Adhesh Reghu Kumar
COE18B001

Operating Systems Lab

Lab Assignment - 4

1. Test drive a C program that creates Orphan and Zombie Processes

Logic Used

What is an Orphan Process ?

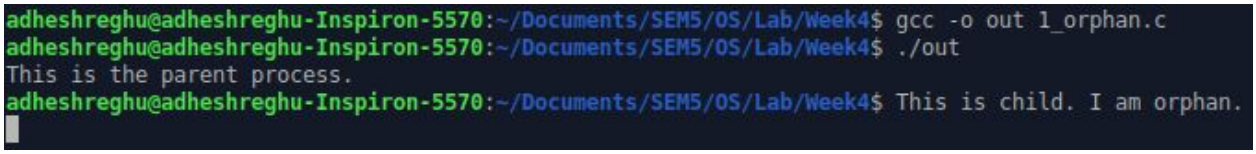
A process whose parent no more exists, i.e., has either terminated or completed without waiting for the child process to terminate.

Code to generate orphan process:

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

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

    if(pid > 0)
       printf("This is the parent process.\n");
    else if(pid == 0)
    {
       sleep(10);
       printf("This is child. I am orphan.\n");
    }

    return 0;
}
```

Output:

The child process sleeps for 10 seconds within which the parent finishes execution and terminates. You can notice that the terminal goes to the next line. After 10 seconds, the child completes execution and prints.



What is a Zombie Process ?

A process which has finished its execution but still has an entry in the process table to report to its parent is known as a zombie process.

Code

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

int main()
{
    pid_t pid = fork();
    if(pid > 0)
    {
       printf("This is the parent process\n");
       sleep(10);
    }
    else if(pid == 0)
    {
       printf("Child process\n");
       exit(0);
    }
    return 0;
}
```

Output

The zombie effect may not be apparent in the output, but the child process entry is maintained in the process tree of the parent.

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 1_zombie.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
This is the parent process
Child process
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

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]

a. Merge Sort

Logic Used

In this program, we have to develop a multiprocessing version of Merge Sort using fork() calls. Since there is a need to share data across processes, we will use vfork() call. In this program, we first accept the array of integers as command line arguments. Then with the help of the validation function, isInteger(), we validate the inputs and print a warning and ignore if the character input is not an integer. Otherwise, we accept the input and store it as integer in an array.

Once we have the array stored in an array, we can now proceed to sorting it. For this we call a recursive function, mergeSort(). In this function, we first find the midpoint of the input array. We then sort the two halves split at midpoint separately using two mergeSort() functions calls. Since we are developing a multiprocessing algorithm, we perform the mergeSort() for each half in a different process created using vfork().

Since the mergeSort() is a recursive function, the split happens until there is only one element left (which is sorted) in each array. Then we start merging the sorted arrays back using merge() function.

Let us take a look at the mergeSort() function:

```
void mergeSort(int arr[], int begin, int end)
{
    if(begin < end)
    {
       int mid = begin + (end - begin)/2;

       // sort both halves - concurrently
       pid_t pid = vfork();

       if(pid == 0)
       {
             // Handle first half within child process
             mergeSort(arr,begin,mid);
             exit(0);
       }
       else if(pid > 0)
       {
             // Handle second half within parent process
             mergeSort(arr,mid+1,end);
       }
       wait(NULL);

       merge(arr,begin,mid,end);
    }
}
```

The merge() function is the same as in a normal Merge Sort algorithm.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#define MAX 1000

int isInteger(char* string);
void merge(int arr[], int begin, int mid, int end);
void mergeSort(int arr[], int begin, int end);
```

```
int main(int argc, char *argv[])
{
    if(argc < 2)
       printf("Insufficent arguments.\n");
    else
    {
       int len = argc - 1;
       int arr[MAX],n=0,order=1;
       int i=1;
       for(;i<=len;++i)
       {
       // Validation function. Warning if not integer.
       if(isInteger(argv[i]))
             arr[n++] = atoi(argv[i]);
       else
             printf("warning: '%s' is not an integer. will be ignored.\n",argv[i]);
       }
       // Warning if entire argument list is invalid.s
       if(n == 0 )
             printf("The input list is invalid. Please enter an integer array.\n");
       else
       {
             // Enter Code Here
             mergeSort(arr,0,n-1);
             int i=0;
             printf("Sorted Array: ");
             for(;i<n;++i)
                   printf("%d ",arr[i]);
             puts("");
       }
    }
    return 0;
}
// Validation Function
int isInteger(char* str)
{
    int character,i=0;
    for(;i<strlen(str);++i)
    {
       character = str[i];
       // In case of '-'' symbol, continue if not last character in string.
```

```
        if(character == 45 && i<strlen(str)-1)
                continue;
        // Else get the decimal value (-48) and check condition.
        character -= 48;
        if(character < 0 || character > 9)
                return 0;
    }
    return 1;
}
void mergeSort(int arr[], int begin, int end)
{
    if(begin < end)
    {
        int mid = begin + (end - begin)/2;

        // sort both halves - concurrently
        pid_t pid = vfork();
        if(pid == 0)
        {
                // Handle first half within child process
                mergeSort(arr,begin,mid);
                exit(0);
        }
        else if(pid > 0)
        {
                // Handle second half within parent process
                mergeSort(arr,mid+1,end);
        }
        wait(NULL);

        merge(arr,begin,mid,end);
    }
}

void merge(int arr[], int begin, int mid, int end)
{
    int lsize = mid - begin + 1;
    int rsize = end - mid;

    // create left and right arrays
    int left[lsize], right[rsize];
```

```
    // fill in the arrays
    int i=0,j=0,k=0;
    for(i=0;i<lsize;++i)
       left[i] = arr[begin + i];
    for(j=0;j<rsize;++j)
       right[j] = arr[mid+1+j];
    k = begin;
    i=0,j=0;

    while(i<lsize && j<rsize)
    {
       if(left[i] < right[j])
             arr[k] = left[i++];
       else
             arr[k] = right[j++];

       ++k;
    }
    // Get the remaining elements from both the arrays into the merged array
    while(i<lsize)
       arr[k++] = left[i++];

    while(j<rsize)
       arr[k++] = right[j++];
}
```

Output:

With only valid inputs

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 2_merge.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out 3 1 2 3 4 5 6 7 1
Sorted Array: 1 1 2 3 3 4 5 6 7
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out 3 1 2 3 4 5 6 7 1 2 3 41 2 1 3 4 1
Sorted Array: 1 1 1 1 2 2 2 3 3 3 3 4 4 5 6 7 41
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

With invalid inputs as well.

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 2_merge.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out 3 1 2 3 4 5 6 7 1 a d g x  1 w d 1 4 5 2
warning: 'a' is not an integer. will be ignored.
warning: 'd' is not an integer. will be ignored.
warning: 'g' is not an integer. will be ignored.
warning: 'x' is not an integer. will be ignored.
warning: 'w' is not an integer. will be ignored.
warning: 'd' is not an integer. will be ignored.
Sorted Array: 1 1 1 1 2 2 3 3 4 4 5 5 6 7
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

b. Quick Sort [Extra Credits]

Logic Used

In this program, we have to develop a multiprocessing version of Quick Sort using fork() calls. Since there is a need to share data across processes, we will use vfork() call. In this program, we first accept the array of integers as command line arguments. Then with the help of the validation function, isInteger(), we validate the inputs and print a warning and ignore if the character input is not an integer. Otherwise, we accept the input and store it as integer in an array.

Now we proceed to sorting the array. For this we call the recursive quickSort() function. In the quickSort function, we pick an element as a pivot and partition the given array around the chosen pivot. Here we always chose the last element to be the pivot. After choosing the pivot, we partition the given array,i.e., we place all elements greater than pivot to the right and all elements less than pivot to the left and put pivot at its correct position. We return the pivot position after the partition. With the pivot position, we perform quickSort() once again for the right and left partitions. This is a divide and conquer strategy.

In the multiprocessing version of the algorithm, the quickSort() call for the left and right partitions have been done in different processes created using vfork().
Let us take a look at the quickSort() function:

```
void quickSort(int arr[], int begin, int end)
{
    if(begin < end)
    {
       int p = partition(arr,begin,end);
       // sort both halves - concurrently
       pid_t pid = vfork();
       if(pid == 0)
       {
             // Handle first half within child process
             quickSort(arr,begin,p-1);
             exit(0);
       }
```

```
        else if(pid > 0)
        {
                // Handle second half within parent process
                quickSort(arr,p+1,end);
        }
        wait(NULL);
    }
}
```

The partition function is the same linear time algorithm used in a normal quicksort algorithm.

Code

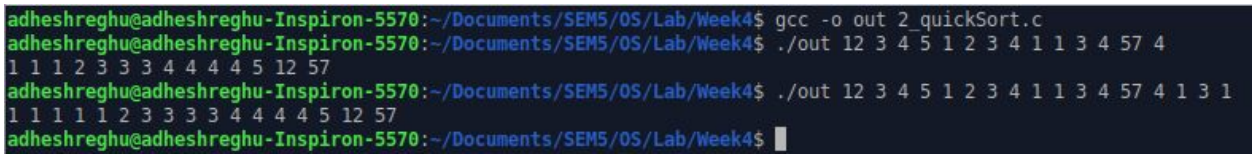
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <inttypes.h>
#include <sys/wait.h>
#include <string.h>
#define MAX 1000

int isInteger(char* string);
void quickSort(int arr[], int begin, int end);
int partition(int arr[],int begin, int end);

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

    if(argc < 2)
        printf("Insufficient arguments.\n");
    else
    {
        int len = argc - 1;
        int arr[MAX],n=0,order=1;
        int i=1;
        for(;i<=len;++i)
        {
        // Validation function. Warning if not integer.
        if(isInteger(argv[i]))
                arr[n++] = atoi(argv[i]);
```

```
        else
                printf("warning: '%s' is not an integer. will be ignored.\n",argv[i]);
        }

        // Warning if entire argument list is invalid.s
        if(n == 0 )
                printf("The input list is invalid. Please enter an integer array.\n");
        else
        {
                // Enter Code Here
                quickSort(arr,0,n-1);
                int i=0;
                for(;i<n;++i)
                        printf("%d ",arr[i]);
                puts("");
        }
    }
    return 0;
}

// Validation Function
int isInteger(char* str)
{
    int character,i=0;
    for(;i<strlen(str);++i)
    {
        character = str[i];
        // In case of '-'' symbol, continue if not last character in string.
        if(character == 45 && i<strlen(str)-1)
                continue;
        // Else get the decimal value (-48) and check condition.
        character -= 48;
        if(character < 0 || character > 9)
                return 0;
    }
    return 1;
}

// Quick Sort Function
void quickSort(int arr[], int begin, int end)
{
    if(begin < end)
```

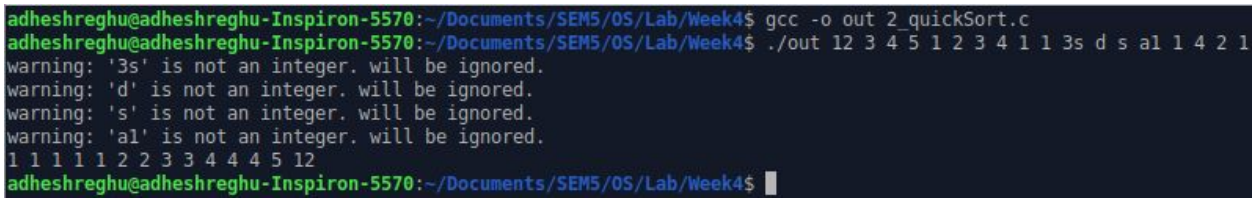
```
    {
       int p = partition(arr,begin,end);

       // sort both halves - concurrently
       pid_t pid = vfork();
       if(pid == 0)
       {
             // Handle first half within child process
             quickSort(arr,begin,p-1);
             exit(0);
       }
       else if(pid > 0)
       {
             // Handle second half within parent process
             quickSort(arr,p+1,end);
       }
       wait(NULL);
    }
}

// QuickSort - Partition Function
int partition(int arr[],int begin, int end)
{
    int pivot = arr[end];
    int i = begin - 1,j,temp;
    for(j= begin;j<=end-1;++j)
    {
       if(arr[j] < pivot)
       {
             i++;
             // swap
             temp = arr[i];
             arr[i] = arr[j];
             arr[j] = temp;
       }
    }

    temp = arr[i+1];
    arr[i+1] = arr[end];
    arr[end] = temp;
    return i + 1;
}
```

Output:

With only valid inputs



With combination of valid and invalid inputs



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

Logic Used

Here we are supposed to count the maximum number of processes that can be created using fork call. We can also see the question as counting the maximum number of child processes that can be created by any given process using fork(). The maximum limit is usually set due to the process entry table size limitations.
To find the limit on max count using code, we call for fork() within a *while* loop for the "main" process alone. Within the *main* process we increment the counter. We exit the loop for the child processes created. In case of pid = -1 condition we break from the loop and print the counter value.

Code

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

int main()
{
    unsigned long int count = 1;
    pid_t cpid = getpid();

    while(1)
    {
       if(getpid() == cpid)
       {
             ++count;
             // printf("%ld\n",count);
             pid_t pid = fork();
             if(pid == 0 || pid == -1)
                   break;
       }
    }

    if(getpid() == cpid)
       printf("Max Count: %ld\n",count);
    return 0;
}
```

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 3.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Max Count: 10343
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Max Count: 10344
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Max Count: 10344
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Max Count: 10345
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

4. Develop your command shell. Extend to support a history feature.

Logic Used

In this program, we have to develop our own custom command shell. The basic shell usually runs on a loop and does the following operations:

a. Read line from stdin
b. Parse the command line input
c. Update the history Log
d. Execute the command line arguments

At the end of each loop, the shell also frees the variables used. Let us take a look at each of the operations in detail.

A. Read line from stdin

We initialize a character pointer "*cmd_line*" to store the input command from stdin. We allocate memory to the character pointer using *malloc()*. Then we use the getline() function to read the line from stdin. If we receive no input (just a line break), we continue our loop from the start again. Else we remove the line break from the character pointer and pass it on to be parsed.

```
// Wait for user input - read a line
char *cmd_line;
size_t line_size = 1024;
cmd_line = (char *)malloc(line_size * sizeof(char));
if(cmd_line == NULL)
{
   perror("Unable to allocate memory");
   exit(0);
}

// getline() gets the entire line of input. replace last character with EOF
size_t len = getline(&cmd_line,&line_size,stdin);
if(cmd_line[0] == '\n')
      continue;
cmd_line[len-1] = 0;
```

B. Parse the command line input

We parse the input command line argument stored in a cmd_line using the parse_args() function. It returns a pointer to a character pointer which stores each command line argument parsed on space from the given input.
Let us take a look at the function.

```
char** parse_args(char *cmd_line)
{
    // Split the cmd_line on spaces
    char **arg_list = malloc(MAX * sizeof(char*));

    // Extract the first argument
    char *arg = strtok(cmd_line," ");
    int pos = 0;
    while(arg!=NULL)
    {
       arg_list[pos++] = arg;
       arg = strtok(NULL," ");
    }
    return arg_list;
}
```

The parsing is done with the help of strtok() function which splits the input character pointer on space and stores it in *arg_list*.

C. Update history Log

As a custom feature we also have to maintain a log of all commands executed (successful or not). So we maintain a *history.txt* log file in the current working directory (initial) and append all commands executed to the file. It is done by the historyLog() function which takes in the command line input and address of the history file as arguments. Let us take a look at the function.

```
void historyLog(char *cmd_line,char cwd[])
{
    FILE *fp = fopen(cwd,"a+");
    fprintf(fp, "%s\n",cmd_line);
    fclose(fp);
}
```

D. Execute the command line arguments

Now after having parsed the command line input to get the arguments, we have to execute them. There are two types of commands

i. InBuilt - exit, history (!num), help, cd
ii. System Commands - all linux system commands

These two types of commands have to be executed differently. An InBuilt command is usually an user defined command. Cd has to be executed differently than the other system commands, so it is also considered to be an inbuilt command.
A system command is executed by creating a new process and calling execvp() on the command line arguments. Execvp() function executes the command line arguments of the system command and prints the output on stdout.
The execute_args() function handles the execution of command line arguments. It uses utility functions such as isInBuiltCmd() to check if the input command is InBuilt or not. In case it is InBuilt we execute it differently using executeInBuiltCmd() function. Let us take a look at the exec_args() function and its helper functions.

```
int execute_args(char **cmd_args,char cwd[])
{
    int status = 0;

    // Check if command is in-built
    if(isInBuiltCmd(cmd_args[0]) == 1)
       status = executeInBuiltCmd(cmd_args,cwd);
    else
    {
       // Execute using execvp and parent waits for process completion
       pid_t pid;

       // Create process
       pid = vfork();
       if(pid == 0)
       {
             // Within child
             status = 1;
             if(execvp(cmd_args[0],cmd_args) == -1)
             {
                   printf("bash: Invalid command.\n");
                   status = 0;
             }
```

```
            exit(0);
        }
        else if(pid > 0)
        {
            // Within parent
            while(wait(NULL) > 0);
        }
    }
    return status;
}
```

Let us take a look at the isInBuiltCmd() function.

```
int isInBuiltCmd(char *cmd)
{
    char *inBuiltCmds[] = {"cd", "exit","help"};
    int num_cmds = sizeof(inBuiltCmds)/sizeof(inBuiltCmds[0]);

    // Compare strings to see if the cmd is In-Built command
    int i;
    for(i=0;i<num_cmds;++i)
        if(strcmp(cmd,inBuiltCmds[i]) == 0)
            return 1;

    if(cmd[0] == 33)
        return 1;

    return 0;
}
```

The cmd[0] = 33 (!) indicates the history command.

Let us take a look at the executeInBuildCmd() function.

```
int executeInBuiltCmd(char **cmd_args,char cwd[])
{
    // cd command.
    if(strcmp("cd",cmd_args[0]) == 0)
    {
```

```
        if(cmd_args[0] == NULL)
        {
                printf("bash: cd: Argument missing.\n");
                return 0;
        }
        else
        {
                if(chdir(cmd_args[1])!=0)
                {
                        printf("bash: cd: No such file or directory.\n");
                        return 0;
                }
        }
        return 1;
    }

    // exit command
    if(strcmp("exit",cmd_args[0]) == 0)
    {
        // quit program
        exit(0);
    }

    // history command
    int num_history = 0;
    if(cmd_args[0][0] == 33)
    {
        if((num_history = isNumber(cmd_args[0])) > 0)
        {
                int retrieve_len = retrieveHistory(num_history,cwd);
        }
        else
        {
                printf("bash: history: Invalid argument passed.\n");
                return 0;
        }
    }

    // printf("inBuilt\n");
    return 0;
}
```

The retrieveHistory() function reads the history.txt log file that we maintained and writes the last num commands executed onto the stdout.

Now let us take a look at the entire code.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <inttypes.h>
#include <string.h>
#define MAX 1024
void init_shell();
char** parse_args(char *cmd_line);
int execute_args(char **cmd_args,char cwd[]);
int isInBuiltCmd(char *cmd);
int executeInBuiltCmd(char **cmd_args,char cwd[]);
void historyLog(char *cmd_line,char cwd[]);

int main()
{
    // start the shell
    init_shell();
}

void init_shell()
{
    // get current directory
    char cwd[MAX];
    getcwd(cwd, sizeof(cwd));
    strcat(cwd,"/history.txt");

    // Initiate shell loop.
    while(1)
    {
       printf("@ ");

       // Wait for user input - read a line
       char *cmd_line;
       size_t line_size = 1024;
```

```
        cmd_line = (char *)malloc(line_size * sizeof(char));
        if(cmd_line == NULL)
        {
              perror("Unable to allocate memory");
              exit(0);
        }

        // getline() gets the entire line of input. replace last character with EOF
        size_t len = getline(&cmd_line,&line_size,stdin);
        if(cmd_line[0] == '\n')
              continue;
        cmd_line[len-1] = 0;


        // Write to history file
        historyLog(cmd_line,cwd);

        // Parse the user input
        char** cmd_args = parse_args(cmd_line);

        // Execute the command line args
        int status;
        status = execute_args(cmd_args,cwd);

        // Free all variables used
        free(cmd_line);
        free(cmd_args);
    }
}

char** parse_args(char *cmd_line)
{
    // Split the cmd_line on spaces
    char **arg_list = malloc(MAX * sizeof(char*));

    // Extract the first argument
    char *arg = strtok(cmd_line," ");
    int pos = 0;
    while(arg!=NULL)
    {
        arg_list[pos++] = arg;
        arg = strtok(NULL," ");
```

```
    }
    return arg_list;
}

int execute_args(char **cmd_args,char cwd[])
{
    int status = 0;

    // Check if command is in-built
    if(isInBuiltCmd(cmd_args[0]) == 1)
       status = executeInBuiltCmd(cmd_args,cwd);
    else
    {
       // Execute using execvp and parent waits for process completion
       pid_t pid;

       // Create process
       pid = vfork();
       if(pid == 0)
       {
             // Within child
             status = 1;
             if(execvp(cmd_args[0],cmd_args) == -1)
             {
                   printf("bash: Invalid command.\n");
                   status = 0;
             }
             exit(0);
       }
       else if(pid > 0)
       {
             // Within parent
             while(wait(NULL) > 0);
       }
    }
    return status;
}

int isInBuiltCmd(char *cmd)
{
    char *inBuiltCmds[] = {"cd", "exit","help"};
    int num_cmds = sizeof(inBuiltCmds)/sizeof(inBuiltCmds[0]);
```

```
    // Compare strings to see if the cmd is In-Built command
    int i;
    for(i=0;i<num_cmds;++i)
       if(strcmp(cmd,inBuiltCmds[i]) == 0)
             return 1;

    if(cmd[0] == 33)
       return 1;

    return 0;
}


int executeInBuiltCmd(char **cmd_args,char cwd[])
{
    // cd command.
    if(strcmp("cd",cmd_args[0]) == 0)
    {
       if(cmd_args[0] == NULL)
       {
             printf("bash: cd: Argument missing.\n");
             return 0;
       }
       else
       {
             if(chdir(cmd_args[1])!=0)
             {
                   printf("bash: cd: No such file or directory.\n");
                   return 0;
             }
       }
       return 1;
    }
    // exit command
    if(strcmp("exit",cmd_args[0]) == 0)
    {
       // quit program
       exit(0);
    }
    // history command
    int num_history = 0;
```

```
    if(cmd_args[0][0] == 33)
    {
       if((num_history = isNumber(cmd_args[0])) > 0)
       {
             int retrieve_len = retrieveHistory(num_history,cwd);
       }
       else
       {
             printf("bash: history: Invalid argument passed.\n");
             return 0;
       }
    }
    return 0;
}
void historyLog(char *cmd_line,char cwd[])
{
    FILE *fp = fopen(cwd,"a+");
    fprintf(fp, "%s\n",cmd_line);
    fclose(fp);
}

int isNumber(char *cmd)
{
    // printf("Number = %s\n",cmd);
    int pos = 1;
    int len = strlen(cmd);
    for(;pos<len;++pos)
    {
       if(cmd[pos] < 48 || cmd[pos] > 57)
             return 0;
    }

    cmd = cmd+1;
    return atoi(cmd);
}

int retrieveHistory(int num_history,char cwd[])
{
    FILE *fp = fopen(cwd,"r");
    char record_list[MAX][MAX];
    char record[MAX];
    int i,num=0;
```

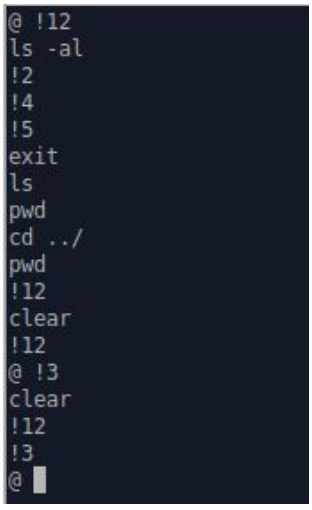
```
    int size = 1024;
    long int record_length = 0;

    while (fgets(record, size, fp))
    {
       record_length = strlen(record);
       record[record_length-1] = 0;
       strcpy(record_list[num],record);
       num = num + 1;
    }

    if(num < num_history)
    {
       for(i=0;i<num;++i)
             printf("%s\n",record_list[i]);
       printf("hist: only %d records available..\n",num);
       return num;
    }
    else
    {
       for(i=(num - num_history);i<num;++i)
             printf("%s\n",record_list[i]);
    }

    return num_history;
}
```

Output:

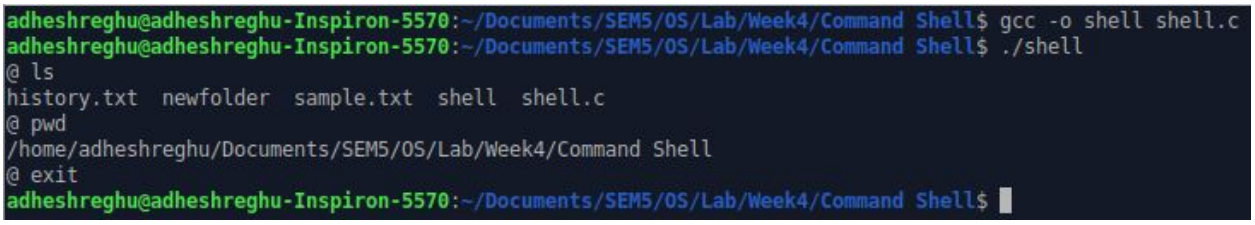
Execution of some system commands

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4/Command Shell$ gcc -o shell shell.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4/Command Shell$ ./shell
@ ls
history.txt  sample.txt  shell  shell.c
@ pwd
/home/adheshreghu/Documents/SEM5/OS/Lab/Week4/Command Shell
@ cd ../
@ pwd
pwd: ignoring non-option arguments
/home/adheshreghu/Documents/SEM5/OS/Lab/Week4
@
```

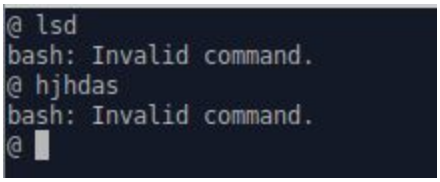
Execution of history command



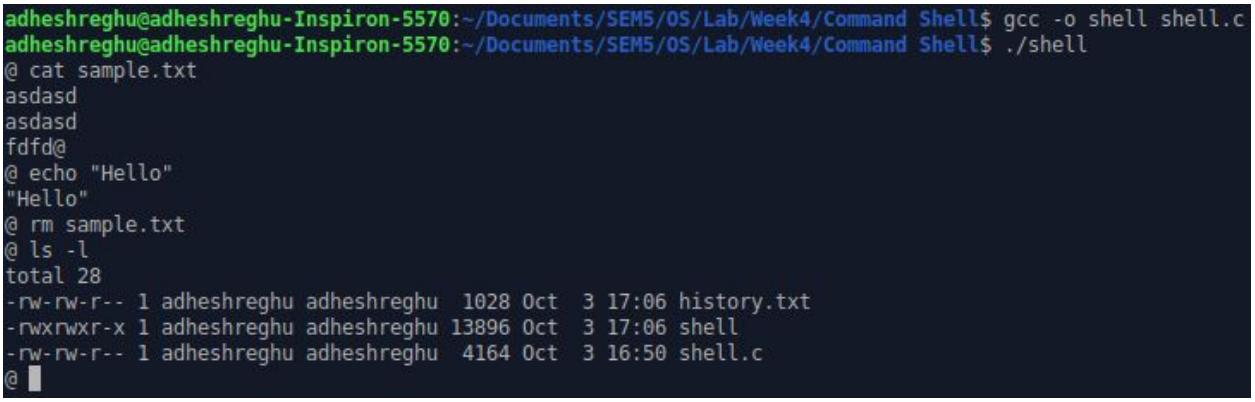
Execution of exit command



Execution of invalid commands



Execution of some more commands



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

Logic Used

In this program, we have to count the occurrences of characters in the input string. We have to develop a multiprocessing algorithm to achieve this.

The most efficient way to count the occurrences of characters is a hash table based approach which is a linear time algorithm. In it we maintain a hash table of size 256 (ASCII extended) with initial values set as 0. We then traverse the input string and update the count of the character by adding 1 to the corresponding position.
We wish to develop a multiprocessing version of this. The best approach would be to split the search area and have multiple processes search different search areas. In a true parallel setup, this algorithm can run faster than the linear algorithm as multiple processes running parallelly are updating the count.

In my code, I have split in the middle of the input string and run a search using a different process (created using vfork()) to update the count of that search area. The extent of split can be determined as per requirement. I have continually split and created new processes until there is only 1 character for each process. It is somewhat similar to the mergeSort implementation (where mergeSort is called for the two halves of the array by different processes).

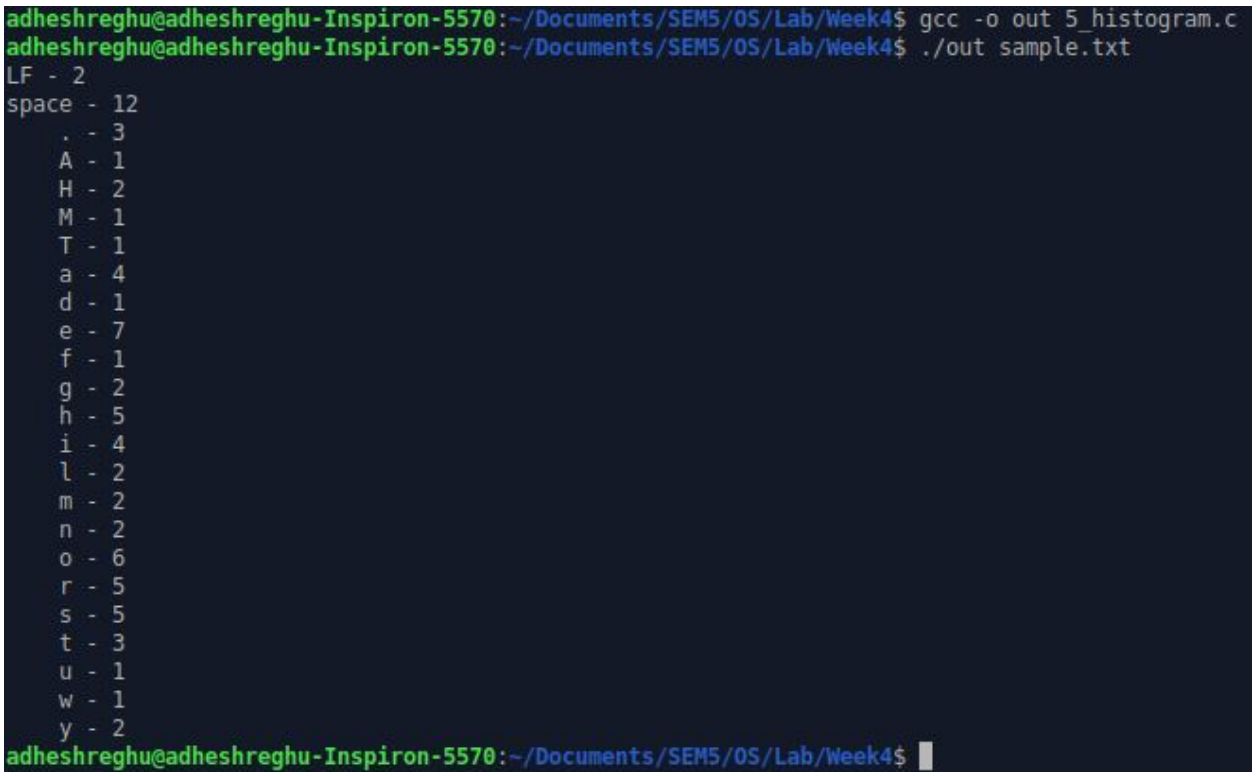
Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <inttypes.h>
#include <string.h>
#include <time.h>
#define MAX 10000
#define MOD 1000000007
#define dd double

int readFile(char src[],char buffer[]);
```

```
void updateCount(int charHash[],char string[],int begin,int end);
void countCharacters(char string[]);
void updateCountSequential(int charHash[],char string[],int end);
int main(int argc, char *argv[])
{
    if(argc <=1)
       printf("Insufficient arguments\n");
    else
    {
       char buffer[MAX];
       int status = readFile(argv[1],buffer);
       countCharacters(buffer);
    }
}


void countCharacters(char string[])
{
    // Maintain a hash table of size 256 corresponding to the 256 characters of the
ASCII table
    int charHash[256] = {0};
    int len = strlen(string);

    updateCount(charHash,string,0,len-1);
    // print character count
    int i=0;
    for(;i<256;++i)
    {
       if(charHash[i]!=0)
       {
             if(i == 32)
                   printf("space - %d\n",charHash[i]);
             else if(i == 10)
                   printf("LF - %d\n",charHash[i]);
             else
                   printf("%5c - %d\n",i,charHash[i]);
       }
    }
}
```

```
// A hash-based concurrent search algorithm.
void updateCount(int charHash[],char string[],int begin,int end)
{
    if(begin==end)
       ++charHash[(int)string[begin]];
    else if(begin<end)
    {
       int mid = begin + (end - begin)/2;
       pid_t pid = vfork();
       if(pid == 0)
       {
             updateCount(charHash,string,begin,mid);
             exit(0);
       }
       else if(pid > 0)
             updateCount(charHash,string,mid+1,end);
    }
}
int readFile(char src[],char buffer[])
{
    int i=0;

    // Open File
    FILE *fp = fopen(src,"r");

    while((buffer[i]=fgetc(fp))!= EOF)
       ++i;

    buffer[i] = '\0';
}
```

Output:



6. Develop a multiprocessing version of Matrix Multiplication.

Logic Used

In this program, we have to develop a multiprocessing version of matrix multiplication. The naive algorithm for matrix multiplication is a $O(n^3)$ algorithm that calculates the the product using the formula

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

[Extra Credit Method]

One way to parallelise is to create a process for each (i,j) position and calculate the product in a different process. The total number of processes required in such an algorithm is

$$m * n \text{ , } where\ dim(A) = m * l \ \ and \ \ dim(B) = l * n$$

Now, we can create m*n process before hand using vfork() calls in a *for()* loop.

```
pid_t pid[dim[0][0]];
pid_t pid_col[dim[0][1]];
for(i=0;i<dim[0][0];++i)
{
   if(getpid() == parent)
   {
      pid[i] = vfork();
      if(pid[i] == 0)
      {
            for(j=0;j<dim[0][1];++j)
            {
                  pid_col[j] = vfork();
                  if(pid_col[j] == 0)
                        break;
            }
            break;
      }
   }
}
```

[Note: vfork does not implement in a parallel sense. However when we translate the same algorithm using fork() and pipes() we can achieve parallelism (or concurrency).]

Once we have created m*n processes, in an ideal parallel setup each of them would run concurrently (or parallely) and each would have the matrix multiplication loop as the image. The matrix multiplication loop looks like this.

```
// Start the multiplication process
int l,m,p;
for(l=0;l<dim[0][0];++l)
{
    if(pid[l] == 0)
    {
       for(m=0;m<dim[1][1];++m)
       {
             if(pid_col[m] == 0)
             {
                   product[l][m] = 0;
                   for(p=0;p<dim[0][1];++p)
```

```
                        product[l][m] = product[l][m] + A[l][p] * B[p][m];
                  exit(0);
            }
            else
                  wait(NULL);
      }
      exit(0);
   }
   else
      wait(NULL);
}
while(wait(NULL) > 0);
```

Here you can notice that a cell multiplication occurs only in a particular process (restricted by if). Even though all processes have to go through the entire outer loop, this would occur faster as all processes are running concurrently (or parallely) and each process has to execute the dominant operation of multiplication only once. All the rest of the times it only has to check the if condition and loop.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <inttypes.h>
#include <time.h>
#define MAX 100

int main()
{
    pid_t parent = getpid();
    // Accept the dimensions of the matrices
    int dim[2][2];
    printf("Enter dimensions (m,n) for Matrix-1:\n");
    scanf("%d %d",&dim[0][0],&dim[0][1]);

    printf("Enter dimensions (m,n) for Matrix-2:\n");
```

```
    scanf("%d %d",&dim[1][0],&dim[1][1]);

    // Do dimensionality check
    if(dim[0][1] != dim[1][0])
    {
       printf("Matrices of given dimensions cannot be multiplied.\n");
       return 0;
    }

    // Accept the matrices
    int A[dim[0][0]][dim[0][1]];
    int B[dim[1][0]][dim[1][1]];
    int i,j,k;

    printf("Enter the elements of Matrix-1\n");
    for(i=0;i<dim[0][0];++i)
       for(j=0;j<dim[0][1];++j)
             scanf("%d",&A[i][j]);

    printf("Enter the elements of Matrix-2\n");
    for(i=0;i<dim[1][0];++i)
       for(j=0;j<dim[1][1];++j)
             scanf("%d",&B[i][j]);
    // declare product matrix
    int product[dim[0][0]][dim[1][1]];

    // Have to create processes for each row and each column. - m*n processes
    pid_t pid[dim[0][0]];
    pid_t pid_col[dim[0][1]];
    for(i=0;i<dim[0][0];++i)
    {
       if(getpid() == parent)
       {
             pid[i] = vfork();
             if(pid[i] == 0)
             {
                   for(j=0;j<dim[0][1];++j)
                   {
                         pid_col[j] = vfork();
                         if(pid_col[j] == 0)
                               break;
                   }
```

```
                    break;
                }
        }
    }

// Start the multiplication process
int l,m,p;
for(l=0;l<dim[0][0];++l)
{
    if(pid[l] == 0)
    {
        for(m=0;m<dim[1][1];++m)
        {
                if(pid_col[m] == 0)
                {
                        product[l][m] = 0;
                        for(p=0;p<dim[0][1];++p)
                                product[l][m] = product[l][m] + A[l][p] * B[p][m];
                        exit(0);
                }
                else
                        wait(NULL);
        }
        exit(0);
    }
    else
        wait(NULL);
}
while(wait(NULL) > 0);

// Print the product matrix
printf("The product of the matrix multiplication is:\n");
for(i=0;i<dim[0][0];++i)
{
    for(j=0;j<dim[1][1];++j)
        printf("%d ",product[i][j]);
    puts("");
}

return 0;
}
```

Output:

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 6_matrixMul.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter dimensions (m,n) for Matrix-1:
4 3
Enter dimensions (m,n) for Matrix-2:
3 4
Enter the elements of Matrix-1
1 2 3
1 2 3
1 2 3
1 2 3
Enter the elements of Matrix-2
1 2 3 4
1 2 3 4
1 2 3 4
The product of the matrix multiplication is:
6 12 18 24
6 12 18 24
6 12 18 24
6 12 18 24
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

Some more outputs

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ gcc -o out 6_matrixMul.c
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter dimensions (m,n) for Matrix-1:
2 2
Enter dimensions (m,n) for Matrix-2:
2 2
Enter the elements of Matrix-1
1 2
3 4
Enter the elements of Matrix-2
5 6
7 8
The product of the matrix multiplication is:
19 22
43 50
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

7. Develop a parallelized application to check for if a user input square matrix is a magic square or not.

Logic Used

In this program, we have to check if the input square matrix is a magic square or not. To check if an input matrix is a magic square, we have to find the sum of rows, sum of columns and sum of both diagonals and verify that they are all equal.

A simple way to parallelize the program is by having a process check if the rows sums are equal, another process checks if the column sums are equal and a third process checks if the diagonal sums are equal.

We can further parallelize the algorithm by creating n child processes within the process calculating the row sums where each of the n children independently calculate the sum of the ith row.

Similarly we can create n child processes within the process checking the column sums and two child processes within the process checking the diagonal sum.
We can then return -1 if they are not equal and return the respective sums if they are equal and then verify if the sums from row, column and diagonal process are equal or not.

So in total we have 2n+2 processes created for this algorithm.

Code:

```
#include <bits/stdc++.h>
using namespace std;
#include <sys/types.h>
#include<sys/wait.h>
#include <unistd.h>
#include <inttypes.h>
#define MAX 100

int rowSum(vector< vector<int> > A,int n);
int columnSum(vector< vector<int> > A,int n);
int diagonalSum(vector< vector<int> > A,int n);
```

```
int main()
{
    // Accept the dimension of the square matrix
    int n;
    printf("Enter order of the matrix (n)\n");
    cin>>n;

    // Accept n*n square matrix
    vector< vector<int> > A(n,vector<int>(n,0));
    printf("Enter the elements of the matrix.\n");
    for(int i=0;i<n;++i)
       for(int j=0;j<n;++j)
             cin>>A[i][j];

    // create 3 processes to handle the row, column and diagonal sums concurrently
    int rstatus,cstatus,dstatus;
    pid_t pid_r = vfork();
    if(pid_r == 0)
    {
       rstatus = rowSum(A,n);
       exit(0);
    }
    else
    {
       pid_t pid_c = vfork();
       if(pid_c == 0)
       {
             cstatus = columnSum(A,n);
             exit(0);
       }
       else
       {
             dstatus = diagonalSum(A,n);
       }
    }

    while(wait(NULL) > 0);
    if(dstatus == cstatus && cstatus == rstatus && rstatus > 0 && dstatus > 0)
       printf("Given matrix is a Magic Square with Maigc Sum = %d\n",cstatus);
    else
       printf("Given matrix is NOT a Magic Square.\n");
```

```
    return 0;
}

int rowSum(vector< vector<int> > A,int n)
{
    int sumR[n];
    pid_t pidR[n];
    pid_t rowParent = getpid();

    // Create n process to fill the sumR array concurrently
    for(int i=0;i<n;++i)
    {
       if(getpid() == rowParent)
       {
             pidR[i] = vfork();
             if(pidR[i] == 0)
                   break;
       }
    }

    // Here fill the sumR[i] using pidR[i] process concurrently
    for(int i=0;i<n;++i)
    {
       if(pidR[i] == 0)
       {
             sumR[i] = 0;
             for(int j=0;j<n;++j)
                   sumR[i] += A[i][j];
             exit(0);
       }
    }
    // wait for all children processes to complete
    while(wait(NULL) > 0);

    // Here linearly check if all sumR[i] are the same
    for(int i=1;i<n;++i)
    {
       if(sumR[i] != sumR[i-1])
             return -1;
    }
    return sumR[0];
}
```

```
int columnSum(vector< vector<int> > A,int n)
{
    int sumC[n];
    pid_t pidC[n];
    pid_t colParent = getpid();

    // Create n process to fill the sumC array concurrently
    for(int i=0;i<n;++i)
    {
       if(getpid() == colParent)
       {
             pidC[i] = vfork();
             if(pidC[i] == 0)
                   break;
       }
    }

    // Here fill the sumC[i] using pidC[i] process concurrently
    for(int i=0;i<n;++i)
    {
       if(pidC[i] == 0)
       {
             sumC[i] = 0;
             for(int j=0;j<n;++j)
                   sumC[i] += A[j][i];
             exit(0);
       }
    }
    // wait for all children processes to complete
    while(wait(NULL) > 0);

    // Here linearly check if all sumC[i] are the same
    for(int i=1;i<n;++i)
    {
       if(sumC[i] != sumC[i-1])
             return -1;
    }
    return sumC[0];
}
```

```
int diagonalSum(vector< vector<int> > A,int n)
{
    // Create two processes
    int sumD[2];
    pid_t pidD = vfork();
    if(pidD == 0)
    {
       sumD[0] = 0;
       for(int i=0;i<n;++i)
             sumD[0] += A[i][i];
       exit(0);
    }
    else if(pidD > 0)
    {
       sumD[1] = 0;
       for(int i=0;i<n;++i)
             sumD[1] += A[i][n-i-1];
    }
    // wait for the child process to complete.
    wait(NULL);

    if(sumD[0]!= sumD[1])
       return -1;

    return sumD[0];
}
```

Output:

When input is a magic square.

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out 7_magSqrSumChk.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter order of the matrix (n)
5
Enter the elements of the matrix.
17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
Given matrix is a Magic Square with Maigc Sum = 65
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter order of the matrix (n)
14
Enter the elements of the matrix.
177 186 195   1 10 19 28 128 137 146 99 108 68 77
185 194 154   9 18 27 29 136 145 105 107 116 76 78
193 153 155 17 26 35 37 144 104 106 115 124 84 86
152 161 16 172 34 36 45 103 112 114 123 132 85 94
160 162 171 33 42 44   4 111 113 122 131 140 93 53
168 170 179 41 43   3 12 119 121 130 139 141 52 61
169 178 187 49   2 11 20 120 129 138 147 100 60 69
30 39 48 148 157 166 175 79 88 97 50 59 117 126
38 47   7 156 165 174 176 87 96 56 58 67 125 127
46   6   8 164 173 182 184 95 55 57 66 75 133 135
 5 14 163 25 181 183 192 54 63 65 74 83 134 143
13 15 24 180 189 191 151 62 64 73 82 91 142 102
21 23 32 188 190 150 159 70 72 81 90 92 101 110
22 31 40 196 149 158 167 71 80 89 98 51 109 118
Given matrix is a Magic Square with Maigc Sum = 1379
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

When input is not a magic square.

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter order of the matrix (n)
4
Enter the elements of the matrix.
16  2  3 13
 5 11 10  8
 9  7  6 12
 4 14 15  8
Given matrix is NOT a Magic Square.
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

8. Extend the above to also support magic square generation.

Logic Used

In this program, we have to generate a magic square of a given order n.
Magic squares can be classified into three types based on the algorithm used to generate them

a. Odd Magic Squares (order = 2 * n + 1)
b. Doubly Even Magic Squares (order = 4 * n)
c. Singly Even Magic Squares (order = 4 * n + 2)

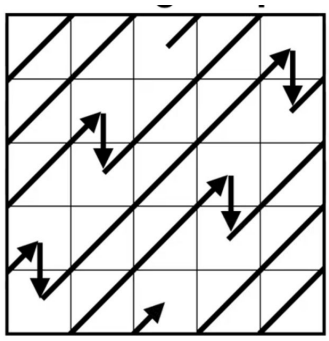
Each of these classes of magic square use a different algorithm. However in all the algorithms we fill the magic square with numbers from 1 to n^2 . Let us start looking at each of these algorithms.

a. Odd Magic Squares

Order = 2 * n + 1

The basic algorithm in generating an odd magic square is we start in the upper middle square (0,n/2) and traverse diagonally to the top-right and back or wrap around whenever we hit an edge. If we encounter a cell that is already filled, we go to the cell below the current one. In this pattern, we fill all the cells by starting at 1.

Trace of filling a 5 * 5 magic square is



		1		
	5			
4				
				3
			2	

		1	8	
	5	7		
4	6			
				3
			2	9

		1	8	15
	5	7	14	
4	6	13		
10	12			3
11			2	9

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

If we notice carefully, we can observe that 13 (ceil($n^2/2$) = ceil(25/2)) occurs at the position (n/2,n/2). This is true for all odd squares,i.e., ceil($n^2/2$) occurs at (n/2,n/2) position. So we can split the entire generation process into two.
First process - can generate numbers from 1 to ceil($n^2/2$)-1 starting at position (0,n/2)
Second process - can generate numbers from ceil($n^2/2$) to n^2 starting at position (n/2,n/2).

Each of the two processes would follow the same algorithm of traversing to the top-right and wrapping around.

In this way we can parallelize the generation of odd magic squares.

The function used to generate odd magic squares is generateOddSquare().

Code:

```
void generateOddSquare(int n, vector<vector <int > > &magicSqr,int row, int col,
int offset)
{
    // Let us generate the magic square of odd order n*n using vfork() - here let's
split the process in two
    pid_t pid = vfork();
    if(pid == 0)
    {
       // In the child process, lets fill from 1 to floor(n^2/2) in the matrix
       int i = row + 0,j = col + ceil(n/2);
       int num = 1 + offset,target = (n*n)/2 + offset;
       while(num <= target)
       {
             magicSqr[i][j] = num;
             ++num;

             // Check for boundary or cell-fill condition
             if(magicSqr[(((i-1)%n + n)%n + row)][(j+1)%n + col] == 0)
             {
                   i = ((i-1)%n + n)%n + row;
                   j = (j+1)%n + col;
             }
             else
                   i = (i + 1)%n + row;
       }
       exit(0);
    }
    else if(pid > 0)
    {
       // In the parent process, let us fill from n^2/2 + 1 to n^2.
       int i = ceil(n/2) + row,j = ceil(n/2) + col;
       int num = (n*n)/2 + 1 + offset,target = n*n + offset;
```

```
        while(num <= target)
        {
                magicSqr[i][j] = num;
                ++num;

                // Check for boundary or cell-fill condition
                if(magicSqr[(((i-1)%n + n)%n) + row][(j+1)%n + col] == 0)
                {
                        i = ((i-1)%n + n)%n + row;
                        j = (j+1)%n + col;
                }
                else
                        i = (i + 1)%n + row;
        }
    }
    else
        perror("Process could not be created.");

    wait(NULL);
}
```

Here parameters like *col*, *row*, *offset* are not used in the generation of odd squares but rather used as part of generation of singly even magic squares (will be explained there).

Output:

- 5*5 magic square with magic sum = 65

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out magicSqr.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
5
Magic Square is
17 24  1  8 15
23  5  7 14 16
 4  6 13 20 22
10 12 19 21  3
11 18 25  2  9
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

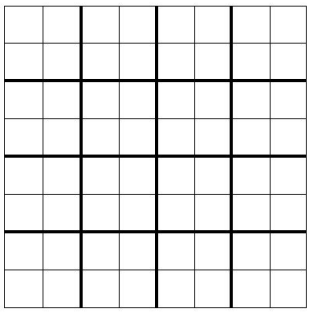
- 3*3 magic square with magic sum = 15

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out magicSqr.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
3
Magic Square is
 8  1  6
 3  5  7
 4  9  2
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

b. Doubly Even Magic Squares

Order = 4 * n

The basic algorithm in generating a doubly even magic square is to first fill the square with numbers from 1 to n^2 starting at (n-1,0) and going to (0,0) traversing from right to left from bottom.
After that we divide the entire matrix into 16 grids of area = n*n/16. For example for a 8*8 matrix we divide the matrix into 16 grids of each size 2*2.



Then for the top-right, top-left, mid-right (both), mid-left (both), bottom-right and bottom-left we update the cell values with

magicSqr[i][j] = $n^2 + 1$ - magicSqr[i][j]

What to parallelize ?
Since the updation to the four grids can be done independently, we execute them in four different processes created using vfork().

Let us take a look at the function generateDoublyEvenSqr().

Code

```
void generateDoublyEvenSqr(int n,vector<vector <int > > &magicSqr)
{
    int i,j;
    // First let us fill the matrix
    for(i=0;i<n;++i)
       for(j=0;j<n;++j)
             magicSqr[i][j] = (n*i) + j + 1;

    pid_t pid = vfork();
    if(pid == 0)
    {
       pid_t pid_0 = vfork();
       if(pid_0 == 0)
       {
             // Bottom-Right
             for (i = 3 * n/4; i < n; i++)
                   for ( j = 3 * n/4; j < n; j++)
                         magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];

             // Center of Matrix
             for (i = n/4; i < 3 * n/4; i++)
                   for (j = n/4; j < 3 * n/4; j++)
                         magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];

             exit(0);
       }
       else if(pid_0 > 0)
       {
             // Bottom Left
             for (i = 3 * n/4; i < n; i++)
                   for (j = 0; j < n/4; j++)
                  magicSqr[i][j] = (n*n+1) - magicSqr[i][j];
       }
       wait(NULL);
       exit(0);
    }
    else if(pid > 0)
    {
       pid_t pid_1 = vfork();
       if(pid_1 == 0)
```

```
        {
                // Top-Right Corner
                for(i=0;i<n/4;++i)
                        for(j=3*(n/4);j<n;++j)
                                magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];
                exit(0);
        }
        else if(pid_1 > 0)
        {
                // Top-Left Corner
                for(i=0;i<n/4;++i)
                        for(j=0;j<n/4;++j)
                                magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];
        }
        wait(NULL);
    }
    else
        perror("Process could not be created.");
    wait(NULL);
}
```

Output:

- 4*4 Magic Square with magic sum = 34

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out magicSqr.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
4
Magic Square is
16  2  3 13
 5 11 10  8
 9  7  6 12
 4 14 15  1
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

- 8*8 Magic Square with magic sum = 260

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out magicSqr.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
8
Magic Square is
64 63  3  4  5  6 58 57
56 55 11 12 13 14 50 49
17 18 46 45 44 43 23 24
25 26 38 37 36 35 31 32
33 34 30 29 28 27 39 40
41 42 22 21 20 19 47 48
16 15 51 52 53 54 10  9
 8  7 59 60 61 62  2  1
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

c. Singly Even Magic Squares

Order = 4 * n + 2

If we want to create a magic square of order n = 4k + 2, then we begin by partitioning the square into four (2k+1) * (2k+1) squares. (All of squares of odd order)

- In the upper left square we place, using the method we used to generate odd magic squares, the integers 1, 2 , 3, $n^2/4$.
- In the lower right-hand square we place the numbers $n^2/4+1$, $n^2/4+2$,..., $n^2/2$, again using the method we used to generate odd magic squares.
- In the upper right-hand square we place the numbers $n^2/2+1$, $n^2/2+2$,...,$3n^2/4$.
- Finally in the lower left-hand square we place the numbers $3n^2/4+1$, $3n^2/4+2$,..., n^2 using the same odd generation method.

Swaps and Adjustments

- Next, in the first (k-1) columns and in the last (k-1) columns we exchange the corresponding elements in the upper square with those in the lower square
- In the kth column we exchange all but the middle elements of this column.
- Finally we exchange the middle element of the upper and lower squares in the (k+1)st column.

The result should be an nxn magic square.
Each of the bullet points given above can be executed in different processes. The first four processes will fill the four odd magic squares.

The next four processes will be responsible for the swaps and adjustments mentioned in the algorithm.

Code

```
void generateSinglyEvenSqr(int n,vector<vector <int > > &magicSqr)
{
    // Divide the matrix into four sub matrices of size
    int size = n/2;
    // Do each quarter generation in a process
    pid_t pid[6];
    pid[0] = vfork();
    if(pid[0] == 0)
    {
       pid[1] = vfork();
       if(pid[1] == 0)
       {
             generateOddSquare(size,magicSqr,size,size,size*size);
             exit(0);
       }
       else
             generateOddSquare(size,magicSqr,size,0,3*(size*size));
       wait(NULL);
       exit(0);
    }
    else
    {
       pid[2] = vfork();
       if(pid[2] == 0)
       {
             generateOddSquare(size,magicSqr,0,size,2*(size*size));
             exit(0);
       }
       else
             generateOddSquare(size,magicSqr,0,0,0);
       wait(NULL);
    }
    wait(NULL);
    // Now let us do the swaps necessary - in four processes
    int k = (n-2)/4;
    pid[3] = vfork();
    if(pid[3] == 0)
```

```
    {
       pid[4] = vfork();
       if(pid[4] == 0)
       {
             // kth column - all but middle
             for(int i=0;i<size;++i)
                   if(i!=size/2)
                         swap(magicSqr[i][k-1],magicSqr[i+size][k-1]);
             exit(0);
       }
       else
       {
             // Swap middle Element of k+1 th col
             swap(magicSqr[size/2][k],magicSqr[size/2 + size][k]);
       }
       wait(NULL);
       exit(0);
    }
    else
    {
       pid[5] = vfork();
       if(pid[5] == 0)
       {
             // Last (k-1) columns
             for(int i=n-k+1;i<n;++i)
                   for(int j=0;j<size;++j)
                         swap(magicSqr[j][i],magicSqr[j+size][i]);
             exit(0);
       }
       else
       {
             // First (k-1) columns
             for(int i=0;i<k-1;++i)
                   for(int j=0;j<size;++j)
                         swap(magicSqr[j][i],magicSqr[j+size][i]);
       }
       wait(NULL);
    }
    wait(NULL);
}
```

Output:

- 6*6 Magic Square with magic sum = 111.

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ g++ -o out magicSqr.cpp
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
6
Magic Square is
35  1  6 26 19 24
 3 32  7 21 23 25
31  9  2 22 27 20
 8 28 33 17 10 15
30  5 34 12 14 16
 4 36 29 13 18 11
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

- 14 * 14 Magic Square with magic sum = 1379

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$ ./out
Enter the order of
14
Magic Square is
177 186 195   1 10 19 28 128 137 146 99 108 68 77
185 194 154   9 18 27 29 136 145 105 107 116 76 78
193 153 155 17 26 35 37 144 104 106 115 124 84 86
152 161 16 172 34 36 45 103 112 114 123 132 85 94
160 162 171 33 42 44  4 111 113 122 131 140 93 53
168 170 179 41 43  3 12 119 121 130 139 141 52 61
169 178 187 49  2 11 20 120 129 138 147 100 60 69
30 39 48 148 157 166 175 79 88 97 50 59 117 126
38 47  7 156 165 174 176 87 96 56 58 67 125 127
46  6  8 164 173 182 184 95 55 57 66 75 133 135
 5 14 163 25 181 183 192 54 63 65 74 83 134 143
13 15 24 180 189 191 151 62 64 73 82 91 142 102
21 23 32 188 190 150 159 70 72 81 90 92 101 110
22 31 40 196 149 158 167 71 80 89 98 51 109 118
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Week4$
```

The entire code is given below:

```
/*****************************************
* AUTHOR : AdheshR*
*****************************************/
#include <bits/stdc++.h>
#include <sys/types.h>
#include <unistd.h>
#include <inttypes.h>
#include <string.h>
#include <math.h>
#include <sys/wait.h>
using namespace std;
#define LL long long
#define MAX 100
#define MOD 1000000007
#define dd double

void generateOddSquare(int n,vector<vector <int > > &magicSqr,int row, int col, int
offset);
void generateDoublyEvenSqr(int n,vector<vector <int > > &magicSqr);
void generateSinglyEvenSqr(int n,vector<vector <int > > &magicSqr);

int main()
{
    int n;
    printf("Enter the order of \n");
    scanf("%d",&n);

    vector<vector <int > > magicSqr(n,vector<int > (n,0));

    if(n%2 == 1)
       generateOddSquare(n,magicSqr,0,0,0);
    else if(n%4 == 0)
       generateDoublyEvenSqr(n,magicSqr);
    else if(n%4 == 2)
       generateSinglyEvenSqr(n,magicSqr);

    printf("Magic Square is\n");
    for(int i=0;i<n;++i)
    {
       for(int j=0;j<n;++j)
```

```
            printf("%2d ",magicSqr[i][j]);
        puts("");
    }
}

void generateOddSquare(int n, vector<vector <int > > &magicSqr,int row, int col, int
offset)
{
    // Let us generate the magic square of odd order n*n using vfork() - here let's split
the process in two
    pid_t pid = vfork();
    if(pid == 0)
    {
        // In the child process, lets fill from 1 to floor(n^2/2) in the matrix
        int i = row + 0,j = col + ceil(n/2);
        int num = 1 + offset,target = (n*n)/2 + offset;
        while(num <= target)
        {
            magicSqr[i][j] = num;
            ++num;

            // Check for boundary or cell-fill condition
            if(magicSqr[(((i-1)%n + n)%n + row)][(j+1)%n + col] == 0)
            {
                i = ((i-1)%n + n)%n + row;
                j = (j+1)%n + col;
            }
            else
                i = (i + 1)%n + row;
        }
        exit(0);
    }
    else if(pid > 0)
    {
        // In the parent process, let us fill from n^2/2 + 1 to n^2.
        int i = ceil(n/2) + row,j = ceil(n/2) + col;
        int num = (n*n)/2 + 1 + offset,target = n*n + offset;
        while(num <= target)
        {
            magicSqr[i][j] = num;
            ++num;
```

```
            // Check for boundary or cell-fill condition
            if(magicSqr[(((i-1)%n + n)%n) + row][(j+1)%n + col] == 0)
            {
                  i = ((i-1)%n + n)%n + row;
                  j = (j+1)%n + col;
            }
            else
                  i = (i + 1)%n + row;
      }
   }
   else
      perror("Process could not be created.");

   wait(NULL);
}

void generateDoublyEvenSqr(int n,vector<vector <int > > &magicSqr)
{
   int i,j;
   // First let us fill the matrix
   for(i=0;i<n;++i)
      for(j=0;j<n;++j)
            magicSqr[i][j] = (n*i) + j + 1;

   pid_t pid = vfork();
   if(pid == 0)
   {
      pid_t pid_0 = vfork();
      if(pid_0 == 0)
      {
            // Bottom-Right
            for (i = 3 * n/4; i < n; i++)
                  for ( j = 3 * n/4; j < n; j++)
                        magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];

            // Center of Matrix
            for (i = n/4; i < 3 * n/4; i++)
                  for (j = n/4; j < 3 * n/4; j++)
                        magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];

            exit(0);
      }
```

```
        else if(pid_0 > 0)
        {
              // Bottom Left
              for (i = 3 * n/4; i < n; i++)
                    for (j = 0; j < n/4; j++)
                   magicSqr[i][j] = (n*n+1) - magicSqr[i][j];
        }
        wait(NULL);
        exit(0);
    }
    else if(pid > 0)
    {
        pid_t pid_1 = vfork();
        if(pid_1 == 0)
        {
              // Top-Right Corner
              for(i=0;i<n/4;++i)
                    for(j=3*(n/4);j<n;++j)
                          magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];
              exit(0);
        }
        else if(pid_1 > 0)
        {
              // Top-Left Corner
              for(i=0;i<n/4;++i)
                    for(j=0;j<n/4;++j)
                          magicSqr[i][j] = (n*n + 1) - magicSqr[i][j];
        }
        wait(NULL);
    }
    else
        perror("Process could not be created.");
    wait(NULL);
}

void generateSinglyEvenSqr(int n,vector<vector <int > > &magicSqr)
{
    // Divide the matrix into four sub matrices of size
    int size = n/2;
    // Do each quarter generation in a process
    pid_t pid[6];
    pid[0] = vfork();
```

```
    if(pid[0] == 0)
    {
       pid[1] = vfork();
       if(pid[1] == 0)
       {
             generateOddSquare(size,magicSqr,size,size,size*size);
             exit(0);
       }
       else
             generateOddSquare(size,magicSqr,size,0,3*(size*size));
       wait(NULL);
       exit(0);
    }
    else
    {
       pid[2] = vfork();
       if(pid[2] == 0)
       {
             generateOddSquare(size,magicSqr,0,size,2*(size*size));
             exit(0);
       }
       else
             generateOddSquare(size,magicSqr,0,0,0);
       wait(NULL);
    }
    wait(NULL);

    // Now let us do the swaps necessary - in four processes
    int k = (n-2)/4;

    pid[3] = vfork();
    if(pid[3] == 0)
    {
       pid[4] = vfork();
       if(pid[4] == 0)
       {
             // kth column - all but middle
             for(int i=0;i<size;++i)
                   if(i!=size/2)
                         swap(magicSqr[i][k-1],magicSqr[i+size][k-1]);
             exit(0);
       }
```

```
        else
        {
                // Swap middle Element of k+1 th col
                swap(magicSqr[size/2][k],magicSqr[size/2 + size][k]);
        }
        wait(NULL);
        exit(0);
    }
    else
    {
        pid[5] = vfork();
        if(pid[5] == 0)
        {
                // Last (k-1) columns
                for(int i=n-k+1;i<n;++i)
                        for(int j=0;j<size;++j)
                                swap(magicSqr[j][i],magicSqr[j+size][i]);
                exit(0);
        }
        else
        {
                // First (k-1) columns
                for(int i=0;i<k-1;++i)
                        for(int j=0;j<size;++j)
                                swap(magicSqr[j][i],magicSqr[j+size][i]);
        }
        wait(NULL);
    }
    wait(NULL);
}
```