Computer Operating System Experiment

Laboratory 4

Threads

Objective:

- Practice working with multi-threaded C programs. This lab will give you a good introduction on how to work with the Pthread library. You will be focusing on applying some of the most important concepts discussed in lecture. (Later labs will delve deeper into the topic.)
- Practice working with C function pointers. The C language has a very interesting feature that allows programs to use functions as parameters to other functions. The concept and the syntax for function pointers can be a little intimidating, but this is something you will have to master in order to use Pthreads.

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1 Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

In general, there are two types of parallelism: data parallelism and task parallelism. Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

2 Work with Pthread

Pthreads are a simple and effective way of creating a multi-threaded application. This introduction to pthreads shows the basic functionality – executing two tasks in parallel and merging back into a single thread when the work has been done.

Multi-threaded applications allow two or more tasks to be executed concurrently. When a thread is created using pthread_create, both the original thread and the new thread share the same code base and the same memory – it's just like making two function calls at the same time. Multi-threaded applications come with a whole host of concurrency issues, which will be discussed further in a future post.

All C programs using pthreads need to include the pthread.h header file (ie: #include <pthread.h>). There are four steps to creating a basic threaded program:

1) Define thread reference variables

The variable type pthread_t is a means of referencing threads. There needs to be a pthread_t variable in existence for every thread being created. Something like pthread_t thread0; will do the trick.

2) Create an entry point for the thread

When creating a thread using pthreads, you need to point it to a function for it to start execution. The function must return void * and take a single void * argument. For example, if you want the function to take an integer argument, you will need to pass the address of the integer and dereference it later. This may sound complicated but, as is shown below, it's pretty simple. An example function signature would be void *my_entry_function(void *param);

3) Create the thread

Once the pthread_t variable has been defined and the entry point function created, we can create the thread using pthread_create. This method takes four arguments: a pointer to the pthread_t variable, any extra attributes (don't worry about this for now – just set it to NULL), a pointer to the function to call (ie: the name of the entry point) and the pointer being passed as the argument to the function. Now there's a lot of pointers in that call, but don't stress – it's not as tricky as it sounds. This call will look something like pthread_create(&thread0, NULL, my_entry_function, &parameter);

4) Join everything back up

When the newly-created thread has finished doing it's bits, we need to join everything back up. This is done by the pthread_join function which takes two parameters: the pthread_t variable used when pthread_create was called (not a pointer this time) and a pointer to the return value pointer (don't worry about this for now – just set it to NULL). This call will look something like pthread_join(thread0, NULL);

And that's all there is to it. The function used as the thread entry point can call other functions, create variables or do anything any other function can do. It can also use the variables set by the other thread.

When compiling the program, you will also need to add -lpthread to the compile command. ie:
gcc program.c -o program -lpthread

You can use gettimeofday() function to time how long your program run

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
The tv argument is a struct timeval (as specified in <sys/time.h>):
  struct timeval {
       time_t        tv_sec;       /* seconds */
       suseconds_t tv_usec;      /* microseconds */
  };
```

example:

```
uint64_t GetTimeStamp() {
       struct timeval tv;
       gettimeofday(&tv,NULL);
       return tv.tv_sec*(uint64_t)1000000+tv.tv_usec;
}
```

3 Experiments

3.1 Experiment 1: data sharing

The program shown in below uses the Pthreads API.

```
#include <pthread.h>
#include <stdio.h>
#include   <sys/types.h>
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    int myvalue = 5;
    pid = fork();
    if (pid == 0) { /* child process */
        int myval = 5;
        pthread_create(&tid,NULL,runner,&myvalue);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d\n",value); /* LINE A */
    }
    else if (pid > 0) { /* parent process */
        myvalue = 10;
        pthread_create(&tid,NULL,runner,&myvalue );
        pthread_join(tid,NULL);
        wait(NULL);
        printf("PARENT: value = %d\n",value); /* LINE B */
    }
}
void *runner(void *param) {
        value = *((int*)param) - 1;
        pthread_exit(0);
}
```

Questions:

1) What would be the output from the program at LINE A and LINE A? please use the data sharing mechanism to explain it.

3.2 Experiment 2: calculates various statistical values using pthread

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

The average value is 82
The minimum value is 72
The maximum value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.

3.3 Experiment 3: calculate the sum of number of squares

The task specified an interval of integer numbers. you need to calculate the result that follows the formula below.

$$1^2 + (1+1)^2 + (1+2)^2 + \ldots + (a-2)^2 + (a-1)^2 \qquad (1)$$

Requirements:

1) According to the formula (1), you need to write a serial sum_serial.c program to get the result.
2) Using **gettimeofday()** function to time how long you program takes.

3.4 Experiment 4: calculate the sum of number of squares using pthread

Modify sum_serial.c program to sum_pthread.c, which requires two command line arguments to execute:

numthreads – an integer value that specifies how many threads will be created, and

the number – the number of integer values.

The task performed by this code is simple: It creates the specified number of threads and assigns to each one of them an interval of integer numbers, say [a,b). Each thread computes a value s, which is the summation of the squares of the integers in its interval, that is:

a^2 + (a+1)^2 + (a+2)^2 + … + (b-2)^2 + (b-1)^2

Essentially, this program "parallelizes" the computation of the summation of the squares of integers in the interval [a,b) balancing the load equally among the number of threads specified. We say "parallelizes" in quotes because in a single-core machine, the threads would execute not in parallel, but rather concurrently: splitting the time of one single CPU. If this code runs on a multi-core machine the threads may really run in parallel (this depends on the set up of the library and the OS)

Suppose that you have *N* threads, every thread gets ⌊ a/N ⌋ interval, so you can assign the task to every thread and calculate.

Thread0, {1, ⌊ a/N ⌋}

Thread1, { ⌊ a/N ⌋+1, 2 ⌊ a/N ⌋}

Thread2, {2 ⌊ a/N ⌋+1, 3 ⌊ a/N ⌋}

…

Threadn-1, {(n-1) ⌊ a/N ⌋+1, a }

```
/ structure for thread arguments
struct thread_args {
    int tid;
    int a;      //start
    int b;      //end
    double result;    // partial results
};
```

Requirements:

1) According to the description, you need to use pthread to write sum_pthread.c to get the result.
2) Using **gettimeofday()** function to time how long you program takes.

Questions:

1) What is the type of parallelism that you use? Is the data parallelism or task parallelism? Please explain it.

3.5 Experiment 5: analysis the performance

Copy you sum_pthread.c to a new file summation.c, which will compute the time it takes to execute to completion. To accomplish this, you should instrument the program to call gettimeofday(2) right when starts and then again when it ends.

With a little bit of arithmetic, *using* the two struct timeval values returned, you can compute the time it took the program to run. Be careful to do the right arithmetic when you subtract two time values. Have this time printed to the standard output exactly before the program terminates.

Update your Makefile so it will compile summation.c properly.

Please input different numthreads and the number of integer values, you can get the final calculating time.

Questions:

1) When you increate the number of threads to get the summation, whether the running time is shorter or not? Why?
2) If there is any problem, explain what causes it and how you can avoid it. Also, remember to modify your source code and submit a version that sure prints out the termination time for each thread.