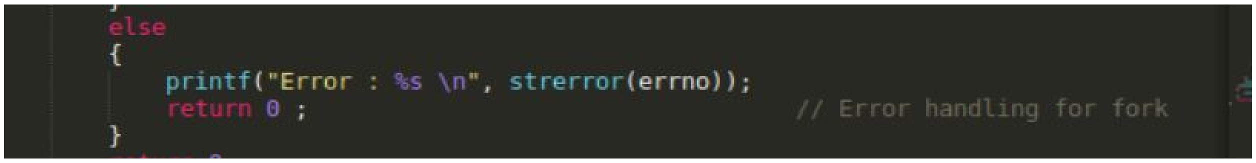
Part 1 Description

I have used the following system calls in my code -

1) **fork()**

   Functionality -
   - Fork system call is used to create a new process from an existing process.
   - It does not take any arguments and returns an integer process ID (of type pid_t).
   - This process Id is -
     - Negative - if the process fails to be created
     - Zero - returns zero to child process
     - Positive - returns process id of child to parent
   - A child process is created from an existing parent process. This child process has the same program counter, registers and files as the parent since a copy of the parent's address space is passed onto the child.

   Error handling -
   - If the process id is negative (-1), then the system is unable to create a child process from the parent process.
   - Hence, fork() command fails and the errno is set. We use the function strerror() to print the error message corresponding to the set errno in case the value returned by fork is negative.



```
else
{
    printf("Error : %s \n", strerror(errno));
    return 0 ;                                  // Error handling for fork
}
```

   For eg -
   1) EAGAIN - **"Resource temporarily unavailable"** - this message occurs when the kernel crosses the maximum number of threads and processes that can be reached.
   2) ENOMEM fork() - **"Not enough space/Cannot allocate memory"** - This message occurs when there isn't enough memory to allocate important kernel structures.

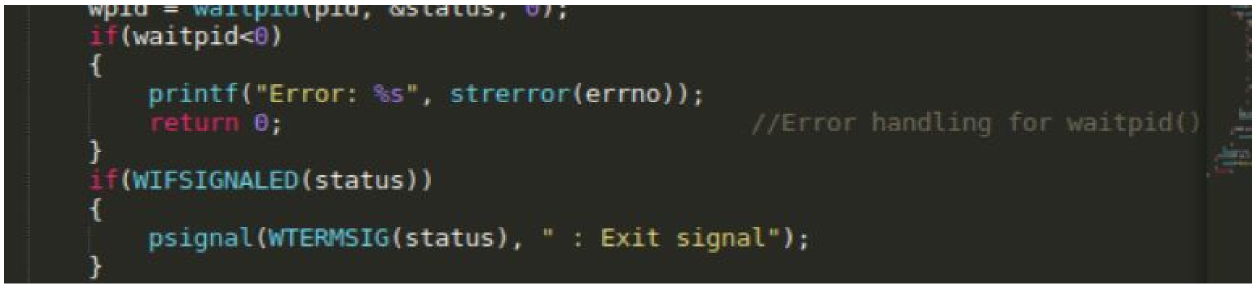
2) waitpid()

Functionality -
- waitpid() system call is used to make the calling process wait until it receives a particular signal. This signal can be the termination of a child process etc.
- It takes in three parameters - pid_t waitid(pid_t *id*, int *status, int *options*);
  1) id is the id of the child/other process for whose signal the calling process has to wait
  2) status stores the status of execution / termination returned by the child process.

   3) options - it specifies the type of signal the parent has to wait for till it starts its own execution.

- I have used the following parameters **waitpid(pid, &status, 0);**
   1) pid is the id of the child process returned in the parent. It is of type pid_t(int).
   2) Status is an integer that stores the execution/termination state of child
   3) 0 means that the parent process has to wait till the child gets terminated.

Error handling -
It returns a pid_t(int) which tells us whether the system call was successfully executed or not.
In my program wpid takes the value returned by waitpid().
If wpid<0 then we display error message corresponding to the set errno using the function strerror() .
Moreover if the child doesn't terminate but gives a signal, I have displayed that as well using the psignal() function.



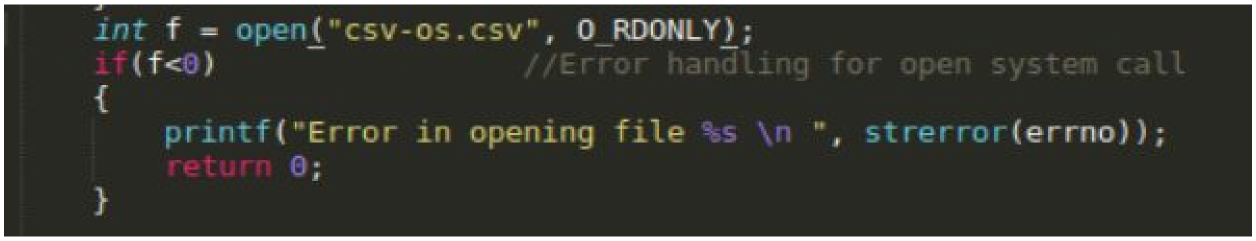
```
    wpid = waitpid(pid, &status, 0);
    if(waitpid<0)
    {
        printf("Error: %s", strerror(errno));
        return 0;                              //Error handling for waitpid()
    }
    if(WIFSIGNALED(status))
    {
        psignal(WTERMSIG(status), " : Exit signal");
    }
```

For eg -
1) ECHILD - **"No child processes"** - This error message occurs when the process corresponding to the specified process id does not exist or isn't the child of that process.
2) EINTR - **"Interrupted function call" -** This error message occurs when there is an interruption due to an interrupt or signal being sent.

3. open()
Functionality -
- It is used for reading or writing or both.
- It takes in two parameters - int open(const char **pathname*, int *flags*);
   1) The first is the path of the file i.e the place where the file is present in the system. It can be the absolute path or the relative path(when file is present in same directory).
   2) The second parameter takes the flags specifying the read and write permissions.
      O_RDONLY - read only
      O_WRONLY - write only
      O_RDWR - read and write both
- I have used the following parameter - open("csv-os.csv", O_RDONLY)
   - "csv-os.csv" is the name of the file to be read. I have not given an absolute path since they are in the same directory
   - O_RDONLY - specifies that we have opened the file in read only mode

- I have opened the file in read only mode since we only need to read the csv file.
- It returns an integer. This integer(file descriptor) can be used for further system calls like lseek(), read() etc.

  Error handling -
  - The system call returns an integer. If the integer is a non negative value then the file has opened successfully.There can be various reasons like we might not have the permission to open the file or it might not exist.
  - I have handled this error by displaying an error message using the strerror() function.



```
int f = open("csv-os.csv", O_RDONLY);
if(f<0)                    //Error handling for open system call
{
    printf("Error in opening file %s \n ", strerror(errno));
    return 0;
}
```

For eg -

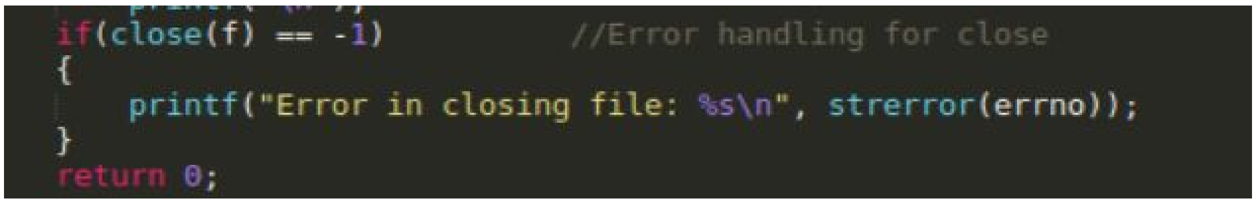
1) EACCES **-"Permission denied**" WHen the desired permission is not provided and hence there is error in accessing the file.
2) EINVAL - **"Invalid arguments"** - When the valid flags haven't been given in the second parameter.

4) close()
Functionality -

- This function call is basically used to close the file descriptor opened by the open() system call.
- It takes one argument -
  An integer file descriptor returned by open() that needs to be closed
- I have given the parameter - close(f) -
  f is the file descriptor of "csv-os.csv" file that we read.

Error handling

- It returns an integer.
- If it is successful in closing the file it returns 0.
- Otherwise in case of an error it returns -1 and sets the errno.
- I have handled this error by printing the value returned by strerror() function when close() returns a value of -1.



```
if(close(f) == -1)          //Error handling for close
{
    printf("Error in closing file: %s\n", strerror(errno));
}
return 0;
```

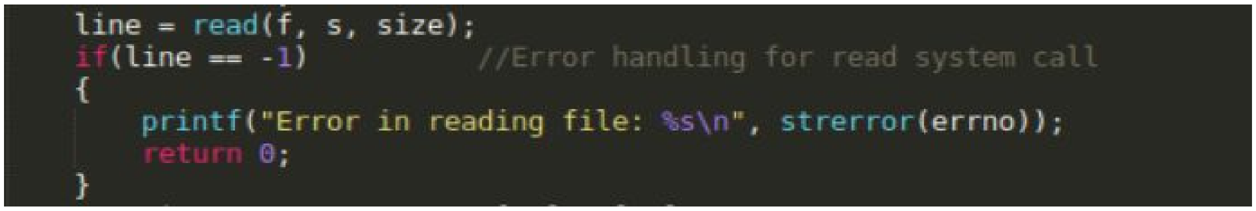
1. EBADF - “**Bad file descriptor” -** When the argument specified as the file descriptor isn’t valid.
2. EINTR - **“Interrupted function call** “ - When the close() command is interrupted by some signal.

5) read()
Functionality -

- It is used to read the specified number of bytes from a file descriptor.
- It takes in three arguments ssize_t read(int *f*, void **ch*, size_t *count*);
    1) f is the file descriptor of the file from which we want to read
    2) ch is the buffer in which the contents of the file are read.
    3) count is the number of bytes to be read
- It returns the number of bytes read.
- I have specified the following parameters - read(f, s, size)
    - f is the file descriptor of “csv-os.csv” file that we read.
    - s is the character array in which i have read my file
    - size is the number of bytes read

Error handling -

- The read() function call returns the total number of bytes read from the file.
- If the returned number is -1, then an error has occurred.
- I have handled this error by printing it on the terminal using strerror() function as errno is set when the function returns a negative value.



```
line = read(f, s, size);
if(line == -1)          //Error handling for read system call
{
    printf("Error in reading file: %s\n", strerror(errno));
    return 0;
}
```

Eg -
1.EBADF - **“Bad file descriptor” -** When the argument specified as the file descriptor isn’t valid.
2.EINTR - “**Interrupted function call** “ - When the close() command is interrupted by some signal.

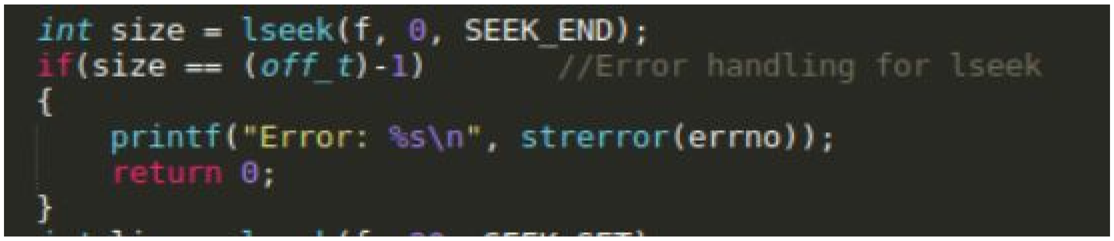
6) lseek()

Functionality -

- It is used to position a pointer on the file according to offset arguments specified.
- It takes three parameters- off_t lseek(int *f*, off_t *off*, int w);
    1. f is the file descriptor of the file on which we want to reposition the pointer
    2. off is the offset value used to determine the position
    3. w specifies the point from where we consider the offset. It can be
       SEEK_SET - starting of file (offset = off)
       SEEK_END - end of file (offset = off+size of file)
       SEEK_CUR - current position of file(offset = off + cur. Position)
- I have used the following parameters - lseek(f, 0, SEEK_END)
  f is the file descriptor of "csv-os.csv" file that we read.
  0 specifies the offset
  SEEK_END specifies that total offset is size of file

Error handling -

- It returns the final offset measured from the starting of file.
- When there is an error it returns the value (off_t)-1 and sets the errno. I have handled this error by printing error message using errno when this value is returned



```
int size = lseek(f, 0, SEEK_END);
if(size == (off_t)-1)        //Error handling for lseek
{
    printf("Error: %s\n", strerror(errno));
    return 0;
}
```

Eg

1. EOVERFLOW -**"Value too large to be stored in datatype"** - This error occurs when the offset reaches beyond (off_t).
2. EBADF **- "Bad file descriptor**" - When the argument specified as the file descriptor isn't valid.

7) write()

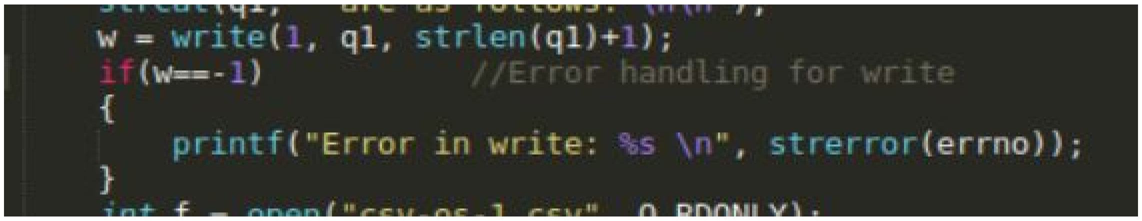
Functionality -

- It is used to write a specific number of bytes to a file.
- It takes three parameters - ssize_t write(int *f*, const void **ch*, size_t *count*);
    - f is the file descriptor of the file on which we want to write
    - ch is the buffer in which the contents of the file are written
    - count is the number of bytes to write

- It returns the number of bytes written .
- it can also write on the terminal by specificting the first argument as ‘1’. I have done the same in my program to write on the terminal.
- I have used the following parameters - write(1, q1, strlen(q1)+1)
    - 1 specifies that we need to write on the terminal
    - q1 contains the contents to be written
    - strlen(q1)+1 is the number of bytes to be written

Error handling -

- The write() function call returns the total number of bytes written in the file. It might be lesser than count due to a variety of reasons - since it was interrupted by a signal or memory wasn’t enough in the disk.
- If the returned number is -1, then an error has occurred.
- I have handled this error by printing it on the terminal using strerror() function as errno is set when the function returns a negative value.



```
w = write(1, q1, strlen(q1)+1);
if(w==-1)           //Error handling for write
{
    printf("Error in write: %s \n", strerror(errno));
}
```

Eg

1. EBADF - **“Bad file descriptor”** - When the argument specified as the file descriptor isn’t valid.
2. EINTR - “**Interrupted function call** “ - When the close() command is interrupted by some signal.

Working of my program

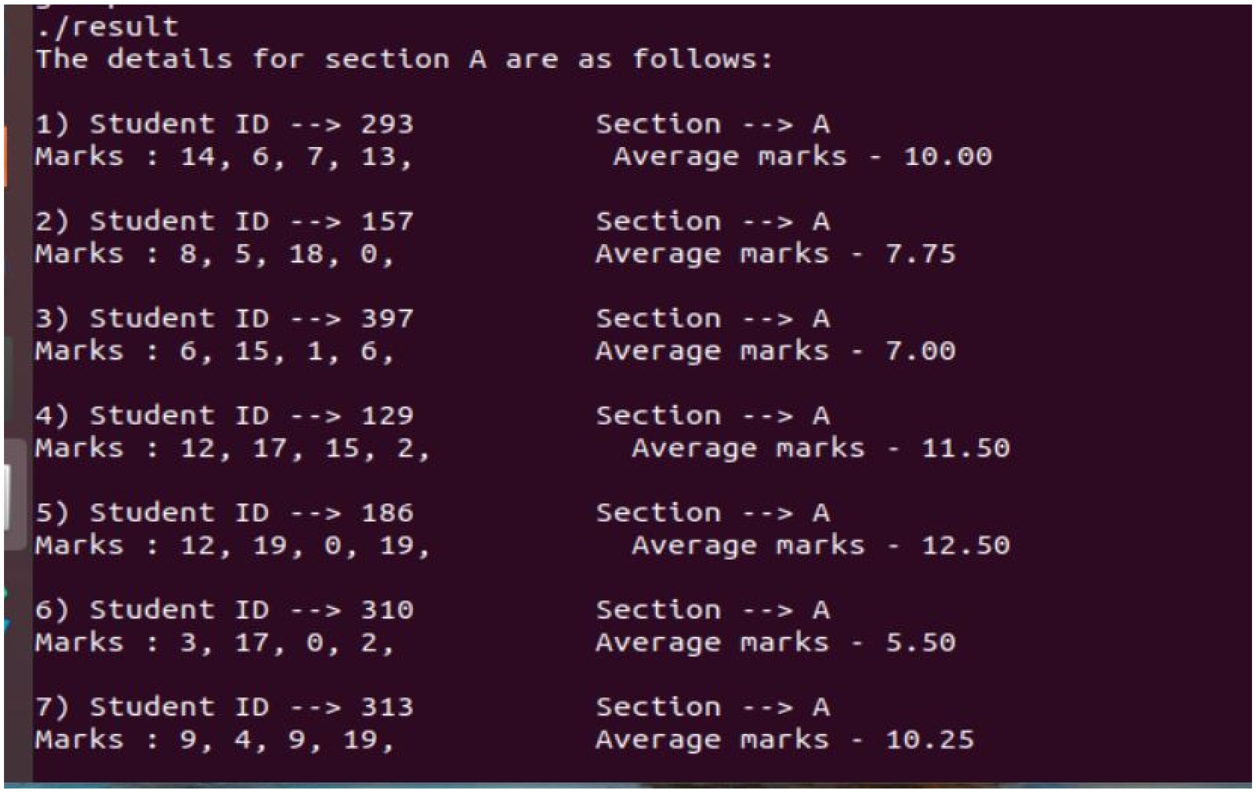
Basic code explanation -

- In the main function of the program I use fork() system call to create a child process.
- If the pid (process id) is 0 , then we are within the child process. In that case, I call my read_csv(char *a) function with the character parameter “A” such that it displays the average marks of the students of section A.
- If the process id is positive, then we are within the parent process as child process id is returned in the parent. In that case, I call my read_csv(char *a) function with the character parameter “B” such that it displays the average marks of the students of section B.The parent process waits for the child to terminate using the waitpid system call.
- The read_csv(char *a) takes a character parameter. It checks the section of the student and prints the marks accordingly in parent and child processes.Within this function I have used read() and lseek() system calls to read the contents of the csv file.
- I have then used strtok to separate the space separated tokens and calculated and print the average marks.

- At the end I print the details of the students and their calculated average marks

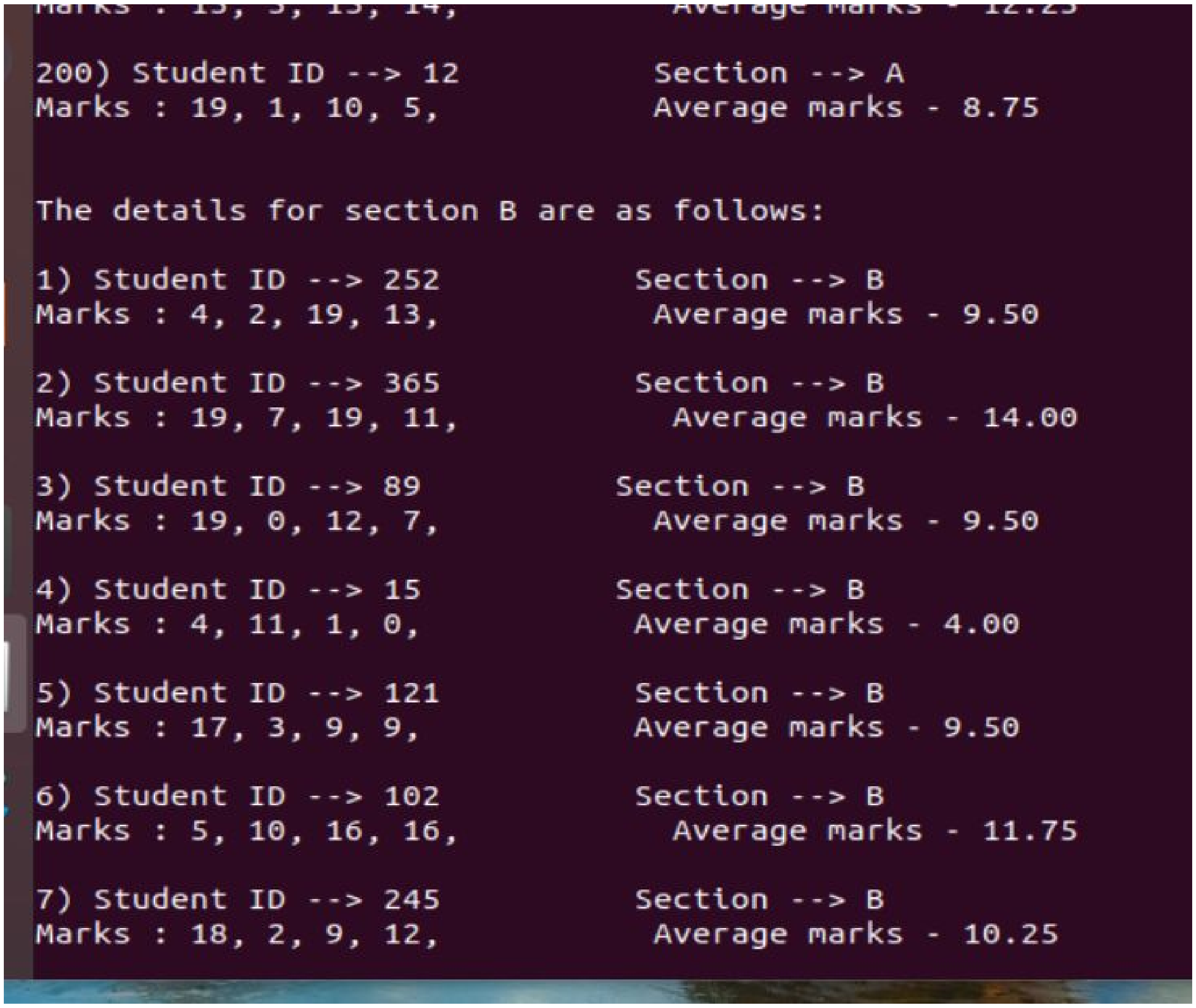
OUTPUT

This is the output of the child process that prints the average marks for the students of section A.



```
./result
The details for section A are as follows:

1) Student ID --> 293          Section --> A
Marks : 14, 6, 7, 13,           Average marks - 10.00

2) Student ID --> 157          Section --> A
Marks : 8, 5, 18, 0,           Average marks - 7.75

3) Student ID --> 397          Section --> A
Marks : 6, 15, 1, 6,           Average marks - 7.00

4) Student ID --> 129          Section --> A
Marks : 12, 17, 15, 2,           Average marks - 11.50

5) Student ID --> 186          Section --> A
Marks : 12, 19, 0, 19,           Average marks - 12.50

6) Student ID --> 310          Section --> A
Marks : 3, 17, 0, 2,           Average marks - 5.50

7) Student ID --> 313          Section --> A
Marks : 9, 4, 9, 19,           Average marks - 10.25
```

This is the output for the parent process that prints the average marks of the students in section B. It starts only after the child process is terminated.



```
Marks : 13, 3, 13, 14,            Average marks - 12.25

200) Student ID --> 12            Section --> A
Marks : 19, 1, 10, 5,             Average marks - 8.75


The details for section B are as follows:

1) Student ID --> 252            Section --> B
Marks : 4, 2, 19, 13,             Average marks - 9.50

2) Student ID --> 365            Section --> B
Marks : 19, 7, 19, 11,             Average marks - 14.00

3) Student ID --> 89            Section --> B
Marks : 19, 0, 12, 7,             Average marks - 9.50

4) Student ID --> 15            Section --> B
Marks : 4, 11, 1, 0,             Average marks - 4.00

5) Student ID --> 121            Section --> B
Marks : 17, 3, 9, 9,             Average marks - 9.50

6) Student ID --> 102            Section --> B
Marks : 5, 10, 16, 16,             Average marks - 11.75

7) Student ID --> 245            Section --> B
Marks : 18, 2, 9, 12,             Average marks - 10.25
```