OPERATING SYSTEMS PRACTICE (COM301P)

Name: Vinayak Sethi **Roll No:** COE18B061

Assignment 4

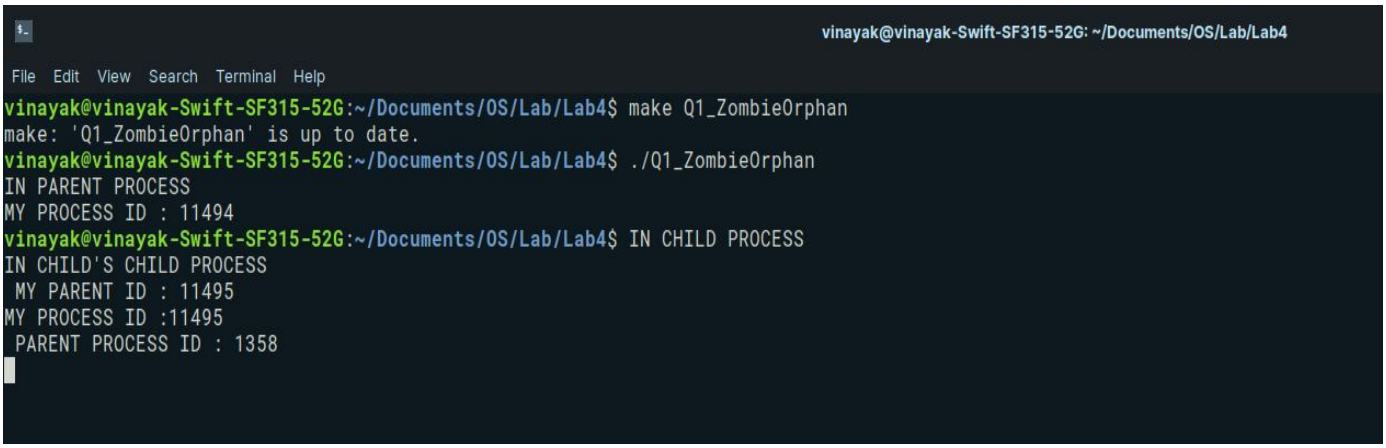
(1) Test drive a C program that creates **Orphan and Zombie Processes**

Filename: Q1_ZombieOrphan.c

```c
// C program to execute zombie and orphan process in a single program
#include<stdio.h>
#include<unistd.h>

int main()
{
   pid_t pid;
   pid = fork();

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

   else if (pid == 0) //child block
   {
       sleep(5);
       pid = fork();

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

    return 0;
}
```

Output:



Explanation:

In the following 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.

(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]

Filename: Q2_MergeSort.c

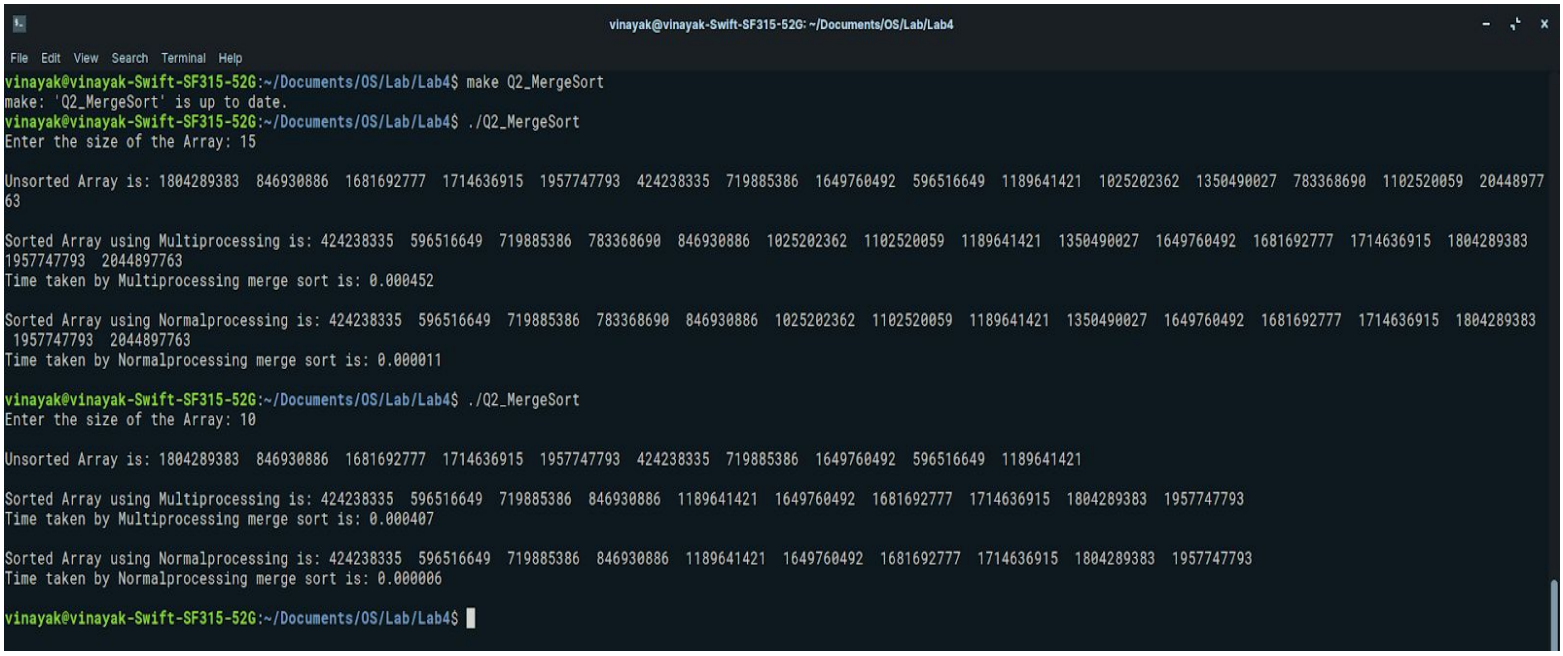
```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<unistd.h>
#include<time.h>

void merge(int arr[], int low, int mid, int high);
void MergeSortParallel(int arr[], int low, int high);
void MergeSort(int arr[], int low, int high);
void print(int arr[], int n);

int main()
{
   int i, size;
   clock_t t1, t2;
   printf("Enter the size of the Array: ");
   scanf("%d", &size);
   int arr1[size], arr2[size];

   for(i=0; i<size; i++)
   {
       int x = rand();  //Filling the random numbers
       arr1[i] = arr2[i] = x;
   }

   printf("\nUnsorted Array is: ");
```

```
   print(arr1, size);
   printf("\n");
   t1 = clock();
   MergeSortParallel(arr1, 0, size-1);
   t2 = clock();
   printf("\nSorted Array using Multiprocessing is: ");
   print(arr1, size);
   printf("\nTime taken by Multiprocessing merge sort is: %lf\n", (t2 -
t1) / (double) CLOCKS_PER_SEC);

   t1 = clock();
   MergeSort(arr2, 0, size-1);
   t2 = clock();
   printf("\nSorted Array using Normalprocessing is: ");
   print(arr2, size);
   printf("\nTime taken by Normalprocessing merge sort is: %lf\n\n", (t2 -
t1) / (double) CLOCKS_PER_SEC);
   return 0;
}

void merge(int arr[], int low, int mid, int high)
{
   int i, j, k;
   int n1 = mid - low + 1;
   int n2 = high - mid;
    /* create temp arrays */
   int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
       L[i] = arr[low + i];
   for (j = 0; j < n2; j++)
       R[j] = arr[mid + 1 + j];
    /* Merge the temp arrays back into arr[low..high]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = low; // Initial index of merged subarray
   while (i < n1 && j < n2)
```

```
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
     /* Copy the remaining elements of L[], if there are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
     /* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void MergeSortParallel(int arr[], int low, int high)
{
    if(low < high)
    {
        int mid = low + (high - low) / 2;
        pid_t pid;
        pid = vfork();
        if(pid == 0)
```

```
        {
            MergeSortParallel(arr, low, mid);
            exit(0);
        }
        else
        {
            MergeSortParallel(arr, mid + 1, high);
            merge(arr, low, mid, high);
        }
    }
}

void MergeSort(int arr[], int low, int high)
{
    if(low < high)
    {
        int mid = low + (high - low) / 2;
        MergeSort(arr, low, mid);
        MergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

void print(int arr[], int n)
{
    for(int i=0; i<n; i++)
    {
        printf("%d  ",arr[i]);
    }
}
```

Output:



Explanation:

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.

When we time the code and compare its performance with the traditional sequential code. We would be surprised to know that sequential sort performance better!
When, say left child, access the left array, the array is loaded into the cache of a processor. Now when the right array is accessed (because of concurrent accesses), there is a cache miss since the cache is filled with the left segment and then the right segment is copied to the cache memory. This to-and-fro process continues and it degrades the performance to such a level that it performs poorer than the sequential code.

Filename: Q2_QuickSort.c

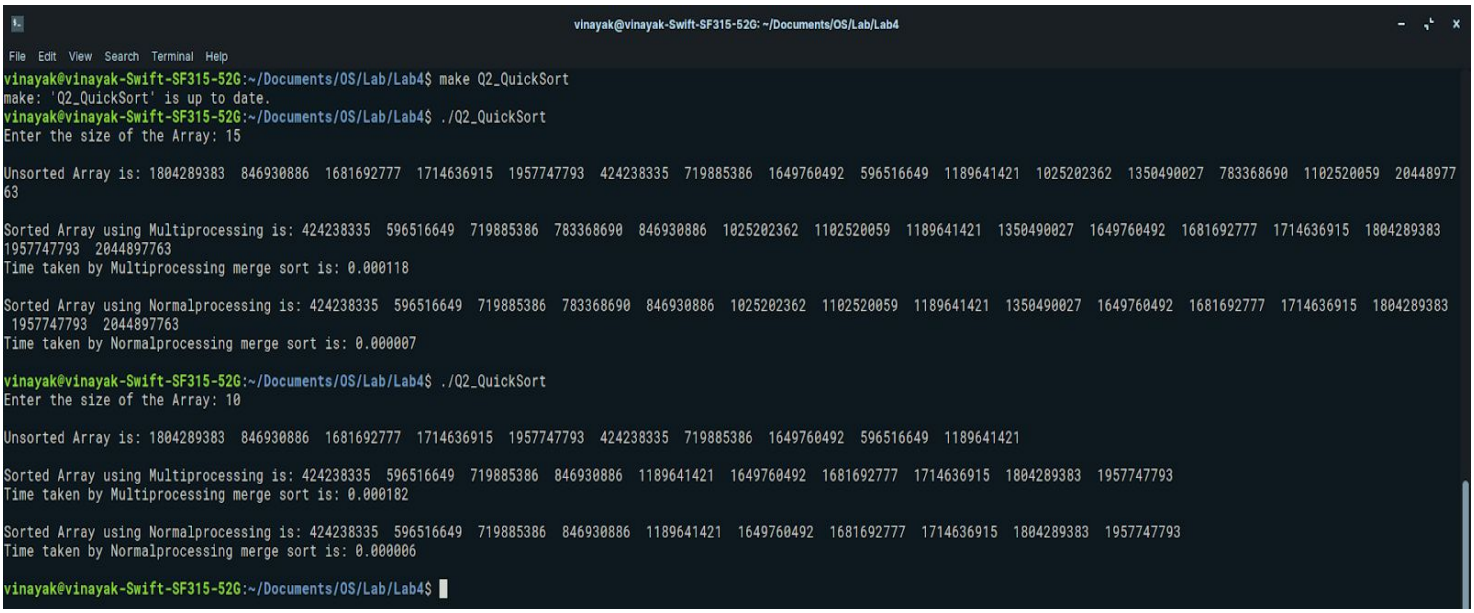
```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<unistd.h>
#include<time.h>

int partition(int arr[], int low, int high);
void QuickSortParallel(int arr[], int low, int high);
void QuickSort(int arr[], int low, int high);
void swap(int *a, int *b);
void print(int arr[], int n);

int main()
{
   int i, size;
   clock_t t1, t2;
   printf("Enter the size of the Array: ");
   scanf("%d", &size);
   int arr1[size], arr2[size];

   for(i=0; i<size; i++)
   {
       int x = rand();  //Filling the random numbers
       arr1[i] = arr2[i] = x;
   }

   printf("\nUnsorted Array is: ");
   print(arr1, size);
   printf("\n");
   t1 = clock();
   QuickSortParallel(arr1, 0, size-1);
   t2 = clock();
   printf("\nSorted Array using Multiprocessing is: ");
   print(arr1, size);
```

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

   t1 = clock();
   QuickSort(arr2, 0, size-1);
   t2 = clock();
   printf("\nSorted Array using Normalprocessing is: ");
   print(arr2, size);
   printf("\nTime taken by Normalprocessing merge sort is: %lf\n\n", (t2 -
t1) / (double) CLOCKS_PER_SEC);
   return 0;
}

int partition(int arr[], int low, int high)
{
   int pivot = arr[high]; // pivot
   int i = (low - 1); // Index of smaller element
    for (int j = low; j <= high- 1; j++)
   {
       // If current element is smaller than the pivot
       if (arr[j] < pivot)
       {
           i++; // increment index of smaller element
           swap(&arr[i], &arr[j]);
       }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}

void QuickSortParallel(int arr[], int low, int high)
{
   if (low < high)
   {
       /* pi is partitioning index, arr[p] is now
          at right place */
       int pi = partition(arr, low, high);
```

```
        pid_t pid;
        pid = vfork();
        // Separately sort elements before partition and after partition
        if(pid == 0)
        {
            QuickSort(arr, low, pi - 1);
            exit(0);
        }
        else
        {
            QuickSort(arr, pi + 1, high);
        }
    }
}

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

void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void print(int arr[], int n)
{
    for(int i=0; i<n; i++)
    {
        printf("%d  ",arr[i]);
```

```
    }
}
```

Output:



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, which in turn leads to parallelization of each partition sort side operation.

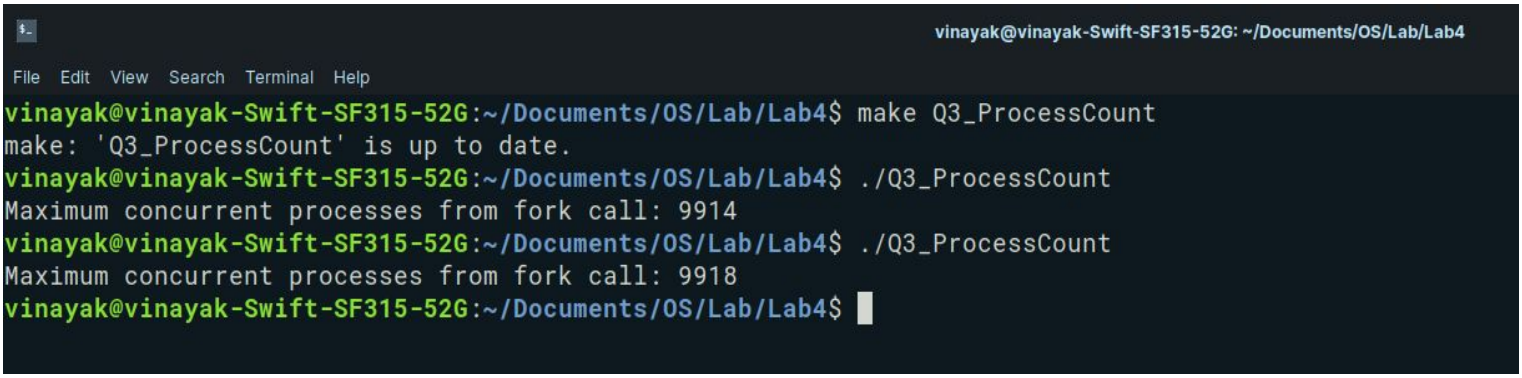
And the argument for time and performance of QuickSort in multiprocessing and normal processing is similar to MergeSort argument.

(3) Develop a C program to count the maximum number of processes that can be created using fork call.

Filename: Q3_ProcessCount.c

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

int main()
{
   long int count = 0;
   int n = 999999;

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

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

   printf("Maximum concurrent processes from fork call: %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.

(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.

Filename: Q4_CommandShell.c

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

```c
#include<sys/wait.h>

struct string_linked_list
{
   char *cmd;
   struct string_linked_list *prev;
};

void run_command(char *cmd, char *cwd, char *prev_wd, char *home_dir,
struct string_linked_list *commands_run, int no_already_run);
void extract_args(char *cmd, char *args[],int argc);
void change_directory(char *args[], char *cwd, char *prev_wd, char
*home_dir);
void history(struct string_linked_list *list, char *args[],int max);
void substring(char s[], char sub[], int p, int l);

int main()
{
   char
cmd[100],cwd[100],prev_wd[100],printing_cwd[100],*home_dir,after_home[100]
,test[100];
   int N=0,home_dir_size,cmd_size;
   struct string_linked_list *tail=NULL,*temp;

   home_dir=getenv("HOME");
   home_dir_size=strlen(home_dir);
   // run_command("clear\n","","",home_dir,tail,N);

   printf("\033[1;33m================================ MY COMMAND SHELL
================================\n");
   while(1)
   {
       getcwd(cwd,100);
       substring(cwd,test,0,home_dir_size);
       if(strcmp(test,home_dir)==0)
       {
           printing_cwd[0]='~';
```

```
            printing_cwd[1]='\0';
            substring(cwd,after_home,home_dir_size,87);
            strcat(printing_cwd,after_home);
        }
        else
            strcpy(printing_cwd,cwd);
        printf("\033[1;32mmy-command-shell\033[0m:\033[1;34m%s\033[0m@
",printing_cwd);
        fgets(cmd,100,stdin);
        N++;
        cmd_size=strlen(cmd)-1;
        temp=malloc(sizeof(struct string_linked_list));
        temp->cmd=malloc(cmd_size);
        strncpy(temp->cmd,cmd,cmd_size);
        temp->prev=tail;
        tail=temp;
        run_command(cmd,cwd,prev_wd,home_dir,tail,N);
    }
    return 0;
}

void run_command(char *cmd,char *cwd,char *prev_wd,char *home_dir,struct
string_linked_list *commands_run,int no_already_run)
{
    int argc=1;
    for(char *c=cmd;*c!='\n';c++)
        if(*c==' ')
            argc++;

    char **args = malloc(sizeof (char *)*(argc+1));
    extract_args(cmd,args,argc);

    if(args[0][0]=='!')
    {
        history(commands_run,args,no_already_run);
        return;
    }
```

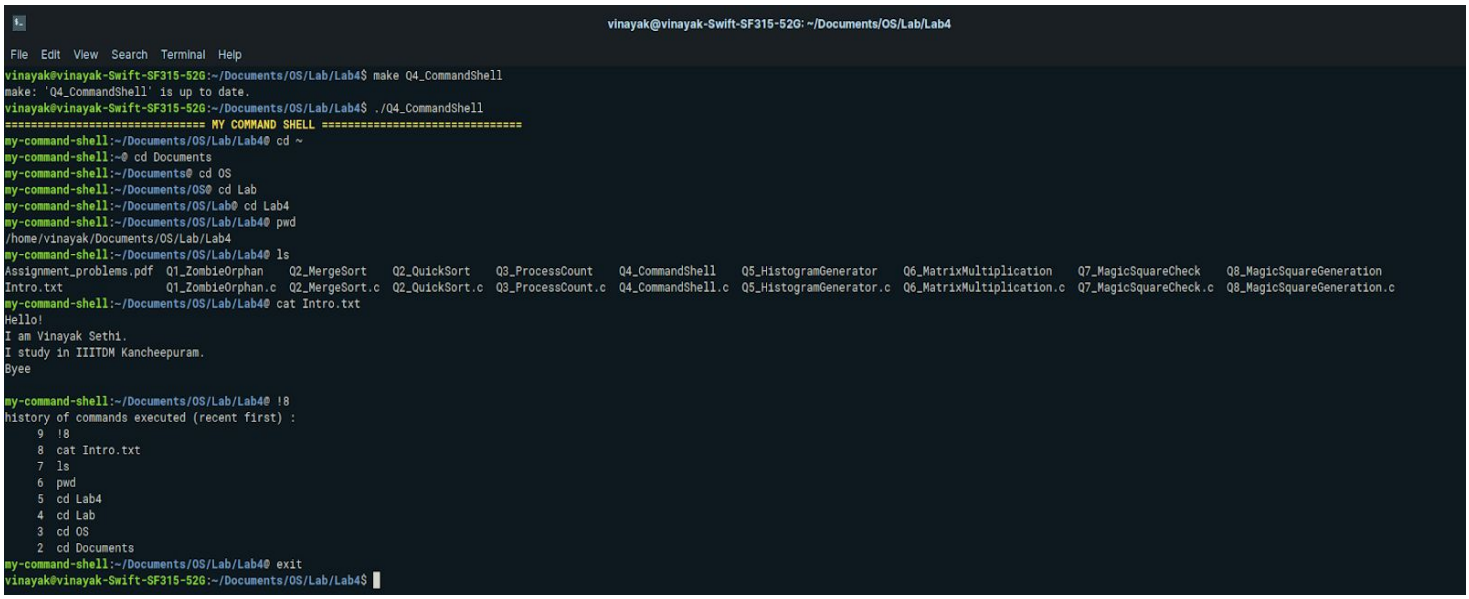
```
    else if(strcmp(args[0],"cd")==0)
    {
        change_directory(args,cwd,prev_wd,home_dir);
        return;
    }
    else if(strcmp(args[0],"exit")==0)
        exit(0);

    pid_t pid;
    pid=fork();
    if(pid==-1)
    {
        printf("myshell: fork failed. could not execute '%s'",cmd);
        exit(1);
    }
    else if(pid>0) // parent block
        wait(NULL);
    else if(pid==0) // child block
    {
        execvp(args[0],args);
        // to terminate the extra process if exec call fails
        printf("myshell: %s: could not execute command\n",args[0]);
        exit(1);
    }
}

void extract_args(char *cmd, char *args[],int argc)
{
    int i;
    char *c,*p;
    for(i=0;i<=argc;i++)
        args[i]=malloc(100);
    i=0;
    p=&args[0][0];
    for(c=cmd;*c!='\n';c++)
    {
        if(*c!=' ')
```

```
        {
            *p=*c;
            p++;
        }
        else
        {
            *p='\0';
            i++;
            p=&args[i][0];
        }
    }
    *p='\0';
    i++;
    args[i]=NULL;
    return;
}

void change_directory(char *args[], char *cwd, char *prev_wd, char
*home_dir)
{
    if(args[1]==NULL||args[1][0]=='~')
    {
        strcpy(args[0],home_dir);
        if(args[1]!=NULL)
            strcat(args[0],&args[1][1]);
        chdir(args[0]);
        strcpy(prev_wd,cwd);
    }
    else if(strcmp(args[1],"-")==0 && args[2]==NULL)
    {
        if(prev_wd[0]=='\0')
        {
            printf("myshell: cd: prev_wd not set\n");
            return;
        }
        printf("%s\n",prev_wd);
        chdir(prev_wd);
```

```
        strcpy(prev_wd,cwd);
    }
    else if(args[2]==NULL)
    {
        int x=chdir(args[1]);
        if(x==-1)
            printf("myshell: cd: %s: No such file or directory\n",args[1]);
        else
            strcpy(prev_wd,cwd);
    }
    else
        printf("myshell: cd: too many arguments\n");
    return;
}

void history(struct string_linked_list *list, char *args[],int max)
{
    struct string_linked_list *temp;
    int x=atoi(&args[0][1]);
    if(x==0||args[1]!=NULL)
        printf("history: incorrect usage\nTry '!n' to view the last n
commands\n");
    else if(max==1)
        printf("YoU JuSt OpeNeD
\033[1;30mm\033[1;31my\033[1;32ms\033[1;33mh\033[1;34me\033[1;35ml\033[1;3
6ml\033[0m wHaT dO yOu ExPeCt To seE iN thE HisToRy?\n");
    else if(x>max)
        printf("history: you have executed only %d commands so far
including '%s'\n",max,list->cmd);
    else
    {
        printf("history of commands executed (recent first) :\n");
        temp=list;
        for(int i=0;i<x;i++)
        {
            printf("%6d  %s\n",max-i,temp->cmd);
            temp=temp->prev;
```

```
        }
    }
}

void substring(char s[], char sub[], int p, int l)
{
    int c=p;
    while(c<l)
    {
        sub[c-p]=s[c];
        c++;
    }
    sub[c]='\0';
}
```

Output:



Explanation:

A linked list is maintained to keep a track of all the commands which are entered. Whenever we need to show the history of the commands executed till now. We print the linked list in reverse order till the number mentioned with '!'.

(5) Develop a multiprocessing version of Histogram generator to count the occurrence of various characters in a given text.

Filename: Q5_HistogramGenerator.c

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

FILE * openFile(char * filename) // open a file in read mode and return
the pointer
{
   FILE * file;
   file = fopen(filename, "r");

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

   return file;
}

void outputResults(int * charCount)
{
   long numbers_letters = 0;
   long total_characters = 0;
```

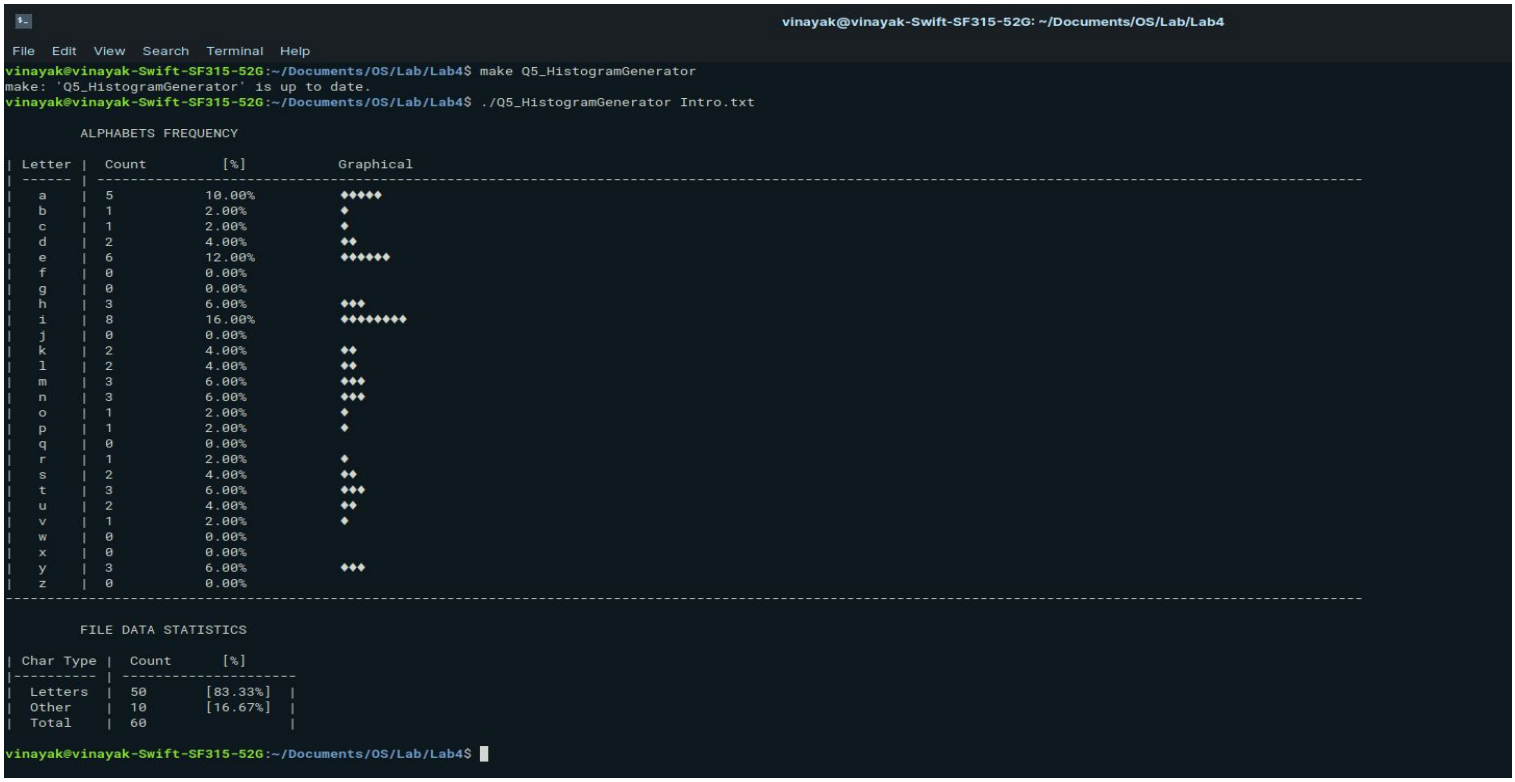
```
   for(int i = 32; i < 128; i = i+1)
   {
       total_characters = total_characters + charCount[i];
       if(i >= 97 && i <= 122)
       {
           numbers_letters = numbers_letters + charCount[i];
       }
   }

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

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

       for(int j = 0; j < charCount[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 * countLetters(char * filename)
{
   int * charCount;
   FILE * file;

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

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

       if((file = openFile(filename)) == NULL)
       {
           printf("Error opening file. %s\n", filename);
           exit(1);
       }

       pid_t pid = fork();

       if (pid == -1)
       {
           exit(1);
       }
       else if (pid == 0)
       {
           while((c = tolower(fgetc(file))) != EOF)
```

```
            {
                if(i == 26 && (c < 97 || c > 122))
                    charCount[c]++; // Count other char
                else if (c == i + 97)
                    charCount[i + 97] += 1; // Count letters
            }

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

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

    return charCount;
}

int main(int argc, char * argv[]) //command line arguments
{
    if(argc != 2)
    {
        printf("Syntax: ./a.out <filename>\n");
        exit(EXIT_FAILURE);
    }

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

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

    outputResults(countLetters(filename));

    if (fclose(file) != 0)
```

```
    {
        printf("Error closing file!\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Output:



Explanation:

For parallelization of processes we have used fork() call, which helps us to do multiple operations in child and parent block.

The child block stores the frequency of each character, while the parent block handles errors like while reading the file, if any error is encountered it

will be handled by the parent block using the rewind function. The rewind function sets the file position to the beginning of the file.

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

Filename: Q6_MatrixMultiplication.c

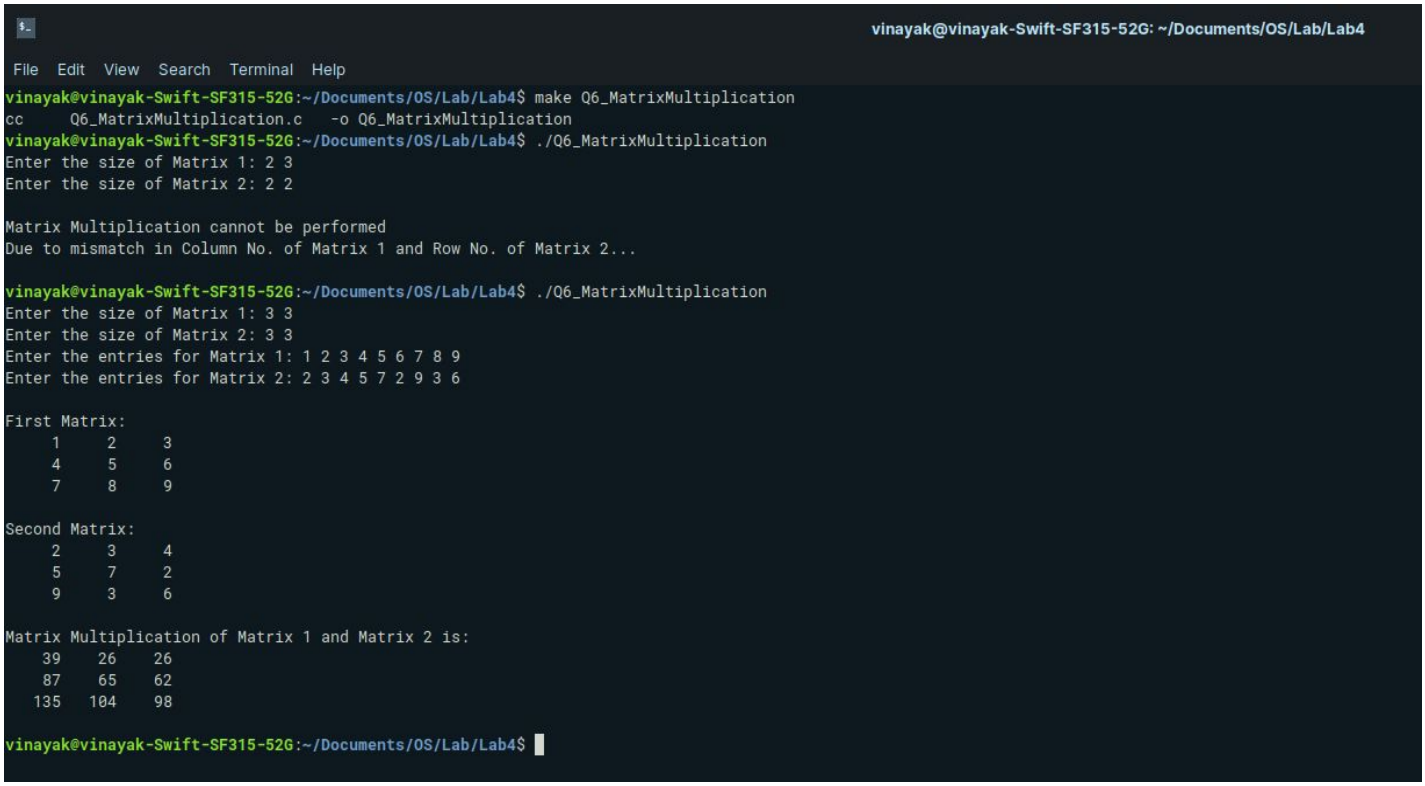
```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int nr1, nr2, nc1, nc2;

void get_input(int row, int column, int arr[][column]);
void displayMatrix(int row, int column, int arr[][column]);
int MatMul(int i, int j, int a[][nc1], int b[][nc2]);

int main()
{
   int status;
   printf("Enter the size of Matrix 1: ");
   scanf("%d %d",&nr1, &nc1);
   printf("Enter the size of Matrix 2: ");
   scanf("%d %d",&nr2, &nc2);

   int a[nr1][nc1];
   int b[nr2][nc2];
```

```
    if(nr2 != nc1)
    {
        printf("\nMatrix Multiplication cannot be performed\nDue to
mismatch in Column No. of Matrix 1 and Row No. of Matrix 2...\n\n");
        exit(EXIT_FAILURE);
    }

    printf("Enter the entries for Matrix 1: ");
    get_input(nr1, nc1, a);
    printf("Enter the entries for Matrix 2: ");
    get_input(nr2, nc2, b);

    printf("\nFirst Matrix: \n");
    displayMatrix(nr1, nc1, a);
    printf("\nSecond Matrix: \n");
    displayMatrix(nr2, nc2, b);

    int c[nr1][nc2];
    printf("\nMatrix Multiplication of Matrix 1 and Matrix 2 is: \n");
    pid_t pid[nr1 * nc2];
    int index = 0;
    int sum1, sum2;

    for(int i=0; i<nr1; i++)
    {
        for(int j=0; j<nc2; j += 2)
        {
            pid[index] = vfork(); //use of vfork()
            if(pid[index++] == 0)
            {
                sum1 = MatMul(i, j, a, b);
                c[i][j] = sum1;
                exit(0);
            }
            else
            {
                if(j+1 < nc2)
```

```
                {
                    sum2 = MatMul(i, j + 1, a, b);
                    c[i][j + 1] = sum2;
                }
            }
        }
    }

    waitpid(-1, & status, 0);
    displayMatrix(nr1, nc2, c);
    printf("\n");
    return 0;
}

void get_input(int row, int column, int arr[][column])
{
    for(int i=0; i<row; i++)
    {
        for(int j=0; j<column; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
}

void displayMatrix(int row, int column, int arr[][column])
{
    for(int i=0; i<row; i++)
    {
        for(int j=0; j<column; j++)
        {
            printf(" %5d", arr[i][j]);
        }
        printf("\n");
    }
}
```

```c
int MatMul(int a, int b, int arr1[][nc1], int arr2[][nc2])
{
    int sum = 0;
    for(int i=0; i<nr2; i++)
    {
        sum += arr1[a][i] * arr2[i][b];
    }
    return sum;
}
```

Output:



Explanation:

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.

Parallelization is done in the part of Row-Column Multiplication**(MatMul)** for Odd Columns is done in the parent process, and for Even Columns, Multiplication is done in the child process.

(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.

Filename: Q7_MagicSquareCheck.c

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

int size, sum1, sum2, rowsum, columnsum;

void get_input(int size, int arr[][size]);
void displayMagicSquare(int size, int arr[][size]);
int DiagonalSum(int arr[][size]);
int RowSum(int arr[][size]);
int ColumnSum(int arr[][size]);

int main()
{
   int sum = 0, flag1, flag2, status;
   printf("Enter order of Square Matrix: ");
   scanf("%d", &size);
   int arr[size][size];

   printf("Enter the entries for Matrix: ");
   get_input(size, arr);
   printf("\nSquare Matrix is: \n\n");
   displayMagicSquare(size, arr);
```

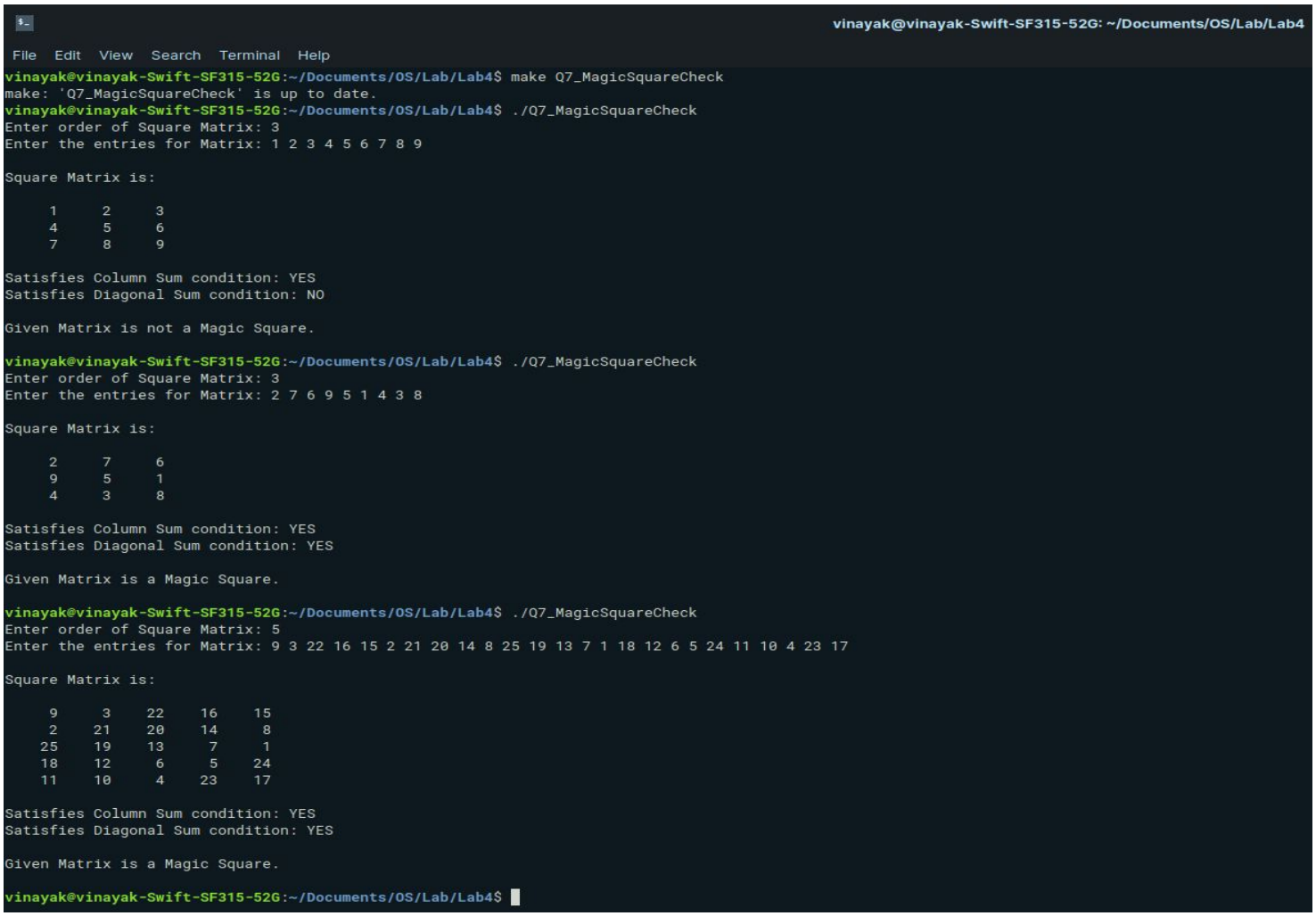
```
pid_t pid1, pid2, pid3;
pid1 = vfork();
if(pid1 == 0)
{
    sum = RowSum(arr);
    exit(0);
}
else
{
    pid2 = vfork();
    if(pid2 == 0)
    {
        printf("\nSatisfies Column Sum condition: ");
        if(sum == ColumnSum(arr))
        {
            flag1 = 1;
            printf("YES\n");
        }
        else
        {
            flag1 = 0;
            printf("NO\n");
        }
        exit(0);
    }
    else
    {
        pid3 = vfork();
        if(pid3 == 0)
        {
            printf("Satisfies Diagonal Sum condition: ");
            if(sum == DiagonalSum(arr))
            {
                flag2 = 1;
                printf("YES\n");
            }
            else
```

```
                {
                    flag2 = 0;
                    printf("NO\n");
                }

                exit(0);
            }
        }
    }

    waitpid(-1, &status, 0);
    if((flag1 == flag2) == 1)
        printf("\nGiven Matrix is a Magic Square.\n\n");
    else
        printf("\nGiven Matrix is not a Magic Square.\n\n");

    return 0;
}

int DiagonalSum(int arr[][size])
{
    // calculate the sum of the prime diagonal
    int i;
    sum1 = 0, sum2 = 0;
    for(i=0; i<size; i++)
        sum1 = sum1 + arr[i][i];
    // the secondary diagonal
    for(i=0; i<size; i++)
        sum2 = sum2 + arr[i][size-1-i];
     if(sum1 != sum2)
        return 0;
    else
        return sum1;
}

int RowSum(int arr[][size])
{
```

```
   // For sums of Rows
   int tempsum;
   for(int i=0; i<size; i++)
   {
       rowsum = 0;
       for(int j=0; j<size; j++)
       {
           rowsum += arr[i][j];

       }    // check if every row sum is equal to prime diagonal sum
           if(i == 0)
           {
               tempsum = rowsum;
               continue;
           }
           if(rowsum != tempsum)
               return 0;
   }

       return rowsum;
}

int ColumnSum(int arr[][size])
{
   // For sums of Columns
   int tempsum;
   for(int i=0; i<size; i++)
   {
       columnsum = 0;
       for(int j=0; j<size; j++)
       {
           columnsum += arr[j][i];
        }  // check if every column sum is equal to prime diagonal sum
           if( i==0 )
           {
               tempsum = columnsum;
               continue;
```

```
            }
            if(tempsum != columnsum)
                return 0;
    }

        return columnsum;
}

void get_input(int size, int arr[][size])
{
    for(int i=0; i<size; i++)
    {
        for(int j=0; j<size; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
}

void displayMagicSquare(int size, int arr[][size])
{
    for(int i=0; i<size; i++)
    {
        for(int j=0; j<size; j++)
        {
            printf(" %5d", arr[i][j]);
        }
        printf("\n");
    }
}
```

Output:



Explanation:

In this we have done parallelization by using vfork() and created 3 child processes, where the first child process stores the row sum of the square matrix.
The second child checks if Column Sum is equal to Row Sum or not and accordingly update the flag.
The third child checks if Diagonal Sum is equal to ROw Sum or not and accordingly updates the flag.

After completion of all the processes, on the basis of the value of the flag, we determine whether the given square matrix is a magic square or not.

(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...)

Filename: Q8_MagicSquareGeneration.c

```
#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 MagicSquareCheck(int size, int arr[][size]);

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

        MagicSquare(size, arr);
        exit(0);
    }
    else
    {
        wait(NULL);
        displayMagicSquare(size, arr);
        int valid = MagicSquareCheck(size, arr);
        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 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)
```

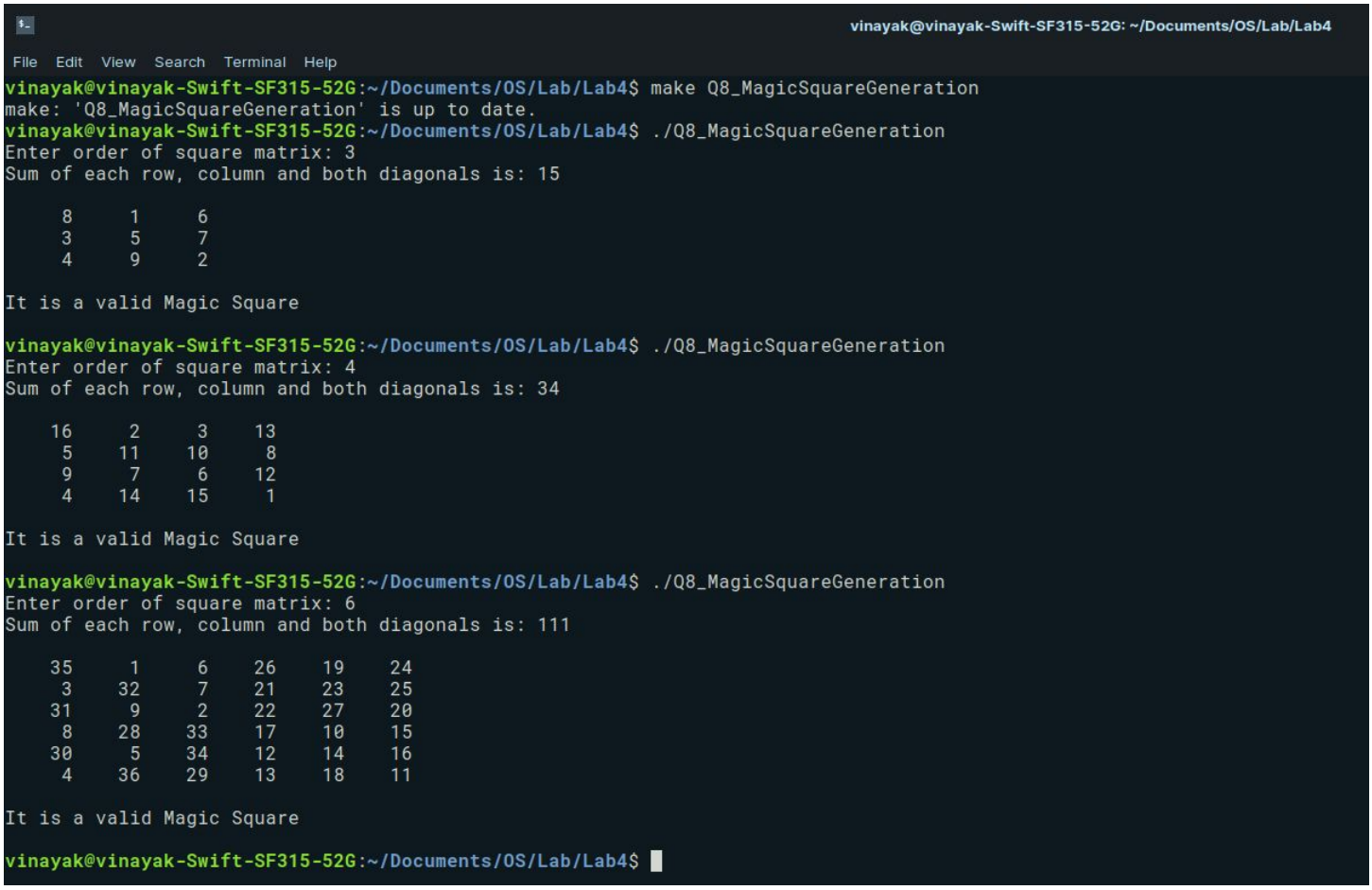
```
    {
        arr[i][j] = k;
        i--;
        j++;

        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;

    //prepare 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+1 - arr[i][j];
       }
}

void SinglyEvenMagicSquare(int size, int arr[][size])
{
   int N = size;
   int halfN = N/2; //size of ABCD boxes
   int k = (N-2)/4; // to get 'noses' of A & D boxes

   int temp;
   int new[N];

   int swapCol[N]; // columns which need to swap between C-B & A-D
   int index=0; // index of swapCol
   int miniMagic[halfN][halfN];

   OddOrderMagicSquare(halfN, miniMagic); //creating odd magic square for
A box

   //creating 4 magic boxes
   for(int i=0; i<halfN; i++)
       for (int j=0; j<halfN; j++)
       {
           arr[i][j] = miniMagic[i][j];              //A box
           arr[i+halfN][j+halfN] = miniMagic[i][j]+halfN*halfN; //B box
           arr[i][j+halfN] = miniMagic[i][j]+2*halfN*halfN;     //C box
           arr[i+halfN][j] = miniMagic[i][j]+3*halfN*halfN;     //D box
       }

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

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

   //swapping values between C-B & A-D by known columns
   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;
       }

   //swapping noses
   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;
   //end of swapping
}

void displayMagicSquare(int size, int arr[][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++)
   {
       for(int j=0; j<size; j++)
       {
           printf(" %5d", arr[i][j]);
       }
       printf("\n");
   }
}
```

```
int MagicSquareCheck(int size, int arr[][size])
{
   // calculate the sum of the prime diagonal
   int i, sum1 = 0, sum2 = 0;
   for(i=0; i<size; i++)
       sum1 = sum1 + arr[i][i];
   // the secondary diagonal
   for(i=0; i<size; i++)
       sum2 = sum2 + arr[i][size-1-i];
    if(sum1 != sum2)
       return 0;
    // For sums of Rows
   for(i=0; i<size; i++)
   {
       int rowSum = 0;
       for(int j=0; j<size; j++)
           rowSum += arr[i][j];

       // check if every row sum is equal to prime diagonal sum
       if(rowSum != sum1)
           return 0;
   }
    // For sums of Columns
   for(i=0; i<size; i++)
   {
       int colSum = 0;
       for(int j=0; j<size; j++)
           colSum += arr[j][i];
        // check if every column sum is equal to prime diagonal sum
       if(sum1 != colSum)
       return 0;
   }
    return 1;
}
```

Output:



Explanation:

Magic squares are generally classified according to their order *n* as:

1. odd if *n* is odd
2. evenly even (also referred to as "doubly even") if $n = 4k$ (e.g. 4, 8, 12, and so on)
3. 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 program vfork() is used for data sharing and in the parent process we have created a magic square, and in the child process we did a check that whether the generated magic square follows all the properties of magic square or not.