增量型安全文件系统 SFS

引言

文件系统是计算机系统中为用户提供数据存储服务的一个很好的选择。然而，根据本文有限的调研，大部分通用文件系统在设计上都没有特别考虑文件的永久性存储。这里的永久性指的是，文件及其每一个版本都在底层存储介质上永久保存。因此，“删库跑路”等事故时有发生。例如，一个对高层不满的系统管理员可能会通过运行`rm -rf /`等命令来删除系统中的所有文件，从而表达他的不满。由于被上述命令删除的文件不难恢复，所以，稍有经验的系统管理员可能还会向文件系统写入垃圾数据来覆盖实际上未被清除的数据。此外，菜鸟系统管理员亦可能受到他人的教唆而运行这样的命令，为企业带来损失。此外，对于数据库系统等其他软件系统，也有同样的安全风险。虽然合理的权限控制可以缓解一部分问题，但拥有最高权限的系统管理员仍然是危险的。虽然可以针对特定应用，设计特定的操作日志等永久性记录，但这些方案不具有良好的灵活性，同时不能很方便地迁移到其他系统。对此，本文认为只有在文件系统层面对文件进行增量的、永久性的存储，才能够同时确保安全性、透明性以及灵活性。由此可见，一个能够对文件永久性存储的文件系统是必要的。

注意到git [Git. 2018]是一个能够跟踪每一个文件版本的版本控制系统，而libfuse [libfuse. 2018]使得开发者可以很方便地开发文件系统，本项目在git仓库的基础上利用libfuse，设计并实现了一种增量型安全文件系统SFS（Secure File System）。

本文分为这样几个部分：“相关工作”介绍了前人的已有工作，“功能设计”和“具体实现”介绍了本项目的设计与实现思路，“测试”部分给出了对于本项目方法的正确性测试和性能测试以及讨论。最后，“结论”一节对全文进行了总结。

相关工作

Git [Git. 2018]是一个能够跟踪每一个文件不同版本的版本控制系统。Libgit2 [libgit2. 2018]是一个C语言开发的操作git仓库的函数库，提供了易于使用的接口方便开发者操作git仓库。gitfs [presslabs. 2018]也是一个基于git的文件系统，不过其核心功能在于让用户能够很方便地以文件系统的方式来访问git仓库及其各个版本和分支。微软公司提出的GVFS [Microsoft. 2018]同样是基于git的文件系统，这个项目的关注点则在于访问大仓库时的性能，以及减少网络传输的数据量。Libfuse [libfuse. 2018]是一个用户态文件系统函数库，使得开发者可以在用户态开发及调试文件系统，而不用在内核态编写和调试内核模块。Fio [Jens Axboe. 2018]是一个灵活的性能测试工具，在本项目中被用来测试本项目的I/O性能。

功能设计

整体系统使用用户态文件系统libfuse实现，利用libgit2函数库操作git仓库。

具体而言，一个使用git管理的文件仓库可分为工作区、缓冲区和版本库三个部分。工作区即文件目录中直接可见的文件，缓冲区即调用`git add`后暂存的文件，版本库即`.git`目录中的信息。要使用git实现文件系统，最简单的方式是将各项文件操作直接转发为对于工作区的文件操作，只有需要保存增量副本时，才调用git程序将指定文件提交（在git中，被称为commit）到版本库。然而，这样实际上将每个文件在工作区和版本库中各存了一份副本，空间开销很大。为了优化空间开销，本项目选用libgit2函数库直接操作版本库。对于大多数操作，例如列出目录内容、获取文件属性等，可以直接在

版本库中操作，而不必提取具体的文件到工作区。当打开（`open`）一个文件时，为了避免直接对版本库进行操作带来的额外时间开销以及不必要的复杂性，本项目将被打开文件的内容缓存到临时文件中。读（`read`）、写（`write`）、改变文件长度（`truncate`）操作直接对该临时文件进行；而关闭（`release`）时在版本库中直接构造一个提交（commit），包含该文件的最新内容，并删除该临时文件。

考虑到文件的打开与关闭操作可能比较频繁，而频繁地创建和删除临时文件可能开销过大，未来可以在关闭文件时不删除临时文件，而将临时文件作为版本库中文件的缓存，利用缓存淘汰算法（LRU等），结合用户配置的最大缓存文件总量等参数来提升性能。

除了在关闭文件时保存增量副本（commit）以外，根据安全性的不同，版本控制的策略也可以有不同的选择。对于文件读写操作，本项目的设计目前支持如下三种版本控制策略：

1. `commit-on-close`：文件每次被关闭时，将缓冲区中的文件数据提交到git仓库；
2. `timed-commit`：可设定一个时间间隔，定时将缓冲区中的文件数据提交到git仓库；
3. `commit-on-write`：文件每次被写入后，将缓冲区中的文件数据提交到git仓库；

以上三种策略的安全性能、时间和空间开销均依次增大。一般而论，`commit-on-close`已经可以提供足够的安全性；对于安全性要求较高的场合，可以尝试`timed-commit`，并根据安全性要求的不同，合理地选择时间间隔；`commit-on-write`安全性最高，但由于开销过大，只适用于安全性要求极其严苛的场合。

对于其他写操作，包括创建目录、删除目录、创建文件、删除文件、修改文件权限以及重命名，每一次操作都会被提交到git仓库。

此外，本项目还提供了一些人性化功能，例如：

- 在挂载时可以指定SFS回滚到某个指定时间之前的最新版本，SFS会在此版本基础上新建一个分支继续存储，而不会改变原来的数据；
- SFS可以以只读模式挂载，防止文件被误修改而带来的不必要的版本回退；

同时，我们的文件系统支持并发操作。

具体实现

相关术语

本小节解释了一些`libgit2`中出现的术语。

- `git_repository`代表整个仓库；
- `index`可以认为是一个存储区，存放被`git add`命令记录的变更，被记录的文件的变更被添加到`index`；
- `blob`直接对应存放在仓库中的数据文件，可以认为是文件系统目录中的文件；
- `oid`指的是`git_object`的标识符，事实上是一个SHA-1值，20个字节；
- `tree`是一个树形数据结构，`tree_entry`是这颗树的节点，可能是没有子节点的文件，也有可能是还有子节点的目录；
- `commit`即git中的提交，一个`commit`与一颗`tree`有关，`commit`是整个仓库进行版本控制的基本单位，所有`commit`构成一幅有向图；
- `reference` / `branch`和git中概念一致，是多个`commit`形成的链。`branch`可被认为是有名字的`reference`。它们在本质上是一致的，在`libgit2`的内部实现也是一致的。

FUSE接口实现

本小节详细说明FUSE各个接口的具体实现。一般地，一个FUSE接口大致对应一个系统调用，但并非完全一致。

`create`

`create`会在git的`index`创建一个指定文件名的空文件，然后提交到git仓库。

`open`

`open`的语义为打开文件。前文提到，本项目使用临时文件的做法，因此`open`实现为从git仓库读取一个文件的内容，并将其写入临时文件。同时，还维护打开的文件相关数据结构。

`read`

`read`负责从文件中读取一块数据。读操作直接转发到对临时文件的读操作，提升了文件系统的性能，降低了开发复杂度。

`write`

`write`负责向文件写入一块数据。首先，写操作会将数据写入临时文件。然后，根据版本控制策略配置的不同，若此次写操作需要被提交到git版本库（开启了`commit-on-write`或`timed-commit`的时间间隔已达到），则会在版本库中利用临时文件中的最新数据构造一个`blob`，添加到`index`中，最后在git仓库创建一个提交。

`release`

在一个文件访问结束时，`release`会被调用。前文提到，本项目使用临时文件的做法，因此`release`操作会在版本库中利用临时文件中的最新数据构造一个`blob`，添加到`index`中，最后在git仓库创建一个提交，并删除该临时文件。

同时，还清理打开的文件相关数据结构。

`getattr`

`getattr`用于获取一个文件或目录的属性。访问权限、文件尺寸、是否为目录的信息，可以直接调用libgit2的接口获取。由于git中不存储用户信息，为了保证挂载SFS的用户有权访问其中的内容，我们令某文件的所属用户和所属用户组与git仓库（`.git`）目录一致。这样做有两点好处：如果此git仓库是原来就存在的，这样可以避免无权用户访问此分区；如果此git仓库是挂载时创建的，也能令挂载者成为此分区的属主。此外，git不存储根目录信息，因此令根目录的所有属性也与git仓库目录一致。

`readdir`

`readdir`列出一个目录中的所有文件或子目录的文件名和属性。调用libgit2的遍历`tree`函数，使其只遍历某一层目录，而不进行递归即可。遍历到目录中的每个文件或子目录时，收集其文件名和属性。

`unlink`

`unlink`主要实现删除文件的文件功能，具体到git实现如下：从仓库`repository`中取得代表当前暂存区的数据结构指针`index`；根据文件的路径`path`，移除`index`中相关项；最后把修改过后的`index`重建为`tree`，更新到仓库的`commit`中即可。

`truncate`

`truncate`主要实现修改文件大小的功能，单位为字节，若文件小则截断尾部多余的字节，反之则填充0。最初的简要实现是利用`mktemp`系统调用把当前文件备份到临时文件，修改临时文件满足要求后提交改到到git的`commit`中。上述操作会对造成磁盘读取不必要的性能损失，我们注意到libgit2中代表文件的结构`blob`可以从内存中创建，这样我们只需要把原始文件的`blob`载入到内存中，通过`memcpy`等操作即可完成对文件大小的修改并避免多余的磁盘读写操作。

`rename`

`rename`主要实现文件及文件夹名字的修改，具体到git实现如下：取出该路径对应的`entry`，判断输入路径对应的是一个文件还是文件夹。若输入路径对应一个文件，则将原文件对应的`entry`从`index`中删除，并将新`entry`插入`index`，新`entry`中除路径名（path）之外的其他成员变量都与原`entry`一致，路径名（path）改为输入中修改后的路径；若输入路径对应一个文件夹，则要在`index`里寻找所有以输入路径为path的前缀的`entry`，又由于所有`entry`在`index`中的存储是有序的（以path为关键字），所以只需要调用函数找到下标最小的、以输入路径为path的前缀的`entry`,之后不断地将下标增加一，直到某个`entry`的path的前缀不再是输入路径即可，之后就将找到的原`entry`都删除，新`entry`插入`index`即可。

`mkdir`与`rmdir`

`mkdir`和`rmdir`用户增删目录。由于git不将目录当作文件，而只是保存仓库中各个文件的路径，这造成：1. 一个目录中至少有一个文件；2. 增删文件时，其所属目录可以自动被增删。所以`mkdir`只需在目录中创建一个特殊文件；`rmdir`只需先检查目录中是否只剩下该特殊文件，若是再将该文件删除即可。我们在FUSE的各个接口处对所涉及的文件名做了转义，使用户不可能访问或操作此特殊文件，此特殊文件对用户实际上是透明的。

版本控制

在配置文件中设置类似与`%d-%d-%d %d:%d:%d`格式回滚的时间，即可回到距离时间最近的一次版本。回滚后我们会新开一个分支来表示当前的工作区，而使历史版本不至于丢失。具体来说，实现如下：

- 通过`branch_iterator`遍历所有分支，顺便统计出分支数目用以命名新的分支；
- 对于每个分支从后往前寻找满足时间要求的`commit`，记录最小值；
- 找到最近的`commit`，据此建立新的`branch`，将`HEAD`指过去；
- 更新`index`为`HEAD`所指向的`tree`。

整个过程最难的部分在于更新`index`，类似的操作可以用git中的`git reset`和`git checkout`完成，libgit2中也有类似的函数接口。可调用了接口却没有任何效果，经过多方排查和阅读原码，猜测可能是`reset`和`checkout`都会对两个`commit`之间不同文件的工作区和暂存区进行操作，而我们的架构设计中没有工作区的这可能函数不能正常工作，于是我们简要的自己实现`reset`功能才到达了目的。

需要注意的是，在libgit2中表示时间使用的是`time_t`，其含义是1970年1月1日0时0分0秒到某一时刻的秒数。`time_t`数据类型本质上是`long`类型，它所表示的时间不能晚于2038年1月18日19时14分07秒。

路径转义

由于git仓库中不能保存特定文件名的文件，例如名为`.git`的文件，本项目在FUSE文件操作及其对应的git仓库操作之间，加入了路径转义。对于各类文件和目录操作，路径转义将路径中的每一项附加一个前缀`$`；特别地，在列目录操作时，返回给调用者的文件名是恢复后的文件名，即对于每一项，删除前缀`$`。这样确保了SFS文件系统存放的文件均在git仓库中以`$`为前缀，此时git仓库中的其他文件名可用于SFS文件系统记录元数据，例如`mkdir`与`rmdir`中提到的特殊文件。

测试

测试环境

- `libfuse`：版本2.9.4，使用`apt install libfuse-dev`安装；
- `libgit2`：版本0.24.1，使用`apt install libgit2-dev`安装；
- `fio`：版本2.2.10，使用`apt install fio`安装；
- 其他必要的GCC工具链。

正确性测试

在分别实现并测试SFS的各个接口之后，我们使用在SFS上编译运行一个真实项目的方式，来测试SFS综合运行各个功能的正确性，以及处理并发请求的正确性。我们选用ucore这个微型操作系统作为测试样本，使用`make -j8`并行编译。编译可以顺利完成，且编译后ucore可以正常运行。

为了进一步测试SFS并发读写的正确性，我们使用fio测试工具，并发地读写4个文件，每个文件使用8个线程进行随机并发读写，利用CRC32校验和检查正确性。SFS可以通过测试。

性能测试

我们使用fio进一步进行了性能测试。测试环境如下：

- 底层存储设备：机械硬盘（HDD），Hitachi HTS547564A9E384(JEDOA50A)；
- 运行基准测试和SFS的底层文件系统：ext4；
- 为确保公平性，测试时开启了操作系统内核文件缓存（buffered mode），这是因为SFS内部使用了临时文件，而fio无法关闭这些临时文件的缓存；
- 写测试中，所有数据写入完毕后，调用`fsync`强制写回底层存储设备。

实验结果

顺序读取64MB - 带宽

sequential read 64MB					
bandwidth (KB/s)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	28285	27115	28909	25640	29116
sfs	66942	67563	68125	68624	68409

顺序读取64MB - 延迟

average latency (us)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	138.72	290.87	547.91	1238.43	2178.23
sfs	4.03	7.1	13.92	26.24	58.11

随机读取4MB - 带宽

random read 4MB					
bandwidth (KB/s)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	2723	4423	7772	9041	12450
sfs	4352	4486	4486	4530	4571

随机读取4MB - 延迟

average latency (us)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	1458.34	1797.25	2039.75	3511.65	5108.09
sfs	40.96	47.1	81.9	98.73	167.48

顺序写入256MB - 带宽

注：由于`commit-on-write`性能相当差，仅对其写入1MB。

sequential write 256MB	(user large size to evaluate timed commit)				
bandwidth (KB/s)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	25804	26102	26309	26549	26346
sfs (commit on close)	74494	73574	74052	72676	60611
sfs (commit every 8s)	75633	73657	74557	29108	10148
sfs (commit every 4s)	28395	26074	74367	73082	10023
sequential write 1MB	(commit on write is too slow)				
bandwidth (B/s)					
sfs (commit on write)	279545	98153	221125	97979	57396

顺序写入256MB - 延迟

注：由于`commit-on-write`性能相当差，仅对其写入1MB。

average latency (us)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	5.39	9.23	16.59	31.5	61.62
sfs (commit on close)	39.07	72.43	142.22	282.62	572.67
sfs (commit every 8s)	38.11	71.77	140.8	941.4	5824.53
sfs (commit every 4s)	126.14	270.31	140.99	279.81	5900.89
average latency (ms)					
sfs (commit on write)	14.61	83.3	73.94	334.11	1140.51

随机写入256MB - 带宽

注：由于`commit-on-write`性能相当差，仅对其写入1MB。

random write 256MB					
bandwidth (KB/s)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	30649	25832	24705	25875	25981
sfs (commit on close)	72939	56827	59946	60527	61812
sfs (commit every 8s)	73082	23700	55823	24654	24114
sfs (commit every 4s)	23096	23883	11946	10356	61263
random write 1MB					
bandwidth (B/s)					
sfs (commit on write)	226817	141489	136497	75769	63584

随机写入256MB - 延迟

注：由于`commit-on-write`性能相当差，仅对其写入1MB。

average latency (us)					
block size	4KB	8KB	16KB	32KB	64KB
baseline	5.67	9.46	16.93	32.36	61.32
sfs (commit on close)	39.37	74.76	143.55	282.74	560.64
sfs (commit every 8s)	39.39	271.43	163.21	1058.37	2181.3
sfs (commit every 4s)	157.92	268.88	1216.57	2472.15	564.42
average latency (ms)					
sfs (commit on write)	18.02	57.8	119.84	432.05	1029.46

从上面的实验结果可以看出，对于SFS的文件操作的性能不比基准文件系统差，本文暂时无法解释此类情况。

对于写操作，分别开启`commit-on-close`、`timed-commit`以及`commit-on-write`后，带宽依次减小、延迟依次增大。特别地，对于`timed-commit`，更小的提交间隔能导致带宽减小、延迟增大。这与上文的理论是相符的。

结论

本文提出了一种增量型安全文件系统SFS，介绍了其设计与实现，给出了系统参数选取的建议，然后对其在不同参数和文件访问模式的情况下进行了性能测试，较好地验证了理论分析。

虽然本项目实现的文件系统能够较好地确保安全性，但本文的方法也有一些不足：

- 本项目实现的文件系统对于底层存储设备空间的占用只增不减，在某些情况下这可能成为经济压力；
- 由于在git中获取某文件的修改时间需要扫描分支树，开销过大，故本项目没有支持文件修改时间的各类操作；
- 由于git只能记录文件是否可执行，因此本项目只能支持文件的0644和0755两种访问权限（差异仅在是否可执行）；对于目录，访问权限固定为0755。这个问题可以通过设置挂载点上级目录的权限来缓解。

未来，可以对这些不足进行改进。

参考文献

- Git. 2018. *git*. https://git-scm.com/
- libgit2. 2018. *libgit2*. https://libgit2.github.com/
- presslabs. 2018. *gitfs*. https://github.com/presslabs/gitfs
- Microsoft. 2018. *GVFS*. https://github.com/Microsoft/GVFS
- libfuse. 2018. *libfuse*. https://github.com/libfuse/libfuse
- Jens Axboe. 2018. *fio*. https://github.com/axboe/fio