CS/SE 3377 Sys. Prog. in Unix and Other Env.
Project 2: Multi-threaded Hash Tree
Due: Nov 13th, 2022

Version 1.0, 10/24/22

This project may be done individually or with a partner. I strongly recommend working with a partner. Sharing your work with anyone other than your partner is strictly prohibited. Also, do not look for solution on the net or any other outside sources. **Any act of plagiarism will be reported to the Dean of OSC.**

You can develop the project in cs1/cs2/cs3. To run your experiments, use cs3. It has a greater number of CPUs than the other machines. If you want to test your code for large number of threads (128 or more), use net* machines.

Multi-threaded hash tree

In this project, you will write a multi-threaded program to compute a hash value of a given file. The size of the file can be very large. So, there is a need for the program to be multi-threaded to achieve speedup. While one thread reads a part (block) of the file, other threads can compute hash values of already read blocks.

Binary tree threads

For assignment 8, you will write (or wrote) a program to create threads that formed a complete binary tree. The threads did not do any useful work. In this project, you will extend the program to compute hash values.

Hash value

The hash value of a string (very long) is a fixed length numeric value. Hash values are typically expressed in hex digits and can be long. In this project we will restrict the hash value type to a 32-bit unsigned integer. A hashing algorithm will read the input string (file) and produce a hash value. There are many hashing algorithms in the literature. For this project we will use Jenkins one_at_a_time hash described in https://en.wikipedia.org/wiki/Jenkins_hash_function.

One of the uses of hash value is to check the integrity of the data (file, database, etc.). One can think of a hash value of a file as a fingerprint of the file. If the file is modified, then the hash value also changes. By comparing the new and old hash values one can easily detect that the file was modified.

Computation of a hash value of a large file (several giga bytes in size) may take a while. Hence a need for a multi-threaded program.

Part 1: Multi-threaded hashing

A file can be divided into multiple blocks. The basic unit of accessing a file on a i/o device is a block. Assume there are n blocks, m threads, and n is divisible by m. Each thread can compute the hash value of n/m consecutive blocks. Several of these hash values are then combined to form a string whose hash value can be computed as before. These steps can be repeated till one final hash value remain.

As in assignment 8, a complete binary tree of threads should be created. Each leaf thread will compute the hash value of n/m consecutive blocks assigned to it and return the hash value to its parent thread (through pthread_exit() call). An interior thread will compute the hash value of the blocks assigned to it, and then wait for its children to return hash values. These hash values should be combined to form a string: <computed hash value + hash value from left child + hash value from right child>. (Note that the + sign here means concatenation and not addition. Also, if the interior thread has no right child, only the hash value of the interior thread and the hash value from the left child are combined.) The interior thread then computes hash value of the concatenated string as before and returns the value to its parent, The value returned by the root thread is the hash value of the file.

How to assign blocks to threads?

Each thread is assigned a number when it is created. The number of root thread is 0. For a thread with number i, the number of its left child is 2 * i + 1, and the number of its right child is 2 * i + 2. For a thread with number i, n/m consecutive blocks starting from block number i * n/m are assigned. For example, let n = 100 and m = 4. Then thread 0 will be assigned blocks 0 through 24, thread 1 will be assigned blocks 25 through 49, thread 2 will be assigned blocks 50 through 74, and thread 3 will be assigned blocks 75 through 99.

Usage and hints

Name your program htree.c. The usage is:

```
htree filename num_threads
```

'filename' is the name of the input file whose hash value needs to be computed. 'num_threads' is the number of threads to be created.

Assume the block size to be 4096 bytes. Number of blocks can be found from the file size in bytes. Check fstat() to find the file size.

Starter code htree.c is in ~sxa173731/3377/hw/ and several testcases are provided in ~sxa173731/3377/hw/p2_testcases/. Trace the verbose output (hash) files in this folder to get a good understanding of the requirements.

You can read any block from the input file using lseek(0 and read(). But there is an easier way to read blocks from a file using mmap(). Check the man page for mmap().

You can use sprintf() to convert an uint to string.

Part 2: Experimental study

The main goal of this study is to find speedup achieved when the number of threads is increased for various input files. For each input file, find the time taken to compute the hash value for 1, 4, 16, 32, 64, and 128 threads.

Run your experiments on cs3. It has 48 CPUs. Other machines have lesser number of CPUs.

Plot a graph with time on y-axis, and #threads on x-axis for each input file. Plot another graph with speedup on y-axis and #threads on x-axis for each input file.

speedup = (time taken for single thread)/(time taken for n threads)

Write a short report on what you observe. It is expected that the time taken to compute hash value decrease when the number of threads increase. Does it always happen? What is the speed-up achieved when the number of threads increased? Is the speedup proportional to the number of threads increased? Also, briefly describe the experimental set-up at the beginning of your report. (Command 'lscpu' provides relevant information for this)

Write a high-quality report. You should showcase the report in your job interview.

Other Requirements

Error handling is an important concept in system programming. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

So, check the input for errors. Check the return values of function calls. Display appropriate error messages.

Testing

Make sure you compile your program as follows:
gcc htree.c –o htree -Wall -Werror -std=gnu99 -pthread

There should not be any error messages and warning during compilation.

Sample testcases are provided.

Hand-in Instructions

Name your program file as htree.c and submit it along the with study report (in pdf format) on elearning.

If you have worked with a partner, only one of you need to submit. But both should upload text file named PARTNER.txt which should contain partners' name and netids.

Grading Policy

Source code should be structured well with adequate comments clearly showing the different parts and functionalities implemented. Up to 10 points will awarded for style.

The TA will compile the code and test it in one of cs* machines. If your program does not compile in cs* machines, you will get 0 points.

Write a good report. 30 points are allocated for the report.

You may also be called by the TA to demonstrate your code. If you are not able to explain the working of your code, you will not be given any points.