CSCI 4210 — Operating Systems
Homework 2 (document version 1.1)
Processes, Pipes, and Files

- This homework is due in Submitty by 11:59PM EST on Wednesday, June 22, 2022
- You can use at most three late days on this assignment
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this assignment, and all submitted 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
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.4 LTS and `gcc` version 9.4.0 (`Ubuntu 9.4.0-1ubuntu1~20.04.1`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate **exactly** the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code.

Make use of `valgrind` to check for errors with dynamic memory allocation and dynamic memory usage. As another helpful hint, close open file descriptors or `FILE` pointers as soon as you are done using them. This includes pipe descriptors, too!

Finally, always read (and re-read!) the `man` pages for library functions, system calls, etc.

Homework specifications

In this second homework, you will use C to implement a multi-process file parser that extracts valid words and sends them to a separate process via a pipe. More specifically, for each command-line argument, create a child process (via `fork()`) to open and read from each file. And for each file, extract valid words and write them to a pipe, which you will connect to a separate executable that is running on Submitty. (This executable is hidden.)

All child processes will be running in parallel, i.e., given n input files, you will have n child processes running in parallel, each child process opening and reading its assigned input file. The parent process must call `waitpid()` for each child process—and the separate executable—before it also terminates.

No square brackets allowed!

To continue to master the use of pointers and pointer arithmetic, **you are not allowed to use square brackets** anywhere in your code! If a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`. (Ouch!)

To detect square brackets, remember you can use the command-line `grep` tool as shown below.

```
bash$ grep '\[' hw2.c
...
bash$ grep '\]' hw2.c
...
```

Command-line arguments and valid words

Each command-line argument specifies an input file. There is no defined limit to the number of input files given. For each input file, create a child process that opens the file, then parses and extracts all words (if any) from the given file.

As with Homework 1, a word is any string of two or more alphanumeric characters, as defined by `isalnum()`. And you can again assume that each word is no more than 127 bytes long. All other characters serve as delimiters. Words are case sensitive (e.g., `Lion` is different than `lion`).

Inter-process communication via the pipe

Before creating any child processes, the parent process should create one pipe via the `pipe()` system call. This results in a "read" descriptor and a "write" descriptor for the pipe. Remember that when you call `fork()`, these descriptors are copied to the child process, i.e., the file descriptor table is inherited by the child process.

The pipe "write" descriptor is used by each child process as follows. When a valid word is detected, the word is written to the pipe with a `'.'` character to mark its end. For example, if valid words `"lion"` and `"mouse"` are extracted, they are sent as shown below in two separate `write()` calls of five bytes and six bytes, respectively.

```
lion.mouse.
```

The pipe "read" descriptor must first be passed as a command-line argument to the `hw2-cache.out` executable. Call `fork()` and `execl()` to execute this hidden executable program as a child process on Submitty. This hidden executable essentially collects words from the pipe descriptor and performs logic similar to that of Homework 1, storing words into a cache structure.

Output from this separate executable will be captured in `hw2-cache.txt` and will be part of the auto-grading in Submitty. When no more data is written to the pipe, the `hw2-cache.out` executable will terminate, which must be acknowledged by your parent process via `waitpid()`.

Required output and exit status values

When you execute your program, you must display a line of output in the parent for each input file, as well as a line of output describing the status of the child process when it terminates. Further, each child process must display the number of valid words encountered.

When each child process terminates, it must return one of the following three values:

- `3` if no valid words were found in the input file;
- `2` if the given input file was not found;
- `EXIT_FAILURE (1)` if some other error occurred;
- `EXIT_SUCCESS (0)` if everything worked as expected.

Given the `lion.txt` example file, you could run your code as follows:

```
bash$ ./a.out lion.txt
```

Below is sample output from the above program execution that shows the format you must follow (**(v1.1)** corrected the output to show number of words in `lion.txt`):

```
PARENT: Created pipe successfully
PARENT: Calling fork() to create child process to execute hw2-cache.out...
PARENT: Calling fork() to create child process for "lion.txt" file...
CHILD: Successfully wrote 174 words on the pipe
PARENT: Child process terminated with exit status 0
PARENT: Child running hw2-cache.out terminated with exit status 0
```

If any child process has an abnormal termination, display one of the following two lines of output:

```
PARENT: Child process terminated abnormally
PARENT: Child running hw2-cache.out terminated abnormally
```

Note that the `hw2-cache.out` executable only outputs to an output file unless an error occurs. Errors are output to `stderr`.

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution. 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( "aaaaaaaaaaaaagggggggghhhh!\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.