CS110

Unix v6 文件系统

文件

数据存储和访问

- 堆栈、堆和其他程序数据段位于内存(RAM)中
  - 访问快速
  - 字节可寻址：通过地址快速访问数据任意字节
  - 不持久：断电丢失
- 文件系统存在磁盘上
  - 访问慢
  - 持久性：断电不会丢失数据
  - 扇区可寻址：只能读写数据的扇区（一般为512字节）

文件系统目标

- 磁盘上创建新文件
- 查找文件在磁盘上的位置
- 从磁盘读取全部或部分现有文件
- 从磁盘编辑现有文件的一部分
- 在磁盘上创建文件夹
- 获取磁盘上文件夹的内容...

扇区和块

> 文件系统通常定义自己的数据单元（块）

- 扇区 = 硬盘存储单元
- 块 = 文件系统存储单元（一个或多个扇区）- 软件抽象

  > Unix v6 文件系统将一个块定义为 1 个扇区（因此它们可以互换）

在磁盘上存储数据

> 使用两种类型的数据

1. 文件负载数据 - 文件内容

   对于小于512字节的文件，仍然保留整个区块

   对于跨越多个块的文件，它们的块不需要相邻
2. 文件元数据 - 有关文件的信息

索引节点

> **inode**是关于单个文件的一组数据，存储以下内容
>
> - 文件大小
> - 存储文件有效负载数据的块编号的有序列表

- 文件系统将磁盘的`inode`一起存储在`inode表`中，以便快速访问
- `inode`存储在从块2开始的保留区域中（块0是包含硬盘驱动信息的引导块，块1是包含文件系统信息的超级块）
- 对于大小为32字节的`inode`，块大小为一个扇区（512字节），16个`inode`对应一个块

> **文件系统从文件名到索引节点号再到文件数据**

如何访问？

```
typedef struct inode {
  uint16_t  i_addr[8];  // 8个块
  ...
} inode;

// 读取 sector 2 中的所有inode
inode inodes[512 / sizeof(inode)];
readSector(2, inodes);
for (size_t i = 0; i < sizeof(inodes) / sizeof(inodes[0]); i++) {
    ...
}
```

那么对于大文件呢？

> 每个 inode 有 8 个块号，一个文件最大可以是 512 * 8 = 4096 字节（~4KB）

间接寻址

Unix v6 文件系统仅对大文件使用单间接寻址（存储有效负载块号的块）

- 检查`inode`中的标志或大小以了解它是小文件（直接寻址）还是大文件（间接寻址）
  - 小文件，inode中的每个块号存储有效负载数据
  - 大文件，inode中的前7个块号存储有效负载数据的块号，8号存储有效负载块号的块

- 假设使用所有8个块号进行单间接寻址，8 * 256 * 512 = ~1MB

对于更大的文件，我们采用双重间接寻址

支持最大文件大小：(7 + 256) * 256 * 512 = ~34MB

目录

目录层次结构

- 文件系统通常支持目录（文件夹）
- 一个目录可以包含文件和多个目录
- 目录是一个文件容器，需要存储其中包含的文件/文件夹，还具有关联的元数据
- 所有文件都位于根目录 \

> 常见文件系统任务：给定路径，获取文件内容

作为文件的目录

> 目录是一个文件容器

- 每个目录都有一个`inode`
- 目录的有效负载数据是关于它包含的文件的信息列表
- 目录的元数据是关于它的信息，例如它的所有者
- `Inode`可以存储一个字段，告诉我们某物是目录还是文件

我们可以在文件实现之上分层支持目录

代表目录

设计决策：Unix v6 文件系统使用目录有效负载包含该目录中每个文件/文件夹的16字节条目

- 前14字节是名称（不一定以null结尾）
- 最后两个字节是编号

从根目录开始查找

链接：硬链接和软链接

硬链接

- 两个不同的文件名可以解析为相同的编号
- 如果您想要一个文件的多个副本而不必复制其内容，则很有用
- 如果更改其中一个的内容，则所有内容的内容都会更改
- inode 存储解析为该 inode **i_nlink**的名称数。当该数字为 0 且没有程序正在读取它时，文件将被删除
- 我们在这里描述的称为*硬链接*。Unix（和 Linux）中的所有普通文件都是硬链接，两个硬链接就它们指向的文件而言是无法区分的。换句话说，没有“真正的”文件名，因为两个文件名都指向同一个 inode。
- `ln` 在 Unix 中，您可以使用命令创建链接

软链接

- 除了硬链接，Unix 文件系统还具有创建*软链接*的能力。软链接是一种特殊的文件，它包含另一个文件的*路径* ，并且没有引用编号。
- 软链接可以“中断”，因为如果它们引用的路径不存在（例如，文件实际上已从磁盘中删除），则链接将不再有效。
- 要在 Unix 中创建软链接，请使用`-s`带有`ln`.
- 当我们创建一个软链接时，`ls`给我们原始文件的路径
- 但是，原始文件的引用计数保持不变
- 同样，通过任一文件名更改文件内容会更改文件。
- 如果我们删除原始文件，软链接就会断开！软链接仍然存在，但它指向的路径不再有效。如果我们删除了文件之前的软链接，原始文件仍然存在。

文件系统调用

文件系统交互

程序员角度与文件系统交互

- 系统调用
- `int open(const char *pathname, int flags)`

  `int open(const char *pathname, int flags, mode_t mode);`
- `int close(int fd);`
- `ssize_t read(int fd, void *buf, size_t count);`
- 文件描述符（当前打开文件的票号）

  > 文件描述符只是整数

  每个文件都默认提供三个特殊的文件描述符
  - 0：标准输入 - STDIN_FILENO
  - 1：标准输出 - STDOUT_FILENO
  - 2：标准错误 - STDERR_FILENO

操作系统数据结构

- 文件描述符表和文件描述符

1. Linux为每个活动进程维护一个数据结构，这些数据结构称为`process control blocks`，它们存储在`process table`
2. PCB存储很多东西（启动它的用户，启动时间，CPU状态），其中就包括`file descriptor table`

3. 文件描述符是一个整数，它是该表的索引

   描述符0、1、2是标准输入、输出、错误；文件描述符3以上没有预定义的含义

- 创建和使用文件描述符
  - 文件描述符是通过系统调用与资源交互所需
  - 许多系统调用分配文件描述符
    - 读取：打开一个文件
    - pipe：在进程之间创建两个单向字节流（一个读，一个写）
    - accept：接受一个TCP连接请求，返回描述符给新的socket
  - 分配新的文件描述符时，内核选择最小的可用编号
- 文件描述符与文件表条目
  - 文件描述表中的条目只是指向文件表目的指针
  - 表中的多个条目可以指向同一个文件表条目
  - 不同文件描述符表（不同进程）中的条目可以指向同一个文件表条目
- 文件表详细信息
  - 每个进程维护自己的描述符表，但有一个系统范围的打开文件表；这允许在进程之间共享文件资源
  - 三个PCB中的描述符0、1、2分别为相同的三个打开文件命名
- 虚拟节点
  - 每个打开的文件条目都有一个指向`vnode`的指针，`vnode`是一个包含有关文件或类文件资源的静态信息的结构
  - `vnode`是内核对实际文件的抽象：它包括关于它是什么类型的文件、有多少文件表条目引用它以及执行操作的函数指针的信息
  - `vnode`的接口是独立于文件系统的，但它的实现是特定于文件系统的；任何文件系统（文件抽象）都可以将它需要的状态放在`vnode`中（inode编号）

系统调用执行

> 操作系统执行普通用户程序无法执行的私有特权任务，以及用户程序无法访问的数据

将一定范围的地址保留为“内核空间”；用户程序无法访问此内存。系统调用将使用内核空间并以“特权模式”执行，而不是使用用户堆栈和内存空间。但这意味着函数调用必须以不同的方式工作

多进程

核心思想：多个进程可以运行同一个程序

创建进程和运行其他程序

fork( )

> `fork( )`创建第二个进程，是第一个进程的克隆

- 父进程分叉出子进程
- 孩子开始执行下一条程序指令，父进程继续执行下一条程序指令
- 一切都在子进程中复制（除了PID不同）
    - 文件描述符表（增加打开文件表条目的引用计数）
    - 映射内存区域（地址空间）
    - 栈、堆等区域被复制
- 在父进程中，`fork( )`将返回孩子的PID（父母获取孩子PID的唯一方法）
- 在子进程中，`fork( )`将返回0

进程克隆

变量和地址会怎么变？父进程和子进程使用相同的地址来存储不同的数据？

- 每个程序都认为它已获得所有要使用的内存地址
- 操作系统将这些虚拟地址映射到物理地址
- 当一个进程`fork`时，它的虚拟地址空间保持不变
- 操作系统会将孩子的虚拟地址映射到与父母不同的物理地址

在`fork`时复制所有内存不是很浪费吗？

- 操作系统只会进行懒复制
- 它将让它们共享物理内存，直到其中一个将其内容更改为与另一个不同
- 这称为写时复制（仅在写入时制作副本）

为什么要使用`fork`呢

- 在`shell`中运行程序：`shell`派生一个新进程来运行程序
- 服务器：大多数网络服务器在不同进程中运行多个服务器副本

等待进程

```
1 pid_t waitpid(pid_t pid, int *status, int options);
```

父进程始终等待其子进程

- 一个已完成但未被其父进程等待的进程称为僵尸进程
- 僵尸进程占用系统资源（直到它们最终被操作系统清除）
- 在父进程中调用waitpid会收获子进程
    - 如果一个子进程还在运行，父进程中的`waitpid`会阻塞，直到完成，然后清除它
    - 如果一个子进程是僵尸进程，`waitpid`会立即返回并清理它

- 孤立的子进程被`init()`进程收养

execvp( )

```
int execvp(const char *path, char *argv[]);
```

进程间通信

管道

```
int pipe(int fds[]);
```

> `fds[0] = read`
>
> `fds[1] = write`

- `pipe`可以让进程进行通信
  - 父进程的文件描述符在子进程中被复制
  - 管道没有全局名称
  - 每个管道都是单向的

> 管道文件描述符在子进程中是重复的，我们需要关闭父进程和子进程的两个管道端

- 重定向进程I/O
  - 每个进程都有特殊的文件描述符`STDIN(0) STDOUT(1) STDERR(2)`
  - 进程假定这些索引用于这些通信方法

```
int dup2(int oldfd, int newfd);
```

> 制作文件描述符条目的副本并将其放入另一个文件描述符索引中

```
dup2(fds[0], STDIN_FILENO); // 将管道读取的文件描述符复制到标准输入中
```

信号

信号是一种通知进程事件已经发生的方式

- 有一个可以发送已定义信号列表：SIGINT、SIGSTOP、SIGKILL、SIGCONT等
- 信号实际上是一个数字
- 程序可以响应接收到的一种信号做一些事情
- 信号由操作系统或另一个进程发送
- 可以向自己或另一个进程发送信号

分段错误实际上是从操作系统发送到程序的信号（SIGSEGV）

- 当你尝试访问不在有效程序段中的内存地址时触发
- 默认行为是终止程序

```
pid_t waitpid(pid_t pid, int *status, int options);
```

默认行为是等待指定的子进程退出

- `WUNTRACED` - 等待一个孩子被阻止
- `WCONTINUED` - 等待孩子继续
- `WNOHANG` - 不要阻止

发送信号

```
int kill(pid_t pid, int signum);
// same as kill(getpid(), signum)
int raise(int signum);
```

- `kill`将指定的信号发送到指定的进程（命名不当）
- `raise`向自己发送指定的信号

接收信号

> 信号处理程序用途广泛，但充满潜在问题

- 将信号处理程序添加到我们的程序中：接收到特定信号时运行的函数
- 阻塞程序直到接收到信号

信号处理器

```
typedef void (*sighandler_t)(int);
...
sighandler_t signal(int signum, sighandler_t handler);
```

- signum是我们感兴趣的信号
- handler是接收到此信号要调用的函数的函数指针

> `SIGSTOP SIGKILL`不允许处理程序

`SIGCHLD`

> 当一个子进程改变状态时，内核向其父进程发送一个SIGCHLD信号
>
> - 这允许通知父母孩子在做其它工作时已终止
> - 我们可以添加一个SIGCHLD处理程序来清理子进程，而无需在父进程中等待它们

等待信号

- 信号处理程序允许我们做其他工作，并在信号到达时得到通知。但这意味着通知是不可预测的
- 一种更可预测的方法是在我们的程序中停止做其他工作并处理任何未决信号的时间

    - 好处：这允许我们控制何时处理程序，避免并发问题
    - 缺点：信号可能无法及时处理，我们的进程在等待时阻塞
- 我们不会有信号处理程序：相反，我们将在主执行中使用代码来处理挂起的信号

```
int sigwait(const sigset_t *set, int *sig);
```

> 不能等待`SIGKILL SIGSTOP`，也不能等待`SIGSEGV SIGFPE`等同步信号

- `set`：要等待的信号集的位置
- `sig`：应存储接收到的信号编号的位置
- 成功返回0，错误返回>0

信号集

`sigset_t`是一种用作位向量的特殊类型（32位int）它必须使用特殊函数创建和初始化

```
// Initialize to the empty set of signals
int sigemptyset(sigset_t *set);
// Set to contain all signals
int sigfillset(sigset_t *set);
// Add the specified signal
int sigaddset(sigset_t *set, int signum);
// Remove the specified signal
int sigdelset(sigset_t *set, int signum);
```

> `sigprocmask`函数让我们可以暂时阻止指定类型的信号。相反，它们将在块被移除时排队并交付

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- `SIG_BLOCK`：将其添加到要阻止的信号列表
- `SIG_UNBLOCK`：将其从要阻止的信号列表中删除
- `SIG_SETMASK`：使其成为要阻止的信号列表

虚拟内存

让进程使用虚拟地址，操作系统将决定它们实际是什么物理地址

> 我们需要一种非常快速的方法来将虚拟地址转换为物理地址
>
> - 内存管理单元(MMU)是CPU中执行此操作的特殊芯片
> - 非常块 - 否则会影响性能

虚拟与物理地址空间

- **地址空间**是整数地址的有序序列，从0开始
- **物理地址空间大小**受硬件限制（RAM）
- **虚拟地址空间大小**仅受指针大小限制（64位操作系统中的64位）

将虚拟页面映射到物理页面

- 虚拟地址是虚拟页码（所在虚拟页）和虚拟页偏移量（在页中的位置）的串联
- MMU可以从虚拟页号转到物理页号
- 相应的物理地址是物理页加上虚拟页偏移量（虚拟页和物理页大小相同）

竞争条件