Lab7 实验报告

习题一

文件系统结构

buffer 层

为了提高 sd 卡的 IO 速度，在和 sd 卡进行直接读写之上构建一层 buffer 层（*bio.c*）。该 buffer 层使用一个队列来保存缓存 buf 块，该队列用一个链表维护。

同步与一致性：在 buffer 的实现中，设置了两把锁：对整个缓冲队列的进行控制的大锁 *cache lock* 和对单个缓冲块进行控制的小锁 *buf lock*。大锁控制的是整个缓冲队列的结构，保证在对队列中每个 buffer 进行计数器增减、释放、重分配操作的原子性，即队列结构在操作中能保持完整；每个缓冲块单独的小锁则是控制缓冲块和磁盘读写操作的原子性，即不会出现一个缓冲块的上一个磁盘读写命令没结束下一个磁盘读写命令就开始的情况（否则上一个磁盘读写的结果可能会唤醒下一个等待中的磁盘读写）注意到这两个锁控制的原子性需求时独立的，一个缓冲块在缓冲队列中位置的变化并不会影响其磁盘读写时的数据。

bget / brelse：

`bget` 函数需要在缓冲队列中找到对应 *blockno* 需要的缓冲块，如果没有对应的序列，则需要在没有被占用的缓冲块中选出一个来进行分配。同时，可以按照 LRU 算法，最久没有被使用过的缓冲块是最有可能不被使用的。于是，对这些缓冲块按照最后被引用的时间进行进行排列，由于一个缓冲块保存的数据是其上一次修改之后的数据，只要该缓冲块未被替换，其数据就一直有效。那么替换掉之后之后不被使用到的块就可以减少由于替换带来的有用数据的丢失，最后导致重复的情况。具体地，令越前面的块被释放的时间越近，当为某块未命的块分配缓冲块时从后往前寻找。

当找到响应的缓冲块之后，将缓冲块调用计数器加一。同时将开始时持有的队列锁释放，表示已经完成对队列结构的检索；同时持有缓冲块的小锁，表示即将开始对缓冲块的读写操作。

`brelse` 函数则将缓冲块的调用数减一。为了保证未调用块能按照最后被释放的时间排序，在释放后若没有被调用时需要把该块通过链表操作移到队列头。

bread / bwrite：

`bread` 和 `bwrite` 是上层访问 sd 卡数据块的唯一接口。`bread` 需检查该块的数据是否有效（该地址之前是否被读过）以对缓冲的数据进行读取优化。对 sd 卡的读写则调用 `sdrw` 函数来实现

bpin / bunpin：

在某些情况下，虽然对某快的写已经完成，但并不希望该块以直通的方式直接将数据写回到 sd 卡上。而此时如果不写回的话对应的缓冲块有可能被释放，导致数据修改没被执行。 `bpin` 可以将缓冲块调用次数加一，保证这个缓冲块不被替换；`bunpin` 则反之。

log 层

log 层的作用是屏蔽 buffer 层直接的写 sd 卡接口，为上层提供一个更安全的写磁盘的接口。

log 层使用了事件这一概念，上层需要按照不同程序、不同进程等特征分事件地调用 log 层提供的写功能。当一个事件结束后通过提交操作来告知 log 层；当当前暂存的 log 满了之后，再由 log 层统一地将当前暂存事件中所有的写命令转换为低层的 `bwrite` 的调用。

log 结构

log 可分为两部分：处于内存中的 log 结构体和处于 sd 卡上 log 块内未被写到 sd 卡上的 log 数据。

begin_op / end_op

在内存中的 log 被用于缓存未完成的事件。 `begin_op` 函数负责判断若不会超过 log 块的大小限制，在内存的 log 缓存中分配出一个新的事件；而 `end_op` 则负责将事件结束，并且在内存中的事件全部上传完成时调用 `commit` 函数将内存中的 log 写到磁盘中，同时清空内存 log 等待之后的事件。

commit / install_trans / write_log / write_head

上述函数被用于将内存中记录的事件写入 sd 卡中。

`write_log` 和 `write_head` 分别用于将内存 log 中的数据块和标号块写入到 sd 卡的 log 块中，而 `install_trans` 则用于读取 sd 卡的 log 块中块序号和块数据，写入到 sd 卡上。

`commit` 函数会调用以上函数，将内存中所有事件的 log 写回到 sd 卡上。具体地，会先调用 `write_log` 往 sd 卡 log 块写数据再调用 `writd_head` 写块序号；再调用 `install_trans` 写到真正位置上。当完成之后，需要将内存中的 log 个数清空再调用一次 `write_head` ，清空 sd 卡上的 log。

log 恢复

在写 log 的过程中，由于先写入了数据块再写入了序号块，当异常发生时只用三种可能：

- 发生在写序号块之前，此时序号块为空，读取序号块再写回不会有任何变化
- 发生在写完序号块未完成往实际地址的写入之前，此时 log 已经写完了，可以把 log 中所有的记录重新执行一次。

综上，`recover_from_log` 先调用 `read_head` 函数读取 sd 卡 log 块中的数据到内存，再和 `commit` 函数一样将内存中的 log 写到 sd 卡上。

inode 层

inode 层包括两种结构体：inode 和 dnode。在磁盘划分中把磁盘块分两部分：存放 dnode 和 inode 的数据块，bitmap 来记录某块是否被占用。在运行过程中，为了加快读写 inode 的速度（inode 访问频繁），也设计了针对 inode 的缓冲策略。在内存中会设计 inode cache 作为 buffer cache 之用。

同步与一致性

类似 buffer 层，在 inode cache 中也设置了两把锁：在外面的对 cache 整体结构（每个 inode buffer 存放的是哪个 inode、是否被占用）的一把大锁，以及对单独每个 buffer 的数据读写的一把小锁。类似地，这两把锁是相互独立的，即考虑读写某块 icache 时可以不需要持有大锁。

balloc / ialloc

`balloc` 和 `ialloc` 类似，都是枚举每个位置，在 bitmap 中查询该位置的块是否被分配过。在 `balloc` 中调用 `BBLOCK` 得到某块在 bitmap 中的占用位，`ialloc` 则用 `IBLOCK`。优化方面，先在 bitmap 中读取整个对应块的数据，再对每个 01 位单独处理。

为了使用在内存中的缓冲， `ialloc` 会通过 `iget` 返回对应 inode 在内存中的缓冲。

iget / iput

`iget` 在持有大锁后遍历 inode cache 查找响应编号的 inode 缓冲块并返回，如果不存在，则选择一个空的缓冲占用后返回。

`iput` 通过令 inode 缓冲块引用量减一释放本次的调用；如果本次释放之后该缓冲块的引用量变为 0，就表明该缓冲块目前不被使用，可能在之后被替换。inode cache 同样采用写回模式，在 inode buffer 被修改时并不写到 sd 卡中；只有当该 buffer 不被使用、可能被替换时才把数据通过 log 机制写回到 sd 卡上。

file 层

file 层为上层提供了关于文件的抽象，上层可以通过文件描述符进行读写操作，不用通过 inode 查找 dnode 来进行数据的读写。

file 层维护了一个存放 file 结构体的数组，每个 file 通过记录用于索引该文件的 inode 和当前进程对该文件中的读写位置 off 来完成对文件的顺序读写。

filestate / fileread / filewrite

由于在 inode 层中实现了从某个偏移量 off 开始对 inode 所指向的数据的读写以及 inode 的声明信息的读取，对于给定 file 只要对该 file 的 inode 以及 offset 调用响应的 `stati` / `readi` / `writei` 即可。

习题二、三

sys_exec

在当前进程中执行某个位置的可执行文件，并将传入的参数 “告诉” 该进程。

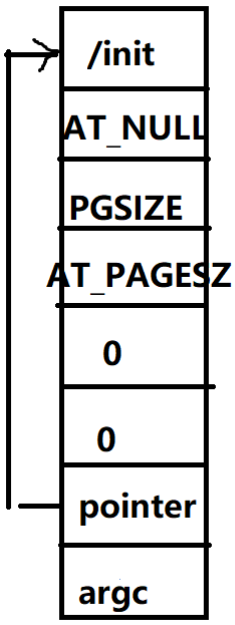
导入可执行文件

可执行文件通过先读取文件路径对应的 inode，再从 inode 中读取 elf 头，然后根据 elf 头中 e_phoff 的位置开始读取的 program table header。ph 给出了对应程序在内存中的位置、占用内存的大小和该段程序的长度。

在从 inode 对应偏置位置拷贝数据时，该段数据需要被放入某个虚拟地址。由于该虚拟地址并不在页表中，需要先调用 `uvm_alloc` 在页表中生成虚拟地址的映射关系。需要注意的是，生成的地址和 `execve` 程序所能访问是在高地址位（内核态地址空间）的，但页表中存放的是物理地址（低地址），而该程序被放入的地址也是低地址（用户态地址空间）。

`loadseg` 函数在完成页表的地址映射之后，从给定 inode 和 offset 中调用 `readi` 读取数据并写入到给定的地址中。

参数传入



如图是栈中参数的存放格式。通过 `uvm_alloc` 分配出栈空间之后，将对应数据压入栈中即可。

进程数据修改

完成上述步骤之后，修改当前进程的进程名、页表地址、栈指针及大小，之后再令 trap frame 中的 *ELR_EL1* 寄存器的值为程序运行的初始地址。这样，当 `sys_exec` 的系统调用完成之后，程序会返回到程序的运行地址开始执行程序。

结果展示

运行 `ls` 和 `cat init` 程序的输出如下：


```
dd if=obj/fs.img of=obj/sd.img seek=$((2048+128*1024)) conv=notrunc
1000+0 records in
1000+0 records out
512000 bytes (512 kB, 500 KiB) copied, 0.209499 s, 2.4 MB/s
qemu-system-aarch64 -M raspi3 -nographic -serial null -serial mon:stdio -drive file=ob
main: [CPU1] is init kernel
main: [CPU3] is init kernel
main: [CPU0] is init kernel
main: [CPU2] is init kernel
irq_init: - irq init
- mbox write: 0x7fd08
- mbox_read: 0x7fd08
- clock rate 50000000
- SD base clock rate from mailbox: 50000000.
- sdInit: reset the card
- Divisor selected = 104, pow 2 shift count = 6
- EMMC: Set clock, status 0x1ff0000 CONTROL1: 0xe6807
- Send IX_GO_IDLE_STATE command
- sdSendCommandA response: 0
- EMMC: Sending ACMD41 SEND_OP_COND status 1ff0000
- Divisor selected = 2, pow 2 shift count = 0
- EMMC: Set clock, status 0x1ff0000 CONTROL1: 0xe0207
- EMMC: SD Card Type 2 SC 128Mb UHS-I 0 mfr 170 'XY:QEMU!' r0.1 2/2006, #ffffffffdeadb
- LBA = 20800, SIZE = 1f800
main: [CPU0] Init success.
main: [CPU2] Init success.
main: [CPU3] Init success.
main: [CPU1] Init success.

******************
* sb.size:       3e8
* sb.nblocks:    3ad
* sb.logstart:   2
* sb.nlog:       1e
* sb.inodestart: 20
* sb.bmapstart:  3a
******************

mknodat: path 'console', major:minor 1:1
stdout 1
stderr 2
init: starting sh
$ ls
.              4000 1 512
..             4000 1 512
cat            8000 2 40888
init           8000 3 49008
ls             8000 4 46080
mkfs           8000 5 47792
sh             8000 6 59256
console        0 7 0
```