ECE 254/MTE 241 Lab2
Thread Concurrency Control
Last updated: 2010/10/04

Objective

This lab is to learn about, and gain practical experience in concurrency control for the purpose of inter-thread communication by shared memory and compare its performance with inter-process/task communication by message passing. In particular, you will use the POSIX pthread library in a general Linux environment and RL-RTX on the Keil LPC1768 board.

Sharing data among threads in a gerneal Linux environment is trivial because threads share the same memory [2]. For the Keil RL-RTX tasks, they share the same addressing space, hence behave similar as threads.

The challenging part of sharing memory is to avoid race conditions. In a general Linux envrionment, pthread semaphore and mutex library calls are to be used for thread mutual exclusion. In RL-RTX, semaphore and mutex management APIs are to be used for task communication mutual exclusion.

After this lab, students will have a good understanding of, and ability to program with

- the pthread `pthread_create()` and `pthread_join()` to create and join threads,
- the pthread `sem_int()`, `sem_post()` and `sem_wait()` library calls for inter-thread communication concurrency control in a general Linux environment, and
- the use of semaphore or mutex management APIs in RL-RTX for inter-task communication concurrency control.

Pre-lab Preparation

1. Read Linux man page of semaphore overview.
   `man 7 sem_overview`
2. Read Sections 4.1, 4.4 and 4.5 about thread programming in [2].
3. Read supplementary materials regarding POSIX pthread in [1]
   (`http://www.cs.cf.ac.uk/Dave/C/node29.html#SECTION00294000000000000000`).
4. Read the Keil μ`Vision4` RL-ARM Real-time Library User's Guide (a help file within the IDE). Under the RL-RTX section, the subsection Function Overview contains more detailed description on semaphore management and mutex management routines.

Lab2 Assignment

Solve the producer-consumer problem with a bounded buffer in a general Linux environment and on the Keil LPC 1768 board. This is a classic multi-tasking problem in which there are one or more tasks that create data (these tasks are referred to as "producers") and one or more tasks that use the data (these tasks are referred to as "consumers"). We will have a system of `P` producers and `C` consumers . Three experimental cases then exist: *single producer/single consumer; single producer/multi-consumer; multi-producer/single consumer*.

The producer tasks will generate a fixed number, `N`, integers, one at a time. The first integer the producer generates is always `0` and then the newly generated integer increments by one sequentially. Each time a new number is created, it is placed into a fixed-size buffer, size `B` integers, shared with the consumer tasks. The producer sleeps for a random period of time, between 0 and 1 second after it deposits an integer to the buffer. When there are `B` integers in the buffer, producers stop producing. Since there is more than one producer, and we do not want the producers to have to coordinate their actions since that would require additional inter-thread/task communication, we will adopt the following approach: each producer has an identity number, `id`, from `0` to `P-1`. The producer with identity number `id` will produce the integers `i` such that `i%P == id`. For example, if there are `7` producers, producer number `3` will produce the integers `3`, `10`, `17`, `...` .

Each consumer is likewise given an integer identity, `cid`, from `0` to `C-1`. Each consumer task reads the integer out of the buffer, one at a time, and calculates the square root of the integer. When the square root of the integer is itself an integer, the consumer prints out its identity, the original integer taken from the buffer, and the value of the square root on the terminal screen (in Linux) or LCD screen (on Keil LPC1768 board). For example, if there are 6 consumers and consumer with `cid = 3` read the value `16` from the buffer, it will display `3 16 4` on the LCD. After displaying the results, the consumer should sleep for a random period of time, between 0 and 1 second. The reason for random sleeps both in producer and consumer is that even though producers and consumers will all try to produce/consume all the time; likely many would be starved; we want to ensure some variation in which producers/consumers are generating/acquiring and displaying data.

Given that the buffer has a fixed size, `B`, and assuming that `N > B`, it is possible for the producers to have produced enough integers that the buffer is filled before any consumer has read any data. If this happens, the producer is blocked, and must wait till there is at least one free spot in the buffer.

Similarly, it is possible for the consumers to read all of the data from the buffer, and yet more data is expected from the producers. In such a case, the consumer is blocked, and must wait for the producers to deposit one or more additional integers into the buffer.

Further, if any given producer or consumer is using the buffer, all other consumers and producers must wait, pending that usage being finished. That is, all access to the buffer represents a critical section, and must be protected as such.

The program terminates when the consumers have read all `N` characters from the producers and finish displaying all square roots that are integers. Note that this is more complex than in the first lab, because you have multiple consumers that are reading from the buffer, and thus will need to determine whether or not some consumer has read the last integer.

Question: Is there a similar problem for the producers to deal with, or is their situation simpler?

Requirements:

Let `N` be the number of integers the producers should produce and `B` be the buffer size. Your program will execute and then print out the time it takes to consume all `N` integers on the terminal/LCD screen;

Let `P` be the number of producers and `C` be the number of consumers. For a set of given `<P,C,N,B>` tuple values, run your application and measure the time it takes. Note for a give value of `<P,C,N,B>`, you may need to run multiple times to compute the average execution time.

In a general Linux environment, compare the performance of multi-thread communication by shared memory with multi-process communication by message queue.

1. Create a process with a fixed buffer size in which producers and consumers are threads within the process.
2. Extend Lab1 Linux implementation to multiple producers and multiple consumers case and use message queue for inter-process communications. Note that you start your program with one process which then forks multiple producer processes and multiple consumer processes.

On the Keil LPC1768 board, compare the performance of multi-task communication by shared memory with multi-task communication by mailbox.

1. Create producers and consumers tasks with a fixed buffer size in which the buffer is a shared global data structure among tasks.
2. Extend Lab1 LPC1768 implementation to multiple producers and multiple consumers case and use mailbox for inter-task communications.

Post-lab Deliverables

Submit the following two items to the course book system before the deadline. Unless you notify the lab TAs and the lab instructor by email, we will mark the latest submission when there are multiple submissions presented.

1. Entire lab2 μVision project folder and all source code for Linux platform.
   Zip the entire source code and name the .zip file `lab2_src.zip`.

2. A lab report named as `lab2_rpt.docx` or `lab2_rpt.pdf`. which contains the following items.

   - Timing analysis for Linux environment
     - Two tables that show the average timing measurement data for the `(P,C,N,B)` values shown in table 1 for a general Linux environment. One table is for the timing result by the approach of inter-thread communication with shared memory. Another table is for the timing result by the approach of inter-process communication with message queue. Note that for each row in the table, you need to run the program `X` (you decide the value of `X`) times and compute the average time.

P	C	N	B	Time
1	1	100	4	
2	1	100	4	
3	1	100	4	
1	2	100	4	
1	3	100	4	
1	1	100	8	
2	1	100	8	
3	1	100	8	
1	2	100	8	
1	3	100	8	
1	1	398	8	
2	1	398	8	
3	1	398	8	
1	2	398	8	
1	3	398	8	

Table 1: Timing measurement data table for given (P, C, N, B) values.

     - Given $(P, C, N, B) = (1, 3, 398, 8)$, run your program X times (you decide the value of X) on Linux platform. Present the average timing measurement data and its standard deviation.
     - Compare the timing results of multi-thread with shared memory and multi-process with message queue. Discuss the advantages and disadvantages of these two approaches to solve the same problem.
   - Timing analysis for Keil LPC1768 board
     - Two tables that show the timing measurement data for for the `(P,C,N,B)` values shown in table 1. One table is for the timing result by the approach of inter-task communication with shared memory. Another table is for the timing result by the approach of inter-task communication with message queue. You only need to run your application once in order to fill the tables.
     - Given $(P, C, N, B) = (1, 3, 398, 8)$, run your program X times (you decide the value of X) on the Keil LPC1768 board. Compute the average timing measurement data and its standard deviation.
     - Compare the timing results of multi-task with shared memory and multi-task with message queue. Discuss the advantages and disadvantages of these two approaches to solve the same problem.
   - Add an appendix in the report which contains your source code of the producer and consumer well as any other routine that create these processes/tasks. Note that you are required to comment the code appropriately so that any other programmer can follow your algorithms and logic.

References

[1] AD Marshall. Programming in c unix system calls and subroutines using c. *Available on-line at http://www.cs.cf.ac.uk/Dave/C/CE.html*, 1999.

[2] M. Mitchell, J. Oldham, and A. Samuel. Advanced linux programming. *Available on-line at http://advancedlinuxprogramming.com*, 2001.