Operating Systems (0368-2162)

Parallel File Find Assignment

Due date (via moodle): **January 12th, 2020, 23:59**

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. Feel free to discuss concepts and ideas with others, but **you should never observe, or have possession of, another student's code** (from this or previous semesters).

Students violating this policy will receive a 250 grade ("did not complete course duties") in the course. *If you're under pressure, contact the course staff or even don't submit, rather than cheat and risk having to repeat the course.*

1 Introduction

The goal of this assignment is to gain experience with threads and filesystem system calls. In this assignment, you will create a program that searches a directory tree for files whose name matches some search term. The program receives a directory D and a search term T, and finds every file in D's directory tree whose name contains T. The program parallelizes its work using threads. Specifically, individual directories are searched by different threads.

2 Assignment description

Implement the following program in a file named `pfind.c`. The following details the specification of the program.

Command line arguments:

- `argv[1]`: search root directory (search for files within this directory and its subdirectories).
- `argv[2]`: search term (search for file names that include the search term).
- `argv[3]`: number of searching threads to be used for the search (assume a valid integer greater than 0)

You should validate that the correct number of command line arguments is passed and that `argv[1]` is a searchable directory.

The flow:

1. Create a FIFO queue that holds directories. (See more details about this queue below.)
2. Put the search root directory (where to start the search, specified in `argv[1]`) in the queue.

3. Create n searching threads (as per the number received in `argv[3]`). Each searching thread removes directories from the queue and searches for file names containing the search term specified in `argv[2]`. The flow of a searching thread is described below.

4. The program exits in one of the following cases: (1) there are no more directories in the queue and all searching threads are idle (not searching for content within a directory), (2) all searching threads have died due to an error, or (3) all searching threads have exited gracefully after a `SIGINT` is delivered. The exit conditions and handling are detailed below.

Flow of a searching thread:

1. Dequeue the head directory from the FIFO queue. If the queue is empty, sleep until it becomes non-empty. **Do not** busy wait (wasting CPU cycles) until the queue becomes non-empty.

2. Iterate through each file in the directory obtained from the queue:

   (a) If the file is one of the directories `"."` or `".."`, ignore it.

   (b) If the file is a directory, do not match its name to the search term. Instead, add that directory to the tail of the shared FIFO queue (which, if some searching threads are sleeping waiting for work, should wake up one of them).

   (c) If the file name contains the search term (as specified in `argv[2]`, **case-sensitive**), print the path of the file (starting from the root search directory and including the file's name) to `stdout` using `printf()`. Print **only** the path followed by `\n` with no other text. For example, if the program is invoked with `argv[1]=foo` and `argv[2]=bar` it will print paths of the form `foo/some/sub/dir/zanzibar.txt`.

3. When done iterating through all files within the directory, repeat from 1.

IMPORTANT: The searching thread flow above is high-level. It doesn't describe how to detect when all directories have been processed and the program should exit. It also doesn't describe how to implement the signal handling requirements described next. These are problems that you need to solve yourself.

Signal handling:

- The program should not terminate immediately upon a receiving a `SIGINT`.

- Upon receiving a `SIGINT`, each searching thread should exit gracefully. For a thread to exit gracefully, read about and use `pthread_cancel()` as well as any additional functions needed for its use (e.g., `pthread_testcancel()` to force cancellation as soon as possible when a cancellation request is registered).

Error handling & termination:

- If an error occurs in a searching thread, print an error message to `stderr` and exit that thread, **but don't exit the program**.

- The program should exit when one of the following occurs:

  – All searching threads have exited due to an error. The exit code should be **1**.

  – All searching threads have exited gracefully due to `SIGINT` being delivered. The exit code should be **0**.

  - There are no more directories in the queue and all searching threads are done searching (idle). The exit code should be **0**.

- Before exiting, the program should print how many files were found, as follows:
  - If exiting because of a `SIGINT`, use **exactly** the following `printf()` format string:
    `"Search stopped, found %d files\n"`
  - Otherwise (error or normal exit), use **exactly** the following `printf()` format string:
    `"Done searching, found %d files\n"`

- No need to free resources (including threads) upon program exit.

Correctness requirements:

- The program should be thread-safe. For example, the same directory should not be searched by different threads. Note that `printf()` is thread-safe; there's no need to protect it with a lock.
- Make sure no deadlocks are possible in your program.
- Threads should run in parallel, i.e., do not turn the entire flow of a searching thread into a critical section protected by a lock. Only accesses to shared data should be synchronized with locks.
- The queue used to distribute directories to searching threads must have the following properties:
  1. Suppose a directory is inserted into an empty queue that has k threads sleeping (waiting for work). Then this directory must get processed by one of those k threads.
  2. A thread may be sleeping (waiting for work) only if the queue is empty.
- The number of matching files printed when the program exits must be equal to the number of file path names printed during the program's execution.

IMPORTANT: You can assume that the directory tree does not change while the program is running.

3 Relevant functions & system calls

1. Learn about and use the following: `pthread_create()`, `pthread_join()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_exit()`, `pthread_cancel()`, `pthread_mutex_lock()/unlock()`, `readdir()`, `stat()`.

4 Submission instructions

1. Submit just your `pfind.c` file. Document your code with explanations for every non-trivial part of your code. Help the grader understand your solution and the flow of your code.
2. The program must compile cleanly (no errors or warnings) when the following command is run in a directory containing the `pfind.c` file:

```
gcc -O3 -D_POSIX_C_SOURCE=200809 -Wall -std=c11 -pthread pfind.c
```