第27讲 设备驱动程序与文件系统

背景回顾：我们可以把设备想象成一组寄存器，以及寄存器基础上的一个设备通信协议，能够实现处理器和设备之间的数据交换：数据可以小到一个字符 (串口)，也可以大到程序和海量的数据 (GPU)。

本讲内容：很自然的问题是，如果操作系统上的程序想要访问设备，就必须把设备抽象成一个进程可以使用系统调用访问的操作系统对象，也就是设备驱动程序和文件系统：

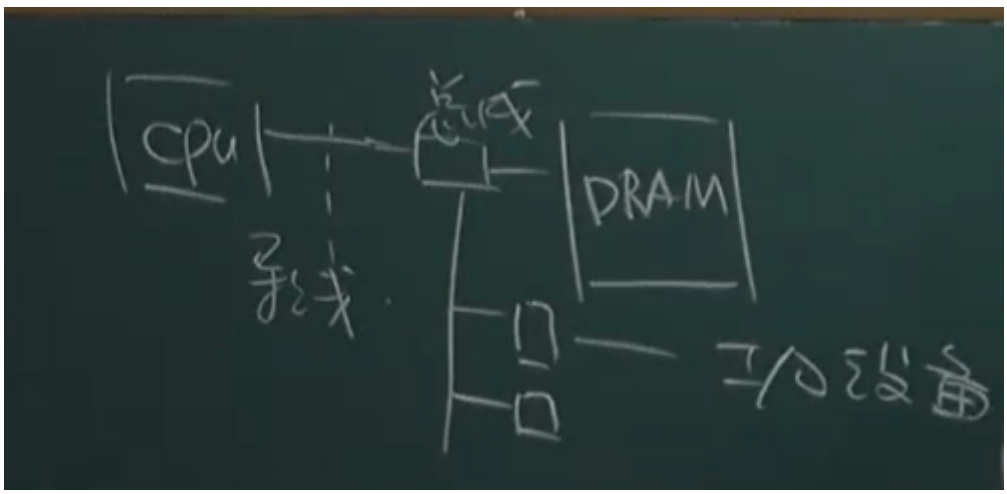
- 设备驱动程序
- 文件系统 API

1. Review

CPU出去一根导线，连到总线；总线是一个特殊的IO设备，可以把一个 CPU 上出去的地址分到一个具体的设备上。

而我们的 IO 设备的模型，它就是很多个寄存器。你甚至可以想象，我们的内存也是很多个寄存器——这是很直观的。当我们说我们要读写物理内存0X123456 这个地址的时候，其实你就是去 RAM 的那个寄存器里面，虽然它的实现未必是一个寄存器——去那个寄存器里面读一个值出来送到 CPU 里。

前面说了有了映射，我们就可以直接用类似load-store的指令来读出和写入设备了。

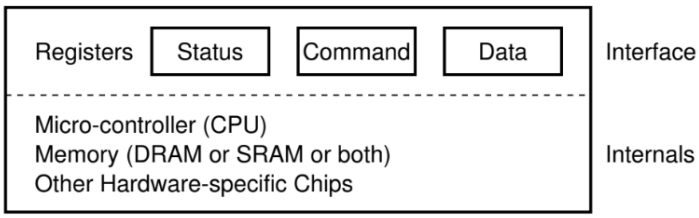


2. 设备驱动程序

I/O 设备的抽象

I/O 设备模型：一个能与 CPU 交换数据的接口/控制器

- 寄存器被映射到地址空间



操作系统：设备也是操作系统中的对象

- 如何“找到”一个对象？
- 对象支持什么操作？

我们为了让软件能够访问这些 IO 设备，那么首先当然可以把所有的这些控制寄存器直接暴露给应用程序。比如，我们可以为应用程序提供一个系统调用：

但这未必是一个好的抽象。

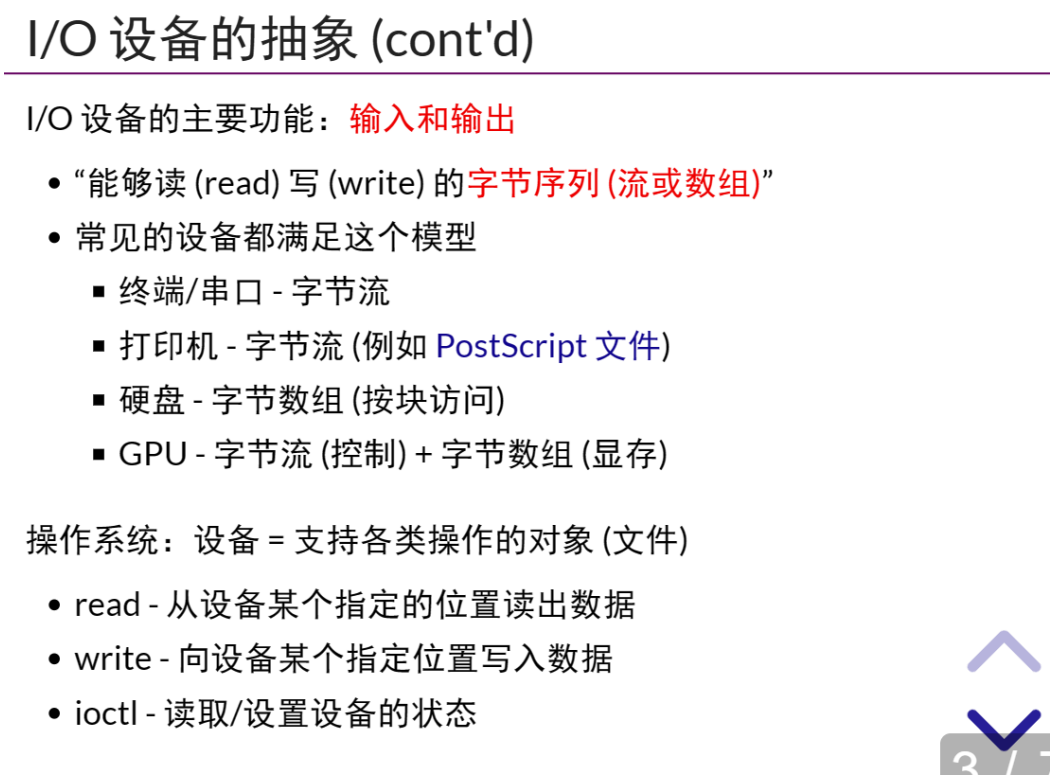
尤其是，我们说OS要把资源很好的管理起来，让应用程序能够比较舒适的使用资源，那舒适就是意味着我们的应用程序在使用资源的时候不要打架，就像你看我们的CPU，我们每一个应用程序他自己的感受是自己独占了整个 CPU ，他感受不到自己被中断了，然后被切出去，然后过了一会又回来，他感受到的是他就是状态机指令执行一直在持续。

同样的，我们的内存的虚拟化，我们有一个 32G 的物理内存，但我们的每一个进程都可以在自己的地址空间里面映射一块可读、可写、可执行的区域，他完全看不到底下的物理内存是什么样的，他就看到了整个地址空间里面所有东西都是他的，他都可以访问。

当然用的是那个 VR 眼镜的方法——用了一个虚拟化的手段。所以同样的操作系统为了把我们的 IO 设备管起来，直接把底层的设备寄存器暴露给各种程序，显然不是一个很好的想法。更好的想法是我们的操作系统应该把它抽象成是一个对象。

比如说如果我们有一个终端字符终端，它应该是操作系统的一个对象，然后如果有一个磁盘，它也是一个对象。然后在对象上面我们可以执行一些操作。那这样我们的不同的用户应用程序，它访问设备的时候就可以用完全相同的接口，他就不用看到这个底下每一个设备都不一样的接口——不同品牌的内存闪存，比如说闪存盘，它可能这个控制寄存器的协议稍微有点不一样，然后有些终端它的这个控制协议也有点不一样，那我们的操作系统做的事情就是不管这个设备是什么样的，都把它抽象成一个操作系统里的对象以及一组操作。

如果你们要把 IO 设备抽象成操作系统里的一个对象的话，你应该给这个对象，实现什么样的操作？这个问题的答案其实也很显然，IO 设备，它的功能就是 input 和output。



它既然是操作系统里的对象，我们就可以用一个指针去指向它。然而我们说到操作系统里面访问操作系统对象的指针就是文件描述符，因为 everything is a file，所以你可以把一个IO设备当成是一个文件，把它打开，打开以后你就得到了一个指向对象的文件描述符，然后你就对着这个文件描述符可以做 read write io control。

这个就是我们设备的抽象，而所谓的设备驱动程序这个概念，你们从刚开始接触计算机的时候就会接触到，因为有的时候你的一个设备接入你的电脑以后，它未必能够正常工作，尤其是比如说显卡这样比较复杂的设备，功能越复杂的设备，我们的操作系统就越轻，越可能没有提供一个比较好的默认的驱动。

那你会看到当你把这个显卡插上去的时候，Windows 会自动的在往互联网上搜索这个显卡的驱动，然后开始安装。比如说你刚插上去的时候，你可能只能以很低的分辨率显示图形，但是如果显卡的驱动安装完成以后，你就可以高清的分辨率来显示图形了。

而我们的显卡的驱动实际上就是设备和操作系统对象之间的桥梁。driver 就是操作系统里面的一份代码，一般来讲设备驱动是由厂商提供的，然后这个代码、这个驱动，知道我们 IO 设备具体的每一个寄存器的它的含义是什么。它可以把你对这个设备的 read write IO control，翻译成设备上寄存器对应的操作，也就是它是一个翻译者。

设备驱动程序

把系统调用 (read/write/ioctl/...) “翻译” 成与设备寄存器的交互

- 就是一段普通的内核代码
- 但可能会睡眠 (例如 P 信号量，等待中断中的 V 操作唤醒)

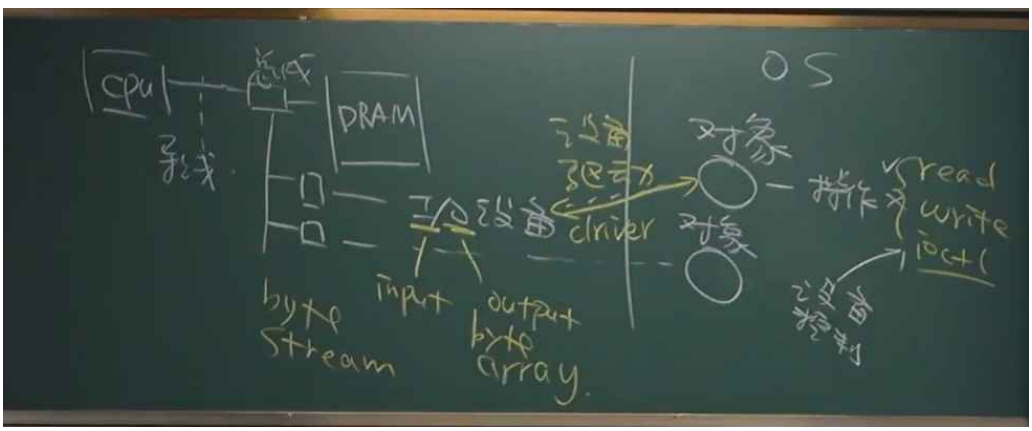
例子：`/dev/` 中的对象

- `/dev/pts/[x]` - pseudo terminal
- `/dev/zero` - “零” 设备
- `/dev/null` - “null” 设备
- `/dev/random`, `/dev/urandom` - 随机数生成器
  - 试一试：`head -c 512 [device] | xxd`
  - 以及观察它们的 strace
    - 能看到访问设备的系统调用

然后有意思的是，其实我们的 Linux 内核里面最多的代码就是设备驱动程序，你们可以想象，因为五花八门的厂商，每当它生产出一个新的产品的时候，我生产了一种新的监控摄像头，这个监控摄像头很好，有人脸识别的功能。那我在人脸识别功能的时候，它就不再是一个标准的摄像头。摄像头当然是有标准的。但如果你自己的厂商嗯造了一个新的摄像头的话，你希望把一些额外的，比如说人脸识别的数据也传出来，那它就不再是一个标准的摄像头。

那你可能会为它添加一个驱动，所以每一个厂商都会希望支持它的设备的话，都会在内核里面添加驱动，而驱动大部分时候也没有办法由让活人去负责它的质量，所以设备驱动是Linux内核里面最多而且也是质量最低的部分的代码，但现在Linux kernal做得很好，它如果驱动里面有 bug 的话，它也未必会直接使得你的kernal就直接就挂掉了。

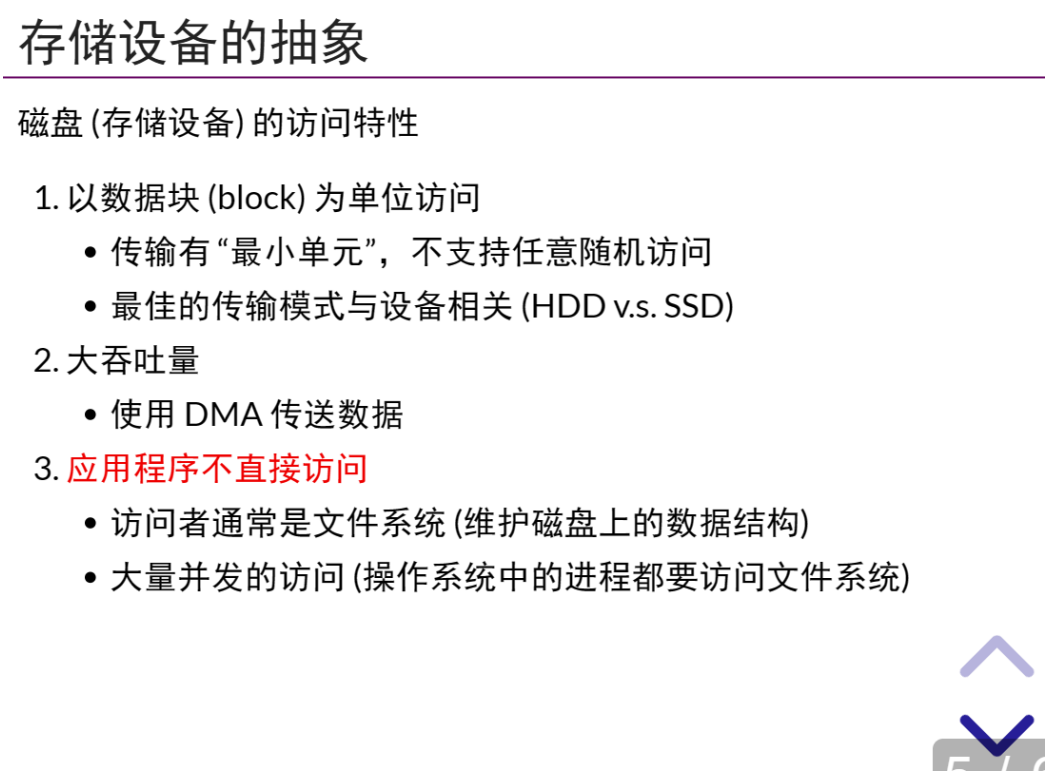
就是 Linux kernel 做了很好的容错，如果比如说驱动 crash 了，你只要你只是看到一个 error message，它并不会使你的整个系统就挂了。



3. 文件系统和API

磁盘和我们刚才讲的一些像终端核弹发射器这种具有这种一定程度的这种独占访问的设备还不一样。总的来说磁盘它有一种给**所有程序**共享，而且所有程序都希望以一个有秩序的方式去访问磁盘里面的数据。那这个时候单单设备驱动程序把它可抽象成是一个可读可写的字节序列，这个抽象就不够了。所以我们需要更高级别的抽象。

> 让所有程序共享磁盘？一个bug操作系统就没了！



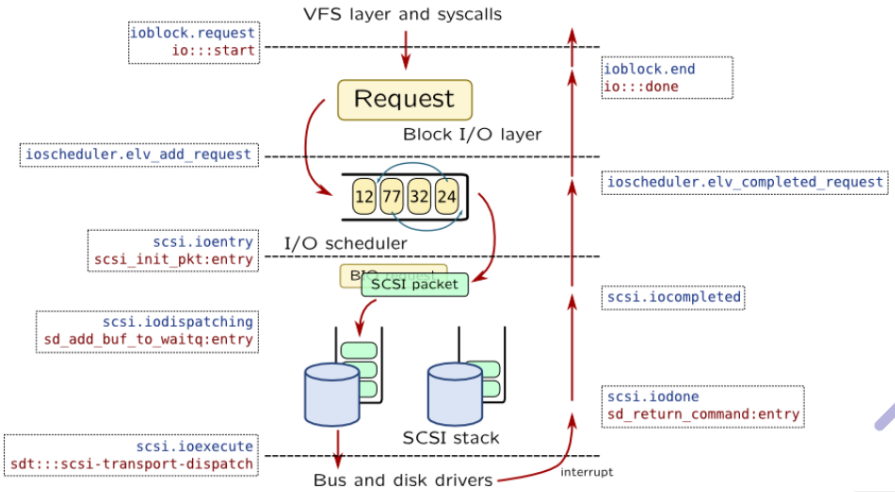
- 对于，比如说一个闪存盘来说，你要读写一个字节必须要以4MB来对齐读写。
- 应用程序都会同时访问。但他也不是直接访问。我们说应用程序确实是共享磁盘的，我们的数据都在磁盘上，但是我们的应用程序看到的不是磁盘，他看到的是文件系统。

Linux 操作系统就是在这样的一个文件系统和这个底层设备之间构建了一整个抽象层，存储的抽象层。我们平时用的比如像 open 这样的系统调用，它看到都是针对文件系统谈的。最终我们底下有一个设备，系统会把设备抽象成可以读、可以写的一部分。

Linux Block I/O Layer

文件系统和磁盘设备之间的接口

- bread (读一块), bwrite (写一块), bflush (等待过往写入落盘)



有了这样的一个机制以后，我们就可以在这个 block 的块设备的基础上构造我们今天看到的文件系统。

3.1 存储设备的虚拟化

那既然我们的文件系统里面已经存了整个计算机世界里面的全部，那你们马上就会知道字节序列并不是磁盘的一个好的抽象。不能像线程一样，我丢给你一个共享内存，好，你们就在这弄共享内存，上去玩去吧，这个显然不是一个负责任的抽象。如果你让所有的应用程序都去共享磁盘的话，如果有一个程序它有一个bug，它很有可能就把你的操作系统搞没了，或者把你另外一个重要的程序一个一个重要的应用程序搞没了，你的整个系统也就起不来了。

文件系统：实现设备在应用程序之间的共享

磁盘中存储的数据

- 程序数据
  - 可执行文件和动态链接库
  - 应用数据 (高清图片、过场动画、3D 模型.....)
- 用户数据
  - 文档、下载、截图
- 系统数据
  - Manpages
  - 配置文件 (/etc)

字节序列并不是磁盘的好抽象

- 让所有应用共享磁盘？一个程序 bug 操作系统就没了

文件 = 虚拟磁盘

文件：虚拟的磁盘

- 磁盘是一个“字节序列”
- 支持读/写操作

文件描述符：进程访问文件 (操作系统对象) 的“指针”

- 通过 open/pipe 获得
- 通过 close 释放
- 通过 dup/dup2 复制
- fork 时继承