The KudOS Architecture for File Systems

Andrew de los Reyes, Chris Frost, Eddie Kohler, Mike Mammarella, and Lei Zhang
University of California, Los Angeles
{adlr,frost,kohler,mikem,leiz}@cs.ucla.edu

INTRODUCTION

For robustness, stability, and reboot speed, file system implementations must ensure that the file system's stored image is kept consistent or easy to return to consistency. Advanced consistency mechanisms such as soft updates [2] and journalling make this possible; unfortunately, they are generally tied to a particular file system, and can't be ported or adapted without significant engineering effort. Furthermore, interfaces like `fsync()` give user code only coarse control over consistency. Applications with custom consistency and performance requirements get little help from conventional file systems, which either impose high overhead (data journalling) or don't guarantee data consistency (soft updates, for example, ensures metadata consistency only).

We propose a new file system implementation architecture, called *KudOS*, where *change descriptor* structures represent any and all changes to stable storage. File systems generate change descriptors for all writes, then send them to block devices for eventual commit. Each change descriptor stores the old state of the block and the change's *dependencies*—other change descriptors that must be committed before it is safe to commit this change. Explicit dependencies let KudOS modules preserve necessary file system invariants without understanding the file system itself; the old state lets KudOS roll back changes when necessary to break cyclic block dependencies. Change descriptors can implement many consistency mechanisms, including soft updates and journalling.

KudOS is decomposed into fine-grained modules which generate, consume, forward, and manipulate change descriptors. A particular innovation of the module design is the separation of the low-level specification of on-disk layout from higher-level file system-independent code, which operates on abstract disk structures.

We have implemented a prototype of the KudOS architecture as part of a new operating system. Although results are premature and performance has not been measured, change descriptors have helped us construct consistent file system structures. Our journalling module should automatically add journalling to any file system, and combinations of simple modules can support, for example, correct consistency on RAID over loop-back devices. Eventually, we plan to support user-defined dependencies, allowing applications to define consistency protocols for the file system to follow.

CHANGE DESCRIPTORS

Each in-memory modification to a cached disk block in KudOS has an associated change descriptor. Different change types correspond to different forms of change descriptors; the change descriptor for a flipped bit—such as in a free-block bitmap—contains an offset and mask, while larger changes contain an offset, a length, and the new data. The change descriptor's dependencies point to other change descriptors that must precede it to stable storage. A change descriptor can be applied or reverted to switch the cached block's state between old and new as necessary. Each change descriptor applies to exactly one block. Figure 1 gives a simplified version of the structure. The ability to revert and re-apply change descriptors is inspired by the soft updates system in BSD's FFS [2], but generalized so that it is not specific to any particular file system.

```
struct chdesc {
    BD_t *device;
    bdesc_t *block;
    enum {BIT, BYTE, NOOP} type;
    union {
        struct {
            uint16_t offset;
            uint32_t xor;
        } bit;
        struct {
            uint16_t offset, length;
            uint8_t *data;
        } byte;
    };
    struct chdesc *dependencies[];
/* ... */ };
```

Figure 1: Partial change descriptor structure.

When a KudOS module first generates change descriptors to write to the disk, it specifies write ordering requirements similar to those of soft updates. For example, Figure 2 depicts change descriptors that allocate and add a new block to an inode. The module passes these change descriptors to another module closer to the disk. This second module can inspect, delay, and even modify them before passing them on further. For instance, the write-back cache module holds on to blocks and their change descriptors instead of forwarding them immediately. When evicting a block and associated change descriptors, the write-back cache enforces an order consistent with the change descriptor dependency information.

Soft updates, journalling, and many application-specific consistency models all correspond to different change descriptor arrangements, so these features can be added to the system as modules which appropriately connect or reconnect the change descriptors. For example, the change descriptors in Figure 2 can be transformed to provide journalling semantics. The original four change descrip-

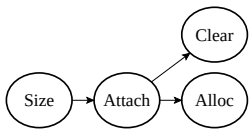

Figure 2: Soft updates change descriptor graph, including the dependencies for adding a newly allocated block to an inode. Writing the new block pointer to an inode (Attach) depends on initializing the block (Clear) and updating the free block map (Alloc). Updating the size of the inode (Size) depends on writing the block pointer.

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

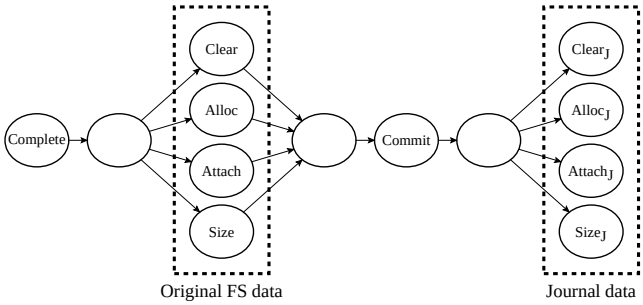

Figure 3: Journal change descriptor graph for the change in Figure 2. Empty circles are "NOOP" change descriptors with no associated block data.

tors are modified to depend on a journal commit record, which depends on blocks journalling the changes. Once the actual changes commit, the journal record is marked as completed. Figure 3 shows these transformed change descriptors. This single journalling module can attach to any file system module; it performs transformations incrementally as change descriptors arrive.

Further, by changing our journal module to journal only change descriptors that modify file system metadata—and by adding additional dependencies to prevent premature reuse of blocks—we could even obtain metadata-only journalling. The journal module can distinguish metadata change descriptors because of the LFS interface described below. Other block device layering systems, like GEOM [1] or JBD in Linux, would or do need special hooks into file system code to determine what disk changes represent metadata in order to do metadata-only journalling. Change descriptors and the LFS interface allow us to do this automatically.

File System Module Interfaces

A complete KudOS configuration is composed of many modules. By breaking file system code into small, stackable modules, we are able to significantly increase code reuse. We add an additional interface that helps to divide file system implementations into common (reusable) code and file system-specific code. We call this intermediate interface the "Low-level File System" (LFS). This new interface is a departure from other stackable module systems, like FiST [3], which stack higher-level operations.

The LFS interface has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-ops. A module implementing the LFS interface should define how bits are laid out on the disk, but doesn't have to know how to combine the micro-ops into larger, more familiar file system operations. A generic VFS-to-LFS module decomposes the larger file write, read, append, and other standard operations into LFS micro-ops. This one module can be used with many different LFS modules implementing different file systems.

Figure 4 shows a contrived example taking advantage of the LFS interface and change descriptors. A file system image is mounted with an external journal, both of which are loop devices on the root file system, which uses soft updates. The journalled file system's ordering requirements are sent through the loop device as change descriptors, allowing dependency information to be maintained across boundaries that might otherwise lose that information. In contrast, without change descriptors and the ability to forward change descriptors through loop devices, BSD cannot express soft updates' consistency requirements through loop-back file systems.

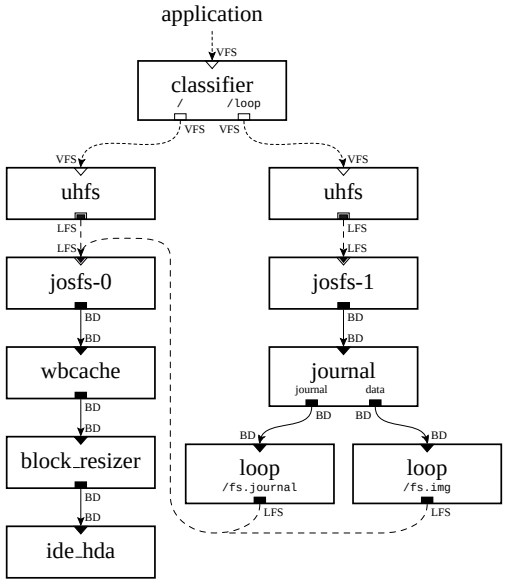

Figure 4: A running KFS configuration. */* is a soft updated file system on an IDE drive; */loop* is an externally journalled file system on loop devices.

Future Work

We would ultimately like to extend the change descriptor APIs into user space, allowing applications like database servers to specify minimal and precise data dependencies for order-sensitive data. Application-defined change descriptors could improve the performance and correctness of applications requiring complicated consistency semantics, although there are many issues to consider; for instance, we must prevent misbehaving or malicious applications from creating excessive numbers of change descriptors, or setting up dependencies such that certain change descriptors stay in the system indefinitely.

We also plan to study change descriptors' performance and memory overhead implications. While we believe end-to-end disk performance will remain similar to native implementations of soft updates and journalling, we would like to better evaluate our implementation to minimize the impact of change descriptors on system performance. In addition, we plan to implement an ext3-like file system and a transactional file system to demonstrate the utility of change descriptors and modules in KudOS.

References

[1] *GEOM – modular disk I/O request transformation framework.* `http://www.freebsd.org/cgi/man.cgi?query=geom&sektion=4`.

[2] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.

[3] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. pages 55–70. USENIX, June 2000.