|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Course Title** | **Unix Operating System** | | | | **Course Type** | | **Integrated** | |
| **Course Code** | **B20CI0402** | **Credits** | **4** | | **Class** | | **IV Semester** | |
| **Course**  **Structure** | TLP | Credits | Contact  Hours | Work  Load | Total Number of  Classes  Per Semester | | Assessment in  Weightage | |
| Theory | 3 | 3 | 3 |
| Practice | 1 | 2 | 2 | Theory | Practical | CIE | SEE |
| - | - | - | - |
| **Total** | **4** | **5** | **5** | **39** | **26** | **50** | **50** |

**COURSE OVERVIEW:**

UNIX operating system provides a practical case of operating systems for the user to understand and master deeply and tangibly the theory and algorithms in operating systems. It gives deeper insights into the hierarchical structure, principles, applications, shells, development, and management of the UNIX operation system multi-dimensionally, systematically and from the elementary to the profound. It makes the user to understand about how UNIX operating system functions.

**COURSE OBJECTIVE(S):**

The objectives of this course are to:

1. Explain the history, basics and structure of UNIX Operating System
2. Describe UNIX process concepts and scheduling techniques
3. Illustrate the use of different memory management techniques of UNIX.
4. Describe UNIX kernel, data structures and internal representation of files in UNIX operating system

**COURSE OUTCOMES (COs):**

After the completion of the course, the students will be able to:

|  |  |  |  |
| --- | --- | --- | --- |
| CO# | Course Outcomes | POs | PSOs |
| CO1 | Outline the history of UNIX environment and its software architecture. | 1,2,5 | 1,3 |
| CO2 | Develop the programs to implement the different process states, attributes and control the process in foreground and background. | 1,4,5 | 1,3 |
| CO3 | Compare and analyze the performance of different memory management techniques. | 1,4,5 | 2,3 |
| CO4 | Make use of UNIX file types, file structure and file system implementation. | 1,2,4, 5 | 1,3 |
| CO5 | Learn new tools and technologies in the Designing of algorithms and apply for suitable application development. | 12 | 1 |
| CO6 | Develop solution to the complex problems, either individually or as a part of the team and report the results with proper analysis and interpretation. | 5,6,10 | 1,2 |

**Lab Exercise:**

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Title of the Experiment** | **Tools and**  **Techniques** | **Expected Skill**  **/Ability** |
| Part-A | | | |
| 1. | a) Execute at least ten UNIX shell commands on the terminal and the use of the shell commands. | Linux OS | Shell commands. |
| b) Write a C program to display the output of any UNIX shell command. | Linux OS |
| 2. | Write a C Program that makes a copy of a file using standard I/O and system calls. | Linux OS | System calls |
| 3. | a) Using an appropriate system calls Implement in C the “cat” Unix command (using command line arguments). | Linux OS | System calls |
| 4. | a) Write a C program to create a child process and allow the parent to display “parent process” and the child to display “child process” on the screen and also display the corresponding pid values. | Linux OS | Process Control |
| b) Write a C program to show the process is an orphan process and print its parent pid value. |
| 5. | a)  Write a C program that creates a zombie and then calls system to execute the ps command to verify that the process is zombie along with process state code. | Linux OS | Process Control |
| b) Write a C program to avoid zombie process by forking twice. |
| 6. | a) Write a C program that outputs the contents of its Environment list. | Linux OS | Environment variables |
| b) Write a C program to illustrate the race condition. | Process management |
| 7. | Write a C program to implement round robin algorithm by considering arrival time and burst time for each process and print the average turn around time and average waiting time for different number of processes and time quantums. | Linux OS | Memory management |
| 8. | Write a C program to implement first fit memory algorithm by considering the number of blocks, size of each block, number of processes and size of each process from the user | Linux OS | Memory management |
| 9 | a) Write a C program to create hard link and soft link and display the hard link count with other attributes of the created file within the sample code. | Linux OS | File Types and Attributes |
| b) Write a C program to list for every file in a directory, its inode number and file name | File Types and Attributes |
| 10. | a) Write a C/C++ program which demonstrates Inter process communication between a reader process and a writer process of a FIFO file by using the corresponding API’s | Linux OS | Inter Process Communication |
| b) Write a C/C++ program which demonstrates the signal handler function to handle the signal sent by the process | UNIX signals |

1. **a. Execute at least ten UNIX shell commands on the terminal and the use of the shell commands.**
2. ls
3. date
4. cal
5. pwd
6. echo
7. cp
8. ps
9. who
10. mkdir
11. cd
12. **Write a C program to display the output of any UNIX shell command.**

#include<stdio.h>

int main()

{

system(“date”);

return 0;

}

Output:

Thu Feb 24 13:00:37 IST 2022

1. **Write a C Program that makes a copy of a file using standard I/O and system calls.**

#include<stdio.h>

#include <fcntl.h>

#include<unistd.h>

int main()

{

int fd, nread;

char buf[1024];

fd = open ("sample", O\_RDONLY);

if (fd == -1)

{

printf("file not found");

return 1;

}

nread = read (fd, buf, sizeof (buf));

write (1, buf, nread);

close (fd);

}

Output:

$ cat sample

hi welcome to UNIX lab

$ gcc 1.c

$ ./a.out

hi welcome to UNIX lab

1. **Using an appropriate system calls Implement in C the “cat” Unix command(using command line arguments).**

#include<unistd.h>

#include<stdio.h>

#include<fcntl.h>

int main( int argc,char \*argv[] )

{

int fd,i; char buf[20];

fd=open(argv[1],O\_RDONLY);

if(fd==-1)

{

printf("file open error");

}

else

{

while((i=read(fd,buf,1))>0)

{

printf("%c",buf[0]);

}

close(fd);

}

return 0;

}

Output:

$ cat a.txt

Good afternoon

$ gcc 2a.c

$ ./a.out

Good afternoon

1. **a. Write a C program to create a child process and allow the parent to display “parent process” and the child to display “child process” on the screen and also display the corresponding pid values.**

#include<stdio.h>

int main()

{

int pid;

pid=fork();

if(pid==0)

{

printf(“child process\n”);

printf(“child process pid is %d”, getpid());

}

else

{

wait();

printf(“parent process\n”);

printf(“parent process pid is %d”, getpid());

}

return 0;

}

Output:

child process

child process pid is 3029

parent process

parent process pid is 3028

**b. Write a C/C++ program to show the process is an orphan process and print its parent pid value.**

#include<stdlib.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

int child\_pid;

child\_pid = fork();

if (child\_pid == 0)

{

sleep(60);

printf("parent process id of child process is %d", getppid());

}

else

{

printf("parent process terminates with exit()\n");

exit(0);

}

return 0;

}

Output:

parent process terminates with exit()

parent process id of child process is 1

1. **a. Write a C program that creates a zombie and then calls system to execute the ps command to verify that the process is zombie along with process state code.**

#include<stdlib.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

int child\_pid;

child\_pid = fork();

if (child\_pid > 0)

{

printf(“child process id is %d”, getpid());

exit(0);

}

else

{

printf(“parent process\n”);

system("ps s");

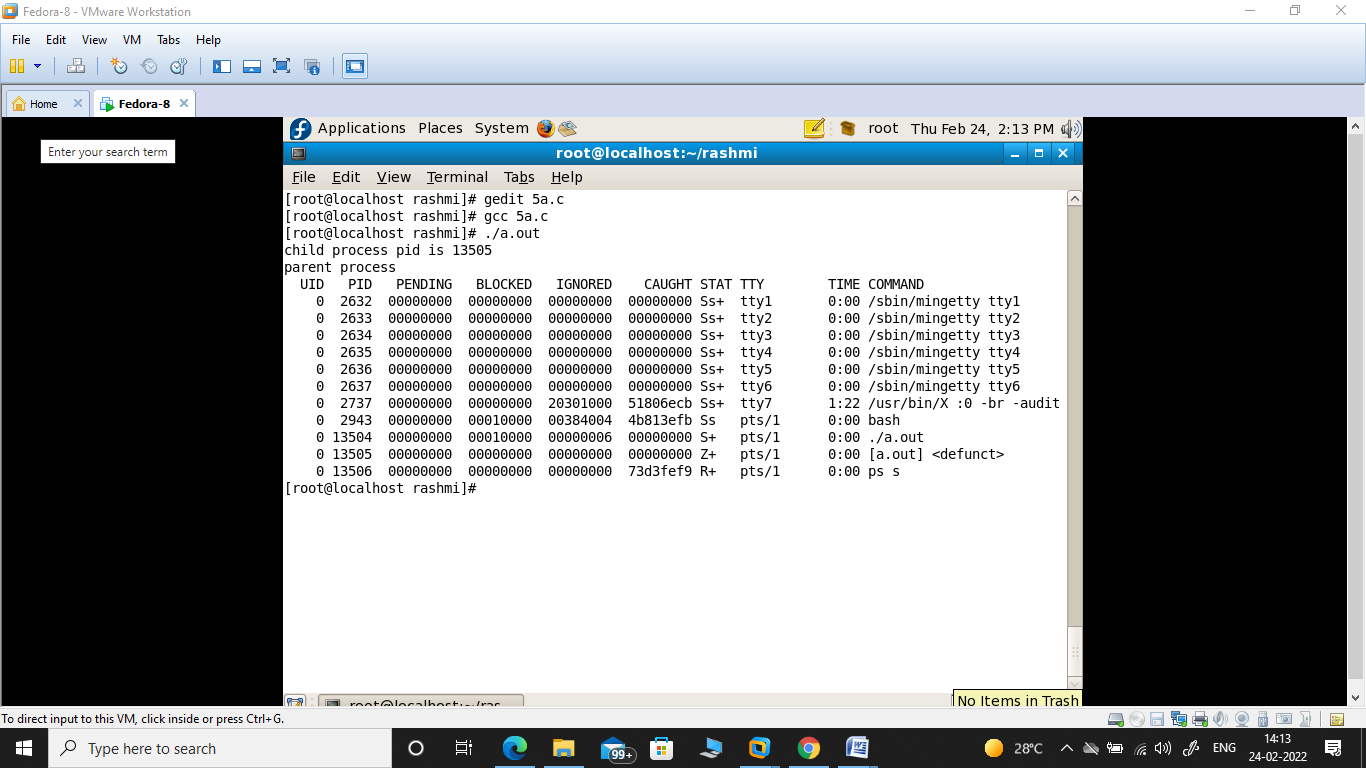
sleep(60);

}

return 0;

}

Output:



**b. Write a C/C++ program to avoid zombie process by forking twice.**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main()

{

    pid\_t pid;

    // fork first time

    pid = fork();

    if (pid == 0)

    {

        // double fork

        pid = fork();

        if (pid == 0)

{

exit(0);

}

else

{

Wait(NULL);

Sleep(10);

             printf("Grandchild pid : %d\n and its parent pid is : %d\n", getpid(), getppid());

     }

}

else

{

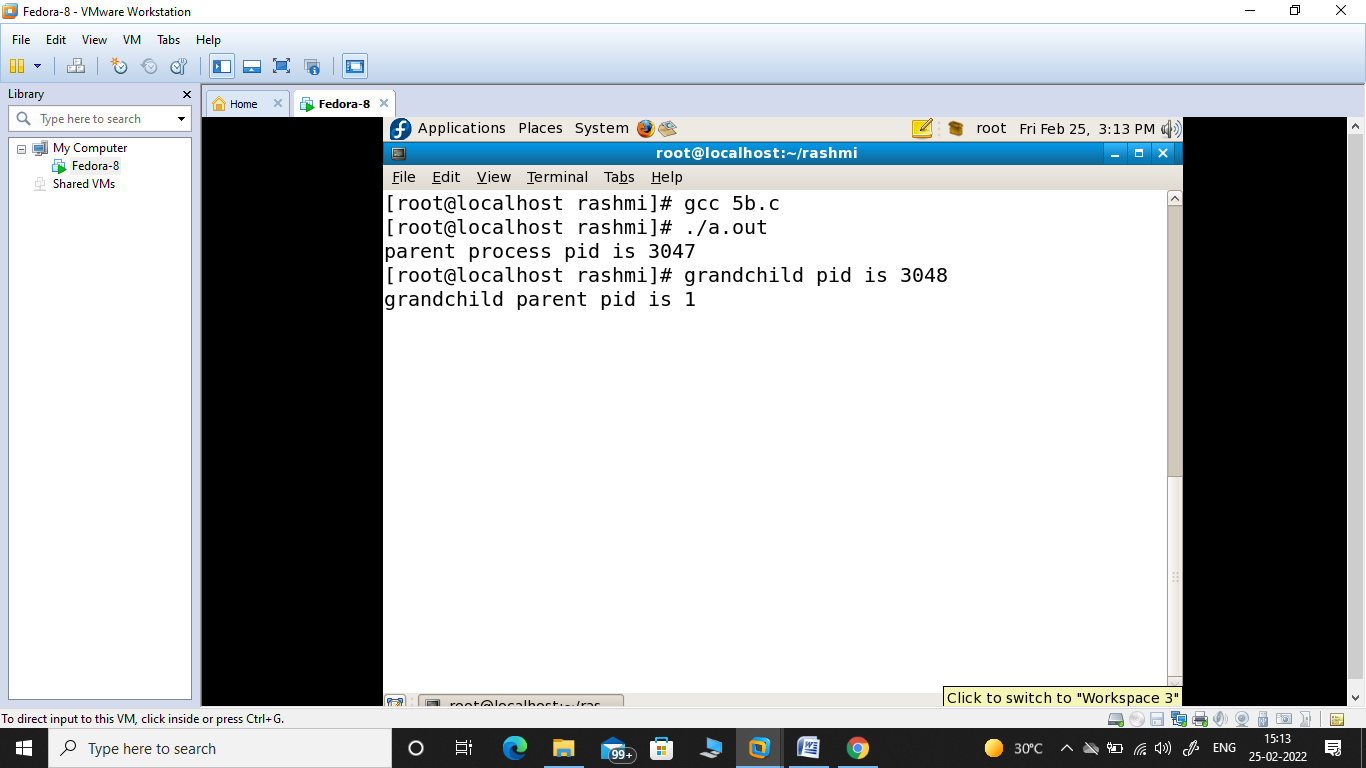
Printf(“parent pid is %d”, getpid());

}

return 0;

}

Output:



1. **a. Write a C/C++ program that outputs the contents of its Environment list.**

#include<stdio.h>

int main(int argc, char\* argv[ ])

{

int i;

char \*\*ptr;

extern char \*\*environ;

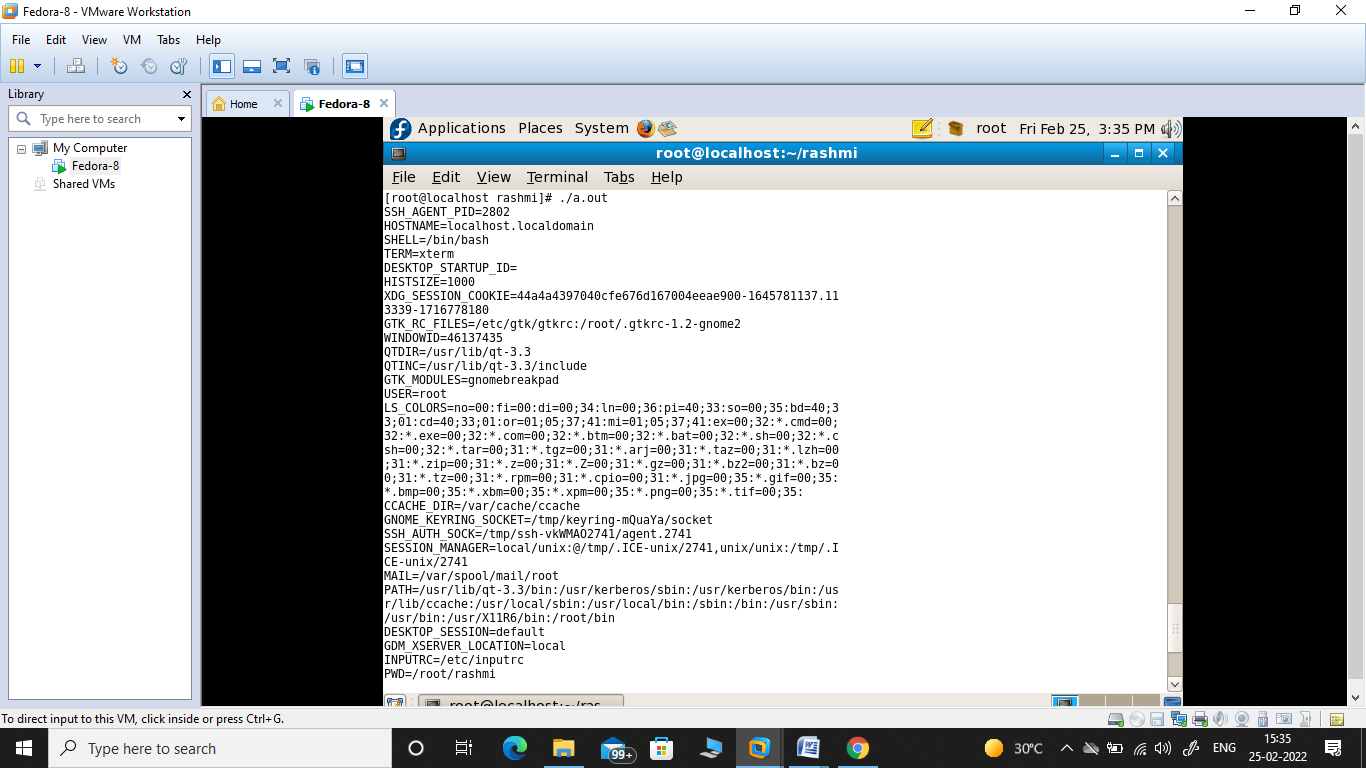
for( ptr = environ; \*ptr != 0; ptr++ ) /\*echo all env strings\*/

printf("%s\n", \*ptr);

return 0;

}

Output:



b**. Write a C/C++ program to illustrate the race condition.**

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

static void charatatime(char \*);

int main(void)

{

pid\_t pid;

if ((pid = fork()) < 0)

{

printf("fork error");

}

else if (pid == 0)

{

charatatime("output from child\n");

}

else

{

charatatime("output from parent\n");

}

return 0;

}

static void charatatime(char \*str)

{

char \*ptr;

int c;

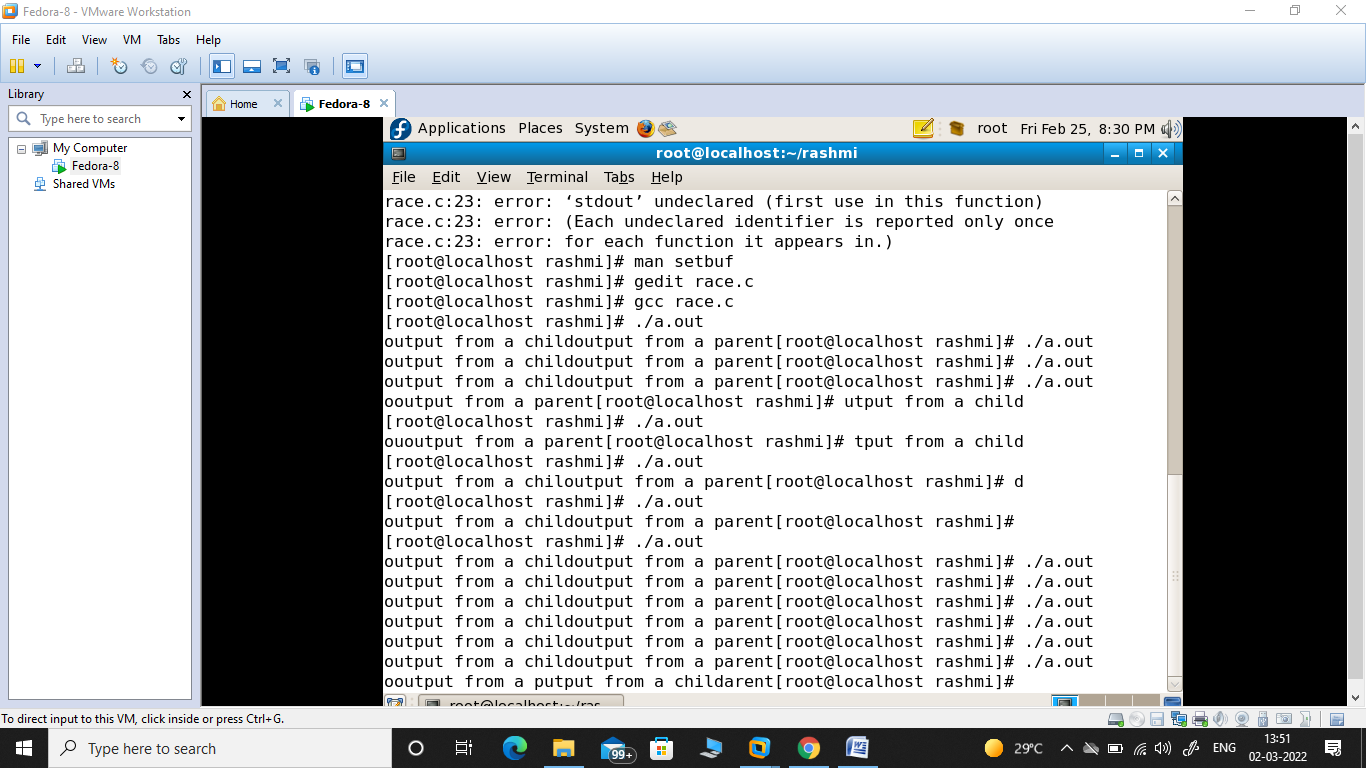
setbuf(stdout, NULL); /\* set unbuffered \*/

for (ptr = str; (c = \*ptr++) != 0; )

putc(c, stdout);

}

Output:



1. **Write a C program to implement round robin algorithm by considering arrival time and burst time for each process and print the average turn around time and average waiting time for different number of processes and time quantums.**

#include<stdio.h>

int 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++)

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");  // Accept arrival time

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 qunat

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\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;

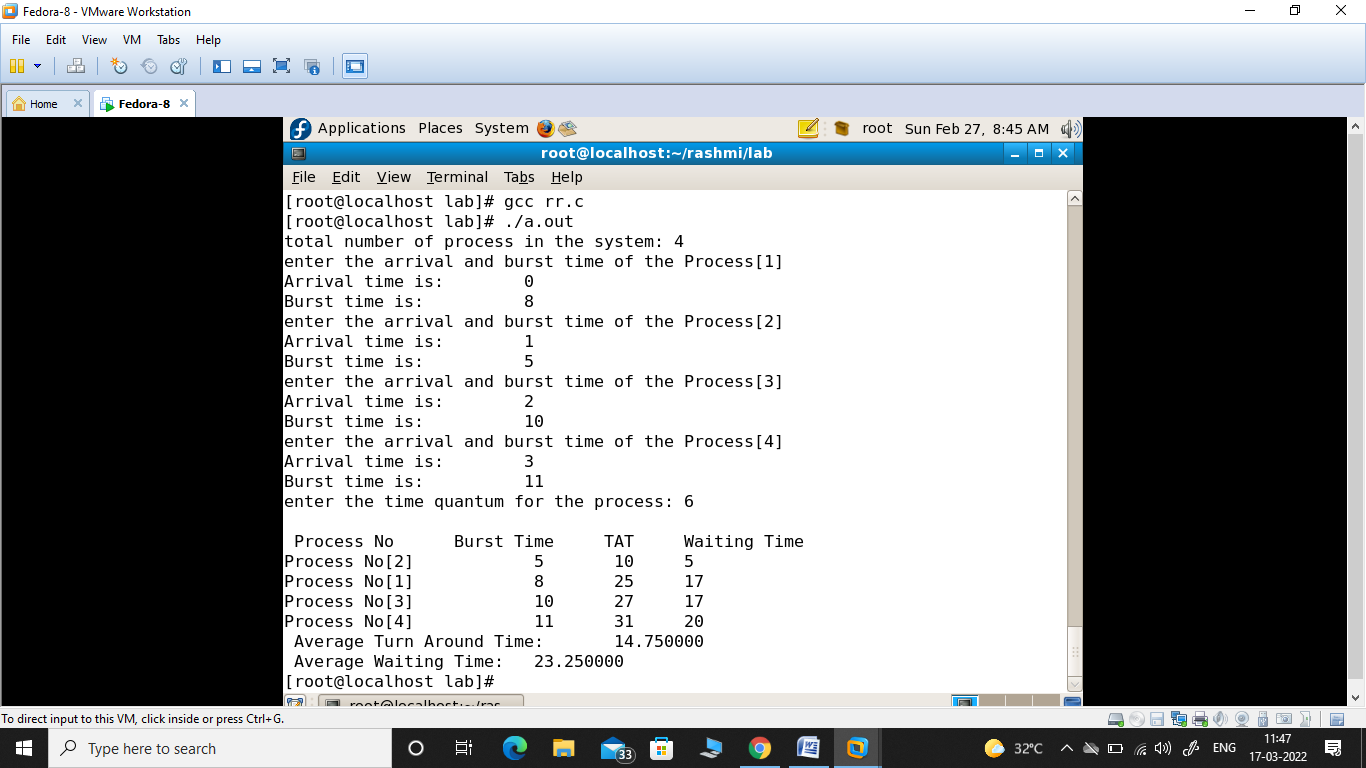
printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

return 0;

}

Output:



1. **Write a C program to implement first fit memory algorithm by considering the number of blocks, size of each block, number of processes and size of each process from the user.**

#include<stdio.h>

int main()

{

int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

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

{

flags[i] = 0;

allocation[i] = -1;

}

printf("Enter no. of blocks: ");

scanf("%d", &bno);

printf("\nEnter size of each block: ");

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

scanf("%d", &bsize[i]);

printf("\nEnter no. of processes: ");

scanf("%d", &pno);

printf("\nEnter size of each process: ");

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

scanf("%d", &psize[i]);

for(i = 0; i < pno; i++)         //allocation as per first fit

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

if(flags[j] == 0 && bsize[j] >= psize[i])

{

allocation[j] = i;

flags[j] = 1;

break;

}

//display allocation details

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");

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

{

printf("\n%d\t\t%d\t\t", i+1, bsize[i]);

if(flags[i] == 1)

printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);

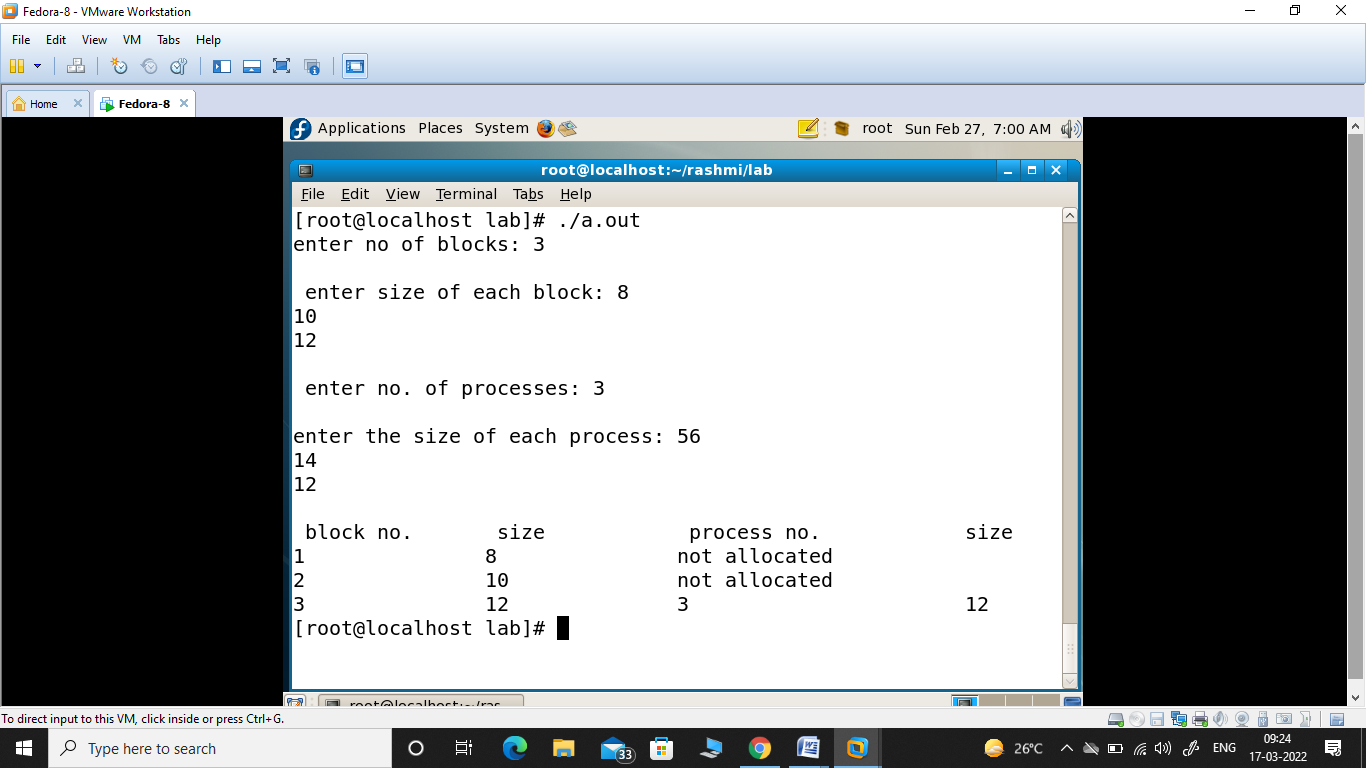
else

printf("Not allocated");

}

}

**Output:**



1. **A. Write a C program to create hard link and soft link and display the hard link count with other attributes of the created file within the sample code.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

 int main(int argc, char\* argv[])

{

    if (argc<2 || argc > 4)

{

printf(“invalid inputs\n”);

exit(0);

}

else if(argc == 3)

{

link(argv[1], argv[2]);

printf(“hard link created successfully\n”);

}

else

{

symlink(argv[2], argv[3]);

printf(“soft link created successfully\n”);

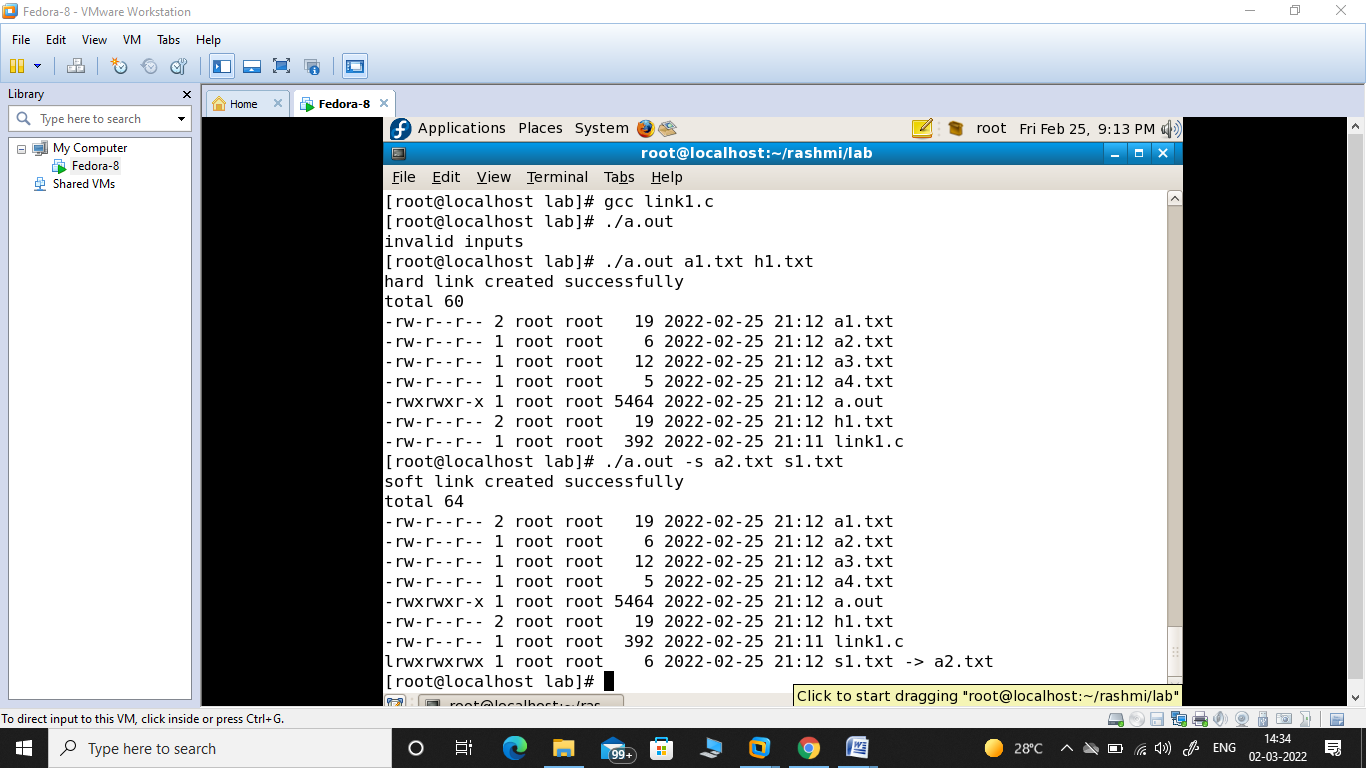
}

system(“ls –l”);

return 0;

}

  Output:



**9 B. Write a C program to list for every file in a directory, its inode number and file name.**

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

int main(int argc, char \*argv[])

{

char d[50];

if(argc==2)

{

bzero(d,sizeof(d));

strcat(d,"ls ");

strcat(d,"-i ");

strcat(d,argv[1]);

system(d);

}

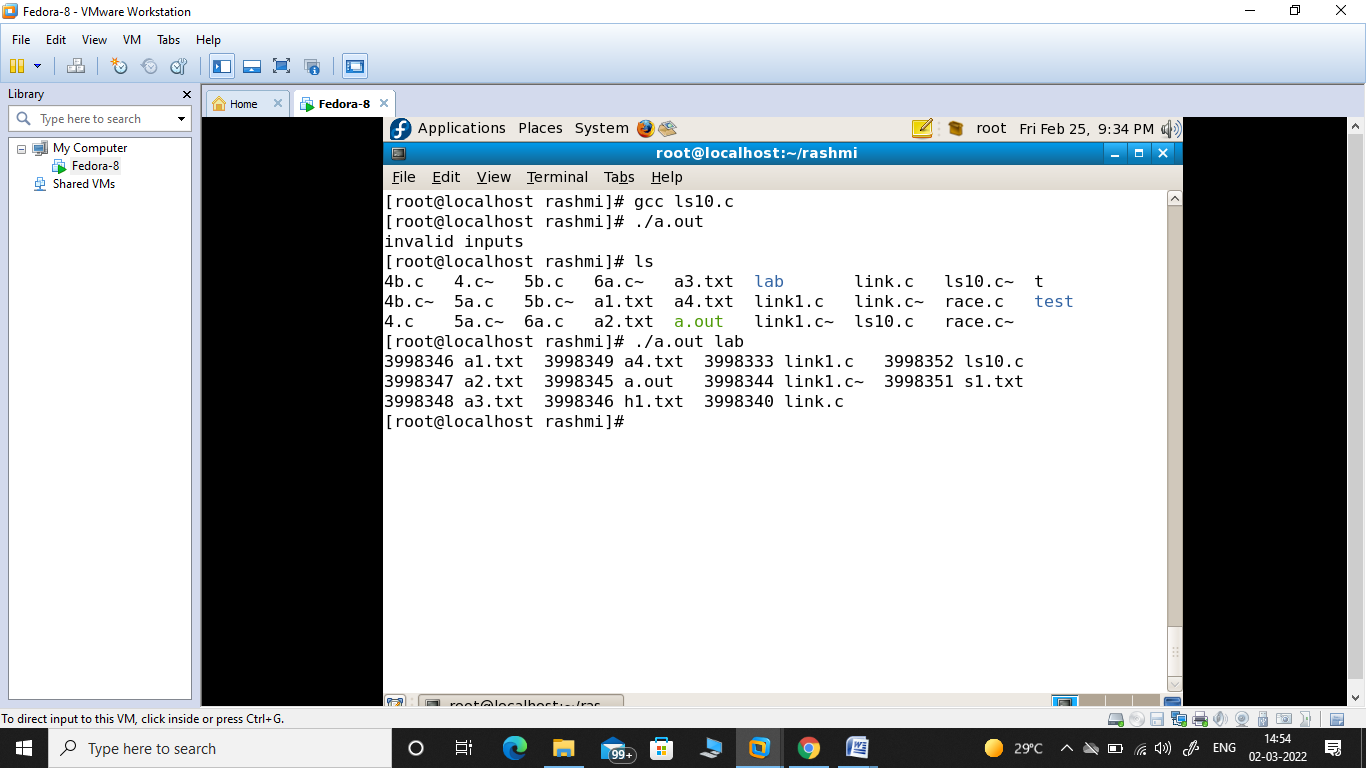
else

printf("\nInvalid No. of inputs");

return 0;

}

Output:



**10 a) Write a C/C++ program which demonstrates Inter process communication between a reader process and a writer process of a FIFO file by using the corresponding API’s.**

**Reader process:**

#include<stdio.h>

#include<unistd.h>

#include<fcntl.h>

int main()

{

int fd;

char a[100];

fd=open("/home/student/rashmi/fifo1",O\_RDONLY);

read(fd,a,100);

printf("%s\n",a);

//close(fd);

return 0;

}

**Writer Process:**

#include<stdio.h>

#include<unistd.h>

#include<fcntl.h>

int main()

{

int fd;

char a[50]=" this program is illustration of FIFO";

mkfifo("fifo1", 0666);

fd=open("/home/student/rashmi/fifo1",O\_WRONLY);

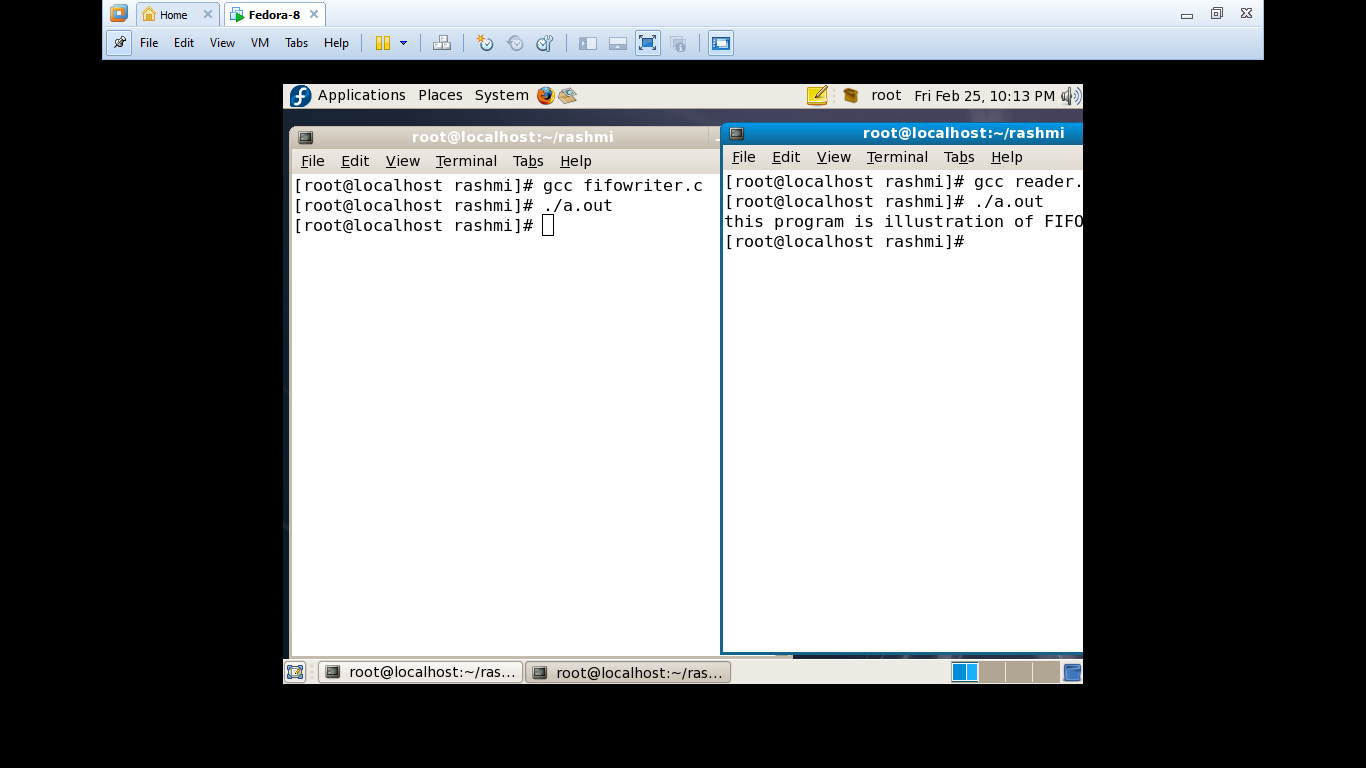
write(fd,a,sizeof(a));

//close(fd);

return 0;

}

Output:



**b) Write a C/C++ program which demonstrates the signal handler function to handle the signal sent by the process**

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

#include<stdlib.h>

void handler(int sig\_no)

{

printf("\ncatched signo is:%d\n\n",sig\_no);

}

int main()

{

signal(SIGTSTP,handler);

printf("\n going to sleep for few seconds\n");

printf(“\n process is interrupted with signal\n”);

printf(“press ctrl+z to interrupt\n”);

sleep(5);

return 0;

}

Output:

