HCMC University Of Technology
Faculty of Computer Science & Engineering

– CO2018 Operating Systems – Fall 2018 –

Lab 5
Thread

October 5, 2018

Goal: Getting familiar with the concept of multithreaded programming.

Contents:

- Study about the stack memory region.
- Review the concept of thread.
- Multithreaded programming

Expected result: After completing this lab, students are expected to (1) perceive the fundamental differences between threads and processes, and (2) be able to develop a multithreaded program.

1 Stack

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

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

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

Listing 1: Tracing function calls

```
#include <stdio.h>
void func (unsigned long number) {
   unsigned long local_f = number;
   printf("#%2l --> %p\n", local_f, &local_f);
   local_f--;
   if (local_f > 0) {
      func(local_f);
   }
}
int main() {
   func(10);
}
```

Similar to heap, stack has a pointer name stack pointer (as heap has a program break) which indicates the top of the stack. To change the stack size, we must modify the value of this pointer. Typically, the value of a stack pointer is stored at the stack pointer register inside the processor. A stack space is limited, we cannot extend the stack to surpass a given size. If we do so, a stack overflow error will occur and crash our program. To identify the default stack size, use the following command:

```
$ ulimit -s
```

Different from heap, data of stack are automatically allocated and cleaned. 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 region 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

OS-related resources, such as opened 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.

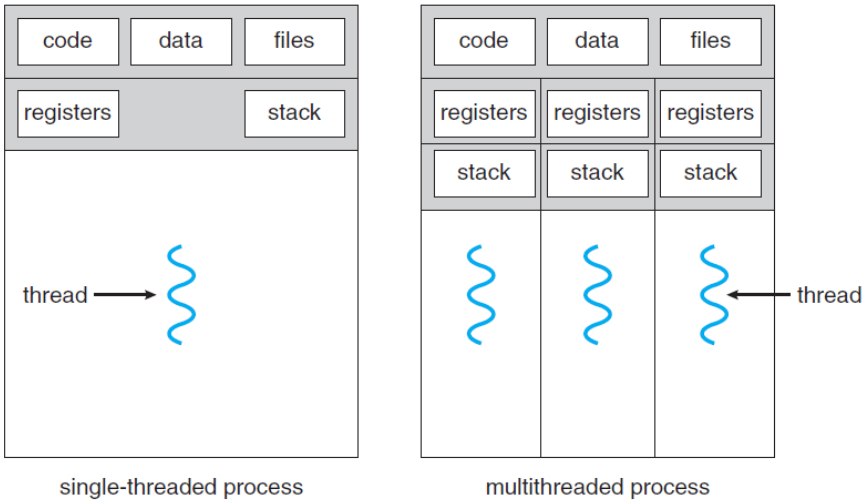

Figure 2.1: Single-threaded and multithreaded processes.

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

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.

3 Multithreaded programming

3.1 Thread libraries

A thread library provides programmers 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 manually created.

- `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 `start_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.

Listing 2: pthread creation and termination

```
#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 arguments to Thread We can pass a structure to each thread in the same manner as the example below (this is just a skeleton code):

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *thread_arg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) thread_arg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

3.2 Multithread programming

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

- 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()`.

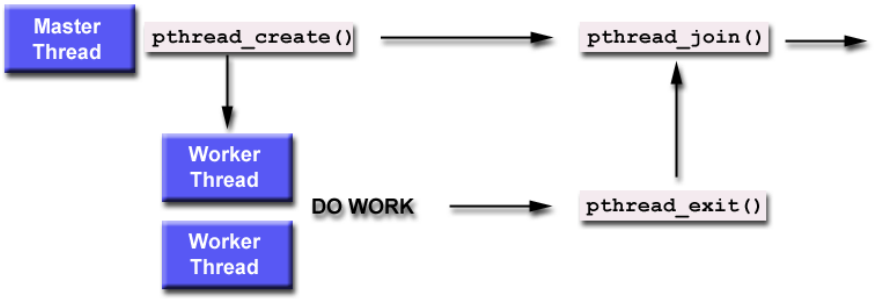

Figure 3.1: Joining threads.

- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.

Below is an example of a multithreaded program that calculates the sum of non-negative integers in a separate thread.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }

   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
```

```
void *runner(void *param)
{
   int i, upper = atoi(param);
35 sum = 0;
   for (i = 1; i <= upper; i++) sum += i;
   pthread_exit(0);
}
```

4 Exercises

4.1 Questions

Question 1: What resources are used when a thread is created? How do they differ from those used when a process is created?

Question 2:

a) How is concurrency different from parallelism? Provide an example to clarify your answer. In developing your answer, be sure to address the definition of concurrency and parallelism.

b) Then explain how it is possible to have concurrency but not parallelism.

4.2 Programming exercises

Problem 1: 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.

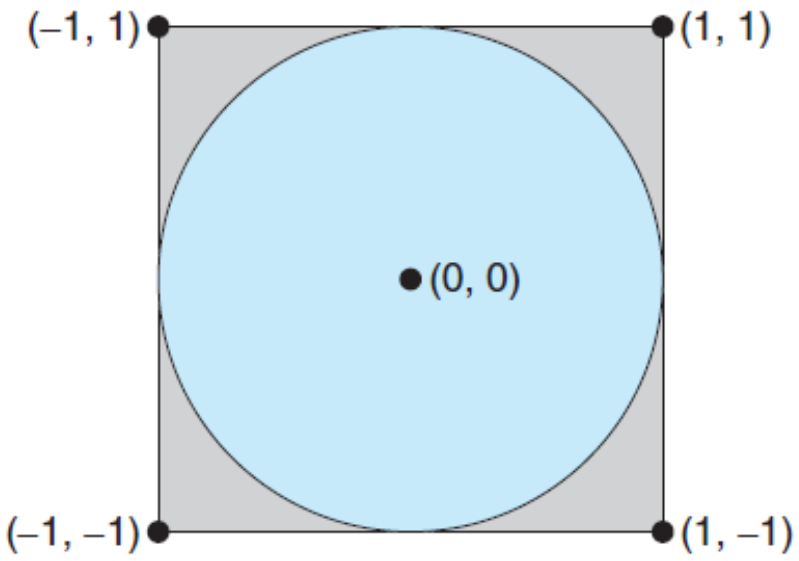

Figure 4.1: Monte Carlo technique for calculating pi.

(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 reside 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. A solution for this problem is to carry out points generation and calculation concurrently.

Write a multithreaded program implementing the mechanism above and it must perform concurrently to reduce the task of counting points. Put all of your code in a single file named *"pi.c"*. The program takes the number of points to be generated from an input argument then creates multiple threads to approximate `pi`. For example, if your executable file is *pi*, use the following command to execute it:

```
./pi 1000000
```

Requirements:

- Explain how your program solve the problem concurrently (e.g., how did you distribute the counting task to each thread?).
- How does your method augment the performance? (i.e., determine the speed-up, the difference in running time between multithreaded and single-threaded program. You should calculate the time reduced in terms of number of threads)
- Calculate the execution time needed to handle 100 million points. It should be within 5 seconds.

Problem 2: Given the content of a C source file named *"code.c"*

Listing 3: code.c

```c
#include <stdio.h>
#include <pthread.h>
void * hello(void * tid) {
   printf("Hello from thread %d\n", (int)tid);
}
int main() {
   pthread_t tid[10];
   int i;
   for (i = 0; i < 10; i++) {
      pthread_create(&tid[i], NULL, hello, (void*)i);
   }
   pthread_exit(NULL)
}
```

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

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

```
Hello from thread 4
Hello from thread 5
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 9
```

4.3 Submission

Please put your report in a PDF file named **report.pdf**. Move all of your code files (**code.c** and **pi.c**) and the report into a single directory <studentID>.zip and submit to BKeL.