<u>OPERATING SYSTEMS PRACTICE</u>
(ASSIGNMENT 4)

<u>Name</u> – Mridul Harish
<u>Roll number</u> – CED18I034

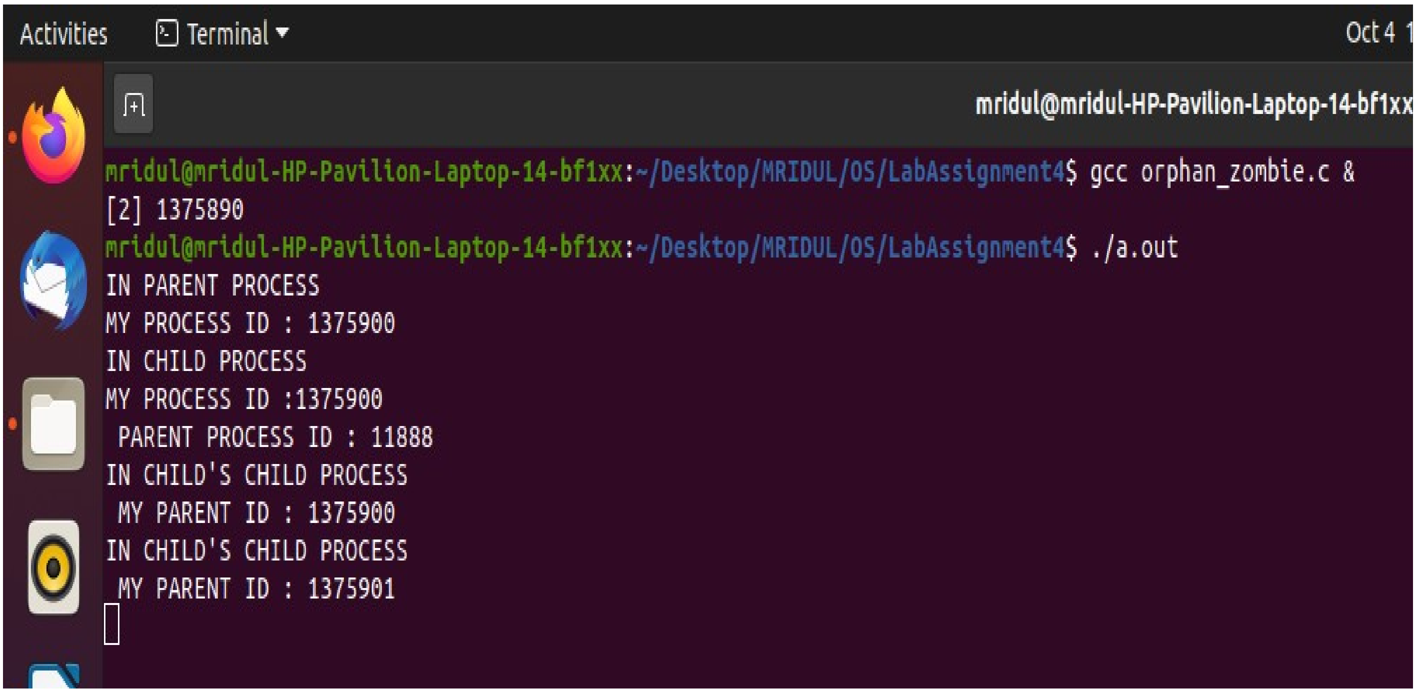
<u>Question 1</u>: Test drive a C program that creates Orphan and Zombie Processes.

<u>Code</u> :-

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{

	pid_t pid = fork();

	if (pid > 0)
	{
	printf("IN PARENT PROCESS\nMY PROCESS ID : %d\n",
getpid());
	}
	else if (pid == 0)
	{
	sleep(5);
	pid_t pid = fork();
	}

	if (pid > 0)
	{
		printf("IN CHILD PROCESS\nMY PROCESS ID :%d\n
PARENT PROCESS ID : %d\n", getpid(), getppid());

	while(1)
```

```
            sleep(1);

        printf("IN CHILD PROCESS\nMY PARENT PROCESS ID : %d\n",
getppid());
        }
        else if (pid == 0)
        {
            printf("IN CHILD'S CHILD PROCESS\n MY PARENT ID :
%d\n", getppid());
        }

        return 0;
}
```

<u>Output</u> :-



<u>Explanation</u> :-

In the above code, we have made a scenario that there is a parent and it has a child and that child also has a child, firstly if our process gets into child process, we put our system into sleep for 5 sec so that we could finish up the parent process so that its child become orphan, then we have made a child's child as zombie process, the child's child finishes its execution while the parent(i.e child) sleeps for 1 seconds, hence the child's child doesn't call terminate, and it's entry still exists in the process table.

<u>Question 2</u>: 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]

<u>Code</u> :-
(Merge Sort)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<unistd.h>
#include<time.h>

void merge(int a[], int low, int mid, int high);
void merge_sort_parallel(int a[], int low, int high);
void merge_sort(int a[], int low, int high);


int main()
{
	int n;
	clock_t t1, t2;

	printf("Enter the size of the Array: ");
	scanf("%d", &n);

	int a1[n]; int a2[n]; int x;

	printf("Enter the elements\n");
	for(int i = 0; i < n; i = i+1)
	{
	scanf("%d", &x);
	a1[i] = a2[i] = x;
	}
```

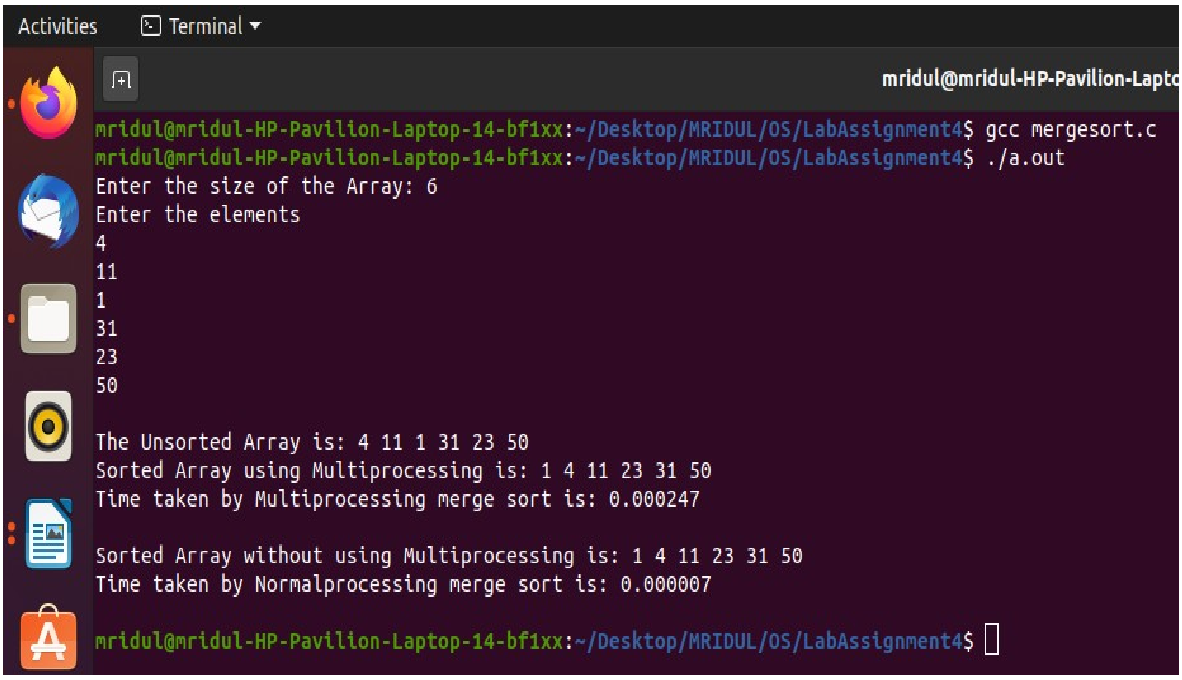
```
	printf("\nThe Unsorted Array is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a1[i]);
	}



	t1 = clock();
	merge_sort_parallel(a1, 0, n-1);
	t2 = clock();

	printf("\nSorted Array using Multiprocessing is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a1[i]);
	}

	printf("\nTime taken by Multiprocessing merge sort is: %lf\n", (t2 -
t1) / (double) CLOCKS_PER_SEC);

	t1 = clock();
	merge_sort(a2, 0, n-1);
	t2 = clock();

	printf("\nSorted Array without using Multiprocessing is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a2[i]);
	}

	printf("\nTime taken by Normalprocessing merge sort is: %lf\n\n",
(t2 - t1) / (double) CLOCKS_PER_SEC);

	return 0;
}

void merge(int a[], int low, int mid, int high)
{
```

```
int i; int j; int k;
int n1 = mid - low + 1;
int n2 = high - mid;

int left[n1], right[n2];

for (i = 0; i < n1; i = i+1)
{
left[i] = a[low + i];
}
for (j = 0; j < n2; j = j+1)
{
right[j] = a[mid + 1 + j];
}

i = 0;
j = 0;
k = low;

while (i < n1 && j < n2)
{
if (left[i] <= right[j])
{
        a[k] = left[i];
        i = i+1;
}
else
{
        a[k] = right[j];
        j = j+1;
}
k = k+1;
}

while (i < n1)
{
a[k] = left[i];
i = i+1;
k = k+1;
```

```
	}

	while (j < n2)
	{
	a[k] = right[j];
	j = j+1;
	k = k+1;
	}
}

void merge_sort_parallel(int a[], int low, int high)
{
	if(low < high)
	{
	int mid = low + (high - low) / 2;
	pid_t pid;
	pid = vfork();

	if(pid == 0)
	{
			merge_sort_parallel(a, low, mid);
			exit(0);
	}
	else
	{
		merge_sort_parallel(a, mid + 1, high);
		merge(a, low, mid, high);
	}
	}
}

void merge_sort(int a[], int low, int high)
{
	if(low < high)
	{
	int mid = low + (high - low) / 2;
	merge_sort(a, low, mid);
	merge_sort(a, mid + 1, high);
	merge(a, low, mid, high);
```

```
	}
}
```

<u>Output</u> :-



<u>Explanation</u> :-

In the above code where we normally execute the “divide” operation of the array into 2^n segments for later “conquer”, we call the “vfork” system call for each divide operation, which in turn leads to parallelization of each conquer operation.

<u>Code</u> :-
(Quick Sort)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<unistd.h>
#include<time.h>

int partition(int a[], int low, int high);
void quick_sort_parallel(int a[], int low, int high);
```

```
void quick_sort(int a[], int low, int high);

int main()
{
	int n;
	clock_t t1, t2;

	printf("Enter the size of the Array: ");
	scanf("%d", &n);

	int a1[n]; int a2[n]; int x;

	printf("Enter the elements\n");
	for(int i = 0; i < n; i = i+1)
	{
	scanf("%d", &x);
	a1[i] = a2[i] = x;
	}

	printf("\nThe Unsorted Array is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a1[i]);
	}

	t1 = clock();
	quick_sort_parallel(a1, 0, n-1);
	t2 = clock();

	printf("\nSorted Array using Multiprocessing is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a1[i]);
	}

	printf("\nTime taken by Multiprocessing merge sort is: %lf\n", (t2 -
t1) / (double) CLOCKS_PER_SEC);

	t1 = clock();
```

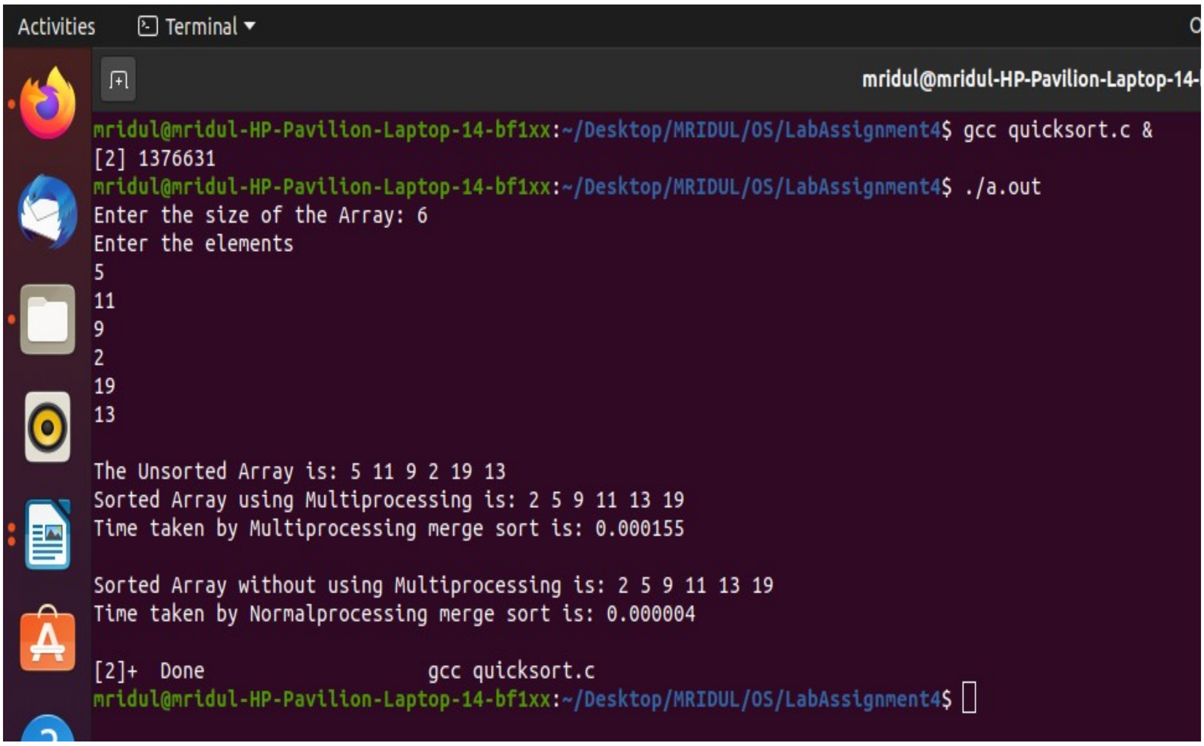
```
	quick_sort(a2, 0, n-1);
	t2 = clock();

	printf("\nSorted Array without using Multiprocessing is: ");
	for(int i = 0; i < n; i = i+1)
	{
		printf("%d ", a2[i]);
	}

	printf("\nTime taken by Normalprocessing merge sort is: %lf\n\n",
(t2 - t1) / (double) CLOCKS_PER_SEC);

	return 0;
}

int partition(int a[], int low, int high)
{
	int pivot = a[high];
	int i = (low - 1);

	for (int j = low; j <= high-1; j = j+1)
	{
	if (a[j] < pivot)
	{
			i = i+1;
			int temp = a[i];
			a[i] = a[j];
			a[j] = temp;
		}
	}
	int temp = a[i+1];
	a[i+1] = a[high];
	a[high] = temp;

	return (i + 1);
}

void quick_sort_parallel(int a[], int low, int high)
{
```

```
	if (low < high)
	{
	int pi = partition(a, low, high);
	pid_t pid;
	pid = vfork();

	if(pid == 0)
	{
			quick_sort(a, low, pi - 1);
			exit(0);
	}
	else
	{
			quick_sort(a, pi + 1, high);
	}
	}
}

void quick_sort(int a[], int low, int high)
{
	if (low < high)
	{
	int pi = partition(a, low, high);
	quick_sort(a, low, pi - 1);
	quick_sort(a, pi + 1, high);
	}
}
```

Explanation :-

In the above code where we normally execute the “partition” and “quicksort” for “left” and “right” partitions, we call the “vfork” for each partition along with the respective partition side sort, whcih in turn leads to parallelization of each partition sort side operation.

<u>Output</u> :-



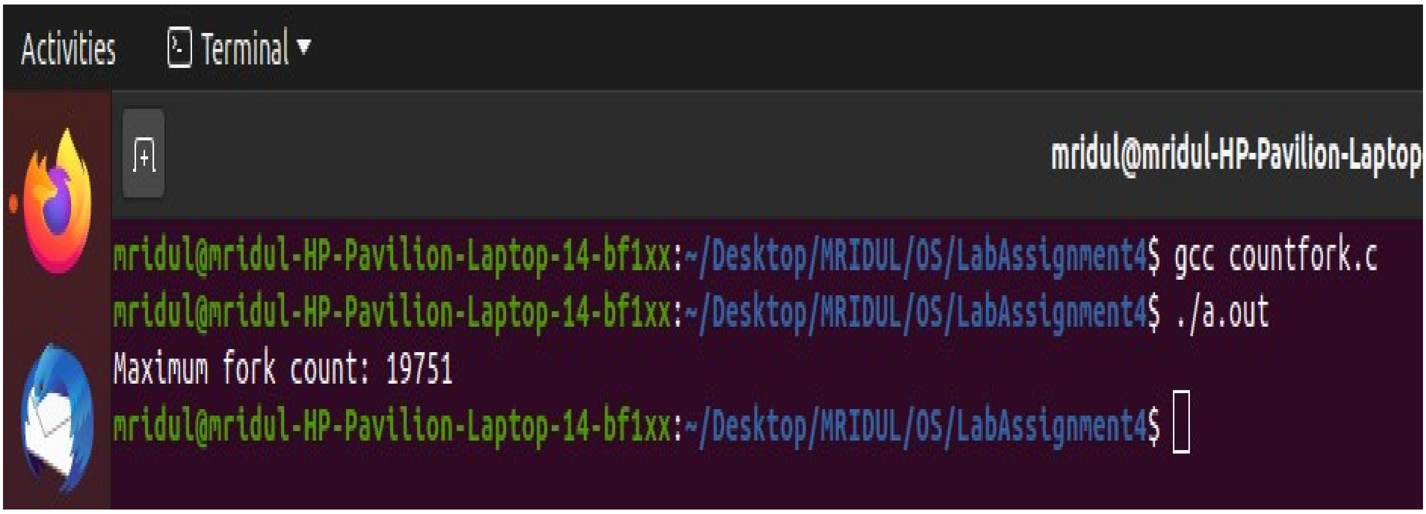
<u>Question 3</u>: Develop a C program to count the maximum number of processes that can be created using fork call.

<u>Code</u> :-

```
#include<stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
        long int count=0;
        int n=900000;

        for(int i = 0; i < n; i = i+1)
        {
        if(fork() == 0)
                exit(1);
        }
```

```
	for(int i = 0; i < n; i = i+1)
	{
	int pid;
	wait(&pid);
	pid /= 255; //the wait catches the child process's exit status 255 times
	count = count + pid;
	}

	printf("Maximum fork count: %ld\n",count);

	return 0;
}
```

Output :-



Explanation :-

In the above code, we are calling fork repeatedly using "for" loop until fork starts to fail and exits the "for" loop. We are keeping a variable "count" to count the number of fork calls after every iteration of the loop and then printing it at the end.

Question 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 shud 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.

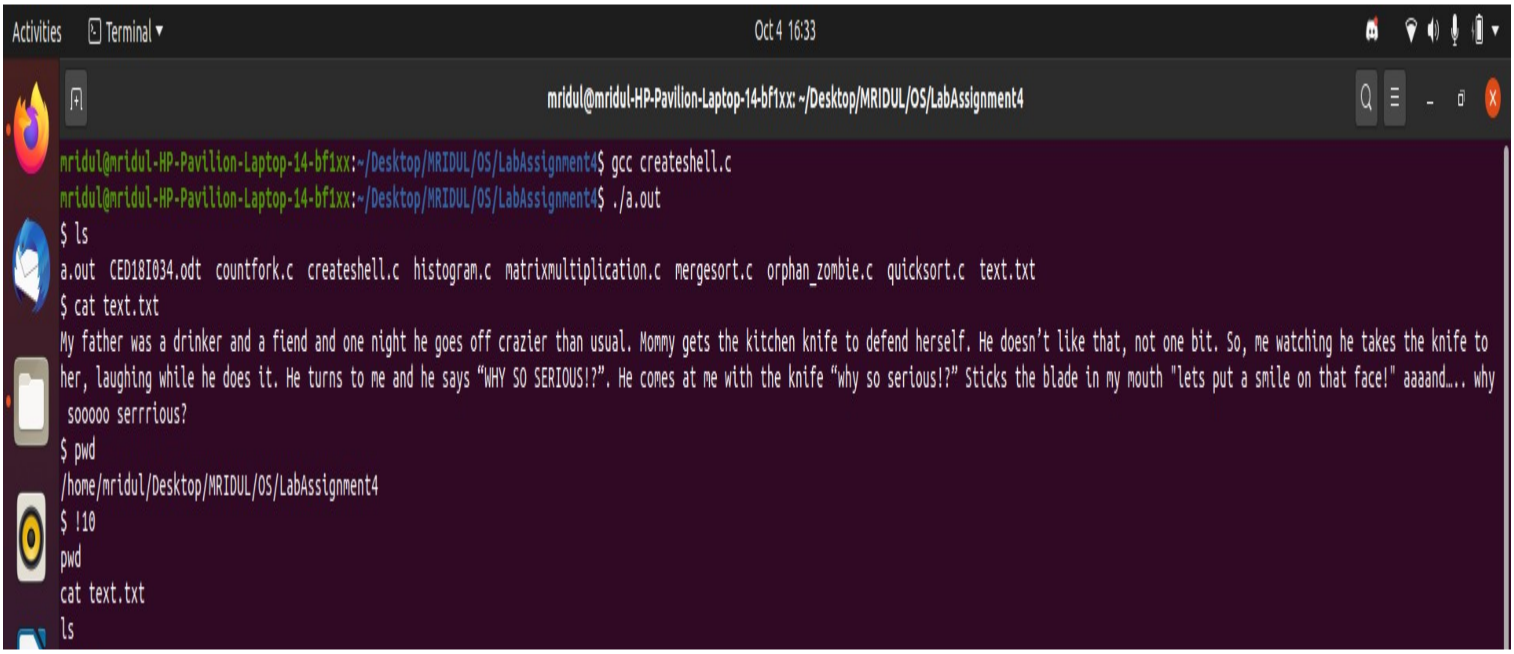
<u>Code</u> :-

```
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<string.h>
#include<stdlib.h>

void history(char* cmd);

int main()
{
	pid_t child;
	char *command[2], *tok, *lineptr = NULL, *line;
	size_t i; size_t n;
	int status;

	while (1)
	{
	printf("$ ");

	if (getline(&lineptr, &n, stdin) == -1)
			break;
	if (strncmp(lineptr, "exit", 4) == 0)
			break;

	history(lineptr);
	command[0] = strtok(lineptr, " \t\n\r");
	command[1] = strtok(NULL, " \t\n\r");

	child = fork();
	if (child == 0)
	{
```

```
			if (strncmp(command[0], "!", 1) == 0)
			{
			command[0][0] = '0';
			int x = atoi(command[0]);
			FILE* fp = fopen(".history", "r");
			getline(&line, &i, fp);
			while(x)
			{
					getline(&line, &i, fp);
					printf("%s", line);
				x = x-1;
			}
			}

			if (strncmp(command[0], "cd", 2) == 0)
			{
			execl("/bin/sh", "cd", (const char *)0);
			}
			if (execlp(command[0], command[0], command[1],
NULL))
			{
			perror("execlp");
			exit(EXIT_FAILURE);
			}
	}

	if (child > 0)
			wait(&status);
	}

	putchar('\n');
	free(lineptr);
	exit(status);
}

void history(char* cmd)
{
   FILE* curr = fopen("1.txt", "w");
```

```
    fputs(cmd, curr);

    fclose(curr);

    system("cp .history 2.txt");
    system("cat 1.txt 2.txt > .history");
    system("rm 1.txt 2.txt");
}
```

<u>Output</u> :-



<u>Question 5</u>: Develop a multiprocessing version of Histogram generator to count the occurrence of various characters in a given text.

<u>Code</u> :-

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/mman.h>

FILE *open_file(char *filename);
```

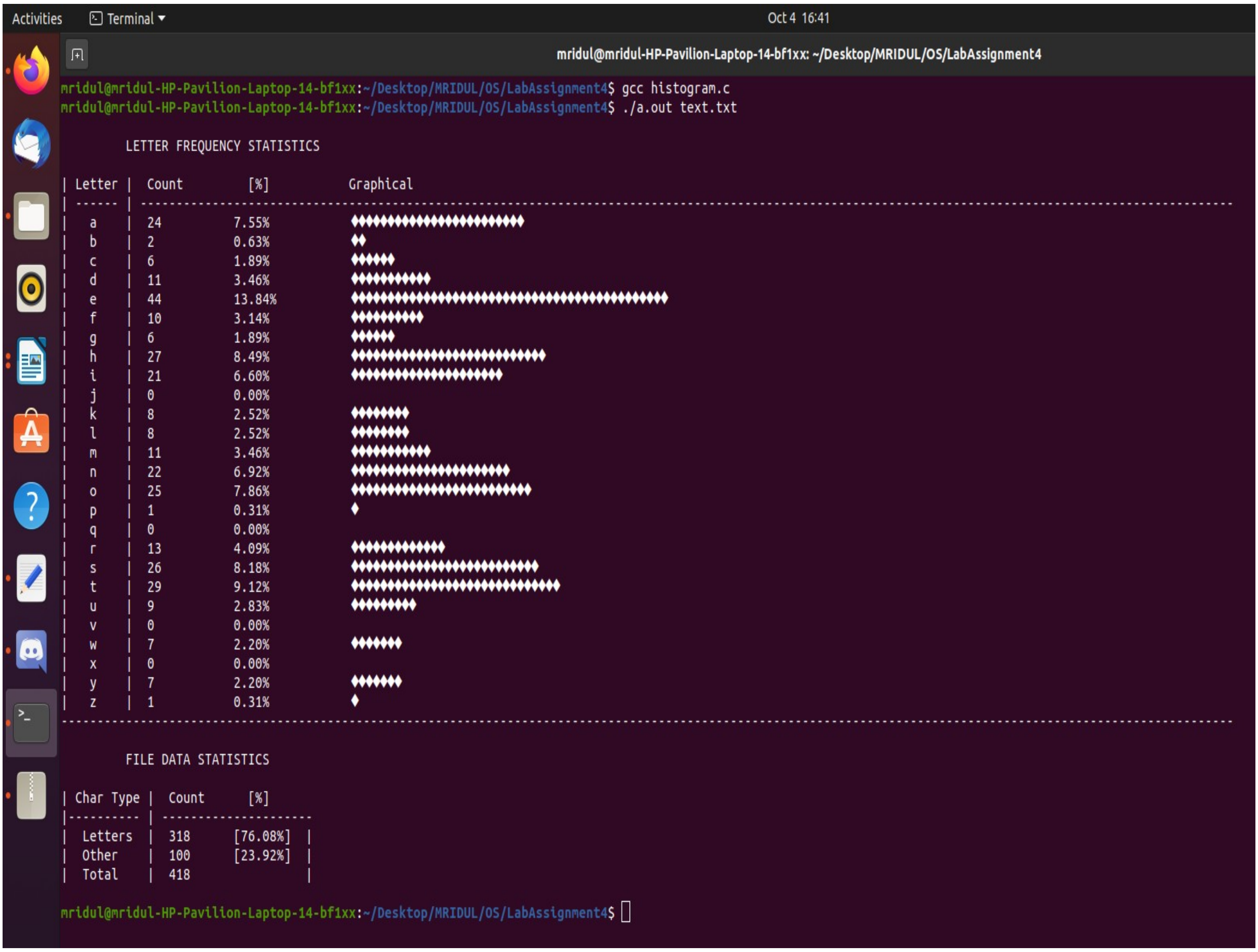
```
void output_results(int *char_count);
int *count_letters(char *filename);

int main(int argc, char *argv[])
{
	if (argc != 2)
	{
	printf("Syntax: %s <filename>\n", argv[0]);
	return 1;
	}

	char *filename = argv[1];
	FILE *file;

	if((file = open_file(filename)) == NULL)
	return 1;

	output_results(count_letters(filename));

	if (fclose(file) != 0)
	{
	printf("Error closing file!\n");
	return 1;
	}

	return 0;
}

FILE *open_file(char *filename)
{
	FILE *file;
	file = fopen(filename, "r");

	if (!file)
	{
	printf("Error opening file!\n");
	return NULL;
	}
```

```
	return file;
}

void output_results(int *char_count)
{
	long numbers_letters = 0;
	long total_characters = 0;

	for (int i = 32; i < 128; i = i+1)
	{
	total_characters = total_characters + char_count[i];
	if (i >= 97 && i <= 122)
	{
			numbers_letters = numbers_letters + char_count[i];
		}
	}

	printf("\n\t LETTER FREQUENCY STATISTICS \n\n");
	printf("| Letter |  Count\t  [%%]\t\tGraphical\n");
	printf("| ------ |
----------------------------------------------------------------------------------------
------------------------------------------------------------\n");

	for (int i = 97; i < 123; i = i+1)
	{
	printf("|   %c    |  %0d ", i, char_count[i]);
	printf("  \t%.2f%%\t\t", ((double)char_count[i] / numbers_letters) *
100);

	for(int j = 0; j < char_count[i]; j = j+1)
	{
			printf("♦");
		}
	printf("\n");
	}
```

```
printf("--------------------------------------------------------------------------------
-----------------------------------------------------------------------------\n");
        printf("\n\t FILE DATA STATISTICS \n\n");
        printf("| Char Type |  Count\t  [%%]\n");
        printf("|---------- | --------------------\n");

        printf("|  Letters  |  %li", numbers_letters);
        printf(" \t[%.2f%%]  |\n", ((double)numbers_letters /
total_characters) * 100);
        printf("|  Other    |  %li", total_characters - numbers_letters);
        printf(" \t[%.2f%%]  |\n", ((double)(total_characters -
numbers_letters) / total_characters) * 100);
        printf("|  Total    |  %li\t\t  |\n\n", total_characters);
}

int *count_letters(char *filename)
{
        int *char_count;
        FILE *file;

        char_count = mmap(NULL, 128 * sizeof(*char_count),
PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

        for(int i = 0; i < 27; i = i+1)
        {
        int c;

        if((file = open_file(filename)) == NULL)
        {
                        printf("Error opening file in child process %d!\n",
getpid());
                        exit(1);
        }

        pid_t pid = fork();

        if(pid == -1)
        {
                        printf("Error forking process!\n");
```

```
                exit(1);
        }
        else if(pid == 0)
        {
                while((c = tolower(fgetc(file))) != EOF)
                {
                if(i == 26 && (c < 97 || c > 122))
                {
                    char_count[c] = char_count[c] + 1;
                }
                else if (c == i + 97)
                {
                    char_count[i + 97] = char_count[i +97] + 1;
                    }
                }

            fclose(file);
            exit(0);
        }
        else
        {
                rewind(file);
        }
        }

        for (int i = 0; i < 27; i = i+1)
        {
        wait(NULL);
        }

        return char_count;
}
```

<u>Output</u> :-



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.

<u>Code</u> :-

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

```
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/wait.h>
#include<time.h>

void print(int a[][3]);
void input(int a[][3]);
void print(int a[][3]);
void multiply(int product[][3], int first[][3], int second[][3], int size);
int component(int pointer1, int pointer2, int first[][3], int second[][3], int
size);

int main()
{
	int first[3][3]; int second[3][3]; int product[3][3];

	printf("Enter the first matrix row by row :-\n");
	for (int i = 0; i < 3; i = i+1)
	{
		for (int j = 0; j < 3; j = j+1)
		{
			scanf("%d", &first[i][j]);
		}
	}

	printf("Enter the second matrix row by row :-\n");
	for (int i = 0; i < 3; i = i+1)
	{
		for (int j = 0; j < 3; j = j+1)
		{
			scanf("%d", &second[i][j]);
		}
	}

	multiply(product, first, second, 3);

	printf("The Product matrix is as below: \n");
	print(product);
```

```
	return 0;
}

void multiply(int product[][3], int first[][3], int second[][3], int size)
{
	for (int i = 0; i < size; i = i+1)
	{
		for (int j = 0; j < size; j = j+1)
		{
			int fd[2];		// this is the file descriptor array
			pipe(fd);
			if (fork() == 0)	//this is the child process
			{
			int prod = component(i, j, first, second, size);

			close(fd[0]); // Writing the Component Product value to the pipe
			write(fd[1], &prod, 10);
			close(fd[1]);
			exit(0);
			}
			else		//this is the parent process
			{
			close(fd[1]); // Reading the Component Product value from the pipe
			read(fd[0], &product[i][j], 10);	//Updating the Product matrix
			close(fd[0]);
			}
		}
	}
}

int component(int pointer1, int pointer2, int first[][3], int second[][3], int
size)
{
	int sum = 0;
	for (int i = 0; i < size; i = i+1)
	{
		sum = sum + (first[pointer1][i] * second[i][pointer2]);
	}
```

```
	return sum;
}

void print(int a[][3])
{
	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < 3; j = j+1)
		{
			printf("%d ", a[i][j]);
		}
		printf("\n");
	}
	printf("\n");
}
```

<u>Output</u> :-

<u>Explanation</u> :-

In the above code each multiplication is parallelized in the most efficient way using vfork() where the data is shared across all the process and the overall output is accumulated and displayed in the end.

<u>Question 7</u>: 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.

<u>Code</u> :-

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define MAX 1024

struct magic_square
{
	int values[MAX][MAX];
};

int coloumn_sum(struct magic_square* matrix, int n, int coloumn);

int row_sum(struct magic_square* matrix, int n, int row);

int diagonal1_sum(struct magic_square* matrix, int n);

int diagonal2_sum(struct magic_square* matrix, int n);

void input_matrix(struct magic_square* matrix, int n);

int check_equivalence(int* a, int n);

void print_array(int* a, int n);
```

```
int main()
{
        printf("Enter the size of the square matrix\n");
        int n;
        scanf("%d", &n);

        int shmid = shmget(IPC_PRIVATE, 2*(n+1)*sizeof(int), 0777 |
IPC_CREAT);

       struct magic_square* magicSq = (struct magic_square*)
malloc(sizeof(struct magic_square));

        input_matrix(magicSq, n);

        for(int i = 0; i < n; i = i+1)
        {
        pid_t pid = fork();

        if(pid == 0)
        {
                int* a = (int*) shmat(shmid, 0, 0);
                a[i] = coloumn_sum(magicSq, n, i);

                shmdt(a);
             exit(0);
        }
        }


        for(int i = 0; i < n; i = i+1)
        {
        pid_t pid = fork();

        if(pid == 0)
        {
                int* a = (int*) shmat(shmid, 0, 0);
             a[i+n] = row_sum(magicSq, n, i);

                shmdt(a);
```

```
            exit(0);
    }
    }

    pid_t pid_1 = fork();
    if(pid_1 == 0)
    {
         int* a = (int*) shmat(shmid, 0, 0);
         a[2*n] = diagonal1_sum(magicSq, n);

         shmdt(a);
         exit(0);
    }

   pid_t pid_2 = fork();
    if(pid_2 == 0)
    {
        int* a = (int*) shmat(shmid, 0, 0);
        a[2*n+1] = diagonal2_sum(magicSq, n);

        shmdt(a);
    exit(0);
    }


    int status;
    wait(NULL);
    waitpid(pid_2, &status, 0);

    int* a = (int*) shmat(shmid, 0, 0);

   if(check_equivalence(a, 2*(n+1)) )
    printf("\nIt is a magic square");
    else
    printf("\nIt's not a magic square");

    shmdt(a);
    shmctl(shmid, IPC_RMID, NULL);
```

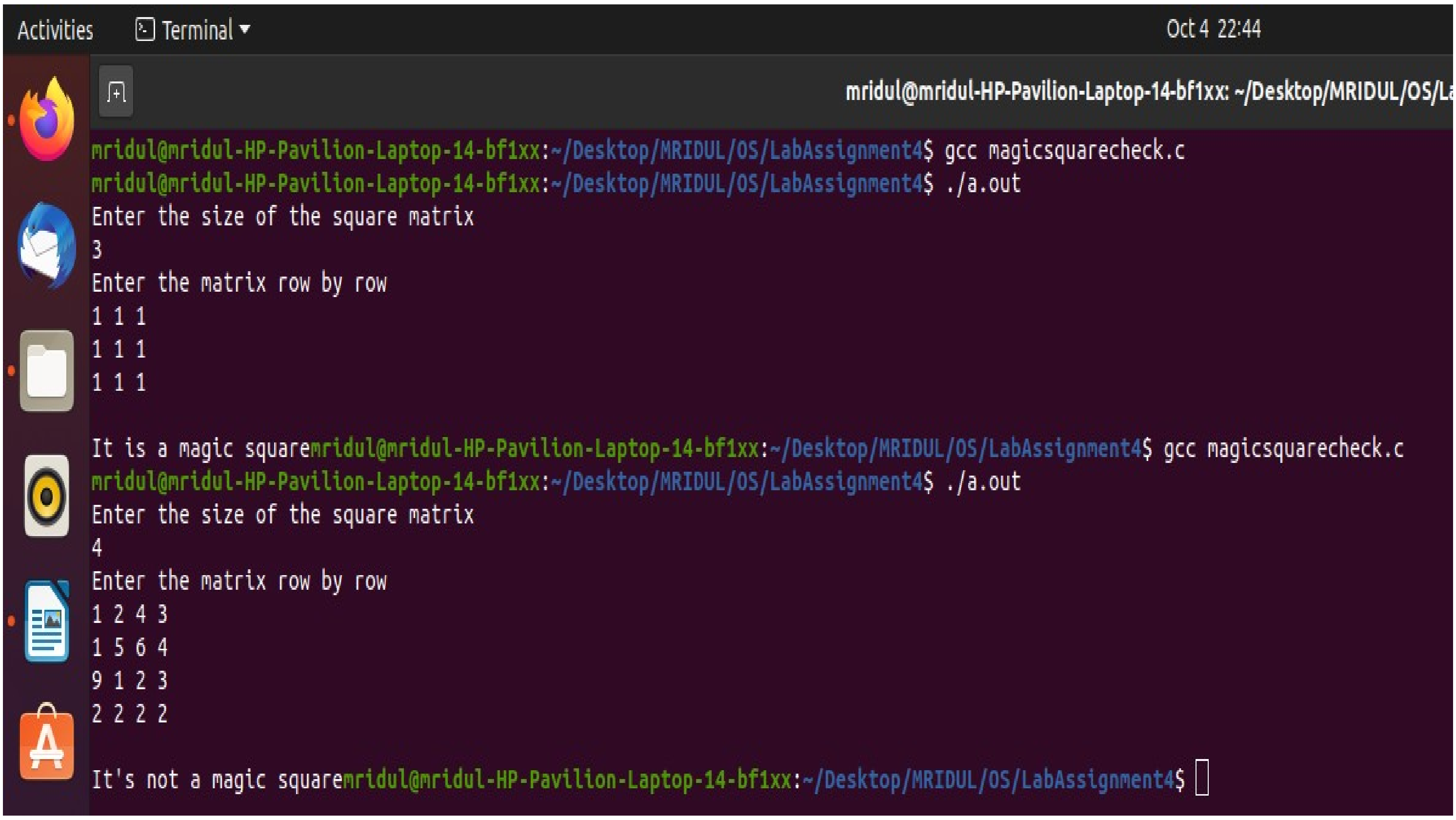
```
    return 0;
}

int coloumn_sum(struct magic_square* matrix, int n, int coloumn)
{
    int sum = 0;

    for(int i = 0; i < n; i = i+1)
    sum = sum + matrix->values[i][coloumn];

    return sum;
}

int row_sum(struct magic_square* matrix, int n, int row)
{
    int sum = 0;

    for(int i = 0; i < n; i = i+1)
    sum = sum + matrix->values[row][i];

    return sum;
}

int diagonal1_sum(struct magic_square* matrix, int n)
{
    int sum = 0;

    for(int i = 0; i < n; i = i+1)
    sum = sum + matrix->values[i][i];

    return sum;
}

int diagonal2_sum(struct magic_square* matrix, int n)
{
    int sum = 0;

    for(int i = 0; i < n; i = i+1)
    sum = sum + matrix->values[i][n - 1 - i];
```

```
	return sum;
}

void input_matrix(struct magic_square* matrix, int n)
{
	printf("Enter the matrix row by row\n");
	for(int i = 0; i < n; i = i+1)
		for(int j = 0; j < n; j = j+1)
			scanf("%d", &(matrix->values[i][j]));

}

int check_equivalence(int* a, int n)
{
	for(int i = 0; i < n-1; i = i+1)
	if(a[i] != a[i+1])
			return 0;

	return 1;
}

void print_array(int *a, int n)
{
	for(int i = 0; i < n; i = i+1)
	printf(" %d", a[i]);
	printf("\n");
}
```

<u>Explanation</u> :-

Magic squares are generally classified according to their order n as: odd if n is odd evenly even (also referred to as "doubly even") if n = 4k (e.g. 4, 8, 12, and so on) oddly even (also known as "singly even") if n = 4k + 2 (e.g. 6, 10, 14, and so on). This classification is based on different techniques required to construct odd, evenly even, and oddly even squares. In the above code, the coloumn, row and diagonal sums are calculated using seperate functions called through “fork” and exit(0) in if-else to share memory accordingly and finally check if the matrix is a magic matrix.

Output :-



Question 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 version...)

Code :-

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

#define MAX 100000
int top =- 1;

void push(int a[], int n);
void magic_square(int size, int a[][size]);
void odd_order_magic_square(int size, int a[][size]);
void doubly_even_magic_square(int size, int a[][size]);
```

```
void singly_even_magic_square(int size, int a[][size]);
void display_magic_square(int size, int a[][size]);
int magic_square_check(int size, int a[][size]);

int main()
{
	int size;
	printf("Enter order of square matrix: ");
	scanf("%d", &size);

	int a[size][size];

	pid_t pid;
	pid = vfork();
	if(pid == 0)
	{
	if(size < 3)
	{
			printf("Error: Order of matrix must be greater than 2\n");
			exit(EXIT_FAILURE);
	}

	magic_square(size, a);
	exit(0);
	}
	else
	{
	wait(NULL);
	display_magic_square(size, a);
	int valid = magic_square_check(size, a);

	if(valid == 1)
			printf("\nIt is a valid Magic Square\n\n");
	else
			printf("It is not a valid Magic Square\n\n");
	}

	return 0;
}
```

```
void push(int a[], int n)
{
	if(top==MAX-1)
	{
		printf("\nStack is full!!");
	}
	else
	{
		top = top+1;
		a[top] = n;
	}
}

void magic_square(int size, int a[][size])
{
	if(size % 2 == 1)
	odd_order_magic_square(size, a);
	else if(size % 4 == 0)
	doubly_even_magic_square(size, a);
	else
	singly_even_magic_square(size, a);
}

void odd_order_magic_square(int size, int a[][size])
{
	int square = size * size;
	int i = 0; int j = size/2; int k;

	for(k = 1; k <= square; ++k)
	{
	a[i][j] = k;
	i = i-1;
	j = j+1;

	if(k % size == 0)
	{
			i = i+2;
			--j;
```

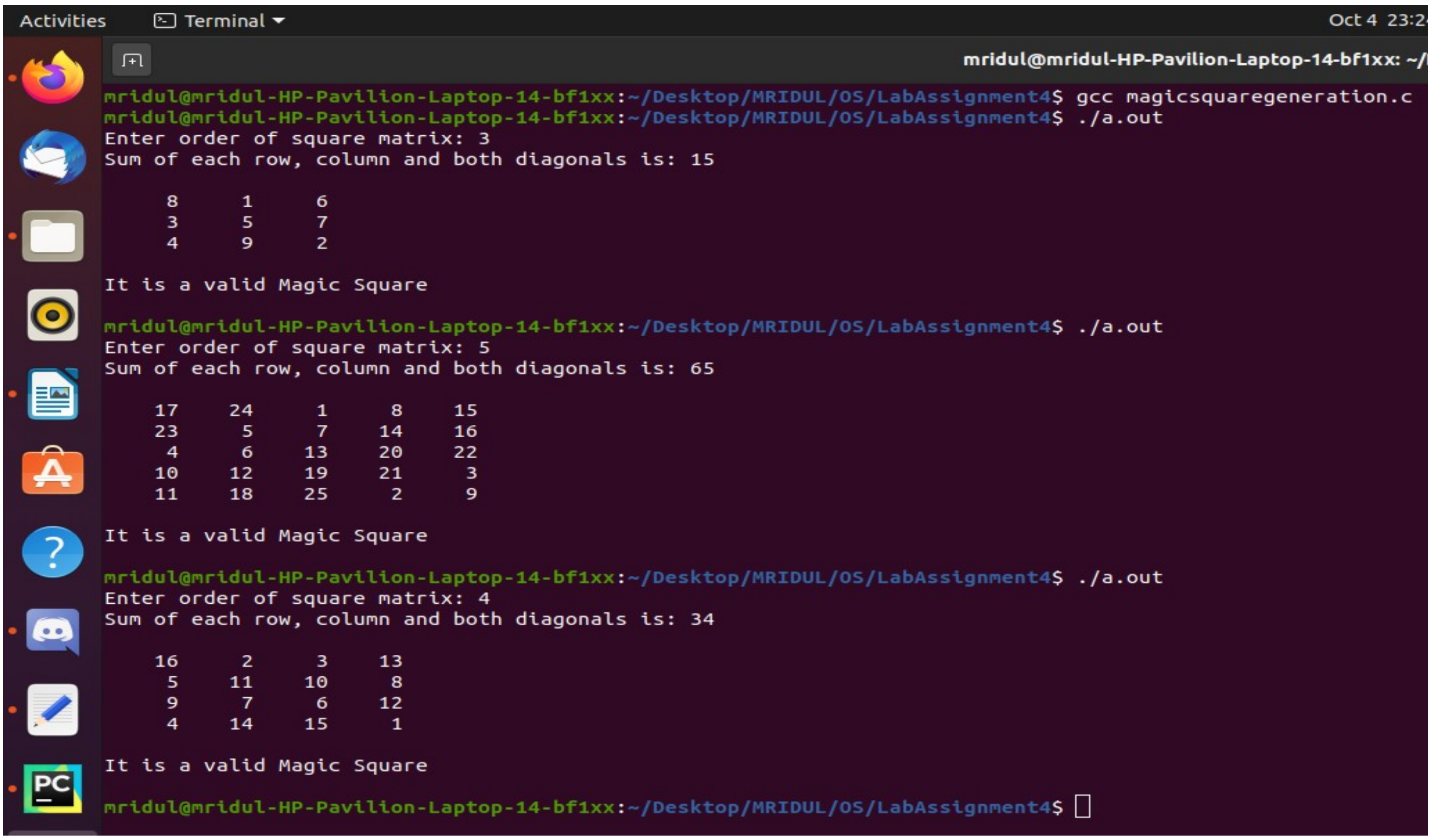
```
        }
        else
        {
                    if(j == size)
            j = j - size;
            else if(i < 0)
            i = i + size;
        }
        }
}

void doubly_even_magic_square(int size, int a[][size])
{
        int I[size][size];
        int J[size][size];

        int i; int j;

        int index=1;
        for(i = 0; i < size; i = i+1)
        for(j=  0; j < size; j = j+1)
        {
                    I[i][j] = ((i+1)%4)/2;
                    J[j][i] = ((i+1)%4)/2;
                    a[i][j] = index;
                    index = index+1;
        }

        for(i = 0; i < size; i = i+1)
        for(j = 0; j < size; j = j+1)
        {
                    if(I[i][j] == J[i][j])
            a[i][j] = size*size+1 - a[i][j];
        }
}

void singly_even_magic_square(int size, int a[][size])
{
        int N = size;
```

```
int halfN = N/2;
int k = (N-2)/4;

int temp;
int new[N];

int swap_coloumn[N];
int index=0;
int mini_magic[halfN][halfN];

odd_order_magic_square(halfN, mini_magic);

for(int i = 0; i < halfN; i = i+1)
        for (int j = 0; j < halfN; j = j+1)
{
        a[i][j] = mini_magic[i][j];
                a[i+halfN][j+halfN] = mini_magic[i][j]+halfN*halfN;
                a[i][j+halfN] = mini_magic[i][j]+2*halfN*halfN;
                a[i+halfN][j] = mini_magic[i][j]+3*halfN*halfN;
        }

for(int i = 1; i <= k; i = i+1)
swap_coloumn[index++] = i;

for (int i = N-k+2; i <= N; i = i+1)
swap_coloumn[index++] = i;



for(int i = 1; i <= halfN; i = i+1)
        for(int j = 1; j <= index; j = j+1)
{
                temp = a[i-1][swap_coloumn[j-1]-1];
                a[i-1][swap_coloumn[j-1]-1] = a[i+halfN-1]
[swap_coloumn[j-1]-1];
                a[i+halfN-1][swap_coloumn[j-1]-1] = temp;
}

temp = a[k][0];
a[k][0] = a[k+halfN][0];
```

```
        a[k+halfN][0] = temp;

        temp = a[k+halfN][k];
        a[k+halfN][k] = a[k][k];
        a[k][k] = temp;
}

void display_magic_square(int size, int a[][size])
{
        printf("Sum of each row, column and both diagonals is: %d\n\n",
size*(size*size + 1) / 2);
        for(int i = 0; i < size; i = i+1)
        {
        for(int j = 0; j < size; j = j+1)
        {
                printf(" %5d", a[i][j]);
        }
       printf("\n");
        }
}

int magic_square_check(int size, int a[][size])
{
        int i; int sum1 = 0; int sum2 = 0;

        for(i = 0; i < size; i = i+1)
        sum1 = sum1 + a[i][i];

       for(i = 0; i < size; i = i+1)
        sum2 = sum2 + a[i][size-1-i];

        if(sum1 != sum2)
        return 0;

        for(i = 0; i < size; i = i+1)
        {
        int row_sum = 0;
        for(int j = 0; j < size; j = j+1)
            row_sum = row_sum + a[i][j];
```

```
            if(row_sum != sum1)
                    return 0;
        }

        for(i = 0; i < size; i = i+1)
        {
        int coloumn_sum = 0;

        for(int j = 0; j < size; j = j+1)
                    coloumn_sum = coloumn_sum + a[j][i];

        if(sum1 != coloumn_sum)
        return 0;
        }

        return 1;
}
```

<u>Output</u> :-



Explanation :-

Magic squares are generally classified according to their order n as: odd if n is odd evenly even (also referred to as "doubly even") if n = 4k (e.g. 4, 8, 12, and so on) oddly even (also known as "singly even") if n = 4k + 2 (e.g. 6, 10, 14, and so on). This classification is based on different techniques required to construct odd, evenly even, and oddly even squares. In the above code, there are important functions such as generation of magic squares and checking if this matrix is a magic square. These two here are parallelized using “vfork” where the buffer is shared between the processes leading to well balanced calculations in turn bringing up the magic square.