TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC QUỐC GIA TP.HCM
TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
268 Lý Thường Kiệt, Phường 14, Quận 10, TP.Hồ Chí Minh
Điện thoại: 84-8-3864 7256 ext 5371 – Fax: 84-8-3865 8687
Website: **www.cce.hcmut.edu.vn**
E-mail: **dientoan@hcmut.edu.vn**

# Lab 3 - Thread

## 1. Theory

### 1.1. Definition

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 conventional (or heavy-weight) 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.

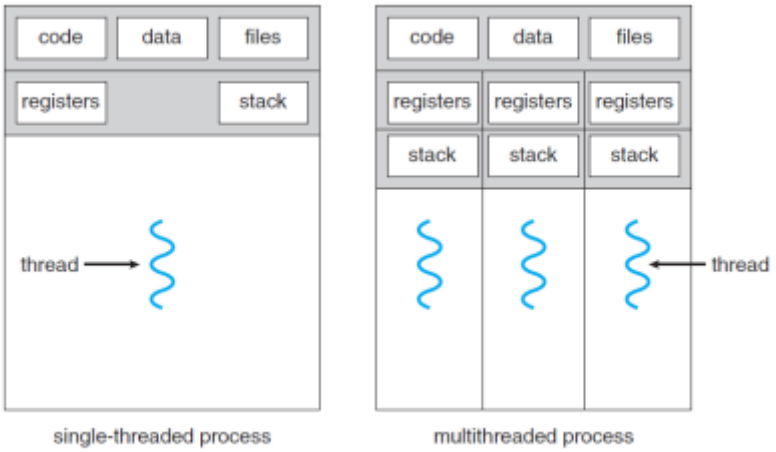


**Fig 1.1: Single-threaded and multithreaded processes**

**Figure 1.1** illustrates the difference between a traditional single-threaded process and a multithreaded process. When a program creates another thread, though, nothing is copied. The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value.

The benefits of multithreaded programming can be broken down into four major categories:

- Responsiveness: Multithreading in an interactive application may allow a program to continue running even if a part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. In a non-multi threaded environment, a server listens to the port for some request and when the request comes, it processes the request and then resumes listening to another request. The time taken while processing a request makes other users wait unnecessarily. Instead a better approach would be to pass the request to a worker thread and continue listening to port.
- Resource sharing: As you've already known, resource sharing in multi-processing is not native and requires other libraries and techniques to do that. In contrast, multithreading can easily share the variables, memories, code, etc..

- Economy: Creating and managing multi processes can consume numerous resources in terms of computer hardware and time. In contrast, threads share memory with the process it belongs to, it is more economical to create and context switch threads.
- Scalability: The benefits of multi-programming greatly increase in case of multiprocessor architecture, where threads may be running parallel on multiple processors *(We only limit the scope of this section in C/C++ language, in other programming languages, for example Python, the situation can be very different!)*. If there is only one thread then it is not possible to divide the processes into smaller tasks that different processors can perform.

## 1.2. Libraries supporting for manipulating threads

There are three primary libraries for using threads in Linux systems:

- std::thread: requires C++11, so that it cannot run on old compilers. It includes very basic and common features. It is included in the standard library therefore you do not need to include any further libraries to use it.
- boost::thread: it was created by the same authors of std::thread. Moreover, it is compatible with both old and new compilers. It has similar features of the standard one however, you should include an external dependency to use it.
- pthread (a.k.a POSIX threads): it has more features than both of above libraries such as: scheduling policies, etc. However, it is only compatible with POSIX systems. This library is also not included in the C/C++ standard library. ***(In this course, we only focus on this library.)***

## 1.3. Thread Creation & Termination

### 1.3.1. Creating and Exiting a thread

Similar to process, threads are also identified by their ID. Initially, a program comprises a single, default thread. In order to create new threads, you can use the following function:

```
int pthread_create(pthread_t *restrict thread, const pthread_attr_t
*restrict attr, void * (*start_routine) (void *), void *restrict arg);
```

- **thread:** An opaque, unique identifier for the new thread returned by the subroutine. (thread ID)
- **attr:** An opaque attribute object that may be used to set thread attributes. You can specify a thread attribute object, or NULL for the default values.
- **start_routine**: the C routine that the thread will execute once it is created.
- **arg:** A single argument that may be passed to textttstart_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.

In some case, you need to terminate the current thread immediately, you can use the following function:

```
void pthread_exit(void *value_ptr);
```

- **value_ptr:** this variable is used to pass the value to the joining function at the main thread after this thread is terminated. More Details in Section 2.1.3.3.

For example, (Note: you should include ***<pthread.h>*** in your source code and add ***-pthread*** argument when you combine the source code containing pthread functions, e.g. *gcc -pthread hello.c*)

```
#include <pthread.h>
#include <cstdio>
void *PrintHello(void *inputString){
    char * castedInput = (char *) inputString;
    printf("Hello %s\n", castedInput);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]){
    pthread_t threadID;
    const char *inputString = "World";
    int status = pthread_create(&threadID, NULL, &PrintHello, (void
*) inputString);
    pthread_exit(NULL);
}
```

### 1.3.2. Sharing complicated data structures between threads

In the previous section, you only pass some pre-defined, simple variable from the main thread to the new thread (the variable named inputString). However, you also can share user-defined structures between new threads. For example,

```
#include <pthread.h>
#include <cstdio>
#include <sys/types.h>
#include <unistd.h>

struct sharedVariable {
  char *value1;
  int value2;
};

void *PrintHello(void * input){
    struct sharedVariable * newInput = (struct sharedVariable *)
input;
    printf("Hello %s at thread %d\n", newInput->value1,
newInput->value2);
    newInput->value2 += 1;
```

```
    newInput->value1 = "Computer";
    pthread_exit(NULL);
}
int main(int argc, char *argv[]){
    pthread_t threadID;
    struct sharedVariable input;
    input.value1 = "World";
    input.value2 = 1;
    pthread_create(&threadID, NULL, &PrintHello,(void *) &input);
    sleep(2);
    pthread_create(&threadID, NULL, &PrintHello,(void *) &input);
    pthread_exit(NULL);
}
```

### 1.3.3. Joining Multi Threads

Normally, the main thread may independently terminate before its child threads finish. It can lead to several bugs and issues. You can force the main thread to wait for these threads by the following function and receive the response values:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- **thread:** thread ID.
- **value_ptr**: pthread_join will receive this pointer from pthread_exit().

For example,

```
#include <pthread.h>
#include <cstdio>
#include <cstdlib>
#define NUM_THREADS 5
void *RandomizeValue(void *){
    int value = rand() % 6 + 1;
    int *result = (int *) malloc(sizeof(int));
    *result = value;
    return (void *) result; // pthread_exit((void *) result);
}
int main(int argc, char *argv[]){
    pthread_t threadIDs[NUM_THREADS];
    for(int i=0; i<NUM_THREADS; i++){
        pthread_create(&threadIDs[i], NULL, &RandomizeValue,(void *)
NULL);
    }
    for (int i=0; i<NUM_THREADS; i++){
```

```
        int* response;
        pthread_join (threadIDs[i], (void **) &response);
        printf("DONE with random value: %d\n", *response);
    }
    pthread_exit(NULL);
}
```

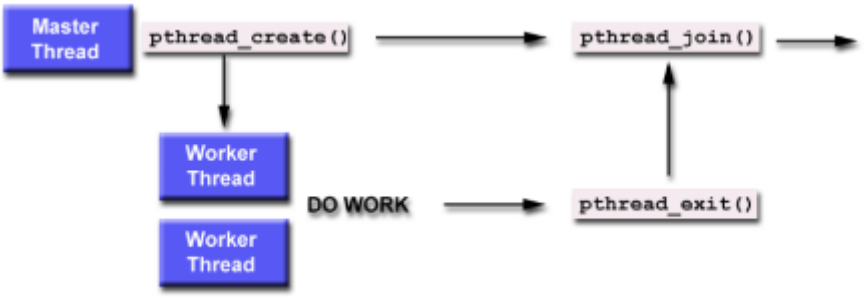


**Fig 1.2: An example of multi-threading workflow with pthread.**

### 1.4. Thread Pool

Naively, you create a new thread for each new task, which can lead to thousands of zombie threads existing by chance and it can lower your system's performance, wasting the system resources. A typical solution is that you can limit the number of available threads and for each incoming task, you can reuse the old threads. A *thread pool* is a similar idea. On most systems, it's impractical to have a separate thread for every task that can potentially be done in parallel with other tasks, but you'd still like to take advantage of the available concurrency where possible. A thread pool allows you to accomplish this; tasks that can be executed concurrently are submitted to the pool, which puts them on a queue of pending work. Each task is then taken from the queue by one of the *worker threads*, which executes the task before looping back to take another from the queue.

```
#include <tbb/concurrent_queue.h>
#include <iostream>
#include <unistd.h>
#include <pthread.h>
using namespace std;
#define THREADS 3

typedef struct {
    int data;
} work_t;
tbb::concurrent_queue<work_t> gqueue;
pthread_t threads[THREADS];
/* function headers */
```

```
void * worker(void *);
void submit_work(int data);

int main(void) {

    /* start all threads */
    for (int i = 0; i < THREADS; i++)
        pthread_create(&threads[i], NULL, worker, NULL);

    // Note: You should init workers before submitting jobs

    submit_work(rand());
    submit_work(100);
    submit_work(90000);
    submit_work(89899);

    /* wait all threads to finish */
    for (int i = 0; i < THREADS; i++)
        pthread_join(threads[i], NULL);

    return EXIT_SUCCESS;
}

/**
 * Adds a calculation task to queue.
 */
void submit_work(int data) {
    /* dynamically allocate a work task */
    work_t * work = (work_t *) malloc(sizeof(work_t));
    work->data = data;
    gqueue.push(*work);
}

/**
 * Worker thread. Looks for new tasks to execute.
 */
void * worker(void *) {
    int result;
    work_t work_info;
    for (;;) {
```

```
        while (!send_queue.try_pop(work_info));
        cout << "WORKER RECEIVE DATA: " << work_info.data << "\n";
        // DO SOMETHING
        sleep(rand()%5);
    }
    pthread_exit(NULL);
}
```

## 2. Exercises:

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 Fig 2.1.

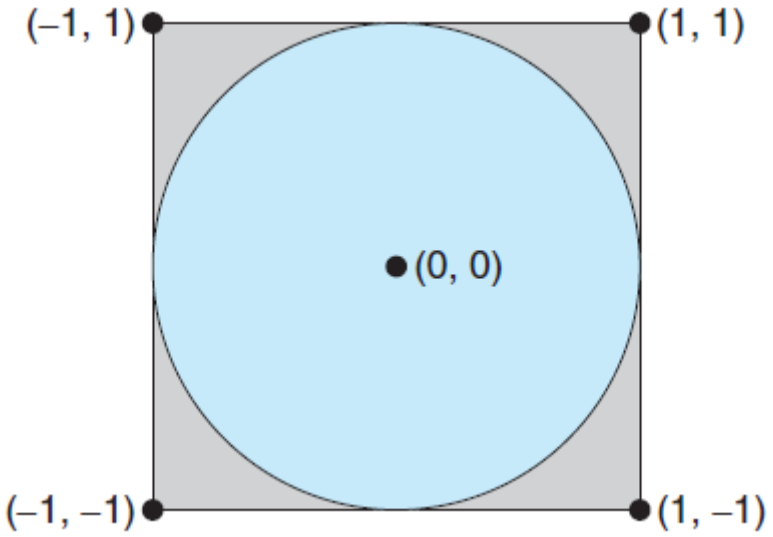


**Fig 2.1. Monte Carlo technique for calculating pi.**

- (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 \texttt{pi} by performing the following calculation:

$$pi = 4 \frac{number\ of\ points\ in\ circle}{total\ number\ of\ points}$$

- 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 (processes) and have each

thread (process) create only $\frac{nPoints}{N}$ points and count the number of points that fall into the circle. After all threads (processes)have done their job we then get the total number of points inside the circle by combining the results from each thread (process). Since the total number of points has been generated equal to ***nPoints***, 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 single process program, we can save a lot of time.

Implement the Monte Carlo technique for calculating pi ('ve been already introduced in the previous session) using:

a. a single thread (serial version).
b. multi-processing programming.
c. multi-threading programming.

2. Thread Pool:

Download the file from url:
*https://drive.google.com/file/d/1vK755T6kyFD_MS7dLFZTDbOvdFrgxC-n/view?usp=sharing*

This file consists of 100 files, each of which contains the ratings of users with movies (from 0->5). Each file is in the following format:

***userID, movieID, rating***
***userID, movieID, rating***
**…**

Write a program to compute the average rating of each movie. There are about 10.681 movies and 71.567 users with about a total of 10.000.054 ratings. (If your computer doesn't have enough memory to run the program, you only need to compute the average ratings of the first 1000 movies, which have IDs ranging from 0 to 9999). You should use a thread pool with 4 workers.