操作系统 L1-L3 实验报告

张天昀 171860508

2019 年 6 月 10 日

L3 虚拟文件存储

总共写了 4404 - 2647 = 1757 行……主要的内容有：

- `./src/vfs.c`：虚拟文件系统，283 行；
- `./src/shell.c`：~~好看~~漂亮多功能 shell，349 行；
- `./src/file.c`：inode 文件树相关 API，42 行（全都是递归挺好写的）；
- `./src/filesystems/naive.c`：naiveFAT 文件系统，523 行（写到疯）；
- `./src/dev/ramdisk/initrd.img`：文件系统磁盘镜像。
  - 用 `vim :%!xxb` 纯手工手写写出来的
  - 内置了 3 个可读写文件和 1 个文件链接，~~包爽~~

naiveFAT

经过和同学~~吵架~~数个小时之后，我终于搞懂了为什么不能直接内存访问而是要通过 `dev->ops` 来访问数据，所以~~抄袭~~借鉴 FAT32 实现了一个 naiveFAT 文件系统。

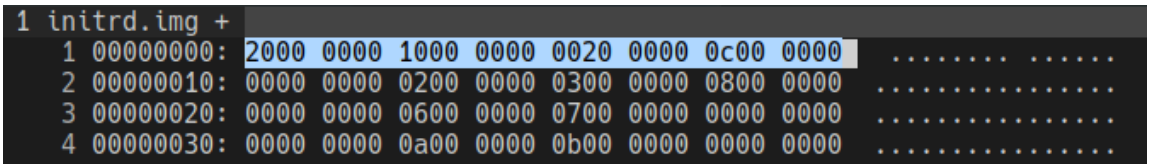

图 1:

naiveFAT 格式的磁盘前 16 字节是四个 `int32_t` 型元素，分别代表：

- 每个单元（簇）的长度，必须为 32；
- 文件跳转表的起始位置；
- 数据区的起始位置；
- 最小可用单元（簇）号。

随后跟跳转表，每个表项均为 `int32_t`，指示下一单元（簇）号。

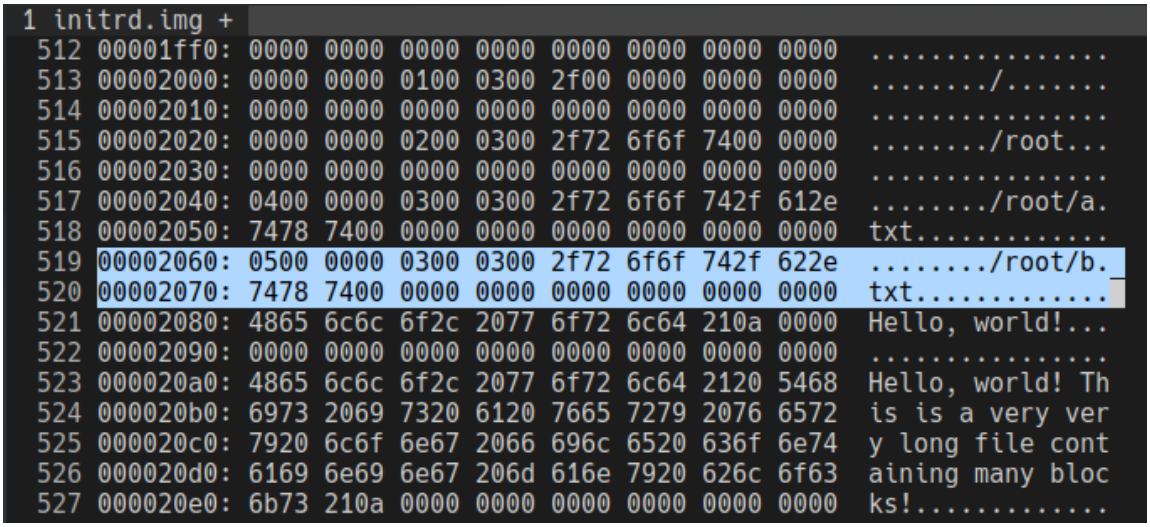

图 2:

对于目录类型的数据簇，每个单元（簇）由 4 个部分组成：

- 一个 `int32_t`，指示首块数据的单元（簇）号；
- 两个 `int16_t`，分别代表文件类型和文件权限；
- 24 字符的字符串，代表文件的相对地址。

sHELL

终端中可使用的指令有：

- `help`：打印指令列表；
- `ping`：输出 `pong`；
- `echo`：复读机；
- `ls`：列举当前目录下的文件；
- `pwd`：打印当前目录；
- `cd`：切换目录；
- `cat`：打印文件；
- `write`：向文件内写数据（会覆盖原有数据）；
- `append`：向文件结尾添加数据；
- `link`：建立文件链接；
- `mkdir`：创建文件夹；

- `rmdir`：删除文件夹（内容必须为空）；
- `rm`：删除文件（不可删除文件夹）。

系统启动后，共有四个文件系统：

- `/`：根目录，使用 naivefs，读取初始化内存盘 `ramdisk0` 的内容；
  - 映射 `kernel/dev/ramdisk/initrd.img` 中的磁盘镜像；
  - `initrd.img` 是用 `vim` 手写出来的二进制镜像（酸爽）；
  - `/root` 内有 `a.txt`、`b.txt`、`main.cpp` 三个文本文件和指向 `main.cpp` 的文件链接 `main.link`；
- `/mnt`：使用 `naivefs`，对应硬件为 `ramdisk1`，内容为空；
- `/dev`：使用 `devfs`，对应各个设备，可读可写；
- `/proc`：使用 `procfs`，对应各个进程，只读不可写。

终端支持的特性(因为篇幅限制报告里就没贴运行截图……可以访问https://doowzs.com/docs/42-os2019/oslabs-l3-report/来查看有截图的版本)：

- 终端所有指令均可使用相对路径或绝对路径；
- 通过对`/dev` 下的硬件进行直接读写可以进行 tty 间的通信；
- `/proc` 下保存进程信息，可通过`/proc/self` 读取本进程信息：
- `write` 和 `append` 提供两种不同的文件操作，但两种操作最后都会加上换行符；
- 支持文件链接（其实是类 FAT 文件系统上模拟出来的硬链接）；
- 对链接的文件进行修改，所有文件信息都会同时更新；
- 支持简单的错误处理（如删除不存在的文件、重名文件等）；
- 支持简单的文件权限检查（如`/proc` 为只读）；
- ……

L2 多线程

L2 太难了 + 要素过多，不知道该从哪里写起。

请注意我的实现里的 `printf` 是没有上锁的所以如果直接不上锁打印结果可能会很混乱。

概要

本次实验新增的内容有：

- `os.[ch]`：定义 `handler` 结构体和中断处理函数；
- `semaphore.[ch]`：信号量和相关函数；
- `thread.[ch]`：多线程调度、信号量等待。

所有的数据结构全都是链表，基本没有容量问题。

信号量睡眠

为了实现信号量等待时的睡眠，我分别尝试了以下方法：

- 设置状态为睡眠然后 `_yield()`，会因为此时别的进程已经发出信号但没有收到而死锁。
- 设置状态为即将睡眠然后 `_yield()`，会在 `_yield()` 前一刻被中断（或者被其他处理器抢先处理）导致切换到了即将睡眠的线程然后 boom。

~~最后我成功浪费了半个五一假期来解决这个问题导致的各种奇葩死锁和 bug。假期结束后在床上突然领悟到只要用一把大锁保证程序进入 `sem_wait` 后一定能够成功睡眠（原子性）即可~~。

这个美妙的用 `int $0x80` 进行系统调用来睡眠的方法成功触发了各种奇怪的错误然后继续浪费了我两天。于是我又修改回了原来的、最傻的、simple is best 的实现方式，能触发神秘 bug 的系统调用版信号量存放在 `L2` 分支中，提交的分支使用 `L2-no-syscall` 分支的代码。

- 先释放信号量的锁 `sem->lock`（先释放以防止死锁），然后获取 `os_trap_lock`，阻止其他线程此时进入中断；
- 此时保存 `sem` 的地址到 `task->alarm` 中（设置闹钟），允许其他进程在进入睡眠前唤醒自己（删除闹钟）；
- 最后，解开 `os_trap_lock`，调用 `_yield()`，此部分过程中如果发生时钟中断不会产生任何影响；
- `_yield()` 处理过程中首先判断是否已经被唤醒，如果没有唤醒 (`task->alarm != NULL`) 则设置状态为睡眠 (`ST_S = sleeping`)，然后切换线程。

键盘中断

当 CPU 关中断时，IO 设备的信号仍然能够被接收到，为了进行处理，在中断处理函数中进行了如下的判断：

- 如果未持有 `os_trap_lock` 中断锁，则按照普通中断处理；
- 如果当前已持有 `os_trap_lock`，首先对中断类型进行判断：不允许中断过程中嵌套 `_yield` 和时间中断。判断嵌套的中断类型合法后，依序调用 handler，但不会调用无对应事件的的 handler（如 `save` 和 `switch`），最后返回中断前的上下文（即在嵌套中断不允许发生进程切换）。

这样就实现了简单的中断嵌套，允许在中断过程中被更高优先级的事件打断，快速处理后返回上一层中断继续处理。

删除 task

为了减轻 L3 工作量我就先把这部分做了……删除一个 task 的顺序如下：

- 获得 `os_trap_lock` 中断锁，设置自尽变量 `task->suicide`；
- 下一次这个进程停止执行时（进行调度时），从任务列表中毁灭自己。

那么这个过程的正确性就依赖于设置 `suicide` 变量后进程仍然会再次被 CPU 执行。如果进程当前正在运行 *running) 或者可被切换 (waken up)，那么下一次中断就会自灭；唯一需要特殊处理的是睡眠进程 (sleeping)。因此，我在时钟中断中添加了防死锁机制：每个一定时间唤醒所有的进程，强制让他们继续运行，这样就可以把睡眠的僵尸进程也拉出来灭了。

祖传 bug

本次实验总共发现并修复了很多个 PA 祖传 bug：

- 提供了 `memmove` 函数，不使用额外内存。
- 修正了 `printf` 打印-2147483648 会因为取反整数溢出，输出 10 个符号的 bug。
- 修正了 `memset` 赋值为负会因为有符号数移位右边补 1 导致赋值错误的 bug。

L1 内存分配

数据结构

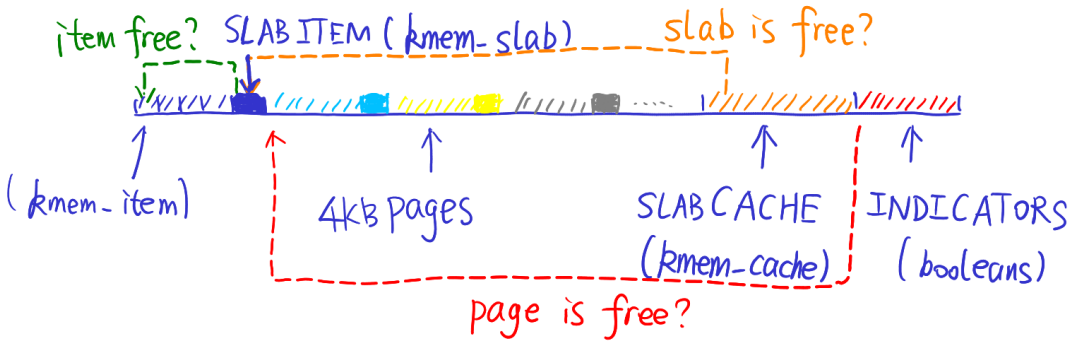

图 3: 内存区域划分与指针链接

为了防止在各种坑人的内存测试中挂掉，我采用了 Slab 内存分配方式，每种内存大小对应一个 cache，每个 cache 有多个 slab 对象，每个 slab 对象对应一个 4KB（或者更大的）内存块，包含多个内存对象。

整个内存被分为三个区域（对应图中从左往右）：

- 页面区：可分配的内存空间
- 缓存区：分配 slab cache 对象
- 指示器：用 bool 类型指示对应的内存块是否可用

每个 slab cache 为 `kmem_cache` 类型，包含两个链表，分别对应可用的 Slab 和已经全部占用（满的）slab。每个 slab 为 `kmem_slab` 类型，也分别保存两个链表，指向可用的内存对象和不可用的内存对象。为了回收方便，我在每个内存对象的前面贴了一个 `kmem_item` 类型的数据，指向 slab。

分配、回收过程

内存分配时，先计算所需 `size + kmem_item` 的大小，找到对应大小的 cache。如果对应的 cache 有空闲的 slab，直接使用，否则进行增长操作。如果所需大小小于 512B 则分配 1 个页面，否则分配 8 个对应对象所需要的内存页面数。在空闲的 slab 中取出一个空闲的内存对象，将指针加上 `kmem_item` 对应的偏移量返回给用户。

回收内存时，将指针减去对应的偏移量，得到 `kmem_item` 的指针，然后取出 `kmem_slab` 的地址，将该内存对象重新标记为可用即可。

接口封装与自旋锁

内存的分配、回收在进入 `alloc` 和 `free` 函数时直接上自旋锁，先计算所需大小，然后调用具体执行的函数 `kmem_alloc` 与 `kmem_free`，内部调用返回后再解开自旋锁，返回获得的结果。

存在的问题与解决方案

在某位超级大佬提供的测试代码中我的~~无懈可击的完美~~程序挂了，因为大佬的测试代码分配的内存大小随机，而我的代码每种大小都会单独生成 cache，很快 cache 分区就爆了。

解决方案很简单，对于每次内存分配，都找到不小于所需要的大小的 2 的次幂。这样 cache 的种类就非常少了（10 种即可覆盖 1KB 以下的所有需求）。而由于对 `alloc` 和 `free` 函数进行了封装，修改起来也非常简单。只需要在进行内部调用前修改所需大小的值，把修改后的值传入调用即可。例如：

```
void* alloc(size_t size) {
    lock();
    size_t real_size = cal_real_size();
void *ret = kmem_alloc(real_size);
    unlock();
return ret;
}
```