**Assignment 4**

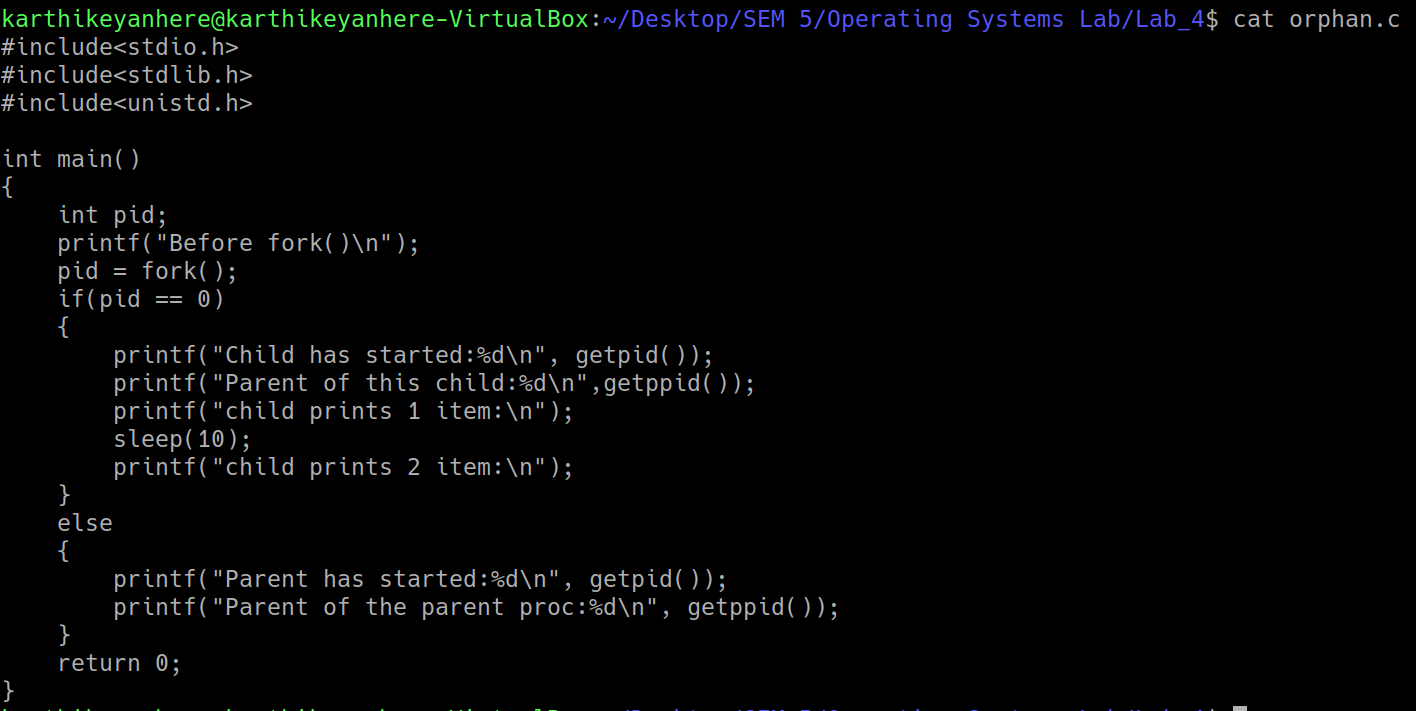
**T Karthikeyan**

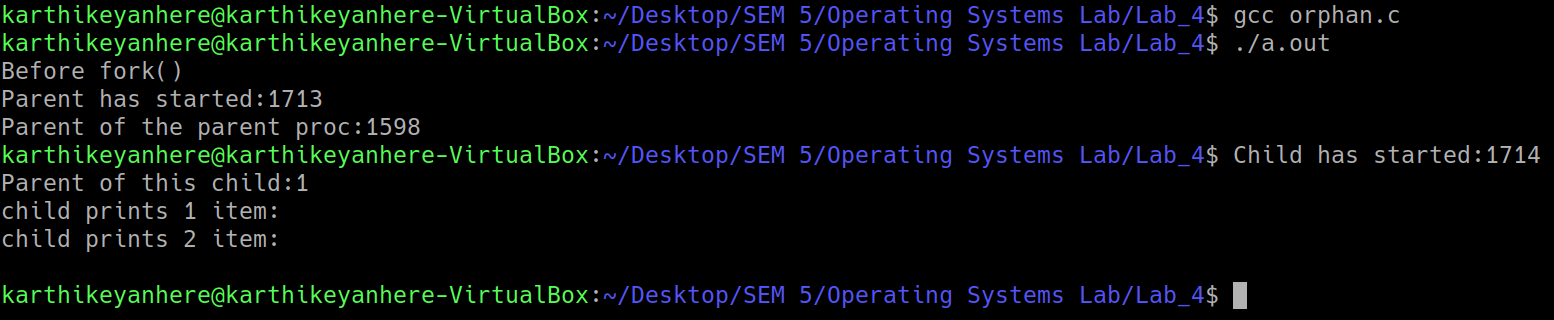
**CED18I064**

**All my codes can be found here** [**CLICK HERE**](https://drive.google.com/drive/folders/11J2WTPnIdweQYlI6MFG7s3kCX3syQ5jb?usp=sharing)

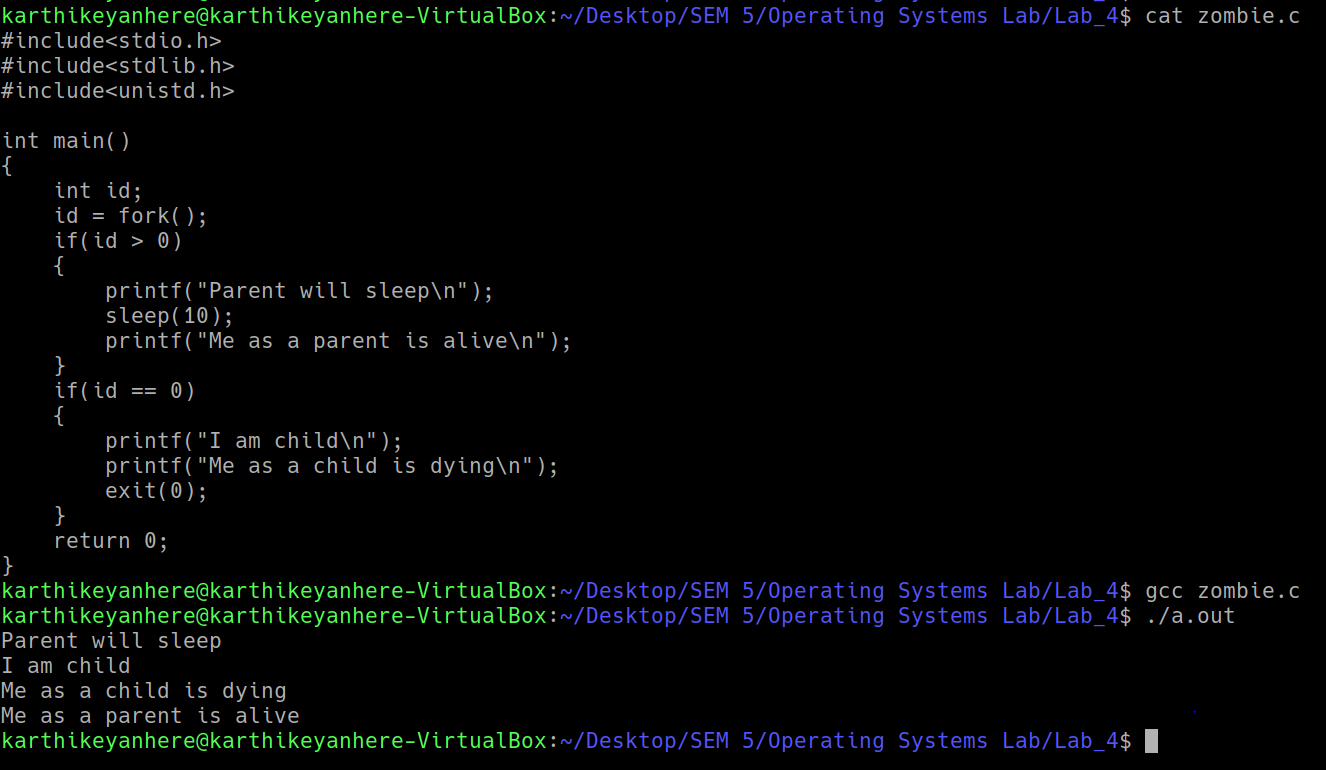
**(1) Test drive a C program that creates Orphan and Zombie Processes**

Orphan process

****

****

Zombie process

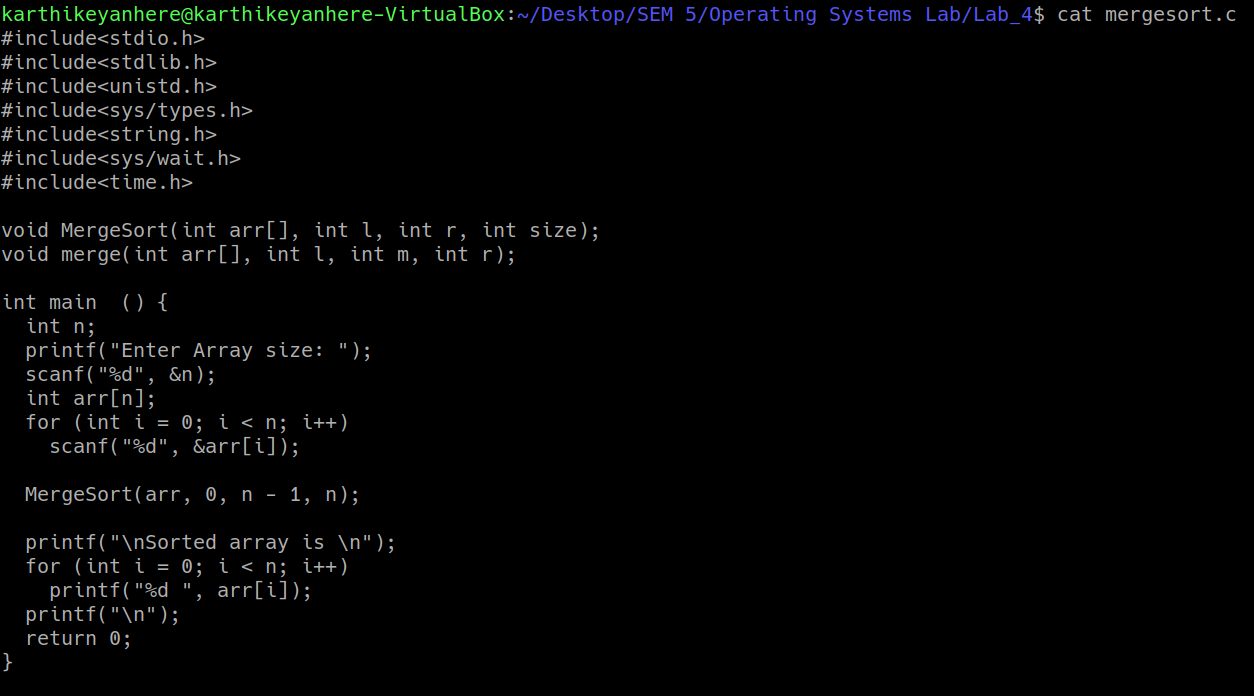
****

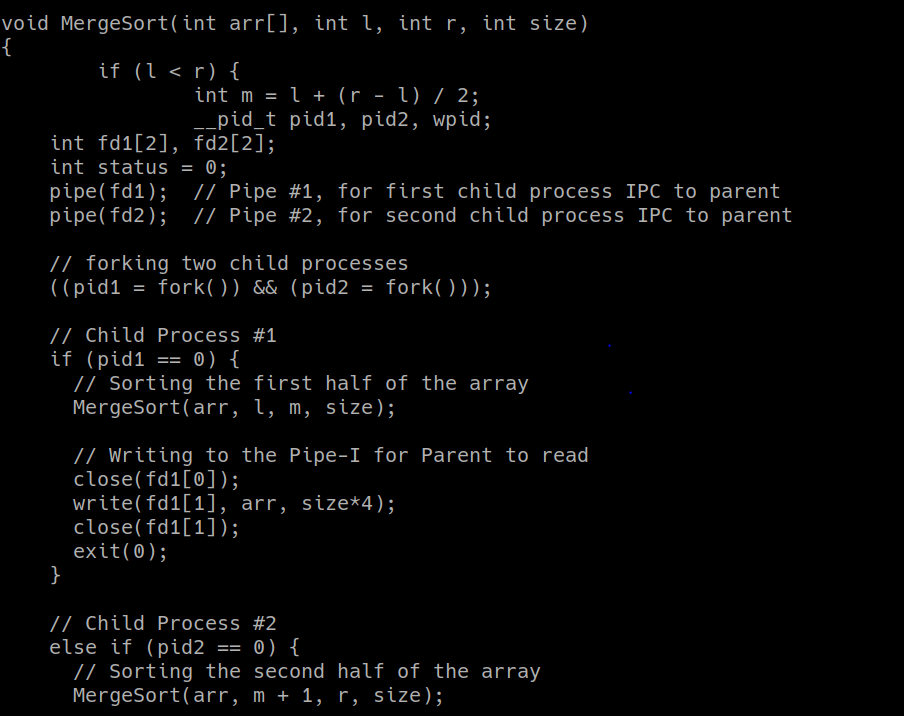
**(2) Develop a multiprocessing version of Merge or Quick Sort. Extra credits would be**

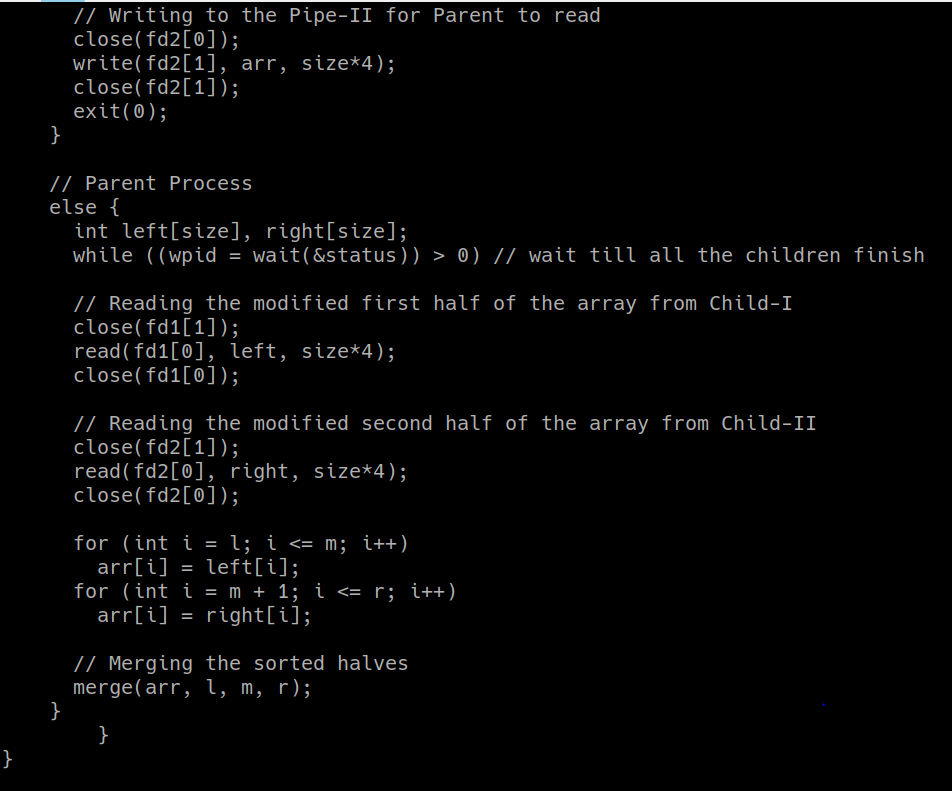
**given for those who implement both in a multiprocessing fashion [ increased no of**

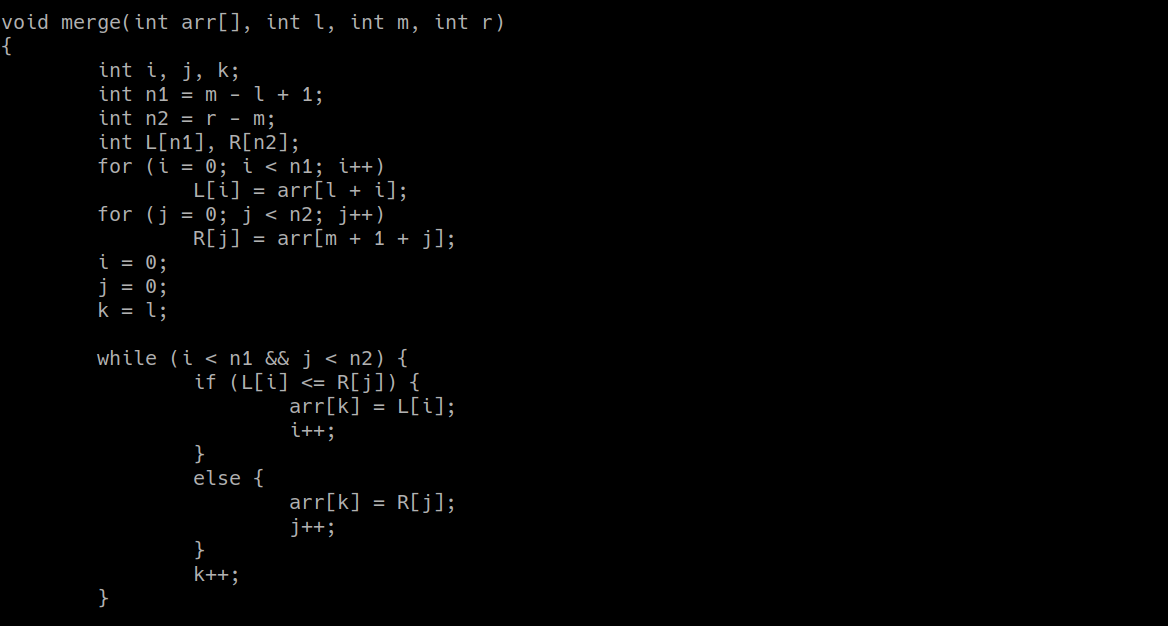
**processes to enhance the effect of parallelization]**

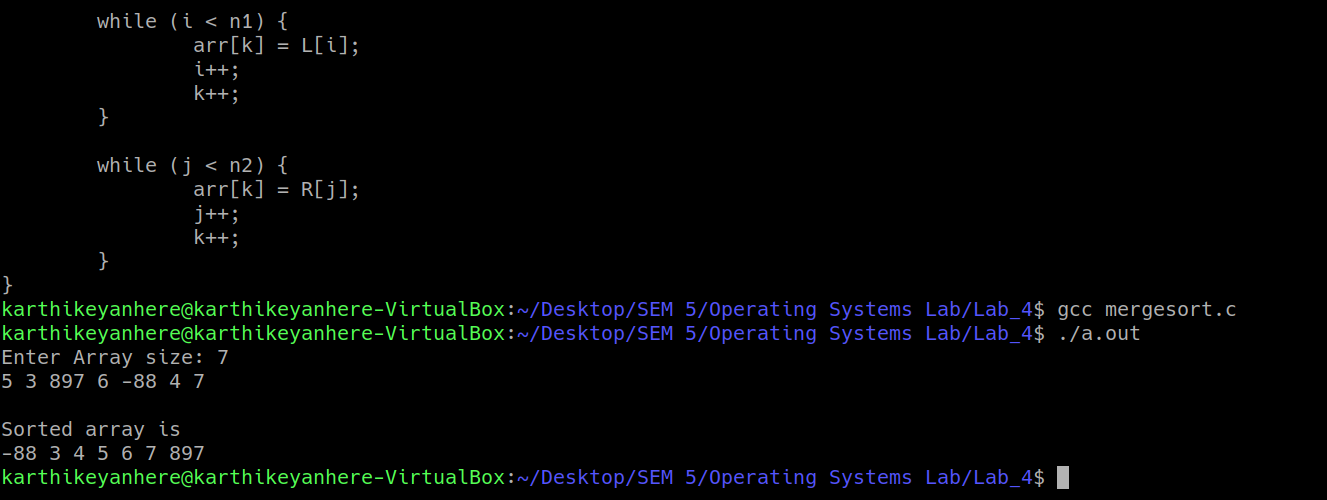
Multiprocessing version of merge sort : Two child processes are created at every mergesort function call recursively, each one of them sorts the half array allotted to them and returns the result to parent through dedicated pipes.











**(3) Develop a C program to count the maximum number of processes that can be**

**created using fork call.**

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int main()

{

int pid;

int i = 1;

for(;;)

{

pid = fork();

if(pid < 0)

{

printf("Maximum number of processes : %d\n", i);

exit(0);

}

else if(pid == 0)

{

i++;

}

else

{

wait(0);

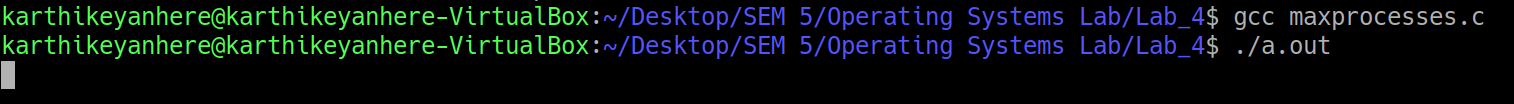
exit(0);

}

}

return 0;

}

****

my program made the system hang during execution, so i have to forcefully stop

**(4) Develop your own command shell [say mark it with @] that accepts user**

**commands (System or User Binaries), executes the commands and returns the prompt**

**for further user interaction. Also extend this to support a history feature (if the user**

**types !6 at the command prompt; it should display the most recent execute 6 commands).**

**You may provide validation features such as !10 when there are only 9 files to display**

**the entire history contents and other validations required for the history feature;**

With the help of Parser(), space separated input is converted into an array of strings which is in turn passed into execvp.

History feature is implemented with the help of global strings and queue.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

char lastsearched[10][128] = {"---", "---", "---", "---", "---", "---", "---", "---", "---", "---"};

void Parser(char \* buff, char \*\*args);

void RemoveNextLine(char \*buff);

void get\_pwd(char \*buff);

void record(char \*cmd);

void showrecord(int n);

int main()

{

while(1)

{

char buff[256];

char \*args[50];

char current\_path[256];

int status = 0;

pid\_t wpid;

get\_pwd(current\_path);

printf("\n[TKarthikeyan@iiitdm]:%s$", current\_path);

fgets(buff, sizeof(buff), stdin);

Parser(buff, args);

record(buff);

if(strcmp(args[0], "exit()") == 0)

{

printf("Thanks for using my shell\n");

exit(0);

}

else if(strcmp(args[0], "cd") == 0)

chdir(args[1]);

else if(args[0][0] == '!')

showrecord(args[0][1]-48);

else if(fork() == 0)

{

if(execvp(args[0], args) < 0)

printf("exec failed\n");

exit(0);

}

else

while((wpid = wait(&status)) > 0);

}

return 0;

}

void get\_pwd(char \*buff)

{

int fd[2];

pipe(fd);

if(fork() == 0)

{

close(fd[0]);

dup2(fd[1], 1);

execlp("pwd", "pwd", NULL);

}

else

{

close(fd[1]);

read(fd[0], buff, 256);

close(fd[0]);

RemoveNextLine(buff);

}

}

void Parser(char \*buff, char \*\*args)

{

int i = 0;

RemoveNextLine(buff);

char \*token = strtok(buff, " ");

args[i] = token;

i++;

while(1)

{

token = strtok(NULL, " ");

if(token == NULL)

break;

args[i] = token;

i++;

}

args[i] = (char \*)NULL;

}

void RemoveNextLine(char \*buff)

{

int i = 0;

while(buff[i] != '\n')

i++;

buff[i] = '\0';

}

void record(char \*cmd)

{

for(int i = 8; i >= 0; i--)

strcpy(lastsearched[i+1], lastsearched[i]);

strcpy(lastsearched[0], cmd);

}

void showrecord(int n)

{

if(n > 10)

{

printf("Sorry only last 10 can be retrieved\n");

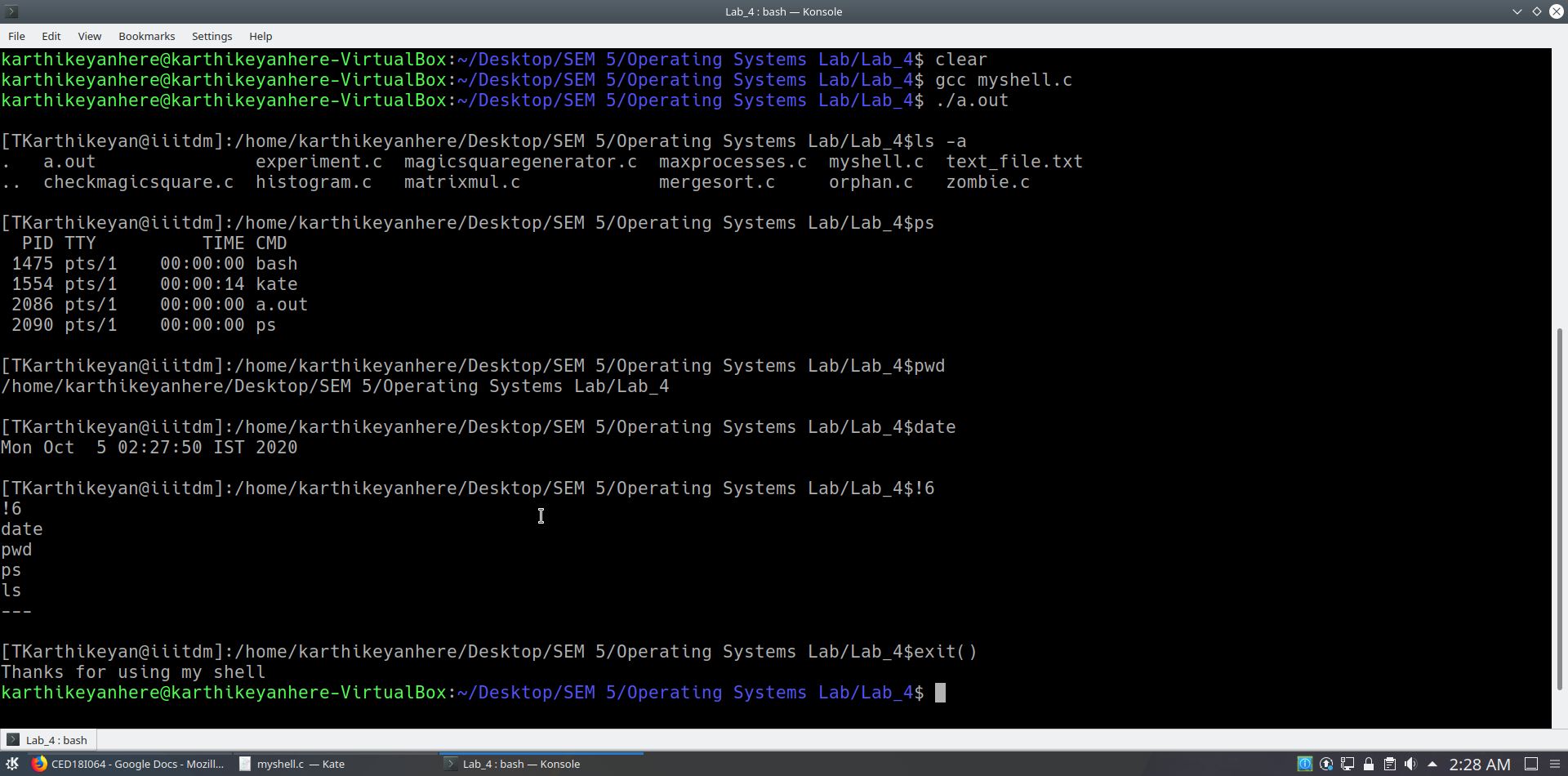
n = 10;

}

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

printf("%s\n", lastsearched[i]);

}



**(5) Develop a multiprocessing version of Histogram generator to count the occurrence**

**of various characters in a given text.**

18 children were created using a fork, each child was given to count the occurrence of 5 characters. Rest characters were managed by the parent

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

void CreateHistogram(char \* content, int size);

int main () {

char \* content = "CED18I064 !!!!!! T Karthikeyan 7438947#$^&#$%#^&&\*&\*@~!#%$?}{:>}?";

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

printf("-------------Histogram generator-------------\n");

CreateHistogram(content, strlen(content));

return 0;

}

void CreateHistogram(char \* content, int size) {

int pid1, pid2, pid3, pid4, pid5, pid6, pid7, pid8, pid9, pid10, pid11, pid12, pid13, pid14, pid15, pid16, pid17, pid18, pid19, pid20, pid21, pid22, pid23, pid24, pid25, wpid;

int status = 0;

// parent will take care of remaining characters

// children have to take care of characters in set of 5

pid1 = fork();

if (pid1 == 0)// Computes the frequency of ASCII 33 - 37

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-33);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",33+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid2 = fork();

if (pid2 == 0)// Computes the frequency of ASCII 38 - 42

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-38);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",38+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid3 = fork();

if (pid3 == 0)// Computes the frequency of ASCII 43 - 47

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-43);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",43+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid4 = fork();

if (pid4 == 0)// Computes the frequency of ASCII 48 - 52

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-48);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",48+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid5 = fork();

if (pid5 == 0)// Computes the frequency of ASCII 53 - 57

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-53);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",53+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid6 = fork();

if (pid6 == 0)// Computes the frequency of ASCII 58 - 62

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-58);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",58+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid7 = fork();

if (pid7 == 0)// Computes the frequency of ASCII 63 - 67

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-63);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",63+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid8 = fork();

if (pid8 == 0)// Computes the frequency of ASCII 68 - 72

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-68);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",68+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid9 = fork();

if (pid9 == 0)// Computes the frequency of ASCII 73-77

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-73);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",73+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid10 = fork();

if (pid10 == 0)// Computes the frequency of ASCII 78-82

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-78);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",78+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid11 = fork();

if (pid11 == 0)// Computes the frequency of ASCII 83-87

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-83);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",83+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid12 = fork();

if (pid12 == 0)// Computes the frequency of ASCII 88-92

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-88);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",88+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid13 = fork();

if (pid13 == 0)// Computes the frequency of ASCII 93-97

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-93);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",93+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid14 = fork();

if (pid14 == 0)// Computes the frequency of ASCII 98-102

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-98);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",98+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid15 = fork();

if (pid15 == 0)// Computes the frequency of ASCII 103-107

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-103);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",103+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid16 = fork();

if (pid16 == 0)// Computes the frequency of ASCII 108-112

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-108);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",108+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid17 = fork();

if (pid17 == 0)// Computes the frequency of ASCII 113-117

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-113);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",113+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else

{

pid18 = fork();

if (pid18 == 0)// Computes the frequency of ASCII 118-122

{

int count[5] = {0,0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-118);

if (temp >=0 && temp <=4)

{

count[temp]++;

}

}

for(int k = 0; k < 5; k++)

{

printf("%c :",118+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

}

else // ASCII 123-126

{

while ((wpid = wait(&status)) > 0);

int count[4] = {0,0,0,0};

for(int i = 0; i < size; i++)

{

int temp = (content[i]-123);

if (temp >=0 && temp <=3)

{

count[temp]++;

}

}

for(int k = 0; k < 4; k++)

{

printf("%c :",123+k);

for(int j = 0; j < count[k]; j++)

printf("(^\_^)");

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

}

printf("\n-------------------------------------\n");

}//19

} //18

} //17

}//16

}//15

} //14

}//13

}//12

} //11

} //10

} //9

} //8

} //7

} //6

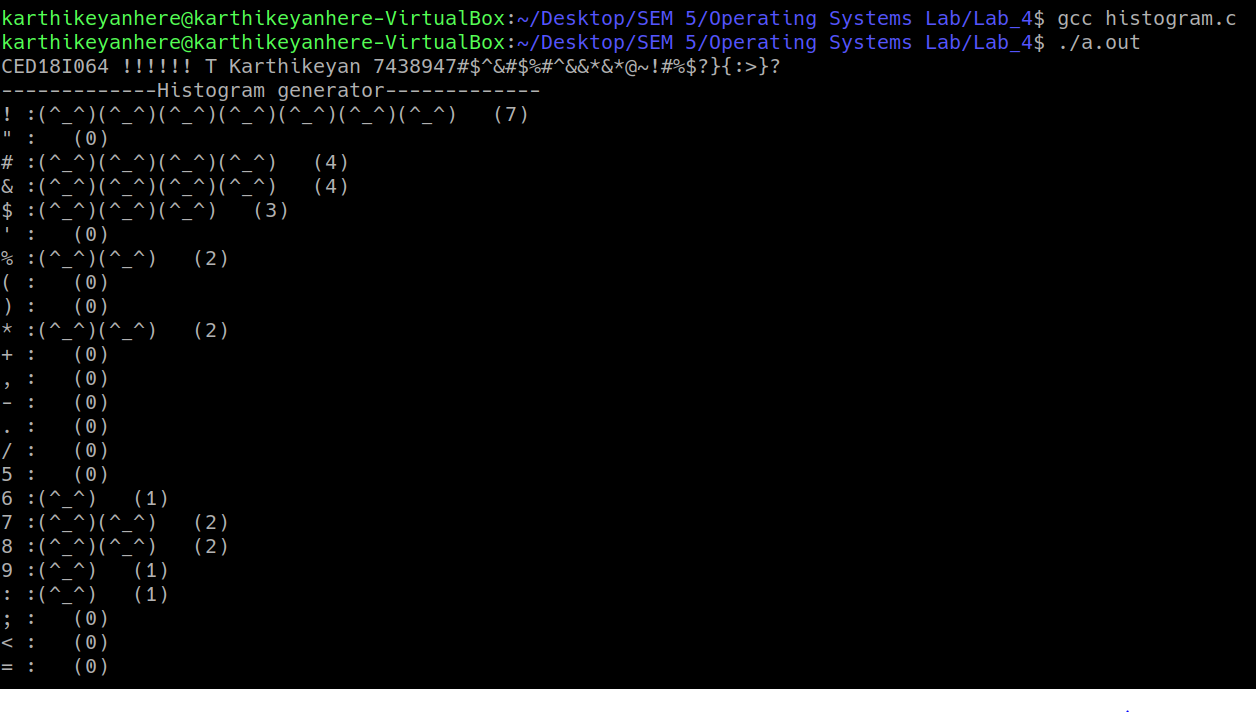
} //5

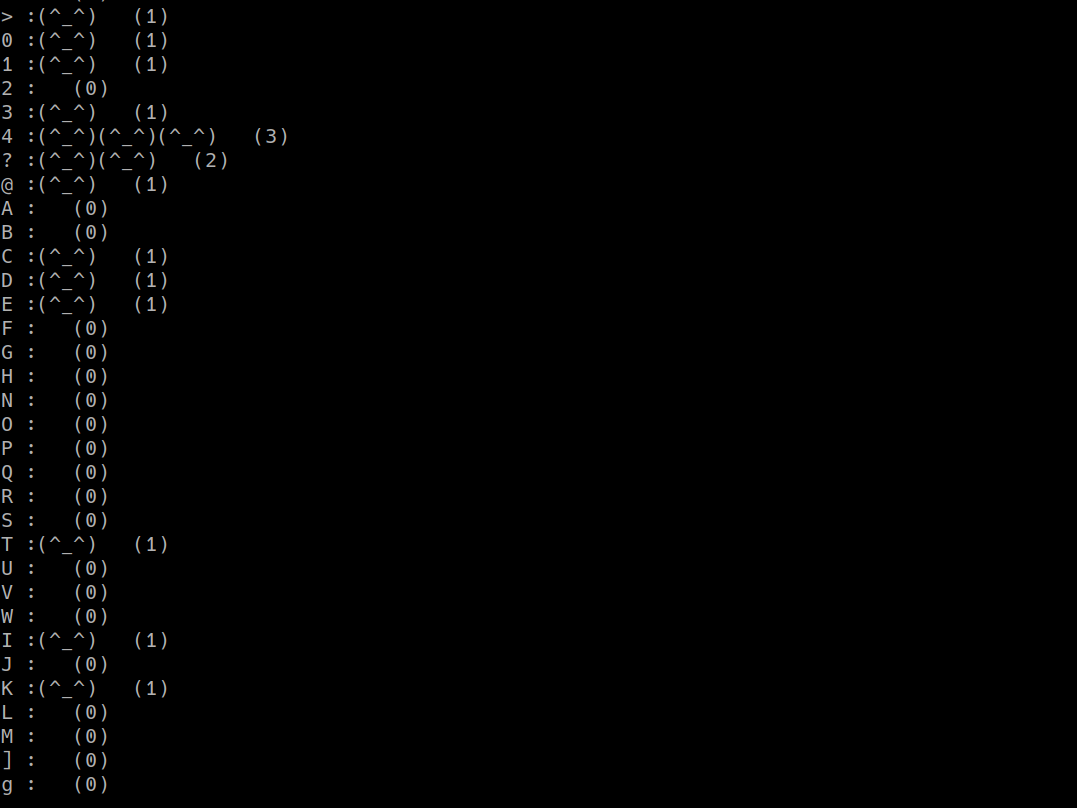
} //4

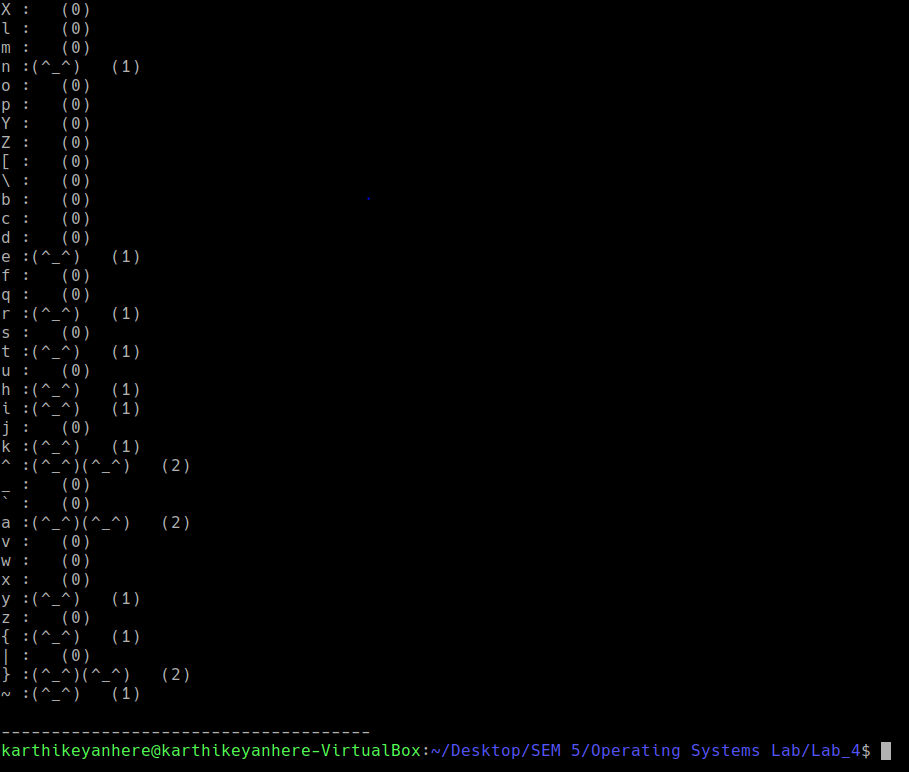
} //3

} //2

} //1

****

****

****

**(6) Develop a multiprocessing version of matrix multiplication. Say for a result 3\*3 matrix**

**the most efficient form of parallelization can be 9 processes, each of which computes**

**the net resultant value of a row (matrix1) multiplied by column (matrix2). For**

**programmers convenience you can start with 4 processes, but as I said each result**

**value can be computed parallel independent of the other processes in execution.**

9 processes(child) are created), each one will calculate sum of respective row and column and pass on the result to parent through dedicated pipes

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<time.h>

int main()

{

int arr[3][3] = {1,2,3,4,5,6,7,8,9};

int brr[3][3] = {1,1,1,1,1,1,1,1,1};

int res[3][3] = {0,0,0,0,0,0,0,0,0};

printf("Printing array A:\n");

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

printf("%d ",arr[i][j]);

printf("\n");

}

printf("Printing array B:\n");

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

printf("%d ",brr[i][j]);

printf("\n");

}

int fd1[2], fd2[2], fd3[2], fd4[2], fd5[2], fd6[2], fd7[2], fd8[2], fd9[2];

pipe(fd1); pipe(fd2); pipe(fd3); pipe(fd4); pipe(fd5); pipe(fd6); pipe(fd7); pipe(fd8); pipe(fd9);

int pid1 = fork();

if(pid1 == 0) // 1st row 1st column

{

int temp = arr[0][0]\*brr[0][0] + arr[0][1]\*brr[1][0] + arr[0][2]\*brr[2][0];

close(fd1[0]);

write(fd1[1], &temp, 4);

close(fd1[1]);

}

else

{

int pid2 = fork();

if(pid2 == 0) // 1st row 2nd column

{

int temp = arr[0][0]\*brr[0][1] + arr[0][1]\*brr[1][1] + arr[0][2]\*brr[2][1];

close(fd2[0]);

write(fd2[1], &temp, 4);

close(fd2[1]);

}

else

{

int pid3 = fork();

if(pid3 == 0) // 1st row 3rd column

{

int temp = arr[0][0]\*brr[0][2] + arr[0][1]\*brr[1][2] + arr[0][2]\*brr[2][2];

close(fd3[0]);

write(fd3[1], &temp, 4);

close(fd3[1]);

}

else

{

int pid4 = fork();

if(pid4 == 0) // 2nd row 1st column

{

int temp = arr[1][0]\*brr[0][0] + arr[1][1]\*brr[1][0] + arr[1][2]\*brr[2][0];

close(fd4[0]);

write(fd4[1], &temp, 4);

close(fd4[1]);

}

else

{

int pid5 = fork();

if(pid5 == 0) // 2nd row 2nd column

{

int temp = arr[1][0]\*brr[0][1] + arr[1][1]\*brr[1][1] + arr[1][2]\*brr[2][1];

close(fd5[0]);

write(fd5[1], &temp, 4);

close(fd5[1]);

}

else

{

int pid6 = fork();

if(pid6 == 0) // 2nd row 3rd column

{

int temp = arr[1][0]\*brr[0][2] + arr[1][1]\*brr[1][2] + arr[1][2]\*brr[2][2];

close(fd6[0]);

write(fd6[1], &temp, 4);

close(fd6[1]);

}

else

{

int pid7 = fork();

if(pid7 == 0) // 3rd row 1st column

{

int temp = arr[2][0]\*brr[0][0] + arr[2][1]\*brr[1][0] + arr[2][2]\*brr[2][0];

close(fd7[0]);

write(fd7[1], &temp, 4);

close(fd7[1]);

}

else

{

int pid8 = fork();

if(pid8 == 0) // 3rd row 2nd column

{

int temp = arr[2][0]\*brr[0][1] + arr[2][1]\*brr[1][1] + arr[2][2]\*brr[2][1];

close(fd8[0]);

write(fd8[1], &temp, 4);

close(fd8[1]);

}

else

{

int pid9 = fork();

if(pid9 == 0) // 3rd row 3rd column

{

int temp = arr[2][0]\*brr[0][2] + arr[2][1]\*brr[1][2] + arr[2][2]\*brr[2][2];

close(fd9[0]);

write(fd9[1], &temp, 4);

close(fd9[1]);

}

else

{

close(fd1[1]);

read(fd1[0], &res[0][0], 4);

close(fd1[0]);

close(fd2[1]);

read(fd2[0], &res[0][1], 4);

close(fd2[0]);

close(fd3[1]);

read(fd3[0], &res[0][2], 4);

close(fd3[0]);

close(fd4[1]);

read(fd4[0], &res[1][0], 4);

close(fd4[0]);

close(fd5[1]);

read(fd5[0], &res[1][1], 4);

close(fd5[0]);

close(fd6[1]);

read(fd6[0], &res[1][2], 4);

close(fd6[0]);

close(fd7[1]);

read(fd7[0], &res[2][0], 4);

close(fd7[0]);

close(fd8[1]);

read(fd8[0], &res[2][1], 4);

close(fd8[0]);

close(fd9[1]);

read(fd9[0], &res[2][2], 4);

close(fd9[0]);

printf("\nPrinting result array:");

for(int i = 0; i < 3; i++)

{

printf("\n");

for(int j = 0; j < 3; j++)

printf("%d ",res[i][j]);

}

printf("\n");

}//9

}//8

}//7

}//6

}//5

}//4

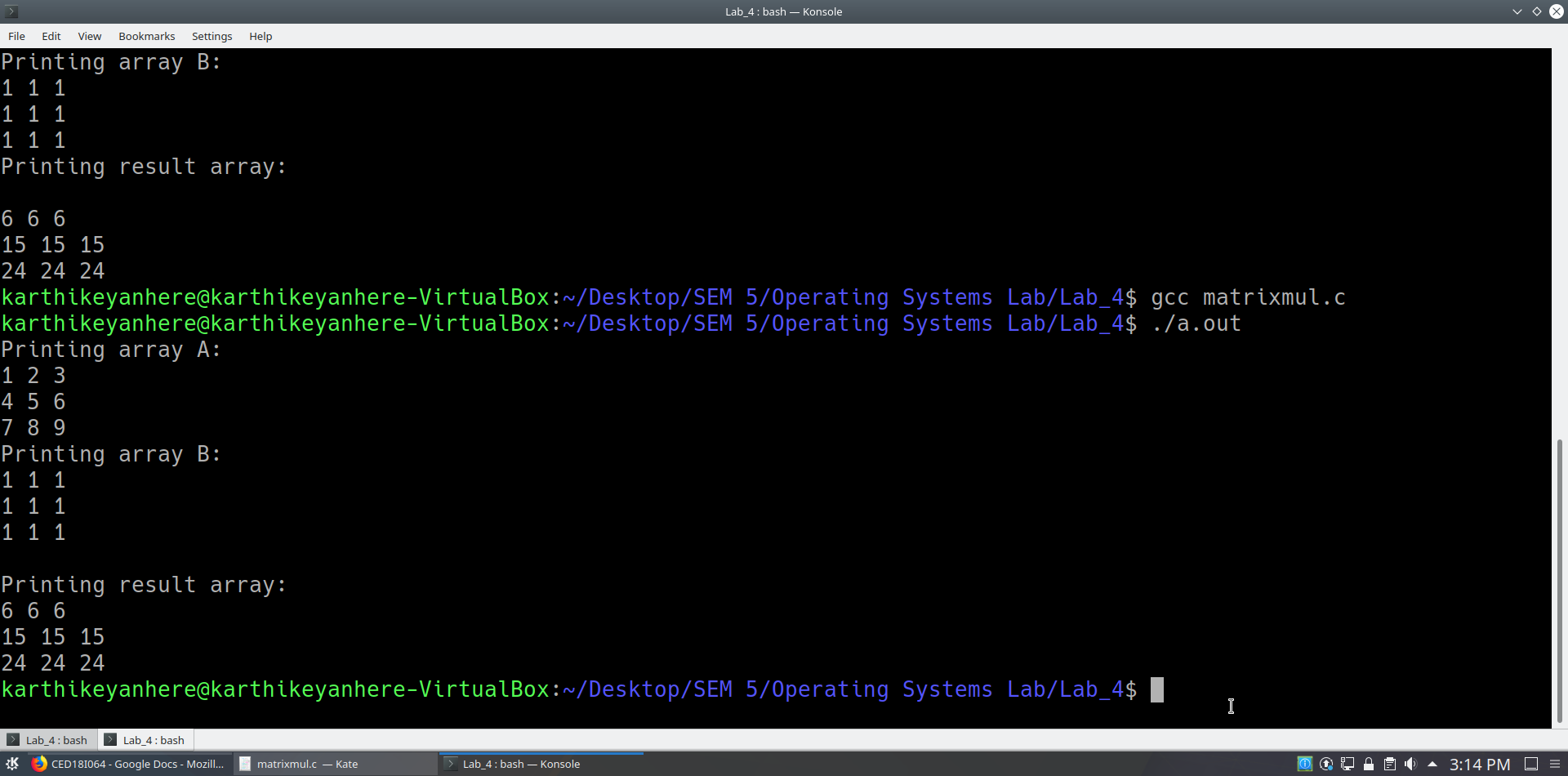
}//3

}//2

}//1

return 0;

}

****

**Non Mandatory (Extra Credits)..**

**(7) Develop a parallelized application to check for if a user input square matrix is a**

**magic square or not. No of processes again can be optimal as w.r.t to matrix exercise**

**above.**

3 child processes are used to verify row sum, column sum and diagonal sum respectively and each one of them will produce their result to parent, parent will make the final decision whether its magic square or not

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<time.h>

int main()

{

int n;

printf("Enter the order of matrix:");

scanf("%d",&n);

int arr[n][n];

printf("Enter the matrix in row-wise manner(space separated):");

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

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

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

printf("Entered matrix:\n");

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

{

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

printf("%d ",arr[i][j]);

printf("\n");

}

int fd1[2], fd2[2], fd3[2];

pipe(fd1);

pipe(fd2);

pipe(fd3);

int pid1 = fork();

if(pid1 == 0) // take care of row sum

{

int temp = 0;

int rowsum[n];

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

{

rowsum[i] = 0;

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

rowsum[i]+=arr[i][j];

}

for(int k = 0; k < n-1; k++)

{

if(rowsum[k]!=rowsum[k+1])

{

temp = -1;

break;

}

}

if(temp!=-1)

temp = rowsum[0];

printf("rowflag:%d\n",temp);

close(fd1[0]);

write(fd1[1], &temp, 4);

close(fd1[1]);

}

else

{

int pid2 = fork();

if(pid2 == 0) // take care of column sum

{

int temp = 0;

int colsum[n];

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

{

colsum[i] = 0;

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

colsum[i]+=arr[j][i];

}

for(int k = 0; k < n-1; k++)

{

if(colsum[k]!=colsum[k+1])

{

temp = -1;

break;

}

}

if(temp!=-1)

temp = colsum[0];

printf("colflag:%d\n",temp);

close(fd2[0]);

write(fd2[1], &temp, 4);

close(fd2[1]);

}

else

{

int pid3 = fork();

if(pid3 == 0) // take care of diagonal sum

{

int temp = 0;

int diagsum[2] = {0,0};

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

{

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

{

if(i==j)

diagsum[0]+=arr[i][j];

if(i+j == n-1)

diagsum[1]+=arr[i][j];

}

}

if(diagsum[0] == diagsum[1])

temp = diagsum[0];

else

temp = -1;

printf("diagflag:%d\n",temp);

close(fd3[0]);

write(fd3[1], &temp, 4);

close(fd3[1]);

}

else // will integrate the separate results

{

int rowflag, colflag, diagflag;

close(fd1[1]);

read(fd1[0], &rowflag, 4);

close(fd1[0]);

close(fd2[1]);

read(fd2[0], &colflag, 4);

close(fd2[0]);

close(fd3[1]);

read(fd3[0], &diagflag, 4);

close(fd3[0]);

if(rowflag == -1 || colflag == -1 || diagflag == -1)

printf("Not a magic square\n");

else

{

if(rowflag == colflag && colflag == diagflag)

printf("It is a magic square\n");

else

printf("Not a magic square\n");

} //4

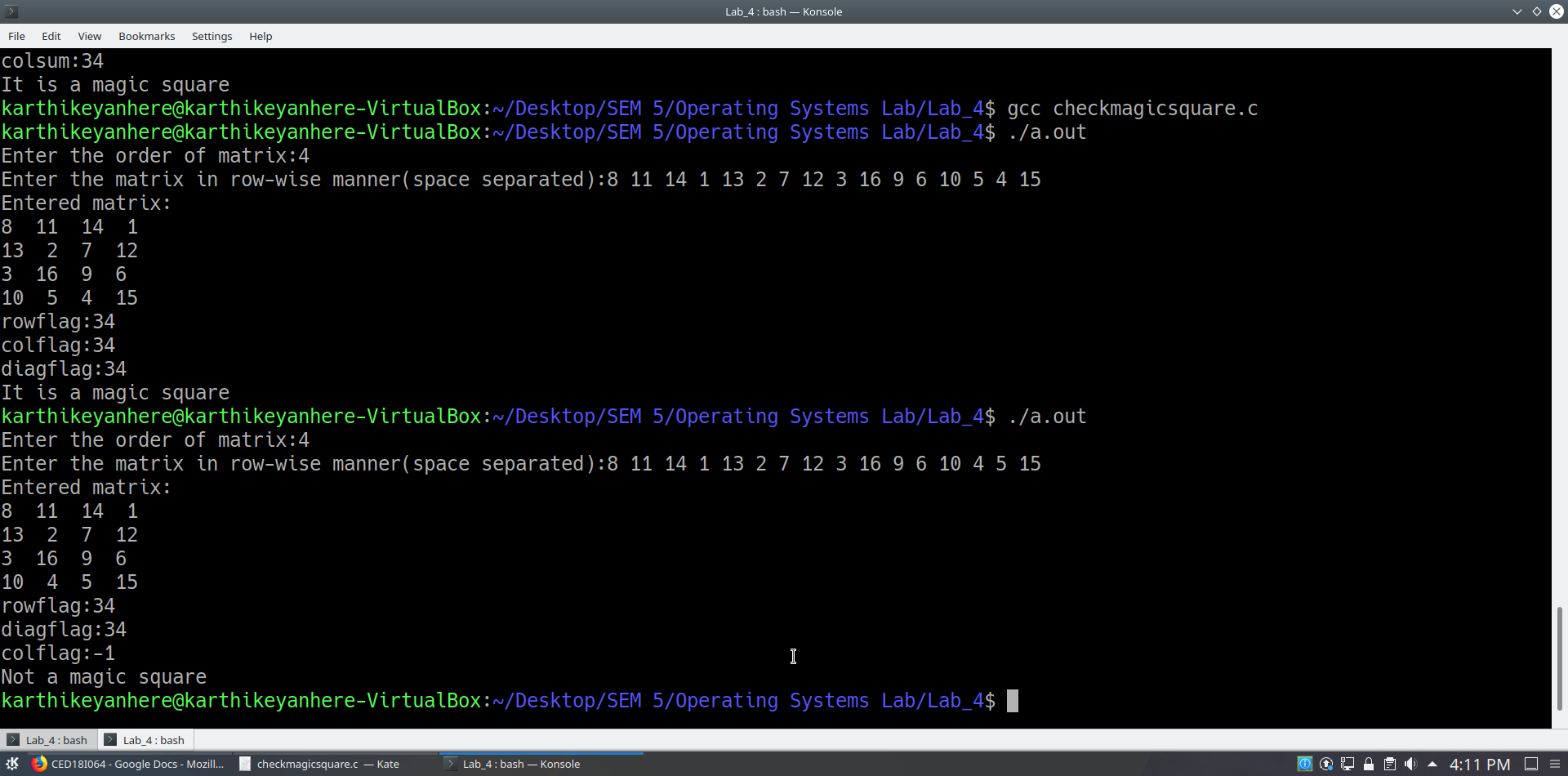
}//3

}//2

}//1

return 0;

}

****

**(8) Extend the above to also support magic square generation (u can take as input the**

**order of the matrix..refer the net for algorithms for odd and even versions…)**

Child process created using vfork generates the magic square using any of the three algorithms based on the order of the matrix and parent waits for the child process to finish, it displays the resultant magic square

1. N = 3, 5, 7, … (odd order magic square)
2. N = 4, 8, 12, 16, … (doubly even order magic square)
3. N = 6, 10, 14, … (singly even order magic square)

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>

#define MAX 100000

void MagicSquare(int size, int arr[][size]);

void OddOrderMagicSquare(int size, int arr[][size]);

void DoublyEvenMagicSquare(int size, int arr[][size]);

void SinglyEvenMagicSquare(int size, int arr[][size]);

void displayMagicSquare(int size, int arr[][size]);

int main()

{

int size;

printf("Enter the order of the matrix:");

scanf("%d",&size);

int arr[size][size];

int pid;

pid = vfork();

if(pid == 0)

{

if(size < 3)

{

printf("Order of the matrix should be greater than 2\n");

exit(-1);

}

MagicSquare(size,arr);

exit(0);

}

else

{

wait(NULL);

displayMagicSquare(size, arr);

}

return 0;

}

void MagicSquare(int size, int arr[][size])

{

if(size%2 == 1)

OddOrderMagicSquare(size, arr);

else if(size%4 == 0)

DoublyEvenMagicSquare(size, arr);

else

SinglyEvenMagicSquare(size, arr);

}

void OddOrderMagicSquare(int size, int arr[][size])

{

int square = size\*size;

int i = 0, j = size/2, k;

for(k = 1; k <= square; ++k)

{

arr[i][j] = k;

i--;

j++;

if(k%size == 0)

{

i = i+2;

--j;

}

else

{

if(j == size)

j = j - size;

else if(i < 0)

i = i + size;

}

}

}

void DoublyEvenMagicSquare(int size, int arr[][size])

{

int I[size][size];

int J[size][size];

int i, j;

int index = 1;

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

{

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

{

I[i][j] = ((i+1)%4)/2;

J[j][i] = ((i+1)%4)/2;

arr[i][j] = index;

index++;

}

}

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

{

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

{

if(I[i][j] == J[i][j])

arr[i][j] = size\*size - arr[i][j] + 1;

}

}

}

void SinglyEvenMagicSquare(int size, int arr[][size])

{

int N = size;

int halfN = N/2;

int k = (N-2)/4;

int temp;

int swapCol[N];

int index = 0;

int miniMagic[halfN][halfN];

OddOrderMagicSquare(halfN, miniMagic);

for(int i = 0; i < halfN; i++)

for(int j = 0; j < halfN; j++)

{

arr[i][j] = miniMagic[i][j]; //A

arr[i+halfN][j+halfN] = miniMagic[i][j] + halfN\*halfN; //B

arr[i][j+halfN] = miniMagic[i][j] + 2\*halfN\*halfN; //C

arr[i+halfN][j] = miniMagic[i][j] + 3\*halfN\*halfN; //D

}

for(int i = 1; i <= k; i++)

swapCol[index++] = i;

for(int i = N-k+2; i <= N; i++)

swapCol[index++] = i;

for(int i = 1; i <= halfN; i++)

for(int j = 1; j <= index; j++)

{

temp = arr[i-1][swapCol[j-1]-1];

arr[i-1][swapCol[j-1]-1] = arr[i+halfN-1][swapCol[j-1] - 1];

arr[i+halfN-1][swapCol[j-1]-1] = temp;

}

temp = arr[k][0];

arr[k][0] = arr[k+halfN][0];

arr[k+halfN][0] = temp;

temp = arr[k+halfN][k];

arr[k+halfN][k] = arr[k][k];

arr[k][k] = temp;

}

void displayMagicSquare(int size, int arr[][size])

{

printf("Magic number:%d\n", size\*(size\*size + 1)/2);

for(int i = 0; i < size; i++)

{

for(int j = 0; j < size; j++)

printf("%5d ", arr[i][j]);

printf("\n");

}

}

