L3: 虚拟文件系统

截止日期：2019年6月16日23:59:59

你的所有操作系统实验将在你之前的代码上完成，你只需要维护一份内容不断增加的实验报告(pdf格式)，其中每次实验的内容不建议超过2页A4纸。请在实验报告中描述你在实验中遇到的特别值得一提的事件，例如你代码的架构设计、特别精巧的实现、遇到印象深刻的bug等。

虚拟文件系统

在上一个实验中，我们有了一个操作系统的雏形——为每个线程实现了CPU的虚拟化。在上一个实验中，每个线程都可以自由访问I/O，直接向设备驱动程序发起调用即可：

```
device_t *tty = dev_lookup("ramdisk");
tty→ops→read(tty, offset, buf, length);
```

在这个实验中，我们实现设备的虚拟化，使线程能够通过文件系统API和文件描述符访问这套统一的接口访问设备和操作系统中的对象。

收到的作业

背景

在这个实验中，我们在多处理器分时多线程的基础上实现线程安全的虚拟文件系统(virtual file system, VFS) API，并且在VFS这一抽象层上实现若干不同的文件系统：

- 在存储设备(例如ramdisk)上支持完整文件和目录操作的文件系统
- 虚拟的procfs，提供一系列只读的、反映操作系统内部状态的文件
- 虚拟的devfs，将操作系统中的设备抽象为可以访问的文件，并为这些文件提供读写操作

实现完成后，系统中的多个线程就能通过文件系统API读写文件——至此我们离现代操作系统就只有一步之遥(下一个实验)：为每个线程附属一个独立的地址空间(通过虚拟存储实现)，线程就变成了我们熟知的进程，操作系统就完整了。

什么？Lab2？

我的Lab2好像运行不起来耶，那些API似乎还没调对，我只是交了个不怎么对的版本(和一本正经胡说八道的实验报告)给助教骗分了。

恭喜你，好好调试吧！有同学还调出了ICS PA时代累积下的klib bug呢。

实验描述

在本次实验中，实现vfs模块API，并且实现磁盘文件系统、procfs和devfs。它们能通过mount API挂载到文件系统中。以下将对实现的三部分进行描述。

开放实验

本次实验是开放的实验，不再提供框架代码，大家可以根据自己的能力和兴趣选择具体实现的部分。举例来说，vfs模块中包含若干文件系统的操作，但如果你选择用类似FAT的方式实现你的文件系统，则可以不必支持link系统调用。

vfs模块

所有对文件系统的操作都是通过调用vfs模块中的API完成的:

```
MODULE {
  void (*init)();
  int (*access)(const char *path, int mode);
  int (*mount)(const char *path, filesystem_t *fs);
  int (*unmount)(const char *path);
  int (*mkdir)(const char *path);
  int (*rmdir)(const char *path);
  int (*link)(const char *oldpath, const char *newpath);
  int (*unlink)(const char *path);
  int (*open)(const char *path, int flags);
  ssize_t (*read)(int fd, void *buf, size_t nbyte);
  ssize_t (*write)(int fd, void *buf, size_t nbyte);
  off_t (*lseek)(int fd, off_t offset, int whence);
  int (*close)(int fd);
} MOD_NAME(vfs);
```

除了mount之外，其他命令的行为建议实现成与Linux一致，但你可以做出简化，例如不允许blocking操作的interrupt、只返回一种类型的错误等。注意 `open` 返回的文件描述符是针对线程有效的，每个线程有自己独立的文件描述符表。关于文件描述符的行为，请参考教科书第39章。

文件系统

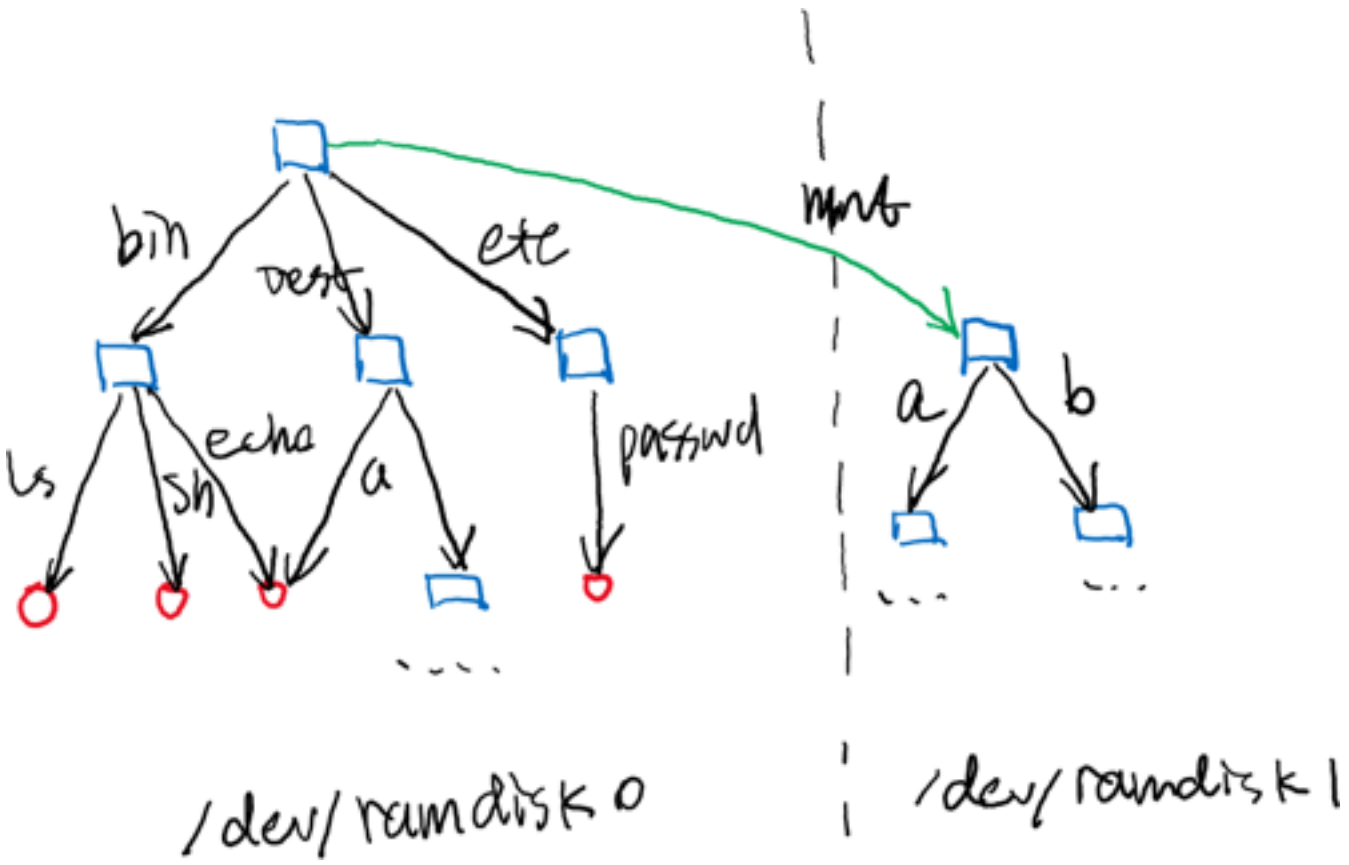
一个文件系统(用一个 `filesystem_t` 结构体表示)维护了树状的目录和文件结构，其中结构体里比较重要的成员有：

```
struct filesystem {
   ...
  fsops_t *ops;
  dev_t *dev;
};
```

每个文件系统都是基于一个I/O设备的(procfs和devfs没有对应的设备，因此dev为 `NULL`)， `fsops_t` 是具体的一些文件系统API的实现，一个合理的fsops定义：

```
typedef struct fsops {
  void (*init)(struct filesystem *fs, const char *name, dev_t *dev);
  inode_t *(*lookup)(struct filesystem *fs, const char *path, int flags
  int (*close)(inode_t *inode);
} fsops_t;
```

之前提到，文件系统是一个树状的数据结构。下图中，圆形节点表示存储数据的文件、方形节点表示目录(用目录文件存储)。



无论是普通文件还是目录文件，它们都对应了一个inode。注意到文件名并没有标记在节点上而是标记在边上，是因为文件系统文件名是存储在目录文件的目录项中的——在允许链接的文件系统中，一个文件可以有多个名字。

上图中，我们展示了一个系统中挂载了两个文件系统(`filesystem_t`)的例子：

- 一个文件系统实现挂载在 `/` ，对应的设备是 `/dev/ramdisk0` (ramdisk0设备)
- 另一个文件系实现挂载在 `/mnt` ，对应的设备是 `/dev/ramdisk1` (ramdisk1设

备)

inode

文件系统 `fileops_t` 中最重要的函数是 `lookup` ，它就给定相对于该文件系统根的路径名，返回路径查找后的inode。因此，为了解析一个全局的文件名，例如 `/mnt/a/test.txt` ，你就必须在系统中保存所有的mount points，路径解析如下：

- 首先，查找系统中的mount point table，找到 `/mnt` 对应的文件系统对象 `fs` ；
- 调用 `fs→ops→lookup(fs, "/a/test.txt", flags)` 获取inode。

对于一个具体的文件，无论它是普通文件还是目录文件，对文件的操作都是通过对它所对应inode的操作完成的：

```
typedef struct inodeops {
  int (*open)(file_t *file, int flags);
  int (*close)(file_t *file);
  ssize_t (*read)(file_t *file, char *buf, size_t size);
  ssize_t (*write)(file_t *file, const char *buf, size_t size);
  off_t (*lseek)(file_t *file, off_t offset, int whence);
  int (*mkdir)(const char *name);
  int (*rmdir)(const char *name);
  int (*link)(const char *name, inode_t *inode);
  int (*unlink)(const char *name);
  // 你可以自己设计readdir的功能
} inodeops_t;

struct inode {
  ...
  int refcnt;
  void *ptr;        // private data
  filesystem_t *fs;
  inodeops_t *ops; // 在inode被创建时，由文件系统的实现赋值
                   // inode ops也是文件系统的一部分
  ...
};
```

系统中打开的文件

最后，线程/进程的文件操作是通过文件描述符完成的，每个文件描述符背后都有一个“打开文件”的数据结构 `file_t` ：

```
typedef struct file {
  int refcnt; // 引用计数
  inode_t *inode;
  uint64_t offset;
  ...
} file_t;
```

这样，我们可以在进程/线程的数据结构中维护一系列“打开文件”的指针，这就是文件描述符！

```
struct task {
  ...
  file_t *fildes[NOFILE];
  ...
};
```

其中 `fildes[i] == NULL` 表示编号为 `i` 的文件描述符可用，你应该在教科书上看到过一样的代码！在这里，让我们整理一下 `open("/mnt/a/test.txt", flags)` 发生的事情:

- 首先，你的代码会解析 `path` ，找到它所在的文件系统 `fs` (类型是 `filesystem_t *`)，调用 `inode = fs->ops->lookup("/a/test.txt", flags)` 得到文件所对应的inode;
- 在线程的 `fildes` 中找到第一个为 `NULL` 的位置，使用 `file = pmm->alloc()` 分配一个新的 `file_t` ;
- 调用 `inode->ops->open(file, flags)` 打开文件。

此后，如果我们访问文件描述符，例如 `read(fd, buf, size)` ，我们会:

- 通过 `file = current->fildes[fd]` 找到对应的打开文件的数据结构;
- 通过 `file->inode->ops->read(file, buf, size)` 进行读取。

这么麻烦，有什么好处?

VFS的最大好处是可以兼容各种不同的文件系统实现。系统启动后，你的文件系统相关的代码将在操作系统中创建三个虚拟文件系统的实例:

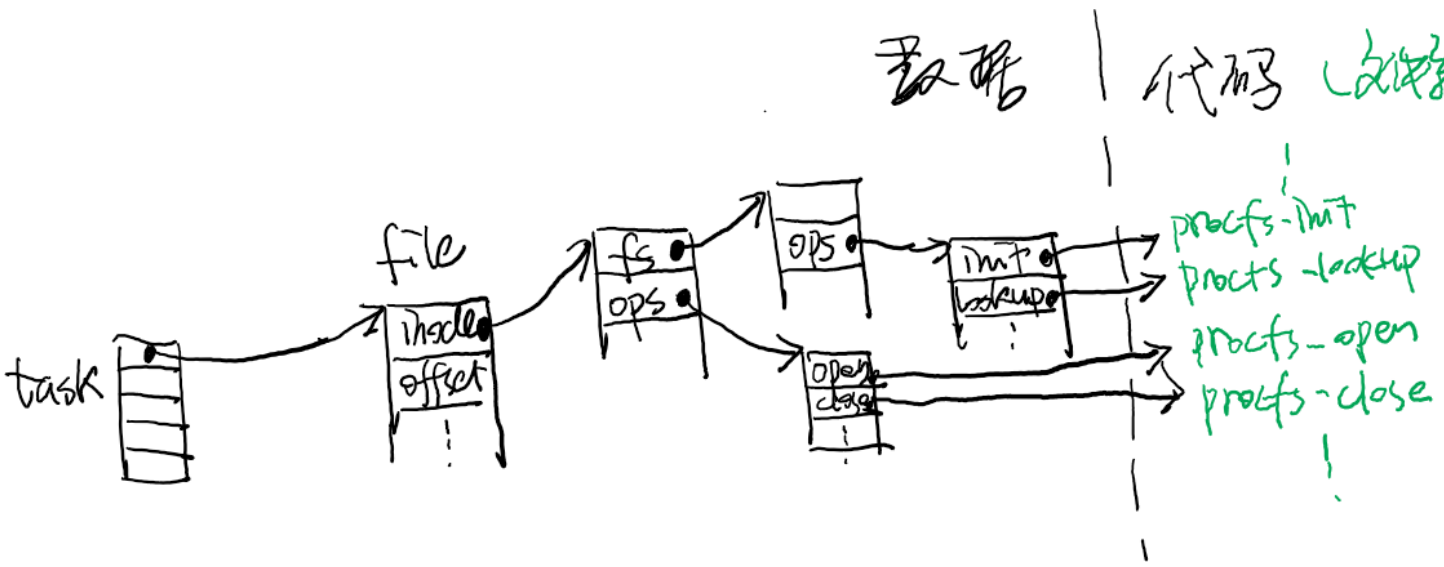
- procfs，挂载在 `/proc` 。procfs中包含每个线程的统计信息， `cpuinfo` 和 `meminfo` ，格式自定。
- devfs，挂载在 `/dev` 。devfs中包含 `random` , `zero` , `null` 设备，你也可以增加任意你想支持的设备。
- blkfs，挂载在 `/` ，维护一个树状的文件系统结构(可以参考教科书Filesystem

Implementation)。

有了vfs，兼容多个文件系统就变得非常简单了。以devfs为例，devfs里的文件是静态创建的，只需要实现 `lookup` ，就能实现 `/dev/ramdisk0` 等的解析。在每一个devfs inode中，inode的私有数据(`ptr` 域)都保存了指向设备(`dev_t`)的指针。因此，inode的读/写就非常容易了：

```
void devfs_read(file_t *file, char *buf, size_t size) {
  dev_t dev = file→inode→ptr;
  …
  ssize_t nread = dev→ops→read(dev, file→offset, buf, size);
  file→offset += size;
  …
}
```

聪明的你一定知道procfs应该如何实现了。实际上，这个实验中的VFS与Linux内核中的机制非常相似，以后你阅读Linux内核代码的时候，一定会感到几分亲切。用一张图回顾一下，是如何从文件描述符“路由”到对应的文件系统实现的。



一些约定

其中，除 `vfs→init()` 是由操作系统启动代码调用外，其他代码均可以由任何线程并发调用。你需要为每个内核线程维护一定数量的文件描述符(例如16个)，每个线程执行 `open` 操作都会按顺序返回编号最小的空闲文件描述符。线程被初始创建时不拥有任何文件描述符。

实现单线程shell

最后，还记得Lab2中的演示程序么？一小段代码可以实现命令行的“echo”功能：

```
void echo_task(void *name) {
  device_t *tty = dev_lookup(name);
  while (1) {
    char line[128], text[128];
    sprintf(text, "(%s) $ ", name); tty_write(tty, text);
    int nread = tty→ops→read(tty, 0, line, sizeof(line));
    line[nread - 1] = '\0';
    sprintf(text, "Echo: %s.\n", line); tty_write(tty, text);
  }
}
```

这其实是一个最简单shell的雏形，你可以把它丰富成如下的一个非常简陋的shell，但瞬间就让你的操作系统有了“能用”的感觉！

```
void shell_thread(int tty_id) {
  char buf[128];
  ...
  sprintf(buf, "/dev/tty%d", tty_id);
  int stdin = vfs→open(buf, O_RDONLY);
  int stdout = vfs→open(buf, O_WRONLY);
  while (1) {
    if (got_a_line()) {
      ...
    } else {
      ssize_t nread = fs→read(stdin, buf, sizeof(buf));
      ...
    }
    // supported commands:
    //   ls
    //   cd /proc
    //   cat filename
    //   mkdir /bin
    //   rm /bin/abc
    //   ...
  }
}
```

基本上，你只需要解析输入的命令，然后翻译成文件系统的API即可。

评分标准

本次实验是开放性的实验，因此你在实验报告中应当提供一些shell的使用指南，例如支持的命令、已知的bug等。评分时，助教会运行你的操作系统，在多个终端之间切换，在每个终端里都执行一些命令。你必须实现的部分有：

- 一个有文件和目录结构的文件系统，它作用于ramdisk上，并且初始时有一定的数据(initrd)。
- proc目录下有`/proc/[pid]`的统计信息(例如被调度的次数、时间等)；`/proc/cpuinfo`和`/proc/meminfo`。
- `/dev/tty1-4`，`/dev/ramdisk0`，`/dev/ramdisk1`。

我们离一个完整的操作系统有多近?

现在我们已经有了可以打开、关闭、读写文件的线程，线程又可以创建和管理其他线程。很自然，文件描述符就是线程的一种资源。同理，我们可以给线程增加任意的资源，比如一个独立的地址空间，并且运行一个用户态的程序。哈！这样就有了进程了。进程在进行系统调用时，同样由 os→interrupt 捕获进行处理，你也许已经想好fork, execve, exit的实现了……