OPERATING SYSTEMS PRACTICE (COM301P)

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

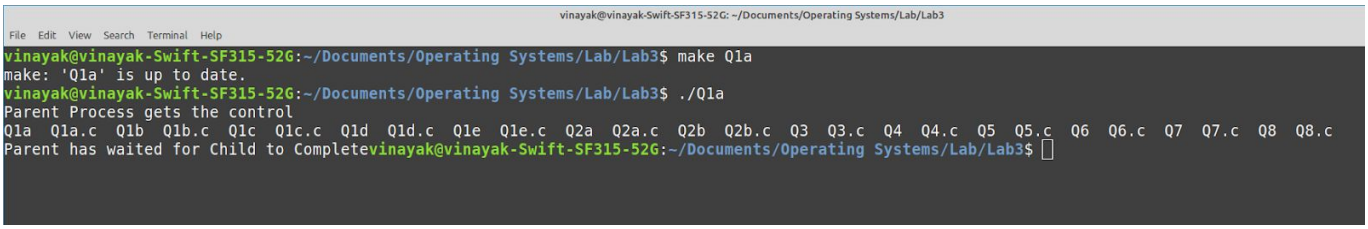
Assignment 3

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

Code: execl()

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

int main()
{
	pid_t pid;
	pid = fork();
	if(pid == 0) //child process
		execl("/bin/ls", "ls", NULL); //child image is now ls command
	else
	{
		printf("Parent Process gets the control \n");
		wait(NULL); //parent waits for the child to complete execution
		printf("Parent has waited for Child to Complete");
	}

	return 0;
}
```

Output:



Explanation:
Here the child function is executed so ‘ls’ is executed.

```
int execl(const char *path, const char *arg, ..., NULL);
```

- **path:** Path of the executable file.
- **arg:** Arguments to be passed on to run the function.

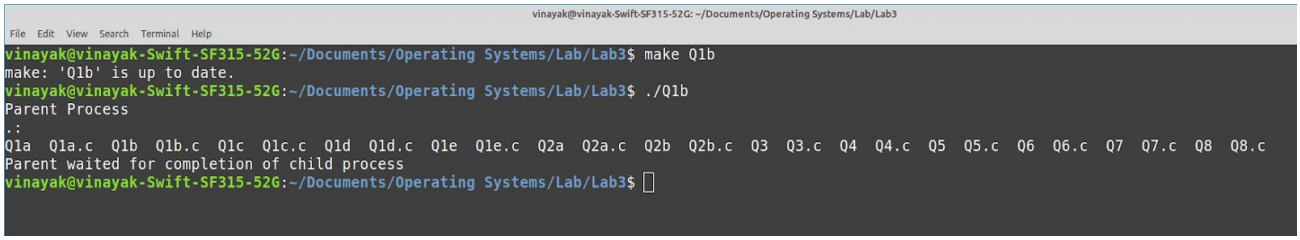
The parent process waits for the child to complete.

Code: execlp()

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

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

	if(pid < 0)
		printf("Fork failed \n");

	else if(pid == 0) //child process
		execlp("ls", "ls", "-LR", NULL);
```

```
        else
        {
                printf("Parent Process \n");
                wait(NULL); //parent waits for the child to complete execution
                printf("Parent waited for completion of child process \n");
        }

        return 0;
}
```

Output:



Explanation:
Here the child function is executed so ‘ls’ is executed.

```
int execlp(const char *file, const char *arg, ..., NULL );
```

- **file:** Executable file.
- **name:** Name of the command to be executed.
- **arg:** Arguments to be passed on to run the function.

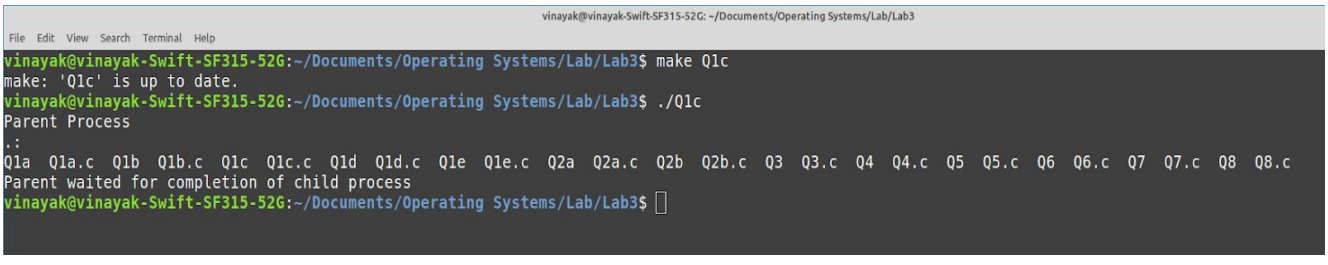
The parent process waits for the child to complete.

Code: execv()

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

```c
#include<unistd.h>

int main()
{
	char *argv[] = {"/bin/ls", "-LR", NULL};

	pid_t pid = fork();

	if(pid < 0)
		printf("Fork failed \n");

	else if(pid == 0) //child process
		execv(argv[0], argv);

	else
	{
		printf("Parent Process \n");
		wait(NULL); //parent waits for the child to complete execution
		printf("Parent waited for completion of child process \n");
	}

	return 0;
}
```

Output:



Explanation:

Here the child function is executed so ‘ls’ is executed.

```
int execv(const char *path, char *const argv[]);
```

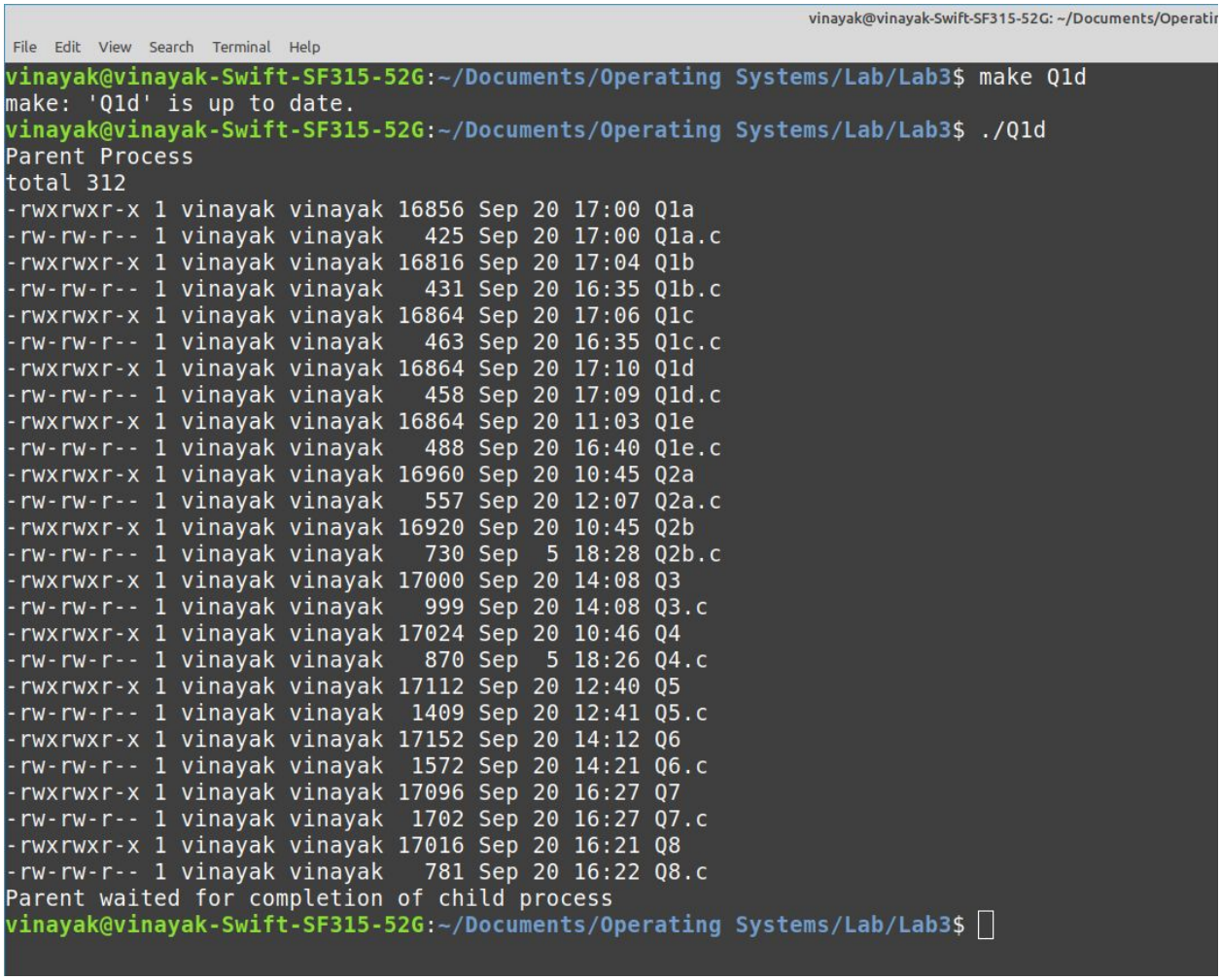
- **path:** Path of the executable file.
- **argv:** Arguments to be passed on to run the function.

The parent process waits for the child to complete.

Code: execvp()

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

int main()
{
	char *argv[] = {"ls", "-l", NULL};

	pid_t pid = fork();

	if(pid < 0)
		printf("Fork failed \n");

	else if(pid == 0) //child process
		execvp(argv[0], argv);

	else
	{
		printf("Parent Process \n");
		wait(NULL); //parent waits for the child to complete execution
		printf("Parent waited for completion of child process \n");
	}

	return 0;
}
```

Output:



Explanation:
Here the child function is executed so ‘ls’ is executed.

```
int execvp(const char *file, char *const argv[]);
```

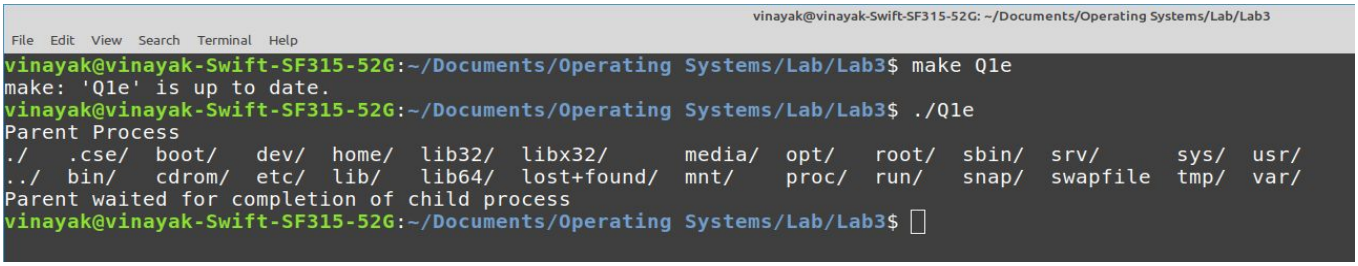
- **path:** Path of the executable file.
- **argv:** Arguments to be passed on to run the function.

The parent process waits for the child to complete.

Code: execve()

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

int main()
{
	char *args[] = {"ls", "-aF", "/", 0};
	char *env[] = {0};

	pid_t pid = fork();

	if(pid < 0)
		printf("Fork failed \n");

	else if(pid == 0) //child process
		execve("/bin/ls", args, env);

	else
	{
		printf("Parent Process \n");
		wait(NULL); //parent waits for the child to complete execution
		printf("Parent waited for completion of child process \n");
	}

	return 0;
}
```

Output:



Explanation:
Here the child function is executed so ‘ls’ is executed.

```
int execve(const char *file, char *const argv[], char *const envp[]);
```

- **file:** Executable file.
- **argv:** Arguments to be passed on to run the function.
- **envp:** Environment variables to the function.

The parent process waits for the child to complete.

(2) 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)

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

int main()
{
	pid_t pid;
	int i,n;

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

	pid = fork();

	if(pid == 0) //child block
	{
		printf("Child Process...\n");
		printf("First %d even numbers are : ", n);
		for(i=0; i<n; i++)
		{
			printf("%d  ",2*i);
```

```
            }
            printf("\n");
      }

      else
      {
            wait(NULL);
            printf("Parent Process...\n");
            printf("First %d odd numbers are : ", n);
            for(i=1; i<=n; i++)
            {
                  printf("%d  ",2*i - 1);
            }
            printf("\n");
      }

      return 0;
}
```

Output:

```
vinayak@vinayak-Swift-SF315-52G: ~/Docume
File  Edit  View  Search  Terminal  Help
vinayak@vinayak-Swift-SF315-52G:~/Documents/Operating Systems/Lab/Lab3$ make Q2a
make: 'Q2a' is up to date.
vinayak@vinayak-Swift-SF315-52G:~/Documents/Operating Systems/Lab/Lab3$ ./Q2a
Enter the vaue of n : 7
Child Process...
First 7 even numbers are : 0  2  4  6  8  10  12
Parent Process...
First 7 odd numbers are : 1  3  5  7  9  11  13
vinayak@vinayak-Swift-SF315-52G:~/Documents/Operating Systems/Lab/Lab3$ ./Q2a
Enter the vaue of n : 10
Child Process...
First 10 even numbers are : 0  2  4  6  8  10  12  14  16  18
Parent Process...
First 10 odd numbers are : 1  3  5  7  9  11  13  15  17  19
vinayak@vinayak-Swift-SF315-52G:~/Documents/Operating Systems/Lab/Lab3$
```

Explanation:

The child process prints the even numbers till then the parent process waits using wait() function.

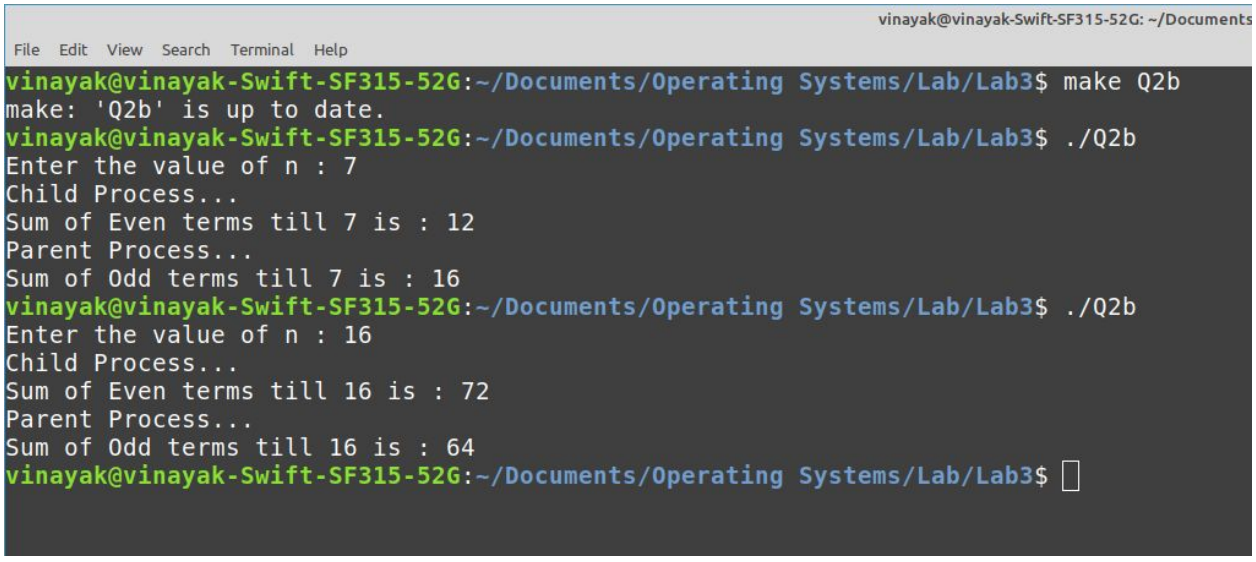
After the child process is completed, the parent process prints the odd numbers.

(b) given a series of n numbers (you 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.

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

int main()
{
	pid_t pid;
	int i,n, sum=0;

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

	pid = fork();

	if(pid == 0) //child block
	{
		printf("Child Process...\n");
		printf("Sum of Even terms till %d is : ",n);
		for(i=1; i<=n/2; i++)
		{
			sum = sum + 2*i;
		}
		printf("%d\n",sum);
	}

	else //parent block
	{
		wait(NULL);
		printf("Parent Process...\n");
		printf("Sum of Odd terms till %d is : ",n);
		if(n%2 == 0)
		{
```

```
                for(i=1; i<=n/2; i++)
                {
                        sum = sum + (2*i - 1);
                }
                printf("%d\n",sum);
        }
        else
        {
                for(i=1; i<=(n/2 + 1); i++)
                {
                        sum = sum + (2*i - 1);
                }
                printf("%d\n",sum);
        }
    }

    return 0;
}
```

Output:



Explanation:

The child process prints the sum of even indices till then the parent process waits using wait() function.

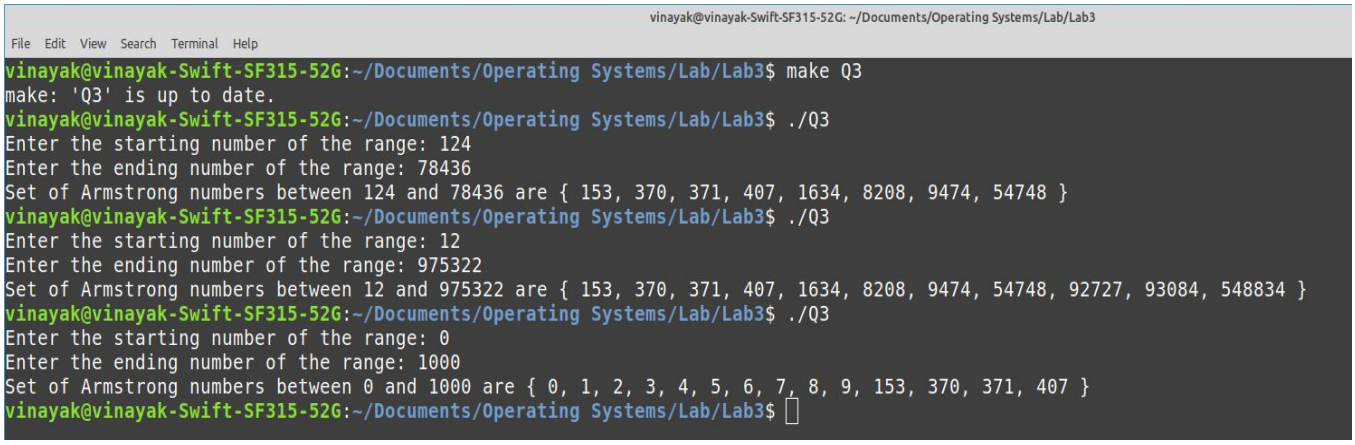
After the child process is completed, the parent process prints the sum of odd indices.

(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 child and the output list in the child.

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

int main()
{
	int start, end;
	printf("Enter the starting number of the range: ");
	scanf("%d", &start);
	printf("Enter the ending number of the range: ");
	scanf("%d", &end);

	int n = end - start + 1, temp, count = 0, digit, i;
	int arr[n];

	for(i = 0; i < n; i++)
		arr[i] = 0;

	pid_t pid = vfork();

	if(pid == 0) //child block
	{
		for(i = start; i < end + 1; i++)
		{
			temp = i;

			while(temp != 0)
			{
				temp /= 10;
				count++;
```

```
                }

                temp = i;

                while(temp != 0)
                {
                        digit = temp % 10;
                        temp /= 10;
                        arr[i - start] += pow(digit, count);
                }
                count = 0;
        }
        exit(0);
    }

    else //parent block
    {
        wait(NULL);
        printf("Set of Armstrong numbers between %d and %d are { ",
start, end);
        for(i = start; i < end + 1; i++)
                if(arr[i - start] == i)
                        printf("%d, ", i);
        printf("\b\b }\n");
    }

    return 0;
}
```

Output:



Explanation:

Here we have used **vfork()** which is used for address sharing for both child and parent process.

Execution of the calling process is blocked until the child process calls one of the exec() family of functions, or calls exit().

The child process computes the sum of the numbers raised to the power of the length of the number. After the child process completes, the parent process prints the armstrong numbers from the given range.

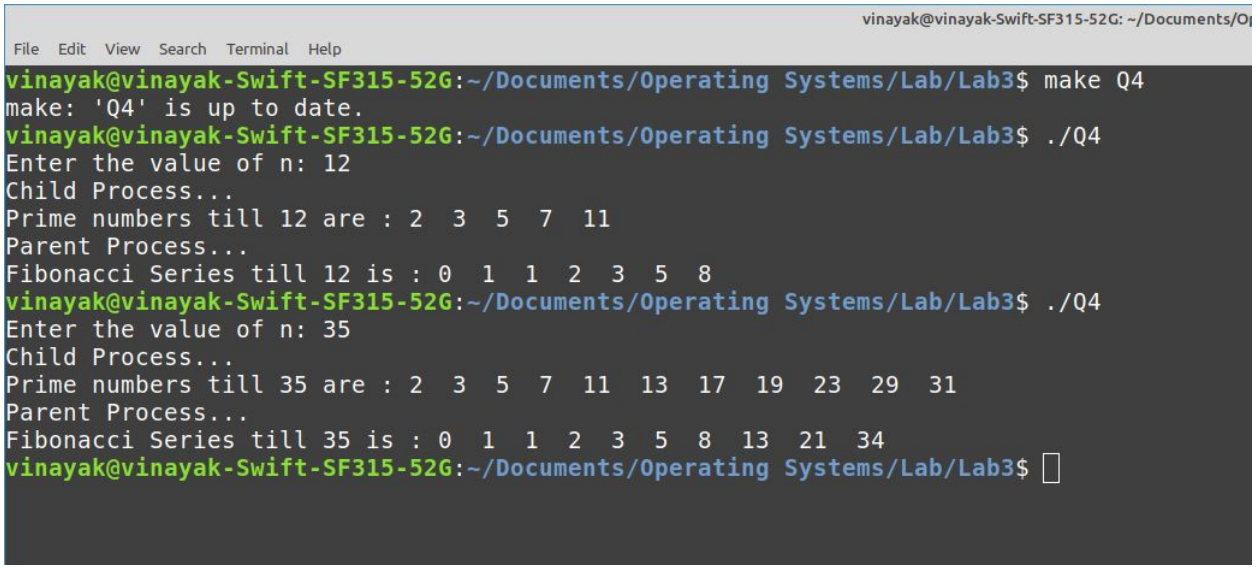
(4) Fibonacci Series and Prime parent child relationship (say parent does fib Number generation using series and child does prime series)

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

void Fibonacci(int n);
void Prime(int n);

int main()
{
      pid_t pid;
      int i,n;
```

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

	pid = fork();

	if(pid == 0) //child block
	{
		printf("Child Process...\n");
		printf("Prime numbers till %d are : ",n);
		Prime(n);
	}

	else //parent block
	{
		wait(NULL);
		printf("Parent Process...\n");
		printf("Fibonacci Series till %d is : ",n);
		Fibonacci(n);
	}

	return 0;
}

void Fibonacci(int n)
{
	int sum = 0, a = 0, b = 1;
	while(sum <= n)
	{
		printf("%d  ", sum);
		a = b; //swap elements
		b = sum;
		sum = a+b; // next term
	}
	printf("\n");
}

void Prime(int n)
{
	int i,j;
	for(i=2; i<=n; i++)
	{
```

```
            int count=0;
            for(j=1; j<=i; j++)
            {
                  if(i%j == 0)
                        count++;
            }

            if(count == 2)
                  printf("%d  ",i);
      }
      printf("\n");
}
```

Output:



Explanation:

The child process prints the fibonacci series till ‘n’. After the child process is completed, the parent process prints the prime numbers which are less than ‘n’.

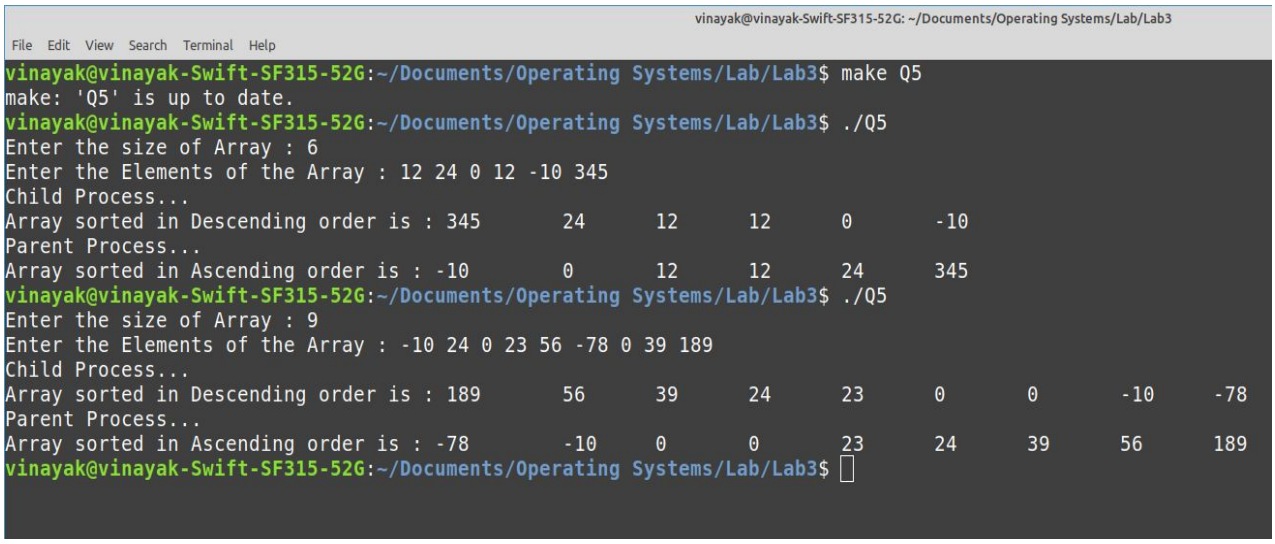
(5) Ascending Order sort within Parent and Descending order sort (or vice versa) within the child process of an input array. (you can view as two different outputs –first entire array is asc order sorted in op and then the second part desc order output)

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

void bubblesort(int *arr, int n, bool comp(const void* , const void*));
bool asc(const void* a, const void* b);
bool desc(const void* a, const void* b);
void swap(int *a,int *b);
void print(int *arr,int n);

int main()
{
	int i,size;
	int arr[size];
	pid_t pid;

	printf("Enter the size of Array : ");
	scanf("%d",&size);
	printf("Enter the Elements of the Array : ");
	for(i=0; i<size; i++)
	{
		scanf("%d", &arr[i]);
	}

	pid = fork();

	if(pid == 0) //child block
	{
		printf("Child Process...\n");
		printf("Array sorted in Descending order is : ");
		bubblesort(arr,size,desc);
		print(arr,size);
	}
```

```
	else //parent block
	{
		wait(NULL);
		printf("Parent Process...\n");
		printf("Array sorted in Ascending order is : ");
		bubblesort(arr,size,asc);
		print(arr,size);
	}

	return 0;
}

void bubblesort(int *arr,int n, bool comp(const void* , const void*))
{
	for(int i=0; i<n-1; i++)
	{
		for(int j=0; j<n-i-1; j++)
		{
			if(comp(&arr[j], &arr[j+1]))
				swap(&arr[j],&arr[j+1]);
		}
	}
}

bool asc(const void* a, const void* b)
{
	return *(int*)a > *(int*)b;
}

bool desc(const void* a, const void* b)
{
	return *(int*)a < *(int*)b;
}

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

void print(int *arr,int n)
```

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

Output:



Explanation:

The child process prints the input array into descending order. After the child process completes, the parent process prints the same array in the ascending order.

(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)

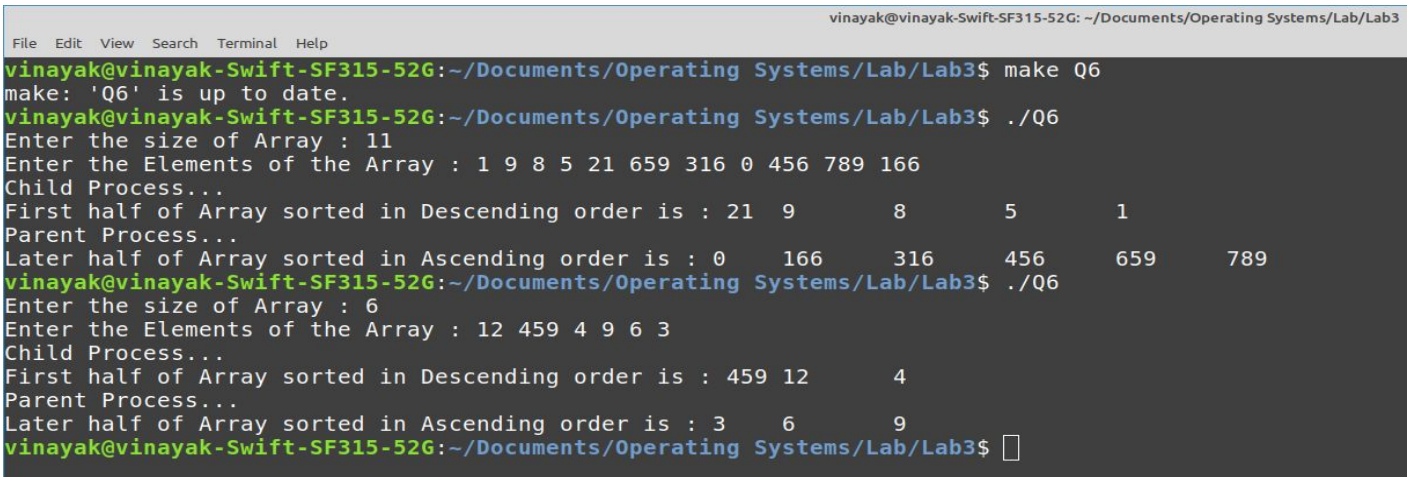
```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
```

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

void bubblesort(int *arr, int n, bool comp(const void* , const void*));
bool asc(const void* a, const void* b);
bool desc(const void* a, const void* b);
void swap(int *a,int *b);
void print(int *arr,int n);

int main()
{
	int i,size,mid;
	pid_t pid;

	printf("Enter the size of Array : ");
	scanf("%d",&size);
	mid  = size/2;
	int arr1[mid], arr2[size-mid];
	printf("Enter the Elements of the Array : ");
	for(i=0; i<mid; i++)
	{
		scanf("%d", &arr1[i]);
	}

	for(i=0; i<size-mid; i++)
	{
		scanf("%d", &arr2[i]);
	}

	pid = fork();

	if(pid == 0) //child block
	{
		printf("Child Process...\n");
		printf("First half of Array sorted in Descending order is : ");
		bubblesort(arr1,mid,desc);
		print(arr1,mid);
		exit(0);
	}

	else //parent block
```

```
	{
		wait(NULL);
		printf("Parent Process...\n");
		printf("Later half of Array sorted in Ascending order is : ");
		bubblesort(arr2,size-mid,asc);
		print(arr2,size-mid);
	}

	return 0;
}

void bubblesort(int *arr,int n, bool comp(const void* , const void*))
{
	for(int i=0; i<n-1; i++)
	{
		for(int j=0; j<n-i-1; j++)
		{
			if(comp(&arr[j], &arr[j+1]))
				swap(&arr[j],&arr[j+1]);
		}
	}
}

bool asc(const void* a, const void* b)
{
	return *(int*)a > *(int*)b;
}

bool desc(const void* a, const void* b)
{
	return *(int*)a < *(int*)b;
}

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

void print(int *arr,int n)
{
```

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

Output:



Explanation:

The child process prints the first half of the array in the descending order. After the child process completes, the parent process prints the later half in the ascending order.

(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)

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

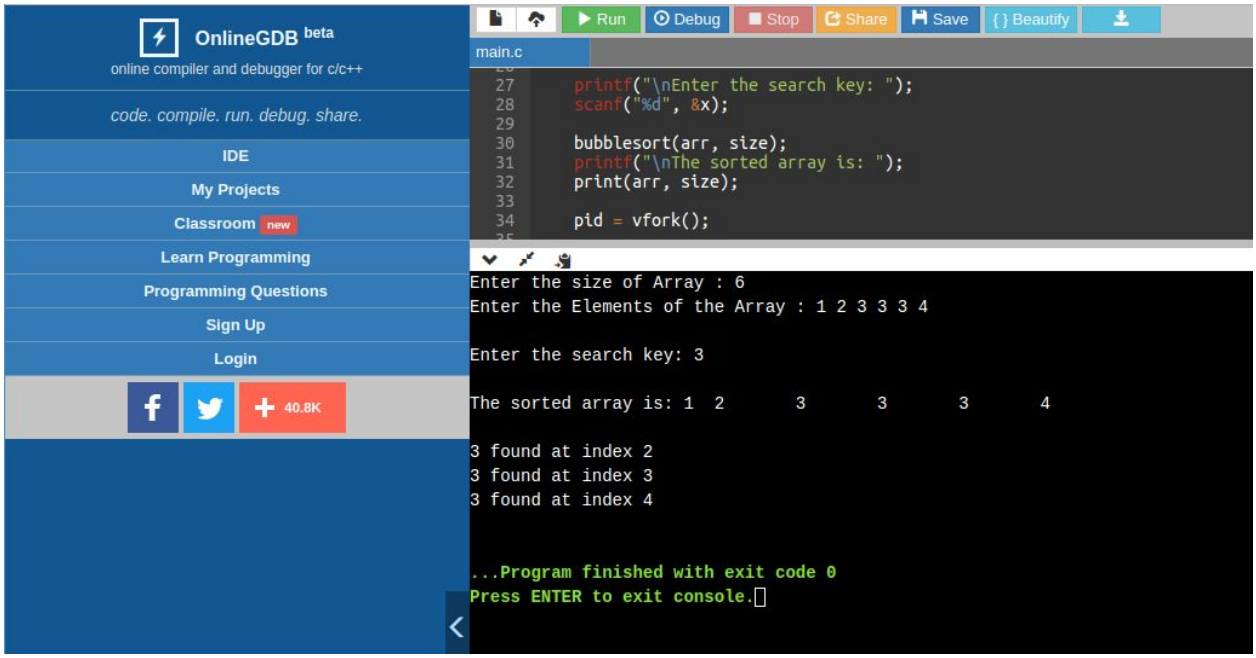
```c
#include<sys/wait.h>
#include<unistd.h>
#include<stdbool.h>

int binarysearch(int *a, int n, int x);
void bubblesort(int *arr, int n);
void swap(int *a,int *b);
void print(int *arr,int n);

int main()
{
	int size, mid, x, i, count = 0;
	int arr[size];
	pid_t pid;

	printf("Enter the size of Array : ");
	scanf("%d",&size);
	printf("Enter the Elements of the Array : ");
	for(i=0; i<size; i++)
	{
		scanf("%d", &arr[i]);
	}

	printf("\nEnter the search key: ");
	scanf("%d", &x);

	bubblesort(arr, size);
	printf("\nThe sorted array is: ");
	print(arr,size);

	pid = vfork();

	if(pid == 0) //child block
	{
		if((mid = binarysearch(arr, size, x)) == -1)
		{
			printf("\n%d is not found in the array\n", x);
			exit(1);
		}
		else
			printf("\n%d found at index %d\n", x, mid);
```

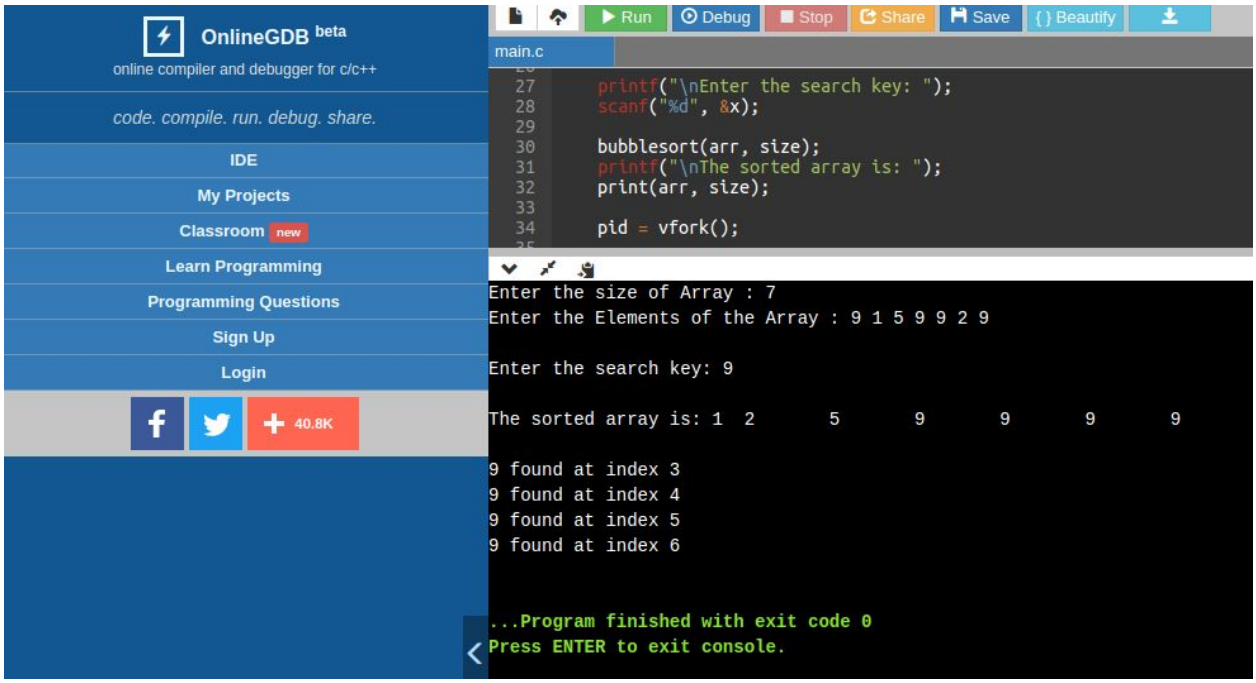
```
            i = mid - 1;
            while(i > -1 && arr[i] == x)
            {
                  printf("%d found at index %d\n", x, i);
                  i--;
            }

            exit(0);
      }

      else //parent block
      {
            wait(NULL);
            i = mid + 1;
            while(i<size && arr[i] == x)
            {
                  printf("%d found at index %d\n", x, i);
                  i++;
            }
      }

      return 0;
}

int binarysearch(int* a, int n, int x)
{
      int left = 0, right = n - 1;
      while (left <= right)
      {
            int mid = left + (right - left) / 2;
            if(a[mid] == x)
                  return mid;
            else if(a[mid] < x)
                  left = mid + 1;
            else
                  right = mid - 1;
      }
      return -1;
}

void bubblesort(int *arr,int n)
{
```

```
	for(int i=0; i<n-1; i++)
	{
		for(int j=0; j<n-i-1; j++)
		{
			if(arr[j] > arr[j+1])
				swap(&arr[j],&arr[j+1]);
		}
	}
}

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

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

Output:

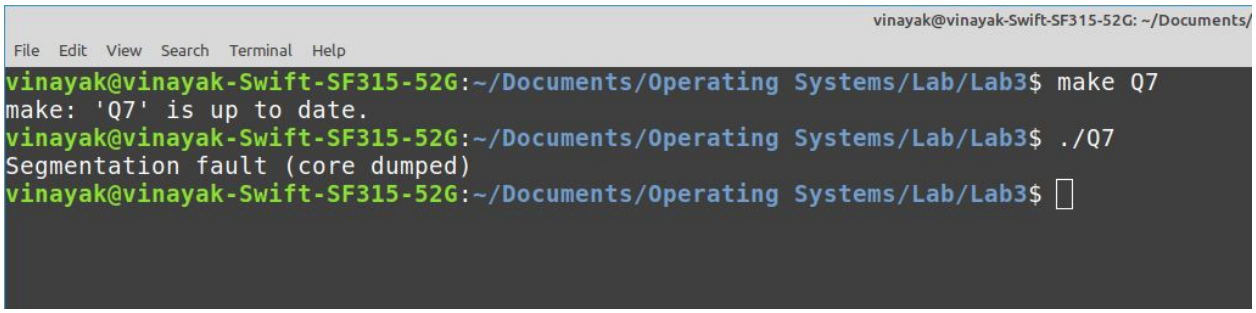
On online compiler





On Kaushik Sai compiler

On my compiler



Explanation:

The input array must be sorted for binary search to work so if the input array is not sorted then the array is sorted first.

Then the child process searches for the input search element. After the child process is completed then the parent process find the same element/duplicates (if any) in the array.

Note: Still figuring out why I am getting segmentation fault in my terminal.

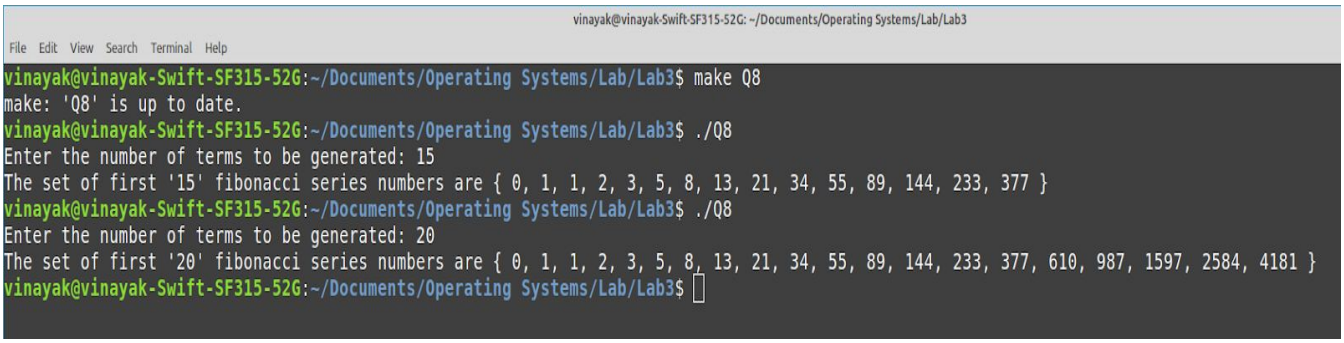
(8) * Non Mandatory [extra credits]

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).

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

int buff[100];
int fib(int n);

int main()
{
	int n, i;
	printf("Enter the number of terms to be generated: ");
	scanf("%d", &n);

	for(i = 0; i <100; i++)
	{
		buff[i] = -1 ;
	}
	printf("The set of first '%d' fibonacci series numbers are { ", n);

	for(i=0; i<n; i++)
	{
		printf("%d, ", fib(i));
	}
	printf("\b\b }\n");

	return 0;
}

int fib(int n)
```

```
{
	int temp = 0;

	if (buff[n] == -1)
	{
		if (n <= 1)
			buff[n] = n;

		else
		{
			pid_t pid = vfork();

			if(pid == 0) //child block
			{
				temp += fib(n - 1);
				exit(0);
			}
			else //parent block
			{
				wait(NULL);
				temp += fib(n - 2);
			}

			buff[n] = temp;
		}
	}

	return buff[n];
}
```

Output:



Explanation:

Using the concept of dynamic programming, memorization, we avoid duplicate calculations in the fibonacci series generation.

The child process finds the fibonacci number, fib(n - 1). After the child process is completed , the parent process finds the fibonacci number, fib(n - 2).

The above sum is stored in buff(buffer) hence the duplicate calculations are avoided.