Write up

In the main function, all the process is performed. So the code starts with calling **fork()** System call, which will generate a child process from the main parent process. fork() returns an integer.

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- It returns a zero to the newly created child process.
- It returns a positive value, the process ID of the child process, to the parent.

Error handled - if the creation of the child process is unsuccessful.

After calling a child process, it will return a positive value stored in pid of type pid_t. As it is +ve value, the command will execute the else part in which it calls the system call **waitpid(),** which will tell the parent to wait for the child to exit.

Syntax

pid_t waitpid(pid_t pid, int *wstatus, int options);

As an **argument to waitpid()**, pass the pid of the waiting child, the status to be returned, and the option. If the state of the child process changes (even if it is not terminated), waitpid() is returned.If it returns successfully, the child's pid is returned. Otherwise, -1 is returned.

Now the child process execution starts. So it will begin by opening a CSV file for reading using an **open()** system call.

Syntax

int open (const char* Path, int flag [, int mode]);

Parameters

- **Path:** the path to file which is to be read

Using a relative path, i.e., the only file name, as it is in the same file's directory.

- **flag : O_RDONLY**(for read-only)

How it works

- Find the existing file on disk

- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

Error handled: when the file doesn't exist or any failure while opening.

After that, first, it read the first line in the file, which is of no use with the help of a read system call, then pointer shift to the next line from where our main details start, then it will read that part and store in char buffer. From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by the buf. A successful **read()** updates the access time for the file.

Syntax

size_t read (int fd, void* buf, size_t cnt);

Parameters

- **fd:** file descriptor
- **buf:** buffer to read data from
- **cnt:** length of the buffer

Returns: return Number of bytes read on success

Important points

buf needs to point to a valid memory location with a length not smaller than the specified size because of overflow.

fd should be a valid file descriptor returned from open() to perform read operation because if fd is NULL, then read should generate the error.

cnt is the requested number of bytes read, while the return value is the actual number of bytes read. Also, sometimes read system calls should read fewer bytes than cnt.

After reading the file, the entire content would store in buf as a single string. Then I first split the string by line using the **strtok()** function by delimiter \r and \n and store it in a new char array. I again split the stored string by space so that I would get every detail separated in the array. Now I get the final array in which

all the data are separated. Now, it will start calculating section A average marks of the assignments and write to the terminal using the **write()** system call. I'll use 1 in the file descriptor to write to the terminal, as 1 is reserve for standard output.

Syntax

write (int fd, void* buf, size_t cnt);

Parameters

- **fd:** file descriptor
- **buf:** buffer to write data to
- **cnt:** length of the buffer
- **Returns:**
- return Number of bytes written on success
- return 0 on reaching the end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- **buf** needs to be at least as long as specified by cnt because if buf size is less than the cnt, buf will lead to the overflow condition.
- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** have a lower number of bytes to write than cnt.

After writing, the child process will end the process by calling the **exit()** system call. Thus, after child termination, the parent process will execute, calculating the average marks of the sec B in the same way as the child process did i.e, opening, reading, and writing.

Reference:
https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/

Makefile :

To compile the program use the **make** command which will compile the program and print the output. If you want to stop after compilation then use **make Compilation** as a command. To see the output of the compiled file use **make output**.when you run the program it will print “child process” when it executes child process. similar to the parent process.