A3 - Parallel Programming with fork and pipes

Submit Assignment

Due Mar 31 by 10pm **Points** 100 **Submitting** a file upload
Available Mar 6 at 12pm - Apr 3 at 11:59pm 28 days

Due Date: 10:00 pm Tuesday, March 31, 2020
Worth: 17% of your final grade.

Introduction:

Nowadays most search engines also provide some support for image retrieval, which allows you, for example, to search the web for images that are similar to a given image. If a search engine used a single process to search a large number of images to find the most similar one it would take a very long time to get the results. For this assignment, you will write a parallel program using fork and pipes to search for similar images. (However, you won't likely see any performance difference between using one process and many because the number and size of the images we are giving you are so small compared to the web-scale operations that search engines do.)

The main program will be provided with the name of a ppm image file and the name of the directory where we want to look for similar images. The program will search the directory for sub-directories and fork a new process for each sub-directory. Each child process will compare the original image to all the images in the sub-directory that it was assigned. It will then send to the parent process information on the most similar image in its sub-directory.

Task 1: Helper functions for image comparison

You will start by implementing two helper functions that will be needed for comparing two images.

-
  ```
  read_image
  ```

  This function expects the name of a ppm P3 image file as an argument and will build a `struct` that represents this image `worker.h` already contains the definition for an Image struct (representing an image) and a Pixel struct (representing an individual pixel). An Image includes width, height, max value and a pointer to pixels. To properly store the pixels for an image you will need to dynamically allocate an array of the proper size and make sure that the pixel pointer in the image struct points to it. `read_image` will make sure that the file starts with the magic number `P3`. If the file does not start with this magic number, `read_image` will return NULL. If it does start with the correct magic number the function assumes that the file contains a properly formatted valid P3 image and does not need to do any further error checking.
  We are providing a function `print_image` in the starter code that you can use to print an image based on an image struct that you created.

-
  ```
  compare_image
  ```

  This function takes an Image struct and the filename of a P3 image as an argument. It will compute the similarity between the image in the struct and the image in the file. In reality, determining how similar two images are is a complex problem, which is in fact an active area of research. We will use a simple metric based on the Euclidean distance between two pixels and for simplicity work only with images that have the same dimension (same width and height). The starter code already contains a function `eucl_distance` that takes two pixels as input and returns their Euclidean distance. The smaller the returned value is the more similar the two pixels are. `compare_images` will return the average Euclidean distance between the pixels in the two images, i.e. it will compute all the pairwise Euclidean distances between the corresponding pixels in the two images and then return the average of all these distances. For example, if the first image consists of four pixels p1, p2, p3, p4 and the second image consists of four pixels pa, pb, pc, pd then the function will return

```
 (eucl_distance(p1,pa) +  eucl_distance(p2,pb) + eucl_distance(p3,pc) + eucl_distance(p4,pd))/4
```

This approach only works if the two images have the same dimensions. If they don't have the same dimension, then `compare_images` returns `FLT_MAX` (the largest float number) as the distance between the two images.

Task 2: A solution based on only one process

Before writing the parallel version of the program we will start by completing a program `one_process` that uses only a single process, so you can make sure that the basic functionality of your image retrieval is working before diving into multiple processes. The starter code already contains a file `one_process.c` with a basic skeleton for the main program. When completed this program will be run from the command line as follows:

```
./one_process -d DIR_NAME IMAGE_FILE
```

`IMAGE_FILE` is the name of the image file that we compare all other files to. The -d option provides the name of the directory, where we search for similar images. This option is optional and the program will simply search the current working directory if it is not provided. When completed the program should look for images in all the sub-directories of `DIR_NAME` and find the one that is most similar to `IMAGE_FILE`. In the final parallel version of your program, each sub-directory will be handled by a separate process. But for this task, `one_process` will handle all the sub-directories itself.

The starter code for `one_process.c` already takes care of processing the command line arguments. It also implements the functionality to open the directory `DIR_NAME` and find all the sub-directories inside `DIR_NAME`. Right now it just prints the name of each sub-directory it finds. You want to change the implementation so that instead it will call the function `process_dir` for each sub-directory.

`process_dir` is a function you will need to complete. It searches the directory specified in its first argument for image files. It will use the helper function `compare_images` to compare each file in the directory to the image `img` that was passed to it as an argument and determine, which one is most similar. It will store its results (i.e. the information for the most similar image that it found) in a struct called `CompRecord`, which contains the name of the file and the distance it has to the original image (as computed by `compare_images`). Once it has processed all images in the directory it will return the `CompRecord` for the most similar one. If the sub-directory contains no image files it will return a CompRecord with a distance of `FLT_MAX`.

The main function will keep track of the `CompRecord` with the smallest distance (highest similarity) to the original image and print the information about it to the screen. The print statement is already included in the starter code. Please don't change its format.

In the parallel version of the program `process_dir` will be run by a child process that the parent process forked. Since process_dir will run in a separate process it will need a different way to pass back the `CompRecord` than using the return value of the function. That is the purpose of the third argument of `process_dir`, which is an integer `out_fd`. This argument is a file descriptor that will correspond to the writing end of the pipe the parent and child will share. In your single process program, you can experiment with writing to this file descriptor by calling process_dir with `STDOUT_FILENO` (the file descriptor corresponding to stdout) as its third argument.

In order to complete `process_dir` you will need to know how to find all the files in a directory. To do this you can follow the example we have already provided in the main program for iterating over all the sub-directories inside a directory. It uses the **opendir (https://linux.die.net/man/3/opendir)** and the **readdir (https://linux.die.net/man/3/readdir)** functions and the **C library function stat (https://linux.die.net/man/2/lstat)**. stat fills in a struct stat with information about the file you ask about. The `st_mode` field of the `struct stat` is particularly interesting, since it can tell you whether a file is a directory or a regular file. A series of macros help you extract data from the field: `S_ISDIR`, which is used in the main function, can be used to check whether something is a directory and `S_ISREG` checks for a regular file.

Task 3: A parallel version of the image retrieval program

You have now set up all the core functionality for image retrieval. The only piece missing is to parallelize the processing of all the sub-directories. Start by creating a new program called `image_retrieval.c`, that is a copy of your `one_process.c`. Now modify it such that the main function forks a new process for each sub-directory. Before doing so it needs to set up a pipe that the child can use to write back the CompRecord for the most similar image in its sub-directory to the parent. The parent will read all `CompRecords` it retrieves from its children and print the information for the most similar one to the screen. Remember to also add a rule to your Makefile for the new `image_retrieval.c` program.

Starter Code

We are providing a Makefile and three source code files as starter code, which you are expected to modify:

- Makefile
- worker.h
- worker.c
- one_process.c

You will also create and submit a new file called `image_retrieval.c`, which is a modification of `one_process.c` and implements the parallel (i.e. multi-process) solution to the problem.
Make sure to submit your version of all of the above files (including the Makefile) and your newly created `image_retrieval.c`.

Important Hints

Here are a few things to remember as you implement and test the functions:

1. When implementing `compare_images` you have two choices for how to handle the image in the file that is provided by the second argument. You could either create a `struct` based on the file contents using `read_image` or you could use the approach of A1 where you process the file and do the comparison as you are reading it, without storing it in a struct. For this assignment we will accept both solutions, however if you do create a struct remember to free all memory for it afterwards, since your program might be comparing a large number of large images, causing it to run out of memory (and losing marks from the assignment).
2. When working on your implementation of `process_dir`, you can test it without setting up pipes by passing it `STDOUT_FILENO` as a third argument, so you are writing to the screen, rather than a pipe. Just remember that `write` writes in binary format, so if you write a number it will write the binary representation of the number rather than the corresponding ASCII characters, which will look garbled on the screen. But if you write strings (e.g. the filename of the `CompRec` struct) they will show up ok.
3. When you first compile the starter code, you will get a couple of warning messages about uninitialized/unused variables. These only occur because the

code is not yet complete.

4. Test thoroughly before submitting a final version. We are using automated grading tools, so your program must compile and run according to the specifications. In particular, your program must produce output in the specified format.
5. As always, build up functionality one small piece at a time.
6. Check for errors on every system or library call that is not a print. Send any potential error messages to stderr.
7. Be very careful to initialize pointers to NULL and to make sure that you copy any strings that you use.

Submission and Marking

We are using automated grading tools to provide functional feedback, so it's important that your submission be fully submitted and compile cleanly.

Your program must compile on a CS lab machine, so please test it there before submission. We will be using gcc to compile program with the flags `-Wall and -std=gnu99`, as demonstrated in the `Makefile`.

Your program should not produce any error or warning messages when compiled. As with previous assignments, programs that do not compile will receive a 0. Programs that produce warning messages will be penalized.

You will be submitting your assignment via Quercus.