ITP 30002-{01, 02} Operating System, Spring 2021

Homework 3. `pfind`: parallel text search program using multiprocessing

(due on 11 PM, 16 Apr (Fri), 2021)

Shin Hong, Handong Global University

1. Overview

In this homework, you are asked to design and construct a text searching program `pfind` which find all lines of given keywords from all text files under a certain directory[1]. For time performance, `pfind` is designed to run one or multiple children processes as *workers* which conduct text searching on multiple directories concurrently. The main process communicates with workers via named pipes (i.e., `fifo`) to give jobs and then receive the results.

Your task is to understand the skeleton of the design given in this assignment description (Section 2) and develop the details (e.g., communication protocol), and then implement the program accordingly by using proper standard system libraries.

The submission deadline is **11 PM, 16 April (Friday)**. Your submission must include all source code files and a presentation video (Section 5). The presentation must include the explanation on the program design and demo of your program with different execution scenarios.

2. Program Design

2.1. Functionality

`pfind` is a text search program that finds all text lines that satisfy specified conditions under a given directory. The usage of `pfind` can be described as follows:

```
$ ./pfind [<option>]* <dir> [<keyword>]+
```

`pfind` receives the path to a directory for which searching will be conducted (i.e., <*dir*>) and a non-empty list of keywords (i.e., one or more <*keyword*>'s). For given directory path and keywords, `pfind` must examine all text files under the given directory to find every line that contains all given keywords at the same time. Once `pfind` detects a line that meets the condition, it prints out the path, the filename, the line number and the content of the line to the standard output. For example, when `pfind` is executed to find all lines that contain two keywords "`regcomp`" and "`char`" together from `/usr/local/llvm` of the peace server, it would find the two matching cases and display the found lines as follows:

```
$./pfind /usr/local/llvm regcomp char

lib/Support/regex_impl.h:97:int
llvm_regcomp(llvm_regex_t *, const char *, int)

lib/Support/regcomp.c:165:llvm_regcomp(llvm_regex_t
*preg, const char *pattern, int cflags)
```

Note that the scope of search is limited to regular text files, thus, `pfind` must not concern any non-regular or binary files. In traversing subdirectories, `pfind` must skip symbolic links since symbolic links may form cycles in the directory hierarchy.

`pfind` may receive a list of option flags to configure the program behaviors. The options for `pfind` should be supported as follows:

`-p <N>`	defines the number of the worker processes (i.e., child processes) as `N` which be greater than 0 and less than 8. The default value is 2.
`-c`	match keywords in the case-insensitive way. The default behavior is the case-sensitive matching.
`-a`	print the absolute path of a file. The default behavior is to print a relative path.

When the searching is completed, `pfind` must print the summary of the searching result to the standard output. A summary must include the total number of found lines, the number of explored files, the number of the explored directories, and the total execution time.

The user can press `Ctrl+C` to forcefully terminate the program execution before the searching is completed. In such a case, `pfind` must print the summary on the searching up to the moment when the signal is given, and terminate all processes quickly.

2.2. Manager-workers architectures[2]

`pfind` is designed to have a manager-workers architecture, as shown in Figure 1 to be able of distributing a searching job over multiple concurrent processes. The main process (i.e., first process) of `pfind` roles as the *manager* which takes the following activities:

- receive commands from the user, and
- spawn child processes as workers, and
- assign tasks to the workers, and
- receive the task output from the workers, and
- print the search summary, and
- terminate the workers before the program termination.

For parallelization, `pfind` divides a searching job into multiple *tasks* each of which is to explore a single directory. A worker takes on one task at a time, thus, multiple tasks can be processed concurrently on multiple workers. The manager and the workers communicate via two pipes: left one for passing tasks from the manager to the workers, and right one for receiving task results from the workers to the manager.

A task is defined as the path of a target directory, which can be either the search target specified by the user (i.e., <*dir*>), or one of its subdirectories. For the directory example of Figure 2, there will be six tasks because there exist total five subdirectories under the search target (i.e., `dir1`). The activities of a task are (1) to find all lines satisfying the given searching conditions from all regular text files in the corresponding directory (not its subdirectories), and (2)

[1] `pfind` is similar with Unix utilities `grep` and `find`.

[2] its old term was master-slaves but replaced for the racial meaning.

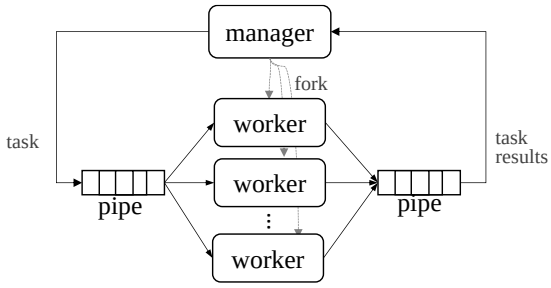

Figure 1. Manager and workers processes

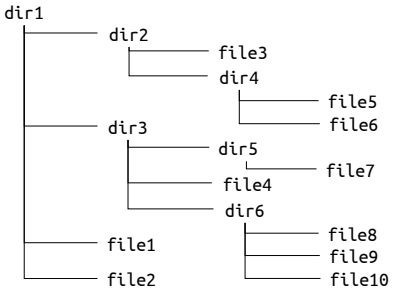

Figure 2. Search target directory example

find all immediate subdirectories to define new tasks. For example, the task for `dir1` is to explore text files `file1` and `file2` and identifies two immediate subdirectories `dir2` and `dir3`.

Worker processes must be created as soon as `pfind` starts. A worker process can take a task at a time from the manager. Once a task is dispatched, the worker proceeds to search for the matching lines from the text files. For each line matching, the worker prints out the result to the standard output. After searching the text files, the worker lists up immediate subdirectories. Subsequently, the worker transfers the information on the task result to the manager to complete the task. From given task result, the manager updates the searching statistics and creates new tasks based on the given list of the subdirectories. After completing a task, a worker starts to take a remaining task if there exists. A worker process may be idle if no task remains.

3. Implementation

This section gives the requirements and the useful information on implementing `pfind`.

Your program must use `fork` to create children processes. You cannot use multithreading in this homework, thus, the manager process (i.e., main process) and each of the worker processes must be single-threaded. The main process uses a signal mechanism to receive a termination command from the user. Also, the main process uses signals to terminate worker processes. The signal handler of the main process must work for printing out the searching summary before existing the program execution. Make sure that no children processes remain after the termination.

There must be two named pipes (i.e., `fifo`), `tasks` and `results`, for transferring task information and task result information, respectively. `pfind` must create these two pipes as a program execution starts and removes them before the termination. Note that at most one process must be able to open `tasks` or `results` at a time. To make the manager process and the worker processes communicate via the two named pipes, you need to design protocols that defines the rules and the structures of messages. A tip is that a message should be started with 4 bytes that gives the length of the content. If so, a reader first reads 4 bytes to determine how much bytes must be read for constructing a message. The details of these protocols should be elaborated in your presentation.

It is possible to retrieve the list of files in a directory by using `readdir`. It is possible to determine whether a file (or directory) is regular or not (or symbolic link) by checking into the `dirent` structure. Checking if a regular file is text is trickier. An easy solution is to use a Unix utility `file`: if the output for a given file mentions "`ASCII`" or "`text`", the file can be considered as a text file.

For there are multiple workers running concurrently, the printings of found lines from multiple workers can be mixed (or intertwined). You do not have to synchronize concurrent worker processes in this homework.

In addition to the aforementioned requirements, your program must reject invalid inputs, handle various error cases to prevent fatal errors, return proper error messages to the user.

4. Your Tasks

This section summarizes what you need to do for this homework and gives information about the points of evaluation.

You are asked to write `pfind` in the C programming language. Your program may consist of one or multiple C source code files. You must include a build script (e.g, `Makefile`) and a `REAMDE` document that instructs how to build and run your program in your homework submission. Note that your program will be tested on `peace.handong.edu`, thus, you are recommended to check if your program works well with the server.

The other obligation of this homework is recording a presentation on your result. Your presentation must include the followings:

- Explanation of the program design, including the protocol of the communication between the manager and the worker processes. It is highly recommended to give details using slide or whiteboard.
- Code review of the program source code.
- Demonstration of your implementation. You must devise different test scenarios to evidence that your program satisfies the various requirements. It will be evaluated by how comprehensively your demonstration covers the given requirements posed in this homework. As a part of the demo, show that multiple worker processes are running concurrently.
- Your idea of improving `pfind`. Tackle the limitation and/or the shortcomings of the current design and suggest ways to resolve these issues.

Your presentation must not exceed 10 minutes. You must use English in the presentation slide and narration. For supplementation of your presentation, you may put subtitle in Korean if you want. Evaluation will be primary based on your presentation, thus, carefully address all key points of your results in the presentation video. And be careful not to overstate your results. It will be treated as a fraudulent attempt if your statement and result are not consistent.

5. Submission

Submit an archive (e.g., as a `tar` or `zip` file) of all source code and results at a submission repository entitled with "Homework 3" in Hisnet by **11:00 PM, 16 April (Fri)**. The archive must contain (1) README file with a link to your presentation video (a link to YouTube, Vimeo, or Google drive), (2) all source code files and build scripts needed for reproducing what you reported in the video.