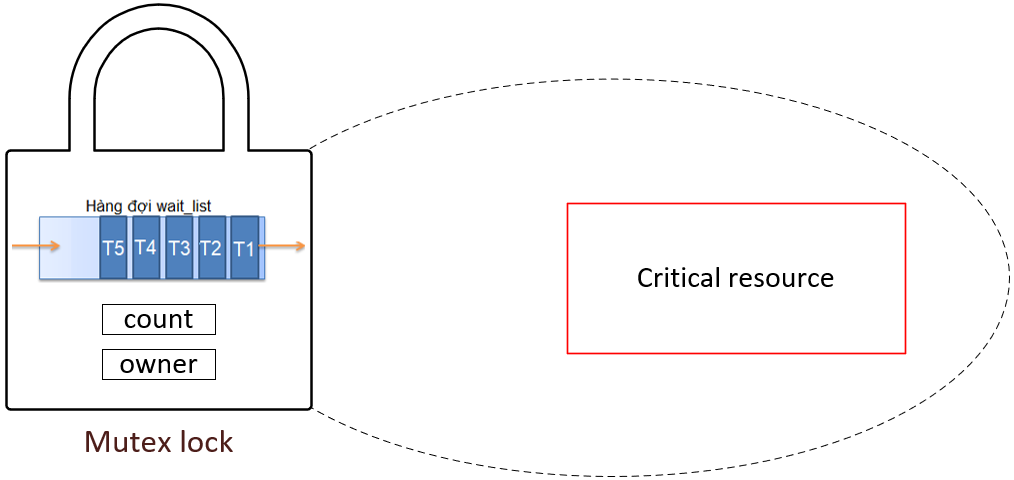
**HỆ ĐIỀU HÀNH**

**MUTEX CLOCK**

**Mutex lock là gì?**

Mutex lock là một cấu trúc dữ liệu, được Linux kernel xây dựng theo nguyên tắc mutual exclusion, dùng để ngăn chặn race condition xảy ra trên các cấu trúc dữ liệu khác. Nói nôm na, mutex lock đảm bảo rằng: tại một thời điểm bất kì, chỉ có tối đa một thread truy cập vào critical resource.

****

*Hình 1 Mutex lock giống như một ổ khóa dùng để bảo vệ critical resource*

**Mutex lock có cấu tạo như thế nào?**

Mutex lock gồm 3 thành phần chính: biến ***count***, biến ***owner*** và hàng đợi ***wait\_list***. Dựa vào đó, Linux kernel đã xây dựng cấu trúc [mutex](https://elixir.bootlin.com/linux/v4.4.137/source/include/linux/mutex.h#L50) để biểu diễn một mutex lock.

struct mutex {

/\*

\* Biến count lưu trạng thái của mutex lock, cũng như trạng thái của

\* critical resource.

\* - Nếu count = 1, thì trạng thái của mutex lock đang là UNLOCKED,

\* còn trạng thái của critical resource đang là AVAILABLE.

\* - Nếu count < 1, thì trạng thái của mutex lock đang là LOCKED,

\* còn trạng thái của critical resource đang là UNAVAILABLE.

\* count = 0: không có thread nào đang phải đợi để được

\* sử dụng critical resource.

\* count < 0: có thread đang phải đợi để được sử dụng

\* critical resource.

\*/

atomic\_t count;

/\*

\* Hàng đợi wait\_list có thể bị nhiều thread truy cập đồng thời.

\* wait\_lock là một spinlock bảo vệ wait\_list

\*/

spinlock\_t wait\_lock;

/\*

\* Hàng đợi wait\_list chứa danh sách các thread đang phải đợi để

\* chiếm được mutex lock, cũng chính là danh sách các thread

\* đang phải đợi để được sử dụng critical resource.

\*/

struct list\_head wait\_list;

/\*

\* Biến owner mô tả thread đang chiếm dụng mutex lock, cũng chính là

\* thread đang sử dụng critical resource.

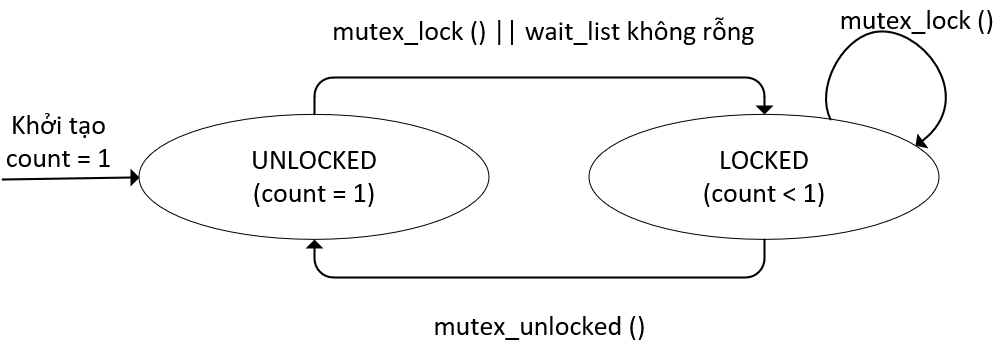
\*/

struct task\_struct \*owner;

...

};

**Mutex lock hoạt động ra sao?**

****

*Hình 2. Sơ đồ biểu diễn các trạng thái của một mutex lock*

Khi ***count*** đang bằng 1 (tức là mutex lock đang ở trạng thái **UNLOCKED**), nếu một thread gọi hàm [mutex\_lock](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L95), thì:

* biến ***count*** bị [giảm](https://elixir.bootlin.com/linux/v4.4.137/source/arch/x86/include/asm/mutex_32.h#L31) thành 0 (tức là mutex lock bị chuyển sang trạng thái **LOCKED**). Ta nói rằng thread đã khóa mutex lock lại.
* biến ***owner*** được [thiết lập bằng thread đó](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L103). Ta nói rằng thread đã chiếm dụng mutex lock.
* CPU bắt đầu thực thi critical section của thread (nói theo ngôn ngữ của CPU), hay thread đang sử dụng critical resource (nói theo ngôn ngữ của Linux kernel).

Khi ***count*** đang nhỏ hơn 1 (tức là đang ở trạng thái **LOCKED**), nếu một thread gọi hàm **mutex\_lock**, thì:

* biến ***count*** sẽ [giảm](https://elixir.bootlin.com/linux/v4.4.137/source/arch/x86/include/asm/mutex_32.h#L31) xuống 1 đơn vị.
* CPU tạm ngừng thực thi thread này rồi chuyển sang thực thi thread khác (nói theo ngôn ngữ của CPU). Hay nói theo ngôn ngữ của Linux kernel, thread được [thêm vào](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L543) hàng đợi ***wait\_list*** và sẽ [đi ngủ](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L578), sau đó Linux kernel sẽ [lập lịch cho thread khác](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L582). Do đó, ta nói rằng, mutex lock áp dụng cơ chế **sleep-waiting**, tức là mutex lock thuộc loại **sleep lock**, trái với spinlock thuộc loại busy lock.

Khi ***count*** đang nhỏ hơn 1 (tức là đang ở trạng thái **LOCKED**), nếu một thread A gọi hàm [mutex\_unlock](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L423), thì:

* biến ***owner*** [được thiết lập thành NULL](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L435). Ta nói rằng, thread A đã giải phóng mutex lock.
* biến ***count*** sẽ được [tăng](https://elixir.bootlin.com/linux/v4.4.137/source/arch/x86/include/asm/mutex_32.h#L77) thành 1 hoặc được [thiết lập](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L737) bằng 1 (tức là mutex lock chuyển sang trạng thái **UNLOCKED**). Ta nói rằng thread A đã mở khóa mutex lock.
* nếu hàng đợi ***wait\_list*** không rỗng và giả sử thread B nằm ở đầu hàng đợi, CPU sẽ chuyển sang thực thi thread B (nói theo ngôn ngữ của CPU). Hay nói theo ngôn ngữ của Linux kernel, Linux kernel sẽ [đánh thức](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L754) thread B dậy. Sau khi thức dậy, thread B sẽ chuyển mutex lock sang trạng thái **LOCKED** (thay đổi biến ***count***  [thành -1](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L560) nếu vẫn còn các thread khác đang đợi, hoặc [thành 0](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L590) nếu không còn thread nào đang đợi). Sau đó, thread B [chiếm lấy](https://elixir.bootlin.com/linux/v4.4.137/source/kernel/locking/mutex.c#L596) mutex lock rồi bắt đầu sử dụng critical resource.

**Mutex lock bảo vệ critical resource như thế nào?**

Trong khi lập trình device driver, ta đặt hàm **mutex\_lock** và **mutex\_unlock** lần lượt vào trước và sau critical section của mỗi thread. Việc làm này giúp bảo vệ critical resource. Để thấy được điều này, ta xét ví dụ sau. Giả sử, hệ thống có kernel thread A và B được thực thi riêng biệt trên 2 lõi CPU0 và CPU1. Cả 2 thread đều có nhu cầu sử dụng critical resource R, và tài nguyên R được bảo vệ bằng mutex lock M. Xét 2 trường hợp sau:

* Trường hợp 1: A muốn truy cập R trong khi B đang sử dụng R.
  + Trước khi thực thi các lệnh trong critical section của thread A, CPU0 sẽ thực thi hàm **mutex\_lock**và thấy rằng M đang ở trạng thái **LOCKED**. Khi đó, CPU0 sẽ dừng thực thi thread A rồi chuyển sang thực thi một thread C nào đó.
  + Sau khi thực thi xong critical section của thread B, CPU1 thực thi tiếp hàm **mutex\_unlock** để chuyển M sang trạng thái **UNLOCKED**. Lúc này, thread A sẽ chiếm lấy M và CPU0 tiếp tục thực thi thread A.
* Trường hợp 2: cả A và B đồng thời muốn truy cập R.
  + Khi đó, cả 2 thread đồng thời thực thi hàm **mutex\_lock**. Tuy nhiên, do hàm **mutex\_lock** dùng thao tác atomic để thay đổi biến ***count***, nên chỉ có một trong hai thread chiếm được M.
  + Thread nào chiếm được M trước thì sẽ sử dụng R trước. Thread nào không chiếm được M thì sẽ đi ngủ cho đến khi thread đầu tiên sử dụng xong R.

Như vậy, tại bất cứ thời điểm nào, tối đa chỉ có một thread được phép chiếm dụng mutex lock, đồng nghĩa với việc, tối đa chỉ có một thread được phép sử dụng critical resource. Do đó, race condition sẽ không xảy ra và critical resource được bảo vệ.

**Sử dụng mutex clock:**

Để sử dụng mutex, trước hết chúng ta phải khai báo và khởi tạo mutex. Trong Posix thread, biến mutex là kiểu dữ liệu có dạng pthread\_mutex\_t và có thể được khởi tạo tĩnh sử dụng macro PTHREAD\_MUTEX\_INITIALIZER hoặc khởi tạo động lúc runtime.

## ****Khởi tạo tĩnh (statically allocation)****

Pthread\_mutex\_t mutex =  PTHREAD\_MUTEX\_INITIALIZER;

## ****Khởi tạo động (dynamically initializing)****

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);

      /\*Trả về 0 nếu thành công, hoặc 1 số dương mã lỗi\*/

Trong cách khởi tạo tĩnh, macro PTHREAD\_MUTEX\_INITIALIZER dùng để khởi tạo một mutex với các thuộc tính (thread attribute) mặc định. Trong khi hàm pthread\_mutex\_init() trong cách khởi tạo động cho phép khởi tạo và thiết lập thuộc tính cho mutex. Nếu không cần quan tâm đến thuộc tính của thread, ta có thể truyền NULL vào đối số pthread\_mutexattr\_t \*attr. Khi khởi tạo động mutex bằng hàm pthread\_mutex\_init(), ta cần phải hủy mutex đó nếu không cần sử dụng nữa bằng hàm pthread\_mutex\_destroy() có prototype như sau (khởi tạo tĩnh bằng macro PTHREAD\_MUTEX\_INITIALIZER không cần destroy mutex):

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

/\*Return 0 nếu thành công, hoặc một số dương mã lỗi nếu không thành công\*/

## ****Lock/unlock mutex****

Sau khi khởi tạo, mutex được khóa và mở khóa bởi 2 hàm sau đây:

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

/\*Trả về 0 nếu thành công, hoặc 1 số dương mã lỗi khi xảy ra lỗi\*/

Để khóa 1 mutex, ta truyền địa chỉ của mutex đó vào hàm pthread\_mutex\_lock(). Nếu một mutex đang ở trạng thái unlock, hàm này sẽ khóa mutex đó và return. Nếu mutex đó đã bị khóa bởi thread khác, hàm này sẽ bị block cho đến khi mutex được mở. Nếu một thread khóa một mutex mà chính nó đang giữ khóa thì sẽ xảy ra deadlock (thread rơi vào trạng thái chờ vô hạn).

Ngoài ra, chuẩn Posix còn cung cấp hai hàm lock mutex sau đây:

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

và

int pthread\_mutex\_timedlock(pthread\_mutex\_t \*restrict mutex,const struct timespec \*restrict abs\_timeout);

Hàm pthread\_mutex\_trylock() hoạt động khác pthread\_mutex\_lock() ở chỗ: nếu mutex đang bị khóa, nó sẽ không block thread mà sẽ return ngay lập tức với mã lỗi là EBUSY. Còn hàm pthread\_mutex\_timedlock() được thêm vào đối số abs\_timeout để thiết lập thời gian tối đa thread có thể chờ; nếu sau khoảng thời gian "abs\_timeout " mà thread đó chưa sở hữu được mutex, nó sẽ return và trả về mã lỗi ETIMEDOUT.

**Ví dụ:**

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

#define MAX\_THREAD 2

pthread\_t tid[MAX\_THREAD];

/\*Counter la bien toan cuc duoc 2 thread su dung\*/

int counter;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;   //khai bao mutex

void \*threadFunc(void \*argv)

{

pthread\_mutex\_lock(&mutex);

counter += 1;

printf("Thread %d has started\n", counter);

sleep(1);

printf("Thread %d has finished\n", counter);

pthread\_mutex\_unlock(&mutex);

pthread\_exit(NULL);

}

int main(void)

{

int i = 0;

int ret = 0;

for (i = 0; i < MAX\_THREAD; i++)

{

ret = pthread\_create(&(tid[i]), NULL, threadFunc, NULL);

if (ret != 0)

{

printf("Thread [%d] created error\n", i);

}

}

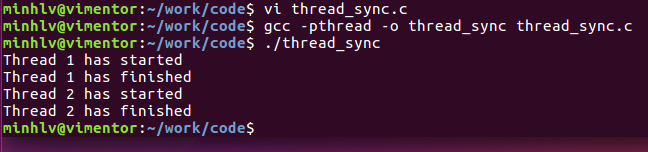
pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

return 0;

}

**Kết quả:**



**Bài tập**

Mỗi hãng máy bay sẽ có một số lượng vé nhất định, các đại lý sẽ truy cập đến hệ thống của hãng để đặt mua vé máy bay.

Hãy xây dựng một chương trình quản lý vé máy bay của hãng. Khi thực hiện chương trình này, sẽ cho người dùng nhập vào số lượng vé máy bay trong ngày của hãng. Dữ liệu này sẽ được nhập vào file: sove.txt.

Hãy xây dựng chương trình cho 2 đại lý: mỗi đại lý khi chạy sẽ đọc từ file: daily1.txt và daily2.txt. Khi chạy, mỗi đại lý sẽ mua vé từ hãng, mỗi lần mua vé thì file sove.txt của hãng sẽ bị giảm theo số lượng vé của mỗi đại lý đặt. Nếu đặt mua thành công, sẽ hiện thông báo đặt mua thành công với số vé, nếu hết vé phải báo là đã hết vé, không thể mua tiếp. Mỗi lần mua xong, bắt buộc mỗi đại lý phải tạm nghỉ 5s.

Nội dung tập tin daily1.txt

2

4

5

10

Các số trên là số vé mà khách đến đại lý để mua ở các lần khác nhau.

Nội dung tập tin daily2.txt

5

3

4

Hãy sử dụng công cụ share memory, mutex clock để giải quyết bài tập trên.