Lab 4: Làm việc với luồng trong Linux

1 Lý thuyết

1.1 Tổng quan về luồng (threads)

1.1.1 Giới thiệu về luồng

Giống như tiến trình, luồng (threads) là một cơ chế cho phép ứng dụng (application) thực hiện các tác vụ đồng thời. Một tiến trình có thể chứa nhiều luồng, và các luồng này đều thực thi chương trình độc lập với nhau. Chúng cùng chia sẻ với nhau bộ nhớ toàn cục (global memory), bao gồm dữ liệu khởi tạo (initialized data), dữ liệu không khởi tạo (uninitialized data) và phân đoạn heap (heap segments).
Luồng trong tiến trình có thể được thực thi song song (execute concurrently). Trên một hệ thống đa vi xử lý (multiprocessor system), nhiều luồng có thể được thực thi song song. Nếu một luồng bị chặn (blocked) trên I/O, các luồng khác vẫn có thể hoạt động.
Cụ thể, vị trí của các ngăn xếp theo luồng (per-thread stacks) xen kẽ với các thư viện (shared library) và các vùng nhớ (shared memory), phụ thuộc vào thứ tự của các luồng được khởi tạo, thư viện tải lên (shared libraries loaded), và vùng nhớ đính kèm (shared memory regions attached). Hơn nữa, vị trí của các ngăn xếp theo luồng (per-thread stack) có thể thay đổi phụ thuộc vào Linux distro tương ứng.
Sử dụng đa luồng đồng thời có nhiều lợi thế (advantages) như sau:

- Chia sẻ thông tin giữa các luồng rất đơn giản và nhanh. Nó chỉ đơn thuần là sao chép dữ liệu vào trong biến toàn cục hoặc là phân đoạn heap (global variables or heap segments).
- Khởi tạo luồng nhanh hơn rất nhiều so với khởi tạo tiến trình (khoảng 10 lần hoặc hơn) do nhiều thuộc tính trong tiến trình con tạo ra bởi câu lệnh `fork()` được chia sẻ giữa các luồng.

Bên cạnh bộ nhớ toàn cục (global memory), luồng đồng thời còn chia sẻ nhiều thuộc tính bao gồm: PID & PPID, user and group IDs, giới hạn tài nguyên (resource limit), ... Thế nhưng, các luồng cũng có những đặc điểm đặc trưng để phân biệt với nhau bằng các thuộc tính quan trọng như thread ID, stack (local variables),...

1.1.2 Khởi tạo luồng (threads creation)

Khi một chương trình (program) bắt đầu, tiến trình kết quả (resulting process) lúc này sẽ chứa một luồng khởi tạo (initial or main thread). Trong mục này ta sẽ trình bày về cách tạo ra các luồng khác (threads creation) với hàm `pthread_create()` với cú pháp như sau:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start)(void*), void *arg);
Returns 0 on sucess, or positive number on error
```

Các tham số (parameter) được sử dụng trong hàm này gồm:

- `thread`: con trỏ trỏ đến biến `pthread_t`, vị trí mà `thread_ID` được lưu trữ.

- `attr`: con trỏ trỏ đến đối tượng (object) `pthread_attr_t` để xác định thuộc tính của thread (thread's attribute) (có thể là `\NULL` với thuộc tính mặc định).
- `start`: con trỏ trỏ tới hàm nơi mà thread mới tạo ra sẽ được thực thi (executed).
- `arg`: đối số (argument) được truyền vào hàm `start()`, dùng `\NULL` trong trường hợp mặc định.

Ví dụ: ta sẽ viết chương trình khởi tạo một luồng mới như sau:

```
#include <stdio.h>
#include <pthread.h>
    void* foo(void* arg){
        printf("Created a new thread");
    }
    int main(){
        pthread_t thread1;
        pthread_create(&thread1, NULL, foo, NULL);
        return 0;
    }
```

Ở đây sau khi sau khi thực thi cả 2 cặp lệnh sau trên terminal, ta đều không thấy có output ra tương ứng:

```
gcc thread.c -o thread
./thread

gcc thread.c -pthread
./thread
```

Ta có thể nhận thấy lý do output không xuất hiện trên terminal là luồng bị dừng (terminate) quá nhanh (lỗi này cũng giống như lỗi đã gặp với tiến trình đã được trình bày rất kĩ trong `lab3.pdf`), vì thế nên ta phải sử dụng thêm một hàm có chức năng chờ, không cho luồng dừng ngay lập tức như `sleep()` hay `scanf()` với tiến trình, đó là hàm `pthread_join()` (sẽ được thảo luận kĩ hơn ở phần sau). Ta sửa lại code khởi tạo luồng mới:

```
#include <stdio.h>
#include <pthread.h>
    void* foo(void* arg){
        printf("Created a new thread");
    }
    int main(){
        pthread_t thread1;
        pthread_create(&thread1, NULL, foo, NULL);
        pthread_join(thread1, NULL); // Wait for thread
        return 0;
    }
```

Sau khi thử cả 2 cặp lệnh để thực thi `thread.c` theo cú pháp dưới đây, ta đã thu được output đúng như mong đợi:

```
Chiaki@mx:~/C
$ gcc thread.c -o thread
Chiaki@mx:~/C
```

```
$ ./thread
Created a new threadChiaki@mx:~/C
$ gcc thread.c -pthread
Chiaki@mx:~/C
$ ./thread
Created a new threadChiaki@mx:~/C
```

1.1.3 Dừng luồng (threads termination)

Một tiến trình có thể bị dừng bởi các cách sau:

- Trả về từ hàm `main()`.
- Gọi hàm `pthread_exit()` (sẽ được thảo luận kĩ dưới đây).
- Bị dừng bởi một luồng khác (dùng hàm `pthread_cancel()`).
- Bị dừng bởi hàm `exit()`, lưu ý khi gọi hàm `exit()` điều đó đồng nghĩa với việc dừng toàn bộ **tiến trình**.

Cú pháp của hàm `pthread_exit()` như sau:

```
#include <pthread.h>
...
void pthread_exit(void *retval);
```

Với `retval` là một exit status tùy ý của luồng bị dừng, để cho đơn giản ta sẽ mặc định truyền cho hàm này tham số (parameter) `NULL`. Ví dụ:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
    void* foo(void* arg){
        printf("Created a new thread");
        pthread_exit(NULL);
    }

    int main(){
        pthread_t thread1;
        pthread_create(&thread1, NULL, foo, NULL);
        pthread_join(thread1, NULL);
        return 0;
    }
```

Output tương ứng là:

```
Chiaki@mx:~/C
$ gcc thread.c -pthread
Chiaki@mx:~/C
$ ./thread
Created a new threadChiaki@mx:~/C
```

1.1.4 ID của luồng (thread ID)

Như đã trình bày ở mục 1.1.1, mỗi luồng đều có một ID đặc trưng. Ở đây ta dùng hàm `pthread_self()` để xác định ID của luồng tương ứng:

```
#include <pthread.h>
pthread_t pthread_self(void);
Return thread ID of calling thread.
```

Ví dụ:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
    void* foo(void* arg){
        printf("Created a new thread\n");
        printf("The thread ID in function is: %d\n", pthread_self());
        pthread_exit(NULL);
    }

    int main(){
        pthread_t thread1;
        pthread_create(&thread1, NULL, foo, NULL);
        printf("Thread ID in main is %d\n", thread1);
        pthread_join(thread1, NULL);
        return 0;
    }
```

Ở đây khi ta thử dùng cặp lệnh để thực thi (execute) file này như ở trên thì có thể thấy vấn đề xuất hiện như sau:

```
Chiaki@mx:~/C
$ gcc thread.c -pthread
Chiaki@mx:~/C
$ ./thread
bash: ./thread: No such file or directory
Chiaki@mx:~/C
$ gcc thread.c -o thread
Chiaki@mx:~/C
$ ./thread
Thread ID in main is -2028964160
Created a new thread
The thread ID is: -2028964160
```

Dễ dàng thấy vấn đề nếu ta gõ lệnh `gcc thread.c -pthread` ở trước (ví dụ trên đều gõ `gcc thread.c -o thread` ở trước), thì không hề có một tiến trình hay file có khả năng thực thi (executable file) nào được tạo ra cả, vậy nên hiển nhiên câu lệnh `./thread` sẽ không hoạt động. Chính vì thế nên ta phải thay đổi cú pháp câu lệnh thành:

```
gcc -pthread thread.c -o thread # manually attach pthread library
                                # to executable file
gcc thread.c -o thread # for system can attach pthread library by default
```

Quá trình thực thi tiến trình (process) này có thể được hình dung qua 2 bước, đầu tiên là gắn cờ (flag) `-pthread` vào `thread.c` để compiler nhận thêm thư viện `pthread` tương ứng, sau đó `-o thread` để tạo tiến trình và file có khả năng thực thi `thread` (executable file) tương ứng. Để cho đầy đủ và chính xác trong việc thực thi file, cú pháp `gcc -pthread filename.c -o filename` sẽ được sử dụng trong toàn bộ bài báo cáo này.
Cũng giống như PID, TID (thread ID) cũng được sử dụng để nhận dạng (identify) các luồng tương ứng, để truyền (pass) chúng vào trong các hàm như `pthread_join()`, `pthread_detach()`,...
Để so sánh ID của luồng (TID comparison) xem chúng có bằng nhau hay không, ta sẽ sử dụng hàm `pthread_equal(pthread_t t1, pthread_t t2)` với cú pháp như sau:

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
Returns a non-zero value if t1 and t2 is the same, otherwise returns 0
```

Thế nhưng ta có thể thấy rõ ràng 2 TID được khởi tạo đồng thời trong cùng một tiến trình sẽ không thể có TID bằng nhau, hay nói cách khác mỗi TID được xác định là **duy nhất**, nên hiển nhiên hàm này được sử dụng để kiểm tra xem TID có là TID của hệ thống hay không như sau:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void* foo(){
    printf("The TID is: %d\n", pthread_self());
    pthread_exit(NULL);
}
int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &foo, NULL);
    pthread_create(&t2, NULL, &foo, NULL);
    printf("This is the first thread\n");
    printf("This is the second thread\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    if(pthread_equal(t1, pthread_self()))
        printf("First thread is system thread\n");
    else    printf("First thread is not system thread\n");
    if(pthread_equal(t2, pthread_self()))
        printf("Second thread is system thread\n");
    else printf("Second thread is not system thread\n");
    return 0;
}
```

Output trả về tương ứng là:

```
$ ./thread
This is the first thread
This is the second thread
The TID is: 905701056
The TID is: 897308352
First thread is not system thread
Second thread is not system thread
```

Hoặc để trả về kết quả ngược lại:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void* foo(){
    printf("The TID is: %d\n", pthread_self());
    pthread_exit(NULL);
}
int main(){
    pthread_t t1;
    pthread_create(&t1, NULL, &foo, NULL);
    t1 = pthread_self(); //Assign system thread value to t1
    pthread_join(t1, NULL);
    if(pthread_equal(t1, pthread_self()))
        printf("First thread is system thread\n");
    else    printf("First thread is not system thread\n");
    return 0;
}
/* Output: First thread is system thread */
```

1.1.5 Chờ luồng kết thúc (join thread)

Hàm `join_thread()` là một phiên bản hàm `wait()` của luồng. Nó cũng thực hiện chức năng tương tự: cho phép một luồng chờ sự dừng (termination) của một luồng khác. Thuật ngữ `join` có thể gây nhầm lẫn lúc đầu, nhưng nó được dùng với ý nghĩa "Wait for the thread to finish and rejoin the main thread's flow of execution.", tức là chờ luồng con thực thi xong rồi tái tham gia lại luồng chính (luồng đang thực thi). Như đã phân tích ở phần 1.1.2, ta có thể thấy hàm này có cú pháp như sau:

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
Returns 0 on success or a positive error number on failure.
```

Với các tham số (parameter) như sau:

- `thread`: TID của luồng được chờ.
- `retval`: con trỏ lưu trữ giá trị trả về của luồng bị dừng (terminated thread). Để cho đơn giản ta mặc định con trỏ này là `NULL`.

Các ví dụ về hàm này đã được phân tích và ứng dụng ở các phần trên.

1.1.6 Ngắt luồng (detach thread)

Hàm `pthread_detach()` được dùng để "ngắt" thẳng một luồng bất kì ra khỏi luồng chính (main thread) và một khi đã bị ngắt, nó không thể nào nhập (join) lại được nữa. Lúc này, hàm `main` sẽ tự động loại bỏ (clean up) luồng bị ngắt. Cú pháp của hàm này như sau:

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
Returns 0 on success or a positive error number on failure.
```

với tham số truyền vào duy nhất là `pthread_t thread`, tức là TID của luồng bị ngắt. Ví dụ:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void *foo(){
    printf("Thread is running\n");
    sleep(4);
    printf("Thread finished\n");
    pthread_exit(NULL);
}
int main(){
    pthread_t thread1;
    pthread_create(&thread1, NULL, &foo, NULL);
    pthread_detach(thread1);
    printf("Main thread continues...\n");
    return 0;
}
```

Output tương ứng sau khi thực thi `thread.c` là:

```
$ ./thread
Main thread continues... # (Instantaneously appears)
Thread is running # (Instantaneously appears)
Thread finished # (Appears after 4s)
```

Hiển nhiên ta có thể thấy hàm `pthead_detach(thread1)` đã ngắt `thread1` ra khỏi luồng chính (main thread), nên output xuất hiện `Main threads continues...` trước khi quay lại thực thi luồng `thread1`. Mặt khác, nếu ta đổi hàm `pthread_detach()` thành hàm `pthread_join()`, thì khối lệnh trong hàm `main` sẽ được thực thi theo thứ tự từ trên xuống dưới, trả về output là:

```
$ ./thread
Thread is running (Instantaneously appears)
Thread finished (Appears after 4s)
Main thread continues... (Instantaneously appears after second line)
```

2 Thực hành

2.1 Exercise 1:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void *foo(){
    int n;
    printf("Enter from keyboard");
    scanf("%d",&n);
    for(int i = 1; i < n + 1; i++)  printf("%d\n", i);
    pthread_exit(NULL);
}
```

```
int main(){
    pthread_t thread1;
    pthread_create(&thread1, NULL, &foo, NULL);
    pthread_join(thread1, NULL);
    printf("Main thread exiting");
    return 0;
}
```

Output trả về như sau:

```
Chiaki@mx:~/C
$ gcc -pthread thread.c -o thread
Chiaki@mx:~/C
$ ./thread
Enter from keyboard 5
1
2
3
4
5
Main thread exiting
```

2.2 Exercise 2:

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

void* compute_sum(void* arg) {
    int n = *((int*)arg);
    int* result = malloc(sizeof(int));
    *result = 0;
    for (int i = 1; i <= n; i++) {
        *result += i;
    }
    return (void*)result;
}

int main() {
    pthread_t thread;
    int n;
    printf("Enter N: ");
    scanf("%d", &n);
    pthread_create(&thread, NULL, compute_sum, &n);
    void* res;
    pthread_join(thread, &res);
    int sum = *((int*)res);
    free(res);
    printf("Sum from 1 to %d is: %d\n", n, sum);
    return 0;
}
```

```
Output:
Chiaki@mx:~/C
$ ./thread
Enter N: 30
Sum from 1 to 30 is: 465
```

2.3 Exercise 3:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *first(){
    printf("This is the first thread. TID1 is: %d\n", pthread_self());
    pthread_exit(NULL);
}
void *second(){
    printf("This is the second thread. TID2 is: %d\n", pthread_self());
    pthread_exit(NULL);
}
void *third(){
    printf("This is the third thread. TID3 is: %d\n", pthread_self());
    pthread_exit(NULL);
}
int main(){
    pthread_t thread1, thread2, thread3;
    pthread_create(&thread1, NULL, &first, NULL);
    pthread_create(&thread2, NULL, &second, NULL);
    pthread_create(&thread3, NULL, &third, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    printf("Main thread is running. TID is : %d\n", pthread_self());
    return 0;
}
Output:
Chiaki@mx:~/C
$ ./thread
This is the first thread. TID1 is: 76932800
This is the second thread. TID2 is: 68540096
This is the third thread. TID3 is: 60147392
Main thread is running. TID is : 76937024
```

2.4 Exercise 4:

```
// Program 1
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int sharedVar = 5;
void* threadFunc(void* arg){
```

```
    sharedVar++;
    pthread_exit(NULL);
}
int main(){
    pthread_t thread;
    pthread_create(&thread,NULL,&threadFunc,NULL);
    pthread_join(thread,NULL);
    printf("Main: sharedVar = %d\n", sharedVar);
    return 0;
}
Output:
Chiaki@mx:~/C
$ ./thread
Main: sharedVar = 6
// Program 2:
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int sharedVar = 5;
int main(){
    pid_t pid = fork();
    if(pid == 0){
        sharedVar++;
        printf("Child process: sharedVar = %d\n", sharedVar);
        exit(0);
    }
    else{
        wait(NULL);
        printf("Parent process: sharedVar = %d\n", sharedVar);
    }
    return 0;
}
Output:
Chiaki@mx:~/C
$ ./thread
Child process: sharedVar = 6
Parent process: sharedVar = 5
```

Lý do dẫn đến sự khác biệt giữa 2 output: do luồng có chia sẻ bộ nhớ nên `sharedVar` được chia sẻ giữa `main` và `thread`, nên output cho ra kết quả bằng `6`. Mặt khác, tiến trình không hề có sự chia sẻ bộ nhớ giữa tiến trình cha và con, nên chỉ có tiến trình con mới được cập nhật kết quả `sharedVar = 6`, còn tiến trình cha vẫn giữ nguyên `sharedVar = 5`.