File System Performance

s5fs, ffs, lfs, etc.

This lecture discussed file system performance, using three example file systems to illustrate how on disk structure can lead to different performance behavior: s5fs, ffs, and lfs.

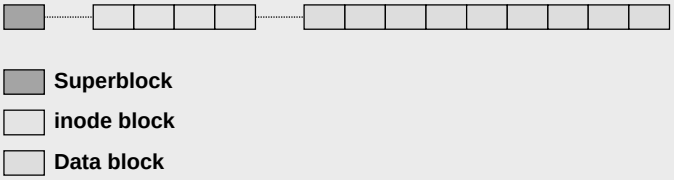
s5fs

- First UNIX file system
- Simple (~1000 lines of C)
- Reasonable performance for the time

s5fs, or the System V file system, is the first UNIX file system. As the rest of UNIX was when initially developed, it was designed for simplicity. At the time, it exhibited reasonable performance characteristics. It's very slow in comparison to modern file systems, but can be implemented in roughly a thousand lines of code, which is much much less than complex systems such as Veritas, which are in the tens of thousands.

s5fs Structure

- File system is a linear array of blocks
- Superblock
- inode blocks
- Data blocks



s5fs makes a single simple abstraction of the disk, that it's a simple linear array of blocks. The first non-boot block is the superblock, which is followed by some number of inode blocks, then the remaining blocks are data blocks.

s5fs Superblock

- Magic number
- Number of inodes
- Free block list head
- Free inode list head
- Last modification time
- etc.

The s5fs superblock contains file system wide metadata. For example, how many inodes the file system has, the index of the first free inode, the index of the first free data block, and the last time the file system was modified. This data is critical to be able to mount the file system. Without it, there's no surefire way to tell some of these things.

```
struct  filsys
{
        ushort  s_isize;              /* size in blocks of i-list */
        daddr_t s_fsize;              /* size in blocks of volume */
        short   s_nfree;              /* number of addresses in s_free */
        daddr_t s_free[NICFREE];      /* free block list */
        short   s_ninode;             /* number of i-nodes in s_inode */
        s5ino_t s_inode[NICINOD];     /* free i-node list */
        char    s_flock;              /* lock during free manip */
        char    s_ilock;              /* lock during i-list manip */
        char    s_fmod;               /* super block modified flag */
        char    s_ronly;              /* mounted read-only flag */
        time_t  s_time;               /* last super block update */
        short   s_dinfo[4];           /* device information */
        daddr_t s_tfree;              /* total free blocks*/
        s5ino_t s_tinode;             /* total free inodes */
        char    s_fname[6];           /* file system name */
        char    s_fpack[6];           /* file system pack name */
        long    s_fill[13];           /* makes sizeof filsys be 512 */
        long    s_magic;              /* magic number */
        long    s_type;               /* type of new file system */
};
```


Here's a C struct that represents the superblock of s5fs. Note the presence of locks for inode and free list manipulation. This struct could be populated by reading the superblock into memory, then casting it to a struct filesys.

s5fs inode

- File size
- Last modification
- Owner, group, permissions
- Data blocks

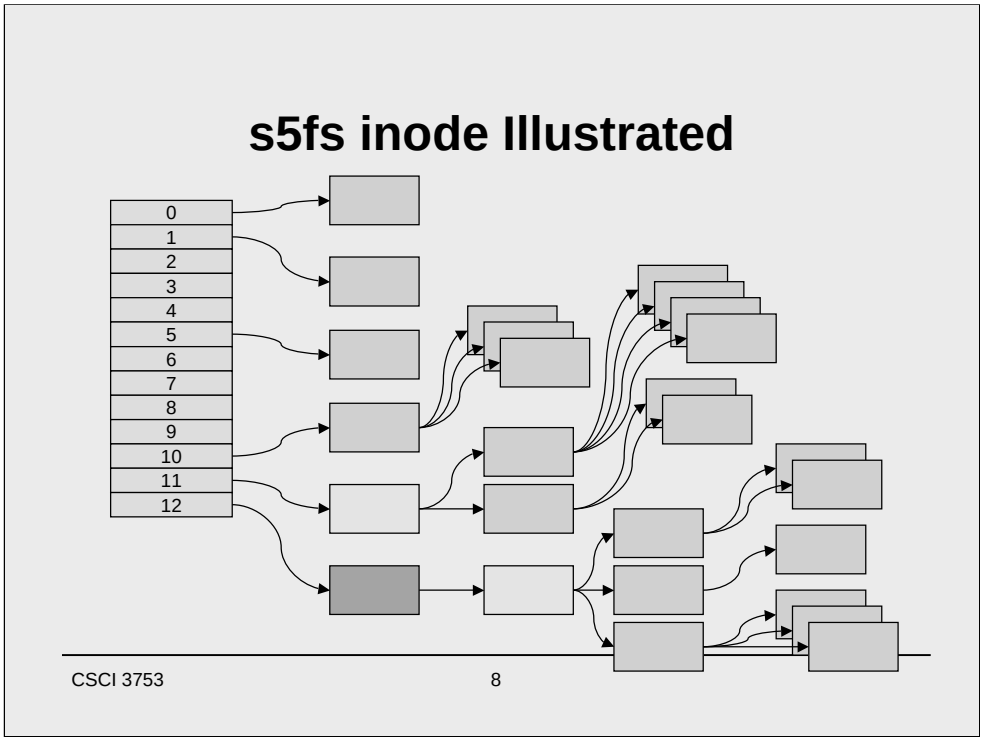
File specific data is stored in an s5fs inode, such as the size of the file and the time of last modification. It also contains the block map, the data structure that stores what blocks belong to the file.

s5fs Data Blocks

- **Direct blocks**
- **Single indirect block**
- **Double indirect block**
- **Triple indirect block**

s5fs has three types of data blocks. First are direct blocks, which store data. Next are single indirect blocks, which store block numbers for direct blocks. This way, instead of having a single large table of direct blocks in the inode, blocks on the disk can be allocated for storing the disk map of large files. Next are double indirect blocks, which store the block numbers of single indirect blocks. Last are triple indirect blocks, which store the block numbers of double indirect blocks.

The hierarchy of single, double, and triple indirect blocks allows for large files while keeping the inode disk map fairly small. It does mean that later blocks in a file might require multiple block reads, but the worst case is 5 reads (inode, triple, double, single, data).



Here's a pictorial representation of the inode block map. An s5fs inode has a total of 13 block entries in its block map. The first ten are direct blocks. The eleventh is a single indirect block. The twelfth is a double indirect block, and the thirteenth is a triple indirect block.

Blocks numbers are ordered in indirect blocks. The eleventh block of a file is the first block number in the single indirect block, while the twelfth is the second.

s5fs supports sparse files. The file size provides the kernel with knowledge of the last valid block in a file. However, if large (block sized areas) of the file are all zeros, then having a large number of zero filled blocks is a waste. Instead, the file system only allocates blocks that are needed. If a process opens a new file, then seeks far into it before writing a single byte, of course a block must be allocated for that byte, but blocks need not be allocated for all of the space that was seeked over.

Indirect blocks are wasted disk space, as they're not actually storing any file data. A degenerate series of files might require up to four times their expected storage. This is a pretty degenerate case, though.

Maximum File Size

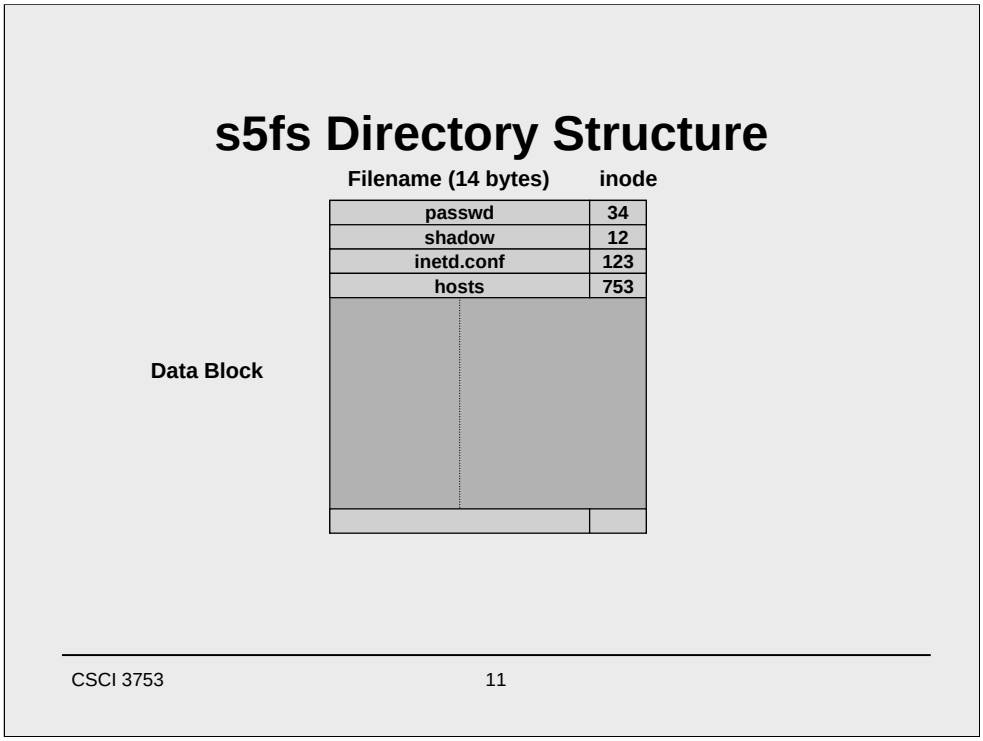
- 4 byte block index
- 1024 byte blocks
- 256 block indices per block
- $10 + 256 + 256^2 + 256^3$
- 16 GB files (actually, 2GB, but hey)

Given a few pieces of data, it's clear why there are triple indirect blocks. Assuming a 4 byte block index and a 1K block, each block can contain 256 block entries. Doing a little arithmetic, it's easy to determine that a triple indirect block allows for files that are about 16GB in length. Only having a double indirect block would make the maximum file size roughly 64MB. Of course, although it's possible to have a 16GB file, a 32 bit architecture could never reference the latter 14GB of the file. (Quick quiz: Why is a 32 bit architecture limited to 2GB instead of 4GB? Hint: what's the type of an offset pointer?)

s5fs Free List

- **Linked list of free blocks**
- **Linked list of free inodes**

Free inodes and blocks are kept track of by the free list. The head of the list is on the superblock. Free inodes and blocks can then reference later ones, forming a linked list of sorts. When a new block or inode is freed, it can be put on the head of the list (you **really** don't want to walk to the end of the list); when one is needed, it can be removed.



Directories in s5fs have a fixed structure. They have data blocks, just like any file, but have a special bit set in the inode to denote that they're directories. The data blocks have a specific format, broken up into directory entries. A directory entry has a 14 byte file name and an inode index. 14 bytes allows for names of reasonable length without wasting a great deal of space. Nowadays, a 14 byte length limit is unheard of, but back then, it was OK, and larger would involve significant waste or a more complex structure. The size of a directory file (in the inode) allows the kernel to know how many entries are in the directory.

s5fs is great, but...

- Disjoint data and meta information
- No locality in data
- Ignores disk characteristics
- Small block size
- Superblock fragility

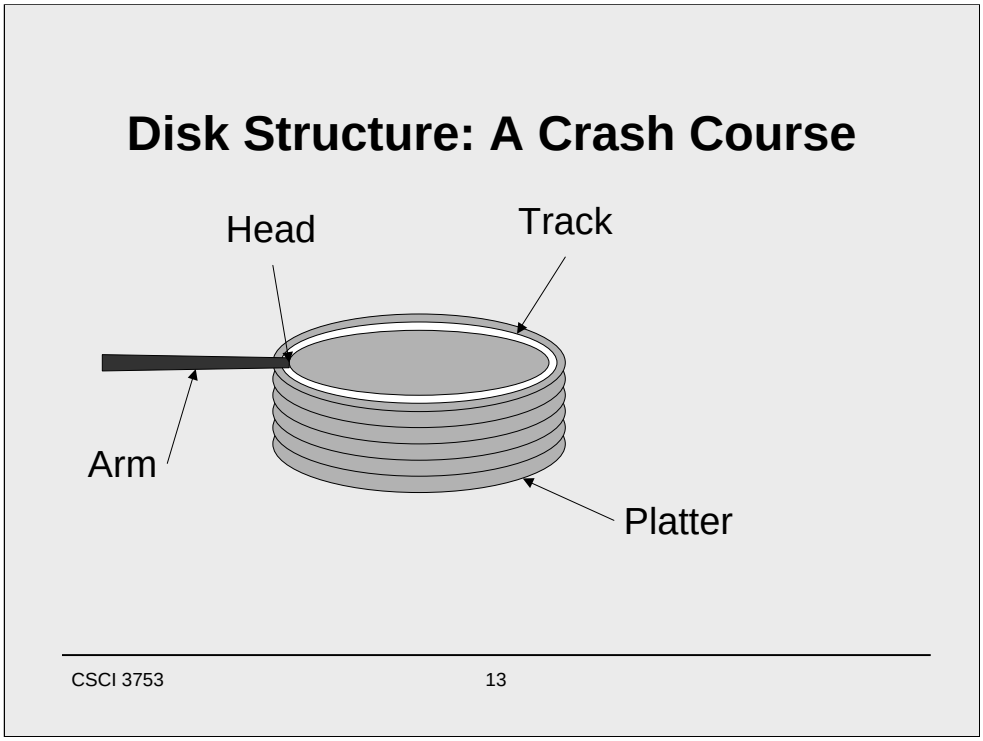
CSCI 3753 12

s5fs was great for its time and is very simple, but has a slew of significant problems that came to light as UNIX grew in popularity. As UNIX transitioned from a research to a production system with the BSD project, s5fs was no longer adequate.

The foremost problem with s5fs is its performance. Metadata (inodes) and data are disjoint and distant on the disk. All of the inodes are next to one another, while all of the data blocks are next to one another. The structure of s5fs ignores the structure and behavior of a disk.

Additionally, s5fs has a very small block size. This means that files over 10K in size start requiring indirect blocks. Since there's no promise that an indirect block is close to its data blocks, this can require a lot of time spent seeking across the disk.

One more major problem is s5fs' fragility in the face of crashes. Notably, there's a single copy of the superblock. If it's lost due to a crash, there's really no good way of getting your file system back. Some sort of redundancy would be nice, such that a single crash is unlikely to destroy your entire file system.



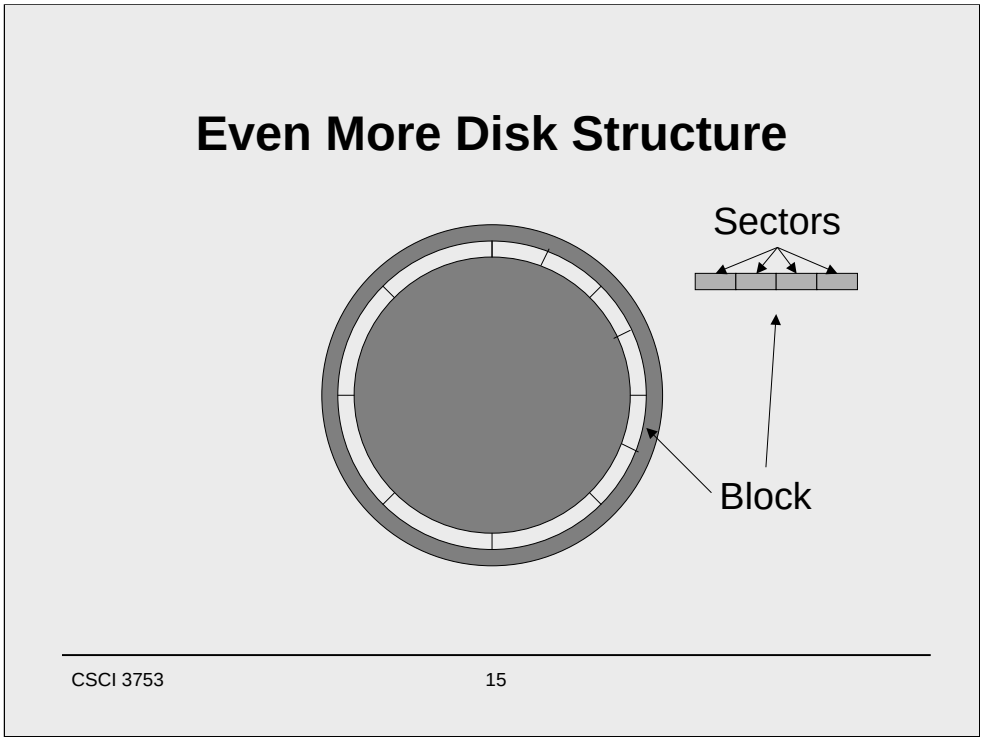
Here's a simple pictorial representation of a disk. The disk has a set of platters, each of which has its own disk head. The platters are separated into tracks, which is the area of the disk that an unmoving head covers as the disk spins.

Disks are really slow in comparison to main memory. This latency is due to the fact that physical parts have to move. Disk latency can be broken into two parts, rotational latency and seek latency. The latter is much worse than the former. Rotational latency refers to the time it takes a block to be rotated under an unmoving disk head; it's determined by how quickly the disk spins. Seek latency is how long it takes the head to move from track to track. For a 5,000 RPM disk (slow by today's standards), the rotational latency is on the average of 200 microseconds. A corresponding seek latency might be 10 milliseconds. While these might seem like short periods of time, they're not in terms of modern CPUs. A 500 MHz processor could execute 100,000 instructions during the rotational latency, or 5 million during the seek latency.

More Disk Structure

- **Cylinder: set of tracks that have same location on different platters**

Generally, all of the heads move as one unit. Therefore, an additional abstraction is useful, that of a cylinder. A cylinder is the set of tracks under all of the heads. You can think of this as the same track of each of the different platters. This distinction of a cylinder is useful because if the heads don't seek, then any block in the cylinder can be accessed within the rotational latency of the disk.



The smallest unit of reading and writing on a disk is the sector. Usually, sectors are quite small (512 or 1K bytes), so file systems often make an abstraction on top of the sector known as a block. A block is the smallest unit of reading and writing for the file system.

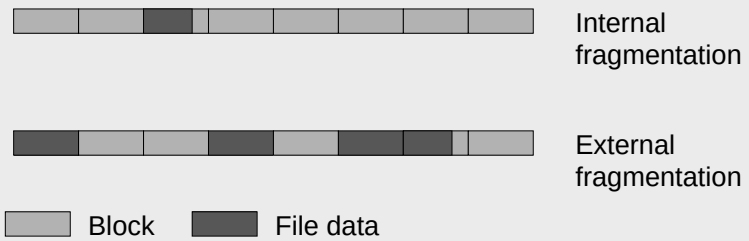
Blocks and Sectors

- A sector is a unit of disk allocation
- A block is a unit of file system allocation
- Block size is a design choice: fragmentation

Choosing the block size properly can lead to markedly increased performance for a file system. However, there's an inherent tradeoff present in deciding the block size, one a tradeoff between different forms of fragmentation.

Fragmentation

- **Internal Fragmentation (wasted space)**
- **External Fragmentation (low locality)**



Here's a picture that demonstrates why block size presents a tradeoff between forms of fragmentation. Internal fragmentation is when not the entire block is used; space is wasted. External fragmentation is when a single piece of data is spread out across distant blocks. This results in lower throughput as the disk has to seek or wait for platters to rotate before it can service requests.

Tradeoffs

- **Large block size**
  - **Low external fragmentation**
  - **High internal fragmentation**
- **Small block size**
  - **High external fragmentation**
  - **Low internal fragmentation**

A large block size reduces external fragmentation but increases internal fragmentation. A small block size does the opposite. Hopefully, there's some perfect middle value that results in little latency and not too much waste.

FFS

- File system organized in cylinder groups
- Blocks can have fragments
- Superblock redundancy
- Data locality
- Many of these are now assumed techniques

CSCI 3753 19

The Berkeley Fast File System (known as ufs or ffs) was developed in the 70s to deal with these limitations. It kept the basic abstractions of s5fs (inodes, data blocks, etc.) while using a completely different on-disk structure.

The most critical abstraction used in ffs is that of a cylinder groups, which is a small number of adjacent cylinders. ffs divides the disk up into cylinder groups, each of which has its own inodes and data blocks. Hopefully, the data blocks in the cylinder group are associated with its inodes, therefore requiring few seeks. Also, blocks of a file are usually close to one another.

FFS deals with the issue of fragmentation by dividing blocks up into fragments. Usually, entire blocks are allocated, but in special circumstances fragments can be used, in order to decrease internal fragmentation.

In addition, ffs has multiple copies of the superblock, to prevent its loss resulting from a crash.

Many of the techniques used in ffs are generally assumed when constructing modern file systems.

FFS has blocks and fragments

- **Every block is comprised of 2^n fragments**
- **Files are allocated in terms of blocks, *except* for the last block**
- **Last part of file can be one or more fragments**
- **Best of both worlds**

Every block in ffs is divided into a small number of fragments (2, 4, 8, etc.). Files are allocated in terms of blocks, except for its end, which can be allocated in terms of a fragments. This way, small files with inopportune sizes (block size + 1 bytes) won't waste entire blocks, but the file isn't broken up into pieces as small as fragments. This allows FFS to keep external fragmentation down with a large block size, but also keep internal fragmentation down at the end of a file with fragments. Of course, a sparse file still wastes disk space based on the block size, but this is less of a concern than the end of a file; users are generally angrier if their 1K file actually takes 4K than if their 200K file with only a few bytes in it actually takes 200K.

s5fs superblock

- s5fs has one superblock
- Contains file system meta-meta-data
  - block size
  - number of inodes
  - number of blocks
- Lose the superblock, good luck recovering the file system!

The s5fs superblock is a big problem due to its fragility. As it contains the file system meta-meta data that exists nowhere else, reconstructing the file system without it can be impossible. Having only one copy of the superblock (in a very conspicuous place) is asking for trouble.

Superblock redundancy

- FFS keeps redundant copies of the superblock
- Different cylinder groups, platters, rotational positions, etc.
- No localized physical failure should be able to destroy the superblock
- Superblock consistency can be resolved at crash recovery

ffs solves the superblock fragility problem by having multiple copies of it scattered across the disk. They're all on different tracks, platters, and rotational positions, so that if a single hardware failure were to destroy them all, you'd be pretty sure you'd lost all of your data as well. If the system crashes in the midst of updating the superblock, inconsistency between them can be resolved when the system reboots, as you're probably going to run fsck.

s5fs' Bad Locality

- s5fs free list becomes randomized after use (two successively allocated blocks can be distant)
- s5fs inodes and data separate

The free lists of s5fs are a simple mechanism, but lead to horrible performance. They're perfectly ordered when the file system is first created, but quickly become randomized. Two adjacent blocks in the free list can be very distant on the disk. Understandably, this is a problem. Chances are if you access block 1 of a file, you'll soon access block 2. Placing them close together on disk saves costly seeks.

Additionally, the separation of inodes and their data is a real problem. Opening a file and reading the first byte can involve two seeks across the entire disk.

The FFS Solution

- FFS data blocks are allocated on a cylinder group basis, not a global basis
- FFS has inodes and data in each cylinder group
- FFS tries to keep related files (same directory) together

ffs solves the data locality problem through its use of cylinder groups. Instead of maintaining a global free list, the cylinder group has a bit field of blocks and fragments in use. ffs tries to keep most of the blocks of most files in the same cylinder group as their inode. This way, not only are adjacent logical file blocks actually nearby physical blocks, but the inode is close as well.

In addition, ffs tries to keep all of the files in a given directory in the same cylinder group.

Data Block Allocation

- If next rotationally available block is free, use it
- Else if a block on cylinder is free, use it
- Else if a block in cylinder group is free, use it
- Else if a block in new cylinder group is free, use it
  - New cylinder group selected through quadratic hashing
- Else try searching all cylinder groups
- Exceptions for when file reaches certain size (don’t want to use all of one cylinder group)

Here's the basic data block allocation algorithm for ffs.

Related File Locality

- **ls isn't a problem…**
- **ls -l is...**

```
itsydev /tmp_mnt/home/cia/levis/public_html -10-> ls -l
total 1016
-rw-r--r--   1 levis    student      4876 Apr 18 20:02 first.txt
drwxr-xr-x   2 levis    student      4096 Aug 16  1999 img
-rw-r--r--   1 levis    student      3310 Sep  5 22:39 index.html
-rw-r--r--   1 levis    student       393 Aug 17  1999 links.html
drwxr-xr-x   2 levis    student      4096 Sep  5 12:15 os
-rw-r--r--   1 levis    disk       994571 Aug 30 22:47 paper.pdf
-rw-r--r--   1 levis    disk         9987 Nov 30  1999 plans.txt
-rw-r--r--   1 levis    student      1974 Aug 17  1999 techlinks.html
```


Recalling the s5fs directory layout, ls doesn't pose a large problem, as all of the directory entries are in the data blocks of the directory file. More complex operations, however, do cause problems. For example, the command ls -l lists a lot of metadata on each file, all of which resides in the inodes. This requires going to the inode of each of the files, reading it into memory, and using some of the data. Although all of the inodes are in one section of the disk, it can still require a good number of costly seeks or rotations to read all of them.

Related File Locality, cont.

- The metadata resides on inodes
- Want inodes in a directory to be close
- Want data to be close to inodes
- Try to make all files in a directory reside on same cylinder group

The ffs solution is to try to place all of the files in a directory in a given cylinder group. This way, only long seek is needed.

Results

- **Correct tuning can result in 900% increase in data throughput over s5fs**
- **Space loss to internal fragmentation roughly equivalent**

ffs provides impressive improvements over s5fs. Notably, its throughput is 900% higher (10 to 1), while its fragment/block division keeps the internal fragmentation roughly equivalent to s5fs.

Steps Beyond FFS

- **Better block allocation (Smith and Seltzer)**
- **Larger block allocation - extents (Veritas, Episode, etc.)**
- **Better crash recovery - journaling, lfs**

Many steps have been taken since ffs. For example, although ffs tries to keep adjacent logical file blocks close, they're often not actually adjacent, which is what's usually wanted. Work has been done to write new block allocation algorithms that keep the blocks closer. Enterprise level file systems (such as Veritas) use a different approach, that of extents. Externs are basically groups of contiguous blocks. Instead of allocating a single block at a time, the file system allocates an extent, part of which might be later freed.

In addition to disk throughput, a major concern for file systems is crash recovery. While ffs prevents the situation of crash recovery being impossible, it makes no promises for its rapidity. In modern computing environments, the time it can take to perform a crash recovery is critical. Take, for example, the recent CSEL crash recovery, which took several days. The basic issue in crash recovery is inconsistent state. For example, if the system crashes during a file write, the block may have been taken off the free list but not yet added to the file. Since we're generally unable to know to which file the block was supposed to be added, we could just return it to the free list. But first, we need to examine every inode to see if it belongs to one.

One way around this is journaling. In essence, one logs what operation is going to be performed just before it is. When recovering from a crash, all one has to do is examine the journaling log and compare it with the disk state. If the state matches the entry in the log, it completed. If it isn't, then it didn't. The rest of the system is assured to be in a consistent state.

lfs: Log Structured File System

- Only one on-disk structure: the log
- File system updates append to the log
- Metadata updates append to the log too

CSCI 3753 30

lfs, or the log structured file system, is an entirely different approach to file system structure, and has very different performance characteristics. Instead of dividing the disk up into inode areas, data areas, and cylinder groups, the entire disk forms one large log. All operations result in an a log entry being appended to the log. For example, when a file is modified, the new version is written in entirety onto the log.

At first glance, this structure is very appealing. Crash recovery is easy, as one merely has to look at the end of the log. Writes are fast, as they're always contiguous. Similarly, although a read might require a long seek to the file, the file is entirely contiguous on the disk, so subsequent reads will be quick.

lfs Benefits

- Fast writes
- Files always contiguous
- Easy crash recovery

lfs Drawbacks

- **Reads can be very slow**
  - **needs a big buffer cache**
- **Fragmentation of log**
- **The answer: *log cleaning***

CSCI 3753 32

Of course, there are always problems. Although reading an entire file requires only one seek, rarely does a system just read in a single file at a time. Instead, the files are spread all across the disk, and so reads are generally fairly slow. A big buffer cache can solve this problem, though, as then the files can just be kept in memory.

In addition, there' s the problem of log fragmentation. As files are updated and rewritten, then their old versions can be reclaimed as space for the log. Understandably, this can lead to external fragmentation, in which there's lots of free space, but none of it is very large. This prevents lfs from having good performance, as it can no longer just continue to simply append.

Log fragmentation is dealt with by a technique known as log cleaning, which basically reorders and restructures the log when the system is idle. This way, in use portions can be compacted, and large contiguous free areas can be made available.