CS 537 - Introduction to Operating Systems

Project 6: Threads and Synchronization

Aman Rakesh Chadha
(9068354597)

1 Objectives

The goals for this project are to:

1. Get experience programming with threads.
2. Get experience with thread synchronization.

2 Desktop search

In this project, you will implement a very simple desktop search engine. Your code will scan files and add each word in each file to an index. The index contains can be searched by word and contains the file name and line number where the word shows up. The search interface allows a user to type in a single word, and spits back a list of files containing that word.

Your code will consist of three pieces:

1. A file system scanner that reads in file names
2. An indexer that find all the words in a file and add them to a hash table
3. A search interface that allows you to type in a word and get back a list of files containing that word

Each of these components should run concurrently as a separate thread, so you can scan files and index them as you query the index.

File system scanner

The file system scanner can be quite simple: it just reads in a list of files to scan. A list of files in a directory can be generated with this command:

```
find . -type f > list-of-files.txt
```

This fills in the file list-of-files.txt with all the files within and beneath the current directory. You can assume that file names will be less than 511 characters (defined as MAXPATH in the provided code).

The scanner is a producer, as in a producer/consumer code. It should write file names to be scanned to a bounded buffer.

Indexer

There should be one or more indexer threads in your program. The scanner should read feil names from a the scanner thread. An indexer takes as input a file name, opens that file, and then reads all the words in the file. You can use @strtok_r@, separating on whitespace (spaces, tabs, newlines) to find words. For each word it finds, the indexer adds an entry to a hash table with the word and the file name/line number where it appears. The code should look something like this:

```
FILE * file;
file = fopen(filename, "r");
while (!feof(file)) {
  int line_number = 0;
  char * word;
  char * saveptr;
```

```
    fgets(buffer, buffer_len,file);
    word = strtok_r(buffer, " \n\t-_!@#$%^&*()_+=,./<>?", &saveptr);
    while (word != NULL) {
     insert_into_index(word, file_name, line_number);
     word = strtok_r(NULL, " \n\t-_!@#$%^&*()_+=,./<>?",&saveptr);
    }
    line_number = line_number+1;
  }
  fclose(file);
```

You need to built a data structure to pass file names from the file system scanner to the indexer. This is a producer-consumer type of problem. There can be more than one indexer running at the same time, so you may want a bounded buffer of file names for the indexers to read from.

The indexer should work well on text files and not crash on other kinds of files, such as pdfs.

3 Multithreaded Hash Table

Code for insert_into_index and other functions for the hash table are provided, but they are not thread safe.

You should modify the code in index.c to be thread safe. You should try to use fine grained locks, so that operations that do not race can proceed in parallel. Be careful, though, because the hash table may resize (hashtable_expand) function. You may want to consider reader/writer locks.

4 Search interface

The search interface reads strings from standard input and looks the words in the index.

Basic Search

If the user enters a single word, you should search for that word across all files in the current index. You can also assume that the input will be less than 128 characters (so you need 130 characters to hold a newline and null terminator).

The search interface searches the index and prints out the words that it find in the following format: If the word is found, it prints:

```
  FOUND: file-name line-number
```

where line-numbers start at 1 for the first line of the file (and not zero). You should do this for each line and each file.

If not found, it prints:

```
  Word not found.
```

The program should exit when the user enters ctrl-D at the search interface (indicating end of file).

Code for searching the hashtable and returning a list of locations is provided with the find_in_index function.

If a search is entered before indexing is complete, it should search the files indexed so far (unless the query references a specific file, in which case it should wait for that file to be indexed).

Advanced search
You can also search a specific file for a string. If the user enters a line with two strings, the first string is the pathname of the file to search, and the second word is the word to search for. For example:

```
   files/test.c printf
```

would look for all lines containing "printf" in the file files/printf.c.

If the file provided has not yet been indexed, the thread should wait until the file is indexed and then print results. NOTE: this requires the use of condition variables. If the all the files have been indexed and the file name is not one of the files, you should print out:

```
  ERROR: File <filename> not found
```

where <filename> is replaced by the filename in the search.

5 Threading

You should write this code so that each component runs as a separate thread, and so that there can be multiple indexer threads. You should use pthreads for threading. Thus, you should compile with "-pthread". Thus, you should have at least three threads (one for each component).

The main task you have for this problem is the synchronization: you must make sure that with threads adding to the hash table at the same time as searching it, there are no data races. You also need to synchronize the scanning thread and the indexer threads. Finally, your search thread may have to wait for files to be indexed.

You should use pthread locks to prevent data races, and pthread condition variables with locks to synchronize between threads.

The main thread of your program can be one of your threads. However, you should wait for all other threads to terminate before exiting your main thread. Thus, you should think about how to make sure your threads exit cleanly when needed.

6 Specification

The program should be started with the following command line:

```
search-engine num-indexer-threads file-list
```

where num-indexer-threads is the number of threads running the indexer and file-list is the list of files for the file system scanner to use.

7 Provided code

Provided code for this project is in ~cs537-2/public/projects/p5/index.c. This code consists of the data structure for the index. It is not multi-thread safe, so you must handling concurrency issues, for example by acquiring locks before invoking this code and releasing locks afterwards. Note that the sample code above does not include this synchronization.

The code is contained in index.c and the header file is index.h. Sample code is available in test.c

The code provides three functions:

1. int init_index() call this when your program initializes.
2. int insert_into_index(char * word, char * file_name, int line_number) adds an entry to the index for word word in file file_name at line number line_number. The function does not keep a copy of any of the strings, so you can free/reuse their memory.

index_search_results_t * find_in_index(char * word) searches for word@@ in the index and returns results

Results are returned as a data structure:

```
 typedef struct index_search_elem_s {
  char file_name[MAXPATH];
  int line_number;
 } index_search_elem_t;

 typedef struct index_search_results_s {
  int num_results;
  index_search_elem_t results[1];
 } index_search_results_t;
```

The sample code shows how to use this structure; you should make sure call free on the pointer returned from find_in_index().

Note: none of these functions use locks, so you must provide any locking or synchronization needed.