# D. Y. Patil college of Engineering, Akurdi, Pune-44

# Department of Information Technology

**Operating System Lab**

**SUBJECT CODE: 314446**

**CLASS: THIRD YEAR (SEMESTER-I)**

**LAB MANUAL**

**Operating System Lab**

**SUBJECT CODE: 314446**

**CLASS: THIRD YEAR**

**SEMESTER - I LAB MANUAL**

## D. Y. PATIL COLLEGE OF ENGINEERING,

## AKURDI, PUNE-44

## DEPARTMENT OF INFORMATION TECHNOLOGY

**D. Y. Patil college of Engineering, Akurdi, Pune-44**

**Department of Information Technology**

**Vision of the Institute:**

“Empowerment through Knowledge”.

**Mission of the Institute:**

To educate the students to transform them as professionally competent and quality conscious engineers by providing conducive environment for teaching, learning, and overall personality development, culminating the institute into an international seat of excellence.

**Vision of the Department**

Developing globally competent IT professional for sustainable growth of humanity.

**Mission of the Department**

**M1** Build a strong foundation and techniques for problem-solving and inculcate communication skills as an integral component of Information Technology

**M2** Develop competency skills in the faculty members and students to serve the societal challenges and needs in lieu of its multidisciplinary applications in the field of Information Technology

**M3** Encourage development of strong technical skills and knowledge and to encourage studentsto undergo research in the field of Information Technology

**M4** Nurture students to become ethical and committed lifelong IT professionals

**M5** Empower students with strong decision-making skills and technical competency to accomplish start-up ideas in the field of IT Engineering

## Program Educational Objectives (PEOs) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune :

1. **Core Competency:** To provide graduates with a solid foundation in Mathematics, Science, Engineering fundamentals required to solve complex Software Engineering Problem.
2. **Breadth:** To impart the knowledge and skills in the field of Information Technology; and to comprehend, analyze, design and create novel products and solutions for the real- time and Complex Engineering problems of any domain with innovative approaches.
3. **Professionalism:** To inculcate in graduates, professional and ethical values, effective communication skills, teamwork, multidisciplinary approach, and ability to relate engineering issues in broader social context.
4. **Learning Environment:** To provide graduates with an academic environment that makes them aware of excellence in leadership, presentation, time management and ethics leading them to become responsible and competent professionals prepared to address challenges in the field of IT at global level.
5. **Attainment:** To empower graduates with an attitude and skills of Research, Entrepreneur and Higher education in the field of Information Technology.

## Program Specific Outcomes (PSO) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune:

1. Apply design methodologies, application development tools, engineering skills in Software Engineering Domains and IT Application areas like Cloud Computing, Software Testing, Mobile App Development, etc.
2. Aspire to pursue Higher Education in the specialized fields of IT Engineering and management programs like Data science, Cyber Security, Artificial Intelligence, etc.
3. Formulate decision-making skills, IT Engineering skills, and knowledge to implement start-up ideas as an entrepreneur in the fields such as Cyber Security, Mobile ApplicationDevelopment, etc.
4. Devise to design, implement, and evaluate IT based software systems to serve the needs of society or IT industries at large.

# PROGRAM OUTCOME (PO)

|  |  |  |
| --- | --- | --- |
| **Students are expected to know and be able to–** | | |
| **PO1** | **Engineering knowledge** | An ability to apply knowledge of mathematics, computing, science, engineering and technology. |
| **PO2** | **Problem analysis** | An ability to define a problem and provide a systematic solution with the help of conducting experiments, analyzing the problem and interpreting  the data. |
| **PO3** | **Design / Developmentof Solutions** | An ability to design, implement, and evaluate software or a software  /hardware system, component, or process to meet desired needs within realistic constraints. |
| **PO4** | **Conduct Investigation of Complex**  **Problems** | An ability to identify, formulate, and provide essay schematic solutionsto complex engineering /Technology problems. |
| **PO5** | **Modern Tool Usage** | An ability to use the techniques, skills, and modern engineering technology tools, standard processes necessary for practice as a IT  professional. |
| **PO6** | **The Engineer and Society** | An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer- based systems with necessary constraints and assumptions. |
| **PO7** | **Environment and Sustainability** | An ability to analyze and provide solution for the local and global impactof information technology on individuals, organizations and society. |
| **PO8** | **Ethics** | An ability to understand professional, ethical, legal, security and social issues and responsibilities. |
| **PO9** | **Individual and Team Work** | An ability to function effectively as an individual or as a team memberto accomplish a desired goal(s). |
| **PO10** | **Communication Skills** | An ability to engage in life-long learning and continuing professional development to cope up with fast changes in the technologies /tools with the help of electives, profession along animations and extra- curricular  activities. |
| **PO11** | **Project Management and**  **Finance** | An ability to communicate effectively in engineering community at large by means of effective presentations, report writing, paper publications, demonstrations. |
| **PO12** | **Life-long Learning** | An ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessaryfor  professional practice. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | **Teaching Scheme:** | | Credits: 02 | |  | | --- | | **Examination Scheme:** | |
| |  | | --- | | Practical:04 Hours/Week | | |  | | --- | | PR:25 Marks | | TW:25 Marks | |

|  |
| --- |
| **Prerequisites:**   1. C Programming 2. Fundamentals of Data Structure |
| **Course Objectives:**   |  | | --- | | 1. To introduce and learn Linux commands required for administration. 2. To learn shell programming concepts and applications. 3. To demonstrate the functioning of OS basic building blocks like processes, threads under the LINUX. 4. To demonstrate the functioning of OS concepts in user space like concurrency control (process synchronization, mutual exclusion), CPU Scheduling, Memory Management and Disk Scheduling in LINUX. 5. To demonstrate the functioning of Inter Process Communication under LINUX. 6. To study the functioning of OS concepts in kernel space like embedding the system call in any LINUX kernel. | |
| **Course Outcomes:**  By the end of the course, students should be able to   1. Apply the basics of Linux commands. 2. Build shell scripts for various applications. 3. Implement basic building blocks like processes, threads under the Linux. 4. Develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling, Memory Management and Disk Scheduling in Linux. 5. Develop system programs for Inter Process Communication in Linux. |

# List of Assignments

|  |  |
| --- | --- |
| **Sr. No.** | **Tile of Assignment** |
| 1 | **A.** Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.  **B.** Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit |
| 2 | Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.  **A.** Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.  **B.** Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order. |
| 3 | Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time. |
| 4 | **A.** Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex.  **B.** Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority. |
| 5 | Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm. |
| 6 | Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three. |
| 7 | Inter process communication in Linux using following.  **A. FIFOS:** Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays onstandard output.  **B. Inter-process Communication using Shared Memory using System V.** Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen. |
| 8 | Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle. |

|  |  |
| --- | --- |
| **Sr. No.** | **Extra Assignment** |
| 9 | **A.** Memory management with Fixed Partitioning technique (MFT)  **B.** Memory management with Variable Partitioning technique (MVT) |

# RUBRICS FOR LABORATORY ASSESSMENT

### Attendance

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Assessment  Outcome | Poor (1) | Satisfactory(2) | Good (3) | Very Good (4) | Excellent (5) |
| Dimensions |
| 1.Attendance with Involvement of Student (5M ) | Passive observer | Very little involvement | Good Involvement in performing  experiment | Individual Involvement in performing experiment | Individual and self - Involvement in performing experiment |

* 1. **Viva**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Assessment  Outcome | Poor (1) | Satisfactory(2) | Good (3) | Very Good (4) | Excellent (5) |
| Dimensions |
| 1.Preparation and Basic Knowledge  (5M ) | No preparation | Little Knowledge | Prepared Well | Very well prepared | Advance Knowledge |
| 2.Program development and execution (5M) | Not Executed | Partially executed | Executed | Executed without additional  modification | Executed with additional modification |
| 3.Punctuality and Ethics (5M) | Attendance Below 50% and not following the lab instructions | Attendance 50% to 75%  And sometimes copies the program | Regular attendance 75-00% and follows the instruction and try to perform on his own | Regular attendance 80-  90% and follows the instruction and try to perform on his own | 90-100 % attendance, follows all instructions and execute the program on his own |

### Presentation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Assessment  Outcome | Poor (1) | Satisfactory(2) | Good (3) | Very Good (4) | Excellent (5) |
| Dimensions |
| Journal Presentation (5M) | Not  Prepared | Incomplete | Completed  documentation | well documented | Very well  documented |

**Outcome:** Student will be able to

Apply knowledge to real life examples and develop practical approach

Design Basic Application.

Note: Students with poor marks should repeat the assignment

**Assignment No-1A**

**Aim:** Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

**Theory:** The Linux command is Linux utility. The commands are executed on the **Linux terminal**. Maximum tasks can be done by executing commands. The terminal is a command-line interface to interact with the operating system, *and is* ***case-sensitive****.*

Linux provides a powerful command-line interface compared to other operating systems such as Windows and MacOS. We can do basic work and advanced work through its terminal. We

can do some basic tasks such as creating a file, deleting a file, moving a file, and more. In addition, we can also perform advanced tasks such as administrative tasks (including package installation, user management), networking tasks (ssh connection), security tasks, and many more.

Linux terminal is a user-friendly terminal as it provides various support options. To open the Linux terminal, press "**CTRL + ALT + T**" keys together, and execute a command by pressing the '**ENTER**' key.

**User defined variables (UDV)**

To define UDV use following syntax

Syntax:

variable name=value

'value' is assigned to given 'variable name' and Value must be on right side = sign.

Example:

To define variable called n having value 10

**$ n=10**

***Shell Arithmetic*** *Use to perform arithmetic operations.*

Syntax:

expr op1 math-operator op2

Examples:

$ expr 1 + 3

$ expr 2 - 1

$ expr 10 / 2

**The read Statement**- Use to get input (data from user) from keyboard and store (data) to variable.

**Syntax:**

read variable1, variable2,...variableNtest command or [ expr ] - test command or [ expr ] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

**Syntax:**

test expression OR [ expression ]

**Example:**

Following script determine whether given argument number is positive.if test $1 -gt 0 then

echo "$1 number is positive"

fi

***test or [ expr ] works with***

***1.Integer ( Number without decimal point)***

***2.File types*** Shell also test for file and directory types

Test Meaning

|  |  |
| --- | --- |
| -s file | Non empty file |
| -f file | Is File exist or normal file and not a directory |
| -d dir | Is Directory exist and not a file |
| -w file | Is writeable file |

-r file Is read-only file

-x file Is file is executable

**Loops in Shell Scripts**

**Bash supports:** 1) for loop 2) while loop

***while* :**The syntax of the ***while*** is:

while **test-commands**

**do**

**commands**

done Execute **commands** as long as **test-commands** has an exit status of zero.

***for* :**The syntax of the ***for*** is:

for variable in list

do

**commands**

done

Each white space-separated word in list is assinged to variable in turn and commands executed until list is exhausted.

**The case Statement-**The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

Syntax:

case $variable-name in

pattern1) command

...

..

command;;

pattern2) command

...

..

command;;

patternN) command

...

..

command;;

\*) command

...

..

command;;

esac

**Sample Commands:** *'cd'*

*'ls'*

*'cp'*

*‘mkdir’*

*‘mv’*

*‘rm’*

*‘more’,’less’*

*‘man’*

*‘touch’*

*‘cat’*

*‘sort’*

*‘sed’*

*‘who’*

*‘pwd’*

*‘ps’*

*‘chmod’*

*‘cut’*

**Conclusion/Result:**

Different commands studied in this experiment gives vast functioning ability to start with shell scripting and perform different tasks. All Commands have been executed successfully.

**Assignment No-1B**

**Aim:** Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

**Theory:**

**The shell**

The shell acts as an interface between the user and the kernel. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt .The adept user can customize his/her own shell and users can use different shells on the same machine. Unix system offers verity of shells like 1) Bourne shell 2) c shell 3) Korn shell 4) Bash shell (very powerful & recommended for use, Linux default shell)

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

**Shell Script**

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands), then you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as shell script. Shell Script is series of command written in plain text file. This manual is meant as a brief introduction to features found in Bash.

**Sample Code:**

#!/bin/bash

createaddressbook()

{

echo

if [ -e addressbook.txt ]

then

echo "address book is already created"

else

touch addressbook.txt

echo " address book created"

fi

}

insertrecord()

{

while true

do

echo -e "\n To add a record to book, please enter the information in following format : "

echo -e "\n lastname,firstname,email,mobile no."

echo -e "\n Eg :

Venkatesh,Vaishnav,vaishnavvenkatesh@gmail.com,8600278899" echo "to quit press 'q'"

read newrecord

if [ "$newrecord" == 'q' ]

then

break

fi

echo $newrecord >> addressbook.txt

echo "record inserted successfully"

echo -e "\n"

done

}

viewrecords()

{

cat addressbook.txt

}

searchrecord()

{

while true

do

echo "To search a record, enter any string :"

echo "eg : last name or email address(case sensitive)" echo "for exit enter 'q'"

read sayrec

echo "record of \"$sayrec\" : "

if [ "$sayrec" == 'q' ]

then

break

fi

echo "record of \"$sayrec\" : "

grep "$sayrec" addressbook.txt

returnstatus=`echo $?`

if [ $returnstatus -eq 1 ]

then

echo "no record found"

fi

done

}

modifyrecord()

{

while true

do

echo "To modify a record, enter any string :"

echo "eg : last name or email address(case sensitive)" echo "for exit enter 'q'"

read mrecord

if [ "$mrecord" == 'q' ]

then

break

fi

echo -e "\n listing records for \"$mrecord\" : "

grep -i -n "$mrecord" addressbook.txt

returnstatus=`echo $?`

if [ $returnstatus -eq 1 ]

then

echo "no record found"

else

echo "Enter the line number (the first number of the entry) that you would like to edit : "

read linenumber

echo

for line in `grep -n "$mrecord" addressbook.txt`

do

number=`echo "$line" | cut -c 1`

if [ $number -eq $linenumber ]

then

echo "what would you like to change it to? use the format : " echo "last name,first name,email id,mobile no."

read erecord

linechange="${linenumber}s"

sed -i -e "$linechange/.\*/$erecord/" addressbook.txt echo -e "\n address book updated successfully"

fi

done

fi

done

}

deleterecord()

{

while true

do

echo "To remove a record, enter any string :"

echo "eg : last name or email address(case sensitive)" echo "for exit enter 'q'"

read drecord

if [ "$drecord" == 'q' ]

then

break

fi

echo -e "\n listing records for \"$drecord\" : "

grep -i -n "$drecord" addressbook.txt

retstatus=`echo $?`

if [ $retstatus -eq 1 ]

then

echo "no record found"

else

echo "Enter the line number (the first number of the entry) that you would like to edit : "

read linenum

echo

for line in `grep -n "$drecord" addressbook.txt`

do

num=`echo "$line" | cut -c 1`

if [ $num -eq $linenum ]

then

lineremove="${linenum}d"

sed -i -e "$lineremove" addressbook.txt

echo -e "\n record deleted successfully"

fi

done

fi

done

}

lastcharoffile=`tail -c 1 addressbook.txt`

if [ -n "$lastcharoffile" ]

then

echo >> addressbook.txt

fi

echo "address book?"

echo "enter your choice"

while true

do

echo "1: create address book"

echo "2: insert a record"

echo "3: view records"

echo "4: modify a record"

echo "5: delete a single record"

echo "6: search a record"

echo "7: exit"

echo

‘

read ch

case $ch in

1)createaddressbook;;

2)insertrecord;;

3)viewrecords;;

4)modifyrecord;;

5)deleterecord;;

6)searchrecord;;

7)exit;;

esac

done

**Conclusion/Result:**

Thus in shell script we can write series of commands and execute as a single program***.***

**Assignment No-2A**

**Aim:** Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

**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() systemcall. 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.

**Let us take an example:**

int main()

{

printf(“Before Forking”);

fork();

printf(“After Forking”);

return 0;

}

If the call to fork() is executed successfully,

Unix will

Make two identical copies of address spaces, one for the parent and the other for the child.

Both processes will start their execution at the next statement following the fork() call. If we run this program, we might see the following on the screen:

* Before Forking
* After Forking
* After Forking

Here print() statement after fork()system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

**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.

**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:**

On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child’s exit status. The term zombie process derives from the common definition of zombie — an undead person. In the term’s metaphor, the child process has “died” but has not yet been “reaped”. Also, unlike normal processes, the kill command has no effect on a zombie process. 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**:

In orphan process is a computer process whose parent process has finished or terminated, though it remains running itself. In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically. Even though technically the process has the init process as its parent, it is still called an orphan process since the process that originally created it no longer exists.

**Daemon Process**:

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

**Sample Code:**

1. FORK SYSTEM CALL

# include <stdio.h>

# include <sys/types.h> #

include <unistd.h>

int main()

{

printf("Statement 1 : %d\n",getpid()); fork();

fork();

fork();

printf("statement 2 %d\n",getpid());

}

# include <stdio.h>

# include <sys/types.h> #

include <unistd.h>

void ChildProcess(); /\* child process prototype \*/ void

ParentProcess(); /\* parent process prototype \*/ int main()

{

pid\_t pid; pid =

fork();

if (pid == 0)

ChildProcess();

else

ParentProcess();

return 0;

}

void ChildProcess()

{

printf("I am Child Process\n");

}

void ParentProcess()

{

printf("I am Parent Process\n");

}

1. **WAIT SYSTEM CALL**

#include<stdio.h>

#include<sys/wait.h>

#include<unistd.h>

int main()

{

if (fork()== 0)

printf("HC: hello from child\n");

else

{

printf("HP: hello from parent\n");

wait(NULL);

printf("CT: child has terminated\n");

}

printf("Bye\n");

return 0;

}

1. **Orphan State**

# include <stdio.h>

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

int main()

{

pid\_t id; id = fork(); if(id == 0 )

{

sleep(30);

printf("I am child Process : %d\n",id);

}

else

{

printf("I am Parent Process : %d\n",getpid());

}

}

1. **Zombie state**

# include <stdio.h>

# include <sys/types.h>

# include <unistd.h>

#include <stdlib.h>

int main()

{

pid\_t id;

id = fork();

if(id == 0 )

{

exit(0);

}

else

{

sleep(50);

printf("Parent process id Active Now");

//printf("I am Parent Process : %d\n",getpid());

}

return 0;

}

**Conclusion/Result:**

We have created new process using fork() call. Also implemented wait() system call.

**Assignment No-2B**

**Aim:** Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

**Theory:**

**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.

**exec family:**

1. **execl() and execlp():**

**execl():** It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL.

e.g. **execl("/bin/ls", "ls", "-l", NULL);**

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

e.g. **execlp("ls", "ls", "-l", NULL);**

1. **execv() and execvp():**

**execv():** It does same job **as** execl() except that command line arguments can be passed to it inthe form of an array of pointers to string.

e.g. **char \*argv[] = ("ls", "-l", NULL);execv**("/bin/ls", argv);

**execvp**(): It does same job expect that it will use environment variable PATH to determinewhich executable to process. Thus a fully qualified path name would not have to be used.

e.g. **execvp("ls", argv);**

1. **execve( ):**

**eg: int execve(const char \*filename, char \*const argv[ ], char \*const envp[ ]);**

It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form: argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:int main(int argc, char \*argv[ ] , char \*envp[ ])] execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

**Sample Code:**

**Code and Output :**

**1) Execvp()**

exec1.c

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

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

{

printf("Into exec1.c file having process id : %d\n",getpid()); char \*args[] = {"./exec2", NULL};

execvp(args[0],args);

printf("exec1.c file end\n");

return 0;

}

exec2.c

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main(int arg,char \*argv[])

{

printf("Into exec2.c file having process id : %d\n",getpid());

printf("exec2.c file end\n");

return 0;

}

**2) Execlp()**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main()

{

char \*file\_name="ls";

char \*arg1="-a";

char \*arg2="-s";

execlp(file\_name,file\_name,arg1,arg2,NULL);

return 0;

}

**3) Execl()**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h> int

main()

{

char \*bin\_path="/bin/ls"; char

\*arg1="-a";

char \*arg2="-s";

execl(bin\_path,bin\_path,arg1,arg2,NULL);

return 0;

**Conclusion/Result:**

Thus, we know how to use exec system calls and implemented binary search program.

**Assignment No-3**

**Aim:** Implement the C program for CPU Scheduling Algorithms: Shortest Job First(Preemptive) and Round Robin with different arrival time.

**Theory:**

**CPU Scheduling** is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution.

**Important algorithms**

1. First Come First Serve (FCFS)

2. Shortest-Job-First (SJF) Scheduling

3. Shortest Remaining Time

4. Priority Scheduling

5. Round Robin Scheduling

6. Multilevel Queue Scheduling

**Shortest Job First (Preemptive) :**

In shortest job first scheduling algorithm, the processor selects the waiting process with the smallest execution time to execute next. In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Advantages

• SJF is frequently used for long term scheduling.

• It reduces the average waiting time over FIFO (First in First Out) algorithm.

• SJF method gives the lowest average waiting time for a specific set of processes.

• It is appropriate for the jobs running in batch, where run times are known in advance.

• For the batch system of long-term scheduling, a burst time estimate can be obtained from the job description.

• For Short-Term Scheduling, we need to predict the value of the next burst time.

• Probably optimal with regard to average turnaround time.

Disadvantages

• SJF can’t be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst.

• This algorithm may cause very long turnaround times or starvation.

• Requires knowledge of how long a process or job will run.

**Round Robin Algorithm:**

A round-robin is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called time **quantum or time slice**. Some of the round-robin processes are pre-empted if it executed in a given time slot, while the rest of the processes go back to the ready queue and wait to run in a circular order with the scheduled time slot until they complete their task. It removes the starvation for each process to achieve CPU scheduling by proper partitioning of the CPU**.** It is a pre-emptive algorithm. It shares an equal time interval between all processes to complete their task. It is a starvation free CPU scheduling algorithm. Hence it is known as the fairest and simple algorithm

**Advantages**

1. It does not face any starvation issues or convoy effect.

2. Each process gets equal priority to the fair allocation of CPU.

3. It is easy to implement the CPU Scheduling algorithm.

4. Each new process is added to the end of the ready queue as the next process's arrival time is reached.

5. Each process is executed in circular order that shares a fixed time slot or quantum.

6. Every process gets an opportunity in the round-robin scheduling algorithm to reschedule after a given quantum period.

**Disadvantage**

1. If the time quantum is lower, it takes more time on context switching between the processes.

2. It does not provide any special priority to execute the most important process.

3. The waiting time of a large process is higher due to the short time slot.

**Sample Code:**

**1)Shortest Job First (Preemptive)**

**Code :**

#include <stdio.h>

int arrival\_t[100], burst\_t[100], ct[100],temp[100];

int main()

{

int i, smallest, count = 0,j, n;

double avg\_wt = 0, avg\_tat = 0, end;

printf("\nEnter the Total Number of Processes: ");

scanf("%d", &n);

printf("\nEnter Details of %d Processes", n);

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

{

printf("\nEnter Arrival Time for P%d: ",i+1);

scanf("%d", &arrival\_t[i]);

printf("Enter Burst Time for P%d: ",i+1);

scanf("%d", &burst\_t[i]);

temp[i]=burst\_t[i];

}

burst\_t[99] = 1000;

for(i = 0; count != n; i++)

{

smallest = 99;

for(j = 0; j < n; j++)

{

if(arrival\_t[j] <= i && burst\_t[j] < burst\_t[smallest] && burst\_t[j] > 0)

{

smallest = j;

}

}

burst\_t[smallest]--;

if(burst\_t[smallest] == 0)

{

count++;

ct[smallest]=i+1;

}

}

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

{

int TAT=ct[i]-arrival\_t[i];

avg\_tat+=TAT;

avg\_wt+=TAT-temp[i];

}

printf("Average Turn Arround Time: %lf\n",avg\_tat/n);

printf("Average Waiting time: %lf\n",avg\_wt/n);

return 0;

}

**2) Round Robin**

**Code :**

#include <stdio.h>

int main()

{

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10]; float average\_wait\_time, average\_turnaround\_time;

printf("\nEnter Total Number ofProcesses:\t");

scanf("%d", &limit);

x = limit;

for (i = 0; i < limit; i++)

{

printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

scanf("%d", &arrival\_time[i]);

printf("Burst Time:\t");

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum:\t");

scanf("%d", &time\_quantum);

printf("\nProcess ID\tBurst Time\t Turnaround Time\t Waiting Time\n");

for (total = 0, i = 0; x != 0;)

{

if (temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if (temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if (temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];

turnaround\_time = turnaround\_time + total - arrival\_time[i];

counter = 0;

}

if (i == limit - 1)

{

i = 0;

}

else if (arrival\_time[i + 1] <= total)

{

i++;

}

else

{

i = 0;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

return 0;

}

**Conclusion/Result:**

Here, we have studied preemptive SJF and round robin algorithms in details and implemented successfully**.**

**Assignment No. 4A:**

**Aim:** Thread synchronization using counting semaphores. Application to demonstrate: producer consumer problem with counting semaphores and mutex.

**Theory:**

The producer and consumer share a fixed-size buffer used as a queue. The producer’s job is to generate data and put this in the buffer. The consumer’s job is to consume the data from this buffer, one at a time.

How do you make sure that producer doesn’t try to put data in buffer when the buffer is full and consumer doesn’t try to consumer data when the buffer is empty?

When producer tries to put data into the buffer when it is full, it wastes cpu cycles. The same is true for consumer it tries to consumer from an empty buffer. It’s better that they go on sleep in these cases so that the scheduler can schedule another process.

Implementation:

Use two semaphores, one for buffer full, and one for buffer empty as a buffer index. Each time a producer queues an element into buffer, the producer posts to the empty buffer semaphore. This will increment its value, and if it were 0 it would wake up consumer threads waiting on that semaphore. The consumer thread that waits on that semaphore, upon waking up it will decrement the value of the semaphore. So empty buffer semaphore basically follows up with the buffer index.

The full buffer semaphore does the opposite. Consumer threads post to it when they dequeue from the buffer, which increments the semaphore value. And producer threads wait in them, which means they sleep if buffer is full, and decrement the semaphore value each time they add element to buffer.

A mutex is used to protect access to buffer. The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Sample Code:

//PRODUCER CONSUMER PROBLEM

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

void \*writer\_thr(int temp);

void \*reader\_thr(int temp);

sem\_t mutex;

sem\_t wrt;

int readcount=0,nwt,nrd;

void main()

{

long int i;

sem\_init(&mutex,0,1);

sem\_init(&wrt,0,1);

pthread\_t reader[100],writer[100];

printf("\n Enter number of readers:");

scanf("%d",&nrd);

printf("\n Enter number of writers:");

scanf("%d",&nwt);

for(i=1;i<=nwt;i++)

{

pthread\_create(&writer[i],NULL,(void \*)writer\_thr,(int \*)i);

pthread\_join(writer[i],NULL);

}

for(i=1;i<=nrd;i++)

{

pthread\_create(&reader[i],NULL,(void \*)reader\_thr,(int \*)i);

}

for(i=1;i<=nrd;i++)

{

pthread\_join(reader[i],NULL);

}

sem\_destroy(&wrt);

sem\_destroy(&mutex);

}

void \*reader\_thr(int temp)

{

printf("\n Reader %d is trying to enter database for reading.",temp);

sem\_wait(&mutex);

readcount++;

if(readcount==1)

sem\_wait(&wrt);

sem\_post(&mutex);

printf("\nReader %d is now reading in database.",temp);

sem\_wait(&mutex);

readcount--;

if(readcount==0)

sem\_post(&wrt);

sem\_post(&mutex);

printf("\nReader %d has left the database.\n",temp);

sleep(3);

}

void \*writer\_thr(int temp)

{

printf("\nWriter %d is trying to enter database for modifying data",temp);

sem\_wait(&wrt);

printf("\n Writer %d is writing in database.",temp);

sleep(3);

printf("\n Writer %d is leaving the database.\n",temp);

sem\_post(&wrt);

}

**Conclusion/Result:**

We have ensured that producer do not produce data when buffer is full and consumer do not remove data when buffer is empty by implementing above code.

**Assignment No-4B**

**Aim**: Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader Writer problem with reader priority.

**Theory:**

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

**Sample Code:**

//READER WRITER PROBLEM

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

void \*writer\_thr(int temp);

void \*reader\_thr(int temp);

sem\_t mutex;

sem\_t wrt;

int readcount=0,nwt,nrd;

void main()

{

long int i;

sem\_init(&mutex,0,1);

sem\_init(&wrt,0,1);

pthread\_t reader[100],writer[100];

printf("\n Enter number of readers:");

scanf("%d",&nrd);

printf("\n Enter number of writers:");

scanf("%d",&nwt);

for(i=1;i<=nwt;i++)

{

pthread\_create(&writer[i],NULL,(void \*)writer\_thr,(int \*)i);

pthread\_join(writer[i],NULL);

}

for(i=1;i<=nrd;i++)

{

pthread\_create(&reader[i],NULL,(void \*)reader\_thr,(int \*)i);

}

for(i=1;i<=nrd;i++)

{

pthread\_join(reader[i],NULL);

}

sem\_destroy(&wrt);

sem\_destroy(&mutex);

}

void \*reader\_thr(int temp)

{

printf("\n Reader %d is trying to enter database for reading.",temp);

sem\_wait(&mutex);

readcount++;

if(readcount==1)

sem\_wait(&wrt);

sem\_post(&mutex);

printf("\nReader %d is now reading in database.",temp);

sem\_wait(&mutex);

readcount--;

if(readcount==0)

sem\_post(&wrt);

sem\_post(&mutex);

printf("\nReader %d has left the database.\n",temp);

sleep(3);

}

void \*writer\_thr(int temp)

{

printf("\nWriter %d is trying to enter database for modifying data",temp);

sem\_wait(&wrt);

printf("\n Writer %d is writing in database.",temp);

sleep(3);

printf("\n Writer %d is leaving the database.\n",temp);

sem\_post(&wrt);

}

**Conclusion/Result:**

This is proved with the help of above reader-writer problem that mutual exclusion can avoid synchronization issues.

**Assignment No- 5**

**Aim:** Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

**Theory:**

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

Let **‘n’** be the number of processes in the system and **‘m’** be the number of resources types. **Available:** A vector of length m. It shows number of available resources of each type. If Available[i] = k, then k instances of resource Ri are available.

**Max:** An n×m matrix that contain maximum demand of each process. If Max[i,j] = k, then process Pi can request maximum k instances of resource type Rj.

**Allocation:** An n×m matrix that contain number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then Pi is currently allocated k instances of resource type Rj. **Need:** An n×m matrix that shows the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Rj to complete the task.

Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ].

**Code:**

#include<stdio.h>

int required[100],allocate[100],need[100];

void main()

{

int n;

int max=12,allocateCount=0;

printf("Enter the number of processes:\n");

scanf("%d",&n);

printf("Enter the required resource for each process: \n");

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

{

printf("Requirement for Process %d:",i+1);

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

}

printf("\nEnter the Allocated resource for each process: \n");

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

{

printf("Allocated for process %d:",i+1);

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

allocateCount+=allocate[i];

need[i]=required[i]-allocate[i];

}

int available=max-allocateCount;

int count=n,sequence=0;

int ans[n],ind=0;

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

{

for(int j=0; j<n; j++)

{

if(need[j]!=0)

{

if(need[j]>available)

{

continue;

}

else

{

ans[ind++]=j+1;

sequence++;

count--;

available+=need[j];

need[j]=0;

}

}

}

}

if(sequence<n)

{

printf("\nThe system is in a unsafe state!!\n");

}

else

{

printf("\nThe system is in a safe state!!\n");

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

{

printf("p%d->",ans[i]);

}

}

}

**Conclusion/Result:**

Thus, Bankers algorithm is implemented to show deadlock avoidance.

**Assignment No.-6**

**Aim:** Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

**Theory:**

Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory. Page Replacement happens when a requested page is not present in the main memory and the available space is not sufficient for allocation to the requested page.

When the page that was selected for replacement was paged out, and referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm tries to select which pages should be replaced so as to minimize the total number of page misses.

**First In First Out (FIFO)**

This is the simplest page replacement algorithm. In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back.

When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time.

**Least Recently Used (LRU)**

Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.

In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

**Optimal Page Replacement**

Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. It is also known as OPT, clairvoyant replacement algorithm, or Belady’s optimal page replacement policy.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behaviour. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

**Sample Code:**

#include<stdio.h>

int n,nf;

int in[100];

int p[50];

int hit=0;

int i,j,k;

int pgfaultcnt=0;

void getData()

{

printf("\nEnter length of page reference sequence:"); scanf("%d",&n);

printf("\nEnter the page reference sequence:"); for(i=0; i<n; i++)

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

printf("\nEnter no of frames:");

scanf("%d",&nf);

}

void initialize()

{

pgfaultcnt=0;

for(i=0; i<nf; i++)

p[i]=9999;

}

int isHit(int data)

{

hit=0;

for(j=0; j<nf; j++)

{

if(p[j]==data)

{

hit=1;

break;

}

}

return hit;

}

int getHitIndex(int data)

{

int hitind;

for(k=0; k<nf; k++)

{

if(p[k]==data)

{

hitind=k;

break;

}

}

return hitind;

}

void dispPages()

{

for (k=0; k<nf; k++)

{

if(p[k]!=9999)

printf(" %d",p[k]);

}

}

void dispPgFaultCnt()

{

printf("\nTotal no of page faults:%d",pgfaultcnt); }

void fifo()

{

initialize();

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

for(k=0; k<nf-1; k++)

p[k]=p[k+1];

p[k]=in[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault");

}

dispPgFaultCnt();

}

void optimal()

{

initialize();

int near[50];

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

for(j=0; j<nf; j++)

{

int pg=p[j];

int found=0;

for(k=i; k<n; k++)

{

if(pg==in[k]) {

near[j]=k; found=1;

break; }

else

found=0; }

if(!found)

near[j]=9999; }

int max=-9999;

int repindex;

for(j=0; j<nf; j++) {

if(near[j]>max) {

max=near[j]; repindex=j;

}

}

p[repindex]=in[i]; pgfaultcnt++;

dispPages();

}

else

printf("No page fault"); }

dispPgFaultCnt();

}

void lru()

{

initialize();

int least[50];

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

for(j=0; j<nf; j++) {

int pg=p[j]; int found=0; for(k=i-1; k>=0; k--) {

if(pg==in[k]) {

least[j]=k; found=1;

break;

}

else

found=0; }

if(!found)

least[j]=-9999; }

int min=9999;

int repindex;

for(j=0; j<nf; j++)

{

if(least[j]<min) {

min=least[j]; repindex=j;

}

}

p[repindex]=in[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault!"); }

dispPgFaultCnt();

}

void lfu()

{

int usedcnt[100];

int least,repin,sofarcnt=0,bn; initialize();

for(i=0; i<nf; i++)

usedcnt[i]=0;

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

{

printf("\n For %d :",in[i]); if(isHit(in[i]))

{

int hitind=getHitIndex(in[i]); usedcnt[hitind]++;

printf("No page fault!"); }

else

{

pgfaultcnt++;

if(bn<nf)

{

p[bn]=in[i];

usedcnt[bn]=usedcnt[bn]+1; bn++;

}

else

{

least=9999;

for(k=0; k<nf; k++) if(usedcnt[k]<least) {

least=usedcnt[k]; repin=k;

}

p[repin]=in[i];

sofarcnt=0;

for(k=0; k<=i; k++) if(in[i]==in[k]) sofarcnt=sofarcnt+1; usedcnt[repin]=sofarcnt; } dispPages();

}

}

dispPgFaultCnt();

}

void secondchance()

{

int usedbit[50];

int victimptr=0;

initialize();

for(i=0; i<nf; i++)

usedbit[i]=0;

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

{

printf("\nFor %d:",in[i]); if(isHit(in[i]))

{

printf("No page fault!"); int hitindex=getHitIndex(in[i]); if(usedbit[hitindex]==0) usedbit[hitindex]=1; }

else

{

pgfaultcnt++;

if(usedbit[victimptr]==1) {

do

{

usedbit[victimptr]=0; victimptr++;

if(victimptr==nf)

victimptr=0; }

while(usedbit[victimptr]!=0); }

if(usedbit[victimptr]==0) {

p[victimptr]=in[i]; usedbit[victimptr]=1; victimptr++;

}

dispPages();

}

if(victimptr==nf)

victimptr=0;

}

dispPgFaultCnt();

}

int main()

{

int choice;

while(1)

{

printf("\nPage Replacement Algorithms\n1.Enter data\n2.FIFO\n3.Optimal\n4.LRU\n5.LFU\n6.Second Chance\n7.Exit\nEnter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1: getData();

break;

case 2: fifo();

break;

case 3:optimal();

break;

case 4: lru();

break;

case 5: lfu();

break;

case 6: secondchance();

break;

default: return 0;

break;

}

}

}

**Conclusion/Result:**

Different Page replacement algorithms were developed and can be used based on their pros and cons as implemented in this assignment.

**Assignment No. 7A:**

**Inter process communication in Linux using following.**

**Aim:** FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

**Theory:**

We used one pipe for one-way communication and two pipes for bi-directional communication. Does the same condition apply for Named Pipes. The answer is no, we can use single named pipe that can be used for two-way communication (communication between the server and the client, plus the client and the server at the same time) as Named Pipe supports bi-directional communication.

Another name for named pipe is **FIFO (First-In-First-Out)**. Let us see the system call (mknod()) to create a named pipe, which is a kind of a special file.

int mknod(const char \*pathname, mode\_t mode, dev\_t dev);

This system call would create a special file or file system node such as ordinary file, device file, or FIFO. The arguments to the system call are pathname, mode and dev. The pathname along with the attributes of mode and device information. The pathname is relative, if the directory is not specified it would be created in the current directory. The mode specified is the mode of file which specifies the file type such as the type of file and the file mode as mentioned in the following tables. The dev field is to specify device information such as major and minor device numbers.

The communication between pipes is meant to be unidirectional. Pipes were restricted to one-way communication in general and need at least two pipes for two-way communication. Pipes are meant for inter-related processes only. Pipes can’t be used for unrelated processes communication, say, if we want to execute one process from one terminal and another process from another terminal; it is not possible with pipes. Do we have any simple way of communicating between two processes, say unrelated processes in a simple way? The answer is YES. Named pipe is meant for communication between two or more unrelated processes and can also have bi-directional communication.

**Sample Code:**

// C program to implement one side of FIFO

// This side reads first, then reads

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int fd1;

// FIFO file path

char \* myfifo = "/tmp/myfifo";

// Creating the named file(FIFO)

// mkfifo(<pathname>,<permission>)

mkfifo(myfifo, 0666);

char str1[80], str2[80];

while (1)

{

// First open in read only and read

fd1 = open(myfifo,O\_RDONLY);

read(fd1, str1, 80);

// Print the read string and close

printf("User1: %s\n", str1);

close(fd1);

// Now open in write mode and write

// string taken from user.

fd1 = open(myfifo,O\_WRONLY);

fgets(str2, 80, stdin);

write(fd1, str2, strlen(str2)+1);

close(fd1);

}

return 0;

}

Read

// This side writes first, then reads #include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{ int fd;

// FIFO file path

char \* myfifo = "/tmp/myfifo"; // Creating the named file(FIFO) //

//mkfifo(<pathname>, <permission>) mkfifo(myfifo, 0666);

char arr1[80], arr2[80]; while (1)

{

// Open FIFO for write only

fd = open(myfifo, O\_WRONLY);

// Take an input arr2ing from user. // //80 is maximum length

fgets(arr2, 80,stdin);

// Write the input arr2ing on FIFO // and close it

write(fd, arr2,strlen(arr2)+1); close(fd);

// Open FIFO for Read only fd = open(myfifo, O\_RDONLY);

// Read from FIFO read(fd,

arr1, sizeof(arr1);

// Print the read message

printf("User2: %s\n", arr1); close(fd);

} return

0;

}

**Conclusion/Result:**

We have implemented two processes communicating with each other using FIFO and counting no of words, letters and lines.

**Assignment No. 7B:**

**Inter process communication in Linux using following.**

**Aim:** Interprocess Communication using Shared Memory using System V.

Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

**Theory:**

Shared memory is one of the three interprocess communication (IPC) mechanisms available under Linux and other Unix-like systems. The other two IPC mechanisms are the message queues and semaphores. In case of shared memory, a shared memory segment is created by the kernel and mapped to the data segment of the address space of a requesting process. A process can use the shared memory just like any other global variable in its address space.

The shared memory mechanism avoids copying overhead. The first process simply writes data into the shared memory segment. As soon as it is written, the data becomes available to the second process. Shared memory is the fastest mechanism for interprocess communication.

System V Shared Memory Calls

4.1 shmget

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget (key\_t key, size\_t size, int shmflg);

As the name suggests, shmget gets you a shared memory segment associated with the given key.

The key is obtained earlier using the ftok function. If there is no existing shared memory segment corresponding to the given key and IPC\_CREAT flag is specified in shmflg, a new shared memory segment is created. Also, the key value could be IPC\_PRIVATE, in which case a new shared memory segment is created. size specifies the size of the shared memory segment to be created; it is rounded up to a multiple of PAGE\_SIZE. If shmflg has IPC\_CREAT | IPC\_EXCL specified and a shared memory segmentfor the given key exists, shmget fails and returns -1, with errno set to EEXIST. The last nine bits of shmflg specify the permissions granted to owner, group and others. The execute permissions are not used. If shmget succeeds, a shared memory identifier is returned. On error, -1 is returned and errno isset to the relevant error. It means the shared memory segment should be attached with read only access.

4.2 shmat

#include <sys/types.h>

#include <sys/shm.h>

void \*shmat (int shmid, const void \*shmaddr, int shmflg);

With shmat, the calling process can attach the shared memory segment identified by *shmid*. The process can specify the address at which the shared memory segment should be attached with *shmaddr*. However, in most cases, we do not care at what address system attaches the shared memory segment and *shmaddr* can conveniently be specified as NULL. *shmflg* specifies the flags for attaching the shared memory segment. If *shmaddr* is not null and SHM\_RND is specified in *shmflg*, the shared memory segment is attached at address rounded down to the nearest multiple of SHMLBA, where SHMLBA stands for Segment low boundary address. The idea is to attach at an address which is a multiple of SHMLBA. On most Linux systems, SHMLBA is the same as PAGE\_SIZE. Another flag is SHM\_RDONLY, On success, shmat returns pointer to the attached shared memory segment. On error, (void \*) -1 is returned, with errno set to the cause of the error.

4.3 shmdt

#include <sys/types.h>

#include <sys/shm.h>

int shmdt (const void \*shmaddr);

shmdt detaches a shared memory segment from the address space of the calling process. *shmaddr* is the address at which the shared memory segment was attached, being the value returned by anearlier shmat call. On success, shmdt returns 0. On error, shmdt returns -1 and errno is set to indicate the reason of error.

4.4 shmctl

#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid\_ds \*buf);

The shmctl call is for control operations of a System V shared memory segment identified by the *shmid* identifier, returned by an earlier shmget call. The data structure for shared memory segments in the kernel. The *cmd* parameter in shmctl specifies the command. The important commands are IPC\_STAT, IPC\_SET and IPC\_RMID. The IPC\_STAT command copiesthe data in the kernel data structure shmid\_ds for the shared memory into the location pointed by the parameter *buf*. With the IPC\_SET command, we can set some of the fields in the shmid\_ds structure in the kernel for the shared memory segment. The fields that can be modified are shm\_perm.uid, shm\_perm.gid and the least significant 9 bits of shm\_perm.mode. The command, IPC\_RMID, marks a shared memory segment for removal from the system. The shared memory segment is actually removed after the last process detaches it from its

address space.

CODE:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666|IPC\_CREAT); /\*creates shared memory segment with key 2345, having size 1024 bytes. IPC\_CREAT is used to create the shared segment if it does not exist. 0666 are the permisions on the shared segment\*/

printf("Key of shared memory is %d\n",shmid);

shared\_memory=shmat(shmid,NULL,0); //process attached to shared memory segment

printf("Process attached at %p\n",shared\_memory); //this prints the address where the Segment is attached with this process

printf("Enter some data to write to shared memory\n");

read(0,buff,100); //get some input from user

strcpy(shared\_memory,buff); //data written to shared memory

printf("You wrote : %s\n",(char \*)shared\_memory);

}

Client

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666);

printf("Key of shared memory is %d\n",shmid);

shared\_memory=shmat(shmid,NULL,0); //process attached to shared memory segment printf("Process attached at %p\n",shared\_memory);

printf("Data read from shared memory is : %s\n",(char \*)shared\_memory); }

**Conclusion/Result:**

We have implemented two processes communicating with each other using IPC

**Assignment No-8**

**Aim:** Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

**Theory:**

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling. served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled. Ore request may be far from each other so can result in greater disk arm

movement. Accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let’s have a quick look at some of the important terms:

data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

speed of the disk and number of bytes to be transferred.

Disk Access Time is: Disk Access Time = Seek Time + Rotational Latency + Transfer Time

**Shortest Seek Time First (SSTF) :**

Basic idea is the tracks which are closer to current disk head position should be serviced first in order to minimize the seek operations.

Algorithm –

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘head’ is the position of disk head.

2. Find the positive distance of all tracks in the request array from head.

3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.

4. Increment the total seek count with this distance.

5. Currently serviced track position now becomes the new head position.

6. Go to step 2 until all tracks in request array have not been serviced.

**SCAN (Elevator) algorithm:**

In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Algorithm

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.

2. Let direction represents whether the head is moving towards left or right. 3. In the direction in which head is moving service all tracks one by one.

4. Calculate the absolute distance of the track from the head.

5. Increment the total seek count with this distance.

6. Currently serviced track position now becomes the new head position.

7. Go to step 3 until we reach at one of the ends of the disk.

8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

**C-LOOK (Circular LOOK) Disk Scheduling Algorithm:**

C-LOOK is an enhanced version of both SCAN as well as LOOK disk scheduling algorithms. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm. We know that C-SCAN is used to avoid starvation and services all the requests more uniformly, the same goes for C-LOOK.

In this algorithm, the head services requests only in one direction(either left or right) until all the requests in this direction are not serviced and then jumps back to the farthest request on the other direction and service the remaining requests which gives a better uniform servicing as well as avoids wasting seek time for going till the end of the disk.

Algorithm

1. Let Request array represents an array storing indexes of the tracks that have been requested in ascending order of their time of arrival and head is the position of the disk head.
2. The initial direction in which the head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction it is moving.
4. The head continues to move in the same direction until all the requests in this direction have been serviced.
5. While moving in this direction, calculate the absolute distance of the tracks from the head.
6. Increment the total seeks count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach the last request in this direction.
9. If we reach the last request in the current direction then reverse the direction and move the head in this direction until we reach the last request that is needed to be serviced in this direction without servicing the intermediate requests.
10. Reverse the direction and go to step 3 until all the requests have not been serviced.

**Code:**

SSTF:

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

// logic for sstf disk scheduling

/\* loop will execute until all process is completed\*/

while(count!=n)

{

int min=1000,d,index;

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

{

d=abs(RQ[i]-initial);

if(min>d)

{

min=d;

index=i;

}

}

TotalHeadMoment=TotalHeadMoment+min;

initial=RQ[index];

// 1000 is for max

// you can use any number

RQ[index]=1000;

count++;

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

SCAN Algorithm:

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for Scan disk scheduling

/\*logic for sort the request array \*/

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

{

for(j=0;j<n-i-1;j++)

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

for(i=index;i<n;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

initial = size-1;

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

initial =0;

for(i=index;i<n;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

C-Look Algorithm:

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for C-look disk scheduling

/\*logic for sort the request array \*/

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

{

for( j=0;j<n-i-1;j++)

{

if(RQ[j]>RQ[j+1])

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

for(i=index;i<n;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

for( i=0;i<index;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Conclusion/Result:**

Thus, disk scheduling algorithms play important role in serving different scenario efficiently. We have implemented the SSTF, SCAN and C-LOOK algorithms to check the difference for same sequence of requests.

**Extra** **Assignment**

**Assignment No 9:**

Memory Management A). Memory Management With Fixed Partitioning Technique (Mft)

**Aim:** To implement and simulate the MFT algorithm.

**Theory:** In this the memory is divided in two parts and process is fit into it. The process which is best suited will be placed in the particular memory where it suits. In MFT, the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. In MVT, each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

**Algorithm**:

Step1: Start the process.

Step2: Declare variables.

Step3: Enter total memory size ms.

Step4: Allocate memory for os. Ms=ms-os

Step5: Read the no partition to be divided n Partition size=ms/n.

Step6: Read the process no and process size.

Step 7: If process size is less than partition size allot alse blocke the process.

While allocating update memory wastage-external fragmentation.

If (pn[i]==pn[j])

f=1;

if(f==0)

{

if(ps[i]<=siz)

{

extft=extft+size

ps[i];avail[i]=1;

count++;

}

}

Step 8: Print the results.

**Sample Code :**

#include<stdio.h> #include<conio.h> main()

{

int ms, bs, nob, ef,n, mp[10],tif=0; int i,p=0;

clrscr();

printf("Enter the total memory available (in Bytes) -- "); scanf("%d",&ms);

printf("Enter the block size (in Bytes) -- "); scanf("%d", &bs);

nob=ms/bs; ef=ms - nob\*bs;

printf("\nEnter the number of processes -- "); scanf("%d",&n);

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

{

printf("Enter memory required for process %d (in Bytes)-- ",i+1); scanf("%d",&mp[i]);

}

printf("\nNo. of Blocks available in memory--%d",nob); printf("\n\nPROCESS\tMEMORYREQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION");

for(i=0;i<n && p<nob;i++)

{

printf("\n %d\t\t%d",i+1,mp[i]); if(mp[i] > bs)

printf("\t\tNO\t\t---"); else

{

printf("\t\tYES\t%d",bs-mp[i]); tif = tif + bs-mp[i];

p++;

}

}

if(i<n)

printf("\nMemory is Full, Remaining Processes cannot be accomodated"); printf("\n\nTotal Internal Fragmentation is %d",tif);

printf("\nTotal External Fragmentation is %d",ef); getch();

}

|  |  |  |  |
| --- | --- | --- | --- |
| ***INPUT***  Enter the total memory available (in Bytes) -- |  | 1000 |  |
| Enter the block size (in Bytes)-- 300  Enter the number of processes – 5 |  |  |
| Enter memory required for process 1 (in Bytes) -- | 275 |  |
| Enter memory required for process 2 (in Bytes) -- | 400 |  |
| Enter memory required for process 3 (in Bytes) -- | 290 |  |
| Enter memory required for process 4 (in Bytes) -- | 293 |  |
| Enter memory required for process 5 (in Bytes) -- | 100 |  |
| No. of Blocks available in memory -- 3 |  |  |
| ***OUTPUT***  PROCESS | ALLOCAT |  | INTERNAL |
| MEMORY REQUIRED | ED |  | FRAGMENTATION |
| 1 275 | YES |  | 25 |
| 2 400 | NO |  | ----- |
| 3 290 | YES |  | 10 |
| 4 293 | YES |  | 7 |

Memory is Full, Remaining Processes cannot be accommodated Total Internal Fragmentation is 42

Total External Fragmentation is 100

**Assignment No 10:**

**AIM:** To write a program to simulate the MVT algorithm

# ALGORITHM:

Step1: start the process. Step2: Declare variables.

Step3: Enter total memory size ms. Step4: Allocate memory for os.

Ms=ms-os

Step5: Read the no partition to be divided n Partition size=ms/n. Step6: Read the process no and process size.

Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation.

if(pn[i]==pn[j]) f=1; if(f==0){ if(ps[i]<=size)

{

extft=extft+size- ps[i];avail[i]=1; count++;

}

}

Step 8: Print the results Step 9: Stop the process.

**SOURCECODE:**

:#include<stdio.h> #include<conio.h>

main()

{

int ms,mp[10],i, temp,n=0; char ch = 'y'; clrscr();

printf("\nEnter the total memory available (in Bytes)-- "); scanf("%d",&ms);

temp=ms; for(i=0;ch=='y';i++,n++)

{

printf("\nEnter memory required for process %d (in Bytes) -- ",i+1); scanf("%d",&mp[i]);

if(mp[i]<=temp)

{

printf("\nMemory is allocated for Process %d ",i+1); temp = temp - mp[i];

}

else

{

printf("\nMemory is Full"); break;

}

printf("\nDo you want to continue(y/n) -- "); scanf(" %c", &ch);

}

printf("\n\nTotal Memory Available -- %d", ms); printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED "); for(i=0;i<n;i++)

printf("\n \t%d\t\t%d",i+1,mp[i]);

printf("\n\nTotal Memory Allocated is %d",ms-temp); printf("\nTotal External Fragmentation is %d",temp);

getch();

}

Output:

Enter the total memory available (in Bytes) – 1000

Enter memory required for process 1 (in Bytes) – 400 Memory is allocated for Process 1

Do you want to continue(y/n) -- y

Enter memory required for process 2 (in Bytes) -- 275 Memory is allocated for Process 2

Do you want to continue(y/n) – y

Enter memory required for process 3 (in Bytes) – 550

Memory is Full

Total Memory Available – 1000

PROCESS MEMORY ALLOCATED

1 400

2 275

Total Memory Allocated is 675 Total External Fragmentation is 325

VIVA QUESTIONS

1. What is MFT?
2. What is MVT?
3. What is the difference between MVT and MFT?
4. What is meant by fragmentation?
5. Give the difference between internal and external fragmentation