Directory Structure

- **Assignment1**
  - **Problem_1_fork** - Implementation of Problem_1 using fork system call
    - averageFork.c
    - student_record.csv
    - makefile
    - average.csv- This file will be created automatically and contain the average as required in a CSV format.
  - **Problem_1_thread** - Implementation of Problem_1 using Thread
    - averageThread.c
    - student_record.csv
    - makefile
    - average.csv- This file will be created automatically and contain the average as required in a CSV format.
  - **Problem_2** - Implementation of problem 2 and modification of stack using inline assembly.
    - main.c - This is the file with main method.
    - funcA.c
    - funcB.c
    - funcC.c
    - function.h - This is the header file
    - makefile

Problem_1_fork

Input

No inputs need to be given for running the program.
However we need student_record.csv in the folder directory

Output

The output averages would be written in a average.csv file.
If the file is not already present in the folder then it would be created.
Otherwise the file would be overwritten with new averages on each run.

Expected Output

For supplied csv file the expected output is

```
Section,Assignment 1,Assignment 2,Assignment 3,Assignment 4,Assignment 5,Assignment 6
A,51.56,55.33,37.67,58.33,54.00,38.56
B,53.06,52.94,39.76,58.71,64.06,50.00
```

Note: The average.csv file could be opened in MS-Excel or in any text editor such as VSCode.

Compiling the Program

For compiling the program a makefile is supplied in the program directory.
First run make in a linux terminal inside the program folder **Problem_1_fork**.
Then run the **averageFork** file using **./averageFork**.

System Calls Used

- open() - This system call is used to open/create a file for reading. After opening the file a file descriptor integer is returned. This file descriptor is used in write system call.
    - open("average.csv", O_CREAT, S_IRWXU) - O_CREAT specifies that a new file is needed to be create. S_IRWXU is used to set the read/write permissions.
    - newFile = open("average.csv", O_RDWR) - O_RDWR opens the average.csv file created above in the Read/Write mode. The file descriptor is passed to newFile.
- write() - This system call is used to write to the file. Arguments are write(int file_descriptor, char* str, write_length).
    - write(newFile, strB, strlen(strB)); - This writes to the file specified by the file descriptor. The string B is written to the file.

- fork() - This system call is used to call the fork command. No argument is called to fork . It returns 0 in the child process and the pid of the child process in the parent process.
- waitpid() - This system call is used to wait for the process with specified ID to exit using the _exit() system call.
  waitpid(id, NULL, 0) here id specifies the ID of the process to wait for.

- _exit() - This system call is used to exit from the process to the parent process returning the status as specified in the status argument. In this case 0 is passed as the argument to the system call.

Description of How program works

- First we create and then open the supplied csv file using open and write system call.
- Then we store the string till the newline in a string array and repeat the process until we encounter an EOF ie. till write system call return 0.
- Then we delimit each stored string by “,” using strtok() . Then we check for the section. We check for section B in the parent process and section A in the child process.
- Then we add the assignment scores to appropriate assignments
- We then calculate the average for each assignment and then add this to a strA in child process and strB in parent process.
- We have used fork system call in this problem.
- We first check if the id returned by fork is 0. If its 0 then we are in the child process and we call the getAverageSectionB function.
- If value returned by fork is non zero then we are in parent process. We use waitpid system call and check whether the child process has exited.If the child process is still running then we make the parent process to wait.
- When the child process finally exits by exit system call then the parent process starts executing. It then calculates the average for section B and then writes the average to the average.csv file.

Problem_1_thread

Input

No inputs need to be given for running the program.
However we need student_record.csv in the folder directory

Output

The output averages would be written in a average.csv file.
If the file is not already present in the folder then it would be created.
Otherwise the file would be overwritten with new averages on each run.

Expected Output

For supplied csv file the expected output is

```
Section,Assignment 1,Assignment 2,Assignment 3,Assignment 4,Assignment 5,Assignment 6
A,51.56,55.33,37.67,58.33,54.00,38.56
B,53.06,52.94,39.76,58.71,64.06,50.00
Combined,52.31,54.14,38.72,58.52,59.03,44.28
```

Note: The average.csv file could be opened in MS-Excel or in any text editor such as VSCode.

Compiling the Program

For compiling the program a makefile is supplied in the program directory.
First run make in a linux terminal inside the program folder **Problem_1_thread**.
Then run the **averageThread** file using **./averageThread**.

System Calls Used

- open() - This system call is used to open/create a file for reading. After opening the file a file descriptor integer is returned. This file descriptor is used in write system call.
    - open("average.csv", O_CREAT, S_IRWXU) - O_CREAT specifies that a new file is needed to be create. S_IRWXU is used to set the read/write permissions.
    - newFile = open("average.csv", O_RDWR) - O_RDWR opens the average.csv file created above in the Read/Write mode. The file descriptor is passed to newFile.
- write() - This system call is used to write to the file. Arguments are write(int file_descriptor, char* str, write_length).

    - write(newFile, strB, strlen(strB)); - This writes to the file specified by the file descriptor. The string B is written to the file.
- pthread_create - This system call is used to create a thread and make it execute the specified function.
    - pthread_create(&childThread, NULL, &getAverageSectionA, (void *)strA) Here &childThread specifies the address of the thread. &getAverageSectionA specifies the address of the function to execute. (void *)strA is the void pointer to the character pointer which is the argument for the function.
- pthread_join() - This system call is used to make the parent thread wait for the specified thread
    - pthread_join(childThread, NULL) - childThread specifies the id of the child thread to wait for.
- _exit() - This system call is used to exit from the process to the parent process returning the status as specified in the status argument. In this case 0 is passed as the argument to the system call.

Description of How program works

- First we create and then open the supplied csv file using open and write system call.
- Then we store the string till the newline in a string array and repeat the process until we encounter an EOF ie. till write system call return 0.
- Then we delimit each stored string by “,” using strtok() . Then we check for the section. We check for section B in the parent process and section A in the child process.
- Then we add the assignment scores to appropriate assignments
- We then calculate the average for each assignment and then add this to a strA in child thread and strB in parent thread.
- We have used p_thread create system call in this problem.
- The pthread_create system call creates a child thread which calculates the average for the child thread.
- Pthread_join system call prevents the parent thread from exiting until the child thread has exited by exit system call.
- When the child process finally exits by exit system call then the parent thread also exits after calculating the average for section B and then writes the average to the average.csv file. The combined averages are then calculated and written to the average.csv file.

Problem_2

Input
We need to input a 64bit binary string representing the 8byte character string.

Output
The output averages would be written in a average.csv file.
If the file is not already present in the folder then it would be created.
Otherwise the file would be overwritten with new averages on each run.

Expected Input

```
We are in main function now
We are in function A now
Enter the 64bit binary string
0100011001000111010001100100011001000110010001100100011001000010
```

Expected Output

```
❯ ./main
We are in main function now
We are in function A now
Enter the 64bit binary string
0100011001000111010001100100011001000110010001100100011001000010
We are in function B now
FGFFFFFB
Modifying the function Stack
We are in function C
exit by syscall
```

Compiling the Program
For compiling the program a makefile is supplied in the program directory.
First run **make** in a linux terminal inside the program folder Problem_2.
Then run the main file using **./main** command

Description of the Program

- Firstly main function calls A. A doesn't take any arguments.
- Then A calls B. A 64bit binary string is passed as an argument to function B. We have called function B in a normal way in C.
- Now the function B first breaks the 64bit string to 8 1byte Strings.
- These strings are then converted into decimal ASCII Codes using the **strtol()** function.
- Now we have got the ASCII Codes of the individual characters of the strings and then they are printed using inline assembly language syscall.
- Now we call another helper function from B to modify the stack.
- The return address for the function call is pushed into the stack by function call command which return command uses to return to the specified address by popping the return address out of the stack and moving that return address to the %rip register.
- Then in the prologue phase of function call the old base pointer is also pushed into the stack so that the parent function can use it after exiting.
- So the idea is to use the pop two times in the assembly and then pushing the appropriate return address to the stack. We again need to restore the original base pointer so we push it back to the stack.
- The helper function pops return address and stack pointer of the stack. Then pushes the return address of C and then again restores the stack by pushing the original base pointer again.
- It then calls C without any explicit function call for C. In this way we modify the stack to call another function without using any explicit function call.
- Now C prints we are in function C and then it exits by using inline assembly and then doing a syscall.