Enhance Strata with Lease

Zeyuan Hu
University of Texas at Austin

Jianwei Chen
University of Texas at Austin

Junkai Liu
University of Texas at Austin

ABSTRACT

Strata [2] is a cross-media file system on a single node. Strata implements multithreading file access control. However, it lacks of concurrent file access control across processes. In this project, we fill this gap by implementing lease [1]. Our design is optimized toward write-intensive workload. Our implementation provides necessary infrastructure to make Strata become distributed file system.

1 INTRODUCTION

Strata is a cross-media file system that leverages strength of one storage media to compensate weaknesses of the other. Its main design principle is to log operations to NVM at user-level (LibFS), and digest and migrate data in kernel (KernelFS). In dosing so, Strata provides high performance, low-cost capacity and crash consistency.

Strata implements multithreading file access control. However, for concurrent file access across processes, there is no mechanism to properly coordinate file access within Strata. We need to design an approach to enforce the file access control. The file lock is a common and effective way used to share data consistently. However, file lock is not ideal for our project since one process may crash while holding the lock of a file, which prevents other processes from accessing the file.

Therefore, we enhance Strata with lease, a mechanism commonly used in distributed systems. A lease is a contract that grants the holders access to some resources with a time limit. We implement lease within Strata that has following features: (1) support leases on files and directories and (2) introduce exclusive writer and shared readers. Experiments demonstrate that our design and implementation are reliable and effective.

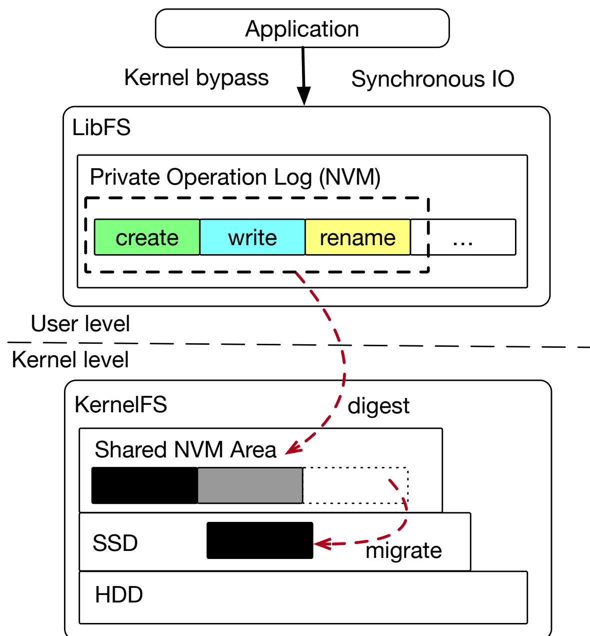

Figure 1: *Strata architecture.*

2 DESIGN

2.1 Overview

Lease[1] is a concept of cache mechanism in distributed systems such as distributed file system like NFS [3], which shares some resemblance with Strata in that per-application file operation logs act as cache and persistent storage (NVM, SSD, HDD) act as server. Lease is similar to a file lock but with a timeout: it allows concurrent read and exclusive write access. Kernel revokes file access of a process when lease is expired. We customized our lease design towards Strata unique features. For example, Strata uses transaction encapsulated application logging. Unlike RAM cache, Strata's logs are written to NVM, which is expensive. To ensure correct lease se-

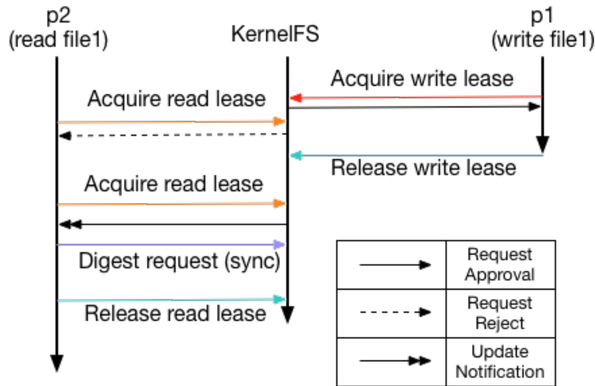

Figure 2: *Digest on read*

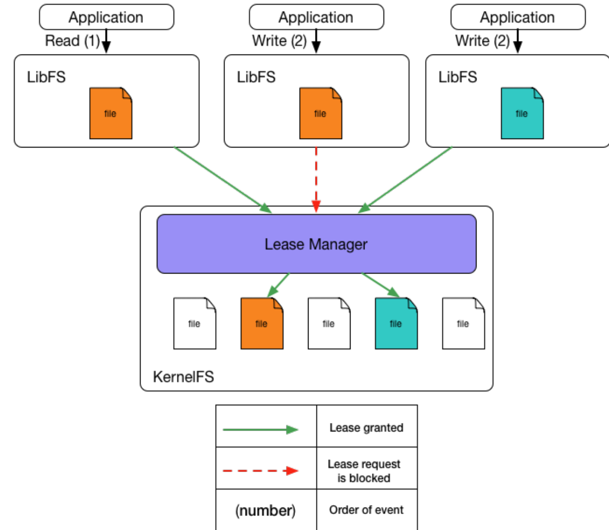

Figure 3: *Strata with Lease architecture.* For any POSIX operation, Libfs will contact lease manager to acquire lease. Lease manager maintains the lease information for a given file path.

mantics, we may need to abort transaction if lease is expired during the transaction and ask application to redo the transaction if necessary. The high cost of such operation forbids us from setting a short lease timeout.

2.2 Optimization

With a relatively long lease timeout, we can avoid aborting transaction but racing processes would need to wait for lease timeout. To mitigate such performance issue, we added following features to our lease design.

Voluntary Lease Releasing Similar to lock concept, we allow applications to call a primitive `mlfs_release_lease` to release lease to a file once they finish their operations. Other processes would then be able acquire lease and avoid long timeout. This is based on the observation that most file operations are fairly short and long lease period could act as a fail safe for long consecutive writes.

Lease Polling Lease releasing do solve part of the utilization ratio problem but is still insufficient for solving racing condition. This is because the other application have no knowledge about how soon its counterpart will finish using the lease when its lease got rejected. To resolve this, we designed a lease polling scheme which is after a relative short interval, an application can send `acquire_lease` again to the kernel to see if it can get the lease. We think this scheme is better because it relieved kernel from keeping track of how many process are waiting on a lease if using a upcall / notification scheme.

Digest on Read Because each application keeps its own version of cache (file operation logs), its critical for applications to make sure the file operations are correctly serialized per file and the processes are reading consistent states. One potential solution is to use kernel upcall to notify application to digest after writing, but we used a different approach to solve this problem. Considering a pure write scenario when multiple processes are trying to write to the same file, there's no need to do digestion because their modification might be overwritten. Therefore we designed a write friendly scheme which only do digestion on read. This requires us to keep an file state alongside the expiration time in the kernel data structure and send the states back to processes on lease acquisition.

3 IMPLEMENTATION

3.1 LibFS

From POSIX operations' perspective, lease is no more than a lock with interface shown below:

```
int
Acquire_lease(const char *path,
           mlfs_time_t *expiration_time,
           file_operation_t operation,
           inode_t type);

void
mlfs_release_lease(const char* path,
           file_operation_t operation,
           inode_t type);
```

To acquire a lease, operation needs to provide file path,

an mlfs_time_t struct to hold expiration time, operation type to indicate current operation (e.g., read, create, write, delete), and whether the lease is for file or directory. All the lease-related logic is encapsulated inside Acquire_lease implementation, which is irrelevant to the caller.

Internally, Acquire_lease will contact KernalFS to initialize expiration_time at the very beginning. In the subsequent lease call, if necessary, Acquire_lease will contact KernalFS to renew the lease. If KernalFS indicates error (e.g., some other process has modified the file), a synchronous digestion request will trigger before returning the error code to operation. Motivation for digestion request is that the current process may want to see the latest file state change. If KernalFS indicates that the lease request is blocked, Acquire_lease execution will be block for a poll time before contacting KernalFS again to acquire (renew) lease. Acquire_lease exits only when acquire lease request is fulfilled. Besides OK and error, Acquire_lease may also return an exit code indicating there is renewal failure inside the function call (i.e., renewal request is blocked at least once). This exit code is critical for certain POSIX operation. For example, if the lease has been given up inside Acquire_lease before, read operation need to invalidate its cache and trigger synchronous digestion request to make sure it can read the latest file change.

Whenever POSIX operations finish with lease, they need to release lease through mlfs_release_lease call. This call is important to lease performance as our lease time is relative long to accommodate the transaction write (i.e., long enough so that write transaction can finish). Thus, we do not want to wait for lease expire whenever possible. Besides lease interface taking file path as a input parameter, we also provide interface that can take in inode number as input parameter. To support this interface, we also maintain a hash table that maps file path with its inode number. The entry to the hash table is added whenever a POSIX open is called.

3.2 KernelFS

The kernel keeps track of lease status of all files by maintaining a hash table from file path to lease states. The lease states contains three parts: current lease expiration time, current state of the file, last process to modify the state of file. The lease expiration time is expressed in mlfs_time_t with zero as indicator of nobody is holding the lease. On lease_acquire, lease manager will check the type of file operation type, for mlfs_read_op it will try to acquire a read lease for that file and other operations (mlfs_write_op, mlfs_create_op , mlfs_delete_op) it will acquire a write lease for that operation. Additionally, for mlfs_create_op, mlfs_delete_op it will first check if the hash table has unreleased lease for its subdirectory if the path is a folder.

/mlfs/file1	{ expiration time; state: UNK }
/mlfs/file2	{ expiration time; state: DEL }
/mlfs/file3	{ expiration time; state: CRT }
...	...

Figure 4: *Lease Manager Data Structure*

The lease acquisition process is basically checking the lease type and compare lease expiration time with current time. Read lease can be shared but write lease is exclusive. If the lease is granted, the lease expiration time is updated to current time plus the lease interval and return to the caller. Otherwise, the lease will be rejected with current lease expiration time returned. Currently, this is indicated by negating the tv_sec part of the return value.

The lease releasing process is just resetting the expiration time. For mlfs_create_op, mlfs_delete_op, the file state will also be reset to indicate the modification. Additionally, on digestion complete, all the file states will be reset to unknown because now the change is visible to every process.

3.3 Connectting LibFS with KernelFS

In this part, we implement a client and a server in the LibFS and KernelFS respectively. Here we adopt the Unix domain socket to enable the communication between different processes executing on the same host operating system. Since it is standard component of POSIX operating systems, we can incorporate this feature in our project conveniently. The socket type we use is the SOCK_DGRAM, which supports connectionless messages of a fixed maximum length. In this way, the server can receive datagrams from multiple clients using recvfrom (which must specify the address to receive requests from) and replies to them with sendto. Unlike the Internet socket, all communication of Unix domain socket occurs entirely within the operating system kernel so we do not need to worry about the packet loss when using SOCK_DGRAM.

In the client, we mainly design and implement an API called send_requests. This API takes the file operation types, lease action types, node types and the file path as input. The first three parameters are encoded into a fix-length header which will then be sent to the server together with the file path.

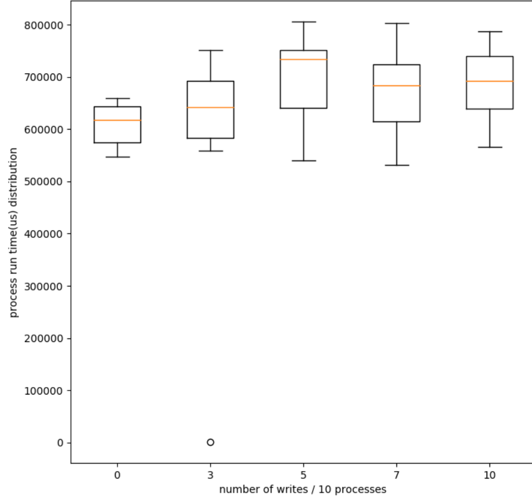

Figure 5: *Process runtime for read-intensive workload vs. write-intensive workload.*

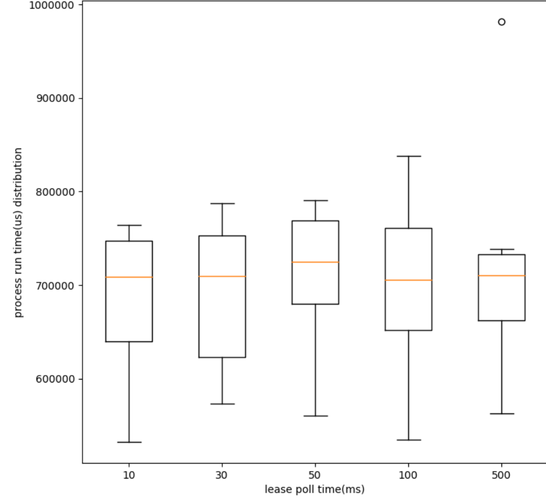

Figure 6: *Process runtime change as poll time increases.*

In the server, the most important function is `process_new_data`. This function will receive a request from the client and decode the header of the message. After that, it will determine which function should be called (`lease_acquire` or `lease_release`) and send the responses to the client. Additionally, we implement the server with synchronous connections handling using epoll() system call and we make the socket to be non-blocking.

4 EVALUATION

We perform all our experiments on a laptop with Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, Kernel version 4.15.0, Ubuntu 16.04 LTS, and 8GB memory. We emulate two persistent memory with size of 474MB each.

We first measure process runtime under read-intensive workload and write-intensive workload. We start 10 LibFS processes and all of them work with the same file. Each process will either read or write 100MB of the file. We measure how long it takes for all the processes finish their work. The result is shown in Figure 5. As shown in the figure, the best performance is achieved When all 10 processes perform read operations. Since lease allows multiple readers and there is no synchronous digestion request involved when no one modifies the file, the observation is expected. When there are 10 write processes, the runtime is around 680 ms, which is lower than five read and five write case. The overhead for 10 write case is the wait time to acquire write lease. However, since there is no wait time for digestion, 10 processes can finish in reasonable time. Worst case happens when there is a mixture of read and write operations. Since each read may need to wait for digestion finish at the very beginning, there is extra overhead besides the wait time to acquire write lease.

Since our lease implementation is polling-based, we also measure the impact of poll time on process performance. As shown in Figure 6, the worst runtime of processes of each scenario increases as the poll time increases becase the more frequent we poll the KernalFS, the more likely we can acquire lease and finish operations.

5 CONCLUSION

We implemented lease mechanism in Strata to support concurrency file access control across multiple processes. Our lease supports multiple readers and exclusive writer. We optimize lease implementation towards write-intensive workload by using voluntary lease releasing, lease polling, and digestion on read.

6 ACKNOWLEDGEMENTS

We thank Simon Peter for his helpful discussion on Strata and his sponsorship for machine.

7 AVAILABILITY

We have made the source code for `Enhance Strata with Lease` available at the following URL: `https://github.com/xxks-kkk/strata`.

References

[1] GRAY, C., AND CHERITON, D. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, vol. 23. ACM, 1989.

[2] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 460–477.

[3] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference* (1985), pp. 119–130.