# EX NO: 1 STUDY OF BASIC UNIX COMMANDS DATE:

**AIM:**

To study the basic UNIX commands.

# COMMANDS:

**FILE HANDLING COMMANDS:**

# Syntax:

$ cat > filename

# Description:

This command is used to create a file.

# Syntax:

$ cat filename

# Description:

This command is used to view the contents of the file.

# Syntax:

$ cat >> filename

# Description:

This command is used to add the contents to the existing file.

# Syntax:

$ cp file1 file2

# Description:

This command is used to copy the contents of one file to another file.

# Syntax:

$ rm filename

# Description:

This command is used to remove the file.

# Syntax:

$ mv file1 file2

# Description:

This command is used to move the file from one place to another. It removes the specified file from the original location.

# Syntax:

$ ls

**Description:** It is used to list all files and directories in the system.

# Syntax:

$ wc filename

# Description:

This command is used to count the number of words, lines and characters in the file.

# FILE SYSTEM COMMANDS:

1. **Syntax:**

$ pwd

# Description:

pwd – print working directory. It displays the current working directory.

# Syntax:

$ mkdir dirname

# Description:

This command is used to create a new directory.

# Syntax:

$ rmdir dirname

# Description:

This command is used to remove the directory.

# Syntax:

$ cd

# Description:

This command is used to change the current working directory.

# Syntax:

$ tput clear

# Description:

This command is used to clear the screen.

# GENERAL PURPOSE COMMANDS:

1. **Syntax:**

$ date

# Description:

This command is used to display the current date with day, month &

time.

# Syntax:

$ echo text

# Description:

This command is used to print the message on the screen.

# Syntax:

$ cal month or year

# Description:

This command is used to display the specified month or year calendar.

# Syntax:

$ bc

# Description:

It includes the calculator.

# Syntax:

$ who

# Description:

This command is used to display the data about all the users who are currently logged into the system.

# Syntax:

$ who am i

# Description:

This command identifies the user and lists the user name, terminal line, the date and time of login.

# Syntax:

$ tty

# Description:

tty - teletype. It is used to know the terminal name that we are using.

# Syntax:

$ passwd

# Description:

This command is used to change the password.

# FILTER COMMANDS:

1. **Syntax:**

$ head filename

# Description:

This command is used to display the first 10 lines of a file.

# Syntax:

$ tail filename

# Description:

This command is used to display the last 10 lines of a file from end.

# Syntax:

$ grep pattern file(s)

# Description:

grep – Global Regular Expression and Print. It is used to search and print specified patterns from a file.

# Syntax:

$ sort filename

# Description:

This command is used to sort the contents of the file.

# Syntax:

$ cmp file1 file2

# Description:

This command is used to compare the 2 given files.

# Syntax:

$ comm file1 file2

# Description:

This command is used to compare each line in the file1 with the corresponding line in the file2.

# Syntax:

$ diff file1 file2

# Description:

This command is used to displays the difference between 2 files.

# RESULT:

Thus the basic UNIX commands were studied successfully.

# EX NO:2(a) FIRST COME FIRST SERVE SCHEDULING DATE:

**AIM**:

To implement the algorithm first come first serve scheduling

# FIRST COME FIRST SERVE SCHEDULING:

**•**



First in, first out (FIFO), also known as first come, first served (FCFS), is the

simplest scheduling algorithm.

# •



FIFO simply queues processes in the order that they arrive in the ready

queue.

**•**



In this, the process that comes first will be executed first and next process

starts only after the previous gets fully execute.

* Easy to implement and understand.
* Average wait time is high.

# ALGORITHM:

Step **1:** Get the CPU burst time and processed for all the processes

Step **2:** Calculate the waiting time such that the process at the head of the Queue first enter in to the CPU.FCFS maintains FIFO queue.

Step **3:** Find the average waiting time

Step **4:** Display all the details for each process

# PROGRAM:

#include<stdio.h> main()

{ char p[10][10];

int b[10],t[10],wt[10],i,j,n,k; float avg,twt=0.0; printf("\tProgram for FCFS\n"); printf("\t \n");

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

printf("\nEnter the process and burst time\n"); for (i=0;i<n;i++)

{ printf("\nEnter process %d name:",i+1); scanf("%s",&p[i]);

printf("\nEnter process %d burst time:",i+1); scanf("%d",&b[i]); }

wt[0]=0;

printf("\nGantt chart for process\n"); printf(" \n\n");

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

{ wt[j]=wt[j-1]+b[j-1]; twt=twt+wt[j-1]; }

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

{

printf("%d---%s---",wt[j],p[j]);

}

printf("%d",wt[n]); printf("\n");

printf("\nCalculating waiting Time\n"); printf(" \n\n");

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

{

printf("Waiting time of %s is %d\n",p[j],wt[j]);

}

printf("\nCalculating Average Time:\n"); avg=twt/(float)n;

printf ("Average time=%f\n", avg);

printf ("Calculating Turn around Time:\n"); k=0;

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

k=k+b[i]+wt[i]; k=k/n;

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

}

# OUTPUT:

Program for FCFS

Enter the number of process : 3

Enter the process and burst time

|  |  |
| --- | --- |
| Enter process 1 name | : a |
| Enter process 1 burst time | : 10 |
| Enter process 2 name | : b |
| Enter process 2 burst time | : 20 |
| Enter process 3 name | : c |
| Enter process 3 burst time Gantt chart for process | : 30 |
| 0---a---10---b---30---c---60  Calculating waiting Time |  |

Waiting time of a is 0 Waiting time of b is 10 Waiting time of c is 30

Calculating Average waiting Time:

Average time =13.333333

Calculating Average Turnaround Time

Average turnaround time =33

# RESULT:

Thus the program to implement First come first serve scheduling was executed successfully.

# EX NO:2(b) SHORTEST JOB SCHEDULING DATE:

**AIM:**

To implement the shortest job scheduling algorithm.

# SHORTEST JOB SCHEDULING:

* Process with less burst time will always execute first.
* This is pre-emptive scheduling algorithm.
* Preferred to minimize waiting time.
* It is much better than FCFS.
* Easy to implement.

# ALGORITHM:

Step **1:** Get the CPU burst time and processed for all the process.

Step **2:** Calculate the waiting time such that the process with the shortest CPU burst time is executed first and the process the largest CPU burst time is executed last.

step **3:** Find the average waiting time

step **4:** Display all the details for each process

# PROGRAM:

#include<stdio.h> #include<string.h> int main ()

{

int b[10], t[10], wt[10], i, j, n; float k, avg, twt=0.0; printf("\tProgram for SJF\n"); printf("\t \n");

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

printf("\nEnter the process and burst time\n"); for(i=0;i<n;i++)

{

printf("\nEnter process %d name:",i+1); scanf("%s", &p[i]);

printf("\nEnter process %d burst time:",i+1); scanf("%d", &b[i]);

}

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

{

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

{

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

{

t[i]=b[i];

b[i]=b[j];

b[j]=t[i]; strcpy(f[i],p[i]);

strcpy(p[i],p[j]);

strcpy( p[j],f[i]);

}

}

}

wt[0]=0;

printf("\nGantt chart for process\n"); printf(" \n\n");

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

{

wt[j]=wt[j-1]+b[j-1]; twt=twt+wt[j-1];

}

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

{

printf("%d...%s...",wt[j],p[j]);

}

printf("%d",wt[n]); printf("\n");

printf("\nCalculating Waiting Time\n"); printf(" \n\n");

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

{

printf("Waiting Time of %s is %d \n",p[j],wt[j]);

}

printf("\nCalculating Average Time\n"); avg=twt/(float)n;

printf("Average Time=%f \n",avg); printf("Calculating Turnaround Time\n"); k=0;

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

k=k+b[i]+wt[i]; k=k/(float)n;

printf("Average Turnaound Time =%f",k);

}

**OUTPUT:** Program for SJF

Enter the number of process:3 Enter the process and burst time Enter process 1 name:a

Enter process 1 burst time:12 Enter process 2 name:b Enter process 2 burst time:14 Enter process 3 name:c Enter process 3 burst time:16 Gantt chart for process

0...a...12...b...26...c...42

Calculating Waiting Time

Waiting Time of a is 0 Waiting Time of b is 12 Waiting Time of c is 26 Calculating Average Time Average Time=12.666667 Calculating Turnaround Time

Average Turn around Time =26.666666

# RESULT:

Thus the program to implement Shortest Job Scheduling was executed successfully.

# EX NO:2(c) ROUND ROBIN SCHEDULING DATE:

**AIM:**

To implement the round robin scheduling algorithm

# ROUND ROBIN SCHEDULING:

**•**



Round robin (RR) is one of the algorithms employed by process and

network schedulers in computing.

# •



As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling

all processes without priority (also known as cyclic executive).

**ALGORITHM:**

Step **1:** Obtain the number of processes from the user

Step **2:** Obtain the process ID and CPU burst time for each process from the user.

Step **3:** Set the time quantum.

Step **4:** Calculate the waiting time for each process based on the time quantum

Step **5:** Calculate the average waiting time. Step **6:** Display all the details for each process

# PROGRAM:

#include<stdio.h> #include<unistd.h> main()

{

int a[10],b[10],p[10],i,j,k,m,n,t,t1=0,w=0,t2[10]; float avg,tot;

printf("enter the number of process"); scanf("%d",&n);

j=n; k=0;

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

{

printf("process name:"); scanf("%d", &p[i]); printf("burst time:"); scanf("%d",&b[i]);

}

printf("enter the quantum number:"); scanf("%d",&t);

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

{

if(b[i]>t)

{

}

else

{

}

printf("processing %d process \n",p[i]); printf("remaining time is =%d \n",b[i]-t); p[j]=p[i];

b[j]=b[i]-t; t1+=t;

j++;

sleep(2);

printf("process removed is%d \n",p[i]); k=p[i]-1;

t1+=b[i];

w=t1-b[k];

t2[k]=w;

printf("the processing time is %d \n" ,t1); printf("waiting time is: %d \n",w); sleep(2);

n=j;

}

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

{

tot+=t2[k];

printf("wait time of process %d is %d \n",k+1,t2[k]);

}

avg=tot/m;

printf("total waiting time is %f\n",tot); printf("average waiting time is %f\n",avg);

}

# OUTPUT:

Enter the number of process2 Process name:1

Burst time:2 Process name:2 Burst time:3

Enter the quantum number:2 Process removed is1

The processing time is 2 Waiting time is: 0 Processing 2 process Remaining time is =1 Process removed is2 The processing time is 5 Waiting time is: 2

Wait time of process 1 is 0 Wait time of process 2 is 2 Total waiting time is 2.000000

Average waiting time is 1.000000

# RESULT:

Thus the program to implement Round Robin scheduling was executed successfully.

**Implementation of File Allocation Strategies**

The main idea behind allocation is effective utilization of file space and fast access of the files. There are three types of allocation:

* + contiguous allocation
  + linked allocation
  + indexed allocation

In addition to storing the actual file data on the disk drive, the file system also stores metadata about the files: the name of each file, when it was last edited, exactly where it is on the disk, and what parts of the disk are "free". Free areas are not currently in use by the file data or the metadata, and so available for storing new files. (The places where this metadata is stored are often called "inodes", "chunks", "file allocation tables", etc.)

To keep track of the free space, the file system maintains a free-space list which tracks all the disk blocks which are free. To create a file, the required space is reserved for the file and the corresponding space is removed from the free list linked to each other.

# EX NO:3(a) SEQUENTIAL FILE ALLOCATION DATE:

**AIM:**

To implement Sequential file allocation in C.

# SEQUENTIAL FILE ALLOCATION

* The main problem is how to allocate disk space to the files so that disk space is utilized effectively band files can be accessed quickly.
* We have 3 space allocation methods.
* Contiguous allocation (Sequential):

It requires each file to occupy a set of contiguous blocks on the hard disk where disk addresses define a linear ordering on the disk.

* Disadvantages:
  1. Difficult for finding space for a new file.
  2. Internal and external fragmentation will be occurred.

# ALGORITHM:

Step 1.Start

Step 2.Declare the starting block no. and the length of the file.

Step 3.Get the Starting block no. and length of the file from the user. Step 4.Allocate files sequentially until end of the file.

Step 5.Display the fragments of the file. Step 6.stop

# PROGRAM:

#include #include main()

{

int f[50],i,st,j,len,c,k; clrscr(); for(i=0;i<50;i++) f[i]=0;

X:

printf("\n Enter the starting block & length of file"); scanf("%d%d",&st,&len);

for(j=st;j<(st+len);j++) if(f[j]==0)

{ f[j]=1;

printf("\n%d->%d",j,f[j]);

}

else

{

printf("Block already allocated"); break;

}

if(j==(st+len))

printf("\n the file is allocated to disk");

printf("\n if u want to enter more files?(y-1/n-0)"); scanf("%d",&c);

if(c==1) goto X; else exit();

getch();

}

# OUTPUT:

Enter the starting block & length of file 4 10 4 ->1

5 ->1

6 ->1

7 ->1

8 ->1

9->1

10 ->1

11 ->1

12 ->1

13 ->1

The file is allocated to disk

If u want to enter more files ? (y-1/n-0)

# RESULT:

Thus the program to implement Sequential allocation of c was executed successfully.

# EX NO:3(b) INDEXED FILE ALLOCATION DATE:

**AIM:**

To implement Indexed file allocation technique in C.

# INDEXED FILE ALLOCATION

**•**



In this scheme, a special block known as the **Index block** contains the

pointers to all the blocks occupied by a file.

# •



Each file has its own index block.

**•**



The entry in the index block contains the disk address of the ith file block.

# •



The directory entry contains the address of the index block as shown in the

image:

* Indexed allocation method solves all the problems in the linked allocation by bringing all the pointers together into one location called index block.

# ALGORITHM:

Step 1.Start

Step 2.Declare the index block no. and total no.of files in a block

Step 3.Get the index block no. and total no.of files in a block from the user. Step 4.Allocate files based on the index block no.

Step 5.Arrange the files based on indexes whichare created for each fragment of the file such that each and every similar indexed file is maintained by the primary index to provide the flow to file fragments.

Step 6.stop

# PROGRAM:

#include

int f[50],i,k,j,inde[50],n,c,count=0,p; main()

{

clrscr(); for(i=0;i<50;i++) f[i]=0;

x:

printf("enter index block\t"); scanf("%d",&p);

if(f[p]==0)

{ f[p]=1;

printf("enter no of files on index\t"); scanf("%d",&n);

}

else

{

printf("Block already allocated\n"); goto x;

}

for(i=0;i<n;i++) scanf("%d",&inde[i]); for(i=0;i<n;i++) if(f[inde[i]]==1)

{

printf("Block already allocated"); goto x;

}

for(j=0;j<n;j++) f[inde[j]]=1; printf("\n allocated");

printf("\n file indexed");

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

printf("\n %d->%d:%d",p,inde[k],f[inde[k]]); printf(" Enter 1 to enter more files and 0 to exit\t"); scanf("%d",&c);

if(c==1) goto x; else exit();

getch();

}</n;k++)

</n;j++)

</n;i++)

</n;i++)

# OUTPUT:

Enter index block 9

Enter no of files on index 3 1 2 3

Allocated File indexed 9 ->1:1

9 ->2:1

9 ->3:1 enter 1 to enter more files and 0 to exit

# RESULT:

Thus the program to implement indexed file allocation of c was executed successfully.

# EX NO:3(c) LINKED FILE ALLOCATION DATE:

**AIM:**

To allocate the files in the secondary storage using Linked allocation

technique.

# LINKED FILE ALLOCATION:

* Linked Allocation Linked allocation of disk space overcomes all the problems of contiguous allocation.
* In linked allocation each file is a linked list of disk blocks where the disk blocks may be scattered anywhere on the disk.
* The directory contains a pointer to the first and last blocks of the file.

# •



In this scheme, each file is a linked list of disk blocks which **need not**



**be** contiguous.

**•**



The disk blocks can be scattered anywhere on the disk.



The directory entry contains a pointer to the starting and the ending file



block.

# •



Each block contains a pointer to the next block occupied by the file.

* Disadvantages : Space required to maintain pointers.

# ALGORITHM:

Step 1.Start

Step 2.Initialize the AVAIL linked list, where each node consist of starting address, size of the empty

block and a link for next available node

Step 3.Initialialize the FAT ( File Allocation Table) which is implemented as array of pointers.

Step 4.Display the AVAIL List

Step 5.Read File allocation request which consist of File name, No of blocks and its contents

Step 6.Traverse the AVAIL linked list from the starting node Step 7.Retrieve the required no of blocks from AVAIL List Step 8.Assign the contents of file to the retrieved blocks Step 9.Update the FAT by making an entry in FAT

Step 10.Update the AVAIL LIST

Step 11.Display the AVAIL List and FAT table Step 12.Stop

# PROGRAM:

#include #include main()

{

int f[50],p,i,j,k,a,st,len,n,c; clrscr();

for(i=0;i<50;i++) f[i]=0;

printf("Enter how many blocks that are already allocated"); scanf("%d",&p);

printf("\nEnter the blocks no.s that are already allocated"); for(i=0;i<p;i++)

{ scanf("%d",&a); f[a]=1; }

X:

printf("Enter the starting index block & length"); scanf("%d%d",&st,&len);

k=len; for(j=st;j<(k+st);j++)

{ if(f[j]==0)

{ f[j]=1;

printf("\n%d->%d",j,f[j]); } else

{ printf("\n %d->file is already allocated",j); k++; }}

printf("\n If u want to enter one more file? (yes-1/no-0)"); scanf("%d",&c);

if(c==1) goto X; else exit();

getch( ); }

# OUTPUT:

Enter how many blocks that are already allocated3 Enter the blocks no.s that are already allocated 4 7 9 Enter the starting index block & length 3 7

3 ->1

4 ->1 file is already allocated

5 ->1

6 ->1

7 ->1 file is already allocated

8 ->1

9 ->1 file is already allocated

10 ->1

11 ->1

12 ->1

If u want to enter one more file? ( yes -1 /no-0)

# RESULT:

Thus the program to implement Linked file allocation of c was executed successfully.

# EX NO:4 SEMAPHORES

**DATE: (PRODUCER-CONSUMER PROBLEM)**

# AIM:

To implement producer/consumer problem using semaphore.

# PRODUCER-CONSUMER PROBLEM:

**•**



The Producer–Consumer Problem is a classic example of a multi-Process Synchronisation problem.

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as queue.

The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from

an empty buffer.

**•**

**•**

**•**

**•**

# ALGORITHM:

Step 1. Declare variable for producer & consumer as pthread-t-tid produce tid consume.

Step 2. Declare a structure to add items, semaphore variable set as struct. Step 3. Read number the items to be produced and consumed.

Step4. Declare and define semaphore function for creation and destroy. Step 5. Define producer function.

Step 6. Define consumer function. Step 7. Call producer and consumer. Step 8. Stop the execution.

# PROGRAM:

#include<stdio.h>

int mutex=1,full=0,empty,x=0; main()

{

int n;

void producer(); void consumer(); int wait(int);

int signal(int); clrscr();

printf("\n\t\t\tPRODUCER-CONSUMER PROBLEM\nEnter the Stack size: "); scanf("%d",&empty);

printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n"); while(1)

{

printf("\nENTER YOUR CHOICE: ");

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

{

case 1: if((mutex==1)&&(empty!=0)) producer();

else

printf("BUFFER IS FULL");

break; case 2:

if((mutex==1)&&(full!=0)) consumer();

else

printf("BUFFER IS EMPTY");

break; case 3: exit(0);

break;

}

}

}

int wait(int s)

{

return(--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex); full=signal(full); empty=wait(empty); x++;

printf("Producer produces the item%d",x); mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex); full=wait(full); empty=signal(empty);

printf("Consumer consumes item%d",x); x--;

mutex=signal(mutex);

}

# OUTPUT:

PRODUCER-CONSUMER PROBLEM

Enter the Stack size: 3 1.PRODUCER

1. CONSUMER
2. EXIT

ENTER YOUR CHOICE: 1

producer produces the item1 ENTER YOUR CHOICE :1

producer produces the item2 ENTER YOUR CHOICE:1

producer produces the item3 ENTER YOUR CHOICE : 1 BUFFER IS FULL

ENTER YOUR CHOICE : 2

consumer consumes item3 ENTER YOUR CHOICE : 2

consumer consumes item2 ENTER YOUR CHOICE : 2

consumer consumes item1 ENTER YOUR CHOICE: 2 BUFFER IS EMPTY

ENTER YOUR CHOICE : 3

# RESULT:

Thus the producer consumer program was executed and verified successfully.

# FILE ORGANIZATION TECHNIQUES

The File is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.

File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.

File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.

The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.

Files of fixed length records are easier to implement than the files of variable length records.

# EX. NO:5(a) SINGLE LEVEL DIRECTORY DATE:

**AIM:**

To simulate the file oraganisation technique of Single Level Directory.

# SINGLE LEVEL DIRECTORY:

**•**



In this type of directory system, there is a root directory which has all files.

**•**



It has a simple architecture and there are no sub directories.

**•**



Advantage of single level directory system:

**•**

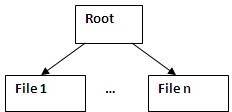


It is easy to find a file in the directory.

**•**



This type of directory system is used in cameras and phones.



# ALGORITHM:

Step 1: Start the Program

Step 2:Obtain the required data through char and int datatypes. Step 3:Enter the filename,index block.

Step 4: Print the file name index loop.

Step 5:Fill is allocated to the unused index blocks

Step 6: This is allocated to the unused linked allocation. Step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<conio.h> #include<stdlib.h> #include<graphics.h> void main()

{ int gd=DETECT,gm,count,i,j,mid,cir\_x; char fname[10][20];

clrscr(); initgraph(&gd,&gm,"c:\\tc\\bgi"); cleardevice(); setbkcolor(GREEN);

printf("enter no of files do u have"); scanf("%d",&count); for(i=0;i<count;i++)

{ cleardevice();setbkcolor(GREEN); printf("enter file %d name ",i+1); scanf("%s",fname[i]); setfillstyle(1,MAGENTA); mid=640/count;

cir\_x=mid/3; bar3d(270,100,370,150,0,0);

settextstyle(2,0,4); settextjustify(1,1); outtextxy(320,125,"Root Directory"); setcolor(BLUE); for(j=0;j<=i;j+,cir\_x+=mid)

{ line(320,150,cir\_x,250); fillellipse(cir\_x,250,30,30); outtextxy(cir\_x,250,fname[j]);

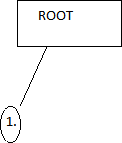
}

getch();

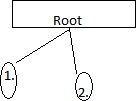
} }

# OUTPUT:

Enter the no of files do u have 3 Enter the 1 name:1.c

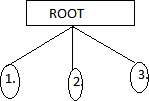


Enter the File 2 name:2.c



,,,,,,,,,,,,,,,,,,,,,,,,,

Enter the file 3 name:3.c



# RESULT:

Thus the Single Level Directory program of File Organization Technique was executed and verified successfully.

# EX. No: 5(b) TWO LEVEL DIRECTORY DATE:

**AIM:**

To simulate the file organization technique of two Level Directory.

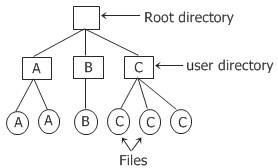
# TWO LEVEL DIRECTORY:

**•**



Two-level directory systems is used to avoid the problem caused by

the single-level directory system**.**

* In two-level directory systems, give each user a private directory.
* Therefore in this two-level directory system, names chosen by one user don't interfere with names chosen by a different user and there is no any problem that is caused by the same name occurring in two or more than two [directories](https://codescracker.com/operating-system/directories.htm).

# ALGORITHM:

Step 1: Start the Program

Step 2:Obtain the required data through char and indatatypes. Step 3:Enter the filename,index block.

Step 4: Print the file name index loop.

Step 5: File is allocated to the unused index blocks

Step 6: This is allocated to the unused linked allocation. Step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<graphics.h> struct tree\_element

{

char name[20];

int x,y,ftype,lx,rx,nc,level; struct tree\_element \*link[5];

};

typedef truct tree\_element node; void main()

{

int gd=DETECT,gm; node \*root; root=NULL; clrscr();

create(&root,0,"null",0,639,320); clrscr(); initgraph(&gd,&gm,"c:\tc\bgi"); display(root);

getch(); closegraph();

}

create(node \* \*root,int lev,char \*dname,int lx,int rx,int x)

{

int i,gap; if(\*root==NULL)

{

(\*root)=(node\*)malloc(sizeof(node)); printf("enter name of dir/file(under %s):",dname); fflush(stdin);

gets((\*root)->name); if(lev==0||lev==1) (\*root)->ftype=1;

else (\*root)->ftype=2; (\*root)->level=lev; (\*root)->y=50+lev\*50; (\*root)->x=x;

(\*root)->lx=lx; (\*root)->rx=rx; for(i=0;i<5;i++)

(\*root)->link[i]=NULL; if((\*root)->ftype==1)

{

if(lev==0||lev==1)

{

if((\*root)->level==0) printf("How many users"); else

printf("hoe many files"); printf("(for%s):",(\*root)->name);

scanf("%d",&(\*root)->nc);

}

else (\*root)->nc=0; if((\*root)->nc==0) gap=rx-lx; else gap=(rx-lx)/(\*root)->nc; for(i=0;i<(\*root)- >nc;i++)

create(&((\*root)>link[i]),lev+1,(\*root)>name,lx+gap\*i,lx+gap\*i+gap,lx+gap\*i+gap/2

);

}

else (\*root)->nc=0;

}

}

display(node \*root)

{

int i; settextstyle(2,0,4);

settextjustify(1,1); setfillstyle(1,BLUE); setcolor(14); if(root!=NULL)

{

for(i=0;i<root->nc;i++)

{

line(root->x,root->y,root->link[i]->x,root->link[i]->y);

}

if(root->ftype==1) bar3d(root->x-20,root->y-10,root->x+20,roo>y+10,0,0); else

fillellipse(root->x,root->y,20,20); outtextxy(root->x,root->y,root->name); for(i=0;i<root->nc;i++)

{

display(root->link[i]);

}

}

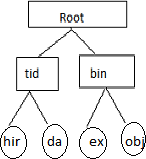
}

# OUTPUT:

Enter the name of dir/file(under null): sld How many users(forsld):2 Enter the name of dir/file(under sld):tld how many file(fortld):2 Enter the name of dir/file(under tld):hir

Enter the name of dir/file(under tld):dag Enter the name of dir/file(under tld):bin how many file(fortld):2

Enter the name of dir/file(under bin):exe Enter the name of dir/file(under bin):obj



# RESULT:

Thus the Two Level Directory program of File Organization Technique was executed and verified successfully.

# PAGE REPLACEMENT ALGORITHM:

In a computer operating system that uses paging for virtual memory management, page replacement algorithms decide which memory pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated. Page replacement happens when a requested page is not in memory (page fault) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the quality of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself

# EX.NO:7(a) FIRST –IN-FIRST-OUT (FIFO) DATE:

**AIM:**

To write a C program to implement page replacement FIFO(First In First Out) algorithm



**First-In-First**-**Out** (**FIFO**) **Replacement:**

**•**



On a **page**fault, the frame that has been in memory the longest is

replaced.

**•**



**FIFO** is not a stack **algorithm.**

# •



In certain cases, the number of **page** faults can actually increase

when more frames are allocated to the process.

**ALGORITHM:**

step 1: Start the Program

step 2:Obtain the required data through char and in datatypes.

step 3:Enter the filename,index block. step 4: Print the file name index loop.

step 5:Fill is allocated to the unused index blocks

step 6: This is allocated to the unused linked allocation. step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<conio.h>

int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1; void main()

{

clrscr();

printf("\n \t\t\t FIFI PAGE REPLACEMENT ALGORITHM");

printf("\n Enter no.of frames: "); scanf("%d",&nof);

printf("Enter number of reference string: "); scanf("%d",&nor);

printf("Enter the reference string: "); for(i=0;i<nor;i++) scanf("%d",&ref[i]);

printf("The given reference string:"); for(i=0;i<nor;i++) printf("%4d",ref[i]); for(i=1;i<=nof;i++)

frm[i]=-1;

printf("\n"); for(i=0;i<nor;i++)

{

flag=0;

printf("\n\tReference np%d->\t",ref[i]); for(j=0;j<nof;j++)

{

if(frm[j]==ref[i])

{

flag=1; break;

}

}

if(flag==0)

{

pf++; victim++;

victim=victim%nof; frm[victim]=ref[i]; for(j=0;j<nof;j++) printf("%4d",frm[j]);

}

printf("\n\nNo.of pages faults : %d",pf);

}

# OUTPUT:

FIFO PAGE REPLACEMENT ALGORITHM

Enter no.of frames: 4

Enter number of reference string: 6 Enter the reference string..

5 6 4 1 2 3

The given reference string: 5 6 4 1 2 3

F1 F2 F3 F4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reference np5-> | 5 | -1 | -1 | -1 |
| Reference np6-> | 5 | 6 | -1 | -1 |
| Reference np4-> | 5 | 6 | 4 | -1 |
| Reference np1-> | 5 | 6 | 4 | 1 |
| Reference np2-> | 2 | 6 | 4 | 1 |
| Reference np3-> | 2 | 3 | 4 | 1 |
| No.of pages faults : 6 |  |  |  |  |

# RESULT:

Thus the page replacement FIFO(First In First Out) algorithm was executed successfully.



**EX NO: 7(b**) **DATE:**

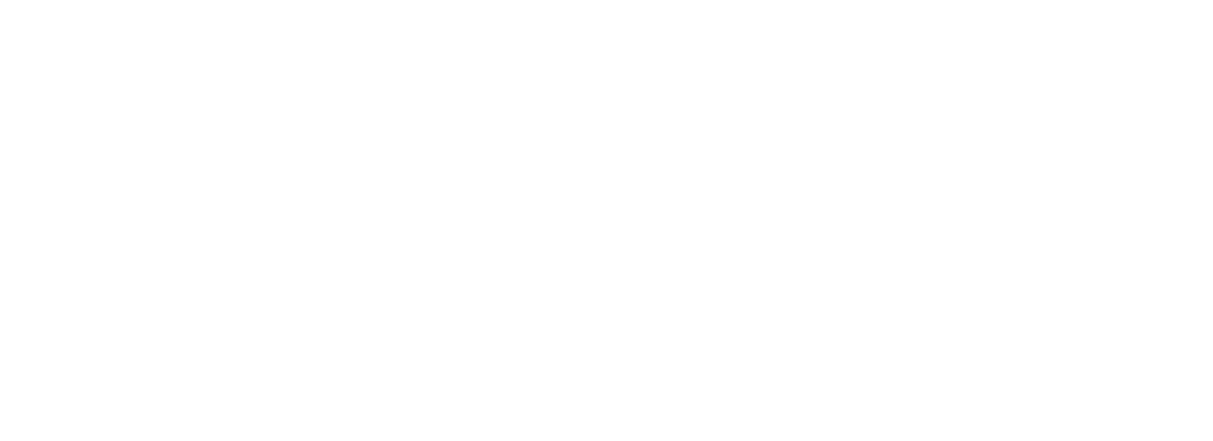
**AIM:**

**LEAST RECENTLY USED (LRU)**

To write a C program to implement page replacement LRU(Least Recently Used) algorithm.



**LEAST RECENTLY USED(LRU):**



* A good approximation to the **optimal** algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few.
* Add a register to every page frame - contain the last time that the page in that frame was accessed
* Use a "logical clock" that advance by 1 tick each time a memory reference is made.
* Each time a page is referenced, update its register

# ALGORITHM:

step 1: Start the Program

Step 2:Obtain the required data through char and int datatypes. Step 3:Enter the filename,index block.

Step 4: Print the file name index loop.

Step 5:Fill is allocated to the unused index blocks

Step 6: This is allocated to the unused linked allocation. Step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<conio.h>

int i,j,nof,nor,flag=0,ref[50],frm[50],pf=0,victim=-1; int recent[10],lrucal[50],count=0;

int lruvictim(); void main()

{

printf("\n\t\t\tLRU PAGE REPLACEMENT ALGORITHM"); printf("\nEnter no.of Frames: ");

scanf("%d",&nof);

printf("Enter no.of reference string: "); scanf("%d",&nor);

printf("Enter reference string: "); for(i=0;i<nor;i++) scanf("%d",&ref[i]);

printf("The given reference string:\n"); for(i=0;i<nor;i++)

printf("%4d",ref[i]); printf("\n\t\t\t\t "); for(i=1;i<=nof;i++)

{

frm[i]=-1; lrucal[i]=0; printf("F%d ",i);

}

for(i=0;i<10;i++) recent[i]=0; for(i=0;i<nor;i++)

{

flag=0;

printf("\n\t Reference NO %d->\t",ref[i]); for(j=0;j<nof;j++)

{

if(frm[j]==ref[i])

{

flag=1; break;

}

}

if(flag==0)

{

count++; if(count<=nof)

victim++; else

victim=lruvictim(); pf++; frm[victim]=ref[i]; for(j=0;j<nof;j++) printf("%4d",frm[j]);

}

recent[ref[i]]=i;

}

printf("\n\nNo.of page faults...%d",pf); getch();

}

int lruvictim()

{

int i,j,temp1,temp2; for(i=0;i<nof;i++)

{

temp1=frm[i]; lrucal[i]=recent[temp1];

}

temp2=lrucal[0];

for(j=1;j<nof;j++)

{

if(temp2>lrucal[j]) temp2=lrucal[j];

}

for(i=0;i<nof;i++) if(ref[temp2]==frm[i]) return i;

return 0;

}

# OUTPUT:

LRU PAGE REPLACEMENT ALGORITHM

Enter no.of Frames: 3

Enter no.of reference string: 6 Enter reference string:

6 5 4 2 3 1

The given reference string: 6 5 4 2 3 1

F1 F2 F3

|  |  |  |  |
| --- | --- | --- | --- |
| Reference NO 6-> | 6 | -1 | -1 |
| Reference NO 5-> | 6 | 5 | -1 |
| Reference NO 4-> | 6 | 5 | 4 |
| Reference NO 2-> | 2 | 5 | 4 |
| Reference NO 3-> | 2 | 3 | 4 |
| Reference NO 1-> | 2 | 3 | 1 |

No.of page faults: 6

# RESULT:

Thus the page replacement Least Recently Used (LRU) algorithm was executed successfully.

# EX NO:8 SHARED MEMORY MULTIPLICATION DATE:

**AIM:**

To write a C program to implement shared memory.

# SHARED MEMORY MULTIPLICATION



**Shared memory refers** to a block of RAM that can be accessed by several different CPUs in a multi processor computer system.

Shared memory **systems** may use:



* Uniform Memory Access (UMA): all the processors share the physical memory uniformly;
* Non Uniform Memory Access (NUMA): memory access time depends on the memory location relative to a processor;
* Cache Only Memory Architecture (COMA): the local memories for the processors at each node are used as cache instead of as actual main memory.

**ALGORITHM:**

step 1: Start the Program

step 2:Obtain the required data through char and int datatypes. step 3:Enter the filename,index block.

step 4: Print the file name index loop.

step 5:Fill is allocated to the unused index blocks

step 6: This is allocated to the unused linked allocation. step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<sys/ipc.h> #include<sys/shm.h> int main()

{

int id,\*a,\*t,\*a,pid,I,j,k,n,m,l,p; printf(“enter the order of first matrix”); scanf(“%d%d”,&m,&n);

printf(“enter the order of second matrix”): scanf(“%d%d”, &l,&p);

if(n!=l)

{

printf(“first matrix column should be same as second matrix row”);

scanf(“%d”, &l);

}

id=shmget(IPC\_PRIVATE,((m\*n)+(l\*p)+(m\*p))\*sizeof (int),0700|IPC\_CREAT|IPC\_EXCL);

q=(int \* ) shmat (id,NULL,0700); if(n==l)

{

printf (“enter the elements of first matrix”): for(i=0;i<m;i++)

for(j=0;j<n;j++) scanf(“%d”,(a+i\*n+j)); t=(q+i\*n+j);

printf(“enter the elements of second matrix”); for(i=0;i<l;i++)

for(j=0;j<p;j++) scanf(“%d”,(t+i\*p+j)); a = t+i\*p+j;

printf(“first matrix \n”); for(i=0;i<m;i++)

{

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

{

printf(“5d”,\*(q+i\*n+j));

printf(“\t”);

}

printf(“\n”);

}

printf(“second matrix \n”); for(i=0;i<l;i++0

{

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

{

printf(“%d”,\* (t+i\*p+j)); printf(“\t”);

}

printf(“\n”);

}

printf(“\n”);

}

pid=fork(); if(pid)

{

for(i=0;i<m\2;i++)

{

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

{

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

{

\*(a+i\*p+j)=\*(a+i\*p+j)+((\*(q+i\*+k))\*(\*(t+k\*p+j)));

}

}

}

printf(resultant Matrix is”); for (i=0; i<m; i++)

{

for (j=0 j<p j++)

{

printf(“%d/t”,\*(a+ip+j)); printf(“/t”);

}

printf(“/n);

}

shmdt(q); shmctl(id,IPC\_RMID,NULL);

}

else

{

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

{

for(j=0:j<p;j++)

{

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

{

\*(a+i\*p+j)=\*(a+i\*p+j)+((\*(q+i\*n+k))\*(\*(t+k\*p+j)));

}

}

shmdt(q);

}

}

getch():

}

# OUTPUT:

Enter the order of first matrix: 2

2

Enter the order of second matrix: 2

2

Enter the elements of 1st matrix: 1

1

2

2

Enter the elements of 2nd matrix: 1

2

1

2

First matrix:

1. 1

2. 2

Second matrix:

1. 2

1 2

Resultant matrix:

2. 4

6 8

# RESULT:

Thus the Shared Memory Multiplication program was executed and

verified successfully.

# EX NO:9 MEMORY MANAGEMENT DATE



**AIM:**

To write a C program to implement the concept of Paging.

# ALGORITHM:

step 1: The Semaphore mutex, full & empty are initialized. step 2: In the case of producer process

1. Produce an item in to temporary variable.
2. If there is empty space in the buffer check the mutex value for enter into the critical section.
3. If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

step 3: In the case of consumer process

1. It should wait if the buffer is empty
2. If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer

iii ) Signal the mutex value and reduce the empty value by 1. iv ) Consume the item.

step 4: Print the result

# PROGRAM:

#include <stdio.h> #include <conio.h> struct pstruct

{

int fno; int pbit;

}

ptable[10];

int pmsize,lmsize,psize,frame,page,ftable[20],frameno; void info()

{

printf("\n\t\tMEMORY MANAGEMENT USING PAGING\n\t\t \n");

printf("Enter the Size of Physical memory: "); scanf("%d",&pmsize);

printf("Enter the size of Logical memory: "); scanf("%d",&lmsize);

printf("Enter the partition size: "); scanf("%d",&psize);

frame = (int) pmsize/psize; page = (int) lmsize/psize;

printf("The physical memory is divided into %d no.of frames\n",frame);

printf("The Logical memory is divided into %d no.of pages\n",page);

}

void assign()

{

int i;

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

{

ptable[i].fno = -1;

ptable[i].pbit= -1;

}

for(i=0; i<frame;i++) ftable[i] = 32555; for (i=0;i<page;i++)

{

printf("Enter the Frame number where page %d must be placed: ",i); scanf("%d",&frameno);

ftable[frameno] = i; if(ptable[i].pbit == -1)

{

ptable[i].fno = frameno; ptable[i].pbit = 1;

}

}

//getch();

//clrscr();

printf("\n\tPAGE TABLE\n\t \n");

printf("PageAddress FrameNo. PresenceBit\n"); for (i=0;i<page;i++)

printf("%d\t\t%d\t\t%d\n",i,ptable[i].fno,ptable[i].pbit); printf("\n\tFRAME TABLE\n\t \n");

printf("FrameAddress PageNo\n"); for(i=0;i<frame;i++) printf("%d\t\t%d\n",i,ftable[i]);} void cphyaddr()

{

int laddr,paddr,disp,phyaddr,baddr;

//getch();

//clrscr();

printf("\n\tProcess to create the Physical Address\n"); printf("Enter the Base Address: "); scanf("%d",&baddr);

printf("Enter theLogical Address: "); scanf("%d",&laddr);

paddr = laddr / psize;

disp = laddr % psize; if(ptable[paddr].pbit == 1 )

phyaddr = baddr + (ptable[paddr].fno\*psize) + disp;

printf("The Physical Address where the instruction present: %d",phyaddr);} void main()

{ clrscr();

info();

assign(); cphyaddr(); getch();}

# OUTPUT:

MEMORY MANAGEMENT USING PAGING

Enter the Size of Physical memory: 16 Enter the size of Logical memory: 8 Enter the partition size: 2

The physical memory is divided into 8 no.of frames The Logical memory is divided into 4 no.of pages

Enter the Frame number where page 0 must be placed: 5 Enter the Frame number where page 1 must be placed: 6 Enter the Frame number where page 2 must be placed: 7 Enter the Frame number where page 3 must be placed: 2

PAGE TABLE

|  |  |  |
| --- | --- | --- |
| PageAddress | FrameNo. | PresenceBit |
| 0 | 5 | 1 |
| 1 | 6 | 1 |
| 2 | 7 | 1 |

FRAME TABLE

FrameAddress PageNo

|  |  |
| --- | --- |
| 0 | 32555 |
| 1 | 32555 |
| 2 | 3 |
| 3 | 32555 |
| 4 | 32555 |
| 5 | 0 |
| 6 | 1 |
| 7 | 2 |

Process to create the Physical Address Enter the Base Address: 1000

Enter theLogical Address: 3

The Physical Address where the instruction present: 1013

# RESULT:

Thus the paging technique of memory management program was executed successfully.

# EX NO:10 THREAD SYNCHRONIZATION DATE:

**AIM:**

To write a C program to implement Threading & Synchronization

# THREAD SYNCHRONIZATION:

**•**



Thread synchronization is the concurrent execution of two or more

threads that share critical resources.

**•**



Threads should be synchronized to avoid critical resource use conflicts.

**•**



Otherwise, conflicts may arise when parallel-running threads attempt to

modify a common variable at the same time.

# ALGORITHM:

step 1: Start the Program

step 2: Obtain the required data through char and in data types. step 3:Enter the filename, index block.

step 4: Print the file name index loop.

step 5: File is allocated to the unused index blocks

step 6: This is allocated to the unused linked allocation. step 7: Stop the execution

# PROGRAM:

#include<stdio.h> #include<string.h> #include<pthread.h> #include<stdlib.h> #include<unistd.h> pthread\_t tid[2];

int counter;

void\* doSomeThing(void \*arg)

{

unsigned long i = 0; counter += 1;

printf(“\n Job %d started\n”,counter); for(i=0;i<(0xFFFFFFFF);i++);

printf(“\n Job %d finished\n”,counter); return NULL;

}

int main(void)

{

int i = 0; int err;

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL,&doSomeThing, NULL); if(err != 0)

printf(“\ncan`t create thread :[%s]”,strerror(err)); i++;

}

pthread\_join(tid[0], NULL); pthread\_join(tid[1], NULL); return 0;

}

# OUTPUT:

$ ./tgsthreads Job 1 started

Job 2 started

Job 1 finished

Job 2 finished

# RESULT:

Thus the threading synchronization program was executed successfully.