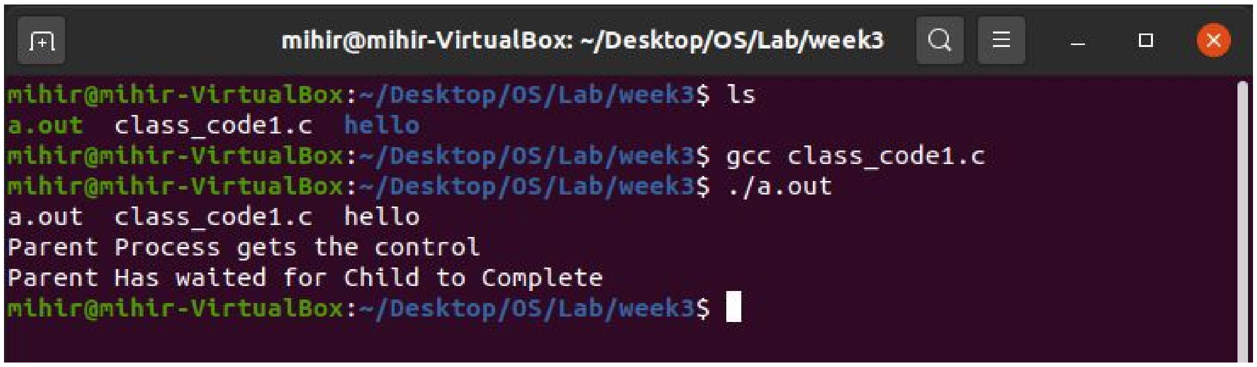
Assignment 3 - Fork

(1) Test Drive all the examples discussed so far in the class for the usage of wait, exec call variants.

Example 1.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ()
{
   pid_t pid;            // this is to use the pid data type – relevant headers above
   pid = fork();
   if (pid == 0)
         execl("/bin/ls", "ls", NULL);         // child image is now ls command
   else
   {
         wait (NULL);          // parent waits for the child to complete execution.
         printf("Parent Process gets the control \n");
         printf ("Parent Has waited for Child to Complete\n");
   }
   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ls
a.out  class_code1.c  hello
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc class_code1.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
a.out  class_code1.c  hello
Parent Process gets the control
Parent Has waited for Child to Complete
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$
```

Reasoning:

At forking point, main() gets split into two processes, main() and C1 (where C1 is the child of main()). Within the parent block (pid > 0), wait(NULL) has been called which means that it will wait for any one of the children to complete its execution and then only it will terminate.

Since, we only have one child, so the parent process waits for the only child process to complete its execution.

Now, in the child block, execl function has been called. The **execl() function** replaces the current child process image with a new process image specified by path. Here, the path is **/bin/ls** and the executable is also **ls**. So, the child process adapts the image of the ls command. When the child process has finished executing, it sends a message to the parent process so that the parent process can finish its execution.

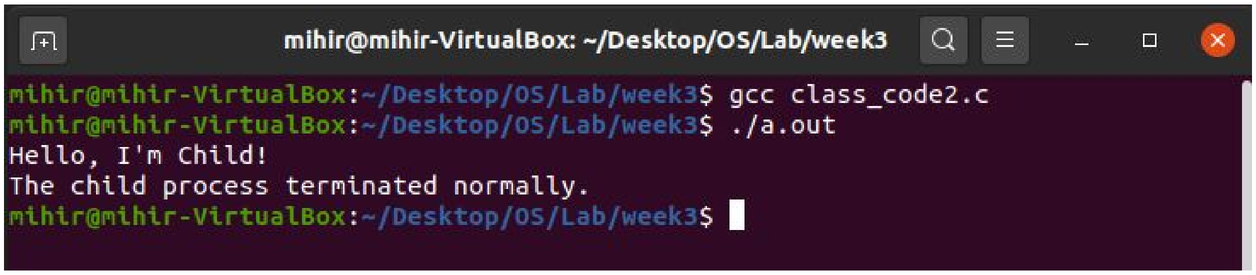
So, first the **ls** command is executed then the **two printf statements inside the parent block** are executed.

Example 2.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main ()
{
   pid_t ChildPid;
   ChildPid = fork();

   if (ChildPid == 0)
   {
         printf("Hello, I'm Child!\n");
         exit(0);
```

```
    }

    else if (ChildPid < 0)
    {
            printf("Error: Fork failed!\n");
    }

    else
    {
            int ReturnStatus;
            waitpid(ChildPid, &ReturnStatus, 0);

            if (ReturnStatus == 0)
                    printf("The child process terminated normally.\n");

            else if (ReturnStatus == 1)
                    printf("The child process terminated with an error!\n");
    }

    return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc class_code2.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Hello, I'm Child!
The child process terminated normally.
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ 
```

Reasoning:

At forking point, main() gets split into two processes, main() and C1 (where C1 is the child of main()). Within the parent block (pid > 0), waitpid() has been called which means that it will wait for the child to complete its execution and then only it will terminate.

So, the control gets transferred to the **child block** where it executes the **only printf statement** and exits.

Now, the control is back with the parent block since the child has died.
Inside the parent block **waitpid() function** was called whose first parameter was ChildPid which has a value >0 since we are inside the parent block. So, waitpid() has been called with the first parameter >0.

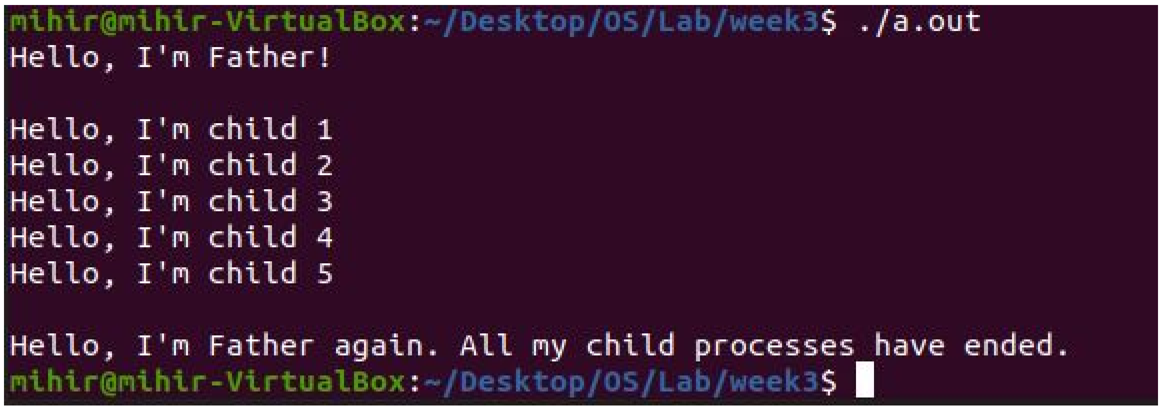
If the child process terminates normally, a value of 0 will be returned to the second argument (&ReturnStatus) of the waitpid() call else a value of 1 will be returned and the corresponding block of the ReturnStatus will be executed and the message will be displayed.

Example 3.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main ()
{
   pid_t ChildPid, wpid;
   int status = 0;
   int n = 5;

   printf("Hello, I'm Father!\n\n");

   for (int id = 0; id < n; id++)
   {
         if ((ChildPid = fork()) == 0)
         {
                  printf("Hello, I'm child %d\n", id+1);
                  exit(0);
```

```
        }
    }

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

    printf("\nHello, I'm Father again. All my child processes have ended.\n");

    return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Hello, I'm Father!

Hello, I'm child 1
Hello, I'm child 2
Hello, I'm child 3
Hello, I'm child 4
Hello, I'm child 5

Hello, I'm Father again. All my child processes have ended.
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$
```

Reasoning:

The **first printf statement will be executed first** as it is before any fork call.

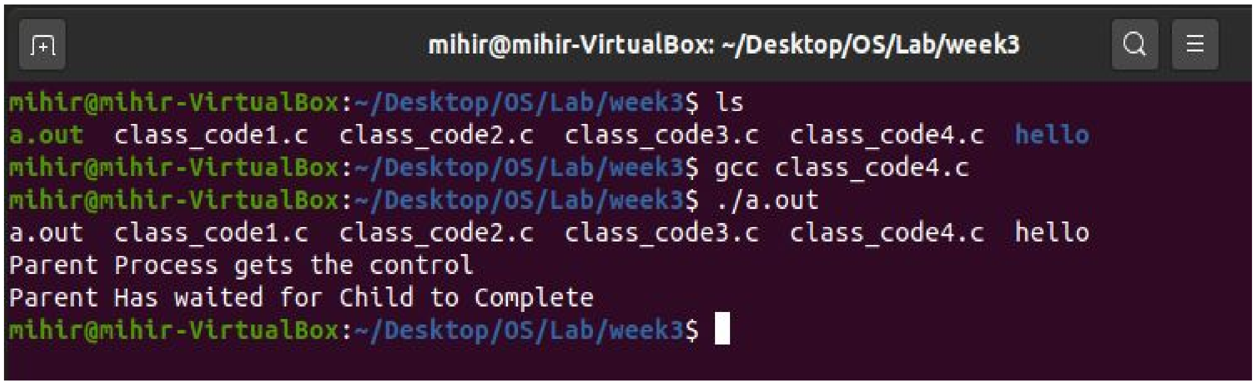
One thing to note is that every time forking happens within the parent block only. So, the number of children created will be equal to the no of times the loop runs (n = 5) and hence **each child will execute the printf statement inside its block**.

Another thing to note is that the **wait() function** has been called again and again within a while loop until the pid returned by the wait() is >0. When all the children have finished their execution then only the value returned by wait() will not be >0 and hence the while loop will terminate causing the **last printf statement inside the parent block to get executed**.

<u>Example 4.</u>

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ()
{
   pid_t pid;          // this is to use the pid data type – relevant headers above
   pid = fork();

   if (pid == 0)
         execlp("ls", "ls", NULL);          // child image is now ls command

   else
   {
         wait (NULL);          // parent waits for the child to complete execution.
         printf("Parent Process gets the control \n");
         printf("Parent Has waited for Child to Complete\n");
   }
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week3
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ls
a.out  class_code1.c  class_code2.c  class_code3.c  class_code4.c  hello
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc class_code4.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
a.out  class_code1.c  class_code2.c  class_code3.c  class_code4.c  hello
Parent Process gets the control
Parent Has waited for Child to Complete
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ 
```

Reasoning:

The reasoning is exactly the same as in **Example 1.** The only difference here is that instead of execl(), the **execlp() function** has been called. The advantage of using the execlp() function over the execl() function is that we don’t have to specify the path of the command. The system does that for us.

For the following questions you are free to decide the responsibility of parent / child processes. As mentioned in the class u r allowed to use vfork / file based approach to avoid the data sharing issues which we will later address using pipes in later classes to follow.

(2) (a) Odd and Even series generation for n terms using Parent Child relationship (say odd is the duty of the parent and even series as that of child)

Logic:

<u>**Approach 1**</u> **-** Create two .c files, odd.c and even.c and create their corresponding executables. Assuming that the child takes up the task of printing even numbers till n and that child will be executed first, ask for the value of n from the user in even.c and store that value in a file. In odd.c, read the stored value of n from the file and supply to the code for printing odd numbers till n.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for even.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for odd.c.

<u>**Approach 2**</u> **-** Capture the value returned by fork in a variable named pid. In the child block (when pid == 0), write the code for printing even series and and in the parent block (when pid > 0) write the code for printing odd series.

Codes:

For approach 1:

<u>even.c</u> -

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

void even_series(int n)
{
   printf("\nEven series till %d:\n", n);
   for (int i = 0; i <= n; i+=2)
         printf("%d ", i);

   printf("\n\n");
}

int main()
{
   int n;

   printf("Enter the value of n: ");
   scanf("%d", &n);
   even_series(n);

   // Store the value of n in a file
   FILE *fptr;
   fptr = fopen("n.txt", "w");
   putw(n, fptr);
   fclose(fptr);

   return 0;
}
```

<u>odd.c</u> -

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

```
#include <stdlib.h>

void odd_series(int n)
{
   printf("Odd series till %d:\n", n);
   for (int i = 1; i <= n; i+=2)
         printf("%d ", i);

   printf("\n");
}

int main()
{
   int n;

   // Read the value of n from the file
   FILE *fptr;
   fptr = fopen("n.txt", "r");
   n = getw(fptr);
   fclose(fptr);

   odd_series(n);

   return 0;
}
```

code1.c (The driver code) -

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

int main ()
{
   pid_t pid;
   pid = fork();
   if (pid == 0)
         execl("/home/mihir/Desktop/OS/Lab/week3/even", "./even", NULL);
```

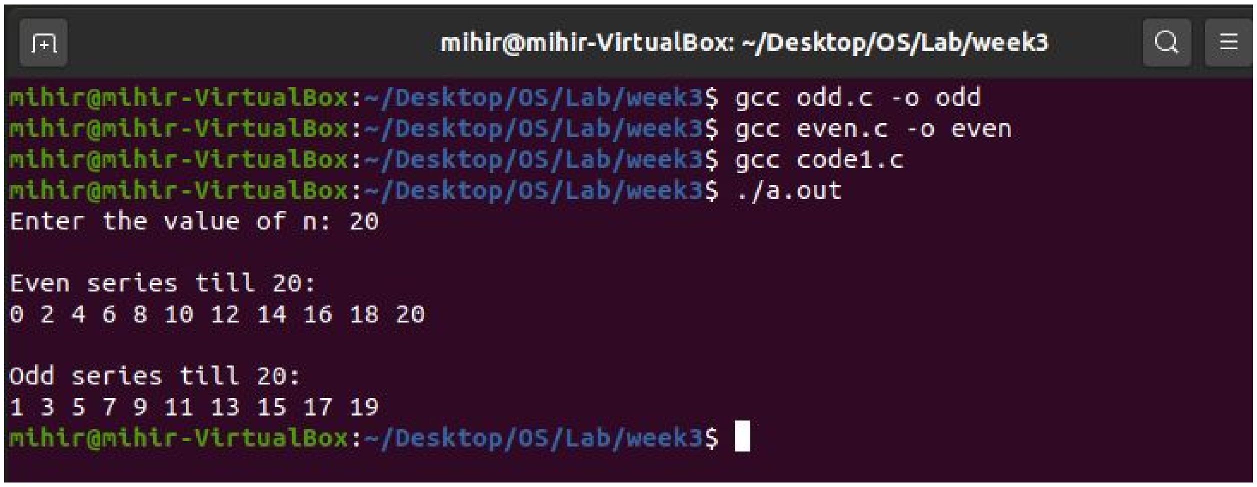
```
    else
    {
        wait (NULL);
        execl("/home/mihir/Desktop/OS/Lab/week3/odd", "./odd", NULL);
    }

    return 0;
}
```

For approach 2:

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

int main ()
{
    pid_t pid;          // this is to use the pid data type – relevant headers above
    int n;

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

    pid = fork();

    if (pid == 0)
    {
        printf("\nEven series till %d:\n", n);
        for (int i = 0; i <= n; i+=2)
            printf("%d ", i);

        printf("\n\n");
    }

    else
    {
            wait (NULL);                    // parent waits for the child to complete execution.
            printf("Odd series till %d:\n", n);
            for (int i = 1; i <= n; i+=2)
                    printf("%d ", i);

            printf("\n");
    }

    return 0;
}
```

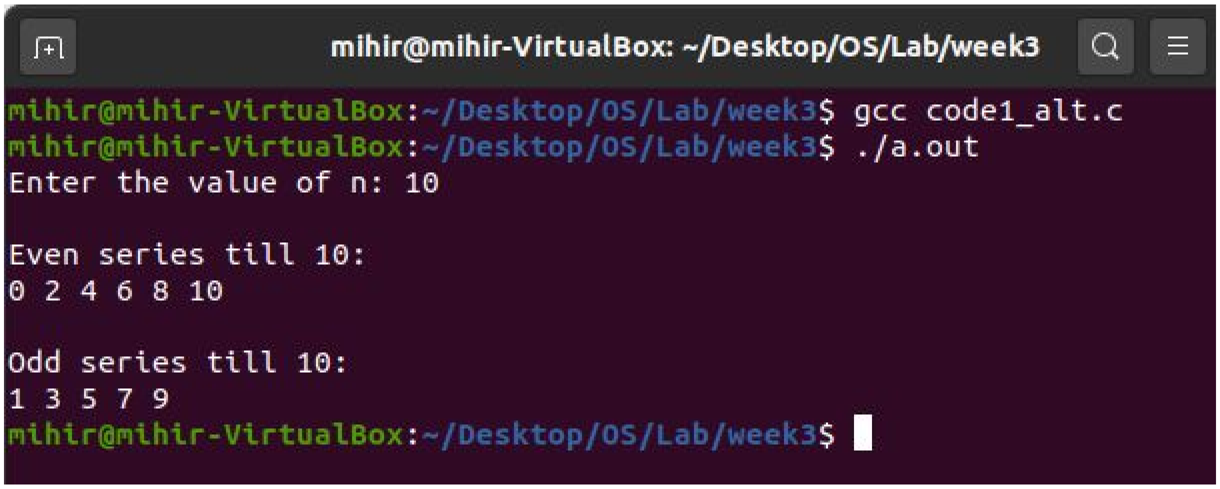
Output ss:

For approach 1:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc odd.c -o odd
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc even.c -o even
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc code1.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Enter the value of n: 20

Even series till 20:
0 2 4 6 8 10 12 14 16 18 20

Odd series till 20:
1 3 5 7 9 11 13 15 17 19
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$
```

For approach 2:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc code1_alt.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Enter the value of n: 10

Even series till 10:
0 2 4 6 8 10

Odd series till 10:
1 3 5 7 9
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$
```

(2) (b) Given a series of n numbers (u can assume natural numbers till n) generate the sum of odd terms in the parent and the sum of even terms in the child process.

Logic:

Approach 1 - Create two .c files, odd_sum.c and even_sum.c and create their corresponding executables. Assuming that the child takes up the task of calculating the sum of even terms till n and that child will be executed first, ask for the value of n from the user in even.c and store that value in a file. In odd.c, read the stored value of n from the file and supply to the code for calculating the sum of odd terms till n.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for even_sum.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for odd_sum.c.

Approach 2 - Capture the value returned by fork in a variable named pid. In the child block (when pid == 0), write the code for calculating the sum of even terms and in the parent block (when pid > 0) write the code for calculating the sum of odd terms.

Codes:

For approach 1 -

even-sum.c -
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int even_sum(int n)
{
   printf("\nSum of even terms till %d: ", n);
   int sum = 0;
   for (int i = 2; i <= n; i+=2)
         sum += i;
```

```
    return sum;
}

int main()
{
    int n;

    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("%d\n", even_sum(n));

    // Store the value of n in a file
    FILE *fptr;
    fptr = fopen("n.txt", "w");
    putw(n, fptr);
    fclose(fptr);

    return 0;
}
```

odd-sum.c -

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

int odd_sum(int n)
{
    printf("Sum of odd terms till %d: ", n);
    int sum = 0;
    for (int i = 1; i <= n; i+=2)
            sum += i;

    return sum;
}

int main()
{
    int n;
    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);
    fclose(fptr);

    printf("%d\n", odd_sum(n));

    return 0;
}
```

code2.c (The driver code)-

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

int main ()
{
    pid_t pid;          // this is to use the pid data type – relevant headers above

    pid = fork();
    if (pid == 0)
        execl("/home/mihir/Desktop/OS/Lab/week3/even", "./even", NULL);

    else
    {
        wait (NULL);            // parent waits for the child to complete execution.
        execl("/home/mihir/Desktop/OS/Lab/week3/odd", "./odd", NULL);
    }

    return 0;
}
```

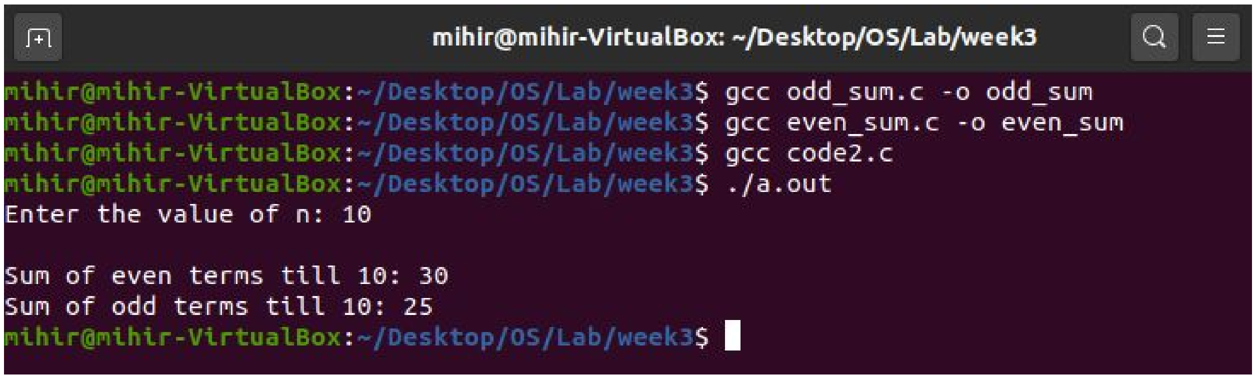
For approach 2 -

```
#include <stdio.h>
#include <sys/types.h>
```

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

int main ()
{
    pid_t pid;              // this is to use the pid data type – relevant headers above
    int n;

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

    pid = fork();

    if (pid == 0)
    {
        printf("\nSum of even numbers till %d: ", n);
        int sum = 0;
        for (int i = 2; i <= n; i+=2)
            sum += i;

        printf("%d", sum);
        printf("\n");
    }

    else
    {
        wait (NULL);                // parent waits for the child to complete execution.
        printf("Sum of odd numbers till %d: ", n);
        int sum = 0;
        for (int i = 1; i <= n; i+=2)
            sum += i;

        printf("%d", sum);
        printf("\n");
    }

    return 0;
}
```

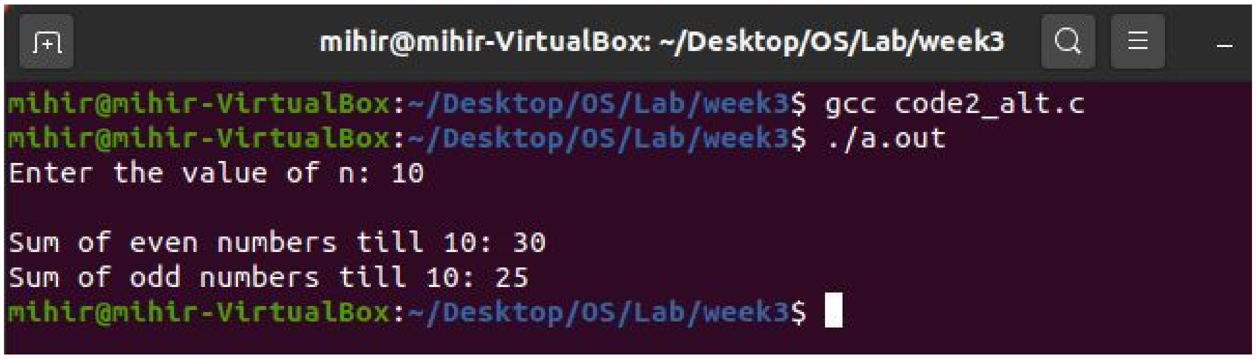
Output ss:

For approach 1 -



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc odd_sum.c -o odd_sum
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc even_sum.c -o even_sum
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc code2.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Enter the value of n: 10

Sum of even terms till 10: 30
Sum of odd terms till 10: 25
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ 
```

For approach 2 -



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc code2_alt.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Enter the value of n: 10

Sum of even numbers till 10: 30
Sum of odd numbers till 10: 25
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ 
```

(3) Armstrong number generation within a range. The digit extraction, cubing can be the responsibility of the child while the checking for sum == no can happen in the parent and the output list in the parent.

Logic:

Approach 1 - Create two .c files, processing.c and check.c and create their corresponding executables. Assuming that the child takes up the task of calculating the cube of digits and summing them up and that child will be executed first, ask for the value of n from the user in processing.c and store that value in a file. Also, store the value of the sums calculated for each number from 1 till n in processing.c in the same

file. In check.c, read the stored value of n and the value of the sums one by one from the file and supply it to the code for checking whether the values are equal or not.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for processing.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for check.c.

<u>**Approach 2**</u> **-** Inside a for loop, capture the value returned by **vfork()** each time in a variable named pid. In the child block (when pid == 0), write the code for calculating the sum of cubes of digits of n and in the parent block (when pid > 0) write the code for checking whether the original number is equal to the sum or not.
Note that, here we're using **vfork()**. This is because we need to supply the value of the sum updated by the child to the parent block. If we use normal fork then the variables have separate memory spaces while in vfork() the variables have a shared memory space.

Codes:

For approach 1 -

<u>**processing.c**</u> **-**
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
   int n, sum;

   FILE *fptr;
   fptr = fopen("n.txt", "w");

   printf("Enter the value of n (range): ");
   scanf("%d", &n);
   putw(n, fptr);
```

```
    for (int i = 1; i <= n; i++)
    {
        int d, sum = 0, k = i;
        while (k > 0)
        {
            d = k % 10;
            sum += d*d*d;
            k /= 10;
        }

        putw(sum, fptr);
    }

    fclose(fptr);

    return 0;
}
```

check.c -

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

int main()
{
    int n, sum;

    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);

    printf("The following are the Armstrong numbers between 1 to %d:\n", n);

    for (int i = 1; i <= n; i++)
    {
        sum = getw(fptr);
        if (i == sum)
            printf("%d\n", i);
    }

    fclose(fptr);

    return 0;
}
```

code3.c (The driver code)

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

int main ()
{
    pid_t pid;              // this is to use the pid data type – relevant headers above

    pid = fork();
    if (pid == 0)
        execl("/home/mihir/Desktop/OS/Lab/week3/processing", "./processing", NULL);

    else
    {
        wait (NULL);                    // parent waits for the child to complete execution.
        execl("/home/mihir/Desktop/OS/Lab/week3/check", "./check", NULL);
    }

    return 0;
}
```

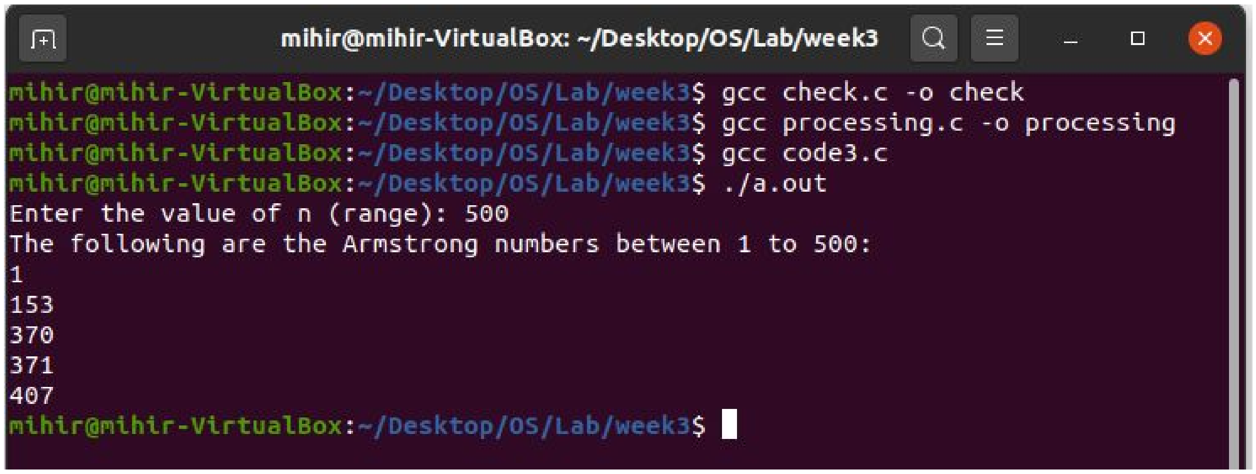
For approach 2 -

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

```
int main ()
{
   pid_t pid;              // this is to use the pid data type – relevant headers above
   int n;

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

   int num = n;

   for (int i = 1; i <= n; i++)
   {
         pid = vfork();
         int sum = 0;

         if (pid == 0)
         {
                  int d, k = i;
                  while (k > 0)
                  {
                           d = k % 10;
                           sum += d*d*d;
                           k /= 10;
                  }
         }

         if (pid > 0)
         {
                  wait (NULL);            // parent waits for the child to complete execution.
                  if (i == sum)
                          printf("%d\n", i);
         }
   }

   return 0;
}
```

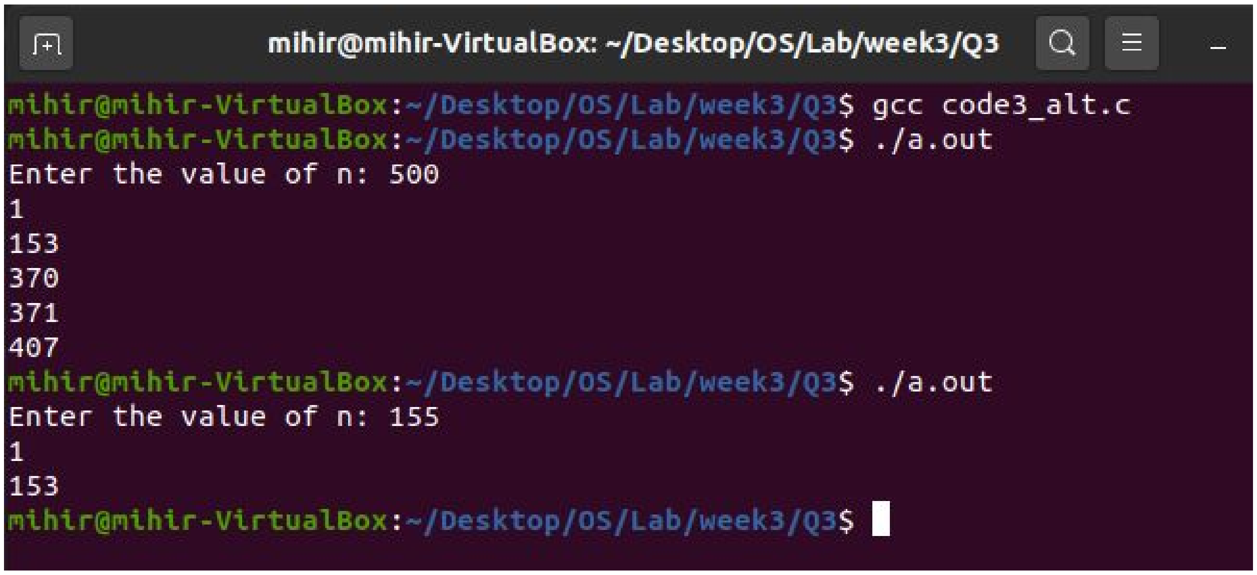
Output ss:

For approach 1 -



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc check.c -o check
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc processing.c -o processing
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ gcc code3.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$ ./a.out
Enter the value of n (range): 500
The following are the Armstrong numbers between 1 to 500:
1
153
370
371
407
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3$
```

For approach 2 -



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q3$ gcc code3_alt.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q3$ ./a.out
Enter the value of n: 500
1
153
370
371
407
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q3$ ./a.out
Enter the value of n: 155
1
153
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q3$
```

(4) Fibonacci Series AND Prime parent child relationship (say parent does fib Number generation using series and child does prime series)

Logic: Create two .c files, fib.c and prime.c and create their corresponding executables. Assuming that the child takes up the task of generating the fibonacci numbers till n and

that child will be executed first, ask for the value of n from the user in fib.c and store that value in a file. In prime.c, read the stored value of n and supply it to the code for generating the prime number series till n.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for processing.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for check.c.

Codes:

<u>fib.c</u> -

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

void fib(int n)
{
   int f1 = 0, f2 = 1;

   if (n == 0)
         printf(" ");

   else
   {
         for (int i = 1; i <= n; ++i)
        {
                 printf("%d ", f1);
                 int f3 = f1 + f2;
                 f1 = f2;
                 f2 = f3;
        }
   }
}

int main()
{
```

```c
    int n;

    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("\nFibonacci series till %d:\n", n);
    fib(n);

    // Store the value of n in a file
    FILE *fptr;
    fptr = fopen("n.txt", "w");
    putw(n, fptr);
    fclose(fptr);

    return 0;
}
```

prime.c -

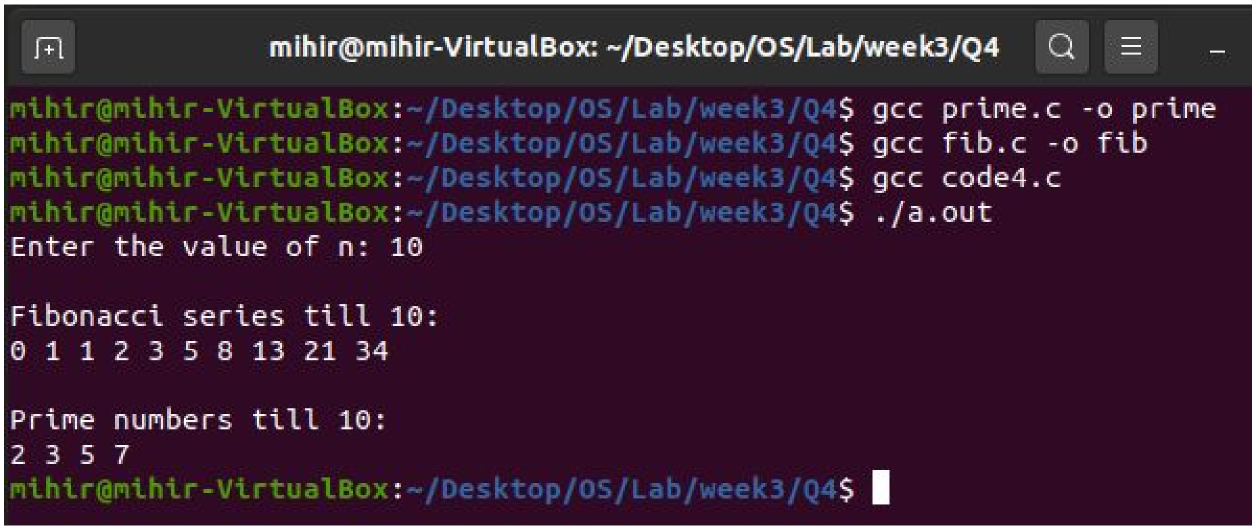
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int prime(int n)
{
    int c=0;
    for(int i=1;i<=n;i++)
    {
        if(n%i==0)
            c++;
    }

    if(c==2)
        return 1;

    else
        return 0;
}
```

```
int main()
{
    int n;

    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);
    fclose(fptr);

    printf("\n\nPrime numbers till %d:\n", n);

    for (int i = 1; i <= n; i++)
    {
        if (prime(i))
            printf("%d ", i);
    }

    printf("\n");

    return 0;
}
```

<u>code4.c</u> -

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

int main ()
{
    pid_t pid;          // this is to use the pid data type – relevant headers above

    pid = fork();
    if (pid == 0)
        execl("/home/mihir/Desktop/OS/Lab/week3/Q4/fib", "./fib", NULL);

    else
    {
        wait (NULL);          // parent waits for the child to complete execution.
        execl("/home/mihir/Desktop/OS/Lab/week3/Q4/prime", "./prime", NULL);
    }

    return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week3/Q4
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q4$ gcc prime.c -o prime
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q4$ gcc fib.c -o fib
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q4$ gcc code4.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q4$ ./a.out
Enter the value of n: 10

Fibonacci series till 10:
0 1 1 2 3 5 8 13 21 34

Prime numbers till 10:
2 3 5 7
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q4$
```

(5) Ascending Order sort within Parent and Descending order sort (or vice versa) within the child process of an input array. (u can view as two different outputs –first entire array is asc order sorted in op and then the second part desc order output)

Logic: Create two .c files, ascending.c and descending.c and create their corresponding executables. Assuming that the child takes up the task of printing the ascending order sort and that child will be executed first, ask for the value of n from the user in asending.c and store that value in a file. Also, Ask the user to input the array elements and keep inserting them in the same file where n was inserted. In descending.c, read the stored value of n and the array elements supply it to the code for printing the descending order sort.

In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for processing.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for check.c.

Codes:

ascending.c -

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

int asc(int a[], int n)
{
   for (int i = 0; i < n; i++)
   {
        for (int j = 0; j < n; j++)
        {
                 if (a[j] > a[i])
                 {
                        int tmp = a[i];
                        a[i] = a[j];
                        a[j] = tmp;
                 }
        }
   }

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

 int main()
{
   int n, a[100];

   printf("Enter the size of the array: ");
```

```
    scanf("%d", &n);

    // Store the value of n in a file
    FILE *fptr;
    fptr = fopen("n.txt", "w");
    putw(n, fptr);

    printf("Enter the array elements: ");

    for (int i = 0; i < n; ++i)
    {
        scanf ("%d", &a[i]);
        putw(a[i], fptr);
    }

    printf("\nAscending order: \n");
    asc(a, n);
    printf("\n");

    fclose(fptr);

    return 0;
}
```

<u>descending.c</u> -

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

int desc(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (a[j] < a[i])
            {
                int tmp = a[i];
                a[i] = a[j];
```

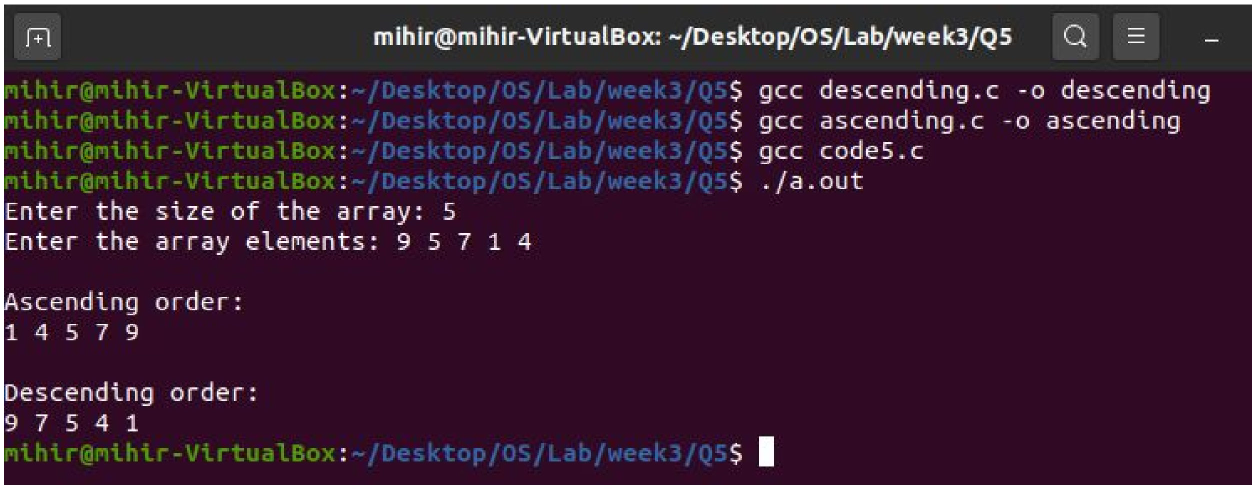
```
                        a[j] = tmp;
                    }
                }
            }

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

int main()
{
    int n, a[100];

    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);

    for (int i = 0; i < n; ++i)
    {
        a[i] = getw(fptr);
    }

    fclose(fptr);

    printf("\nDescending order:\n");
    desc(a, n);
    printf("\n");

    return 0;
}
```

code5.c -

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

int main ()
```

```
{
   pid_t pid;    // this is to use the pid data type – relevant headers above

   pid = fork();
   if (pid == 0)
         execl("/home/mihir/Desktop/OS/Lab/week3/Q5/ascending", "./ascending",
NULL);        // child image is now ./even

   else
   {
         wait (NULL);          // parent waits for the child to complete execution.
         execl("/home/mihir/Desktop/OS/Lab/week3/Q5/descending", "./descending",
NULL);        // parent image is now ./odd
   }

   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week3/Q5
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q5$ gcc descending.c -o descending
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q5$ gcc ascending.c -o ascending
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q5$ gcc code5.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q5$ ./a.out
Enter the size of the array: 5
Enter the array elements: 9 5 7 1 4

Ascending order:
1 4 5 7 9

Descending order:
9 7 5 4 1
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q5$
```

(6) Given an input array use parent child relationship to sort the first half of array in ascending order and the trailing half in descending order (parent / child is ur choice)

Logic: Create two .c files, asc.c and desc.c and create their corresponding executables. Assuming that the child takes up the task of sorting the first half of the array in ascending order and that child will be executed first, ask for the value of n from the user in asc.c and store that value in a file. Also, Ask the user to input the array elements and keep inserting them in the same file where n was inserted. In desc.c, read the stored value of n and the array elements supply it to the code for sorting the second half of the array in descending order.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for processing.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for check.c.

Codes:

asc.c -

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

int asc(int a[], int n)
{
   for (int i = 0; i < n/2; i++)
   {
        for (int j = 0; j < n/2; j++)
        {
                if (a[j] > a[i])
                {
                        int tmp = a[i];
                        a[i] = a[j];
                        a[j] = tmp;
                }
        }
```

```
    }

    for (int i = 0; i < n/2; i++)
            printf("%d ", a[i]);
 }

 int main()
{
    int n, a[100];

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

    // Store the value of n in a file
    FILE *fptr;
    fptr = fopen("n.txt", "w");
    putw(n, fptr);

    printf("Enter the array elements: ");

    for (int i = 0; i < n; ++i)
    {
            scanf ("%d", &a[i]);
            putw(a[i], fptr);
    }

    printf("\nFirst half in ascending order:\n");
    asc(a, n);
    printf("\n");

    fclose(fptr);

    return 0;
}
```

desc.c -

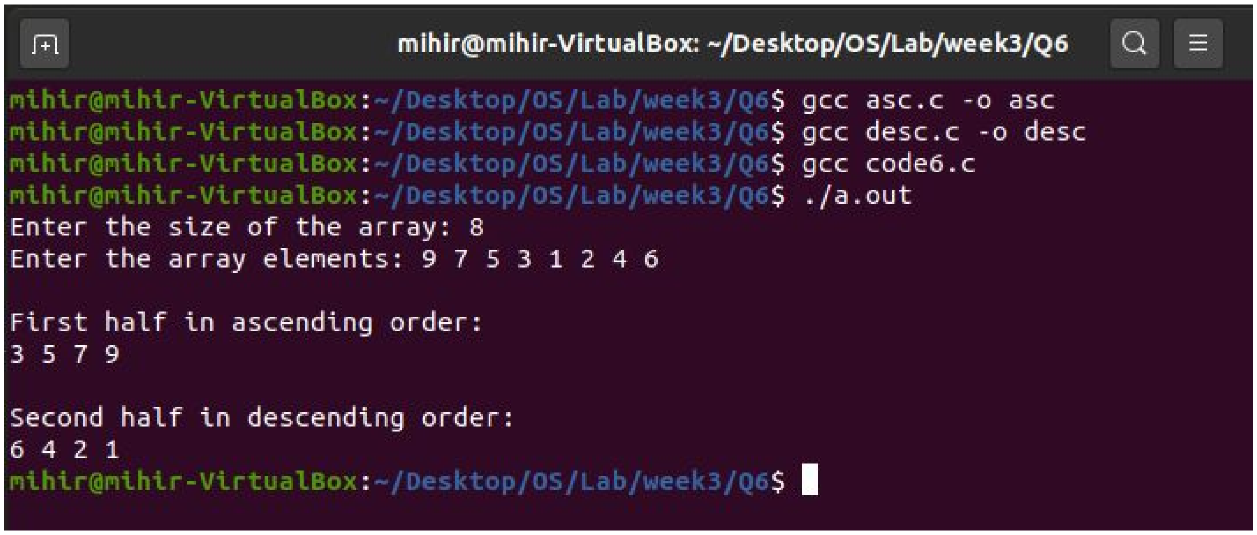
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
int desc(int a[], int n)
{
    for (int i = n/2; i < n; i++)
    {
           for (int j = n/2; j < n; j++)
           {
                       if (a[j] < a[i])
                       {
                                 int tmp = a[i];
                                 a[i] = a[j];
                                 a[j] = tmp;
                       }
           }
    }

    for (int i = n/2; i < n; i++)
            printf("%d ", a[i]);
 }
 int main()
{
    int n, a[100];

    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);

    for (int i = 0; i < n; ++i)
    {
            a[i] = getw(fptr);
    }
    fclose(fptr);
    printf("\nSecond half in descending order:\n");
    desc(a, n);
    printf("\n");

    return 0;
}
```

<u>code6.c</u> -

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ()
{
   pid_t pid;            // this is to use the pid data type – relevant headers above

   pid = fork();
   if (pid == 0)
         execl("/home/mihir/Desktop/OS/Lab/week3/Q6/asc", "./asc", NULL);

   else
   {
         wait (NULL); // parent waits for the child to complete execution.
         execl("/home/mihir/Desktop/OS/Lab/week3/Q6/desc", "./desc", NULL);
   }
   return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week3/Q6
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q6$ gcc asc.c -o asc
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q6$ gcc desc.c -o desc
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q6$ gcc code6.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q6$ ./a.out
Enter the size of the array: 8
Enter the array elements: 9 7 5 3 1 2 4 6

First half in ascending order:
3 5 7 9

Second half in descending order:
6 4 2 1
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q6$
```

(7) Implement a multiprocessing version of binary search where the parent searches for the key in the first half and subsequent splits while the child searches in the other half of the array. By default u can implement a search for the first occurrence and later extend to support multiple occurrence (duplicated elements search as well)

Logic: Create two .c files, bin_first.c and bin_second.c and create their corresponding executables. Assuming that the child takes up the task of searching the first half of the array and that child will be executed first, ask for the value of n from the user in asc.c and store that value in a file. Also, Ask the user to input the array elements and the key (the element to search) and keep inserting them in the same file where n was inserted. For the binary search, first sort the first half of the array and then apply the binary search.
To tackle multiple occurrences of an element, note the position at which the element is once found. Now, since the array is sorted, hence the other occurrences of the element will be at positions immediately before it or immediately after it i.e. if the element is found at position **i** the other occurrences of that element can only be at positions **i-1, i-2,...** or **i+1, i+2,... .**
In bin_second.c, read the stored value of n and the array elements and the key and supply it to the code for searching in the second half of the array.
In the driver code (where forking will take place) create two processes using fork, the child process and the parent process. Inside the child process (pid == 0), using **execl() function**, execute the binary file created for processing.c.
In the parent block, call **wait() function** to wait for the child process to finish its execution and then using the **execl() function,** execute the binary file created for check.c.

Codes:

<u>bin-first.c</u> -
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void bin_search(int a[], int n, int key)
{
   for (int i = 0; i < n/2; i++)
   {
        for (int j = 0; j < n/2; j++)
        {
                if (a[j] > a[i])
                {
                        int tmp = a[i];
                        a[i] = a[j];
                        a[j] = tmp;
                }
        }
}

int mid, beg = 0, end = n/2-1, flag = 0;
while(beg<=end)
{
        mid=(beg+end)/2;

        if(a[mid]==key)
        {
                flag += 1;
                break;
        }

        else if(a[mid]<key)
                beg=mid+1;

        else
                end=mid-1;
}

int i = mid, j = mid;
if (flag == 1)
{
        while (a[i] == a[i-1] && i >= 0)
        {
                flag += 1;
                i--;
        }

        while (a[j] == a[j+1] && j <= n/2 - 1)
```

```
            {
                    flag += 1;
                    j++;
            }

            printf("Element %d found %d times in the first half of the array\n", key, flag);
    }

    else
            printf("Element %d not found in the first half of the array\n", key);
 }

 int main()
{
    int n, a[100], key;

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

    // Store the value of n in a file
    FILE *fptr;
    fptr = fopen("n.txt", "w");
    putw(n, fptr);

    printf("Enter the array elements: ");

    for (int i = 0; i < n; ++i)
    {
            scanf ("%d", &a[i]);
            putw(a[i], fptr);
    }

    printf("Enter the element to search: ");
    scanf ("%d", &key);
    putw(key, fptr);

    bin_search(a, n, key);

    fclose(fptr);
    return 0;
}
```

<u>bin-second.c</u> -

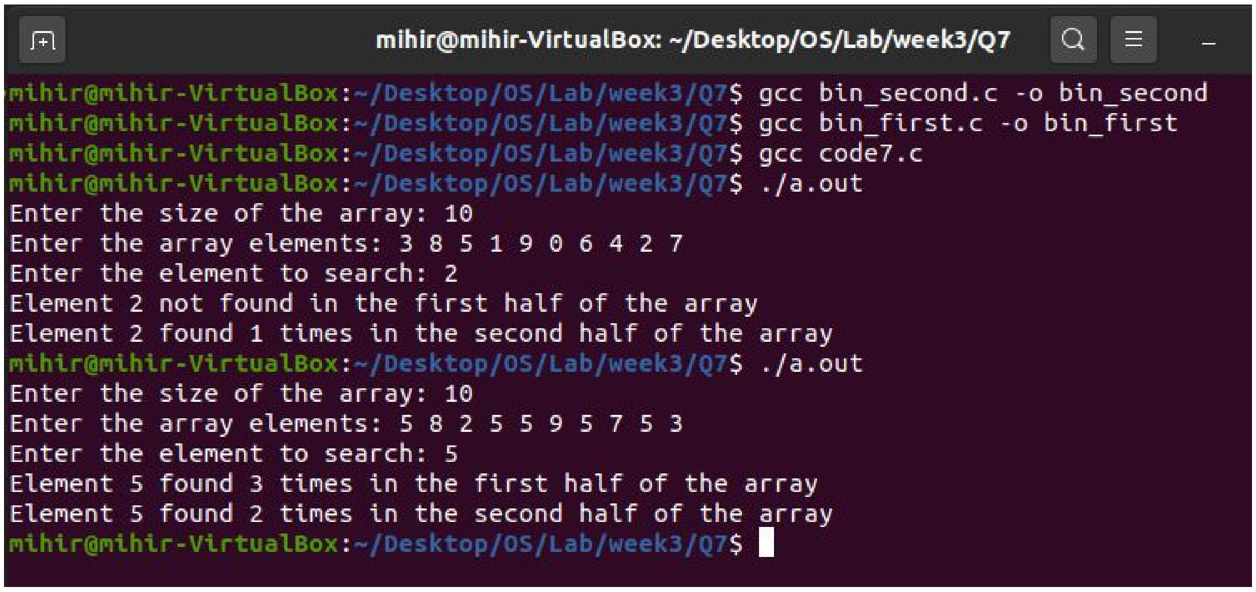
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void bin_search(int a[], int n, int key)
{
    for (int i = n/2; i < n; i++)
    {
        for (int j = n/2; j < n; j++)
        {
            if (a[j] > a[i])
            {
                int tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
        }
    }

    int mid, beg = n/2, end = n-1, flag = 0;
    while(beg<=end)
    {
        mid=(beg+end)/2;

        if(a[mid]==key)
        {
            flag += 1;
            break;
        }

        else if(a[mid]<key)
            beg=mid+1;

        else
```

```
                end=mid-1;
    }

    int i = mid, j = mid;
    if (flag == 1)
    {
        while (a[i] == a[i-1] && i >= n/2)
        {
            flag += 1;
            i--;
        }

        while (a[j] == a[j+1] && j <= n-1)
        {
            flag += 1;
            j++;
        }

        printf("Element %d found %d times in the second half of the array\n", key, flag);
    }

    else
        printf("Element %d not found in the second half of the array\n", key);
 }

int main()
{
    int n, a[100];

    // Read the value of n from the file
    FILE *fptr;
    fptr = fopen("n.txt", "r");
    n = getw(fptr);

    for (int i = 0; i < n; ++i)
    {
        a[i] = getw(fptr);
    }
    int key = getw(fptr);

    fclose(fptr);

    bin_search(a, n, key);

    return 0;
}
```

code7.c (The driver code) **-**

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

int main ()
{
    pid_t pid;              // this is to use the pid data type – relevant headers above

    pid = fork();
    if (pid == 0)
    {
            execl("/home/mihir/Desktop/OS/Lab/week3/Q7/bin_first", "./bin_first", NULL);
    }

    else
    {
            wait (NULL);
            execl("/home/mihir/Desktop/OS/Lab/week3/Q7/bin_second", "./bin_second",
NULL);
    }

    return 0;
}
```

Output ss:



```
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ gcc bin_second.c -o bin_second
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ gcc bin_first.c -o bin_first
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ gcc code7.c
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ ./a.out
Enter the size of the array: 10
Enter the array elements: 3 8 5 1 9 0 6 4 2 7
Enter the element to search: 2
Element 2 not found in the first half of the array
Element 2 found 1 times in the second half of the array
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ ./a.out
Enter the size of the array: 10
Enter the array elements: 5 8 2 5 5 9 5 7 5 3
Enter the element to search: 5
Element 5 found 3 times in the first half of the array
Element 5 found 2 times in the second half of the array
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q7$ 
```

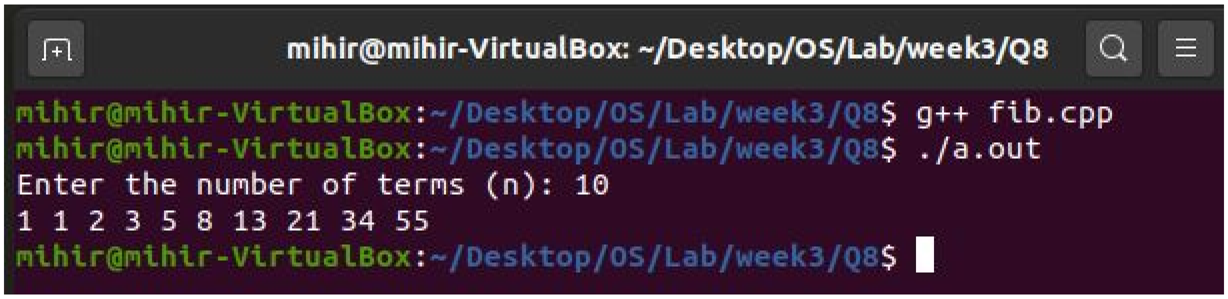
(8) * Non Mandatory [extra credits]
Read upon efficient ways of parallelizing the generation of Fibonacci series and apply the logic in a parent child relationship to contributes a faster version of fib series generation as opposed to sequential logic in (4)

Logic: The logic here is that the child calculates the first term fib(0) and the parent calculates the next term fib(1) and then the sum of these terms is returned by the parent. We use vfork() so that the term calculated by the child can be passed on to the parent for use. From then onwards, the one term is calculated by the child and one term by parent. So, parent and child make the process of generating fibonacci series efficient by generating it term by term.

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

```
#include <bits/stdc++.h>
using namespace std;

int fib (int n)
{

    if(n <= 2)
            return 1;

            else
            {
                        pid_t pid = vfork();

                        long int fib_0, fib_1;

                        if(pid == 0)
                        {
                                    fib_0 = fib(n-2);
                                    _exit(0);
                        }

                        else if(pid > 0)
                                    fib_1 = fib(n-1);

                        wait(NULL);

                        return fib_1 + fib_0;
   }
}

int main()
{
          int i, n;

          cout << "Enter the number of terms (n): ";
          cin >> n;

          for(int i = 1; i <= n; i++)
           cout << fib(i) << " ";
          cout << endl;

          return 0;

}
```

Output ss:



```
mihir@mihir-VirtualBox: ~/Desktop/OS/Lab/week3/Q8
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q8$ g++ fib.cpp
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q8$ ./a.out
Enter the number of terms (n): 10
1 1 2 3 5 8 13 21 34 55
mihir@mihir-VirtualBox:~/Desktop/OS/Lab/week3/Q8$
```