

第28讲 文件系统

背景回顾：任何实现了 `struct file_operations` 的操作系统对象可以都是“文件”：有驱动程序的设备；`procfs` 中的虚拟文件、管道……

本讲内容：操作系统将这些文件组织起来的方式：文件系统。

1. 文件系统API

1. jyy

文件系统

文件 = struct file_operations

- 数据文件：hello.c, a.out, ...
 - 虚拟的磁盘
 - `vector<char>`
- 设备驱动
 - null, nuke0, ...
- 虚拟文件
 - 进程的地址空间, ...

新需求：怎么管理系统中众多的文件？

- `find_file_by_name?`

思路：信息的局部性

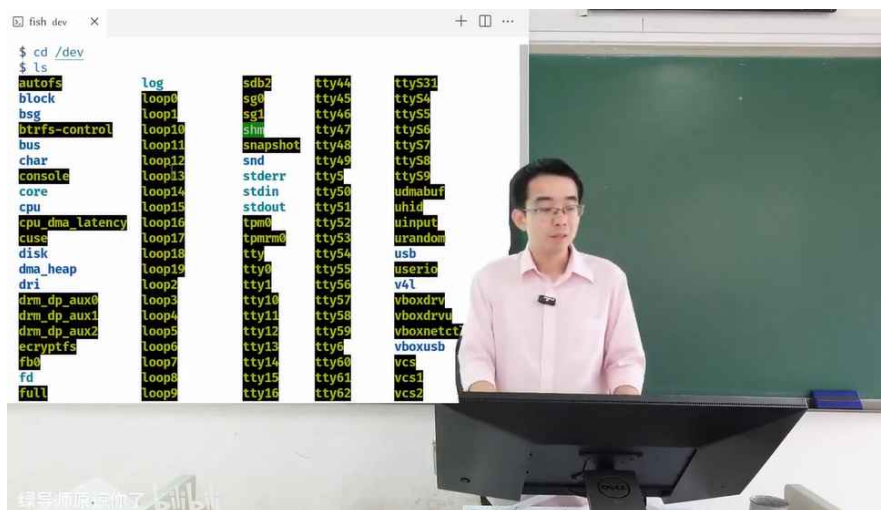
树状层次结构

- 逻辑相关的数据存放在相近的目录

```
├─ 学习资料
│  ├── .学习资料(隐藏)
│  ├── 问题求解1
│  ├── 问题求解2
│  ├── 问题求解3
│  ├── 问题求解4
│  └── 操作系统
```

我们用这样的信息的局部性就可以一层一层——如果你们学过一点机器学习的话，就有点像一个决策树，如果你要找到一个你内心想要的东西，然后你会不断的从最顶层的特征开始，把所有的东西给分开。（top down，从上到下分类的想法）

我们同样的也知道，如果你想要存储一些这样非数据的文件，比如说设备驱动，根据局部性原理，我们也应该把像设备这样的文件都放在一起。所以我们有 `/dev`：



你会发现这个第一层分类大概把系统里面最重要的一些文件或者说功能给分开了：



麻烦的是“非数据”的文件

UNIX/Linux: Everything is a File

- 一切都在“/”中 (例子: 中文语言包, fstab)

Windows 的设计

- 一个驱动器一棵树: A:, B:, C:, ...
- 其他命名空间: [Windows Driver Model](#), [Registry](#)
 - (也可以“everything is a file”)

```
HANDLE hDevice = CreateFile(  
    "\\.\PhysicalDrive0",  
    GENERIC_READ | GENERIC_WRITE,  
    FILE_SHARE_READ | FILE_SHARE_WRITE,  
    NULL, OPEN_EXISTING, 0, NULL  
);
```

那么接下来的问题是这些不完全是用户数据的文件到底应该怎么管理?

- 当然, 对 Unix 来说, 这个答案是 everything is file! 所有的一切都是在文件系统里面, 而且都在以 / 为根的文件系统里面是可达的。
- 如果是另外一个操作系统的设计者, 或者说你不是, 你不想设计一个 Linux 这样的操作系统, 其实这个答案并不是唯一的。Linux 只是做了一个 everything is file 这样的大家很容易接受也很容易理解的设计——大家所有的文件都在 / 里。上一节课, 我给大家看了中文的语言包。你们有时候会好奇, 语言到底是怎么实现的啊? 答案就是我有一个环境变量 LC, 然后你可以把它配置成任何的语言, 然后当你有了这个环境变量以后, 我们的进程就会根据这个环境变量去文件系统里面找有没有对应的语言包, 如果找到了他就会用那个语言包。
- 但 Windows 有一个很不一样的设计, 它不是一棵单个的树, 对吧? Unix 说好文件系统跟所有的东西, 不管我这有多少文件系统, 我这有多少块磁盘, 我都是从根可以走到的。但 Windows 不是, 因为 Windows 有一个很有意思的设计, 是每一个磁盘的驱动器是一棵树。这也是一个很自然的设计, 尤其是在以前, 大家可能还没有想到可以像 Unix、Linux 一样自由的组合文件系统的时候, 这就是一个很自然的设计——有一个驱动器就一棵树。为什么从 C 盘开始? 从 dos 时代就开始, A 和 B 盘是软盘, 装 dos 系统的。

然后 Windows 和 Linux 还有一个不一样。比如说系统里面很多的配置不像我刚才讲到 Linux 里面, 我的 nginx 可以直接用一个文本文件去修改它的配置, 它的 SSL 证书。

Windows 有一个机制叫注册表, 叫 registry, 当然你也可以把它理解成就是一个文本文件, ——你可以用一个文本文件来实现注册表, 我相信大家都在 Windows 里面玩过那个注册表编辑器。如果你们要安装南京大学给你们提供的正版的 Microsoft office 或者是 Windows 的话, 它会有一个脚本文件, 然后那个脚本文件就有可能会修改注册表。比如说他要把以你的 Windows 的一个 key 写进去, 他就可能会用修改注册表的方式, 现在 Windows 也有别的方式, 但这就相当于是一个巨大无比的配置文件, 所有的软件系统都可以, 包括 Windows 自己都可以把配置写在这里。

打比方如果你们觉得像 Windows defender 的那个病毒扫描, 就实时的病毒防护有时候会把 CPU 拉高, 尤其是在你的比较老的电脑上, 你可以通过写注册表的方式把它关闭。或者比如说 Windows

every reporting 这样的机制都可以在注册表里面把它关闭。但这是一个不在文件系统目录树里面的文件，这是有一点点不一样的，a 盘 b 盘 c 盘，你的下载的文件都在文件系统的目录树里，但是注册表你可以想象是一个单独的驱动器，对吧？比如说可以叫一个叫注册表的驱动器。

- 然后以及 Windows 的设备也不像雷尼斯一样，它是因为 everything is a file，所以我一定是通过文件的 API 去打开一个文件，然后我对它做 IO control。Windows 有一整套设备的 API，有一套设备的 API，所以 c 或者 C++ 的程序，它可以直接用那个设备的 API 去访问这个设备 Windows driver model。

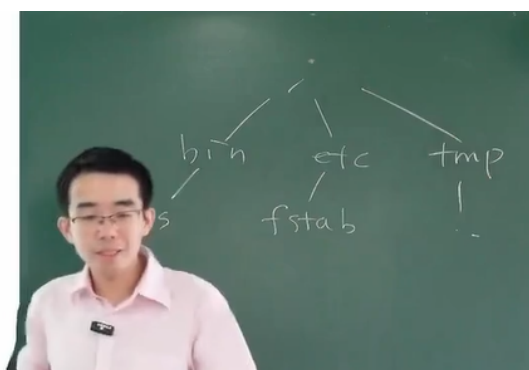
UNIX 的设计：目录树的拼接

mount: 将一个目录解析为另一个文件系统的根

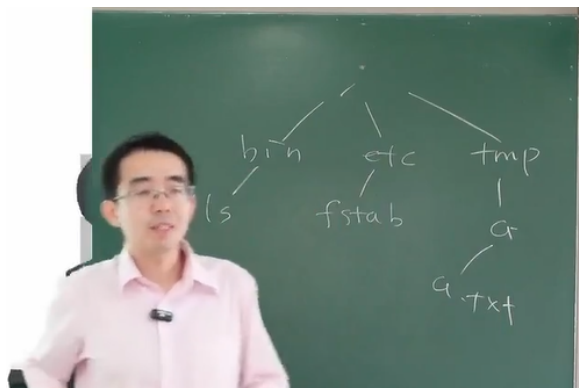
```
mount(source, target, filesystemtype, mountflags, data);
```

- 再看“最小 Linux”
 - 初始时只有 /dev/console 和几个文件
 - /proc, /sys, 甚至 /tmp 都没有
 - 答案：它们都是 mount 系统调用创建出来的
- UNIX 一贯的设计哲学：灵活
 - Linux 安装时的“mount point”
 - /, /home, /var 可以是独立的磁盘设备

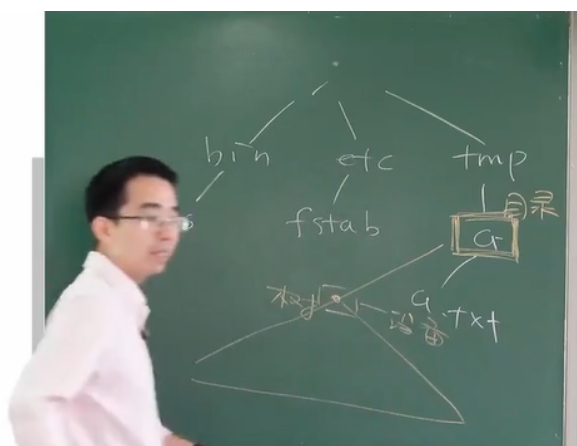
Unix 在设计的最初，它就支持一个操作叫目录树的拼接。我们的文件系统是有根的，而且根上一般来讲是有一个会挂一个实在的文件系统上：



他做的行为是他可以把一个 source，挂载到一个 target 上，这个 target 一定是一个目录。比如，我们在 tmp 下面创建 a，在 a 里面创建 a.txt：

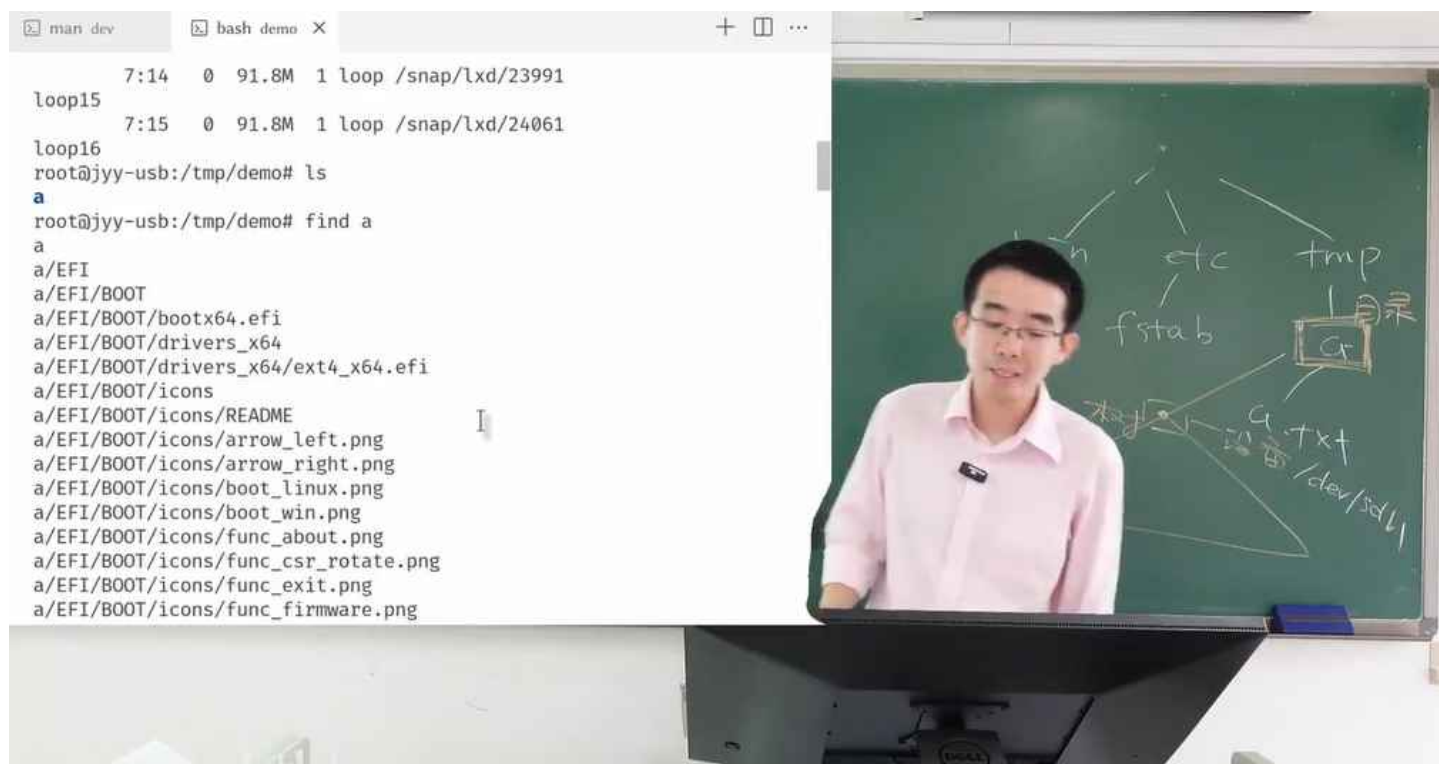


然后 mount 可以做一件什么事？ mount 可以选择整个文件系统树里面的任何一个目录（前提是它支持），他可以把这个目录换成是一棵文件系统的树，然后这个文件系统数是由一个设备相关的。

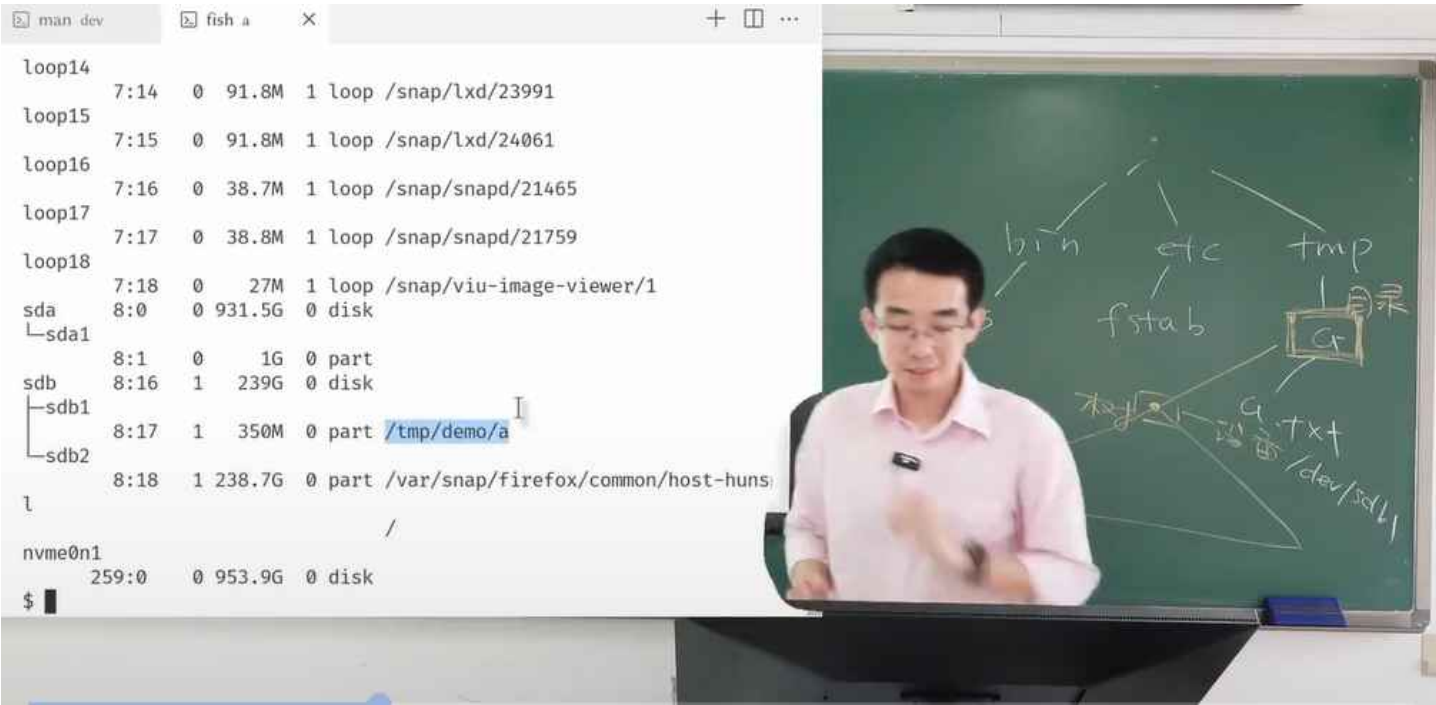


还是 tmp 下的 a，但是我现在做了一个偷梁换柱——有一个设备叫做 /dev/sdb1，然后因为它是一个设备，所以它上面可以把它解析成上面有一个文件系统。这个很好理解：一个磁盘或者一个磁盘的分区，它就是一个字节序列，就是一个存储设备，那我自然可以把它变成一个目录树。

然后我就可以把这个目录树直接嫁接到文件系统树的这个地方，这时候a文件夹还在，a这个名字还在，但是a里面所有东西都看不见了，a.txt没有了！



取而代之的是/dev/sdb1里面的一些东西。



我应该给大家看一下这 a.txt 有没有被删掉，你们觉得会还在吗？还在的，我挂载的时候就相当于是我画了一棵这样的树，然后我这里有一个目录叫a，然后我现在有一棵新的树，我就把这个树直接往上一贴，旧的东西还在，但是永远看不到了，把这个名字换成了另外一棵树，然后等我把那棵树扯下来的时候，这些东西又回来了，所以这个 a.txt 他又回来了。

gpt:

在 Linux 系统中，`/proc` 文件夹是一个虚拟文件系统，称为 proc 文件系统（procfs）。它提供了一个接口，用于访问内核数据结构。通过 `proc` 文件系统，用户和程序可以查看和操作内核信息，而不需要直接与内核数据结构交互。`/proc` 文件夹包含了很多文件和子目录，这些文件和目录并不是真正存储在磁盘上的数据，而是由内核在运行时动态生成的。

以下是 `/proc` 文件夹中一些常见的文件和目录，以及它们的用途：

1. 进程目录：

- `/proc/[pid]`：每个正在运行的进程都有一个对应的目录，名称是该进程的 PID（进程 ID）。这个目录包含了关于该进程的信息，例如：
 - `/proc/[pid]/cmdline`：进程的启动命令行。
 - `/proc/[pid]/cwd`：进程当前工作目录的符号链接。
 - `/proc/[pid]/environ`：进程的环境变量。
 - `/proc/[pid]/exe`：进程可执行文件的符号链接。
 - `/proc/[pid]/fd`：进程打开的文件描述符。
 - `/proc/[pid]/status`：进程的状态信息。

2. 系统信息文件：

- `/proc/cpuinfo`：包含关于 CPU 的信息，例如型号、频率、核数等。
- `/proc/meminfo`：提供系统内存的使用情况，包括总内存、空闲内存、缓冲区和缓存使用情况等。
- `/proc/uptime`：显示系统已经运行的时间和空闲时间。
- `/proc/version`：显示内核版本信息。
- `/proc/loadavg`：显示系统负载平均值。

3. 内核参数：

- `/proc/sys`：包含内核可调参数，可以通过这个目录查看和调整内核参数。例如：
 - `/proc/sys/kernel`：包含与内核相关的参数。
 - `/proc/sys/net`：包含与网络相关的参数。

4. 设备信息：

- `/proc/devices`：列出当前系统中的字符设备和块设备。
- `/proc/interrupts`：显示中断请求（IRQ）的统计信息。
- `/proc/ioports`：显示当前系统中的 I/O 端口信息。
- `/proc/partitions`：显示系统中的分区信息。

总的来说，`/proc` 文件系统提供了一种方便的方式来查看和修改内核和系统的状态信息。它是系统管理员和开发人员进行系统监控和调试的重要工具。

提问：所以procfs是一种文件系统，然后挂载到了/proc下？

回答：是的，`procfs`（proc 文件系统）确实是一种特殊的虚拟文件系统，并且通常挂载在 `/proc` 目录下。

具体来说，`procfs` 不是一个常规的磁盘文件系统，而是由内核直接在内存中动态生成的。它提供了一种访问内核数据结构的接口，通过这种接口，用户和应用程序可以方便地获取系统和进程的信息。

挂载 `procfs` 的步骤通常在系统启动时由启动脚本或内核自动完成。例如，在大多数 Linux 系统中，启动过程中的某个时刻会执行以下命令来挂载 `procfs`：

```
1 mount -t proc proc /proc
```

这条命令使用 `mount` 命令来挂载 `proc` 文件系统到 `/proc` 目录。选项 `-t proc` 指定了文件系统的类型是 `proc`。

这里有几个关键点可以进一步说明：

1. 虚拟文件系统：

- `procfs` 是内核提供的一种虚拟文件系统，它直接反映内核的数据结构和状态，因此可以动态变化，而不需要实际的磁盘存储。

2. 挂载点：

- 挂载点是 `/proc`，这意味着所有通过 `procfs` 提供的信息都可以通过访问 `/proc` 目录及其子目录来获得。

3. 动态内容：

- `proc` 文件系统中的内容是动态生成的。例如，当一个进程启动时，会在 `/proc` 中创建一个以该进程 PID 命名的目录，当进程结束时，该目录会被删除。

4. 访问接口：

- 通过读取和写入 `/proc` 中的文件，用户和程序可以获取内核信息和调整内核参数。例如，读取 `/proc/cpuinfo` 可以获取 CPU 信息，写入 `/proc/sys/net/ipv4/ip_forward` 可以启用或禁用 IP 转发。

下面是一个简单的示例，展示了如何通过 `/proc` 获取系统的 CPU 信息：

```
1 cat /proc/cpuinfo
```

执行这条命令后，你会看到关于系统中每个 CPU 的详细信息，包括型号、频率、缓存大小等。

总之，`procfs` 是一个非常强大的工具，它通过文件系统接口提供了对内核和系统状态的访问，大大简化了系统监控和调试的复杂性。

我当然可以自己 `mkdir` 一个 `myproc`，然后把 `procfs` 挂载过去——当然原来挂载的 `proc` 不受影响（我可以挂载好几份！）。

这个设计有什么作用？他给了我们非常灵活的机制，这个机制可以帮我们干什么呢？可以帮我们在文件系统里面给不同的目录分不同的设备，因为你的机器可能有多个磁盘。总的来说我们的 `mount` 可以支持的是把我们很多的设备都同时放到文件系统里来，比如说我们的服务器就会给 `/` 和 `/home` 分不同的磁盘。之前的时候我们的有一块比较小的但比较快的盘做系统盘，然后比较大但是不那么快的盘做 `/home` 给同学们存实验的数据，这样我们就可以把它们独立开来，在物理意义上独立开来可以并行的访问两块磁盘。

`mount` 还有一个，killer application，近乎杀手级的应用——我们不仅可以挂载刚才一个设备，我们还可以挂载一个文件系统里面的文件——文件是一个虚拟的磁盘！

Quiz

如何挂载一个 filesystem.img?

- 一个微妙的循环
 - 文件 = 磁盘上的虚拟磁盘
 - 挂载文件 = 在虚拟磁盘上虚拟出的虚拟磁盘 🤔
- 试试[镜像](#)

我能不能把这个虚拟的磁盘挂载到文件系统里呢？答案是可以的！

Linux 的处理方式

创建一个 loopback (回环) 设备

- 设备驱动把设备的 read/write 翻译成文件的 read/write
- [drivers/block/loop.c](#)
 - 实现了 loop_mq_ops (不是 file_operations)

观察挂载文件的 strace

- lsblk 查看系统中的 block devices (strace)
- strace 观察挂载的流程
 - `ioctl(3, LOOP_CTL_GET_FREE)`
 - `ioctl(4, LOOP_SET_FD, 3)`

gpt:

文件系统中的回环设备

文件系统中的回环设备（Loop Device）是一种虚拟的块设备，它允许你将一个普通文件当作块设备来挂载和访问。这在需要访问磁盘镜像文件或创建临时文件系统时非常有用。

使用场景

1. **挂载磁盘镜像文件：**你可以使用回环设备将 ISO 文件、磁盘镜像文件等挂载为实际的文件系统，从而可以浏览和操作其内容。
2. **创建加密文件系统：**通过回环设备，可以在一个文件中创建加密的文件系统，从而提供额外的安全性。

3. 临时存储：在开发和测试过程中，可以快速创建一个文件系统，而不需要实际的物理硬件支持。

操作示例

以下是如何使用回环设备挂载一个磁盘镜像文件的具体步骤：

1. 创建一个空文件：

```
1 dd if=/dev/zero of=mydisk.img bs=1M count=100
```

1. 格式化这个文件为一个文件系统：

```
1 mkfs.ext4 mydisk.img
```

1. 将文件与回环设备关联：

```
1 sudo losetup /dev/loop0 mydisk.img
```

1. 挂载回环设备：

```
1 sudo mount /dev/loop0 /mnt
```

现在，你可以访问 `/mnt` 目录，就像访问一个实际的磁盘分区一样。

1. 卸载和分离回环设备：

```
1 sudo umount /mnt
2 sudo losetup -d /dev/loop0
```

常用命令

- 查看当前系统中的回环设备：

```
1 losetup -a
```

- 自动设置回环设备：

```
1 sudo losetup -f mydisk.img
```

- 这条命令会自动找到一个未使用的回环设备，并将其与 `mydisk.img` 关联。
- 分离回环设备：

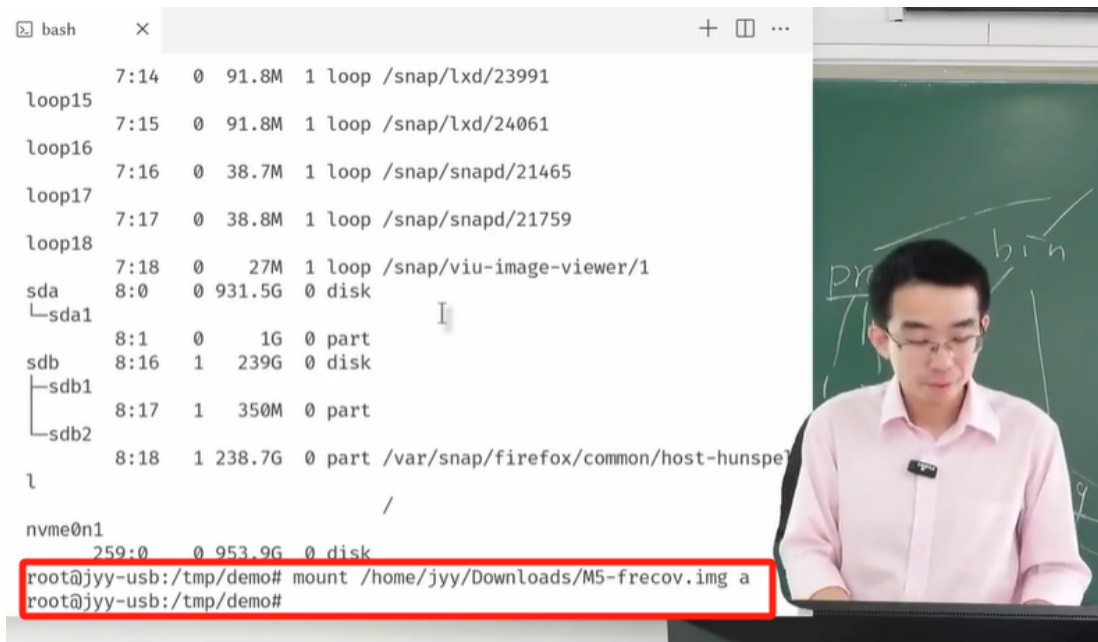
```
1 sh
2 复制代码
3 sudo losetup -d /dev/loop0
```

回环设备的优势

1. **灵活性**：可以将任何文件作为虚拟磁盘使用，非常方便。
2. **测试环境**：开发和测试过程中，可以轻松创建和销毁虚拟文件系统。
3. **安全性**：通过在文件中创建加密文件系统，可以增加数据存储的安全性。

总结

文件系统中的回环设备在 Linux 中提供了一种灵活的方式来使用普通文件作为块设备，从而可以挂载和操作磁盘镜像文件或创建临时文件系统。这种机制在开发、测试和数据管理中非常有用。



```

bash
loop15  7:15  0  91.8M  1 loop /snap/lxd/24061
loop16  7:16  0  38.7M  1 loop /snap/snapd/21465
loop17  7:17  0  38.8M  1 loop /snap/snapd/21759
loop18  7:18  0   27M  1 loop /snap/viu-image-viewer/1
loop19  7:19  0   64M  0 loop /tmp/demo/a
sda     8:0    0 931.5G  0 disk
├─sda1   8:1    0    1G  0 part
├─sdb    8:16   1  239G  0 disk
│  └─sdb1  8:17   1   350M  0 part
│      └─sdb2  8:18   1 238.7G  0 part /var/snap/firefox/common/host-hunspel
└─
nvme0n1 259:0   0 953.9G  0 disk
root@jyy-usb:/tmp/demo#

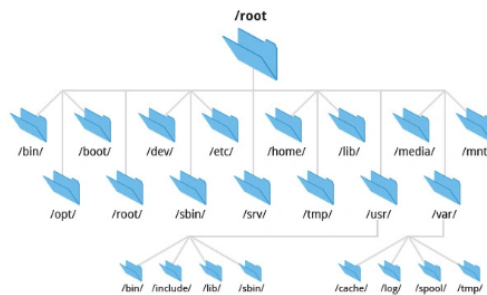
```

真正挂载的时候linux做什么呢？在操作系统你看不见的地方创建一个新的设备叫 LOOP 19，然后这个 LOOP 19 就会和这个文件关联起来，他们俩是同一个东西。然后因为我的挂载的时候必须要挂载一个存储设备，所以它会把它虚拟成一个这样的设备，这样我就可以把这个设备挂上去了，所以实际上我实际上挂载的是一个叫 LOOP 19 的设备。

Filesystem Hierarchy Standard

我们看到的目录结构

- **FHS** enables *software and user* to predict the location of installed files and directories.
 - 例子：macOS 是 UNIX 的内核 (BSD), 但不遵循 Linux FHS



就是这样的一整套机制，运行了很多年以后，形成了我们今天的file system hierarchy standard，比如说我们为什么要有 `/`？`/` 里面为什么有 `bin`？你们每一个安装的 Linux 都有 `/bin`、`/media`、`/home`，对吧？而其他的Unix，比如说像 macos 就不遵守这个规则。它有一整套系统世界的规定，而都是在长期的实践当中形成的。所以那个笑话说 Unix is friendly，但是它很挑谁是他的friends。是因为只有跟他一起共事过很长时间的人，才形成了 Linux 世界里面的基本概念。整个文件系统的目录树就是这样被创建出来的。

下来我们的 Linux 上面的文件系统还有一个有意思的feature，是我们的文件其实是在目录里面有两个不同名字的——甚至同名也可以的文件，他们可以实际上是同一个文件。

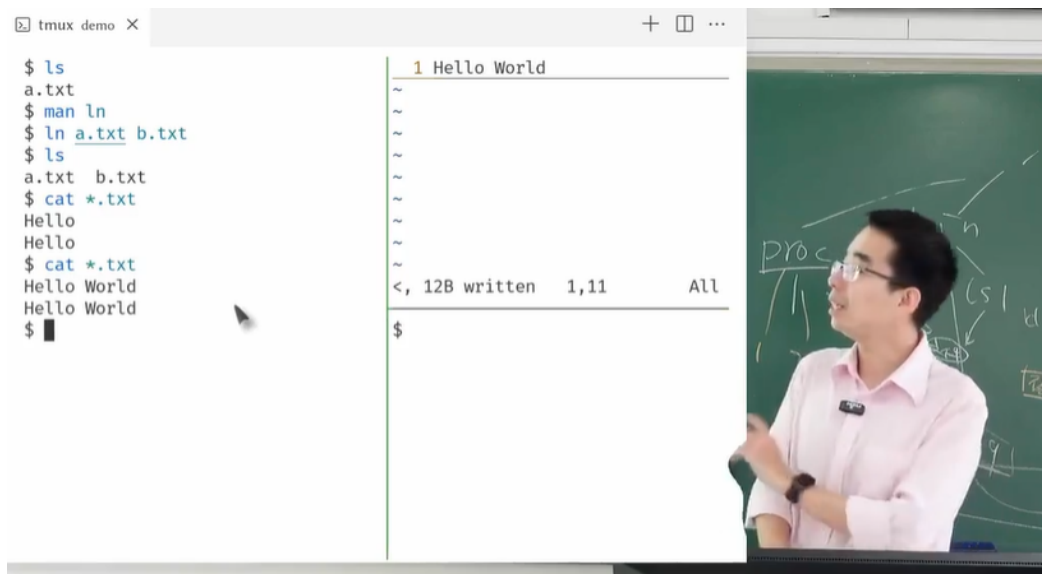
硬 (hard) 链接

需求：系统中可能有同一个运行库的多个版本

- `libc-2.27.so`, `libc-2.26.so`, ...
- 还需要一个“当前版本的 libc”
 - 程序需要链接“`libc.so.6`”，能否避免文件的一份拷贝？

(硬) 链接：允许一个文件被多个目录引用

- 文件系统实现的特性 (ls -li 查看)
 - 不能链接目录、不能跨文件系统
 - 删除文件的系统调用称为“`unlink`” (`refcount--`)



这两个文件除了文件名不一样，所有信息都是完全相同。

硬链接不能跨文件系统，因为这本身就是fs实现的一部分。

软 (symbolic) 链接

软链接：在文件里存储一个“跳转提示”

- 软链接也是一个文件
 - 当引用这个文件时，去找另一个文件
 - 另一个文件的绝对/相对路径以文本形式存储在文件里
 - 可以跨文件系统、可以链接目录、.....

几乎没有任何限制

- 类似“快捷方式”
 - 链接指向的位置不存在也没关系
 - (也许下次就存在了)

- 符号链接就是windows里面的快捷方式！

你可以想象什么是一个符号链接，符号链接就是任何一个目录里面我都可以有一个叫做 `a.txt`，然后这个 `a.txt`，我可以在这个文件头上写，这是一个快捷方式。他要 `goto` 另外一个地方去找，然后操作系统在读到这个文件的这个内容的时候，就会自动的做一个向另外一个地方的查找，这样原则上我可以做任何的跨文件系统的符号链接——他只要一个名字就可以了，所以他只要去这个地方找就可以了。所以你们有过这样的经历，会把一个快捷方式拷给别人，让别人是没有办法运行的，这是一个很经典的，你们在使用 Windows 的时候会遇到的问题。

2. 来点CSE

1. Hard link

Q：文件名是不是文件的一部分？

A：文件名和文件到底是什么关系？没有关系……你的名字和你有关系吗？如果有一天你失忆了并且到了一个没人认识你的地方，这个名字就没有用处了……

那你的名字是什么时候和你有关系的呢？

你的出生证！然后国家把这个对应关系写到database里面了。

接下来一个问题就是目录名太深了，要打好几层目录。在我们前面说的目录结构里面，并没有任何地方组织我们把一个文件名对应的inode-number设置成同一个。也就是不同文件名对应到同一个inode-number。如果我们为一个inode创建多个name，我们就可以为一个很深目录的文件名创建一个很短的文件名，最后对应的inode是一个。（具体的实现方法往目录里面增加一行就行了，多个文件名对应同一个inode）

假设一个文件有两个名字，当我们删掉一个名字的时候，用另一个去访问是能访问还是不能访问呢？

什么是删文件：本质操作只是删除目录中的helloworld.txt -> 12删除掉，并不是删除block中的数据，因为可能别的地方还引用了。所以我们要在inode中维护一个reference counter，如果删除完reference counter == 0就可以删除了。同时我们还需要加一个type来标识是一个普通文件还是目录。

Links

Path name	File name	Inode num	File (inode)	Block num	Disk Block
-----------	-----------	-----------	--------------	-----------	------------

Reference count

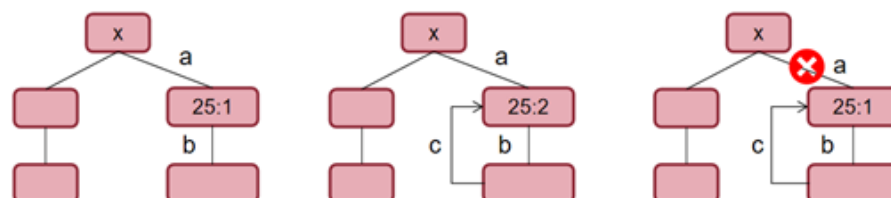
- An inode can bind multiple file names
- +1 when **LINK**, -1 when **UNLINK**
- A file will be deleted when reference count is 0
- **No cycle allowed**
 - Except for '.' and '..'
 - Naming current and parent directory with no need to know their names

```
struct inode
{
    integer block_nums[N]
    integer size
    integer type
    integer refcnt
}
```

在unix的v6版本文件系统里面，有一个特点，虽然inode有ref counter，但是目录不能有多个name。点和点点的实现就是在表里加一个.和..，对应当前目录和上一级目录的inode-number。我们能不能建立一个指向目录的link呢？也就是一个目录能不能在不同地方有多个name呢？

No Cycle for LINK

Path name	File name	Inode num	File (inode)	Block num	Disk Block
-----------	-----------	-----------	--------------	-----------	------------



- /a/b is a directory
- The refcnt of a is 1
- a's inode num is 25

- LINK ("/a/b/c", "a")
- Cause a cycle!
- Refcnt of a is 2

- UNLINK ("/a")
- Refcnt of a is 1, so the inode 25 is not deleted
- Now inode 25 is disconnected from graph
- No one can get it!

Think: Why deleting a directory requires to delete all the files in it first?

25

为什么我们要禁止给目录加link【其实有一个例外，.（当前目录）和..（上一级目录），一会再说】？假设inode有个x，记录了目录a，目录a中记录了目录b。假设我们现在来link把/a/b/c对应到/a，也就是在a的子目录下建立了一个指向a的link。a现在有两个人指向它，也就是/a和/a/b/c，它的ref count是2。现在我们尝试删除/a，它的ref是1，但是从x再也找不到a了。因为它唯一的名字被删除了。它既没有被删掉，又不能被另外的地方访问到。所以就会成为内存垃圾。所以禁止指向目录。

当我们删除一个非空目录的时候，其实我们是递归删除的，在真实的linux里面，我们已经考虑了这个问题了，但这个rm -r在unix第6版是不存在的。

下面来看rename。当我们要去rename的时候，改文件名是文件系统中非常复杂的事情。假设我们要去做改文件名的操作，比如mv (from,to)，这一个指令牵涉到3件事情。

1. 把to删了，to的ref count变成0.
2. 把from加一个to的名字，from的ref count变成2.
3. 再把from的ref count变成1

2. Symbolic link layer

而第七层symbolic link layer，我们在文件系统中创建了一个新的文件，内容是一个字符串，指向需要链接的文件路径和文件名。它不会仅仅停留在打开symbolic link本身，它会读取数据，并且指向进一步的操作。所以在inode_type中要添加symbolic link。

```
$ ln -s "/tmp/abc" s-link

$ ls -l s-link
7536945 lrwxrwxrwx 1 xiayubin 8 Sep 20 08:01 s-link -> /tmp/abc

$ readlink s-link
/tmp/abc

$ cat s-link
cat: slink: No such file or directory

$ echo "hello, world" > /tmp/abc

$ cat s-link
hello, world
```

What does "8" means? File size

详细解释

1. 创建符号链接

```
1 ln -s /tmp/abc s-link
```

这条命令创建一个符号链接 `s-link`，指向 `/tmp/abc`。 `-s` 参数表示创建符号链接。

2. 查看符号链接信息

```
1 ls -l s-link
```

输出示例：

```
1 lrwxrwxrwx 1 user group 8 Sep 20 08:01 s-link -> /tmp/abc
```

这里是对输出的解释：

- `l`：表示这是一个符号链接（link）。
- `rw-rw-rw-`：表示所有用户都对这个符号链接有读、写和执行权限。
- `1`：硬链接数目（对于符号链接总是 `1`）。
- `user`：文件所有者。
- `group`：文件所属的组。
- `8`：符号链接的大小，即符号链接所指向路径字符串的长度，这里 `/tmp/abc` 的长度是 `8`。
- `Sep 20 08:01`：文件的最后修改时间。
- `s-link -> /tmp/abc`：符号链接 `s-link` 指向 `/tmp/abc`。

3. 读取符号链接的目标

```
1 readlink s-link
```

这条命令输出符号链接 `s-link` 指向的目标路径：

```
1 /tmp/abc
```

4. 查看符号链接的目标文件内容

```
1 cat s-link
```

如果 `/tmp/abc` 文件不存在，则会报错：

```
1 cat: s-link: No such file or directory
```

5. 写入内容到目标文件

```
1 echo "hello, world"/tmp/abc
```

这条命令将字符串 `"hello, world"` 写入到文件 `/tmp/abc` 中。

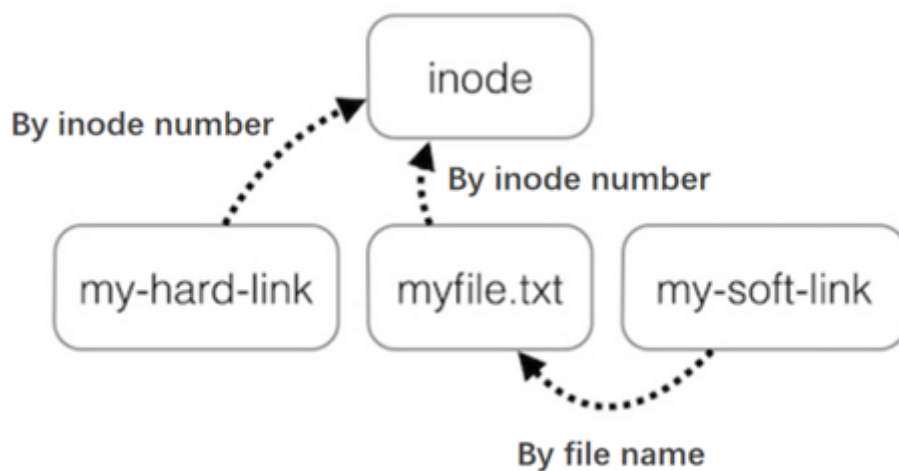
6. 再次查看符号链接的目标文件内容

```
1 cat s-link
```

这次可以成功读取 `/tmp/abc` 文件的内容：

```
1 hello, world
```

我们创建s-link的信息的时候，我们发现是指向/tmp/abc的。我们可以通过命令readlink来读出symbolic link文件的内容。8是/tmp/abc的大小，它其实不需要去存/0。如果我们尝试使用 cat来读取数据的时候，会报错。当我们把hello, world重定向到/tmp/abc的时候，才会读到hello, world。所以这个告诉我们哪怕原先的路径的文件不存在的话，我们也可以为那个不存在的文件创建一个symbolic link，但是对于hard link来说，我们不能为不存在的文件创建一个hard link，因为inode都不存在。



我们可以通过上图来审视一下hard link和symbolic link。Hard link是通过inode-number绑定，而symbolic link(soft link)是通过路径和文件名来绑定。Symbolic link是可以指向目录的，而hard link除了.和..是不能指向目录的。

2. 文件系统：实现

刚才我们提到这个文件系统里面的最核心的机制就是mount，也就是挂载。挂载就意味着我有一个设备，有一个device，是一个字节的序列，或者说在现实的存储设备中它叫 block device，叫块设备，因为这个设备是按块来访问的，比如说按照4KB，然后我可以把一个 device 和一个文件系统里面的位置一个目录关联起来。

如果是《数据结构》课？

借助 RAM 自由布局目录和文件

- 文件系统就是一个 Abstract DataType (ADT)

```
class FSObject {  
};  
  
class File: FSObject {  
    std::vector<char> content;  
};  
  
class Directory: FSObject {  
    std::map<std::string, FSObject*> children;  
};
```

回到《操作系统》课

对不起，没有 Random Access Memory

- 我们只有 **block device**
- 两个 API
 - `bread(int bid, struct block *b);`
 - `bwrite(int bid, struct block *b);`

实现：

- `read, write, ftruncate, ...`
- `mkdir, rmdir, readdir, link, unlink, ...`
 - 用 `bread/bwrite` 模拟 RAM → 严重的读/写放大
 - 我们需要更适合磁盘的数据结构

我们的敌人和朋友

敌人：读/写放大

- 被迫读写连续的一块数据

朋友：局部性 + 缓存

- 适当地排布数据，使得临近的数据有“一同访问”的倾向
- 数据暂时停留在内存，延迟写回

3. Take-away Messages

把文件系统理解成一个“数据结构”，就不难分析其中的重点和实现要点——我们总是把数据按照局部性组织起来，无论是 FAT 还是 bitmap/inode 的设计，都利用了这一点。另一个重要的设计是“目录也是文件”——文件系统实现将目录文件中的数据作出解读，从而解析出其中的目录结构。