Ex. No. 1a	CPU SCHEDULING – FCFS
27-02-2024	

AIM: To write a program to simulate non-pre-emptive CPU algorithm to find turnaround time and waiting time using FCFS.

ALGORITHM:

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait waiting time for p[1] will be 0 i.e. wt[0] = 0.
4. Find waiting time for all other processes i.e. for process i -> wt[i] = bt[i-1] + wt[i-1] .
5. Find turnaround time = waiting_time + burst_time for all processes.
6. Find average waiting time = total_waiting_time / no_of_processes.
7. Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

SOURCE CODE:

```
// FCFS CPU scheduling
#include<stdio.h>
// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
        wt[0] = 0; // waiting time for first process is 0
        for (int i = 1; i < n ; i++ )  // calculating waiting time
                wt[i] = bt[i-1] + wt[i-1] ; }
// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[]) {
        for (int i = 0; i < n ; i++) // calculating turnaround time by adding  bt[i] + wt[i]
                tat[i] = bt[i] + wt[i];  }
void findavgTime( int processes[], int n, int bt[]) { //Function to calculate average time
        int wt[n], tat[n], total_wt = 0, total_tat = 0;
        findWaitingTime(processes, n, bt, wt);  //Function to find wt  of all processes
        findTurnAroundTime(processes, n, bt, wt, tat); //Function to find tat for all processes
        //Display processes along with all details
        printf("Processes Burst time Waiting time Turn around time\n");
// Calculate total waiting time and total turn  around time
        for (int i=0; i<n; i++) {
                total_wt = total_wt + wt[i];
                total_tat = total_tat + tat[i];
                printf(" %d\n ",(i+1), "  %d \n", bt[i], "  %d\n",wt[i], "   %d\n",tat[i]); }
        float s=(float)total_wt / (float)n;
        float t=(float)total_tat / (float)n;
        printf("Average waiting time = %f\n",s, "Average turn around time = %f\n",t); }
int main() {
         int processes[] = { 1, 2, 3};  //process id's
        int n = sizeof processes / sizeof processes[0];
        int burst_time[] = {12,4,7};  //Burst time of all processes
        findavgTime(processes, n, burst_time);
        return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"FCFS"
CPU scheduling using FCFS
Processes Burst time Waiting time Turn around time
 1        12         0            12
 2        4          12           16
 3        7          16           23
Average waiting time = 9.333333
Average turn around time = 17.000000
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 1b	CPU SCHEDULING – SJF
27-02-2024	

AIM: To write a program to simulate non-pre-emptive CPU algorithm to find turnaround time and waiting time using SJF.

ALGORITHM:

1. Sort all the processes according to the arrival time.
2. Then select that process that has minimum arrival time and minimum Burst time.
3. After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

SOURCE CODE:

```
#include <stdio.h>
int main(){
	// Matrix for storing Process Id, BurstTime, Average Waiting Time & Average, Turn Around Time.
	int A[100][4];
	int i, j, n, total = 0, index, temp;
	float avg_wt, avg_tat;
	printf("Enter number of process: ");
	scanf("%d", &n);
	printf("Enter Burst Time:\n");
	for (i = 0; i < n; i++) {  // User Input Burst Time and alloting Process Id.
		printf("P%d: ", i + 1);
		scanf("%d", &A[i][1]);
		A[i][0] = i + 1;
	}  // Sorting process according to their Burst Time
	for (i = 0; i < n; i++) {
		index = i;
		for (j = i + 1; j < n; j++)
			if (A[j][1] < A[index][1])
				index = j;
		temp = A[i][1];
		A[i][1] = A[index][1];
		A[index][1] = temp;

		temp = A[i][0];
		A[i][0] = A[index][0];
		A[index][0] = temp;
	}
	A[0][2] = 0;
	for (i = 1; i < n; i++) {  // Calculation of Waiting Times
		A[i][2] = 0;
		for (j = 0; j < i; j++)
			A[i][2] += A[j][1];
		total += A[i][2];
	}
	avg_wt = (float)total / n;
	total = 0;
	for (i = 0; i < n; i++) { // Calculation of Turn Around Time and printing the data.
		A[i][3] = A[i][1] + A[i][2];
		total += A[i][3];
		printf("P%d	%d	%d	%d\n", A[i][0],
			A[i][1], A[i][2], A[i][3]);
	}
	avg_tat = (float)total / n;
	printf("Average Waiting Time= %f", avg_wt);
```

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

OUTPUT:

```
[aani@jefe output]$ ./"SJF"
Enter number of process: 3
Enter Burst Time:
P1: 12
P2: 4
P3: 7
P        BT      WT      TAT
P2       4       0       4
P3       7       4       11
P1       12      11      23
Average Waiting Time= 5.000000
Average Turnaround Time= 12.666667
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 1c	CPU SCHEDULING – ROUND ROBIN
27-02-2024	

AIM: To write a program to simulate non-pre-emptive CPU algorithm to find turnaround time and waiting time using Round-Robin.

ALGORITHM:

1. Create an array rem_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)
2. Create another array wt[] to store wt of processes. Initialize this array as 0.
3. Initialize time : t = 0
4. Keep traversing all the processes while they are not done. Do following for i'th process if it is not done yet.
5. If rem_bt[i] > quantum ,t = t + quantum, rem_bt[i] -= quantum;
6. Else // Last cycle for this process t = t + rem_bt[i];

    - wt[i] = t – bt[i]
    - rem_bt[i] = 0; // This process is over

SOURCE CODE:

```
#include<stdio.h>
 void main()
   {
      // initlialize the variable name
      int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
      float avg_wt, avg_tat;
      printf(" Total number of process in the system: ");
      scanf("%d", &NOP);
      y = NOP; // Assign the number of process to variable y
   // Use for loop to enter the details of the process like Arrival time and the Burst Time
   for(i=0; i<NOP; i++)
   {  // Accept arrival time
   printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1, (" Arrival time is: \t");
   scanf("%d", &at[i]);
   printf(" \nBurst time is: \t"); // Accept the Burst time
   scanf("%d", &bt[i]);
   temp[i] = bt[i]; // store the burst time in temp array
   }
   // Accept the Time quantum
   printf("Enter the Time Quantum for the process: \t");
   scanf("%d", &quant);
   // Display the process No, burst time, Turn Around Time and the waiting time
   printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
   for(sum=0, i = 0; y!=0; )  {
   if(temp[i] <= quant && temp[i] > 0) // define the conditions
   {
      sum = sum + temp[i];
      temp[i] = 0;
      count=1;
      }
      else if(temp[i] > 0)
      {
         temp[i] = temp[i] - quant;
         sum = sum + quant;
      }
      if(temp[i]==0 && count==1)
      {
         y--; //decrement the process no.
         printf("\nProcess No[%d] \t\t %d\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
         wt = wt+sum-at[i]-bt[i];
         tat = tat+sum-at[i];
         count =0;
```

```
    }
    if(i==NOP-1)
      i=0;
    else if(at[i+1]<=sum)
      i++;
    else
      i=0;
    }   // represents the average waiting time and Turn Around time
  avg_wt = wt * 1.0/NOP;
  avg_tat = tat * 1.0/NOP;
  printf("\n Average Turn Around Time: \t%f", avg_wt, ("\n Average Waiting Time: \t%f",
avg_tat);
  }
```

OUTPUT:

```
[aani@jefe output]$ ./"round_robin"
 Total number of process in the system: 3

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

Burst time is:  12

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

Burst time is:  4

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

Burst time is:  7
Enter the Time Quantum for the process:         2

 Process No              Burst Time              TAT             Waiting Time
Process No[2]            4                               9                       5
Process No[3]            7                               17                      10
Process No[1]            12                              23                      11
 Average Turn Around Time:      8.666667
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 1d	CPU SCHEDULING – PRIORITY
05-03-2024	

AIM: To write a program to simulate non-pre-emptive CPU algorithm to find turnaround time and waiting time using Round-Robin.

ALGORITHM:

1. First input the processes with their arrival time, burst time and priority.
2. First process will schedule, which have the lowest arrival time, if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.
3. Now further processes will be schedule according to the arrival time and priority of the process. (Here we are assuming that lower the priority number having higher priority).
4. If two process priority are same, then sort according to process number.
5. Once all the processes have been arrived, we can schedule them based on their priority.

SOURCE CODE:

```
#include<stdio.h>
struct process
{
   int WT,AT,BT,TAT,PT;
};
struct process a[10];
int main()
{
   int n,temp[10],t,count=0,short_p;
   float total_WT=0,total_TAT=0,Avg_WT,Avg_TAT;
   printf("Enter the number of the process\n");
   scanf("%d",&n);
   printf("Enter the arrival time , burst time and priority of the process\n");
   printf("AT BT PT\n");
   for(int i=0;i<n;i++)
   {
      scanf("%d%d%d",&a[i].AT,&a[i].BT,&a[i].PT);
      // copying the burst time in a temp array fot futher use
      temp[i]=a[i].BT;
   }   // we initialize the burst time of a process with maximum
   a[9].PT=10000;
   for(t=0;count!=n;t++)
   {
      short_p=9;
      for(int i=0;i<n;i++)
      {
         if(a[short_p].PT>a[i].PT && a[i].AT<=t && a[i].BT>0)
            short_p=i;
       }

      a[short_p].BT=a[short_p].BT-1;

      // if any process is completed
      if(a[short_p].BT==0)
      {
         // one process is completed so count increases by 1
         count++;
         a[short_p].WT=t+1-a[short_p].AT-temp[short_p];
         a[short_p].TAT=t+1-a[short_p].AT;

         // total calculation
         total_WT=total_WT+a[short_p].WT;
         total_TAT=total_TAT+a[short_p].TAT;

      }
```

```
    }

    Avg_WT=total_WT/n;
    Avg_TAT=total_TAT/n;

    // printing of the answer
    printf("ID WT TAT\n");
    for(int i=0;i<n;i++)
    {
        printf("%d %d\t%d\n",i+1,a[i].WT,a[i].TAT);
    }

    printf("Avg waiting time of the process  is %f\n",Avg_WT);
    printf("Avg turn around time of the process is %f\n",Avg_TAT);

    return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"priority"
Enter the number of the process
3
Enter the arrival time , burst time and priority of the process
AT BT PT
0 12 3
1 4 2
2 7 1
ID WT TAT
1 11    23
2 7     11
3 0     7
Avg waiting time of the process  is 6.000000
Avg turn around time of the process is 13.666667
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 2a	SEQUENTIAL FILE ALLOCATION
05-03-2024	

AIM: To write a program to simulate Sequential file allocation strategy.

ALGORITHM:

1. In contiguous file allocation, the block is allocated in such a manner that all the allocated blocks in the hard disk are adjacent.
2. Assuming a file needs 'n' number of blocks in the disk and the file begins with a block at position'x',
3. the next blocks to be assigned to it will be x+1,x+2,x+3,...,x+n-1 so that they are in a contiguous manner
4. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).

SOURCE CODE:

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

void recurse(int files[]){
   int flag = 0, startBlock, len, j, k, ch;
   printf("Enter the starting block and the length of the files: ");
   scanf("%d%d", &startBlock, &len);
   for (j=startBlock; j<(startBlock+len); j++){
      if (files[j] == 0)
         flag++;
   }
   if(len == flag){
      for (int k=startBlock; k<(startBlock+len); k++){
         if (files[k] == 0){
            files[k] = 1;
            printf("%d\t%d\n", k, files[k]);
         } }
      if (k != (startBlock+len-1))
         printf("The file is allocated to the disk\n");
   }
   else
      printf("The file is not allocated to the disk\n");

   printf("Do you want to enter more files?\n", "Press 1 for YES, 0 for NO:”);
   scanf("%d", &ch);
   if (ch == 1)
      recurse(files);
   else
      exit(0);
   return;
}

int main()
{
int files[50];
for(int i=0;i<50;i++)
files[i]=0;
printf("Files Allocated are :\n");

recurse(files);
return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"sequential_file_alloc"
Files Allocated are :
Enter the starting block and the length of the files: 6 4
6       1
7       1
8       1
9       1
The file is allocated to the disk
Do you want to enter more files?
Press 1 for YES, 0 for NO: 1
Enter the starting block and the length of the files: 8 1
The file is not allocated to the disk
Do you want to enter more files?
Press 1 for YES, 0 for NO: 1
Enter the starting block and the length of the files: 10 2
10      1
11      1
The file is allocated to the disk
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 2b	INDEXED FILE ALLOCATION
12-03-2024	

AIM: To write a program to simulate Indexed file allocation strategy.

ALGORITHM:

1. In this scheme, each file is a linked list of disk blocks which **need not be** contiguous.
2. The disk blocks can be scattered anywhere on the disk.
3. The directory entry contains a pointer to the starting and the ending file block.
4. Each block contains a pointer to the next block occupied by the file.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
for(k=0;k<n;k++)
printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n", "Enter another file indexed \n”);
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
```

```
goto x;
else
exit(0);

}
```

OUTPUT:

```
[aani@jefe output]$ ./"indexed_file_alloc"
Enter the index block: 3
Enter no of blocks needed and no of files for the index 3 on the disk :
2 1 3 1
Allocated
File Indexed
3-------->1 : 1
3-------->3 : 1
Do you want to enter more file(Yes - 1/No - 0)Enter the index block: 1
1 index is already allocated
Enter the index block: 4
Enter no of blocks needed and no of files for the index 4 on the disk :
2 3 4 5
File in the index is already allocated
Enter another file indexed2
Allocated
File Indexed
4-------->5 : 1
4-------->2 : 1
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 2c	LINKED FILE ALLOCATION
12-03-2024	

AIM: To write a program to simulate Linked file allocation strategy.

ALGORITHM:

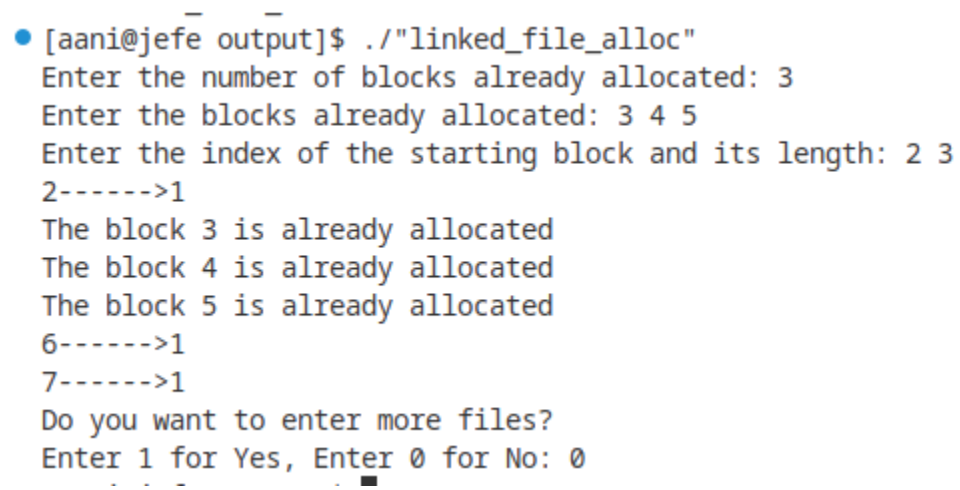
1. In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file.
2. Each file has its own index block.
3. The ith entry in the index block contains the disk address of the ith file block.
4. This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
5. It overcomes the problem of external fragmentation.

SOURCE CODE:

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

void recursivePart(int pages[]){
   int st, len, k, c, j;
   printf("Enter the index of the starting block and its length: ");
   scanf("%d%d", &st, &len);
   k = len;
   if (pages[st] == 0){
      for (j = st; j < (st + k); j++){
         if (pages[j] == 0){
            pages[j] = 1;
            printf("%d------>%d\n", j, pages[j]);
         }
         else {
            printf("The block %d is already allocated \n", j);
            k++;
         }
      }
   }
   else
      printf("The block %d is already allocated \n", st);
   printf("Do you want to enter more files? \n");
   printf("Enter 1 for Yes, Enter 0 for No: ");
   scanf("%d", &c);
   if (c==1)
      recursivePart(pages);
   else
      exit(0);
   return;
}

int main(){
   int pages[50], p, a;

   for (int i = 0; i < 50; i++)
      pages[i] = 0;
   printf("Enter the number of blocks already allocated: ");
   scanf("%d", &p);
   printf("Enter the blocks already allocated: ");
   for (int i = 0; i < p; i++){
      scanf("%d", &a);
```

```
        pages[a] = 1;
    }

    recursivePart(pages);
    return 0;
}
```

OUTPUT:



RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 3	PAGING TECHNIQUE
19-03-2024	

AIM: To write a program to simulate Paging technique of memory management.

ALGORITHM:

1. In a paging scheme, the logical deal with the region is cut up into steady-duration pages.
2. Every internet web page is mapped to a corresponding body within the physical deal with the vicinity.
3. The going for walks tool keeps a web internet web page desk for every method, which maps the system's logical addresses to its corresponding bodily addresses.
4. When a method accesses memory, the CPU generates a logical address, that is translated to a bodily address using the net page table.
5. The reminiscence controller then uses the physical cope to get the right of entry to the reminiscence.

SOURCE CODE:

```
#include<stdio.h>
main()
{
 int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
 int s[10], fno[10][20];


printf("\nEnter the memory size -- ");
scanf("%d",&ms);

printf("\nEnter the page size -- ");
scanf("%d",&ps);

nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);

printf("\nEnter number of processes -- ");
 scanf("%d",&np);
rempages = nop;
for(i=1;i<=np;i++)

{

printf("\nEnter no. of pages required for p[%d]-- ",i);
 scanf("%d",&s[i]);

if(s[i] >rempages)
{

printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];

printf("\nEnter pagetable for p[%d] --- ",i);
 for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}

printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");
```

```
scanf("%d %d %d",&x,&y, &offset);



if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");

else
{ pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);

}
}
```

OUTPUT:

```
[aani@jefe output]$ ./"paging_tec"

Enter the memory size -- 1000

Enter the page size -- 100

The no. of pages available in memory are -- 10
Enter number of processes -- 3

Enter no. of pages required for p[1]-- 4

Enter pagetable for p[1] --- 8 6 9 5

Enter no. of pages required for p[2]-- 1 4 7 5 3

Enter pagetable for p[2] ---
Enter no. of pages required for p[3]--
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2 3 60
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 4a	SINGLE LEVEL DIRECTORY FILE ORGANIZATION
19-03-2024	

AIM: To write a program to simulate Single level directory file organization technique.

ALGORITHM:

1. The single-level directory structure in file organization is the simplest way to store any number of files in a single directory.
2. It doesn't require creating multiple sub-directories inside it, all the files are stored in the same directory or folder.
3. It follows a very straightforward approach, but the files that are being stored inside the directory must have unique names.
4. No two files can have the same name and reside inside the same directory.
5. But in the single-level directory, the user can store multiple types of files inside a single directory, meaning that even if the extensions of the files are different from each other or the same.
6. They can reside inside the same directory, but only the name must be unique.

SOURCE CODE:

```
#include<stdio.h>
// #include<conio.h>
#include<string.h>
void main()
{
int nf=0,i=0,j=0,ch;
char mdname[10],fname[10][10],name[10];
printf("Enter the directory name:");
scanf("%s",mdname);
printf("Enter the number of files:");
scanf("%d",&nf);
do
{
printf("Enter file name to be created:");
scanf("%s",name);
for(i=0;i<nf;i++)
{
if(!strcmp(name,fname[i]))
break;
}
if(i==nf)
{
strcpy(fname[j++],name);
nf++;
}
else
printf("There is already %s\n",name, “Do you want to enter another file(yes - 1 or no - 0):");
scanf("%d",&ch);
}
while(ch==1);
printf("Directory name is:%s\n",mdname);
Files names are:");
for(i=0;i<j;i++)
printf("\n%s\n",fname[i]);
}
```

OUTPUT:

```
[aani@jefe output]$ ./"single_level_dir"
Enter the directory name:2
Enter the number of files:folder1
Enter file name to be created:Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:file1
Do you want to enter another file(yes - 1 or no - 0):0
Directory name is:2
Files names are:
folder1

file1
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 4b	TWO LEVEL DIRECTORY FILE ORGANIZATION
19-03-2024	

AIM: To write a program to simulate Two level directory file organization technique.

ALGORITHM:

1. In two level directory structure, user can create directory inside the root directory.
2. Through two-level file directory structure, each user can create his/her own directory and store files.
3. Start by making a user directory for each individual person and load it into the root directory. These folders will be special to them.
4. Now your users have the ability to save things like documents, images, and subfolders within their own folder.
5. The final step is making it easy to locate their files. Simple enough, all they have to do is go through the root directory and select their personal folder

SOURCE CODE:

```
#include<stdio.h>
struct st
{
char dname[10];
char sdname[10][10];
char fname[10][10][10];
int ds,sds[10];
}dir[10];
void main()
{
int i,j,k,n;
printf("enter number of directories:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter directory %d names:",i+1);
scanf("%s",&dir[i].dname);
printf("enter size of directories:");
scanf("%d",&dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("enter subdirectory name and size:");
scanf("%s",&dir[i].sdname[j]);
scanf("%d",&dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
{
printf("enter file name:");
scanf("%s",&dir[i].fname[j][k]);
}
}
}
printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");
for(i=0;i<n;i++)
{
printf("%s\t\t%d",dir[i].dname,dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("\t%s\t\t%d\t",dir[i].sdname[j],dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
printf("%s\t",dir[i].fname[j][k]);
printf("\n\t\t");
} }    }
```

OUTPUT:

```
[aani@jefe output]$ ./"two_level_dir"
enter number of directories:2
enter directory 1 names:folder1
enter size of directories:2
enter subdirectory name and size:file1 2
enter file name:book1
enter file name:book2
enter subdirectory name and size:file2 1
enter file name:book3
enter directory 2 names:folder2
enter size of directories:1
enter subdirectory name and size:subfolder 1
enter file name:book4

dirname         size    subdirname      size    files
*****************************************************
folder1         2       file1           2       book1   book2
                        file2           1       book3

folder2         1       subfolder               1       book4
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 5	BANKERS ALGORITHM
26-03-2024	

AIM: To write a program to simulate Bankers algorithm for the purpose of deadlock avoidance.

ALGORITHM:

1. It is a 1-d array of size **'m'** indicating the number of available resources of each type.
2. Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$
3. It is a 2-d array of size '**n*m'** that defines the max demand of each process in a system.
4. Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.
5. It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
6. Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$.

SOURCE CODE:

```
#include <stdio.h>
int main()
{
   int n, m, i, j, k;
   n = 5; // Number of processes
   m = 3; // Number of resources
   int alloc[5][3] = { { 0, 1, 0 },   { 2, 0, 0 }, { 3, 0, 2 }, { 2, 1, 1 }, { 0, 0, 2 } };  // Allocation matrix
   int max[5][3] = { { 7, 5, 3 }, { 3, 2, 2 }, { 9, 0, 2 }, { 2, 2, 2 }, { 4, 3, 3 } };  // Maximum matrix
   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("\nFollowing 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);
}
```

OUTPUT:

```
[aani@jefe output]$ ./"bankers_algo"
The allocation matrix is:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
The Max matrix is:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
The available resources are:
3 3 2
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 6a	FCFS DISK SCHEDULING
26-03-2024	

AIM: To write a program to simulate FCFS disk scheduling algorithm.

ALGORITHM:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,n,TotalHeadMoment=0,initial;
   printf("Enter the number of Requests\n");
   scanf("%d",&n);
   printf("Enter the Requests sequence\n");
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d",&initial);

   // logic for FCFS disk scheduling
 for(i=0;i<n;i++)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }
   printf("Total head moment is %d\n",TotalHeadMoment);
   return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"FCFS_disk_scheduling"
Enter the number of Requests
2
Enter the Requests sequence
23 69 89 32
Enter initial head position
Total head moment is 112
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 6b	SCAN DISK SCHEDULING
02-04-2024	

AIM: To write a program to simulate SCAN disk scheduling algorithm.

ALGORITHM:

1. Let the Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of the disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which the head is moving, service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach one of the ends of the disk
8. If we reach the end of the disk reverse the direction and go to step 2 until all tracks in the request array have not been serviced.

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
   printf("Enter the number of Requests\n");
   scanf("%d",&n);
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d",&initial);
   scanf("%d",&size);
   scanf("%d",&move);

   // logic for Scan disk scheduling
   for(i=0;i<n;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(RQ[j]>RQ[j+1])
         {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
         }}}
   int index;
   for(i=0;i<n;i++)
   {
      if(initial<RQ[i])
      {
         index=i;
         break;
      }}
   if(move==1)    // if movement is towards high value
   {
      for(i=index;i<n;i++)
      {
         TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];
      }
      TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
```

```
    initial = size-1;
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];

    }}
  Else     // if movement is towards low value
  {
    for(i=index-1;i>=0;i--)
    {
       TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
       initial=RQ[i];
    }
    //  last movement for min size
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    initial =0;
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }}
  printf("Total head movement is %d\n",TotalHeadMoment);
  return 0;}
```

OUTPUT:

```
[aani@jefe output]$ ./"SCAN_disk_scheduling"
Enter the number of Requests
3
Enter the Requests sequence
10 29 38
Enter initial head position
25
Enter total disk size
40
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 43
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 6c	C-SCAN DISK SCHEDULING
09-04-2024	

AIM: To write a program to simulate C-SCAN disk scheduling algorithm.

ALGORITHM:

1. Let the Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of the disk head.
2. The head services only in the right direction from 0 to the disk size.
3. While moving in the left direction do not service any of the tracks.
4. When we reach the beginning(left end) reverse the direction.
5. While moving in the right direction it services all tracks one by one.
6. While moving in the right direction calculate the absolute distance of the track from the head.

SOURCE CODE:

```
#include<stdio.h>
   #include<stdlib.h>
   int main()
   {
      int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
      scanf("%d",&n);
      for(i=0;i<n;i++)
       scanf("%d",&RQ[i]);
      scanf("%d",&initial);
      scanf("%d",&size);
      scanf("%d",&move);
      // logic for C-Scan disk scheduling
      for(i=0;i<n;i++)
      {
         for( j=0;j<n-i-1;j++)
         {
            if(RQ[j]>RQ[j+1])
            {
               int temp;
               temp=RQ[j];
               RQ[j]=RQ[j+1];
               RQ[j+1]=temp;
            }     }       }

      int index;
      for(i=0;i<n;i++)
      {
         if(initial<RQ[i])
         {
            index=i;
            break;
         }   }
      if(move==1)        // if movement is towards high value
      {
         for(i=index;i<n;i++)
         {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
         }
         TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
         TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
         initial=0;
```

```
        for( i=0;i<index;i++)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
          } }
    Else        // if movement is towards low value
    {
        for(i=index-1;i>=0;i--)
        {
           TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
           initial=RQ[i];
        }
        //  last movement for min size
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        /*movement min to max disk */
        TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
        initial =size-1;
        for(i=n-1;i>=index;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
          }  }
    printf("Total head movement is %d\n",TotalHeadMoment);
    return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"SCAN_C_disk"
Enter the number of Requests
4
Enter the Requests sequence
21 90 34 68
Enter initial head position
55
Enter total disk size
100
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 177
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 7a	FIFO PAGE REPLACEMENT
16-04-2024	

AIM: To write a program to simulate FIFO page replacement algorithm.

ALGORITHM:

1. Start traversing the pages. If set holds less pages than capacity.
2. Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
3. Maintain the pages in the queue to perform FIFO. Increment page fault.
4. Else If current page is present in set, do nothing.
5. Else Remove the first page from queue as it was the first to be entered in memory.
6. Replace the first page in the queue with the current page in the string. Store current page in the queue. Increment page faults.
7. Return page faults.

SOURCE CODE:

```
#include <stdio.h>
int main()
{
   int incomingStream[] = {3, 1, 2, 4, 2};
   int pageFaults = 0;
   int frames = 3;
   int m, n, s, pages;
   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
   printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
   int temp[frames];
   for(m = 0; m < frames; m++)
      temp[m] = -1;
   for(m = 0; m < pages; m++)
   {
      s = 0;
      for(n = 0; n < frames; n++)
      {
         if(incomingStream[m] == temp[n])
         {
            s++;
            pageFaults--;
         }
      }
      pageFaults++;
      if((pageFaults <= frames) && (s == 0))
      {
         temp[m] = incomingStream[m];
      }
      else if(s == 0)
      {
         temp[(pageFaults - 1) % frames] = incomingStream[m];
      }
      printf("\n %d\t\t\t",incomingStream[m]);
      for(n = 0; n < frames; n++)
      {
         if(temp[n] != -1)
            printf(" %d\t\t\t", temp[n]);
         else
            printf(" - \t\t\t");
      }
   }
   printf("\nTotal Page Faults:\t%d\n", pageFaults);
```

```
    return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"FIFO_page_replacement"
Incoming         Frame 1         Frame 2         Frame 3
3                       3                       -                       -
1                       3                       1                       -
2                       3                       1                       2
4                       4                       1                       2
2                       4                       1                       2
Total Page Faults:      4
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 7b	LRU PAGE REPLACEMENT
23-04-2024	

AIM: To write a program to simulate LRU page replacement algorithm.

ALGORITHM:

1. Start traversing the pages. **If set holds less pages than capacity.** Insert page into the set one by one until the size of **set** reaches **capacity** or all page requests are processed.
2. Simultaneously maintain the recent occurred index of each page in a map called **indexes**. Increment page fault.
3. **Else If** current page is present in **set**, do nothing.
4. **Else** Find the page in the set that was least recently used. We find it using index array.
5. Replace the found page with current page and Increment page faults.
6. Update index of current page.
7. Return page faults

SOURCE CODE:

```
#include<stdio.h>
int findLRU(int time[], int n){
int i, minimum = time[0], pos = 0;
for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum = time[i];
pos = i;
}}
return pos;
}
int main()
{
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
   for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
   }
for(i = 0; i < no_of_frames; ++i)
    frames[i] = -1;
   for(i = 0; i < no_of_pages; ++i){
    flag1 = flag2 = 0;
    for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
  flag1 = flag2 = 1;
  break;
  }  }
    if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
    counter++;
    faults++;
    frames[j] = pages[i];
    time[j] = counter;
    flag2 = 1;
    break;   }  }  }
```

```
    if(flag2 == 0){
    pos = findLRU(time, no_of_frames);
    counter++;
    faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
    }
    for(j = 0; j < no_of_frames; ++j){
    printf("\n %d\t", frames[j]);
    } }
printf("\n\nTotal Page Faults = %d\n", faults);
   return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"LRU_page_replacement"
Enter number of frames: 4
Enter number of pages: 2
Enter reference string: 1 2 7 8

1       -1      -1      -1
1       2       -1      -1

Total Page Faults = 2
```

RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 7c	LFU PAGE REPLACEMENT
07-05-2024	

AIM: To write a program to simulate LFU page replacement algorithm.

ALGORITHM:

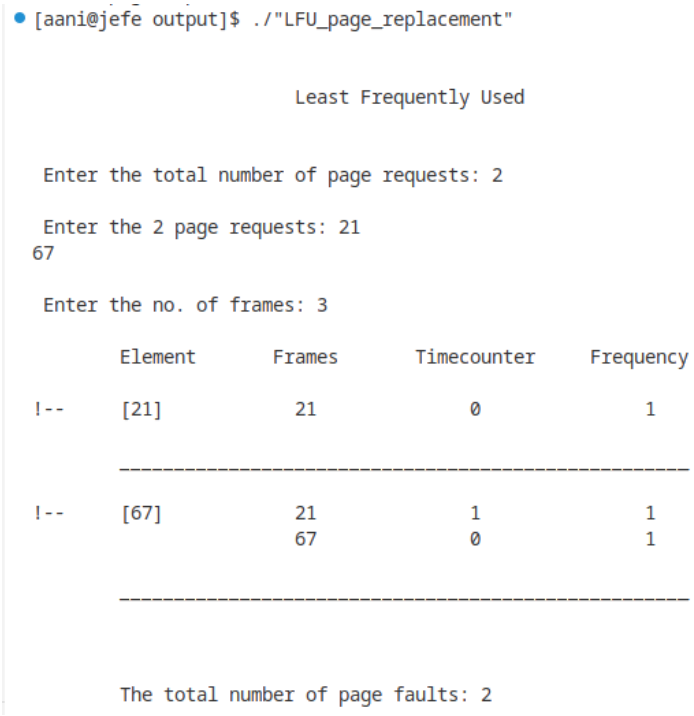
1. Initialize count as 0. Create a vector / array of size equal to memory capacity.Create a map to store frequency of pages. Traverse elements of pages[]**.**
2. In each traversal: if(element is present in memory): remove the element and push the element at the end increase its frequency.
3. Else: if(memory is full) remove the first element and decrease frequency of 1st element Increment count push the element at the end and increase its frequency
4. Compare frequency with other pages starting from the 2nd last page.
5. Sort the pages based on their frequency and time at which they arrive.
6. *I*f frequency is same, then, the page arriving first must be placed first

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int f,p, pages[50],frame[10],hit=0,count[50],time[50];
int i,j,page,flag,least,minTime,temp;
printf("Enter no of frames : ");
scanf("%d",&f);
printf("Enter no of pages : ");
scanf("%d",&p);
for(i=0;i<50;i++)
frame[i]=-1;
for(i=0;i<50;i++)
count[i]=0;
printf("Enter page no : \n");
for(i=0;i<50;i++)
scanf("%d",&pages[i]);
printf("\n");
for(i=0;i<50;i++)
{
count[pages[i]]++;
time[pages[i]]=i;
flag=1;
least=frame[0];
for(j=0;j<40;j++)
{
if(frame[j]==-1 || frame[j]==pages[i])
{
if(frame[j]!=-1)
{
hit++;
}
flag=0;
frame[j]=pages[i];
break;
}
if(count[least]>count[frame[j]])
{
least=frame[j];
}
}
if(flag)
{
```

```
minTime=50;
for(j=0;j
{
if(count[frame[j]]==count[least] && time[frame[j]]
{
temp=j;
minTime=time[frame[j]];
}
}
count[frame[temp]]=0;
frame[temp]=pages[i];
}
for(j=0;j
{
printf("%d ",frame[j]);
}}
printf("\n Page hit = %d",hit);
return 0;
}
```

OUTPUT:



RESULT :

The program is executed successfully, and the output is verified.

Ex. No. 8	PRODUCER-CONSUMER PROBLEM
14-05-2024	

AIM: To write a program to simulate Producer-consumer problem using semaphores.

ALGORITHM:

1. **Initialization of semaphores –** mutex = 1
2. Full = 0 // Initially, all slots are empty. Thus full slots are 0
3. Empty = n // All slots are empty initially
4. When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now.
5. The value of mutex is also reduced to prevent consumer to access the buffer.
6. Now, the producer has placed the item and thus the value of “full” is increased by 1.
7. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

SOURCE CODE:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#define MaxItems 5 // Max items a producer can produce or a consumer can consume
#define BufferSize 5 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;
void *producer(void *pno)
{
   int item;
   for(int i = 0; i < MaxItems; i++) {
      item = rand(); // Produce an random item
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);
      buffer[in] = item;
      in = (in+1)%BufferSize;
      pthread_mutex_unlock(&mutex);
      sem_post(&full);
   }}
void *consumer(void *cno)
{
   for(int i = 0; i < MaxItems; i++) {
      sem_wait(&full);
      pthread_mutex_lock(&mutex);
      int item = buffer[out];
      out = (out+1)%BufferSize;
      pthread_mutex_unlock(&mutex);
      sem_post(&empty);
   }
}
int main(){
   pthread_t pro[5],con[5];
   pthread_mutex_init(&mutex, NULL);
   sem_init(&empty,0,BufferSize);
   sem_init(&full,0,0);
   int a[5] = {1,2,3,4,5}; //Just used for numbering the producer and consumer
```

```
    for(int i = 0; i < 5; i++)
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
    for(int i = 0; i < 5; i++)
        pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);
     for(int i = 0; i < 5; i++)
        pthread_join(pro[i], NULL);
    for(int i = 0; i < 5; i++)
        pthread_join(con[i], NULL);
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"producer_consumer"
Producer 1: Insert Item 1804289383 at 0
Producer 1: Insert Item 846930886 at 1
Producer 1: Insert Item 1681692777 at 2
Producer 1: Insert Item 1714636915 at 3
Producer 1: Insert Item 1957747793 at 4
Consumer 5: Remove Item 1804289383 from 0
Consumer 5: Remove Item 846930886 from 1
Consumer 5: Remove Item 1681692777 from 2
Consumer 5: Remove Item 1714636915 from 3
Consumer 5: Remove Item 1957747793 from 4
Producer 5: Insert Item 424238335 at 0
Producer 5: Insert Item 1649760492 at 1
Producer 5: Insert Item 596516649 at 2
Producer 5: Insert Item 1189641421 at 3
Producer 4: Insert Item 719885386 at 4
Consumer 1: Remove Item 424238335 from 0
Consumer 1: Remove Item 1649760492 from 1
Producer 5: Insert Item 1025202362 at 0
Producer 4: Insert Item 1350490027 at 1
Consumer 1: Remove Item 596516649 from 2
Consumer 1: Remove Item 1189641421 from 3
Consumer 1: Remove Item 719885386 from 4
Producer 3: Insert Item 1102520059 at 2
Consumer 3: Remove Item 1025202362 from 0
Consumer 3: Remove Item 1350490027 from 1
Producer 3: Insert Item 1967513926 at 3
Producer 3: Insert Item 1365180540 at 4
Consumer 3: Remove Item 1102520059 from 2
Consumer 3: Remove Item 1967513926 from 3
Consumer 2: Remove Item 1365180540 from 4
Producer 4: Insert Item 2044897763 at 0
Consumer 4: Remove Item 2044897763 from 0
Producer 3: Insert Item 1540383426 at 1
Producer 3: Insert Item 1303455736 at 2
Producer 2: Insert Item 783368690 at 3
Producer 2: Insert Item 35005211 at 4
Consumer 3: Remove Item 1540383426 from 1
Consumer 4: Remove Item 1303455736 from 2
```

RESULT:

The program is executed successfully, and the output is verified.

Ex. No. 9	DINING-PHILOSOPHERS PROBLEM
21-05-2024	

AIM: To write a program to simulate Dining-philosophers problem.

ALGORITHM:

1. Initialize the semaphores for each fork to 1.
2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.
3. For each philosopher process, create a separate thread that executes the following code:
4. While true: Think for a random amount of time. Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
5. Attempt to acquire the semaphore for the fork to the left.
6. If successful, attempt to acquire the semaphore for the fork to the right.
7. If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
8. If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.
9. Run the philosopher threads concurrently.

SOURCE CODE:

```
#include<stdio.h>
#define n 4
int compltedPhilo = 0,i;
struct fork{
int taken;
}ForkAvil[n];
struct philosp{
int left;
int right;
}Philostatus[n];

void goForDinner(int philID){ //same like threads concept here cases implemented
if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
      printf("Philosopher %d completed his dinner\n",philID+1);
//if already completed dinner
else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){
         //if just taken two forks
         printf("Philosopher %d completed his dinner\n",philID+1);
         Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he
completed dinner by assigning value 10
         int otherFork = philID-1;
         if(otherFork== -1)
            otherFork=(n-1);

        ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks
         printf("Philosopher %d released fork %d and fork
%d\n",philID+1,philID+1,otherFork+1);
         compltedPhilo++; }
      else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){
//left already taken, trying for right fork
            if(philID==(n-1)){
               if(ForkAvil[philID].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST
PHILOSOPHER TRYING IN reverse DIRECTION
                  ForkAvil[philID].taken = Philostatus[philID].right = 1;
                  printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);
               }else{
                  printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);
               }
            }else{ //except last philosopher case
               int dupphilID = philID;
               philID-=1;
```

```
            if(philID== -1)
               philID=(n-1);

            if(ForkAvil[philID].taken == 0){
               ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
               printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);
            }else{
               printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);
            }
         }
      }
      else if(Philostatus[philID].left==0){ //nothing taken yet
            if(philID==(n-1)){
               if(ForkAvil[philID-1].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST
PHILOSOPHER TRYING IN reverse DIRECTION
                  ForkAvil[philID-1].taken = Philostatus[philID].left = 1;
                  printf("Fork %d taken by philosopher %d\n",philID,philID+1);
               }else{
                  printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);
               }
            }else{ //except last philosopher case
               if(ForkAvil[philID].taken == 0){
                  ForkAvil[philID].taken = Philostatus[philID].left = 1;
                  printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);
               }else{
                  printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);
               }
            }
      }else{}
}

int main(){
for(i=0;i<n;i++)
      ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){
for(i=0;i<n;i++)
         goForDinner(i);
printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);
}

return 0;
}
```

OUTPUT:

```
[aani@jefe output]$ ./"dining_philo"
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 2

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4
```

RESULT :

The program is executed successfully, and the output is verified.