Process creation and termination

1. Forking and averaging:
    a. First the main program starts running (later referred as parent process), the parent **open**s (syscall 2) the file, gets the file descriptor and passes it to syscall **read** which then reads the data into a 8KiB buffer.
    b. After that **fork** (which issues syscall 57) is used to create a child process. The wrapper **fork** (if successful), returns 0 in the child process and pid of the child process in the parent process. A simple if else condition detects which process it is and whether there was an error or not.
    c. In case the system wants to execute the parent process first; I have prevented it using a **waitpid** call (which is a wrapper around syscall 61 **wait4**). It basically waits for the process specified to end (here the child's pid is supplied).
    d. *Part common to both processes* is performed by a function named **operate**. When **operate** is called by the child process, it sends the argument 'A', asking the function to *operate* on section 'A' students only. Similarly, the parent process sends 'B'. **operate** then notifies which section it is calculating the average for, by using a syscall 1, or **write** on STDOUT.
    e. An <u>int</u> array of assignments is then created (initialized to 0). A loop traverses over the buffer using a string tokenizer (strtok_r POSIX) and keeps adding the marks of those students belonging to section specified in to the assignments array. The count of the students is also maintained.
    f. Finally, to compute the average, the whole array is then divided by the number of students in that section. Each average score gets written on the STDOUT. (If the process is the child process, the exit (syscall 60) is also used to mark the end of execution and exit.)
2. Threading and averaging:
    a. Again, start by opening the file and writing it to a buffer. However this time a 2D matrix is created (assuming there are only 6 assignments for each student). The array stores the section and the marks.
        i. Since threads share the same address space, the 2D matrix is accessible to both simultaneously. We need not worry about Writing hazards (WAW, WAR, RAW) since we will just need to read.
        ii. A function similar to **operate** is created to facilitate section wise average. In addition to printing, this function also returns the array.
    b. pthread_create is then used twice to create 2 threads which executes the function with arguments 'A' and 'B' respectively.
    c. Finally, and pthread_join captures the return values into 2 variables which is used to get the final average (i.e. $(x_1+x_2)/2$). And is also printed to STDOUT.

3. Error handling (appropriate messages included)
    a. Forking: **fork** returns negative value incase it fails
    b. Waitpid: **waitpid** returns -1 incase of an error.
    c. Threads: **pthread_create/pthread_join** returns non 0 value incase it fails.
    d. Write: **write** returns written bytes or -1 incase of failure.
    e. Opening file: incase of opening error, **open** returns -1
    f. Allocation: **malloc/calloc** both return a NULL pointer incase of failure.