

Operating System

Lab Manual

NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SECTION: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DEGREE: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**IQRA UNIVERSITY,**

**Operating System**

***(LIST OF LABS)***

***Lab #1\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Introduction to UNIX/LINUX Shell

***Lab # 2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

UNIX/LINUX Shell programming

* Variable
* Basic Variable

***Lab # 3\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

UNIX/LINUX Shell programming

* Using Array
* Basic Operator

***Lab # 4\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

UNIX/LINUX Shell programming

* Decision Making
* Shell Loop
* Loop Control
* Shell Function

***Lab # 5\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

UNIX/LINUX Shell programming

* Shell Substitution
* Quoting Mechanisms
* I/O Redirection
* Man\_page Help

***Lab # 6\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Linux File Management.

***Lab # 7\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Linux Directory Management

***Lab # 8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Linux File Permission / Access Mode

***Lab # 9\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

CPU Scheduling Algorithms

***Lab # 10\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

File Allocation Strategies

***Lab # 11\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Memory Management Techniques

***Lab # 12\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Memory Management Techniques using Paging.

***Lab # 13\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Deadlock Management Techniques

***Lab # 14\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Page Replacement Algorithms

***Lab # 15\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Process Synchronization



Operating System

**LAB-1**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Introduction to UNIX/LINUX Shell

**Objective: To Understand UNIX/ LINUX Shell**

## **Introduction:**

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

## **Shell Prompt**

The prompt, $, which is called command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

The shell reads your input after you press Enter. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of **date** command which displays current date and time:

$date

Thu Jun 25 08:30:19 MST 2009

You can customize your command prompt using environment variable PS1 explained in Environment tutorial.

## **Shell Types**

In UNIX there are two major types of shells:

* The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
* The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows −

* Bourne shell ( sh)
* Korn shell ( ksh)
* Bourne Again shell ( bash)
* POSIX shell ( sh)

The different C-type shells follow −

* C shell ( csh)
* TENEX/TOPS C shell ( tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.

In this tutorial, we are going to cover most of the Shell concepts based on Borne Shell.

## **Shell Scripts**

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign, #, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

## **Example Script**

Assume we create a test.sh script. Note all the scripts would have **.sh**extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −

#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.*

To create a script containing these commands, you put the shebang line first and then add the commands −

#!/bin/bash

pwd

ls

## **Shell Comments**

You can put your comments in your script as follows −

#!/bin/bash

# Script follows here:

pwd

ls

Now you save the above content and make this script executable as follows −

$chmod +x test.sh

Now you have your shell script ready to be executed as follows −

$./test.sh

This would produce following result −

/home/amrood

index.htm unix-basic\_utilities.htm unix-directories.htm

test.sh unix-communication.htm unix-environment.htm

## **Extended Shell Scripts**

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is sample run of the script −

$./test.sh

What is your name?

Iqra Uni Student

Hello, Iqra Uni Student

$

**Task:**

Write Linux bash Shell Script, which will ask and display all information required to Student admission in Engineering and Sciences University.

**Bash Script:**

#!/bin/sh

#Students Details For Admission:

echo "Enter your Name:"

read NAME

echo "Enter Your Father Name:"

read FNAME

echo "Enter Your Age:"

read AGE

echo "Enter Your Email Address"

read EMAIL

echo "Enter Your Current Address"

read CADDRESS

echo "Enter Your Gender:"

read GENDER

echo "Enter Your Mobile Number:"

read MOBILE

echo "Enter Your Previous Grade:"

read GRADE

echo "Enter Your Intresting Field:"

read FIELD

echo "Enter Your Hobby:"

read HOBBY

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo "=================================== IQRA UNIVERSITY ADDMISSION PORTAL============================================================"

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo "Name: $NAME"

echo "Father Name: $FNAME"

echo "Age: $AGE"

echo "Email: $EMAIL"

echo "Gender: $GENDER"

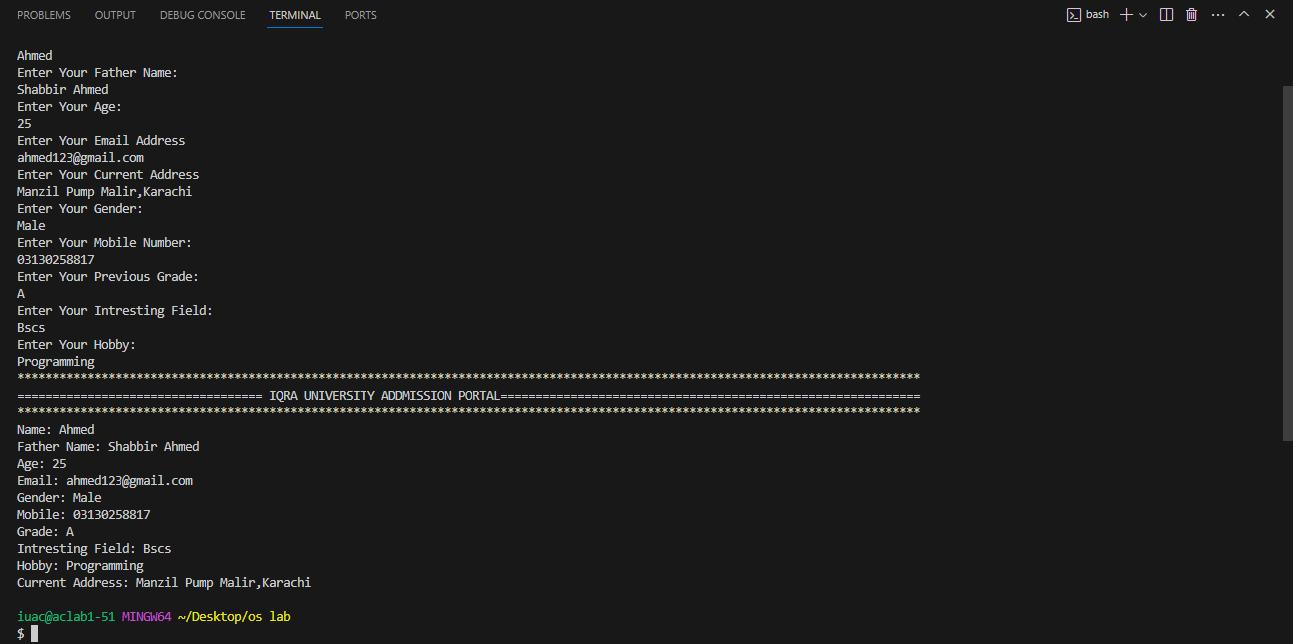
echo "Mobile: $MOBILE"

echo "Grade: $GRADE"

echo "Intresting Field: $FIELD"

echo "Hobby: $HOBBY"

echo "Current Address: $CADDRESS"

**Output:**

.



Operating System

**LAB-2**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Variable, Special Variable.

**Objective: Understand and implement Variable, Special Variable in Linux Shell.**

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

## **Variable Names**

The name of a variable can contain only letters ( a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

By convention, Unix Shell variables would have their names in UPPERCASE.

The following examples are vUniversityd variable names −

\_UNIVERSITY

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invUniversityd variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as !,\*, or - is that these characters have a special meaning for the shell.

## **Defining Variables**

Variables are defined as follows −

variable\_name=variable\_value

For example:

NAME="IU\_stuudent"

Above example defines the variable NAME and assigns it the value "IU\_Student". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The shell enables you to store any value you want in a variable. For example −

VAR1="IU\_Student"

VAR2=100

## **Accessing Values**

To access the value stored in a variable, prefix its name with the dollar sign ( $) −

For example, following script would access the value of defined variable NAME and would print it on STDOUT −

#!/bin/sh

NAME="IU\_Student"

echo $NAME

This would produce following value −

IU\_Student

## **Read-only Variables**

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME −

#!/bin/sh

NAME="IU\_Student"

readonly NAME

NAME="New\_student"

This would produce following result −

/bin/sh: NAME: This variable is read only.

## **Unsetting Variables**

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

Above command would unset the value of a defined variable. Here is a simple example −

#!/bin/sh

NAME="IU\_Student"

unset NAME

echo $NAME

Above example would not print anything. You cannot use the unset command to**unset** variables that are marked **readonly**.

## **Variable Types**

When a shell is running, three main types of variables are present −

* **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
* **Environment Variables** − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
* **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.
* The following table shows a number of special variables that you can use in your shell scripts

|  |  |
| --- | --- |
| **Variable** | **Description** |
| **$0** | The filename of the current script. |
| **$n** | These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| **$#** | The number of arguments supplied to a script. |
| **$\*** | All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| **$@** | All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| **$?** | The exit status of the last command executed. |
| **$$** | The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| **$!** | The process number of the last background command. |

## **Command-Line Arguments**

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./test.sh IU Student

File Name : ./test.sh

First Parameter : IU

Second Parameter : Student

Quoted Values: IU Student

Quoted Values: IU Student

Total Number of Parameters : 2

.

## **Special Parameters $\* and $@**

There are special parameters that allow accessing all of the command-line arguments at once. $\* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameter specifies all command-line arguments but the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $\* or $@ special parameters −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

There is one sample run for the above script −

$./test.sh IU Student 17 Years Old

IU

Student

17

Years

Old

**Exit Status**

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

$./test.sh IU Student

File Name : ./test.sh

First Parameter : IU

Second Parameter : Student

Quoted Values: IU Student

Quoted Values: IU Student

Total Number of Parameters : 2

$echo $?

0

$

**Task: Use following variables to write your CV in Linux bash script.**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Script** | **Output** |
| **$0** | echo "---------------------------------------------\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: Muhammad Ahmed"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$n** | echo "----------------------------------------------\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename: $0"  echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$#** | echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "total no. of argument:$#"  echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$\*** | echo "\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "Quoted arguments:$\*"  echo "total no. of argument:$#"  echo "\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$@** | echo "\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "Quoted arguments:$@"  echo "total no. of argument:$#"  echo "\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$?** | echo "  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename: $0"  echo "exit status: $?"  echo "total no. of argument: $#"  echo "\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$$** | echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "Process ID:$$"  echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |
| **$!** | echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  echo "==============CV===================="  echo "Name: $1 $2"  echo "Phone: 03130258817"  echo "Email: ahmed@gmail.com"  echo "Gender: male"  echo "Degree: BSCS"  echo "filename:$0"  echo "Process number of background command:$!"  echo "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" |  |



Operating System

**LAB-3**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Array, Operator

**Objective: Understand and implement Array and Operator in Linux Shell.**

A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.

Shell supports a different type of variable called an array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

## **Defining Array Values**

The difference between an array variable and a scalar variable can be explained as follows.

Say that you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Iqra"

NAME02="University"

NAME03="Computer Lab"

NAME04="Engineering Lab"

NAME05="Telecom Lab"

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows −

array\_name[index]=value

Here *array\_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

NAME[0]="Iqra"

NAME[1]="University"

NAME[2]="Computer Lab"

NAME[3]="Engineering Lab"

If you are using **ksh** shell the here is the syntax of array initiUniversityzation −

set -A array\_name value1 value2 ... valuen

If you are using **bash** shell the here is the syntax of array initiUniversityzation −

array\_name=(value1 ... valuen)

## **Accessing Array Values**

After you have set any array variable, you access it as follows −

${array\_name[index]}

Here *array\_name* is the name of the array, and *index* is the index of the value to be accessed. Following is the simplest example −

#!/bin/sh

NAME[0]="Iqra"

NAME[1]="University"

NAME[2]="Computer Lab"

NAME[3]="Engineering Lab"

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

This would produce following result −

$./test.sh

First Index: Iqra

Second Index: University

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

Here array\_name is the name of the array you are interested in. Following is the simplest example −

#!/bin/sh

NAME[0]="Iqra"

NAME[1]="University"

NAME[2]="Computer Lab"

NAME[3]="Engineering Lab"

echo "First Method: ${NAME[\*]}"

echo "Second Method: ${NAME[@]}"

This would produce following result −

$./test.sh

First Method: Iqra University Computer Lab Engineering Lab

Second Method: Iqra University Computer Lab Engineering Lab

**Operator**

There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

There are following operators which we are going to discuss −

* Arithmetic Operators.
* Relational Operators.
* Boolean Operators.
* String Operators.
* File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers −

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

This would produce following result −

Total value : 4

There are following points to note down −

* There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
* Complete expression should be enclosed between **``**, called inverted commas.

## **Arithmetic Operators**

There are following arithmetic operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / | Division - Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = | Assignment - Assign right operand in left operand | a=$b would assign value of b into a |
| == | EquUniversityty - Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != | Not EquUniversityty - Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a == $b ] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

## **Relational Operators:**

Bourne Shell supports following relational operators which are specific to numeric values. These operators would not work for string values unless their value is numeric.

For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable a holds 10 and variable b holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| -eq | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is not true. |
| -ne | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a -ne $b ] is true. |
| -gt | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is not true. |
| -lt | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | [ $a -lt $b ] is true. |
| -ge | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | [ $a -ge $b ] is not true. |
| -le | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | [ $a -le $b ] is true. |

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a <= $b ] is correct where as [$a <= $b] is incorrect.

## **Boolean Operators**

There are following boolean operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ! | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| -o | This is logical OR. If one of the operands is true then condition would be true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| -a | This is logical AND. If both the operands are true then condition would be true otherwise it would be false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

## **String Operators**

There are following string operators supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the value of two operands are equal or not, if yes then condition becomes true. | [ $a = $b ] is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | [ $a != $b ] is true. |
| -z | Checks if the given string operand size is zero. If it is zero length then it returns true. | [ -z $a ] is not true. |
| -n | Checks if the given string operand size is non-zero. If it is non-zero length then it returns true. | [ -n $a ] is not false. |
| str | Check if str is not the empty string. If it is empty then it returns false. | [ $a ] is not false. |

**Task: Write bash scrip to test all Athematic and logical operators in Linux**

|  |  |  |
| --- | --- | --- |
| **Operator: Task** | **Script** | **Output** |
| **+ : add two number** | #!/bin/bash  a=20  b=3  add=`expr $a + $b`  echo ""  echo "--------------"  echo "Addition: $add" |  |
| **- : make decrement operator** | #!/bin/bash  a=20  b=3  sub=`expr $a - $b`  echo ""  echo "-----------------"  echo "subtraction: $sub" |  |
| **\* : Mutiply negative and positive number** | #!/bin/bash  a=20  b=-3  mul=`expr $a \\* $b`  echo ""  echo "--------"  echo "multiplication: $mul" |  |
| **/: divide any number by zero** | #!/bin/bash  a=20  b=0  div=`expr $a / $b`  echo ""  echo "--------"  echo "division: $div" |  |
| **Less than: Compare two value** | #!/bin/bash  x=21  y=23  if [ $x -lt $y ]  then  echo "$x is less than $y"  fi |  |
| **Greater than: Compare two value** | #!/bin/bash  x=25  y=23  if [ $x -gt $y ]  then  echo "$x is greater than $y"  fi |  |
| **Equity: Compare two value** | #!/bin/bash  x=25  y=25  if [ $x -eq $y ]  then  echo "$x is equal to $y"  fi |  |
| **Not equal: Compare two value** | #!/bin/bash  x=25  y=22  if [ $x -ne $y ]  then  echo "$x is not equal to  $y"  fi |  |
| **And: make two simple and gate** | #!/bin/bash  x=09  y=45  z=88  if [ $x -gt $y -a $x -gt $z ]  then  echo "$x is greater than $y and also $z"  elif [ $x -lt $y -a $x -lt $z ]  then  echo "$x is less than $y and less than $z"  else  echo "$x is greater than $y and less than $z"  fi |  |
| **Or: make two input or gate** | #!/bin/bash  x=94  y=17  z=101  if [ $x -gt $y -o $x -gt $z ]  then  echo "$x is greater than $y or greater than $z"  else  echo "$x is less than both values"  fi |  |
| **Not: make 1 input not** | #!/bin/bash  x=7  y=87  z=44  if [ ! $x -gt $y -o $x -gt $z ]  then  echo "$x is less than $y or less than $z"  else  echo "$x is less"  fi |  |



Operating System

**LAB-4**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Decision Making, Loop, Shell Function

**Objective: Understand and implement Decision making, Loop, and Shell function in Linux Shell.**

Linux Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements −

* The **if...else** statements
* The **case...esac** statement

## **The if...else statements:**

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement −

[**if...fi statement**](https://www.tutorialspoint.com/unix/if-fi-statement.htm)**:**

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

## **Syntax**

if [ expression ]

then

Statement(s) to be executed if expression is true

fi

Here Shell *expression* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error.

If **expression** is a shell command then it would be assumed true if it return 0 after its execution. If it is a boolean expression then it would be true if it returns true.

## **Example**

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

This will produce following result −

a is not equal to b

[**if...else...fi statement**](https://www.tutorialspoint.com/unix/if-else-statement.htm)**:**

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

## **Syntax**

if [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi

Here Shell *expression* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

## **Example**

If we take above example then it can be written in better way using *if...else*statement as follows −

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

This will produce following result −

a is not equal to b

[**if...elif...else...fi statement**](https://www.tutorialspoint.com/unix/if-elif-statement.htm)**:**

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

## **Syntax**

if [ expression 1 ]

then

Statement(s) to be executed if expression 1 is true

elif [ expression 2 ]

then

Statement(s) to be executed if expression 2 is true

elif [ expression 3 ]

then

Statement(s) to be executed if expression 3 is true

else

Statement(s) to be executed if no expression is true

fi

There is nothing special about this code. It is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if non of the condition is true then *else* block is executed.

## **Example**

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

elif [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -lt $b ]

then

echo "a is less than b"

else

echo "None of the condition met"

fi

This will produce following result −

a is less than b

**Loops**:

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

[**The while loop**](https://www.tutorialspoint.com/unix/while-loop.htm)

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

## **Syntax**

while command

do

Statement(s) to be executed if command is true

done

Here Shell *command* is evaluated. If the resulting value is *true*, given*statement(s)* are executed. If *command* is *false* then no statement would be not executed and program would jump to the next line after done statement.

## **Example**

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

[**The for loop**](https://www.tutorialspoint.com/unix/for-loop.htm)

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

## **Syntax**

for var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

## **Example**

Here is a simple example that uses for loop to span through the given list of numbers −

#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9

do

echo $var

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

Following is the example to display all the files starting with **.bash** and available in your home. I'm executing this script from my root −

#!/bin/sh

for FILE in $HOME/.bash\*

do

echo $FILE

done

This will produce following result −

/root/.bash\_history

/root/.bash\_logout

/root/.bash\_profile

/root/.bashrc

[**The until loop**](https://www.tutorialspoint.com/unix/until-loop.htm)

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

## **Syntax**

until command

do

Statement(s) to be executed until command is true

done

Here Shell *command* is evaluated. If the resulting value is *false*, given*statement(s)* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

## **Example**

Here is a simple example that uses the until loop to display the numbers zero to nine −

#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5 6 7 8 9

[**The select loop**](https://www.tutorialspoint.com/unix/select-loop.htm)

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

The *select* loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

## **Syntax**

select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

For every selection a set of commands would be executed with-in the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

## **Example**

Here is a simple example to let the user select a drink of choice −

#!/bin/ksh

select DRINK in tea cofee water juice appe all none

do

case $DRINK in

tea|cofee|water|all)

echo "Go to canteen"

;;

juice|appe)

echo "Available at home"

;;

none)

break

;;

\*) echo "ERROR: InvUniversityd selection"

;;

esac

done

The menu presented by the select loop looks like the following −

$./test.sh

1) tea

2) cofee

3) water

4) juice

5) appe

6) all

7) none

#? juice

Available at home

#? none

$

So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this tutorial you will learn following two statements used to control shell loops −

* The **break** statement
* The **continue** statement

## **The infinite Loop**

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

## **Example**

Here is a simple example that uses the while loop to display the numbers zero to nine −

#!/bin/sh

a=10

until [ $a -lt 10 ]

do

echo $a

a=`expr $a + 1`

done

This loop would continue forever because a is alway greater than or equal to 10 and it would never become less than 10.

## **The break statement**

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

## **Syntax**

The following **break** statement would be used to come out of a loop −

break

The break command can also be used to exit from a nested loop using this format −

break n

Here **n** specifies the nth enclosing loop to exit from.

## **Example**

Here is a simple example which shows that loop would terminate as soon as a becomes 5:

#!/bin/sh

a=0

while [ $a -lt 10 ]

do

echo $a

if [ $a -eq 5 ]

then

break

fi

a=`expr $a + 1`

done

This will produce following result −

0 1 2 3 4 5

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 –

#!/bin/sh

for var1 in 1 2 3

do

for var2 in 0 5

do

if [ $var1 -eq 2 -a $var2 -eq 0 ]

then

break 2

else

echo "$var1 $var2"

fi

done

done

This will produce following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from inner loop as well.

1 0

1 5

## **The continue statement**

The **continue** statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

## **Syntax**

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here n specifies the nth enclosing loop to continue from.

## **Example**

The following loop makes use of continue statement which returns from the continue statement and start processing next statement −

#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

Q=`expr $NUM % 2`

if [ $Q -eq 0 ]

then

echo "Number is an even number!!"

continue

fi

echo "Found odd number"

done

This will produce following result −

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

**Functions**

Functions enable you to break down the overall functionUniversityty of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

## **Creating Functions**

To declare a function, simply use the following syntax −

function\_name () {

list of commands

}

The name of your function is function\_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

## **Example**

Following is the simple example of using function −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World"

}

# Invoke your function

Hello

When you would execute above script it would produce following result −

$./test.sh

Hello World

$

## **Pass Parameters to a Function**

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters *Iqra* and *University* and then we capture and print these parameters in the function.

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

}

# Invoke your function

Hello Iqra University

This would produce following result −

$./test.sh

Hello World Iqra University

$

## **Returning Values from Functions**

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the**return** command whose syntax is as follows −

return code

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

## **Example**

Following function returns a value 1 −

#!/bin/sh

# Define your function here

Hello () {

echo "Hello World $1 $2"

return 10

}

# Invoke your function

Hello Iqra University

# Capture value returnd by last command

ret=$?

echo "Return value is $ret"

This would produce following result −

$./test.sh

Hello World Iqra University

Return value is 10

$

## **Nested Functions**

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.

Following simple example demonstrates a nesting of two functions −

#!/bin/sh

# Calling one function from another

number\_one () {

echo "This is the first function speaking..."

number\_two

}

number\_two () {

echo "This is now the second function speaking..."

}

# Calling function one.

number\_one

This would produce following result −

This is the first function speaking...

This is now the second function speaking...

**Task:**

1. Sort integer Array in ascending order.

**BashScript:**

#!/bin/bash

# Function to sort an integer array in ascending order

sort\_array() {

local arr=("$@")

# Use printf to format the array elements, then sort, and finally read the sorted elements into a new array

sorted\_arr=($(printf "%s\n" "${arr[@]}" | sort -n))

echo "Sorted array in ascending order: ${sorted\_arr[@]}"

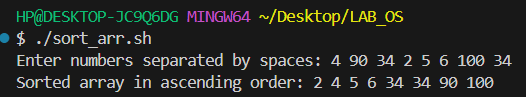
}

# Example usage

read -p "Enter numbers separated by spaces: " -a input\_array

sort\_array "${input\_array[@]}"

**Output:**



1. Find shortest value in integer array.

**BashScript:**

#!/bin/bash

# Function to find the smallest value in an integer array

find\_smallest() {

local arr=("$@")

local smallest=${arr[0]}

for num in "${arr[@]}"; do

if (( num < smallest )); then

smallest=$num

fi

done

echo "The smallest value in the array is: $smallest"

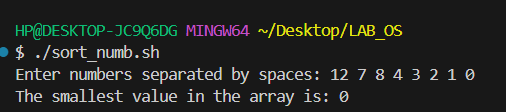
}

# Example usage

read -p "Enter numbers separated by spaces: " -a input\_array

find\_smallest "${input\_array[@]}"

**Output:**

****

1. Make shell function which can find out that input number is prime or not

**BashScript:**

#!/bin/bash

# Function to check if a number is prime

is\_prime() {

local num=$1

# Check if number is less than 2

if (( num < 2 )); then

echo "$num is not a prime number."

return 1

fi

# Check if number is 2 (the only even prime number)

if (( num == 2 )); then

echo "$num is a prime number."

return 0

fi

# Check if number is even

if (( num % 2 == 0 )); then

echo "$num is not a prime number."

return 1

fi

# Check for factors from 3 to the square root of the number

local sqrt\_num=$(echo "sqrt($num)" | bc)

for (( i=3; i<=sqrt\_num; i+=2 )); do

if (( num % i == 0 )); then

echo "$num is not a prime number."

return 1

fi

done

echo "$num is a prime number."

return 0

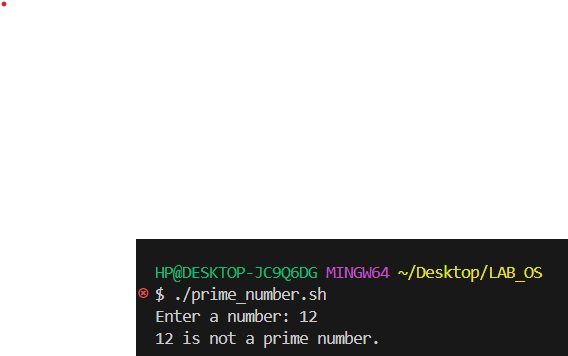
}

# Example usage

read -p "Enter a number: " input\_number

is\_prime "$input\_number"

**Output:**

****

1. Make shell function which can eliminate all odd numbers from integer array.

**BashScript:**

eliminate\_odd\_numbers() {

local even\_numbers=()

for num in "${@}";

do

if (( num % 2 == 0 )); then

even\_numbers+=("$num")

fi

done

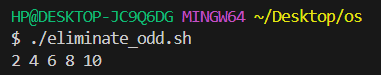
echo "${even\_numbers[@]}"

}

integer\_array=(1 2 3 4 5 6 7 8 9 10)

eliminate\_odd\_numbers "${integer\_array[@]}"

**Output:**



1. Make shell function which can find the last three prime number lesser than 100.

**BashScript:**

last\_three\_primes\_below\_100() {

primes=()

for ((num = 2; num < 100; num++));

do

is\_prime=1

for ((i = 2; i <= num / 2; i++));

do

if ((num % i == 0)); then

is\_prime=0

break

fi

done

if ((is\_prime == 1)); then

primes+=("$num")

fi

done

last\_three\_primes=("${primes[@]: -3}")

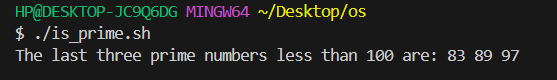
echo "The last three prime numbers less than 100 are: ${last\_three\_primes[@]}"

}

# Call the function

last\_three\_primes\_below\_100

**Output:**

****



Operating System

**LAB-5**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Shell Substitution, Quoting Mechanisms, I/O Redirection, Man\_page Help.

**Objective: Understand and implement shell substitution, Quoting mechanisms, I/O redirection and Man\_page Help in Linux.**

## **Shell Substitution:**

When performing command substitution make sure that you are using the backquote, not the single quote character.

## **Example**

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution −

#!/bin/sh

DATE=`date`

echo "Date is $DATE"

USERS=`who | wc -l`

echo "Logged in user are $USERS"

UP=`date ; uptime`

echo "Uptime is $UP"

This will produce following result −

Date is Thu Jul 2 03:59:57 MST 2009

Logged in user are 1

Uptime is Thu Jul 2 03:59:57 MST 2009

03:59:57 up 20 days, 14:03, 1 user, load avg: 0.13, 0.07, 0.15

## **Variable Substitution**

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions −

|  |  |
| --- | --- |
| **Form** | **Description** |
| **${var}** | Substitue the value of *var*. |
| **${var:-word}** | If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change. |
| **${var:=word}** | If *var* is null or unset, *var* is set to the value of **word**. |
| **${var:?message}** | If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly. |
| **${var:+word}** | If *var* is set, *word* is substituted for var. The value of *var*does not change. |

## **Example**

Following is the example to show various states of the above substitution −

#!/bin/sh

echo ${var:-"Variable is not set"}

echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}

echo "2 - Value of var is ${var}"

unset var

echo ${var:+"This is default value"}

echo "3 - Value of var is $var"

var="Prefix"

echo ${var:+"This is default value"}

echo "4 - Value of var is $var"

echo ${var:?"Print this message"}

echo "5 - Value of var is ${var}"

This would produce following result −

Variable is not set

1 - Value of var is

Variable is not set

2 - Value of var is Variable is not set

3 - Value of var is

This is default value

4 - Value of var is Prefix

Prefix

5 - Value of var is Prefix

## **Quoting mechanisms:**

## **The Metacharacters**

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example **?** matches with a single charater while listing files in a directory and an **\*** would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters) −

\* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

## **Example**

Following is the example which show how to print a **\*** or a **?** −

#!/bin/sh

echo Hello; Word

This would produce following result −

Hello

./test.sh: line 2: Word: command not found

shell returned 127

Now let us try using a quoted character −

#!/bin/sh

echo Hello\; Word

This would produce following result −

Hello; Word

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell −

#!/bin/sh

echo "I have \$1200"

This would produce following result −

I have $1200

There are following four forms of quotings −

|  |  |
| --- | --- |
| **Quoting** | **Description** |
| **Single quote** | All special characters between these quotes lose their special meaning. |
| **Double quote** | Most special characters between these quotes lose their special meaning with these exceptions:   * $ * ` * \$ * \' * \" * \\ |
| **Backslash** | Any character immediately following the backslash loses its special meaning. |
| **Back Quote** | Anything in between back quotes would be treated as a command and would be executed. |

## **The Single Quotes**

Consider an echo command that contains many special shell characters −

echo <-$1500.\*\*>; (update?) [y|n]

Putting a backslash in front of each special character is tedious and makes the line difficult to read −

echo \<-\$1500.\\*\\*\>\; \(update\?\) \[y\|n\]

There is an easy way to quote a large group of characters. Put a single quote ( ') at the beginning and at the end of the string −

echo '<-$1500.\*\*>; (update?) [y|n]'

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

echo 'It\'s Shell Programming'

## **The Double Quotes**

Try to execute the following shell script. This shell script makes use of single quote −

VAR=IQRA

echo '$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]'

This would produce following result −

$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows −

VAR=IQRA

echo "$VAR owes <-\$1500.\*\*>; [ as of (`date +%m/%d`) ]"

Now this would produce following result −

IQRA owes <-$1500.\*\*>; [ as of (07/02) ]

Double quotes take away the special meaning of all characters except the following −

* $ for parameter substitution.
* Backquotes for command substitution.
* \$ to enable literal dollar signs.
* \` to enable literal backquotes.
* \" to enable embedded double quotes.
* \\ to enable embedded backslashes.
* All other \ characters are literal (not special).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

echo 'It\'s Shell Programming'

## **The Back Quotes**

Putting any Shell command in between back quotes would execute the command

## **Syntax:**

Here is the simple syntax to put any Shell **command** in between back quotes −

## **Example**

var=`command`

Following would execute **date** command and produced result would be stored in DATA variable.

DATE=`date`

echo "Current Date: $DATE"

This would produce following result −

Current Date: Thu Jul 2 05:28:45 MST 2009

## **I/O Redirection**

Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is also your terminal by default.

## **Output Redirection**

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal −

Check following **who** command which would redirect complete output of the command in users file.

$ who > users

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check *users* file then it would have complete content −

$ cat users

oko tty01 Sep 12 07:30

ai tty15 Sep 12 13:32

ruth tty21 Sep 12 10:10

pat tty24 Sep 12 13:07

steve tty25 Sep 12 13:03

$

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example −

$ echo line 1 > users

$ cat users

line 1

$

You can use >> operator to append the output in an existing file as follows −

$ echo line 2 >> users

$ cat users

line 1

line 2

$

## **Input Redirection**

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows –

$ wc -l users

2 users

$

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file *users* −

$ wc -l < users

2

$

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

## **Here Document**

A *here document* is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is −

command << delimiter

document

delimiter

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count total number of line −

$wc -l << EOF

This is a simple lookup program

for good (and bad) restaurants

in Cape Town.

EOF

3

$

You can use *here document* to print multiple lines using your script as follows −

#!/bin/sh

cat << EOF

This is a simple lookup program

for good (and bad) restaurants

in Cape Town.

EOF

This would produce following result −

This is a simple lookup program

for good (and bad) restaurants

in Cape Town.

The following script runs a session with the vi text editor and save the input in the file test.txt.

#!/bin/sh

filename=test.txt

vi $filename <<EndOfCommands

i

This file was created automatically from

a shell script

^[

ZZ

EndOfCommands

If you run this script with vim acting as vi, then you will likely see output like the following −

$ sh test.sh

Vim: Warning: Input is not from a terminal

$

After running the script, you should see the following added to the file test.txt −

$ cat test.txt

This file was created automatically from

a shell script

$

## **Discard the output**

Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file /dev/null −

$ command > /dev/null

Here command is the name of the command you want to execute. The file /dev/null is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect STDERR to STDOUT −

$ command > /dev/null 2>&1

Here 2 represents STDERR and 1 represents STDOUT. You can display a message on to STDERR by redirecting STDOUT into STDERR as follows −

$ echo message 1>&2

## **Redirection Commands**

Following is the complete list of commands which you can use for redirection −

|  |  |
| --- | --- |
| **Command** | **Description** |
| pgm > file | Output of pgm is redirected to file |
| pgm < file | Program pgm reads its input from file. |
| pgm >> file | Output of pgm is appended to file. |
| n > file | Output from stream with descriptor n redirected to file. |
| n >> file | Output from stream with descriptor n appended to file. |
| n >& m | Merge output from stream n with stream m. |
| n <& m | Merge input from stream n with stream m. |
| << tag | Standard input comes from here through next tag at start of line. |
| | | Takes output from one program, or process, and sends it to another. |

## **Man\_page Help**

All the Unix commands come with a number of optional and mandatory options. It is very common to forget complete syntax of these commands.

Because no one can possibly remember every Unix command and all its options, there has been online help available since Unix's earliest days.

Unix's version of help files are called **man pages**. If you know any command name but you do not know how to use it, then Man Pages are here to help you at every step.

## **Syntax**

Here is the simple command to get the detail of any Unix command while working with the system −

$man command

## **Example**

Now you imagine any command for which you want to get help. Assuming you want to know about **pwd** then you simply need to use the following command −

$man pwd

The above command would open a help for you which would give you complete information about **pwd** command. Try it yourself at your command prompt to get more detail on

You can get complete detail on **man** command itself using the following command −

$man man

## **Man Page Sections**

Man pages are generally divided into sections, which generally vary by the man page author's preference. Here are some of the more common sections −

|  |  |
| --- | --- |
| **Section** | **Description** |
| NAME | Name of the command |
| SYNOPSIS | General usage parameters of the command. |
| DESCRIPTION | Generally describes of the command and what it does |
| OPTIONS | Describes all the arguments or options to the command |
| SEE ALSO | Lists other commands that are directly related to the command in the man page or closely resembling its functionUniversityty. |
| BUGS | Explains any known issues or bugs that exist with the command or its output |
| EXAMPLES | Common usage examples that give the reader an idea of how the command can be used. |
| AUTHORS | The author of the man page/command. |



Operating System

**LAB-6**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Objective: To understand UNIX/Linux file management by creating, reading, editing and writing in file.

All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with UNIX, one way or another you spend most of your time working with files. This tutorial would teach you how to create and remove files, copy and rename them, create links to them etc.

In UNIX there are three basic types of files −

* **Ordinary Files** − An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.
* **Directories** − Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders.
* **Special Files** − Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to Universityases or shortcuts and enable you to access a single file using different names.

## **Listing Files**

To list the files and directories stored in the current directory. Use the following command −

$ls

Here is the sample output of the above command −

$ls

bin hosts lib res.03

ch07 hw1 pub test\_results

ch07.bak hw2 res.01 users

docs hw3 res.02 work

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

$ls -l

total 1962188

drwxrwxr-x 2 amrood amrood 4096 Dec 25 09:59 uml

-rw-rw-r-- 1 amrood amrood 5341 Dec 25 08:38 uml.jpg

drwxr-xr-x 2 amrood amrood 4096 Feb 15 2006 univ

drwxr-xr-x 2 root root 4096 Dec 9 2007 urlspedia

-rw-r--r-- 1 root root 276480 Dec 9 2007 urlspedia.tar

drwxr-xr-x 8 root root 4096 Nov 25 2007 usr

drwxr-xr-x 2 200 300 4096 Nov 25 2007 webthumb-1.01

-rwxr-xr-x 1 root root 3192 Nov 25 2007 webthumb.php

-rw-rw-r-- 1 amrood amrood 20480 Nov 25 2007 webthumb.tar

-rw-rw-r-- 1 amrood amrood 5654 Aug 9 2007 yourfile.mid

-rw-rw-r-- 1 amrood amrood 166255 Aug 9 2007 yourfile.swf

drwxr-xr-x 11 amrood amrood 4096 May 29 2007 zlib-1.2.3

$

Here is the information about all the listed columns −

* First Column: represents file type and permission given on the file. Below is the description of all type of files.
* Second Column: represents the number of memory blocks taken by the file or directory.
* Third Column: represents owner of the file. This is the Unix user who created this file.
* Fourth Column: represents group of the owner. Every Unix user would have an associated group.
* Fifth Column: represents file size in bytes.
* Sixth Column: represents date and time when this file was created or modified last time.
* Seventh Column: represents file or directory name.

In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

|  |  |
| --- | --- |
| **Prefix** | **Description** |
| **-** | Regular file, such as an ASCII text file, binary executable, or hard link. |
| **b** | Block special file. Block input/output device file such as a physical hard drive. |
| **c** | Character special file. Raw input/output device file such as a physical hard drive |
| **d** | Directory file that contains a listing of other files and directories. |
| **l** | Symbolic link file. Links on any regular file. |
| **p** | Named pipe. A mechanism for interprocess communications |
| **s** | Socket used for interprocess communication. |

## **Meta Characters**

Meta characters have special meaning in Unix. For example **\*** and **?** are metacharacters. We use **\*** to match 0 or more characters, a question mark **?**matches with single character.

For Example −

$ls ch\*.doc

Displays all the files whose name start with ch and ends with .doc −

ch01-1.doc ch010.doc ch02.doc ch03-2.doc

ch04-1.doc ch040.doc ch05.doc ch06-2.doc

ch01-2.doc ch02-1.doc c

Here **\*** works as meta character which matches with any character. If you want to display all the files ending with just **.doc** then you can use following command −

$ls \*.doc

## **Hidden Files**

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files −

* **.profile** − the Bourne shell ( sh) initiUniversityzation script
* **.kshrc** − the Korn shell ( ksh) initiUniversityzation script
* **.cshrc** − the C shell ( csh) initiUniversityzation script
* **.rhosts** − the remote shell configuration file

To list invisible files, specify the -a option to ls −

$ ls -a

. .profile docs lib test\_results

.. .rhosts hosts pub users

.emacs bin hw1 res.01 work

.exrc ch07 hw2 res.02

.kshrc ch07.bak hw3 res.03

$

* Single dot **.** − This represents current directory.
* Double dot **..** − This represents parent directory.

## **Creating Files**

You can use **vi** editor to create ordinary files on any Unix system. You simply need to give following command −

$ vi filename

Above command would open a file with the given filename. You would need to press key **i** to come into edit mode. Once you are in edit mode you can start writing your content in the file as below −

This is unix file....I created it for the first time.....

I'm going to save this content in this file.

Once you are done, do the following steps −

* Press key **esc** to come out of edit mode.
* Press two keys **Shift + ZZ** together to come out of the file completely.

Now you would have a file created with **filename** in the current directory.

$ vi filename

$

## **Editing Files**

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows −

$ vi filename

Once file is opened, you can come in edit mode by pressing key **i** and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key **esc** and then you can use following keys to move inside a file −

* **l** key to move to the right side.
* **h** key to move to the left side.
* **k** key to move up side in the file.
* **j** key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use **i** key to come in edit mode. Edit the file, once you are done press **esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

## **Display Content of a File**

You can use **cat** command to see the content of a file. Following is the simple example to see the content of above created file −

$ cat filename

This is unix file....I created it for the first time.....

I'm going to save this content in this file.

$

You can display line numbers by using **-b** option along with **cat** command as follows

$ cat -b filename

1 This is unix file....I created it for the first time.....

2 I'm going to save this content in this file.

$

## **Counting Words in a File**

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is the simple example to see the information about above created file .

$ wc filename

2 19 103 filename

$

Here is the detail of all the four columns −

* First Column: represents total number of lines in the file.
* Second Column: represents total number of words in the file.
* Third Column: represents total number of bytes in the file. This is actual size of the file.
* Fourth Column: represents file name.

You can give multiple files at a time to get the information about those file. Here is simple syntax −

$ wc filename1 filename2 filename3

## **Copying Files:**

To make a copy of a file use the **cp** command. The basic syntax of the command is

$ cp source\_file destination\_file

Following is the example to create a copy of existing file **filename**.

$ cp filename copyfile

$

Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

## **Renaming Files**

To change the name of a file use the **mv** command. Its basic syntax is −

$ mv old\_file new\_file

Following is the example which would rename existing file **filename** to **newfile**:

$ mv filename newfile

$

The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory.

## **Deleting Files**

To delete an existing file use the **rm** command. Its basic syntax is −

$ rm filename

**Caution:** It may be dangerous to delete a file because it may contain useful information. So be careful while using this command. It is recommended to use**-i** option along with **rm** command.

Following is the example which would completely remove existing file **filename**:

$ rm filename

$

You can remove multiple files at a tile as follows −

$ rm filename1 filename2 filename3

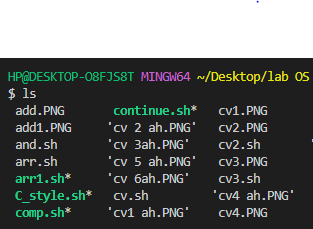
$

## **Standard Unix Streams**

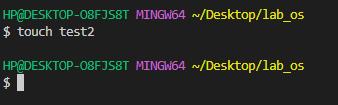
Under normal circumstances every Unix program has three streams (files) opened for it when it starts up −

* **stdin** − This is referred to as *standard input* and associated file descriptor is 0. This is also represented as STDIN. Unix program would read default input from STDIN.
* **stdout** − This is referred to as *standard output* and associated file descriptor is 1. This is also represented as STDOUT. Unix program would write default output at STDOUT
* **stderr** − This is referred to as *standard error* and associated file descriptor is 2. This is also represented as STDERR. Unix program would write all the error message at STDERR.

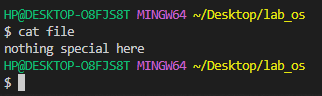
## **Listing Files**



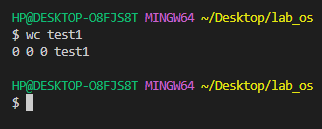
## **Creating Files**



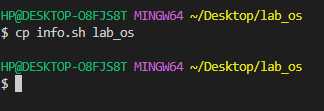
## **Display Content of a File**



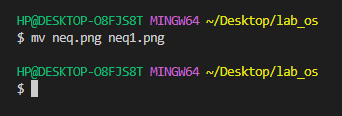
## **Counting Words in a File**



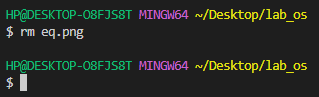
## **Copying Files:**



## **Renaming Files**



## **Deleting Files**





Operating System

**LAB-7**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Directory Management

Objective: To Understand UNIX / Linux Directory management by Creating, Listining and Removing, Changing and Renaming Directories.

A directory is a file whose sole job is to store file names and related information. All files, whether ordinary, special, or directory, are contained in directories.

UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character ( /), and all other directories are contained below it.

## **Home Directory**

The directory in which you find yourself when you first login is called your home directory.

You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command −

$cd ~

$

Here **~** indicates home directory. If you want to go in any other user's home directory then use the following command −

$cd ~username

$

To go in your last directory you can use following command −

$cd -

$

## **Absolute/Relative Pathnames**

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /.

These are some example of absolute filenames.

/etc/passwd

/users/sjones/chem/notes

/dev/rdsk/Os3

A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user amrood' home directory, some pathnames might look like this −

chem/notes

personal/res

To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory −

$pwd

/user0/home/amrood

$

## **Listing Directories**

To list the files in a directory you can use the following syntax −

$ls dirname

Following is the example to list all the files contained in /usr/local directory −

$ls /usr/local

X11 bin gimp jikes sbin

ace doc include lib share

atalk etc info man ami

## **Creating Directories**

Directories are created by the following command −

$mkdir dirname

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command −

$mkdir mydir

$

Creates the directory mydir in the current directory. Here is another example −

$mkdir /tmp/test-dir

$

This command creates the directory test-dir in the /tmp directory. The **mkdir**command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, mkdir creates each of the directories. For example −

$mkdir docs pub

$

Creates the directories docs and pub under the current directory.

## **Creating Parent Directories**

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, mkdir issues an error message as follows −

$mkdir /tmp/amrood/test

mkdir: Failed to make directory "/tmp/amrood/test";

No such file or directory

$

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example −

$mkdir -p /tmp/amrood/test

$

Above command creates all the required parent directories.

## **Removing Directories**

Directories can be deleted using the **rmdir** command as follows −

$rmdir dirname

$

**Note** − To remove a directory make sure it is empty which means there should not be any file or sub-directory inside this directory.

You can remove multiple directories at a time as follows −

$rmdir dirname1 dirname2 dirname3

$

Above command removes the directories dirname1, dirname2, and dirname2 if they are empty. The rmdir command produces no output if it is successful.

## **Changing Directories**

You can use the **cd** command to do more than change to a home directory: You can use it to change to any directory by specifying a vUniversityd absolute or relative path. The syntax is as follows −

$cd dirname

$

Here, dirname is the name of the directory that you want to change to. For example, the command −

$cd /usr/local/bin

$

Changes to the directory /usr/local/bin. From this directory you can cd to the directory /usr/home/amrood using the following relative path −

$cd ../../home/amrood

$

## **Renaming Directories**

The mv (move) command can also be used to rename a directory. The syntax is as follows −

$mv olddir newdir

$

You can rename a directory **mydir** to **yourdir** as follows −

$mv mydir yourdir

$

## **The directories . (dot) and .. (dot dot)**

The filename . (dot) represents the current working directory; and the filename .. (dot dot) represent the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories files and use the -a option to list all the files and the -l option provides the long listing, this is the result.

$ls -la

drwxrwxr-x 4 teacher class 2048 Jul 16 17.56 .

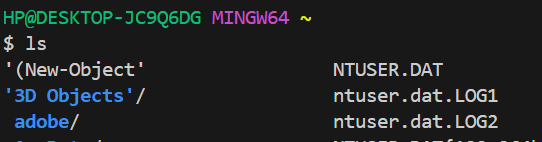
drwxr-xr-x 60 root 1536 Jul 13 14:18 ..

---------- 1 teacher class 4210 May 1 08:27 .profile

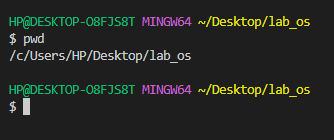
-rwxr-xr-x 1 teacher class 1948 May 12 13:42 memo

$

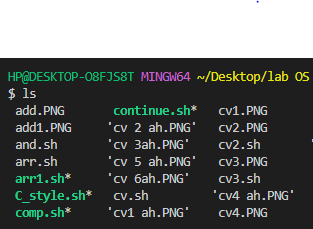
**Home Directories:**

****

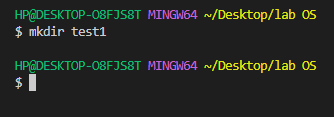
**Relative Pathnames:**



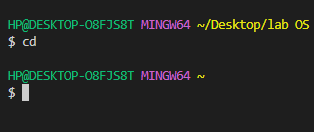
**Listing Directories:**



**Creating Directories:**



**Changing Directories:**





Operating System

**LAB-8**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Linux File Permission/ Access Modes

Objective: Changing permission, access and ownership of file in UNIX/Linux operating system.

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes −

* **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
* **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
* **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

## **The Permission Indicators**

While using **ls -l** command it displays various information related to file permission as follows −

$ls -l /home/amrood

-rwxr-xr-- 1 amrood users 1024 Nov 2 00:10 myfile

drwxr-xr--- 1 amrood users 1024 Nov 2 00:10 mydir

Here first column represents different access mode ie. permission associated with a file or directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

* The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that owner has read (r), write (w) and execute (x) permission.
* The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--**represents that group has read (r) and execute (x) permission but no write permission.
* The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

## **File Access Modes**

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and**execute** permissions, which are described below −

## **1. Read**

Grants the capability to read ie. view the contents of the file.

## **2. Write**

Grants the capability to modify, or remove the content of the file.

## **3. Execute**

User with execute permissions can run a file as a program.

## **Directory Access Modes**

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

## **1. Read**

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

## **2. Write**

Access means that the user can add or delete files to the contents of the directory.

## **3. Execute**

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.

A user must have execute access to the **bin** directory in order to execute ls or cd command.

## **Changing Permissions**

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

## **Using chmod in Symbolic Mode**

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

|  |  |
| --- | --- |
| **Chmod operator** | **Description** |
| **+** | Adds the designated permission(s) to a file or directory. |
| **-** | Removes the designated permission(s) from a file or directory. |
| **=** | Sets the designated permission(s). |

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows −

$ls -l testfile

-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes −

$chmod o+wx testfile

$ls -l testfile

-rwxrwxrwx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod u-x testfile

$ls -l testfile

-rw-rwxrwx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod g=rx testfile

$ls -l testfile

-rw-r-xrwx 1 amrood users 1024 Nov 2 00:10 testfile

Here's how you could combine these commands on a single line:

$chmod o+wx,u-x,g=rx testfile

$ls -l testfile

-rw-r-xrwx 1 amrood users 1024 Nov 2 00:10 testfile

## **Using chmod with Absolute Permissions**

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

|  |  |  |
| --- | --- | --- |
| **Number** | **Octal Permission Representation** | **Ref** |
| **0** | No permission | --- |
| **1** | Execute permission | --x |
| **2** | Write permission | -w- |
| **3** | Execute and write permission: 1 (execute) + 2 (write) = 3 | -wx |
| **4** | Read permission | r-- |
| **5** | Read and execute permission: 4 (read) + 1 (execute) = 5 | r-x |
| **6** | Read and write permission: 4 (read) + 2 (write) = 6 | rw- |
| **7** | All permissions: 4 (read) + 2 (write) + 1 (execute) = 7 | rwx |

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows −

$ls -l testfile

-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes –

$ chmod 755 testfile

$ls -l testfile

-rwxr-xr-x 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 743 testfile

$ls -l testfile

-rwxr---wx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 043 testfile

$ls -l testfile

----r---wx 1 amrood users 1024 Nov 2 00:10 testfile

## **Changing Owners and Groups**

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files −

* **chown** − The chown command stands for "change owner" and is used to change the owner of a file.
* **chgrp** − The chgrp command stands for "change group" and is used to change the group of a file.

## **Changing Ownership**

The chown command changes the ownership of a file. The basic syntax is as follows −

$ chown user filelist

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.

Following example −

$ chown amrood testfile

$

Changes the owner of the given file to the user **amrood**.

**NOTE:** The super user, root, has the unrestricted capability to change the ownership of a any file but normal users can change only the owner of files they own.

Changing Group Ownership

The chrgp command changes the group ownership of a file. The basic syntax is as follows −

$ chgrp group filelist

The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

Following example −

$ chgrp special testfile

$ Changes the group of the given file to special group.

SUID and SGID File Permission:

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the passwd command, your new password is stored in the file /etc/shadow.

As a regular user, you do not have read or write access to this file for security reasons, but when you change your password, you need to have write permission to this file. This means that the passwd program has to give you additional permissions so that you can write to the file /etc/shadow.

Additional permissions are given to programs via a mechanism known as the Set User ID ( SUID) and Set Group ID ( SGID) bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is true for SGID as well. Normally programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter "s" if the permission is available. The SUID "s" bit will be located in the permission bits where the owners execute permission would normally reside. For example, the command

$ ls -l /usr/bin/passwd

-r-sr-xr-x 1 root bin 19031 Feb 7 13:47 /usr/bin/passwd\*

$

Which shows that the SUID bit is set and that the command is owned by the root. A capital letter S in the execute position instead of a lowercase s indicates that the execute bit is not set.

If the sticky bit is enabled on the directory, files can only be removed if you are one of the following users −

* The owner of the sticky directory
* The owner of the file being removed
* The super user, root

To set the SUID and SGID bits for any directory try the following −

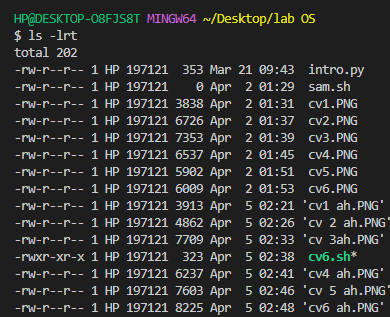
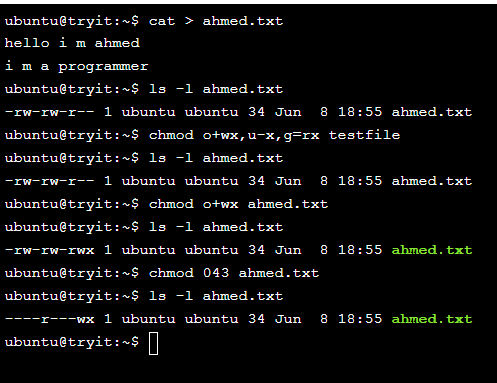
$ chmod ug+s dirname

$ ls -l

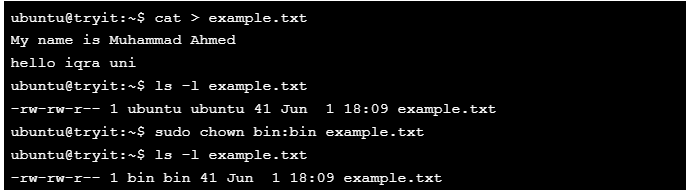
drwsr-sr-x 2 root root 4096 Jun 19 06:45 dirname

$

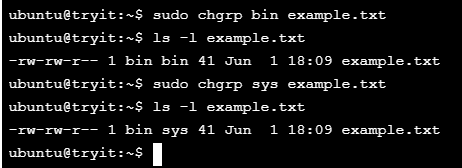
## **Changing Permissions**



**USE OWNERSHIP COMMAND:**



**USE GROUP COMMAND:**

****



Operating System

**LAB-9**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

CPU Scheduling Algorithms

**Objective:** Write a C program to simulate the following non-pre-emptive CPU scheduling algorithms to find turnaround time and waiting time.

* FCFS
* SJF
* Round Robin (pre-emptive)
* Priority

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario.

* All the processes in the system are divided into two categories – system processes and user processes.
* System processes are to be given higher priority than user processes.

Use FCFS scheduling for the processes in each queue.

**TASK 1:**

**DESCRIPTION**

Assume all the processes arrive at the same time.

**FCFS CPU SCHEDULING ALGORITHM**

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

**SJF CPU SCHEDULING ALGORITHM**

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

**ROUND ROBIN CPU SCHEDULING ALGORITHM**

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

**PRIORITY CPU SCHEDULING ALGORITHM**

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

**PROGRAM**

**FCFS CPU SCHEDULING ALGORITHM**

#include<stdio.h>

#include<conio.h>

main()

{

int bt[20], wt[20], tat[20], i, n; float wtavg, tatavg;

clrscr();

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

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

{

printf("\nEnter Burst Time for Process %d -- ", i); scanf("%d", &bt[i]);

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];tat[i] = tat[i-1] +bt[i]; wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

{

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();

}

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| ***INPUT*** |  |  |  |
| Enter the number of processes -- | | 3 |  |
| Enter Burst Time for Process 0 -- | | 24 |  |
| Enter Burst Time for Process 1 -- | | 3 |  |
| Enter Burst Time for Process 2 -- | | 3 |  |
| ***OUTPUT*** |  |  |  |
| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
| P0 | 24 | 0 | 24 |
| P1 | 3 | 24 | 27 |
| P2 | 3 | 27 | 30 |
| Average Waiting Time-- 17.000000 | | |  |
| Average Turnaround Time -- | | 27.000000 |  |

**FCFS Algorithm :**

#include<stdio.h>

#include<conio.h>

int main()

{

int bt[20], wt[20], tat[20], i, n; float wtavg, tatavg;

printf("\nEnter the number of processes -- ");

scanf("%d", &n);

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

{

printf("\nEnter Burst Time for Process %d -- ", i);

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

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];tat[i] = tat[i-1] +bt[i]; wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

{

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

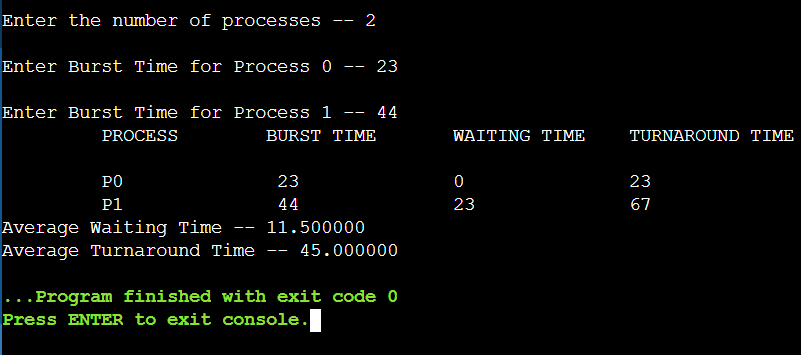
}

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();

}

**OUTPUT:**

****

**SJF CPU SCHEDULING ALGORITHM\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

#include<stdio.h>

#include<conio.h>

main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;

clrscr();

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

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

{

p[i]=i;

printf("Enter Burst Time for Process %d -- ", i); scanf("%d", &bt[i]);

}

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

for(k=i+1;k<n;k++)

if(bt[i]>bt[k])

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

getch();

}

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| ***INPUT*** |  |  |  |
| Enter the number of processes -- | | 4 |  |
| Enter Burst Time for Process 0 -- | | 6 |  |
| Enter Burst Time for Process 1 -- | | 8 |  |
| Enter Burst Time for Process 2 -- | | 7 |  |
| Enter Burst Time for Process 3 -- | | 3 |  |
| ***OUTPUT*** |  |  |  |
| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
| P3 | 3 | 0 | 3 |
| P0 | 6 | 3 | 9 |
| P2 | 7 | 9 | 16 |
| P1 | 8 | 16 | 24 |
| Average Waiting Time -- | | 7.000000 |  |
| Average Turnaround Time -- | | 13.000000 |  |

**SJF Algorithm :**

#include<stdio.h>

#include<conio.h>

int main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;

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

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

{

p[i]=i;

printf("Enter Burst Time for Process %d -- ", i);

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

}

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

for(k=i+1;k<n;k++)

if(bt[i]>bt[k])

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0];

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

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

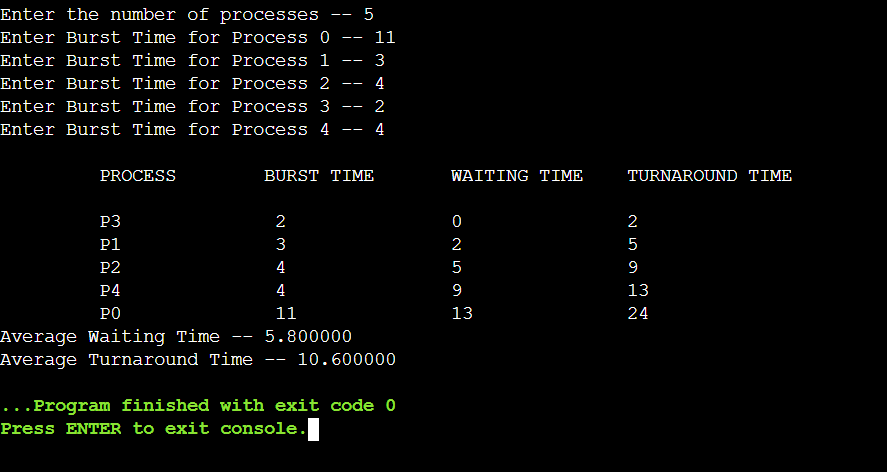
printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

getch();

}

**OUTPUT:**

****

**ROUND ROBIN CPU SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float awt=0,att=0,temp=0;

clrscr();

printf("Enter the no of processes -- "); scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d -- ", i+1); scanf("%d",&bu[i]);

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- "); scanf("%d",&t);

max=bu[0];

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

if(max<bu[i])

max=bu[i];

for(j=0;j<(max/t)+1;j++)

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

if(bu[i]!=0)

if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;temp=temp+t;

}

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

{

wa[i]=tat[i]-ct[i];att+=tat[i];

awt+=wa[i];}

printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

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

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

getch();

}

|  |  |  |  |
| --- | --- | --- | --- |
| ***INPUT*** |  |  |  |
| Enter the no of processes – 3 | |  |  |
| Enter Burst Time for process 1 – | | 24 |  |
| Enter Burst Time for process 2 -- | | 3 |  |
| Enter Burst Time for process 3 -- | | 3 |  |
| Enter the size of time slice – 3 | |  |  |
| ***OUTPUT*** |  |  |  |
| The Average Turnaround time is – 15.666667 | | |  |
| The Average Waiting time is -- | | 5.666667 |  |
| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
| 1 | 24 | 6 | 30 |
| 2 | 3 | 4 | 7 |
| 3 | 3 | 7 | 10 |

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

**ROUND ROBIN Algorithm :**

#include<stdio.h>

int main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float awt=0,att=0,temp=0;

printf("Enter the no of processes -- "); scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d -- ", i+1);

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

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- "); scanf("%d",&t);

max=bu[0];

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

if(max<bu[i])

max=bu[i];

for(j=0;j<(max/t)+1;j++)

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

if(bu[i]!=0)

if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;temp=temp+t;

}

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

{

wa[i]=tat[i]-ct[i];att+=tat[i];

awt+=wa[i];}

printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

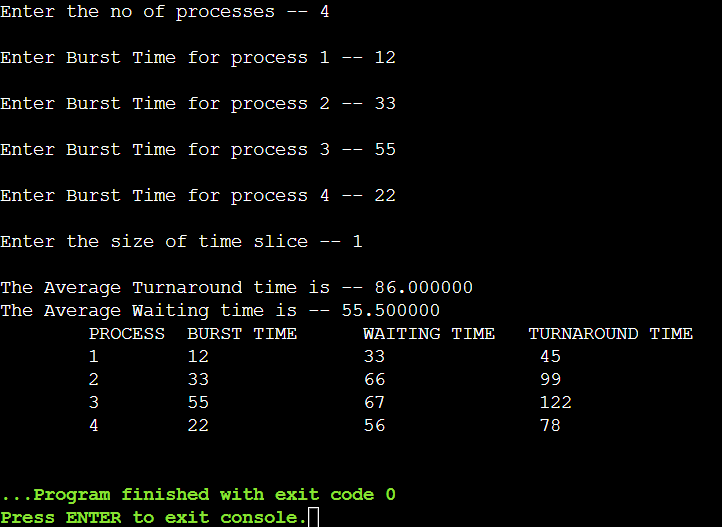
for(i=0;i<n;i++)

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

getch();

}

**OUTPUT:**

****

**PRIORITY CPU SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg, tatavg;

clrscr();

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

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

{

p[i] = i;

printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d %d",&bt[i], &pri[i]);

}

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

for(k=i+1;k<n;k++) if(pri[i] > pri[k])

{

temp=p[i];

p[i]=p[k];

p[k]=temp;

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=pri[i];

pri[i]=pri[k];

pri[k]=temp;

}

wtavg = wt[0] = 0; tatavg = tat[0] = bt[0];

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

{

wt[i] = wt[i-1] + bt[i-1];tat[i] = tat[i-1] + bt[i];

wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME"); for(i=0;i<n;i++)

printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

printf("\nAverage Waiting Time is --- %f",wtavg/n);

printf("\nAverage Turnaround Time is --- %f",tatavg/n); getch();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| } |  |  |  |  |
| ***INPUT*** |  |  |  |  |
| Enter the number of processes -- 5 | | |  |  |
| Enter the Burst Time & Priority of Process 0 --- 10 | | | 3 |  |
| Enter the Burst Time & Priority of Process 1 --- 1 | | | 1 |  |
| Enter the Burst Time & Priority of Process 2 --- 2 | | | 4 |  |
| Enter the Burst Time & Priority of Process 3 --- 1 | | | 5 |  |
| Enter the Burst Time & Priority of Process 4 --- 5 | | | 2 |  |
| ***OUTPUT*** |  |  |  |  |
| PROCESS | PRIORITY | BURST TIME | WAITING TIME | TURNAROUND TIME |
| 1 | 1 | 1 | 0 | 1 |
| 4 | 2 | 5 | 1 | 6 |
| 0 | 3 | 10 | 6 | 16 |
| 2 | 4 | 2 | 16 | 18 |
| 3 | 5 | 1 | 18 | 19 |
| Average Waiting Time is --- | | 8.200000 |  |  |
| Average Turnaround Time is | | --- 12.000000 |  |  |

**PRIORITY QUEUE Algorithm :**

#include<stdio.h>

int main()

{

int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg, tatavg;

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

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

{

p[i] = i;

printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d %d",&bt[i], &pri[i]);

}

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

for(k=i+1;k<n;k++) if(pri[i] > pri[k])

{

temp=p[i];

p[i]=p[k];

p[k]=temp;

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=pri[i];

pri[i]=pri[k];

pri[k]=temp;

}

wtavg = wt[0] = 0; tatavg = tat[0] = bt[0];

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

{

wt[i] = wt[i-1] + bt[i-1];tat[i] = tat[i-1] + bt[i];

wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME"); for(i=0;i<n;i++)

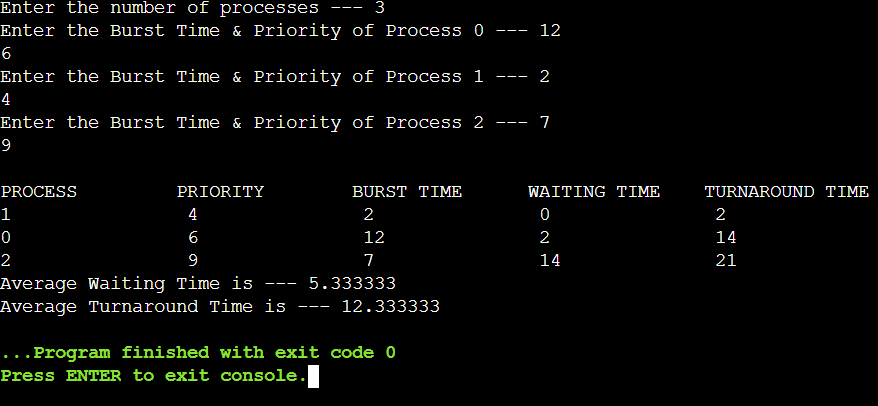
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

printf("\nAverage Waiting Time is --- %f",wtavg/n);

printf("\nAverage Turnaround Time is --- %f",tatavg/n); getch();

}

**OUTPUT*:***

****

**TASK2:**

**DESCRIPTION**

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

**PROGRAM**

main()

{

int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp; float wtavg, tatavg;

clrscr();

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

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

{

p[i] = i;

printf("Enter the Burst Time of Process %d --- ", i); scanf("%d",&bt[i]);

printf("System/User Process (0/1) ? --- "); scanf("%d", &su[i]);

}

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

for(k=i+1;k<n;k++)

if(su[i] > su[k])

{

temp=p[i];

p[i]=p[k];

p[k]=temp;

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=su[i];

su[i]=su[k];

su[k]=temp;

}

wtavg = wt[0] = 0; tatavg = tat[0] = bt[0];

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

{

wt[i] = wt[i-1] + bt[i-1];tat[i] = tat[i-1] + bt[i]

wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME"); for(i=0;i<n;i++)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]); | | | |
| printf("\nAverage Waiting Time is --- %f",wtavg/n); | | | |  |
| printf("\nAverage Turnaround Time is --- %f",tatavg/n); | | | |  |
| getch(); | |  |  |  |
|  | |  |  |  |

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***INPUT*** |  |  |  |  |
| Enter the number of processes --- 4 | | |  |  |
| Enter the Burst Time of Process 0 --- 3 | | |  |  |
| System/User Process (0/1) ? --- 1 | | |  |  |
| Enter the Burst Time of Process 1 --- 2 | | |  |  |
| System/User Process (0/1) ? --- 0 | | |  |  |
| Enter the Burst Time of Process 2 --- 5 | | |  |  |
| System/User Process (0/1) ? --- 1 | | |  |  |
| Enter the Burst Time of Process 3 --- 1 | | |  |  |
| System/User Process (0/1) ? --- 0 | | |  |  |
| ***OUTPUT*** |  |  |  |  |
| PROCESS | SYSTEM/USER PROCESS BURST TIME | | WAITING TIME | TURNAROUND TIME |
| 1 | 0 | 2 | 0 | 2 |
| 3 | 0 | 1 | 2 | 3 |
| 2 | 1 | 5 | 3 | 8 |
| 0 | 1 | 3 | 8 | 11 |
| Average Waiting Time is --- | | 3.250000 |  |  |
| Average Turnaround Time is | | --- 6.000000 |  |  |

**MULTI LEVEL QUEUE ALGORITHM:**

#include <stdio.h>

int main() {

int p[20], bt[20], su[20], wt[20], tat[20], i, k, n, temp;

float wtavg, tatavg;

printf("Enter the number of processes --- ");

scanf("%d", &n);

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

p[i] = i;

printf("Enter the Burst Time of Process %d --- ", i);

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

printf("System/User Process (0/1) ? --- ");

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

}

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

for (k = i + 1; k < n; k++) {

if (su[i] > su[k]) {

temp = p[i];

p[i] = p[k];

p[k] = temp;

temp = bt[i];

bt[i] = bt[k];

bt[k] = temp;

temp = su[i];

su[i] = su[k];

su[k] = temp;

}

}

}

wtavg = wt[0] = 0;

tatavg = tat[0] = bt[0];

for (i = 1; i < n; i++) {

wt[i] = wt[i - 1] + bt[i - 1];

tat[i] = wt[i] + bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME");

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

printf("\n%d \t\t %d \t\t\t %d \t\t %d \t\t %d", p[i], su[i], bt[i], wt[i], tat[i]);

}

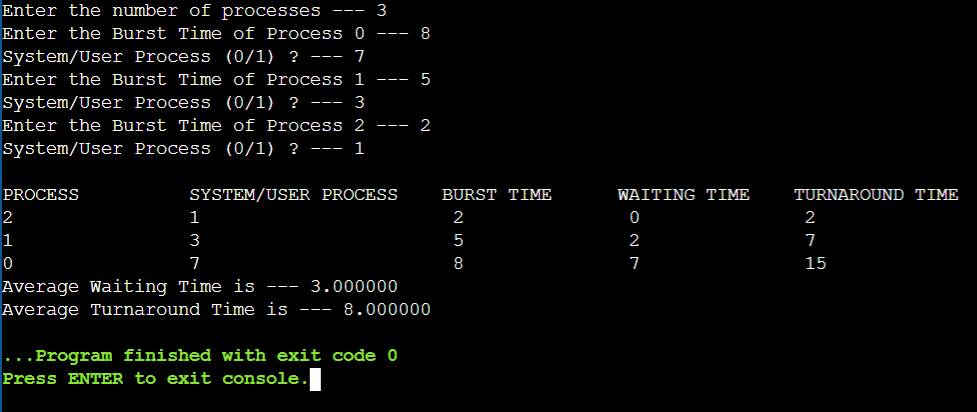
printf("\nAverage Waiting Time is --- %f", wtavg / n);

printf("\nAverage Turnaround Time is --- %f", tatavg / n);

return 0;

}

**OUTPUT:**





Operating System

**LAB-10**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

File Allocation Strategies

**Objective:**

Write a C program to simulate the following file allocation strategies.

* Sequential
* Indexed
* Linked

**TASK1**

**DESCRIPTION**

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

**SEQUENTIAL FILE ALLOCATION**

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

**LINKED FILE ALLOCATION**

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

**INDEXED FILE ALLOCATION**

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block. To find and read the ith block, the pointer in the ithindex-block entry is used.

**PROGRAM**

***SEQUENTIAL FILE ALLOCATION***

#include<stdio.h>

#include<conio.h>

struct fileTable

{

char name[20]; int sb, nob;

}ft[30];

void main()

{

int i, j, n; char s[20]; clrscr();

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 starting block of file %d | | :",i+1); |

scanf("%d",&ft[i].sb);

printf("Enter no of blocks in file %d :",i+1);

scanf("%d",&ft[i].nob);

}

printf("\nEnter the file name to be searched-- ");

scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0)

break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n"); printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob); for(j=0;j<ft[i].nob;j++)

printf("%d, ",ft[i].sb+j);

}

getch();

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| } | | | | | |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | | | |  |  |
|  |  |  |  |  |  |
|  |  | |  | |  |
|  |  |  |  |  |  |
| ***INPUT:*** |  |  |  |  |  |
| Enter no of files | :3 |  |  |  |  |
| Enter file name 1 | :A |  |  |  |  |
| Enter starting block of file 1 | | :85 |  |  |  |
| Enter no of blocks in file 1 | | :6 |  |  |  |
| Enter file name 2 | :B |  |  |  |  |
| Enter starting block of file 2 | | :102 |  |  |  |
| Enter no of blocks in file 2 | | :4 |  |  |  |
| Enter file name 3 | :C |  |  |  |  |
| Enter starting block of file 3 | | :60 |  |  |  |
| Enter no of blocks in file 3 | | :4 |  |  |  |
| Enter the file name to be searched -- B | | | |  |  |
| ***OUTPUT:*** |  |  |  |  |  |
| FILE NAME | START BLOCK | | NO OF BLOCKS | | BLOCKS OCCUPIED |
| B | 102 |  | 4 |  | 102, 103, 104, 105 |
| ***BashScript:***    #include<stdio.h>  struct fileTable  {  char name[20]; int sb, nob;  }ft[30];  int main()  {  int i, j, n; char s[20];  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 starting block of file %d :",i+1);  scanf("%d",&ft[i].sb);  printf("Enter no of blocks in file %d :",i+1);  scanf("%d",&ft[i].nob);  }  printf("\nEnter the file name to be searched-- ");  scanf("%s",s);  for(i=0;i<n;i++)  if(strcmp(s, ft[i].name)==0)  break;  if(i==n)  printf("\nFile Not Found");  else  {  printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n"); printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob); for(j=0;j<ft[i].nob;j++)  printf("%d, ",ft[i].sb+j);  }  }    ***Output:***    LINKED FILE ALLOCATION | |  |  |  |  |
| #include<stdio.h> | |  |  |  |  |
| #include<conio.h> | |  |  |  |  |
| struct fileTable |  |  |  |  |  |
| { |  |  |  |  |  |
| char name[20]; |  |  |  |  |  |
| int nob; |  |  |  |  |  |
| struct block \*sb; |  |  |  |  |  |
| }ft[30]; |  |  |  |  |  |
| struct block |  |  |  |  |  |
| { |  |  |  |  |  |
| int bno; |  |  |  |  |  |
| struct block \*next; | |  |  |  |  |
| }; |  |  |  |  |  |
| void main() |  |  |  |  |  |
| { |  |  |  |  |  |
| int i, j, n; | |  |  |  |  |
| char s[20]; | |  |  |  |  |
| struct block \*temp; | | |  |  |  |
| clrscr(); |  |  |  |  |  |
| 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 blocks of the file :");

 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 the file name to be searched -- ");

scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME 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;

}

}

getch();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| } |  |  |  |  |
| ***INPUT:*** |  |  |  |  |
| Enter no of files : 2 | |  |  |  |
| Enter file 1 | : A |  |  |  |
| Enter no of blocks in file 1 | | : 4 |  |  |
| Enter the blocks of the file 1 | |  | : 12 23 9 4 |  |
| Enter file 2 | : G |  |  |  |
| Enter no of blocks in file 2 | | : 5 |  |  |
| Enter the blocks of the file 2 | |  | : 88 77 66 55 44 |  |
| Enter the file to be searched : G | | |  |  |
| ***OUTPUT:*** |  |  |  |  |
| FILE NAME | NO OF BLOCKS | | BLOCKS OCCUPIED | |
| G | 5 |  | 88 🡪77🡪 | 66🡪55🡪44 |

***BashScript:***

#include<stdio.h>

struct fileTable

{

char name[20];

int nob;

struct block \*sb;

}ft[30];

struct block

{

int bno;

struct block \*next;

};

int main()

{

int i, j, n;

char s[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 blocks of the file :");

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 the file name to be searched -- ");

scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED"); printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob); temp=ft[i].sb;

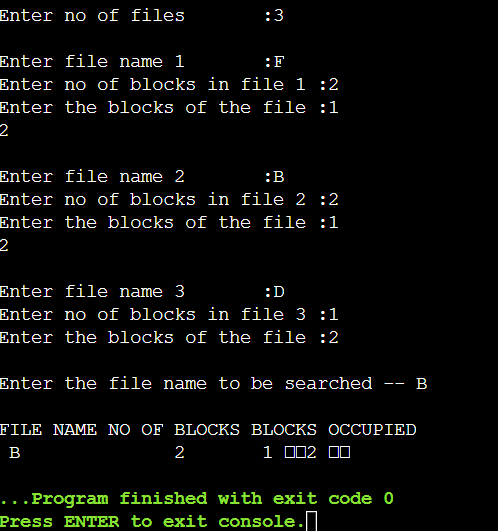
for(j=0;j<ft[i].nob;j++)

{

printf("%d ",temp->bno);temp = temp->next;

}}}

***Output:***

******

***INDEXED FILE ALLOCATION***

#include<stdio.h>

#include<conio.h>

struct fileTable

{

char name[20];

int nob, blocks[30];

}ft[30];

void main()

{

int i, j, n; char s[20]; clrscr();

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);

printf("Enter the blocks of the file :"); for(j=0;j<ft[i].nob;j++)

scanf("%d",&ft[i].blocks[j]);

}

printf("\nEnter the file name to be searched-- "); scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");

printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob);

for(j=0;j<ft[i].nob;j++)

printf("%d, ",ft[i].blocks[j]);

}

getch();

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |  |  |
| ***INPUT:*** |  |  |  |  |  |  |  |
| Enter no of files : 2 | |  |  |  |  |  |  |
| Enter file 1 | : A |  |  |  |  |  |  |
| Enter no of blocks in file 1 | | : 4 |  |  |  |  |  |
| Enter the blocks of the file 1 | |  | : 12 23 9 4 |  |  |  |  |
| Enter file 2 | : G |  |  |  |  |  |  |
| Enter no of blocks in file 2 | | : 5 |  |  |  |  |  |
| Enter the blocks of the file 2 | |  | : 88 77 66 55 44 | |  |  |  |
| Enter the file to be searched : G | | |  |  |  |  |  |
| ***OUTPUT:*** |  |  |  |  |  |  |  |
| FILE NAME | NO OF BLOCKS | | BLOCKS OCCUPIED | | | | |
| G | 5 |  | 88, | 77, | 66, | 55, | 44 |

***BashScript:***

#include<stdio.h>

struct fileTable

{

char name[20];

int nob, blocks[30];

}ft[30];

int main()

{

int i, j, n; char s[20];

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);

printf("Enter the blocks of the file :"); for(j=0;j<ft[i].nob;j++)

scanf("%d",&ft[i].blocks[j]);

}

printf("\nEnter the file name to be searched-- "); scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");

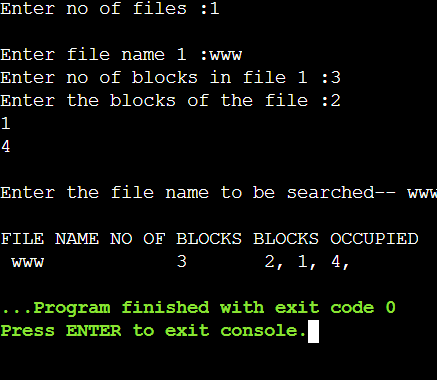
printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob);

for(j=0;j<ft[i].nob;j++)

printf("%d, ",ft[i].blocks[j]);

}

}

***Output:***



Operating System

**LAB-11**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Memory Management Techniques

**Objective:**

Write a C program to simulate the MVT and MFT memory management techniques.

Write a C program to simulate the following contiguous memory allocation techniques

* Worst-fit
* Best-fit
* First-fit

Write a C program to simulate paging technique of memory management.

**TASK1:**

**DESCRIPTION**

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

**PROGRAM**

**MFT MEMORY MANAGEMENT TECHNIQUE**

#include<stdio.h>

#include<conio.h>

main()

{

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

clrscr();

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

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

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

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

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

{

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

}

printf("\nNo. of Blocks available in memory -- %d",nob);

printf("\n\nPROCESS\tMEMORY REQUIRED\t ALLOCATED\tINTERNAL FRAGMENTATION"); for(i=0;i<n && p<nob;i++)

{

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

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

else

{

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

p++;

}

}

if(i<n)

printf("\nMemory is Full, Remaining Processes cannot be accomodated");

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |
| ***INPUT*** |  |  |  |  |  |
| Enter the total memory available (in Bytes) -- | | | | 1000 |  |
| Enter the block size (in Bytes)-- | | 300 |  |  |  |
| Enter the number of processes – 5 | |  |  |  |  |
| Enter memory required for process 1 (in Bytes) -- | | | | 275 |  |
| Enter memory required for process 2 (in Bytes) -- | | | | 400 |  |
| Enter memory required for process 3 (in Bytes) -- | | | | 290 |  |
| Enter memory required for process 4 (in Bytes) -- | | | | 293 |  |
| Enter memory required for process 5 (in Bytes) -- | | | | 100 |  |
| No. of Blocks available in memory -- | | | 3 |  |  |
| ***OUTPUT*** |  |  |  |  |  |
| PROCESS | MEMORY REQUIRED | | | ALLOCATED | INTERNAL FRAGMENTATION |
| 1 | 275 |  |  | YES | 25 |
| 2 | 400 |  |  | NO | ----- |
| 3 | 290 |  |  | YES | 10 |
| 4 | 293 |  |  | YES | 7 |
| Memory is Full, Remaining Processes cannot be accommodated | | | | |  |
| Total Internal Fragmentation is | | 42 |  |  |  |
| Total External Fragmentation is | | 100 |  |  |  |

***Bash script:***

#include <stdio.h>

int main()

{

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

int i, p = 0;

// Clear screen not needed in modern systems, so removed clrscr();

printf("Enter the total memory available (in Bytes) -- ");

scanf("%d", &ms);

printf("Enter the block size (in Bytes) -- ");

scanf("%d", &bs);

nob = ms / bs;

ef = ms - nob \* bs;

printf("\nEnter the number of processes -- ");

scanf("%d", &n);

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

{

printf("Enter memory required for process %d (in Bytes) -- ", i + 1);

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

}

printf("\nNo. of Blocks available in memory -- %d", nob);

printf("\n\nPROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION");

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

{

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

if (mp[i] > bs)

{

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

}

else

{

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

tif = tif + (bs - mp[i]);

p++;

}

}

if (i < n)

{

printf("\nMemory is Full, Remaining Processes cannot be accommodated");

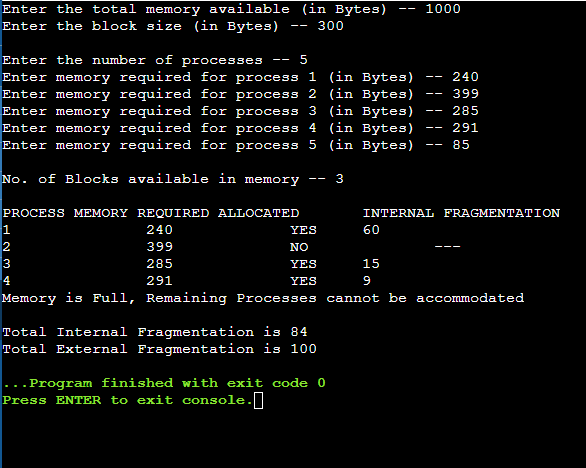
}

printf("\n\nTotal Internal Fragmentation is %d", tif);

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

return 0;

}

***Output:***

**MVT MEMORY MANAGEMENT TECHNIQUE**

#include<stdio.h>

#include<conio.h>

main()

{

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

clrscr();

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

temp=ms;

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

{

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

if(mp[i]<=temp)

{

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

}

else

{

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

}

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

}

printf("\n\nTotal Memory Available -- %d", ms);

printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED "); for(i=0;i<n;i++)

printf("\n \t%d\t\t%d",i+1,mp[i]); printf("\n\nTotal Memory Allocated is %d",ms-temp);printf("\nTotal External Fragmentation is %d",temp);

getch();

|  |  |  |  |
| --- | --- | --- | --- |
| } |  |  |  |
| ***INPUT*** |  |  |  |
| Enter the total memory available (in Bytes) -- | | | 1000 |
| Enter memory required for process 1 (in Bytes) -- | | | 400 |
| Memory is allocated for Process 1 | |  |  |
| Do you want to continue(y/n) -- | | y |  |
| Enter memory required for process 2 (in Bytes) -- | | | 275 |
| Memory is allocated for Process 2 | |  |  |
| Do you want to continue(y/n) -- | | y |  |
| Enter memory required for process 3 (in Bytes) -- | | | 550 |
| ***OUTPUT*** |  |  |  |
| Memory is Full |  |  |  |
| Total Memory Available -- 1000 | |  |  |
| PROCESS | MEMORY ALLOCATED | |  |
| 1 | 400 |  |  |
| 2 | 275 |  |  |
| Total Memory Allocated is 675 | |  |  |
| Total External Fragmentation is | | 325 |  |

***Bash script:***

#include <stdio.h>

int main()

{

int ms, mp[10], i, temp, n = 0;

char ch = 'y';

// Clear screen not needed in modern systems, so removed clrscr();

printf("\nEnter the total memory available (in Bytes) -- ");

scanf("%d", &ms);

temp = ms;

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

{

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

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

if (mp[i] <= temp)

{

printf("\nMemory is allocated for Process %d", i + 1);

temp = temp - mp[i];

}

else

{

printf("\nMemory is Full");

break;

}

printf("\nDo you want to continue (y/n) -- ");

scanf(" %c", &ch); // Added a space before %c to skip any leading whitespace

}

printf("\n\nTotal Memory Available -- %d", ms);

printf("\n\n\tPROCESS\t\tMEMORY ALLOCATED");

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

{

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

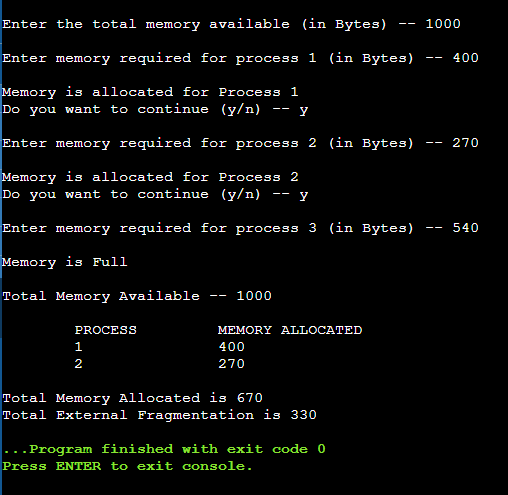
}

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

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

return 0; // Changed from getch() to return 0 for proper program termination

}



***Output:***

**TASK2:**

**DESCRIPTION**

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

**PROGRAM**

**WORST-FIT**

#include<stdio.h>

#include<conio.h> #define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp; static int bf[max],ff[max];

clrscr();

printf("\n\tMemory Management Scheme - First Fit"); printf("\nEnter the number of blocks:"); scanf("%d",&nb);

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

printf("\nEnter the size of the blocks:-\n");for(i=1;i<=nb;i++)

{

printf("Block %d:",i); scanf("%d",&b[i]);

}

printf("Enter the size of the files :-\n");for(i=1;i<=nf;i++)

{

printf("File %d:",i); scanf("%d",&f[i]);

}

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

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];if(temp>=0)

{

ff[i]=j;

break;

}

}

}

frag[i]=temp;

bf[ff[i]]=1;

}

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement"); for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();

}

***INPUT***

Enter the number of blocks: 3 Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***OUTPUT*** |  |  |  |  |
| File No | File Size | Block No | Block Size | Fragment |
| 1 | 1 | 1 | 5 | 4 |
| 2 | 4 | 3 | 7 | 3 |

***Bash script:***

#include <stdio.h>

#define MAX 25

int main()

{

int frag[MAX], b[MAX], f[MAX], i, j, nb, nf, temp;

static int bf[MAX], ff[MAX];

// Clear screen not needed in modern systems, so removed clrscr();

printf("\n\tMemory Management Scheme - First Fit");

printf("\nEnter the number of blocks: ");

scanf("%d", &nb);

printf("Enter the number of files: ");

scanf("%d", &nf);

printf("\nEnter the size of the blocks:-\n");

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

{

printf("Block %d: ", i);

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

}

printf("Enter the size of the files:-\n");

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

{

printf("File %d: ", i);

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

}

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

{

for (j = 1; j <= nb; j++)

{

if (bf[j] != 1) // if bf[j] is not allocated

{

temp = b[j] - f[i];

if (temp >= 0)

{

ff[i] = j;

frag[i] = temp;

bf[j] = 1; // mark block as allocated

break;

}

}

}

}

printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment");

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

{

if (ff[i] != 0)

{

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

}

else

{

printf("\n%d\t\t%d\t\tNot Allocated", i, f[i]);

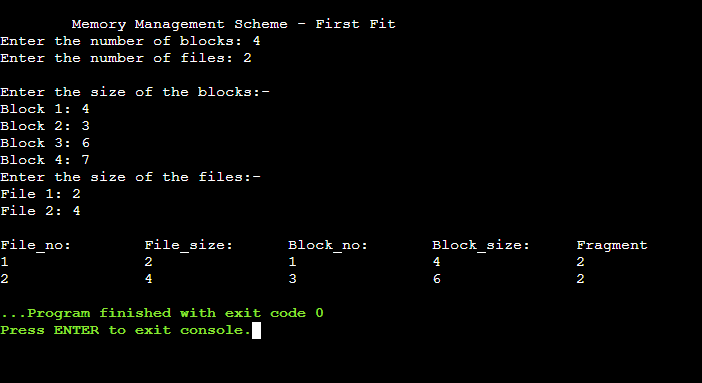
}

}

// Using getchar() instead of getch() for standard console applications

getchar();

return 0;

****}

***Output:***

**BEST-FIT**

#include<stdio.h>

#include<conio.h> #define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000; static int bf[max],ff[max];

clrscr();

printf("\nEnter the number of blocks:"); scanf("%d",&nb);

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

printf("\nEnter the size of the blocks:-\n");for(i=1;i<=nb;i++)

printf("Block %d:",i);scanf("%d",&b[i]);

printf("Enter the size of the files :-\n");for(i=1;i<=nf;i++)

{

printf("File %d:",i); scanf("%d",&f[i]);

}

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

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];if(temp>=0)

if(lowest>temp)

{

ff[i]=j;

lowest=temp;

}

}

}

frag[i]=lowest;

bf[ff[i]]=1;

lowest=10000;

}

printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();

}

***INPUT***

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***OUTPUT*** |  |  |  |  |
| File No | File Size | Block No | Block Size | Fragment |
| 1 | 1 | 2 | 2 | 1 |
| 2 | 4 | 1 | 5 | 1 |

**FIRST-FIT**

#include<stdio.h>

#include<conio.h> #define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0; static int bf[max],ff[max];

clrscr();

printf("\n\tMemory Management Scheme - Worst Fit"); printf("\nEnter the number of blocks:"); scanf("%d",&nb);

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

printf("\nEnter the size of the blocks:-\n");for(i=1;i<=nb;i++)

{

printf("Block %d:",i); scanf("%d",&b[i]);

}

printf("Enter the size of the files :-\n");for(i=1;i<=nf;i++)

{

printf("File %d:",i); scanf("%d",&f[i]);

}

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

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1) //if bf[j] is not allocated

{

temp=b[j]-f[i];if(temp>=0)

if(highest<temp)

{

ff[i]=j;

highest=temp;

}

}

}

frag[i]=highest;

bf[ff[i]]=1;

highest=0;

}

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement"); for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

getch();

}

***INPUT***

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***OUTPUT*** |  |  |  |  |
| File No | File Size | Block No | Block Size | Fragment |
| 1 | 1 | 3 | 7 | 6 |
| 2 | 4 | 1 | 5 | 1 |



Operating System

**LAB-12**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Objective:**

Write a C program to simulate the pagging memory management techniques.

**TASK:**

**DESCRIPTION**

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-managementscheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

**PROGRAM**

#include<stdio.h>

#include<conio.h>

main()

{

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

clrscr();

printf("\nEnter the memory size -- "); scanf("%d",&ms);

printf("\nEnter the page size -- "); scanf("%d",&ps);

nop = ms/ps;

printf("\nThe no. of pages available in memory are -- %d ",nop);

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

rempages = nop;

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

{

printf("\nEnter no. of pages required for p[%d]-- ",i); scanf("%d",&s[i]);

if(s[i] >rempages)

{

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

}

rempages = rempages - s[i];

printf("\nEnter pagetable for p[%d] --- ",i); for(j=0;j<s[i];j++)

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

}

printf("\nEnter Logical Address to find Physical Address "); printf("\nEnter process no. and pagenumber and offset -- ");

scanf("%d %d %d",&x,&y, &offset);

if(x>np || y>=s[i] || offset>=ps)

printf("\nInvalid Process or Page Number or offset");

else

{

pa=fno[x][y]\*ps+offset;

printf("\nThe Physical Address is -- %d",pa);

}

getch();

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |  |
| ***INPUT*** |  |  |  |  |  |  |
| Enter the memory size – 1000 | |  |  |  |  |  |
| Enter the page size -- | 100 |  |  |  |  |  |
| The no. of pages available in memory are -- 10 | | | |  |  |  |
| Enter number of processes -- | | 3 |  |  |  |  |
| Enter no. of pages required for p[1]-- | | | 4 |  |  |  |
| Enter pagetable for p[1] --- | | 8 | 6 | 9 | 5 |  |
| Enter no. of pages required for p[2]-- | | | 5 |  |  |  |
| Enter pagetable for p[2] --- | | 1 | 4 | 5 | 7 | 3 |
| Enter no. of pages required for p[3]-- | | | 5 |  |  |  |
| ***OUTPUT*** |  |  |  |  |  |  |
| Memory is Full |  |  |  |  |  |  |
| Enter Logical Address to find Physical Address | | | |  |  |  |
| Enter process no. and pagenumber and offset -- 2 | | | |  | 3 | 60 |
| The Physical Address is -- | 760 |  |  |  |  |  |

***Bash script:***

#include <stdio.h>

int main()

{

int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;

int s[10], fno[10][20];

// Clear screen not needed in modern systems, so removed clrscr();

printf("\nEnter the memory size -- ");

scanf("%d", &ms);

printf("\nEnter the page size -- ");

scanf("%d", &ps);

nop = ms / ps;

printf("\nThe no. of pages available in memory are -- %d ", nop);

printf("\nEnter number of processes -- ");

scanf("%d", &np);

rempages = nop;

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

{

printf("\nEnter no. of pages required for p[%d] -- ", i);

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

if (s[i] > rempages)

{

printf("\nMemory is Full");

break;

}

rempages -= s[i];

printf("\nEnter pagetable for p[%d] --- ", i);

for (j = 0; j < s[i]; j++)

{

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

}

}

printf("\nEnter Logical Address to find Physical Address ");

printf("\nEnter process no. and page number and offset -- ");

scanf("%d %d %d", &x, &y, &offset);

if (x > np || y >= s[x] || offset >= ps) // Fixed condition check

{

printf("\nInvalid Process or Page Number or Offset");

}

else

{

pa = fno[x][y] \* ps + offset;

printf("\nThe Physical Address is -- %d", pa);

}

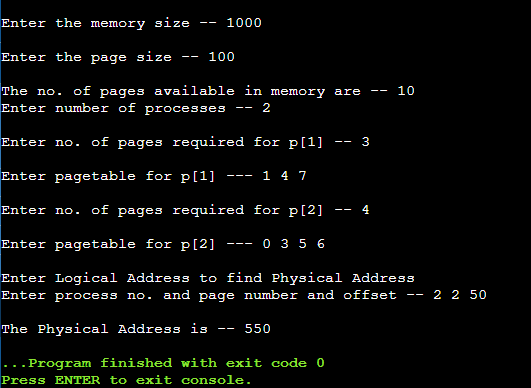
// Using getchar() instead of getch() for standard console applications

getchar();

return 0;

}

***Output:***





Operating System

**LAB-13**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Deadlock Management Techniques

**Objective:**

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Write a C program to simulate disk scheduling algorithms

* FCFS
* SCAN
* C-SCAN

**TASK1:**

**DESCRIPTION**

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker’s algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

**PROGRAM**

#include<stdio.h> struct file

{

int all[10]; int max[10]; int need[10]; int flag;

};

void main()

{

struct file f[10]; int fl;

int i, j, k, p, b, n, r, g, cnt=0, id, newr; int avail[10],seq[10];

clrscr();

printf("Enter number of processes -- "); scanf("%d",&n);

printf("Enter number of resources -- "); scanf("%d",&r);

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

{

printf("Enter details for P%d",i); printf("\nEnter allocation\t -- \t"); for(j=0;j<r;j++)

scanf("%d",&f[i].all[j]); printf("Enter Max\t\t -- \t"); for(j=0;j<r;j++)

scanf("%d",&f[i].max[j]);

f[i].flag=0;

}

printf("\nEnter Available Resources\t -- \t"); for(i=0;i<r;i++)

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

printf("\nEnter New Request Details -- "); printf("\nEnter pid \t -- \t"); scanf("%d",&id);

printf("Enter Request for Resources \t -- \t"); for(i=0;i<r;i++)

{

scanf("%d",&newr); f[id].all[i] += newr;

avail[i]=avail[i] - newr;

}

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

{

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

{

f[i].need[j]=f[i].max[j]-f[i].all[j];if(f[i].need[j]<0)

f[i].need[j]=0;

}

}

cnt=0;

fl=0;

while(cnt!=n)

{

g=0;

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

{

if(f[j].flag==0)

{

b=0;

for(p=0;p<r;p++)

{

if(avail[p]>=f[j].need[p])

b=b+1;

else

b=b-1;

}

if(b==r)

{

printf("\nP%d is visited",j); seq[fl++]=j;

f[j].flag=1;

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

avail[k]=avail[k]+f[j].all[k];

cnt=cnt+1;

printf("(");

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

printf("%3d",avail[k]);

printf(")");

g=1;

}

}

}

if(g==0)

{

printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED"); printf("\n SYSTEM IS IN UNSAFE STATE");

goto y;

}

}

printf("\nSYSTEM IS IN SAFE STATE"); printf("\nThe Safe Sequence is -- ("); for(i=0;i<fl;i++)

printf("P%d ",seq[i]); printf(")");

y:printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");

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

{

printf("P%d\t",i);

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

printf("%6d",f[i].all[j]);

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

printf("%6d",f[i].max[j]);

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

printf("%6d",f[i].need[j]);

printf("\n");

}

getch();

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ***INPUT*** |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Enter number of processes | | | | | |  | – | 5 |  |  |  |  |  |
| Enter number of resources | | | | | |  | -- | 3 |  |  |  |  |  |
| Enter details for P0 | | | | |  |  |  |  |  |  |  |  |  |
| Enter allocation | |  |  |  | -- |  | 0 | 1 | | 0 |  |  |  |
| Enter Max |  |  |  |  | -- |  |  | 7 |  | 5 | 3 |  |  |
| Enter details for P1 | | | | |  |  |  |  |  |  |  |  |  |
| Enter allocation | |  |  |  | -- |  | 2 | 0 |  | 0 |  |  |  |
| Enter Max |  |  |  |  | -- |  | 3 | 2 |  | 2 |  |  |  |
| Enter details for P2 | | | | |  |  |  |  |  |  |  |  |  |
| Enter allocation | |  |  |  | -- |  | 3 | 0 |  | 2 |  |  |  |
| Enter Max |  |  |  |  | -- |  | 9 | 0 |  | 2 |  |  |  |
| Enter details for P3 | | | | |  |  |  |  |  |  |  |  |  |
| Enter allocation | |  |  |  | -- |  | 2 | 1 |  | 1 |  |  |  |
| Enter Max |  |  |  |  | -- |  | 2 | 2 |  | 2 |  |  |  |
| Enter details for P4 | | | | |  |  |  |  |  |  |  |  |  |
| Enter allocation | |  |  |  | -- |  | 0 | 0 |  | 2 |  |  |  |
| Enter Max |  |  |  |  | -- |  | 4 | 3 |  | 3 |  |  |  |
| Enter Available Resources -- | | | | | | 3 | 3 | 2 |  |  |  |  |  |
| Enter New Request Details -- | | | | | |  |  |  |  |  |  |  |  |
| Enter pid | -- |  | 1 | |  |  |  |  |  |  |  |  |  |
| Enter Request for Resources | | | | | | -- | 1 |  | 0 | 2 |  |  |  |
| ***OUTPUT*** |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P1 is visited( | 5 | 3 | 2) | |  |  |  |  |  |  |  |  |  |
| P3 is visited( | 7 | 4 | 3) | |  |  |  |  |  |  |  |  |  |
| P4 is visited( | 7 | 4 | 5) | |  |  |  |  |  |  |  |  |  |
| P0 is visited( | 7 | 5 | 5) | |  |  |  |  |  |  |  |  |  |
| P2 is visited( 10 | | 5 | | 7) |  |  |  |  |  |  |  |  |  |
| SYSTEM IS IN SAFE STATE | | | | |  |  |  |  |  |  |  |  |  |
| The Safe Sequence is -- (P1 P3 P4 P0 P2 ) | | | | | | | |  |  |  |  |  |  |
| Process |  |  | Allocation | | |  |  |  | Max |  | Need | |  |
| P0 |  |  | 0 | 1 | 0 |  |  | 7 | 5 | 3 | 7 | 4 | 3 |
| P1 |  |  | 3 | 0 | 2 |  |  | 3 | 2 | 2 | 0 | 2 | 0 |
| P2 |  |  | 3 | 0 | 2 |  |  | 9 | 0 | 2 | 6 | 0 | 0 |
| P3 |  |  | 2 | 1 | 1 |  |  | 2 | 2 | 2 | 0 | 1 | 1 |
| P4 |  |  | 0 | 0 | 2 |  |  | 4 | 3 | 3 | 4 | 3 | 1 |

**TASK2:**

**DESCRIPTION**

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

**PROGRAM**

**FCFS DISK SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int t[20], n, I, j, tohm[20], tot=0; float avhm;

clrscr();

printf(“enter the no.of tracks”); scanf(“%d”,&n);

printf(“enter the tracks to be traversed”); for(i=2;i<n+2;i++)

scanf(“%d”,&t\*i+);

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

{

tohm[i]=t[i+1]-t[i];if(tohm[i]<0) tohm[i]=tohm[i]\*(-1);

}

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

tot+=tohm[i];

avhm=(float)tot/n;

printf(“Tracks traversed\tDifference between tracks\n”); for(i=1;i<n+1;i++)

printf(“%d\t\t\t%d\n”,t\*i+,tohm\*i+); printf("\nAverage header movements:%f",avhm); getch();

}

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***INPUT*** | | |  |  |  |  |  |  |  |  |  |
| Enter no.of tracks:9 | | |  |  |  |  |  |  |  |  |  |
| Enter track position:55 | | | | 58 | 60 | 70 | 18 | 90 | 150 | 160 | 184 |
| ***OUTPUT*** | | |  |  |  |  |  |  |  |  |  |
| Tracks traversed | | | Difference between tracks | | | |  |  |  |  |  |
| 55 | | |  |  | 45 |  |  |  |  |  |  |
| 58 | | |  |  | 3 |  |  |  |  |  |  |
| 60 | | |  |  | 2 |  |  |  |  |  |  |
| 70 | | |  |  | 10 |  |  |  |  |  |  |
| 18 | | |  |  | 52 |  |  |  |  |  |  |
| 90 | | |  |  | 72 |  |  |  |  |  |  |
|  | Average header movements:30.888889 | | | |
|  |  | | | |
|  |  | | | |
|  |  | | | |
|  |  | | | |

**SCAN DISK SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

clrscr();

printf("enter the no of tracks to be traveresed"); scanf("%d'",&n);

printf("enter the position of head"); scanf("%d",&h);

t[0]=0;t[1]=h; printf("enter the tracks"); for(i=2;i<n+2;i++)

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

for(i=0;i<n+2;i++)

{

for(j=0;j<(n+2)-i-1;j++)

{if(t[j]>t[j+1])

{

temp=t[j];

t[j]=t[j+1];

t[j+1]=temp;

} } }

for(i=0;i<n+2;i++)

if(t[i]==h)

j=i;k=i;

p=0;

while(t[j]!=0)

{

atr[p]=t[j]; j--;

p++;

}

atr[p]=t[j];

for(p=k+1;p<n+2;p++,k++)

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

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

{

if(atr[j]>atr[j+1]) d[j]=atr[j]-atr[j+1];

else

d[j]=atr[j+1]-atr[j];sum+=d[j];

}

printf("\nAverage header movements:%f",(float)sum/n); getch();

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |  |  |  |  |
| ***INPUT*** |  |  |  |  |  |  |  |  |  |
| Enter no.of tracks:9 |  |  |  |  |  |  |  |  |  |
| Enter track position:55 | | 58 | 60 | 70 | 18 | 90 | 150 | 160 | 184 |
|  |  |  |  |  |  |  |  |  |  |
|  |  | | | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**C-SCAN DISK SCHEDULING ALGORITHM**

#include<stdio.h>

main()

{

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0; clrscr();

printf("enter the no of tracks to be traveresed"); scanf("%d'",&n);

printf("enter the position of head"); scanf("%d",&h);

t[0]=0;t[1]=h; printf("enter total tracks"); scanf("%d",&tot); t[2]=tot-1;

printf("enter the tracks"); for(i=3;i<=n+2;i++)

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

for(i=0;i<=n+2;i++)

for(j=0;j<=(n+2)-i-1;j++)if(t[j]>t[j+1])

{

temp=t[j];

t[j]=t[j+1];

t[j+1]=temp;

}

for(i=0;i<=n+2;i++)

if(t[i]==h)

j=i;break;

p=0; while(t[j]!=tot-1)

{

atr[p]=t[j];

j++;

p++;

}

atr[p]=t[j];

p++;

i=0;

while(p!=(n+3) && t[i]!=t[h])

{

atr[p]=t[i];

i++;

p++;

}

for(j=0;j<n+2;j++)

{

if(atr[j]>atr[j+1]) d[j]=atr[j]-atr[j+1];

else

d[j]=atr[j+1]-atr[j];sum+=d[j];

}

printf("total header movements%d",sum); printf("avg is %f",(float)sum/n);

getch();

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| } |  |  |  |  |  |  |  |  |
| ***INPUT*** |  |  |  |  |  |  |  |  |
| Enter the track position : 55 | 58 | 60 | 70 | 18 | 90 | 150 | 160 | 184 |
| Enter starting position : 100 |  |  |  |  |  |  |  |  |



Operating System

**LAB-14**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Page Replacement Algorithms

**Objective:**

Write a C program to simulate page replacement algorithms

* FIFO
* LRU
* LFU

Write a C program to simulate page replacement algorithms

* Optimal

**TASK1:**

**DESCRIPTION**

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

**PROGRAM**

**FIFO PAGE REPLACEMENT ALGORITHM**

#include<stdio.h>

#include<conio.h>

main()

{

int i, j, k, f, pf=0, count=0, rs[25], m[10], n; clrscr();

printf("\n Enter the length of reference string -- "); scanf("%d",&n);

printf("\n Enter the reference string -- "); for(i=0;i<n;i++)

scanf("%d",&rs[i]); printf("\n Enter no. of frames -- "); scanf("%d",&f);

for(i=0;i<f;i++) m[i]=-1;

printf("\n The Page Replacement Process is -- \n"); for(i=0;i<n;i++)

{

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

{

if(m[k]==rs[i])

break;

}

if(k==f)

{

m[count++]=rs[i];

pf++;

}

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

printf("\t%d",m[j]);

if(k==f)

printf("\tPF No. %d",pf); printf("\n");

if(count==f)

count=0;

}

printf("\n The number of Page Faults using FIFO are %d",pf); getch();

***INPUT***

Enter the length of reference string – 20

|  |  |  |
| --- | --- | --- |
| Enter the reference string -- | | 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 |
| Enter no. of frames -- | 3 |  |

**LRU PAGE REPLACEMENT ALGORITHM**

#include<stdio.h>

#include<conio.h>

main()

{

int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1; clrscr();

printf("Enter the length of reference string -- "); scanf("%d",&n);

printf("Enter the reference string -- "); for(i=0;i<n;i++)

{

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

flag[i]=0;

}

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

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

{

count[i]=0; m[i]=-1;

}

printf("\nThe Page Replacement process is -- \n"); for(i=0;i<n;i++)

{

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

{

if(m[j]==rs[i])

{

flag[i]=1;

count[j]=next;

next++;

}

}

if(flag[i]==0)

{

if(i<f)

{

m[i]=rs[i];

count[i]=next;

next++;

}

else

{

min=0;

for(j=1;j<f;j++)

if(count[min] > count[j]) min=j;

m[min]=rs[i];

count[min]=next;

next++;

}

pf++;

}

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

if(flag[i]==0)

printf("PF No. -- %d" , pf); printf("\n");

}

printf("\nThe number of page faults using LRU are %d",pf); getch();

}

***INPUT***

Enter the length of reference string -- 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 Enter the number of frames -- 3

**LFU PAGE REPLACEMENT ALGORITHM**

#include<stdio.h>

#include<conio.h>

main()

{

int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0; clrscr();

printf("\nEnter number of page references -- "); scanf("%d",&m);

printf("\nEnter the reference string -- "); for(i=0;i<m;i++)

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

printf("\nEnter the available no. of frames -- "); scanf("%d",&f);

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

{

cntr[i]=0; a[i]=-1;

}

Printf(“\nThe Page Replacement Process is – \n“); for(i=0;i<m;i++)

{

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

if(rs[i]==a[j])

{

cntr[j]++;

break;

}

if(j==f)

{

min = 0; for(k=1;k<f;k++)

if(cntr[k]<cntr[min])

min=k;

a[min]=rs[i];

cntr[min]=1;

pf++;

}

printf("\n");

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

printf("\t%d",a[j]);

if(j==f)

printf(“\tPF No. %d”,pf);

}

printf("\n\n Total number of page faults -- %d",pf); getch();

|  |  |
| --- | --- |
| } |  |
| ***INPUT*** |  |
| Enter number of page references --10 |  |
| Enter the reference string -- | 1 2 3 4 5 2 5 2 5 1 4 3 |
| Enter the available no. of frames -- 3 |  |
|  |  |

**TASK2:**

**DESCRIPTION**

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

**PROGRAM**

#include<stdio.h> int n;

main()

{

int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s; int count=1,pf=0,p=0;

float pfr; clrscr();

printf("Enter maximum limit of the sequence: "); scanf("%d",&max);

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

scanf("%d",&seq[i]); printf("\nEnter no. of frames: "); scanf("%d",&n);

fr[0]=seq[0];

pf++;

printf("%d\t",fr[0]);

i=1;

while(count<n)

{

flag=1;

p++;

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

{

if(seq[i]==seq[j])

flag=0;

}

if(flag!=0)

{

fr[count]=seq[i];

printf("%d\t",fr[count]);

count++;

pf++;

}

i++;

}

printf("\n");

for(i=p;i<max;i++)

{

flag=1;

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

{

if(seq[i]==fr[j])

flag=0;

}

if(flag!=0)

{

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

{

m=fr[j];

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

{

if(seq[k]==m)

{

pos[j]=k;

break;

}

else

pos[j]=1;

}

}

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

{

if(pos[k]==1)

flag=0;

}

if(flag!=0)

s=findmax(pos);

if(flag==0)

{

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

{

if(pos[k]==1)

{

s=k;

break;

}

}

}

fr[s]=seq[i];

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

printf("%d\t",fr[k]);

pf++;

printf("\n");

}

}

pfr=(float)pf/(float)max;

printf("\nThe no. of page faults are %d",pf); printf("\nPage fault rate %f",pfr);

getch();

|  |  |
| --- | --- |
| } |  |
| int findmax(int a[]) |  |
| { |  |
| int max,i,k=0; |  |
| max=a[0]; |  |
| for(i=0;i<n;i++) |  |
| { |  |
| if(max<a[i]) |  |
| { |  |
| max=a[i]; |  |
| k=i; |  |
| } |  |
| } |  |
| return k; |  |
| } |  |
| ***INPUT*** |  |
| Enter number of page references --10 |  |
| Enter the reference string -- | 1 2 3 4 5 2 5 2 5 1 4 3 |

Enter the available no. of frames -- 3



Operating System

**LAB-15**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Process Synchronization

**Objective:**

1. Write a C program to simulate producer-consumer problem using semaphores.

2. Write a C program to simulate the concept of Dining-Philosophers problem

**TASK1:**

**DESCRIPTION**

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

**12.3PROGRAM**

#include<stdio.h> void main()

{

int buffer[10], bufsize, in, out, produce, consume, choice=0; in = 0;

out = 0; bufsize = 10;

while(choice !=3)

{

printf(“\n1. Produce \t 2. Consume \t3. Exit”); printf(“\nEnter your choice: ”);

scanf(“%d”, &choice);

switch(choice) {

case 1: if((in+1)%bufsize==out) printf(“\nBuffer is Full”);

else

{

printf(“\nEnter the value: “); scanf(“%d”, &produce); buffer[in] = produce;

in = (in+1)%bufsize;

}

Break; case 2: if(in == out)

printf(“\nBuffer is Empty”);

else

{

consume = buffer[out];

printf(“\nThe consumed value is %d”, consume); out = (out+1)%bufsize;

|  |  |  |
| --- | --- | --- |
|  |  | } |
|  |  | break; |
| } } } |  |  |
| ***OUTPUT*** |  |  |
| 1. Produce | 2. Consume | 3. Exit |
| Enter your choice: 2 | |  |
| Buffer is Empty |  |  |
| 1. Produce | 2. Consume | 3. Exit |
| Enter your choice: 1 | |  |
| Enter the value: 100 | |  |
| 1. Produce | 2. Consume | 3. Exit |
| Enter your choice: 2 | |  |
| The consumed value is 100 | |  |
| 1. Produce | 2. Consume | 3. Exit |

Enter your choice: 3

**TASK2:**

**DESCRIPTION**

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

**PROGRAM**

int tph, philname[20], status[20], howhung, hu[20], cho; main()

{

int i; clrscr();

printf("\n\nDINING PHILOSOPHER PROBLEM"); printf("\nEnter the total no. of philosophers: "); scanf("%d",&tph);

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

{

philname[i] = (i+1); status[i]=1;

}

printf("How many are hungry : "); scanf("%d", &howhung); if(howhung==tph)

{

printf("\nAll are hungry..\nDead lock stage will occur"); printf("\nExiting..");

}

else

{

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

{

printf("Enter philosopher %d position: ",(i+1)); scanf("%d", &hu[i]);

status[hu[i]]=2;

}

do

{

printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:"); scanf("%d", &cho);

switch(cho)

{

|  |  |
| --- | --- |
| case 1: | one(); |
|  | break; |
| case 2: | two(); |
|  | break; |
| case 3: | exit(0); |

default: printf("\nInvalid option..");

}

}while(1);

}

}

one()

{

int pos=0, x, i;

printf("\nAllow one philosopher to eat at any time\n"); for(i=0;i<howhung; i++, pos++)

{

printf("\nP %d is granted to eat", philname[hu[pos]]); for(x=pos;x<howhung;x++)

printf("\nP %d is waiting", philname[hu[x]]);

}

}

two()

{

int i, j, s=0, t, r, x;

printf("\n Allow two philosophers to eat at same time\n"); for(i=0;i<howhung;i++)

{

for(j=i+1;j<howhung;j++)

{

if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)

{

printf("\n\ncombination %d \n", (s+1)); t=hu[i];

r=hu[j];

s++;

printf("\nP %d and P %d are granted to eat", philname[hu[i]], philname[hu[j]]);

for(x=0;x<howhung;x++)

{

if((hu[x]!=t)&&(hu[x]!=r))

printf("\nP %d is waiting", philname[hu[x]]);

}

}

}

}

}

***INPUT***

DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry : 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5