Program #1
Multi-threaded Programming

Nathan Balon
SN# 3210 1717
CIS 450
Winter 2004
Due: 2/ 17/ 2004

Introduction

The use of the multithreads in a program can have either a positive or a negative impact on the performance of a program. Simply by making a program multithreaded, it will not guarantee improved performance for the program. A program must be properly design to use multiple-threads. If a program is poorly designed, it will very likely be the case that the threads are hindering the performance of a program. Using threads can also increase the complexity of the program. For instance, the programmer must worry about synchronization of the global data in the program when multi-threads are used. If the programmer fails to provide a synchronization method, it will very likely be the case that the data in the program can become corrupt over time. If the synchronization methods aren't properly applied to the program, deadlocks will inevitably develop, and bring the programs execution to a halt.

With these problems aside, a properly designed multi-threaded program can gain a significant performance advantage. Two typical cases where using threads is beneficial to performance, is with server programs such as a web or a database server and with user interfaces. For example, a web server can listen for connection to be made and when one is established, a new thread can be allocated to handle the communication from a poll of available threads. Another instance where threads can also prove beneficial is they allow a program to be logically divided. In the case of a user interface, one thread can be used to read in data from a user and another thread can be used to write data to the user. There are numerous other positive examples of how threads can be used to benefit a program. The assignment that follows is to show instances of when the threads can be beneficial in use with operating systems and when using threads can detract from performance.

Assignment

The assignment includes creating two programs, which run on the Solaris operating system. Each program is to have two different versions of it. One version is multithreaded using pthreads and the other version is implemented with a single thread. One program is to show a gain in performance because of using threads and the other on does not improve performance. The programs are to be written in either C or C++.

Implementation

The programs for this assignment are all written in C. Each program is written as a function in C so they can all be called from main in one program eliminating the need for separate files. The programs were designed to be very static, so that runtime results would be consistent for each execution of the program. There are many cases where using threads would have a great improvement on performance but some such programs would be hard to implement to produce reliable results. This severely limits the type of programs that can be written to simple programs that are easily reproducible. Both of the programs in this assignment are of limited real value in there current state. They are used just to show simple examples of the impact of threads on performance.

Computing the correct program runtime is crucial in all the programs. Each of the four programs in the assignment uses the timespec structure to store the beginning and end times of the program. A function print_prog_time() prints the runtime results of the program in a user friendly format.

Aside from just comparing the results on Solaris I also ran the programs on in a single processor computer running Redhat Linux 9.0. I wanted to determine the differences on execution based on how the operating system used threads. In some of the case I noticed a great difference in the relative performance of the program based on the computer the program was run on.

Program 1

The first of the two programs to be compared are set_total and set_total_th. The purpose of these programs is to show the effect that threads have on computing the total of a number of sets in parallel. For this program a large multi-dimensional array is used to hold the values of the sets. The arrays are then randomly filled with integer values to be read later in the program. The arrays may not be a set in the true sense because for this program there is the possibility that the number is repeated. After the arrays are given values, the real work of the program starts that is be compared for its performance. Each of the arrays in the program is totaled. While this totally has no real purpose it is used by the program to show the effects of an intensive computation performed in parallel with threads. For example, if there is a number array, which happens to be number[3][1000]. Then, there will be three number arrays, that each holds 1000 elements. The total the 1000 integer values in each array taken and the results from each of the set is displayed to the user along with the total runtime of the program. In the case of the multithreaded program three separate threads are created for this instance of the program. The number of threads correlates to the number of sets. In the case of this program the runtime clock in not started until all of the data in the sets is initialized.

In almost all cases that this program was ran the multi-threaded version of the program received the best results. In the source code the size of the sets and the number of them are defined at the top so they can easily be changed. So, during the execution of the program the results were frequently changed to observe the impact of using different numbers of threads for each set and a different number of values in it. The only time that the performance of the program was found to significantly worsen, when using multiple threads was when only a few numbers were in a set to be totaled. The cause of this is associated with the overhead of creating the threads. When there were only 20 or 30 integers to be totaled in each set the non-threaded version outperformed the threaded version. In the case of performing computation on a large set of data, the threaded version of the program consistently returned the best results. The runtime of the multi-threaded program on the average is about half of that of the single threaded program.

Below in figure 1 are the results of the program. The top portion of the figure is the execution of the single threaded program has a runtime of 0.070s and the bottom is the

multi-threaded program that has a runtime of 0.035s. From the figure it is clear that the multi-threaded program achieves the best performance.

From these results it can be deduced that the server that the program was ran on contains multiple processors. Using pthreads does not guarantee a performance gains from design the threads to be ran in parallel. If only a single processor is available this program will not achieve any performance gains. I ran the two programs on my home pc which has Linux on it and got quite different results. On my home pc with Linux the threaded version actually had worse performance. I concluded that the difference in the execution came from the difference is how the operating systems support thread and possible hardware differences.

```
Program Set Total
***Initializing data***
Randomly filling 3 sets, with 100000 items
Set 0's total is: 1296820
Set 1's total is: 1302632
Set 2's total is: 1299850
------------------------------------------------------------------------
start time sec  = 1076848621  nsec =  381284698
  end time sec  = 1076848621  nsec =  451754794
Total program runtime of program set_total is sec = 0.070470096s

Program Threaded Set Total
***Initializing data***
Randomly filling 3 sets, with 100000 items
Set 0's total is: 1794691
Set 1's total is: 1799412
Set 2's total is: 1795462
------------------------------------------------------------------------
start time sec  = 1076848621  nsec =  858415749
  end time sec  = 1076848621  nsec =  894261877
Total program runtime of program set_total_th is sec = 0.035846128s
```

Figure 1 - Results of Program 1 Set Total

Program 2

The second program used in the assignment counts the number of lines in a file. The program reads the files in a certain directory and then counts the number of lines that are contained within each file. The program will also display the result of the total number of line read from all the files. This program works similar to the wc –l command in UNIX. Each program is started from main by calling the function for the program. Each of the functions takes the name of the directory to search through. Originally the program was to accept command line arguments but to limit the involvement in conducting the performance test that option was eliminated. Now it will only read from what ever directory is passed to the function. The result of the execution of this program is a printing of the files read from and the number of lines that are contained within each file.

The results of this program are the opposite of program one. In this case, the use of threads degrades the performance of the program. The program is completed almost twice as fast when only a single thread is used. Figure 2 shows the output of the program line count. The single thread version takes about 0.56s to complete compared to the threaded version which takes 1.08s.

There are a few reasons for the poor performance with this program. Most of which result from poor us of threads by the programmer. For an illustration purposes, this program was designed so that threads would have a negative impact on performance. The multi-threaded version of the line count program could achieve some performance gains if the program was modified. One problem with the program is that a new thread is used for each file that is read. This causes a high amount of overhead for the program. Even if the file contains only a few lines a new thread is created to read from the file. Performance could be improved if threads were continued to be used but instead of creating a new thread each time the program would acquire one from a pool of available threads. After the thread completes reading from the file the thread is returned to the pool to be further reused. This approach would cut down on a lot of the overhead in the program.

```
Program Line Count
------------------------------------------------------------------
File:   .                                   Lines:      0
File:   ..                                  Lines:      0
File:   time_test.c                         Lines:      8
File:   line_count.c                        Lines:     70
File:   a.out                               Lines:      3
File:   lines.c                             Lines:    434
File:   lines                               Lines:      7
File:   th_line_count_result                Lines:      0
File:   line_count_result                   Lines:   9002
File:   lines.c.bak                         Lines:    331
File:   lines.c.bak.save                    Lines:    277
File:   proc_test.c                         Lines:     25
File:   prog1                               Lines:      0
File:   proc_test                           Lines:      4
File:   new_lines.c                         Lines:    297
File:   almost.c                            Lines:    393
File:   lines2.c                            Lines:    221
File:   lines2                              Lines:      6
------------------------------------------------------------------
Total lines read from directory /students/grad/n/nbalon/cis450 =  11078
------------------------------------------------------------------
start time sec  = 1076847396  nsec =  385861108
  end time sec  = 1076847396  nsec =  949136576
Total program runtime of program set_total_th is sec = 0.563275468us

Program Threaded Line Count
------------------------------------------------------------------
File:   .                                   Lines:      0
File:   ..                                  Lines:      0
File:   time_test.c                         Lines:      8
File:   line_count.c                        Lines:     70
File:   a.out                               Lines:      3
File:   lines.c                             Lines:    434
File:   lines                               Lines:      7
File:   th_line_count_result                Lines:      0
File:   lines2.c                            Lines:    221
File:   lines2                              Lines:      6
File:   proc_test                           Lines:      4
File:   prog1                               Lines:      0
File:   proc_test.c                         Lines:     25
File:   almost.c                            Lines:    393
File:   new_lines.c                         Lines:    297
File:   lines.c.bak                         Lines:    331
File:   lines.c.bak.save                    Lines:    277
File:   line_count_result                   Lines:   9002
------------------------------------------------------------------
Total lines read from directory /students/grad/n/nbalon/cis450 = 11078
------------------------------------------------------------------
start time sec  = 1076847396  nsec =  949840111
  end time sec  = 1076847398  nsec =   31861390
Total program runtime of program set_total_th is sec = 1.082021279us
```

Figure 2 - Line count programs output