Government Engineering College, Idukki

Affiliated to Kerala Technological University | Approved by AICTE

ഗവൺമെന്റ് എഞ്ചിനീയറിംഗ് കോളേജ്, ഇടുക്കി

ITL331 OPERATING SYSTEM AND NETWORK PROGRAMMING LAB MANUAL

Government Engineering College Idukki
Painavu, Kerala - 685603
Phone: 04862-233250/232477
Fax : 04862-232477
Email: info@gecidukki.ac.in

Course Code	Course Name	L	T	P	Credits
ITL331	Operating System and Network Programming	0	0	3	2

Course Outcomes: After completion of the course the student will be able to

CO #	CO Description	Bloom's Taxonomy Level
CO 1	Analyze CPU scheduling algorithms like FCFS, SJF, Round Robin and Priority	Analyze
CO 2	Implement inter process communication and process synchronization problems	Apply
CO 3	Implement memory management schemes – first fit, best fit and worst fit	Apply
CO 4	Implement client-server communication using sockets	Apply
CO 5	Implement MAC protocols	Apply
CO 6	Familiarization of network simulation tool	Understand

Mapping of Course Outcomes with Program Outcomes

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3	3	3	2	-	-	-	-	1	-	3
CO 2	3	3	3	2	1	-	-	-	-	1	-	3
CO 3	3	3	3	2	1	-	-	-	-	1	-	3
CO 4	3	3	3	2	2	-	-	-	-	1	-	3
CO 5	3	3	3	2	2	-	-	-	-	1	-	3
CO 6	2	2	2	2	3	-	-	-	-	1	-	3

List of Experiments and Course Outcomes (COs)

Sl. No.	Experiment	CO
1	Familiarization of Basic Linux Commands	CO 6
2	Familiarization of Network Programming API	CO 6
3	Implementation of Echo Server	CO 4
4	Implementation of Two-way Chat	CO 4
5	Implementation of System Calls in Operating System	CO 2
6	Implementation of Non-preemptive CPU Scheduling Algorithms	CO 1
7	Implementation of Fixed Partition Memory Allocation Methods	CO 3
8	Implementation of Bankers Algorithm for Deadlock Avoidance	CO 1
9	Implementation of Interprocess Communication Using Shared Memory	CO 2

Contents

1. Familiarization of Basic Linux Commands

Experiment 1 – Familiarization of basic Linux commands for directory and file operations such as pwd, ls, cd, mkdir, rmdir, man, rm, touch, man, cp, mv, cat, echo, grep, find, sort, and wc

Aim

To familiarize basic Linux commands for directory and file operations such as pwd, ls, cd, mkdir, rmdir, man, rm, touch, man, cp, mv, cat, echo, grep, find, sort, and wc

Procedure

An Operating System (OS) is a software that acts as an interface between computer hardware components and the user. Every computer system must have at least one operating system to run other programs.

An operating system is a program that acts as a user-computer GUI (Graphical user interface). It controls the execution of all types of applications.

The operating system performs the following functions in a device.

1. Instruction
2. Input/output Management
3. Memory Management4. File Management
5. Processor Management
6. Job Priority
7. Special Control Program
8. Scheduling of resources and jobs
9. Security
10. Monitoring activities
11. Job accounting

Instruction: The operating system establishes a mutual understanding between the various instructions given by the user.

Input/output Management: What output will come from the input given by the user, the operating system runs this program. This management involves coordinating various input and output devices. It assigns the functions of those devices where one or more applications are executed.

Memory Management: The operating system handles the responsibility of storing any data, system

programs, and user programs in memory. This function of the operating system is called memory management.

File Management: The operating system is helpful in making changes in the stored files and in replacing them. It also plays an important role in transferring various files to a device.

Processor Management: The processor is the execution of a program that accomplishes the specified work in that program. It can be defined as an execution unit where a program runs.

Job Priority: The work of job priority is creation and promotion. It determines what action should be done first in a computer system.

Special Control Program: The operating systems make automatic changes to the task through specific control programs. These programs are called Special Control Program.

Scheduling of resources and jobs: The operating system prepares the list of tasks to be performed for the device of the computer system. The operating system decides which device to use for which task. This action becomes complicated when multiple tasks are to be performed simultaneously in a computer system. The scheduling programs of the operating system determine the order in which tasks are completed. It performs these tasks based on the priority of performing the tasks given by the user. It makes the tasks available based on the priority of the device.

Security: Computer security is a very important aspect of any operating system. The reliability of an operating system is determined by how much better security it provides us. Modern operating systems usea firewall for security. A firewall is a security system that monitors every activity happening in the computer and blocks that activity in case of any threat.

Monitoring activities: The operating system takes care of the activities of the computer system during various processes. This aborts the program if there are errors. The operating system sends instant messages to the user for any unexpected error in the input/output device. It also provides security to the system when the operating system is used in systems operated by multiple users. So that illegal users cannot get data from the system.

Job accounting: It keeps track of time & resources used by various jobs and users.

ESSENTIAL LINUX COMMANDS:

cat --- for creating and displaying short files

chmod --- change permissions

cd --- change directory

cp --- for copying files

date --- display date

echo --- echo argument

ftp --- connect to a remote machine to download or upload files

grep --- search file

head --- display first part of file

ls --- see what files you have

lpr --- standard print command

more --- use to read files

mkdir --- create directory

mv --- for moving and renaming files

ncftp --- especially good for downloading files via anonymous ftp.

print --- custom print command

pwd --- find out what directory you are in

rm --- remove a file

rmdir --- remove directory

rsh --- remote shellsetenv --- set an environment variable

sort --- sort file

tail --- display last part of file

tar --- create an archive, add or extract files

telnet --- log in to another machine

wc --- count characters, words, lines

1. To create a file.

Syntax: $ cat>filename

Example: $ cat>ex1

2. To view the content of the file.

Syntax: $ cat filename

Example: $ cat ex1

3. To append some details with the existing details in the file

Syntax: $ cat>>filename

Example: $ cat>>ex1

4. To concatenate multiple files

Syntax: $ cat file1 file2 > file3

Example: $ cat computer compiler>world

5. To know the list of all files in directory

Syntax: $ ls

6. To copy the file to another file

Syntax: $ cp source destination

Example: $ cp ex1 ex2

7. To rename the file

Syntax: $ mv oldfile newfile

Example: $ mv ex1 ex3

8. To delete a file

Syntax: $ rm filename

Example: $ rm ex19. To delete all files

Syntax: $ rm *

10. To create a directory

Syntax: $ mkdir dirname

11. To change the name of the directory

Syntax: $ cd dirname

12. To remove the directory

Syntax: $ rmdir dirname

Example: $ rmdir flower

13. Echo

i. To display the filename starting with single letter

Syntax: $ echo?

ii. To display the filename starting with two letters

Syntax: $ echo??

iii. To display the filename starting with the letter f

Syntax: $ echo f*

iv. To display the filename ending with letter f.

Syntax: $ echo *f

We u can use the find command to search for files and directories based on their permissions, type, date, ownership, size, and more. It can also be combined with other tools such as grep or sed

find [options] [path...] [expression]

Expected Output

Your output should display the basic linux commands for directory and file operations such as pwd, ls, cd, mkdir, rmdir, man, rm, touch, man, cp, mv, cat, echo, grep, find, sort, and wc, along with their result.

2. Familiarization of Network Programming API

Experiment 2 – Familiarization of network programming API in Java for socket programming

Aim

To familiarize network programming API in Java for socket programming

Procedure

Client-Side Programming

Establish a Socket Connection

To connect to another machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port. The java.net.Socket class represents a Socket. To open a socket:

Socket socket = new Socket("127.0.0.1", 5000)

- The first argument –**IP address of Server**. (127.0.0.1 is the IP address of localhost, where code will run on the single stand-alone machine).
- The second argument –**TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

Communication

To communicate over a socket connection, streams are used to both input and output the data.

Closing the connection

The socket connection is closed explicitly once the message to the server is sent.

Server-Side Programming

Establish a Socket Connection

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

Communication

getOutputStream() method is used to send the output through the socket.

Close the Connection

After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Expected Output

Your output should display the server and client terminal windows along with appropriate messages showing connection establishment, message passing and connection close.

3. Implementation of Echo Server

Experiment 3 – Implementation of an echo server using client-server communication and socket programming in Java

Aim

To implement an echo server using client-server communication and socket programming in Java

Procedure

Java Socket programming is used for communication between the applications running on different JRE. Java Socket programming can be connection-oriented or connection-less. Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number

Server.java

```
import java.net.*;
import java.io.*;
public class Server {
public static void main(String args[]) throws Exception,UnknownHostException{
ServerSocket ss=new ServerSocket(8088);
Socket s=ss.accept();;
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
String str="",str2="";
while(str!="stop")
{
System.out.println("Waiting for client's Reply...");
str=din.readUTF();
System.out.println("Client: "+str);
```

```
System.out.println("Enter Message:");
str2=br.readLine();
dout.writeUTF(str2);
dout.flush();
}
din.close();
s.close();
ss.close();
}
}
```

Client.java

```
import java.net.*;
import java.io.*;
public class Client {
public static void main(String[] args) throws Exception {
Socket s=new Socket("localhost",8088);
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
String str="",str2="";
while(!str.equals("stop")){
System.out.println("\nEnter Response: ");
str=br.readLine();
dout.writeUTF(str);
dout.flush();
System.out.println("Waiting for Server's Reply...");
str2=din.readUTF();
System.out.println("Server says: "+str2);
}
dout.close();
s.close();
}
}
```

Expected Output

Your output should be displayed in two terminals – one client terminal and another one, the server terminal. Whatever you type in through the client terminal should be echoed back by the server and displayed in the client terminal.

4. Implementation of Two-way Chat

Experiment 4 – Implementation of a two-way chat application using client-server communication and socket programming in Java

Aim

To implement a two-way chat application using client-server communication and socket programming in Java

Procedure

It is possible to send data from the server and receive a response from the client. Similarly, the client can also send and receive data to-and-from.

Below are the various steps to do so:

- We need additional streams both at server and client. For example, to receive data into the server, it is a better idea to use a BufferedReader object, as shown in the following code snippet:

  InputStream obj = s.getInputStream();
  BufferedReader br = new BufferedReader(new InputStreamReader(obj);

- Then read() or readLine() methods of the BufferedReader object can be used to read data. To send data from the client we can take the help of the DataOutputStream class as shown in the following code snippet:

  OutputStream obj + s.getOutputStrean();
  DataOutputStream dos = new DataOutputStream(obj);

- Then, the writeBytes() method of the DataOutputStream class can be used to send strings in the form of a group of bytes. To establish the two-way communication between a client and server perform the following steps:
  - Creating the Server Program: Create a class named Server.java to create server such that the server receives data from the client using a BufferedReader object and then sends a reply to the client using a PrintStream object.

Server.java

```
import java.io.*;
import java.net.*;
 class Server2 {
    public static void main(String args[])
     throws Exception
```

```
{

    // Create server Socket
    ServerSocket ss = new ServerSocket(888);
      // connect it to client socket
    Socket s = ss.accept();
    System.out.println("Connection established");
      // to send data to the client
    PrintStream ps
        = new PrintStream(s.getOutputStream());
      // to read data coming from the client
    BufferedReader br
        = new BufferedReader(
            new InputStreamReader(
                s.getInputStream()));
      // to read data from the keyboard
    BufferedReader kb
        = new BufferedReader(
            new InputStreamReader(System.in));
      // server executes continuously
    while (true) {
          String str, str1;
          // repeat as long as the client
        // does not send a null string
          // read from client
        while ((str = br.readLine()) != null) {
            System.out.println(str);
            str1 = kb.readLine();
              // send to client
            ps.println(str1);
        }
          // close connection
        ps.close();
        br.close();
```

```
        kb.close();
        ss.close();
        s.close();
          // terminate application
        System.exit(0);
       } // end of while
   }
}
```

Creating the Client Program: Create a client, named Client2.Java, which first connects to a server, then starts the communication by sending a string to the server. The server sends a response to the client. When ‘exit’ is typed at the client side, the program terminates.

Client.java

```
import java.io.*;
import java.net.*;
 class Client {
    public static void main(String args[])
     throws Exception
  {
       // Create client socket
     Socket s = new Socket("localhost", 888);
       // to send data to the server
     DataOutputStream dos
        = new DataOutputStream(
           s.getOutputStream());
      // to read data coming from the server
     BufferedReader br
        = new BufferedReader(
           new InputStreamReader(
              s.getInputStream()));

     // to read data from the keyboard
     BufferedReader kb
        = new BufferedReader(
           new InputStreamReader(System.in));
```

```
        String str, str1;
         // repeat as long as exit
        // is not typed at client
        while (!(str = kb.readLine()).equals("exit")) {
            // send to the server
           dos.writeBytes(str + "\n");
            // receive from the server
           str1 = br.readLine();
            System.out.println(str1);
        }
         // close connection.
        dos.close();
        br.close();
        kb.close();
        s.close();
    }
}
```

Expected Output

Your output should be displayed in two terminals – one client terminal and another one, the server terminal. Whatever you type in through the client terminal should be displayed in the server terminal and vice-versa.

5. Implementation of System Calls in Operating System

Experiment 5 – Implementation of the following system calls in Operating System using C programming - fork, exec, getpid, exit, wait, close, stat, opendir, and readdir

Aim

To implement the following system calls in Operating System using C programming - fork, exec, getpid, exit, wait, close, stat, opendir, and readdir.

Procedure

The “fork()” system call

A process is almost an identical clone of the parent The fork() is called once, but returns twice! After fork()both the parent and the child are executing the same program. The exec() call replaces a current process’ image with new one (i.e. loads a new program within current process).

The “wait()” system call

Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (terminate). When the child process dies, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution. A child process that dies but is never waited on by its parent becomes a zombie process. Such a process continues to exist as an entry in the system process table even though it is no longer an actively executing program.

The “exit()” system call

This call gracefully terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the zombie state. By calling wait(), the parent cleans up all its zombie children. When the child process dies, an exit status is returned to the operating system and a signal is sent to the parent process. The exit status can then be retrieved by the parent process via the wait system call Stat is a system call that you can use to get information about files - information that is contained in the files' inodes.

Functions for working with directories

A directory can be read as a file by anyone whoever has reading permissions for it. Writing a directory as a file can only be done by the kernel. The structure of the directory appears to the user as a succession of structures named directory entries. A directory entry contains, among other information, the name of the

file and the i-node of this. For reading the directory entries one after the other we can use the following functions:

#include <sys/types.h>

#include <dirent.h>

DIR opendir(const char* pathname);*

struct dirent readdir(DIR* dp);*

void rewinddir(DIR dp);*

int closedir(DIR dp);*

The opendir() function opens a directory. It returns a valid pointer if the opening was successful and NULL otherwise. The readdir function, at every call, reads another directory entry from the current directory. The first readdir will read the first directory entry; the second call will read the next entry and so on. In case of a successful reading the function will return a valid pointer to a structure of type directory and NULL otherwise (in case it reached the end of the directory, for example). The rewinddir function repositions the file pointer to the first directory entry (the beginning of the directory). The closedir function closes a previously opened directory. In case of an error it returns the value -1.The structure dirent is defined in the dirent.h file. It contains at least two elements:

struct dirent {

ino_t d_fileno;

// i-node nr.

char d_name[MAXNAMLEN + 1]; // file name

}

Expected Output

The program should successfully execute and display the output of the following system calls in Operating System using C programming - fork, exec, getpid, exit, wait, close, stat, opendir, and readdir.

6. Implementation of Non-preemptive CPU Scheduling Algorithms

Experiment 6 – Implementation of the following non-preemptive CPU scheduling algorithms in C programming to find average turnaround time and average waiting time:

(a) FCFS (b) SJF (c) Round Robin (d) Priority

Aim

To implement the following non-preemptive CPU scheduling algorithms in C programming to find average turnaround time and average waiting time

(a) FCFS (b) SJF (c) Round Robin (d) Priority

Procedure

First Come First Serve (FCFS)

In priority scheduling algorithm each process has a priority associated with it and as each process hits the queue, it is stored in based on its priority so that process with higher priority are dealt with first. It should be noted that equal priority processes are scheduled in FCFS order.

Algorithm

Step 1: Start

Step 2: Input the number of processes

Step 3: Input the burst time, arrival time of each process

Step 4: Set the waiting of the first process as ‗0‘and its burst time as its turnaround time

Step 5: for each process calculate a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b).

Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate a) Average waiting time = Total waiting Time / Number of process b) Average

Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop

Program

```
#include<stdio.h>
#include<conio.h>
main()
{
```

```
int bt[20], wt[20], tat[20], i, n;

float wtavg, tatavg; clrscr();

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

scanf("%d", &n); for(i=0;i<n;i++)

{

printf("\nEnter Burst Time for Process %d -- ", i);scanf("%d", &bt[i]); } wt[0] = wtavg = 0; tat[0] = tatavg =
bt[0];

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

{

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

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

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

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

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

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

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

getch();

}
```

Shortest Job First (SJF)

In shortest job first scheduling algorithm, the processor selects the waiting process with the smallest execution time to execute next.

Algorithm

Step 1: Start

Step 2: Input the number of processes

Step 3: Input the burst time, arrival time of each process

Step 4: Sort the process according to lowest to highest burst time

Step 5: Set the waiting of the first process as ‗0‘and its burst time as its turnaround time

Step 6: for each process calculate a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b).

Turnaround time (n)= waiting time(n)+Burst time(n)

Step 7: Calculate a) Average waiting time = Total waiting Time / Number of process b) Average

Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop

Program

```
#include<stdio.h>
#include<conio.h>
main(){
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
```

```
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

Round Robin

The queue structure in ready queue is of First In First Out (FIFO) type. A fixed time is allotted to every process that arrives in the queue. This fixed time is known as time slice or time quantum. The first process that arrives is selected and sent to the processor for execution. If it is not able to complete its execution within the time quantum provided, then an interrupt is generated using an automated timer. The process is then stopped and is sent back at the end of the queue. However, the state is saved and context is thereby stored in memory. This helps the process to resume from the point where it was interrupted. The scheduler selects another process from the ready queue and dispatches it to the processor for its execution. It is executed until the time Quantum does not exceed. The same steps are repeated until all the process are finished. The round robin algorithm is simple and the overhead in decision making is very low. It is the best scheduling algorithm for achieving better and evenly distributed response time.

Algorithm

Step 1: Start

Step 2: Input the number of processes

Step 3: Input the burst time, arrival time of each process

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: for each process calculate a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b). Turnaround time (n)= waiting time(n)+Burst time(n)

Step 7: Calculate a) Average waiting time = Total waiting Time / Number of process b) Average Turnaround time = Total Turnaround Time / Number of processStep

8: Stop

Program

```
#include<stdio.h>
#include<conio.h>
main()
```

```
{
Int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]); ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<max/t;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t)
{
tat[i]=temp+bu[i];
temp=temp+bu[i]; bu[i]=0;
}
else
{
bu[i]=bu[i]-t;temp=temp+t;
}
for(i=0;i<n;i++)
{
wa[i]=tat[i]- ct[i];
att+=tat[i]; awt+=wa[i];
}
```

```
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
}
```

Priority Scheduling

In priority scheduling algorithm each process has a priority associated with it and as each process hits the queue, it is stored in based on its priority so that process with higher priority are dealt with first. It should be noted that equal priority processes are scheduled in FCFS order.

Algorithm

Step 1: Start

Step 2: Input the number of processes

Step 3: Input the burst time, arrival time of each process

Step 4: Sort the ready queue according to the priority number

Step 5: Set the waiting of the first process as ‗0‘ and its burst time as its turnaround time

Step6: Arrange the processes based on process priority

Step 7: for each process calculate a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b). Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate a) Average waiting time = Total waiting Time / Number of process b) Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop

Program

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg; clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++)
```

```
{
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i);
scanf("%d %d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++}
if(pri[i] > pri[k])
{
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f",tatavg/n);
getch();
}
```

Expected Output

The output should be displayed in a table format showing the waiting time and turnaround time of all processes in the system. The average waiting time and average turnaround time should also be printed.

7. Implementation of Fixed Partition Memory Allocation Methods

Experiment 7 – Implementation of the following memory allocation methods for fixed partition using doubly linked list in C programming:

(a) First fit (b) Worst Fit (c) Best Fit

Aim

To implement the following memory allocation methods for fixed partition using doubly linked list in C programming:

(a) First fit (b) Worst Fit (c) Best Fit

Procedure

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

First Fit

Program

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highes t=0; static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - Worst Fit");
```

```
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");for(i=1;i<=nb;i++)
{
printf(
"Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]
-
f[i];
if(temp>=0)
if(highest<temp)
{
}
}
frag[i]=highest;
bf[ff[i]]=1; highest=0;
}
ff[i]=j;
highest=temp;
```

```
}
printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)printf("\n%d\t\t%d\t\t%d\tt%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
}
```

Best Fit

Program

```
#include<stdio.h>
#include<conio.h>
#define max
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
clrscr();
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
printf("Block %d:",i);
scanf("%d",&b[i]);
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
```

```
{if(
bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
if(lowest>temp)
{
ff[i]=j;
lowest=temp;
}
}}
frag[i]=lowest; bf[ff[i]]=1;
lowest=10000;
}
printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

Worst Fit

Program

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,t
emp; static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme – Worst Fit");printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
```

```
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

Expected Output

The program should display the process requests and the blocks allocated to each of them.

8. Implementation of Bankers Algorithm for Deadlock Avoidance

Experiment 8 – Implementation of Bankers algorithm for deadlock avoidance

Aim

To implement Bankers algorithm for deadlock avoidance

Procedure

Deadlock is a situation where in two or more competing actions are waiting for the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether them allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$
- Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task. Banker's algorithm consists of Safety algorithm and Resource request algorithm.

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, 3, 4….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 a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$ <= $Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available

Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Requesti

$Allocation_i$ = $Allocation_i$ + $Request_i$

$Need_i$ = $Need_i$– $Request_i$

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

Program

```
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here
    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
```

```
                        int flag = 0;
                        for (j = 0; j < m; j++) {
                                if (need[i][j] > avail[j]){
                                        flag = 1;
                                        break;
                                }
                        }

                        if (flag == 0) {
                                ans[ind++] = i;
                                for (y = 0; y < m; y++)
                                        avail[y] += alloc[i][y];
                                f[i] = 1;
                        }
                }
        }
}
int flag = 1;
for(int i=0;i<n;i++)
{
if(f[i]==0)
{
        flag=0;
        printf("The following system is not safe");
        break;
}
}

if(flag==1)
{
printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
```

```
        printf(" P%d", ans[n - 1]);
        }
        return (0);
}
```

Expected Output

The program should display the Need matrix, as well as whether the system is in safe state or not. If the system is in safe state, the safe sequence(s) should also be printed.

9. Implementation of Interprocess Communication Using Shared Memory

Experiment 9 –Implementation of Interprocess Communication Using Shared Memory

Aim

To implement interprocess communication (IPC) in C programming language using shared memory concept.

Procedure

Shared Memory is the fastest inter-process communication (IPC) method. The operating system maps a memory segment in the address space of several processes so that those processes can read and write in that memory segment. The overview is as shown below:

Two functions: shmget() and shmat() are used for IPC using shared memory. Shmget() function is used to create the shared memory segment while shmat() function is used to attach the shared segment with the address space of the process.

Syntax (shmget()):

#include <sys/ipc.h>

#include <sys/shm.h>int shmget(key_t key, size_t size, int shmflg);

The first parameter specifies the unique number (called key) identifying the shared segment. The second parameter is the size of the shared segment e.g. 1024 bytes or 2048 bytes. The third parameter specifies the permissions on the shared segment. On success the shmget() function returns a valid identifier while on failure it return -1.

Syntax (shmat()):

#include <sys/types.h>

#include <sys/shm.h>

*void *shmat(int shmid, const void *shmaddr, int shmflg);*

shmat() is used to attach the created shared segment with the address space of the calling process. The first parameter here is the identifier which shmget() function returns on success. The second parameter is the address where to attach it to the calling process. A NULL value of second parameter means that the

system will automatically choose a suitable address. The third parameter is ‘0’ if the second parameter is NULL, otherwise, the value is specified by SHM_RND.

Program

// This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with
key 2345, having size 1024 bytes. IPC_CREAT is used to create the shared segment if it does not
exist. 0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the
segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

// This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

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

```
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

Expected Output

The program should display the data that is written to the shared memory and also data that is read from the shared memory. In addition, to make sure that both processes are accessing the same memory segment, the HEX address of the shared memory segment should also be printed.