BÀI 3: C NÂNG CAO & MAKEFILE

Advanced C Programming & Build System

🎯 Mục tiêu bài học

1. **C Nâng cao:** Hiểu sâu về Con trỏ hàm (Function Pointer), `void *` và Struct (Cốt lõi của Linux Driver).
2. **Quy trình biên dịch:** Hiểu rõ 4 bước từ Code $\rightarrow$ File chạy.
3. **Makefile:** Viết được Makefile chuyên nghiệp để quản lý dự án lớn.
4. **Tư duy tách file:** Quản lý code source (`.c`) và header (`.h`).

1. C Nâng cao: Tại sao cần học lại C?

Trong Embedded Linux, chúng ta không dùng C cơ bản (Vòng lặp, if/else). Kernel Linux sử dụng các kỹ thuật cao cấp:

- **Pointer to Void (`void *`):** Để truyền dữ liệu đa năng (Generic programming).
- **Function Pointer:** Để làm Callback, Plugin, và mô phỏng hướng đối tượng (OOP) trong C.
- **Struct & Bitfields:** Để thao tác trực tiếp với thanh ghi phần cứng và Protocol header.

2. Con trỏ hàm (Function Pointer)

Đây là khái niệm quan trọng nhất để viết Driver.

```c
#include <stdio.h>

void chao_tieng_anh() {
    printf("Hello\n");
}

void chao_tieng_viet() {
    printf("Xin chao\n");
}

int main() {
    // Khai báo con trỏ hàm
    void (*my_func_ptr)();

    // Trỏ tới hàm tiếng Việt
    my_func_ptr = chao_tieng_viet;
    my_func_ptr(); // In ra "Xin chao"

    return 0;
}
```

3. Quy trình biên dịch GCC

Lệnh `gcc main.c -o app` thực chất chạy 4 bước ngầm:

1. **Preprocessing (`-E`):** Xử lý `#include`, `#define`.
2. **Compilation (`-S`):** Chuyển C sang Assembly.
3. **Assembly (`-c`):** Chuyển Assembly sang mã máy (Object file `.o`).
4. **Linking:** Liên kết các file `.o` và thư viện thành file chạy cuối cùng.

4. Makefile là gì?

- **Vấn đề:** Dự án có 100 file `.c`. Gõ lệnh `gcc file1.c file2.c ... file100.c` rất mệt mỏi và dễ sai.
- **Giải pháp: Make**. Công cụ tự động hóa quy trình biên dịch.
- **Ưu điểm:** Chỉ biên dịch lại những file có thay đổi (Tiết kiệm thời gian build).

Cấu trúc cơ bản của Makefile

Makefile hoạt động dựa trên các **Quy tắc (Rules)**:

```
Target: Dependencies
    Command
```

- **Target:** File đích muốn tạo ra (ví dụ: `app`).
- **Dependencies:** Các file cần thiết để tạo ra target (ví dụ: `main.c`).
- **Command:** Lệnh thực thi (Bắt buộc phải thụt đầu dòng bằng **TAB**, không dùng Space).

Ví dụ Makefile (Cơ bản)

```
# Khai báo biến
CC = gcc
CFLAGS = -Wall -I.

# Rule chính
my_app: main.o hello.o
    $(CC) -o my_app main.o hello.o

# Rule biên dịch file object
main.o: main.c
    $(CC) $(CFLAGS) -c main.c

hello.o: hello.c
    $(CC) $(CFLAGS) -c hello.c

# Rule dọn dẹp
clean:
    rm -f *.o my_app
```

Makefile nâng cao (Mẫu chuẩn)

Sử dụng **Automatic Variables** để viết ngắn gọn và hỗ trợ Cross-compile.

```
CC = $(CROSS_COMPILE)gcc  # Hỗ trợ cross-compile
TARGET = app_demo
SRCS = main.c utils.c
OBJS = $(SRCS:.c=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) -o $@ $^  # $@ = Target, $^ = All dependencies

%.o: %.c
    $(CC) -c $< -o $@  # $< = First dependency

clean:
    rm -f $(OBJS) $(TARGET)
```

🛠️ PHẦN THỰC HÀNH (LAB 03)

Viết chương trình C mô phỏng Driver & Build bằng Makefile

Yêu cầu Lab 03

1. **Tạo cấu trúc dự án:**

```
Lab_03/
├── include/
│   └── calculation.h
├── src/
│   ├── main.c
│   └── calculation.c
└── Makefile
```

2. **Code C:** Sử dụng Function Pointer để thực hiện phép cộng/trừ (mô phỏng callback).
3. **Makefile:** Viết Makefile để build project, hỗ trợ biến `CROSS_COMPILE`.

Hướng dẫn Code (src/main.c)

```c
#include <stdio.h>
#include "calculation.h"

int main() {
    // Khai báo con trỏ hàm nhận 2 số int, trả về int
    int (*operation)(int, int);

    // Trỏ vào hàm cộng
    operation = add;
    printf("Add: 10 + 5 = %d\n", operation(10, 5));

    // Trỏ vào hàm trừ
    operation = sub;
    printf("Sub: 10 - 5 = %d\n", operation(10, 5));

    return 0;
}
```

Hướng dẫn Makefile

```
CC = $(CROSS_COMPILE)gcc
CFLAGS = -I./include -Wall

# Tự động tìm tất cả file .c trong src/
SRCS = $(wildcard src/*.c)
# Đổi đuôi .c thành .o
OBJS = $(SRCS:.c=.o)

TARGET = my_calculator

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f src/*.o $(TARGET)
```

Thử nghiệm (Test)

1. **Build cho PC (Native):**

```
make
./my_calculator
```

2. **Build cho Board (Cross-compile):**

```
make clean
make CROSS_COMPILE=arm-linux-gnueabihf-
file my_calculator
# Kết quả: ARM, EABI5 version 1 (SYSV), dynamically linked...
```

📝 Bài tập về nhà

1. Tìm hiểu thêm về biến đặc biệt trong Makefile: `?=` vs `=` vs `:=`.
2. Nâng cấp Makefile: Tạo thư mục `build/` riêng để chứa các file `.o` (để không làm rác thư mục `src`).
3. Ôn tập về **Struct Padding/Packing** (Rất quan trọng khi giao tiếp UART/Network).

Q & A

Hẹn gặp lại ở Bài 4: Toolchain & Cross-Compilation!