# Operating System

# (OS type)

**A Mini-Project Report *Submitted by***

## NAME OF THE CANDIDATE (Roll No.)

Rushang Phira (E011)

Alok Rajpurohit (E014)

Aniket Raman (E015)

Shaurya Rawat (E018)

***Under the Guidance of***

**Prof. Swarnalata Bollavarapu**

## B. TECH. (COMPUTER ENGINEERING)

## At



**MUKESH PATEL SCHOOL OF TECHNOLOGY MANAGEMENT & ENGINEERING**

**MONTH & YEAR**

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| Sr. No. | Topic |
| 1. | Introduction |
| 2. | Problem Statement |
| 3. | Scope of the Project |
| 4. | Literature Survey |
| 5. | Analysis & Design |
| 6. | Source Code |
| 7. | Output Screenshots |
| 8. | Conclusion |
| 9. | References |

**INTRODUCTION**

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Following are some of important functions of an operating System.

* Memory Management
* Processor Management
* Device Management
* File Management
* Security
* Control over system performance
* Job accounting
* Error detecting aids
* Coordination between other software and users

A mobile operating system (mobile OS) is an OS built exclusively for a mobile device, such as a smartphone, personal digital assistant (PDA), tablet or other embedded mobile OS. Popular mobile operating systems are Android, Symbian, iOS, BlackBerry OS and Windows Mobile.

A mobile OS is responsible for identifying and defining mobile device features and functions, including keypads, application synchronization, email, thumbwheel and text messaging. A mobile OS is like a standard OS (like Windows, Linux, and Mac) but is relatively simple and light and primarily manages the wireless variations of local and broadband connections, mobile multimedia and various input methods.

To adapt to inherent mobile device environments, a mobile OS runs on limited resources emphasizing communication, such as random access memory (RAM), storage and central processing unit (CPU) speed.

Below is an example describing how text messaging works on a mobile OS:

A mobile application allows a user to read and write a message for delivery to a mobile device through radio signal waves. After the device receives the message signals, the device notifies the mobile OS, which stores the message and notifies the messaging application.

The user reads the message and responds with a reply message.

The OS uses the hardware antennae to transmit the message.

**PROBLEM STATEMENT**

Design a working simulation of a Mobile Operating System. The simulated O.S. should include the following functional areas:

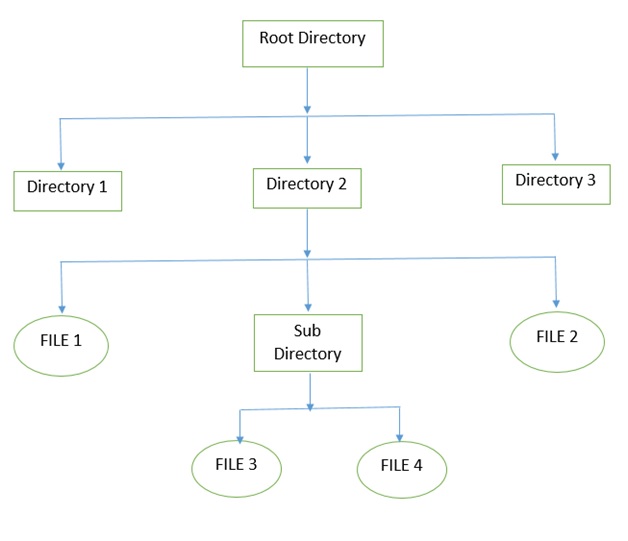
* File Management (Hash Table Algorithm)
* I/O Management (Deadline Scheduling [Earliest Deadline First Algorithm])
* Memory Management (Buddy Algorithm)
* Process Management (Highest Response Ratio Next Algorithm)

**SCOPE OF THE PROJECT**

**File Management:**

The following are some of the tasks performed by file management of mobile operating system:

* It helps to create new files in computer system and placing them at the specific locations.
* It helps in easily and quickly locating these files in computer system.
* It makes the process of sharing of the files among different users very easy and user friendly.
* It helps to stores the files in separate folders known as directories. These directories help users to search file quickly or to manage the files according to their types or uses.
* It helps the user to modify the data of files or to modify the name of the file in the directories.



The above figure shows the general hierarchy of the storage in an operating system. In this figure the root directory is present at the highest level in the hierarchical structure. It includes all the subdirectories in which the files are stored. Subdirectory is a directory present inside another directory in the file storage system. The directory base storage system ensures better organization of files in the memory of the computer system.

There is the number of algorithms by using which, the directories can be implemented. However, the selection of an appropriate directory implementation algorithm may significantly affect the performance of the system.

**I/O Management:**

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation.

I/O Subsystems must contend with two (conflicting?) trends:

(1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and,

(2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

Device drivers are modules that can be plugged into an OS to handle a device or category of similar devices.

**Memory Management:**

Memory Management is the way memory is allocated and de allocated to processes according to their need and following a certain algorithm. Most of the time memory addresses are not used up ideally or efficiently. There are many memory addresses in the middle which are left unallocated. This is known as fragmentation.

**Process Management:**

Process management in a typical operating system involves many complex data structures and algorithms but doesn’t go much beyond the level managing the typical process data structure. Android is similar in that at the base level the control structures look the same.

**LITERATURE SURVEY**

4 algorithms were implemented in the source code of this project which according to us best fitted the problem statement at hand. However as one could expect, a lot of algorithms in all fields were rejected/not seen as suitable enough to apply to the given problem statement. Some of them are listed as follows:

**Memory Management:**  
[First-Fit Memory Allocation](https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/)**:**  
This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to it’s size. The operating system doesn’t search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

[Best-Fit](https://www.geeksforgeeks.org/program-best-fit-algorithm-memory-management/)**:**  
This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.

Worst Fit:

The algorithm searches for free-space in memory in which it can store the desired information. The algorithm selects the largest possible free space that the information can be stored on (i.e. that is bigger than the information needing to be stored) and stores it there. This is directly opposed to the best fit algorithm which searches the memory in much the same way as before.

Next Fit:

Next fit is a modified version of [‘first fit’](https://tutorialspoint.dev/slugresolver/program-first-fit-algorithm-memory-management/). It begins as the first fit to find a free partition but when called next time it starts searching from where it left off, not from the beginning. This policy makes use of a roving pointer. The pointer moves along the memory chain to search for a next fit. This helps in, to avoid the usage of memory always from the head (beginning) of the free block chain.

**Process Management:**

[First Come First Serve (FCFS)](https://www.geeksforgeeks.org/program-fcfs-scheduling-set-1/):

Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.

Note:First come first serve suffers from [convoy effect](https://www.geeksforgeeks.org/convoy-effect-operating-systems/).

[Shortest Job First (SJF)](https://www.geeksforgeeks.org/program-shortest-job-first-sjf-scheduling-set-1-non-preemptive/):

Process which have the shortest burst time are scheduled first.If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

[Shortest Remaining Time First (SRTF)](https://www.geeksforgeeks.org/program-shortest-job-first-scheduling-set-2srtf-make-changesdoneplease-review/)**:**

It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

[Round Robin Scheduling](https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1/):

Each process is assigned a fixed time(Time Quantum/Time Slice) in cyclic way.It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

[Priority Based scheduling (Non-Preemptive)](https://www.geeksforgeeks.org/operating-system-priority-scheduling-different-arrival-time-set-2/):

In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first. If priorities of two processes match, then schedule according to arrival time. Here starvation of process is possible.

**I/O Management:**

Elevator Algorithm:  
The elevator algorithm (also SCAN) is a disk-scheduling algorithm to determine the motion of the disk's arm and head in servicing read and write requests.

This algorithm is named after the behavior of a building elevator, where the elevator continues to travel in its current direction (up or down) until empty, stopping only to let individuals off or to pick up new individuals heading in the same direction.

N-Step-SCAN:

N-Step-SCAN (also referred to as N-Step LOOK) is a disk scheduling algorithm to determine the motion of the disk's arm and head in servicing read and write requests. It segments the request queue into subqueues of length N. Breaking the queue into segments of N requests makes service guarantees possible

FScan:

FScan is a disk scheduling algorithm to determine the motion of the disk's arm and head in servicing read and write requests. It uses two sub-queues. During the scan, all of the requests are in the first queue and all new requests are put into the second queue. Thus, service of new requests is deferred until all of the old requests have been processed. When the scan ends, the arm is taken to the first queue entries and is started all over again.

**File Management:**

Continuous Allocation: A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1.

Linked Allocation(Non-contiguous allocation):

Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

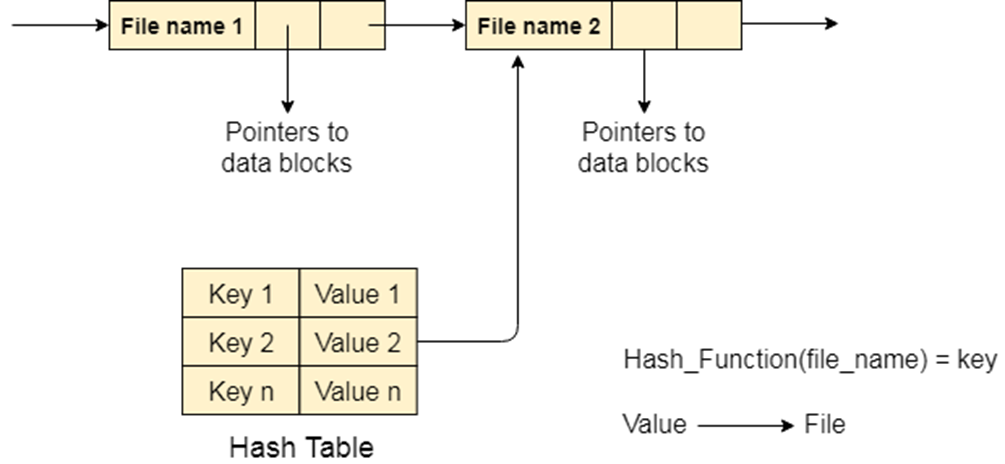
**ANALYSIS & DESIGN**

**File Management:**

*Hash Table Algorithm:*

In this algorithm, all the files in a directory are maintained as singly lined list. Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory.

To overcome the drawbacks of singly linked list implementation of directories, there is an alternative approach that is hash table. This approach suggests to use hash table along with the linked lists.



**I/O Management:**

*Deadline Scheduling (Earliest Deadline First Algorithm):*

The user enters the number of processes as well as the deadlines, time-period and the burst times of each. Process with the earliest deadline gets executed first for the duration of its time-period. Deadlines and burst times are examined again and provided no deadlines have been reached and processes are yet to be completed, the step is repeated again and again until all processes get completed without a single deadline having passed (in which case it is successful) or if any process is incomplete with its deadline having passed (i.e. it has failed).

**Memory Management:**

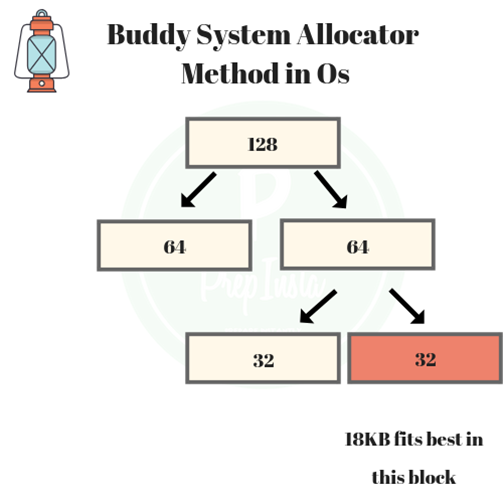
*Buddy Algorithm:*

The buddy system is a memory allocation and management algorithm that manages memory in power of two increments. Assume the memory size is 2U, suppose a size of S is required.

• If 2U-1<S<=2U: Allocate the whole block

• Else: Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block and get out the loop.

For example-If there is a address block of 256 kb and a process with 37 kb has to be allocated memory, the 256 kb block keeps getting fragmented into two halves(buddies) until we get the minimum memory space in the form 2^n in which process can fit. In this case 256 gets fragmented in to 2 128 kb blocks. Similarly, 1 128 kb block is fragmented into 2 64 kb blocks. This is the block with minimum memory space which can fit the process.



**Process Management:**

*Highest Response Ratio Next Algorithm:*

Given n processes with their Arrival times and Burst times, the task is to find average waiting time and average turnaround time using HRRN scheduling algorithm.

The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Criteria – Response Ratio

Mode – Non-Preemptive

Response Ratio = (W + S)/S

Here, W is the waiting time of the process so far and S is the Burst time of the process.

Implementation of HRRN Scheduling –

1. Input the number of processes, their arrival times and burst times.

2. Sort them according to their arrival times.

3. At any given time calculate the response ratios and select the appropriate process to be scheduled.

4. Calculate the turnaround time as completion time – arrival time.

5. Calculate the waiting time as turnaround time – burst time.

6. Turnaround time divided by the burst time gives the normalized turnaround time.

7. Sum up the waiting and turnaround times of all processes and divide by the number of processes to get the average waiting and turnaround time.

**SOURCE CODE**

#include<iostream>

#include<string.h>

#include<math.h>

#include<stdlib.h>

#include<conio.h>

#include<stdio.h>

#include<dos.h>

using namespace std;

void validate\_command(string);

void command\_words\_calculate(string,string\*);

void accept\_command();

void display();

int check\_directory(string);

void create\_directory(string);

int check\_file(string,string);

void create\_file(string , string);

void delete\_file(string directory,string file);

void delete\_directory(string directory);

void io\_manager();

void memory\_manager();

void process\_manager();

void main\_screen1();

void main\_screen2();

int keycounter=0;

//double dimensional linked list with key generated by simple hash function where columns store directories and rows store files

struct node1

{

string data;

int key;

struct node1\* row;

struct node1\* column;

};

struct node1\* head1=NULL;

//function to create a new directory

void create\_directory(string directory)

{

int flag=check\_directory(directory);

if(flag==1)

{

cout<<"directory already exists...\n"<<endl;

accept\_command();

}

else

{

keycounter++;

node1\* p=new node1;

p->key=keycounter;

p->data=directory;

p->row=NULL;

p->column=NULL;

if(head1==NULL)

{

head1=p;

}

else

{

node1\* current=head1;

while(current->column!=NULL)

{

current=current->column;

}

current->column=p;

}

cout<<"directory '"<<directory<<"' created.\n"<<endl;

accept\_command();

}

}

// function to check whether directory exists in file system or not

int check\_directory(string directory)

{

node1\* current=head1;

while(current!=NULL)

{

if(current->data==directory)

{

return 1;

}

current=current->column;

}

return 0;

}

//function to create a new file

void create\_file(string directory,string file)

{

int flag1=check\_directory(directory);

if(flag1==0)

{

cout<<"directory '"<<directory<<"' does not exist...\n"<<endl;

accept\_command();

}

int flag=check\_file(directory,file);

if(flag==1)

{

cout<<"file already exists...\n"<<endl;

accept\_command();

}

else

{

node1\* p=new node1;

node1\* current = head1;

p->data=file;

p->row=NULL;

p->column=NULL;

while(current!=NULL)

{

if(current->data==directory)

break;

current=current->column;

}

p->key=current->key;

while(current->row!=NULL)

{

current=current->row;

}

current->row=p;

cout<<"file '"<<directory<<"\\"<<file<<"' created.\n"<<endl;

accept\_command();

}

}

// function to check whether file exists in file system or not

int check\_file(string directory, string file)

{

node1\* current=head1;

while(current!=NULL)

{

if(current->data==directory)

break;

current=current->column;

}

while(current->row!=NULL)

{

current=current->row;

if(current->data==file)

return 1;

}

return 0;

}

//function to display all directories in file system

void show\_directories()

{

int counter=0;

node1\* current=head1;

if(head1==NULL)

{

cout<<"no directories.\n"<<endl;

accept\_command();

}

else

{

while(current!=NULL)

{

counter++;

cout<<"- "<<current->data<<endl;

current=current->column;

}

cout<<"\n"<<counter<<" directories fetched.\n"<<endl;

}

}

//function to display all files in a directory of the file system

void show\_files(string directory)

{

int flag=check\_directory(directory);

if(flag==0)

{

cout<<"directory does not exist...\n"<<endl;

accept\_command();

}

else

{

int counter=0;

node1\* current=head1;

while(current!=NULL)

{

if(current->data==directory)

break;

current=current->column;

}

while(current->row!=NULL)

{

current=current->row;

cout<<"- "<<current->data<<endl;

counter++;

}

if(counter==0)

cout<<"no files."<<endl;

cout<<"\n"<<counter<<" files fetched.\n"<<endl;

accept\_command();

}

}

//function to delete a directory from the file system

void delete\_directory(string directory)

{

int flag=check\_directory(directory);

if(flag==0)

{

cout<<"directory does not exist...\n"<<endl;

accept\_command();

}

else

{

node1 \*current1=head1;

node1 \*current2=head1;

if(head1->data==directory)

{

head1=current1->column;

delete current1;

delete current2;

}

else

{

while(current1->column!=NULL)

{

current2=current1;

current1=current1->column;

if(current1->data==directory)

{

current2->column=current1->column;

delete current1;

break;

}

}

cout<<"directory '"<<directory<<"' deleted.\n"<<endl;

accept\_command();

}

}

}

//function to delete a file from the file system

void delete\_file(string directory,string file)

{

int flag=check\_directory(directory);

if(flag==0)

{

cout<<"directory does not exist...\n"<<endl;

accept\_command();

}

else

{

int flag1=check\_file(directory,file);

if(flag1==0)

{

cout<<"file does not exist...\n"<<endl;

accept\_command();

}

else

{

node1 \*current1=head1;

node1 \*current2=head1;

while(current1->column!=NULL)

{

if(current1->data==directory)

break;

current1=current1->column;

}

while(current1->row!=NULL)

{

current2=current1;

current1=current1->row;

if(current1->data==file)

{

current2->row=current1->row;

delete current1;

break;

}

}

cout<<"file '"<<directory<<"\\"<<file<<"' deleted.\n"<<endl;

accept\_command();

}

}

}

//function to accept command to operate the file manager/system of the OS

void accept\_command()

{

string command=""; // local variable

void validate\_command(string command);

cout<<"root\\:>";

getline(cin,command);

validate\_command(command);

}

////function to command given to operate the file manager/system of the OS

void validate\_command(string command)

{

void command\_words\_calculate(string command, string\* command\_words);

void accept\_command();

string command\_words[4]={""};

command\_words\_calculate(command, command\_words);

if(command\_words[1]=="")

{

cout<<"incomplete command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="create")

{

if(command\_words[1]=="dir")

{

if(command\_words[2]=="")

{

cout<<"directory name missing...\n"<<endl;

}

else

create\_directory(command\_words[2]);

}

else if(command\_words[1]=="file")

{

if(command\_words[2]=="")

cout<<"directory name missing...\n"<<endl;

else if(command\_words[3]=="")

cout<<"file name or directory name missing...\n"<<endl;

else

{

string temp=command\_words[3];

int i,k=0;

for(i=0;i<temp.length();i++)

{

char ch= temp.at(i);

if(ch=='.')

{

k=1;

break;

}

}

if(k==1)

{

create\_file(command\_words[2],command\_words[3]);

}

else

{

cout<<"file type/extension missing...\n"<<endl;

}

}

}

else

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="show")

{

if(command\_words[1]=="dir")

{

show\_directories();

}

else if(command\_words[1]=="files")

{

if(command\_words[2]=="")

{

cout<<"directory name missing...\n"<<endl;

}

else

{

show\_files(command\_words[2]);

}

}

else

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="delete")

{

if(command\_words[1]=="dir")

{

if(command\_words[2]=="")

{

cout<<"directory name missing...\n"<<endl;

}

else

{

delete\_directory(command\_words[2]);

}

}

else if(command\_words[1]=="file")

{

if(command\_words[2]=="")

{

cout<<"directory name missing...\n"<<endl;

}

else if(command\_words[3]=="")

{

cout<<"file name or directory name missing...\n"<<endl;

}

else

{

delete\_file(command\_words[2],command\_words[3]);

}

}

else

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="open")

{

if(command\_words[1]=="file")

{

if(command\_words[2]=="" || command\_words[3]=="")

cout<<"directory name missing...\n"<<endl;

else

{

int flag=check\_directory(command\_words[2]);

if(flag==0)

cout<<"directory does not exist...\n"<<endl;

else

{

int flag1=check\_file(command\_words[2],command\_words[3]);

if(flag1==0)

cout<<"file does not exist...\n"<<endl;

else

cout<<"file '"<<command\_words[2]<<"\\"<<command\_words[3]<<"' opened.\n"<<endl;

}

}

}

else

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="close")

{

if(command\_words[1]=="file")

{

if(command\_words[2]=="" || command\_words[3]=="")

cout<<"directory name missing...\n"<<endl;

else

{

int flag=check\_directory(command\_words[2]);

if(flag==0)

cout<<"directory does not exist...\n"<<endl;

else

{

int flag1=check\_file(command\_words[2],command\_words[3]);

if(flag1==0)

cout<<"file does not exist...\n"<<endl;

else

cout<<"file '"<<command\_words[2]<<"\\"<<command\_words[3]<<"' closed.\n"<<endl;

}

}

}

else

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

else if(command\_words[0]=="clr")

{

if(command\_words[1]=="scr"&&command\_words[2]==""&&command\_words[3]=="")

{

display();

}

else

{

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

}

else if(command=="system exit")

{

main\_screen1();

}

else if(command=="process manager")

{

process\_manager();

}

else if(command=="i/o manager")

{

io\_manager();

}

else if(command=="memory manager")

{

memory\_manager();

}

else

{

cout<<"'"<<command\_words[0]<<"' is not a recognized internal or external command...\n"<<endl;

accept\_command();

}

}

//function to split the command into words

void command\_words\_calculate(string command, string\* command\_words)

{

int i,k=0;

char ch,ch1;

string duplicate="";

for(i=0;i<command.length();i++)

{

ch=command.at(i);

if(ch==' ')

continue;

else

break;

}

command=command.substr(i,command.length());

for(i=0;i<command.length();i++)

{

ch=command.at(i);

if(i!=command.length()-1)

ch1=command.at(i+1);

if(ch==' ')

{

if(i!=command.length()-1&&ch1!=' ')

{

k++;

continue;

}

else if(i==command.length()-1)

{

break;

}

else

continue;

}

else

command\_words[k]=command\_words[k]+ch;

}

}

//start screen of the file manager

void display()

{

void accept\_command();

system("cls");

cout<<" A.B.A.M OPERATING SYSTEM [Beta Version 1.0]\n"<<endl;

cout<<" -------------------------------------------------\n"<<endl;

cout<<" File Manager.\n"<<endl;

cout<<" -------------------------------------------------\n\n\n"<<endl;

accept\_command();

}

//function to implement memory manager of the OS

void memory\_manager()

{

system("cls");

cout<<" A.B.A.M OPERATING SYSTEM [Beta Version 1.0]\n"<<endl;

cout<<" -------------------------------------------------\n"<<endl;

cout<<" Memory Manager.\n"<<endl;

cout<<" -------------------------------------------------\n\n\n"<<endl;

int totalmemory[16];

int partitions[16];

int i;

int choice=0;

int reqd;

int reqdideal=1;

int allocated=0;

int lb=0,ub=0,cntr=0;

int diff=-1;

int j;

int b=0;

int a=0;

int todelete;

int dup;

int k,i2;

int cntr1=0;

//function to implement memory manager of the OS

for(i=0;i<16;i++)

{

totalmemory[i]=9;

partitions[i]=0;

}

//Setting last element of array to 1. i.e. partition at the end of memory

partitions[15]=1;

while(choice!=4)

{

cntr=0;

lb=0,ub=0;

reqdideal=1;

allocated=0;

diff=-1;

cout<<"1.Allocate memory 2.Deallocate Memory 3.Display Memory used and unused 4.Exit ";

cin>>choice;

switch(choice)

{

case 1:cout<<"Enter amount of memory needed ";

cin>>reqd;

while(reqd>reqdideal)

{

reqdideal=reqdideal\*2;

}

//finding out lower and upper bounds and difference between them

for(i=0;i<16;i++)

{

if(partitions[i]==1)

{

lb=ub;

ub=i+1;

diff=ub-lb;

}

if(diff==reqdideal && totalmemory[lb]==9)

{

for(j=lb;j<ub;j++)

{

if(cntr<reqd)

{

totalmemory[j]=1;

}

if(cntr>=reqd)

{

totalmemory[j]=0;

}

cntr++;

allocated=1;

}

}

else if(diff>reqdideal && totalmemory[lb]==9)

{

while(diff!=reqdideal)

{

partitions[lb+((ub-lb-1)/2)]=1;

ub=(ub+lb)/2;

diff=ub-lb;

}

for(j=lb;j<ub;j++)

{

if(cntr<reqd)

totalmemory[j]=1;

else

totalmemory[j]=0;

cntr++;

allocated=1;

}

}

else

continue;

if(allocated==1)

break;

}

//checking if memory has been allocated

cout<<"Memory Index |";

for(i=0;i<16;i++)

{

cout<<i<<" ";

}

cout<<endl;

cout<<"Memory Occupied/Unoccupied/Unusable |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<totalmemory[i]<<" ";

else

cout<<totalmemory[i]<<" ";

}

cout<<endl;

cout<<"Memory Partitions |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<partitions[i]<<" ";

else

cout<<partitions[i]<<" ";

}

cout<<endl;

cout<<endl;

cout<<endl;

cout<<"MEMORY OCCUPIED/UNOCCUPIED/UNUSABLE TABLE KEY"<<endl;

cout<<"1 indicates memory index is currently occupied"<<endl;

cout<<"0 indicates memory index is currently unoccupied but unusable because of current partitioning"<<endl;

cout<<"9 indicates memory index is currently unoccupied and can be used"<<endl;

cout<<endl;

cout<<"MEMORY PARTITIONS TABLE KEY"<<endl;

cout<<"0 indicates no partition at that index"<<endl;

cout<<"1 indicates partition at that index"<<endl;

cout<<endl;

cout<<endl;

break;

case 2:a=0,b=0,cntr1=0;

cout<<"Enter starting position to delete";

cin>>todelete;

if(totalmemory[todelete]==0 || totalmemory[todelete]==9)

{

cout<<"No process resides in this memory location"<<endl;

continue;

}

totalmemory[todelete]=9;

a++;

i=todelete;

while(partitions[i]==0)

{

i++;

totalmemory[i]=9;

a++;

}

cout<<endl;

while(1)

{

b=0;

dup=i;

if(partitions[i]==0)

{

i2=i;

for(k=i2;k<16;k++)

{

if(partitions[k]==1)

{

i=k;

break;

}

}

}

if((i+1)%(2\*a)!=0)

{

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

{

if(totalmemory[j]==9)

b++;

}

if(a==b)

partitions[i]=0;

else

break;

}

else

{

i=i-(2\*a);

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

{

if(totalmemory[j]==9)

b++;

}

if(a==b)

partitions[i]=0;

else

break;

i=dup;

}

a=a\*2;

}

for(i=0;i<16;i++)

{

if(totalmemory[i]==9)

{

cntr1++;

}

}

if(cntr1==16)

{

partitions[7]=0;

}

cout<<"Memory Index |";

for(i=0;i<16;i++)

{

cout<<i<<" ";

}

cout<<endl;

cout<<"Memory Occupied/Unoccupied/Unusable |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<totalmemory[i]<<" ";

else

cout<<totalmemory[i]<<" ";

}

cout<<endl;

cout<<"Memory Partitions |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<partitions[i]<<" ";

else

cout<<partitions[i]<<" ";

}

cout<<endl;

cout<<endl;

cout<<endl;

cout<<"MEMORY OCCUPIED/UNOCCUPIED/UNUSABLE TABLE KEY"<<endl;

cout<<"1 indicates memory index is currently occupied"<<endl;

cout<<"0 indicates memory index is currently unoccupied but unusable because of current partitioning"<<endl;

cout<<"9 indicates memory index is currently unoccupied and can be used"<<endl;

cout<<endl;

cout<<"MEMORY PARTITIONS TABLE KEY"<<endl;

cout<<"0 indicates no partition at that index"<<endl;

cout<<"1 indicates partition at that index"<<endl;

cout<<endl;

cout<<endl;

break;

case 3:cout<<"Memory Index |";

for(i=0;i<16;i++)

{

cout<<i<<" ";

}

cout<<endl;

cout<<"Memory Occupied/Unoccupied/Unusable |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<totalmemory[i]<<" ";

else

cout<<totalmemory[i]<<" ";

}

cout<<endl;

cout<<"Memory Partitions |";

for(i=0;i<16;i++)

{

if(i>=10)

cout<<partitions[i]<<" ";

else

cout<<partitions[i]<<" ";

}

cout<<endl;

cout<<endl;

cout<<endl;

cout<<"MEMORY OCCUPIED/UNOCCUPIED/UNUSABLE TABLE KEY"<<endl;

cout<<"1 indicates memory index is currently occupied"<<endl;

cout<<"0 indicates memory index is currently unoccupied but unusable because of current partitioning"<<endl;

cout<<"9 indicates memory index is currently unoccupied and can be used"<<endl;

cout<<endl;

cout<<"MEMORY PARTITIONS TABLE KEY"<<endl;

cout<<"0 indicates no partition at that index"<<endl;

cout<<"1 indicates partition at that index"<<endl;

cout<<endl;

cout<<endl;

break;

case 4:break;

default:cout<<"Wrong choice "<<endl;

}

}

main\_screen2();

}

//function to implement process manager of the OS

void process\_manager()

{

system("cls");

cout<<" A.B.A.M OPERATING SYSTEM [Beta Version 1.0]\n"<<endl;

cout<<" -------------------------------------------------\n"<<endl;

cout<<" Process Manager.\n"<<endl;

cout<<" -------------------------------------------------\n\n\n"<<endl;

int i,j,t,n,temp1,temp2,temp3,temp4,sumburst=0;

char c;

float avgwt=0,avgtat=0;

cout<<"Enter number of processes"<<endl;

cin>>n;

int art[n],bt[n],complete[n],wt[n],tat[n],ntat[n];

char pname[n];

cout<<"Enter the burst times"<<endl;

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

{

cin>>bt[i];

}

cout<<"Enter the arrival times"<<endl;

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

{

cin>>art[i];

}

cout<<"\n";

for(i=0,c='A';i<n;i++,c++)

{

pname[i]=c;

complete[i]=0;

sumburst=sumburst+bt[i];

}

cout<<"Name "<<" Arrival Time "<<" Burst Time "<<endl;

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

{

cout<<pname[i]<<"\t"<<art[i]<<"\t\t"<<bt[i]<<endl;

}

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

{

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

{

if(art[i]>art[j])

{

temp1=art[i];

art[i]=art[j];

art[j]=temp1;

temp2=bt[i];

bt[i]=bt[j];

bt[j]=temp2;

temp3=pname[i];

pname[i]=pname[j];

pname[j]=temp3;

temp4=complete[i];

complete[i]=complete[j];

complete[j]=temp4;

}

}

}

cout<<"\n";

cout<<"After sorting, the order of processes is: "<<endl;

cout<<"Name "<<" Arrival Time "<<" Burst Time "<<endl;

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

{

cout<<pname[i]<<"\t"<<art[i]<<"\t\t"<<bt[i]<<endl;

}

cout<<"\n";

cout<<"By applying the Response Ratio formula: "<<endl;

cout<< "Name " << " Arrival Time " << " Burst Time " << " Turnaround Time " << " Waiting Time " << " Normalized TAT ";

for(t=art[0];t<sumburst;)

{

float hrr= -9999;

float z;

int next;

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

{

if(art[i]<=t && complete[i]!=1)

{

z=(bt[i] + (t-art[i]-bt[i]))/bt[i];

if(hrr<z)

{

hrr=z;

next=i;

}

}

}

t=t+bt[next];

tat[next]=t-art[next];

wt[next]=tat[next]-bt[next];

avgtat=avgtat+tat[next];

ntat[next]=((float)tat[next]/bt[next]);

complete[next]=1;

avgwt=avgwt+wt[next];

cout<<"\n"<<pname[next]<<"\t"<<art[next];

cout<<"\t\t"<<bt[next]<<"\t\t"<<tat[next];

cout<<"\t\t"<<wt[next]<<"\t\t"<<ntat[next];

}

cout<<"\nAverage waiting time: "<<avgwt/n<<endl;

cout<<"Average turnaround time: "<<avgtat/n<<endl;

main\_screen2();

}

//function to implement I/O manager of the OS

void io\_manager()

{

system("cls");

cout<<" A.B.A.M OPERATING SYSTEM [Beta Version 1.0]\n"<<endl;

cout<<" -------------------------------------------------\n"<<endl;

cout<<" I/O Manager.\n"<<endl;

cout<<" -------------------------------------------------\n\n\n"<<endl;

int n, i, j, temp, processincomplete=0;

cout<<"Enter the number of I/O instructions to be completed"<<endl;

cin>>n;

int deadline[n];

cout<<"Enter the deadlines(in seconds) of each I/O instruction "<<endl;

int complete[n];

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

{

complete[i]=0;

}

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

{

cin>>deadline[i];

}

//Sorting the processes in increasing order of their deadlines

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

{

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

{

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

{

temp= deadline[j];

deadline[j]=deadline[j+1];

deadline[j+1]=temp;

}

}

}

cout<<"Instructions are sorted according to their deadlines and are as follows:"<<endl;

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

{

cout<<"P"<<i+1<<": "<<deadline[i]<<endl;

}

//Creating a duplicate of deadline[] array

int deadlinecopy[n];

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

{

deadlinecopy[i]=deadline[i];

}

cout<<"Enter the time period for each instruction"<<endl;

int period[n];

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

{

cin>>period[i];

}

system("cls");

cout<<"Enter the burst times of each instruction(in seconds)"<<endl;

int burst[n];

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

{

cin>>burst[i];

}

system("cls");

cout<<"Instructions along with their deadlines and their burst times are:\n"<<endl;

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

{

cout<<"P"<<i+1<<" Deadline: "<<deadline[i]<<" Burst time: "<<burst[i]<<endl;

}

cout<<"\n Press e to begin the I/O scheduling algorithm"<<endl;

char begin; cin>>begin;

if(begin=='e')

system("cls");

int count=0; //Used later to check if all instructions have been completed

i=0;

while(1)

{

//Finding out the instruction which is closest to its deadline

int min = 1000;

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

{

if(deadline[j]<min)

min=j;

}

//Letting instruction run for its allotted time period

if((burst[min]-period[min])>=0)

{

burst[min]=burst[min]-period[min];

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

{

deadline[j]=deadline[j]-period[min];

}

}

else

{

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

{

deadline[j]=deadline[j]-burst[min];

}

burst[min]=0;

}

//Checking if any deadline have passed without the respective instruction being completed

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

{

if(deadline[j]<=0 && burst[j]>>0)

{

processincomplete=1;

}

}

//Checking if the current instructions has been completed before its deadline has passed

if(burst[min]==0 && deadline[min]>>0)

{

complete[min]=1;

}

if(processincomplete==1)

break;

//Checking if all instructions have been completed

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

{

if(complete[j]==1)

count=count+1;

}

if(count==n)

break;

i++;

}

for(j=0; j<=10; j++)

{

system("cls");

cout<<"LOADING..."<<(j\*10)<<"%";

}

if(processincomplete==1)

cout<<"\nDeadline scheduler failed to complete as process "<<i<<" failed to meet its deadline"<<endl;

else

cout<<"\nAll instructions were successfully executed"<<endl;

main\_screen2();

}

//command screen of the OS

void main\_screen2()

{

string command; // local variable

cout<<"\nroot\\:>";

getline(cin,command);

if(command=="i/o manager")

io\_manager();

else if(command=="memory manager")

memory\_manager();

else if(command=="file manager")

display();

else if(command=="process manager")

process\_manager();

else if(command=="system exit")

main\_screen1();

else

{

cout<<"'"<<command<<"' is not a recognized internal or external command...\n"<<endl;

main\_screen2();

}

}

//Boot screen of the OS

void main\_screen1()

{

system("cls");

void display();

for(int i=1;i<=50;i++)

{

system("cls");

cout<<"\n\n\t\t\t\t\t\t Loading\n\t\t\t\t ";

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

{

cout<<"²";

}

cout<<"\n\n\t\t\t\t\t "<<2\*i<<"%";

}

cout<<" Complete.\n\n"<<endl;

cout<<" A.B.A.M OPERATING SYSTEM [Beta Version 1.0]\n"<<endl;

cout<<" -------------------------------------------------\n"<<endl;

cout<<" @ A.S.A.R 2020(c). All Rights Reserved.\n"<<endl;

cout<<" -------------------------------------------------\n\n\n"<<endl;

main\_screen2();

}

int main()

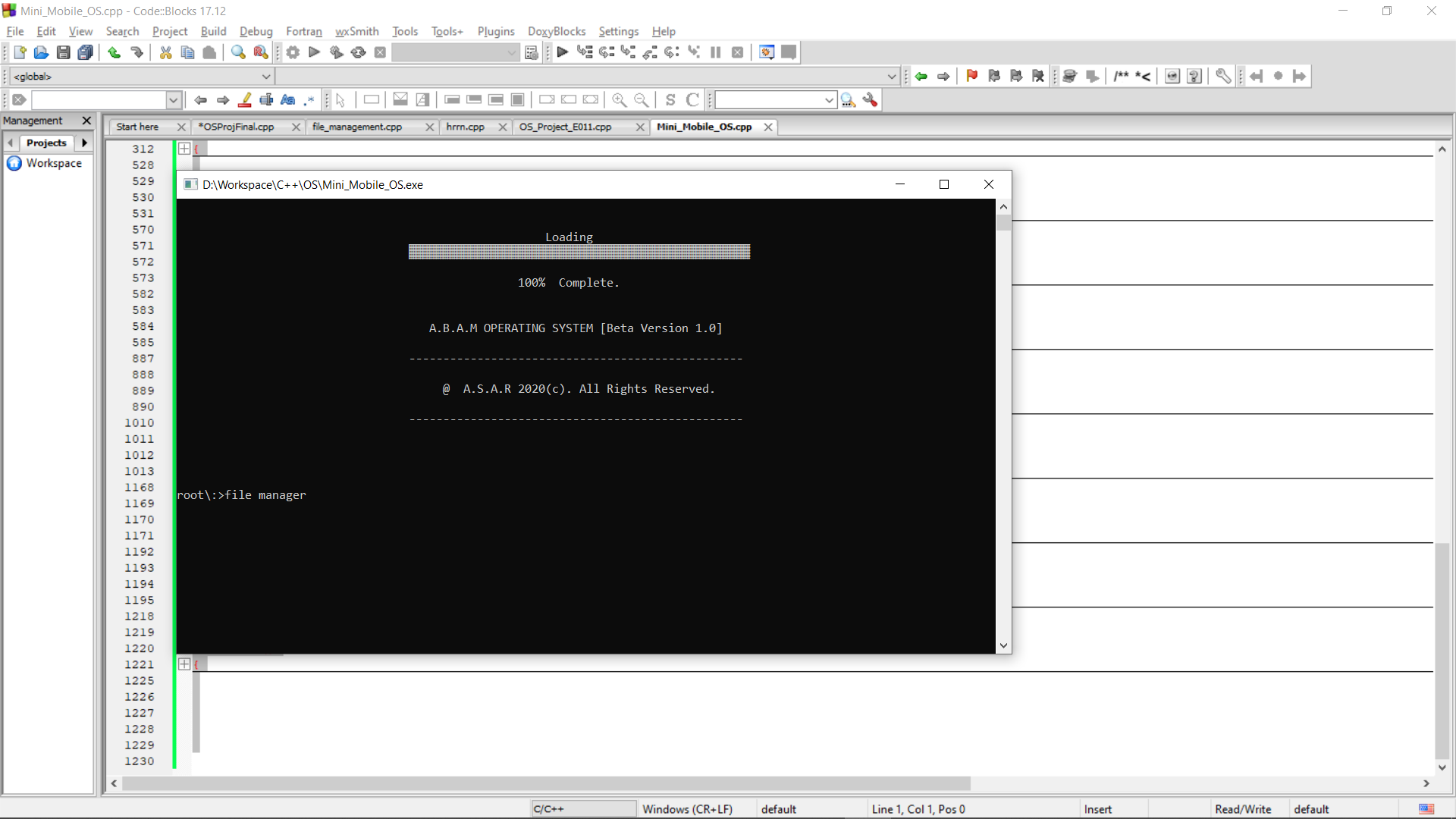
{

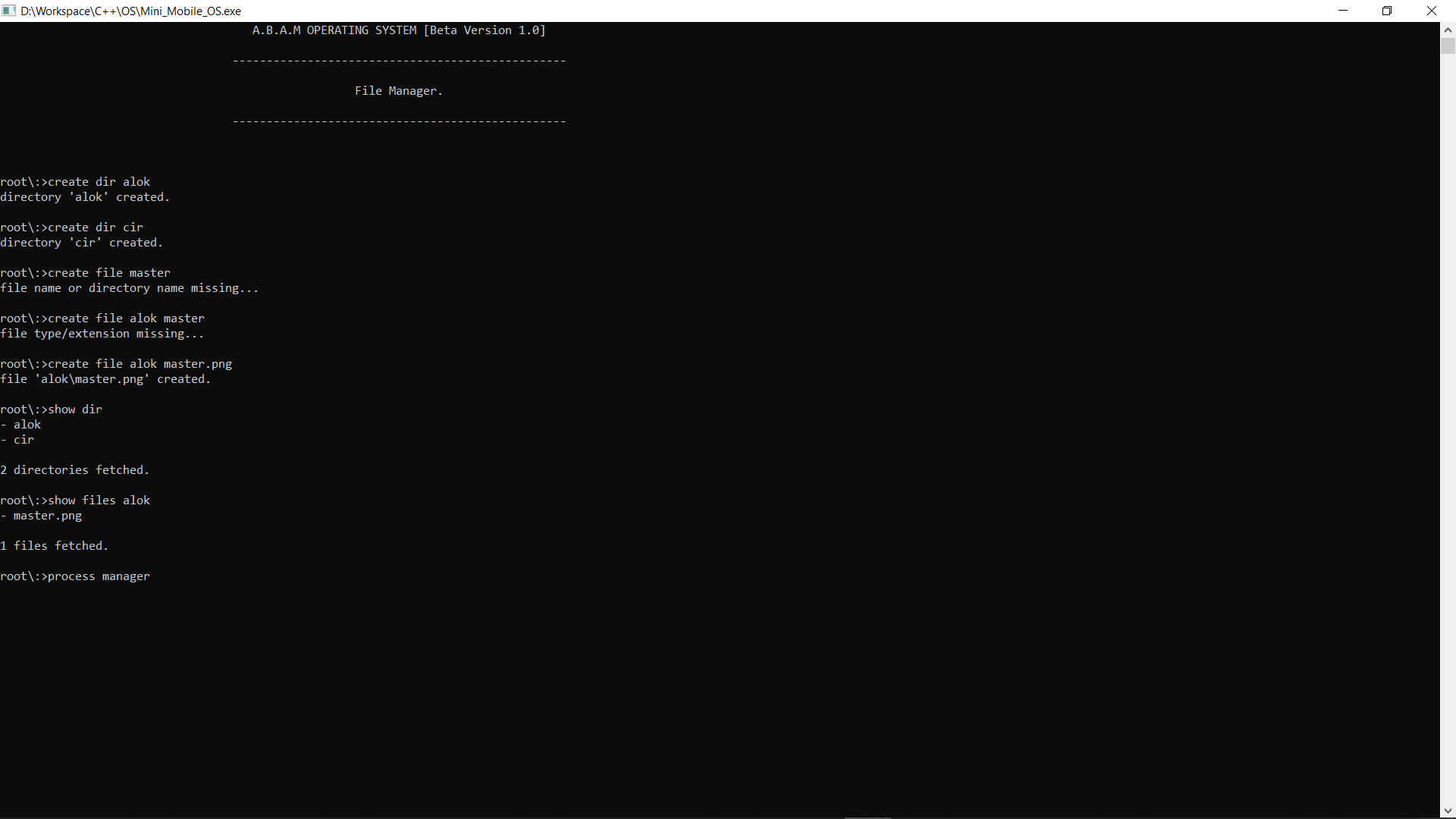
main\_screen1();

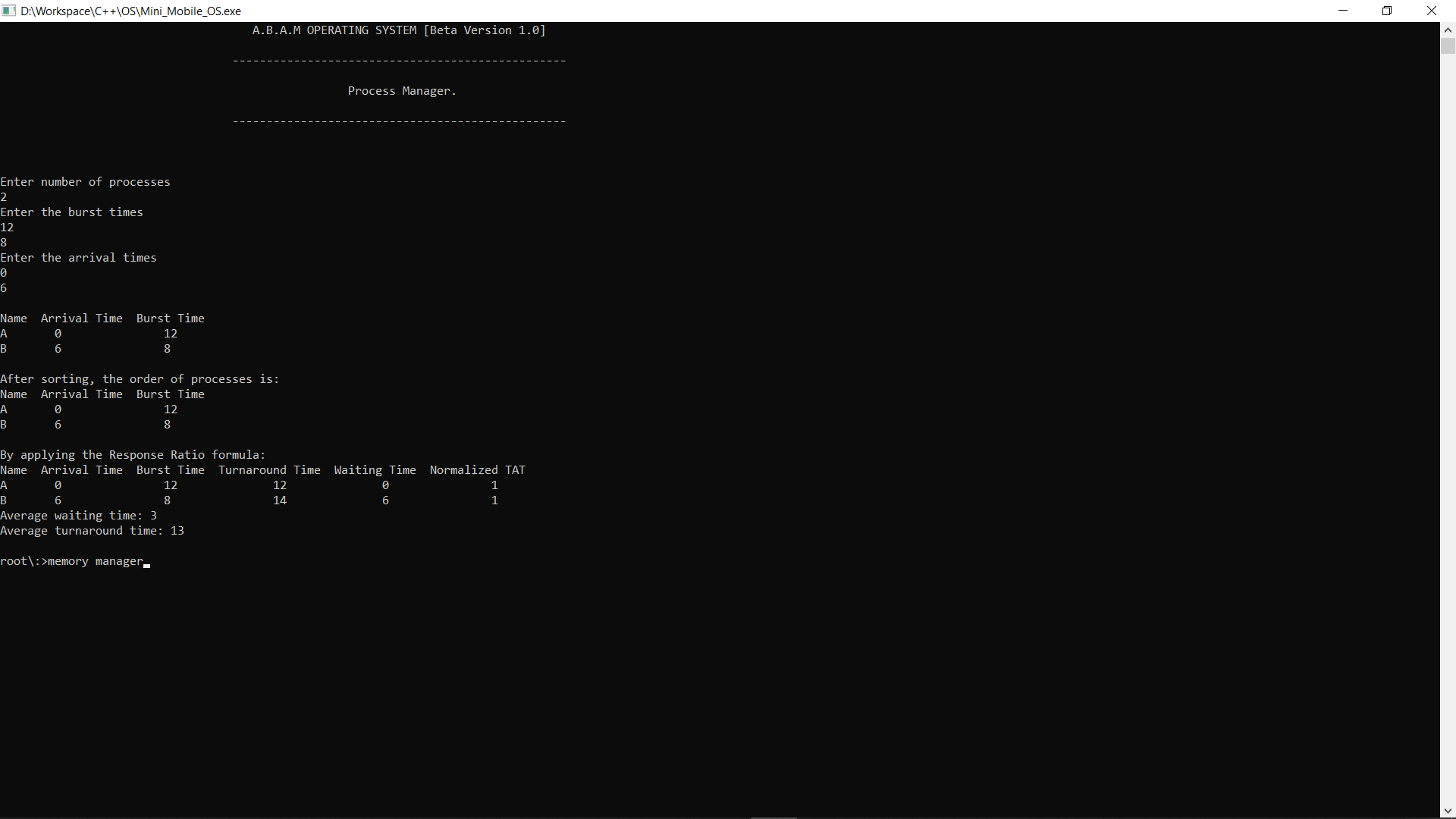
return 0;

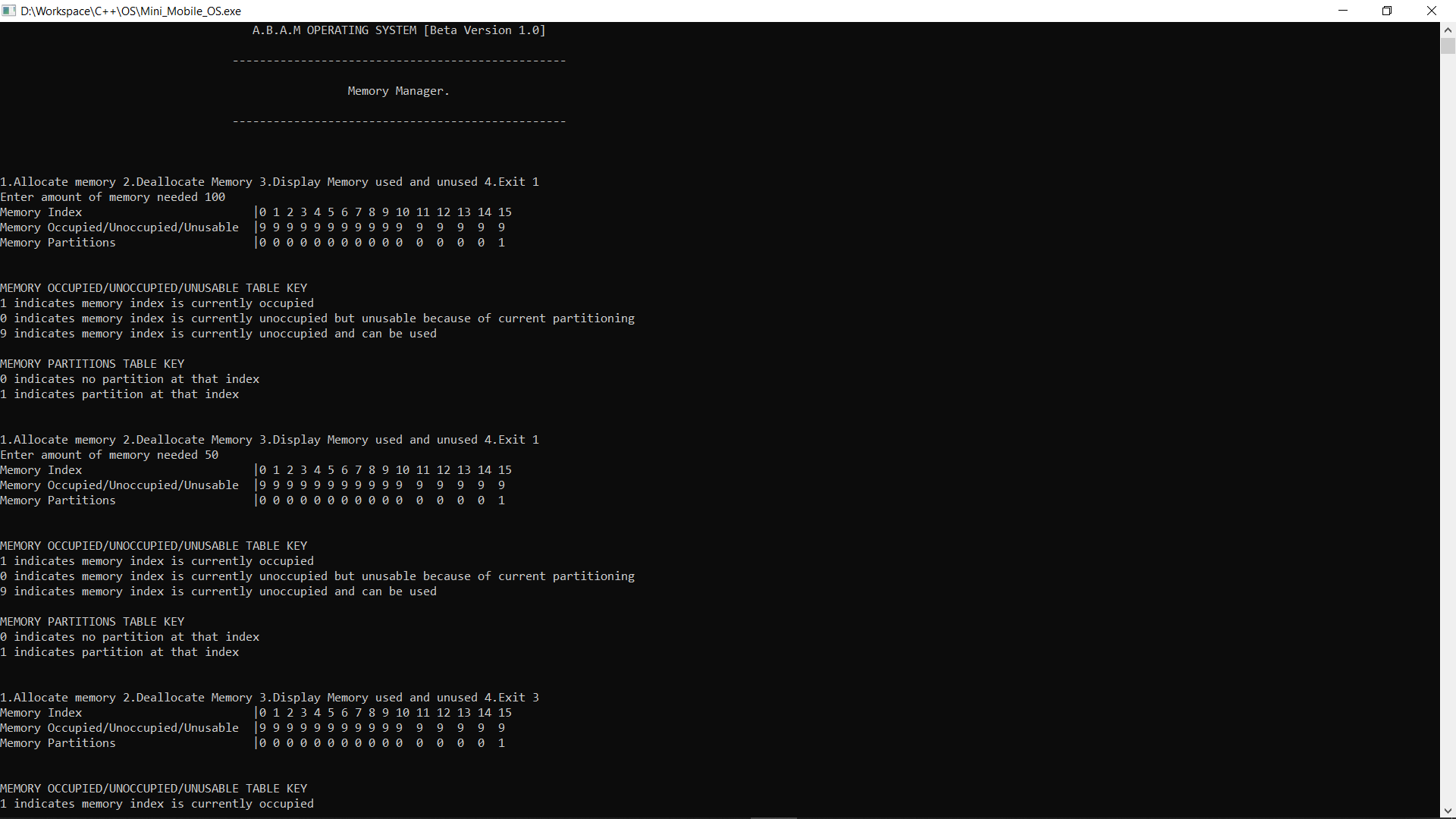
}

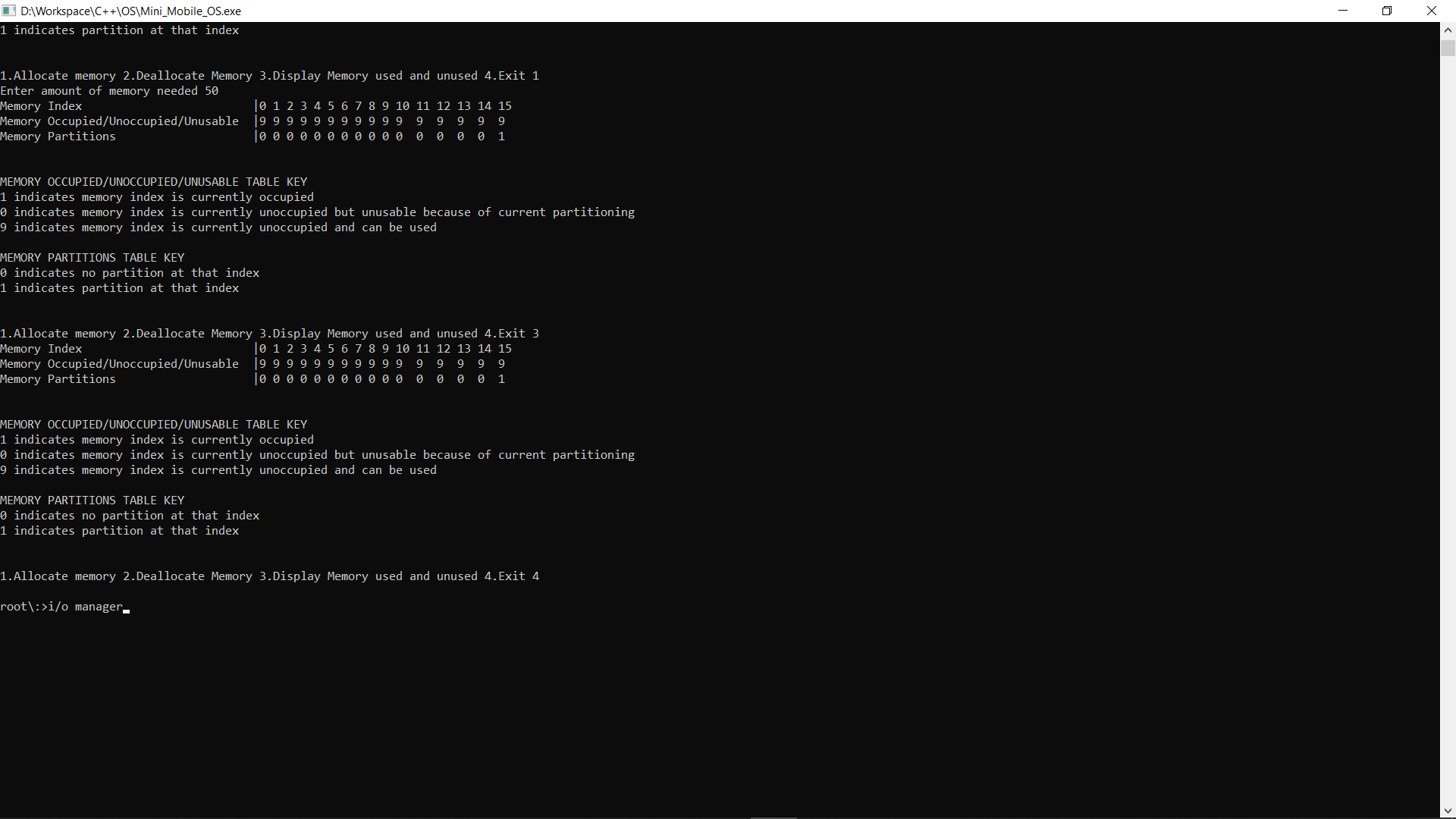
**OUTPUT SCREENSHOTS**

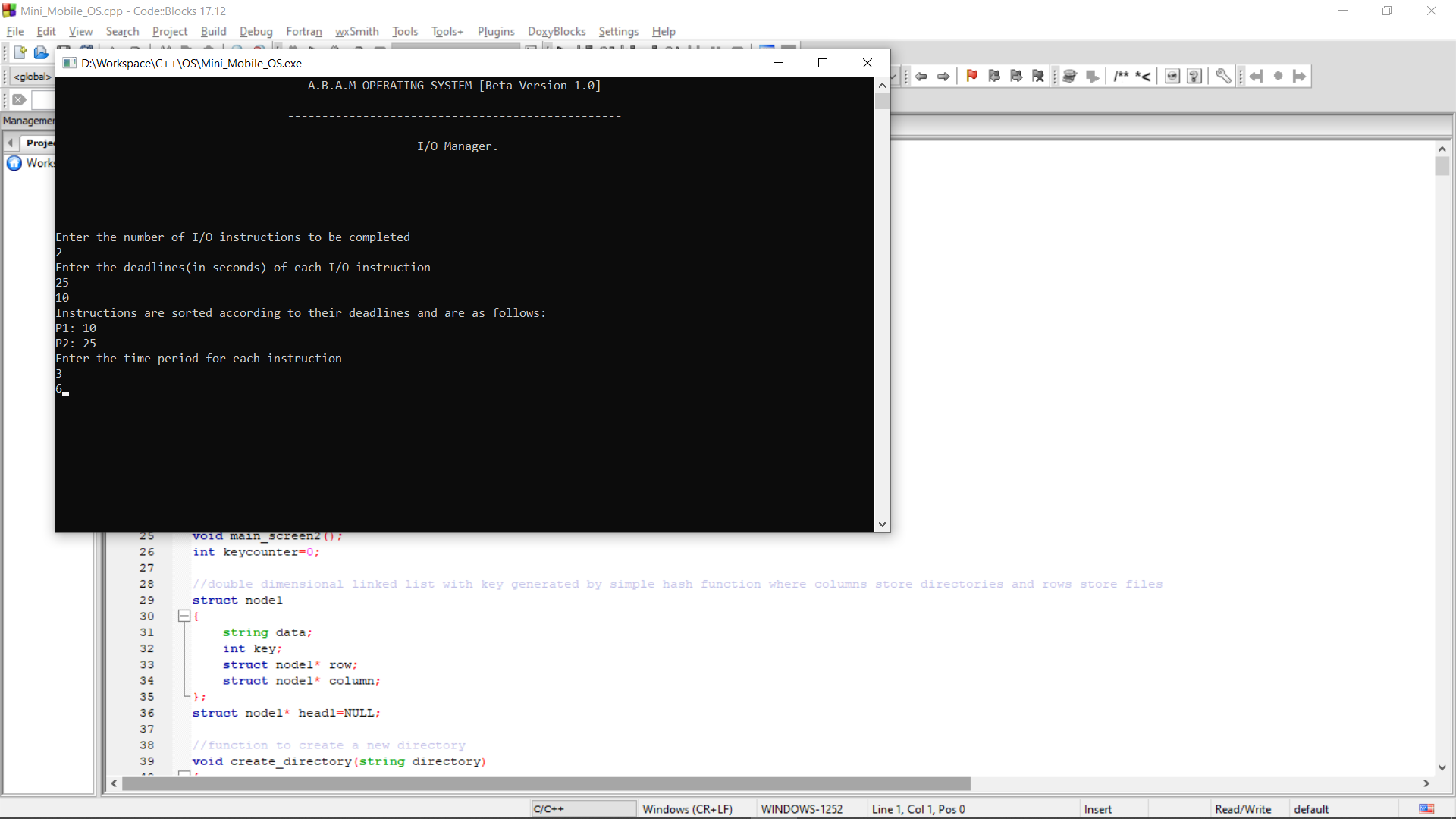


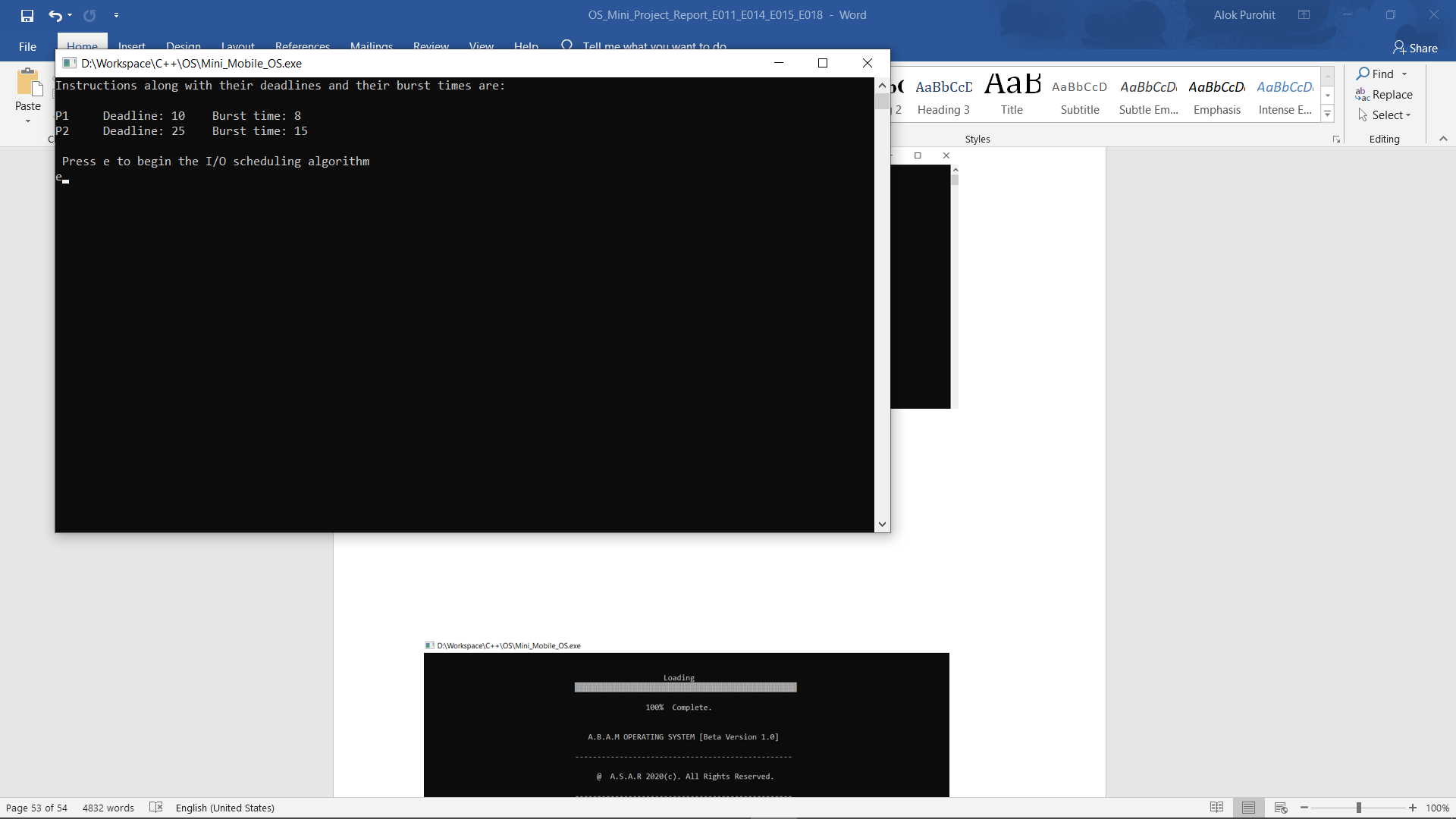


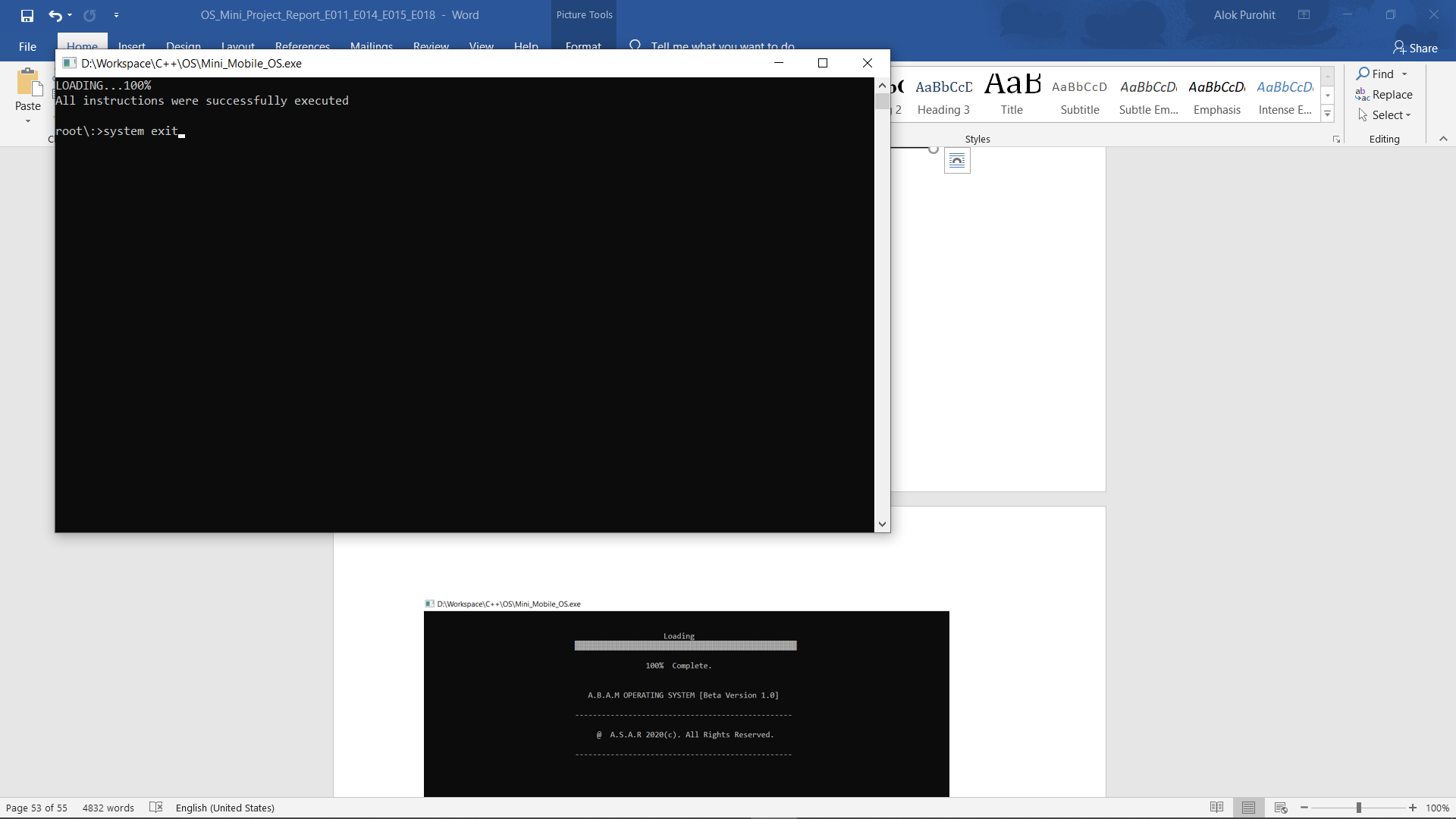


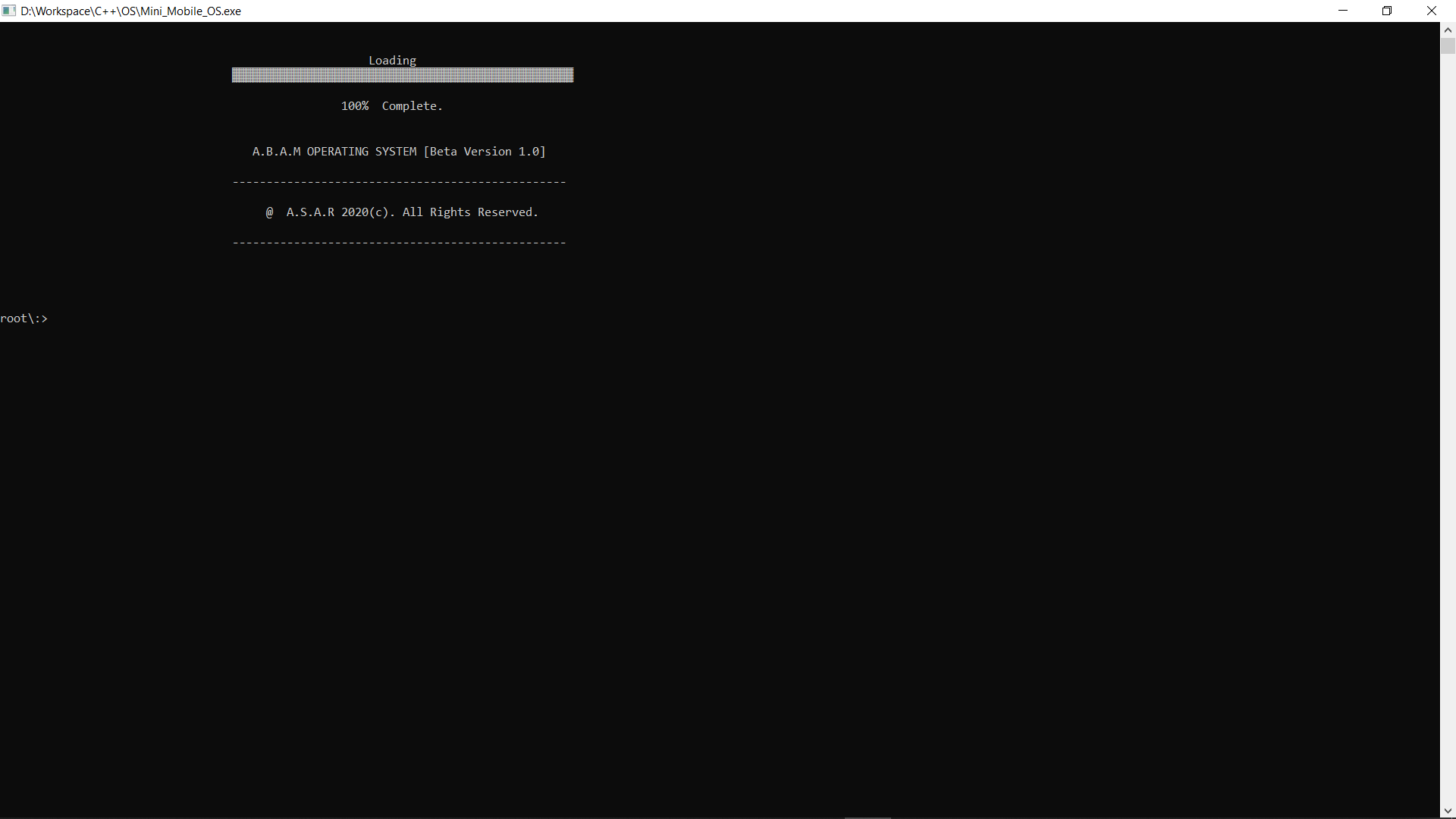












**CONCLUSION**

We designed a Mobile Operating System which involves the use of an efficient algorithm for the functions- File Management, Memory Management, I/O Management and Process Management. We implemented the algorithms in our Mini OS project in C++.

We had to learn about the implemented algorithms as well as the model of an operating system in detail from various online sites due to our limited knowledge in these concepts. While working on this project we grasped these concepts in a very efficient manner as we had to think from several perspectives to implement them in the project. The learning along with the first-hand experience of building an OS will surely help us in our future operations where we will have to deal with highly advanced functionalities which will require a great deal of knowledge and past experience.

**REFERENCES**

<https://www.geeksforgeeks.org/>

<https://www.javatpoint.com/>

<https://www.tutorialspoint.com/index.htm>