Multithreading

Thư viện pthread (POSIX Threads) cung cấp một tập hợp các hàm và kiểu dữ liệu để lập trình đa luồng trong ngôn ngữ C trên các hệ điều hành hỗ trợ POSIX. Dưới đây là một số hàm quan trọng trong thư viện pthread:

1. Quản lý threads:
   * pthread\_create: Tạo một thread mới.
   * pthread\_join: Đợi cho một thread kết thúc.
   * pthread\_detach: Đặt một thread trong chế độ tự giải phóng tài nguyên sau khi kết thúc.
   * pthread\_exit: Kết thúc thread hiện tại và trả về giá trị.
2. Đồng bộ hóa:
   * pthread\_mutex\_init: Khởi tạo một mutex.
   * pthread\_mutex\_destroy: Hủy một mutex.
   * pthread\_mutex\_lock: Khóa một mutex.
   * pthread\_mutex\_unlock: Mở khóa một mutex.
   * pthread\_cond\_init: Khởi tạo một biến điều kiện.
   * pthread\_cond\_destroy: Hủy một biến điều kiện.
   * pthread\_cond\_wait: Chờ cho đến khi nhận được tín hiệu từ một biến điều kiện.
   * pthread\_cond\_signal: Gửi một tín hiệu đến một biến điều kiện để đánh thức một thread đang chờ.
   * pthread\_cond\_broadcast: Gửi một tín hiệu đến tất cả các thread đang chờ trên một biến điều kiện.
3. Thu thập thông tin:
   * pthread\_self: Trả về ID của thread gọi hàm.
   * pthread\_equal: So sánh hai ID thread.
   * pthread\_cancel: Hủy bỏ một thread.
4. Điều khiển thuật toán lập lịch:
   * pthread\_yield: Yêu cầu chuyển quyền điều khiển đến một thread khác.

<https://www.youtube.com/watch?v=ldJ8WGZVXZk>

**Demo how to use thread:**

#include <stdio.h>

#include <pthread.h>

void \*computation(){

printf("Computation");

return NULL;

}

int main()

{

//tao bien luu tru thong tin ve thread

pthread\_t thread1;

//tao thread moi

pthread\_create(&thread1,NULL,computation,NULL);

//NULL thu 2 thuong la NULL, truyen thuoc tinh cho luong

//computation la ham duoc truyen vao thread

//NULL cuoi cung pass argument vao computation o dang void pointer:

/\*

void \*computation(void\* add){

long \*add\_num = (long \*)(add);

printf("Add: %ld\n", \*add\_num);

}

long value1 = 1;

pthread\_create(&thread1,NULL,computation,(void\*) &value1);\*/

pthread\_join(thread1,NULL);//thread1 la luong chung ta dang doi

//NULL la return cua thread1 , neu can gia tri return tu thread1 thi:

//void\* result;

//pthread\_join(thread1, &result);

return 0;

}

**Argument Passing**

#include <stdio.h>

#include <pthread.h>

void \*computation(void\* add){

//long \*add\_num = (long \*)(add);

printf("Add: %ld\n", \*((long \*)add));

return NULL;

}

int main()

{

pthread\_t thread1;

long value1 = 1;

pthread\_create(&thread1,NULL,computation,(void\*) &value1);

pthread\_join(thread1,NULL);

return 0;

}

**Add 2nd thread**

#include <stdio.h>

#include <pthread.h>

void \*computation(void\* add){

//long \*add\_num = (long \*)(add);

printf("Add: %ld\n", \*((long \*)add));

return NULL;

}

int main()

{

pthread\_t thread1;

pthread\_t thread2;

long value1 = 1;

long value2 = 2;

pthread\_create(&thread1,NULL,computation,(void\*) &value1);

pthread\_create(&thread2,NULL,computation,(void\*) &value2);

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

return 0;

}

**Note!**:

* 2 thread cùng join -> nhanh gấp đôi

pthread\_create(&thread1,NULL,computation,(void\*) &value1);

pthread\_create(&thread2,NULL,computation,(void\*) &value2);

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

* Lần lượt join - > code xử lý chậm như single thread

pthread\_create(&thread1,NULL,computation,(void\*) &value1);

pthread\_join(thread1,NULL);

pthread\_create(&thread2,NULL,computation,(void\*) &value2);

pthread\_join(thread2,NULL);

**Process**

<https://www.geeksforgeeks.org/difference-between-process-and-thread/>

<https://www.youtube.com/watch?v=RQ5_lulx5jk&list=PLuhsSjG-BPPKC3Mom3rvNnTOCELLYRxj0&index=14&pp=iAQB>

[](https://www.youtube.com/watch?v=RQ5_lulx5jk&list=PLuhsSjG-BPPKC3Mom3rvNnTOCELLYRxj0&index=14&pp=iAQB)

<https://www.youtube.com/watch?app=desktop&v=RQ5_lulx5jk&list=PLuhsSjG-BPPKC3Mom3rvNnTOCELLYRxj0&index=14>

Việc sử dụng fork() và wait() có thể được ứng dụng trong nhiều tình huống khác nhau, bao gồm:

**1. Tạo tiến trình con:**

* fork() là một cách đơn giản để tạo ra một **bản sao** của tiến trình hiện tại.
* Tiến trình con có thể được sử dụng để thực hiện các **nhiệm vụ khác nhau** đồng thời với tiến trình cha.
* Ví dụ:
  + Tạo ra một tiến trình con để xử lý dữ liệu trong khi tiến trình cha tiếp tục nhận đầu vào từ người dùng.
  + Tạo ra một tiến trình con để thực hiện một tác vụ lâu dài và hiển thị thông báo tiến trình cho người dùng.

**2. Giao tiếp giữa tiến trình:**

* fork() có thể được sử dụng để tạo ra các tiến trình **cha-con** có thể giao tiếp với nhau.
* Các tiến trình có thể sử dụng các phương thức như **ống dẫn (pipe)** hoặc **bộ nhớ dùng chung (shared memory)** để trao đổi dữ liệu.
* Ví dụ:
  + Tạo ra một tiến trình con để đọc dữ liệu từ một tệp và gửi dữ liệu đó đến tiến trình cha.
  + Tạo ra một tiến trình con để thực hiện tính toán và gửi kết quả đến tiến trình cha.

**3. Xử lý lỗi:**

* fork() có thể được sử dụng để tạo ra một **tiến trình con** để xử lý các **lỗi** xảy ra trong tiến trình cha.
* Tiến trình con có thể ghi lại thông tin lỗi và thực hiện các hành động để **phục hồi** hoặc **thoát** chương trình một cách an toàn.
* Ví dụ:
  + Tạo ra một tiến trình con để xử lý các lỗi truy cập tệp.
  + Tạo ra một tiến trình con để ghi lại nhật ký lỗi và gửi email thông báo cho người dùng.

**4. Thực hiện các tác vụ song song:**

* fork() có thể được sử dụng để tạo ra nhiều **tiến trình con** thực hiện các tác vụ **song song** với nhau.
* Điều này có thể giúp **tăng tốc độ** thực thi chương trình cho các tác vụ có thể chia nhỏ thành các phần nhỏ độc lập.
* Ví dụ:
  + Tạo ra nhiều tiến trình con để tải xuống các tệp khác nhau đồng thời.
  + Tạo ra nhiều tiến trình con để xử lý các phần khác nhau của một hình ảnh lớn.

**5. Tạo các chương trình server:**

* fork() có thể được sử dụng để tạo ra các **chương trình server** có thể xử lý nhiều yêu cầu từ các **client** khác nhau.
* Tiến trình cha sẽ lắng nghe các yêu cầu mới và tạo ra một **tiến trình con** để xử lý mỗi yêu cầu.
* Ví dụ:
  + Tạo ra một chương trình server web sử dụng fork() để xử lý các yêu cầu HTTP từ các trình duyệt web.
  + Tạo ra một chương trình server trò chơi sử dụng fork() để xử lý các kết nối từ các người chơi.

**Lưu ý:**

* Việc sử dụng fork() và wait() có thể **phức tạp** và dễ dẫn đến lỗi nếu không được sử dụng đúng cách.
* Cần **cân nhắc kỹ lưỡng** trước khi sử dụng fork() và wait trong chương trình của bạn.

**Mutex**

<https://www.youtube.com/watch?v=raLCgPK-Igc&list=PLA1FTfKBAEX4hblYoH6mnq0zsie2w6Wif&index=98>

Multithread is great, but sometime it access a same memory and change it without queue -> mutex is a solution

Mutex is like a lock, only one thread can open the lock and use the code -> lock and wait for next use

Create mutex

// mutex variable

pthread\_mutex\_t mutex;

// initialize the mutex right after create all thread

pthread\_mutex\_init(&mutex, NULL);

// destroy the mutex after joining

pthread\_mutex\_destroy(&mutex);

//install lock on sensitive function

void\* deposit(void \*amount)

{

//lock and wait

pthread\_mutex\_lock(&mutex);

// retrieve the bank balance

int account\_balance = read\_balance();

// make the update locally

account\_balance += \*((int \*) amount);

// write the new bank balance

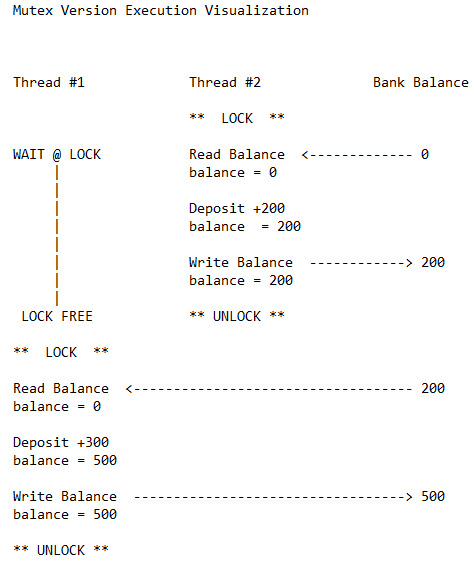
write\_balance(account\_balance);

// unlock to make the critical section available to other threads

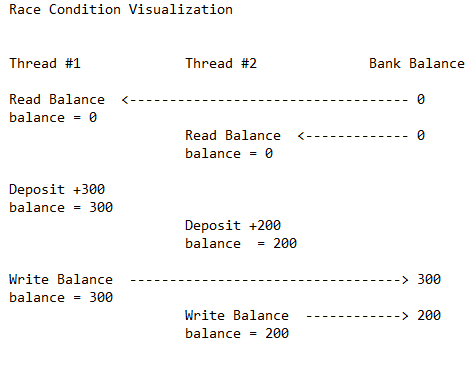
pthread\_mutex\_unlock(&mutex);

return NULL;

}



Code run correctly after using mutex lock



Race condition occurs if we don’t use mutex to lock sensitive code

**Message Queue** <https://www.youtube.com/watch?v=YjxKYxpf51E&list=PLA1FTfKBAEX6dPcQitk_7uL3OwDdjMn90&index=226&pp=iAQB>

[POSIX MQ IN C!! (Message queue) - C TUTORIAL #23 (youtube.com)](https://www.youtube.com/watch?v=iVr1wcBqvfw)

[](https://www.youtube.com/watch?v=iVr1wcBqvfw)

Sử dụng lib: <fcntl.h> (for 0\_\* constant)

<sys/stat.h> (for mode constant)

<mqueue.h>

Gồm 2 phần send và receive:

* Sender:

**Cấu trúc** struct mq\_attr**:**

Cấu trúc struct mq\_attr định nghĩa các thuộc tính sau của một hàng đợi tin nhắn:

* **mq\_flags:** Thuộc tính này cho phép thiết lập các cờ điều khiển cho hàng đợi tin nhắn. Giá trị 0 cho mq\_flag tương ứng với việc không thiết lập cờ điều khiển nào.
* **mq\_maxmsg:** Thuộc tính này cho biết số lượng tin nhắn tối đa mà hàng đợi có thể lưu trữ.
* **mq\_msgsize:** Thuộc tính này cho biết kích thước tối đa của một tin nhắn trong hàng đợi.
* **mq\_curmsgs:** Thuộc tính này cho biết số lượng tin nhắn hiện đang có trong hàng đợi.

Ex:

struct mq\_attr attributes = {

.mq\_flags = 0,

.mq\_maxmsg = 10,

.mq\_curmsgs = 0,

.mq\_msgsize = sizeof(message)

};

Tạo queue:

mqd\_t queue = mq\_open(MQ\_NAME, O\_CREAT | O\_WRONLY, S\_IRUSR | S\_IWUSR, &attributes);

Câu lệnh này sử dụng hàm mq\_open() để mở một hàng đợi tin nhắn (message queue) có tên MQ\_NAME.

**Cấu trúc của câu lệnh:**

* mqd\_t queue**:** Biến được sử dụng để lưu trữ mã nhận dạng của hàng đợi tin nhắn.
* mq\_open(MQ\_NAME, ...)**:** Hàm mq\_open() được sử dụng để mở một hàng đợi tin nhắn.
  + MQ\_NAME**:** Tên của hàng đợi tin nhắn.
  + O\_CREAT | O\_WRONLY**:** Các cờ điều khiển được sử dụng để mở hàng đợi tin nhắn.
    - O\_CREAT**:** Tạo mới hàng đợi tin nhắn nếu nó không tồn tại.
    - O\_WRONLY**:** Mở hàng đợi tin nhắn với quyền ghi.
  + S\_IRUSR | S\_IWUSR**:** Quyền truy cập của chủ sở hữu đối với hàng đợi tin nhắn.
    - S\_IRUSR**:** Quyền đọc.
    - S\_IWUSR**:** Quyền ghi.
  + &attributes**:** Con trỏ đến cấu trúc struct mq\_attr chứa các thuộc tính của hàng đợi tin nhắn.

**Hàm** mq\_open()**:**

Hàm mq\_open() trả về một mã nhận dạng (descriptor) của hàng đợi tin nhắn nếu thành công. Nếu thất bại, hàm trả về giá trị -1 và đặt errno để biểu thị lỗi.

Các cấu trúc như O\_CREAT là các cờ điều khiển (flag) được sử dụng để thiết lập các thuộc tính cho các thao tác tập tin và IPC (Inter-Process Communication) trong hệ thống Unix/Linux.

Nguồn gốc:

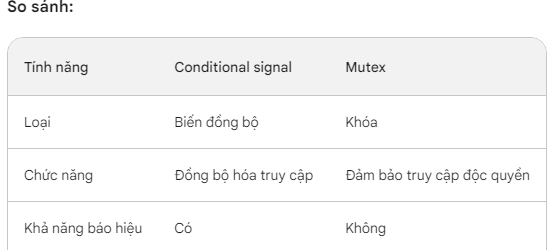
Cấu trúc cờ được định nghĩa trong các tập tin header khác nhau, tùy thuộc vào chức năng cụ thể. Ví dụ:

* Cờ O\_CREAT được định nghĩa trong tập tin <fcntl.h>.
* Cờ O\_WRONLY được định nghĩa trong tập tin <sys/stat.h>.
* Cờ S\_IRUSR và S\_IWUSR được định nghĩa trong tập tin <sys/stat.h>.

Code sử dụng mq\_close để đóng hàng đợi và mq\_unlink để xóa hàng đợi (tùy chọn).

**Conditional signal**

<https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/>

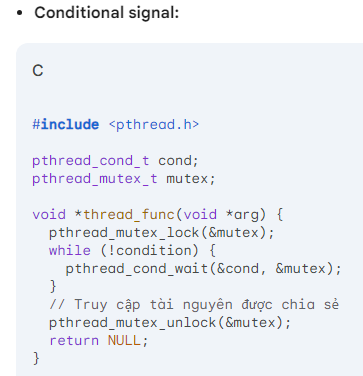


**Conditional signal:**

* Là một **biến đồng bộ** được sử dụng để **đồng bộ hóa** việc truy cập vào tài nguyên được chia sẻ giữa các luồng.
* Khi một luồng muốn truy cập vào tài nguyên được bảo vệ bởi conditional signal, nó phải **giữ (acquire)** conditional signal.
* Nếu conditional signal đã được giữ bởi một luồng khác, luồng hiện tại sẽ **bị chặn (blocked)** cho đến khi luồng đang giữ conditional signal **nhả (release)** nó.
* Conditional signal cũng có thể được sử dụng để **báo hiệu (signal)** cho các luồng đang chờ đợi rằng tài nguyên đã sẵn sàng để truy cập.

**Mutex:**

* Là một **khóa (lock)** được sử dụng để **đảm bảo** rằng chỉ có một luồng có thể truy cập vào tài nguyên được chia sẻ tại một thời điểm.
* Khi một luồng muốn truy cập vào tài nguyên được bảo vệ bởi mutex, nó phải **khóa (lock)** mutex.
* Nếu mutex đã được khóa bởi một luồng khác, luồng hiện tại sẽ **bị chặn (blocked)** cho đến khi luồng đang giữ mutex **mở khóa (unlock)** nó.
* Mutex không có khả năng báo hiệu cho các luồng đang chờ đợi.



***Signal:*Giải thích về**pthread\_cond\_signal(&cond1);**:**

**Hàm** pthread\_cond\_signal()**:**

* Hàm này được sử dụng để **báo hiệu cho một hoặc nhiều luồng đang chờ đợi trên biến điều kiện cụ thể**.
* Khi một luồng gọi pthread\_cond\_signal(), nó sẽ **đánh thức một luồng đang chờ đợi trên biến điều kiện được chỉ định**.
* Luồng được đánh thức sẽ tiếp tục thực thi sau khi nó **nhận được quyền truy cập vào mutex được liên kết với biến điều kiện**.

**Cú pháp:**

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

**Tham số:**

* cond: Biến điều kiện mà tín hiệu được gửi đến.

**Giá trị trả về:**

* 0 nếu thành công.
* Lỗi nếu không thành công.

**Ví dụ:**

Giả sử bạn có hai luồng:

* **Luồng A:** Cập nhật dữ liệu trong một bộ đệm.
* **Luồng B:** Đọc dữ liệu từ bộ đệm.

Để đảm bảo rằng luồng B chỉ đọc dữ liệu sau khi luồng A cập nhật xong, bạn có thể sử dụng pthread\_cond\_signal().

**Cách thực hiện:**

1. Khai báo một biến condition variable.
2. Trong luồng A, sau khi cập nhật dữ liệu, hãy gọi pthread\_cond\_signal() để báo hiệu cho luồng B.
3. Trong luồng B, trước khi đọc dữ liệu, hãy gọi pthread\_cond\_wait() để chờ tín hiệu từ luồng A.

**Lưu ý:**

* Luồng gọi pthread\_cond\_signal() **phải sở hữu mutex được liên kết với biến điều kiện**.
* Nếu không có luồng nào đang chờ đợi trên biến điều kiện, pthread\_cond\_signal() sẽ **không có tác dụng gì**.

**Kết luận:**

pthread\_cond\_signal() là một công cụ mạnh mẽ để đồng bộ hóa luồng trong các ứng dụng pthread. Sử dụng pthread\_cond\_signal() đúng cách có thể giúp nâng cao hiệu quả, tính an toàn và đơn giản cho mã của bạn.

**Ngoài ra:**

* pthread\_cond\_signal(&cond1); sẽ gửi tín hiệu đến biến điều kiện có tên cond1.
* Tín hiệu này sẽ đánh thức một luồng đang chờ đợi trên biến điều kiện cond1.
* Luồng được đánh thức sẽ tiếp tục thực thi sau khi nó nhận được quyền truy cập vào mutex được liên kết với cond1.

***Wait***

pthread\_cond\_wait(&cond1, &lock);**trong Pthreads:**

Hàm pthread\_cond\_wait() là một hàm đồng bộ hóa trong thư viện Pthreads. Nó sử dụng **biến điều kiện (**cond**)** và **mutex (**lock**)** để tạm thời dừng luồng hiện tại chờ đợi một điều kiện cụ thể được đáp ứng.

**Phân tích từng thành phần:**

* pthread\_cond\_wait(&cond1, &lock);:
  + cond1: Biến điều kiện mà luồng sẽ chờ đợi tín hiệu trên đó.
  + &lock: Con trỏ tới mutex được sử dụng để đồng bộ hóa truy cập vào biến điều kiện và tài nguyên chung.
* **Cơ chế hoạt động:**
  + Luồng gọi pthread\_cond\_wait() sẽ giải phóng **mutex (**lock**)**, nhưng vẫn duy trì quyền sở hữu về mặt logic.
  + Luồng bị **ngăn chặn** cho đến khi có luồng khác gọi pthread\_cond\_signal() trên cùng một biến điều kiện (cond1).
  + Khi nhận được tín hiệu, luồng sẽ **cố gắng lấy lại mutex (**lock**)**.
  + Nếu lấy được mutex, luồng sẽ **tiếp tục thực thi** từ điểm bị ngắt.
  + Nếu không lấy được mutex ngay lập tức, luồng sẽ tiếp tục chờ đợi trong **hàng đợi** của biến điều kiện.

**Lưu ý:**

* Luồng **phải sở hữu mutex (**lock**)** trước khi gọi pthread\_cond\_wait().
* Luồng sẽ tự động giải phóng mutex khi gọi pthread\_cond\_wait().
* Luồng **chỉ nên gọi**pthread\_cond\_wait() khi nó đang chờ đợi một điều kiện cụ thể được đáp ứng bởi luồng khác.
* Sử dụng sai pthread\_cond\_wait() và pthread\_cond\_signal() có thể dẫn đến tình trạng **deadlock**.

**Ví dụ:**

Giả sử bạn có hai luồng:

* **Luồng A:** Cập nhật dữ liệu trong một bộ đệm.
* **Luồng B:** Đọc dữ liệu từ bộ đệm.

Để đảm bảo rằng luồng B chỉ đọc dữ liệu sau khi luồng A cập nhật xong, bạn có thể sử dụng pthread\_cond\_wait() và pthread\_cond\_signal().

**Cách thực hiện:**

1. Khai báo một biến condition variable.
2. Trong luồng A, sau khi cập nhật dữ liệu, hãy gọi pthread\_cond\_signal() để báo hiệu cho luồng B.
3. Trong luồng B, trước khi đọc dữ liệu, hãy gọi pthread\_cond\_wait() để chờ tín hiệu từ luồng A.