# Q1 Write-Up

## Process Creation and Termination Using System Calls

### Working of the Program(s)

At first, the program(s) will calculate the Buffer Size and the number of students present in the CSV file, so that this information can be used.

**For Fork Program**: After the above calculations, the fork() command would create a new child process. In both the parent and child process, we first store the data in proper format into arrays for our ease of use by calling getInfo(). In the getInfo() function, we use the open() and read() system calls to extract the data into a buffer and break the data into a series of tokens and store them in the appropriate arrays. Using these arrays, now the data is processed easily, and averages are computed. After child program execution ends, exit(0) is called to end the child. In the parent program (whose id returned by fork() > 0), we use the waitpid() system call, so that the parent waits for the child process to terminate and then begins its execution.

**For Pthread Program**: Here, instead of fork(), we make two new threads for the parent and child using pthread_create(). The extraction of data in both threads is in the same way as described above for the fork program, by using getInfo() and extracting data stored in the arrays for computation. The only addition in the threads, is that we also increase the total marks & total student count, for each assignment (both global variable arrays) from each thread. This is possible because memory is shared between threads. Hence, by using the result obtained by the two threads for each section, we can compute the average score of each assignment across the sections. Before starting the parent thread, we do a pthread_join() on child thread, so that the parent thread creation & execution would wait until the child thread finishes. We also do pthread_join() on parent thread, so that parent thread is executed before program main() ends.

### System Calls Used

1. **fork()** is used to create the child process. No arguments are passed in it. For Error Handling, I used the fact that fork() would return a negative value if creation was unsuccessful.
2. **open()** is used to open the file for reading and/or writing. It takes 2 arguments, the File Path Name and Flags, specifying the read and write permissions and returns the file descriptor. On error, it returns -1, and I've done the error handling accordingly.
3. **close()** is used to close the file pointed by the file descriptor. It takes the fd to be closed as an argument. On error, it would return -1 and error handling is done accordingly.
4. **read()** is used to read some amount of bytes into the buffer from the file indicated by the file descriptor. It takes 3 arguments: fd-> file descriptor, buffer-> buffer to read data into, count-> buffer_size and returns number of bytes read on success. On error, it returns negative value, and error handling is done accordingly.
5. **waitpid()** is used to wait for state changes in a child. It suspends execution of the current process until a child specified by pid argument has changed state. It takes 3 arguments: pid-> process to wait for, status-> pointer to an integer which will be filled with the exit status, options-> used for flags. It returns process ID of the child process and on failure, it returns -1 and error handling is done accordingly.

6. **pthread_create()** is used to create a new thread within the same process. It takes 4 arguments: thread-> location where ID of thread is stored, attr-> specifies attributes for the thread, start-> function for the thread, arg-> argument passed. It returns 0 if creation was successful, else returns errno through which error handling is done accordingly.

7. **pthread_join()** is used to suspend execution until the target thread terminates. It takes 2 arguments: thread->thread to wait for, status-> location where the exit status of joined thread is stored. It returns 0 if successful, else returns errno through which error handling is done accordingly.

8. **stat()** is used return information about a file, in the buffer pointed to by statbuf. It takes two arguments, the File Path Name and statbuf address. On error, it returns -1, and I've done the error handling accordingly. We have used stat() in order to know the buffer_size in advance.

## Compiling and running of Program(s)

Simply running 'make' in the same directory as the question, would compile step by step, generate the intermediate files, and execute the program. The actual step by step process is as follows:

```
gcc -E q1fork.c -o q1fork.i
gcc -S q1fork.i -o q1fork.s
gcc -c q1fork.s -o q1fork.o
gcc q1fork.o -o q1fork
./q1fork
```