

# EXPERIMENT NO:1

Name: Saylee Shirke

Batch:S22

Roll No: 101

**Aim:** Explore usage of basic linux commands and system calls for files, directory and process management.

## **Theory:**

- 1.who : it is used to find out the current user who is logged into the system.
- 2.pwd : present working directory, lets you know the current directory you are in.
- 3.cal : show the calendar of the complete month.
- 4.date : It shows you the current date, along with the time, along with the day, along with the year.
- 5.mkdir: to create a new directory under any directory
- 6.chdir/cd : to change the current working directory
- 7.cat : to create the file and display the contents of the file
- 8.chmod: to change the mode of the file. There are three modes read(r), write(w) and execute(e)
- 9.ls : to list all directories and subdirectories
  - a.ls-l : to show the long listing information about the directory
  - b.ls-lh : human readable format.
  - c.ls-ld : shows the details of the directory content.
  - d.ls-d\* : to show the sub directories in a directory
  - e.ls-a : to show hidden files
  - f.ls-lhs : show files in the descending order in which you have used your files.
10. sort-r file name.txt : sorts the list in reverse order
11. sort-n file name.txt : its sorts the numerical list in ascending order
12. sort nr file name.txt : its sorts the numerical list in reverse order
13. sort u file name.txt : to remove the duplicates

14. sort m file name.txt : Sorts the months in ascending order
15. awk : it is used for the user that defines text patterns that are to be searched for each line of the file.

Syntax , awk '{print}' file name.txt

awk '/faculty/{print}' file name.txt :

awk '{print}NR, \$0}' file name.txt :

NR - specifies the number of lines.

### Code Execution:

```
tsec6@ubuntu:~$ mkdir saylee
tsec6@ubuntu:~$ cd saylee
tsec6@ubuntu:~/saylee$ date
Mon Jan 15 19:48:29 PST 2024
tsec6@ubuntu:~/saylee$ cal dec 2022
    December 2022
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

tsec6@ubuntu:~/saylee$ cat ->lin3
^C
tsec6@ubuntu:~/saylee$ gedit lin3
tsec6@ubuntu:~/saylee$ sort ->nr lin3
^C
tsec6@ubuntu:~/saylee$ sort -nr lin3
786
234
56
45
34
24
1
tsec6@ubuntu:~/saylee$ mkdir rithika
tsec6@ubuntu:~/saylee$ ls
lin3  nr  rithika
```

**Conclusion:** Therefore, the Linux command is a utility of the Linux operating system. Linux commands facilitate efficient file, directory, and process management, enabling users to navigate, manipulate, and control system resources effectively, enhancing productivity and system functionality.

## EXPERIMENT NO:2

Name: Saylee Shirke

Batch: S22

Roll No: 101

**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.

**Theory :**

A]

i) `uname - r{username}`

To show the OS version, release no. OS release version

ii) `uname - v`

Shows Kernel version (last when launched)

iii) `uname - a`

Shows all the details including OS used, PC and other details.

`us - c + u - r → us - a`

B]

i) `ps-aux (i) sort : nk +41 tail`

Sorts 10 processes in descending order. It shows logged in users, modes of memory & storage.

OR

ii) `ps-aux | tail`

Same as above

OR

iii) `ps-aux | sort -nl+4 | tail - n15` (shows top 15 processes)

same with 15

C]

i) `sudo apt install htop`

shows date of processes with highest memory

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

same as above same as above

D]

i) `ps -p$$`

Shows process ID and current shell

ii) `echo $HOME` display home directory

`echo $HOME`

To display home directory

## Code Execution:

*Command execution:*

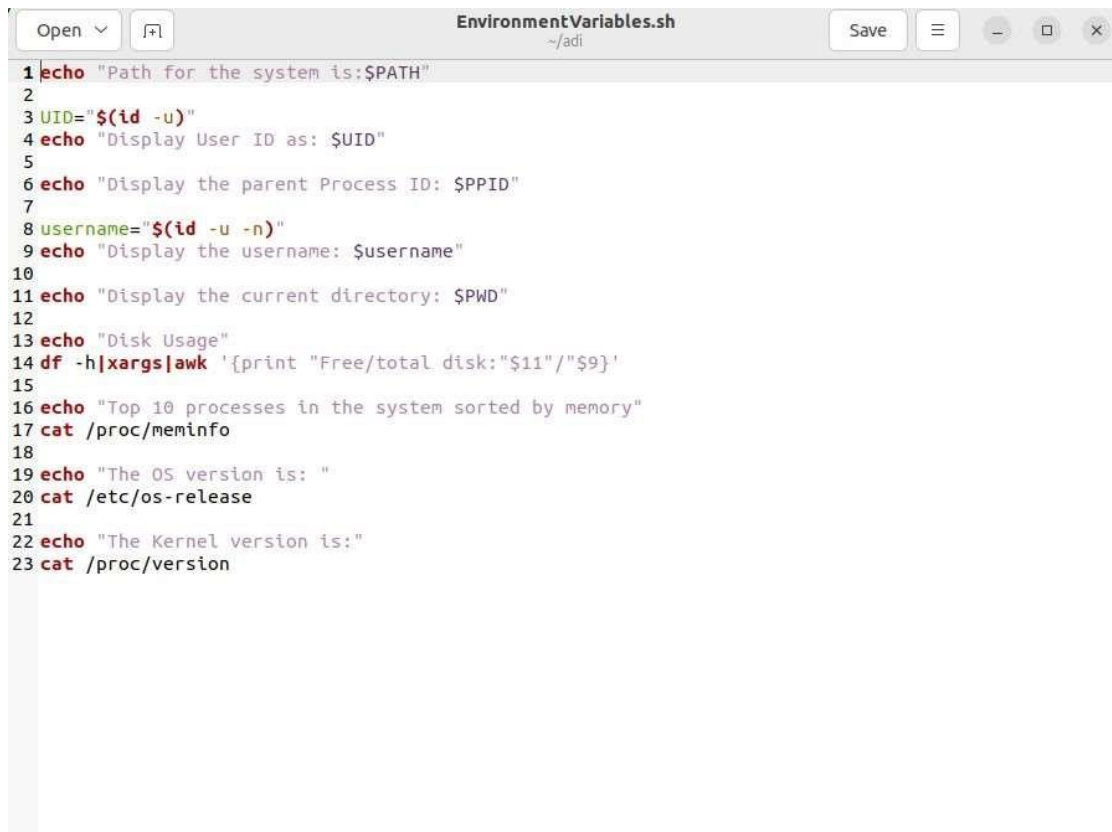
```
File Edit View Search Terminal Help
tsec10@ubuntu:~$ uname -v
#167-18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2023
tsec10@ubuntu:~$ uname -r
5.4.0-150-generic
tsec10@ubuntu:~$ uname -a
Linux ubuntu 5.4.0-150-generic #167-18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2023 x86_64 x86_64 GNU/Linux
tsec10@ubuntu:~$ ps -aux|tail
tsec10  2214  1.0  3.6 1267372 144828 ?        Ssl  19:32  0:02 /usr/bin/gnome-software --gapplication-service
root    2223  0.1  2.0 623192 80568 ?        Ssl  19:32  0:00 /usr/lib/fwupd/fwupd
tsec10  2264  0.3  1.2 825444 51668 ?        Sl   19:32  0:00 /usr/bin/nautilus --gapplication-service
root    2282  0.0  0.0  0  0 ?        T   19:32  0:00 [worker/0:4-cgi]
root    2283  0.0  0.0  0  0 ?        I   19:32  0:00 [worker/0:5]
tsec10  2293  0.2  0.9 791680 36292 ?        Ssl  19:33  0:00 /usr/lib/gnome-terminal/gnome-terminal-server
tsec10  2347  0.0  0.1 22452 4880 pts/0    Ss   19:33  0:00 bash
tsec10  2359  0.0  0.6 586872 25200 tty2     Sl+  19:33  0:00 update-notifier
tsec10  2420  0.0  0.0 39672 3712 pts/0     R+   19:35  0:00 ps -aux
tsec10  2421  0.0  0.0  7516  824 pts/0     S+   19:35  0:00 tail
tsec10@ubuntu:~$ ps -aux|tail -n15
tsec10  2160  0.0  0.1 187908 5128 ?        Sl   19:32  0:00 /usr/lib/dconf/dconf-service
tsec10  2168  0.1  1.5 1129008 62000 ?       Sl   19:32  0:00 /usr/lib/evolution/evolution-calendar-factory-subprocess --factory all --bus-name org.gnome.evolution.dataserver.Subprocess.Backend.Calend
arx2137x2 -own-path /org/gnome/evolution/dataserver/Subprocess/Backend/Calendar/2137/2
tsec10  2178  0.0  0.6 723716 24588 ?        Ssl  19:32  0:00 /usr/lib/evolution/evolution-addressbook-factory
tsec10  2193  0.0  0.6 1875640 26204 ?       Sl   19:32  0:00 /usr/lib/evolution/evolution-addressbook-factory-subprocess --factory all --bus-name org.gnome.evolution.dataserver.Subprocess.Backend.Add
dressBookx2178x2 -own-path /org/gnome/evolution/dataserver/Subprocess/Backend/AddressBook/2178/2
tsec10  2208  0.0  0.1 197360 5628 ?        Ssl  19:32  0:00 /usr/lib/gvfs/gvfsd-metadata
tsec10  2214  0.0  3.6 1267372 144828 ?       Ssl  19:32  0:02 /usr/bin/gnome-software --gapplication-service
root    2223  0.1  2.0 623192 80568 ?        Ssl  19:32  0:00 /usr/lib/fwupd/fwupd
tsec10  2264  0.3  1.2 825444 51668 ?        Sl   19:32  0:00 /usr/bin/nautilus --gapplication-service
root    2282  0.0  0.0  0  0 ?        T   19:32  0:00 [worker/0:4-cgi]
root    2283  0.0  0.0  0  0 ?        I   19:32  0:00 [worker/0:5]
tsec10  2293  0.2  0.9 791680 36344 ?        Ssl  19:33  0:00 /usr/lib/gnome-terminal/gnome-terminal-server
tsec10  2347  0.0  0.1 22452 4884 pts/0    Ss   19:33  0:00 bash
tsec10  2359  0.0  0.6 586872 25200 tty2     Sl+  19:33  0:00 update-notifier
tsec10  2422  0.0  0.0 39672 3700 pts/0     R+   19:36  0:00 ps -aux
tsec10  2423  0.0  0.0  7516  828 pts/0     S+   19:36  0:00 tail -n15
tsec10@ubuntu:~$ ps -p$$
  PID TTY          TIME CMD
 2347 pts/0    00:00:00 bash
```

```

tsec10@ubuntu:~$ sudo apt install htop
[sudo] password for tsec10:
1
Sorry, try again.
[sudo] password for tsec10:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
 fonts-liberation2 fonts-opensymbol gir1.2-gst-plugins-base-1.0 gir1.2-gstreamer-1.0 gir1.2-gudev-1.0 gir1.2-udisks-2.0
 grilo-plugins-0.3-base gstreamer1.0-gtk3 libboost-date-time1.65.1 libboost-filesystem1.65.1 libboost-iostreams1.65.1
 libboost-locale1.65.1 libcdr-0.1-1 libclucene-contribs1v5 libclucene-core1v5 libcmis-0.5-5v5 libcolamd2 libdazzle-1.0-0
 libe-book-0.1-1 libedataserverui-1.2-2 libeot0 libepubgen-0.1-1 libetonyek-0.1-1 libevent-2.1-6 libexiv2-14
 libfreerdp-client2-2 libfreerdp2-2 libgic2 libgee-0.8-2 libgexiv2-2 libgom-1.0-0 libgpgmepp6 libgpod-common libgpod4
 liblangtag-common liblangtag1 liblirc-client0 liblua5.3-0 libmediaart-2.0-0 libmspub-0.1-1 libodfgen-0.1-1 libqwing2v5
 libraw16 libvenge-0.0-0 libsgutils2-2 libssh-4 libsuitesparseconfig5 libvncclient1 libwinpr2-2 libxapian30 libxmlsec1-nss
 linux-hwe-5.4-headers-5.4.0-137 linux-hwe-5.4-headers-5.4.0-84 lp-solve media-player-info python3-mako python3-markupsafe
 syslinux syslinux-common syslinux-legacy usb-creator-common
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
 htop
0 upgraded, 1 newly installed, 0 to remove and 53 not upgraded.
Need to get 80.0 kB of archives.
After this operation, 221 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu bionic/main amd64 htop amd64 2.1.0-3 [80.0 kB]
Fetched 80.0 kB in 1s (84.8 kB/s)
Selecting previously unselected package htop.
(Reading database ... 186880 files and directories currently installed.)
Preparing to unpack .../htop_2.1.0-3_amd64.deb ...
Unpacking htop (2.1.0-3) ...
Setting up htop (2.1.0-3) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...
Processing triggers for mime-support (3.60ubuntu1) ...
tsec10@ubuntu:~$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem|head
  PID  PPID  CMD                                %MEM  %CPU
  1901   1778 /usr/bin/gnome-shell                  4.6    1.5
  2214   1749 /usr/bin/gnome-software --g          3.6    0.3
  1021   1015 /usr/bin/gnome-shell                  3.5    0.2
  2106   1778 nautilus-desktop                    2.7    0.1
  2223     1 /usr/lib/fwupd/fwupd                 2.0    0.0
  1769   1767 /usr/lib/xorg/Xorg vt2 -dis          1.9    0.6
  2137   1749 /usr/lib/evolution/evolutio          1.6    0.0
  1281     1 /usr/lib/packagekit/package          1.6    0.5
  2168   2137 /usr/lib/evolution/evolutio          1.5    0.0

```

*Program execution:*



```
1 echo "Path for the system is:$PATH"
2
3 UID="$(id -u)"
4 echo "Display User ID as: $UID"
5
6 echo "Display the parent Process ID: $PPID"
7
8 username="$(id -u -n)"
9 echo "Display the username: $username"
10
11 echo "Display the current directory: $PWD"
12
13 echo "Disk Usage"
14 df -h|xargs|awk '{print "Free/total disk:"$11/"$9}'
15
16 echo "Top 10 processes in the system sorted by memory"
17 cat /proc/meminfo
18
19 echo "The OS version is: "
20 cat /etc/os-release
21
22 echo "The Kernel version is:"
23 cat /proc/version
```

```

lab-503@lab503-virtual-machine:~$ pwd
Path for the system is: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
Display User ID as: 1000
Display the parent Process ID: 2430
Display the username: lab-503
Display the current directory: /home/lab-503/adl
Disk Usage
Free/total disk:386M/388M
Top 10 processes in the system sorted by memory
MemTotal: 3983512 kB
MemFree: 666456 kB
MemAvailable: 2640720 kB
Buffers: 16936 kB
Cached: 2123076 kB
SwapCached: 8 kB
Active: 1135852 kB
Inactive: 1536368 kB
Active(anon): 547812 kB
Inactive(anon): 28740 kB
Active(file): 588040 kB
Inactive(file): 1507628 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 1939924 kB
SwapFree: 1939656 kB
Zswap: 0 kB
Zswapped: 0 kB
Dirty: 0 kB
Writeback: 0 kB
AnonPages: 532200 kB
Mapped: 301384 kB
Shmem: 44344 kB
KReclaimable: 166112 kB
Slab: 309624 kB
SReclaimable: 166112 kB
SUnreclaim: 143512 kB
KernelStack: 10556 kB
PageTables: 14996 kB
SecPageTables: 0 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 3921000 kB
Committed_AS: 4050420 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 29448 kB
VmallocChunk: 0 kB
Percpu: 100352 kB
HardwareCorrupted: 0 kB

```

```

Percpu: 100352 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
ShmemHugeMmapped: 0 kB
FileHugePages: 0 kB
FileHugeMmapped: 0 kB
Unaccepted: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 0 kB
DirectMap4k: 180224 kB
DirectMap2M: 4005888 kB
DirectMap1G: 2097152 kB
The OS version is:
PRETTY_NAME="Ubuntu 22.04.3 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.3 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=jammy
The kernel version is:
Linux version 6.5.0-14-generic (build@lgcy02-amd64-110) (x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1-22.04) 12.3.0, GNU ld (GNU Binutils for Ubuntu) 2.38) #14-22.04.1-Ubuntu SMP PREEMPT_
v 20 18:15:30 UTC 2
lab-503@lab503-virtual-machine:~$ pwd

```

**Conclusion:** Therefore, the Linux shell scripting commands can be a powerful tool for automating tasks and manipulating files and directories, also they provide insights into system information, process management, and user activity and offer valuable insights for system administrators and users to monitor system health and resource usage efficiently.

## EXPERIMENT NO:03

Name: Saylee Shirke

Roll No: 101

Batch: S22

**Aim:** Implement Basic Commands of Linux like ls, cp, mv using Kernel APIs.

### **Theory:**

1) cp → cp - r → copies the file

cp - backups → It copies the backups of the destination file in the source folder.

cp - i → It asks for confirmation of the user whether yes or no and gives warning to the user before overriding the destination file.

cp -ie.txt      b.txt

cp -f → Unable to open destination file for writing because the user has not allowed the writing operation. By writing the command, the destination file is deleted and the content is copied from source to destination.

cp - v → it is used for which command is running in backup.

cp - p → It preserves the attributes of files such as last date modification time, time of last access owners and the file permissions.

*Syntax* : cp -p source file destination file.

cp - l → To create a hard link file

2) mv → (move or rename).

mv -i → Ask the user for confirmation before moving the file.

mv - f → It provides the protection and overrides destination file forcefully and deletes the source file.

mv - f a.txt    b.txt

mv - n → It takes/prevents the existing file from being overridden.

mv - n a.txt b.txt

mv - b → To take the backup of an existing file



## Output:

```
File Edit View Search Terminal Help
tsec10@ubuntu:~$ uname -v
#167-18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2023
tsec10@ubuntu:~$ uname -r
5.4.0-150-generic
tsec10@ubuntu:~$ uname -a
Linux ubuntu 5.4.0-150-generic #167-18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2023 x86_64 x86_64 GNU/Linux
tsec10@ubuntu:~$ ps -aux|tail
tsec10 2214 1.0 3.0 1267372 144828 ? Sll 19:32 0:02 /usr/bin/gnome-software --gapplication-service
root 2223 0.1 2.0 623192 80568 ? Ssl 19:32 0:00 /usr/lib/fwupd/fwupd
tsec10 2264 0.3 1.2 825444 51668 ? Sl 19:32 0:00 /usr/bin/nautilus --gapplication-service
root 2282 0.0 0.0 0 0 ? I 19:32 0:00 [worker/0:4-cgr]
root 2283 0.0 0.0 0 0 ? I 19:32 0:00 [worker/0:5]
tsec10 2293 0.2 0.9 791680 36292 ? Ssl 19:33 0:00 /usr/lib/gnome-terminal/gnome-terminal-server
tsec10 2347 0.0 0.1 22452 4800 pts/0 Ss 19:33 0:00 bash
tsec10 2359 0.0 0.0 568872 25208 tty2 Sl+ 19:33 0:00 update-notifier
tsec10 2420 0.0 0.0 30672 3712 pts/0 R+ 19:35 0:00 ps -aux
tsec10 2421 0.0 0.0 7516 824 pts/0 S+ 19:35 0:00 tail
tsec10@ubuntu:~$ ps -aux|tail -n15
tsec10 2160 0.0 0.1 107900 5128 ? Sl 19:32 0:00 /usr/lib/dconf/dconf-service
tsec10 2168 0.1 1.5 1129008 62600 ? Sl 19:32 0:00 /usr/lib/evolution/evolution-calendar-factory-subprocess --factory all --bus-name org.gnome.evolution.dataserver.Subprocess.Backend.Calend
arx2137x2 --own-path /org/gnome/evolution/dataserver/Subprocess/Backend/Calendar/2137/2
tsec10 2178 0.0 0.0 275716 24508 ? Ssl 19:32 0:00 /usr/lib/evolution/evolution-addressbook-factory
tsec10 2193 0.0 0.0 1075540 26204 ? Sl 19:32 0:00 /usr/lib/evolution/evolution-addressbook-factory-subprocess --factory all --bus-name org.gnome.evolution.dataserver.Subprocess.Backend.Add
ressBookx2178x2 --own-path /org/gnome/evolution/dataserver/Subprocess/Backend/AddressBook/2178/2
tsec10 2208 0.0 0.1 107500 560 ? Ssl 19:32 0:00 /usr/lib/gvfs/gvfsd-metadata
tsec10 2214 0.9 3.0 1267372 144828 ? Sll 19:32 0:02 /usr/bin/gnome-software --gapplication-service
root 2223 0.1 2.0 623192 80568 ? Ssl 19:32 0:00 /usr/lib/fwupd/fwupd
tsec10 2264 0.3 1.2 825444 51668 ? Sl 19:32 0:00 /usr/bin/nautilus --gapplication-service
root 2282 0.0 0.0 0 0 ? I 19:32 0:00 [worker/0:4-cgr]
root 2283 0.0 0.0 0 0 ? I 19:32 0:00 [worker/0:5]
tsec10 2293 0.2 0.9 791680 36344 ? Ssl 19:33 0:00 /usr/lib/gnome-terminal/gnome-terminal-server
tsec10 2347 0.0 0.1 22452 4804 pts/0 Ss 19:33 0:00 bash
tsec10 2359 0.0 0.0 568872 25208 tty2 Sl+ 19:33 0:00 update-notifier
tsec10 2420 0.0 0.0 30672 3700 pts/0 R+ 19:36 0:00 ps -aux
tsec10 2423 0.0 0.0 7516 838 pts/0 S+ 19:36 0:00 tail -n15
tsec10@ubuntu:~$ ps -p55
PID TTY
2347 pts/0 00:00:00 bash
tsec10@ubuntu:~$ sudo apt install httpd
[sudo] password for tsec10:
1
Sorry, try again.
[sudo] password for tsec10:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
 fonts-liberation2 fonts-opensymbol gir1.2-gst-plugins-base-1.0 gir1.2-gstreamer-1.0 gir1.2-gudev-1.0 gir1.2-udisks-2.0
 grilo-plugins-0.3-base gstreamer1.0-gtk3 libboost-date-time1.65.1 libboost-filesystem1.65.1 libboost-iostreams1.65.1
 libboost-locale1.65.1 libbdr-0.1-1 libclucene-contribs1v5 libclucene-core1v5 libcmis-0.5-5v5 libcolamd libdazzle-1.0-0
 libe-book-0.1-1 libedataserverui-1.2-2 libeot0 libepubgen-0.1-1 libetonyek-0.1-1 libevent-2.1-6 libexiv2-14
 libfreerdp-client-2.2 libfreerdp2-2 libgic2 libgee-0.8-2 libgexiv2-2 libgom-1.0-0 libgpgmepp6 libgpod-common libgpod4
 liblangtag-common liblangtag1 liblirc-client0 liblua5.3-0 libmediaart-2.0-0 libmspub-0.1-1 libodfgen-0.1-1 libqquwing2v5
 libraw16 librevenge-0.0-0 libsgutils2-2 libssh-4 libsuitesparseconfig5 libvncclient1 libwinpr2-2 libxapian30 libxmlsec1-ns
 linux-hwe-5.4-headers-5.4.0-137 linux-hwe-5.4-headers-5.4.0-84 lp-solve media-player-info python3-mako python3-markupsafe
 syslinux syslinux-common syslinux-legacy usb-creator-common
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
 httpd
0 upgraded, 1 newly installed, 0 to remove and 53 not upgraded.
Need to get 80.0 kB of archives.
After this operation, 221 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu bionic/main amd64 httpd amd64 2.1.0-3 [80.0 kB]
Fetched 80.0 kB in 1s (84.8 kB/s)
Selecting previously unselected package httpd.
(Reading database ... 186880 files and directories currently installed.)
Preparing to unpack .../httpd.2.1.0-3_amd64.deb ...
Unpacking httpd (2.1.0-3) ...
Setting up httpd (2.1.0-3) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...
Processing triggers for mime-support (3.60ubuntu1) ...
tsec10@ubuntu:~$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem|head
PID PPID CMD %MEM %CPU
1901 1778 /usr/bin/gnome-shell 4.6 1.5
2214 1749 /usr/bin/gnome-software --g 3.6 0.3
1021 1015 /usr/bin/gnome-shell 3.5 0.2
2106 1778 nautilus-desktop 2.7 0.1
2223 1 /usr/lib/fwupd/fwupd 2.0 0.0
1769 1767 /usr/lib/xorg/Xorg vt2 -dis 1.9 0.6
2137 1749 /usr/lib/evolution/evolutio 1.6 0.0
1281 1 /usr/lib/packagekit/package 1.6 0.5
2168 2137 /usr/lib/evolution/evolutio 1.5 0.0
tsec10@ubuntu:~$
tsec10@ubuntu:~$
```

```
tsec10@ubuntu:~$ sudo apt install httpd
[sudo] password for tsec10:
1
Sorry, try again.
[sudo] password for tsec10:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
 fonts-liberation2 fonts-opensymbol gir1.2-gst-plugins-base-1.0 gir1.2-gstreamer-1.0 gir1.2-gudev-1.0 gir1.2-udisks-2.0
 grilo-plugins-0.3-base gstreamer1.0-gtk3 libboost-date-time1.65.1 libboost-filesystem1.65.1 libboost-iostreams1.65.1
 libboost-locale1.65.1 libbdr-0.1-1 libclucene-contribs1v5 libclucene-core1v5 libcmis-0.5-5v5 libcolamd libdazzle-1.0-0
 libe-book-0.1-1 libedataserverui-1.2-2 libeot0 libepubgen-0.1-1 libetonyek-0.1-1 libevent-2.1-6 libexiv2-14
 libfreerdp-client-2.2 libfreerdp2-2 libgic2 libgee-0.8-2 libgexiv2-2 libgom-1.0-0 libgpgmepp6 libgpod-common libgpod4
 liblangtag-common liblangtag1 liblirc-client0 liblua5.3-0 libmediaart-2.0-0 libmspub-0.1-1 libodfgen-0.1-1 libqquwing2v5
 libraw16 librevenge-0.0-0 libsgutils2-2 libssh-4 libsuitesparseconfig5 libvncclient1 libwinpr2-2 libxapian30 libxmlsec1-ns
 linux-hwe-5.4-headers-5.4.0-137 linux-hwe-5.4-headers-5.4.0-84 lp-solve media-player-info python3-mako python3-markupsafe
 syslinux syslinux-common syslinux-legacy usb-creator-common
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
 httpd
0 upgraded, 1 newly installed, 0 to remove and 53 not upgraded.
Need to get 80.0 kB of archives.
After this operation, 221 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu bionic/main amd64 httpd amd64 2.1.0-3 [80.0 kB]
Fetched 80.0 kB in 1s (84.8 kB/s)
Selecting previously unselected package httpd.
(Reading database ... 186880 files and directories currently installed.)
Preparing to unpack .../httpd.2.1.0-3_amd64.deb ...
Unpacking httpd (2.1.0-3) ...
Setting up httpd (2.1.0-3) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...
Processing triggers for mime-support (3.60ubuntu1) ...
tsec10@ubuntu:~$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem|head
PID PPID CMD %MEM %CPU
1901 1778 /usr/bin/gnome-shell 4.6 1.5
2214 1749 /usr/bin/gnome-software --g 3.6 0.3
1021 1015 /usr/bin/gnome-shell 3.5 0.2
2106 1778 nautilus-desktop 2.7 0.1
2223 1 /usr/lib/fwupd/fwupd 2.0 0.0
1769 1767 /usr/lib/xorg/Xorg vt2 -dis 1.9 0.6
2137 1749 /usr/lib/evolution/evolutio 1.6 0.0
1281 1 /usr/lib/packagekit/package 1.6 0.5
2168 2137 /usr/lib/evolution/evolutio 1.5 0.0
tsec10@ubuntu:~$
tsec10@ubuntu:~$
```

## Output 2:

```
Open ▾  copy.c  Save  ≡  ⌵  ✖
#include<stdlib.h>
int main()
{
FILE *fptr1, *fptr2;
char filename [100], c;
printf("Enter the filename to open for reading \n");
scanf("%s", filename); // Open one file for reading

fptr1 = fopen(filename, "r");

if(fptr1 ==NULL)
{printf("Cannot open file %s in", filename); exit(0);}
printf("Enter the filename to open for writing \n");
scanf("%s", filename);
// Open another file for writing

fptr2= fopen(filename, "w"); if (fptr2 == NULL)
{printf("Cannot open file %s\n", filename); exit(0);} // Read contents from file
c = fgetc(fptr1); while (c != EOF) {fputc(c, fptr2);
c = fgetc(fptr1);}printf("\nContents copied to %s", filename);
fclose(fptr1); fclose(fptr2); return 0;
}
```

```
tsec@ubuntu:~$ gcc -o copy.out copy.c
tsec@ubuntu:~$ ./copy.out
Enter the filename to open for reading
example.txt
Enter the filename to open for writing
example1.txt
Contents copied to example1.txt

tsec@ubuntu:~$ cat example1.txt
Hello!!!
```

**Conclusion:** Thus we have successfully studied and implemented various basic commands of Linux like ls, cp and mv using kernel APIs

# **EXPERIMENT NO:04**

Name: Saylee Shirke

Batch: S22

Roll no: 101

**AIM:** a. 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.  
b. Explore wait and waitpid before termination of process

## **THEORY:**

### 1] System call

When a program is in user mode and requires access to Ram or hardware resources, it must ask the kernel to provide access to that resource. This is done via something called a system call. When a program makes a system call, the mode is switched from user mode to Kernel mode. This is called context switch. The kernel provides the resources which the program requested. System calls are made by user level programmes in following cases:

Creating, opening, closing and delete files in the system

Creating and managing new processes

Creating a connection in the network, sending and receiving packets

Requesting access to a hardware device like a mouse or a printer

### 2] Fork ( )

The fork system call is used to create processes. When a process makes a fork().call, an exact copy of the process is created.

There are now two processes, one being the parent process and the other being the child process. The process which called fork().call is the parent process and the process which is created newly is called child process. The child process will be exactly the same as the parent. The process state of the parent i.e the address space, variables, open files etc is copied into the child process. The change of values in the parent process doesn't affect the child and vice versa.

**Conclusion:** Hence we implemented and studied how to create child process in Linux using for system calls

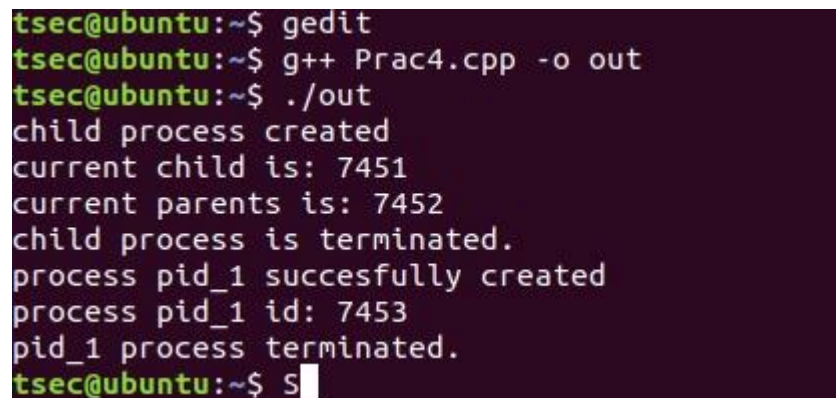
# Implementation with code:

## CODE:

```
#include <iostream>
#include<sys/wait.h>
#include<unistd.h>

using namespace std;
int wait_func()
{
    int pid_1 = fork();
    if(pid_1==0)
    {
        cout<<"process pid_1 succesfully created"<<"\n";
        cout<<"process pid_1 id: "<< getpid()<<"\n";
        exit(0);
    }
    waitpid(pid_1, NULL,0);
    cout << "pid_1 process terminated."<<"\n";
    return 0;
}
int main()
{
    int pid=fork();
    if(pid==0)
    {
        cout<<"child process created"<<"\n";
        cout<<"current child is: "<<getppid()<<"\n";
        cout<<"current parents is: "<<getpid()<<"\n";
        exit(0);
    }
    wait(NULL);
    cout<<"child process is terminated."<<"\n";
    wait_func();
    return 0;
}
```

## OUTPUT:



```
tsec@ubuntu:~$ gedit
tsec@ubuntu:~$ g++ Prac4.cpp -o out
tsec@ubuntu:~$ ./out
child process created
current child is: 7451
current parents is: 7452
child process is terminated.
process pid_1 succesfully created
process pid_1 id: 7453
pid_1 process terminated.
tsec@ubuntu:~$ S
```

## **EXPERIMENT NO: 05**

Name: Saylee Shirke

Batch: S22

Roll no: 101

### **Aim:**

- a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms.
  
- b. Write a program to demonstrate the concept of preemptive scheduling algorithms.

### **Theory :**

#### **Non-preemptive Scheduling Algorithms**

Non-preemptive scheduling algorithms constitute a class of scheduling techniques utilized in operating systems and real-time systems to manage resource allocation, such as CPU time, among competing processes or tasks. Unlike preemptive scheduling algorithms, which permit the scheduler to interrupt and forcibly terminate a running process to allocate the CPU to another process, non-preemptive scheduling algorithms do not interrupt the currently executing process until it voluntarily relinquishes the CPU or completes its execution. These algorithms are designed to be simpler to implement and have lower overhead compared to preemptive algorithms because they do not require mechanisms for process interruption and context switching.

One of the most common non-preemptive scheduling algorithms is First-Come, First-Served (FCFS). In FCFS scheduling, processes are executed in the order they arrive in the ready queue. The CPU is assigned to the first process in the queue, and it continues executing until it completes its execution or is blocked for I/O operations. Once a process finishes, the CPU is assigned to the next process in the queue. FCFS is straightforward to implement and understand but may lead to poor performance, especially in scenarios where short processes are waiting behind long processes, causing long turnaround times and low throughput.

Another non-preemptive scheduling algorithm is Shortest Job Next (SJN), also known as Shortest Job First (SJF). SJN selects the process with the shortest estimated burst time next for execution. This algorithm aims to minimize the average waiting time and turnaround time by prioritizing shorter jobs over longer ones. However, SJN requires accurate estimates of the burst times, which may not always be available in practice.

Similarly, another variant of SJN is the Shortest Remaining Time (SRT) scheduling algorithm, which preempts the currently running process if a new process with a shorter burst time arrives. SRT dynamically adjusts to the arrival of new processes and aims to minimize the remaining time for each process to complete.

Priority scheduling is another non-preemptive algorithm where each process is assigned a priority. The CPU is allocated to the process with the highest priority that is ready to run. Priority scheduling can be implemented using either static priorities assigned to processes or dynamic priorities that change over time based on factors like aging process behavior.

While non-preemptive scheduling algorithms are simpler to implement and have lower overhead compared to preemptive algorithms, they may suffer from drawbacks such as poor response time and throughput, especially in dynamic and time-sensitive environments. Additionally, non-preemptive algorithms may not be suitable for real-time systems or scenarios where fairness and responsiveness are critical.

## Preemptive Scheduling Algorithms

Preemptive scheduling algorithms represent a category of scheduling techniques employed in operating systems and real-time systems to manage resource allocation, particularly CPU time, among competing processes or tasks. Unlike non-preemptive scheduling algorithms, which allow a process to run until completion or voluntarily relinquish the CPU, preemptive algorithms enable the scheduler to interrupt the currently running process and allocate the CPU to another process based on certain criteria.

One of the widely used preemptive scheduling algorithms is Round Robin (RR). In Round Robin scheduling, each process is assigned a fixed time quantum or time slice. The CPU is allocated to a process for the duration of its time slice, and if the process does not complete within that time, it is moved to the back of the ready queue, and the CPU is given to the next

process in line. Round Robin is fair and ensures that every process gets a chance to execute, but it may lead to higher waiting times for short processes.

Another preemptive scheduling algorithm is Priority Scheduling, which assigns priorities to processes. The process with the highest priority currently in the ready queue is given the CPU. If a new process arrives with a higher priority than the currently running process, a preemption occurs, and the higher-priority process is given the CPU. Priority scheduling can be implemented using static priorities assigned to processes or dynamic priorities that change during runtime based on factors such as aging or process behavior.

In preemptive algorithms, the concept of aging is crucial. Aging refers to gradually increasing the priority of a process that has been waiting for a long time. This helps prevent starvation, ensuring that even lower-priority processes eventually get a chance to execute.

Shortest Remaining Time (SRT) is another preemptive scheduling algorithm, a variation of Shortest Job Next (SJN). In SRT, the scheduler continuously monitors the remaining time of each process. If a new process arrives with a shorter remaining time than the currently running process, a preemption occurs, and the CPU is assigned to the new process. SRT dynamically adapts to the changing characteristics of the processes and aims to minimize the remaining time for each process to complete.

Preemptive scheduling algorithms are essential for real-time systems where responsiveness and meeting deadlines are critical. They provide better control over the allocation of CPU time and can respond promptly to changing system conditions. However, they come with higher overhead due to frequent context switches, leading to potential performance implications. Additionally, priority inversion is a concern in preemptive scheduling, where a low-priority task holds a resource needed by a high-priority task, causing the high-priority task to be delayed. Techniques like priority inheritance or priority ceiling protocols are employed to mitigate priority inversion issues in preemptive scheduling.

A] Non-preemptive Scheduling:

**CODE:**

```

#include <stdio.h>

void calculateTimes(int n, int bt[], int at[], int wt[], int tat[])
{
    wt[0] = 0;
    for (int i = 1; i < n; i++)
    {
        wt[i] = wt[i - 1] + bt[i - 1];
    }
    for (int i = 0; i < n; i++)
    {
        tat[i] = wt[i] + bt[i];
    }
}

void displayResults(int n, int bt[], int at[], int wt[], int tat[])
{
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround\n");
    printf("Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], wt[i],
            tat[i]);
    }
}

int main()
{
    int n;
    // Input the number of processes
    printf("Name:Saylee Shirke\nBatch:S22\nRoll no:101\n");
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int burstTime[n], arrivalTime[n], waitingTime[n], turnaroundTime[n];
    printf("Enter the arrival times and burst times of processes:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrivalTime[i]);
        printf("Burst Time: ");
        scanf("%d", &burstTime[i]);
    }
    calculateTimes(n, burstTime, arrivalTime, waitingTime,
turnaroundTime);
}

```



```

    displayResults(n, burstTime, arrivalTime, waitingTime,
turnaroundTime);
    return 0;
}

```

## OUTPUT:

```

Name: Saylee Shirke
Batch: S22
Roll no: 101
Enter the number of processes: 5
Enter the arrival times and burst times of processes:
Process 1:
Arrival Time: 2
Burst Time: 3
Process 2:
Arrival Time: 1
Burst Time: 2
Process 3:
Arrival Time: 3
Burst Time: 4
Process 4:
Arrival Time: 4
Burst Time: 1
Process 5:
Arrival Time: 5
Burst Time: 5

```

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	2	3	0	3
2	1	2	3	5
3	3	4	5	9
4	4	1	9	10
5	5	5	10	15

## B] Preemptive Scheduling

### CODE:

```

#include <stdio.h>
void srtf(int n, int at[], int bt[]);
int main()
{
    int n;
    printf("Name: Saylee Shirke\nBatch: S22\nRoll no: 101\n");
    printf("Enter the number of processes: ");
}

```

```

scanf("%d", &n);
int arrival_time[n];
int burst_time[n];
for (int i = 0; i < n; i++)
{
    printf("Enter Arrival time of process %d: ", i + 1);
    scanf("%d", &arrival_time[i]);
    printf("Enter Burst time of process %d: ", i + 1);
    scanf("%d", &burst_time[i]);
}
srtf(n, arrival_time, burst_time);
return 0;
}
void srtf(int n, int at[], int bt[])
{
    int remaining_time[n];
    int waiting_time[n];
    int turnaround_time[n];
    int completed = 0;
    int current_time = 0;
    for (int i = 0; i < n; i++)
    {
        remaining_time[i] = bt[i];
        waiting_time[i] = 0;
    }
    while (completed < n)
    {
        int shortest = -1;
        for (int i = 0; i < n; i++)
        {
            if (at[i] <= current_time && remaining_time[i] > 0)
            {
                if (shortest == -1 || remaining_time[i] <
                    remaining_time[shortest])
                {
                    shortest = i;
                }
            }
        }
        if (shortest == -1)
        {
            current_time++;
        }
    }
}

```

```

else
{
    remaining_time[shortest]--;
    current_time++;
    if (remaining_time[shortest] == 0)
    {
        turnaround_time[shortest] = current_time - at[shortest];
        waiting_time[shortest] = turnaround_time[shortest] -
            bt[shortest];
        completed++;
    }
}
}
printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");
for (int i = 0; i < n; i++)
{
    printf("%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], waiting_time[i],
        turnaround_time[i]);
}
float average_waiting_time = 0, average_turnaround_time = 0;
for (int i = 0; i < n; i++)
{
    average_waiting_time += waiting_time[i];
    average_turnaround_time += turnaround_time[i];
}
average_waiting_time /= n;
average_turnaround_time /= n;
printf("\nAverage Waiting Time: %.2f\n", average_waiting_time);
printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}

```

## OUTPUT:

```
Name: Saylee Shirke
Batch: S22
Roll no: 101
Enter the number of processes: 5
Enter Arrival time of process 1: 2
Enter Burst time of process 1: 1
Enter Arrival time of process 2: 1
Enter Burst time of process 2: 3
Enter Arrival time of process 3: 4
Enter Burst time of process 3: 2
Enter Arrival time of process 4: 5
Enter Burst time of process 4: 3
Enter Arrival time of process 5: 3
Enter Burst time of process 5: 2

Process  Arrival  Burst    Waiting  Turnaround
1         2        1        0         1
2         1        3        1         4
3         4        2        1         3
4         5        3        4         7
5         3        2        4         6

Average Waiting Time: 2.00
Average Turnaround Time: 4.20
```

## CONCLUSION:

In summary, non-preemptive scheduling algorithms manage the allocation of CPU resources among competing processes without forcibly interrupting running processes.

## **EXPERIMENT NO.06**

Name: Saylee Shirke

Batch: S22

Roll no: 101

### **AIM:**

Process Management : Synchronization : To implement solution of Producer/ Consumer problem using Semaphore

### **THEORY:**

The Producer Consumer problem is a process synchronization problem. In this problem, there is a memory buffer of a fixed size. Two processes access the shared buffer: Producer and Consumer. A producer creates new items and adds to the buffer, while a consumer picks items from the shared buffer. The problem is to ensure synchronization between the producer and the consumer such that while the producer is adding an item to the buffer, the consumer does not start accessing it.

Further, the producer should not try to add items to the buffer when it is full and the consumer should not access the buffer when it is empty. If the producer finds the buffer full, it should either sleep or discard the new item until the consumer frees a location in the buffer.

Similarly, a consumer may go to sleep if it finds the buffer empty. When the producer adds an item, the consumer can continue retrieving the data.

The Producer-Consumer problem is also known as the Bounded Buffer problem.

A solution is required for the Producer-Consumer problem such that both processes can perform their tasks without ending up in a deadlock

### **CODE:**

```
// Online C compiler to run C program online
```

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

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

```
int mutex = 1;
```

```
int full = 0;
```

```
int empty = 10, x = 0;
```

```
void producer()
```

```
{  
    --mutex;  
    ++full;  
    --empty;  
    x++;  
    printf("Producer produces item %d",x);  
    ++mutex;  
}
```

```
void consumer()
```

```
{  
    --mutex;  
    --full;  
    ++empty;  
    printf("Consumer consumes item %d",x);  
    x--;  
    ++mutex;  
}
```

```
int main()
```

```
{  
    int n, i;  
    printf("Name:Saylee Shirke\nBatch:S22\nRoll No:101\n");  
    printf("\n1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 for Exit");
```

```
for (i = 1; i > 0; i++) {
```

```
printf("\n\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;
```

```
}
```

```
}
```

```
}
```



## OUTPUT:

```
Name: Saylee Shirke
Batch: S22
Roll No: 101

1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit

Enter your choice: 1
Producer produces item 1

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 1
Producer produces item 1

Enter your choice: 1
Producer produces item 2

Enter your choice: 2
Consumer consumes item 2

Enter your choice: 2
Consumer consumes item 1

Enter your choice: 2
Buffer is empty!

Enter your choice: 3

...Program finished with exit code 0
Press ENTER to exit console.
```

## CONCLUSION:

Thus we implemented and studied how to solve a Producer/ Consumer problem using Semaphore successfully

## EXPERIMENT NO.7

Name: Saylee Shirke

Batch: S22

Roll no: 101

### AIM:

- a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
- b. Write a program demonstrate the concept of Dining Philosopher's Problem

### Theory:

#### Bankers Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that simulates resource allocation for predetermined maximum possible amounts of all resources before performing an "s-state" check to look for potential activities and determining whether allocation should be permitted to continue

The reason the banker's algorithm is so titled is because it is employed in the banking sector to determine whether or not to authorize a loan to an individual. Assume a bank has  $n$  account holders, each of whom has an individual balance of  $S$ . When someone requests for a loan, the bank first deducts the loan amount from the total amount of money it possesses, and only approves the loan if the balance is more than  $S$ . It is done because the bank can simply do it if every account holder shows up to get their money.

In other words, the bank would never arrange its funds in a way that would prevent it from meeting the demands of all of its clients. The bank would strive to maintain safety at all times.

#### Dining Philosopher's Problem

The dining philosopher's problem is the classical problem of synchronization which says that five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right)

chopstick and starts thinking again. The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

### **Code:**

#### **Banker's Algorithm**

```
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    printf("Name: Saylee Shirke\nBatch: S22\nRoll no: 101\n");
    int alloc[5][3] = { { 2, 1, 0 }, // P0    // Allocation Matrix
                        { 1, 0, 4 }, // P1
                        { 3, 1, 2 }, // P2
                        { 2, 1, 3 }, // P3
                        { 1, 0, 2 } }; // P4

    int max[5][3] = { { 6, 5, 3 }, // P0    // MAX Matrix
                     { 3, 1, 2 }, // P1
                     { 9, 0, 1 }, // P2
                     { 1, 2, 2 }, // P3
                     { 4, 2, 3 } }; // P4

    int avail[3] = { 2, 3, 3 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
```

```

int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {

            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                    break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
}

```

```

int flag = 1;

```

```

for(int i=0;i<n;i++)
{
    if(f[i]==0)
    {

```

```

        flag=0;
        printf("The following system is not safe");
        break;
    }
}

if(flag==1)
{
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
}

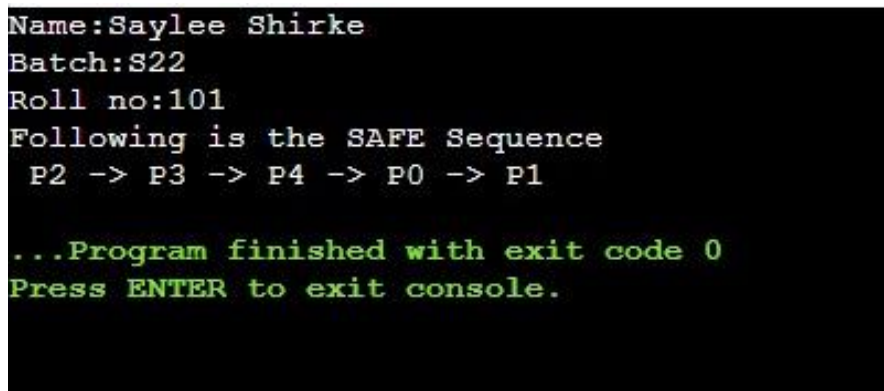
return (0);

// This code is contributed by Deep Baldha (CandyZack)
}

```

## Output:

### Banker's Algorithm



```

Name: Saylee Shirke
Batch: S22
Roll no: 101
Following is the SAFE Sequence
P2 -> P3 -> P4 -> P0 -> P1

...Program finished with exit code 0
Press ENTER to exit console.

```

# Code:

## Dining Philosopher's Problem

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
sem_t chopstick[5];
void * philos(void *);
void eat(int);
int main()
{
    printf("Name:Saylee Shirke\nBatch:S22\nRoll no:101\n");

    int i,n[5];
    pthread_t T[5];
    for(i=0;i<5;i++)
        sem_init(&chopstick[i],0,1);
    for(i=0;i<5;i++){
        n[i]=i;
        pthread_create(&T[i],NULL,philos,(void *)&n[i]);
    }
    for(i=0;i<5;i++)
        pthread_join(T[i],NULL);
}
void * philos(void * n)
{
    int ph=*(int *)n;
    printf("Philosopher %d wants to eat\n",ph);
    printf("Philosopher %d tries to pick left chopstick\n",ph);
    sem_wait(&chopstick[ph]);
    printf("Philosopher %d picks the left chopstick\n",ph);
```

```

    printf("Philosopher %d tries to pick the right chopstick\n",ph);
    sem_wait(&chopstick[(ph+1)%5]);
    printf("Philosopher %d picks the right chopstick\n",ph);
    eat(ph);
    sleep(2);
    printf("Philosopher %d has finished eating\n",ph);
    sem_post(&chopstick[(ph+1)%5]);
    printf("Philosopher %d leaves the right chopstick\n",ph);
    sem_post(&chopstick[ph]);
    printf("Philosopher %d leaves the left chopstick\n",ph);
}
void eat(int ph)
{
    printf("Philosopher %d begins to eat\n",ph);
}

```

# OUTPUT:

## Dining Philosopher's Problem

```
Name: Saylee Shirke
Batch: S22
Roll no: 101

Philosopher 3 wants to eat
Philosopher 3 tries to pick left chopstick
Philosopher 3 picks the left chopstick
Philosopher 3 tries to pick the right chopstick
Philosopher 3 picks the right chopstick
Philosopher 3 begins to eat
Philosopher 4 wants to eat
Philosopher 4 tries to pick left chopstick
Philosopher 0 wants to eat
Philosopher 0 tries to pick left chopstick
Philosopher 0 picks the left chopstick
Philosopher 0 tries to pick the right chopstick
Philosopher 0 picks the right chopstick
Philosopher 0 begins to eat
Philosopher 1 wants to eat
Philosopher 1 tries to pick left chopstick
Philosopher 2 wants to eat
Philosopher 2 tries to pick left chopstick
Philosopher 2 picks the left chopstick
Philosopher 2 tries to pick the right chopstick
Philosopher 3 has finished eating
Philosopher 3 leaves the right chopstick
Philosopher 4 picks the left chopstick
Philosopher 2 picks the right chopstick
Philosopher 4 tries to pick the right chopstick
Philosopher 3 leaves the left chopstick
Philosopher 2 begins to eat
Philosopher 0 has finished eating
Philosopher 0 leaves the right chopstick
Philosopher 0 leaves the left chopstick
Philosopher 4 picks the right chopstick
Philosopher 4 begins to eat
Philosopher 1 picks the left chopstick
Philosopher 1 tries to pick the right chopstick
Philosopher 2 has finished eating
Philosopher 2 leaves the right chopstick
Philosopher 2 leaves the left chopstick
Philosopher 1 picks the right chopstick
```



```
Philosopher 1 begins to eat
Philosopher 4 has finished eating
Philosopher 4 leaves the right chopstick
Philosopher 4 leaves the left chopstick
Philosopher 1 has finished eating
Philosopher 1 leaves the right chopstick
Philosopher 1 leaves the left chopstick

...Program finished with exit code 0
Press ENTER to exit console. □
```

## CONCLUSION:

The Banker's Algorithm ensures resource allocation in a way that prevents deadlock, while the Dining Philosophers problem illustrates challenges in resource sharing and deadlock avoidance among concurrent processes. Both highlight the importance of careful resource management and synchronization techniques in multi-process environments.

## **EXPERIMENT NO:08**

Name: Saylee Shirke

Batch: S22

Roll no: 101

**AIM:** Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst- Fit

### **THEORY:**

Memory management in operating systems is a crucial aspect that involves the efficient allocation and management of system memory resources. Among the various techniques used in memory management, dynamic partitioning is a significant concept. In dynamic partitioning, the memory is divided into variable-sized partitions to accommodate processes with varying memory requirements. Placement algorithms play a vital role in dynamic partitioning, as they determine how processes are allocated to these partitions. In this essay, we will delve into the concept of dynamic partitioning and explore different placement algorithms in detail.

Dynamic partitioning involves dividing the available memory into variable-sized partitions to accommodate processes as per their memory requirements. When a process requests memory allocation, the operating system searches for a suitable partition to accommodate the process. If a partition is found that satisfies the memory requirement of the process, it is allocated to that partition. However, if no such partition is available, the operating system needs to perform memory allocation by either creating a new partition or combining adjacent free partitions to meet the process's requirements.

Placement algorithms are employed to decide how processes are allocated to memory partitions. Several placement algorithms are used in dynamic partitioning, each with its advantages and disadvantages. Let's explore some of the commonly used placement algorithms:

### **First Fit:**

In the First Fit algorithm, the operating system allocates the process to the first partition that is large enough to accommodate it.

It is a simple and efficient algorithm as it requires scanning the memory partitions sequentially.

However, it may lead to fragmentation, as smaller partitions might remain unused even if they could accommodate smaller processes that arrive later.

### **Best Fit:**

The Best Fit algorithm allocates the process to the smallest partition that is large enough to accommodate it.

It aims to minimize fragmentation by utilizing memory efficiently.

However, it requires searching the entire memory space to find the smallest suitable partition, leading to higher time complexity compared to First Fit.

### **Worst Fit:**

The Worst Fit algorithm allocates the process to the largest partition available that can accommodate it.

It tends to leave the largest holes in memory, potentially leading to faster fragmentation compared to other algorithms.

Despite its name, Worst Fit can sometimes perform better in certain scenarios where larger processes are common.

### **Next Fit:**

The Next Fit algorithm is similar to First Fit but starts searching for suitable partitions from the last allocated partition rather than from the beginning.

It reduces search time compared to First Fit, especially when there are many small processes being allocated successively.

However, it can still suffer from fragmentation issues similar to First Fit.

Each placement algorithm has its trade-offs in terms of memory utilization, time complexity, and fragmentation. The choice of algorithm depends on the specific requirements and characteristics of the system.

Fragmentation is a critical issue in dynamic partitioning, leading to inefficient use of memory resources. Fragmentation can be of two types: external fragmentation and internal fragmentation. External fragmentation occurs when free memory blocks are scattered throughout the memory space, making it challenging to allocate contiguous memory to processes even though the total free memory might be sufficient. Internal fragmentation, on the other hand, happens when a partition allocated to a process is larger than what the process actually requires, leading to wasted memory within the partition.

To mitigate fragmentation, various techniques are employed:

**Compaction:** This involves moving processes in memory to consolidate free memory blocks and reduce external fragmentation. However, compaction can be expensive in terms of time and resources.

**Memory Compaction:** Memory compaction involves moving allocated memory to eliminate internal fragmentation. This can be achieved through techniques like memory paging or segmentation.

**Virtual Memory:** Virtual memory allows processes to use more memory than physically available by utilizing disk space as an extension of RAM. This helps in reducing the effects of fragmentation by providing a larger address space for processes.

**CODE:**

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

```
#define MAX_PARTITIONS 100
```

```
#define MAX_PROCESS 100
```

```
int partitions[MAX_PARTITIONS];
```

```
int processes[MAX_PROCESS];
```

```
int num_partitions, num_processes;
```

```
// Function prototypes
```

```
void initialize();
```

```
void displayPartitions();
```

```
void displayProcesses();
```

```
void firstFit();
```

```
void bestFit();
```

```
void worstFit();
```

```
int main() {
```

```
initialize();

printf("Initial partitions:\n");
displayPartitions();
printf("Processes:\n");
displayProcesses();

// Perform allocation using different algorithms
printf("\nFirst Fit Allocation:\n");
firstFit();
displayPartitions();

printf("\nBest Fit Allocation:\n");
bestFit();
displayPartitions();

printf("\nWorst Fit Allocation:\n");
worstFit();
displayPartitions();

return 0;
}

void initialize() {
    printf("Enter number of partitions: ");
```

```
scanf("%d", &num_partitions);
printf("Enter sizes of partitions:\n");
for (int i = 0; i < num_partitions; i++) {
    scanf("%d", &partitions[i]);
}

printf("Enter number of processes: ");
scanf("%d", &num_processes);
printf("Enter sizes of processes:\n");
for (int i = 0; i < num_processes; i++) {
    scanf("%d", &processes[i]);
}
}
```

```
void displayPartitions() {
    printf("Partitions:\n");
    for (int i = 0; i < num_partitions; i++) {
        printf("%d ", partitions[i]);
    }
    printf("\n");
}
```

```
void displayProcesses() {
    printf("Processes:\n");
    for (int i = 0; i < num_processes; i++) {
```

```
    printf("%d ", processes[i]);  
}  
printf("\n");  
}
```

```
void firstFit() {  
    for (int i = 0; i < num_processes; i++) {  
        for (int j = 0; j < num_partitions; j++) {  
            if (processes[i] <= partitions[j]) {  
                partitions[j] -= processes[i];  
                break;  
            }  
        }  
    }  
}
```

```
void bestFit() {  
    for (int i = 0; i < num_processes; i++) {  
        int best_index = -1;  
        for (int j = 0; j < num_partitions; j++) {  
            if (processes[i] <= partitions[j] && (best_index == -1 ||  
partitions[j] < partitions[best_index])) {  
                best_index = j;  
            }  
        }  
    }  
}
```



```

    if (best_index != -1) {
        partitions[best_index] -= processes[i];
    }
}
}

void worstFit() {
    for (int i = 0; i < num_processes; i++) {
        int worst_index = -1;
        for (int j = 0; j < num_partitions; j++) {
            if (processes[i] <= partitions[j] && (worst_index == -1 ||
partitions[j] > partitions[worst_index])) {
                worst_index = j;
            }
        }
        if (worst_index != -1) {
            partitions[worst_index] -= processes[i];
        }
    }
}

```

**OUTPUT:**

```
Enter number of partitions: 5
Enter sizes of partitions:
100 200 150 300 250
Enter number of processes: 4
Enter sizes of processes:
50 100 200 150
Initial partitions:
Partitions:
100 200 150 300 250
Processes:
Processes:
50 100 200 150

First Fit Allocation:
Partitions:
50 100 0 100 250

Best Fit Allocation:
Partitions:
0 0 0 100 50

Worst Fit Allocation:
Partitions:
0 0 0 50 50

...Program finished with exit code 0
Press ENTER to exit console.
```

## CONCLUSION:

In conclusion, dynamic partitioning is a fundamental concept in memory management, allowing the efficient allocation of memory to processes with varying memory requirements. Placement algorithms play a crucial role in dynamic partitioning by determining how processes are allocated to memory partitions. While each algorithm has its advantages and disadvantages, fragmentation remains a significant challenge in dynamic partitioning, which needs to be addressed using various techniques like compaction and virtual memory to ensure optimal memory utilization and system performance.

## **EXPERIMENT NO:09**

Name: Saylee Shirke

Batch: S22

Roll no: 101

**AIM:** Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU

### **THEORY:**

Page replacement policies are an integral part of virtual memory management in operating systems. When a program accesses data or instructions, the operating system loads the corresponding pages into physical memory (RAM) from secondary storage (usually a hard disk). However, physical memory has limited capacity, so not all pages can reside in memory simultaneously. When a program requests a page that is not in memory, a page fault occurs, triggering the need for a page replacement policy to determine which page to evict from memory to make room for the new one. Two commonly used page replacement policies are First-In-First-Out (FIFO) and Least Recently Used (LRU). Let's delve into these policies and understand their mechanisms and trade-offs.

#### **First-In-First-Out (FIFO):**

FIFO is one of the simplest page replacement algorithms. It evicts the oldest page in memory, based on the assumption that the page that has been in memory the longest is least likely to be needed in the near future.

#### **Mechanism:**

When a page fault occurs and memory is full, the operating system selects the page that entered memory earliest (the oldest page) for replacement.

The selected page is evicted from memory, and the new page is loaded in its place.

The page table is updated accordingly.

Advantages:

**Simplicity:** FIFO is easy to implement and understand.

**Low Overhead:** The overhead of maintaining data structures is minimal.

Disadvantages:

**Belady's Anomaly:** FIFO can suffer from Belady's anomaly, where increasing the number of frames can actually increase the number of page faults.

**Poor Performance:** FIFO does not consider the access history of pages, leading to suboptimal performance in many cases, especially when the access patterns are irregular.

Least Recently Used (LRU):

LRU is based on the principle that the page that has not been accessed for the longest time is least likely to be used in the near future.

Mechanism:

LRU keeps track of the time of the last access for each page.

When a page fault occurs, the operating system selects the page that was least recently accessed for replacement.

The selected page is evicted from memory, and the new page is loaded in its place.

The page table is updated accordingly, and the access time of the new page is recorded.

Advantages:

**Optimality:** LRU provides better performance than FIFO in terms of reducing the number of page faults in many scenarios.

**Flexibility:** LRU can adapt to varying access patterns by considering the access history of pages.

Disadvantages:

**Implementation Complexity:** Implementing an efficient LRU algorithm requires maintaining a data structure to track the access times of pages, which can be resource-intensive.

**High Overhead:** The overhead of maintaining access times for each page can be significant, especially in systems with a large number of pages.

Comparison and Trade-offs:

**Optimality:** LRU is generally considered more optimal than FIFO because it takes into account the actual access history of pages rather than just the order of arrival. However, implementing a true LRU algorithm can be complex and resource-intensive.

**Overhead:** FIFO has lower overhead compared to LRU since it does not require tracking access times for each page. However, this simplicity comes at the cost of potentially poorer performance.

**Belady's Anomaly:** FIFO can suffer from Belady's anomaly, where increasing the number of frames can paradoxically increase the number of page faults. LRU does not suffer from this anomaly.

**Adaptability:** LRU is more adaptable to varying access patterns since it considers the actual access history of pages. FIFO, on the other hand, may perform poorly in scenarios with irregular access patterns.

**CODE:**

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

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

```
#define MAX_FRAMES 3 // Maximum number of page frames
```

```
#define MAX_PAGES 10 // Maximum number of pages in reference  
string
```

```
// Function prototypes

void fifo(int pages[], int n, int frames);
void lru(int pages[], int n, int frames);


int main() {
    int pages[MAX_PAGES];
    int n, frames;

    printf("Enter the number of pages: ");
    scanf("%d", &n);

    printf("Enter the page reference string: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &frames);

    printf("\nFIFO Page Replacement:\n");
    fifo(pages, n, frames);

    printf("\nLRU Page Replacement:\n");
    lru(pages, n, frames);
}
```

```

    return 0;
}

void fifo(int pages[], int n, int frames) {
    int frame[MAX_FRAMES];
    int page_faults = 0;
    int current_frame = 0;

    for (int i = 0; i < frames; i++) {
        frame[i] = -1; // Initialize frames as empty
    }

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;

        // Check if page is already in a frame
        for (int j = 0; j < frames; j++) {
            if (frame[j] == page) {
                found = 1;
                break;
            }
        }

        // Page fault: replace the oldest page
    }
}

```

```

    if (!found) {
        printf("Page %d caused a page fault.\n", page);
        frame[current_frame] = page;
        current_frame = (current_frame + 1) % frames; // Circular
queue
        page_faults++;
    }

    // Display current state of frames
    printf("Frames: ");
    for (int j = 0; j < frames; j++) {
        printf("%d ", frame[j]);
    }
    printf("\n");
}

printf("Total page faults: %d\n", page_faults);
}

```

```

void lru(int pages[], int n, int frames) {
    int frame[MAX_FRAMES];
    int page_faults = 0;
    int counter[MAX_FRAMES] = {0};

    for (int i = 0; i < frames; i++) {

```



```

    frame[i] = -1; // Initialize frames as empty
}

for (int i = 0; i < n; i++) {
    int page = pages[i];
    int found = 0;

    // Check if page is already in a frame
    for (int j = 0; j < frames; j++) {
        if (frame[j] == page) {
            found = 1;
            counter[j] = i + 1; // Update counter to indicate recent
access
            break;
        }
    }

    // Page fault: replace the least recently used page
    if (!found) {
        printf("Page %d caused a page fault.\n", page);
        int min_counter = counter[0];
        int min_index = 0;
        for (int j = 1; j < frames; j++) {
            if (counter[j] < min_counter) {
                min_counter = counter[j];
            }
        }
        // Replace the least recently used page
        int lru_index = min_index;
        frame[lru_index] = page;
        counter[lru_index] = i + 1;
    }
}

```

```

        min_index = j;
    }
}
frame[min_index] = page;
counter[min_index] = i + 1; // Update counter to indicate
recent access
page_faults++;
}

// Display current state of frames
printf("Frames: ");
for (int j = 0; j < frames; j++) {
    printf("%d ", frame[j]);
}
printf("\n");
}

printf("Total page faults: %d\n", page_faults);
}

```

## OUTPUT:

```
Enter the number of pages: 10
Enter the page reference string: 1 2 3 4 1 2 5 1 2 3
Enter the number of frames: 3
```

### FIFO Page Replacement:

```
Page 1 caused a page fault.
Frames: 1 -1 -1
Page 2 caused a page fault.
Frames: 1 2 -1
Page 3 caused a page fault.
Frames: 1 2 3
Page 4 caused a page fault.
Frames: 4 2 3
Page 1 caused a page fault.
Frames: 4 1 3
Page 2 caused a page fault.
Frames: 4 1 2
Page 5 caused a page fault.
Frames: 5 1 2
Frames: 5 1 2
Frames: 5 1 2
Page 3 caused a page fault.
Frames: 5 3 2
Total page faults: 8
```

### LRU Page Replacement:

```
Page 1 caused a page fault.
Frames: 1 -1 -1
Page 2 caused a page fault.
Frames: 1 2 -1
Page 3 caused a page fault.
Frames: 1 2 3
Page 4 caused a page fault.
Frames: 4 2 3
Page 1 caused a page fault.
Frames: 4 1 3
Page 2 caused a page fault.
Frames: 4 1 2
Page 5 caused a page fault.
Frames: 5 1 2
Frames: 5 1 2
Frames: 5 1 2
Page 3 caused a page fault.
Frames: 3 1 2
Total page faults: 8
```

## **CONCLUSION:**

In summary, page replacement policies such as FIFO and LRU play a crucial role in virtual memory management by determining which pages to evict from memory when page faults occur. While FIFO is simple to implement and has low overhead, it may perform poorly in scenarios with irregular access patterns and can suffer from Belady's anomaly. LRU, on the other hand, provides better performance by considering the actual access history of pages, but it comes with higher implementation complexity and overhead. The choice between FIFO and LRU depends on factors such as the system's requirements, workload characteristics, and available resources.

## **EXPERIMENT NO:10**

Name: Saylee Shirke

Batch: S22

Roll no: 101

**AIM:** Write a program in C to do disk scheduling - FCFS, SCAN, C-SCAN

### **THEORY:**

File management and I/O (Input/Output) management are crucial components of operating systems responsible for handling the storage and retrieval of data from disk storage devices efficiently. Disk scheduling algorithms play a vital role in optimizing I/O operations by determining the order in which disk requests are serviced. Among the various disk scheduling algorithms, First-Come, First-Served (FCFS), SCAN, and C-SCAN are widely used. Let's delve into these concepts and understand how each algorithm works and its implications in disk management.

#### **File Management:**

File management involves organizing and managing files on disk storage devices to facilitate efficient storage, retrieval, and manipulation of data. It encompasses various operations such as file creation, deletion, access control, and directory management. A file system is responsible for managing these operations and maintaining the structure and integrity of files and directories.

#### **I/O Management:**

I/O management deals with managing Input/Output operations between the CPU, memory, and I/O devices such as disks, printers, and network interfaces. Disk I/O operations are particularly significant as they involve relatively slow mechanical devices compared to the CPU and memory. Disk scheduling algorithms are employed to optimize the order in which disk requests are serviced to minimize seek time and maximize disk throughput.

#### **Disk Scheduling Algorithms:**

## 1. First-Come, First-Served (FCFS):

FCFS is the simplest disk scheduling algorithm, where disk requests are serviced in the order they arrive. It operates on the principle of fairness, as requests are processed based on their arrival times without any consideration for their locations on the disk.

### Mechanism:

When a disk request arrives, it is added to the end of the request queue.

The disk scheduler services requests in the order they are queued.

Each request is processed sequentially, starting from the innermost track to the outermost track of the disk.

### Implications:

FCFS is easy to implement and ensures fairness in servicing requests.

However, it may lead to longer seek times, especially if requests are scattered across the disk, resulting in poor disk performance.

## 2. SCAN (Elevator) Algorithm:

SCAN, also known as the elevator algorithm, simulates the movement of an elevator moving up and down a building. It services requests in one direction until reaching the end of the disk, then reverses direction and continues servicing requests in the opposite direction.

### Mechanism:

The disk head starts from one end of the disk and moves towards the other end while servicing requests along its path.

When it reaches the end of the disk, it reverses direction and starts moving towards the opposite end, servicing requests along the way.

SCAN prevents the disk head from unnecessarily traversing the entire disk by changing direction at the disk boundaries.

Implications:

SCAN reduces the average seek time by prioritizing requests closer to the current position of the disk head.

However, it may result in starvation for requests located at the extremes of the disk if there is a continuous stream of requests in one direction.

### 3. C-SCAN (Circular SCAN) Algorithm:

C-SCAN is an enhancement of the SCAN algorithm that overcomes the potential starvation issue by treating the disk as a circular buffer. After reaching one end of the disk, the disk head jumps to the other end without servicing requests, ensuring fairness in request servicing.

Mechanism:

Similar to SCAN, the disk head moves in one direction, servicing requests until reaching the end of the disk.

Instead of reversing direction immediately, C-SCAN jumps to the other end of the disk without servicing requests.

It then continues servicing requests in the same direction, preventing starvation for requests located at the disk boundaries.

Implications:

C-SCAN provides fairness in request servicing by preventing starvation for requests at the extremes of the disk.

However, it may result in slightly higher average seek times compared to SCAN due to the jump between disk ends.

**CODE:**

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

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

```
#define MAX_REQUESTS 100 // Maximum number of disk requests
```

```
#define MAX_CYLINDERS 200 // Maximum number of cylinders  
on the disk
```

```
// Function prototypes
```

```
void fcfs(int requests[], int n, int initial_position);
```

```
void scan(int requests[], int n, int initial_position, int cylinders);
```

```
void c_scan(int requests[], int n, int initial_position, int cylinders);
```

```
int main() {
```

```
    int requests[MAX_REQUESTS];
```

```
    int n, initial_position, cylinders;
```

```
    printf("Enter the number of disk requests: ");
```

```
    scanf("%d", &n);
```

```
    printf("Enter the disk requests: ");
```

```
    for (int i = 0; i < n; i++) {
```

```
        scanf("%d", &requests[i]);
```

```
    }
```

```
    printf("Enter the initial position of the disk head: ");
```

```
    scanf("%d", &initial_position);
```



```
printf("Enter the total number of cylinders on the disk: ");
scanf("%d", &cylinders);

printf("\nFirst-Come, First-Served (FCFS):\n");
fcfs(requests, n, initial_position);

printf("\nSCAN:\n");
scan(requests, n, initial_position, cylinders);

printf("\nC-SCAN:\n");
c_scan(requests, n, initial_position, cylinders);

return 0;
}
```

```
void fcfs(int requests[], int n, int initial_position) {
    int total_seek_time = 0;

    printf("Sequence of servicing requests:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
        total_seek_time += abs(requests[i] - initial_position);
        initial_position = requests[i];
    }
```

```

printf("\nTotal seek time: %d\n", total_seek_time);
}

void scan(int requests[], int n, int initial_position, int cylinders) {
    int total_seek_time = 0;
    int direction = 1; // 1 for right, -1 for left

    printf("Sequence of servicing requests:\n");

    // Sort requests to service in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }

    // Find the index where the disk head should reverse direction
    int reverse_index = 0;
    while (reverse_index < n && requests[reverse_index] <
initial_position) {

```

```

        reverse_index++;
    }

    // Service requests in one direction
    for (int i = reverse_index - 1; i >= 0; i--) {
        printf("%d ", requests[i]);
        total_seek_time += abs(requests[i] - initial_position);
        initial_position = requests[i];
    }

    // Service requests in the reverse direction
    for (int i = reverse_index; i < n; i++) {
        printf("%d ", requests[i]);
        total_seek_time += abs(requests[i] - initial_position);
        initial_position = requests[i];
    }

    printf("\nTotal seek time: %d\n", total_seek_time);
}

void c_scan(int requests[], int n, int initial_position, int cylinders) {
    int total_seek_time = 0;

    printf("Sequence of servicing requests:\n");

```

```
// Sort requests to service in ascending order
```

```
for (int i = 0; i < n - 1; i++) {  
    for (int j = 0; j < n - i - 1; j++) {  
        if (requests[j] > requests[j + 1]) {  
            int temp = requests[j];  
            requests[j] = requests[j + 1];  
            requests[j + 1] = temp;  
        }  
    }  
}
```

```
// Service requests in one direction
```

```
for (int i = 0; i < n && requests[i] < initial_position; i++) {  
    printf("%d ", requests[i]);  
    total_seek_time += abs(requests[i] - initial_position);  
    initial_position = requests[i];  
}
```

```
// Jump to the beginning of the disk
```

```
printf("%d ", 0);  
total_seek_time += initial_position;
```

```
// Service requests in the reverse direction
```

```
for (int i = n - 1; i >= 0 && requests[i] >= initial_position; i--) {  
    printf("%d ", requests[i]);
```

```

        total_seek_time += abs(requests[i] - initial_position);
        initial_position = requests[i];
    }

    printf("\nTotal seek time: %d\n", total_seek_time);
}

```

## OUTPUT:

```

Enter the number of disk requests: 5
Enter the disk requests: 98 183 37 122 14
Enter the initial position of the disk head: 53
Enter the total number of cylinders on the disk: 200

First-Come, First-Served (FCFS):
Sequence of servicing requests:
98 183 37 122 14
Total seek time: 469

SCAN:
Sequence of servicing requests:
37 14 98 122 183
Total seek time: 208

C-SCAN:
Sequence of servicing requests:
14 0 183
Total seek time: 222

...Program finished with exit code 0
Press ENTER to exit console.

```

## CONCLUSION:

File management and I/O management are essential components of operating systems, responsible for efficient storage and retrieval of data from disk storage devices. Disk scheduling algorithms such as FCFS, SCAN, and C-SCAN play a crucial role in optimizing disk I/O operations by determining the order in which disk requests are

served. While FCFS provides simplicity and fairness, SCAN and C-SCAN aim to minimize seek times and prevent starvation for requests located at the extremes of the disk. The choice of disk scheduling algorithm depends on factors such as disk workload characteristics, system requirements, and performance considerations.