pthread

dqmdang@hcmus.edu.vn

Nội dung

Nội dung bài học hôm nay tìm hiểu về pthread và các API cơ bản trong thư viện pthread trên Linux:

- Giới thiệu pthread
- Tạo ra và kết thúc một pthread
- Lỗi race condition
- Đồng bộ giữa các pthread bằng mutex
- Các lưu ý khi dùng pthread

pthread

Thread

- Thread (thread of execution) là khái niệm trừu tượng đại diện cho việc thực thi code của một process.
- Mỗi process luôn có ít nhất 1 thread. Process có nhiều task độc lập có thể có nhiều thread chạy “đồng thời” (multithread), mỗi thread phụ trách một task.
- Trong cùng process, mỗi thread sử dụng một stack riêng và chia sẻ môi trường thực thi của process (process address space, process id, file descriptor table, ...)
- Scheduler của kernel có nhiệm vụ sắp xếp và phân phối tất cả các thread trong hệ thống đến các CPU để thực thi

Schedule threads

Hình minh họa các thread trong hệ thống:

- Mỗi khối là một thread, có tất cả 8 thread cần thực thi
- Mỗi màu đại diện cho một process, có tất cả 5 process đang chạy
- Hệ thống có 2 CPU: CPU1 và CPU2 đang thực thi 2 thread ở đầu hàng đợi
- Khi một thread thực thi “xong”, nó sẽ được chuyển đến cuối hàng và CPU sẽ thực thi thread tiếp theo
- Do thời gian xoay vòng ngắn, các thread gần như chạy “liên tục” và “đồng thời”

Lợi ích của multithread

Sử dụng multithread mang lại những lợi ích:

- Dễ dàng xử lý các sự kiện bất đồng bộ (asynchronously) bằng các thread riêng
- Cải thiện performance/responsiveness của chương trình bằng cách thực thi nhiều thread đồng thời
- Chia sẻ dữ liệu nhanh chóng, dễ dàng giữa các thread so với các process
- Thread được tạo ra nhanh hơn và nhẹ hơn so với process

pthread

- POSIX threads (pthread) là một tiêu chuẩn được định nghĩa bởi IEEE, nó gồm một tập các API quy định việc quản lý thực thi của các thread trong một chương trình
- Việc chuẩn hóa giúp một chương trình pthread có thể thực thi trên nhiều platform khác nhau, miễn là các platform đó hỗ trợ chuẩn pthread
- Trên Linux, pthread được thực hiện bởi thư viện *Native POSIX Threads Library (NPTL).* Chương trình phải include header <*pthread.h*> để sử dụng pthread và thêm option *-lpthread* khi build bằng gcc

pthread API

Giới thiệu

- Khi một process bắt đầu thực thi, nó được tạo sẵn 1 thread gọi là **main thread**. Thread này bắt đầu thực thi từ hàm main()
- Các thread khác được tạo ra và bắt đầu thực thi từ một hàm gọi là hàm **start()** của thread
- Mỗi thread được định danh bởi **thread id (tid)** bên trong process

pthread_create

```
#include <pthread.h>

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

Returns 0 on success, or a positive error number on error

- Tạo ra một thread mới <thread> (trỏ tới tid) với thuộc tính <attr>, <thread> bắt đầu thực thi từ hàm <start> với tham số <arg>

pthread_create

- Giá trị trỏ đến bởi <attr> và <arg> phải nằm ở vùng nhớ hiện hữu trong suốt thời gian thực thi của <thread>
- Hàm <start> return con trỏ trỏ đến giá trị nằm ở vùng nhớ hiện hữu khi thread kết thúc thực thi
- <attr> có thể nhận giá trị NULL để tạo ra <thread> có thuộc tính mặc định
- <arg> có thể nhận giá trị NULL nếu hàm <start> của thread không cần nhận tham số

pthread_exit

```
include <pthread.h>

void pthread_exit(void *retval);
```

- Kết thúc thực thi thread, và trả về <retval>
- Giá trị trỏ đến bởi <retval> nên nằm ngoài stack của thread, bởi vì stack của thread sẽ bị thu hồi khi thread kết thúc thực thi

Thread termination

- main thread kết thúc thực thi (các thread khác vẫn chạy) khi:
  - main thread gọi hàm pthread_exit()
- Một thread kết thúc thực thi (các thread khác vẫn chạy) khi:
  - Hàm start() của thread return
  - Thread gọi hàm pthread_exit()
- Tất cả các thread kết thúc thực thi khi:
  - Bất kì thread nào gọi hàm exit()
  - main thread return (bên trong hàm main())

pthread_self / pthread_equal

```
include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

- Trả về tid của thread

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if *t1* and *t2* are equal, otherwise 0

- So sánh tid của 2 thread

pthread_join

```
include <pthread.h>

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

Returns 0 on success, or a positive error number on error

- Chờ đến khi <thread> kết thúc thực thi và nhận giá trị trả về của nó thông qua <retval>. Quá trình này thường được gọi là joining
- <retval> có thể bằng NULL nếu không cần nhận giá trị trả về của <thread>

Ví dụ

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

void * threadFunc(void *arg){
    char *s = (char *) arg;
    printf("%s", s);
    return (void *) strlen(s);
}

int main(int argc, char *argv[]){
    pthread_t t1;
    void *res;
    pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    pthread_join(t1, &res);
    printf("Thread returned %ld\n", (long) res);
    return 0;
}
```

pthread_detach

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

- Chuyển trạng thái của <thread> thành detach từ trạng thái joinable mặc định
- Thread ở trạng thái detach không cần (và không thể) join khi kết thúc thực thi
- Thread có thể tự mình detach
- Thread ở trạng thái detach vẫn kết thúc thực thi trong cùng điều kiện với các thread khác

race condition

- **Race condition** là trạng thái lỗi khi các thread cùng truy cập vào một vùng nhớ trong đó có ít nhất một thread ghi
- Để khắc phục race condition, đoạn code truy cập vào cùng nhớ dùng chung cần phải được thực thi **atomic** (không bị chia cắt, không bị gián đoạn bởi một đoạn code truy cập khác)
- Đoạn code cần được thực thi atomic được gọi là **critical section**

Ví dụ

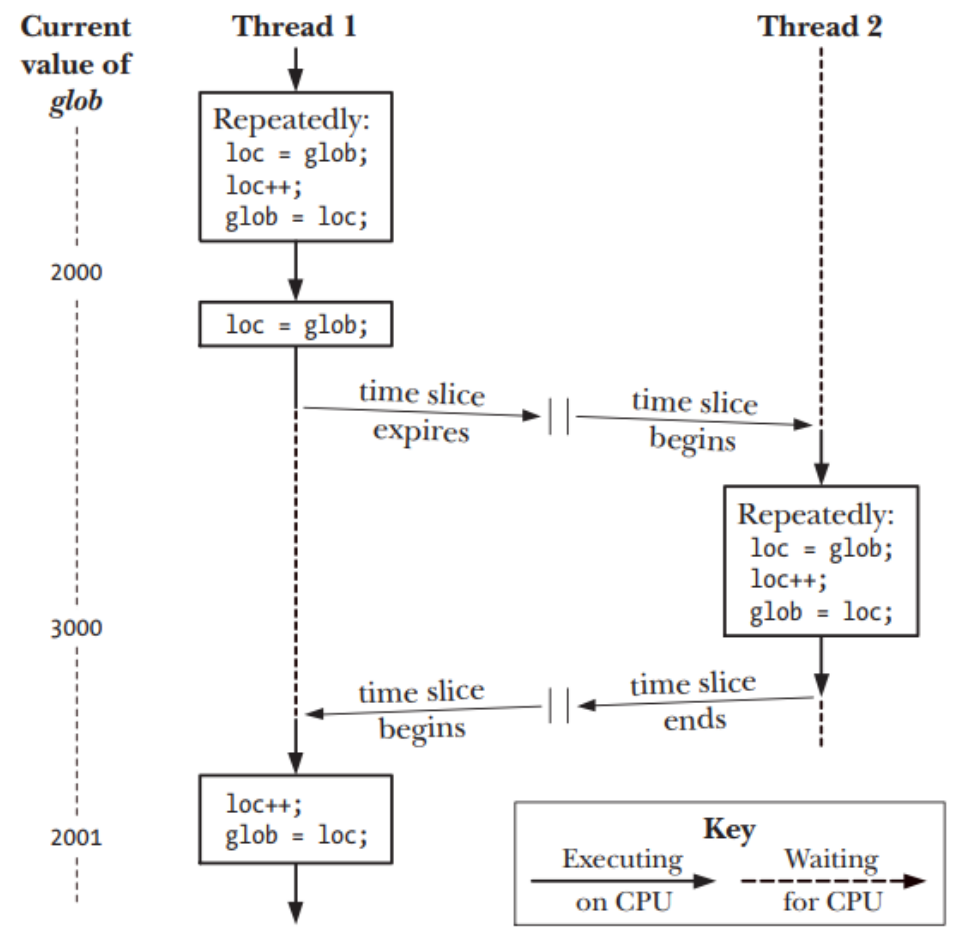
```c
#include <pthread.h>
#include <stdio.h>

static int glob = 0;
static void * threadFunc(void *arg) {
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    int loops = 10000000;
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("glob = %d\n", glob);
    return 0;
}
```

Race condition xảy ra như thế nào? Đâu là critical section?

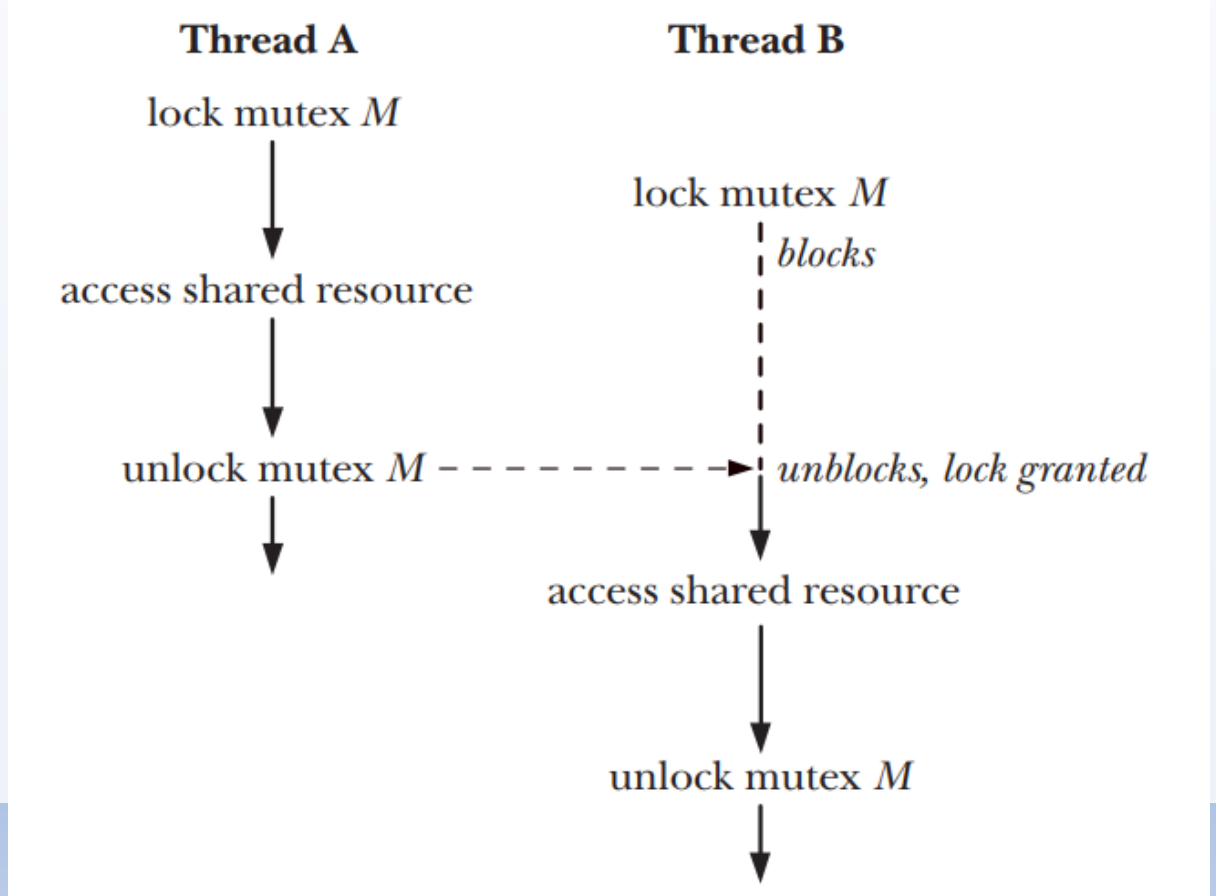
Ví dụ



- race condition xảy ra khi 2 thread truy cập đồng thời vào biến dùng chung *glob*
- Hậu quả là biến *glob* có giá trị sai và sai khác nhau với mỗi lần chạy chương trình
- critical section là đoạn code truy cập biến *glob* bên trong vòng lặp *for*
- Thay đoạn code này bằng *glob*++ có sửa được lỗi?

mutex

- **mutex** (mutual exclusion) là tính năng giúp cho critical section thực thi atomic.
- Nó hoạt động giống như một cái khóa, chỉ cho phép một critical section truy cập vùng nhớ dùng chung tại một thời điểm



mutex API

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

- Khởi tạo mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Both return 0 on success, or a positive error number on error

- Khóa và mở khóa mutex

Ví dụ

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

static int glob = 0;
static pthread_mutex_t mtx =
PTHREAD_MUTEX_INITIALIZER;

static void * threadFunc(void *arg) {
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mtx);
        loc = glob;
        loc++;
        glob = loc;
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    int loops = 10000000;
    pthread_create(&t1, NULL, threadFunc, &loops);
    pthread_create(&t2, NULL, threadFunc, &loops);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("glob = %d\n", glob);
    return 0;
}
```

mutex deadlock

Thread A	Thread B
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);*	2. *pthread_mutex_lock(mutex1);*
blocks	blocks

- Deadlock xảy ra khi hai thread đã khóa một mutex mà thread kia đang muốn khóa
- Deadlock xảy ra tương tự với trường hợp nhiều hơn 2 thread
- Luôn khóa các mutex theo thứ tự quy ước để tránh deadlock

pthread notes

API

- Các kiểu dữ liệu định nghĩa bởi pthread (pthread_t, pthread_mutex_t, ...) chỉ được sử dụng với pthread API. Để đảm bảo tính tương thích, chương trình không được khai thác định nghĩa của các kiểu dữ liệu này
- Khác với quy ước truyền thống, pthread API trả về giá trị bằng 0 nếu thành công và giá trị > 0 (tương đương errno) nếu có lỗi xảy ra

detach/join

- Trừ khi thread ở trạng thái detach, thread phải được join khi kết thúc thực thi
- Khác với process, các thread là ngang hàng với nhau. Không có mối liên hệ giữa thread gọi hàm pthread_create() và thread được tạo ra từ lời gọi hàm này
- Một thread bất kì có thể join với một thread khác miễn là nó biết tid của thread đó

memory

- Các thread trong cùng process chia sẻ process address space (text, global data, heap data, ...)
- Mỗi thread sở hữu một giá trị errno riêng
- Các biến cấp phát trên stack là riêng của mỗi thread, và chỉ tồn tại bên trong hàm nơi biến được khai báo
- Khi truyền tham số và nhận giá trị trả về từ thread, cần đảm bảo tính hiện hữu của các vùng nhớ

race condition

- Race condition thường xảy ra ở thời điểm không xác định và gây ra lỗi có hậu quả không xác định. Do đó rất khó để phát hiện và sửa lỗi race condition, nhất là trong các chương trình phức tạp
- Các thread truy cập dữ liệu dùng chung cần dùng mutex đồng bộ để tránh race condition
- Luôn khởi tạo mutex trước khi sử dụng
- Khi sử dụng nhiều mutex, luôn khóa các mutex theo thứ tự quy ước để tránh deadlock

Exercise

Bài tập

1. Viết chương trình **sleep_thread** có chức năng tương tự command sleep, nhưng có N thread

Gợi ý: Nên tổ chức các thread theo kĩ thuật thread pool

Ví dụ:

$./sleep_thread 2 (chạy command sleep_thread với cấu hình 2 thread)

>> (dấu nhắc do sleep_thread in ra, báo hiệu còn thread để sleep)

>>5 (nhập 5 vào và nhấn enter, thread1 sleep 5s)

>> (hiện dấu nhắc do vẫn còn thread để sleep)

>>3 (nhập 3 vào và nhấn enter, thread2 sleep 3s)

(không in ra dấu nhắc, do tất cả các thread đã sleep)

>> (3s trôi qua, hiện dấu nhắc do thread2 đã thức dậy và sẵn sàng sleep)

Bài tập

2. Viết chương trình **primecnt_thread** nhận tham số N (0<N<100) và M (0<M<2^{64}), trong đó N là số thread được tạo ra để đếm số lượng số nguyên tố (snt) có trong khoảng 1 tới M. M luôn chia hết cho N.

Ví dụ:

$./primecnt_thread 3 12

5

main thread tạo ra thêm 3 thread, thread1 đếm snt có trong các số 1,4,7,10; thread2 đếm snt có trong các số 2,5,8,11 và thread 3 đếm snt có trong các số 3,6,9,12

Kết quả in ra là 5 vì có 5 snt >=1 và <=12 là 2, 3, 5, 7, 11

Bài tập

3. Viết chương trình **primecnt_process** có chức năng tương tự như primecnt_thread, nhưng sử dụng process thay vì thread

Bài tập

4. Command *time* được dùng để đo thời gian thực thi (tx) của một program. Vẽ đồ thị biểu diễn tx theo N, với M là tham số, cho primecnt_thread và primecnt_process. Nhận xét và giải thích hai đồ thị đã vẽ.

Gợi ý:

- Với mỗi giá trị của M và N, so sánh tx của 2 đồ thị
- Với mỗi giá trị M, tx nhỏ nhất với N bằng bao nhiêu? Tại sao?

Lưu ý: Thời gian thực thi của một program tùy thuộc vào các process đang chạy trên hệ thống. Do đó cần đảm bảo:

- Các process đang chạy là như nhau với mỗi lần đo
- Chọn các giá trị M sao cho tx có giá trị đủ lớn