CSE 3100 Systems Programming	Fall 2025
Homework #6	

Submit your .c file to gradescope. A Makefile is provided.

Your programs should close unused file descriptors in each process. They should not leave processes running in background, leave zombies behind, or have any memory leaks.

Unless stated otherwise in the starter code, you may call `die()` to exit from the process if a system call fails. For example,

```
die("fork() failed.");
die("open() failed.");
```

Problem 1. (100 points) runpipeline A pipeline is a sequence of external programs chained together to perform a task. The standard output of stage i of the pipeline is fed to the standard input of stage $i+1$. In shells like bash, stages are separated by `"|"`. For example, the following pipeline contains 7 stages and counts the number of occurrences of each word in a text file.[1] The output of the last stage is redirected into file `counts.txt`. `whitman.txt` is provided with the starter code. You can also try this pipeline with other text files (e.g. your C source code).

```
cat whitman.txt | tr -s [:space:] '\n' | tr -d [:punct:] | tr A-Z a-z | sort | uniq -c | \
    sort -nr > counts.txt
```

The seven stages do the following. The command in each stage prints its result to `stdout`, and the output of the last command is redirected to `counts.txt`.

1. Send the contents of the file `whitman.txt` to `stdout`
2. Replace every sequence of consecutive spaces in `stdin` with a single line-feed
3. Delete all punctuation characters from `stdin` and send remaining characters to `stdout`
4. Replace uppercase letters in `stdin` with lowercase letters
5. Sort the lines from `stdin` alphabetically
6. Collapse adjacent matching lines to a single copy preceded by the number of copies
7. Sort the lines from `stdin` in reverse numerical order

In this problem, you will complete three functions in `runpipeline.c` so the program can start a pipeline with the programs specified at the command line. To avoid interference with the shell, pipeline stages are separated with `"--"` instead of `"|"`. To run the above bash pipeline with `runpipeline`, you would run the following command in bash and the resulting `counts.txt` should be the same.

```
./runpipeline cat whitman.txt -- tr -s [:space:] '\n' -- tr -d [:punct:] -- tr A-Z a-z -- \
        sort -- uniq -c -- sort -nr > counts.txt
```

In `runpipeline.c`, the commands for all stages are already stored in an array of `Program` structures, which are defined as follows.

```
typedef struct program_tag {
    char**   argv;      // array of pointers to arguments
    int      argc;      // number of arguments
    int      pid;       // process ID of the program
    int      fd_in;     // pipe fd for stdin
    int      fd_out;    // pipe fd for stdout
} Program;
```

[1] The "\" at the end of first line allows the pipeline commands to continue on the next line; you can also just join the two lines when you try the pipeline in bash.

`argv` is the array of arguments to be passed to an `execv*` function, and `args[0]` is the executable of the program. `argc` is the number of arguments in `argv`. `pid` is the process ID of the child process for this program. If `fd_in` is non-negative, the file descriptor will be used for `stdin` for the program. If `fd_out` is non-negative, it will be used for `stdout` for the program.

Note that `runpipeline` does not redirect the input or output for the pipeline itself (that is, the input of the first command or the output of the last command). If needed, the redirection can be set on `runpipeline` by the shell. So the first command in the pipeline may have redirected `stdin` and the last one may have redirected `stdout`, but this should not need to be handled in `runpipeline`.

The functions to be completed in this problem are listed below. Information on how the functions are used and what they should return can be found in the starter code.

```
void prepare_pipes(Program *programs, int num_programs);
int start_program(Program *programs, int num_programs, int cur);
int wait_on_program(Program *prog);
```

There are a few ways to create pipes. One can create all necessary pipes for the entire pipeline in `prepare_pipes()`, or create pipes in `start_program()` just before they are needed. If you choose to create pipes in `start_program()`, you do not need to add any code to `prepare_pipes()`.

`start_program()` starts a program indexed by `cur`. The function performs necessary redirection for the program. Make sure unused file descriptors are closed.

`wait_on_program()` waits for the specified process to finish and returns the exit value of the process.

Dealing with many pipeline stages may look scary at the beginning. However, if we start with two stages, and go on to three stages, four stages, and more, we could find a pattern and perform operations in a loop. Then, increasing the number of stages does not increase complexity.

Several programs can be helpful for testing. For example, `tee` saves `stdin` to a file, which allows us to examine the data stream at the middle stages.

The following examples show how to create pipelines with various numbers of stages. To check if your code is producing correct result or behaves correctly, run the same pipeline in bash (and use `"|"`, instead of `"--"`, to connect stages). Note: you may find the test cases for this homework difficult to decipher. Comparing the results of `runpipeline()` to the resutls of the same pipeline in bash will be very helpful for debugging your code, as will `tee` and `checkof`.

```
./runpipeline echo 'Hello, world!'
./runpipeline echo 'Hello, world!' -- wc
./runpipeline echo 'Hello, world!' -- cat -- wc
./runpipeline echo 'Hello, world!' -- cat -- cat -- wc
./runpipeline ls -- cat -- tee t.out -- wc
./runpipeline cat -- cat -- tee t.out -- cat -- wc
```