CIS*3110 Assignment 1
Parallel computation using processes
Due date: Friday, February 16, 11:59pm,
Weight: 15%

1. Description

For this assignment you will write a parallel program that takes a list of file names as input and produces a new set of files, which contain the english alphabet histograms for each of the input files.

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details.

Histograms are simply counts - in this case, counts of alphabet characters in the file. For example, given a file with contents
`hello world`
the characters `a` thorough `d` appear 0 times, `e` - 1 time, etc..

Your program will consist of several processes - a parent and multiple children. These processes will communicate using pipes and signals. The structure of your program is described below.

Parent process

- The parent process will accept multiple file names as command line arguments. The parent must be able to take any number of files as inputs. If no input files are provided through the command line, your program must display an error message and exit.
- The parent program registers a signal handler for `SIGCHLD`
- For every file, the parent will fork a child and pass the file name to it. How you do this is up to you - you can use a pipe, a variable that gets initialized by the parent and copied by the child, etc. Each child also gets a pipe for sending the data back to the parent.
- The parent then goes in a loop and does nothing, avoid busy waits. Terminating children are dealt with by the `SIGCHLD` handler:
  - The process id of the terminated child is retrieved using `waitpid(-1, &child_status, WNOHANG)` - more on this later
  - It must display a message indicating it caught a `SIGCHLD`, and specify the child's PID.
  - The signal handler function checks how the child terminated:
    - If the child exited normally, the parent reads the character counts from that child's pipe, then closes the pipe's read end. The character counts are saved in the file called `filePID.hist` where PID is the process ID of the corresponding child. The format for the output file is stated below.
    - If the child terminated abnormally - either exited with a value other than 0, or was killed by a signal - the signal handler does not read the data, since there's nothing to read.

    - The examples discussed in class and posted on the course website showed you how to get the exit status of a process.
    - The macros `WIFSIGNALED` and `WTERMSIG` will help you figure out if the child was killed by a signal (and which one), and the function `strsignal` will provide the name of the signal given its code.
- Once the parent knows that all the children have terminated, it exits.

In other words, the parent handles the terminating children and the data they send asynchronously. Instead of using a loop with `wait`, it catches every `SIGCHLD` generated by a terminating child process, checks the termination status, and reads the data from the children that terminated normally.

It is possible that some children will terminate nearly simultaneously. This can be a problem - when multiple child processes terminate simultaneously and hence multiple `SIGCHLD`s are raised, the parent may get delivered only the first one of them and the others would be lost, since the kernel does not queue signals.

It is somewhat unlikely in our case, since each child would be working on a file of different lengths. However, to avoid this problem, we will use a non-blocking version of `waitpid` with the `WNOHANG` flag. In addition, each process will sleep for a few seconds after it completes the task. These steps will prevent the race condition.

Child processes

- Once the child gets the file name, it opens it and computes the histogram.
    - If the file fails to open, the child closes the write end of the pipe and terminates with the return / exit value 1.
- The calculated histogram is sent through a dedicated pipe to the parent as an array. Make sure you get the array size right, since you will be sending array of multi-byte values (e.g. ints).
- The child goes to sleep for *10+3*i* seconds, where *i* is the index of the spawned child, starting at 0. So the child process that was forked first would sleep for 10 seconds, second - 13 seconds, third - 16 seconds, etc..
- The child then closes the write end of the pipe and terminates with exit value 0.

You will need to make the array of pipes global and keep track of what pipe corresponds to what child PID. This will give the signal handler access to the array of pipes and allow the signal handler figure out which pipe to read from depending on which child has terminated. How you do this is up to you. You will not be penalized for use of global variables - they are often used in low-level systems programming code out of necessity.

Histogram calculation

- Read the file into an array of chars.

- Go through every character in the array and calculate the number of occurrences of every letter. You can assume that the text is written using the 26-letter English alphabet. All other characters must be ignored. Upper- and lower-case forms of the same letter must be treated the same - e.g. 'A' counts as an occurrence of 'a'.
- Be careful with the character array size - **do not** hardcode it! It must be determined at runtime and you must malloc the array in the main process and the necessary arrays in the additional worker processes. Remember to free all data array when you are done.

Once the parent has received the histogram - which the child must send through a pipe as an array - the parent will save the histogram to a file using the following format (one letter count per line, each line has the letter and its corresponding count):

```
a 12
b 0
c 17
d 3
```
...

Special inputs

The command line arguments passed to the parent process are treated as file names - with one exception. If a command line argument is the string `SIG`, the corresponding child will not receive the filename - instead, the parent will send `SIGINT` to that child using the `kill` function. The child does not need to catch it - it should just terminate, using the default `SIGINT` disposition.

For example, given the command:

```
A1 file1.txt SIG file2.txt
```

The child process 0 (the first fork) would be given `file1.txt`, child 1 would be sent `SIGINT`, and child 2 would be given `file2.txt`

Using signal functions

To use functions related to signals, we need to tell the C compiler to use the 2008-09 POSIX standard. You can put the statement `#define _POSIX_C_SOURCE 200809L` as the very first line in your code, above all other `#include` or `#define` statements.

Alternatively, you can pass this macro value to your code through the compiler:

```
gcc -D_POSIX_C_SOURCE=200809L file.c
```

You also need to include the relevant headers: `unistd.h`, `signal.h`, `string.h`, etc.. The first few lines of your code can look like this:

```
#define _POSIX_C_SOURCE 200809L
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

Compilation

Include a `Makefile` that compiles all your source code. The `Makefile` must compile your code with the flags `-Wall -g -std=c11`have the following targets:
- `A1`: the main executable fine implementing the assignment functionality
- `clean`: clean: the target that deletes the executable, as well as any other temporary files you may have created when compiling the executable

You are welcome to use the Makefile provided in the code examples on the course website as a starting point.

Also, please include a `README` indicating known issues / bugs, as well as whatever additional instructions we may need to run your code.

2. Evaluation

Your code must compile, run, and have all of the required functionality implemented. It must compile and run with no errors on the CIS*3110 Docker image provided on the course website and discussed in class.

Any compiler errors will result in the **automatic grade of zero** for the assignment. Compiler warnings will result in mark deductions.

Your code will be tested with several text files. Marks will be deducted for:
- Incorrect and missing functionality, including a missing makefile
- Deviations from the assignment requirements
- File descriptions that you opened yourself and that are still open when your program terminates - `valgrind` will be used for checking this
- Memory leaks and memory errors - these will also be checked for using `valgrind`
- Run-time errors, including infinite loops, crashes, race conditions, etc.
- Failure to follow submission instructions

Your code will be compiled with the `-Wall`, `-g`, and `-std=c11` flags. Make sure your code compiles with no errors and no warnings to avoid mark deductions. You can create a makefile for your code to speed up the compilation. If you do, the makefile must use the above flags.

3. Submission

Submit your source code and makefile as a Zip file using Moodle. The name of your file must me l`oginName.zip`, where the `loginName` name is your standard UofG login. Make sure the archive a Zip format. As stated above, failure to follow these instructions will result in a mark deduction.

The assignment will be marked using the standard CIS*3110 Docker image provided on the course website and discussed in class. We will download and install a fresh image, create a container, and use Docker to run and grade your code. It will be compiled using the Makefile you have provided and we will execute the created file. Make sure you test your code accordingly.