CSE231: Operating Systems

Assignment 1, Part 1

Name: Mihir Chaturvedi
Roll no.: 2019061
Section: A
Branch: CSE

Introduction

I have attempted to make my code **as general as possible**. By this I mean, wherever possible I've tried not to hard-code any values that might need to be changed, if at all the given CSV file changes.

This has helped me to remove many of the possible *corner-cases* that might have risen due to use of hard-coded magic numbers, strings, etc., but also contributed to the verbosity of the code. I believe that can be excused.

Program flow

In the `main()` function, fork the current process:

- Unsuccessful fork
    - Throw error and exit program
- Child process
    - Open CSV file
    - Get file length (number of bytes)
    - Get file content
    - Write out heading ("Section A:")
    - Parse the content, sending `section='A'`
        - Tokenize content with `'\n'` character as the delimiter
        - While the pointer has not reach the file-end
            - Skip CSV header
            - Parse the line
                - Tokenize line using `' '` (space) as the delimiter
                - Get `student_id`, `student_section`
                - Check whether section matches requested section
                    - If no, return
                - In a loop, calculate sum of marks, and number of assignments

        - Calculate (float) average
        - Write out final "`id: average`" to stdout
    - Close the file
    - Exit the process
- Parent process
    - Wait for the child process with `child_pid` to exit.
    - Perform same functions as in child process except with `section='B'`
- Return main function

System calls

This program has used, in total, **eight** system calls.

1. **`fork(void)`**
   Fork the main, parent process and return the PID of the child process. This function is immediately called when the program initiates and does not take any arguments.

2. **`waitpid(pid_t pid, int *wstatus, int options)`**
   Wait for the child process to terminate. The first option we supply is the child process' PID, the second is to save the exit status of the child process and the third are bitwise OR'ed flags that we can supply. The latter two are `NULL` and `0` in our implementation.

3. **`open(const char *pathname, int flags)`**
   Open's the stream for the file described by `pathname` and returns its file descriptor. The `flags` describe the mode to open the file in.

4. **`lseek(int fd, off_t offset, int whence)`**
   Returns the new offset or position the pointer is set at. In this program, I set the `whence` first to `SEEK_END` (EOF) which returns the file length (bytes), and set the pointer back to the start with `whence`=`SEEK_SET`. `offset` in both cases is `0`.

5. **`read(int fd, void *buf, size_t count)`**
   Read the contents of the file described by the `fd` file descriptor upto `count` bytes, and store it in the string `buf`.

6. **`write(int fd, const void *buf, size_t count)`**
   Write out `count` bytes into the stream described by `fd` from the string `buf`.

7. **`close(int fd)`**
   Close the file stream described by file descriptor `fd`.

8. **`_exit(int status)`**
   Terminate the program with exit code `status`.

Errors handled

Each of the following errors prints out the error message to `STDERR_FILENO` (2) using the `write()` syscall and immediately exits the program with a non-zero exit-code (`EXIT_FAILURE`).

- **`"ERR! File could not be opened properly."`**
  The `open()` syscall could not open the file properly in the specified mode.
- "**`ERR! Could not seek in file."`**
  The `lseek()` syscall could seek the read pointer to the requested location.
- **`"ERR! Could not read file."`**
  The `read()` syscall could not read the contents of the specified file.
- **`"ERR! Could not close file."`**
  The `close()` syscall could not close the specified file, supplied with an integer file descriptor.
- **`"ERR! Could not write to specified stream"`**
  The `write()` syscall could not write to the specified output stream, the standard output or the standard error stream.
- **`"ERR! Could not fork parent."`**
  The `fork()` syscall could not fork the parent process successfully.