HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 5 Thread
Course: Operating Systems

Thanh Le-Hai Hoang
Email: thanhhoang@hcmut.edu.vn

April 28, 2021

Goal: This lab helps student to distinguish the differences between thread and process; How to implement a multithreading program in Linux using C programming language.

Content In detail, this lab requires student practice with examples such as creating a multithread program, showing the memory region of threads.

- Write a multithread program
- Solve an example which can run in parallel using thread
- Show the differences between process and thread in term of memory region

Result After doing this lab, student can understand the difference of thread and process. Student can write a program with multiple thread created.

Requirement Student need to review the theory of thread.

1 Stack

Stack is one of the most important memory region of a process. It is used to store temporary data used by the process (or thread). The name "stack" is used to described the way data put and retrieved from this region which is identical to stack data structure: The last item pushed to the stack is the first one to be removed (popped).

Stack organization makes it suitable from handling function calls. Each time a function is called, it gets a new stack frame. This is an area of memory which usually contains, at a minimum, the address to return when it complete, the input arguments to the function and space for local variables.

In Linux, stack starts at a high address in memory and grows down to increase its size. Each time a new function is called, the process will create a new stack frame for this function. This frame will be place right after that of its caller. When the function returns, this frame is clean from memory by shrinking the stack (stack pointer goes up). The following program illustrates to identify the relative location of stack frames create by nested function calls.

Example: Trace function calls

```
1  #include <stdio.h>
2  void func (unsigned long number) {
3          unsigned long local_f = number;
4          printf("%2lu -> %p\n", local_f, &local_f);
5          local_f--;
6          if (local_f > 0) {
7                  func(local_f);
8          }
9  }
10 int main() {
11         func(10);
12 }
```

Similar to heap, stack has a pointer name stack pointer (as heap has program break) which indicate the top of the stack. To change stack size, we must modify the value of this pointer. Usually, the value of stack pointer is hold by stack pointer register inside the processor. Stack space is limited, we cannot extend the stack exceed a given size. If we do so, stack overflow will occurs and crash our program. To identify the default stack size, use the following command

```
1  ulimit -s
```

Different from heap, data of stack are automatically allocated and cleaned through procedure invocation termination, Therefore, in C programming, we do not need to

allocate and free local variables. In Linux, a process is permitted to have multiple stack regions. Each regions belongs to a thread.

2 Introduction to thread

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.

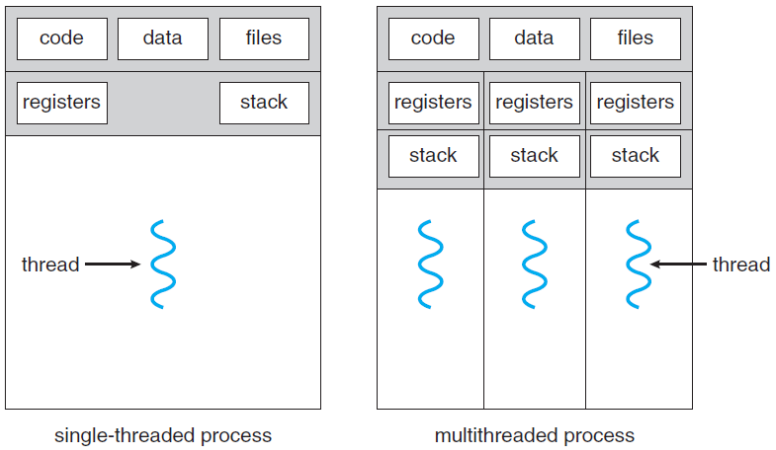

Figure 2.1: Single-threaded and multithreaded processes.

Figure 2.1 illustrates the difference between a traditional single-threaded process and a multithreaded process. The benefits of multithreaded programming can be broken down into four major categories:

- Responsiveness
- Resource sharing
- Economy
- Scalability

Q1. What resources are used when a thread is created? How do they differ from those used when a process is created?

Multicore programming Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system.

Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

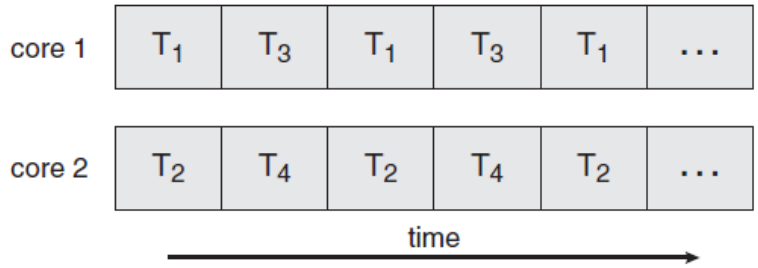

Figure 2.2: Parallel execution on a multicore system.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core, as Figure 2.2 shown.

Q2. Is it possible to have concurrency but not parallelism? Explain.

3 How to create multiple threads?

3.1 Thread libraries

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java. In this lab, we use POSIX Pthread on Linux and Mac OS to practice with multithreading programming.

Creating threads

```
pthread_create (thread, attr, start_routine, arg)
```

Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
- `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or `NULL` for the default values.
- `start`: the C routine that the thread will execute once it is created.
- `arg`: A single argument that may be passed to texttttstart_routine. It must be passed by reference as a pointer cast of type void. `NULL` may be used if no argument is to be passed.

Example: Pthread Creation and Termination

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
        long tid;
        tid = (long)threadid;
        printf("Hello World! It's me, thread #%ld!\n", tid);
        pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t],NULL, PrintHello, (void *)t);
        if (rc){
          printf("ERROR; return from pthread_create() is %d\n", rc);
          exit(-1);
        }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Passing argument to Thread We can pass a structure to each thread such as the example below. Using the previous example to implement this example:

```c
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *thread_arg)
{
```

```
11      struct thread_data *my_data;
12      ...
13      my_data = (struct thread_data *) thread_arg;
14      taskid = my_data->thread_id;
15      sum = my_data->sum;
16      hello_msg = my_data->message;
17      ...
18  }
19
20  int main (int argc, char *argv[])
21  {
22      ...
23      thread_data_array[t].thread_id = t;
24      thread_data_array[t].sum = sum;
25      thread_data_array[t].message = messages[t];
26      rc = pthread_create(&threads[t], NULL, PrintHello,
27           (void *) &thread_data_array[t]);
28      ...
29  }
```

Joining and Detaching Threads "Joining" is one way to accomplish synchronization between threads, For example:

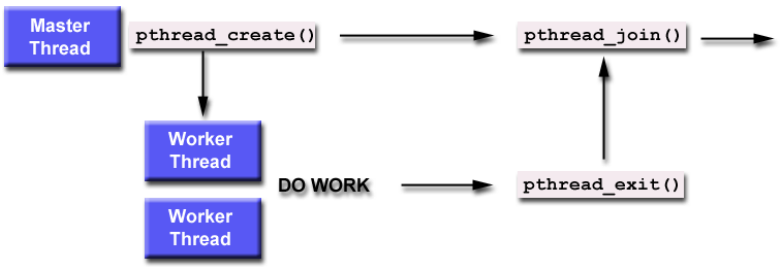

Figure 3.1: Joining threads.

- The `pthread_join()` subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to `pthread_exit()`.
- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.

3.2 Multithread programming

Problem: Constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  int sum; /* this data is shared by the thread(s) */
5  void *runner(void *param); /* threads call this function */
6
7  int main(int argc, char *argv[])
8  {
9          pthread_t tid; /* the thread identifier */
10         pthread_attr_t attr; /* set of thread attributes */
11
12         if (argc != 2) {
13                 fprintf(stderr,"usage: a.out <integer value>\n");
14                 return -1;
15         }
16
17         if (atoi(argv[1]) < 0) {
18                 fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
19                 return -1;
20         }
21         /* get the default attributes */
22         pthread_attr_init(&attr);
23         /* create the thread */
24         pthread_create(&tid, &attr, runner, argv[1]);
25         /* wait for the thread to exit */
26         pthread join(tid,NULL);
27
28         printf("sum = %d\n",sum);
29         }
30
31 /* The thread will begin control in this function */
32 void *runner(void *param)
33 {
34         int i, upper = atoi(param);
35         sum = 0;
36         for (i = 1; i <= upper; i++)
37         sum += i;
38         pthread exit(0);
39 }
```

4 Exercise (Required)

4.1 Personal Exercise (5 points)

Problem 1 (2.5 points) An interesting way of calculating `pi` is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows: Suppose you have a circle inscribed within a square, as shown in Figure 4.1.

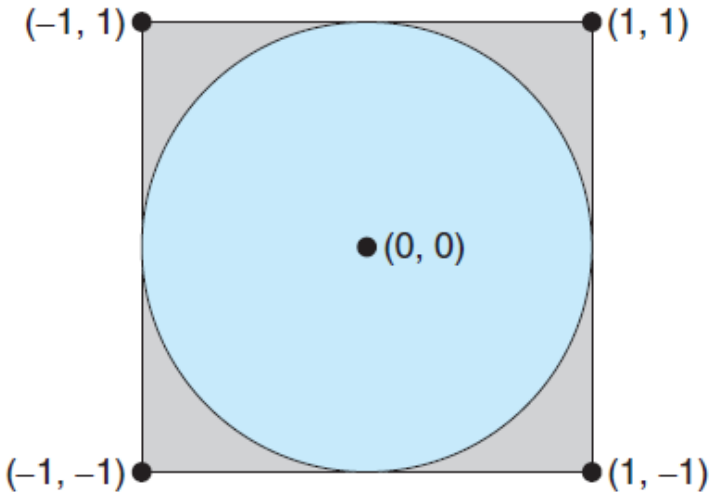

Figure 4.1: Monte Carlo technique for calculating pi.

(Assume that the radius of this circle is 1.) First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate `pi` by performing the following calculation:

```
pi = 4 x (number of points in circle) / (total number of points)
```

As a general rule, the greater the number of points, the closer the approximation to `pi`. However, if we generate too many points, this will take a very long time to perform our approximation. Solution for this problem is to carry out point generation and calculation concurrently. Suppose the number of points to be generated is *nPoints*. We create N separate threads and have each thread to create only *nPoints* / N points and count the number of points fall into the circle. After all threads have done their job we then get the total number of points inside the circle by combining the results from each thread. Since the total number of points has been generated equal to *nPoint*, the results of this method is equivalent to that of running a single process program. Furthermore, as threads run concurrently and the number of points each thread has to handle is much fewer than that of a singe process program, we can save a lot of time.

Write two programs implementing algorithm describe above (one serial version and one multi-thread version). The program takes the number of points to be generated from user then creates multiple threads to approximate pi. Put all of your code in two files named *"pi_serial.c"* and *"pi_multi-thread.c"*. The number of points is passed to your program as an input parameter. For example, if your executable file is *pi* then to have your program calculate pi by generating one million points, we will use the follows command:

```
./pi_serial 1000000
./pi_multi-thread 1000000
```

Requirement: The multi-thread version must have some speed-up compared to the serial version. You must print the execution time of both versions to demonstrate this speedup, together with the caculated `pi` value. There must be at least 2 targets in the Makefile (*Makefile1*): *pi_serial* and *pi_multi-thread* to compile the two program.

PROBLEM 2 (2.5 POINTS) Given the content of a C source file named *"code.c"*

```
1  #include <stdio.h>
2  #include <pthread.h>
3  void * hello(void * tid) {
4          printf("Hello from thread %d\n", (int)tid);
5  }
6  int main() {
7          pthread_t tid[10];
8          int i;
9          for (i = 0; i < 10; i++) {
10                 pthread_create(&tid[i], NULL, hello, (void*)i);
11         }
12         pthread_exit(NULL)
13 }
```

Since threads run concurrently, the output produced by threads will appear randomly and in an unpredictable way. Modify this program to have those threads to print their thread ID in ascending order every time we run the program as follows:

```
1  ## ./code
2  Hello from thread 0
3  Hello from thread 1
4  Hello from thread 2
5  Hello from thread 3
6  Hello from thread 4
7  Hello from thread 5
8  Hello from thread 6
9  Hello from thread 7
10 Hello from thread 8
11 Hello from thread 9
```

Write a Makefile (*Makefile2*) to compile the program. The executable file name is *code*.

Please remember to put all of your source code in a ZIP file in the correct format (*<id>.zip*). You must submit your ZIP file to E-learning before the deadline. No need to put each exercise in a separated folder.

4.2 Group Exercise (5 points)

There are two questions (Q1 and Q2) in the section 2. Your group have to answer them with detailed explanation together, with the following questions:

1. Describe an example that we need to create a new thread instead of invoking a new process? How about the contrary?
2. Is multithreading faster than multiprocessing (in case using the same number of threads and processes)?

The file format must be *answer.txt*. You only need one representative of your group to submit your answer to BKEL before your classes finishes.

All other filenames will not be accepted.

Revision History

Revision	**Date**	**Author(s)**	**Description**
1.0	22.09.16	Phuong-Duy Nguyen, Duc-Hai Nguyen, Minh Thanh Chung	created
2.0	10.03.19	TK Pham	Change Exercises
3.0	06.05.20	Thanh Le-Hai Hoang	Update Exercises requirements