

## **LAB ASSIGNMENT 1**

**Problem Statement:** Design suitable data structures and implement pass-1 of a two-pass assembler. Implementation should consist of a few instructions from each category and few assembler directives.

**Theory:** An assembler generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. The assembler divide these tasks in two passes.

During the first pass, the assembler checks to see if the instructions are legal in the current assembly mode.

On the first pass, the assembler performs the following tasks:

- Checks to see if the instructions are legal in the current assembly mode.
- Allocates space for instructions and storage areas you request.
- Fills in the values of constants, where possible.
- Builds a symbol table, also called a cross-reference table, and makes an entry in this table for every symbol it encounters in the label field of a statement.

The assembler reads one line of the source file at a time. If this source statement has a valid symbol in the label field, the assembler ensures that the symbol has not already been used as a label. If this is the first time the symbol has been used as a label, the assembler adds the label to the symbol table and assigns the value of the current location counter to the symbol.

Next, the assembler examines the instruction's mnemonic. If the mnemonic is for a machine instruction that is legal for the current assembly mode, the assembler determines the format of the instruction (for example, XO format). The assembler then allocates the number of bytes necessary to hold the machine code for the instruction. The contents of the location counter are incremented by this number of bytes.

When the assembler encounters a comment or an end-of-line character, the assembler starts scanning the next instruction statement. The assembler keeps scanning statements and building its symbol table until there are no more statements to read.

At the end of the first pass, all the necessary space has been allocated and each symbol defined in the program has been associated with a location counter value in the symbol table. When there are no more source statements to read, the second pass starts at the beginning of the program.

### **Code:**

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

```
#include<string.h>
```

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

```
void main()
```

```

{

char opcode[10], operand[10], label[10], code[10], mnemonic[3];

int locctr, start, length;

FILE *fp1, *fp3, *fp4, *fp2;


fp1=fopen("input.txt", "r");

fp2=fopen("optab.txt", "r");

fp3=fopen("symtab.txt", "w");

fp4=fopen("output.txt", "w");


fscanf(fp1, "%s\t%s\t%s", label, opcode, operand);


if(strcmp(opcode, "START")==0)
{
    start=atoi(operand);

    locctr=start;

    fprintf(fp4, "\t%s\t%s\t%s\n", label, opcode, operand);

    fscanf(fp1, "%s\t%s\t%s\n", label, opcode, operand);
}
else

    locctr=0;


while(strcmp(opcode,"END")!=0)
{
    fprintf(fp4, "%d\t", locctr);

    if (strcmp(label, "***")!=0)

        fprintf(fp3, "%s\t%d\n", label, locctr);
}

```

```
fscanf(fp2, "%s\t%s", code, mnemonic);
```

```
while(strcmp(code, "END")!=0)
```

```
{
```

```
    if(strcmp(opcode, code)==0)
```

```
    {
```

```
        locctr+=3;
```

```
        break;
```

```
    }
```

```
    fscanf(fp2, "%s\t%s", code, mnemonic);
```

```
}
```

```
if(strcmp(opcode, "WORD")==0)
```

```
    locctr+=3;
```

```
else if(strcmp(opcode, "RESW")==0)
```

```
    locctr+=(3*(atoi(operand)));
```

```
else if (strcmp(opcode, "RESB")==0)
```

```
    locctr+=atoi(operand);
```

```
else if (strcmp(opcode, "BYTE")==0)
```

```
    ++locctr;
```

```
fprintf(fp4, "%s\t%s\t%s\t\n", label, opcode, operand);
```

```
fscanf(fp1, "%s\t%s\t%s", label, opcode, operand);
```

```
}
```

```
fprintf(fp4, "%d\t%s\t%s\t%s\n", locctr, label, opcode, operand);
```

```
length=locctr-start;
```

```
printf("The length of the code: %d\n", length);
```

```
fclose(fp1);
```

```
fclose(fp2);
```

```
fclose(fp3);
```

```
fclose(fp4);
```

```
}
```

### **Input:**

#### Optab.txt

```
1  START  *
2  LDA 03
3  STA 0F
4  LDCH 53
5  STCH 57
6  END *
```

#### Symtab.txt

```
1  ALPHA 2012
2  FIVE 2018
3  CHARZ 2021
4  C1 2022
```

#### Input.txt

```
1  ** START 2000
2  ** LDA FIVE
3  ** STA ALPHA
4  ** LDCH CHARZ
5  ** STCH C1
6  ALPHA RESW 2
7  FIVE WORD 5
8  CHARZ BYTE C'Z'
9  C1 RESB 1
10 ** END **
```

## Output:

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>

```
PS D:\Desktop_II\SPOS Lab\Lab assignment 1> cd "d:\Desktop_II\SPOS Lab\Lab assignment 1\Pass 1\" ; if ($?) { gcc ASSEMBLER  
.c -o ASSEMBLER } ; if ($?) { .\ASSEMBLER }
```

The length of the code: 23

```
PS D:\Desktop_II\SPOS Lab\Lab assignment 1\Pass 1>
```

## Output.txt:

1		**	START	2000	
2	2000	**	LDA	FIVE	
3	2003	**	STA	ALPHA	
4	2006	**	LDCH	CHARZ	
5	2009	**	STCH	C1	
6	2012		ALPHA	RESW	2
7	2018		FIVE	WORD	5
8	2021		CHARZ	BYTE	C'Z'
9	2022		C1	RESB	1
10	2023	**	END	**	

**Conclusion:** Thus we have implemented the pass-1 of a two-pass assembler.

PRANJAL KUMAR

ISA-I

2203263

## LAB ASSIGNMENT 2

**Problem Statement:** Implement pass-II of a two-pass assembler. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment.

**Theory:** An assembler generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. The assembler divide these tasks in two passes.

During the first pass, the assembler checks to see if the instructions are legal in the current assembly mode.

On the second pass, the assembler performs the following tasks:

1. Generate object code by converting symbolic op-code into respective numeric op-code
2. Generate data for literals and look for values of symbols.

The assembler now generates the machine code file along some other information for the loader. This file can now be executed by the operating system.

### Code:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
void main()
{
    char a[10], ad[10], label[10], opcode[10], operand[10], symbol[10], ch;
    int st, diff, i, address, add, len, actual_len, finaddr, prevaddr, j = 0;
    char mnemonic[15][15] = {"LDA", "STA", "LDCH", "STCH"};
    char code[15][15] = {"33", "44", "53", "57"};
    FILE *fp1, *fp2, *fp3, *fp4;
    // clrscr();
    system("cls");
    fp1 = fopen("ASSMLIST.DAT", "w");
    fp2 = fopen("SYMTAB.DAT", "r");
    fp3 = fopen("INTERMED.DAT", "r");
    fp4 = fopen("OBJCODE.DAT", "w");
    fscanf(fp3, "%s%s%s", label, opcode, operand);
```

```

while (strcmp(opcode, "END") != 0)
{
    prevaddr = address;
    fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
}
finaddr = address;
fclose(fp3);
fp3 = fopen("INTERMED.DAT", "r");

fscanf(fp3, "%s%s%s", label, opcode, operand);
if (strcmp(opcode, "START") == 0)
{
    fprintf(fp1, "\t%s\t%s\t%s\n", label, opcode, operand);
    fprintf(fp4, "H^s^oo^oo%d\n", label, operand, finaddr);
    fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
    st = address;
    diff = prevaddr - st;
    fprintf(fp4, "T^oo%d^%d", address, diff);
}
while (strcmp(opcode, "END") != 0)
{
    if (strcmp(opcode, "BYTE") == 0)
    {
        fprintf(fp1, "%d\t%s\t%s\t%s\t", address, label, opcode, operand);
        len = strlen(operand);
        actual_len = len - 3;
        fprintf(fp4, "^");
        for (i = 2; i < (actual_len + 2); i++)
        {
            itoa(operand[i], ad, 16);
            fprintf(fp1, "%s", ad);
            fprintf(fp4, "%s", ad);
        }
        fprintf(fp1, "\n");
    }
    else if (strcmp(opcode, "WORD") == 0)
    {
        len = strlen(operand);

```

```

    itoa(atoi(operand), a, 10);
    fprintf(fp1, "%d\t%s\t%s\t%s\ttooooo%s\n", address, label, opcode, operand, a);
    fprintf(fp4, "^oooo%s", a);
}
else if ((strcmp(opcode, "RESB") == 0) || (strcmp(opcode, "RESW") == 0))
    fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
else
{
    while (strcmp(opcode, mnemonic[j]) != 0)
        j++;
    if (strcmp(operand, "COPY") == 0)
        fprintf(fp1, "%d\t%s\t%s\t%s\tsoooo\n", address, label, opcode, operand, code[j]);
    else
    {
        rewind(fp2);
        fscanf(fp2, "%s%d", symbol, &add);
        while (strcmp(operand, symbol) != 0)
            fscanf(fp2, "%s%d", symbol, &add);
        fprintf(fp1, "%d\t%s\t%s\t%s\t%s%d\n", address, label, opcode, operand, code[j], add);
        fprintf(fp4, "^%s%d", code[j], add);
    }
}
fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
}
fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
fprintf(fp4, "\nE^oo%d", st);
printf("\n Intermediate file is converted into object code");
// fcloseall();
fclose(fp1);
fclose(fp2);
fclose(fp3);
fclose(fp4);

printf("\n\nThe contents of Intermediate file:\n\n\t");
fp3 = fopen("INTERMED.DAT", "r");
ch = fgetc(fp3);
while (ch != EOF)
{
    printf("%c", ch);

```



```
    ch = fgetc(fp3);
}
printf("\n\nThe contents of Symbol Table :\n\n");
fp2 = fopen("SYMTAB.DAT", "r");
ch = fgetc(fp2);
while (ch != EOF)
{
    printf("%c", ch);
    ch = fgetc(fp2);
}
printf("\n\nThe contents of Output file :\n\n");
fp1 = fopen("ASSMLIST.DAT", "r");
ch = fgetc(fp1);
while (ch != EOF)
{
    printf("%c", ch);
    ch = fgetc(fp1);
}
printf("\n\nThe contents of Object code file :\n\n");
fp4 = fopen("OBJCODE.DAT", "r");
ch = fgetc(fp4);
while (ch != EOF)
{
    printf("%c", ch);
    ch = fgetc(fp4);
}
// fcloseall();
fclose(fp1);
fclose(fp2);
fclose(fp3);
fclose(fp4);
getch();
}
```

Input:

INTERMED.DAT

```
1  COPY      START      2000
2  2000      **      LDA      FIVE
3  2003      **      STA      ALPHA
4  2006      **      LDCH     CHARZ
5  2009      **      STCH     C1
6  2012      ALPHA     RESW     1
7  2015      FIVE      WORD     5
8  2018      CHARZ     BYTE     C'EOF'
9  2019      C1        RESB     1
10 2020      **      END      **
```

SYMTAB.DAT

```
1  ALPHA     2012
2  FIVE      2015
3  CHARZ     2018
4  C1        2019
```

## Output:

Intermediate file is converted into object code

The contents of Intermediate file:

	COPY	START	2000
2000	**	LDA	FIVE
2003	**	STA	ALPHA
2006	**	LDCH	CHARZ
2009	**	STCH	C1
2012	ALPHA	RESW	1
2015	FIVE	WORD	5
2018	CHARZ	BYTE	C'EOF'
2019	C1	RESB	1
2020	**	END	**

The contents of Symbol Table :

ALPHA	2012
FIVE	2015
CHARZ	2018
C1	2019

The contents of Output file :

	COPY	START	2000	
2000	**	LDA	FIVE	332015
2003	**	STA	ALPHA	442012
2006	**	LDCH	CHARZ	532018
2009	**	STCH	C1	572019
2012	ALPHA	RESW	1	
2015	FIVE	WORD	5	000005
2018	CHARZ	BYTE	C'EOF'	454f46
2019	C1	RESB	1	
2020	**	END	**	

The contents of Object code file :

H^COPY^002000^002020  
T^002000^19^332015^442012^532018^572019^000005^454f46  
E^002000

### ASSIMLIST.DAT :

1		COPY	START	2000	
2	2000	**	LDA FIVE	332015	
3	2003	**	STA ALPHA	442012	
4	2006	**	LDCH	CHARZ	532018
5	2009	**	STCH	C1	572019
6	2012	ALPHA	RESW	1	
7	2015	FIVE	WORD	5	000005
8	2018	CHARZ	BYTE	C'EOF'	454f46
9	2019	C1	RESB	1	
10	2020	**	END **		

### OBJCODE.DAT

1	H^COPY^002000^002020
2	T^002000^19^332015^442012^532018^572019^000005^454f46
3	E^002000

**Conclusion:** Thus we have implemented the pass-2 of a two-pass assembler.

### **LAB ASSIGNMENT 3**

**Problem Statement:** Design suitable data structure and implement macro definition and macro expansion and processing for a sample macro with positional and keyword parameters.

**Theory:** Macros are portions of assembly language statements that are textually substituted at various points in the source file prior to assembly. Sequences of statements that are called very often can be replaced by macros to make the source code shorter and easier to read. Macros are an alternative to procedures for short sequences of code. The advantage of using macros is that they are fast since the actual code is duplicated every time the macro is used. However, since the code is duplicated, the executable file becomes quite large.

A macro call leads to macro expansion. During macro expansion, the macro statement is replaced by sequence of assembly statements. Every macro begins with MACRO keyword at the beginning and ends with the ENDM (end macro). Whenever a macro is called the entire code is substituted in the program where it is called.

#### **Code:**

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

```
#include<string.h>
```

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

```
char line[80],t1[10],t2[20],t3[10],FPN[20],APN[20],mname[10];
```

```
int count , v1,v2,v3,v4;
```

```
FILE *ifp;
```

```
int main()
```

```
{
```

```
    int t21,t31,index=1;
```

```
    ifp= fopen("int.txt","r");
```

```
    while(!feof(ifp))
```

```
    {
```

```
        fgets(line,179,ifp);
```

```

count = sscanf(line,"%s%s%s",t1,t2,t3);

if(strcmp("MACRO",t1)==0)

{

strcpy(mname,t2);

printf("\n macro name table");

printf("\n-----\n");

}

if(strcmp(mname,t2)==0)

{

strcpy(FPN,t3);

printf("\n\n\n**FORMAL PARAMETER NAME TABLE**");

printf("\n-----:\n");

printf("\nINDEX\t\t:MACRO NAME");

printf("\n%d\t:%s",index,FPN);

}

if(strcmp(mname,t1)==0)

{

strcpy(APN,t2);

printf("\n\n\n**ACTUAL PARAMETER NAME TABLE**");

printf("\n-----:\n");

printf("\nINDEX\t\t:MACRO NAME");

printf("\n%d\t:%s",index,APN);

}

}

}

}

```

## Input:

### Input.txt

```
1  MACRO ADDS X
2  ADD AREG BREG MEND
3  START 100
4  MOVER AREG CREG
5  ADDS X1
6  SUB AREG CREG
7  END
```

## Output:

```
PS D:\Desktop_II\SPOS Lab\Macro assignment> cd "d:\Desktop_II\SPOS Lab\Macro assignment\" ; if ($?) { gcc MACRO.c -o MACRO } ; if ($?) { .\MACRO }
```

```
macro name table
-----
```

```
**FORMAL PARAMETER NAME TABLE**
-----;
```

```
INDEX      :MACRO NAME
1          :X
```

```
**ACTUAL PARAMETER NAME TABLE**
-----;
```

```
INDEX      :MACRO NAME
1          :X1
```

**Conclusion:** Thus we have implemented macro definition and macro expansion.

PRANJAL KUMAR

ISA-I

2203263

## **LAB ASSIGNMENT 4**

**Problem Statement:** Implement the following using shell scripting

Bash programs to

- a) Find the factorial of a number.
- b) Find the greatest of the three numbers.
- c) Find a prime number.
- d) Find whether a number is palindrome.
- e) Find whether a string is palindrome.

**Theory: Shell Scripting** is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. A shell script usually has comments that describe the steps. The different operations performed by shell scripts are program execution, file manipulation and text printing. A wrapper is also a kind of shell script that creates the program environment, runs the program etc.

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a **command prompt (usually \$)**, where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

### **Types of Shells**

There are two major types of shells in Unix. These are:

#### **1. Bourne Shell**

This is the default shell for version 7 Unix. The character \$ is the default prompt for the Bourne shell. The different subcategories in this shell are Korn shell, Bourne Again shell, POSIX shell etc.

#### **2. C Shell**

This is a Unix shell and a command processor that is run in a text window. The character % is the default prompt for the C shell. File commands can also be read easily by the C shell, which is known as a script.

### **Code:**

#### **1. Factorial**

```
#!/bin/bash
```



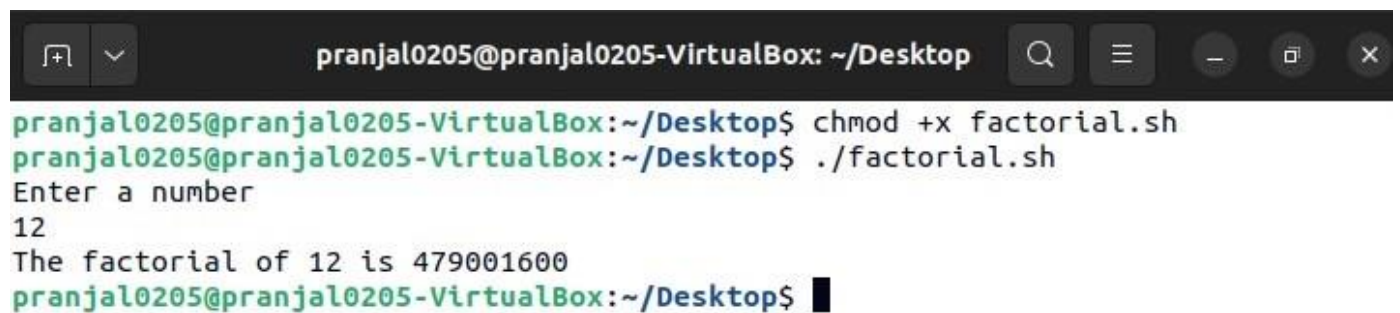
```
echo "Enter a number"
read num
input=$num

fact=1

while [ $num -gt 1 ]
do
fact=$((fact * num))
num=$((num - 1))
done

echo "The factorial of $input is $fact"
```

### Output:

A terminal window titled 'pranjal0205@pranjal0205-VirtualBox: ~/Desktop'. The prompt is 'pranjal0205@pranjal0205-VirtualBox:~/Desktop\$'. The user enters 'chmod +x factorial.sh'. The prompt is 'pranjal0205@pranjal0205-VirtualBox:~/Desktop\$'. The user enters './factorial.sh'. The prompt is 'pranjal0205@pranjal0205-VirtualBox:~/Desktop\$'. The user enters '12'. The prompt is 'pranjal0205@pranjal0205-VirtualBox:~/Desktop\$'. The output is 'The factorial of 12 is 479001600'. The prompt is 'pranjal0205@pranjal0205-VirtualBox:~/Desktop\$' followed by a cursor.

```
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x factorial.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./factorial.sh
Enter a number
12
The factorial of 12 is 479001600
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

## 2. Greatest of 3 numbers

```
#!/bin/bash
echo "Enter the first number"
read num1
echo "Enter the second number"
read num2
echo "Enter the third number"
read num3

if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
echo "$num1 is the greatest number"
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
echo "$num2 is the greatest number"
else
echo "$num3 is the greatest number"
fi
```

### Output:

```
pranjal0205@pranjal0205-VirtualBox: ~/Desktop
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x greatest.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./greatest.sh
Enter the first number
287
Enter the second number
879
Enter the third number
345
879 is the greatest number
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

### 3. Prime number

```
#!/bin/bash
echo "Enter a number"
read num
for((i=2; i<=num/2; i++)) #used num and not dollarnum
do
if [  $((num \% i))$  -eq 0 ]
then
echo "$num is not a prime number"
exit
fi
done
echo "The number is prime"
```

### Output:

```
pranjal0205@pranjal0205-VirtualBox: ~/Desktop
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x prime.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./prime.sh
Enter a number
43
The number is prime
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

#### 4. Number palindrome

```
#!/bin/bash
echo "Enter a number"
read num
s=0
rev=""

temp=$num

while [ $num -gt 0 ]
do
s=$(( $num % 10 ))
num=$(( $num / 10 ))
rev=$(( echo ${rev}${s} ))
done

if [ $temp -eq $rev ];
then
echo "The number is a palindrome"
else
echo "Number is not a palindrome"
fi
```

#### Output:

```
pranjal0205@pranjal0205-VirtualBox: ~/Desktop
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x num_palindrome.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./num_palindrome.sh
Enter a number
45654
The number is a palindrome
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

## 5. String palindrome

```
#!/bin/bash
echo "Enter a string"
read input
reverse=""

len=${#input}
for (( i=len-1; i>=0; i-- ))
do
reverse="$reverse${input:$i:1}"
done
if [ $input == $reverse ]
then
echo "$input is a palindrome"
else
echo "$input is not a palindrome"
fi
```

### Output

A terminal window titled 'pranjal0205@pranjal0205-VirtualBox: ~/Desktop' with standard window controls. The terminal shows the following commands and output:

```
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x string_palindrome.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./string_palindrome.sh
Enter a string
tenet
tenet is a palindrome
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

**Conclusion:** Thus we have executed several shell scripting programs.

## **LAB ASSIGNMENT 5**

**Problem Statement:** Implement the following using shell scripting

Menu driven program for

- a) Find the factorial of a number.
- b) Find the greatest of the three numbers.
- c) Find a prime number.
- d) Find whether a number is palindrome.
- e) Find whether a string is palindrome.

**Theory: Shell Scripting** is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. A shell script usually has comments that describe the steps. The different operations performed by shell scripts are program execution, file manipulation and text printing. A wrapper is also a kind of shell script that creates the program environment, runs the program etc.

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a **command prompt (usually \$)**, where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

### **Types of Shells**

There are two major types of shells in Unix. These are:

#### **1. Bourne Shell**

This is the default shell for version 7 Unix. The character \$ is the default prompt for the Bourne shell. The different subcategories in this shell are Korn shell, Bourne Again shell, POSIX shell etc.

#### **2. C Shell**

This is a Unix shell and a command processor that is run in a text window. The character % is the default prompt for the C shell. File commands can also be read easily by the C shell, which is known as a script.

### **Code:**

```
#!/bin/bash
```

```
#function to get the factorial
```

```
function factorial() {  
    echo "Enter a number"  
  
    read num  
  
    input=$num  
  
    fact=1  
  
    while [ $num -gt 1 ]  
    do  
        fact=$((fact * num))  
        num=$((num - 1))  
    done  
  
    echo "The factorial of $input is $fact"  
}
```

```
#function for the greatest number  
function greatest(){  
    echo "Enter the first number"  
  
    read num1  
  
    echo "Enter the second number"  
  
    read num2  
  
    echo "Enter the third number"  
  
    read num3  
  
    if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
```

then

echo "\$num1 is the greatest number"

elif [ \$num2 -gt \$num1 ] && [ \$num2 -gt \$num3 ]

then

echo "\$num2 is the greatest number"

else

echo "\$num3 is the greatest number"

fi

}

#function to check whether the number is prime or not

function prime(){

echo "Enter a number"

read num

for((i=2; i<=num/2; i++)) #used num and not dollarnum

do

if [ \$((num%i)) -eq 0 ]

then

echo "\$num is not a prime number"

exit

fi

done

echo "The number is prime"

}

#function to check if the number is a palindrome

```
function num_palindrome(){
```

```
    echo "Enter a number"
```

```
    read num
```

```
    s=0
```

```
    rev=""
```

```
    temp=$num
```

```
    while [ $num -gt 0 ]
```

```
    do
```

```
        s=$(( $num % 10 ))
```

```
        num=$(( $num / 10 ))
```

```
        rev=$( echo ${rev}${s} )
```

```
    done
```

```
    if [ $temp -eq $rev ];
```

```
    then
```

```
        echo "The number is a palindrome"
```

```
    else
```

```
        echo "Number is not a palindrome"
```

```
    fi
```

```
}
```

```
#function to check if the string is a palindrome
```

```
function string_palindrome(){
```



```
echo "Enter a string"

read input

reverse=""

len=${#input}

for (( i=$len-1; i>=0; i--))

do

    reverse="$reverse${input:$i:1}"

done

if [ $input == $reverse ]

then

    echo "$input is a palindrome"

else

    echo "$input is not a palindrome"

fi

}
```

```
echo "Select the operation to be performed"

echo "1. Find the factorial of a number"

echo "2. Find the greatest of 3 numbers"

echo "3. Find whether the number is prime or not"

echo "4. Find if the number is a palindrome"

echo "5. Find if the string is a palindrome"

echo -n "Enter your choice [1-5]: "
```

while :

do

read choice

case \$choice in

1) factorial;;

2) greatest;;

3) prime;;

4) num\_palindrome;;

5) string\_palindrome;;

\*) echo "invalid option";;

esac

echo -n "Do you want to continue [1/o] :"

read option

if [ \$option -eq 1 ]

then

echo -n "Enter your choice [1-5] : "

else

exit

fi

done

## Output:

```
pranjal0205@pranjal0205-VirtualBox: ~/Desktop
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ chmod +x new.sh
pranjal0205@pranjal0205-VirtualBox:~/Desktop$ ./new.sh
Select the operation to be performed
1. Find the factorial of a number
2. Find the greatest of 3 numbers
3. Find whether the number is prime or not
4. Find if the number is a palindrome
5. Find if the string is a palindrome
Enter your choice [1-5]: 1
Enter a number
7
The factorial of 7 is 5040
Do you want to continue [1/0] :1
Enter your choice [1-5] : 2
Enter the first number
23
Enter the second number
34
Enter the third number
45
45 is the greatest number
Do you want to continue [1/0] :1
Do you want to continue [1/0] :1
Enter your choice [1-5] : 3
Enter a number
39
39 is not a prime number
Do you want to continue [1/0] :1
Enter your choice [1-5] : 4
Enter a number
90094
Number is not a palindrome
Do you want to continue [1/0] :1
Enter your choice [1-5] : 5
Enter a string
racecar
racecar is a palindrome
Do you want to continue [1/0] :0
pranjal0205@pranjal0205-VirtualBox:~/Desktop$
```

**Conclusion:** Thus we have executed a menu driven shell scripting program.

PRANJAL KUMAR

ISA-I

2203263

## **LAB ASSIGNMENT 6**

**Problem Statement:** Process Control System Calls – Fork, exec and wait system calls along with the demonstration of zombie and orphan states.

- Application should consist of Fork-wait combination (parent with one application and child with another application) and students must demonstrate zombie and orphan states.
- Application should consist of Fork-exec combination (parent with one application and child with another application)

### **Theory:**

#### **1. fork()**

The fork() is one of the syscalls that is very special and useful in Linux/Unix systems. It is used by processes to create the processes that are copies of themselves. With the help of such system calls, the child process can be created by the parent process. Until the child process is executed completely, the parent process is suspended.

Some of the important points on fork() are as follows:-

- The parent will get the child process ID with non-zero value.
- Zero Value is returned to the child.
- If there will be any system or hardware errors while creating the child, -1 is returned to the fork().
- With the unique process ID obtained by the child process, it does not match the ID of any existing process group.

#### **2. wait()**

As in the case of a fork, child processes are created and get executed but the parent process is suspended until the child process executes. In this case, a wait() system call is activated automatically due to the suspension of the parent process. After the child process ends the execution, the parent process gains control again.

#### **3. exec()**

The exec() is such a system call that runs by replacing the current process image with the new process image. However, the original process remains as a new process but the new process replaces the head data, stack data, etc. It runs the program from the entry point by loading the program into the current process space.

4. A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.
5. Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally. An intentionally orphaned process runs in the background without any manual support. This is

usually done to start an indefinitely running service or to complete a long-running job without user attention. An unintentionally orphaned process is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.

### **Code:**

```
#include <stdio.h>
#include<sys/types.h> //fork, sleep, getpid, getppid
#include<sys/wait.h> //system call - wait
#include<stdlib.h>
#include<unistd.h>

int main()
{
    pid_t cpid; //Declaring a variable cpid with the pid_t data type
    int *status=NULL; //Intilizing a pointer var status to NULL

    cpid = fork(); //The process fork_wait creates a new process --clone

    if( cpid == 0 ) { //CHILD PROCESS as it is not creating any new process
        printf("\n***** This is child process *****\n "); //write sys call

        printf("\n\t Process id is : %d", getpid());
        printf("\n\t Parent's process id is : %d", getppid());
        sleep(15);

        printf("\n*****Child process terminates *****\n");
    }
    else { /*Parent process waiting for child process, to complete the task*/

        printf("\n\t My process id is : %d", getpid());
        printf("\n\t My Parent process id is : %d", getppid());
        cpid = wait(status); //Forceful wait; that collects the exit status of child process with
cpid

        printf("\n\n\t Parent process collected the exit status of child process with PID
%d\n\n", cpid);
    } //end of if-else

    return 0;
} //end of main
```

## **Output:**

```
pranjal@DESKTOP-GOI233F:/mnt/c/Users/PRAMOD KUMAR/Downloads$ gcc fork_wait.c
pranjal@DESKTOP-GOI233F:/mnt/c/Users/PRAMOD KUMAR/Downloads$ ./a.out
```

```
My process id is : 95
```

```
***** This is child process *****
```

```
Process id is : 96
```

```
Parent's process id is : 95
```

```
*****Child process terminates *****
```

```
My Parent process id is : 9
```

```
Parent process collected the exit status of child process with PID 96
```

```
pranjal@DESKTOP-GOI233F:/mnt/c/Users/PRAMOD KUMAR/Downloads$ █
```

**Conclusion:** Thus we have demonstrated system calls and zombie and orphan states.

## **LAB ASSIGNMENT 7**

**Problem Statement:** Simulate the following scheduling algorithms using C/Python/Java language.

- FCFS (Non-preemptive by default)
- SRTN (Preemptive version of SJF)
- Priority (Non Preemptive)
- Round Robin (Preemptive by default)

**Theory:** CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer. Whenever the CPU becomes idle, the operating system must select one of the processes in the line ready for launch. The selection process is done by a temporary (CPU) scheduler. The Scheduler selects between memory processes ready to launch and assigns the CPU to one of them.

There are mainly two types of scheduling methods:

- **Preemptive Scheduling:** Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- **Non-Preemptive Scheduling:** Non-Preemptive scheduling is used when a process terminates, or when a process switches from running state to waiting state.

### **1. First Come First Serve:**

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

### **2. Shortest Time Remaining Next:**

A Shortest remaining Time Next scheduling algorithm is also referred as preemptive SJF scheduling algorithm. When a new process arrives at ready queue while one process is still executing then SRTN algorithm is performed to decide which process will execute next. This algorithm compare CPU burst time of newly arrived process with remaining (left) CPU burst time of currently executing process. If CPU burst time of new process is less than remaining time of current process then SRTN algorithm preempts current process execution and starts executing new process.

### **3. Round robin:**

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

### **Code:**

1. C program for implementation of FCFS

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

```
// Function to find the waiting time for all processes
```

```
void findWaitingTime(int processes[], int n,
```

```
    int bt[], int wt[])
```

```
{
```

```
    // waiting time for first process is 0
```

```
    wt[0] = 0;
```

```
    // calculating waiting time
```

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

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

```
}
```

```
// Function to calculate turn around time
```

```
void findTurnAroundTime( int processes[], int n,
```

```
    int bt[], int wt[], int tat[])
```

```
{
```

```
    // calculating turnaround time by adding
```

```
    // bt[i] + wt[i]
```

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

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

```
}
```

```
//Function to calculate average time
```

```
void findavgTime( int processes[], int n, int bt[])
```

```
{
```

```
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
```

```
    //Function to find waiting time of all processes
```



```
findWaitingTime(processes, n, bt, wt);
```

```
//Function to find turn around time for all processes
```

```
findTurnAroundTime(processes, n, bt, wt, tat);
```

```
//Display processes along with all details
```

```
printf("Processes  Burst time  Waiting time  Turn around time\n");
```

```
// Calculate total waiting time and total turn around time
```

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

```
{
```

```
    total_wt = total_wt + wt[i];
```

```
    total_tat = total_tat + tat[i];
```

```
    printf("  %d ",(i+1));
```

```
    printf("    %d ", bt[i] );
```

```
    printf("    %d",wt[i] );
```

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

```
}
```

```
int s=(float)total_wt / (float)n;
```

```
int t=(float)total_tat / (float)n;
```

```
printf("Average waiting time = %d",s);
```

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

```
printf("Average turn around time = %d ",t);
```

```
}
```

```
// Driver code
```

```
int main()
```

```
{
```

```

//process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[o];


//Burst time of all processes

int burst_time[] = {11, 4, 7};


findavgTime(processes, n, burst_time);

return o;

}

```

## Output:

```

PS D:\Desktop_II\MIT-SOE\SPOS\leftover experiments> cd "d:\Desktop_II\MIT-SOE\SPOS\leftover experiments\" ; if ($?) { gcc fofs.c -o fofs } ; if ($?) { .\fofs }
Processes  Burst time  Waiting time  Turn around time
1          11          0           11
2           4          11          15
3           7          15          22
Average waiting time = 8
Average turn around time = 16
PS D:\Desktop_II\MIT-SOE\SPOS\leftover experiments>

```

## 2. C program for implementation of SRTN

```

#include <stdio.h>

int main()

{

int A[100][4]; // Matrix for storing Process Id, Burst

// Time, Average Waiting Time & Average

// Turn Around Time.

int i, j, n, total = 0, index, temp;

float avg_wt, avg_tat;

```

```

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

// User Input Burst Time and allotting Process Id.

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

    printf("P%d: ", i + 1);

    scanf("%d", &A[i][1]);

    A[i][0] = i + 1;

}

// Sorting process according to their Burst Time.

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

    index = i;

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

        if (A[j][1] < A[index][1])

            index = j;

    temp = A[i][1];

    A[i][1] = A[index][1];

    A[index][1] = temp;

    temp = A[i][0];

    A[i][0] = A[index][0];

    A[index][0] = temp;

}

A[0][2] = 0;

// Calculation of Waiting Times

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

    A[i][2] = 0;

```

```

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

        A[i][2] += A[j][1];

    total += A[i][2];

}

avg_wt = (float)total / n;

total = 0;

printf("P   BT   WT   TAT\n");

// Calculation of Turn Around Time and printing the

// data.

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

    A[i][3] = A[i][1] + A[i][2];

    total += A[i][3];

    printf("P%d   %d   %d   %d\n", A[i][0],

        A[i][1], A[i][2], A[i][3]);

}

avg_tat = (float)total / n;

printf("Average Waiting Time= %f", avg_wt);

printf("\nAverage Turnaround Time= %f", avg_tat);

}

```

## Output:

```

PS D:\Desktop_II\MIT-SOE\SPOS\leftover experiments> cd "d:\Desktop_II\MIT-SOE\SPOS\leftover experiments\" ; if ($?) { gcc srtn.c -o srtn } ; if ($?) { .\srtn
n }
Enter number of process: 4
Enter Burst Time:
P1: 8
P2: 4
P3: 9
P4: 5
P   BT   WT   TAT
P2   4   0   4
P4   5   4   9
P1   8   9  17
P3   9  17  26
Average Waiting Time= 7.500000
Average Turnaround Time= 14.000000

```

### 3. C program for implementation of RR

```
#include<stdio.h>

#include<conio.h>

void main()

{

    // initialize the variable name

    int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

    float avg_wt, avg_tat;

    printf(" Total number of process in the system: ");

    scanf("%d", &NOP);

    y = NOP; // Assign the number of process to variable y


    // Use for loop to enter the details of the process like Arrival time and the Burst Time

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

    {

        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

        printf(" Arrival time is: \t"); // Accept arrival time

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

        printf(" \nBurst time is: \t"); // Accept the Burst time

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

        temp[i] = bt[i]; // store the burst time in temp array

    }

    // Accept the Time quanat

    printf("Enter the Time Quantum for the process: \t");

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

```

// Display the process No, burst time, Turn Around Time and the waiting time

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0) // define the conditions

{

    sum = sum + temp[i];

    temp[i] = 0;

    count=1;

}

else if(temp[i] > 0)

{

    temp[i] = temp[i] - quant;

    sum = sum + quant;

}

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

{

    y--; //decrement the process no.

    printf("\nProcess No[%d] \t\t %d\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

    wt = wt+sum-at[i]-bt[i];

    tat = tat+sum-at[i];

    count =0;

}

if(i==NOP-1)

{

    i=0;

}

```

```
else if(at[i+1]<=sum)

{

    i++;

}

else

{

    i=0;

}

}

// represents the average waiting time and Turn Around time

avg_wt = wt * 1.0/NOP;

avg_tat = tat * 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg_wt);

printf("\n Average Waiting Time: \t%f", avg_tat);

getch();

}
```

## Output:

```
PS D:\Desktop_II\MIT-SOE\SPOS\leftover experiments> cd "d:\Desktop_II\MIT-SOE\SPOS\leftover experiments\" ; if ($?) { gcc rr.c -o rr } ; if ($?) { .\rr }
Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is:      0

Burst time is:  5

Enter the Arrival and Burst time of the Process[2]
Arrival time is:      1

Burst time is:  4

Enter the Arrival and Burst time of the Process[3]
Arrival time is:      2

Burst time is:  2

Enter the Arrival and Burst time of the Process[4]
Arrival time is:      4

Burst time is:  1
Enter the Time Quantum for the process:      2

Process No      Burst Time      TAT      Waiting Time
Process No[3]    2              4          2
Process No[4]    1              3          2
Process No[2]    4             10          6
Process No[1]    5             12          7
Average Turn Around Time:      4.250000
Average Waiting Time:  7.250000
```

**Conclusion:** Thus we have simulated various CPU Scheduling algorithms.

PRANJAL KUMAR

ISA-1

2203263



## LAB ASSIGNMENT 8

**Problem Statement:** A Simulation of Banker's algorithm using C/Python/Java.

**Theory:** Banker's algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.

When working with a banker's algorithm, it requests to know about three things:

- How much each process can request for each resource in the system. It is denoted by the [MAX] request.
- How much each process is currently holding each resource in a system. It is denoted by the [ALLOCATED] resource.
- It represents the number of each resource currently available in the system. It is denoted by the [AVAILABLE] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose  $n$  is the number of processes, and  $m$  is the number of each type of resource used in a computer system.

1. Available: It is an array of length ' $m$ ' that defines each type of resource available in the system. When  $\text{Available}[j] = K$ , means that ' $K$ ' instances of Resources type  $R[j]$  are available in the system.
2. Max: It is a  $[n \times m]$  matrix that indicates each process  $P[i]$  can store the maximum number of resources  $R[j]$  (each type) in a system.
3. Allocation: It is a matrix of  $m \times n$  orders that indicates the type of resources currently allocated to each process in the system. When  $\text{Allocation}[i, j] = K$ , it means that process  $P[i]$  is currently allocated  $K$  instances of Resources type  $R[j]$  in the system.
4. Need: It is an  $M \times N$  matrix sequence representing the number of remaining resources for each process. When the  $\text{Need}[i][j] = k$ , then process  $P[i]$  may require  $K$  more instances of resources type  $R_j$  to complete the assigned work.

$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$ .

Finish: It is the vector of the order  $m$ . It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

**Code:**

```
#include<stdio.h>

int max[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n,r;

void input();

void show();

void cal();

int main()

{

    int i,j;

    printf("**** Banker's Algo ****\n");

    input();

    cal();

    show();

    return 0;

}

void input()

{

    int i,j;

    printf("Enter the no of Processes\t");

    scanf("%d",&n);

    printf("Enter the no of resources instances\t");

    scanf("%d",&r);

    printf("Enter the Max Matrix\n");
```

```

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

{

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

{

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

}

}

printf("Enter the Allocation Matrix\n");

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

{

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

{

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

}

}

printf("Enter the available Resources\n");

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

{

scanf("%d",&avail[j]);

}

}

void show()

{

int i,j;

printf("Process\t\tAllocation\tMax\t\tAvailable\t");

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

{

```

```

printf("\nP%d\t\t",i+1);

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

{

printf("%d ",alloc[i][j]);

}

printf("\t\t");

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

{

printf("%d ",max[i][j]);

}

printf("\t\t\t");

if(i==0)

{

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

printf("%d ",avail[j]);

}

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

{

printf("%d ",need[i][j]);

}

printf("\t");

}

}

void cal()

{

int finish[100],temp,flag=1,k,c1=0;

int safe[100];

```

```
int i,j;

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

{

finish[i]=0;

}

//find need matrix

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

{

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

{

need[i][j]=max[i][j]-alloc[i][j];

}

}

printf("\n");

while(flag)

{

flag=0;

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

{

int c=0;

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

{

if((finish[i]==0)&&(need[i][j]<=avail[j]))

{

c++;

if(c==r)

{
```

```
for(k=0;k<r;k++)  
  
{  
  
    avail[k]+=alloc[i][j];  
  
    finish[i]=1;  
  
    flag=1;  
  
}  
  
    printf("P%d->",i);  
  
if(finish[i]==1)  
  
{  
  
    i=n;  
  
}  
  
}  
  
}  
  
}  
  
}  
  
}  
  
for(i=0;i<n;i++)  
  
{  
  
    if(finish[i]==1)  
  
{  
  
        c1++;  
  
    }  
  
    else  
  
{  
  
        printf("P%d->",i);  
  
    }  
  
}
```

```

}

if(c1==n)

{

printf("\n The system is in safe state\n");

}

else

{

printf("\n Process are in dead lock");

printf("\n System is in unsafe state");

}

}

```

### Output:

```

PS D:\Desktop_II\MIT-SOE\SPOS\leftover experiments> cd "d:\Desktop_II\MIT-SOE\SPOS\leftover experiments\" ; if ($?) { gcc bankers.c -o bankers } ; if ($?) {
.\bankers }
**** Banker's Algo ****
Enter the no of Processes      5
Enter the no of resources instances    3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available Resources
3 3 2

P1->P3->P4->P2->P0->
The system is in safe state
Process      Allocation      Max      Available
P1           0 1 0           7 5 3           8 8 7 7 4 3
P2           2 0 0           3 2 2           1 2 2
P3           3 0 2           9 0 2           6 0 0
P4           2 1 1           2 2 2           0 1 1
P5           0 0 2           4 3 3           4 3 1

```

**Conclusion:** Thus we have implemented the Banker's algorithm in C language.

## LAB ASSIGNMENT 9

**Problem Statement:** Implement the following page replacement algorithms

- FIFO
- LRU

**Theory:** Page replacement is needed in the operating systems that use virtual memory using Demand Paging. As we know that in Demand paging, only a set of pages of a process is loaded into the memory. This is done so that we can have more processes in the memory at the same time.

When a page that is residing in virtual memory is requested by a process for its execution, the Operating System needs to decide which page will be replaced by this requested page. This process is known as page replacement and is a vital component in virtual memory management.

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Some page replacement algorithms are as follows:

1. **First In First Out (FIFO):** This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.
2. **Least Recently Used (LRU):** The least recently used page replacement algorithm keeps the track of usage of pages over a period of time. This algorithm works on the basis of the principle of locality of a reference which states that a program has a tendency to access the same set of memory locations repetitively over a short period of time. So pages that have been used heavily in the past are most likely to be used heavily in the future also. In this algorithm, when a page fault occurs, then the page that has not been used for the longest duration of time is replaced by the newly requested page.

**Code:**

### 1) Program for FIFO Page replacement algorithm

```
#include <stdio.h>

int main()
{
    int i, j, n, a[50], frame[10], no, k, avail, count = 0;

    printf("\n ENTER THE NUMBER OF PAGES:\n");

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



```

printf("\n ENTER THE PAGE NUMBER :\n");

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

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

printf("\n ENTER THE NUMBER OF FRAMES :");

scanf("%d", &no);

for (i = 0; i < no; i++)

    frame[i] = -1;

j = 0;

printf("\tref string\t page frames\n");

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

{

    printf("%d\t\t", a[i]);

    avail = 0;

    for (k = 0; k < no; k++)

        if (frame[k] == a[i])

            avail = 1;

    if (avail == 0)

    {

        frame[j] = a[i];

        j = (j + 1) % no;

        count++;

        for (k = 0; k < no; k++)

            printf("%d\t", frame[k]);

    }

    printf("\n");

}

printf("Page Fault Is %d", count);

```

```

return o;
}

```

### **Output:**

```

ENTER THE NUMBER OF PAGES:
15
ENTER THE PAGE NUMBER :
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2
ENTER THE NUMBER OF FRAMES :3
ref string    page frames
7           7   -1  -1
0           7   0  -1
1           7   0   1
2           2   0   1
0
3           2   3   1
0           2   3   0
4           4   3   0
2           4   2   0
3           4   2   3
0           0   2   3
3
2
1           0   1   3
2           0   1   2
Page Fault Is 12

```

### **Input:**

#### **2) Program for LRU Page replacement algorithm**

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

```
void main()
```

```
{
```

```
    int q[20], p[50], c = 0, c1, d, f, i, j, k = 0, n, r, t, b[20], c2[20];
```

```
    printf("Enter no of pages:");
```

```

scanf("%d", &n);

printf("Enter the reference string:");

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

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

printf("Enter no of frames:");

scanf("%d", &f);

q[k] = p[k];

printf("\n\t%d\n", q[k]);

c++;

k++;

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

    c1 = 0;

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

    {

        if (p[i] != q[j])

            c1++;

    }

    if (c1 == f)

    {

        c++;

        if (k < f)

        {

            q[k] = p[i];

            k++;

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

                printf("\t%d", q[j]);

```

```

        printf("\n");
    }
else
{
    for (r = o; r < f; r++)
    {
        c2[r] = o;
        for (j = i - 1; j < n; j--)
        {
            if (q[r] != p[j])
                c2[r]++;
            else
                break;
        }
    }
    for (r = o; r < f; r++)
        b[r] = c2[r];
    for (r = o; r < f; r++)
    {
        for (j = r; j < f; j++)
        {
            if (b[r] < b[j])
            {
                t = b[r];
                b[r] = b[j];
                b[j] = t;
            }
        }
    }
}

```

```
        }  
    }  
    for (r = o; r < f; r++)  
    {  
        if (c2[r] == b[o])  
            q[r] = p[i];  
        printf("\t%d", q[r]);  
    }  
    printf("\n");  
}  
}  
  
}  
  
printf("\nThe no of page faults is %d", c);
```

### **Output:**

```
Enter no of pages:10
Enter the reference string:7 5 9 4 3 7 9 6 2 1
Enter no of frames:3
7
  7  5
  7  5  9
  4  5  9
  4  3  9
  4  3  7
  9  3  7
  9  6  7
  9  6  2
  1  6  2

The no of page faults is 10
```

**Conclusion:** Thus we have implemented FIFO and LRU page Replacement algorithms.

## **LAB ASSIGNMENT 10**

**Problem Statement:** Implement matrix multiplication using multithreading with pthread library

**Theory:** A thread is a path which is followed during a program's execution. Majority of programs written now a days run as a single thread. Lets say, for example a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

Multitasking is of two types: Processor based and thread based. Processor based multitasking is totally managed by the OS, however multitasking through multithreading can be controlled by the programmer to some extent.

The concept of multi-threading needs proper understanding of these two terms – a process and a thread. A process is a program being executed. A process can be further divided into independent units known as threads. A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user. The extremely fast processing speeds of today's microprocessors make multithreading possible. Even though the processor executes only one instruction at a time, threads from multiple programs are executed so fast that it appears multiple programs are executing concurrently.

Each CPU cycle executes a single thread that links to all other threads in its stream. This synchronization process occurs so quickly that it appears all the streams are executing at the same time. This can be described as a multithreaded program, as it can execute many threads while processing. Each thread contains information about how it relates to the overall program. While in the asynchronous processing stream, some threads are executed while others wait for their turn. Multithreading requires programmers to pay careful attention to prevent race conditions and deadlock.

### **Code:**

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

```
#include <pthread.h>
```

```
#include <unistd.h>
```

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

```
#define MAX 4
```

```
// Each thread computes single element in the resultant matrix
```

```

void *mult(void *arg)
{
    int *data = (int *)arg;

    int k = 0, i = 0;

    int x = data[0];
    for (i = 1; i <= x; i++)
        k += data[i] * data[i + x];

    int *p = (int *)malloc(sizeof(int));

    *p = k;

    // Used to terminate a thread and the return value is passed as a pointer
    pthread_exit(p);
}

```

// Driver code

```

int main()
{

    int matA[MAX][MAX];

    int matB[MAX][MAX];

    int r1 = MAX, c1 = MAX, r2 = MAX, c2 = MAX, i, j, k;

    // Generating random values in matA

    for (i = 0; i < r1; i++)

```



```

        for (j = 0; j < c1; j++)

            matA[i][j] = rand() % 10;


// Generating random values in matB

for (i = 0; i < r1; i++)

    for (j = 0; j < c1; j++)

        matB[i][j] = rand() % 10;


// Displaying matA

printf("Matrix A : \n");

for (i = 0; i < r1; i++)

{

    for (j = 0; j < c1; j++)

        printf("%d ", matA[i][j]);

    printf("\n");

}

printf("\n");

// Displaying matB

printf("Matrix B : \n");

for (i = 0; i < r2; i++)

{

    for (j = 0; j < c2; j++)

        printf("%d ", matB[i][j]);

    printf("\n");

}


int max = r1 * c2;

```

```

// declaring array of threads of size r1*c2

pthread_t *threads;

threads = (pthread_t *)malloc(max * sizeof(pthread_t));


int count = 0;

int *data = NULL;

for (i = 0; i < r1; i++)
    for (j = 0; j < c2; j++)
    {

        // storing row and column elements in data

        data = (int *)malloc((20) * sizeof(int));

        data[0] = c1;


        for (k = 0; k < c1; k++)

            data[k + 1] = matA[i][k];


        for (k = 0; k < r2; k++)

            data[k + c1 + 1] = matB[k][j];


        // creating threads

        pthread_create(&threads[count++], NULL,

                        mult, (void *) (data));

    }


printf("\nRESULTANT MATRIX IS :- \n");

```

```

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

        void *k;

        // Joining all threads and collecting return value

        pthread_join(threads[i], &k);

        int *p = (int *)k;

        printf("%d ", *p);

        if ((i + 1) % c2 == 0)

            printf("\n");

    }

    return 0;

}

```

## **Output:**

```

PS D:\Desktop_II\MIT-SOE\SPOS\Lab> cd "d:\Desktop_II\MIT-SOE\SPOS\Lab\" ; if ($?) { gcc matrix_multithreading.c -o matrix_multithreading } ; if ($?) { .\mat
rix_multithreading }
Matrix A :
1 7 4 0
9 4 8 8
2 4 5 5
1 7 1 1

Matrix B :
5 2 7 6
1 4 2 3
2 2 1 6
8 5 7 6

RESULTANT MATRIX IS :-
20 38 25 51
129 90 135 162
64 55 62 84
22 37 29 39
PS D:\Desktop_II\MIT-SOE\SPOS\Lab>

```

**Conclusion:** Thus we have implemented matrix multiplication using the pthread library.