

#### 1.

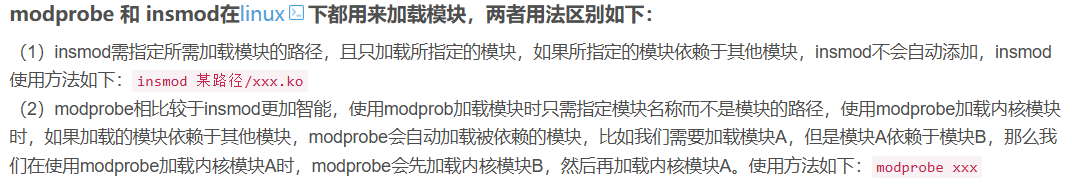
Linux系统一般有四个主要部分：内核、shell（系统用户界面）、文件系统、应用程序。

2.

Linux内核组成部分：5个

* 中断向量：中断源的识别标志，中断服务程序的入口地址

##### 3.



##### 4.ioremap



内核启动会初始化MMU，必须访问虚拟地址

涉及到用户空间与内核空间知识。内核空间可以直接访问物理内存，用户空间只能访问mmu转化的虚拟内存。

5.of\_device\_id

6.gpioled.nd = of\_find\_node\_by\_path("/gpioled"); //获取节点

gpioled.led\_gpio = of\_get\_named\_gpio(gpioled.nd, "led-gpio", 0); //获取gpio属性

1. 创建设备号

创建字符设备cdev

创建类class

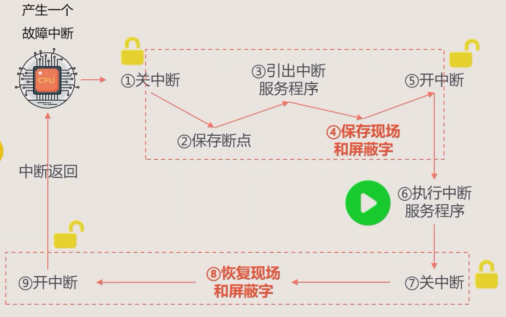
创建设备

1. 在具体应用上要不要先注册驱动再注册设备？有先后顺序没？

我现在做的所有都是先注册设备，后期再加载驱动的。

热插拔设备可以先注册驱动，后注册设备。

1. 中断知识



中断控制器接收中断信号、识别中断号

控制器通知CPU，CPU关中断、保存断点（程序计数器PC）和识别中断源

根据中断向量表跳转到对应的中断服务程序

先保护现场，然后再执行服务程序 （保护现场就是压栈，保存一些寄存器）

恢复现场，返回断点，重新打开中断

10.tasklet和workqueue

基于软中断，由于tasklet的优先级低于硬件中断，因此在执行过程中也可能被硬件中断打断，不能执行睡眠操作，因为软中断上下文没有像进程上下文那样的睡眠和唤醒机制。

Workqueue 开辟一个新线程给任务，可以用于延迟执行一些任务，可运行睡眠和重新调度，执行复杂任务。

1. 中断申请函数

int ret = request\_irq(my\_device\_irq\_num, my\_device\_interrupt, IRQF\_TRIGGER\_RISING, "my\_device", NULL);

这个函数的主要参数包括中断号（irq）、中断处理函数（handler）、中断触发方式（如上升沿触发、下降沿触发等）以及一些标志位（如共享中断标志等）。

1. linux内核的启动过程

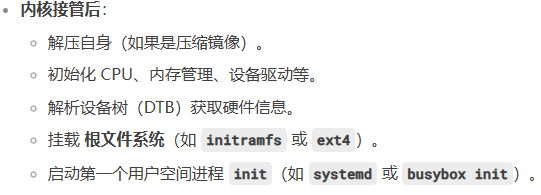
⑴引导加载程序阶段（Boot Loader）

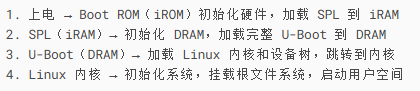
⑵内核解压与初步设置（Start\_kernel）

⑶设备驱动初始化与根文件系统挂载

⑷用户空间初始化与启动第一个用户进程  
deepseek给的详细内容







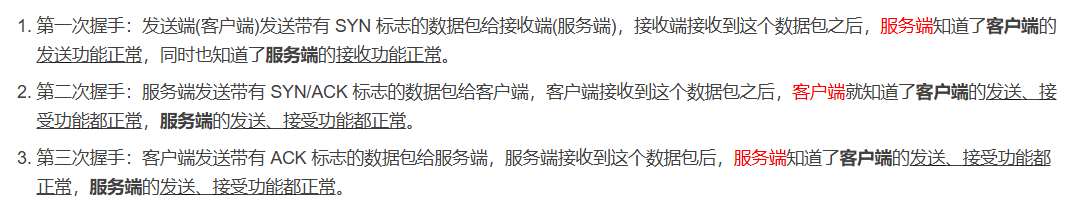
1. linux调度原理？

目前只接触了解过两个；

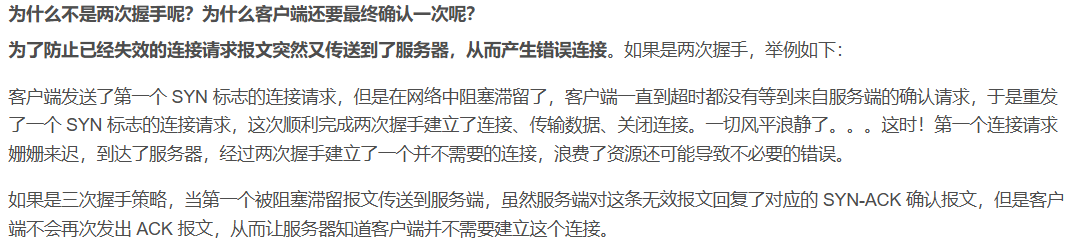
时间片轮转调度

抢占优先级调度

1. 网络tcp/ip



#### **为什么不是两次握手呢？**



1. FIQ和IRQ  
   FIQ 模式下，CPU 会自动屏蔽其他中断（包括 IRQ 和 FIQ），直到 FIQ 中断处理程序执行完毕。这种方式保证了 FIQ 中断处理过程不会被其他中断干扰，从而能够快速、独占式地完成紧急任务。

当进入 IRQ 模式处理中断时，根据 CPU 的设置，可能会允许其他更高优先级的中断（如 FIQ）打断当前的 IRQ 处理过程。

1. Static关键字  
   延长变量生命周期，只要程序还没结束，保存上一次的值。

如果修饰全局变量或函数，表示只能在此文件中调用，其他文件无法使用。

1. sizeof以字节的形式给出操作数的存储空间的大小
2. 内存泄露

内存泄露：是指在程序运行过程中，动态分配的内存没有被正确释放或者释放的时机不当，导致这部分内存无法再被程序访问和利用，从而造成系统内存资源的浪费。

避免内存泄露的几种方式

显式释放内存:有new就有delete，有malloc就有free

避免重复分配内存:程序在使用动态分配的内存时，应该避免重复分配内存，特别是在循环中。如果需要多次分配内存，可以使用realloc函数重新调整内存块的大小，以减少内存碎片的产生。

使用智能指针:智能指针是一种自动管理内存的工具，可以避免手动释放内存的繁琐操作。智能指针会在对象不再被使用时自动释放内存，并且可以避免内存泄漏和悬空指针等问题。

1. 野指针

指针未定义；

指针越界访问；（超出数组地址、自定义的不可访问内存地址）

指针指向的空间释放

1. 指针和函数（void（int \*xxx））

地址传递：会改变实参的值；

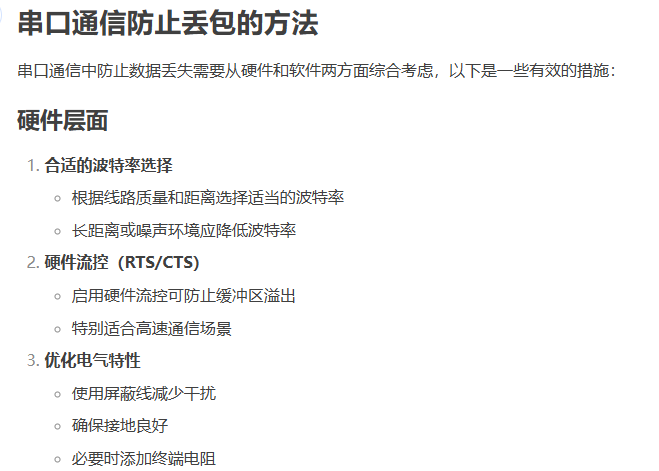
数值传递：不会改变实参的值；

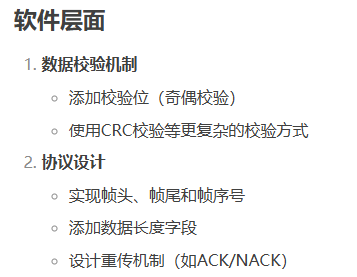
int（int a，int b）{ }这样不会改变原来的a和b

int（int \*a，int \*b）这样传递的是实参的地址，会改变原值；

20.C语言中有malloc和free，为什么c++还需要new和delete  


21.串口防丢包



软件上没法防止丢包，但是可以摒弃掉坏数据  


1. const常量 volatile易变的



一个是防止被程序员改，一个是防止编译器优化把变量优化掉；

1. sizeof和strlen

Sizeof（）是对象（函数——函数的返回类型所占的空间大小。函数的返回类型不能是void）的全部大小；strlen是chr类型的实际大小；

1. 管道

目前自己使用到的只有在调试的时候使用，

dmesg | grep "initialized"

25.基础define  


26.

我们定义一个数组char a[10]；数组第一个元素就是数组首地址；

是int \*类型的指针；我们可以使用\*（a++）来逐个输出数组；

&a则是数组指针，大小为数组大小，为10，

指针大小由系统决定，不管是int \*、char \*，在64位系统里都是8字节。

1. rtos和linux实时性

Rtos采用抢占式调度，实时性很高；

Linux 完全公平调度，时间片轮转调度，注重数据吞吐量，实时性不高

28.linux启动存储和内存的运行

一般讲uboot存在存储中，启动之后，内存会暂存uboot，然后bootz解压zmage和dtb；内存运行zmage和设备树；

根文件系统不是完全加载在内存中的，都是随用随取，所以存储会比较大，我们使用的板子emmc有8g，内存只有512m

SWAP：当内存中物理空间不够用时，会将不常用数据放到存储emmc中，扩大物理内存，也就是我们常说的内存拓展（非内存映射）；  
每个进程都认为自己拥有整个内存空间，32位系统就是拥有4G，但肯定不会真用到4G，用多少就在RAM里取用，超过实际RAM内存512m就使用上面说的swap换入换出使用emmc存储。

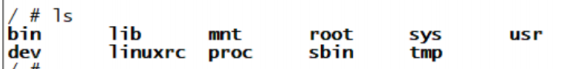
这个和ioremap又不一样了，后面再说。芯片MMU初始化，内存映射，无法访问物理地址。



1.   
   在驱动端加入输出日志代码



1. 熟悉根文件系统每个文件的内容？



1. 什么是MMU，MMU有什么用？

MMU（Memory Management Unit）是一种硬件设备，主要用于实现虚拟内存管理。

有了MMU之后，每个进程都有自己的虚拟地址空间，不会互相干扰。MMU还可以根据进程的访问权限，对虚拟内存保护。

此外，MMU还可以通过虚拟地址和物理地址的映射关系，实现了虚拟内存技术，使得进程能够访问大于物理内存的虚拟地址空间，从而提高了内存利用率和系统性能。

通过在用户空间和内核空间之间共享同一块物理内存来避免数据拷贝。减少CPU和内存资源消耗。

1. 上下文

上下文是指当前程序或进程执行的环境和状态，包括程序的执行位置、寄存器内容、堆栈信息、打开文件等。操作系统需要在多个进程之间进行快速的切换，这就需要在进程间保存和恢复上下文信息。

①Freertos中的vtaskdelay，

②中断服务程序结束时，

③任务挂起或者恢复，

1. 系统调用是什么，你用过哪些系统调用，和库函数有什么区别？

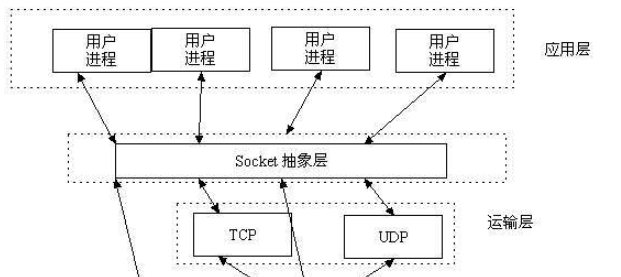
①进程管理 fork

②文件读写 open

③进程间通信 pipe

④网络通信 socket

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。



库函数在用户空间执行，系统调用是在内核空间执行，库函数运行时间属于用户时间，系统调用属于系统时间，库函数开销较小，系统调用开销较大

库函数是有缓冲的，系统调用是无缓冲的 （库函数（如 **fprintf**、**fgets**、**fwrite** 等）在用户空间维护了一个****缓冲区****（内存区域），数据会先写入缓冲区，积累到一定量后再通过系统调用批量写入内核（或从内核读取），减少频繁系统调用的开销。）

库函数并不依赖平台，库函数调用与系统无关，不同的系统，调用库函数，库函数会调用不同的底层函数实现，因此可移植性好。系统调用依赖平台。

14443a协议：

### 1. ****REQA / WUPA (请求命令)****

PCD 发出 REQA（全场请求）或 WUPA（唤醒请求）

所有在场的卡会回应 ATQA，表示它们在场，并告知 UID 类型（单倍/双倍）

### 2. ****Anti-collision（防碰撞）阶段****

以 **4 字节 UID** 为例，操作如下：

#### ➤ 步骤1：PCD 发送 SELECT 命令 + NVB 参数

Command: 0x93 (表示第一级)

Format:

SEL + NVB + <UID已知前缀>

SEL: 0x93 // Level 1 anti-collision

NVB: 0x20 // NVB = 0x20 表示启动防碰撞，从UID第1字节第0位开始

#### ➤ 步骤2：PICC 返回部分 UID（如没有冲突）

PICC 会返回完整 UID 或部分 UID。

如果多个卡同时应答且部分位不同，就发生冲突。

#### ➤ 步骤3：PCD 检测冲突

冲突位 = 当前响应中发生歧义的位置（如某一位两个卡分别是 0 和 1）

PCD 将冲突位记录下来，并用 **新的 NVB 和已知 UID 前缀** 再次发送 SELECT 请求。

加 0 或 1 来分支选中某一个标签。

#### ➤ 步骤4：PCD 不断迭代构造 UID

每次冲突后选择某一个路径（0 或 1）

最终唯一识别出一个标签的完整 UID

此时卡回应完整 UID + BCC（Block Check Character）

### 3. ****SELECT 阶段：唯一选中卡片****

当 UID 被完全识别后，PCD 发送：

SEL + NVB = 0x70 + UID[4] + BCC

卡回应 SAK（Select Acknowledge）确认是否需要第二级防碰撞。

假设有两张卡：

卡1 UID: 04 A1 B2 C3

卡2 UID: 04 A1 B2 D4

它们前三个字节一样，在第四个字节冲突。

流程如下：

PCD 发送 93 20

两张卡都回应 04 A1 B2, 冲突在第4字节

PCD 发现冲突位，比如第4字节第3位冲突

PCD 构造 93 3x（NVB = 3字节+某位），并加上冲突前前缀，尝试0或1分支

最终每次迭代只剩一张卡回应，得到完整 UID

int8\_t pcd\_anticoll(unsigned char sel\_code, unsigned char \*uid) 这个函数是获取uid的关键函数。

包括发送一级防冲撞命令0x93，没有冲突直接获取前四位uid。

有冲突会记录冲突字节和bit位，返回此函数开头重新组装发送数据，再次获取uid数据，知道不冲突。

冲突的判断：在pcd\_com\_transceive(pi);函数中，我们组装完数据发送出去，并接收所有返回的数据，接收完后，recstate\_reg寄存器会将第一位置1。当recstate\_reg寄存器的第二位为1时，表示有冲突位，就读取另外一个寄存器得到冲突字节和bit位