SYSTEMS GOSSIP MEETUP

Learning large systems using peer-to-peer gossip

Policy Against Harassment at ACM Activities

OS Meetup wants to encourage and preserve this open exchange of ideas, which requires an environment that enables all to participate without fear of personal harassment. We define harassment to include specific unacceptable factors and behaviors listed in the ACM's policy against harassment. Unacceptable behavior will not be tolerated.
https://www.acm.org/about-acm/policy-against-harassment

File System

- Recall
  - Scheduling
- File System Introduction
  - Impression
  - File System Layers
- Disk Layout
- iNode
  - Content
- Directory
  - Structure of Directory
  - Path & Search
  - How a file got created with iNode?
- Cache -> Buffer

Recall

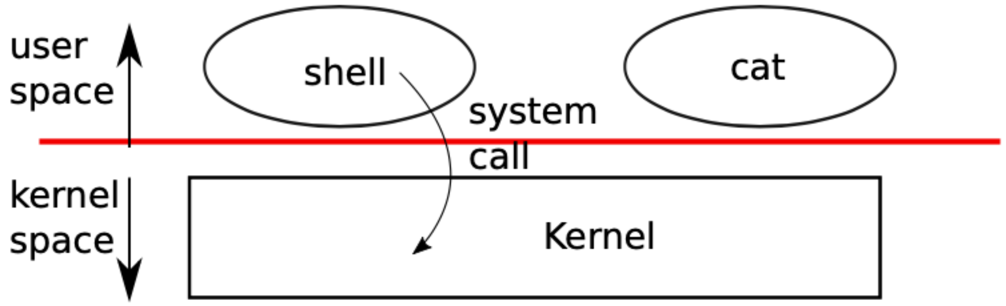


Figure 1.1: A kernel and two user processes.

File System Impression

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

Frangipani: A Scalable Distributed File System

Chandramohan A. Thekkath
Timothy Mann
Edward K. Lee

Systems Research Center
Digital Equipment Corporation
130 Lytton Ave, Palo Alto, CA 94301

Abstract

The ideal distributed file system would provide all its users with coherent, shared access to the same set of files, yet would be arbitrarily scalable to provide more storage space and higher performance to a growing user community. It would be highly available in spite of component failures. It would require minimal human administration, and administration would not become more complex as more components were added.

Frangipani is a new file system that approximates this ideal, yet was relatively easy to build because of its two-layer structure. The lower layer is Petal (described in an earlier paper), a distributed storage service that provides incrementally scalable, highly available, automatically managed virtual disks. In the upper layer, multiple machines run the same Frangipani file system code on top of a shared Petal virtual disk, using a distributed lock service to ensure coherence.

Frangipani is meant to run in a cluster of machines that are under a common administration and can communicate securely. Thus the machines trust one another and the shared virtual disk approach is practical. Of course, a Frangipani file system can be exported to untrusted machines using ordinary network file access protocols.

We have implemented Frangipani on a collection of Alphas running DIGITAL Unix 4.0. Initial measurements indicate that Frangipani has excellent single-server performance and scales well as servers are added.

1 Introduction

File system administration for a large, growing computer installation built with today's technology is a laborious task. To hold more files and serve more users, one must add more disks, attached to more machines. Each of these components requires human administration. Groups of files are often manually assigned to particular disks, then manually moved or replicated when components fill up, fail, or become performance hot spots. Joining multiple disk drives into one unit using RAID technology is only a partial solution; administration problems still arise once the system grows large enough to require multiple RAIDs and multiple server machines.

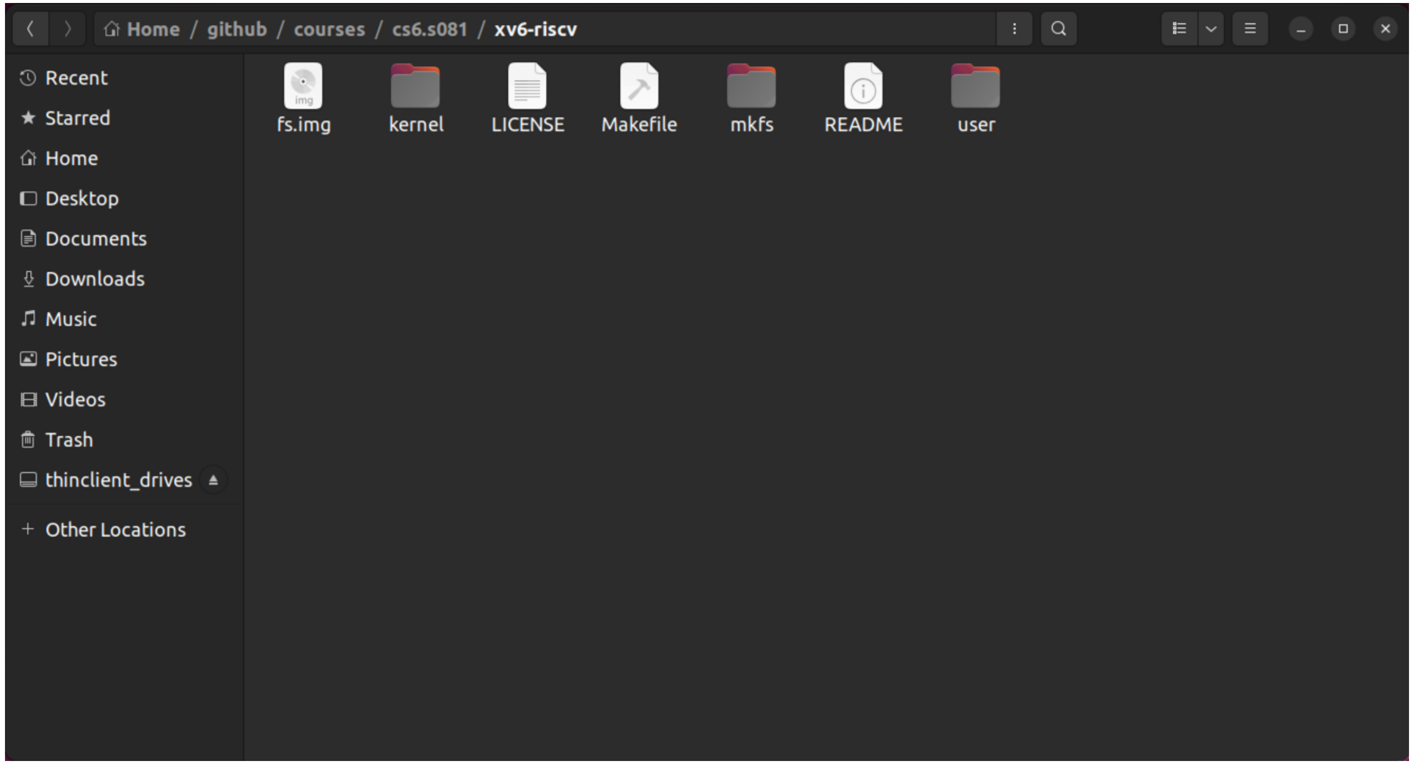
Frangipani is a new scalable distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage. The machines are assumed to be under a common administration and to be able to communicate securely. There have been many earlier attempts at building distributed file systems that scale well in throughput and capacity [1, 11, 19, 20, 21, 22, 26, 31, 33, 34]. One distinguishing feature of Frangipani is that it has a very simple internal structure—a set of cooperating machines use a common store and synchronize access to that store with locks. This simple structure enables us to handle system recovery, reconfiguration, and load balancing with very little machinery. Another key aspect of Frangipani is that it combines a set of features that makes it easier to use and administer Frangipani than existing file systems we know of.

1. All users are given a consistent view of the same set of files.
2. More servers can easily be added to an existing Frangipani installation to increase its storage capacity and throughput, without changing the configuration of existing servers, or interrupting their operation. The servers can be viewed as "bricks" that can be stacked incrementally to build as large a file system as needed.
3. A system administrator can add new users without concern for which machines will manage their data or which disks will store it.
4. A system administrator can make a full and consistent backup of the entire file system without bringing it down. Backups can optionally be kept online, allowing users quick access to accidentally deleted files.
5. The file system tolerates and recovers from machine, network, and disk failures without operator intervention.

Frangipani is layered on top of Petal [24], an easy-to-administer distributed storage system that provides *virtual disks* to its clients. Like a physical disk, a Petal virtual disk provides storage that can be read or written in blocks. Unlike a physical disk, a virtual disk provides a sparse 2^{64} byte address space, with physical storage allocated only on demand. Petal optionally replicates data for high availability. Petal also provides efficient snapshots [7, 10] to support consistent backup. Frangipani inherits much of its scalability, fault tolerance, and easy administration from the underlying storage system, but careful design was required to extend these properties to the file system level. The next section describes the structure of Frangipani and its relationship to Petal in greater detail.

File System Impression

- Read
- Write
- Directory
- Path
- Cache
- Crash Recovery
- Sharing



File System Layers

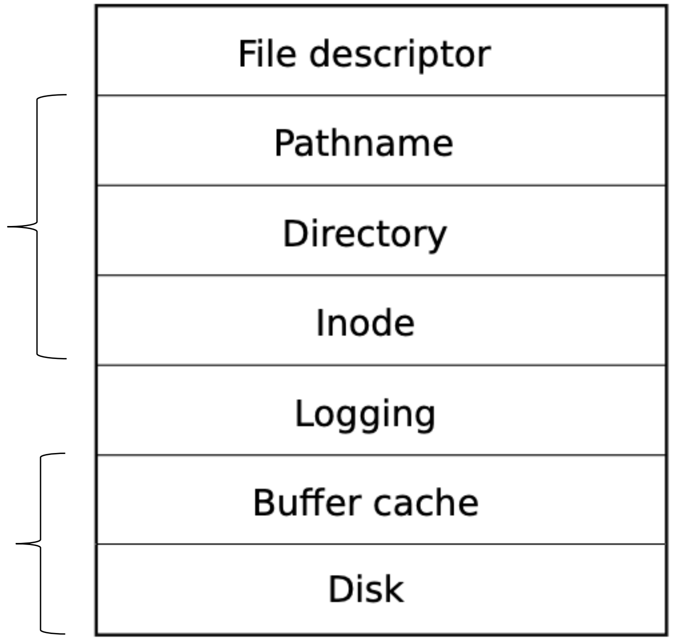
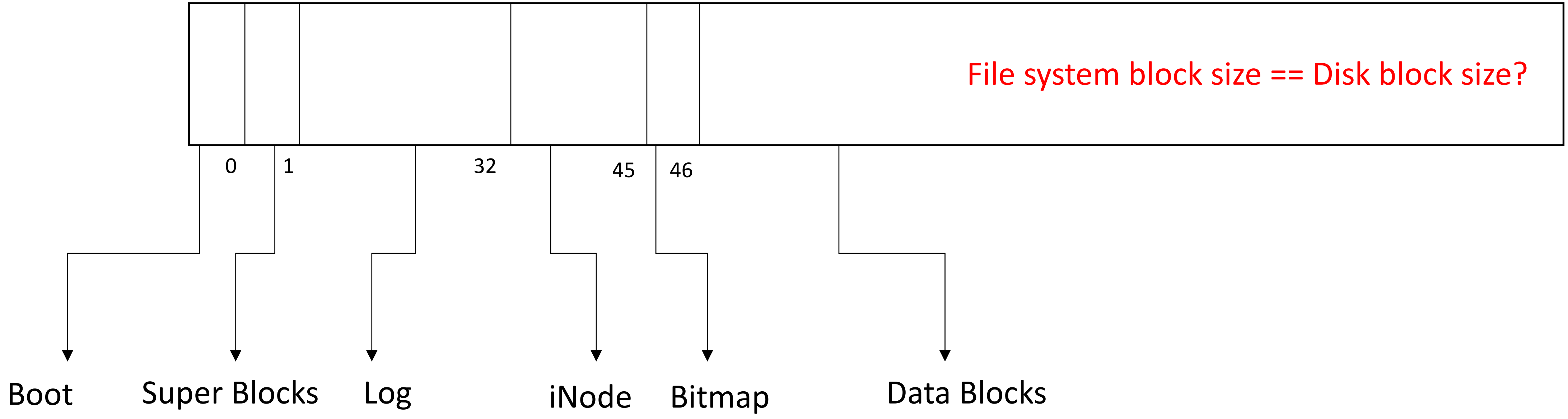


Figure 8.1: Layers of the xv6 file system.

File System: Disk Layout

File System: Disk Layout



File system block size == Disk block size?

```c
struct superblock {
  uint magic;
  uint size;
  uint nblocks;
  uint ninodes;
  uint nlog;
  uint logstart;
  uint inodestart;
  uint bmapstart;
};
```

```c
struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

What is the size of xv6 file system?

File System: iNode

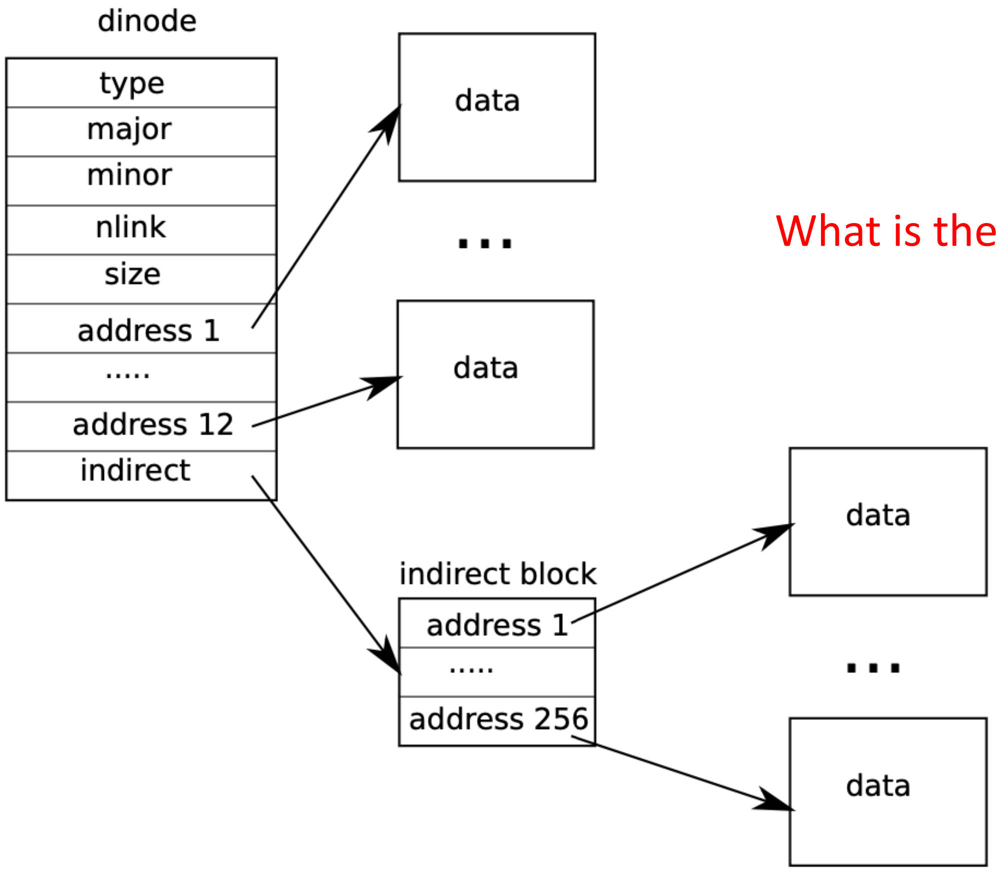
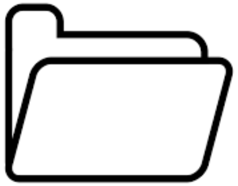


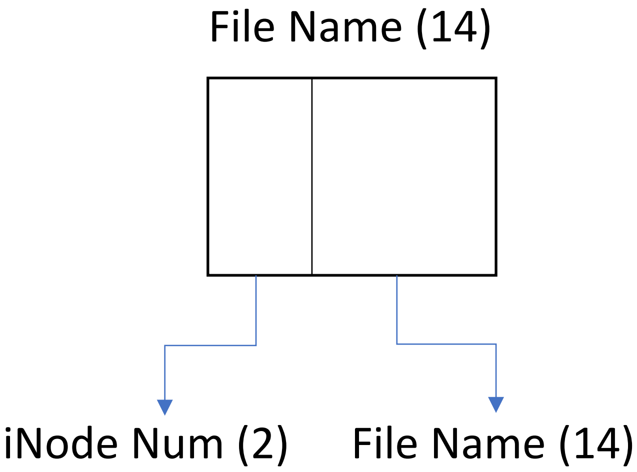
Figure 8.3: The representation of a file on disk.

What is the largest file size in xv6?

File System: Directory



```c
struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```



File System: Path Search

File System: How a file got created?

sys_open

create

ialloc

iupdate

dirlink

writei

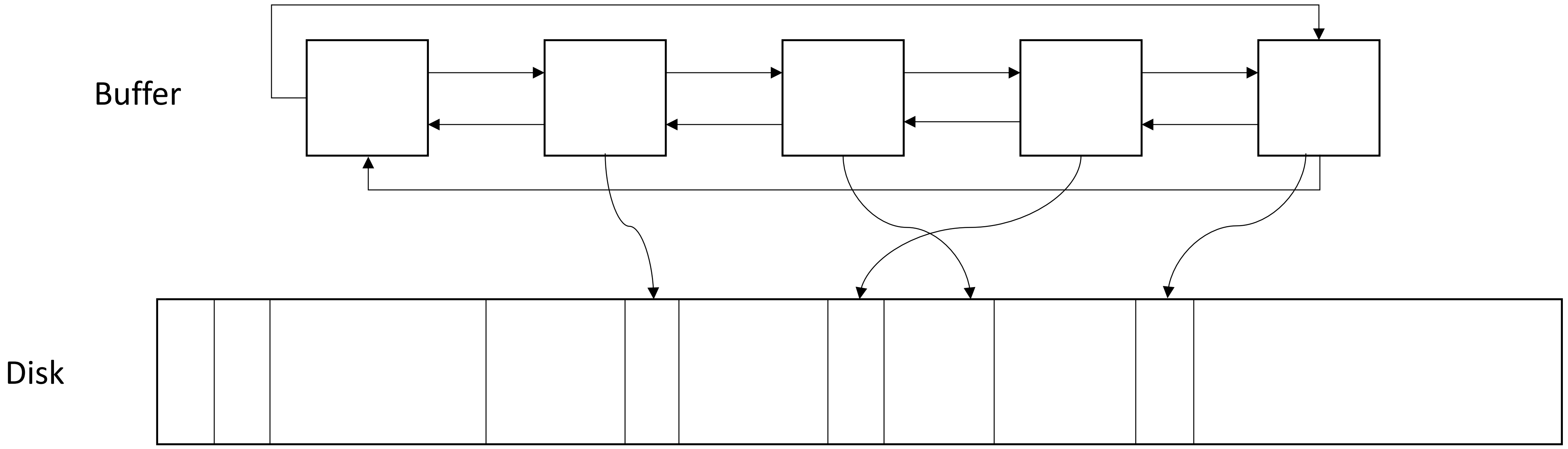
iupdate

itrunc

iupdate

File System: Cache

- Synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy;
- Cache popular blocks so that they don't need to be re-read from the slow disk;



Summary

- Disk Layout
- inode
- Directory and Path
- Cache
- Next
  - File System Logging