Operating Systems (0368-2162)

Parallel File Find Assignment

Due date (via moodle): **December 23rd, 2021, 23:59**

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. You may discuss concepts and ideas with others, but **you must program individually**. **You should never observe another student's code**, from this or previous semesters.

Students violating this policy will receive a **250 grade in the course** ("did not complete course duties").

REMEMBER TO SUBMIT THE THEORETICAL PART TOO

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 the directory specified in `argv[1]` can be searched (see below for how to check this).

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. After all searching threads are created, the main thread signals them to start searching.
5. The program exits in one of the following cases: (1) there are no more directories in the queue and all live searching threads are idle (not searching for content within a directory), or (2) all searching threads have died due to an error. The exit conditions and handling are detailed below.

Flow of a searching thread:

1. Wait for all other searching threads to be created and for the main thread to signal that the searching should start.
2. Dequeue the head directory from the FIFO queue. If the queue is empty:
   - Wait until the queue becomes non-empty. If all other searching threads are already waiting, that means there are no more directories to search. In this case, all searching threads should exit.
   - **Do not** busy wait (wasting CPU cycles) until the queue becomes non-empty.
3. Using `readdir()`, iterate through each directory entry (dirent) in the directory obtained from the queue:
   (a) If the name in the dirent is one of `"."` or `".."`, ignore it.
   (b) If the dirent is for a directory (check this with `stat()`), do not match its name to the search term. Instead:
      - If the directory can be searched (defined below), 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).
      - If the directory can't be searched, print the following message and continue (don't treat this case as an error!). Use **exactly** the following `printf()` format string:
        `"Directory %s: Permission denied.\n"`
        In this message, use the full path to the directory (starting from the root search directory).
      - A directory can be searched if the process has both read and execute permissions for it. You can `opendir()` the directory to check this; assume that if `opendir()` fails, the directory can't be searched.
   (c) If the dirent is not for a directory and its 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`.
   You can assume the maximum length of a path is `PATH_MAX` (defined in `<limits.h>`).
4. When done iterating through all files within the directory, repeat from 2.

IMPORTANT: The searching thread flow above is high-level. For instance, it doesn't describe how to detect when all directories have been processed and the program should exit. This is a problems that you need to solve yourself.

Error handling & termination:

- If an error occurs in the main thread, print an error message to `stderr` and exit the program with exit code 1.
- 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, or an error occurs in the main thread.
  - There are no more directories to search.
- When exiting, the exit code should be **0** if and only if no thread (searching or main) has encountered an error. Otherwise, the exit code should be **1**.
- Before exiting, the program should print how many files were found during its execution. Use **exactly** the following `printf()` format string:

  `"Done searching, found %d files\n"`
- If exiting because of an error in the main thread that occurs before all searching threads have been created or after all searching threads have died, no need to print anything besides the error message. In any other error case, the program should print how many files were found before exiting, as described above.
- No need to free resources (including threads) upon program exit.
- No need to check for errors in calls to `pthread_mutex_*` and `pthread_cond_*` functions.

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 resource 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 satisfy the following:
  1. Work should be distributed to sleeping threads in FIFO order. Formally: If the queue is empty and there are k threads sleeping, then the next k directories added to the queue should be handled by these threads according to the order in which they went to sleep (i.e., the first directory added should be handled by the first thread that went to sleep, etc.).
  2. A thread may put itself to sleep (to wait for work) only if the queue was empty when the thread arrived at the queue.
- 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 (possibly) use the following: `pthread_create()`, `pthread_join()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_exit()`, `pthread_mutex_lock()/trylock()` `opendir()`, `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
   ```

   IMPORTANT: If you are running on nova, use `gcc-5.3.0` instead of `gcc`, because the default `gcc` version on nova doesn't support C11 atomics. On the course VM, you can use the default `gcc`.

3. Did we mention not to forget to also submit the theoretical part?