ARUNAI ENGINEERING COLLEGE

(Affiliated to Anna University)

Velu Nagar, Thiruvannamalai-606 603
www.arunai.org

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

BACHELOR OF ENGINEERING

2021 - 2022

FOURTH SEMESTER

CS8461 – OPERATING SYSTEMS LAB

CS8461 OPERATING SYSTEMS LABORATORY L T P C 0 0 4 2

OBJECTIVES

- To learn Unix commands and shell programming
- To implement various CPU Scheduling Algorithms
- To implement Process Creation and Inter Process Communication.
- To implement Deadlock Avoidance and Deadlock Detection Algorithms
- To implement Page Replacement Algorithms
- To implement File Organization and File Allocation Strategies

LIST OF EXPERIMENTS

1. Basics of UNIX commands
2. Write programs using the following system calls of UNIX operating system
3. fork, exec, getpid, exit, wait, close, stat, opendir, readdir
4. Write C programs to simulate UNIX commands like cp, ls, grep, etc.
5. Shell Programming
6. Write C programs to implement the various CPU Scheduling Algorithms
7. Implementation of Semaphores
8. Implementation of Shared memory and IPC
9. Bankers Algorithm for Deadlock Avoidance
10. Implementation of Deadlock Detection Algorithm
11. Write C program to implement Threading & Synchronization Applications
12. Implementation of the following Memory Allocation Methods for fixed partition
    a) First Fit b) Worst Fit c) Best Fit
13. Implementation of Paging Technique of Memory Management
14. Implementation of the following Page Replacement Algorithms
    a) FIFO b) LRU c) LFU
15. Implementation of the various File Organization Techniques
16. Implementation of the following File Allocation Strategies
    a) Sequential b) Indexed c) Linked

TOTAL: 60 PERIODS

PROGRAMME OUTCOMES (POs)

After going through the four years of study, computer science & engineering graduates will exhibit:

	Graduate Attribute	Programme Outcome
1	Engineering knowledge	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.
2	Problem analysis	Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3	Design/development of solutions	Design solutions for complex engineering problems and designsystem components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4	Conduct investigations of complex problems	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, andsynthesis of the information to provide valid conclusions
5	Modern tool usage	Create, select, and apply appropriate techniques, resources, andmodern engineering and IT tools, including prediction and modeling to complex engineering activities, with an understanding of the limitations.
6	The engineer and society	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice
7	Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8	Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
9	Individual and team work	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10	Communication	Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions
11	Project management and finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments
12	Life-long learning	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAM SPECIFIC OUTCOMES (PSOs)

By the completion of Information Technology program the student will have following Programspecific outcomes

1. Design secured database applications involving planning, development and maintenance usingstate of the art methodologies based on ethical values.
2. Design and develop solutions for modern business environments coherent with the advancedtechnologies and tools.
3. Design, plan and setting up the network that is helpful for contemporary business environmentsusing latest hardware components.
4. Planning and defining test activities by preparing test cases that can predict and correct errorsensuring a socially transformed product catering all technological needs

LIST OF EQUIPMENT FOR A BATCH OF 30 STUDENTS:

SOFTWARE:

- C / C++ / Unix OS

HARDWARE:

Standalone desktops - 30 Nos. (or) Server supporting 30 terminals or more.

OUTCOMES:

At the end of the course, the students will be able to:

Course Outcomes	Description	Level in Bloom's Taxonomy
C217.1	Illustrate the various CPU scheduling algorithms.	K3
C217.2	Implement deadlock avoidance and detection algorithms.	K3
C217.3	Implement semaphore concepts.	K3
C217.4	Create processes and implement IPC.	K3
C217.5	Analyze the performance of the various page replacement algorithms.	K3
C217.6	Implement file organization and file allocation strategies.	K3
C217.7	Exhibit ethical principles in engineering practices	A3
C217.8	Perform task as an individual and / or team member to manage the task in time	A3
C217.9	Express the Engineering activities with effective presentation and report.	A3
C217.10	Interpret the findings with appropriate technological / research citation.	A2

CO - PO MATRIX

Course Outcomes	Programme Outcome (POs)											
	K3	K4	K4	K5	K3,K4,K5	A3	A2	A3	A3	A3	A3	A2
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	2	2	-	-	-	-	-	-	-	-	-
CO2	3	2	2	-	-	-	-	-	-	-	-	-
CO3	3	2	2	-	-	-	-	-	-	-	-	-
CO4	3	2	2	-	-	-	-	-	-	-	-	-
CO5	3	2	2	-	-	-	-	-	-	-	-	-
CO6	3	2	2	-	-	-	-	-	-	-	-	-
CO7	-	-	-	-	-	-	-	3	-	-	-	-
CO8	-	-	-	-	-	-	-	-	3	-	3	-
CO9	-	-	-	-	-	-	-	-	-	3	-	-
CO10	-	-	-	-	-	-	-	-	-	-	-	3
	3	2	2	-	-	-	-	3	3	3	3	3

CO - PSO MATRIX

	PSO1	PSO2	PSO3
CO1	3	2	1
CO2	3	2	1
CO3	3	2	1
CO4	3	2	1
CO5	3	2	1
CO6	3	2	1
CO7	-	-	-
CO8	-	-	-
CO9	-	-	-
CO10	-	-	
	3	2	1

MODE OF ASSESSMENT

EVALUATION PROCEDURE FOR EACH EXPERIMENT

S.No	Description	Mark
1.	Aim & Pre-Lab discussion	20
2.	Observation	20
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	20
	Total	**100**

INTERNAL ASSESSMENT FOR LABORATORY

S.No	Description	Mark
1.	Observation	05
2.	Performance	05
3.	Viva voce	05
4.	Record	05
	Total	**20**

ABOUT THE SOFTWARE

UNIX OS

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X. Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.
As an illustration of the way that the shell and the kernel work together, suppose a user types **rm myfile** (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm** on **myfile**. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. 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 (% on our systems).

The adept user can customise his/her own shell, and users can use different shells on the same machine. Staff and students in the school have the **tcsh shell** by default. The tcsh shell has certain features to help the user inputting commands.
Filename Completion - By typing part of the name of a command, filename or directory and pressing the [**Tab**] key, the tcsh shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

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.

Everything in UNIX is either a file or a process.
A process is an executing program identified by a unique PID (process identifier).
A file is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language
- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

C / C++

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications. C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

C++ is a special-purpose programming language developed by **Bjarne Stroustrup** at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

Differences between C and C++:

- **Definition**

  C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Subset**

  C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.
- **Type of approach**

  C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.

- **Security**

  In C, the data can be easily manipulated by the outsiders as it does not support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.
- **Function Overloading**

  Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.
- **Function Overriding**

  Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.
- **Keywords**

  C contains 32 keywords, and C++ supports 52 keywords.
- **Namespace feature**

  A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.
- **Exception handling**

  C does not provide direct support to the exception handling; it needs to use functions that support exception handling. C++ provides direct support to exception handling by using a try-catch block.
- **Input/Output functions**

  In C, scanf and printf functions are used for input and output operations, respectively, while in C++, cin and cout are used for input and output operations, respectively.
- **Memory allocation and de-allocation**

  C supports calloc() and malloc() functions for the memory allocation, and free() function for the memory de-allocation. C++ supports a new operator for the memory allocation and delete operator for the memory de-allocation.

LIST OF EXPERIMENTS

Exp. No. 1a Study of UNIX OS

Date:

Aim

To introduce the concepts of UNIX Operating System

Operating System

- An Operating System is a set of programs that:
  - Functions as an virtual machine by presenting an interface that is easier toprogram than the underlying hardware
  - Acts as resource management through orderly and controlled allocation of theprocessors, memories, and I/O devices among the programs competing for it.

UNIX Features

1. *Multi-user system*—Multi-user capability of UNIX allows several users to use the same computer to perform their tasks. Several terminals [Keyboards and Monitors] are connected to a single powerful server.
2. *Multi-tasking system*—Multitasking is the capability of the operating system to perform various task simultaneously, i.e. a user can run multiple tasks concurrently.
3. *Programming Facility*—the UNIX shell has all the necessary ingredients like conditional and control structures, etc.
4. *Security*—Every user must have a single login name and password. So, accessing another user's data is impossible without his permission.

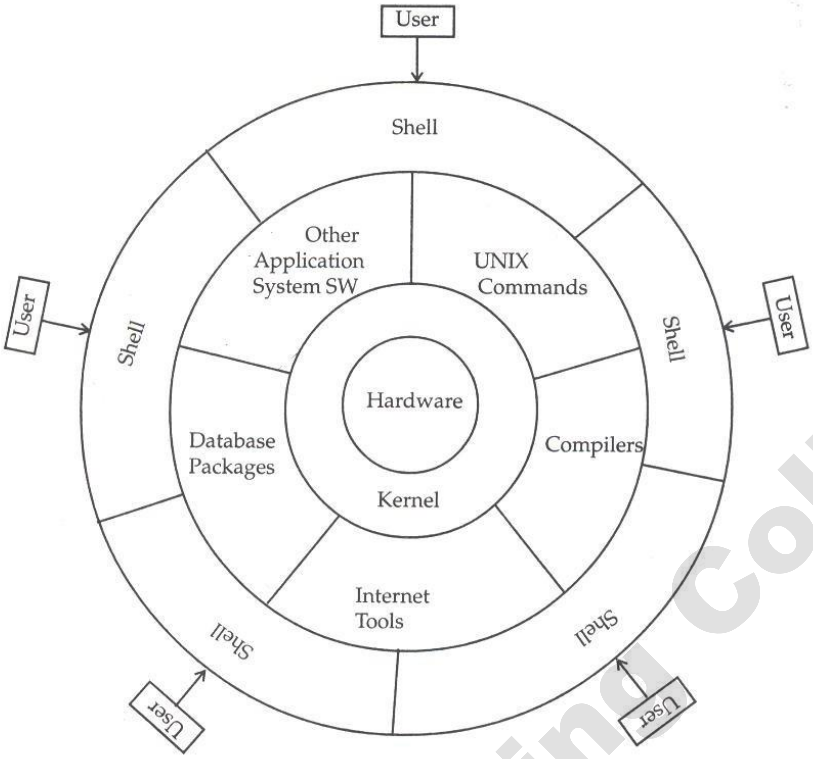
Apart from these features, UNIX has an extensive Tool kit, exhaustive system calls and Libraries and enhanced GUI (X Window).

Organization of UNIX

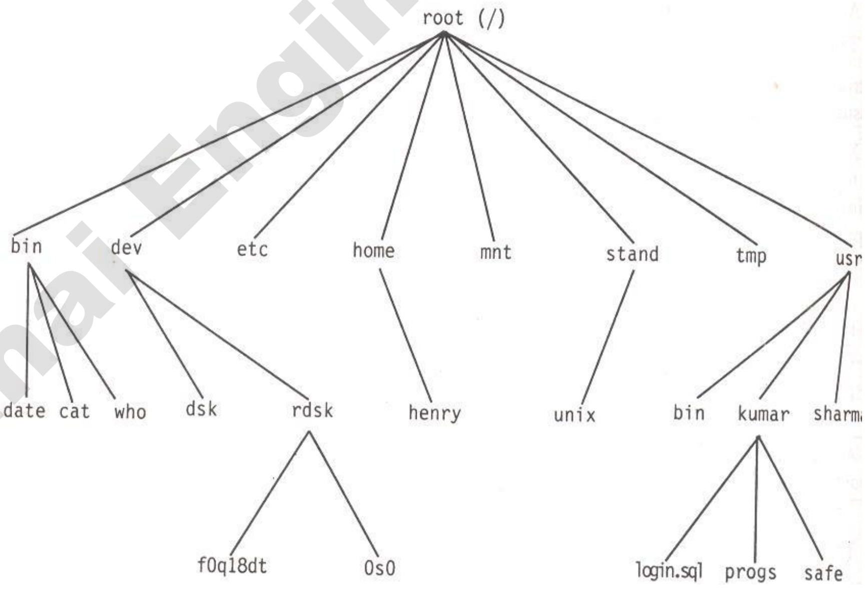
1. The kernel is the heart of the system, a collection of programs written in C that directly communicate with the hardware. It manages the system resources, allocates time between user and processes, decides process priorities, and performs all other tasks. The kernel, in traditional parlance, is often called the Operating system.
2. The shell, on the other hand, is the "sleeping beauty" of UNIX. It is actually the interface between the user and the kernel. The shell is the agency which takes care ofthe features of redirection and has a programming capability of its own.
3. The Tools and Applications consist of Application Software, Compilers, Database Package, Internet tools, UNIX commands, etc.

File System

All files in UNIX are related to one another. The file system of UNIX resembles a tree that grows from top to bottom as shown in the figure. The file system begins with a directory called root (at the top). The root directory is denoted by a slash (\). Branching from root there are several directories such as bin, lib, etc, tmp, dev. Each of these directories contains several sub-directories and files.



Organization



File System

Result

Thus the study of UNIX Operating System has been completed successfully.

Exp. No. 1b **Unix Commands**

Date :

Aim

To study and execute Unix commands.

Login

Type **telnet***server_ipaddress* in **run** window.

User has to authenticate himself by providing *username* and *password*. Once verified, a greeting and **$** prompt appears. The shell is now ready to receive commands from the user. Options suffixed with a hyphen (–) and arguments are separated by space.

General commands

Command	**Function**
Date	Used to display the current system date and time.
date +%D	Displays date only
date +%T	Displays time only
date +%Y	Displays the year part of date
date +%H	Displays the hour part of time
Cal	Calendar of the current month
cal*year*	Displays calendar for all months of the specified year
cal*month year*	Displays calendar for the specified month of the year
Who	Login details of all users such as their IP, Terminal No, User name,
who am i	Used to display the login details of the user
Uname	Displays the Operating System
uname –r	Shows version number of the OS (kernel).
uname –n	Displays domain name of the server
echo$HOME	Displays the user's home directory
Bc	Basic calculator. Press Ctrl+dto quit
lp *file*	Allows the user to spool a job along with others in a print queue.
man*cmdname*	Manual for the given command. Press qto exit
history	To display the commands used by the user since log on.
exit	Exit from a process. If shell is the only process then logs out

Directory commands

Command	**Function**
Pwd	Path of the present working directory
mkdir*dir*	A directory is created in the given name under the current directory
mkdir*dir1 dir2*	A number of sub-directories can be created under one stroke
cd*subdir*	Change Directory. If the *subdir* starts with **/** then path starts from **root** (absolute) otherwise from current working directory.
cd	To switch to the home directory.
cd /	To switch to the root directory.
cd ..	To move back to the parent directory
rmdir*subdir*	Removes an empty sub-directory.

File commands

Command	**Function**
cat >*filename*	To create a file with some contents. To end typing press Ctrl+d.The > symbol means redirecting output to a file.
cat*filename*	Displays the file contents.
cat>>*filename*	Used to append contents to a file
cp*src des*	Copy files to given location. If already exists, it will be overwritten
cp–i *src des*	Warns the user prior to overwriting the destination file
cp –r *src des*	Copies the entire directory, all its sub-directories and files.
mv *old new*	To rename an existing file or directory. –ioption can also be used
mv *f1 f2 f3 dir*	To move a group of files to a directory.
mv –v *old new*	Display name of each file as it is moved.
rm *file*	Used to delete a file or group of files. –ioption can also be used
rm *	To delete all the files in the directory.
rm –r *	Deletes all files and sub-directories
rm –f *	To forcibly remove even write-protected files
Ls	Lists all files and subdirectories (blue colored) in sorted manner.
ls*name*	To check whether a file or directory exists.
ls*name**	Short-hand notation to list out filenames of a specific pattern.
ls –a	Lists all files including hidden files (files beginning with **.**)
ls –x*dirname*	To have specific listing of a directory.
ls –R	Recursive listing of all files in the subdirectories
ls –l	Long listing showing file access rights (read/write/execute-**rwx** foruser/group/others-**ugo**).
cmp *file1 file2*	Used to compare two files. Displays nothing if files are identical.
wc *file*	It produces a statistics of lines (**l**), words(**w**), and characters(**c**).
chmod *perm file*	Changes permission for the specified file. (r=4, w=2, x=1) chmod 740 *file*sets all rights for user, read only for groups and no rights for others

The commands can be combined using the pipeline (|) operator. For example, number ofusers logged in can be obtained as.

who | wc -l

Finally to terminate the unix session execute the command **exit** or **logout**.

Output

```
$ date
Sat Apr   9 13:03:47 IST 2011

$ date +%D
04/09/11

$ date +%T
13:05:33

$ date +%Y
2011

$ date +%H
13


$ cal 08 1998
       August 1998
Su  M  Tu  W  Th  Fr  Sa
    o       e
                       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

$ who
root         :0            Apr   9 08:41
vijai        pts/0         Apr   9 13:00 (scl-64)
cse4001   pts/3            Apr   9 13:18 (scl-41.smkfomra.com)

$ uname
Linux

$ uname -r
```

```
2.4.20-8smp

$ uname -n
localhost.localdomain


$ echo $HOME
/home/vijai

$ echo $USER
vijai

$ bc
3+5
8
pwd
/home/vijai/shellscripts/loops

$ mkdir filter
$ ls
filter   list.sh   regexpr   shellscripts

$ cd shellscripts/loops/
$

$ cd
$

$ cd / [vijai@localhost  /]$

[vijai@localhost  /]$  cd  /home/vijai/shellscripts/loops/
$ cd ..
[vijai@localhost  shellscripts]$

$ rmdir filter
$ ls
list.sh   regexpr   shellscripts

$ cat > greet
hi cse
wishing u the best

$ cat greet
hi ece-a
wishing u the best
```

```
$ cat >> greet
bye
$ cat greet
hi cse
wishing u the bestbye

$ ls
greet   list.sh   regexpr   shellscripts

$ ls -a
.                 .bash_logout     .canna   .gtkrc       regexpr          .viminfo.tmp
..                .bash_profile    .emacs    .kde        shellscripts   .xemacs
.bash_history     .bashrc           greet     list.sh     .viminfo
ls -l
-rw-rw-r--         1  vijai        vijai                32  Apr  11   14:52  greet
-rw-rw-r--         1  vijai        vijai                30  Apr   4   13:58  list.sh
drwxrwxr-x         2  vijai        vijai              4096  Apr   9   14:30  regexpr


$ cp greet ./regexpr/
$ ls
greet   list.sh   regexpr   shellscripts
$ ls ./regexpr
demo greet

$ cp -i greet ./regexpr/
cp: overwrite 'greet'? n

$ mv greet greet.txt
$ ls
greet.txt   list.sh   regexpr   shellscripts

$ mv greet.txt ./regexpr/
$ ls
list.sh   regexpr   shellscripts

$ rm -i *.sh
rm: remove regular file 'fact.sh'? yrm:
remove regular file 'prime.sh'? y
$ ls
list.sh   regexpr   shellscripts

$ wc list.sh
         4          9         30 list.sh
```

```
$ wc -l list.sh
        4 list.sh

$ cmp list.sh fact.sh
list.sh fact.sh differ: byte 1, line 1

$ ls -l list.sh
-rw-rw-r--      1 vijai      vijai               30 Apr   4 13:58 list.sh

$ chmod ug+x list.sh

$ ls -l list.sh
-rwxrwxr--      1 vijai      vijai               30 Apr   4 13:58 list.sh
$ chmod 740 list.sh
$ ls -l list.sh
-rwxr-----      1 vijai      vijai               30 Apr   4 13:58 list.sh
```

Result

Thus the study and execution of Unix commands has been completed successfully.

Exp. No. 2a Study of vi Editor

Date :

Aim

To introduce the concept of text editing using **vi** editor.

vi Editor

Unix provides a versatile editor **vi,** a full-screen editor. "vi" stands for *visual* editor. A vi session begins by invoking vi with or without a filename

```
$vi filename
```

An empty screen, each line beginning with a ~ is displayed. **vi**functions in three modes.

Input Mode

vi starts with command mode. To insert text press I or i. In *Input* mode the editor displays INSERTin the last line. To quit *input* mode press *Esc* key.

Edit Commands

Command	**Function**
x	Deletes the character in the current cursor position
?*text*	Locates the *text* in the file. Use **n** to repeat the search.
u	Reverses the last change made to the buffer.
dd(or) dw	Cuts the entire line / word
yy(or) yw	Copies the entire line / word
p	Pastes the text

Navigation commands

Command	**Function**
b(or) w	Moves back to beginning / end of a word
|(or) $	Moves to start of the line
***l*G**	To move to the specific line

ex Mode

Press **:** (colon) in command mode to switch to **ex** mode. The **:** is displayed in the lastline. Type the command and press *Enter* key to execute the same.

Command	**Function**
w	Saves file,
q!	Quits vi session without saving any changes made since the last save
wq	Save and exit
%s/*Sstr*/*Rstr*/g	It is Find and Replace. **%** represents all lines, **g** makes it global.

Result

Thus the study of text manipulation using vi editor has been completed successfully.

Exp. No. 2b **Shell Programming**

Date:

Aim

To create shell scripts using shell programming constructs.

The activities of a shell are not restricted to command interpretation alone. The shellalso has rudimentary programming features. Shell programs are stored in a file (with extension .**sh**). Shell programs run in interpretive mode. Bourne shell (**sh**), C shell (**csh**) and Korn shell (**ksh**) are also widely used. Linux offers Bash shell (**bash**) .

Preliminaries

1. Comments in shell script start with **#**.
2. Shell variables are loosely typed i.e. not declared. Variables in an expression or output must be prefixed by **$**.
3. The **read**statement is shell's internal tool for making scripts interactive.
4. Output is displayed using **echo**statement.
5. Expressions are computed using the **expr**command. Arithmetic operators are + -
   * / %. Meta characters *** ( )** should be escaped with a \.
6. The shell scripts are executed

   $ sh*filename*

Decision-making

Shell supports decision-making using **if**statement. The **else**statement is optional.

```
if [ condition ]
then
    statements
else
    statements
fi
```

The else-if ladder has the following syntax.

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

The set of relational operators are –eq –ne –gt –ge –lt –le and logical operators used in conditional expression are –a –o !

Multi-way branching

The casestatement is used to compare a variables value against a set of constants. If it matches a constant, then the set of statements followed after) is executed till a ;; is encountered. The optional *default* block is indicated by *****. Multiple constants can be specified in a single pattern separated by **|**.

```
casevariable in
   constant1)
       statements ;;
   constant2)
       statements ;;
    . . .
       *)
       statements
esac
```

Loops

Shell supports a set of loops such as **for**, **while** and **until** to execute a set of statements
repeatedly. The body of the loop is contained between **do** and **done**

statement.The **for** loop doesn't test a condition, but uses a list instead.

```
forvariable inlist
do
        statements
done
```

The **while** loop executes the *statements* as long as the condition remains true.

```
while [ condition ]
do
        statements
done
```

The **until** loop complements the while construct in the sense that the *statements* are executedas long as the condition remains false.

```
until [ condition ]do
        statements
done
```

A) Swapping values of two variables

```
# Swapping values – swap.sh echo -n
"Enter value for A : "read a
echo -n   "Enter value for B : "read b
t=$a
a=$b
b=$t
echo "Values after Swapping"
echo "A Value is $a and B Value is $b"
```

Output

```
$ sh swap.sh
Enter value for A : 12
Enter value for B : 23Values
after Swapping
A Value is 23 and B Value is 12
```

B) Farenheit to Centigrade Conversion

```
# Degree conversion – degconv.shecho -n
"Enter Fahrenheit : " read f
c=`expr \( $f - 32 \) \* 5 / 9`echo
"Centigrade is : $c"
```

Output

```
$ sh degconv.sh
Enter     Fahrenheit     :     213
Centigrade is : 100
```

C) Biggest of 3 numbers

```
# Biggest – big3.sh
echo -n "Give value for A B and C: "read a b c
if [ $a -gt $b -a $a -gt $c ]then
      echo "A is the Biggest number"elif [ $b -
gt $c ]
then
      echo "B is the Biggest number"else
      echo "C is the Biggest number"
fi
```

Output

```
$ sh big3.sh
Give value for A B and C: 4 3 4
C is the Biggest number
```

D) Grade Determination

```
# Grade – grade.sh
echo -n "Enter the mark : "read mark
if [ $mark -gt 90 ]then
    echo "S Grade" elif [
$mark -gt 80 ]then
    echo "A Grade" elif [
$mark -gt 70 ]then
    echo "B Grade" elif [
$mark -gt 60 ]then
    echo "C Grade" elif [
$mark -gt 55 ]then
    echo "D Grade" elif [
$mark -ge 50 ]then
    echo "E Grade"
else
    echo "U Grade"
fi
```

Output

```
$ sh grade.sh
Enter the mark : 65C
Grade
```

E) Vowel or Consonant

```
# Vowel   - vowel.sh
echo -n "Key in a lower case character : "read choice
case $choice in
        a|e|i|o|u) echo "It's a Vowel";;
                    *) echo "It's a Consonant"
esac
```

Output

```
$ sh vowel.sh
Key in a lower case character : eIt's a Vowel
```

F) Simple Calculator

```
# Arithmetic operations — calc.shecho -n
"Enter the two numbers : "read a b
echo " 1. Addition" echo " 2.
Subtraction"
echo " 3. Multiplication" echo " 4.
Division"
echo -n "Enter the option : "read option
case $option in
     1) c=`expr $a + $b` echo "$a
        + $b = $c";;
     2) c=`expr $a - $b` echo "$a -
        $b = $c";;
     3) c=`expr $a \* $b` echo "$a *
        $b = $c";;
     4) c=`expr $a / $b` echo "$a /
        $b = $c";;
     *) echo "Invalid Option"esac
```

Output

```
$ sh calc.sh
Enter the two numbers : 2 4
  1. Addition
  2. Subtraction
  3. Multiplication
  4. Division
Enter the option : 12 + 4 = 6
```

G) Multiplication Table

```
#Multiplication table – multable.shclear
echo -n "Which multiplication table? : "read n
for x in 1 2 3 4 5 6 7 8 9 10do
     p=`expr $x \* $n`
     echo -n "$n X $x = $p"sleep 1
done
```

Output

```
$ sh multable.sh
Which multiplication table? : 66 X 1 = 6
6 X 2 = 12
.....
```

H) Number Reverse

```
# To reverse a number – reverse.shecho -n
"Enter a number : "
read n
rd=0
while [ $n -gt 0 ]do
       rem=`expr $n % 10` rd=`expr $rd
       \* 10 + $rem`n=`expr $n / 10`
done
echo "Reversed number is $rd"
```

Output

```
$ sh reverse.sh Enter a
number : 234
Reversed number is 432
```

I) Prime Number

```
# Prime number – prime.sh echo -n
"Enter the number : "read n
i=2
m=`expr $n / 2` until [ $i
-gt $m ]do
       q=`expr $n % $i`if [
       $q -eq 0 ]then
                echo "Not a Prime number"exit
       fi
       i=`expr $i + 1`done
echo "Prime number"
```

Output

```
$ sh prime.sh
Enter the number : 17Prime
number
```

Result

Thus shell scripts were executed using different programming constructs

Exp. No. 3a **FCFS Scheduling**

Date:

Aim

To schedule snapshot of processes queued according to FCFS scheduling.

Process Scheduling

- CPU scheduling is used in multiprogrammed operating systems.
- By switching CPU among processes, efficiency of the system can be improved.
- Some scheduling algorithms are FCFS, SJF, Priority, Round-Robin, etc.
- Gantt chart provides a way of visualizing CPU scheduling and enables to understand better.

First Come First Serve (FCFS)

- Process that comes first is processed first
- FCFS scheduling is non-preemptive
- Not efficient as it results in long average waiting time.
- Can result in starvation, if processes at beginning of the queue have long bursts.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. The *wtime* for first process is 0.
5. Compute *wtime* and *ttime* for each process as:
   a. $wtime_{i+1} = wtime_i + btime_i$
   b. $ttime_i = wtime_i + btime_i$
6. Compute average waiting time *awat* and average turnaround time *atur*
7. Display the *btime*, *ttime* and *wtime* for each process.
8. Display *awat* time and *atur*
9. Display GANTT chart for the above scheduling
10. Stop

Program

/* FCFS Scheduling - fcfs.c */

```
#include <stdio.h>
struct process
{
    int pid; int
    btime;int
    wtime;int
    ttime;
} p[10];

main()
{
    int  i,j,k,n,ttur,twat;float awat,atur;
    printf("Enter no. of process : ");scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ",(i+1));scanf("%d",
        &p[i].btime);
        p[i].pid = i+1;
    }
    p[0].wtime = 0; for(i=0;
    i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
        p[i].wtime + p[i].btime;
    }
    ttur = twat = 0; for(i=0;
    i<n; i++)
    {
        ttur += p[i].ttime;twat +=
        p[i].wtime;
    }
    awat = (float)twat / n;atur =
    (float)ttur / n;

    printf("\n                    FCFS  Scheduling\n\n");
    for(i=0; i<28; i++)
        printf("-");
    printf("\nProcess B-Time T-Time W-Time\n");for(i=0; i<28; i++)
        printf("-");
    for(i=0; i<n; i++)
        printf("\n P%d\t%4d\t%3d\t%2d", p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
    printf("\n"); for(i=0;
    i<28; i++)
        printf("-");

    printf("\n\nAverage waiting time                    : %5.2fms", awat);
    printf("\nAverage turn around time : %5.2fms\n", atur);

    printf("\n\nGANTT  Chart\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n");
    printf("|"); for(i=0; i<n;
    i++)
    {
        k = p[i].btime/2; for(j=0;
        j<k; j++)
              printf(" "); printf("P%d",p[i].pid);
        for(j=k+1;  j<p[i].btime;  j++)
              printf(" ");
        printf("|");
    }
    printf("\n");
    printf("-");
    for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
    printf("\n");
    printf("0"); for(i=0; i<n;
    i++)
    {
        for(j=0;  j<p[i].btime;  j++)printf(" ");
        printf("%2d",p[i].ttime);
    }
}
```

Output

```
Enter no. of process : 4
Burst time for process P1 (in ms) : 10Burst time for
process P2 (in ms) : 4Burst time for process P3 (in
ms) : 11Burst time for process P4 (in ms) : 6

          FCFS Scheduling

----------------------------------
Process   B-Time  T-Time  W-Time
----------------------------------
   P1       10      10       0
   P2        4      14      10
   P3       11      25      14
   P4        6      31      25
----------------------------------

Average waiting time         : 12.25ms
Average turn around time : 20.00ms


GANTT  Chart
--------------------------------------------------
|          P1        |    P2 |      P3       |    P4    |
--------------------------------------------------
0                  10      14               25        31
```

Result

Thus waiting time & turnaround time for processes based on FCFS scheduling wascomputed and the average waiting time was determined.

Exp. No. 3b SJF Scheduling

Date:

Aim

To schedule snapshot of processes queued according to SJF scheduling.

Shortest Job First (SJF)

- Process that requires smallest burst time is processed first.
- SJF can be preemptive or non–preemptive
- When two processes require same amount of CPU utilization, FCFS is used to break the tie.
- Generally efficient as it results in minimal average waiting time.
- Can result in starvation, since long critical processes may not be processed.

Algorithm

```
1.  Define an array of structure process with members pid, btime, wtime & ttime.
2.  Get length of the ready queue, i.e., number of process (say n)
3.  Obtain btime for each process.
4.  Sort the processes according to their btime in ascending order.
        a. If two process have same btime, then FCFS is used to resolve the tie.
5.  The wtime for first process is 0.
6.  Compute wtime and ttime for each process as:
        a. $wtime_{i+1} = wtime_i + btime_i$
        b. $ttime_i = wtime_i + btime_i$
7.  Compute average waiting time awat and average turn around time atur.
8.  Display btime, ttime and wtime for each process.
9.  Display awat and atur
10. Display GANTT chart for the above scheduling
11. Stop
```

Program

```
/* SJF Scheduling – sjf.c */#include
<stdio.h>
struct process
{
    int pid; int
    btime;int
    wtime;int
    ttime;
} p[10], temp;

main()
{
    int  i,j,k,n,ttur,twat;float awat,atur;

    printf("Enter no. of process : ");scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ",(i+1));scanf("%d",
        &p[i].btime);
        p[i].pid = i+1;
    }
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].btime > p[j].btime) ||
                (p[i].btime == p[j].btime && p[i].pid > p[j].pid))
            {
                temp  =  p[i]; p[i]
                =   p[j];   p[j]   =
                temp;
            }


        }
    }
    p[0].wtime = 0; for(i=0;
    i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
        p[i].wtime + p[i].btime;
    }
    ttur = twat = 0;
```

```
        for(i=0; i<n; i++)
        {
            ttur += p[i].ttime;twat +=
            p[i].wtime;
        }
        awat = (float)twat / n;atur =
        (float)ttur / n;

        printf("\n                SJF  Scheduling\n\n");
        for(i=0; i<28; i++)
            printf("-");
        printf("\nProcess B-Time T-Time W-Time\n");for(i=0; i<28; i++)
            printf("-"); for(i=0;
        i<n; i++)
            printf("\n P%-4d\t%4d\t%3d\t%2d", p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
        printf("\n"); for(i=0;
        i<28; i++)
            printf("-");
        printf("\n\nAverage waiting time                    : %5.2fms", awat);
        printf("\nAverage turn around time : %5.2fms\n", atur);

        printf("\n\nGANTT  Chart\n");
        printf("-");
        for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
        printf("\n|"); for(i=0;
        i<n; i++)
        {
            k = p[i].btime/2;for(j=0;
            j<k; j++)
                printf(" "); printf("P%d",p[i].pid);
            for(j=k+1;  j<p[i].btime;  j++)
                printf(" ");
            printf("|");
        }
        printf("\n-");
        for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
        printf("\n0"); for(i=0;
        i<n; i++)
        {
            for(j=0;  j<p[i].btime;  j++)printf(" ");
            printf("%2d",p[i].ttime);
        }
    }
```

Output

```
Enter no. of process : 5
Burst time for process P1 (in ms) : 10Burst time for
process P2 (in ms) : 6Burst time for process P3 (in
ms) : 5Burst time for process P4 (in ms) : 6Burst
time for process P5 (in ms) : 9

          SJF Scheduling

- - - - - - - - - - - - - - - - - - - - - - - - - - -
Process B-Time T-Time W-Time
-----------------------------
   P3        5      5      0
   P2        6     11      5
   P4        6     17     11
   P5        9     26     17
   P1       10     36     26
-----------------------------

Average   waiting    time        :  11.80ms

Average turn around time : 19.00ms


GANTT   Chart
-----------------------------------------------------
|  P3       |    P2    |   P4   |     P5     |     P1     |
-----------------------------------------------------
0           5         11        17          26          36
```

Result

Thus waiting time & turnaround time for processes based on SJF scheduling wascomputed and the average waiting time was determined.

Exp. No. 3c **Priority Scheduling**

Date:

Aim

To schedule snapshot of processes queued according to Priority scheduling.

Priority

- Process that has higher priority is processed first.
- Prioirty can be preemptive or non–preemptive
- When two processes have same priority, FCFS is used to break the tie.
- Can result in starvation, since low priority processes may not be processed.

Algorithm

```
1.  Define an array of structure process with members pid, btime, pri, wtime & ttime.
2.  Get length of the ready queue, i.e., number of process (say n)
3.  Obtain btime and pri for each process.
4.  Sort the processes according to their pri in ascending order.
      a. If two process have same pri, then FCFS is used to resolve the tie.
5.  The wtime for first process is 0.
6.  Compute wtime and ttime for each process as:
      a. $wtime_{i+1} = wtime_i + btime_i$
      b. $ttime_i = wtime_i + btime_i$
7.  Compute average waiting time awat and average turn around time atur
8.  Display the btime, pri, ttime and wtime for each process.
9.  Display awat and atur
10. Display GANTT chart for the above scheduling
11. Stop
```

Program

```
/* Priority Scheduling   - pri.c */
#include <stdio.h>
struct process
{
    int pid; int
    btime;int pri;
    int wtime;int
    ttime;
} p[10], temp;

main()
{
    int  i,j,k,n,ttur,twat;float awat,atur;

    printf("Enter no. of process : ");scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ", (i+1));scanf("%d",
        &p[i].btime);
        printf("Priority for process P%d : ", (i+1));scanf("%d",
        &p[i].pri);
        p[i].pid = i+1;
    }
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].pri > p[j].pri) ||
                (p[i].pri == p[j].pri && p[i].pid > p[j].pid) )
            {
                temp  =  p[i]; p[i]
                =    p[j];   p[j]    =
                temp;
            }

        }
    }
    p[0].wtime = 0; for(i=0;
    i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime;p[i].ttime =
        p[i].wtime + p[i].btime;
    }
```

```
        ttur = twat = 0; for(i=0;
        i<n; i++)
        {
            ttur += p[i].ttime;twat +=
            p[i].wtime;
        }
        awat = (float)twat / n;atur =
        (float)ttur / n;

        printf("\n\t  Priority  Scheduling\n\n");for(i=0; i<38;
        i++)
            printf("-");
        printf("\nProcess B-Time Priority T-Time   W-Time\n");
        for(i=0; i<38; i++)
            printf("-");
        for (i=0; i<n; i++)
            printf("\n P%-4d\t%4d\t%3d\t%4d\t%4d",
                p[i].pid,p[i].btime,p[i].pri,p[i].ttime,p[i].wtime);
        printf("\n");
        for(i=0; <38; i++)
            printf("-");
        printf("\n\nAverage waiting time                    : %5.2fms", awat);
        printf("\nAverage turn around time : %5.2fms\n", atur);
        printf("\n\nGANTT  Chart\n");
        printf("-");
        for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
        printf("\n|"); for(i=0;
        i<n; i++)
        {
            k = p[i].btime/2;for(j=0;
            j<k; j++)
                printf(" "); printf("P%d",p[i].pid);
            for(j=k+1;  j<p[i].btime;  j++)
                printf(" ");
            printf("|");
        }
        printf("\n-");
        for(i=0; i<(p[n-1].ttime + 2*n); i++)printf("-");
        printf("\n0"); for(i=0;
        i<n; i++)
        {
            for(j=0;  j<p[i].btime;  j++)printf(" ");
            printf("%2d",p[i].ttime);
        }
    }
```

Output

```
Enter no. of process : 5
Burst time for process P1          (in     ms) :  10
Priority for process P1 :          3
Burst time for process P2          (in     ms) :  7
Priority for process P2 :          1
Burst time for process P3          (in     ms) :  6
Priority for process P3 :          3
Burst time for process P4          (in     ms) :  13
Priority for process P4 :          4
Burst time for process P5          (in     ms) :  5
Priority for process P5 :          2


           Priority  Scheduling


----------------------------------------------------
Process   B-Time    Priority   T-Time   W-
                                        Time
----------------------------------------------------
  P2         7          1          7        0
  P5         5          2         12        7
  P1        10          3         22       12
  P3         6          3         28       22
  P4        13          4         41       28
----------------------------------------------------

Average waiting time              : 13.80ms
Average turn around time : 22.00ms

GANTT Chart

-----------------------------------------------------------------------
|     P2      |    P5   |        P1        |    P3     |         P4         |
-----------------------------------------------------------------------
0             7        12                 22          28                   41
```

Result

Thus waiting time & turnaround time for processes based on Priority scheduling was computed and the average waiting time was determined.

Exp. No. 3d Round Robin Scheduling

Date:

Aim

To schedule snapshot of processes queued according to Round robin scheduling.

Round Robin

- All processes are processed one by one as they have arrived, but in rounds.
- Each process cannot take more than the time slice per round.
- Round robin is a fair preemptive scheduling algorithm.
- A process that is yet to complete in a round is preempted after the time slice and putat the end of the queue.
- When a process is completely processed, it is removed from the queue.

Algorithm

1. Get length of the ready queue, i.e., number of process (say *n*)
2. Obtain *Burst* time B_i for each processes P_i.
3. Get the *time slice* per round, say *TS*
4. Determine the number of rounds for each process.
5. The wait time for first process is 0.
6. If $B_i > TS$ then process takes more than one round. Therefore turnaround and waiting time should include the time spent for other remaining processes in the same round.
7. Calculate *average* waiting time and turn around time
8. Display the GANTT chart that includes
    a. order in which the processes were processed in progression of rounds
    b. Turnaround time T_i for each process in progression of rounds.
9. Display the *burst* time, *turnaround* time and *wait* time for each process (in order ofrounds they were processed).
10. Display *average* wait time and turnaround time
11. Stop

Program

```
/* Round robin scheduling   - rr.c */
#include <stdio.h>
main()
{
      int  i,x=-1,k[10],m=0,n,t,s=0;
      int a[50],temp,b[50],p[10],bur[10],bur1[10];int
      wat[10],tur[10],ttur=0,twat=0,j=0;
      float  awat,atur;
      printf("Enter no. of process : ");scanf("%d",
      &n);
      for(i=0; i<n; i++)
      {
            printf("Burst time for process P%d : ", (i+1));scanf("%d",
            &bur[i]);
            bur1[i] = bur[i];
      }
      printf("Enter the time slice (in ms) : ");scanf("%d", &t);

      for(i=0; i<n; i++)
      {
            b[i] = bur[i] / t;
            if((bur[i]%t) != 0)
                  b[i] += 1;
            m += b[i];
      }


      printf("\n\t\tRound  Robin  Scheduling\n");

      printf("\nGANTT  Chart\n");for(i=0; i<m;
      i++)
            printf("------------------ ");
      printf("\n");

      a[0] = 0;
      while(j < m)
      {
            if(x == n-1)x =
                  0;
            else
                  x++;
            if(bur[x] >= t)
            {
                  bur[x] -= t;
                  a[j+1] = a[j] + t;
```

```
        if(b[x] == 1)
        {
            p[s] = x;
            k[s] = a[j+1];s++;
        }
        j++;
        b[x] -= 1;
        printf("   P%d              |", x+1);
    }
    else if(bur[x] != 0)
    {
        a[j+1] = a[j] + bur[x];bur[x] =
        0;
        if(b[x] == 1)
        {
            p[s] = x;
            k[s] = a[j+1];s++;
        }
        j++;
        b[x] -= 1;
        printf("   P%d   |",x+1);
    }
}
printf("\n");
for(i=0;i<m;i++)
    printf("-----------------");
printf("\n");

for(j=0; j<=m; j++) printf("%d\t",  a[j]);

for(i=0; i<n; i++)
{
    for(j=i+1; j<n; j++)
    {
        if(p[i] > p[j])
        {
            temp  =  p[i]; p[i]
            =    p[j];   p[j]    =
            temp;

            temp  =  k[i]; k[i]
            =    k[j];   k[j]    =
            temp;
        }
    }
}
```

```
    for(i=0; i<n; i++)
    {
        wat[i] = k[i] - bur1[i];tur[i] =
        k[i];
    }
    for(i=0; i<n; i++)
    {
        ttur += tur[i];twat +=
        wat[i];
    }

    printf("\n\n"); for(i=0;
    i<30; i++)
        printf("-"); printf("\nProcess\tBurst\tTrnd\tWait\n");
    for(i=0; i<30; i++)
        printf("-");
    for (i=0; i<n; i++)
        printf("\nP%-4d\t%4d\t%4d\t%4d",   p[i]+1,   bur1[i],tur[i],wat[i]);
    printf("\n"); for(i=0;
    i<30; i++)
        printf("-");

    awat = (float)twat / n;atur =
    (float)ttur / n;
    printf("\n\nAverage waiting time                    : %.2f ms", awat);
    printf("\nAverage turn around time : %.2f ms\n", atur);
}
```

Output

```
Enter no. of process : 5 Burst time for
process P1 : 10Burst time for process P2
: 29Burst time for process P3 : 3Burst
time for process P4 : 7Burst time for
process P5 : 12
Enter the time slice (in ms) : 10

                    Round Robin Scheduling

GANTT Chart
-------------------------------------------------------------------------------
P1       |   P2     |      P3  |   P4   |  P5     |    P2     |    P5    |     P2   |
-------------------------------------------------------------------------------
0       10          20         23          30         40          50         52          61

- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Process Burst            Trnd        Wait
-------------------------------------
P1             10          10           0
P2             29          61          32
P3              3          23          20
P4              7          30          23
P5             12          52          40
-------------------------------------

Average waiting time              : 23.00 ms
Average turn around time : 35.20 ms
```

Result

Thus waiting time and turnaround time for processes based on Round robin scheduling was computed and the average waiting time was determined.

Exp. No. 4a **Contiguous Allocation**

Date:

Aim

To implement file allocation on free disk space in a contiguous manner.

File Allocation

- The three methods of allocating disk space are:
  1. Contiguous allocation
  2. Linked allocation
  3. Indexed allocation

Contiguous

- Each file occupies a set of contiguous block on the disk.
- The number of disk seeks required is minimal.
- The directory contains address of starting block and number of contiguous block(length) occupied.
- Supports both sequential and direct access.
- First / best fit is commonly used for selecting a hole.

Algorithm

1. Assume no. of blocks in the disk as 20 and all are free.
2. Display the status of disk blocks before allocation.
3. For each file to be allocated:
   a. Get the *filename*, *start* address and file *length*
   b. If *start* + *length* > 20, then goto step 2.
   c. Check to see whether any block in the range (start, start + length-1) isallocated. If so, then go to step 2.
   d. Allocate blocks to the file contiguously from start block to start + length – 1.
4. Display directory entries.
5. Display status of disk blocks after allocation
6. Stop

Program

/* Contiguous Allocation - cntalloc.c */

```
#include <stdio.h>
#include <string.h>
int num=0, length[10], start[10];char
fid[20][4], a[20][4];

void directory()
{
    int i;
    printf("\nFile  Start  Length\n");for(i=0;
    i<num; i++)
        printf("%-4s  %3d  %6d\n",fid[i],start[i],length[i]);
}


void display()
{
    int i;
    for(i=0; i<20; i++)
        printf("%4d",i);
    printf("\n"); for(i=0;
    i<20; i++)
        printf("%4s",  a[i]);
}


main()
{
    int  i,n,k,temp,st,nb,ch,flag;char id[4];
    for(i=0; i<20; i++)strcpy(a[i],  "");
    printf("Disk space before allocation:\n");display();

    do
    {
        printf("\nEnter File name (max 3 char) : ");scanf("%s",id);
        printf("Enter start block : ");scanf("%d",
        &st);
        printf("Enter no. of blocks : ");scanf("%d",
        &nb); strcpy(fid[num], id);
        length[num] = nb;
        flag = 0;
```

```
            if((st+nb) > 20)
            {
                printf("Requirement exceeds range\n");continue;
            }

            for(i=st; i<(st+nb); i++)
                if(strcmp(a[i], "") != 0)
                    flag = 1;
            if(flag == 1)
            {
                printf("Contiguous allocation not possible.\n");continue;
            }

            start[num] = st; for(i=st;
            i<(st+nb); i++)
                strcpy(a[i], id);; printf("Allocation
            done\n");num++;

            printf("\nAny more allocation (1. yes / 2. no)? : ");
            scanf("%d", &ch);
        } while (ch == 1);

        printf("\n\t\t\tContiguous Allocation\n");printf("Directory:");
        directory();
        printf("\nDisk space after allocation:\n");display();
        printf("\n");
    }
```

Output

```
Disk space before allocation:
   0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19

Enter File name (max 3 char) : ls
Enter start block : 3
Enter no. of blocks : 4
Allocation done

Any more allocation (1. yes / 2. no)? : 1Enter

File name (max 3 char) : cp
Enter start block : 14
Enter no. of blocks : 3
Allocation done

Any more allocation (1. yes / 2. no)? : 1Enter

File name (max 3 char) : tr
Enter start block : 18
Enter no. of blocks : 3
Requirement exceeds
range

Enter File name (max 3 char) : tr
Enter start block : 10
Enter no. of blocks : 3
Allocation done

Any more allocation (1. yes / 2. no)? : 1Enter

File name (max 3 char) : mv
Enter start block : 0
Enter no. of blocks : 2
Allocation done

Any more allocation (1. yes / 2. no)? : 1Enter

File name (max 3 char) : ps
Enter start block : 12
Enter no. of blocks : 3
Contiguous allocation not possible.

Any more allocation (1. yes / 2. no)? : 2

                    Contiguous Allocation
Directory:
File Start Length
ls      3      4
cp     14      3
tr     10      3
mv      0      2
```

```
Disk space after allocation:

0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
mv            ls  ls ls ls              tr  tr tr       cp  cp  cp
mv
```

Result

Thus contiguous allocation is done for files with the available free blocks.

Exp. No. 4b. **Linked Allocation**

Date:

Aim

To implement file allocation on free disk space as a linked list of disk blocks.

Linked

- Each file is a linked list of disk blocks.
- The directory contains a pointer to first and last blocks of the file.
- The first block contains a pointer to the second one, second to third and so on.
- File size need not be known in advance, as in contiguous allocation.
- No external fragmentation.
- Supports sequential access only.

Indexed

- In indexed allocation, all pointers are put in a single block known as index block.
- The directory contains address of the index block.
- The ith entry in the index block points to ith block of the file.
- Indexed allocation supports direct access.
- It suffers from pointer overhead, i.e wastage of space in storing pointers.

Algorithm

```
1. Define file table as a linked list structure
2. Get number of files to be stored.
3. For each file:
    a. Obtain number of disk blocks
    b. Obtain randomly allocated disk blocks
    c. Create a single linked list of nodes for the specified blocks.
4. Get the filename to be searched.
5. List disk blocks of that file as a linked list
6. Stop
```

Program

/* Linked list file allocation */

```
#include <stdio.h>

struct filetable
{
      char name[20];int
      nob;
      struct block *sb;
} ft[30];

struct block
{
      int bno;
      struct block *next;
};


main()
{
      int i, j, n;char
      str[20];
      struct block *temp;

      printf("Enter no. of files: ");scanf("%d",
      &n);
      for(i=0; i<n; i++)
      {
            printf("\nEnter file name %d : ",i+1);scanf("%s",
            ft[i].name);
            printf("Enter no of blocks in file %d : ", i+1);scanf("%d",
            &ft[i].nob);

            ft[i].sb = (struct block*)malloc(sizeof(struct block));temp = ft[i].sb;
            printf("Enter the disk blocks : ");scanf("%d",
            &temp->bno);
            temp->next = NULL; for(j=1;
            j<ft[i].nob; j++)
            {
                  temp->next = (struct block*)malloc(sizeof(struct
block));
                  temp = temp->next; scanf("%d",
                  &temp->bno);
            }

            temp->next = NULL;
```

```
    }
    printf("\nEnter file name to be searched : ");scanf("%s", str);
    for(i=0; i<n;  i++) if(strcmp(str,
        ft[i].name)==0)
            break;if(i
    == n)
        printf("\nFile  Not  Found");else
    {
        printf("\nFilename   No. of Blocks                  Blocks  Occupied");
        printf("\n             %s\t\t%d\t", ft[i].name, ft[i].nob); temp = ft[i].sb;
        for(j=0; j<ft[i].nob; j++)
        {
            printf("%d -> ", temp->bno);temp =
            temp->next;
        }
        printf("NULL");
    }
}
```

Output

```
Enter no. of files: 3 Enter file name

1 : hello.c
Enter no. of blocks in file 1 : 3Enter the disk
blocks : 12 23 34

Enter file name 2 : first.cpp Enter no. of
blocks in file 2 : 3Enter the disk blocks : 22
33 44

Enter file name 3 : profile.doc Enter no. of
blocks in file 3 : 3Enter the disk blocks : 87
76 65

Enter file name to be searched : first.cpp

Filename   No. of Blocks    Blocks Occupied first.cpp     3
                            22 -> 33 -> 44 -> NULL
```

Result

Thus linked list allocation is done for files with the available free blocks.

Exp. No. 5 **Producer-Consumer Synchronization**

Date:

Aim

To synchronize producer and consumer processes using semaphore.

Semaphore

- A semaphore is a counter used to synchronize access to a shared data amongst multiple processes.
- To obtain a shared resource, the process should:
    - Test the semaphore that controls the resource.
    - If value is positive, it gains access and decrements value of semaphore.
    - If value is zero, the process goes to sleep and awakes when value is > 0.
- When a process relinquishes resource, it increments the value of semaphore by 1.

Producer-Consumer problem

- A producer process produces information to be consumed by a consumer process
- A producer can produce one item while the consumer is consuming another one.
- With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if buffer is full.
- The buffer can be implemented using any IPC facility.

Algorithm

```
1.  Define semaphore variables full, empty and mutex
2.  Define wait and signal operation
3.  Display menu-driven and accept user choice.
4.  If choice = 1 then
      i.   Call wait (empty)
      ii.  Call wait (mutex)
      iii. If buffer is not full then append item to buffer
      iv.  Call signal (full)
      v.   Call signal (mutex)
5.  If choice = 2 then
      i.   Call wait (full)
      ii.  Call wait (mutex)
      iii. If buffer is not empty then remove first item from the buffer
      iv.  Call signal (mutex)
      v.   Call signal (empty)
6.  If choice = 3 then display buffer contents
7.  Stop
```

Program

```
/* Producer-Consumer problem using semaphore – pcsem.c */
#include <stdio.h>
#include <string.h>
#define size 5struct
process
{
    char item[10];
}p[10];
int flag=0, full=0, empty=size, mutex=1;
int wait(int s)
{
    if(s==0)
        flag=1;else
        s--;
    return s;
}
int signal(int s)
{
    s++;
    return s;
}


main()
{
    int c, i;
    printf("\nProducer-Consumer  Problem\n");

    while(1)
    {
        printf("\n1.Produce  2.Consume  3.Display  4.Exit\n");printf("Enter
        your choice : ");
        scanf("%d", &c);

        switch(c)
        {
            case 1:
                empty    =    wait(empty);
                mutex    =    wait(mutex);
                if(flag == 0)
                {
                    printf("Enter the item to produce : ");scanf("%s",
                    p[full].item);
                    full = signal(full);
                }
                else
                {
                    printf("\nBuffer  is  FULL\n");flag = 0;
                }
                mutex  =  signal(mutex);break;

            case 2:
                full = wait(full); mutex =
                wait(mutex);if(flag == 0)
                {
                    printf("Item  %s  is  consumed\n",p[0].item);for(i=0; i<size; i++)
                            strcpy(p[i].item,   p[i+1].item);flag=0;
                }
                else
                {
                    printf("\nBuffer  is  EMPTY\n");flag = 0;
                }
                mutex = signal(mutex); empty
                = signal(empty);break;

            case 3:
                if(full != 0)
                {
                    printf("\nItems in the buffer : ");for(i=0; i<full;
                    i++)
                            printf("\n%s",  p[i].item);
                }
                else
                {
                    printf("\nBuffer  is  EMPTY\n");flag = 0;
                }
                break;

            case 4:
                exit(0);
                break;
        }
    }
}
```

Output

```
Producer-Consumer  Problem

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 1
Enter the item to produce : bread

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 1
Enter the item to produce : butter

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 1
Enter the item to produce : bun

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 1
Enter the item to produce : jam

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 2
Item bread is consumed

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 2
Item butter is consumed

1.Produce 2.Consume 3.Display 4.ExitEnter
your choice : 3

Items in the buffer :
bun
jam

1. Produce 2.Consume 3.Display 4.ExitEnter
your choice : 4
```

Result

Thus synchronization between producer and consumer process for access to a sharedmemory segment is implemented.

Exp. No. 6a **Single-Level Directory**

Date:

Aim

To create directory structure as a single level directory structure.

Single-Level Directory

- All files are contained in the same directory,
- Easy to implement
- Filenames must be unique within a directory
- Difficult to remember all filenames
- Leads to anamoly in a multi-user system

Algorithm

```
1. Read number of directories
2. For each directory
      a. Read directory name
      b. Read number of files in that directory
      c. Read filenames for that directory
3. Display directory name and their corresponding files
4. Stop
```

Program

/* Single Level Directory - singlev.c */

```
#include <stdio.h>
int nod,  nof[20]; char
file[20][20][20];
char dir[20][20];int i,j;

main()
{
      printf("No. of Directories : ");scanf("%d",
      &nod);

      printf("\nEnter the directory details\n");for(i=0; i<nod; i++)
      {
            printf("\nDirectory Name : ");
            scanf("%s", &dir[i]);
            printf("No. of Files in the directory : ");scanf("%d",
            &nof[i]);
            printf("Enter the filenames :\n");for(j=0;
            j<nof[i]; j++)
                      scanf("%s",  &file[i][j]);
      }

      printf("\nDirectory  Filenames\n");for(i=0;
      i<nod; i++)
      {
            printf("%s\t",  dir[i]);for(j=0; j<nof[i];
            j++)
                  printf("%s  ",  file[i][j]);printf("\n");
      }
}
```

Output

```
No. of Directories : 3 Enter the

directory details

Directory Name : pds2
No. of Files in the directory : 3Enter the
filenames :
inherit.cpp
poly.cpp
ovld.cpp

Directory Name : os
No. of Files in the directory : 4Enter the
filenames :
fcfs.c pcsem.c
deadlock.clru.c

Directory Name : java
No. of Files in the directory : 2Enter the
filenames :
hello.javaswing.java

Directory   Filenames
pds2        inherit.cpp poly.cpp ovld.cpp os
            fcfs.c pcsem.c deadlock.c lru.cjava
            hello.java swing.java
```

Result

Thus single-level directory structure has been demonstrated.

Exp. No. 6b Two-Level Directory

Date:

Aim

To create directory structure as a two-level directory structure.

Two-Level Directory

- Each user has a user file directory (UFD) that lists folders and files of that user
- When a user refers to a particular file, only his own UFD is searched.
- The two-level directory structure solves the name-collision problem
- It isolates one user from another.

Algorithm

```
1. Read number of users
2. For each user
     i.   Read directory name
     ii.  Read number of folders
     iii. Read filenames for that folder
3. Display files and folders for that user
4. Stop
```

Program

```
/* Two-level directory */

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

struct st
{
    char uname[10]; char
    dname[10][10];
    char  fname[10][10][15];int
    ds,sds[10];
}dir[10];

int main()
{
    int i,j,k,n;

    printf("No. of Users : ");
    scanf("%d", &n);

    for(i=0; i<n; i++)
    {
        printf("\nUser-%d Name : ", i+1);
        scanf("%s", &dir[i].uname);

        printf("No. of folders : ");scanf("%d",
        &dir[i].ds);

        for(j=0; j<dir[i].ds; j++)
        {
            printf("\nEnter folder name : ");scanf("%s",
            &dir[i].dname[j]); printf("No. of files : ");
            scanf("%d", &dir[i].sds[j]); printf("Enter
            filenames:\n"); for(k=0; k<dir[i].sds[j]; k++)
                scanf("%s",   &dir[i].fname[j][k]);
        }
    }
```

```
        printf("\n\tTwo-Level  Directory  Structure\n");
        printf("\nUser\tFolders\tFiles\n\n");
        for(i=0; i<n; i++)
        {
              printf("%s",  dir[i].uname);for(j=0;
              j<dir[i].ds; j++)
              {
                    printf("\t%s\t",  dir[i].dname[j]);
                    for(k=0; k<dir[i].sds[j]; k++) printf("%-15s  ",
                          dir[i].fname[j][k]);
                    printf("\n");
              }
              printf("\n");
        }
  }
```

Output

```
No. of Users : 2

User-1 Name : vijaiNo. of
folders : 2

Enter folder name : networkNo. of
files : 2
Enter  filenames:
udpdns.javatcpchat.java

Enter folder name : pds2No. of
files : 2
Enter  filenames:
inherit.cppvirtual.cpp

User-2 Name : anandNo.
of folders : 2

Enter folder name : osNo.  of
files  :  3 Enter filenames:
sjf.c pcsem.c
bankeralgo.c

Enter folder name : networkNo. of
files : 2
Enter  filenames:
tcpchat.javasniffdata.c

          Two-Level  Directory  StructureUser

          Folders Files

vijai     network udpdns.java         tcpchat.javapds2
                    inherit.cpp         virtual.cpp

anand     os        sjf.c               pcsem.c             bankeralgo.c
          network tcpchat.java        sniffdata.c
```

Result

Thus two-level directory structure has been demonstrated.

Exp. No. 6c Hirearchical Directory Structure

Date:

Aim

To demonstrate tree-like hierarchical directory structure graphically.

Tree-Structured Directories

- A tree is the most common directory structure.
- Extends two-level directory structure to a tree of arbitrary height.
- It allows users to create their own subdirectories and organize their files.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- Current directory contains files that are required for that process.
- Path names can be of two types: absolute and relative.

Algorithm

1. Define tree structure
2. Initialize graphics
3. Recursively obtain user files and folders under root in hierarchy
4. Display the directory structure graphically
5. Stop

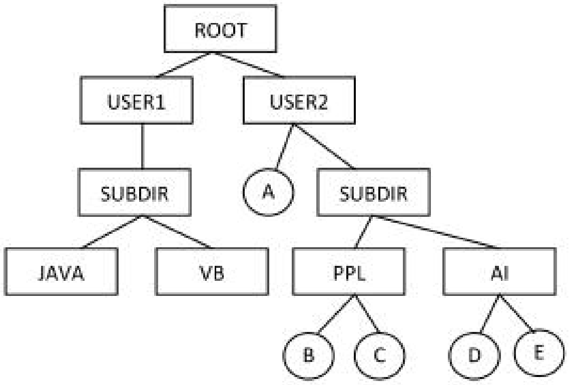
Program

/* Hierarchical directory structure - treedir.c */

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

struct  tree_element
{
    char name[20];
    int x, y, ftype, lx, rx, nc, level;struct
    tree_element *link[5];
};

typedef struct tree_element node;

main()
{
    int gd=DETECT, gm;
    node *root;
    root = NULL;

    clrscr();
    create(&root, 0, "root", 0, 639, 320);clrscr();
    initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
    display(root);
    getch(); closegraph();
}

create(node **root,int lev,char *dname,int lx,int rx,int x)
{
    int i, gap; if(*root ==
    NULL)
    {
        (*root) = (node *)malloc(sizeof(node));
        printf("Enter name of dir/file(under %s) : ", dname);fflush(stdin);
        gets((*root)->name);
        printf("enter 1 for Dir / 2 for file : ");scanf("%d",
        &(*root)->ftype);
        (*root)->level = lev; (*root)->y =
        50 + lev * 50;(*root)->x = x;
        (*root)->lx = lx;
        (*root)->rx = rx;

        for(i=0;i<5;i++)
            (*root)->link[i] = NULL;
            if((*root)->ftype == 1)
            {
                  printf("No of sub directories/files(for %s): ",(*root)->name);
                  scanf("%d",  &(*root)->nc);
                  if((*root)->nc == 0)
                        gap = rx - lx;else
                        gap = (rx - lx) / (*root)->nc;for(i=0;
                  i<(*root)->nc; i++)
                        create(&((*root)->link[i]),  lev+1,  (*root)->name,lx+gap* i,
lx+gap*i+gap, lx+gap*i+gap/2);
            }
            else
            (*root)->nc = 0;
      }
}

display(node *root)
{
      int i;
      settextstyle(2, 0, 4);
      settextjustify(1, 1);setfillstyle(1,  BLUE);
      setcolor(14);
      if(root != NULL)
      {
            for(i=0; i<root->nc; i++)
                  line(root->x,              root->y,           root->link[i]->x,                root-
>link[i]->y);
            if(root->ftype == 1)
                  bar3d(root->x-20, root->y-10, root->x+20, root->y+10,
0, 0);
            else
                  fillellipse(root->x, root->y, 20, 20);
            outtextxy(root->x,  root->y,  root->name);for(i=0; i<root-
            >nc; i++)
                  display(root->link[i]);
      }
}
```

Output

```
Enter Name of dir/file(under root):   ROOTEnter 1
for Dir / 2 for File:   1
No of subdirectories/files(for ROOT): 2 Enter
Name of dir/file(under ROOT): USER1Enter 1 for
Dir / 2 for File: 1
No of subdirectories/files(for USER1): 1 Enter Name
of dir/file(under USER1): SUBDIR1Enter 1 for Dir / 2
for File: 1
No of subdirectories/files(for SUBDIR1): 2Enter
Name of dir/file(under USER1): JAVAEnter 1 for
Dir / 2 for File: 1
No of subdirectories/files(for JAVA): 0 Enter
Name of dir/file(under SUBDIR1): VBEnter 1 for
Dir / 2 for File: 1
No of subdirectories/files(for VB): 0 Enter Name
of dir/file(under ROOT): USER2Enter 1 for Dir / 2
for File: 1
No of subdirectories/files(for USER2): 2Enter
Name of dir/file(under ROOT): A Enter 1 for Dir
/ 2 for File: 2
Enter Name of dir/file(under USER2): SUBDIR2Enter
1 for Dir / 2 for File: 1
No  of  subdirectories/files(for  SUBDIR2):  2 Enter
Name of dir/file(under SUBDIR2): PPLEnter 1 for
Dir / 2 for File: 1
No of subdirectories/files(for PPL): 2Enter
Name of dir/file(under PPL): B Enter 1 for Dir
/ 2 for File: 2
Enter Name of dir/file(under PPL): CEnter 1
for Dir / 2 for File: 2
Enter Name of dir/file(under SUBDIR): AIEnter
1 for Dir / 2 for File: 1
No of subdirectories/files(for AI): 2Enter
Name of dir/file(under AI): D Enter 1 for Dir
/ 2 for File: 2 Enter Name of dir/file(under
AI): E
```



Result

Thus a hierarchical directory structure has been created and shown graphically

Exp. No. 7 **Bankers Algorithm**

Date:

Aim

To avoid deadlocks to a resource allocation system with multiple instances using bankersalgorithm.

Banker's Algorithm

- Data structures maintained are:
  - Available—vector of available resources
  - Max—matrix contains demand of each process
  - Allocation—matrix contains resources allocated to each process
  - Need—matrix contains remaining resource need of each process
- Safety algorithm is used to determine whether system is in a safe state
- Resource request algorithm determines whether requests can be safely granted

Algorithm

1. Read number of resources
2. Read max. instances of each resource type
3. Read number of process
4. Read allocation matrix for each process
5. Read max matrix for each process
6. Display available resources
7. Display need matrix using formula Need = Max - Allocation
8. Determine the order of process to be executed for a safe state
9. Stop

Program

```
/* Banker algorithm for deadlock avoidance - bankersalgo.c */
#include <stdio.h>
#include  <conio.h>
main()
{
    int  output[10],  ins[5],  avail[5],  allocated[10][5];int need[10][5],
    max[10][5], p[10];
    int k=0, d=0, t=0, i, pno, j, nor, count=0;

    printf("Enter number of resources : ");scanf("%d",
    &nor);

    printf("\nEnter max instances of each resources\n");for (i=0; i<nor;
    i++)
    {
        avail[i]=0;
        printf("%c = ",(i+65));
        scanf("%d",  &ins[i]);
    }
    printf("\nEnter the No. of processes : ");
    scanf("%d", &pno);
    printf("\nEnter Allocation matrix \n ");
    for(i=0; i<nor; i++)
        printf("\t%c",  (i+65));printf("\n");

    for(i=0; i<pno; i++)
    {
        p[i]=i; printf("P%d\t",  p[i]);
        for (j=0; j<nor; j++)
        {
            scanf("%d",   &allocated[i][j]);avail[j] +=
            allocated[i][j];
        }
    }
    printf("\nEnter Max matrix \n ");for(i=0;
    i<nor; i++)
    {
        printf("\t%c",  (i+65));
        avail[i] = ins[i] - avail[i];
    }
    printf("\n");
    for (i=0; i<pno; i++)
    {
        printf("P%d\t",i); for (j=0;
        j<nor; j++)
```

```
            scanf("%d",  &max[i][j]);
    }
    printf("\n");
    printf("Available resources are : \n");
    for(i=0; i<nor; i++)
        printf("%c = %d \n", (i+65), avail[i]);

    printf("\nNeed matrix is :\n");for(i=0;
    i<nor; i++)
        printf("\t%c",  (i+65));printf("\n");
    for (i=0; i<pno; i++)
    {
        printf("P%d\t",i); for (j=0;
        j<nor; j++)
            printf("%d\t", max[i][j]-allocated[i][j]);printf("\n");
    }
A:
    d = -1;
    for (i=0;i <pno; i++)
    {
        count = 0;t =
        p[i];
        for (j=0; j<nor; j++)
        {
            need[t][j] = max[t][j] - allocated[t][j];if(need[t][j] <=
            avail[j])
                count++;
        }
        if(count == nor)
        {
            output[k++] = p[i]; for (j=0;
            j<nor; j++)
                avail[j] += allocated[t][j];
        }
        else
            p[++d] = p[i];
    }
    if(d != -1)
    {
        pno = d + 1;
        goto A;
    }
    printf("\n Process Execution Order : ");printf("<");
    for (i=0; i<k; i++)
        printf(" P%d ", output[i]);printf(">");
}
```

Output

```
Enter number of resources : 3

Enter max instances of each resourcesA = 10
B = 5
C = 7

Enter the No. of processes : 5Enter

Allocation matrix
          A         B         C
P0        0         1         0
P1        2         0         0
P2        3         0         2
P3        2         1         1
P4        0         0         2

Enter Max matrix
          A         B         C
P0        7         5         3
P1        3         2         2
P2        9         0         2
P3        2         2         2
P4        4         3         3
Available resources are :
A = 3
B = 3
C = 2

Need matrix is :
          A         B         C
P0        7         4         3
P1        1         2         2
P2        6         0         0
P3        0         1         1
P4        4         3         1

Process Execution Order : < P1   P3   P4   P0   P2 >
```

Result

Thus deadlock is avoided for multiple instances of resources using bankers algorithm.

Exp. No. 8 **Deadlock Detection**

Date:

Aim

To detect whether the given system is in a deadlocked state or not.

Deadlock Detection

- Data structures used are Available, Allocation and Request
- Detection algorithm checks every possible allocation sequence for all processes
- Resources allocated to deadlocked processes will be idle until deadlock is broken
- Deadlocks occur only when process request cannot be granted immediately.
- Deadlock eventually cripples system throughput and causes CPU utilization to drop

Algorithm

1. See if any Processes Requests can be satisfied.
2. If so satisfy the needs and remove that Process and all the Resources it holds
3. Repeat step1 for all processes
4. If all Processes are finally removed then there is no Deadlock
5. List the deadlocked process

Program

/* Deadlock detection - deaddeduct.c */

```
#include <stdio.h>
main()
{
    int found, flag, l, i, j, k=1, sum=0, tp, tr; int p[8][8], c[8][8],
    m[8], r[8], a[8], temp[8];

    printf("Enter No. of Processes : ");scanf("%d",
    &tp);
    printf("Enter No. of Resources : ");scanf("%d",
    &tr);

    printf("\nEnter Claim / Request matrix :\n");for(i=1; i<=tp;
    i++)
        for(j=1; j<=tr; j++) scanf("%d",
            &c[i][j]);

    printf("\nEnter Allocation matrix : \n");for(i=1; i<=tp;
    i++)
        for(j=1; j<=tr; j++) scanf("%d",
            &p[i][j]);

    printf("\nEnter  Total  resources  :\n");for(i=1; i<=tr;
    i++)
        scanf("%d", &r[i]);

    printf("\nEnter  Availability  vector  :\n");for(i=1; i<=tr; i++)
    {
        scanf("%d",  &a[i]);temp[i] = a[i];
    }

    for(i=1; i<=tp; i++)
    {
        sum = 0;
        for(j=1; j<=tr; j++)sum +=
            p[i][j];
        if(sum == 0)
        {
            m[k] = i;
            k++;
        }
    }
    for(i=1; i<=tp; i++)
    {
        for(l=1; l<k; l++)if(i !=
        m[l])
        {
            flag = 1;
            for(j=1; j<=tr; j++)
            {
                if(c[i][j] < temp[j])
                {
                    flag = 0;
                    break;
                }
            }
        }
        if(flag == 1)
        {
            m[k] = i;
            k++;
            for(j=1; j<=tr; j++) temp[j]
                += p[i][j];
        }
    }

    printf("Deadlock causing processes are : ");for(j=1; j<=tp;
    j++)
    {
        found = 0; for(i=1; i<k;
        i++)
        {
            if(j == m[i])
                found = 1;
        }
        if(found == 0) printf("P%d
            ", j);
    }
}
```

Output

```
Enter No. of Processes : 4Enter No.
of Resources : 5

Enter Claim / Request matrix :
0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

Enter Allocation matrix :
1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0

Enter Total resources :
2 1 1 2 1

Enter Availability vector :
0 0 0 0 1

Deadlock causing processes are : P2   P3
```

Result

Thus given system is checked for deadlock and deadlocked processes are listed out.

Exp. No. 9a **FIFO Page Replacement**

Date:

Aim

To implement demand paging for a reference string using FIFO method.

FIFO

- Page replacement is based on when the page was brought into memory.
- When a page should be replaced, the oldest one is chosen.
- Generally, implemented using a FIFO queue.
- Simple to implement, but not efficient.
- Results in more page faults.
- The page-fault may increase, even if frame size is increased (Belady's anomaly)

Algorithm

```
1.  Get length of the reference string, say l.
2.  Get reference string and store it in an array, say rs.
3.  Get number of frames, say nf.
4.  Initalize frame array upto length nf to -1.
5.  Initialize position of the oldest page, say j to 0.
6.  Initialize no. of page faults, say count to 0.
7.  For each page in reference string in the given order, examine:
        a. Check whether page exist in the frame array
        b. If it does not exist then
              i.   Replace page in position j.
              ii.  Compute page replacement position as (j+1) modulus nf.
              iii. Increment count by 1.
              iv.  Display pages in frame array.
8.  Print count.
9.  Stop
```

Program

/* FIFO page replacement - fifopr.c */

```
#include <stdio.h>

main()
{
    int  i,j,l,rs[50],frame[10],nf,k,avail,count=0;

    printf("Enter length of ref. string : ");scanf("%d", &l);
    printf("Enter reference string :\n");for(i=1; i<=l; i++)
        scanf("%d", &rs[i]); printf("Enter number of
    frames : ");scanf("%d", &nf);

    for(i=0; i<nf; i++)
        frame[i] = -1;
    j = 0;

    printf("\nRef. str  Page frames");for(i=1; i<=l;
    i++)
    {
        printf("\n%4d\t", rs[i]);avail = 0;
        for(k=0; k<nf; k++) if(frame[k]
            == rs[i])
                avail = 1;
        if(avail == 0)
        {
            frame[j] = rs[i];j =
            (j+1) % nf; count++;
            for(k=0; k<nf; k++) printf("%4d",
                frame[k]);
        }
    }
    printf("\n\nTotal no. of page faults : %d\n",count);
}
```

Output

```
Enter length of ref. string : 20Enter
reference string :
1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter number of frames : 5Ref. str

Page frames
    1        1   -1   -1   -1   -1
    2        1    2   -1   -1   -1
    3        1    2    3   -1   -1
    4        1    2    3    4   -1
    2
    1
    5        1    2    3    4    5
    6        6    2    3    4    5
    2
    1        6    1    3    4    5
    2        6    1    2    4    5
    3        6    1    2    3    5
    7        6    1    2    3    7
    6
    3
    2
    1
    2
    3
    6
Total no. of page faults : 10
```

Result

Thus page replacement was implemented using FIFO algorithm.

Exp. No. 9b **LRU Page Replacement**

Date:

Aim

To implement demand paging for a reference string using LRU method.

LRU

- Pages used in the recent past are used as an approximation of future usage.
- The page that has not been used for a longer period of time is replaced.
- LRU is efficient but not optimal.
- Implementation of LRU requires hardware support, such as counters/stack.

Algorithm

1. Get length of the reference string, say *len*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Create *access* array to store counter that indicates a measure of recent usage.
5. Create a function *arrmin* that returns position of minimum of the given array.
6. Initalize *frame* array upto length *nf* to -1.
7. Initialize position of the page replacement, say *j* to 0.
8. Initialize *freq* to 0 to track page frequency
9. Initialize no. of page faults, say *count* to 0.
10. For each page in reference string in the given order, examine:
    - a. Check whether page exist in the *frame* array.
    - b. If page exist in memory then
        - i. Store incremented *freq* for that page position in *access* array.
    - c. If page does not exist in memory then
        - i. Check for any empty frames.
        - ii. If there is an empty frame,
            - Assign that frame to the page
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        - iii. If there is no free frame then
            - Determine page to be replaced using *arrmin* function.
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        - iv. Display pages in *frame* array.
11. Print *count*.
12. Stop

Program

/* LRU page replacement - lrupr.c */

```
#include <stdio.h>

int arrmin(int[], int);

main()
{
    int i,j,len,rs[50],frame[10],nf,k,avail,count=0;int access[10],
    freq=0, dm;

    printf("Length of Reference string : ");scanf("%d",
    &len);
    printf("Enter reference string :\n");for(i=1; i<=len;
    i++)
        scanf("%d", &rs[i]); printf("Enter no. of
    frames : ");scanf("%d", &nf);

    for(i=0; i<nf; i++)
        frame[i] = -1;
    j = 0;

    printf("\nRef. str Page frames");for(i=1;
    i<=len; i++)
    {
        printf("\n%4d\t", rs[i]);avail = 0;
        for(k=0; k<nf; k++)
        {
            if(frame[k] == rs[i])
            {
                avail = 1; access[k] =
                ++freq;break;
            }
        }
        if(avail == 0)
        {
            dm = 0;
            for(k=0; k<nf; k++)
            {
                if(frame[k] == -1)
                    dm = 1;
                    break;
            }
            if(dm == 1)
            {
                frame[k] = rs[i];
                access[k] = ++freq;
                count++;
            }
            else
            {
                j = arrmin(access, nf);frame[j]
                = rs[i]; access[j] = ++freq;
                count++;
            }
            for(k=0; k<nf; k++) printf("%4d",
                frame[k]);
        }
    }
    printf("\n\nTotal no. of page faults : %d\n", count);
}

int arrmin(int a[], int n)
{
     int i, min = a[0];for(i=1;
      i<n; i++)if (min > a[i])
            min = a[i];
    for(i=0; i<n; i++)
        if (min == a[i])return
            i;
}
```

Output

```
Length of Reference string : 20Enter
reference string :
1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter no. of frames : 5

  Ref.  str        Page frames
     1               1   -1   -1      -1     -1
     2             1      2     -1     -1     -1
     3             1      2      3     -1     -1
     4             1      2      3      4     -1
     2
     1
     5             1      2      3      4      5
     6             1      2      6      4      5
     2
     1
     2
     3             1      2      6      3      5
     7             1      2      6      3      7
     6
     3
     2
     1
     2
     3
     6

Total no. of page faults : 8
```

Result

Thus page replacement was implemented using LRU algorithm.

Exp. No. 9c **Optimal Page Replacement**

Date:

Aim

To implement demand paging for a reference string using Optimal method.

Optimal

- Optimal page replacement has the lowest page fault rate of all algorithms.
- It does not suffer from Belady's anomaly.
- The page replaced is the one that will not be used for the longest period of time.
- It is difficult to implement, because it requires future knowledge of reference string.

Algorithm

1. Get number of pages.
2. Get number of frames
3. Get the reference string
4. Initialize the frame array
5. Display header
6. Create *access* array to store counter that indicates a measure of usage.
7. Initialize no. of page faults, say *count* to 0.
8. For each page in reference string in the given order, examine:
    - a. Check whether page exist in the *frame* array.
    - b. If page exist in memory then
        - i. Store incremented *freq* for that page position in *access* array.
    - c. If page does not exist in memory then
        - i. Check for any empty frames.
        - ii. If there is an empty frame,
            - Assign that frame to the page
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        - iii. If there is no free frame then
            - Replace page using optimal algorithm.
            - Store incremented *freq* for that page position in *access* array.
            - Increment *count*.
        - iv. Display pages in *frame* array.
9. Print *count*.
10. Stop

Program

```
/* Optimal Page Replacement - optimalpr.c */
#include <stdio.h>
int n, page[20], f, fr[20], i, pf=0, flag=0;

void display(int k, int flg)
{
    printf("\nPage  %d\t\t",k);for(i=0; i<f; i++)
        if(flg == 1)
                printf("%d\t",  fr[i]);
}

void optimal()
{
    int j, max, lp[10], index, m;for(j=0; j<f;
    j++)
    {
        fr[j] = page[j];flag =
        1;
        pf++;
        display(page[j],  flag);
    }

    for(j=f; j<n; j++)
    {
        flag =  1; for(i=0; i<f;
        i++)
            if(fr[i] == page[j])flag = 0;
        if(flag == 1)
        {
            for(i=0; i<f; i++)lp[i] =
                0;
            for(i=0; i<f; i++)
            {
                for(m=j+1; m<n; m++)
                {
                    if(fr[i] == page[m])
                    {
                           lp[i] = m - j;break;
                    }
                }
            }
            max  = lp[0]; index = 0;
            for(i=0;i<f;i++)
            {
                if(lp[i] == 0)
                {
                    index = i;
                    break;
                }
                else
                {
                    if(max < lp[i])
                    {
                         max = lp[i];
                         index = i;
                    }
                }
            }
            fr[index] = page[j];pf++;
            display(page[j],  flag);
        }
        else
                display(page[j],  flag);
    }

    printf("\n\nTotal No. of Page Faults : %d", pf);
}

main()
{
    printf("Enter No. of Pages: ");scanf("%d",
    &n);
    printf("\nEnter No. of Frames: ");scanf("%d",
    &f);
    printf("\nEnter Reference String : \n");for(i=0; i<n;
    i++)
        scanf("%d",  &page[i]);

    printf("\n\n\tOptimal Page Replacement \n");
    printf("\n=======================================\n");
    printf("Reference\t");for(i=0; i<f; i++)
        printf("F%d\t", i);
    printf("\n=======================================");

    for(i=0;i<f;i++)fr[i] = -
        1;

    optimal();
}
```

Output

```
Enter No. of Pages: 20Enter

No. of Frames: 3

Enter Reference String :
7  0  1  2  0  3  0  4  2  3  0  3  2  1  2  0  1  7  0  1


          Optimal Page Replacement

=====================================
Reference            F0        F1        F2
=====================================
Page   7             7         -1        -1
Page   0             7         0         -1
Page   1             7         0         1
Page   2             2         0         1
Page   0
Page   3             2         0         3
Page   0
Page   4             2         4         3
Page   2
Page   3
Page   0             2         0         3
Page   3
Page   2
Page   1             2         0         1
Page   2
Page   0
Page   1
Page   7             7         0         1
Page   0
Page   1

Total No. of Page Faults : 9
```

Result

Thus page replacement was implemented using Optimal algorithm.

Exp. No. 10a **Pipes**

Date:

Aim

To generate 25 fibonacci numbers and determine prime amongst them using pipe.

Interprocess Communication

- Inter-Process communication (IPC), is the mechanism whereby one process cancommunicate with another process, i.e exchange data.
- IPC in linux can be implemented using pipe, shared memory, message queue,semaphore, signal or sockets.

fork()

- The fork system call is used to create a new process called *child* process.
  - The return value is 0 for a child process.
  - The return value is negative if process creation is unsuccessful.
  - For the parent process, return value is positive
- The child process is an exact copy of the parent process.
- Both the child and parent continue to execute the instructions following fork call.
- The child can start execution before the parent or vice-versa.

wait()

- The wait system call causes the parent process to be blocked until a child terminates.
- When a process terminates, the kernel notifies the parent by sending a signal.
- Without wait, the parent may finish first leaving a *zombie* child

Pipe

- Pipes are unidirectional byte streams which connect the standard output from oneprocess into the standard input of another process.
- A pipe is created using the system call *pipe* that returns a pair of file descriptors.
- The descriptor pfd[0] is used for reading and pfd[1] is used for writing.
- Can be used only between parent and child processes.

Algorithm

```
1. Declare a array to store fibonacci numbers
2. Decalre a array pfd with two elements for pipe descriptors.
3. Create pipe on pfd using pipe function call.
     a. If return value is -1 then stop
4. Using fork system call, create a child process.
5. Let the child process generate 25 fibonacci numbers and store them in a array.
6. Write the array onto pipe using write system call.
7. Block the parent till child completes using wait system call.
8. Store fibonacci nos. written by child from the pipe in an array using read system
   call
```

9. Inspect each element of the fibonacci array and check whether they are prime
    a. If prime then print the fibonacci term.
10. Stop

Program

/* Fibonacci and Prime using pipe - fibprime.c */

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

main()
{
    pid_t pid;int
    pfd[2];
    int i,j,flg,f1,f2,f3;
    static unsigned int ar[25],br[25];

    if(pipe(pfd) == -1)
    {
        printf("Error in pipe");exit(-1);
    }


    pid=fork(); if
    (pid == 0)
    {
        printf("Child process generates Fibonacci series\n" );f1 = -1;
        f2 = 1;
        for(i = 0;i < 25; i++)
        {
            f3 = f1 + f2;
            printf("%d\t",f3);f1 = f2;
            f2 = f3; ar[i] =
            f3;
        }
        write(pfd[1],ar,25*sizeof(int));
    }
    else if (pid > 0)
    {
        wait(NULL);
        read(pfd[0], br, 25*sizeof(int));
        printf("\nParent prints Fibonacci that are Prime\n");
        for(i = 0;i < 25; i++)
        {
            flg = 0;
            if (br[i] <= 1)flg = 1;
            for(j=2; j<=br[i]/2; j++)
            {
                if (br[i]%j == 0)
                {
                    flg=1;
                    break;
                }
            }
            if (flg == 0) printf("%d\t",  br[i]);
        }
        printf("\n");
    }
    else
    {
        printf("Process  creation  failed");exit(-1);
    }
}
```

Output

```
$ gcc fibprime.c

$ ./a.out
Child process generates Fibonacci series
0        1        1        2        3        5        8        13
21       34       55       89       144      233      377      610
987      1597     2584     4181     6765     10946    17711    28657
46368
Parent   prints   Fibonacci   that   are Prime
2        3        5        13       89       233      1597     28657
```

Result

Thus fibonacci numbers that are prime is determined using IPC pipe.

Exp. No. 10b **Shared Memory**

Date:

Aim

To demonstrate communication between process using shared memory.

Shared memory

- Two or more processes share a single chunk of memory to communicate randomly.
- Semaphores are generally used to avoid race condition amongst processes.
- Fastest amongst all IPCs as it does not require any system call.
- It avoids copying data unnecessarily.

Algorithm

Server

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (some random value).
3. Create a shared memory segment using shmget with *key* & IPC_CREAT as parameter.
   a. If shared memory identifier *shmid* is -1, then stop.
4. Display *shmid*.
5. Attach server process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
6. Clear contents of the shared region using memset function.
7. Write a–z onto the shared memory.
8. Wait till client reads the shared memory contents
9. Detatch process from the shared memory using shmdt system call.
10. Remove shared memory from the system using shmctl with IPC_RMID argument
11. Stop

Client

1. Initialize size of shared memory *shmsize* to 27.
2. Initialize *key* to 2013 (same value as in server).
3. Obtain access to the same shared memory segment using same *key*.
   a. If obtained then display the *shmid* else print "Server not started"
4. Attach client process to the shared memory using shmmat with *shmid* as parameter.
   a. If pointer to the shared memory is not obtained, then stop.
5. Read contents of shared memory and print it.

6. After reading, modify the first character of shared memory to '*'
7. Stop

Program

Server

/* Shared memory server - shms.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/un.h>
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define shmsize 27
main()
{
    char c; int
    shmid;
    key_t key =   2013;char
    *shm, *s;

    if ((shmid = shmget(key, shmsize, IPC_CREAT|0666)) < 0)
    {
        perror("shmget");exit(1);
    }
    printf("Shared memory id : %d\n", shmid);

    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");exit(1);
    }

    memset(shm, 0, shmsize);s =
    shm;
    printf("Writing (a-z) onto shared memory\n");for (c = 'a'; c <=
    'z'; c++)
        *s++ = c;
    *s = '\0';

    while (*shm != '*');
    printf("Client  finished  reading\n");

    if(shmdt(shm) != 0)
        fprintf(stderr, "Could not close memory segment.\n");

    shmctl(shmid,  IPC_RMID,  0);
}
```

Client

```
/* Shared memory client - shmc.c */

#include <stdio.h>
#include <stdlib.h>
 #include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define shmsize 27
main()
{
     int shmid;
     key_t key = 2013;char
     *shm, *s;

     if ((shmid = shmget(key, shmsize, 0666)) < 0)
     {
          printf("Server  not  started\n");exit(1);
     }
     else
          printf("Accessing shared memory id : %d\n",shmid);

     if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
     {
          perror("shmat");exit(1);
     }

     printf("Shared  memory  contents:\n");for (s = shm; *s
     != '\0'; s++)
          putchar(*s);
     putchar('\n');

     *shm = '*';
}
```

Output

Server

```
$ gcc shms.c -o shms

$ ./shms
Shared memory id : 196611 Writing (a-z)
onto shared memoryClient finished reading
```

Client

```
$ gcc shmc.c -o shmc

$ ./shmc
Accessing shared memory id : 196611Shared
memory contents:
abcdefghijklmnopqrstuvwxyz
```

Result

Thus contents written onto shared memory by the server process is read by the clientprocess.

Exp. No. 10c **Message Queues**

Date:

Aim

To exchange message between server and client using message queue.

Message Queue

- A message queue is a linked list of messages stored within the kernel
- A message queue is identified by a unique identifier
- Every message has a positive long integer type field, a non-negative length, and theactual data bytes.
- The messages need not be fetched on FCFS basis. It could be based on type field.

Algorithm

Server

```
1. Decalre a structure mesgq with type and text fields.
2. Initialize key to 2013 (some random value).
3. Create a message queue using msgget with key & IPC_CREAT as parameter.
    a. If message queue cannot be created then stop.
4. Initialize the message type member of mesgq to 1.
5. Do the following until user types Ctrl+D
    a. Get message from the user and store it in text member.
    b. Delete the newline character in text member.
    c. Place message on the queue using msgsend for the client to read.
    d. Retrieve the response message from the client using msgrcv function
    e. Display the text contents.
6. Remove message queue from the system using msgctl with IPC_RMID as parameter.
7. Stop
```

Client

```
1. Decalre a structure mesgq with type and text fields.
2. Initialize key to 2013 (same value as in server).
3. Open the message queue using msgget with key as parameter.
    a. If message queue cannot be opened then stop.
4. Do while the message queue exists
    a. Retrieve the response message from the server using msgrcv function
    b. Display the text contents.
    c. Get message from the user and store it in text member.
    d. Delete the newline character in text member.
    e. Place message on the queue using msgsend for the server to read.
5. Print "Server Disconnected".
6. Stop
```

Program

Server

```
/* Server chat process - srvmsg.c */

#include <stdio.h> #include
<stdlib.h> #include
<string.h> #include
<sys/types.h>#include
<sys/ipc.h> #include
<sys/msg.h>

struct mesgq
{
    long type;
    char text[200];
} mq;

main()
{
    int msqid, len; key_t
    key = 2013;

    if((msqid = msgget(key, 0644|IPC_CREAT)) == -1)
    {
        perror("msgget");exit(1);
    }


    printf("Enter text, ^D to quit:\n");mq.type = 1;

    while(fgets(mq.text, sizeof(mq.text), stdin) != NULL)
    {
        len = strlen(mq.text);
        if (mq.text[len-1] == '\n')
            mq.text[len-1] = '\0';
        msgsnd(msqid, &mq, len+1, 0);


        msgrcv(msqid, &mq, sizeof(mq.text), 0, 0);printf("From
        Client: \"%s\"\n", mq.text);
    }
    msgctl(msqid,  IPC_RMID,  NULL);
}
```

Client

```
/* Client chat process - climsg.c */

#include <stdio.h> #include
<stdlib.h> #include
<string.h> #include
<sys/types.h>#include
<sys/ipc.h> #include
<sys/msg.h>

struct mesgq
{
    long type;
    char text[200];
} mq;

main()
{
    int msqid, len; key_t
    key = 2013;

    if ((msqid = msgget(key, 0644)) == -1)
    {
        printf("Server not active\n");exit(1);
    }


    printf("Client ready :\n");
    while (msgrcv(msqid, &mq, sizeof(mq.text), 0, 0) != -1)
    {
        printf("From Server: \"%s\"\n", mq.text);

        fgets(mq.text, sizeof(mq.text), stdin);len =
        strlen(mq.text);
        if (mq.text[len-1] == '\n')
            mq.text[len-1] = '\0';
        msgsnd(msqid, &mq, len+1, 0);
    }
    printf("Server   Disconnected\n");
}
```

Output

Server

```
$ gcc srvmsg.c -o srvmsg

$ ./srvmsg
Enter text, ^D to quit:hi
From Client: "hello"Where
r u?
From Client: "I'm where i am"bye
From Client: "ok"
^D
```

Client

```
$ gcc climsg.c -o climsg

$ ./climsg Client
ready:
From Server: "hi"hello
From Server: "Where r u?"I'm
where i am
From Server: "bye"ok
Server  Disconnected
```

Result

Thus chat session between client and server was done using message queue.

Exp. No. 11a Paging

Date:

Aim

To implement paging technique for memory management.

Paging

- Paging permits physical address space of a process to be noncontiguous.
- It avoids external fragmentation and the need for compaction.
- Physical memory is broken into frames.
- Logical memory is broken into pages , where page size = frame size
- Address consist of two parts: page number and page offset
- Page number is used as an index into page table to obtain base address
- Base address is added with offset to obtain physical memory address

Algorithm

1. Read physical memory size
2. Read page size
3. Read number of processes
4. Read page table entry for each process
5. Read page number and offset for a procese
6. Compute base address from page table
7. Add offset to base address
8. Display the physical memory address
9. Stop

Program

```
#include  <stdio.h>

main()
{
        int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;int s[10], fno[10][20];

        printf("Enter Physical memory size : ");
        scanf("%d", &ms);

        printf("\nEnter Page size : ");
        scanf("%d", &ps);
        nop = ms / ps;
        printf("\nNo. of Frames available are : %d \n",nop);

        printf("\nEnter no. of processes : ");
        scanf("%d",&np);

        rempages = nop;
        for(i=1; i<=np; i++)
        {
                printf("\nEnter no. of pages for process P%d : ",i);
                scanf("%d", &s[i]);

                if(s[i] > rempages)
                {
                        printf("\nMemory  is  Full");break;
                }
                rempages = rempages - s[i];

                printf("Enter Page table for process P%d : ", i);
                for(j=1; j<=s[i]; j++)
                        scanf("%d",  &fno[i][j]);
        }

        printf("\nEnter Process No. Page No. and Offset : ");scanf("%d%d%d",
        &x, &y, &offset);

        if(x>np || y>=s[i] || offset>=ps)
                printf("\nInvalid Process or Page No. or offset");
        else
        {
                pa = fno[x][y]* ps + offset; printf("Physical
                Address is : %d",pa);
        }
}
```

Output

```
Enter Physical memory size : 4096Enter

Page size : 512

No. of Frames available are : 8

Enter    no. of processes : 3

Enter    no. of pages for process P1              :  3
Enter    Page table for process P1 :              1  3  5

Enter    no. of pages for process P2              :  3
Enter    Page table for process P2 :              2  4  6

Enter    no. of pages for process P3              :  2
Enter    Page table for process P3 :              7  8

Enter Process No. Page No. and Offset : 2 1 120Physical
Address is : 2168
```

Result

Thus the program has been successfully executed.

Exp. No. 11b **First Fit Allocation**

Date:

Aim

To allocate memory requirements for processes using first fit allocation.

First fit

- The first-fit, best-fit, or worst-fit strategy is used to select a free hole from the set ofavailable holes.
- Allocate the first hole that is big enough.
- Searching starts from the beginning of set of holes.

Algorithm

```
1. Declare structures hole and process to hold information about set of
   holes andprocesses respectively.
2. Get number of holes, say nh.
3. Get the size of each hole
4. Get number of processes, say np.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
      a. If hole size > process size then
            i. Mark process as allocated to that hole.
           ii. Decrement hole size by process size.
      b. Otherwise check the next from the set of hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop
```

Program

/* First fit allocation - ffit.c */

```
#include <stdio.h>

struct  process
{
    int size; int
    flag; int
    holeid;
} p[10];
struct hole
{
    int size; int
    actual;
} h[10];

main()
{
    int i, np, nh, j;

    printf("Enter the number of Holes : ");scanf("%d",
    &nh);
    for(i=0; i<nh; i++)
    {
        printf("Enter size for hole H%d : ",i);scanf("%d",
        &h[i].size);
        h[i].actual =   h[i].size;
    }

    printf("\nEnter number of process : " );
    scanf("%d",&np);
    for(i=0;i<np;i++)
    {
        printf("enter the size of process P%d : ",i);scanf("%d",
        &p[i].size);
        p[i].flag = 0;
    }
    for(i=0; i<np; i++)
    {
        for(j=0; j<nh; j++)
        {
            if(p[i].flag != 1)
            {
                if(p[i].size  <=  h[j].size)
                {
                    p[i].flag = 1; p[i].holeid = j;
                    h[j].size -= p[i].size;
                }
            }
        }
    }

    printf("\n\tFirst fit\n");
    printf("\nProcess\tPSize\tHole");for(i=0;
    i<np; i++)
    {
        if(p[i].flag != 1)
            printf("\nP%d\t%d\tNot  allocated",  i,  p[i].size);else
            printf("\nP%d\t%d\tH%d",  i,  p[i].size,  p[i].holeid);
    }

    printf("\n\nHole\tActual\tAvailable");for(i=0; i<nh
    ;i++)
        printf("\nH%d\t%d\t%d",  i,  h[i].actual,  h[i].size);printf("\n");
}
```

Output

```
Enter     the number of Holes : 5size for
Enter     hole H0 : 100size for hole H1
Enter     : 500size for hole H2 : 200size
Enter     for hole H3 : 300size for hole
Enter     H4 : 600
Enter

Enter     number of process : 4
enter     the size of process P0 :          212
enter     the size of process P1 :          417
enter     the size of process P2 :          112
enter     the size of process P3 :          426

             First fit

Process      PSize      Hole
P0           212        H1
P1           417        H4
P2           112        H1
P3           426        Not allocated

Hole         Actual     Available
H0           100        100
H1           500        176
H2           200        200
H3           300        300
H4           600        183
```

Result

Thus processes were allocated memory using first fit method.

Exp. No. 11c **Best Fit Allocation**

Date:

Aim

To allocate memory requirements for processes using best fit allocation.

Best fit

- Allocate the smallest hole that is big enough.
- The list of free holes is kept sorted according to size in ascending order.
- This strategy produces smallest leftover holes

Worst fit

- Allocate the largest hole.
- The list of free holes is kept sorted according to size in descending order.
- This strategy produces the largest leftover hole.

Algorithm

1. Declare structures *hole* and *process* to hold information about set of holes andprocesses respectively.
2. 8Get number of holes, say *nh*.
3. Get the size of each hole
4. Get number of processes, say *np*.
5. Get the memory requirements for each process.
6. Allocate processes to holes, by examining each hole as follows:
    a. Sort the holes according to their sizes in ascending order
    b. If hole size > process size then
        i. Mark process as allocated to that hole.
        ii. Decrement hole size by process size.
    c. Otherwise check the next from the set of sorted hole
7. Print the list of process and their allocated holes or unallocated status.
8. Print the list of holes, their actual and current availability.
9. Stop

Program

```
/* Best fit allocation - bfit.c */

#include <stdio.h>

struct  process
{
    int size; int
    flag; int
    holeid;
} p[10];
struct hole
{
    int hid; int
    size; int actual;
} h[10];

main()
{
    int i, np, nh, j;
    void bsort(struct hole[], int);

    printf("Enter the number of Holes : ");scanf("%d",
    &nh);
    for(i=0; i<nh; i++)
    {
        printf("Enter size for hole H%d : ",i);scanf("%d",
        &h[i].size);
        h[i].actual =  h[i].size;h[i].hid = i;
    }

    printf("\nEnter number of process : " );
    scanf("%d",&np);
    for(i=0;i<np;i++)
    {
        printf("enter the size of process P%d : ",i);scanf("%d",
        &p[i].size);
        p[i].flag = 0;
    }

    for(i=0; i<np; i++)
    {
        bsort(h, nh);
        for(j=0; j<nh; j++)
        {
            if(p[i].flag != 1)
            {
                if(p[i].size  <=  h[j].size)
                {
                    p[i].flag = 1; p[i].holeid =
                    h[j].hid;
                    h[j].size  -=  p[i].size;
                }
            }
        }
    }

    printf("\n\tBest fit\n");
    printf("\nProcess\tPSize\tHole");for(i=0;
    i<np; i++)
    {
        if(p[i].flag != 1)
            printf("\nP%d\t%d\tNot  allocated",  i,  p[i].size);else
            printf("\nP%d\t%d\tH%d",  i,  p[i].size,  p[i].holeid);
    }

    printf("\n\nHole\tActual\tAvailable");for(i=0; i<nh
    ;i++)
        printf("\nH%d\t%d\t%d",  h[i].hid,  h[i].actual,h[i].size);
    printf("\n");
}

void bsort(struct hole bh[], int n)
{
    struct hole temp;int i,j;
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if(bh[i].size > bh[j].size)
            {
                temp = bh[i];bh[i]
                = bh[j];bh[j] =
                temp;
            }
        }
    }
}
```

Output

```
Enter     the number of Holes : 5size for
Enter     hole H0 : 100size for hole H1
Enter     : 500size for hole H2 : 200size
Enter     for hole H3 : 300size for hole
Enter     H4 : 600
Enter

Enter     number of process : 4
enter     the size of process P0 :            212
enter     the size of process P1 :            417
enter     the size of process P2 :            112
enter     the size of process P3 :            426


            Best fit

Process     PSize       Hole
P0          212         H3
P1          417         H1
P2          112         H2
P3          426         H4

Hole        Actual      Available
H1          500         83
H3          300         88
H2          200         88
H0          100         100
H4          600         174
```

Result

Thus processes were allocated memory using best fit method.

Exp. No. 12 **Multi-Threading**

Date:

Aim

To understand multithreading concepts.

Multi-Threading

- An application task can be split into many "threads" that all execute concurrently.
- Each thread acts as an individual program, but work in shared memory space.
- Communication between threads is simple.
- Switching between threads is cheaper than switching between processes.
- Multithreaded applications often require synchronization objects.
- For POSIX systems, header file pthread.hmust be included
- The function pthread_create is used to create a thread.
- A thread stop and wait for another thread to finish using pthread_join

Algorithm

1. Create thread using pthread_create function
2. Let the threads consume time using usleep function
3. Wait for child threads to terminate first using pthread_join function
4. Stop

Program

/* Multi-threading demo - multithread.c */

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

pthread_t tid[2];int
counter;

void* doThings(void *arg)
{
        unsigned long i = 0;counter
        += 1;
        printf("\n Job %d started\n", counter);

        for(i=0;  i<(0xFFFFFFFF);i++);
        printf("\n Job %d finished\n", counter);return NULL;
}

main()
{
        int i = 0;int
        err;

        while(i < 2)
        {
              err = pthread_create(&(tid[i]), NULL, &doThings, NULL);if (err != 0)
                     printf("\nCan't create thread : %s", strerror(err));i++;
        }

        pthread_join(tid[0],  NULL);
        pthread_join(tid[1],  NULL);
}
```

Output

```
$ gcc multithread.c –lpthread

$ ./a.out

Job   1   started

Job   2   started

Job   2   finished

Job   2   finished
```

Result

Thus multiple threads were created and thread functions were demonstrated.