**Lingaya’s Vidyapeeth, Faridabad**

(Deemed to be University under Section 3 of UGC Act, 1956)

**Operating System Lab**

**CS-252**

**LAB File**

**B.TECH 2nd Year [C.S.E]**



Nachauli, Old Faridabad-Jasana Road,

Faridabad 121002, Phone: (129) 2201008, 2201009;

WebSite: www.lingayasuniversity.edu.in

Submitted to:SubmittedBy**:**

Dr. Tapsi NAME –S Kashaf Alam

ROLL No – 22CS75CL

SEMESTER – 3rd

**LIST OF EXPERIMENTS**

1. Basics of UNIX commands.
2. Shell programming
3. Implementation of CPU scheduling. a) Round Robin b) SJF c) FCFS d) Priority
4. Implement all file allocation strategies
5. Implement Process System call.
6. Implement I/O system call.
7. Implement Bankers algorithm for Dead Lock Avoidance
8. Implement Producer/Consumer problem using semaphore.
9. Implement the all page replacement algorithms a) FIFO b) LRU c) LFU
10. Implement first fit, best fit algorithm for memory management.
11. To write a program for file manipulation for displays the file and directory in memory

*OPERATING SYSTEM LAB MANUAL*

|  |  |
| --- | --- |
| **Ex.No:1.a** | **BASICS OF UNIX COMMANDS** |
| **INTRODUCTION TO UNIX** |

**AIM:**

To study about the basics of UNIX

**UNIX:**

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed

Information and Computing Service) OS.

By1980, UNIX had been completely rewritten using C langua ge.

**LINUX**:

It is similar to UNIX, which is created by Linus Torualds. All UNIX commands works in Linux. Linux is a open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

**STRUCTURE OF A LINUXSYSTEM:**

It consists of three parts.

a)UNIX kernel

1. Shells
2. Tools and Applications

**UNIX KERNEL:**

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one programs tops and another starts.

**SHELL:**

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them

|  |  |
| --- | --- |
| **Ex.No:1.b** | **BASICS OF UNIX COMMANDS** |
| **BASIC UNIX COMMANDS** |

**AIM:**

To study of Basic UNIX Commands and various UNIX editors such as vi, ed, ex and EMACS.

**CONTENT:**

**Note: Syn->Syntax**

1. **date**

–used to check the date and time

Syn:$date

|  |  |  |  |
| --- | --- | --- | --- |
| Format | Purpose | Example | Result |
| +%m | To display only month | $date+%m | 06 |
| +%h | To display month name | $date+%h | June |
| +%d | To display day of month | $date+%d | O1 |
| +%y | To display last two digits of years | $date+%y | 09 |
| +%H | To display hours | $date+%H | 10 |
| +%M | To display minutes | $date+%M | 45 |
| +%S | To display seconds | $date+%S | 55 |

1. **cal**

–used to display the calendar

Syn:$cal 2 2009

c)**echo**

–used to print the message on the screen.

Syn:$echo “text”

d)**ls**

–used to list the files. Your files are kept in a directory.

Syn:$lsls–s

All files (include files with prefix) ls–l Lodetai (provide file statistics) ls–t Order by creation time

ls– u Sort by access time (or show when last accessed together with –l) ls–s Order by size ls–r Reverse order

ls–f Mark directories with /,executable with\* , symbolic links with @, local sockets with =, named pipes(FIFOs)with ls–s Show file size

ls– h“ Human Readable”, show file size in Kilo Bytes & Mega Bytes (h can be used together with –l or) ls[a-m]\*List all the files whose name begin with alphabets From „a‟ to „m‟ ls[a]\*List all the files whose name begins with „a‟ or „A‟

Eg:$ls>my list Output of „ls‟ command is stored to disk file named „my list‟

e)**lp**

–used to take printouts Syn:$lp filename f)**man**

–used to provide manual help on every UNIX commands.

Syn:$man unix command

$man cat

g)**who** & **whoami**

–it displays data about all users who have logged into the system currently. The next command displays about current user only.

Syn:$who$whoami

h)**uptime**

–tells you how long the computer has been running since its last reboot or power-off. Syn:$uptime

i)**uname**

–it displays the system information such as hardware platform, system name and processor, OS type. Syn:$uname–a

j)**hostname**

–displays and set system host name

Syn:$ hostname

k)**bc**

–stands for „best calculator‟

|  |  |
| --- | --- |
| $bc $ bc $ bc | $ bc |
| 10/2\*3 scale =1 ibase=2 | sqrt(196) |
| 15 2.25+1 obase=16  3.35 11010011  quit 89275  1010  Ā  Quit  $bc $ bc-l  for(i=1;i<3;i=i+1)I scale=2   1. s(3.14) 2. 0 3. quit | 14 quit |

**FILE MANIPULATION COMMANDS**

a)**cat**–this create, view and concatenate files.

**Creation**:

Syn:$cat>filename

**Viewing**:

Syn:$cat filename

**Add text to an existing file:**

Syn:$cat>>filename

**Concatenate**:

Syn:$catfile1file2>file3

$catfile1file2>>file3 (no over writing of file3)

b)**grep**–used to search a particular word or pattern related to that word from the file.

Syn:$grep search word filename

Eg:$grep anu student

c)**rm**–deletes a file from the file system

Syn:$rm filename

d)**touch**–used to create a blank file.

Syn:$touch file names

e)**cp**–copies the files or directories

Syn:$cpsource file destination file

Eg:$cp student stud

f)**mv**–to rename the file or directory syn:$mv old file new file

Eg:$mv–i student student list(-i prompt when overwrite)

g)**cut**–it cuts or pickup a given number of character or fields of the file.

Syn:$cut<option><filename>

Eg: $cut –c filename

$cut–c1-10emp

$cut–f 3,6emp

$ cut –f 3-6 emp

-c cutting columns

-f cutting fields

h)**head**–displays10 lines from the head(top)of a given file

Syn:$head filename

Eg:$head student

To display the top two lines:

Syn:$head-2student

i)**tail**–displays last 10 lines of the file

Syn:$tail filename

Eg:$tail student

To display the bottom two lines;

Syn:$ tail -2 student

j)**chmod**–used to change the permissions of a file or director y.

Syn:$ch mod category operation permission file

Where, Category–is the user type

Operation–is used to assign or remove permission

Permission–is the type of permission

File–are used to assign or remove permission all

Examples:

$chmodu-wx student

Removes write and execute permission for users

$ch modu+rw,g+rwstudent

Assigns read and write permission for users and groups

$chmodg=rwx student

Assigns absolute permission for groups of all read, write and execute permissions

k)**wc**–it counts the number of lines, words, character in a specified file(s) with the options as –l,-w,-c

|  |  |  |
| --- | --- | --- |
| Category | Operation | Permission |
| u– users g–group o– others | +assign  -remove  =assign absolutely | r– read w– write x-execute |

Syn: $wc –l filename

$wc –w filename

$wc–c filename

|  |  |
| --- | --- |
| **Ex.No:2** | **SHELL PROGRAMMING** |

**AIM:**

To write simple shell programs by using conditional, branching and looping statements.

**1. Write a Shell program to check the given number is even or odd**

**ALGORITHM:**

SEPT 1: Start the program.

STEP 2: Read the value of n.

STEP 3: Calculate „r=expr $n%2‟.

STEP 4: If the value of r equals 0 then print the number is even STEP 5: If the value of r not equal to 0 then print the number is odd.

**PROGRAM:**

echo "Enter the Number"

read n

r=`expr $n % 2` if [ $r -eq 0 ] then

echo "$n is Even number" else

echo "$n is Odd number"

fi

**OUTPUT**

**2. Write a Shell program to check the given year is leap year or not**

**ALGORITHM:**

SEPT 1: Start the program.

STEP 2: Read the value of year.

STEP 3: Calculate „b=expr $y%4‟.

STEP 4: If the value of b equals 0 then print the year is a leap year

STEP 5: If the value of r not equal to 0 then print the year is not a leap year.

**PROGRAM:**

echo "Enter the year" read y

b=`expr $y % 4` if [ $b -eq 0 ] then

echo "$y is a leap year" else

echo "$y is not a leap year" fi

**OUTPUT**

**3. Write a Shell program to find the factorial of a number**

**ALGORITHM:**

SEPT 1: Start the program.

STEP 2: Read the value of n.

STEP 3: Calculate „i=expr $n-1‟.

STEP 4: If the value of i is greater than 1 then calculate „n=expr $n \\* $i‟ and „i=expr $i – 1‟ STEP 5: Print the factorial of the given number.

**PROGRAM:**

echo "Enter a Number"

read n

i=`expr $n - 1` p=1

while [ $i -ge 1 ]

do n=`expr $n \\* $i`

i=`expr $i - 1` done

echo "The Factorial of the given Number is $n"

**OUTPUT**

**4. Write a Shell program to swap the two integers**

**ALGORITHM:**

SEPT 1: Start the program.

STEP 2: Read the value of a,b.

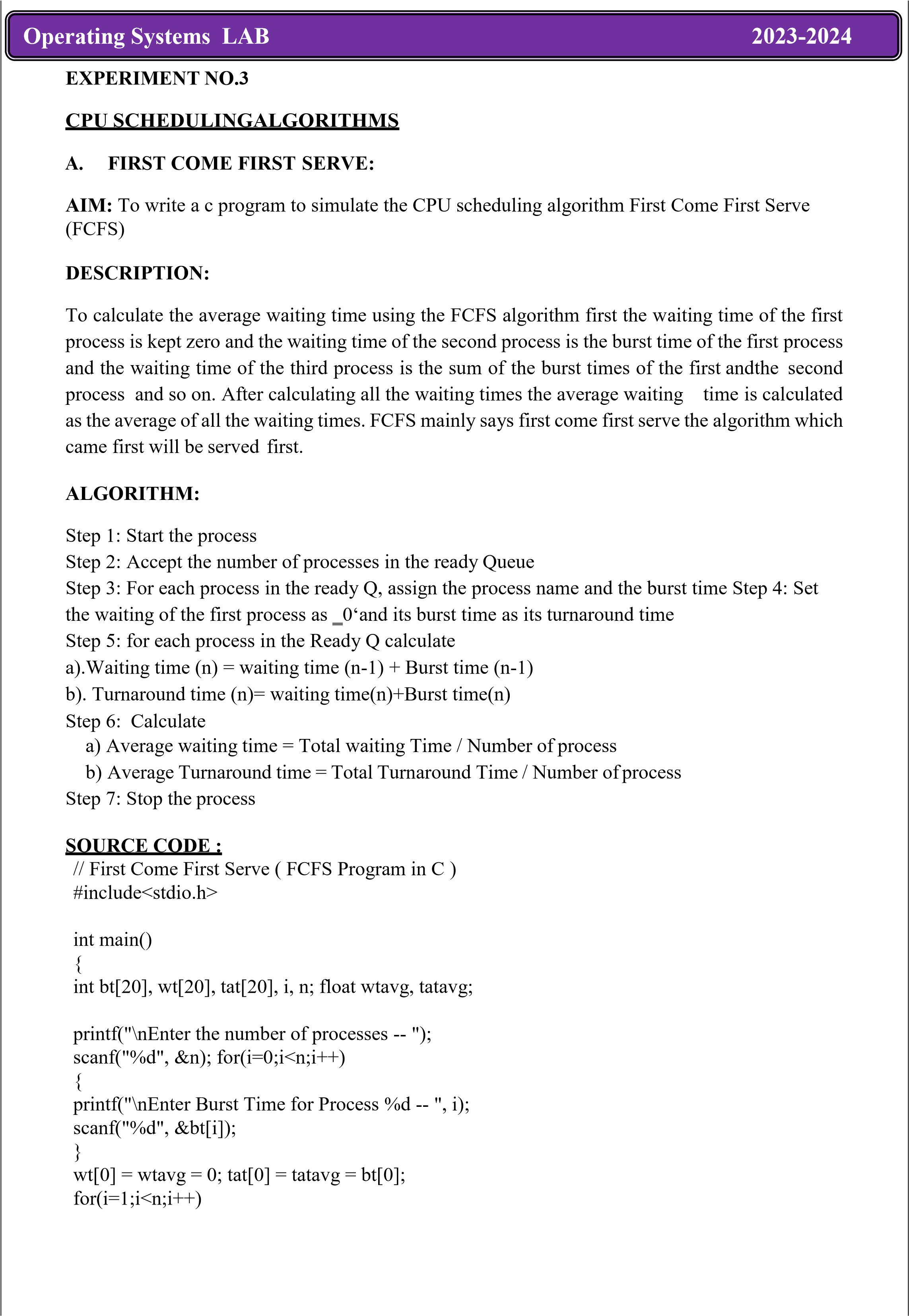
STEP 3: Calculate the swapping of two values by using a temporary variable temp. STEP 4: Print the value of a and b.

**PROGRAM:**

echo "Enter Two Numbers"

read a b temp=$a a=$b b=$temp echo "after swapping" echo $a $b

**OUTPUT**



|  |  |  |
| --- | --- | --- |
|  | {  wt[i] = wt[i-1] +bt[i-1];  tat[i] = tat[i-1] +bt[i]; wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];  }  printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n"); for(i=0;i<n;i++)  printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);  printf("\nAverage Waiting Time --%f", wtavg/n); printf("\nAverage Turnaround Time -- %f", tatavg/n); return 0; | |
| } | **INPUT:**  Enter the number of processes --  3  Enter Burst Time for Process 0 --  24  Enter Burst Time for Process 1 --  3  Enter Burst Time for Process 2 -- 3  **OUTPUT**  PROC BURST WAITING  ESS TIME TIME | TURNAROUND TIME |
|  | P0 24 0 | 24 |
|  | P1 3 24 | 27 |
|  | P2 3 27  Average Waiting Time-- 17.000000  Average Turnaround Time -- 27.000000 | 30 |

**B. SHORTEST JOB FIRST:**

**AIM:** To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

**DESCRIPTION:**

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

**ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst**2** time.

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

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue, calculate

1. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
2. Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Step 9: Stop the process

**SOURCE CODE :**

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

int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;

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

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

{ p[i]=i;

printf("Enter Burst Time for Process %d -- ", i); scanf("%d", &bt[i]);

}

for(i=0;i<n;i++) for(k=i+1;k<n;k++) if(bt[i]>bt[k])

{ temp=bt[i]; bt[i]=bt[k]; bt[k]=temp; temp=p[i]; p[i]=p[k]; p[k]=temp;

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)

{

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

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n"); for(i=0;i<n;i++)

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

printf("\nAverage Waiting Time-- %f", wtavg/n); printf("\nAverage Turnaround Time -- %f", tatavg/n); return 0;

}

INPUT:

Enter the number of processes --4

Enter Burst Time for Process 0 --6 Enter Burst Time for Process 1 --8

Enter Burst Time for Process 2 --7

Enter Burst Time for Process 3 --3

**OUTPUT:**

PROCES BURST WAITING TURNAROUND

S TIME TIME TIME

P3 3 0 3

P0 6 3 9

P2 7 9 16

P1 8 16 24

Average Waiting Time -- 7.000000

Average Turnaround Time --13.000000

**C. ROUND ROBIN:**

**AIM:** To simulate the CPU scheduling algorithm round-robin.

**DESCRIPTION:**

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

**ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

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

Step 6: Consider the ready queue is a circular Q, calculate

1. Waiting time for process (n) = waiting time of process(n-1)+ burst timeof process(n-1 ) + the time difference in getting the CPU fromprocess(n-1)
2. Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number ofprocess Step 8: Stop the process

**SOURCE CODE**

#include<stdio.h> int main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float awt=0,att=0,temp=0; printf("Enter the no of processes -- "); scanf("%d",&n); for(i=0;i<n;i++)

{

printf("\nEnter Burst Time for process %d -- ", i+1); scanf("%d",&bu[i]);

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- ");

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

for(j=0;j<(max/t)+1;j++) for(i=0;i<n;i++) if(bu[i]!=0) if(bu[i]<=t)

{ tat[i]=temp+bu[i]; temp=temp+bu[i];

bu[i]=0; } else {

bu[i]=bu[i]-t;

temp=temp+t;

}

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

{ wa[i]=tat[i]-ct[i];

att+=tat[i]; awt+=wa[i];

} printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is --%f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n"); for(i=0;i<n;i++)

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]); return 0;

}

**INPUT:**

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 -- 3

Enter Burst Time for process 3 – 3

Enter the size of time slice – 3

**OUTPUT:**

PROCESS BURST TIME WAITING TIME TURNAROUNDTIME

1 24 6 30 2 3 4 7

3 3 7 10

The Average Turnaround time is – 15.666667

The Average Waiting time is------------ 5.666667

**D. PRIORITY:**

**AIM**: To write a c program to simulate the CPU scheduling priority algorithm.

**DESCRIPTION:**

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

**ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time Step 4: Sort the ready queue according to the priority number.

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

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate

1. Waiting time(n)= waiting time (n-1) + Burst time (n-1)
2. Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. Average Turnaround time = Total Turnaround Time / Number of process Print the results in an order. Step9:Stop

**SOURCE CODE :**

#include<stdio.h> int main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float awt=0,att=0,temp=0; printf("Enter the no of processes -- ");

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

{

printf("\nEnter Burst Time for process %d -- ", i+1); scanf("%d",&bu[i]); ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- ");

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

for(j=0;j<(max/t)+1;j++) for(i=0;i<n;i++) if(bu[i]!=0) if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0; } else

{ bu[i]=bu[i]-t;

temp=temp+t;

}

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

{ wa[i]=tat[i]-ct[i];

att+=tat[i]; awt+=wa[i];

} printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is --%f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n"); for(i=0;i<n;i++)

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]); return 0;

}

**INPUT:**

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 -- 3

Enter Burst Time for process 3 – 3

Enter the size of time slice – 3

**OUTPUT:**

PROCESS BURST TIME WAITING TIME TURNAROUNDTIME

1 24 6 30 2 3 4 7

3 3 7 10

The Average Turnaround time is – 15.666667

The Average Waiting time is------------ 5.666667

**EXPERIMENT.NO.4**

**FILE ALLOCATION STRATEGIES**

**A) SEQUENTIAL:**

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program

**SOURCE CODE :**

#include<stdio.h> int main()

{

int n,i,j,b[20],sb[20],t[20],x,c[20][20]; printf("Enter no.of files:");

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

{ printf("Enter no. of blocks occupied by file%d",i+1); scanf("%d",&b[i]);

printf("Enter the starting block of file%d",i+1);

scanf("%d",&sb[i]); t[i]=sb[i]; for(j=0;j<b[i];j++) c[i][j]=sb[i]++;

}

printf("Filename\tStart block\tlength\n"); for(i=0;i<n;i++)

printf("%d\t %d \t%d\n",i+1,t[i],b[i]); printf("Enter file name:"); scanf("%d",&x); printf("File name is:%d",x); printf("length is:%d",b[x-1]); printf("blocks occupied:"); for(i=0;i<b[x-1];i++) printf("%4d",c[x-1][i]);

return 0;

}

**OUTPUT:**

Enter no.of files: 2

Enter no. of blocks occupied by file1 4

Enter the starting block of file1 2

Enter no. of blocks occupied by file2 10

Enter the starting block of file2 5

Filename Start block length

1. 2 4
2. 5 10

Enter file name: abhishek

File name is:12803 length is:0blocks occupied

**B) INDEXED:**

**AIM:** To implement allocation method using chained method

**DESCRIPTION:**

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation.

**ALGORITHM:**

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer. If yes check whether the buffer is empty Step 8: If no the consumer consumes them from the buffer Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required Step 11: Terminate the process.

**SOURCE CODE :**

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

int n,m[20],i,j,sb[20],s[20],b[20][20],x; printf("Enter no. of files:"); scanf("%d",&n); for(i=0;i<n;i++)

{ printf("Enter starting block and size of file%d:",i+1); scanf("%d%d",&sb[i],&s[i]); printf("Enter blocks occupied by file%d:",i+1);

scanf("%d",&m[i]); printf("enter blocks of file%d:",i+1); for(j=0;j<m[i];j++) scanf("%d",&b[i][j]); } printf("\nFile\t index\tlength\n"); for(i=0;i<n;i++)

{

printf("%d\t%d\t%d\n",i+1,sb[i],m[i]); }printf("\nEnter file name:"); scanf("%d",&x); printf("file name is:%d\n",x); i=x-1; printf("Index is:%d",sb[i]); printf("Block occupied are:"); for(j=0;j<m[i];j++) printf("%3d",b[i][j]); return 0;

}

OUTPUT:

Enter no. of files:2

Enter starting block and size of file1: 2 5 Enter blocks occupied by file1:10 enter blocks of file1:3

2 5 4 6 7 2 6 4 7

Enter starting block and size of file2: 3 4 Enter blocks occupied by file2:5 enter blocks of file2: 2 3 4 5 6

File index length

1. 2 10
2. 3 5

Enter file name: ankit file name is:12803

Index is:0Block occupied are:

**EXPERIMENT.NO.5**

**IMPLEMENT PROCESS SYSTEM CALL**

**AIM : Write a C program to implement UNIX Process system calls.**

**System Calls:**

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

**Process Control:** These system calls deal with processes such as process creation, process termination etc.

Fork system call is used for creating a new process, which is called child process. The child process runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same PC (Program Counter), same CPU registers, same open files which use in the parent process.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the variables pid and child id.

Step 3: Get the child id value using system call fork().

Step 4: If child id value is greater than zero then print as “i am in the parent process”.

Step 5: If child id! = 0 then using getpid() system call get the process id.

Step 6: Print “i am in the parent process” and print the process id.

Step 7: If child id! = 0 then using getppid() system call get the parent process id.

Step 8: Print “i am in the parent process” and print the parent process id. Step 9: Else If child id value is less than zero then print as “i am in the child process”.

Step 10: If child id! = 0 then using getpid() system call get the process id.

Step 11: Print “i am in the child process” and print the process id.

Step 12: If child id! = 0 then using getppid() system call get the parent process id.

Step 13: Print “i am in the child process” and print the parent process id. Step 14: Stop the program.

**SOURCE CODE:**

/\* fork system call \*/

#include<stdio.h>

#include <unistd.h> #include<sys/types.h> int main() { int id,childid; id=getpid(); if((childid=fork())>0)

{ printf("\n i am in the parent process %d",id); printf("\n i am in the parent process %d",getpid()); printf("\n i am in the parent process %d\n",getppid());

} else { printf("\n i am in child process %d",id); printf("\n i am in the child process %d",getpid()); printf("\n i am in the child process %d",getppid());

}

}

OUTPUT:

$ vi fork.c

$ cc fork.c $ ./a.out i am in child process 3765 i am in the child process 3766 i am in the child process 3765 i am in the parent process 3765 i am in the parent process 3765 i am in the parent process 3680

RESULT:

Thus the program was executed and verified successfully

**B) Wait () and Exit () System Calls:**

**AIM:To write the program to implement the system calls wait ( ) and exit ( ).**

DESCRIPTION:

1. **fork ( )**

Used to create new process. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero. **Syntax: fork ( );**

1. **wait (** )

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

**Syntax: wait (NULL);**

1. **exit ( )**

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call). **Syntax: exit (0);**

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the variables pid and i as integers.

Step 3: Get the child id value using the system call fork ().

Step 4: If child id value is less than zero then print “fork failed”.

Step 5: Else if child id value is equal to zero, it is the id value of the child and then start the child process to execute and perform Steps 7 & 8.

Step 6: Else perform Step 9.

Step 7: Use a for loop for almost five child processes to be called.

Step 8: After execution of the for loop then print “child process ends”.

Step 9: Execute the system call wait ( ) to make the parent to wait for the child process to get over.

Step 10: Once the child processes are terminated, the parent terminates and hence prints “Parent process ends”.

Step 11: After both the parent and the child processes get terminated it execute the wait ( ) system call to permanently get deleted from the OS. Step 12: Stop the program.

**SOURCE CODE:**

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

{

int i, pid; pid=fork( ); if(pid== -1) {

printf("fork failed"); exit(0); } else if(pid==0)

{

printf("\n Child process starts"); for(i=0; i<5; i++)

{

printf("\n Child process %d is called", i);

}

printf("\n Child process ends");

} else { wait(0);

printf("\n Parent process ends");

} exit(0);

}

OUTPUT:

$ vi waitexit.c

$ cc waitexit.c

$ ./a.out

Child process starts

Child process 0 is called

Child process 1 is called

Child process 2 is called

Child process 3 is called

Child process 4 is called

Child process ends

Parent process ends

RESULT:

Thus the program was executed and verified successfully **SOURCE CODE:**

/\* wait system call \*/

#include <stdlib.h>

#include <errno.h>

#include<stdio.h>

#include <unistd.h>

#include <sys/types.h> #include <sys/wait.h> main() { pid\_t pid;

int rv;

switch(pid=fork())

{ case -1: perror("fork"); exit(1); case 0: printf("\n CHILD: This is the child process!\n"); fflush(stdout); printf("\n CHILD: My PID is %d\n", getpid()); printf("\n CHILD: My parent's PID is %d\n",getppid()); printf("\n CHILD: Enter my exit status (make it small):\n "); printf("\n CHILD: I'm outta here!\n"); scanf(" %d", &rv); exit(rv); default:

printf("\nPARENT: This is the parent process!\n"); printf("\nPARENT: My PID is %d\n", getpid());

fflush(stdout); wait(&rv); fflush(stdout);

printf("\nPARENT: My child's PID is %d\n", pid); printf("\nPARENT: I'm now waiting for my child to exit()...\n"); fflush(stdout); printf("\nPARENT: My child's exit status is: %d\n",WEXITSTATUS(rv));

printf("\nPARENT: I'm outta here!\n");

}

}

**OUTPUT:**

$ vi wait.c

$ cc wait.c $ ./a.out

CHILD: This is the child process! CHILD: My PID is 3821

CHILD: My parent's PID is 3820 CHILD: Enter my exit status (make it small):

CHILD: I'm outta here!

PARENT: This is the parent process!

PARENT: My PID is 3820

10

PARENT: My child's PID is 3821

PARENT: I'm now waiting for my child to exit()... PARENT: My child's exit status is: 10 PARENT: I'm outta here!

RESULT:

Thus the program was executed and verified successfully