
Operating System Assignment-3

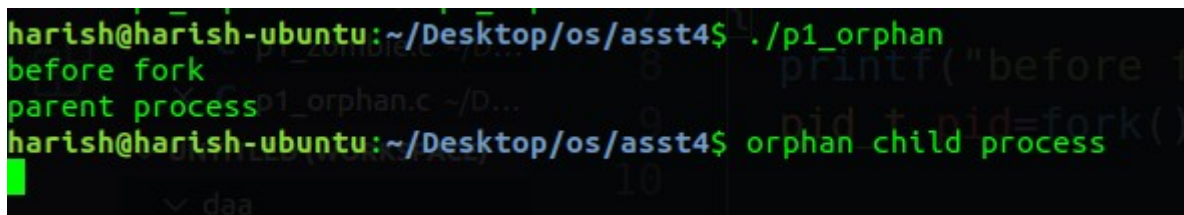
(1) Test drive a C program that creates Orphan and Zombie Processes

Orphan process:

Here the child is made to sleep until parent process ends.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
int main()
{
pid_t pid=fork();
printf("before fork\n");
if(pid>0)
{
printf("parent process\n");
}
else
{
sleep(20);
printf("orphan child process\n");
}
}
```

Output:



```
harish@harish-ubuntu:~/Desktop/os/asst4$ ./p1_orphan
before fork
parent process
harish@harish-ubuntu:~/Desktop/os/asst4$ orphan child process
```

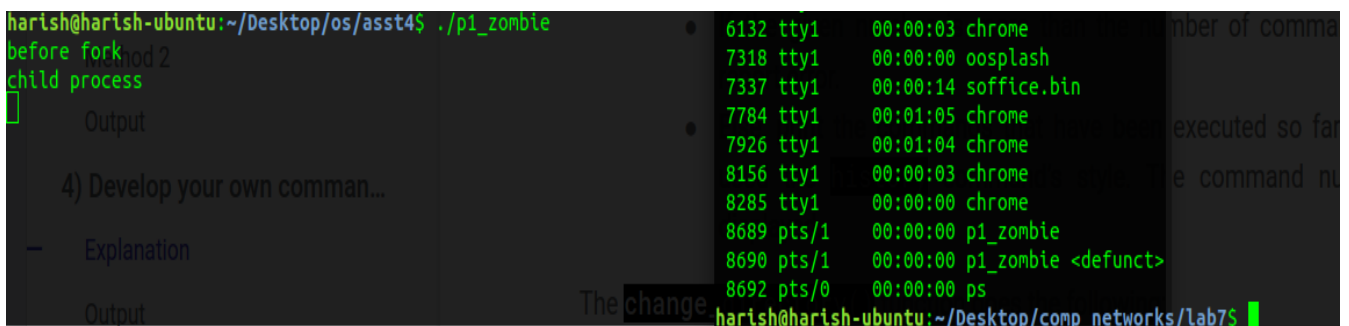
Zombie process:

Here the parent process sleeps until child completes.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdlib.h>
int main()
{
printf("before fork\n");
pid_t pid=fork();
if(pid==0)
{
printf("child process\n");
exit(0);
}
else
{
sleep(20);
printf("parent process\n");
}
}
```

Output:

while parent process is still sleeping, the child has completed. Hence in the “ps -a” output the child process is shown as <defunct>. Hence a zombie process is created



```
harish@harish-ubuntu:~/Desktop/os/asst4$ ./p1_zombie
before fork
child process
Output
4) Develop your own command...
Explanation
Output
The change
6132 tty1 00:00:03 chrome
7318 tty1 00:00:00 oosplash
7337 tty1 00:00:14 soffice.bin
7784 tty1 00:01:05 chrome
7926 tty1 00:01:04 chrome
8156 tty1 00:00:03 chrome
8285 tty1 00:00:00 chrome
8689 pts/1 00:00:00 p1_zombie
8690 pts/1 00:00:00 p1_zombie <defunct>
8692 pts/0 00:00:00 ps
harish@harish-ubuntu:~/Desktop/comp_networks/Lab7$
```

(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]

Approach:

Both quick and merge sort are done as follows:

- Input is got from user
- Shared memory array is declared and all elements are copied into the array
- On every divide of the array, a fork is called and child takes care of each of the divided half.
- Recursively during conquer, the array is sorted and the child terminates, for which the parent waits
- After all elements are sorted recursively output is printed.

Merge sort:

```
#include <stdio.h>
#include <stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
int* create_shm_array(int size)
{
    key_t key = IPC_PRIVATE;

    int shm_id;
    size_t shm_size=size*sizeof(int);
    if((shm_id=shmget(key, shm_size, IPC_CREAT | 0666))== -1)
    {
        perror("shmget");
        exit(1);
    }
}
```

```

}

int *shm_arr;
if((shm_arr=shmat(shm_id,NULL,0))==(int *)-1)
{
perror("shmat");
exit(1);
}

return shm_arr;
}

void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
/* 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[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // 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++;
}
}
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
if (l < r) {
// Same as (l+r)/2, but avoids overflow for
// large l and h
int m = l + (r - l) / 2;
// Sort first and second halves
pid_t qid, pid=fork();
if(pid==0)
{
mergeSort(arr, l, m);
exit(0);
}
if(pid>0)
{
qid=fork();
if(qid==0)
{
mergeSort(arr, m + 1, r);

```

```

exit(0);
}
}
int status;
waitpid(pid,&status,0);
waitpid(qid,&status,0);
merge(arr, l, m, r);
}
}
/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
int i;
for (i = 0; i < size; i++)
printf("%d ", A[i]);
printf("\n");
}
/* Driver program to test above functions */
int main()
{
int arr_size;
printf("enter the value of n\n");
scanf("%d",&arr_size);
int a[20];
int *arr = create_shm_array(arr_size);
for(int i=0;i<arr_size;++i)
scanf("%d",&a[i]);
//int a[]={ 14, 11, 13, 15, 6, 7 };
for(int i=0;i<6;++i)
arr[i]=a[i];
//int arr_size = sizeof(a) / sizeof(a[0]);
printf("Given array is \n");
printArray(arr, arr_size);
mergeSort(arr, 0, arr_size - 1);
printf("\nSorted array is \n");
printArray(arr, arr_size);

```

```
return 0;
}
```

Quick sort:

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

```
int* create_shm_array(int size)
{
    key_t key = IPC_PRIVATE;
```

```
    int shm_id;
    size_t shm_size=size*sizeof(int);
    if((shm_id=shmget(key,shm_size,IPC_CREAT | 0666))== -1)
    {
        perror("shmget");
        exit(1);
    }
```

```
    int *shm_arr;
    if((shm_arr=shmat(shm_id,NULL,0))==(int *)-1)
    {
        perror("shmat");
        exit(1);
    }
```

```
    return shm_arr;
}
```

```
void swap(int* a, int* b)
{
```

```

int temp = *a;
*a = *b;
*b = temp;
}

int partition (int arr[], int low, int high)
{
int pivot = arr[high];
int i = (low - 1);
for (int j = low; j <= high- 1; j++)
{
if (arr[j] < pivot)
{ i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i+1],&arr[high]);
return i+1;
}

void quicksort(int arr[], int low, int high,int k)
{
if (high>low)
{
int q = partition(arr, low, high);
pid_t qid,pid=fork();
if(pid==0)
{
quicksort(arr, low, q-1,k);
exit(0);
}
else if(pid>0)
{
qid=fork();
if(qid==0)
{
quicksort(arr, q+1, high,k);
exit(0);
}
}
}
}

```



```

}
}
int status;
waitpid(pid,&status,0);
waitpid(qid,&status,0);
}
}

int main()
{
int n;
int k;

printf("enter the value of n\n");
scanf("%d",&n);

int *a=create_shm_array(n);
for(int i=0;i<n;++i)
scanf("%d",&a[i]);

//printf("\nenter k:\n");
//scanf("%d",&k);
k=n;
quicksort(a,0,n-1,k);

printf("\nSorted array:\n");

for(int i=0;i<k;++i)
printf("%d ",a[i]);
printf("\n");
return 0;
}

```

Output:

Merge sort:

```
harish@harish-ubuntu:~/Desktop/os/asst4$ ./p2
enter the value of n
5=0;i<6;++i)
1 5 3 2 7
Given array is
1 5 3 2 7
Sorted array is
1 2 3 5 7
harish@harish-ubuntu:~/Desktop/os/asst4$
```

Quick sort:

```
harish@harish-ubuntu:~/Desktop/os/asst4$ ./p2_quick
enter the value of n
8
9 5 2 6 3 1 0 55
Sorted array:
0 1 2 3 5 6 9 55
harish@harish-ubuntu:~/Desktop/os/asst4$
```

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

Code:

Using shared memory for count variable so that all the forked child can access count variable. Parent waits till child completes then prints the count.

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

```
int main()
```

```

{

// create a shared memory integer variable
// to count the total no of fork calls
key_t key = IPC_PRIVATE;
int shm_id;
size_t shm_size=sizeof(int);
if((shm_id=shmget(key,shm_size,IPC_CREAT | 0666))== -1)
{
perror("shmget");
exit(1);
}
int *shm_count;
if((shm_count=shmat(shm_id,NULL,0))==(int *)-1)
{
perror("shmat");
exit(1);
}
*shm_count=1; // initally 1 fork is done

pid_t pid=fork();

if(pid==-1)
{
printf("fork failed!\n");
exit(1);
}
else if (pid==0) // child block
{
while(1)
{
pid=fork(); // fork repeatedly until fork returns -1
if(pid==-1)
break;
else if(pid==0) // increment count by 1 in each child
{
*shm_count=*shm_count+1;

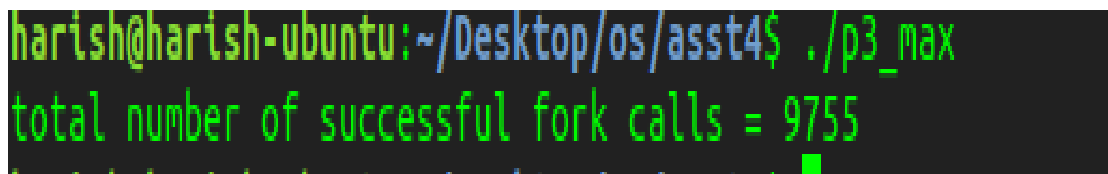
```

```

exit(0);
}
}
exit(0);
}
else
{
wait(NULL); // wait for the child to finish all fork calls
printf("total number of successful fork calls = %d\n", *shm_count);
}
}

```

Output:



```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./p3_max
total number of successful fork calls = 9755

```

(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;

Approach:

The command to be executed is received from user and passed to getcommand function.

getcommand()- A function to split arguments from the input string

A stack is maintained into which the command is pushed-**Push()**. On entering command for history, with an '!' sign the display function is called and based on input number the stack elements are displayed.- **display()**

Other commands are executed using execvp.

Exit – if exit is entered as input then the command prompt function exits

Until then the prompt gets input from user using while(1) loop.

Code: Please refer to commandpmt.c file uploaded.

Output:

```
harish@harish-ubuntu:~/Desktop/os/asst4$ ./commandpmt
-----
MY COMMAND PROMPT
-----
my-cmd-pmt:/home/harish/Desktop/os/asst4@ls
a.out      hist      histogram.c  p1_orphan  p1_zombie.c  p2_quick    p3.c      p3_max2.c  p6_matrix.c
commandpmt hist.c     input.txt   p1_orphan.c p2            p2_quick.c  p3_max    p3_max.c   p7
commandpmt.c histogram ip.txt      p1_zombie   p2.c         p3          p3_max2   p6_matrix  p7.c

my-cmd-pmt:/home/harish/Desktop/os/asst4@ls -l
total 236
-rwxr-xr-x 1 harish harish 13032 Oct  2 18:27 a.out
-rwxr-xr-x 1 harish harish 13120 Oct  4 16:42 commandpmt
-rw-rw-r-- 1 harish harish  3290 Oct  4 16:42 commandpmt.c
-rwxr-xr-x 1 harish harish  8400 Oct  2 11:48 hist
-rw-rw-r-- 1 harish harish   470 Oct  2 11:48 hist.c
-rwxrwxr-x 1 harish harish 12192 Oct  3 08:01 histogram
-rw-rw-r-- 1 harish harish  1059 Oct  2 17:56 histogram.c
-rw-r--r-- 1 harish harish  1475 Aug 18 11:09 input.txt
-rw-r--r-- 1 harish harish    24 Oct  2 17:50 ip.txt
-rwxr-xr-x 1 harish harish  8392 Oct  4 10:37 p1_orphan
-rw-rw-r-- 1 harish harish   271 Oct  4 10:37 p1_orphan.c
-rwxr-xr-x 1 harish harish  8432 Oct  4 10:49 p1_zombie
-rw-rw-r-- 1 harish harish   274 Oct  4 10:39 p1_zombie.c
-rwxr-xr-x 1 harish harish 12984 Oct  4 16:20 p2
-rw-rw-r-- 1 harish harish  3049 Oct  4 16:20 p2.c
-rwxr-xr-x 1 harish harish 12992 Oct  4 16:22 p2_quick
-rw-rw-r-- 1 harish harish  1653 Oct  4 16:22 p2_quick.c
```

```
-rw-rw-r-- 1 harish harish  1653 Oct  4 16:22 p2_quick.c
-rwxr-xr-x 1 harish harish  8296 Sep 20 10:09 p3
-rw-rw-r-- 1 harish harish   184 Sep 20 10:08 p3.c
-rwxr-xr-x 1 harish harish  8600 Oct  3 18:40 p3_max
-rwxr-xr-x 1 harish harish  8480 Oct  3 18:45 p3_max2
-rw-rw-r-- 1 harish harish   627 Oct  3 18:44 p3_max2.c
-rw-rw-r-- 1 harish harish  1026 Oct  3 18:40 p3_max.c
-rwxr-xr-x 1 harish harish 12800 Oct  4 08:53 p6_matrix
-rw-rw-r-- 1 harish harish  1130 Oct  4 08:53 p6_matrix.c
-rwxr-xr-x 1 harish harish 12624 Sep 21 22:11 p7
-rw-rw-r-- 1 harish harish  1350 Sep 21 22:11 p7.c
```

```
my-cmd-pmt:/home/harish/Desktop/os/asst4@pwd
/home/harish/Desktop/os/asst4
```

```
my-cmd-pmt:/home/harish/Desktop/os/asst4@!3
1.!3
2.pwd
3.ls -l
```

```
my-cmd-pmt:/home/harish/Desktop/os/asst4@factor 25
25: 5 5
```

```
my-cmd-pmt:/home/harish/Desktop/os/asst4@!6
1.!6
2.factor 25
3.!3
4.pwd
5.ls -l
6.ls
```

```
my-cmd-pmt:/home/harish/Desktop/os/asst4@exit
harish@harish-ubuntu:~/Desktop/os/asst4$
```

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

Approach:

Printable characters have ASCII value from 32 to 126. For every character, a fork is made and the child process counts the occurrence of that particular character and saves in the shared memory.

Once all child process complete, the parent process prints the count for all the characters that occurred at least once.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include<sys/wait.h>
#include<string.h>
int* create_shm_array(int size)
{
    key_t key = IPC_PRIVATE;

    int shm_id;
    size_t shm_size=size*sizeof(int);
    if((shm_id=shmget(key, shm_size, IPC_CREAT | 0666))== -1)
    {
        perror("shmget");
        exit(1);
    }

    int *shm_arr;
    if((shm_arr=shmat(shm_id, NULL, 0))==(int *)-1)
    {
        perror("shmat");
    }
}
```

```

exit(1);
}

return shm_arr;
}

int main()
{
int *shm_arr=create_shm_array(95);
for(int i=0;i<95;++i)
{
pid_t pid=fork();
if(pid==0)
{
FILE *ptr=fopen("ip.txt","r");
shm_arr[i]=0;
char c;
while((c=fgetc(ptr))!=EOF)
{
if(c==i+32)
{
shm_arr[i]++;
}
}
fclose(ptr);
exit(0);
}
if(pid>0)
wait(NULL);
}
for(int i=0; i<95; ++i)
{
if(shm_arr[i]>0)
{
if(i!=94)
putchar(i+32);

```

```

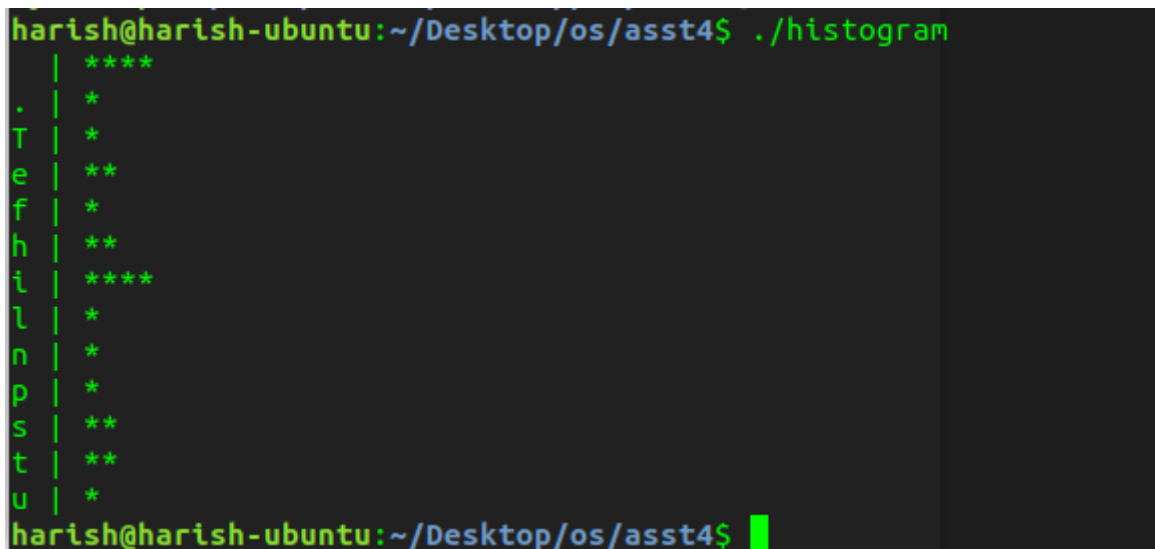
else
printf("others");
printf(" | ");
for(int j=0; j < shm_arr[i]; ++j)
putchar('*');

putchar('\n');
}
}
return 0;
}

```

Output:

(for file ip.txt)



```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./histogram
. | ****
T | *
e | **
f | *
h | **
i | *****
l | *
n | *
p | *
s | **
t | **
u | *
harish@harish-ubuntu:~/Desktop/os/asst4$

```

(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.

Approach:

For a generalised case, if the matrices of size $m \times n$ and $p \times q$ are multiplied (given $n=p$) then $m \times q$ processes are created and each process handles the result of $m \times q$ elements of resultant matrix.

This is achieved by having a shared memory of 2d array called `res` and for every result entry calculation a fork is done.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int* create_shm_array(int size)
{
    key_t key = IPC_PRIVATE;

    int shm_id;
    size_t shm_size=size*sizeof(int);
    if((shm_id=shmget(key, shm_size, IPC_CREAT | 0666))== -1)
    {
        perror("shmget");
        exit(1);
    }

    int *shm_arr;
    if((shm_arr=shmat(shm_id, NULL, 0))==(int *) -1)
    {
        perror("shmat");
        exit(1);
    }

    return shm_arr;
}
```

```

int main()
{
int m=2,n=3,p=3,q=2;
if(n!=p)
printf("matrix mult cant be done");

int *res[m];
for(int i=0;i<m;++i)
res[i]=create_shm_array(q);
int a[][3]={2,3,4},{5,6,1};
int b[][2]={7,8},{9,0},{1,2};

int i, j, k;
pid_t pid=fork();
if(pid==0)
{
for (i = 0; i < m; i++)
{
for (j = 0; j < q; j++)
{
res[i][j] = 0;
for (k = 0; k < p; k++)
{
pid=fork();
if(pid==0)
{
res[i][j] += a[i][k]*b[k][j];
exit(0);
}
}
}
}
exit(0);
}
else if(pid>0)

```

```

{
wait(NULL);
for(int ti=0;i<m;++i)
{
for(int j=0;j<q;++j)
printf("%d ",res[i][j]);
printf("\n");
}
}
return 0;
}

```

Output:

(input given in code as

```

int a[][3]={ {2,3,4},{5,6,1}};
int b[][2]={ {7,8},{9,0},{1,2}};)

```

```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./p6_matrix
45 24
90 42
harish@harish-ubuntu:~/Desktop/os/asst4$ █

```

(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.

Approach:

The following checks are done parallely,

- Diagonal elements sum
- Cross diagonal elements sum
- Each row sum carried out by forks.Each child checking one row sum.
- Each coloumn sum carried out by forks.Each child checking one coloumn sum.
- To check for uniqueness of each element, a separate process sorts the elements of 2d array and checks if any 2 adjacent numbers are same.

Each of the above check is handled by different processes using multiprocessing setup.

Code:

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

int* create_shm_array(int size)
{
    key_t key = IPC_PRIVATE;

    int shm_id;
    size_t shm_size=size*sizeof(int);
    if((shm_id=shmget(key, shm_size, IPC_CREAT | 0666))== -1)
    {
        perror("shmget");
        exit(1);
    }
}
```

```

}

int *shm_arr;
if((shm_arr=shmat(shm_id,NULL,0))==(int *)-1)
{
perror("shmat");
exit(1);
}

return shm_arr;
}

int main()
{
int n;
int mat[10][10];
printf("enter n\n");
scanf("%d",&n);
for(int i=0;i<n;++i)
for(int j=0;j<n;++j)
scanf("%d",&mat[i][j]);
int sum=(n*(n*n+1))/2;
int total_sum=sum*n;
int *flag=create_shm_array(1);
flag[0]=1;
pid_t qid,rid,pid=vfork();//A
if(pid==0)
{
pid_t id=fork();//B
if(id==0){
int sumd1=0;
for(int i=0;i<n;++i)
sumd1+=mat[i][i];

if(sumd1!=sum)
flag[0]=0;

```

```

exit(0);
}
else
{
int sumd2=0;
for(int i=0;i<n;++i)
sumd2+=mat[i][n-1-i];

if(sumd2!=sum)
flag[0]=0;
exit(0);
}
}
else
{
qid=vfork();//C
if(qid==0)
{
for (int i = 0; i < n; i++) {
pid_t id=fork();//D
if(id==0){
int rowSum = 0;
for (int j = 0; j < n; j++)
rowSum += mat[i][j];
// check if every row sum is
// equal to prime diagonal sum
if (rowSum != sum)
flag[0]=0;
exit(0);
}
}
exit(0);
}
else
{
rid=vfork();//E
if(rid==0)

```

```

{
pid_t fid=fork();//F
if(fid==0){
for (int i = 0; i < n; i++) {
pid_t id=fork();//G
if(id==0){
int colSum = 0;
for (int j = 0; j < n; j++)
colSum += mat[j][i];
// check if every column sum is
// equal to prime diagonal sum
if (sum != colSum)
flag[0]=0;
exit(0);
}
}
}
else
{
int arr[100],k=0;
for(int i=0;i<n;++i)
for(int j=0;j<n;++j)
arr[k++]=mat[i][j];
bubbleSort(arr,n*n);
for(int i=0;i<n*n-1;++i)
if(arr[i]==arr[i+1])
flag[0]=0;
exit(0);
}
exit(0);
}
}
}
int status;
waitpid(pid,&status,0);
waitpid(qid,&status,0);
waitpid(rid,&status,0);

```

```

if(flag[0]==1)
printf("matrix is magic square\n");
else
printf("matrix is not magic square\n");
return 0;

}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./p7
enter n
3
2 7 6 9 5 1 4 3 8
matrix is magic square

```

```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./p7
enter n
3
2 7 6 9 5 1 4 3 7
matrix is not magic square

```

```

harish@harish-ubuntu:~/Desktop/os/asst4$ ./p7
enter n
3
5 5 5 5 5 5 5 5 5
matrix is not magic square
harish@harish-ubuntu:~/Desktop/os/asst4$ █

```

(8) Extend the above to also support magic square generation

To generate magic square the following procedure is followed:

In any magic square, the first number i.e. 1 is stored at position $(n/2, n-1)$. Let this position be (i,j) . The next number is stored at position $(i-1, j+1)$ where we can consider each row & column as circular array i.e. they wrap around.

Three conditions hold:

1. The position of next number is calculated by decrementing row number of the

previous number by 1, and incrementing the column number of the previous number by 1. At any time, if the calculated row position becomes -1, it will wrap around to $n-1$. Similarly, if the calculated column position becomes n , it will wrap around to 0.

1. Similarly, if the calculated column position becomes n , it will wrap around to 0.
2. If the magic square already contains a number at the calculated position, calculated column position will be decremented by 2, and calculated row position will be incremented by 1.
3. If the calculated row position is -1 & calculated column position is n , the new position would be: $(0, n-2)$.

As we can see every step depends on previous iteration for its execution, which makes it only serially executable. This generation cannot be done parallelly as every step mandates its previous step to get over.
