CSCI 4210 — Operating Systems
Homework 2 (document version 1.2)
Inter-Process Communication (IPC) with Pipes in C

- This homework is due by 11:59PM ET on Wednesday, March 3, 2021
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (`Ubuntu 7.5.0-3ubuntu1~18.04`)

Hints and reminders

As noted in Homework 1, pay close attention to the lower-level details of C, which includes allocating exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. This also includes deallocating dynamically allocated memory (via `free()`) at the earliest possible point in your code, as well as closing any open file descriptors as soon as you are done with them.

Always read and re-read the corresponding `man` pages for library functions, system calls, etc. And continue to make use of the posted code examples, video lectures, and Submitty's Discussion Forum.

As you code, write only a few lines of code, compile, and thoroughly test. In other words, for each bit of code you add, make sure it does exactly what you want and what you expect it to do.

(v1.1) To help ensure the parent and child processes are displaying their output correctly, consider also displaying each process's `pid` (via the `getpid()` system call).

Homework specifications

In this assignment, you will use C to implement the parallel processing of input files using the `fork()`, `waitpid()`, and `pipe()` system calls. You will also use `stat()` and `write()` in each child process, and you will use `read()` in the parent process.

Overall, the parent process creates a child process for each file given as a command-line argument. Each child process attempts to open and read the given file, counting the number of bytes, words, lines, and digit characters in the file. These counts are communicated back to the parent process via a dedicated pipe. Note that there is a pipe for each file and therefore for each child process.

(v1.1) See the lecture notes from February 18 for the minimum level of parallelization required.

To continue your work in mastering the use of pointers, **you are again not allowed to use square brackets** anywhere in your code! If a `'['` or `']'` character is detected, including within comments, you will receive a zero for this assignment. (Ugh!)

Command-line arguments and counts

As noted above, your program must open and read the regular files specified by the command-line arguments, each of which is a filename or path.

The parent process creates a child process for each given filename. It is up to the child process to determine whether the file exists (via `stat()`), open the file if it does exist, read the file contents, calculate the counts, etc.

The counts to calculate for each file are as follows:

- **Bytes.** Count the number of bytes in the file or use `stat()` to obtain this count directly.
- **Words.** Count the number of words in the file, with words defined as containing only alpha characters and having a length of at least one byte. You can assume that each word is no more than `127` bytes long.
- **Lines.** Count the number of lines in the file. If the file does not end in a newline, still count that last line (e.g., if the file contains `"ABCD\nEFGH"` then count this as two lines).
- **Digits.** Count the number of digit characters in the file (i.e., characters `'0'` to `'9'`).

IPC via pipes and the "sentinel" process

Key to this assignment is creating a pipe in the parent process before calling `fork()` for each child process. Since a pipe is unidirectional, all data will be sent from the child process to the parent process.

The protocol for sending the counts is to send the four `int` variables directly in the order shown above. This means that you write each four-byte `int` variable to the pipe in the child process, then read each four-byte `int` variable in the parent process. In total, 16 bytes are sent via each pipe.

To verify this specific protocol is followed, there is one extra "sentinel" child process. Set up the pipe and call `fork()` as you would for the other child processes, then for this sentinel process, use `execl()` to execute the `sentinel.out` executable, which will be available only on Submitty and will send back four counts for you to display in the parent process.

(v1.1) In addition to the `sentinel.out` argument (as `argv[0]`), the file descriptor corresponding to the write end of the pipe is `argv[1]`. Remember that the file descriptor table is copied to the child process as part of the `fork()` call—and this descriptor is available to the process also after the `execl()` call.

As a suggestion, get your code to work first without the sentinel, then add the sentinel in by reusing your working code.

Required Output

Given the parallel processing required for this assignment, some interleaving of the output is expected to occur.

Using the example `lion.txt` file, program output would generally appear as follows:

```
bash$ ./a.out lion.txt
PARENT: Collecting counts for 1 file and the sentinel...
PARENT: Calling fork() to create child process for "lion.txt"
PARENT: Calling fork() to create child process for the sentinel
CHILD: Processing "lion.txt"
CHILD: Calling execl() to execute sentinel.out...
CHILD: Done processing "lion.txt"
PARENT: File "lion.txt" -- 918 bytes, 184 words, 14 lines, 0 digits
PARENT: Sentinel -- 59317 bytes, 2642 words, 1094 lines, 1 digit
PARENT: All done -- exiting...
```

With two input files, example program output would generally appear as follows:

```
bash$ ./a.out lion.txt sleepy.txt
PARENT: Collecting counts for 2 files and the sentinel...
PARENT: Calling fork() to create child process for "lion.txt"
PARENT: Calling fork() to create child process for "sleepy.txt"
CHILD: Processing "lion.txt"
PARENT: Calling fork() to create child process for the sentinel
CHILD: Processing "sleepy.txt"
CHILD: Calling execl() to execute sentinel.out...
CHILD: Done processing "lion.txt"
CHILD: Done processing "sleepy.txt"
PARENT: File "lion.txt" -- 918 bytes, 184 words, 14 lines, 0 digits
PARENT: File "sleepy.txt" -- 71619 bytes, 12500 words, 1092 lines, 15 digits
PARENT: Sentinel -- 59317 bytes, 2642 words, 1094 lines, 1 digit
PARENT: All done -- exiting...
```

In these examples (and in general), interleaving could occur for all lines of output **except for the first and last line**, as illustrated below:

```
<parent process starts>
PARENT: Collecting counts for 2 files and the sentinel...
 |
PARENT: Calling fork()...  <child starts>        <child starts>        <child starts>
PARENT: Calling fork()...  CHILD: Processing...  CHILD: Processing...  <execl()>
PARENT: Calling fork()...  CHILD: Done...        CHILD: Done...        <child ends>
PARENT: File "lion.txt"..  <child ends>          <child ends>
PARENT: File "sleepy.txt"...
PARENT: Sentinel...
 |
PARENT: All done -- exiting...
<parent process ends>
```

A larger example is shown below, including a non-existent file (see the next page). Errors are displayed to `stderr`. And note that specifying the same file multiple times as input is fine.

```
bash$ ./a.out lion.txt sleepy.txt lion.txt lion.txt nosuchfile.txt
PARENT: Collecting counts for 5 files and the sentinel...
PARENT: Calling fork() to create child process for "lion.txt"
PARENT: Calling fork() to create child process for "sleepy.txt"
CHILD: Processing "lion.txt"
PARENT: Calling fork() to create child process for "lion.txt"
CHILD: Processing "sleepy.txt"
PARENT: Calling fork() to create child process for "lion.txt"
CHILD: Processing "lion.txt"
PARENT: Calling fork() to create child process for "nosuchfile.txt"
CHILD: Processing "lion.txt"
PARENT: Calling fork() to create child process for the sentinel
CHILD: Processing "nosuchfile.txt"
ERROR: stat() failed: No such file or directory
CHILD: Failed to process "nosuchfile.txt"
CHILD: Calling execl() to execute sentinel.out...
CHILD: Done processing "lion.txt"
CHILD: Done processing "lion.txt"
CHILD: Done processing "lion.txt"
CHILD: Done processing "sleepy.txt"
PARENT: File "lion.txt" -- 918 bytes, 184 words, 14 lines, 0 digits
PARENT: File "sleepy.txt" -- 71619 bytes, 12500 words, 1092 lines, 15 digits
PARENT: File "lion.txt" -- 918 bytes, 184 words, 14 lines, 0 digits
PARENT: File "lion.txt" -- 918 bytes, 184 words, 14 lines, 0 digits
PARENT: Sentinel -- 59317 bytes, 2642 words, 1094 lines, 1 digit
PARENT: All done -- exiting...
```

Error handling

If no command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered in the parent process, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution.

If an error is detected in a child process, report the error in the same way (i.e., to `stderr`) and return an exit status of `EXIT_FAILURE` to the parent process.

As a reminder, only use `perror()` if the given library or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of q is %d\n", q );
    printf( "here12\n" );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw2.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.

(v1.2) The above technique is no longer recommended for this assignment! Instead, call `fflush()` directly before every call to `fork()` and `execl()` that you make.