Chapter 1. Introduction

1.1. Why Linux?

1.2. Embedded Linux Today

1.3. Open Source and the GPL

Chapter 2. Your First Embedded Experience

2.1. Embedded or Not?

2.1.1. BIOS Versus Bootloader

2.2. Anatomy of an Embedded System

2.2.1. Typical Embedded Linux Setup

2.2.2. Starting the Target Board

2.2.3. Booting the Kernel

+ Uboot ~ BIOS mini, khởi tạo phần cứng nhưng không giao diện người dùng và sẽ off ngay khi kernel linux khởi động.

2.2.4. Kernel Initialization: Overview

+ Linux mounts a root file system.

+ Linux requires a file system.

2.2.5. First User Space Process: init

+ Kernel context.

+ User space

2.3. Storage Considerations

2.3.1. Flash Memory:

Bộ nhớ Flash, như thẻ Compact Flash và SD, là bộ nhớ thể rắn không có bộ phận chuyển động, bền bỉ và tiết kiệm năng lượng. Nó được sử dụng phổ biến trong các hệ thống nhúng, với dung lượng từ vài MB đến vài GB. Bộ nhớ Flash có thể ghi và xóa dữ liệu dưới sự điều khiển của phần mềm, nhưng tốc độ ghi và xóa chậm hơn ổ cứng truyền thống.

Bộ nhớ Flash được chia thành các **erase blocks** lớn, và khi thay đổi dữ liệu, cả block phải bị xóa và ghi lại. Điều này dẫn đến thời gian ghi lâu hơn so với ổ cứng. Ngoài ra, Flash có **tuổi thọ ghi hạn chế** (khoảng 100.000 vòng ghi mỗi block), vì vậy cần tránh ghi quá nhiều lần vào bộ nhớ Flash, như trong việc ghi log hệ thống.

2.3.2. NAND Flash

NAND Flash là công nghệ Flash mới, cải tiến so với **NOR Flash** truyền thống. NAND Flash có **block nhỏ hơn**, giúp ghi nhanh và hiệu quả hơn. **NOR Flash** truy cập dữ liệu theo cách song song, có thể được vi xử lý truy cập trực tiếp, nhưng hiệu suất thấp do không thể cache lệnh. **NAND Flash** sử dụng giao diện chuỗi, phù hợp hơn cho lưu trữ dữ liệu lớn, và có tuổi thọ ghi lâu hơn, nhưng thời gian xóa ngắn hơn so với NOR Flash.

2.3.3. Flash Usage

Figure 2-4. Example Flash memory layout

Bootloader thường được đặt ở đầu hoặc cuối của mảng bộ nhớ Flash. Sau bootloader, không gian được phân bổ cho hình ảnh kernel Linux và hình ảnh hệ thống tập tin ramdisk, trong đó chứa hệ thống tập tin gốc (root file system). Thông thường, hình ảnh kernel Linux và hệ thống tập tin ramdisk được nén, và bootloader sẽ xử lý nhiệm vụ giải nén trong quá trình khởi động.

2.3.4. Flash File Systems:

Hệ thống tệp Flash giúp khắc phục hạn chế của bố trí bộ nhớ Flash đơn giản, với các cải tiến như **wear leveling** (cân bằng mài mòn) để kéo dài tuổi thọ Flash. Tuy nhiên, một vấn đề lớn là **nguy cơ mất dữ liệu** khi mất điện, do quá trình ghi vào Flash có thể chậm và cần xóa toàn bộ khối Flash. **JFFS2** là một hệ thống tệp phổ biến với các tính năng như cải thiện cân bằng mài mòn, nén và giải nén dữ liệu, và hỗ trợ liên kết cứng Linux, giúp cải thiện hiệu suất và giảm nguy cơ mất dữ liệu.

2.3.5. Memory Space

Hầu hết các hệ điều hành nhúng cũ đều xem và quản lý bộ nhớ hệ thống như một không gian địa chỉ lớn, phẳng duy nhất. Các thiết kế phần cứng thường bắt đầu DRAM từ dưới cùng của phạm vi và bộ nhớ Flash từ trên xuống.

Trong các hệ thống nhúng truyền thống dựa trên hệ điều hành cũ, hệ điều hành và tất cả các tác vụ đều có quyền truy cập ngang nhau vào tất cả các tài nguyên trong hệ thống. Một lỗi trong một tiến trình có thể làm mất dữ liệu bộ nhớ ở bất kỳ đâu trong hệ thống, dù đó là của chính tiến trình đó, của hệ điều hành, của một tác vụ khác, hoặc thậm chí là một thanh ghi phần cứng nào đó trong không gian địa chỉ. Mặc dù cách tiếp cận này có đặc điểm giá trị nhất là sự đơn giản, nhưng nó cũng dẫn đến những lỗi khó chẩn đoán.

Bộ Quản Lý Bộ Nhớ (MMUs): Quyền truy cập cho phép hệ điều hành chỉ định các quyền truy cập bộ nhớ cụ thể cho các tác vụ cụ thể. Việc chuyển đổi bộ nhớ cho phép hệ điều hành ảo hóa không gian địa chỉ của nó, điều này mang lại nhiều lợi ích.

2.3.6. Execution Contexts

Khi Linux khởi động, một trong những nhiệm vụ đầu tiên của nó là cấu hình **Bộ Quản Lý Bộ Nhớ (MMU)** và kích hoạt **chuyển đổi địa chỉ**, thiết lập bộ nhớ ảo. Kernel hoạt động trong không gian bộ nhớ ảo riêng của nó, với địa chỉ ảo mặc định bắt đầu từ

0xC0000000 trong các phiên bản gần đây. Các tiến trình có thể thực thi trong hai ngữ cảnh:

- **Ngữ cảnh User Space**: Các chương trình ứng dụng hoạt động trong không gian này và chỉ có thể truy cập bộ nhớ của chính chúng. Chúng tương tác với các tài nguyên có quyền hạn (ví dụ: tệp, thiết bị) thông qua các cuộc gọi hệ thống đến kernel.
- **Ngữ cảnh Kernel**: Ngữ cảnh này thực thi mã kernel thay mặt cho một tiến trình, thường là khi một cuộc gọi hệ thống được thực hiện, chẳng hạn như yêu cầu đọc tệp.

Ví dụ, khi một ứng dụng đọc một tệp:

a. Tiến trình bắt đầu ở user space (hàm read() của thư viện C).
b. Một cuộc gọi hệ thống chuyển ngữ cảnh sang kernel, nơi phát ra yêu cầu đọc từ ổ cứng.
c. Ứng dụng sẽ được đưa vào hàng đợi chờ cho đến khi dữ liệu sẵn sàng.
d. Khi phần cứng phát tín hiệu ngắt cho biết dữ liệu đã sẵn sàng, kernel đọc dữ liệu và tiếp tục ứng dụng.

2.3.7. Process Virtual Memory

Khi một tiến trình được tạo ra (ví dụ: chạy lệnh ls), kernel của Linux sẽ gán cho nó các địa chỉ bộ nhớ ảo độc lập với không gian địa chỉ của kernel hoặc các tiến trình khác. Không có mối quan hệ trực tiếp giữa bộ nhớ ảo và các vị trí bộ nhớ vật lý trên board. Thêm vào đó, trong suốt vòng đời của nó, một tiến trình có thể chiếm các địa chỉ vật lý khác nhau do việc phân trang và hoán đổi bộ nhớ.

Các khái niệm chính trong ví dụ:

- **Bộ nhớ ảo và không gian địa chỉ**:
  - Tiến trình nhìn thấy các địa chỉ ảo được gán bởi kernel, có thể không trùng khớp với các địa chỉ vật lý.
  - Ví dụ:
    - Thực thi mã (main): 0x10000418 (địa chỉ ảo trong RAM cao).
    - Biến stack: 0x7ff8ebb0 (trong phần trên của không gian địa chỉ 32-bit).
    - BSS (biến toàn cục chưa khởi tạo): 0x10010a1c.
    - Phần dữ liệu (biến toàn cục đã khởi tạo): 0x10010a18.
  - Các địa chỉ ảo này được ánh xạ tới RAM vật lý ở đâu đó trong 256MB bộ nhớ có sẵn trên board AMCC Yosemite.
- **Hoán đổi và phân trang**:
  - Khi RAM vật lý ít, kernel hoán đổi các trang bộ nhớ ra lưu trữ ngoài (ví dụ: ổ cứng) để giải phóng bộ nhớ cho các tiến trình đang hoạt động.
  - Trong các hệ thống nhúng, việc hoán đổi thường bị vô hiệu hóa vì:
    - Bộ nhớ Flash (thường được sử dụng) có số chu kỳ ghi hạn chế.
    - Bộ nhớ Flash chậm hơn nhiều so với RAM, dẫn đến các vấn đề về hiệu suất.
- **Lưu ý cho hệ thống nhúng**:

- Khi không có hoán đổi, các nhà phát triển phải thiết kế ứng dụng sao cho phù hợp với giới hạn bộ nhớ vật lý có sẵn.

Điều cần lưu ý: Bộ nhớ ảo của Linux cung cấp sự trừu tượng và linh hoạt, cho phép các tiến trình có không gian địa chỉ riêng. Tuy nhiên, đối với các hệ thống nhúng, việc quản lý bộ nhớ cẩn thận là rất quan trọng vì việc hoán đổi ra đĩa là không khả thi hoặc không mong muốn.

2.3.8. Cross-Development Environment

Khi phát triển ứng dụng và driver cho hệ thống nhúng, chúng ta cần các công cụ như trình biên dịch và tiện ích có khả năng tạo ra các tệp thực thi nhị phân ở định dạng phù hợp với kiến trúc mục tiêu. Quá trình phát triển phần mềm trong môi trường này được gọi là **cross-development** (phát triển chéo), trái ngược với phát triển bản địa (native development):

Phát triển bản địa (Native Development):

- Trình biên dịch chạy trên cùng một kiến trúc mà mã nguồn được nhắm tới.
- Ví dụ: Biên dịch và chạy chương trình "Hello World" trên máy tính để bàn sử dụng trình biên dịch bản địa như gcc.

Phát triển chéo (Cross-Development):

- Trình biên dịch chạy trên một máy chủ phát triển (ví dụ: máy tính để bàn) nhưng tạo ra các tệp thực thi nhị phân cho một kiến trúc mục tiêu khác (ví dụ: PowerPC).
- Phát triển chéo là cần thiết vì hệ thống nhúng thường thiếu các tài nguyên cần thiết (sức mạnh CPU, bộ nhớ) để phát triển và biên dịch phần mềm trực tiếp.

Các khái niệm chính:

- **Tệp Header và Thư viện**:
  - Trình biên dịch có các đường dẫn tích hợp để tìm các tệp header (ví dụ: stdio.h).
  - Trình liên kết có thư viện mặc định (ví dụ: libc-*) để giải quyết các tham chiếu ngoài như printf().
- **Thách thức trong phát triển chéo**:
  - Trình biên dịch chéo phải sử dụng đúng các tệp header và thư viện riêng cho mục tiêu để tạo ra các tệp nhị phân tương thích.
  - Nếu trình biên dịch chéo vô tình liên kết với thư viện từ hệ thống máy chủ (ví dụ: thư viện C của x86 cho mục tiêu PowerPC), tệp nhị phân sẽ không sử dụng được, dẫn đến lỗi biên dịch hoặc lỗi liên kết (hoặc lỗi khi chạy).
- **Cấu hình đúng trình biên dịch chéo**:
  - Trình biên dịch chéo phải được cấu hình để:
    - Tìm kiếm các vị trí không chuẩn cho các tệp header dành riêng cho mục tiêu.
    - Sử dụng các thư viện đặc biệt cho kiến trúc mục tiêu.
  - Môi trường phát triển chéo được cấu hình đúng là yếu tố quan trọng để thành công.

Điều cần lưu ý: Môi trường phát triển chéo phức tạp hơn môi trường phát triển bản địa, yêu cầu cấu hình cẩn thận các trình biên dịch, trình liên kết và thư viện. Những môi trường này là rất quan trọng để phát triển ứng dụng và driver cho các hệ thống nhúng hạn chế tài nguyên. Cấu hình đúng giúp đảm bảo rằng các tệp nhị phân tương thích với kiến trúc mục tiêu và tránh các vấn đề như liên kết thư viện sai.

2.4. Embedded Linux Distributions

Trong macro của C (định nghĩa bằng #define), có một số toán tử đặc biệt giúp thao tác trên tham số của macro. Dưới đây là danh sách các toán tử quan trọng:

1. Toán tử Dán Chuỗi (##)

- Dùng để **nối hai token lại với nhau** trong macro.
- Giúp tạo ra mã nguồn động từ các tham số macro.

🔹 **Ví dụ:**

```
#define CONCAT(a, b) a##b

int CONCAT(var, 1) = 10;  // Biến trở thành: int var1 = 10;
```

int CONCAT(var, 1) = 10; // Biến trở thành: int var1 = 10;

📌 **Ứng dụng thực tế trong kernel:**

- Thường dùng để tạo tên biến hoặc function theo tham số truyền vào.
- Ví dụ: LIST_HEAD(name) -> struct list_head name = { &(name), &(name) }

2. Toán tử Chuyển Thành Chuỗi (#)

- Biến tham số macro thành **chuỗi ký tự** (string literal).

🔹 **Ví dụ:**

```
#define STRINGIFY(x) #x

printf("%s\n", STRINGIFY(Hello World));  // Kết quả: "Hello World"
```

printf("%s\n", STRINGIFY(Hello World)); // Kết quả: "Hello World"

📌 **Ứng dụng thực tế trong kernel:**

- Chuyển các macro thành chuỗi để log hoặc debug.
- Ví dụ: WARN_ON(condition) trong kernel có thể sử dụng #condition để in thông báo lỗi.

3. Toán tử Biến Số (__VA_ARGS__)

- Dùng trong **macro có số lượng tham số không xác định**.
- Cho phép truyền số lượng tham số linh hoạt vào macro.

🔹 **Ví dụ:**

```c
#define PRINTF(fmt, ...) printf(fmt, __VA_ARGS__)

PRINTF("Number: %d, String: %s\n", 42, "Hello");
```

👉 **Tương đương với:**

```c
printf("Number: %d, String: %s\n", 42, "Hello");
```

📌 **Ứng dụng thực tế trong kernel:**

- Dùng trong các macro như pr_info(), pr_err(), dev_dbg() để log thông tin linh hoạt.

4. Toán tử #@ (GCC extension - Characterizing tokens)

- Chuyển tham số macro thành **ký tự đơn**.
- Đây là mở rộng của GCC, không có trong chuẩn C.

🔹 **Ví dụ:**

```c
#define CHARIFY(x) #@x

char ch = CHARIFY(A);  // Tương đương: char ch = 'A';
```

📌 **Ít gặp trong kernel** nhưng có thể dùng khi cần chuyển token thành ký tự.

5. Toán tử defined (Dùng với #if)

- Kiểm tra xem một macro có được định nghĩa hay không.

🔹 **Ví dụ:**

```c
#if defined(DEBUG)
    printf("Debug mode is ON\n");
#endif
```

6.36.1 Common Variable Attributes

attribute giống như một chiếc note để cung cấp thông tin cho các function GCC chạy đúng , chuẩn. Không ảnh hưởng đến nội dung đoạn mã, nó chỉ giúp đoạn mã chạy tối ưu theo đúng mong muốn của lập trình viên.

alias ("target")

1. Ý nghĩa:

- alias là một **variable attribute** trong GCC, cho phép một biến hoặc hàm được định nghĩa như một alias (bí danh) của một biến hoặc hàm khác.
- Khi sử dụng alias, cả hai tên đều trỏ đến cùng một vị trí trong bộ nhớ.
    - Việc sử dụng __attribute__((alias("target"))) tuy chỉ giới hạn trong cùng một file .c, nhưng nó vẫn có những lợi ích thực tế nhất định, đặc biệt trong kernel và thư viện hệ thống. Dưới đây là một số lý do:

1. **Tương thích ngược (Backward Compatibility)**

- Khi muốn đổi tên một hàm/biến mà vẫn giữ cho mã cũ hoạt động, alias giúp duy trì cả hai tên mà không cần sửa mã nguồn cũ. Ví dụ:

```
int new_function() { return 42; }

int __attribute__((alias("new_function"))) old_function;
```

- Điều này hữu ích trong các API hệ thống hoặc thư viện khi muốn đổi tên mà không phá vỡ chương trình cũ.

2. **Tối ưu hóa và giảm mã dư thừa**

- Trong một số trường hợp, alias giúp tránh duplicate code bằng cách tạo nhiều tên cho cùng một thực thể thay vì tạo nhiều phiên bản giống nhau của một hàm/biến.

3. **Hỗ trợ cho các nền tảng khác nhau**

- Một số hệ thống có thể cần đặt tên hàm khác nhau tùy vào kiến trúc, alias giúp ánh xạ một tên phổ biến đến các phiên bản khác nhau mà không phải sửa code chính.

4. **Cách tổ chức mã nguồn**

- Trong kernel hoặc thư viện lớn, một hàm có thể có nhiều alias để phản ánh các mục đích sử dụng khác nhau mà không cần tạo wrapper function tốn hiệu năng.
- Mặc dù alias bị giới hạn trong cùng một file .c, nhưng nếu cần dùng nó giữa các file khác nhau, có thể kết hợp với #pragma weak hoặc sử dụng linker script.

2. Cách sử dụng:

- Alias phải có **cùng kiểu dữ liệu** với biến hoặc hàm gốc (ngoại trừ các qualifier như const, volatile).
- Alias và biến/hàm gốc **phải được định nghĩa trong cùng một translation unit** (cùng file .c khi biên dịch).
- Nếu target không tồn tại trong cùng một file, trình biên dịch sẽ báo lỗi.

3. Ví dụ về alias cho biến:

```
int var_target;

extern int __attribute__((alias("var_target"))) var_alias;
```

=> Ở đây, var_alias là một alias của var_target, tức là cả hai biến này cùng trỏ đến một địa chỉ trong bộ nhớ.

4. Ví dụ về alias cho hàm:

```
void my_function() {
    printf("Hello, World!\n");
}

void new_function() __attribute__((alias("my_function")));
```

=> new_function() thực chất là một alias của my_function(), nên gọi new_function() cũng tương đương với gọi my_function().

5. Lưu ý:

- Nếu không có alias, mỗi định nghĩa có extern sẽ được coi là một đối tượng khác nhau.
- Một số hệ thống có thể không hỗ trợ alias do yêu cầu hỗ trợ từ assembler và object file format.

6. Ứng dụng thực tế:

- Sử dụng alias để tạo các tên thay thế mà không cần sao chép code.
- Hữu ích khi làm việc với ABI (Application Binary Interface) hoặc duy trì khả năng tương thích với mã cũ.

copy

copy (function)

Thuộc tính **copy (function)** trong GCC được dùng để sao chép các thuộc tính của một hàm (hoặc biến) khác sang một hàm (hoặc biến) mới. Điều này đặc biệt hữu ích khi làm việc với các **alias** (tên thay thế) hoặc **function resolver** mà cần đảm bảo chúng có cùng tập thuộc tính với hàm gốc.

Cách hoạt động

- **copy(function)** sao chép tất cả **thuộc tính cú pháp và ngữ nghĩa** (syntactic and semantic attributes) từ function sang một hàm khác.
- Tuy nhiên, **không** sao chép các thuộc tính liên quan đến **liên kết hoặc khả năng hiển thị** như:
  - alias
  - visibility
  - weak
- Các thuộc tính đặc biệt như deprecated hoặc target_clones cũng **không** được sao chép.

Ví dụ minh họa

Tạo một alias có cùng thuộc tính với hàm gốc

c Sao chép Chỉnh sửa

```c
#include <stdlib.h>

// Định nghĩa macro StrongAlias để tạo alias có cùng thuộc tính
#define StrongAlias(TargetFunc, AliasDecl)  \
  extern __typeof__ (TargetFunc) AliasDecl  \
    __attribute__ ((alias (#TargetFunc), copy (TargetFunc)));

// Hàm allocate có các thuộc tính đặc biệt như alloc_size, malloc, nothrow
extern __attribute__ ((alloc_size (1), malloc, nothrow))
void* allocate(size_t);

// Tạo alias `alloc` cho `allocate`, đồng thời giữ nguyên các thuộc tính
StrongAlias(allocate, alloc);

int main() {
    void* ptr = alloc(100); // Gọi alias `alloc`, nhưng thực chất là `allocate`
    free(ptr);
    return 0;
}
```

Cách viết **__attribute__ ((alias (#TargetFunc), copy (TargetFunc)))** thể hiện rằng **alias và copy là hai thuộc tính độc lập (attributes) được áp dụng đồng thời lên AliasDecl:**

- __attribute__ ((...)) chứa **danh sách các attributes**.
- Các attributes **độc lập** và có thể hoán đổi thứ tự.
- alias tạo alias, còn copy đảm bảo alias có cùng thuộc tính với gốc.
- Toán tử # biến tham số macro thành chuỗi (string), cần thiết cho alias.

Giải thích

1. **Hàm gốc allocate(size_t)** có các thuộc tính:
   - alloc_size(1): Cho biết kích thước bộ nhớ cấp phát dựa trên tham số đầu tiên.
   - malloc: Thông báo cho trình biên dịch rằng đây là một hàm cấp phát bộ nhớ.
   - nothrow: Chỉ ra rằng hàm không ném ngoại lệ.
2. **Tạo alias alloc trỏ đến allocate** bằng macro StrongAlias:
   - __typeof__(TargetFunc) AliasDecl: Đảm bảo alloc có cùng kiểu với allocate.
   - alias (#TargetFunc): Định nghĩa alias alloc cho allocate.
   - copy(TargetFunc): Sao chép tất cả thuộc tính của allocate sang alloc.

Với cách này, alloc() thực chất gọi allocate(), nhưng **vẫn giữ nguyên các thuộc tính tối ưu hóa của allocate()**, giúp trình biên dịch tối ưu code tốt hơn.

Ứng dụng thực tế

- **Thư viện chuẩn (Standard Library):**
  Khi tạo các alias cho hàm cấp phát bộ nhớ (malloc, calloc, realloc...), thuộc tính copy đảm bảo các alias này vẫn giữ được thông tin tối ưu hóa.
- **Function resolver (Trình phân giải hàm):**
  Khi có nhiều phiên bản của một hàm trên các kiến trúc khác nhau, copy giúp duy trì các thuộc tính quan trọng mà không cần khai báo lại.
- **Tối ưu hóa hiệu suất:**
  Việc sử dụng copy giúp trình biên dịch nhận diện đúng các thuộc tính của alias, từ đó có thể thực hiện các tối ưu hóa như **eliminate redundant calls** hoặc **improve memory safety**.

Nhìn chung, copy giúp đảm bảo rằng các alias hoặc function resolver không chỉ trùng khớp về kiểu mà còn giữ nguyên các thuộc tính quan trọng, tránh lỗi và giúp trình biên dịch tối ưu tốt hơn.

4o

aligned (aligment)

1. **Định nghĩa chung:**
   - Thuộc tính **aligned** chỉ định mức căn lề tối thiểu cho một biến hoặc trường của cấu trúc.
   - Mức căn lề được đo bằng byte và phải là một hằng số nguyên, dạng lũy thừa của 2.
   - Nếu không chỉ định giá trị căn lề, trình biên dịch sẽ sử dụng mức căn lề mặc định của kiến trúc (thường là 8 hoặc 16 byte, nhưng không phải lúc nào cũng như vậy).
2. **Ví dụ sử dụng:**
   - **Biến toàn cục:**

     ```
     int x __attribute__ ((aligned (16))) = 0;
     ```

     - Biến x được cấp phát trên đường biên (boundary) căn lề 16 byte.
     - Trên vi xử lý 68040, điều này có thể được dùng cùng với biểu thức asm để truy cập lệnh **move16** (đòi hỏi toán hạng căn lề 16 byte).
   - **Trường của cấu trúc:**

     ```
     struct foo { int x[2] __attribute__ ((aligned (8))); };
     ```

     - Tạo cặp số nguyên được căn lề theo double-word, thay thế cho cách tạo union chứa thành viên kiểu double.
1. **Chỉ định mặc định và tùy chọn:**
   - Có thể bỏ qua tham số căn lề để dùng mức căn lề mặc định của kiến trúc mục tiêu.
   - Mức căn lề mặc định thường đủ cho các kiểu dữ liệu vô hướng, nhưng có thể không phù hợp cho các kiểu vector trên một số kiến trúc hỗ trợ vector.
2. **Macro BIGGEST_ALIGNMENT:**

- GCC cung cấp macro **BIGGEST_ALIGNMENT** là mức căn lề lớn nhất được sử dụng cho bất kỳ kiểu dữ liệu nào trên máy mục tiêu.
- Ví dụ:

  short array[3] __attribute__ ((aligned (__BIGGEST_ALIGNMENT__)));

- Giúp tối ưu các phép sao chép dữ liệu bằng cách cho phép trình biên dịch sử dụng các lệnh sao chép khối bộ nhớ lớn nhất.

- **Giới hạn và lưu ý khi sử dụng:**
  - **Trình liên kết và định dạng tệp đối tượng:**
    - Một số hệ thống chỉ hỗ trợ căn lề tối đa nhất định (ví dụ: 8 byte); dù chỉ định 16 byte, thực tế có thể chỉ căn lề đến mức tối đa mà trình liên kết cho phép.
  - **Struct và thuộc tính packed:**
    - Khi áp dụng cho struct, **aligned** chỉ có thể tăng mức căn lề.
    - Nếu muốn giảm mức căn lề, cần kết hợp với thuộc tính **packed**.
  - **Biến tĩnh vs. biến trên ngăn xếp:**
    - Các biến tĩnh có thể bị giới hạn bởi trình liên kết.
    - Các biến trên ngăn xếp không gặp hạn chế này, GCC có thể căn lề theo yêu cầu.
- **Ứng dụng trong GCC:**
  - **Mục đích:**
    - Kiểm soát mức căn lề của biến, trường cấu trúc, và hàm nhằm đáp ứng yêu cầu của phần cứng và tối ưu hiệu năng xử lý bộ nhớ.
  - **Tác dụng đối với hàm:**
    - Thuộc tính **aligned** cũng có thể áp dụng cho các hàm, đảm bảo chúng được căn lề phù hợp với các tối ưu hóa cụ thể của kiến trúc máy tính.

Như vậy, việc sử dụng **aligned** trong GCC giúp lập trình viên có thể điều chỉnh vị trí lưu trữ dữ liệu trong bộ nhớ theo yêu cầu, từ đó tối ưu hiệu suất và đảm bảo tương thích với phần cứng.

assume_aligned (alignment)

assume_aligned (alignment, offset)

1. **Mục đích của thuộc tính:**
   - Thuộc tính này được áp dụng cho hàm trả về con trỏ, nhằm chỉ ra với trình biên dịch rằng con trỏ trả về được căn lề (aligned) theo một ranh giới nhất định.
2. **Cú pháp và ý nghĩa tham số:**
   - **assume_aligned(alignment):**
     - Xác định rằng con trỏ trả về được căn lề theo một biên (boundary) có kích thước alignment byte.
     - Ví dụ: Nếu alignment là 16, điều đó có nghĩa rằng địa chỉ trả về luôn là số chia hết cho 16.
   - **assume_aligned(alignment, offset):**
     - Tương tự như trên, nhưng có thêm tham số thứ hai offset cho biết độ lệch căn lề.
     - Nghĩa là: giá trị của con trỏ trả về khi chia cho alignment sẽ có dư bằng offset.
     - Ví dụ: Với alignment = 32 và offset = 8, con trỏ trả về có địa chỉ thỏa mãn điều kiện: địa chỉ % 32 = 8.

3. **Các giá trị ý nghĩa:**
   - Giá trị của alignment phải là một lũy thừa của 2 và lớn hơn 1.
   - Giá trị của offset phải lớn hơn 0 và nhỏ hơn alignment.
- **Ví dụ minh họa:**

```c
void* my_alloc1 (size_t) __attribute__((assume_aligned (16)));
void* my_alloc2 (size_t) __attribute__((assume_aligned (32, 8)));
```

  - **my_alloc1:** Trả về con trỏ được căn lề 16 byte, tức là địa chỉ trả về chia hết cho 16.
  - **my_alloc2:** Trả về con trỏ mà khi chia cho 32, dư bằng 8 (địa chỉ % 32 == 8).
- **Lợi ích:**
  - Thông tin căn lề này giúp trình biên dịch tối ưu hóa mã (ví dụ: sử dụng các lệnh SIMD cần căn lề đặc biệt) và cải thiện hiệu năng xử lý dữ liệu.
  - Giúp trình biên dịch đưa ra các giả định an toàn về cách dữ liệu được lưu trữ, từ đó tránh các kiểm tra không cần thiết trong quá trình chạy.

Tóm lại, **assume_aligned** là một công cụ cho phép bạn chỉ định cho trình biên dịch biết rằng con trỏ trả về từ một hàm có căn lề nhất định (hoặc có độ lệch căn lề cụ thể) nhằm hỗ trợ tối ưu hóa mã nguồn và cải thiện hiệu năng.

counted_by (count)

1. **Định nghĩa chung:**
   - Thuộc tính **counted_by** được gắn cho thành viên **mảng linh hoạt (flexible array member) - Ví dụ phần 3.** trong cấu trúc C99.
   - Nó cho biết số lượng phần tử của mảng được xác định bởi trường **count** trong cùng cấu trúc.
   - Lưu ý: Thuộc tính này chỉ có hiệu lực trong C; trong C++ nó sẽ bị bỏ qua.
2. **Mục đích và ứng dụng:**
   - GCC sử dụng thông tin này để **cải thiện việc phát hiện kích thước đối tượng** của các cấu trúc.
   - **Giúp nâng cao chẩn đoán thời gian biên dịch và hỗ trợ các tính năng thời gian chạy như array bound sanitizer và __builtin_dynamic_object_size.**
     - **__builtin_dynamic_object_size** bằng cách truyền vào một con trỏ tới đối tượng cần kiểm tra và một tham số kiểu số nguyên xác định cách tính kích thước. Ví dụ:

       size_t size = __builtin_dynamic_object_size(ptr, 0);

     - Trong đó:
       - **ptr** là con trỏ đến đối tượng (ví dụ: một mảng linh hoạt hoặc một biến được cấp phát động).
       - Tham số thứ hai (ở đây là **0**) chỉ định kiểu tính kích thước. Các giá trị khác nhau (như 1, 2, hoặc 3) có thể được sử dụng để chỉ định các cách tính khác nhau, nhưng **0** là lựa

chọn phổ biến.

- Nếu không xác định được kích thước đối tượng, hàm sẽ trả về giá trị **(size_t)-1**.

3.Ví dụ minh họa:

```
struct P {
  size_t count;
  char other;
  char array[] __attribute__ ((counted_by (count))); // array[] là 1 mảng linh hoạt
} *p;
size_t n = 10;
struct P*buf = malloc(sizeof(struct P) + n * sizeof(char));
```

- Trong đó, mảng **array** là thành viên linh hoạt, số phần tử của nó được xác định bởi trường **count** trong cùng cấu trúc.
- **Thuộc tính counted_by báo cho trình biên dịch biết rằng số phần tử hợp lệ của mảng linh hoạt buf->data được xác định bởi giá trị của trường buf->count. Nhờ đó, khi cần kiểm tra truy cập mảng hoặc thực hiện các kiểm tra an toàn như array bound sanitizer, trình biên dịch sẽ dựa vào giá trị buf->count để xác định phạm vi truy cập hợp lệ của mảng.**

4.Yêu cầu về kiểu dữ liệu và xử lý giá trị:

- Trường biểu diễn số lượng phần tử phải là kiểu số nguyên.
- Nếu không phải, trình biên dịch sẽ báo lỗi và bỏ qua thuộc tính.
- Nếu trường này được gán giá trị số nguyên âm, giá trị đó được xem như bằng 0.

5.Ràng buộc giữa các đối tượng:

- **p->count** phải được khởi tạo trước khi truy cập lần đầu tiên đến **p->array**.
- **p->array** luôn phải có ít nhất **p->count** phần tử khả dụng, kể cả sau khi các giá trị liên quan được cập nhật.
- Lập trình viên cần đảm bảo các yêu cầu này luôn được duy trì; nếu không, trình biên dịch sẽ đưa ra cảnh báo và các tính năng kiểm tra như array bound sanitizer có thể cho kết quả không xác định.

6. Tính năng cập nhật giá trị:

- Mỗi lần tham chiếu đến thành viên mảng **array** sử dụng giá trị mới nhất của trường **count** được gán trước đó.
- Ví dụ:

```
p->count = val1;
p->array[20] = 0;  // Tham chiếu này sử dụng val1 làm số phần tử
p->count = val2;
p->array[30] = 0;  // Tham chiếu này sử dụng val2 làm số phần tử
```

Như vậy, thuộc tính **counted_by** trong GCC giúp xác định rõ mối liên hệ giữa số lượng phần tử của mảng linh hoạt và một trường cụ thể trong cấu trúc, từ đó hỗ trợ các tối ưu hóa và kiểm tra an toàn bộ nhớ.

alloc_size (position)

alloc_size (position-1, position-2)

1. **Mục đích:**
   Thuộc tính **alloc_size** được sử dụng để cung cấp cho trình biên dịch **thông tin về kích thước của đối tượng được cấp phát bởi một hàm cấp phát bộ nhớ**. Điều này giúp trình biên dịch có thể xác định kích thước của đối tượng khi gọi **__builtin_object_size** nhằm mục đích kiểm tra an toàn bộ nhớ, tối ưu hóa hoặc phát hiện lỗi.
   - **__builtin_object_size** là một hàm tích hợp của GCC, dùng để xác định kích thước của đối tượng mà con trỏ trỏ đến, giúp phát hiện lỗi tràn bộ nhớ và hỗ trợ tối ưu hóa mã. Dưới đây là cách sử dụng và giải thích chi tiết:

     size_t __builtin_object_size (const void *ptr, int type)

     - **ptr:** Con trỏ trỏ tới đối tượng cần kiểm tra.
     - **type:** Một giá trị nguyên (thường là 0, 1, 2 hoặc 3) chỉ định cách tính kích thước đối tượng. Các giá trị này xác định mức độ "chắc chắn" của thông tin về kích thước đối tượng mà trình biên dịch có thể xác định.
   - Ý Nghĩa Các Giá Trị type
     - **type = 0:**
       Trả về kích thước tối đa có thể của đối tượng, nếu có thể xác định được. Nếu không, thường trả về (size_t)-1.
     - **type = 1, 2, 3:**
       Các giá trị này được dùng nội bộ bởi trình biên dịch để điều chỉnh mức độ an toàn của việc xác định kích thước; trong hầu hết các trường hợp, bạn có thể dùng type = 0.

2. Cách hoạt động:

- **alloc_size(position):**
  - Chỉ định rằng kích thước của đối tượng được cấp phát bằng giá trị của đối số ở vị trí được chỉ định trong danh sách tham số của hàm.
  - Ví dụ: Với khai báo __attribute__ ((alloc_size (1))), kích thước đối tượng được lấy từ đối số thứ nhất.
- **alloc_size(position-1, position-2):**
  - Chỉ định rằng kích thước của đối tượng được cấp phát bằng tích của hai đối số, cụ thể là đối số tại vị trí **position-1** nhân với đối số tại vị trí **position-2**.
  - Ví dụ: Với khai báo __attribute__ ((alloc_size (1, 2))), kích thước được tính là tích của đối số thứ nhất và đối số thứ hai (giống như hàm **calloc** trong thư viện C).

3.Giới hạn kích thước:

- Kích thước trả về phải là một giá trị dương và nhỏ hơn **PTRDIFF_MAX**. Nếu không, trình biên dịch sẽ báo lỗi hoặc cảnh báo.

4.Ứng dụng:

- Thông tin này được sử dụng bởi GCC để cải thiện việc xác định kích thước đối tượng khi sử dụng __**builtin_object_size**, từ đó hỗ trợ các kiểm tra an toàn truy cập bộ nhớ.
- Ví dụ, khi gọi **malloc** hay **calloc**, trình biên dịch có thể biết trước kích thước của vùng nhớ được cấp phát dựa trên đối số truyền vào, giúp phát hiện lỗi nếu có truy cập vượt quá phạm vi bộ nhớ.

5.Ví dụ minh họa:

- **calloc_ptr:**

  typedef __attribute__ ((alloc_size (1, 2))) void*

  (*calloc_ptr) (size_t, size_t);

  Ở đây, khi gọi hàm thông qua **calloc_ptr**, kích thước đối tượng sẽ được tính bằng cách nhân đối số thứ nhất và thứ hai, giống như cách hoạt động của hàm **calloc**.

- **malloc_ptr:**

  typedef __attribute__ ((alloc_size (1))) void*

  (*malloc_ptr) (size_t);

  Ở đây, khi gọi hàm thông qua **malloc_ptr**, kích thước của đối tượng được xác định trực tiếp từ đối số thứ nhất, giống như hàm **malloc**.

Tóm Lại

- Thuộc tính **alloc_size** cung cấp thông tin cho trình biên dịch về kích thước bộ nhớ mà hàm cấp phát sẽ trả về.
- Nó không thực hiện cấp phát bộ nhớ hay tính toán kích thước, mà chỉ định nghĩa mối quan hệ giữa các đối số của hàm và kích thước của đối tượng được cấp phát.
- Thông tin này giúp GCC cải thiện việc xác định kích thước đối tượng, hỗ trợ kiểm tra an toàn và phát hiện lỗi trong quá trình biên dịch và runtime.

Như vậy, thuộc tính **alloc_size** là một công cụ mạnh mẽ để cung cấp thông tin về kích thước đối tượng cho các hàm cấp phát bộ nhớ, từ đó giúp trình biên dịch có thể tối ưu hóa và kiểm tra mã nguồn hiệu quả hơn.

cleanup(cleanup_function)

Thuộc tính cleanup trong GCC cho phép bạn chỉ định một hàm dọn dẹp (cleanup_function) sẽ được tự động gọi khi một biến ra khỏi phạm vi (out of scope). Điều này tương tự như việc sử dụng destructor trong C++, nhưng dành cho C.

Quy tắc sử dụng

- Chỉ áp dụng cho **biến cục bộ** (auto storage), không thể dùng với tham số hàm hoặc biến có static storage duration.
- Hàm cleanup_function phải nhận **một tham số** là **con trỏ** trỏ đến kiểu dữ liệu của biến cần dọn dẹp.
- Giá trị trả về của cleanup_function (nếu có) sẽ **bị bỏ qua**.
- Nếu nhiều biến trong cùng một phạm vi có thuộc tính cleanup, chúng sẽ được dọn dẹp theo **thứ tự ngược lại** với thứ tự khai báo (Last In, First Out - LIFO).
- Nếu -fexceptions được bật, hàm cleanup vẫn được gọi trong quá trình **stack unwinding** khi có ngoại lệ xảy ra.
  - **-fexceptions**
    - Đây là một **tùy chọn biên dịch của GCC** cho phép hỗ trợ **xử lý ngoại lệ (exception handling)** trong C++ và C (nếu có hỗ trợ).
    - Khi bật -fexceptions, GCC sẽ tạo **metadata** để theo dõi việc cấp phát tài nguyên và gọi **hàm dọn dẹp (cleanup function)** khi xảy ra ngoại lệ.

"Metadata" là thông tin mô tả về dữ liệu khác. Nó không phải là dữ liệu gốc mà là các thông tin bổ sung giúp xác định đặc điểm, cấu trúc, ngữ cảnh hoặc thuộc tính của dữ liệu đó. Dưới đây là một số điểm chính:

- **Mô tả dữ liệu:**
  Metadata cho biết dữ liệu chứa những thông tin gì, ví dụ như kích thước, kiểu dữ liệu, thời gian tạo, tác giả, v.v.
- **Ứng dụng trong lập trình và biên dịch:**
  Trong quá trình biên dịch, metadata có thể bao gồm thông tin về:
  - Kiểu dữ liệu, địa chỉ bộ nhớ của biến.
  - Thông tin về exception handling (xử lý ngoại lệ).
  - Các chỉ dẫn tối ưu hóa cho trình biên dịch.
  - Thông tin debug, như dấu hiệu của hàm, dòng lệnh, v.v.
- **Lợi ích:**
  Metadata giúp cho:
  - Trình biên dịch hiểu rõ hơn về cách xử lý dữ liệu.
  - Hệ thống quản lý tài nguyên tự động (như stack unwinding) hoạt động hiệu quả.
  - Các công cụ kiểm tra an toàn bộ nhớ và debug có thể đưa ra thông báo chính xác.

- **Metadata** là thông tin bổ sung mà trình biên dịch sinh ra hoặc thu thập từ mã nguồn (ví dụ: thông tin về kiểu dữ liệu, kích thước, thuộc tính như aligned hay alloc_size, v.v.)
- Các **function tích hợp** hoặc các cơ chế nội bộ của GCC (như __builtin_object_size, exception handling, stack unwinding, ...) sử dụng metadata này để đưa ra các kiểm tra an toàn, tối ưu hóa hoặc chẩn đoán lỗi.

Nói cách khác, metadata không phải là thứ mà lập trình viên trực tiếp "cung cấp" cho các hàm của GCC, mà là thông tin mà GCC tự động tổng hợp từ mã nguồn (nhờ vào các **attribute** và cú pháp khác) để hỗ trợ quá trình biên dịch và thực thi.

    - Nếu -fexceptions không được bật, một số hành vi xử lý ngoại lệ có thể không hoạt động chính xác, hoặc chương trình có thể bị kết thúc ngay lập tức mà không có cơ chế thu hồi tài nguyên.
  - **Stack Unwinding**
    - Khi một **ngoại lệ (exception) được ném ra** trong một khối mã, chương trình sẽ cần quay lại (unwind) để tìm **khối catch phù hợp** để xử lý ngoại lệ.
    - **Stack Unwinding** là quá trình:

1. **Từng bước quay lại (unwind) qua các stack frame**.
2. **Gọi các destructor** của đối tượng tự động (auto storage). (C++)
3. **Gọi các hàm cleanup (__attribute__((cleanup)))** cho các biến có gán thuộc tính này.
4. **Dừng lại tại catch phù hợp hoặc kết thúc chương trình** nếu không có handler (catch).

Ví dụ 1: Cleanup chạy trong quá trình stack unwinding

c — Sao chép — Chỉnh sửa

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

void cleanup_function(int *p) {
    printf("Cleaning up: %d\n", *p);
}

void function_with_exception() {
    int x __attribute__((cleanup(cleanup_function))) = 42;
    printf("Before throwing exception\n");
    throw "Error!";  // Ném một ngoại lệ
}

int main() {
    try {
        function_with_exception();
    } catch (const char* msg) {
        printf("Caught exception: %s\n", msg);
    }
    return 0;
}
```

Output:

sh — Sao chép — Chỉnh sửa

```
Before throwing exception
Cleaning up: 42
Caught exception: Error!
```

- Khi throw "Error!"; xảy ra, chương trình **unwind stack** để tìm khối catch.
- Trước khi thoát khỏi function_with_exception(), **cleanup_function được gọi để dọn dẹp biến x**.
- Sau đó, catch trong main() bắt ngoại lệ và in ra "Caught exception: Error!".

- Hành vi không xác định (undefined behavior) nếu cleanup_function không kết thúc bình thường (ví dụ: exit(), longjmp(), hoặc ném ngoại lệ).
  - Một **hàm kết thúc không bình thường** là hàm **bị dừng đột ngột** mà **không qua lệnh return thông thường**. Các nguyên nhân phổ biến:

a. **Ngoại lệ (throw) mà không có catch phù hợp**
b. **Gọi exit() hoặc _exit()**
c. **Gọi abort()**
d. **Gọi longjmp() để nhảy khỏi stack frame hiện tại**

Ví dụ 2: Cleanup bị bỏ qua khi `exit()`

c — Sao chép — Chỉnh sửa

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

void cleanup_function(int *p) {
    printf("Cleaning up: %d\n", *p);
}

void function_with_exit() {
    int x __attribute__((cleanup(cleanup_function))) = 42;
    printf("Before exit\n");
    exit(1);  // Kết thúc chương trình ngay lập tức
}

int main() {
    function_with_exit();
    printf("This will never be printed\n");
    return 0;
}
```

Output:

sh — Sao chép — Chỉnh sửa

```sh
Before exit
```

  - exit(1); **bỏ qua stack unwinding**, vì vậy **cleanup_function không được gọi**.
  - Chương trình kết thúc ngay lập tức, bỏ qua các lệnh sau exit(1).

Ví dụ sử dụng

Giả sử bạn muốn tự động đóng file khi biến quản lý file ra khỏi phạm vi:

Giả sử bạn muốn tự động đóng file khi biến quản lý file ra khỏi phạm vi:

c

Sao chép Chỉnh sửa

```c
#include <stdio.h>

void cleanup_file(FILE **fp) {
    if (fp && *fp) {
        printf("Closing file automatically\n");
        fclose(*fp);
    }
}

#define auto_cleanup __attribute__((cleanup(cleanup_file)))

int main() {
    auto_cleanup FILE *fp = fopen("example.txt", "w");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    fprintf(fp, "Hello, world!\n");

    // Không cần gọi fclose(fp), vì cleanup_file sẽ tự động đóng file khi fp ra khỏi scope
    return 0;
}
```

Diễn giải

- fp được khai báo với __attribute__((cleanup(cleanup_file))), nghĩa là khi fp ra khỏi phạm vi, hàm cleanup_file sẽ được gọi tự động.
- Hàm cleanup_file kiểm tra fp và gọi fclose(*fp), đảm bảo file được đóng đúng cách.

Lợi ích của cleanup

1. **Giảm rủi ro rò rỉ tài nguyên** (memory leak, file descriptor leak).
2. **Tăng tính an toàn** bằng cách tự động dọn dẹp tài nguyên khi biến ra khỏi phạm vi.
3. **Dễ đọc, dễ bảo trì** hơn so với việc phải nhớ gọi hàm cleanup thủ công.

Hạn chế

- **Không hoạt động với biến static hoặc global**.
- **Không thể bắt ngoại lệ**, chỉ có thể thực hiện hành động dọn dẹp.
- **Hành vi không xác định nếu cleanup_function không kết thúc bình thường**.

Khi nào nên sử dụng?

- Khi làm việc với **file, socket, mutex, bộ nhớ cấp phát động** để đảm bảo tài nguyên luôn được giải phóng đúng cách.
- Khi muốn **viết mã gọn gàng hơn**, tránh quên gọi các hàm cleanup thủ công.

always_inline

always_inline là một thuộc tính của GCC dùng để chỉ định rằng một hàm nên được chèn (inline) vào chỗ gọi nó, bỏ qua các hạn chế thông thường của quá trình inlining. Dưới đây là các điểm chính:

- **Mục đích:**
  - Ép buộc trình biên dịch chèn mã của hàm trực tiếp vào chỗ gọi, nhằm giảm thiểu chi phí gọi hàm và tối ưu hóa hiệu năng.
  - Đối với các hàm được khai báo inline, thuộc tính always_inline buộc hàm đó phải được chèn nội tuyến, bất kể các hạn chế thông thường của việc inlining.
- **Hành vi:**
  - Nếu trình biên dịch không thể chèn hàm được, sẽ báo lỗi.
  - Lưu ý rằng nếu hàm được gọi gián tiếp (indirect call), việc inlining có thể không được thực hiện tùy thuộc vào mức độ tối ưu hóa và trình biên dịch có thể không báo lỗi nếu không thể inlining trong trường hợp này.
    - Khi một hàm được gọi gián tiếp (indirect call), nghĩa là bạn không gọi trực tiếp theo tên hàm mà gọi thông qua con trỏ hàm. Trong trường hợp này, trình biên dịch không có đủ thông tin tĩnh (compile-time) để biết chính xác hàm nào sẽ được gọi, vì vậy việc chèn mã (inlining) có thể trở nên khó khăn hơn.

```c
c                                                    Sao chép  Chỉnh sửa

// Hàm luôn được inlined nếu gọi trực tiếp
__attribute__((always_inline)) inline void foo() {
    // ... code ...
}

// Gọi trực tiếp: inlining sẽ được thực hiện
foo();

// Gọi gián tiếp: qua con trỏ hàm, inlining có thể không xảy ra
void (*func_ptr)() = foo;
func_ptr();
```

    - Trong trường hợp gọi gián tiếp:
      - **Trình biên dịch không chắc chắn:** Vì con trỏ hàm có thể trỏ đến nhiều hàm khác nhau, trình biên dịch không thể đảm bảo rằng hàm nào sẽ được gọi lúc runtime.
      - **Tùy thuộc vào tối ưu hóa:** Khi tối ưu hóa mạnh, một số trình biên dịch có thể cố gắng theo dõi và inlining hàm ngay cả khi gọi gián tiếp nếu chúng có thể xác định được giá trị của con trỏ hàm, nhưng thường thì điều này không xảy ra.
      - **Không báo lỗi:** Nếu bạn đã khai báo hàm với always_inline nhưng gọi nó qua con trỏ hàm (gọi gián tiếp) và trình biên dịch không thể inlining, trình biên dịch có thể không báo lỗi. Thay vào đó, nó sẽ thực hiện gọi hàm theo cách thông thường (không inlined).
- **Điều kiện áp dụng:**
  - Thông thường chỉ có hiệu lực khi bật tối ưu hóa (optimization) trong quá trình biên dịch.

Tóm lại, always_inline là một chỉ thị mạnh mẽ cho trình biên dịch, buộc hàm phải được inlined tại nơi gọi, giúp cải thiện hiệu năng nhưng cũng đòi hỏi rằng các điều kiện để inlining được đáp ứng, đặc

biệt là trong các trường hợp gọi trực tiếp.

noinline

Thuộc tính **noinline** được dùng để ngăn trình biên dịch xét đến việc inlining (chèn trực tiếp nội dung của hàm vào chỗ gọi) cho một hàm nào đó. Điều này có nghĩa là, khi một hàm được đánh dấu là noinline, GCC sẽ không thực hiện inlining cho hàm đó, và một số tối ưu hóa liên ngành (interprocedural optimizations) cũng bị vô hiệu hóa.

Chi tiết:

- **Ngăn inlining:**
  Hàm được đánh dấu với noinline sẽ không bị chèn nội tuyến, cho dù có tối ưu hóa cao hay không. Điều này có ích khi bạn muốn giữ nguyên lời gọi hàm trong binary để dễ theo dõi hoặc debug.
- **Tối ưu hóa liên ngành:**
  Noinline còn vô hiệu hóa một số tối ưu hóa giữa các hàm. Tuy nhiên, nếu mục tiêu của bạn là vô hiệu hóa tất cả các tối ưu hóa liên ngành, thì nên dùng thuộc tính **noipa** (no interprocedural analysis) thay thế, vì noipa bao phủ nhiều khía cạnh tối ưu hóa hơn.
- **Các tối ưu hóa khác có thể loại bỏ lời gọi hàm:**
  Ngay cả khi một hàm được đánh dấu noinline, nếu hàm đó không có tác dụng phụ (side effects), các tối ưu hóa khác (không phải inlining) có thể làm cho lời gọi hàm bị loại bỏ (tức là bị tối ưu đi) nếu kết quả của hàm không được sử dụng.
  - Khi một hàm không có tác dụng phụ, mặc dù bạn đã ngăn inlining bằng noinline, GCC có thể áp dụng một số tối ưu hóa khác để loại bỏ hoặc rút gọn lời gọi hàm nếu giá trị trả về của nó không ảnh hưởng đến hành vi observable của chương trình. Một số tối ưu hóa điển hình bao gồm:
    i. **Loại bỏ mã chết (Dead Code Elimination):**
    Nếu kết quả trả về của hàm không được sử dụng và hàm không có tác dụng phụ, compiler có thể hoàn toàn loại bỏ lời gọi hàm đó khỏi mã cuối cùng.
    ii. **Loại bỏ lời gọi hàm (Call Elision):**
    Nếu compiler có thể xác định rằng lời gọi hàm không thay đổi trạng thái của chương trình (vì không có side effects) và giá trị trả về có thể được tính sẵn, nó có thể thay thế lời gọi bằng giá trị đã tính, hoặc thậm chí bỏ qua lời gọi hoàn toàn.
    iii. **Elimination of redundant calls (Common Subexpression Elimination):**
    Nếu cùng một hàm được gọi nhiều lần với cùng các đối số, compiler có thể tính toán kết quả một lần và tái sử dụng kết quả đó cho các lần gọi sau thay vì gọi hàm nhiều lần.
    iv. **Constant Propagation:**
    Nếu đầu vào của hàm là hằng số và kết quả có thể tính được tại thời điểm biên dịch, compiler có thể thay thế lời gọi hàm bằng kết quả hằng số đó.
- Để tránh trường hợp này và đảm bảo rằng lời gọi hàm luôn tồn tại (không bị tối ưu đi), bạn có thể thêm một câu lệnh:

  asm ("");

- vào bên trong hàm đó. Câu lệnh asm ("") được xem như một tác dụng phụ đặc biệt, buộc cho trình biên dịch giữ lại lời gọi hàm.

cold

Thuộc tính **cold** dùng để thông báo cho trình biên dịch rằng hàm được đánh dấu là không được thực thi thường xuyên (ít khi được gọi). Dưới đây là các điểm chính:

- **Mục đích:**
  - Thông báo cho trình biên dịch rằng hàm này "lạnh" (cold), tức là ít khi xảy ra.
  - Giúp tối ưu hóa mã nguồn cho kích thước (size) thay vì tốc độ (speed).
- **Tác động đến cấu trúc mã:**
  - Trên nhiều kiến trúc, các hàm được đánh dấu cold sẽ được đặt vào một phân đoạn (subsection) riêng trong phần text, giúp các phần "nóng" (hot) của chương trình được tập trung lại, cải thiện tính địa phương (locality) của mã.
  - Các nhánh dẫn đến các hàm cold được đánh dấu là ít xảy ra bởi cơ chế dự đoán nhánh (branch prediction).
- **Ứng dụng:**
  - Thích hợp để đánh dấu các hàm xử lý các tình huống hiếm khi xảy ra, như hàm perror hoặc các hàm xử lý lỗi đặc biệt.
  - Trong C++, thuộc tính cold cũng có thể áp dụng cho các kiểu dữ liệu (types) và được truyền xuống cho các thành viên (member functions).
- **Tương tác với profile feedback:**
  - Khi sử dụng tùy chọn -fprofile-use (phản hồi từ profiling), các hàm cold có thể được phát hiện tự động và thuộc tính cold sẽ bị bỏ qua.

Tóm lại, **cold** là một chỉ thị tối ưu hóa cho biết hàm không thường được thực thi, giúp trình biên dịch tối ưu hóa các phần hot của chương trình và sắp xếp lại mã để cải thiện hiệu suất dựa trên dự đoán của nhánh.

hot

huộc tính hot giúp trình biên dịch biết rằng một hàm được gọi thường xuyên (hot spot) và cần được tối ưu hóa mạnh hơn để tăng hiệu suất.

Tác dụng chính của hot

1. **Tăng cường tối ưu hóa**:
   - Trình biên dịch sẽ ưu tiên tối ưu hóa hàm này mạnh hơn so với các hàm khác.
   - Có thể áp dụng các chiến lược tối ưu hóa như **loop unrolling**, **function inlining**, **branch prediction**, v.v.
2. **Cải thiện locality (tính cục bộ của bộ nhớ)**:
   - Trên nhiều nền tảng, các hàm có hot sẽ được đặt gần nhau trong **text section** của binary.
   - Điều này giúp CPU tận dụng cache hiệu quả hơn, giảm cache miss khi chạy chương trình.

3. **Tự động bỏ qua khi dùng -fprofile-use**:
    - Nếu bạn sử dụng **Profile-Guided Optimization (PGO)** với -fprofile-use, GCC sẽ **tự động xác định các hot function** dựa trên dữ liệu thực thi, nên hot sẽ bị bỏ qua.

Ví dụ sử dụng hot:

```
__attribute__((hot))
void frequently_used_function() {
    // Code được gọi nhiều lần trong chương trình
}
```

- Trình biên dịch sẽ tối ưu hóa frequently_used_function() mạnh hơn.

Khi nào dùng hot?

- Khi bạn **biết chắc chắn** rằng một hàm là "hot spot" và được gọi nhiều lần trong luồng xử lý chính.
- Khi không sử dụng -fprofile-use, bạn có thể gán hot để đảm bảo GCC tối ưu hóa ngay từ đầu.

Ví dụ trong Linux Kernel:

- Các hàm xử lý ngắt, lịch CPU (scheduler), driver I/O quan trọng thường được đánh dấu là hot để cải thiện hiệu suất.

const

Thuộc tính **const** trong GCC dùng để đánh dấu rằng **một hàm có đặc tính "hằng số",** nghĩa là:

- **Kết quả của hàm chỉ phụ thuộc vào đối số truyền vào:**
  Hàm không đọc hoặc thay đổi bất kỳ đối tượng nào có thể ảnh hưởng đến giá trị trả về giữa các lần gọi. Nói cách khác, nếu gọi hàm với cùng một đối số, kết quả trả về luôn giống nhau, bất kể các lệnh khác giữa các lần gọi.
- **Không có tác dụng phụ (side effects):**
  Hàm chỉ trả về giá trị và không có bất kỳ hiệu ứng nào đến trạng thái "observable" (các biến, I/O,...) của chương trình.
- **Lợi ích cho tối ưu hóa:**
  Vì kết quả của hàm luôn giống nhau với cùng một đối số, trình biên dịch có thể thực hiện các tối ưu hóa như loại bỏ các biểu thức chung (common subexpression elimination) bằng cách thay thế các lần gọi hàm lặp lại bằng kết quả đã tính từ lần gọi đầu tiên.

Ví dụ:

int square (int) __attribute__ ((const));

Điều này cho biết rằng các lần gọi hàm square với cùng một giá trị đối số có thể được thay thế bằng kết quả đã tính trước đó, giúp giảm số lần thực hiện hàm không cần thiết.

- **Giới hạn và quy định:**
  - Hàm được đánh dấu với **const** không được phép đọc các đối tượng có thể thay đổi giá trị giữa các lần gọi (ví dụ: biến toàn cục thay đổi, biến thông qua con trỏ, ...).
    - Nếu xảy ra **có thể sẽ dẫn đến tối ưu hóa sai ví dụ:**

```c
#include <stdio.h>

int global = 5;

// Hàm myfunc được đánh dấu là const, tuy nhiên nó lại sử dụng biến toàn cục 'global'
int myfunc(int x) __attribute__((const));
int myfunc(int x) {
    // Kết quả phụ thuộc vào 'global' nên nếu 'global' thay đổi,
    // kết quả của myfunc(3) cũng nên thay đổi.
    return x * x + global;
}

int main(void) {
    int a = myfunc(3);  // Dự kiến: 3*3 + 5 = 14
    // Thay đổi giá trị của biến toàn cục sau lần gọi đầu tiên.
    global = 10;
    int b = myfunc(3);  // Dự kiến: 3*3 + 10 = 19

    // Nếu GCC tối ưu hóa dựa trên giả định hàm const, nó có thể
    // tối ưu hóa thành: b = a (vì myfunc(3) được cho là không thay đổi).
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

    - **Kết quả tối ưu hóa sai (nếu xảy ra):**
      Trình biên dịch có thể thay thế lần gọi thứ hai bằng kết quả của lần gọi đầu tiên, làm cho cả a và b đều bằng 14, gây ra lỗi về mặt logic trong chương trình.
  - Vì hàm không có tác dụng phụ, nên nó không có ý nghĩa khi trả về void.
  - Nếu hàm có đối số là con trỏ hoặc tham chiếu (trong C++), và dữ liệu được trỏ tới có thể thay đổi, thì không nên khai báo hàm đó là **const** vì kết quả có thể không ổn định giữa các lần gọi.
  - So với thuộc tính **pure**, **const** đặt ra các hạn chế chặt chẽ hơn về việc không được phép đọc bất kỳ dữ liệu nào có thể ảnh hưởng đến giá trị trả về giữa các lần gọi.

constructor

destructor

constructor (priority)

destructor (priority)

Thuộc tính **constructor** và **destructor** trong GCC được dùng để chỉ định các hàm được gọi tự động vào những thời điểm đặc biệt trong vòng đời của chương trình:

- **constructor**
  - **Mục đích:**
    - Hàm đánh dấu với **constructor** sẽ được gọi tự động trước khi hàm main() bắt đầu thực thi.
    - Thường được sử dụng để khởi tạo các dữ liệu hoặc tài nguyên cần thiết cho chương trình.
- **destructor**
  - **Mục đích:**
    - Hàm đánh dấu với **destructor** sẽ được gọi tự động sau khi hàm main() kết thúc hoặc khi hàm exit() được gọi.
    - Dùng để dọn dẹp, giải phóng tài nguyên đã được cấp phát trong quá trình chạy chương trình.
- **constructor(priority) và destructor(priority):**
  - **Mục đích thêm:**
    - Trên một số kiến trúc, bạn có thể chỉ định một số ưu tiên (priority) cho các hàm constructor hoặc destructor để kiểm soát thứ tự thực thi của chúng.
  - **Cách hoạt động:**
    - **Constructor:**
      - Một hàm constructor có số ưu tiên nhỏ hơn sẽ được gọi **trước** hàm constructor có số ưu tiên lớn hơn.
    - **Destructor:**
      - Ngược lại, một hàm destructor có số ưu tiên nhỏ hơn sẽ được gọi **sau** hàm destructor có số ưu tiên lớn hơn.
  - **Lưu ý:**
    - Các giá trị ưu tiên từ **0 đến 100** được dành riêng (reserved).
    - Nếu bạn có một hàm constructor để cấp phát tài nguyên và một hàm destructor để giải phóng tài nguyên đó, chúng thường nên có cùng mức ưu tiên.
    - Với C++ và các đối tượng có static storage duration, thứ tự khởi tạo của các đối tượng và hàm constructor có thể không được xác định rõ ràng. Để áp đặt thứ tự cụ thể, bạn có thể sử dụng thuộc tính init_priority trong C++.
    - Nếu target không hỗ trợ dạng có đối số cho constructor hoặc destructor, việc sử dụng dạng này sẽ dẫn đến lỗi.

Sao chép Chỉnh sửa

```c
#include <stdio.h>

// Hàm constructor với ưu tiên 150 (ưu tiên cao hơn so với giá trị lớn hơn)
void init_resource(void) __attribute__((constructor(150)));
void init_resource(void) {
    printf("Khởi tạo tài nguyên\n");
}

// Hàm constructor khác với ưu tiên 200
void init_other(void) __attribute__((constructor(200)));
void init_other(void) {
    printf("Khởi tạo tài nguyên khác\n");
}

// Hàm destructor với ưu tiên 150 (sẽ được gọi sau destructor có ưu tiên lớn hơn)
void cleanup_resource(void) __attribute__((destructor(150)));
void cleanup_resource(void) {
    printf("Giải phóng tài nguyên\n");
}

// Hàm destructor khác với ưu tiên 200
void cleanup_other(void) __attribute__((destructor(200)));
void cleanup_other(void) {
    printf("Giải phóng tài nguyên khác\n");
}

int main(void) {
    printf("Chương trình đang chạy...\n");
    return 0;
}
```

returns_nonnull

Thuộc tính **returns_nonnull** chỉ định rằng giá trị trả về của hàm là một con trỏ không bao giờ bằng NULL. Điều này giúp trình biên dịch có thể tối ưu hóa các đoạn mã gọi hàm đó, bởi vì nó biết rằng không cần phải kiểm tra giá trị trả về có phải là NULL hay không.

Ví dụ:

```c
extern void *
mymalloc (size_t len) __attribute__((returns_nonnull));
```

- Ở đây, hàm mymalloc được khai báo có thuộc tính returns_nonnull, có nghĩa là bất kỳ giá trị nào mà hàm trả về luôn là con trỏ hợp lệ, không bao giờ là NULL.
- Với thông tin này, trình biên dịch có thể bỏ qua các kiểm tra lỗi liên quan đến việc kiểm tra con trỏ NULL sau khi gọi hàm, từ đó có thể tạo ra mã máy tối ưu hơn.

Tóm lại, **returns_nonnull** giúp cải thiện hiệu năng thông qua việc tối ưu hóa các đoạn mã dựa trên giả định rằng hàm luôn trả về con trỏ không NULL.

flatten

Thuộc tính **flatten** được dùng để buộc trình biên dịch inlining tất cả các lời gọi hàm bên trong một hàm được đánh dấu, nếu có thể. Điều này có nghĩa là:

- **Tác động đối với các lời gọi bên trong:**
  Khi một hàm được gắn với flatten, mọi lời gọi hàm bên trong (bao gồm cả những lời gọi được thêm vào bởi việc inlining vào hàm đó) đều sẽ được chèn trực tiếp (inline) vào trong hàm, giúp giảm chi phí gọi hàm và cải thiện hiệu suất.
  - Lưu ý: Các lời gọi đệ quy (recursive) đến chính hàm đó sẽ không được inlined.
- **Ngoại lệ:**
  Các hàm được khai báo với noinline hoặc các thuộc tính tương tự sẽ không bị inlined, bất kể có thuộc tính flatten hay không.
- **Quyết định về việc inlining:**
  Dù hàm được đánh dấu flatten, **việc inlining chính hàm đó vào nơi gọi - nơi gọi ở đây là một hàm lớn hơn call đến hàm flatten** (chứ không chỉ các lời gọi bên trong) phụ thuộc vào kích thước của hàm và các tham số tối ưu hóa hiện tại của trình biên dịch.

Tóm lại, **flatten** giúp mở rộng inlining nội bộ bên trong hàm, cố gắng inlining tất cả các lời gọi bên trong để tối ưu hóa hiệu suất, tuy nhiên, các lời gọi đến hàm với noinline và các yếu tố như kích thước hàm vẫn có thể ảnh hưởng đến việc thực hiện inlining.

target (string, ...)

Attribute target(...) cho phép biên dịch một hàm với các tùy chọn kiến trúc khác với mặc định của chương trình. Nó đặc biệt hữu ích khi cần tối ưu một số phần của code cho các tập lệnh CPU khác nhau mà không ảnh hưởng đến toàn bộ chương trình.

1. Cách hoạt động:

• Khi một hàm có attribute target(...), trình biên dịch sẽ bỏ qua các tùy chọn target mặc định từ dòng lệnh và sử dụng các tùy chọn được chỉ định trong attribute đó.

• Các tùy chọn này thường là các phần mở rộng tập lệnh (ISA) như sse4.1, avx2, hoặc các tùy chọn kiến trúc CPU cụ thể như arch=core2.

Ở đây:

```c
#include <stdio.h>

int core2_func (void) __attribute__ ((__target__ ("arch=core2")));
int sse3_func (void) __attribute__ ((__target__ ("sse3")));

int core2_func(void) {
    return 42; // Chỉ biên dịch với -march=core2
}

int sse3_func(void) {
    return 84; // Chỉ biên dịch với -msse3
}

int main() {
    printf("core2_func: %d\n", core2_func());
    printf("sse3_func: %d\n", sse3_func());
    return 0;
}
```

• core2_func() được biên dịch với -march=core2, sử dụng các tối ưu hóa dành riêng cho CPU Core 2.

• sse3_func() được biên dịch với -msse3, đảm bảo rằng nó tận dụng được tập lệnh SSE3.

2. Lưu ý quan trọng:

• Chương trình chỉ có thể gọi các hàm này trên hệ thống hỗ trợ các tập lệnh tương ứng.

• Trên x86, cần kiểm tra hỗ trợ phần cứng trước khi gọi, thường thông qua cpuid.

• Không thể sử dụng target(...) để thay đổi ABI (Application Binary Interface), vì điều đó có thể gây lỗi tương thích.

3. Ứng dụng:

• Tối ưu hiệu năng trên các CPU khác nhau mà không cần biên dịch toàn bộ chương trình cho mỗi kiến trúc.

• Viết thư viện có thể chạy trên nhiều loại CPU, với các hàm được tối ưu riêng cho từng tập lệnh.

target_clone (..., ...)

Attribute target_clones cho phép GCC tạo ra nhiều phiên bản của cùng một hàm, mỗi phiên bản được biên dịch với các tùy chọn kiến trúc (ISA) khác nhau. Khi chương trình chạy, một **resolver function** sẽ tự động chọn phiên bản phù hợp nhất với phần cứng hiện tại.

Cách hoạt động:

- GCC tạo **nhiều bản sao** của một hàm (clones), mỗi bản được tối ưu hóa theo một tập lệnh CPU khác nhau.
- Một **resolver function** (tương tự như ifunc) sẽ **chọn** phiên bản tối ưu nhất khi chương trình chạy.
- Nếu CPU không hỗ trợ tập lệnh nào, phiên bản mặc định của hàm sẽ được sử dụng.

Ví dụ:

```
#include <stdio.h>

__attribute__((target_clones("default,sse4.1,avx2")))
int compute(int x) {
    return x * x;
}

int main() {
    printf("compute(5) = %d\n", compute(5));
    return 0;
}
```

Phân tích:

- GCC tạo **3 phiên bản** của compute():
  - default: Biên dịch với tùy chọn mặc định.
  - sse4.1: Tận dụng tập lệnh SSE4.1 nếu CPU hỗ trợ.
  - avx2: Dùng AVX2 nếu có.
- Khi chương trình chạy, **resolver function** sẽ chọn phiên bản phù hợp với CPU hiện tại.

Lưu ý quan trọng:

1. **Cần hỗ trợ ifunc**:

    - target_clones **phụ thuộc vào cơ chế ifunc** của linker để chọn hàm phù hợp tại runtime.
    - Yêu cầu **glibc 2.23+** để hoạt động.
2. **Hàm được clone phải được gọi ít nhất một lần**:
    - Nếu không có lời gọi nào đến hàm, GCC có thể bỏ qua việc tạo resolver function.
3. **Hàm được gọi từ một hàm có target_clones không tự động bị clone theo**:
    - Nếu muốn các hàm con cũng được clone, cần dùng **flatten** để buộc inline toàn bộ.

So sánh với target(...)

Attribute	Cách hoạt động
target(...)	Một hàm, biên dịch với tùy chọn CPU khác mặc định.
target_clones(...)	Tạo nhiều bản sao của hàm, chọn bản phù hợp khi runtime.

Ứng dụng thực tế:

- **Tối ưu hiệu năng trên nhiều kiến trúc CPU** mà không cần biên dịch riêng cho từng kiến trúc.
- **Viết thư viện hiệu suất cao** có thể tận dụng các tập lệnh mở rộng như AVX, SSE, NEON trên ARM...
- **Giảm công sức bảo trì** so với việc phải viết nhiều hàm riêng biệt cho từng CPU.

pure (..., ...)

Attribute pure cho trình biên dịch biết rằng **hàm không thay đổi trạng thái có thể quan sát được của chương trình**, ngoài việc trả về một giá trị. Điều này giúp trình biên dịch tối ưu hóa mạnh hơn, như **loại bỏ các lời gọi lặp lại với cùng tham số** (common subexpression elimination - CSE).

Quy tắc của pure:

- **Được phép đọc** các biến **không phải volatile**, kể cả biến toàn cục.
- **Không được ghi** vào bất kỳ biến nào **có thể ảnh hưởng đến chương trình** ngoài việc trả về giá trị.

- **Không có tác dụng phụ (side effects)** như thay đổi biến toàn cục, thao tác I/O, hoặc sử dụng volatile.
- **Không được khai báo đồng thời với const**, vì const có ràng buộc chặt hơn.

```c
int hash(const char *str) __attribute__((pure));

int hash(const char *str) {
    int h = 0;
    while (*str) {
        h = h * 31 + *str++;
    }
    return h;
}
```

📌 **Tại sao hợp lệ?**

- Không ghi vào biến toàn cục.
- Chỉ đọc từ str (không phải volatile).
- Có thể gọi lại với cùng tham số mà không làm thay đổi chương trình.

Hàm KHÔNG thể là pure

```c
int counter = 0;

int next_id() __attribute__((pure));  // ❌ Sai!

int next_id() {
    return counter++; // Ghi vào biến toàn cục
}
```

📌 **Tại sao sai?**

- Hàm thay đổi giá trị của counter, làm thay đổi trạng thái chương trình.

So sánh pure và const

Attribute	Được đọc	Được ghi	Ví dụ
pure	**Bất kỳ biến không volatile**	❌ Không được ghi vào biến toàn cục hoặc có thể ảnh hưởng đến chương trình	strlen, memcmp, hash
const	**Chỉ đọc tham số hoặc biến static const**	❌ Không được ghi vào biến nào	sqrt, abs, sin

📌 **Lưu ý:**

- pure có thể đọc **biến toàn cục không đổi**, còn const chỉ có thể đọc tham số.
- pure **có thể nhận con trỏ** (char *), nhưng **không được thay đổi dữ liệu bên trong**.

volatile trong C

volatile là một **qualifier** (bổ ngữ) được dùng khi khai báo biến để báo cho trình biên dịch rằng **giá trị của biến có thể thay đổi bất ngờ ngoài tầm kiểm soát của chương trình**.

1. Khi nào dùng volatile?

volatile thường dùng cho các biến có thể thay đổi bởi:
✅ **Phần cứng (Hardware Registers)** → Khi đọc giá trị từ một thanh ghi của thiết bị ngoại vi.
✅ **Ngắt (Interrupts/ISR - Interrupt Service Routine)** → Khi biến có thể bị thay đổi bởi một ngắt.
✅ **Đa luồng (Multithreading/Shared Memory)** → Khi biến được chia sẻ giữa các thread.

2. Ví dụ volatile trong thực tế

(1) Thanh ghi phần cứng

```
#define STATUS_REG  (*(volatile unsigned int *) 0x4000)

void wait_for_ready() {
    while (STATUS_REG & 0x01 == 0);  // Chờ bit sẵn sàng
}
```

📌 **Tại sao cần volatile?**

- Nếu không có volatile, trình biên dịch có thể **tối ưu hóa vòng lặp** bằng cách đọc STATUS_REG **một lần**, khiến vòng lặp trở thành vô hạn.
- volatile đảm bảo STATUS_REG **luôn được đọc trực tiếp từ phần cứng**.

(2) Biến bị thay đổi trong ISR (Interrupt Service Routine)

```
volatile int flag = 0;

void ISR() {  // ISR: Interrupt Service Routine
    flag = 1;  // Đánh dấu đã có ngắt
}

void loop() {
    while (flag == 0);  // Chờ ngắt xảy ra
    // Thực hiện xử lý khi flag = 1
}
```

📌 **Tại sao cần volatile?**

- Nếu không có volatile, trình biên dịch có thể **bỏ qua vòng lặp while(flag == 0);** vì nó nghĩ rằng flag không bao giờ thay đổi.

(3) Biến dùng trong đa luồng

```
volatile int shared_var = 0;

void *thread_func(void *arg) {
    shared_var = 1;  // Ghi giá trị từ một thread khác
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);

    while (shared_var == 0);  // Chờ giá trị thay đổi từ thread khác
    printf("shared_var changed!\n");
}
```

📌 **Tại sao cần volatile?**

- Nếu không có volatile, trình biên dịch có thể **bỏ qua** vòng lặp while(shared_var == 0); vì nó nghĩ shared_var không bao giờ thay đổi.

3. volatile KHÔNG đảm bảo an toàn trong đa luồng

🔴 volatile **không** thay thế mutex, atomic hoặc các cơ chế đồng bộ khác.

- Nó **chỉ đảm bảo rằng trình biên dịch không tối ưu hóa việc đọc/ghi biến**, nhưng **không đảm bảo tính nhất quán dữ liệu khi có nhiều luồng truy cập cùng lúc**.
- Trong đa luồng, **cần dùng atomic hoặc mutex thay vì chỉ volatile**.

Ví dụ **không an toàn** dù có volatile:

```
volatile int counter = 0;

void *increment() {
    for (int i = 0; i < 10000; i++) {
        counter++;  // Không an toàn trong đa luồng!
    }
}
```

📌 **Tại sao sai?**

- counter++ không phải là **atomic operation** → Hai thread có thể đọc cùng một giá trị, rồi cả hai ghi giá trị mới, dẫn đến **race condition**.

✅ **Giải pháp đúng**: Dùng atomic

```
#include <stdatomic.h>

atomic_int counter = 0;

void *increment() {
    for (int i = 0; i < 10000; i++) {
        atomic_fetch_add(&counter, 1);
    }
}
```

4. volatile có ảnh hưởng đến tốc độ không?

✅ **Có!** Vì trình biên dịch **không thể tối ưu hóa** các biến volatile, nó sẽ luôn đọc trực tiếp từ bộ nhớ **thay vì cache**.
🔴 Dùng volatile không đúng chỗ có thể **làm chậm chương trình** vì mất tối ưu hóa.

5. So sánh volatile và const

Tính chất	volatile	const
Bảo vệ dữ liệu	❌ Không	✅ Có
Tránh tối ưu hóa	✅ Có	❌ Không
Thay đổi được không?	✅ Có	❌ Không
Dùng trong đa luồng?	❌ Không an toàn	✅ Có thể nhưng cần thêm đồng bộ

✅ **Có thể kết hợp volatile và const**

```
const volatile int sensor_value = 0x4000;
```

- const: Không thể thay đổi sensor_value trong code.
- volatile: Trình biên dịch vẫn phải **đọc lại giá trị từ thanh ghi phần cứng mỗi lần**.

noreturn

noreturn là một **function attribute** giúp trình biên dịch biết rằng **hàm sẽ không bao giờ trả về (Không bao giờ quay lại nơi gọi nó)**. Điều này giúp tối ưu hóa mã máy và tránh cảnh báo không cần thiết.

Thông thường hàm sẽ luôn trả về kể cả kiểu void.

1. Cách sử dụng noreturn

Dùng __attribute__((noreturn)) để khai báo một hàm không bao giờ quay lại nơi gọi:

```
#include <stdlib.h>

void fatal_error(const char *msg) __attribute__((noreturn));

void fatal_error(const char *msg) {
    printf("ERROR: %s\n", msg);
    exit(1);  // Không bao giờ quay lại nơi gọi
}
```

✅ **Trình biên dịch hiểu rằng fatal_error() không bao giờ return**, nên:

- Không cần generate mã để xử lý trường hợp fatal_error() quay lại.
- Không cảnh báo về biến chưa khởi tạo nếu chương trình sẽ thoát trước khi sử dụng.

2. Lợi ích của noreturn

(1) Tối ưu hóa mã máy

- Khi trình biên dịch biết một hàm không bao giờ quay lại, nó **có thể bỏ qua việc xử lý return**.
- **Không lưu trạng thái CPU hoặc thanh ghi** để quay lại caller.

(2) Tránh cảnh báo biến chưa khởi tạo

```
int compute(int x) {
    if (x < 0)
        fatal_error("x must be non-negative!");

    int y;
    return y + 10;  // ⚠️ Cảnh báo nếu `fatal_error()` không có `noreturn`
}
```

📌 **Nếu fatal_error() không có noreturn, GCC sẽ cảnh báo "y có thể chưa được khởi tạo"**.
📌 **Với noreturn, trình biên dịch hiểu rằng fatal_error() sẽ kết thúc chương trình, nên không báo lỗi.**

3. noreturn KHÔNG cấm ném exception hoặc longjmp

noreturn chỉ báo hiệu **hàm không return theo cách thông thường**. Nó vẫn có thể:

- **Ném exception (throw trong C++)**
- **Nhảy đến một điểm khác trong chương trình (longjmp)**

```
#include <setjmp.h>

jmp_buf env;

void jump_somewhere() __attribute__((noreturn));

void jump_somewhere() {
    longjmp(env, 1);  // Nhảy về điểm `setjmp()`, không return bình thường
}
```

📌 **Mặc dù jump_somewhere() không return, nó vẫn quay lại setjmp() theo cách khác.**

4. GCC KHÔNG chuyển noreturn thành tail call

Thông thường, nếu một hàm chỉ gọi một hàm khác rồi return, GCC có thể tối ưu hóa thành **tail call optimization** (TCO).

📌 **Với noreturn, GCC KHÔNG làm vậy để bảo toàn backtrace (chuỗi gọi hàm khi debug).**

```
void die() __attribute__((noreturn));

void die() {
    abort();  // GCC không tối ưu hóa thành tail call
}
```

📌 **GCC vẫn gọi abort() như bình thường để không mất stack trace khi debug.**

📌 Stack Frame & Stack Trace là gì?

1️⃣ Stack Frame là gì?

🔹 Mỗi lần một hàm được gọi, nó cần một **khoảng không gian trên stack** để lưu trữ thông tin cần thiết như:

- **Địa chỉ trả về** (nơi chương trình quay lại sau khi hàm kết thúc).
- **Các biến cục bộ** của hàm.
- **Các tham số truyền vào hàm**.
- **Các giá trị tạm thời** phục vụ tính toán.

📌 **Khu vực bộ nhớ này trên stack được gọi là *Stack Frame của hàm đó*.**

Ví dụ:

Giả sử ta có chương trình này:

```c
void funcB() {
    int y = 20;  // Biến cục bộ
}

void funcA() {
    int x = 10;  // Biến cục bộ
    funcB();     // Gọi funcB
}

int main() {
    funcA();     // Gọi funcA
}
```

Khi chương trình chạy, stack sẽ như sau:

```mathematica
| Stack Frame của funcB |
| --------------------- |
| y = 20                |
| Return Address (funcA)|
-------------------------
| Stack Frame của funcA |
| --------------------- |
| x = 10                |
| Return Address (main) |
-------------------------
| Stack Frame của main  |
```

📌 Khi `funcB()` kết thúc, stack frame của nó bị xóa và chương trình quay về `funcA()`.

2️⃣ Stack Trace là gì?

🔹 **Stack Trace** là danh sách các stack frame từ hàm đang chạy cho đến main(), giúp ta biết chương trình đã gọi qua những hàm nào trước khi lỗi xảy ra.

📌 **Khi chương trình bị crash hoặc ta gỡ lỗi bằng GDB, stack trace cho ta thấy chuỗi các lời gọi hàm.**

Ví dụ thực tế

Chạy chương trình sau:

c — Sao chép — Chỉnh sửa

```c
#include <stdio.h>

void funcC() {
    int *ptr = NULL;
    *ptr = 42;  // 💥 Lỗi Segmentation Fault!
}

void funcB() { funcC(); }
void funcA() { funcB(); }

int main() {
    funcA();
    return 0;
}
```

💡 **GDB Stack Trace khi chương trình crash:**

shell — Sao chép — Chỉnh sửa

```
(gdb) backtrace
#0  funcC () at example.c:6
#1  funcB () at example.c:10
#2  funcA () at example.c:13
#3  main () at example.c:17
```

📌 **Stack Trace này cho thấy:**

1. `funcC()` **bị lỗi** (Segmentation Fault xảy ra ở đây).
2. `funcC()` được gọi bởi `funcB()`.
3. `funcB()` được gọi bởi `funcA()`.
4. `funcA()` được gọi bởi `main()`.

📌Tail Call là gì?

> **Tail Call là một lời gọi hàm xảy ra ở vị trí cuối cùng trong một hàm**, nghĩa là **không có thao tác nào khác được thực hiện sau lời gọi đó**.
>
> Nếu một hàm **gọi chính nó** ở vị trí cuối cùng, nó được gọi là **Tail Recursion** (đệ quy đuôi).

1️⃣ Tail Call thông thường

🔹 GCC có thể tối ưu hóa Tail Call như sau:

Ví dụ một hàm gọi exit():

```c
void quit(int code) {
    exit(code);  // Đây là Tail Call
}
```

📌 Lý do nó là Tail Call:

- exit(code) là **lệnh cuối cùng** trong quit().
- GCC có thể **tối ưu hóa bằng cách nhảy trực tiếp vào exit()** mà không cần giữ stack frame của quit() (Tail Call Optimization - TCO).

💡 **Khi TCO xảy ra:**

- quit() không cần stack frame riêng.
- Trình biên dịch thay thế quit() bằng một lệnh nhảy (jmp exit), tiết kiệm stack.

2️⃣ Tail Call với noreturn

🔹 **Khi một hàm có noreturn, GCC không thực hiện Tail Call Optimization.**

```
void fatal_error() __attribute__((noreturn));

void quit(int code) {
    fatal_error();  // ✅ Đây có thể là Tail Call, nhưng vì fatal_error() có noreturn, GCC sẽ
}
```

📌 **Điều gì xảy ra?**

- fatal_error() **không bao giờ quay lại nơi gọi** vì nó có noreturn.
- **GCC không thay call fatal_error() bằng jmp fatal_error**, tức là **nó vẫn tạo stack frame cho quit()**, dù quit() sẽ không bao giờ kết thúc bình thường.

5. noreturn và kiểu trả về của hàm

✅ **Hàm noreturn phải có kiểu void**

```
void exit_with_error() __attribute__((noreturn));  // ✅ Hợp lệ
```

🔴 **Không hợp lệ nếu có kiểu trả về khác void**

```
int fatal_error() __attribute__((noreturn));  // ⚠️ Sai! Hàm `noreturn` không thể return int
```

6. Tóm tắt

Tính chất	Mô tả
Chức năng	Báo cho GCC biết hàm không bao giờ return
Lợi ích	Tối ưu mã máy, tránh cảnh báo không cần thiết
Có ảnh hưởng đến tail call không?	✅ Có, GCC sẽ **không** chuyển thành tail call để bảo toàn backtrace
Có thể ném exception (throw) không?	✅ Có
Có thể dùng longjmp không?	✅ Có
Có thể trả về giá trị không?	❌ Không, chỉ có thể dùng với void

📌Khi nào Linux sử dụng noreturn?

Loại hàm	Ví dụ	Mục đích
Hàm dừng kernel	panic(), BUG(), oops()	Xử lý lỗi nghiêm trọng, dừng hệ thống
Hàm thoát tiến trình	do_exit(), _exit()	Thoát tiến trình, không quay lại
Hàm xử lý lỗi CPU	die(), handle_fault()	Xử lý lỗi CPU, lỗi bộ nhớ
Hàm xử lý lỗi truy cập bộ nhớ	do_page_fault(), bad_area()	Xử lý lỗi bộ nhớ không hợp lệ