CMPE-380: Applied Programming

Laboratory Exercise 13

Threading

Table of Contents

Threading

Pre-Lab – 10 pts

In this lab session, we will review Threading concepts in C. Read and understand the class notes on pthreads.

Threads allow us to run multiple code segments concurrently. OS kernel schedule threads asynchronously and may interrupt them if needed. A thread has its own program counter (PC), a register set, and a stack space. Threads share data and program memory and other OS resources with other threads.

Threads have an obvious performance benefit because they may allow some classes of problems to be broke into multiple parallel parts and then executed in parallel on multiple CPUs. Threads have another advantage in that many embedded processes naturally break into individual pieces that are easier to implement as individual threads even if there is only one CPU and therefore no performance benefit. Most programs use multi-threading: Web Browsers, Microsoft Office etc.

Your lab TA will conduct a brief oral quiz on pthreads. Be prepared to explain what thread safe coding is, what a thread safe library is, why threading can improve performance, why you might use threads even if you are on a single CPU, how threads impact your stack space. Be prepared to explain what the basic threading functions like: pthread_create(), pthread_mutex_init()/lock()/unlock(), pthread_join(), pthread_exit() and sleep() do.

Interactive exercises – 30 pts

In this section you will be writing some simple threading code. Place all your documentation and answers to questions in **exercise.txt**. Be sure to make **backup copies** of your **exercise.txt** file as you are working. The threading application template lab_a.c has been provided. All appropriate thread error checking is required, print and error message and return with a non-zero return code.

1) Review the provided **lab_a.c** code and understand what it does.
2) Add the call to create a thread as indicated in the file. Your argument passed to the thread is **&num**
3) Compile your code with: **gcc -g -std=c99 lab_a.c -lpthread -o lab_a**
4) Run you code and save the output in **exercise.txt.** You code should look like:

```
Main: Starting thread...
Main: All threads started...
Main will sleep 1 second.
   Thread 1, will sleep 1 second.
Main will sleep 1 second.
   Thread 1, will sleep 1 second.
Exiting main thread
   Thread 1, will sleep 1 second.
   Thread 1, will sleep 1 second.
   Thread 1, will sleep 1 second.
   Thread 1, will sleep 1 second.
```

5) Run valgrind using the provided **memtst** script. Capture the output in your exercise.txt file. Did valgrind report memory leaks?

6) Copy **lab_a.c** and call it **lab_b.c**
7) Add a pthread_join() function call in the section " Student should add wait for the thread to finish when instructed". Use **&rcp** for the status pointer.
8) Print the value of the process return code **rcp**
9) Run you code and save the output in **exercise.txt.** How does the new code differ from the previous?

10) Run valgrind using the provided **memtst** script. Capture the output in your exercise.txt file. Did valgrind report memory leaks? What is the basic difference between lab_a.c and lab_b.c?

11) Copy **lab_b.c** and call it **lab_c.c**
12) Expand code to three threads. **taskRC** and **num** will now be arrays.
13) Set the value of **taskRC** to the task number **num** in the thread to prove your return codes are unique. E.g:

    Thread RC 1
    Thread RC 2
    Thread RC 3

14) Run you code and save the output in **exercise.txt.** How does the new code differ from the previous?
15) Run valgrind using the provided **memtst** script. Capture the output in your exercise.txt file. Did valgrind report memory leaks?

16) Copy **lab_c.c** and call it **lab_d.c**
17) Add a global variable called "processed", initialize it to zero and use mutex locks to protect it. Did you make "processed" volatile" Why or why not?
18) Modify your thread code to increment "processed" once for each thread sleep loop.
19) Modify your main program to report progress of the "processed" variable.
20) Modify your program to report the total "wall time" used in main(). Use the **time(NULL)** function.
21) Run you code and save the output in **exercise.txt.** How does the new code differ from the previous?

Show your exercise.txt and all your code to your TA. Your TA will verify that you are properly checking thread status and using mutex locks properly.

Assignment – 60 pts

Objective

The purpose of this assignment is to exercise your ability to write dynamically scalable threading code. You are going to implement threaded code to initialize a massive 1D array to the value 3*index value. E.g: **array [i] = 3*i;**

This problem is naturally threadable because the problem can be broken into parallel chunks. Each chunk is assigned a start and end range and the results of one chunk don't affect other chunks. A frame work program called **lab13.c** is included. You will need to copy your **Timers.h** file from a previous lab.

Program Specification

The requirements:

1) The program shall support 1 to 8 threads, inclusive. Use the provided MAX_THREADS define.
2) The program shall use get_opt_long() and support the following command line parameters.

   Syntax: hw13 - t[hreads] num [-s[tatus]] [-f[ast]] [-v[erbose]]
   Where: -t[hreads] num - number of threads 1 to 8,required
   -s[tatus] - display thread progress, optional
   -v[erbose] - verbose flag, optional
   -f[ast] - shorter data run for Valgrind, optional

3) The array size to be initialized will be either of the provided defines:

   DATA_SIZE (136*3*5*7*146*512) bytes (default)

   VALGRIND_DATA_SIZE (30*3*5*7*8*1024) bytes (fast size)

4) If any command line parameter is incorrect, a friendly error message will be displayed, and the full usage syntax displayed
5) When the program starts it shall display an overall status message of the form:

   Starting 2 threads generating 1067458560 numbers

6) The program shall utilize proper return code checking.
7) In verbose mode the following style message will be display after creating a thread:

   Thread:0 ID:140288510658304 started

8) In verbose mode the following style message will be displayed by the created thread:

   Thread:0 track status:1 seg size:25200KB data ptr:0x7f9776d90010

9) In verbose mode the following style message will be displayed after the thread rejoins the main thread:

   Task 0 join 0, task rc 10

10) In status mode the main thread will display the progress of the array initialization every **TWO SECONDS** in the following style:

    Processed: 106745853 lines 10% complete

11) Each thread shall update the global progress value for each 10% processing completed by that thread. E.g Update the global variable after +10% of the process specific array entries have been initialized. Use the provided update rate define: STATUS_UPDATE_RATE
12) The application shall flush() buffers as necessary to guarantee output.
13) At the conclusion of the threads, program total CPU and wall time will be displayed in the following style:

    Elapsed CPU Time (timer) = 1.8125 sec.

    Total wall time = 1 sec

14) Prior to the overall conclusion of main() the program will verify that the resulting matrix was properly initialized and then free the array
15) Due to the performance of modern processor, the threaded initialization code must be slowed down using the following:

```
// Slow the CPU
int delay = 1<<DELAY_LOOPS_EXP;
while (delay) { delay--; } during each iteration of the initialization.
```

16) Due to the performance of modern processors, compile with -O0. Use the following compile line**: gcc -g -O0 -std=c99 hw13.c -lpthread -o hw13 -Wall -pedantic**
17) At the completion of a thread, the thread shall set a unique return code to **10+thread index** to show that return codes are unique.
18) The data structure **struct ThreadData_s** has been provided in the code and should be used for thread process setup communication.

19) Run valgrind using the following: **-f -t 8 -v**

Note: The data set size we are using in this lab is VERY large and might not always fit safely into a 32 bit signed integer. Try to take advantage of order of operations to avoid overflow.

E.g. **a*b/c** vs **a*(b/c)**

In the first case a*b generates a very large temporary value that will overflow.
Forcing the (b/c) operation first will avoid the overflow.

Makefile

You must provide a quality Makefile with the following targets: all, test, mem, help and clean. Quality Makefiles utilize .SILENT, .PHONY, uses make variables and only compile and/or link the minimum number of files. A Quality Makefile is robust to usage. E.g. Make clean followed by Make mem will automatically rebuild the binary.

Make Targets;

"all"	-should make hw13.c
"test"	- should run **hw13** with the each of the following options and redirect/append the output to **out.txt**. Each command should echo the test to "**out.txt**". (see results.txt)
	./(hw3 -t 1 -s -v
	./hw13 -t 5 -s
	./hw13 -t 8 -v
	-./hw13 -t 0
	-./hw13 -t 9
"mem"	**-** should run the standard valgrind with: **./hw13 -f -s** redirected to **mem.txt**
help, clean	- should do the normal things

Analysis

No analysis is required for this lab.

Grading Criteria

1. (30 points) Correct implementation of variable threading code.
2. (20 points) No memory leaks or access errors
3. (10 points) Correct make file

Student Name: _____

Laboratory Grading Sheet

Lab 13 - Threading

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: verbal quiz	10		
Exercise: simple thread - lab_a	5		
Exercise: join() lab_b	5		
Exercise: multi threads lab_c	10		
Exercise: mutex lock lab_d	10		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!