**EXP1: Design suitable data structures and implement pass-I of a two-pass assembler. Implementation should consist of a few instructions from each category and few assembler directives.**

**THEORY:**

**Assembly Language**

Assembly Language is a low-level programming language that is used for a computer or other programmable devices. There is a very strong correspondence between the assembly language and the architecture’s machine code instructions.

**Types of Assembler**

Assemblers generate instruction. On the basis of a number of phases used to convert to machine code, assemblers have two types:

**1. One-Pass Assembler**

These assemblers perform the whole conversion of assembly code to machine code in one go.

**2. Multi-Pass/Two-Pass Assembler**

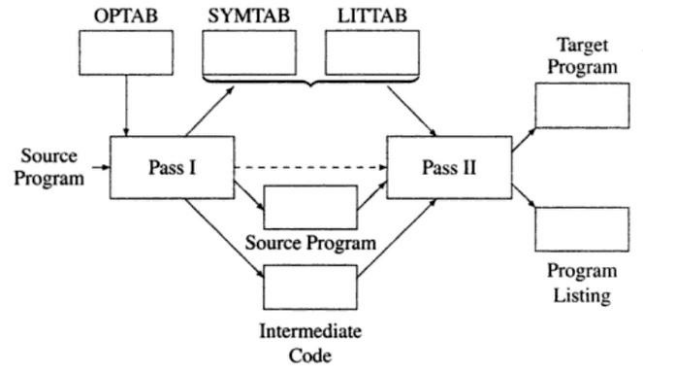
These assemblers first process the assembly code and store values in the opcode table and symbol table. And then in the second step, they generate the machine code using these tables.

***a) Pass 1***

It is an assembler that generally generates the object code directly in memory for immediate execution. It parses through your source code only once and you are done.The one-pass assembler cannot resolve forward references of data symbols. It requires all data symbols to be defined prior to being used.

***b) Pass 2***

A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass. If a data symbol depends on another and this another depends on yet another, the assembler resolved this recursively.

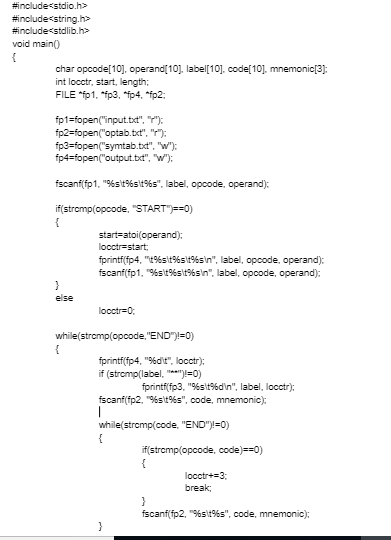
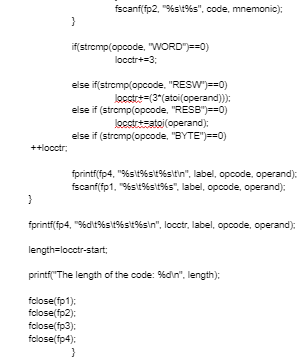


**Working of Pass-1 :**

The primary function performed by the analysis phase is the building of the symbol table. For this purpose it must determine the addresses with which the symbol names used in a program are associated. It is possible to determine some address directly, e.g. the address of the first instruction in the program, however others must be inferred.

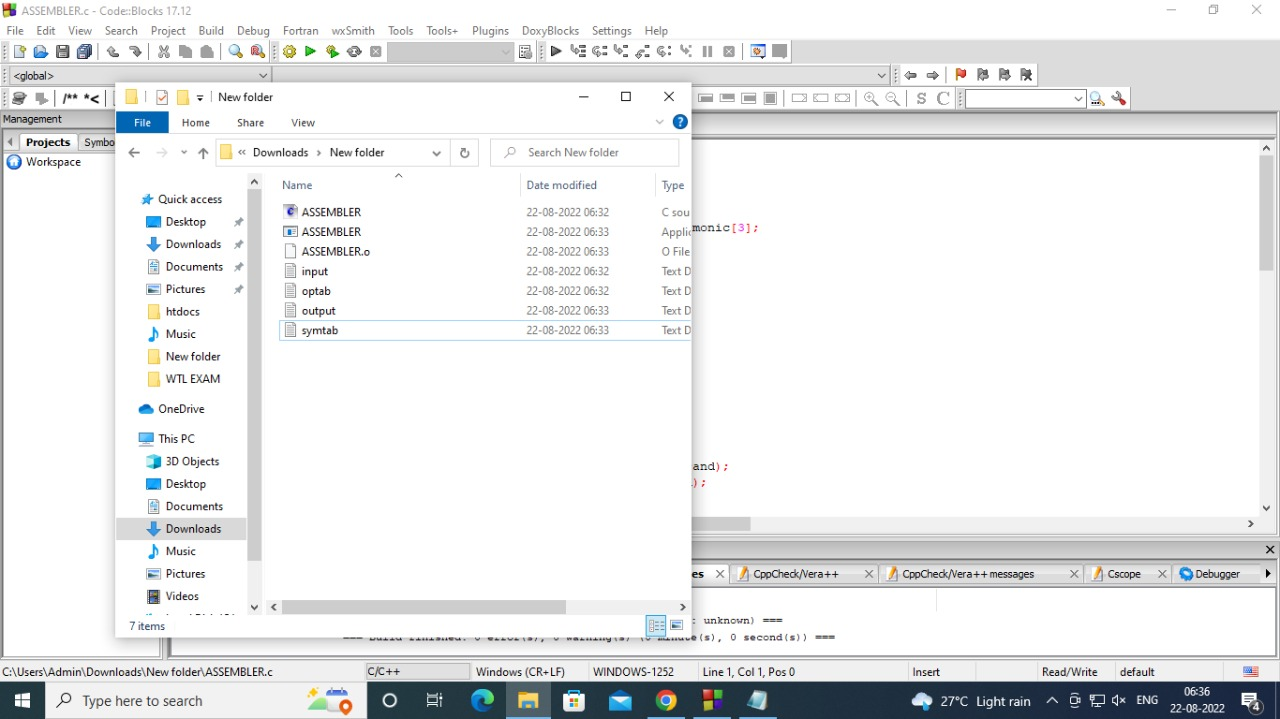
*The location counter (LC)* is always made to contain the address of the next memory word in the target program, initialized to the constant. Whenever the analysis phase sees a label in an assembly statement, it enters the label and the contents of LC in a new entry of the *symbol table*. It then finds the number of memory words required by the assembly statement and updates; the LC contents. This ensures that LC points to the next memory word in the target program even when machine instructions have different lengths and DS/DC statements reserve different amounts of memory. To update the contents of LC, the analysis phase needs to know lengths of different instructions. This information simply depends on the assembly language hence the *mnemonics table* can be extended to include this information in a new field called length. We refer to the processing involved in maintaining the location counter as LC processing.

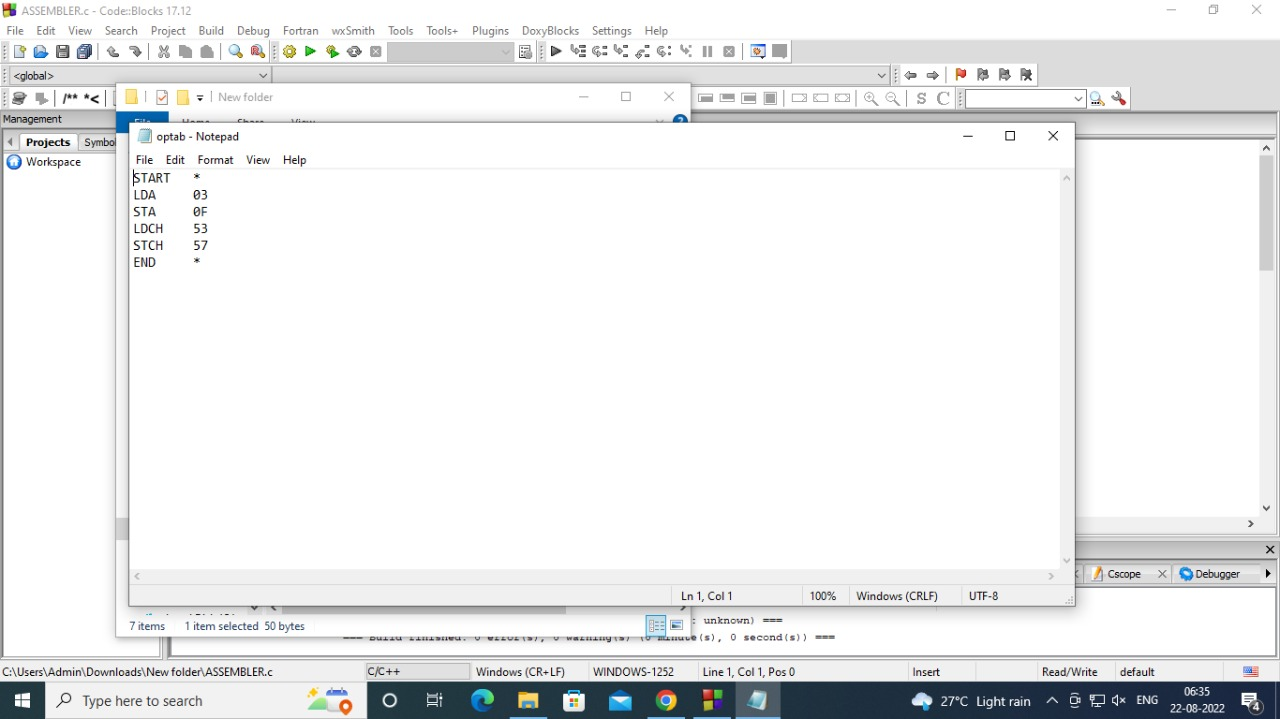
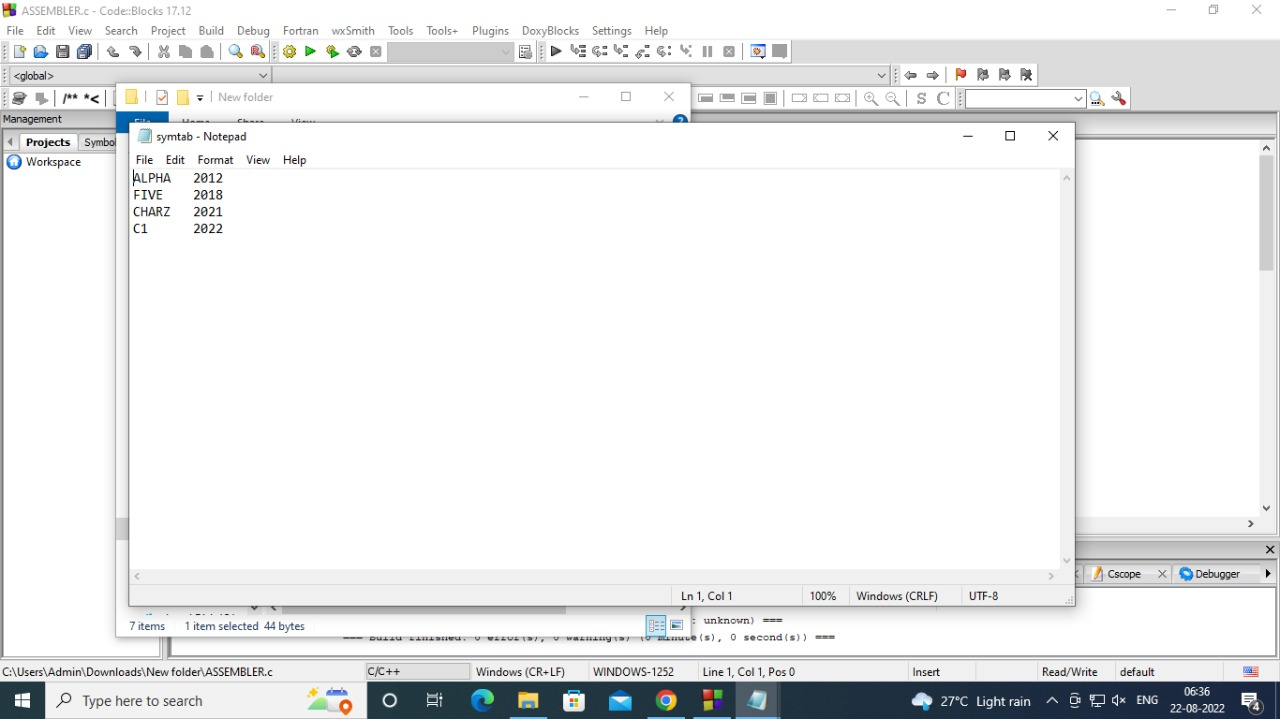
**PROGRAM CODE:**

** **

**PROGRAM OUTPUT:**

After running the code, we observe that two new files *symtab* and *optab* are generated

****

** **

**CONCLUSION:**

We have succesfully designed suitable data structures and implemented pass-I of a two-pass assembler.

**EXP2:Design suitable data structures and implement pass-II of a two-pass assembler. Implementation should consist of a few instructions from each category and few assembler directives.**

**THEORY:**

**Assembly Language**

Assembly Language is a low-level programming language that is used for a computer or other programmable devices. There is a very strong correspondence between the assembly language and the architecture’s machine code instructions.

**Types of Assembler**

Assemblers generate instruction. On the basis of a number of phases used to convert to machine code, assemblers have two types:

**1. One-Pass Assembler**

These assemblers perform the whole conversion of assembly code to machine code in one go.

**2. Multi-Pass/Two-Pass Assembler**

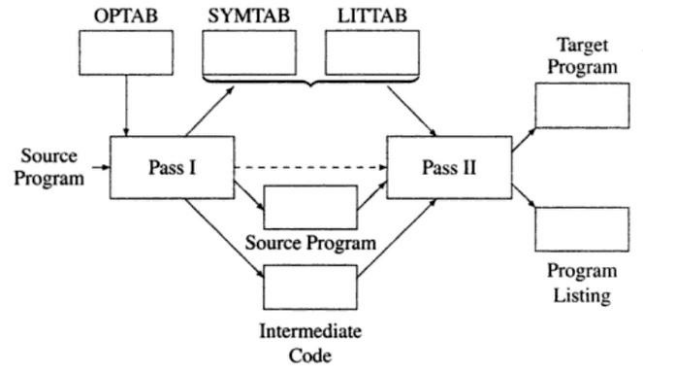
These assemblers first process the assembly code and store values in the opcode table and symbol table. And then in the second step, they generate the machine code using these tables.

***a) Pass 1***

It is an assembler that generally generates the object code directly in memory for immediate execution. It parses through your source code only once and you are done.The one-pass assembler cannot resolve forward references of data symbols. It requires all data symbols to be defined prior to being used.

***b) Pass 2***

A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass. If a data symbol depends on another and this another depends on yet another, the assembler resolved this recursively.



**Working of Pass-2:**

The two pass assembler performs two passes over the source program.

In the first pass, all it does is looks for label definitions and introduces them in the *Symbol Table*, no instructions are assembled. To assign addresses to labels, the assembler maintains a *Location Counter (LC).*

In the second pass the instructions are again read and are assembled using the symbol table and generates machine code for that instruction. In this way, the entire machine code program is created. For most instructions this process works fine, for example for instructions that only reference registers, the assembler can compute the machine code easily, since the assembler knows where the registers are.

Various Data Structures required by pass-2:

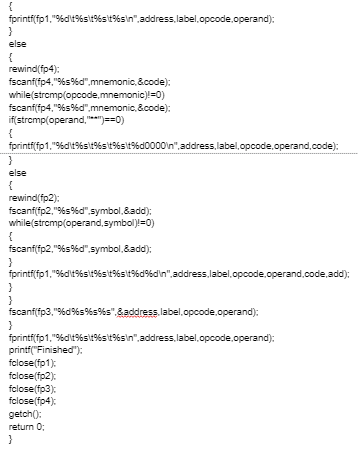
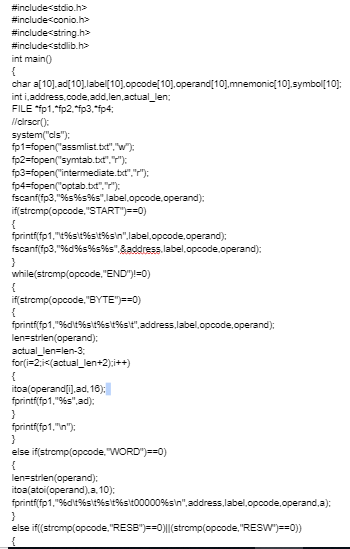
1. MOT table(machine opcode table)

2. POT table(pseudo opcode table)

3. Base table(storing value of base register)

4. LC ( location counter)

**PROGRAM CODE:**

****

**PROGRAM OUTPUT:**

****

**CONCLUSION:**

We have succesfully designed suitable data structures and implemented pass-II of a two-pass assembler.

**EXP3:** **Implement the following using shell scripting programs for**

**a) Find the factorial of a no.**

**b) Find greatest of three numbers**

**c) Find a prime no**

**d) Find whether a number is palindrome**

**e) Find whether a string is palindrome**

**THEORY:**

**Shell Scripting:**

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.

**Advantages of Shell Script**

Some of the advantages of shell script are −

* The commands and syntax of the shell script are the same as that entered at the command line. Because of this, there is no need to switch to a completely different syntax.
* Shell scripting can be used by users that are not experts to modify and tailor the behaviour of their programs according to their requirements.

**Disadvantages of Shell Script**

Some of the disadvantages of shell script are −

* There may be errors in shell scripting that prove to be quite costly.
* The programs in shell script are quite slow while executing and a new process is required for every shell command executed.
* Different platforms in shell scripting may also have compatibility problems.

**Executing the file**

Now that we have seen what a typical shell script looks like, let’s look at how to execute the file.

1. Save the file with a .sh extension
2. To execute the file, first we need to give it execute permissions.

chmod +x filepath/filename.sh

1. To execute the file, we can do it in the following ways
   * If you are using a GUI file navigation system, right-click on the file and click on run or execute.
   * If you are using the terminal, ./filename.sh will execute the script. (Make sure you are in the correct directory!)

**Commands used:**

**echo "Welcome!! Please Enter Your Name"**

The echo keyword is used to output the strings that it is being passed as arguments. It outputs onto the stdout.

**read name**

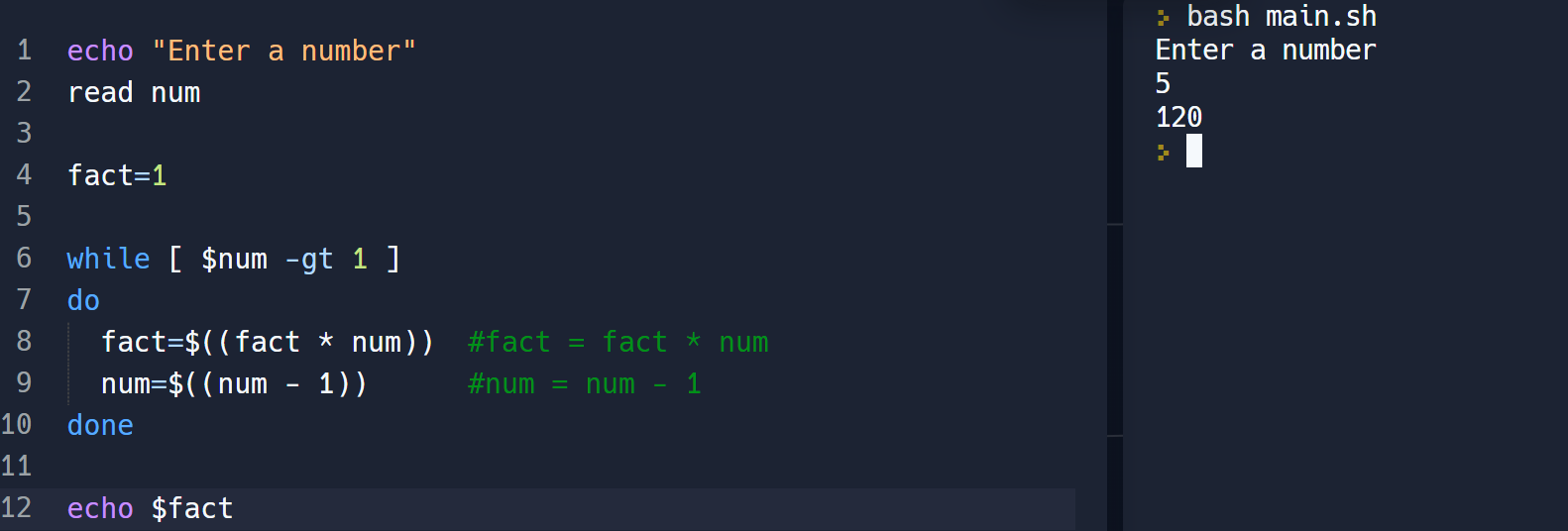
The read command is used to receive input while running the script. When we wrote read name it initialized a variable named name and stored the input in it.

**echo "Hello $name"**

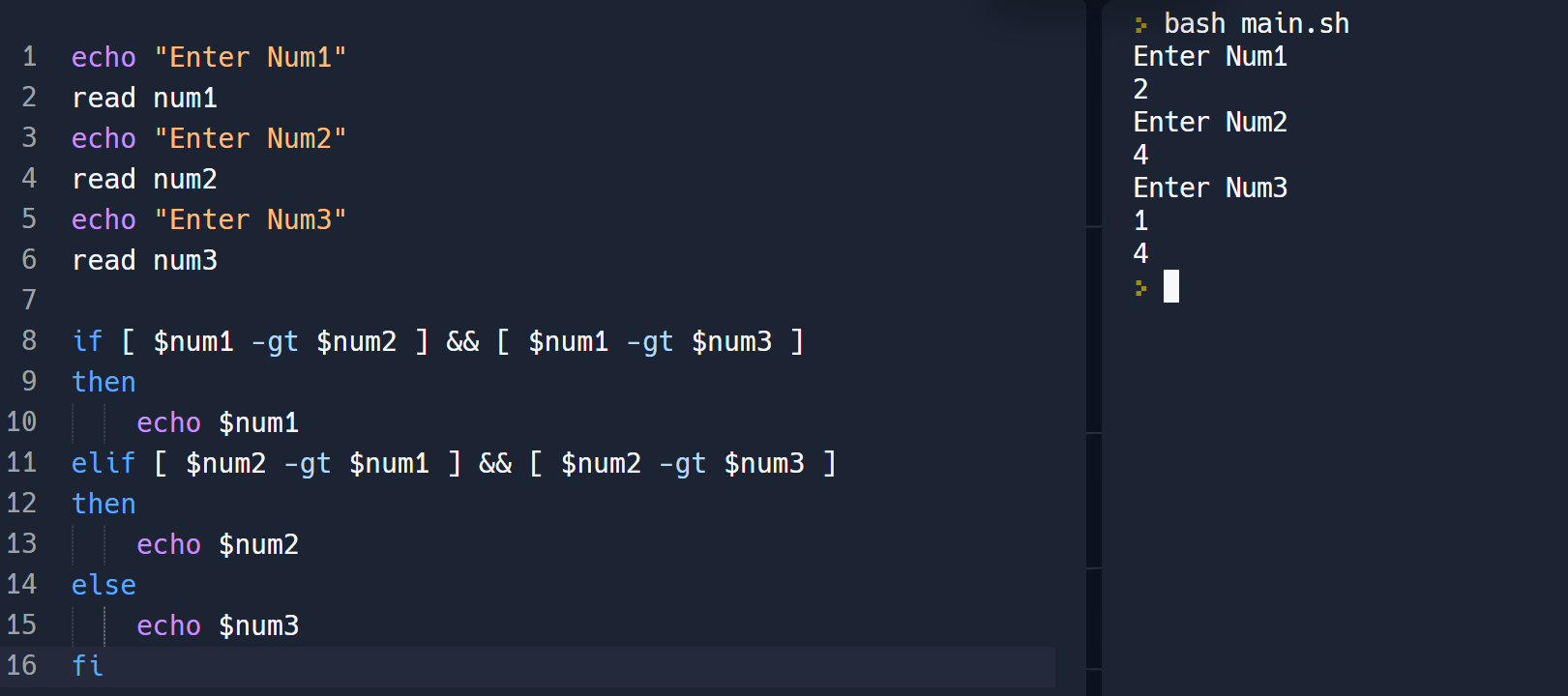
The $ symbol is used to print the **value** of a variable. In the above example, it prints whatever input was given in the previous step.

**PROGRAM CODE:**

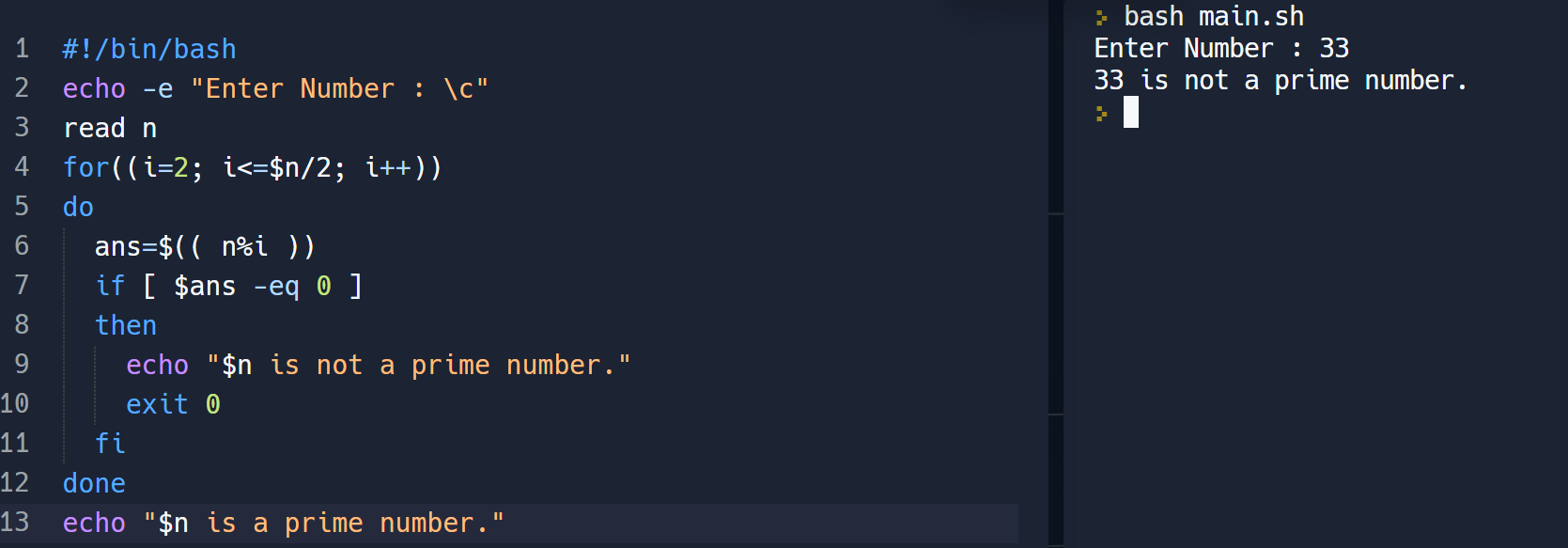
**a) Find the factorial of a no.**

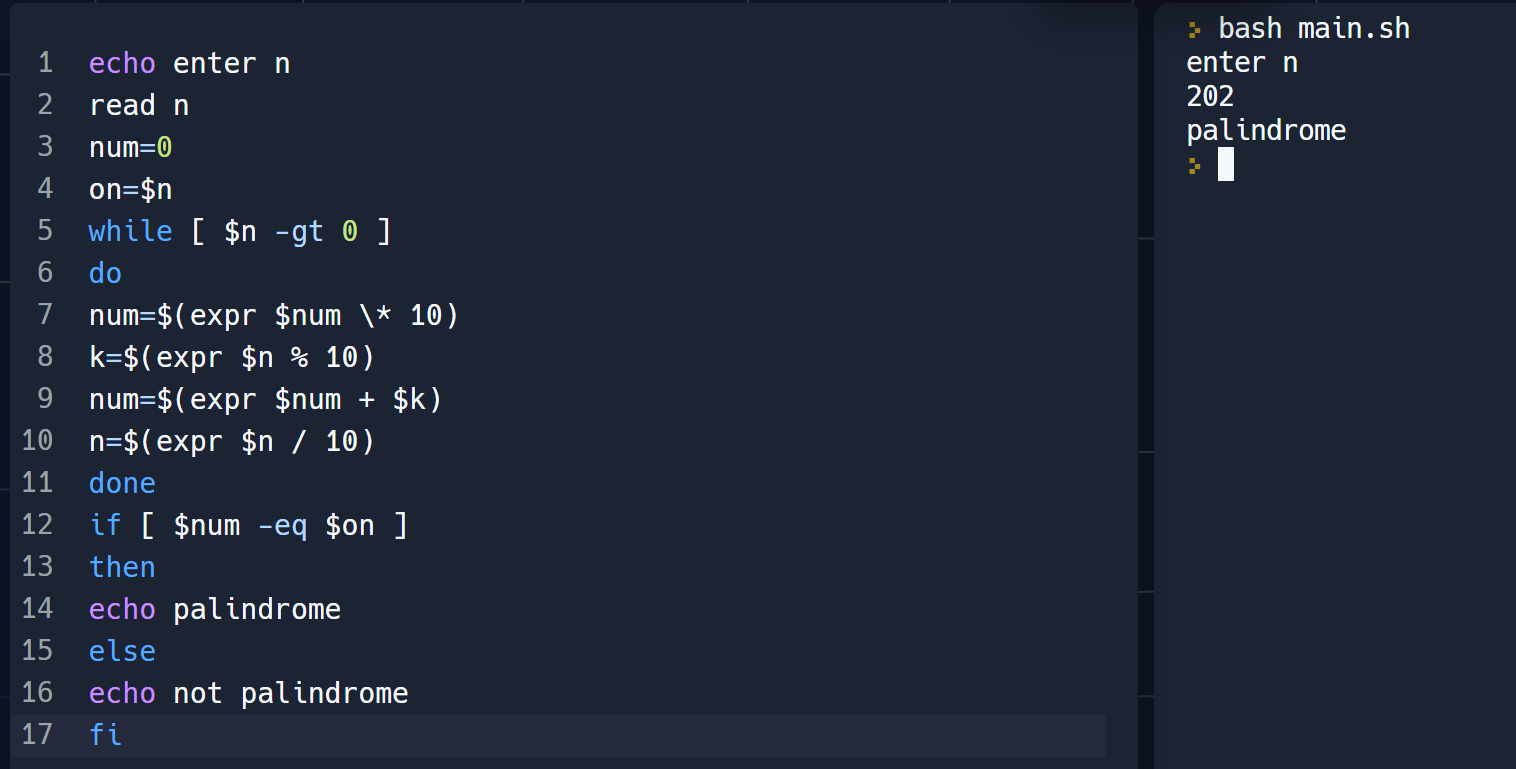


**b) Find greatest of three numbers**

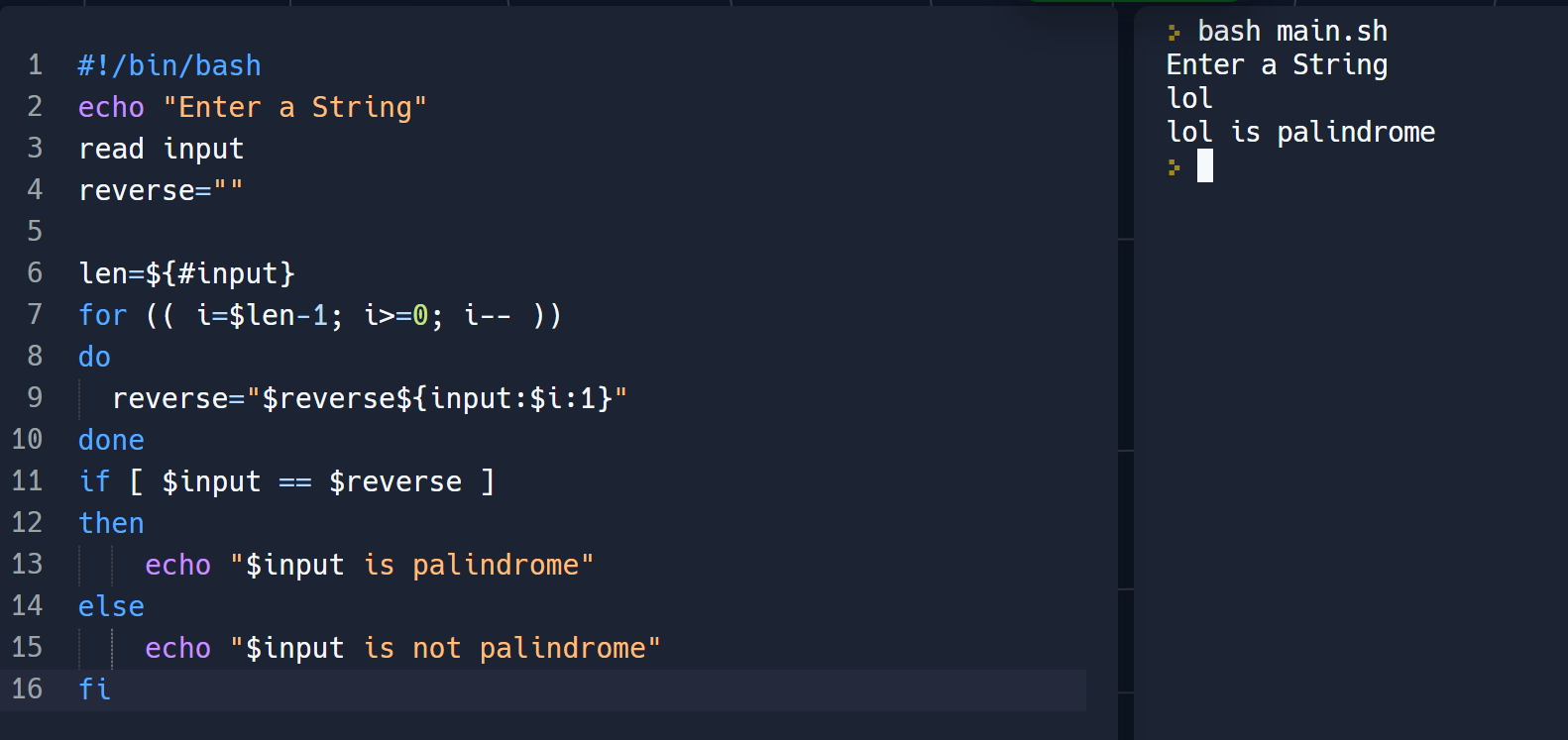


**c) Find a prime no**

****

****

**e) Find whether a string is palindrome**

****

**CONCLUSION:**

We have successfully implemented required shell scripting programs using bash programming language.

**EXP3:** **Implement the following using shell scripting Menu driven program for**

**a) Find the factorial of a no.**

**b) Find greatest of three numbers**

**c) Find a prime no**

**d) Find whether a number is palindrome**

**e) Find whether a string is palindrome**

**THEORY:**

**Shell Scripting:**

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.

**Executing the file**

Now that we have seen what a typical shell script looks like, let’s look at how to execute the file.

1. Save the file with a .sh extension
2. To execute the file, first we need to give it execute permissions.

chmod +x filepath/filename.sh

1. To execute the file, we can do it in the following ways
   * If you are using a GUI file navigation system, right-click on the file and click on run or execute.
   * If you are using the terminal, ./filename.sh will execute the script. (Make sure you are in the correct directory!)

**Commands used:**

**echo "Welcome!! Please Enter Your Name"**

The echo keyword is used to output the strings that it is being passed as arguments. It outputs onto the stdout.

**read name**

The read command is used to receive input while running the script. When we wrote read name it initialized a variable named name and stored the input in it.

**echo "Hello $name"**

The $ symbol is used to print the **value** of a variable. In the above example, it prints whatever input was given in the previous step.

**Switch-Case**

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression. In shell scripting, we have the case command for this.

case "$var" in

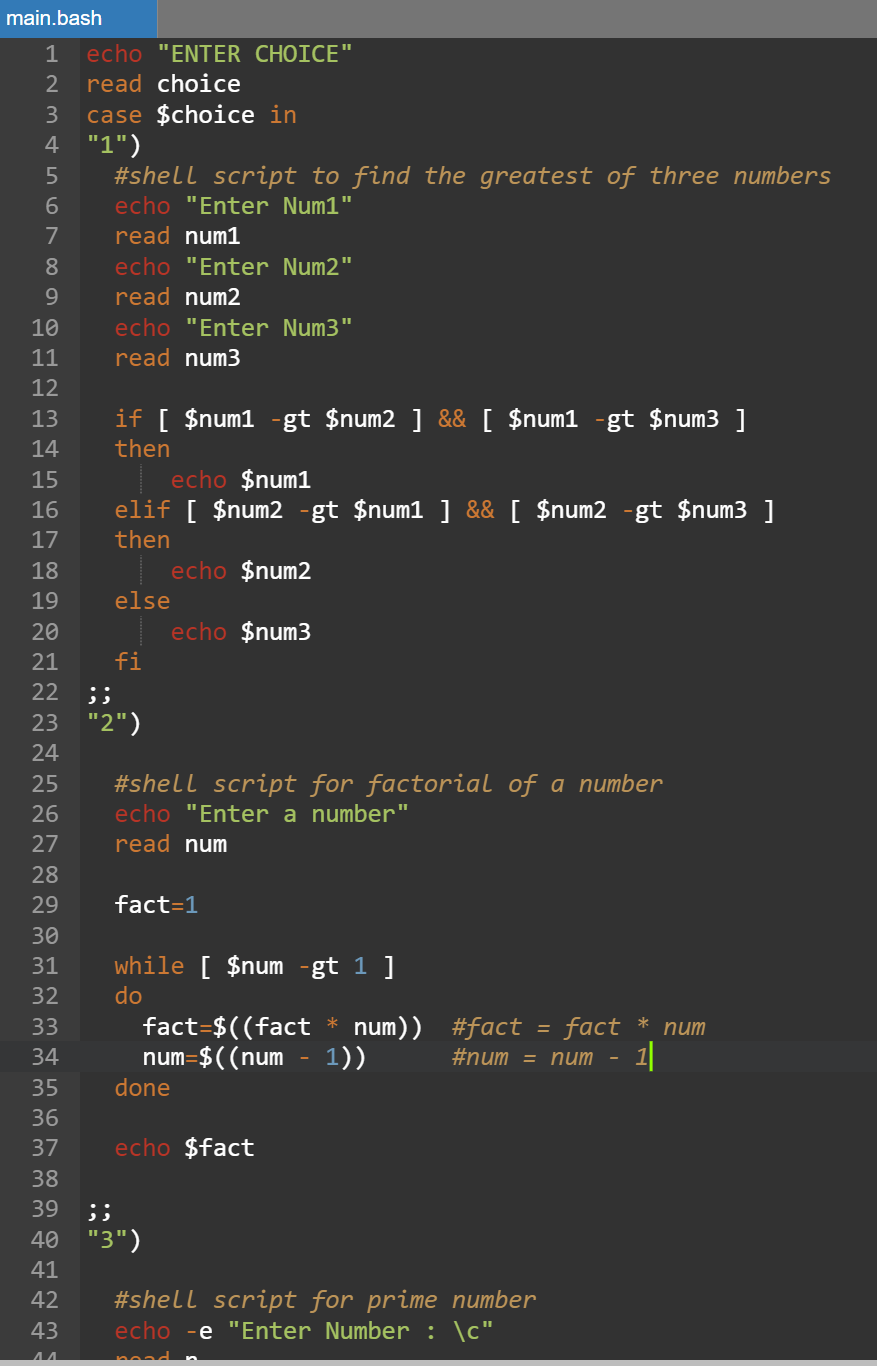
#case 1

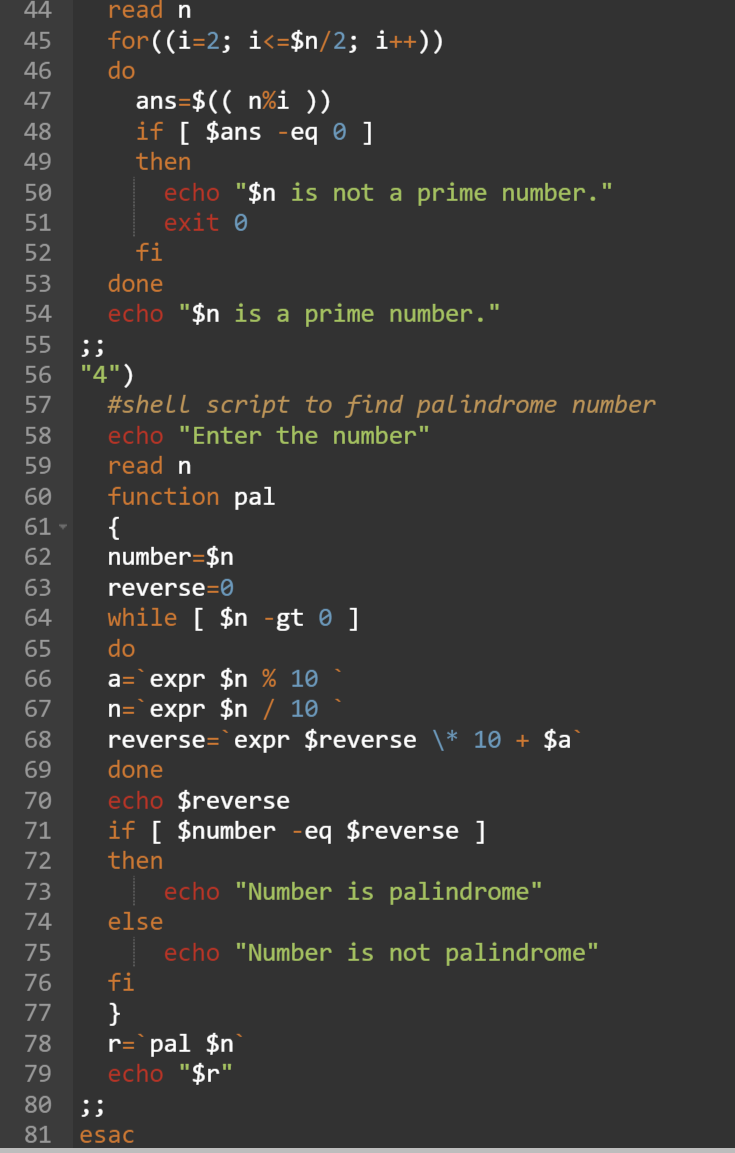
pattern\_1) statement\_1;;

#case 2

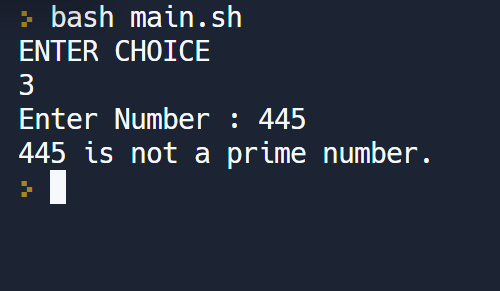
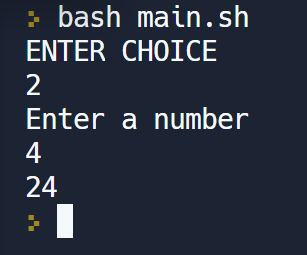
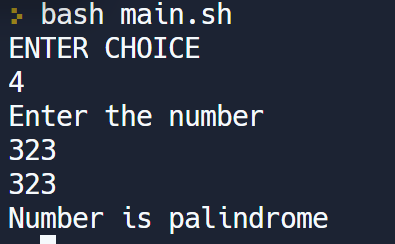
patter\_2) statement\_2;;

**PROGRAM CODE:**





**PROGRAM OUTPUT:**



**CONCLUSION:**

We have successfully implemented required shell scripting program using menu driven interface using bash programming language.

**EXP4: Design suitable data structures and implement macro definition and macro expansion processing for a sample macro with positional and keyword parameters.**

**THEORY:**

**Macros** is a piece of programming code that runs in excel environment, and it helps to automate routine tasks. In other words, a macro is a recording of your regular steps in excel, which you can replay using a single button.

Macros are useful for the following purposes:

· To simplify and reduce the amount of repetitive coding

· To reduce errors caused by repetitive coding

· To make an assembly program more readable.

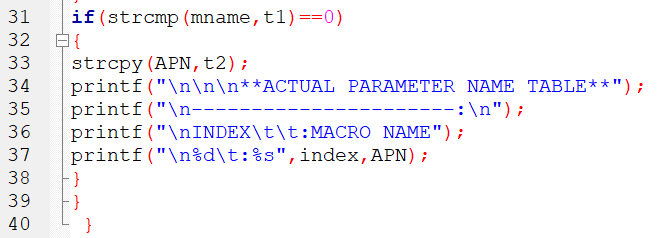
**Macro Expansion.**

A macro call leads to macro expansion. During macro expansion, the macro statement is replaced by sequence of assembly statements.

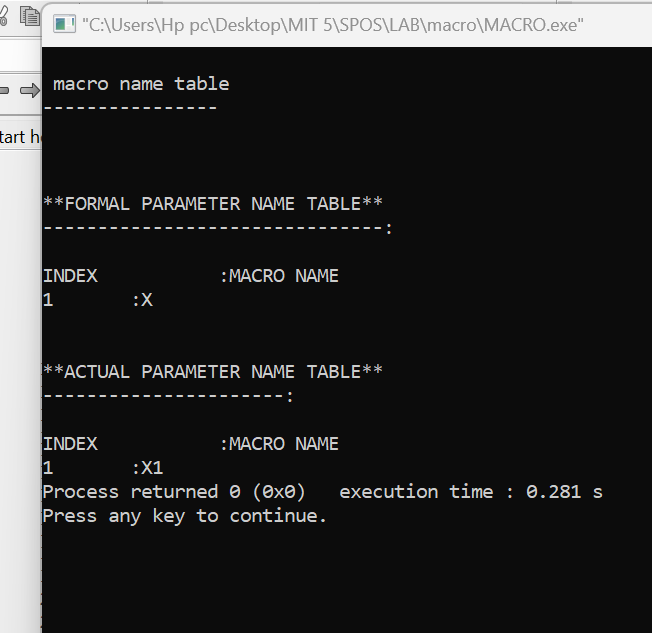
**Parameters in Macro**

1. **Positional parameters** are symbolic parameters that must be specified in a specific order every time the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called.
2. **Keyword parameters** are symbolic parameters that can be specified in any order when the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called. These parameters can be given a default value.

**PROGRAM CODE:**



**PROGRAM OUTPUT:**



**CONCLUSION:**

We have successfully designed suitable data structures and implemented macro definition and macro expansion processing for a sample macro with positional and keyword parameters.

**EXP5: 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:**

A **round-robin** is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called time quantum or time slice**.**

1. **Completion Time**: It defines the time when processes complete their execution.
2. **Turn Around Time**: It defines the time difference between the completion time (CT) and the arrival time (AT).  
   Turn Around Time (TAT) = Completion Time (CT) - Arrival Time (AT)
3. **Waiting Time**: It defines the total time between requesting action and acquiring the resource.  
   Waiting Time (WT) = Turn Around Time (TAT) - Burst Time (BT)
4. **Response Time**: It is the time that defines at which time the **system** response to a process.

**Algorithm:**

**Step 1:** Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

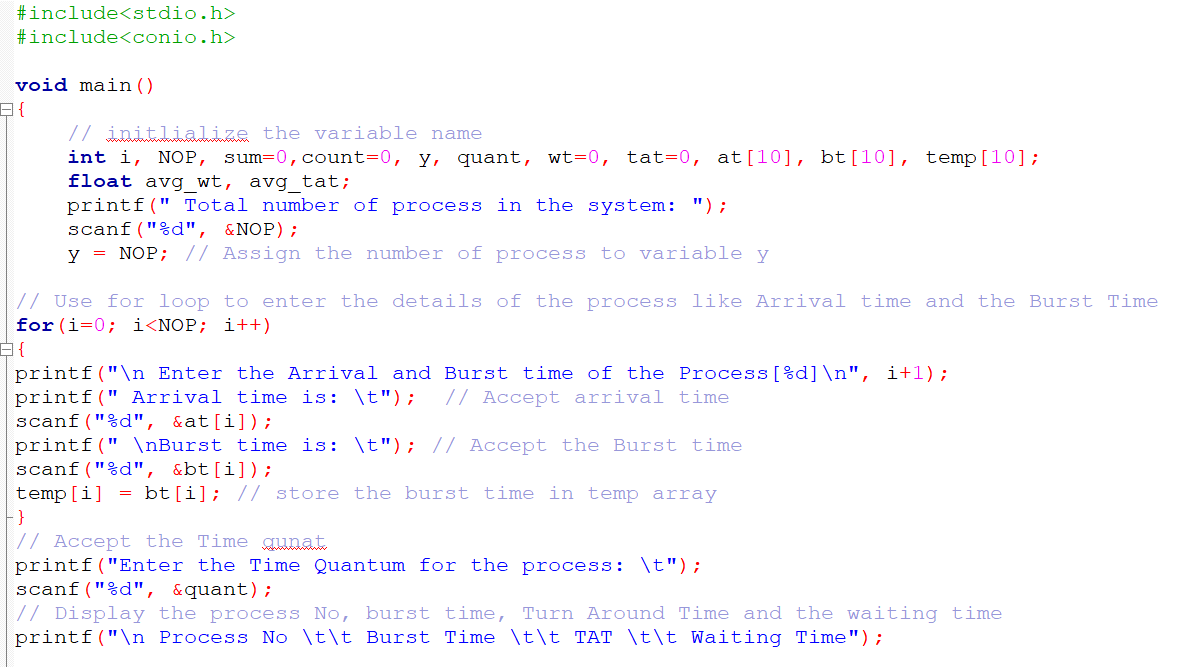
**Step 2:** Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

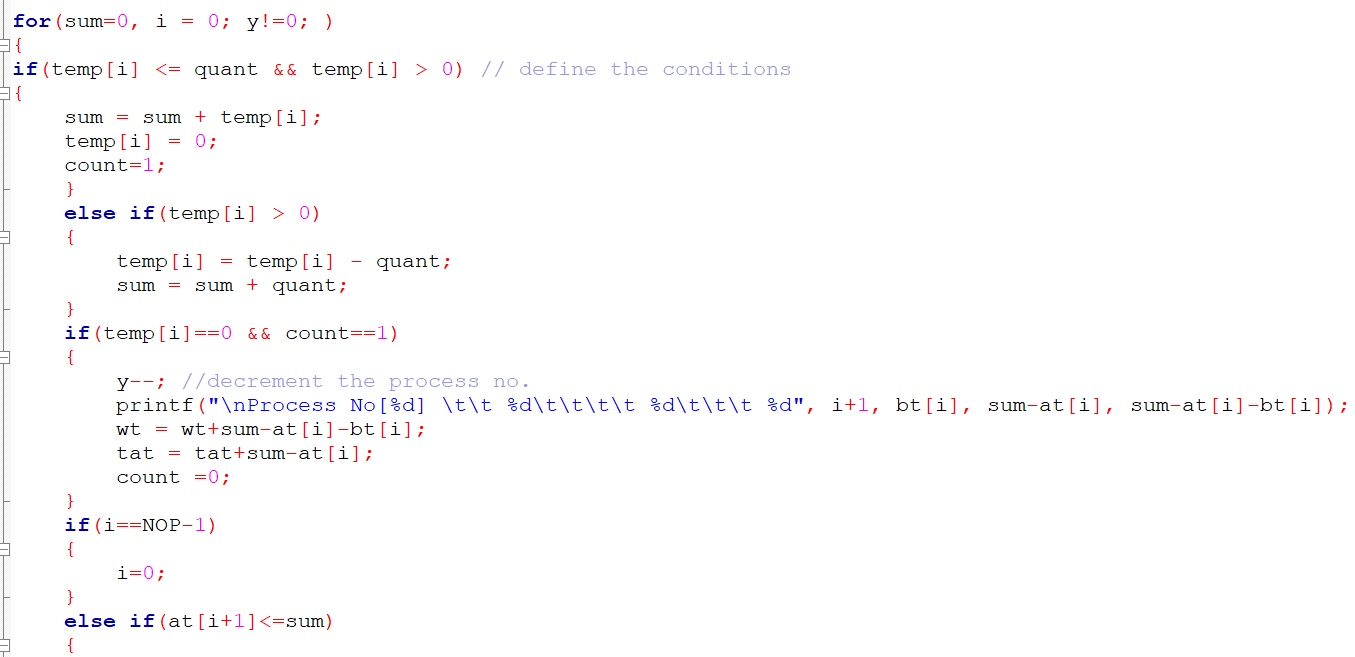
**Step 3:** If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

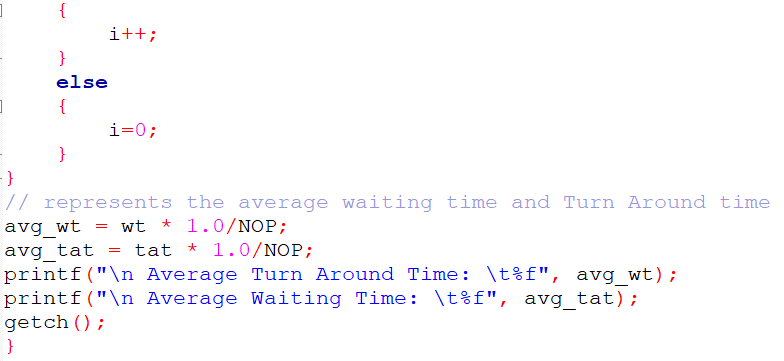
**Step 4:** Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

**Step 5:** Similarly, we repeat all the steps to execute the process until the work has finished.

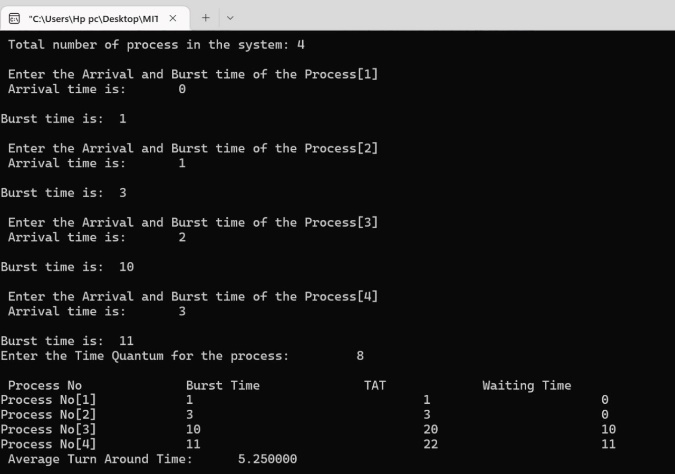
**PROGRAM:**

****

****

****

**OUTPUT:**

****

**● FCFS (Non-preemptive by default)**

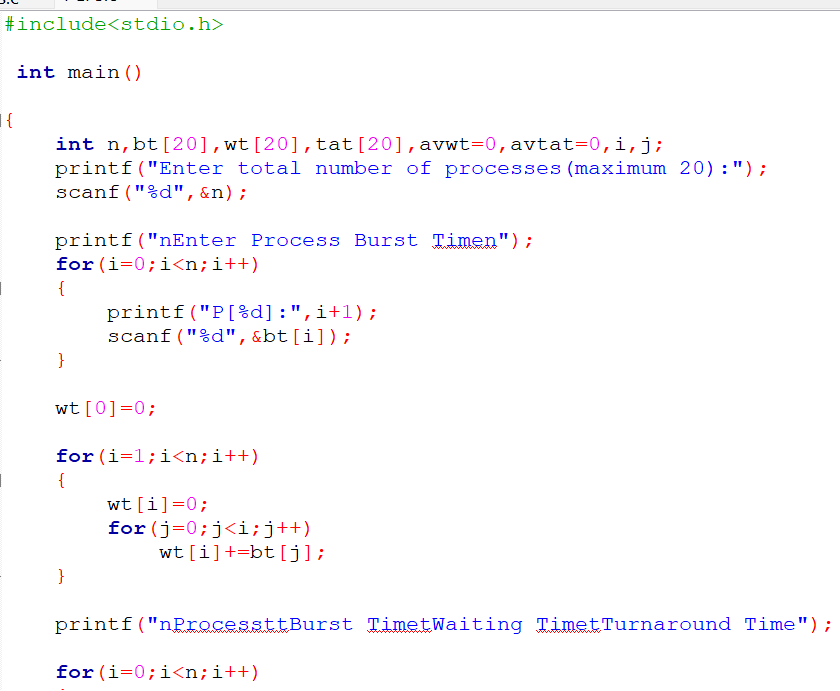
**THEORY:**

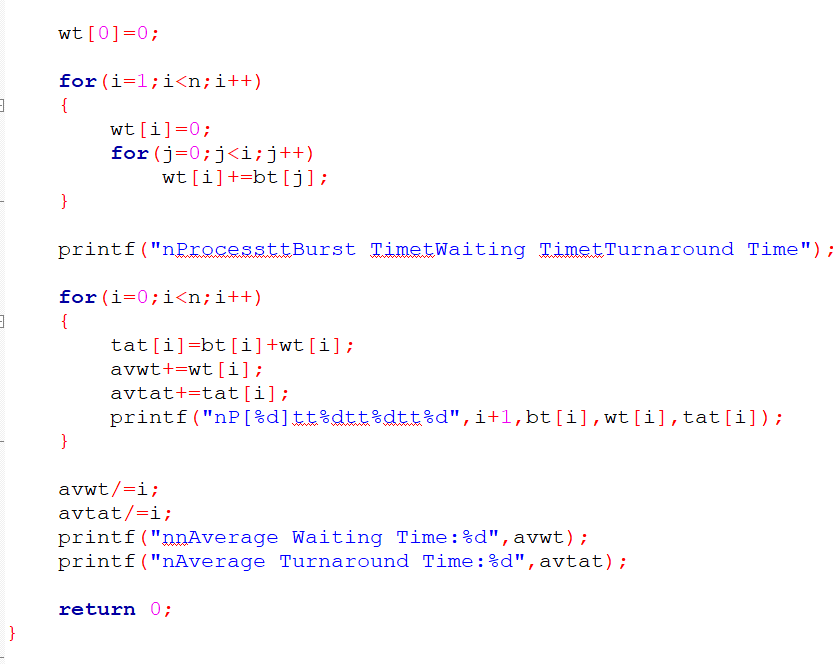
**First Come First Serve** is a scheduling algorithm used by the CPU to schedule jobs. It is a Non-Preemptive Algorithm. Priority is given according to which they are requested by the processor.

The process that requests the services of CPU first, get the CPU first. This is the philosophy used by the first come first serve algorithm.

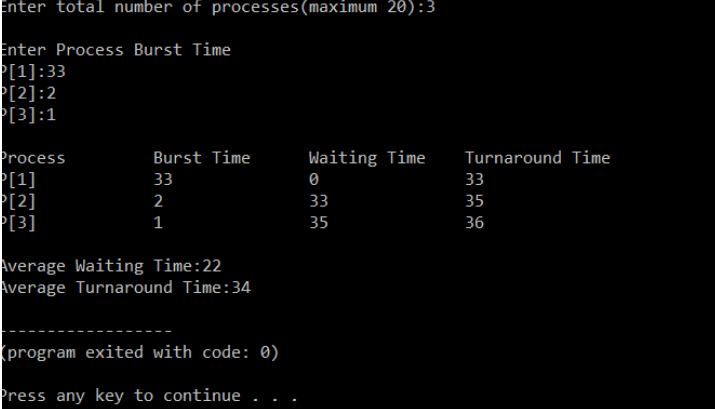
* Completion time: Time when the execution is completed.
* Turn Around Time: The sum of the burst time and waiting time gives the turn-around time
* Waiting time: The time the processes need to wait for it to get the CPU and start execution is called waiting time.

**PROGRAM:**

****



**OUTPUT:**

****

**● SRTN (Preemptive version of SJF)**

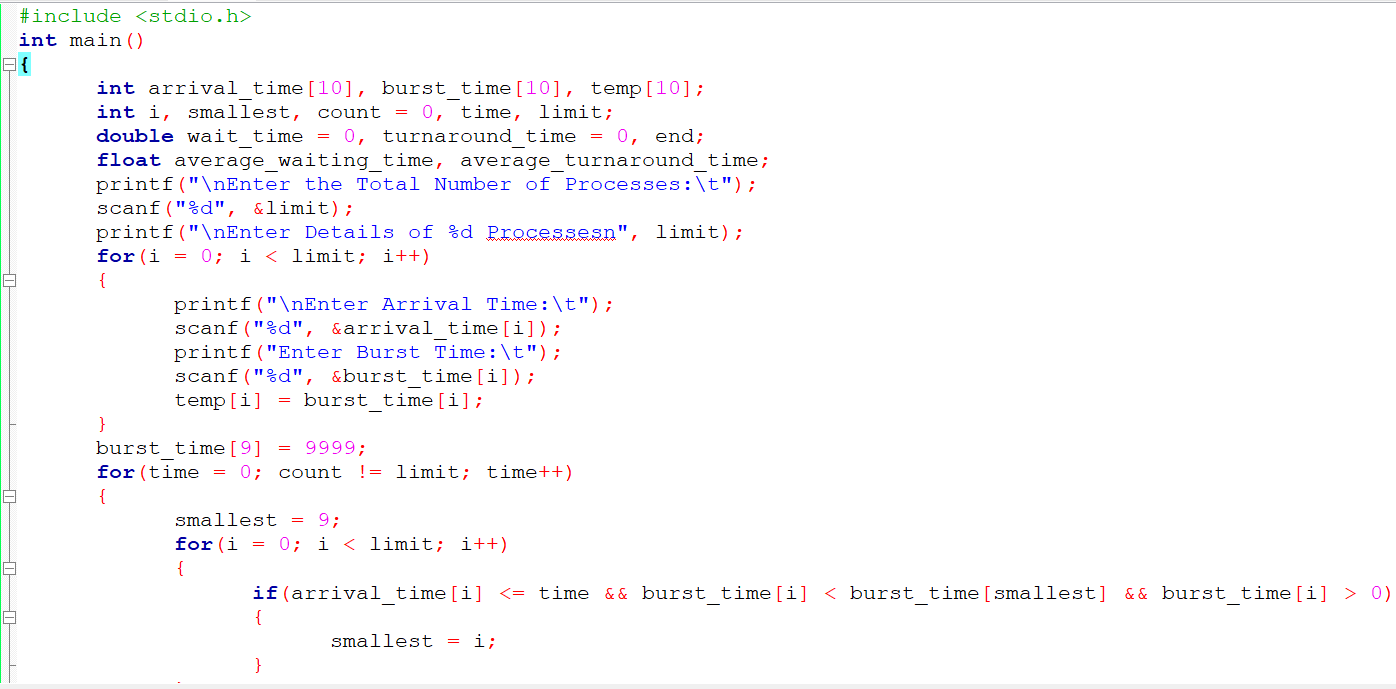
**THEORY:**

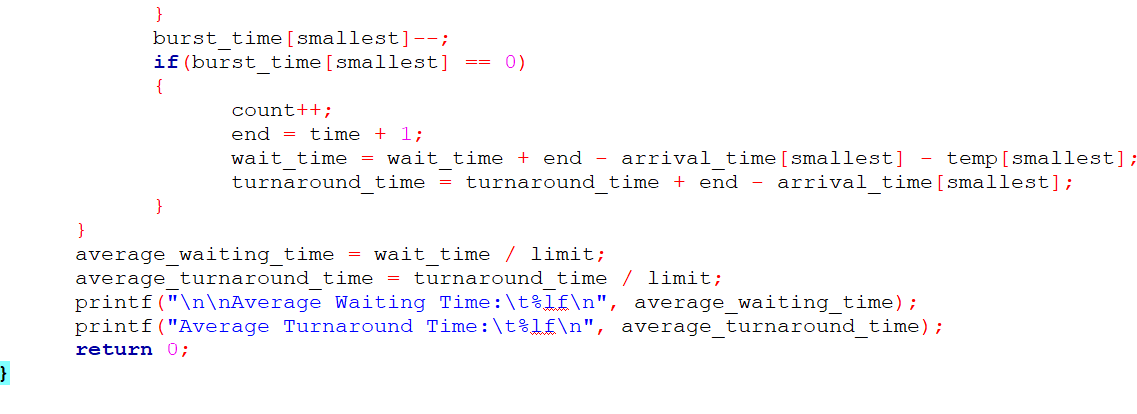
In the **Shortest Remaining Time First (SRTF**) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Algorithm:

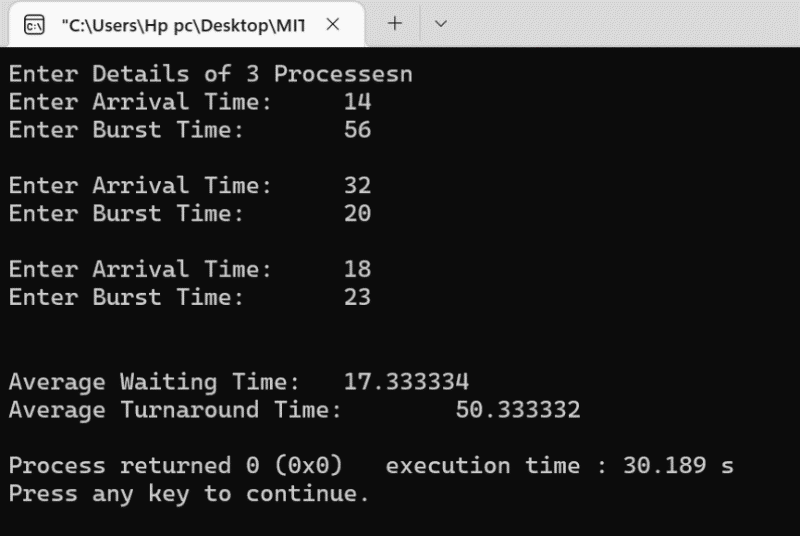
* Traverse until all process gets completely executed.
* Find process with minimum remaining time at every single time lap.
* Reduce its time by 1.
* Check if its remaining time becomes 0
* Increment the counter of process completion.
* Completion time of current process = current\_time + 1;
* Calculate waiting time for each completed process.
  + - **wt[i]= Completion time – arrival\_time-burst\_time**
* Increment time lap by one.
* Find turnaround time (waiting\_time + burst\_time).

**PROGRAM:**

****

****

**OUTPUT:**

****

**CONCLUSION:**

We have successfully implemented scheduling programs using c programming language

**EXP6: Simulation of Banker’s algorithm using ‘C’/Python/Java language**

**THEORY:**

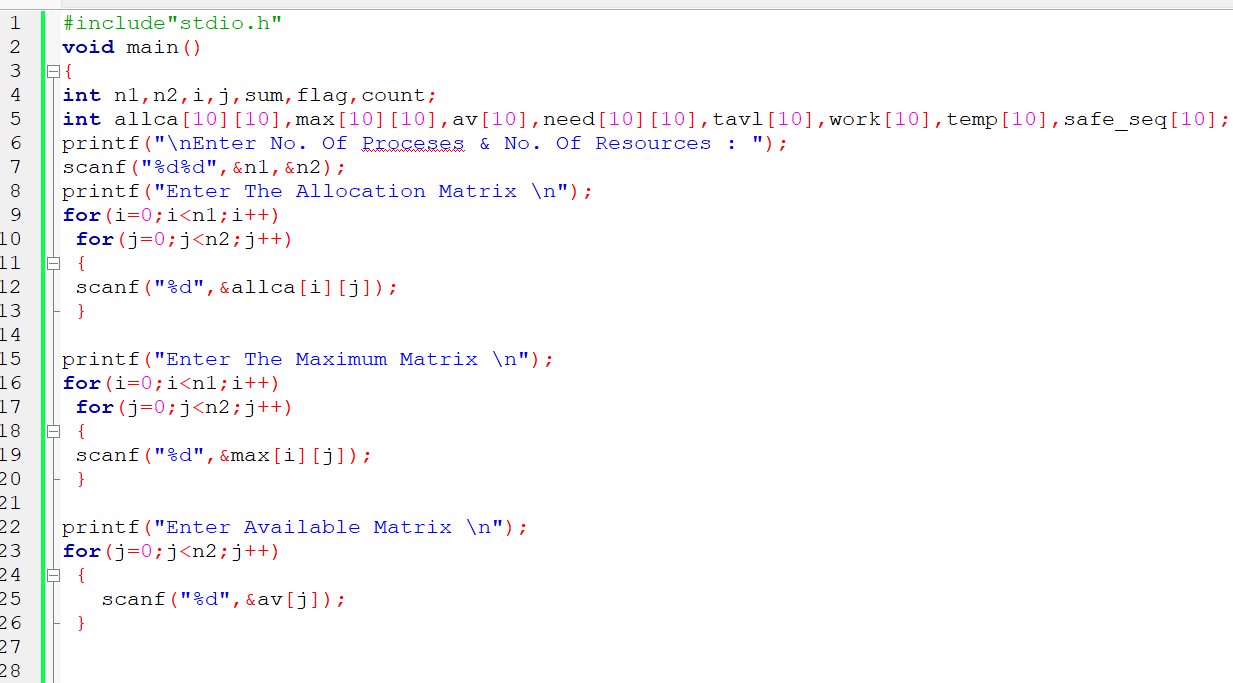
The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be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

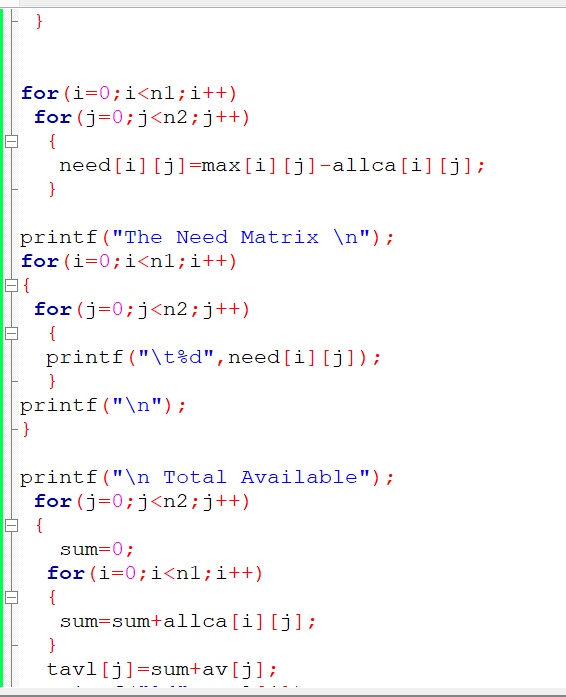
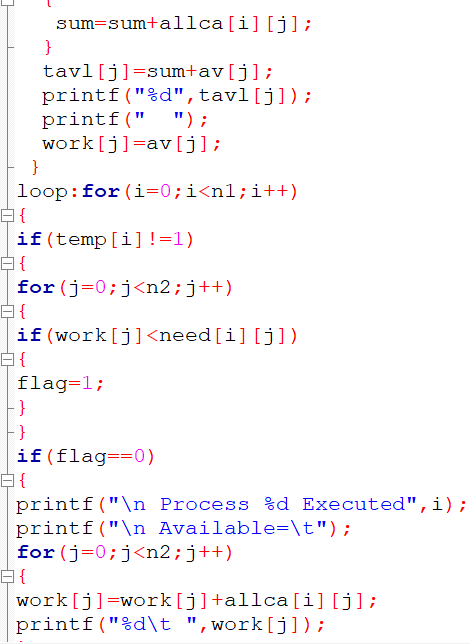
**Safety Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

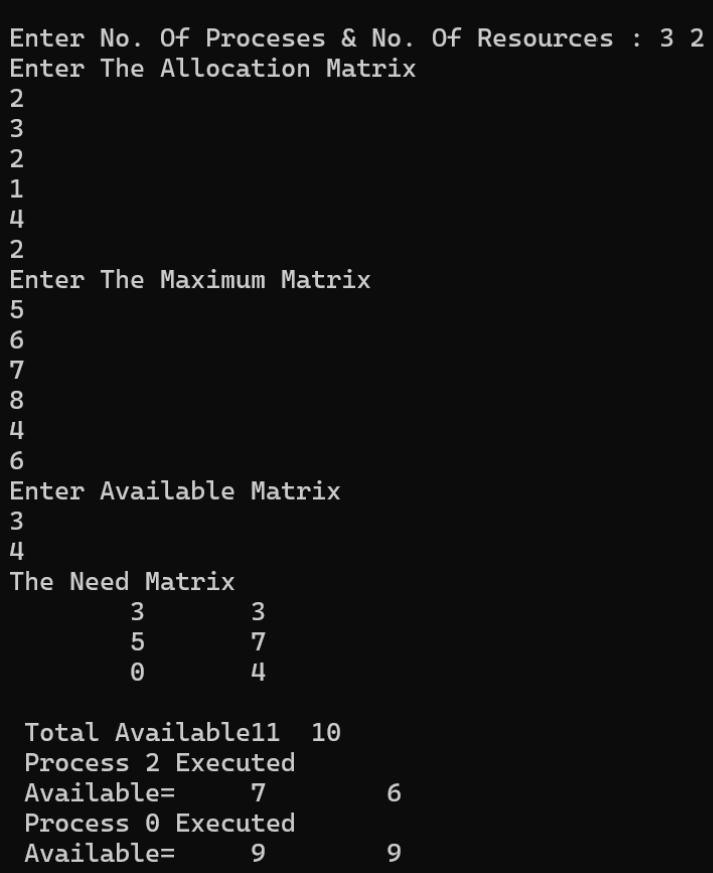
1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.  
   Initialize: Work= Available  
   Finish [i]=false; for i=1,2,……,n
2. Find an i such that both  
   a) Finish [i]=false  
   b) Need\_i<=work  
     
   if no such i exists goto step (4)
3. Work=Work + Allocation\_i  
   Finish[i]= true  
   goto step(2)
4. If Finish[i]=true for all i,  
   then the system is in safe state.

**PROGRAM:**



**OUTPUT:**

****

**CONCLUSION:**

We have successfully implemented Bankers scheduling program using c programming language

**EXP7: Implement the following page replacement algorithms**

**● FIFO**

**● LRU**

**THEORY:**

Page Replacement Algorithms: The operating system uses the method of paging for memory management. This method involves page replacement algorithms to make a decision about which pages should be replaced when new pages are demanded. The demand occurs when the operating system needs a page for processing, and it is not present in the main memory. The situation is known as a page fault.

FIFO ALGORITHM:

Step 1. Start to traverse the pages.

Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.

Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.

Step 4. If the current page is present in the memory, do nothing.

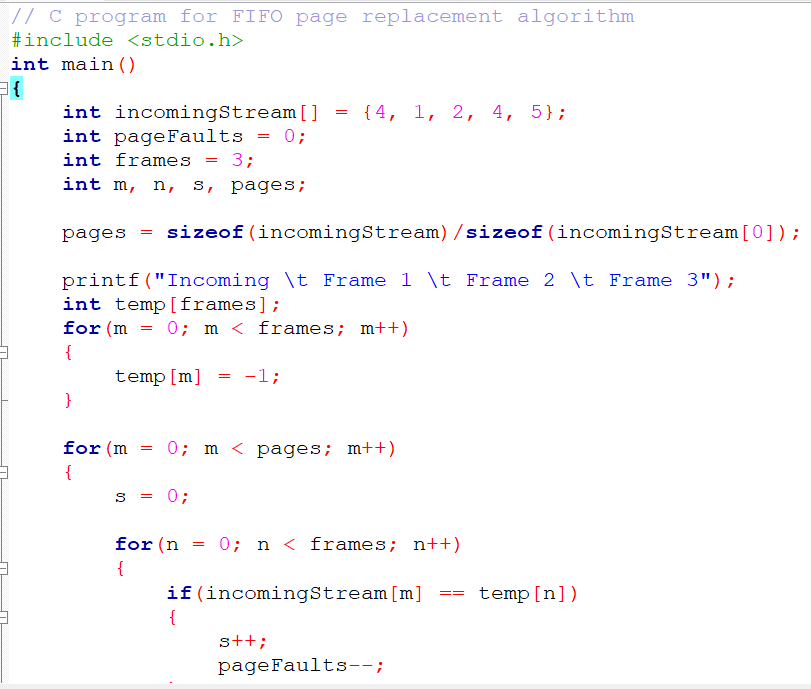
Step 5. Else, pop the topmost page from the queue as it was inserted first.

Step 6. Replace the topmost page with the current page from the string.

Step 7. Increment the page faults.

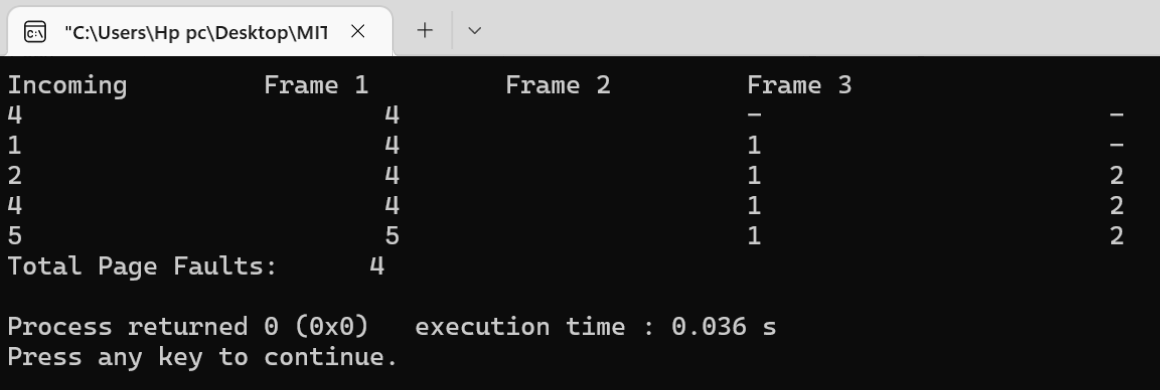
Step 8. Stop

**PROGRAM:**

****

****

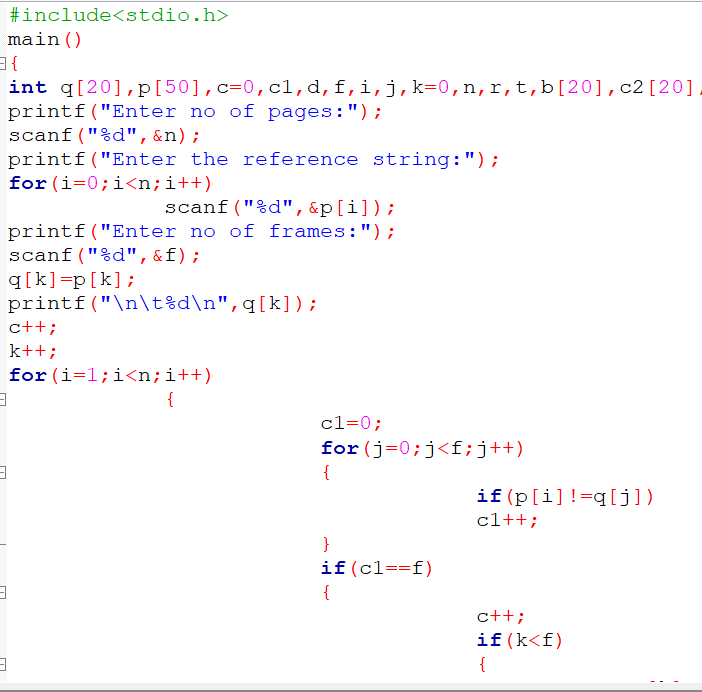
**OUTPUT:**

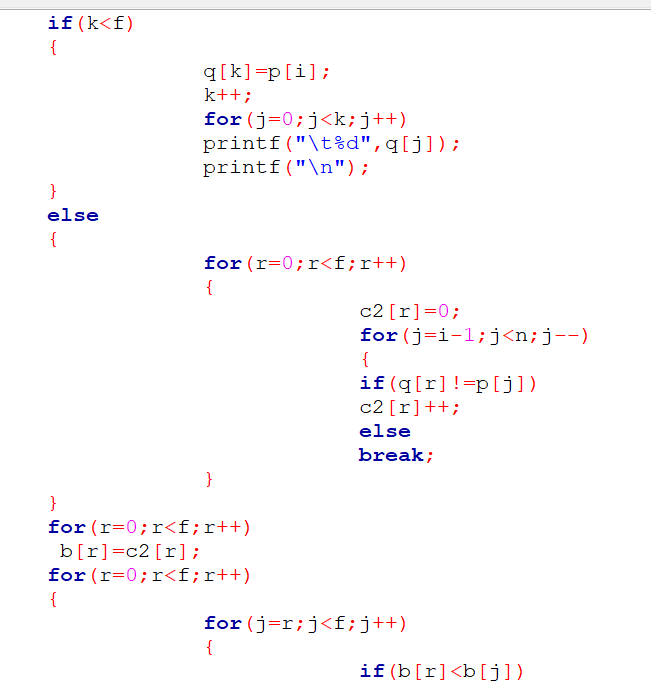
****

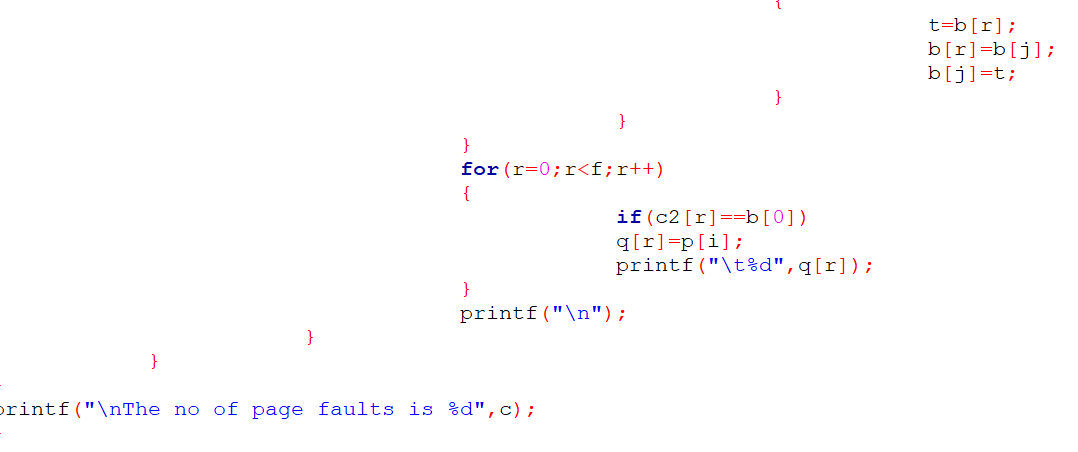
**LRU ALGORITHM:**

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

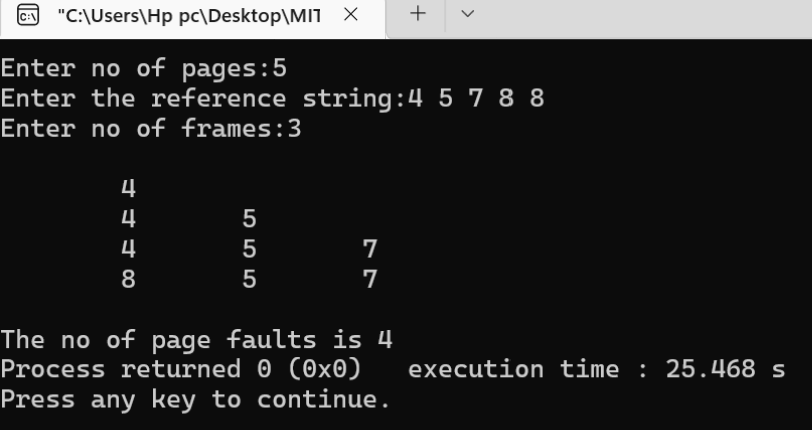
**PROGRAM:**

****

****

****

**OUTPUT:**

****

**CONCLUSION:**

We have successfully implemented FIFO & LRU page replacement programs using c programming language

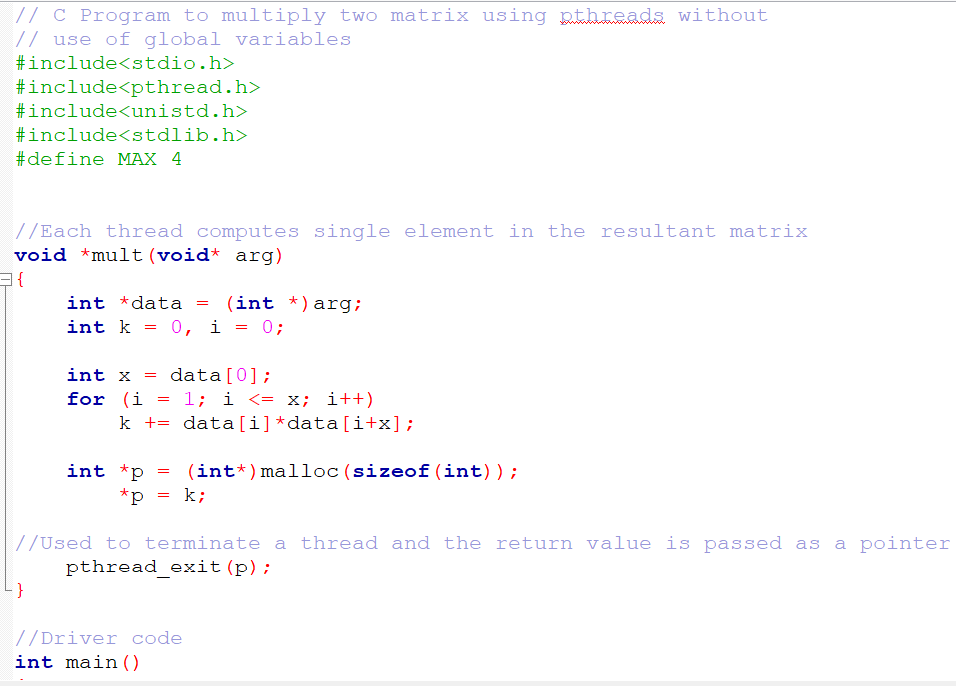
**EXP8: Implement matrix multiplication using multithreading with pthread library.**

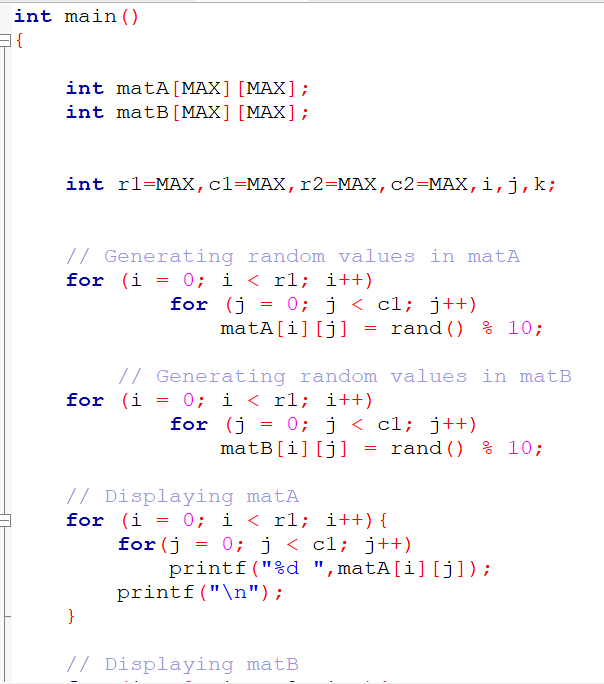
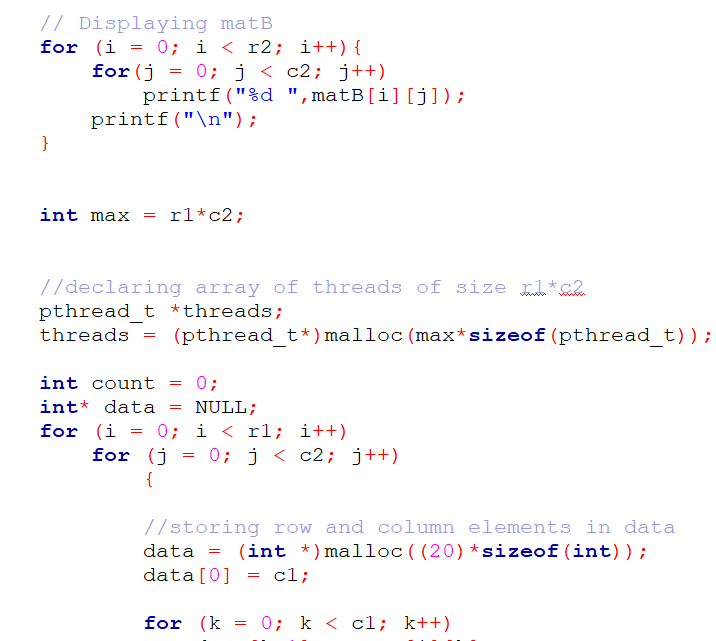
**THEORY:**

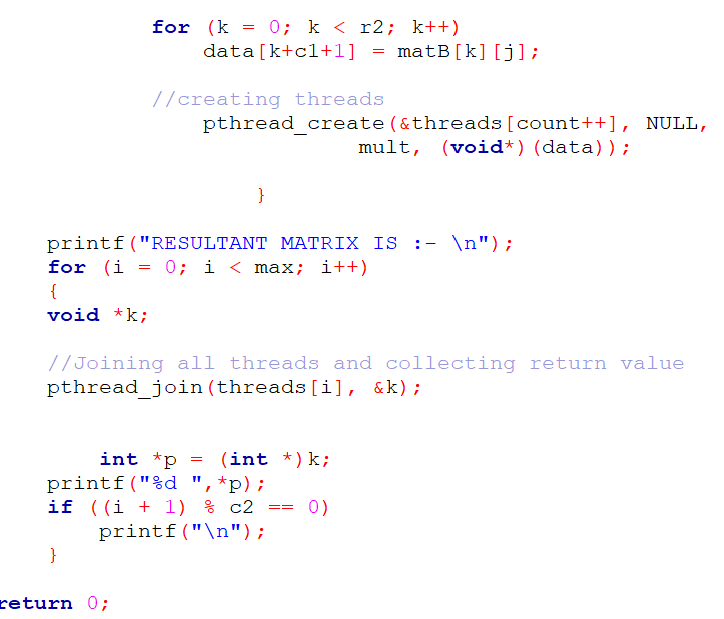
Thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically as part of an operating system). The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is a component of a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. Threads are created using pthread create().

**Algorithm**  
Input: two matrices.  
Output: Output matrix C.  
procedure Matrix-Multiply(A, B)  
1. if columns [A] != rows [B]  
2.then error “incompatible dimensions”  
3. else  
4.for i = 1 to rows [A]  
5.for j = 1 to columns [B]  
6.C[i, j] =0  
7.for k = 1 to columns [A]  
8.C[i, j]=C[i, j]+A[i, k]\*B[k, j]  
9. return C

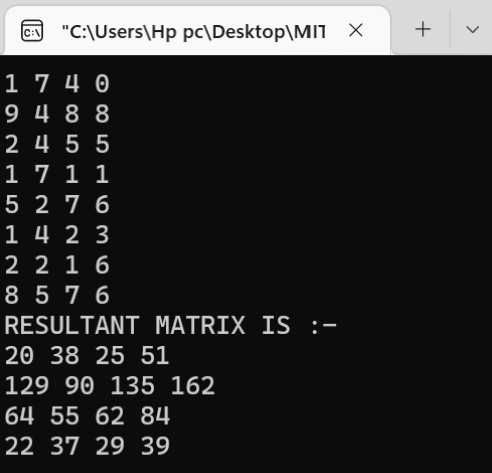
**PROGRAM:**

****

** **

****

**OUTPUT:**

****

**CONCLUSION:**

We have successfully implemented matrix multiplication using multithreading with pthread library.