一、实验名称

设备树

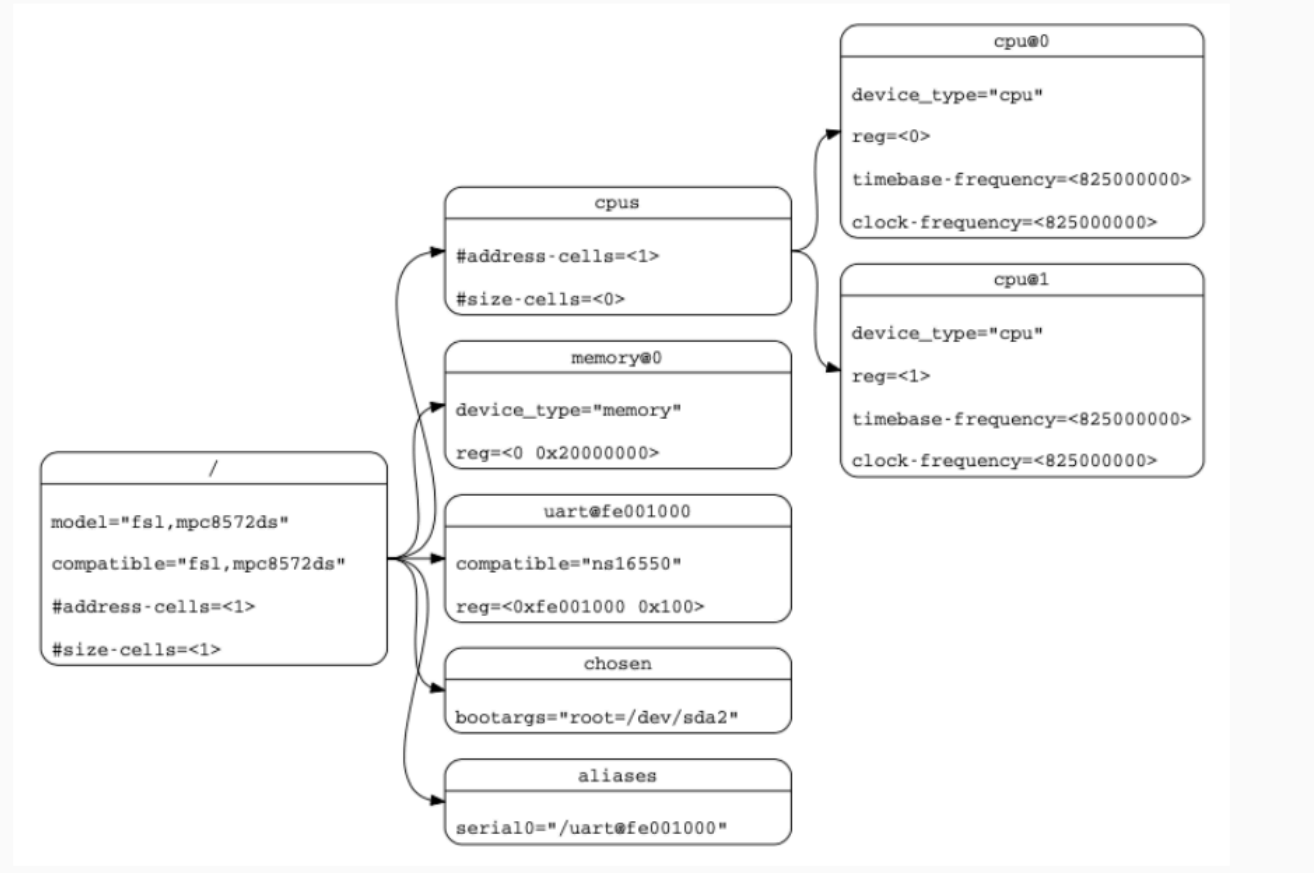
二、实验目的

依据 设备树规范 选择你熟悉的语言编写程序解析virt.dtb文件

三、实验内容

介绍

设备树是用于解决ARM等嵌入式系统由于设备种类纷繁复杂导致的与平台相关的大量内核代码被大量重复的问题。通过设备树来描述系统硬件及其属性，然后通过bootLoader将其传递给kernel，以便kernel可以有较大的灵活性。如下图所示设备树的例子。



> [补充说明]
> 根节点（/）：根节点是整个设备树的起点，所有的其他节点都是它的子节点。
> /cpus 节点：这个节点描述了系统中的CPU资源。**address-cells=<1>** 和 **size-cells=<0>**：这两个属性定义了地址和大小信息的编码方式。**cpu@0** 和 **cpu@1**：这两个子节点分别代表系统中的两个CPU核心。**device_type="cpu"**：标识这是一个CPU设备。**reg=<0>** 和 **reg=<1>**：分别表示这两个CPU核心的地址编号。 **timebase-frequency=<825000000>** 和 **clock-frequency=<825000000>**：分别表示时间基准频率和时钟频率，单位为Hz。

> /memory@0 节点：这个节点描述了系统的内存资源。**device_type="memory"**：标识这是一个内存设备。 **reg=<0 0x20000000>**：表示内存的起始物理地址为0，大小为0x20000000字节（即512MB）。
> /uart@fe001000 节点：这个节点描述了一个UART（通用异步收发器）串口设备。**compatible="ns16550"**：表示该设备兼容ns16550驱动。 **reg=<0xfe001000 0x100>**：表示设备的物理地址范围从0xfe001000开始，长度为0x100字节。
> /chosen 节点：这个节点通常用于指定系统启动时的参数。**bootargs="root=/dev/sda2"**：指定了根文件系统位于/dev/sda2设备上。
> /aliases 节点：这个节点用于定义设备别名。**serial0="uart@fe001000"**：为uart@fe001000设备定义了一个别名serial0，方便在系统中引用。

设备树是描述硬件的数据结构。 无需将设备的每个细节都硬编码到操作系统中，可以在引导时通过DTB这种形式的数据结构对硬件的各方面进行描述，然后传递给操作系统从而增强操作系统的灵活性。 设备树由 OpenFirmware、OpenPOWER 抽象层 (OPAL)、Power Architecture Platform Requirements (PAPR) 以独立的 Flattened Device Tree (FDT) 形式使用。

基本上，那些可以动态探测到的设备是不需要描述的，例如USB device。不过对于SOC上的usb host controller，它是无法动态识别的，需要在device tree中描述。同样的道理，在computer system中，PCI device可以被动态探测到，不需要在device tree中描述，但是PCI bridge如果不能被探测，那么就需要描述 1。

很多系统工具是OS开发过程中自然产生的需求，如将.dtb文件进行解析并转换成可读的.dts格式。

提示：由于本系列实验暂时没有使用bootloader，而是通过QEMU将内核直接加载到内存，所以不会借由设备树往kernel传递参数。

代码解析

依据 设备树规范 选择你熟悉的语言编写程序解析virt.dtb文件。

设备树解析关键点

1. **节点命名规则**
   节点名称格式为 `label:node-name@unit-address`，例如 `uart0: serial@101f0000`，其中 `unit-address` 表示寄存器基地址。
2. **属性值类型**
   - **字符串**：如 `compatible = "arm,cortex-a7"`。
   - **整数数组**：如 `reg = <0x1000 0x2000>`，需按大端字节序解析。
   - **二进制数据**：如 `local-mac-address = [00 11 22 33 44 55]`
3. **内存对齐要求**
   设备树规范要求数据按4字节对齐，代码通过 `(*offset + 3) & ~3` 实现自动对齐

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <endian.h>
#include <ctype.h>
```

设备树头部结构体

```
struct fdt_header {
    uint32_t magic;               // 魔数（0xd00dfeed），标识DTB文件合法性
    uint32_t totalsize;           // DTB文件总大小
    uint32_t off_dt_struct;       // 结构体块在DTB中的偏移
    uint32_t off_dt_strings;      // 字符串表在DTB中的偏移
    uint32_t off_mem_rsvmap;      // 内存保留映射表偏移（未在代码中使用）
    uint32_t version;             // 设备树版本
    uint32_t last_comp_version;   // 最后兼容版本
    uint32_t boot_cpuid_phys;     // 启动CPU的物理ID
    uint32_t size_dt_strings;     // 字符串表总大小
    uint32_t size_dt_struct;      // 结构体块总大小
};
```

缩进打印

```
static void print_indent(int depth) {
    for(int i = 0; i < depth; i++) {
        printf("\t");  // 每层节点缩进一个制表符
    }
}
```

字符串合法性检查

```
static int is_printable_string(const char *data, int len) {
    for(int i = 0; i < len; i++) {
        if(data[i] == 0 && i == len-1) continue;  // 忽略末尾的空字符
        if(!isprint(data[i]) && !isspace(data[i])) return 0;  // 非可打印字符
则非法
    }
    return 1;
}
```

属性值格式化输出

```
static void print_property_value(const char *data, int len, const char
*prop_name) {
    // 检查是否为字符串
    if(is_printable_string(data, len)) {
        printf("\"%s\"", data);  // 字符串类型用双引号包裹
        return;
    }
    // 检查是否为32位整数数组（cells）
    if(len % 4 == 0) {
        printf("<");
        for(int i = 0; i < len; i += 4) {
            uint32_t value = be32toh(*(uint32_t *)(data + i));  // 大端转主机
字节序
            printf("0x%02x", value);  // 以十六进制显示
        }
        printf(">");
        return;
    }
    // 其他情况按字节数组输出
    printf("[");
    for(int i = 0; i < len; i++) {
        printf("%02x", (unsigned char)data[i]);
    }
    printf("]");
}
```

解析节点名称并输出，处理地址对齐

```
static void parse_node(const char *dtb, uint32_t *offset, int depth) {
    print_indent(depth);
    char node_name[256];
    strcpy(node_name, dtb + *offset);  // 读取节点名称
    printf("%s {\n", node_name);
    *offset += strlen(node_name) + 1;  // 移动到名称结束位置
    *offset = (*offset + 3) & ~3;      // 对齐到4字节边界（设备树规范要求）
}
```

读取属性名和值，处理地址对齐，输出键值对

```
static void parse_prop(const char *dtb, const char *strings, uint32_t
*offset, int depth) {
    uint32_t len = be32toh(*(uint32_t *)(dtb + *offset));  // 属性值长度
```

```
    uint32_t nameoff = be32toh(*(uint32_t *)(dtb + *offset + 4));  // 属性名在字符串表中的偏移
    const char *name = strings + nameoff;  // 获取属性名
    const char *data = dtb + *offset + 8;  // 属性值数据地址

    print_indent(depth);
    printf("%s = ", name);
    print_property_value(data, len, name);  // 调用值解析函数
    printf(";\n");

    *offset += 8 + ((len + 3) & ~3);  // 移动到下一个属性（8字节头部 + 数据长度对齐）
}
```

主函数

```
// 主函数入口，接收命令行参数
int main(int argc, char *argv[]) {
    // 检查参数数量是否正确（程序名 + 1个dtb文件路径）
    if(argc != 2) {
        // 参数错误时打印使用说明
        printf("Usage: %s <dtb file>\n", argv[0]);
        return 1; // 返回错误码
    }

    // 尝试以二进制读模式打开指定的dtb文件
    FILE *f = fopen(argv[1], "rb");
    if(!f) { // 文件打开失败处理
        perror("Failed to open DTB file"); // 输出系统错误信息
        return 1;
    }

    // 获取文件大小：移动文件指针到末尾
    fseek(f, 0, SEEK_END);
    // 获取当前位置（即文件大小）
    long size = ftell(f);
    // 重置文件指针到起始位置
    fseek(f, 0, SEEK_SET);

    // 分配内存用于存储整个dtb文件内容
    char *dtb = malloc(size);
    if(!dtb) { // 内存分配失败处理
        perror("Failed to allocate memory");
```

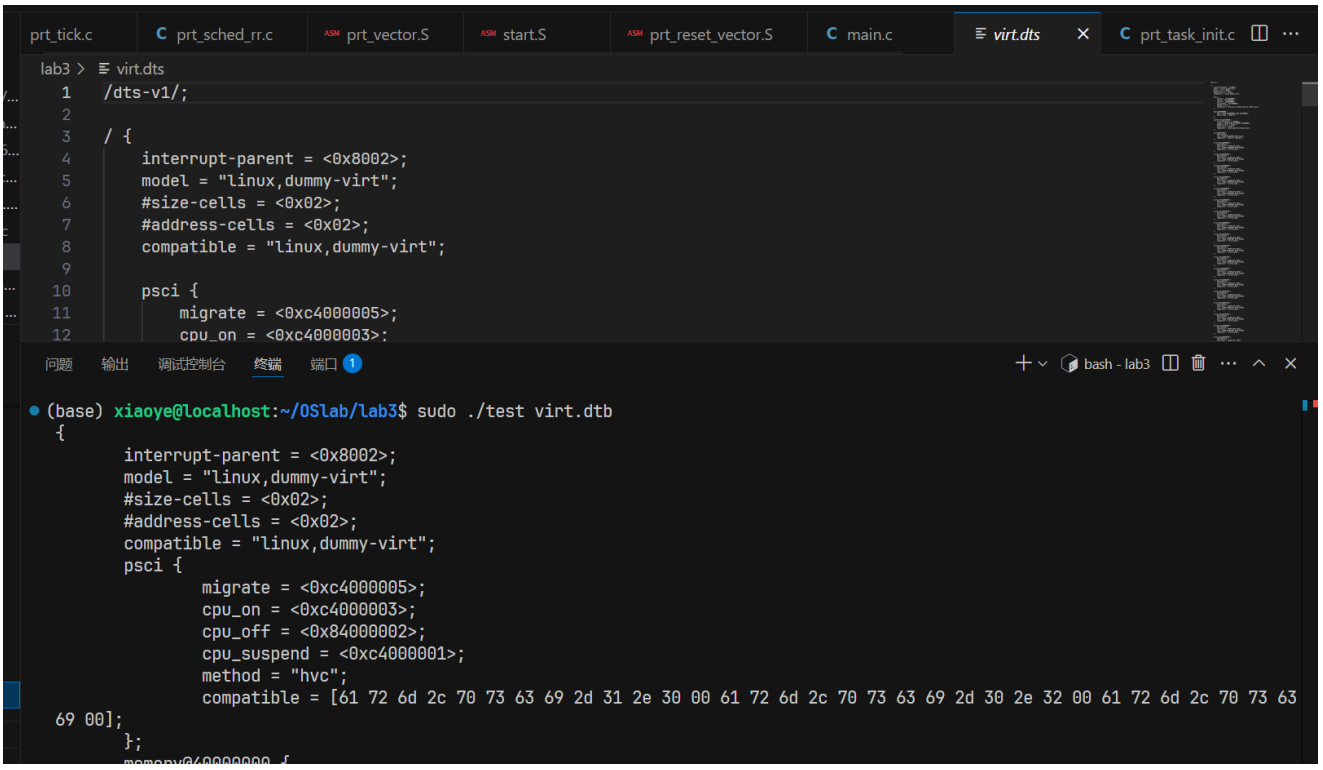
```
        fclose(f); // 关闭文件句柄
        return 1;
    }

    // 将文件内容读取到分配的内存中
    if(fread(dtb, 1, size, f) != size) { // 读取失败处理
        perror("Failed to read DTB file");
        free(dtb); // 释放已分配内存
        fclose(f); // 关闭文件句柄
        return 1;
    }
    fclose(f); // 成功读取后关闭文件

    // 解析dtb头部信息：将内存起始地址强制转换为头部结构体指针
    struct fdt_header *header = (struct fdt_header *)dtb;
    // 验证魔数（大端转主机字节序），检查是否是有效的dtb文件
    if(be32toh(header->magic) != FDT_MAGIC) {
        printf("Invalid DTB magic number\n"); // 魔数不匹配时报错
        free(dtb); // 释放内存
        return 1;
    }

    // 获取设备树结构和字符串表的偏移量（大端转主机字节序）
    uint32_t struct_offset = be32toh(header->off_dt_struct);
    // 字符串表存储位置：dtb起始地址 + 字符串表偏移
    const char *strings = dtb + be32toh(header->off_dt_strings);
    // 初始化当前解析位置为结构体起始偏移
    uint32_t offset = struct_offset;
    // 初始化嵌套深度计数器
    int depth = 0;

    // 主解析循环：遍历整个结构体区域
    while(offset < (struct_offset + be32toh(header->size_dt_struct))) { //
循环直到结构体结束
        // 读取当前token（大端转主机字节序），每次读取4字节
        uint32_t token = be32toh(*(uint32_t *)(dtb + offset));
        offset += 4; // 移动到下一个token位置

        // 根据token类型进行分支处理
        switch(token) {
            case FDT_BEGIN_NODE: // 节点开始标记
                parse_node(dtb, &offset, depth); // 解析节点信息
                depth++; // 嵌套深度增加
```

```
                break;

            case FDT_END_NODE: // 节点结束标记
                depth--; // 嵌套深度减少
                print_indent(depth); // 打印缩进
                printf("};\n"); // 输出节点结束符号
                break;

            case FDT_PROP: // 属性定义
                parse_prop(dtb, strings, &offset, depth); // 解析属性内容
                break;

            case FDT_NOP: // 空操作（保留字段）
                break;

            case FDT_END: // 结构体结束标记
                goto done; // 跳转到结束处理

            default: // 未知token处理
                printf("Unknown token: %x\n", token);
                goto done; // 异常退出解析
        }
    }

done: // 解析完成或异常退出标签
    free(dtb); // 释放分配的内存
    return 0; // 返回成功状态码
}
```

代码功能概述：这是一个DTB（设备树二进制）文件解析器，主要完成以下工作：

1. 检查命令行参数有效性
2. 读取指定DTB文件的二进制内容到内存
3. 验证文件格式有效性（魔数校验）
4. 解析设备树结构：
   - 处理节点开始/结束标记
   - 解析属性字段
   - 维护嵌套层级结构
   - 输出格式化的设备树结构
5. 包含完整的错误处理机制，确保内存安全和错误状态提示

解析结果



心得

总的来说，对于硬件设备、属性及其规范的描述细节，有了更清楚的认识。