rcore with LFS 结题报告

李一苇 liyw19@mails.tsinghua.edu.cn

李之尧 lizhiyao19@mails.tsinghua.edu.cn

实验概述

本项目借助 rcore-tutorial[2]和 rcore-tutorial-OS20[3]的已有工作，使用 Rust 语言构建了基于 RISC-V 的操作系统原型。在构建基础上，重点探索了 rcore 的文件系统，为现有 OS 提供了多文件系统支持，并实现了一个不同于 ucore 和 rcore 现有的 Simple Fast System (SFS)[4]的全新的日志文件系统 Log-structured File System (LFS)[1]。本项目还基于 LFS 和 SFS 进行了一些性能比较。

实验目标

使用 Rust 实现基于 RISC-V 的操作系统原型，具体包括：

- 支持进程/线程切换与调度，包括进行基于时间片的抢占式调度和进程提出的 yield()；
- 支持基于页表的虚实地址转换和动态物理内存的分配；
- 实现与 rcore-fs 兼容的驱动体系，支持将块设备挂载到 qemu 上访问；
- 支持多文件系统，能跑通现有的 rcore-fs-sfs；
- 其他必须的 kernel 功能：处理系统调用、解析 ELF 文件等。

根据参考文献[1]，实现了基于日志的文件系统 LFS，具体包括：

- 用追加写的方式实现基本 FS 的操作：文件/目录的创建、修改、删除、硬链接；
- 兼容 rcore-fs 中的 VFS 接口；
- 实现以 Segment 为粒度的垃圾回收。

基于实现的 LFS 和已有的 SFS 系统进行评估，包括：

- 功能性验证：编写脚本对 LFS 的各个功能单元进行测试；
- 性能测试：使用 SimpleSSD 模拟器对不同系统生成的读写 trace 进行性能评估，进行延时和能耗的比较。

实验环境

本项目在 Linux 环境，rust 工具链版本 nightly-2020-04-23 下编译通过。为了减少重复劳动，本项目还复用若干 rcore-os 的提供的库，包括 rcore-fs，rcore-fs-sfs，riscv，device_tree 和 virtio-drivers，均位于 https://github.com/rcore-os 目录下，在 2020 年 5 月 31 日的最新版可以通过编译。

最终的操作系统在 qemu 模拟器（版本 4.1.1）上可正常运行，包括 boot、预设进程调度和文件读写。

实验过程简述

大致分工安排

1-4 周，两人对现有 rcore 和 rcore-tutorial 进行大致调研，明确了操作系统原型的实现目标；

5-9 周，李之尧借助 rcore-tutorial 文档，设计实现了操作系统原型，达到部分目标；

10-11 周，两人和本科生赵成刚等本科生同学进行交流后，复用了他们的驱动系统的实现；

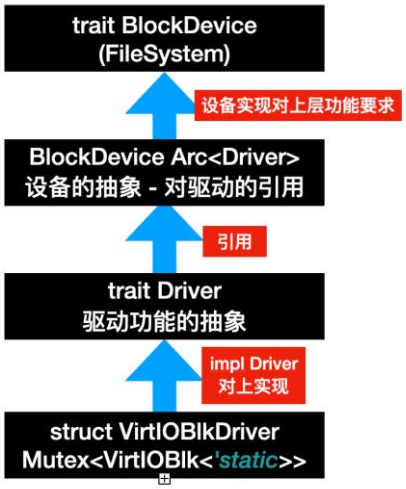
12-15 周，李一苇基于日志文件系统文献，设计实现了 LFS，并完成了测试和评估工作。

14-15 周，两人撰写结题报告等。

下面对上述进度进行概要说明：

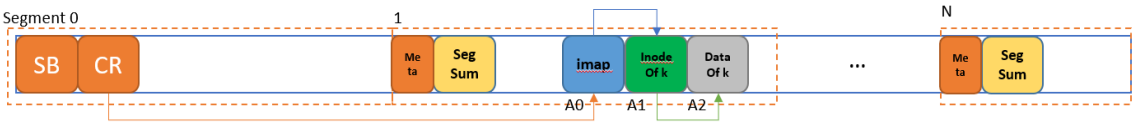
块设备驱动

现有的 rcore-tutorial 不具备完整的驱动系统，生成的磁盘镜像被直接链接到内核中，其地址区间由两个内建的符号标识，通过 MemBuf 结构伪造一个 Device 实现进行文件的读写。通过复用 Virtio-drivers 库，本项目实现了块设备的 Device 接口，并支持块设备的 VirtIO 协议，使得文件磁盘镜像可被 Qemu 直接挂载，流程更为自然。本项目还对生成的镜像进行 qcow2 格式压缩。实现结构图如下所示：



LFS

本项目设计了如下所示的日志文件系统格式：



按 Segment 划分，第 0 个 Segment 存储 Super Block (SB)，即文件系统元信息，以及 Checkpoint Region (CR)，对一段时间内读写的 segment 进行跟踪，CR 中还存储到所有 imap 的物理块位置。其余 Segments 正常存储 inode 和数据块。在各 Segment 内部，Meta 结构存储目前使用的 inode 数、blocks 数等信息，Segment Summary Block 用于存储数据块编号到 inode 和 dataentry 编号的映射。

定位文件时，首先通过 CR 检索并整合最新的 imaps，imap 结构记录 inode 编号到 inode 所在物理块的映射。通过文件 inode 编号找到对应 inode 所在的物理块位置，成功访问 inode 之后的流程同 SFS 几乎一致。

为了实现追加写的功能，任何对 inode 及其数据块的修改，都要用新分配的数据块，经过 merge 后补加到日志尾部，以利用好顺序读写的优势。更进一步，本项目在 rcore-fs 的 Dirty 结构上扩展了新的 Stale 位，由设计人员手动维护，用以表示该数据块返回时是否采用追加方式还是原地写回的方式。通过调整 Stale 位是否设置，可以平衡随机访问和读写容量的开销。在未来，我们会考虑使用一些自适应的方法进行调节。

评估工作

正确性验证上，本项目使用脚本随机生成大量文件并读写，然后用 fuse 工具的 zip 选项，分别打包成 SFS 和 LFS 的镜像。随后，使用 unzip 拆成对应的目录，检查这三个目录（本地脚本创建、SFS 恢复和 LFS 恢复）是否一致。下图是使用 tree 工具进行对比的部分结果，发现两者是一样的。

```
disk
├── dir0
│   ├── dir32
│   │   ├── dir39
│   │   │   ├── dir67
│   │   │   │   └── dir87
│   │   │   └── dir92
│   │   ├── dir45
│   │   │   ├── dir49
│   │   │   │   ├── dir52
│   │   │   │   │   ├── dir75
│   │   │   └── file85
│   ├── file46
│   ├── file70
│   └── file80
├── file12
├── file17
├── file25
├── file3
├── file48
├── file6
└── file7

56 directories, 44 files
```

```
disk-sfs
├── dir0
│   ├── dir32
│   │   ├── dir39
│   │   │   ├── dir67
│   │   │   │   └── dir87
│   │   │   └── dir92
│   │   ├── dir45
│   │   │   ├── dir49
│   │   │   │   ├── dir52
│   │   │   │   │   ├── dir7
│   │   │   └── file85
│   ├── file46
│   ├── file70
│   └── file80
├── file12
├── file17
├── file25
├── file3
├── file48
├── file6
└── file7

56 directories, 44 files
```

另外，将 LFS 生成的镜像挂载到 OS 原型中，OS 能正确识别 magic number 并打开文件，访问磁盘内 binary 文件。二进制文件进一步对文件系统进行创建文件、读写文件操作，并对结果进行比对，截图如下：

```
mod memory initialized
mod interrupt initialized
mod driver initialized
Loading a LFS disk
content of this directory:
.
..
test
write
notebook
read
user_shell
temp111
writecreate
read2
hello_world

mod fs initialized

Task1: write to a new file
created "temp123"
ready to write file temp123
write to file 'temp123' successfully...
read from file 'temp123' successfully...
content = Hello world!
Thread 1 exit with code 0

Task2: write to a existing file
ready to write file temp111
write to file 'temp111' successfully...
read from file 'temp111' successfully...
content = Hello world again!
Thread 2 exit with code 0

Task3: drop inode and reopen file 1
ready to read file temp123
content = Hello world!
read from file 'temp123' successfully...
Thread 3 exit with code 0

Task4: drop inode and reopen file 2
ready to read file temp111
content = Hello world again!
read from file 'temp111' successfully...
Thread 4 exit with code 0
src/process/processor.rs:98: 'all threads terminated, shutting down'
```

性能上，通过 Rust info 级日志记录两个文件系统进行相同文件操作后对设备产生的读写 trace。基本信息如下，可见 LFS 在读写长度开销上有优势，但相比 SFS 存在 53%的存储开销。主要是由于每个 segment 的 meta 信息，inode 占用一整个 block 的空间，以及不清空的删除操作等。

	Write count	Read count	IO Transfer length	Image capacity (compressed)
LFS	34582	11419	47120712	4.9MB

SFS	34215	22442	93869060	3.2MB

脚本中使用的文件块主要为 1M-4M。但由于脚本产生的为空数据文件，所以压缩率很低。

同时采用 SimpleSSD[5]仿真工具测试不同文件系统在 SSD 上的表现，收集到的数据如下：

	LFS	SFS
Simulation Tick	1215821831777725	1221120730998634
Min Latency	2.871473ms	3.376592ms
Max Latency	3052.497074ms	3983.285137
Average Latency	1019.119791	1481.942723
Stdev Latency	3425.016061	1460.266471
Event Handled	156314	184877

LFS 由于采用更多的顺序读写，因此延迟的平均水平相对更低，但波动变化较大，原因未知。但 SSD 上总体上两者的周期数没有太大区别，推测是 SSD 模型的 page cache 等措施有效消除了随机写的开销。

相关工作参考

[1] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 26-52.

[2] rcore-tutorial. https://rcore-os.github.io/rCore_tutorial_doc/

[3] rcore-tutorial-OS20. https://os20-rcore-tutorial.github.io/rCore-Tutorial-deploy

[4] SFS Analysis. https://os20-rcore-tutorial.github.io/rCore-Tutorial-deploy/docs/lab-5/files/rcore-fs-analysis.pdf

[5] Jung M, Zhang J, Abulila A, et al. Simplessd: modeling solid state drives for holistic system simulation[J]. IEEE Computer Architecture Letters, 2017, 17(1): 37-41.