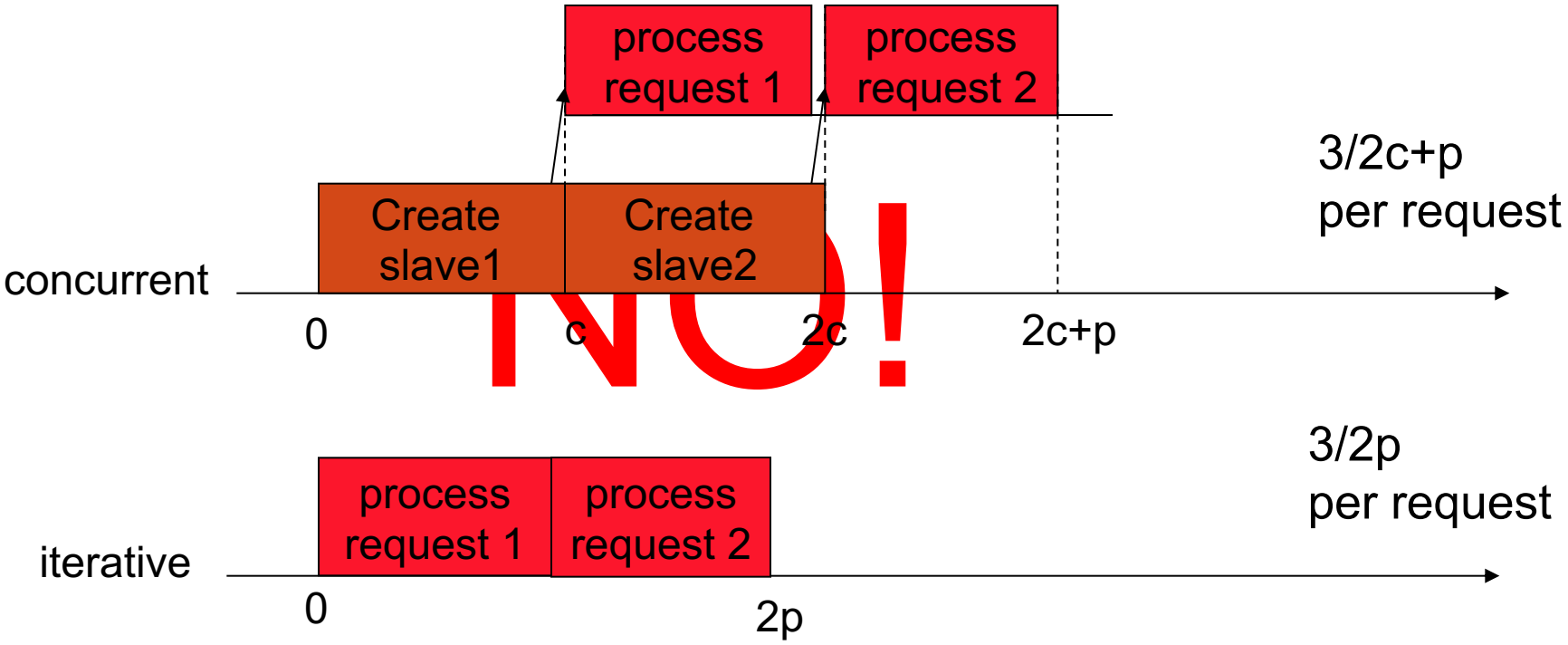
Máy chủ xử lý đồng thời, đa luồng

Giảng viên: Nguyễn Hoài Sơn
Bộ môn Mạng và Truyền thông máy tính
Khoa Công nghệ thông tin

Nội dung bài học

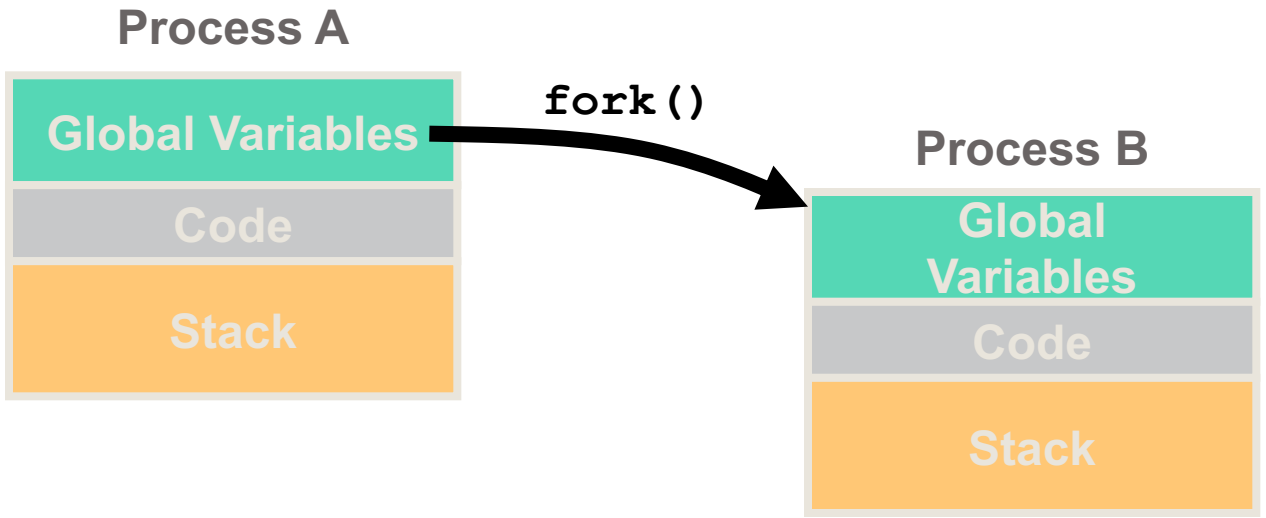
- Xử lý đa luồng là gì?
- Ví dụ về xử lý đa luồng
- Các vấn đề liên quan đến đồng bộ hóa

Xử lý đồng thời đa tiến trình có luôn tốt hơn xử lý tuần tự ?



Vấn đề của hàm fork()

- Chi phí cao
  - Việc tạo tiến trình con giống hệt tiến trình mẹ làm tốn tài nguyên bộ nhớ và thời gian
- Khó chia xẻ thông tin giữa tiến trình mẹ và tiến trình con

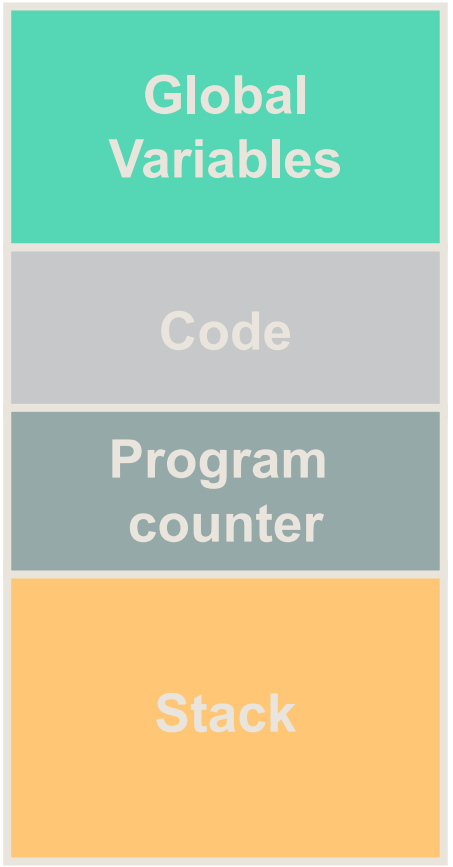


Giải pháp

- Xử lý đa luồng

Luồng là gì ?

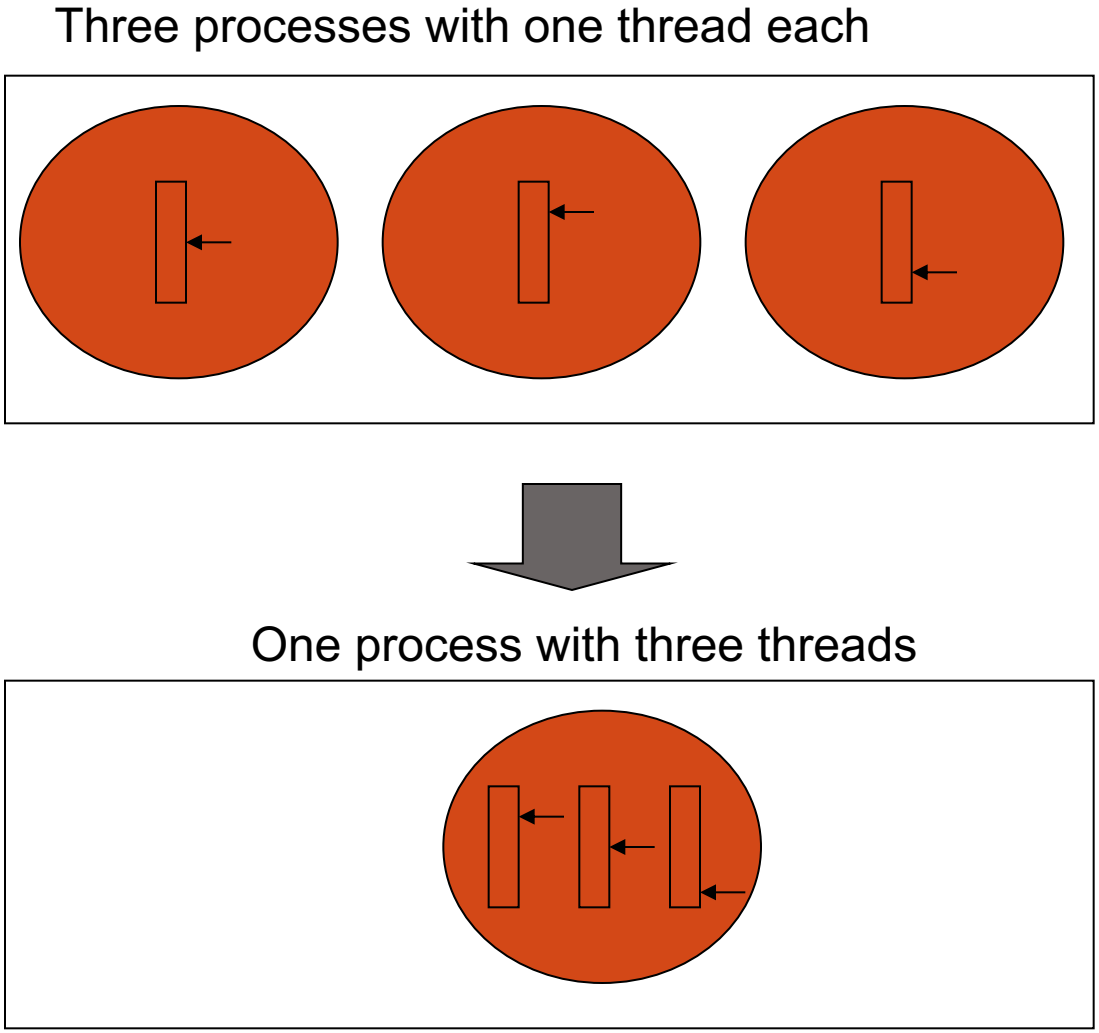
- Thread (luồng) là một dòng điều khiển trong một tiến trình
  - Tiến trình nhẹ (lightweight process)
  - có riêng
    - Định danh luồng (thread ID)
    - Bộ đếm chương trình (PC)
    - Tập thanh ghi (register set)
    - Ngăn xếp (stack): chứa các biến cục bộ
    - Độ ưu tiên
  - Có chung
    - Phần mã chương trình
    - Phần dữ liệu toàn cục
    - Tài nguyên hệ điều hành



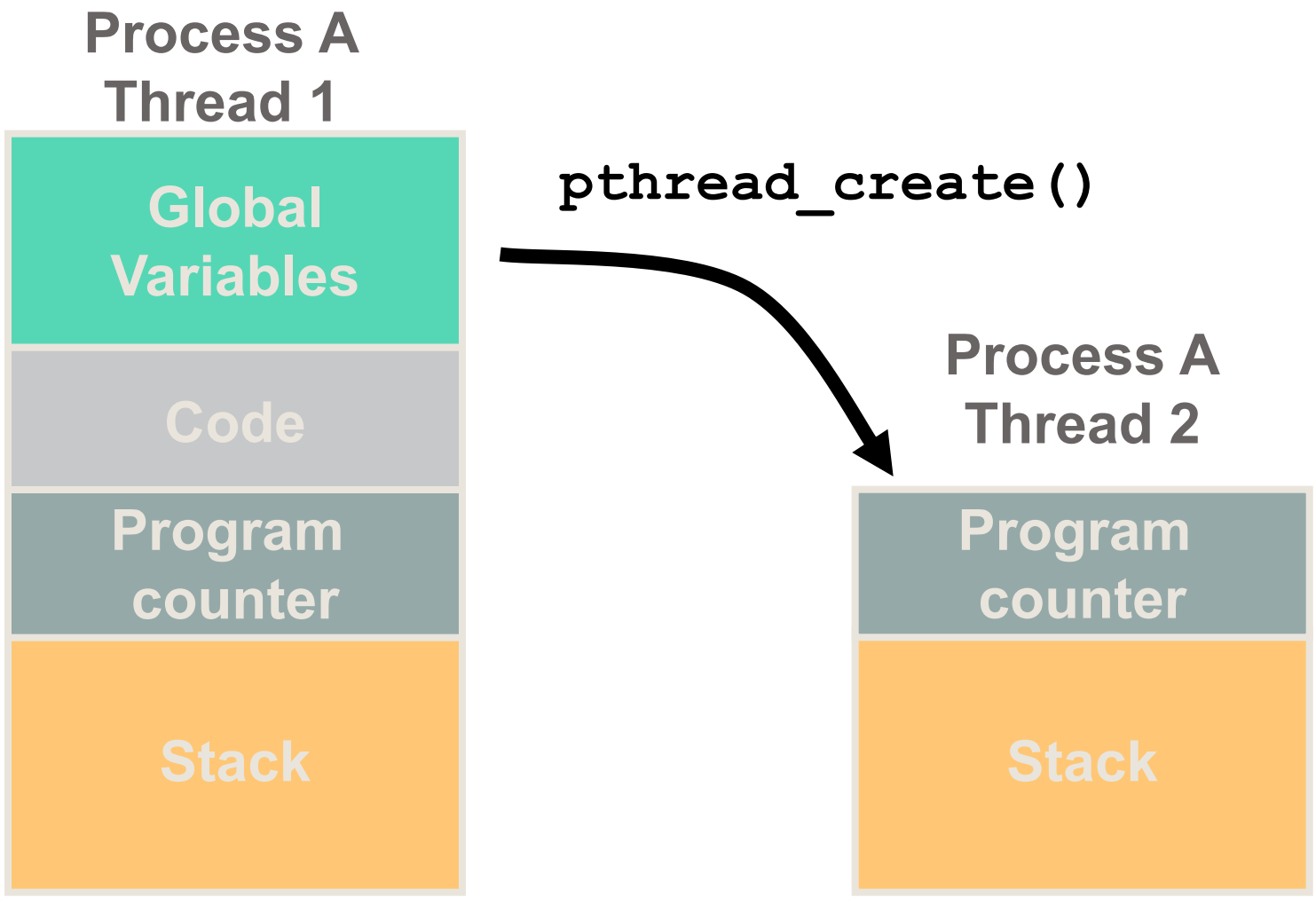
Tại sao lại là xử lý đa luồng?

- Một tiến trình với nhiều luồng có thể thực hiện nhiều công việc khác nhau tại cùng một thời điểm.
- Ưu điểm của xử lý đa luồng với xử lý đa tiến trình
  - Tạo luồng nhanh hơn
    - từ 10–100 lần so với tạo tiến trình
  - Tiêu tốn ít tài nguyên bộ nhớ
  - Chia xẻ thông tin giữa các luồng sẽ dễ dàng hơn

Xử lý đa tiến trình và xử lý đa luồng



Khởi tạo luồng mới



pthread_create(): Khởi tạo luồng

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                                    void *(*func) (void *), void *arg);
                    Returns: 0 if OK, positive Exxx value on error
```

- tid: con trỏ tới biến định danh luồng kiểu pthread_t
  - pthread_t: thường là unsigned int
- attr: con trỏ tới cấu trúc pthread_attr_t
  - pthread_attr_t: xác định các thuộc tính của luồng như độ ưu tiên, kích thước ban đầu của ngăn xếp, …
  - NULL nếu sử dụng mặc định của hệ thống
- func: Hàm gọi khi luồng bắt đầu
- arg: một con trỏ tới một cấu trúc các tham số của hàm *func*

Tuổi sống của một luồng

- Khi một luồng được tạo ra, nó sẽ chạy hàm func() được thiết lập trong lệnh gọi pthread_create().
- Khi hàm func() trả về giá trị, luồng sẽ kết thúc thực thi
- Một luồng cũng có thể kết thúc thực thi bằng hàm pthread_exit().
- Nếu luồng chính kết thúc thực thi hoặc một luồng nào đó trong tiến trình gọi lệnh exit() thì tất cả các luồng khác đều kết thúc thực thi

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

Does not return to caller

Luồng con phụ thuộc(joinable)/Luồng con độc lập (detached)

- Luồng con phụ thuộc
  - Sau khi xử lý của luồng kết thúc, trạng thái và ID của luồng con vẫn được giữ lại trong bộ nhớ cho đến khi hàm pthread_join được gọi
  - Trạng thái của một luồng khi được tạo ra sẽ được mặc định là phụ thuộc (joinable)
- Luồng con độc lập
  - Trạng thái và ID của luồng con sẽ được xóa ra khỏi bộ nhớ sau khi xử lý của luồng kết thúc

pthread_join(): đợi một luồng phụ thuộc dừng thực thi

```
#include <pthread.h>

int pthread_join (pthread_t tid, void ** status);
                                    Returns: 0 if OK, positive Exxx value on error
```

- *tid:* thread ID
- *status:* con trỏ tới giá trị trả về từ luồng
- Tương đương với hàm waitpid trong xử lý đa tiến trình

pthread_detach(): chuyển một luồng sang trạng thái "độc lập" (deteched)

```
#include <pthread.h>

int pthread_detach (pthread_t tid);
                                    Returns: 0 if OK, positive Exxx value on error
```

- *tid*: thread ID
- Khi một luồng độc lập kết thúc thực thi, tất cả tài nguyên của nó sẽ được giải phóng

pthread_self(): Lấy định danh của chính luồng đó

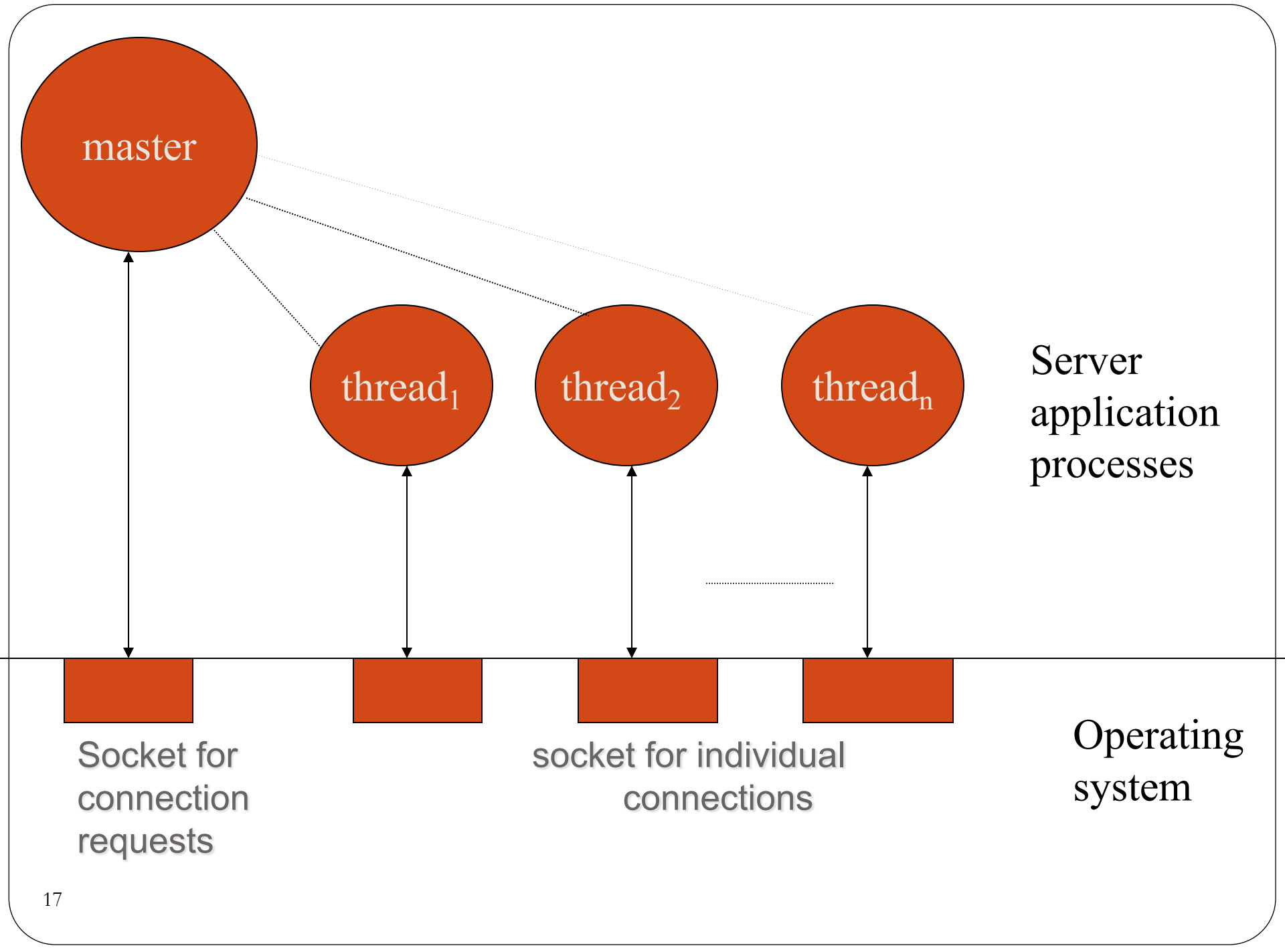
```
#include <pthread.h>

pthread_t pthread_self (void);
                                Returns: thread ID of calling thread
```

- Có thể sử dụng định danh luồng trong hàm pthread_detach()

Máy chủ xử lý đồng thời đa luồng hướng kết nối

- Tạo mỗi luồng cho một kết nối mới đến máy khách


master
thread1
thread2
threadn
Server application processes
Socket for connection requests
socket for individual connections
Operating system

Các bước thực thi máy chủ xử lý đồng thới, hướng kết nối, đa luồng

- ***Bước 1 luồng chính:*** Khởi tạo socket, gán thông tin cho socket và chuyển socket sang trạng thái thụ động, chờ kết nối
- ***Bước 2 luồng chính:*** Lặp lại lệnh gọi `accept()` để chấp nhận một kết nối từ máy khách và khởi tạo một luồng con để xử lý yêu cầu của máy khách

Các bước thực thi máy chủ xử lý đồng thới, hướng kết nối, đa luồng (2)

- ***Bước 1 luồng con:*** Thực thi hàm nhận và xử lý yêu cầu của máy khách thông qua socket kết nối và gửi trả lại kết quả trả lời
- ***Bước 2 luồng con:*** Đóng socket kết nối và kết thúc thực thi

doit() function

```
static void *doit(void *arg)
{
    pthread_detach(pthread_self());
    str_echo(* ( ( int *) arg));   /* same function as before */
    close(* ( ( int *) arg));      /* done with connected socket */
    return (NULL);
}
```

- Luồng con không cần đóng socket lắng nghe
- Luồng con phải đóng socket kết nối trước khi kết thúc thực thi

Ví dụ về máy chủ TCP echo xử lý đồng thời đa luồng

```
int main(int argc, char **argv){
       int     listenfd, connfd;
       pthread_t tid;
       socklen_t addrlen, len;
       struct sockaddr cliaddr;
      int *iptr;

       listenfd = passiveTCP(argv[1], QLEN);
      len = sizeof(cliaddr);

       for (; ; ) {
          connfd = accept(listenfd, cliaddr, &len);
           pthread_create(&tid, NULL, &doit, (void *) &connfd);
      }
 }
```

- Luồng chính không cần đóng socket kết nối

Vấn đề của xử lý đa luồng

- Thread-safe
  - Nếu một hàm sử dụng biến toàn cục, có thể không an toàn khi sử dụng hàm này với đa luồng
  - Một hàm là thread-safe nếu nó hoạt động đúng khi được sử dụng tại nhiều luồng khác nhau
- Đồng bộ hóa bộ nhớ giữa các luồng
  - Các luồng tranh chấp việc truy cập vào cùng một tài nguyên

Thread Safe library functions

- POSIX.1 requires that all the functions defined by POSIX.1 and by the ANSI C standard be thread-safe
- POSIX says nothing about thread safety with regard to the networking API functions

Need not be thread-safe	Must be thread safe
ctime	ctime_r
rand	rand_r
gethostXXX	
getnetXXX	
getprotoXXX	
getservXXX	
inet_ntoa	
…	…

Cải tiến với hàm thread-safe

```
int main(int argc, char **argv){
       int     listenfd;
       pthread_t tid;
       socklen_t addrlen, len;
       struct sockaddr cliaddr;
      int *iptr;

       listenfd = passiveTCP(argv[1], QLEN);
      len = sizeof(cliaddr);

       for (; ; ) {
         iptr = malloc(sizeof(int));
          *iptr = accept(listenfd, cliaddr, &len);
          pthread_create(&tid, NULL, &doit, (void *) iptr);
      }
 }
```

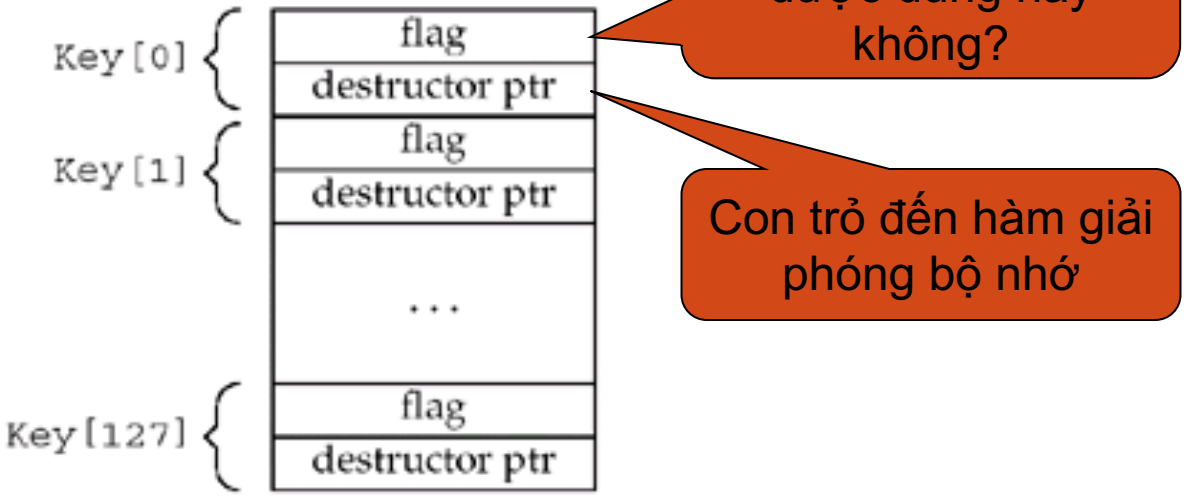
```
static void *doit(void *arg)
{
   int connfd;

   connfd = * ( ( int *) arg);
   free(arg);
    pthread_detach(pthread_self());
    str_echo(connfd);   /* same function as before */
    close(connfd);        /* done with connected socket */
    return (NULL);
}
```

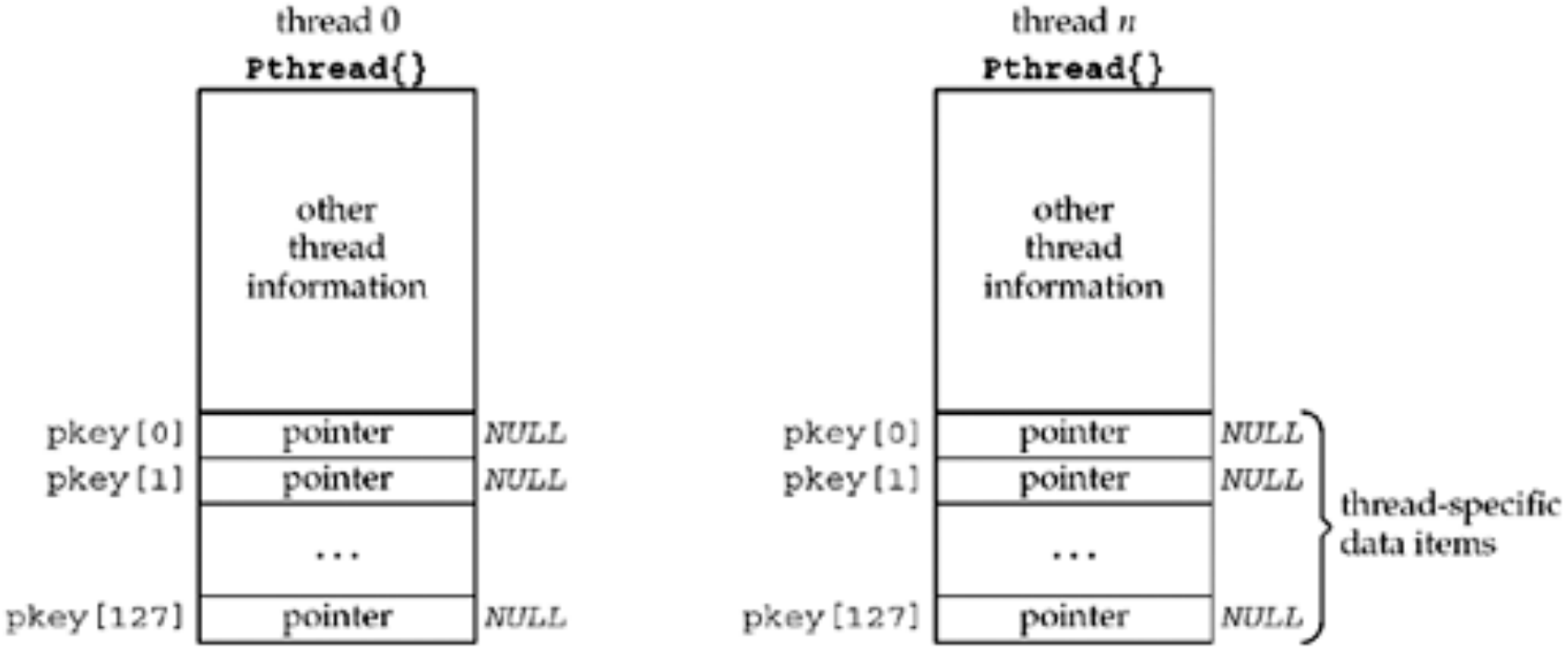
- Cẩn thận với các biến chung

Thread-Specific Data

- Dữ liệu riêng của luồng
  - Quản lý vùng bộ nhớ riêng cho mỗi luồng
  - Được sử dụng để đảm bảo các hàm là thread-safe
- Mỗi tiến trình duy trì một số lượng nhất định các thông tin về dữ liệu riêng của luồng
  - Theo POSIX, >= 128
  - Cấu trúc *Key*



Thông tin về dữ liệu riêng tại mỗi luồng



Sử dụng dữ liệu riêng của luồng như thế nào?

- *int pthread_once(pthread_once_t *onceptr, void (*init) (void));*
  - Được gọi khi hàm sử dụng dữ liệu riêng được gọi
  - Thực hiện hàm init 1 lần duy nhất đối với mỗi tiến trình
- *int pthread_key_create(pthread_key_t *keyptr, void (*destructor) (void *value));*
  - Lấy chỉ số (key/index) của dữ liệu riêng hiện đang trống từ cấu trúc *key* và gán hàm giải phóng bộ nhớ cho dữ liệu riêng này
  - Thường được gọi trong hàm khởi tạo *pthread_once()*
- *int pthread_setspecific(pthread_key_t key, const void *value);*
  - Gán con trỏ vào dữ liệu riêng
  - Chỉ thực hiện một lần
- void *pthread_getspecific(pthread_key_t *key*);
  - Lấy dữ liệu riêng

Ví dụ về hàm readline

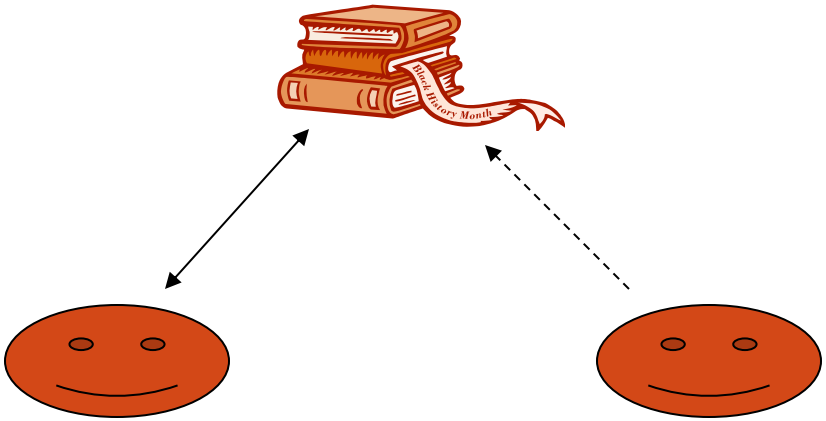
- *ssize_t readline(int fd, void *vptr, size_t maxlen)*
  - Đọc tối đa *maxlen* ký tự từ mô tả file *fd* cho đến khi gặp ký tự xuống dòng hoặc đến cuối file và lưu các ký tự vào vùng bộ nhớ trỏ bởi *vptr*
- Hàm không phải thread-safe
  - *test/readline1.c*
    - Đọc từng ký tự từ file và kiểm tra có phải là ký tự xuống dòng không
    - Rất chậm
  - *lib/readline.c*
    - Đọc chuỗi ký tự từ file và lần lượt kiểm tra từng ký tự xem có là ký tự xuống dòng không
    - Sử dụng biến static → không phải là thread-safe
- Hàm thread-safe
  - threads/realine.c

Luồng xử lý của hàm readline

- Ban đầu, 1 tiền trình với nhiều luồng được khởi tạo
- Khi 1 luồng gọi hàm readline
  - Gọi hàm pthread_key_create() để tìm Key chưa sử dụng
    - Key này được tạo ra duy nhất với tất cả các luồng vì hàm pthread_key_create() được gọi duy nhất 1 lần trong hàm pthread_once()
  - Gọi hàm pthread_getspecific() để lấy giá trị pkey[Key] ứng với số thứ tự Key ở trên
    - Nếu pkey[Key] = null, tạo vùng bộ nhớ cho dữ liệu riêng và gọi hàm pthread_setspecific() để gán con trỏ cho pkey[Key]
  - Sử dụng dữ liệu riêng của luồng để thực hiện việc đọc dữ liệu

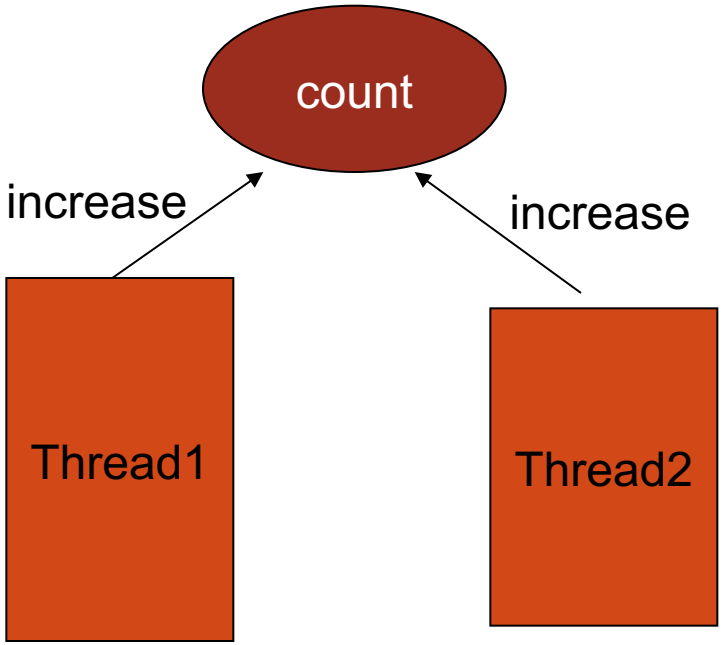
Loại trừ lẫn nhau

- Các luồng chia xẻ bộ nhớ, xử lý file (file handles), sockets, và các tài nguyên khác
- Nếu hai luồng muốn sử dụng cùng một tài nguyên tại một thời điểm nào đó, một trong 2 luồng phải đợi luồng kia kết thúc việc sử dụng tài nguyên



Ví dụ về sự cần thiết của loại trừ lẫn nhau

- threads/example01.c



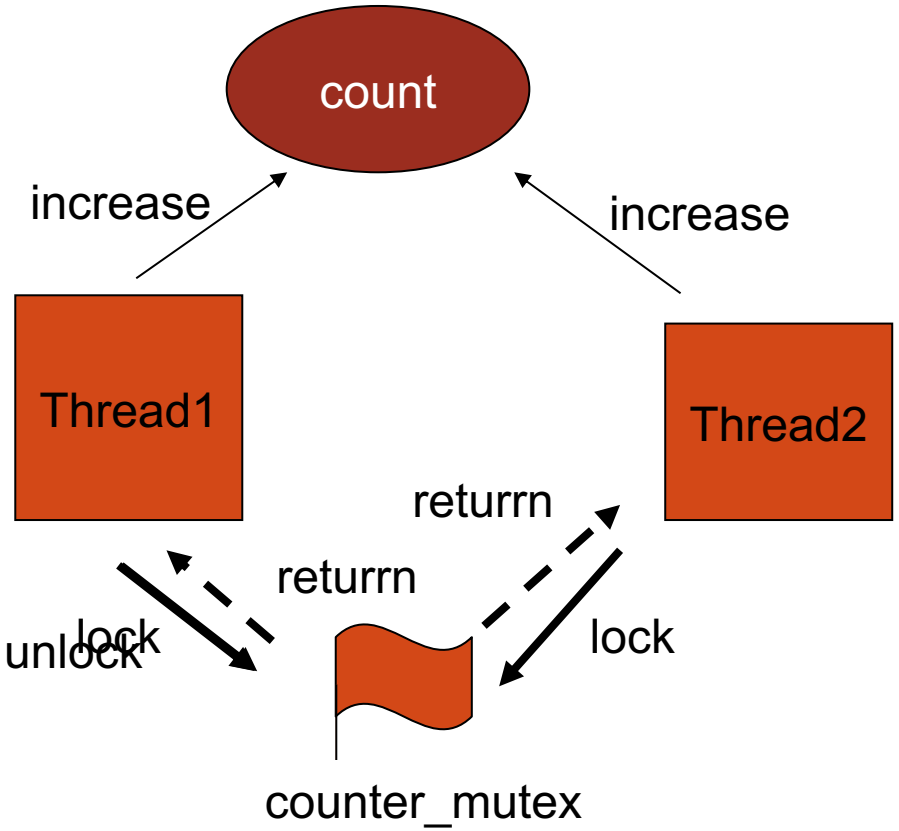
Loại trừ lẫn nhau (2)

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mptr);
int pthread_mutex_unlock(pthread_mutex_t * mptr);
                                Both return: 0 if OK, positive Exxx value on error
```

- mptr: con trỏ tới biến mutex ("mutual exclusion") kiểu pthread_mutex_t
  - Biến mutex được khởi tạo với giá trị PTHREAD_MUTEX_INITIALIZER nếu được gán tĩnh
- Thiết lập truy cập loại trừ lẫn nhau đến một biến chung
  - Chỉ cho phép truy cập vào biến chung khi không bị khóa

Ví dụ về loại trừ lẫn nhau

- threads/example02.c



Đặt vấn đề: Web client với nhiều kết nối đồng thời

- Luồng chính tạo nhiều luồng con để download nhiều đối tượng Web cùng một lúc
  - threads/web01.c
- Luồng chính đợi các luồng con kết thúc download
  - Tạo biến đếm số lượng luồng con đã thực hiện xong *ndone*
  - Khi luồng con kết thúc, tăng biến đếm *ndone* lên 1
  - Luồng chính chờ đến khi biến đếm *ndone* khác 0 thì xử lý dữ liệu đọc được

Làm thế nào để luồng A theo dõi sự thay đổi tại luồng B?

- Giải pháp
  - Tạo một biến toàn cục X
  - Luồng B update giá trị của biến X
  - Luồng A đọc giá trị của biến X
- Vấn đề
  - Luồng A phải tạo một vòng lặp để theo dõi biến X

  => Lãng phí CPU
- threads/web02.c

Biến điều kiện Condition Variables

- Cho phép một luồng chờ một tín hiệu gửi từ một luồng khác thông qua một biến điều kiện
  - *int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);*
    - Từ bỏ *mutex* và chờ tín hiệu gửi đến
- Cho phép một luồng gửi tín hiệu qua một biến trạng thái
  - *int pthread_cond_signal(pthread_cond_t *cptr)*

Với Web client với nhiều kết nối đồng thời

- Luồng chính
  - Khai báo biến trạng thái, biến đếm *ndone*, biến loại trừ
    - pthread_cond_t ndone_cond = PTHREAD_COND_INITIALIZER;
  - Tạo các luồng con để download file
  - Khi biến đếm bằng 0, luồng chính chờ tín hiệu từ luồng con gửi qua biến trạng thái
- Luồng con
  - Sau khi kết thúc download, tăng biến đếm *ndone* và gửi tín hiệu qua biến trạng thái
- Luồng chính
  - Xử lý dữ liệu download từ luồng con

Ví dụ

- threads/web03.c