BÀI 6
LẬP TRÌNH ĐA TUYẾN (MULTI –THREAD)

I. Lý Thuyết

1. Tuyến là gì ? Tại sao phải dùng tuyến (thread)

Tuyến là một phần của tiến trình sở hữu riêng ngăn xếp (stack) và thực thi độc lập ngay trong mã lệnh của tiến trình. Nếu như một HĐH có nhiều tiến trình thì bên trong mỗi tiến trình lại có thể tạo ra nhiều tuyến hoạt động song song với nhau tương tự như cách tiến trình hoạt động song song bên trong HĐH.

Ưu điểm của tuyến là chúng hoạt động trong cùng không gian địa chỉ của tiến trình. Cơ chế liên lạc giữa các tuyến đơn giản và hiệu quả.

Đối với HĐH, chi phí chuyển đổi ngữ cảnh của tiến trình cao và chậm hơn chí phí chuyển đổi ngữ cảnh dành cho tuyến.

2. Tạo lập và hủy tuyến

Khi chương trình chính bắt đầu, nó chính là một tuyến. Tuyến điều khiển hàm **`main()`** được gọi là tuyến chính. Các tuyến khác do tiến trình tạo ra sau đó được gọi là tuyến phụ. Mỗi tuyến được cung cấp cho một số định danh gọi là thread ID. Để tạo ra một tuyến mới ngoài tuyến chính, bạn gọi hàm **`pthread_create().`** Hàm này được khai báo như sau:

```
#include <pthread.h>

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

Hàm **`pthread_create ()`** nhận 4 tham số, tham số thứ nhất có kiểu cấu trúc **`pthread_t`** để lưu các thông tin về tuyến sau khi tạo ra. Tham số thứ hai dùng để đặt thuộc tính cho tuyến (trong trường hợp ta đặt giá trị **`NULL`** thì tuyến được tạo ra với các thuộc tính mặc định). Tham số thứ ba là địa chỉ của hàm mà tuyến sẽ dùng để thực thi. Tham số thứ tư là địa chỉ đến vùng dữ liệu sẽ truyền cho hàm thực thi tuyến.

3. Chờ tuyến kết thúc

a. Chờ tuyến hoàn thành xong tác vụ

Tương tự như tiến trình dùng hàm **`wait()`** để đợi tiến trình con kết thúc, bạn có thể gọi hàm **`pthread_join()`** để đợi một tuyến kết thúc.

```
#include <pthread.h>

int pthread_join (pthread_t th, void* thread_return);
```

`th` là tuyến mà bạn muốn chờ, **`thread_return`** là con trỏ đến vùng chưa giá trị trở về của tuyến.

b. Chờ đồng thời nhiều tuyến

Thường trong các ứng dụng dịch vụ hoạt động theo mô hình khách chủ (client/server), trình chủ (server) của bạn phải mở nhiều tuyến để phục vụ trình khách. Hay trong các ứng dụng chò trơi bạn phải mở cùng lúc nhiều tuyến, mỗi tuyến thực hiện thao tác điều khiển một nhân vật hoạt hình nào đó. Kiểm soát và chờ đồng thời nhiều tuyến, bạn cũng dùng hàm **`pthread_join ().`**

4. Đồng bộ hóa tuyến với đối tượng mutex

Một trong những vấn đề quan tâm hàng đầu của việc điều khiển lập trình đa tuyến trong cùng không gian địa chỉ của tiến trình đó là đồng bộ hóa. Bạn phải đảm bảo được nguyên tắc 'các tuyến không dẫm chân lên nhau'. Ví dụ một tuyến chuẩn bị để đọc dữ liệu từ đĩa, thao tác đọc chưa hoàn tất thì một tuyến khác đã ghi đè dữ liệu mới lên dữ liệu cũ. Hay đơn giản và thường gặp hơn đó là xảy ra đụng độ khi truy cập và xử lý biến chung.

Để giải quyết tranh chấp và xử lý đồng bộ hóa chúng ta sử dụng một khái niệm gọi là mutex.

a.Mutex là gì

Mutex thực sự là một cờ hiệu, hay đối với hệ thống, mutex là một đối tượng mang hai trạng thái: đang được sử dụng và chưa sử dụng (trạng thái sẵn sàng).

Khi mutex bật, một tuyến sẽ bước vào sử dụng tài nguyên và tắt mutex. Tuyến khác sẽ không sử dụng được tài nguyên cho đến khi tuyến trước đó bật lại mutex ở trạng thái sẵn sàng.

b. Tạo và khởi động mutex

Để tạo ra đối tượng mutex, trước hết bạn cần khai báo biến kiểu cấu trúc **pthread_mutex_t**, đồng thời khởi tạo giá trị ban đầu cho biến này. Các đơn giản nhất để khởi tạo cấu trúc mutex là dùng hằng định nghĩa trước **PTHREAD_MUTEX_INITIALIZER**. Mã khai báo mutex thường có dạng sau:

```
pthread_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Một điều quan trọng bạn cần lưu ý là mutex khởi tạo theo cách này gọi là "mutex cấp tốc". Đối tượng mutex này không thể bị khóa hai lần bởi cùng một tuyến. Trong tuyến, nếu bạn đã gọi hàm khóa mutex này và thực hiện khóa mutex lần nữa, bạn sẽ rơi vào trạng thái khóa chết (deadlock).

Có một kiểu mutex khắc phục được nhược điểm trên, đó là mutex cho phép khóa lặp (recursive mutex). Trong cùng một tuyến, nếu bạn khóa mutex nhiều lần thì không có vấn đề gì xảy ra, nhưng bù lại muốn giải phóng mutex, bạn phải tháo khóa bằng đúng số lần bạn đã thực hiện gọi hàm khóa mutex. Mutex kiểu này thường được khởi động bằng hằng **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP.**

Bạn cũng có thể gọi hàm **pthread_mutex_init ()** để thực hiện cùng chức năng khởi tạo mutex:

```
#include <pthread.h>
int pthread_mutex_init ( pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* mutexattr);
```

mutex là con trỏ đến biến cấu trúc **pthread_mutex_t** mà bạn muốn khởi tạo. **mutexattr** là các thuộc tính của mutex (mutex đơn hay mutex cho phép khóa lặp). Nếu bạn đặt trị NULL thì **mutex** với thuộc tính mặc định sẽ được tạo ra. Cách thứ hai để khởi tạo mutex sẽ là:

```
int res;
pthread_mutex_t* mutex;
res = thread_mutex_init (mutex, NULL);
if (res != 0)
{
    perror ("Initialize mutex fail");
}
```

c. Khóa và tháo khóa cho mutex

Để khóa mutex bạn có thể sử dụng hàm **pthread_mutex_lock (),** nếu không khóa được (mutex đã bị tuyến khác khóa trước đó) hàm sẽ đặt tuyến hiện hành vào trạng thái ngủ (chờ). Trong trường hợp này

khi mutex được tháo khóa, tuyến hiện hành sẽ được đánh thức dậy để tiếp tục thử khóa mutex trước khi đi vào sử dụng tài nguyên. Dưới đây là cách khóa mutex:

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
int rc = pthread_mutex_lock (&a_mutex);
if (rc)/*Lỗi phát sinh*/
{
    perror (“pthread_mutex_lock_error”);
    pthread_exit (NULL);
}
/*Mutex đã được khóa, tuyến của bạn có thể sử dụng tài nguyên một cách an toàn ở đây*/
...
…
```

Một khi không cần sử dụng độc quyền tài nguyên nữa, bạn nên gọi hàm **pthread_mutex_unlock ()** để tháo khóa mutex trả lại quyền sử dụng tài nguyên cho tuyến khác. Bạn tháo khóa mutex như sau:

```
rc = pthread_mutex_unlock (&a_mutex);
if (rc)
{
    perror (“pthread_mutex_unlock error”);
    pthread_exit (NULL);
}
```

d. Hủy mutex

Sau khi sử dụng xong mutex bạn nên hủy nó. Sử dụng xong có nghĩa là không còn tuyến nào cần chiếm giữ mutex cho các cho tác khóa/tháo khóa nữa. Hàm **pthread_mutex_destroy ()** được dùng để hủy mutex.

```
rc = pthread_mutex_destroy (&a_mutex);
```

Sau khi gọi hàm hủy mutex, bạn không còn sử dụng được biến mutex được nữa. Để sử dụng lại biến mutex bạn cần thực hiện lại bước khởi tạo.

II. Thực hành

Bài 1: Chương trình tạo lập tuyến: chúng ta tạo hàm **do_loop ()** để in ra các số nguyên. Hàm **do_loop ()** này được gọi thực thi ở hai nơi: một trong tuyến chính (hàm main) và một trong tuyến phụ tạo ra bởi hàm **pthread_create ().**

thread_create.c

```
#include <stdio.h>
#include <pthread.h> /*Khai báo các hàm xử lý tuyến*/

/*Hàm thực thi tuyến*/
void* do_loop (void* data)
{
      int i; /*Bộ đếm cho tuyến*/
      /*Dữ liệu cho hàm pthread_create() truyền vào cho tuyến*/
      int me = (int*) data;

      for (i = 0; i < 5; i++)
```

```
	{
		sleep (1); /*Dừng*/
		printf (" '%d' - Got '%d' \n", me, i);
	}
	/*Chấm dứt tuyến*/
	pthread_exit (NULL);
}

/*Chương trình chính*/
int main (int agrc, char* argv[])
{
	int thr_id; /*Định danh tuyến*/
	pthread_t p_thread; /*Cấu trúc lưu trữ các thông tin về tuyến*/
	int a = 1; /*Định danh cho tuyến thứ nhất*/
	int b = 2; /*Định danh cho tuyến thứ hai*/

	/*Tạo tuyến*/
	thr_id = pthread_create (&p_thread, NULL, do_loop, (void*) a);

	/*Chạy do_loop trong tuyến chính*/
	do_loop ((void*)b);
	return 0;
}
```

Để biên dịch chương trình này, bạn cần phải dùng đến thư viện liên kết hỗ trợ lập trình tuyến là **libpthread**. Chúng ta biên dịch chương trình như sau:

```
$gcc thread_create.c -o thread_create -lpthread
```

Chạy chương trình với kết quả kết xuất

```
./thread_create
'2' - Got '0'
'1' - Got '0'
'2' - Got '1'
'1' - Got '1'
'2' - Got '2'
'1' - Got '3'
'2' - Got '3'
'1' - Got '3'
'2' - Got '4'
'1' - Got '4'
```

Bài 2: Chờ tuyến thực thi xong tác vụ

thread_wait.c

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

char message[] = "Hello World";

/*Hàm xử lý tuyến*/
void* do_thread (void* data)
{
	printf ("Thread function is executing ... \n");
```

```
	printf ("Thread data is %s\n", (char*) message);
	sleep (3);
	strcpy (message, "Bye !");
	pthread_exit ("Thank you for using my thread");
}

/*Chương trình chính*/
int main ()
{
	int res;
	pthread_t a_thread;
	void* thread_result;

	/*Tạo và thực thi tuyến*/
	res = pthread_create (&a_thread, NULL, do_thread, (void*) message);

	if (res != 0)
	{
		perror ("Thread created error\n");
		exit (EXIT_FAILURE);
	}

	/*Đợi tuyến kết thúc*/
	printf ("Waiting for thread to finish ...\n");
	res = pthread_join (a_thread, &thread_result);

	if (res != 0)
	{
		perror ("Thread wait error\n");
		exit(EXIT_FAILURE);
	}

	/*In kết quả trả về của tuyến*/
	printf ("Thread completed, it returned %s \n", (char*) thread_result);
	printf ("Message is now %s \n", message);
	return 0;
}
```

Biên dịch và chạy chương trình từ dòng lệnh. Kết quả kết xuất sẽ như sau:

```
$gcc thread_wait.c -o thread_wait -lpthread
$./thread_wait
Thread function is executing ...
Thread data is Hello World
Waiting for thread to finish ...
Thread completed, it returned Thank you for using my thread
Message is now Bye !
```

Bài 3: Chờ đồng thời nhiều tuyến: dùng mảng để lưu thông tin về danh sách các tuyến. Sau đó chương trình chính sẽ gọi **`pthread_join ()`** để chờ các tuyến trong danh sách kết thúc.

thread_multiwait.c

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

```c
#include <stdlib.h>
#include <pthread.h>

#define MAX_THREADS 6

void* do_thread (void* data);

int main ()
{
	int res;
	int thread_num;
	pthread_t a_thread [MAX_THREADS];

	void* thread_result;

	/*Khởi tạo danh sách các tuyến*/
	for (thread_num =1; thread_num < MAX_THREADS; thread_num++)
	{
		/*Tạo tuyến và lưu vào phần tử mảng*/
		res = pthread_create (&(a_thread [thread_num]), NULL, do_thread,
                                                        (void*) thread_num);

		if (res != 0)
		{
			perror ("Thread created error");
			exit (EXIT_FAILURE);
		}
		/*Dừng 1 giây*/
		sleep (1);
	}
	printf ("Waiting for threads to finish ...\n");

	/*Chờ danh sách các tuyến kết thúc*/
	for(thread_num = MAX_THREADS - 1; thread_num > 0;  thread_num--)
	{
		res = pthread_join (a_thread [thread_num], &thread_result);
		if (res != 0)
		{
			perror ("Thread exited error");
		}
		else
		{
			printf ("Pickup a thread\n");
		}
	}
	printf ("All thread completed \n");
	return 0;
}

/*Cài đặt hàm điều khiển tuyến*/
void* do_thread (void* data)
{
	int my_number = (int) data;
	printf ("Thread function is running. Data argument was %d\n", my_number);
	sleep (3);
	printf ("Finish - bye from %d\n", my_number);
```

```
}
```

Biên dịch chương trình:

$gcc thread_multiwait.c -o thread_multiwait -lpthread

Chạy chương trình:

$./thread_multiwait

```
Thread function is running. Data argument was 1
Thread function is running. Data argument was 2
Thread function is running. Data argument was 3
Thread function is running. Data argument was 4
Finish - bye from 1
Finish - bye from 2
Thread function is running. Data argument was 5
Finish - bye from 3
Waiting for threads to finish ...
Finish - bye from 4
Finish - bye from 5
Pickup a thread
Pickup a thread
Pickup a thread
Pickup a thread
Pickup a thread
All thread completed
```

Bài 4: Sử dụng mutex

Chương trình tạo ra hai tuyến. Tuyến thứ nhất liên tục tăng biến toàn cục **global_var** lên một đơn vị và dừng chờ trong 1 giây. Tuyến thứ hai ngược lại liên tục giảm biến toàn cục **global_var** đi một đơn vị và dừng chờ trong 2 giây.

thread_race.c

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

/*Biến dữ liệu toàn cục có thể truy xuất bởi cả hai tuyến*/
int global_var;

pthread_mutex_t a_mutex; /*Khai báo biến mutex toàn cục*/

/*Khai báo các hàm dùng thực thi tuyến*/
void* do_thread1 (void* data);
void* do_thread2 (void* data);

/*Chương trình chính*/
int main ()
{
      int res;
      int i;
      pthread_t p_thread1;
      pthread_t p_thread2;
```

```
	/*Khởi tạo mutex*/
	res = pthread_mutex_init (&a_mutex, NULL);
	/*Bạn cũng có thể khởi tạo mutex như sau
	a_mutex = PTHREAD_MUTEX_INITIALIZER;
	*/

	if (res != 0)
	{
		perror ("Mutex create error");
		exit (EXIT_FAILURE);
	}

	/*Tạo tuyến thứ nhất*/
	res = pthread_create (&p_thread1, NULL, do_thread1, NULL);
	if (res != 0)
	{
		perrror ("Thread create error");
		exit (EXIT_FAILURE);
	}

	/*Tạo tuyến thứ hai*/
	res = pthread_create (&p_thread2, NULL, do_thread2, NULL);
	if (res != 0)
	{
		perror ("Thread create error");
		exit (EXIT_FAILURE);
	}

	/*Tuyến chính của chương trình*/
	for (i = 1; i < 20; i++)
	{
		printf ("Main thread waiting %d second ... \n", i);
		sleep (1);
	}
	return 0;
}
/*Cài đặt hàm thực thi tuyến thứ nhất*/
void* do_thread1 (void* data)
{
	int i;
	pthread_mutex_lock (&a_mutex); /*Khóa mutex*/

	for (i=1; i <= 5; i++)
	{
		printf ("Thread 1 count: %d with global value %d \n", i, global_var++);
		sleep(1);
	}
	pthread_mutex_unlock (&a_mutex); /*Tháo khóa mutex*/

	printf ("Thread 1 completed !");
}

void* do_thread2 (void* data)
{
	int i;
```

```
      pthread_mutex_lock (&a_mutex); /*Khóa mutex*/

      for (i=1; i <= 5; i++)
      {
            printf ("Thread 2 count: %d with global value %d \n", i, global_var--);
            sleep(2);
      }
      pthread_mutex_unlock (&a_mutex); /*Tháo khóa mutex*/

      printf ("Thread 2 completed !");
}
```

Biên dịch và thực thi chương trình bạn sẽ nhận được kết xuất từ dòng lệnh như sau:

```
$./thread_race
Thread 1 count: 1 with global value 0
Main thread waiting 1 second ...
Main thread waiting 2 second ...
Thread 1 count: 2 with global value 1
Main thread waiting 3 second ...
Thread 1 count: 3 with global value 2
Main thread waiting 4 second ...
Thread 1 count: 4 with global value 3
Main thread waiting 5 second ...
Thread 1 count: 5 with global value 4
Thread 1 completed!
Main thread waiting 6 second ...
Thread 2 count: 1 with global value 5
Main thread waiting 7 second ...
Thread 2 count: 2 with global value 4
Main thread waiting 8 second ...
Main thread waiting 9 second ...
Thread 2 count: 3 with global value 3
Main thread waiting 10 second ...
Main thread waiting 11 second ...
Thread 2 count: 4 with global value 2
Main thread waiting 12 second ...
Main thread waiting 13 second ...
Thread 2 count: 5 with global value 1
Thread 2 completed!
Main thread waiting 14 second ...
```