CS 3013 Operating Systems — WPI, A Term 2019
Craig E. Wills — Project 4 (20 pts)
Assigned: Friday, September 27, 2019 — Due: Thursday, October 10, 2019 (5:00pm)

Important: This project is to be done by each individual student. This is NOT a group project.

Introduction

This assignment is intended to help you learn about different architectures for building a server. In particular, you will build a program that works using a single process using multiple threads within a single process. You will also learn about file attributes.

Problem

The basic idea of this assignment is to handle "requests" to a server. Each request will be a file name. The server needs to determine the type and possibly the size for the given file. Requests will be read in from stdin (using *cin.getline()* in C++ or *gets()* or *fgets()* in C) with one file name given on each line of input. Your server should use one of two architectures to handle requests. The default architecture is a serial architecture, which is a single process with no threads.

Serial Architecture

In the serial architecture version of your program (the default), your program will read one file name from input, process it and then continue until all input file names are processed. Your program should continue to read file names until an end-of-file (EOF) is detected on input. The serial version of your program will allow you to get the file statistic pieces working. The output of the both architectures should be identical to this architecture.

Your server will be using the Linux system call *stat()* to determine information about each file. This system call takes a file name and a statistic buffer of type `struct stat`. There is much information returned about a file, but the fields you will need to use are `st_mode` (the file mode) and `st_size` (the file size in bytes). More information about this system call can be found by looking at its man page.

The output of your program will be to print out the following information:

- total number of "bad files." These are file names causing *stat()* to return an error.
- total number of directories. You can test if a file is a directory by using the `S_ISDIR` macro on the `st_mode` field.
- total number of "regular" files. These are non-directory and non-special files. Most files are of this type. You can test for a regular file by using the `S_ISREG` macro on the `st_mode` field.

- total number of "special" files. There are three types—block, character and fifo. If a file is not a directory or a regular file then it is a special file. For example files in the directory `/dev/` are often special types of files as they refer to devices.
- total number of bytes used by regular files. Accumulate the sizes of all regular files.
- total number of regular files that contain all text (see more details below).
- total number of bytes used by text files. Accumulate the sizes of all regular text files.

Text Files

The *stat()* system call cannot be used to determine if a regular file contains only text. Rather your program will need to read the contents of the file to determine if it is text. Because the contents of a file are unknown you cannot use routines in C/C++ that only work for text input. Rather you need to use the *read()* system call to read in the *bytes* of a file and then determine if each byte is actually a printable character. The files `fileio.C` and `fileio.c` show sample usage of opening a file (or using stdin if no argument is given) then using the *read()* system call to read the contents of the file into a buffer and writing the buffer to stdout.

If a file cannot be successfully opened then it should not be counted as a text file. As the file contents are processed, your program needs to check if each byte in the buffer is a printable character. To do so you should use two routines: *isprint()*, which determines if a byte value is a printable character; and *isspace()*, which determines if a byte is a space, newline, tab, etc. Check the man pages of these routines for details and the needed include file. You should classify a file as a text file if *all* bytes of the file are either printable or a space. If any byte in the file fails a test then the file is not a text file. Be sure and close the file when done processing as there is a limit on the number of open file descriptors a process may have.

Testing

You can test your code by manually entering file names or you can put a list of files into a text file and redirect it as stdin to your code. Another way to test a set of files is to combine your program with the *ls* command using a pipe. If your code is compiled as *proj4* then the following command line will check all files in the current directory.

```
% ls -1d * | ./proj4
```

The options to *ls* are "*1*" (the number one), which causes the files to be listed one per line and "*d*", which causes just directory names and not directory contents to be displayed.

Other directories can be specified with *ls*. For example, here is sample output obtained from running the code on the `/dev/` directory of a Computer Science Department machine.

```
% ls -1d /dev/* | ./proj4
Bad Files: 0
Directories: 3
Regular Files: 3
Special Files: 140
Regular File Bytes: 32881
Text Files: 2
Text File Bytes: 16541
```

Multi-Threaded Architecture

Once your server can handle each request in a serial manner, it should be modified to also support a multi-thread architecture. You should use this architecture when the argument "`thread`" is specified on the command line. In this architecture you should create a worker thread (using the routine *pthread_create()*) for each new file name. The worker thread should use the *stat()* and *open()/read()* calls to obtain statistics about the file. It should update the appropriate type count and total byte variables as appropriate. You can store these variables as globals because all threads can access global variables, but because multiple threads will be accessing these variables you will need to prevent concurrent access by using thread routines to implement mutual exclusion for a critical region of code.

Once a worker thread has updated the appropriate variables for its file, the thread terminates. This approach of creating a worker thread for each new request allows these requests to be processed in parallel. However to avoid too much parallelism, the maximum number of worker threads that can be executing at one time is also specified on the command line. This limit is an integer between one and the maximum of 15.

Your main thread should work by creating a new thread for each new file request until the limit is reached. At this point, your main thread must wait (using *pthread_join()*) for a worker thread to complete. However unlike waiting for a process, your main thread must wait for a thread with a specific identifier. Your program will need to remember the thread ids that it creates and wait for them in the order of creation. After your main thread (the one reading in file names) has read all files by detecting EOF on input, it should wait to make sure all threads have completed and then print out the output statistics.

When all processing is done your main thread should print out results, which should be the same result as for the serial architecture.

Watch out! Threads are nice, but they can cause programming problems due to all threads running in the same address space. Specifically, be wary of passing the file name to a newly created thread. If you use the same static character array for passing file names to each thread then each time your main thread reads a new file name it will write over the previous file name. Even worse, all threads will be pointing to this same buffer and hence the file name could change after the thread is created. Moral of the story: allocate a unique buffer for the file name passed to each thread.

The following shows how to compile your program (if written in C++) using the pthread library for the multi-threaded architecture. It also shows invoking your program with the same example using the multi-threaded architecture with at most 10 threads executing at a

time.

```
% g++ -o proj4 proj4.C -lpthread
% ls -1d * | ./proj4 thread 10
```

Additional Work

Satisfactory completion of a serial architecture version of the basic objective of this assignment is worth 10 of the 20 points. The multi-threaded architecture is worth an additional 8 points. For two additional points, you need to also add code to calculate the total wall-clock time, system and user time used by your program. You should test your code on different directories with both architectures varying the thread limits.

Prepare a short report (1-2 pages of text) with separate graphs of program performance versus the maximum number of threads. The report should be in `pdf` format. Also include results obtained for the serial architecture. Make sure your graphs are correctly labeled and you include an explanation of the results. Be sure to describe the testing environment that you used.

Submission of Project

Please compress all the files together as a single .zip archive for submission. In addition to source files and makefile that compiles your code, you should include a `typescript` file (created using *script* program) showing sample execution of your program.

Please upload your .zip archive via InstructAssist with the project name of *proj4*.