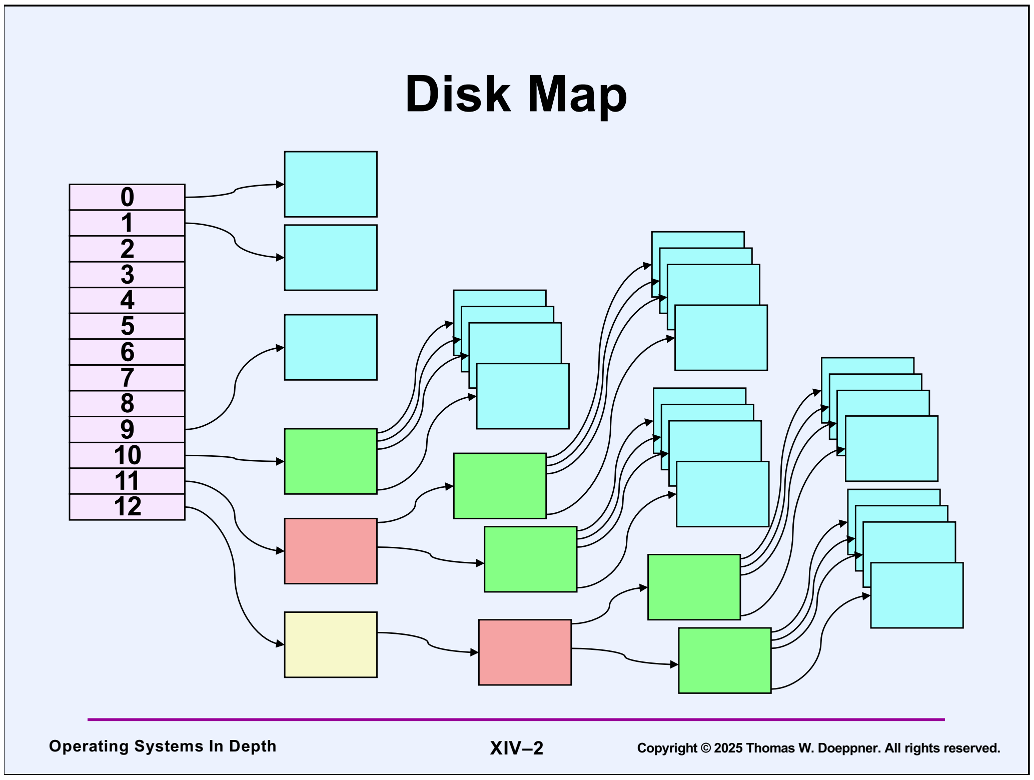
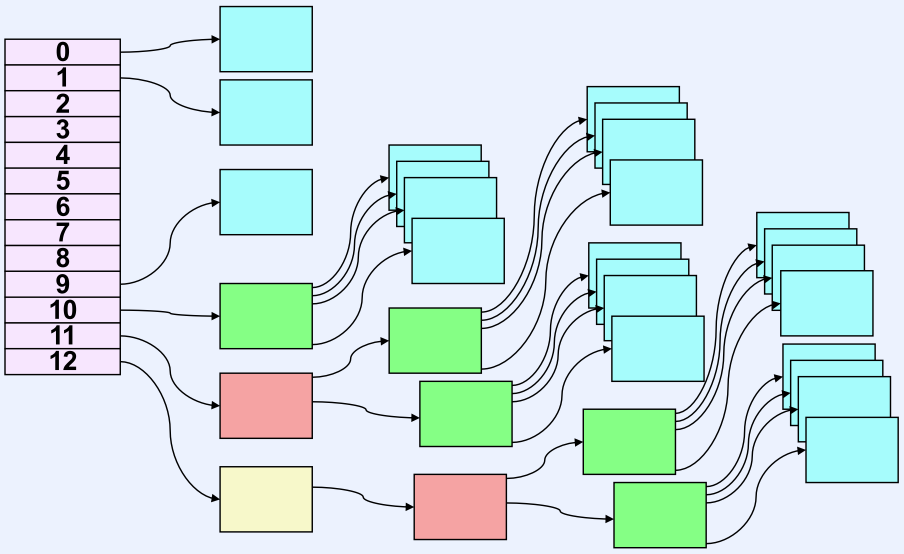
File Systems Part 2

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



Disk Map



The purpose of the disk-map portion of the inode is to represent where the blocks of a file are on disk. I.e., it maps block numbers relative to the beginning of a file into block numbers relative to the beginning of the file system. Each block is 1024 (1K) bytes long. (It was 512 bytes long in the original Unix file system.) The data structure allows fast access when a file is accessed sequentially, and, with the help of caching, reasonably fast access when the file is used for paging (and other "random" access).

The disk map consists of 13 pointers to disk blocks, the first 10 of which point to the first 10 blocks of the file. (Thus the pointers are actually block numbers on the disk.) The first 10Kb of a file are accessed directly. If the file is larger than 10Kb, then pointer number 10 points to a disk block called an **indirect block**. This block contains up to 256 (4-byte) pointers to **data blocks** (i.e., 256KB of data). If the file is bigger than this (256K +10K = 266K), then pointer number 11 points to a **double indirect block** containing 256 pointers to indirect blocks, each of which contains 256 pointers to data blocks (64MB of data). If the file is bigger than this (64MB + 256KB + 10KB), then pointer number 12 points to a **triple indirect block** containing up to 256 pointers to double indirect blocks, each of which contains up to 256 pointers pointing to single indirect blocks, each of which contains up to 256 pointers pointing to data blocks (potentially 16GB, although the real limit is 2GB, since the file size, a signed number of bytes, must fit in a 32-bit word).

If a disk pointer (block number) is zero, it's treated as if it points to a block containing all zeroes (without the need for this block of zeroes to actually exist on disk).

The Weenix version of S5FS supports the indirect block, but not the double or triple indirect blocks.

Friday’s Quiz

Suppose a new file is created. (At this point it occupies zero blocks.) Then one byte is written to it at byte offset $(266 \times 2^{10}) + 1$. Assume the block size is 2^{10} and block addresses occupy four bytes. How many blocks are required to represent the file, not counting its inode?

a) more than 270

b) 270

c) 3

d) 1

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Quiz 1

Suppose one now writes to all locations in the file, from its beginning up to the location written to in the previous slide (byte offset (266×2^{10}) + 1). How many blocks are required to represent the file, not counting its inode?

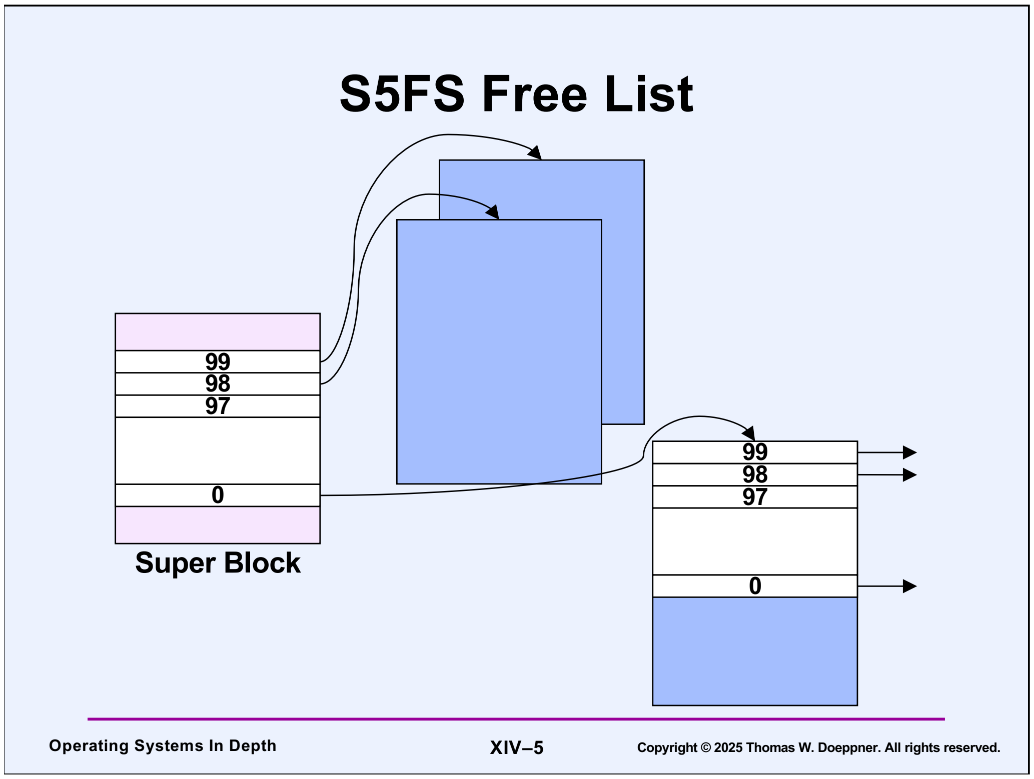
a) more than 270

b) 270

c) 3

d) 1

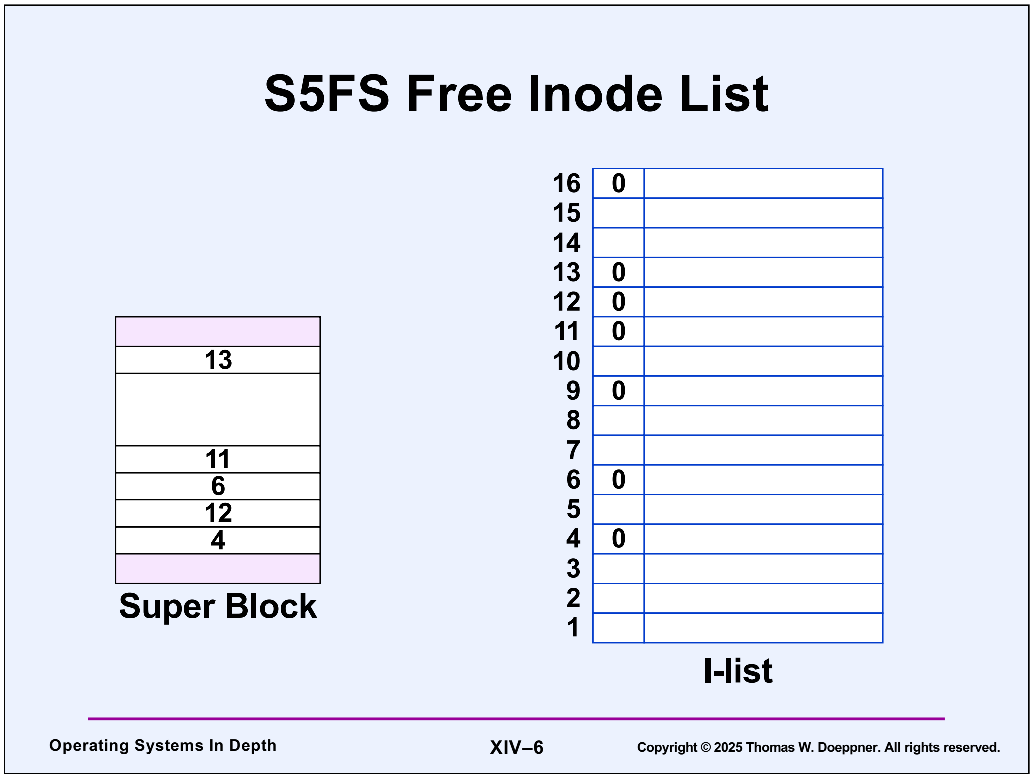
Copyright © 2025 Thomas W. Doeppner. All rights reserved.



S5FS Free List

Free disk blocks are organized as shown in the picture. The superblock contains the address of up to 100 free disk blocks. The last of these disk blocks contains 100 pointers to additional free disk blocks. The last of these pointers points to another block containing up to *n* free disk blocks, etc., until all free disk blocks are represented. Thus, most requests for a free block can be satisfied by merely getting an address from the superblock. When the last block reference by the superblock is consumed, however, a disk read must be done to fetch the addresses of up to 100 more free disk blocks. Freeing a disk block results in reconstructing the list structure.

This organization, though very simple, scatters the blocks of files all over the surface of the disk. When allocating a block for a file, one must always use the next block from the free list; there is no way to request a block at a specific location. No matter how carefully the free list is ordered when the file system is initialized, it becomes fairly well randomized after the file system has been used for a while. This is an issue that we'll address when we look at other file systems.



S5FS Free Inode List

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

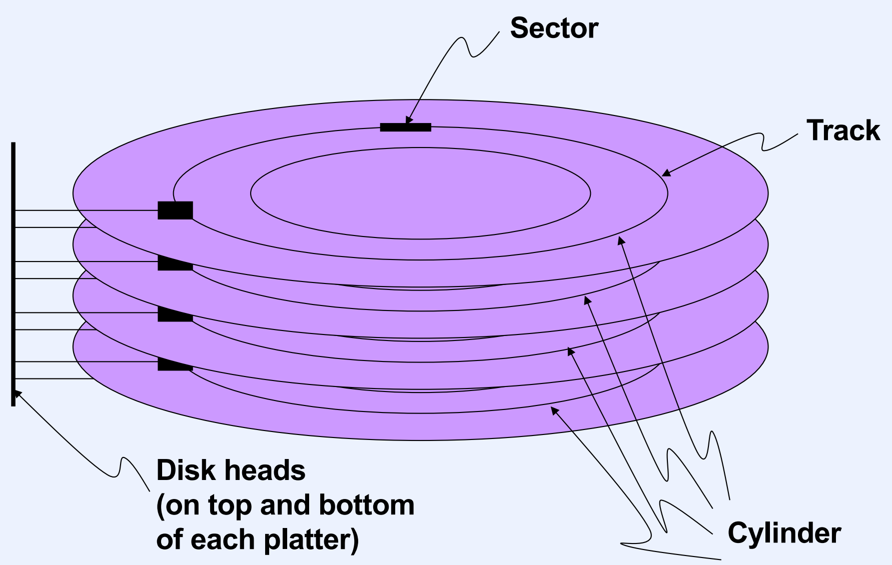
Inodes are allocated from the I-list. Free inodes are represented simply by zeroing their mode bits. The superblock contains a cache of indices of free inodes. When a free inode is needed (i.e., to represent a new file), its index is taken from this cache. If the cache is empty, then the I-list is scanned sequentially until enough free inodes are found to refill the cache.

To speed this search somewhat, the cache contains a reference to the inode with the smallest index that is known to be free (inode 4 in the slide). When an inode is free, it is added to the cache if there is room, and its mode bits are zeroed on disk.

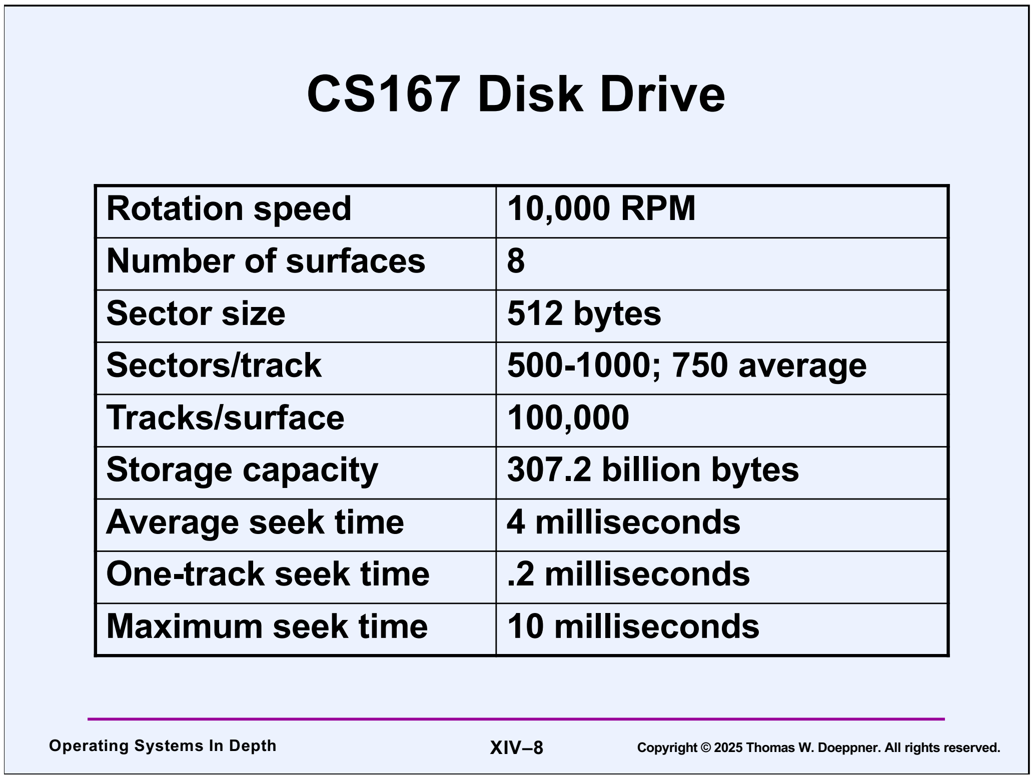
The Weenix version of the free inode list is done differently. It's a singly linked list of free inodes, with the index of the first in the list maintained in the superblock.

Inode 0 is special: it refers to the root directory of the file system.

Disk Architecture



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

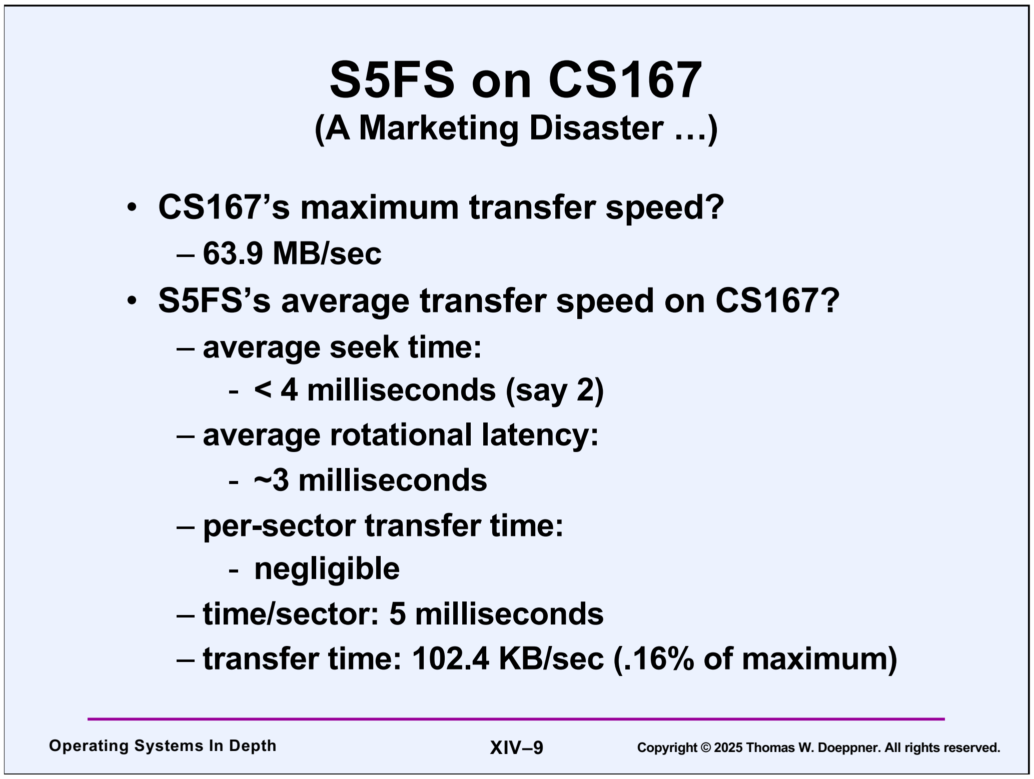


CS167 Disk Drive

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500-1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 milliseconds
One-track seek time	.2 milliseconds
Maximum seek time	10 milliseconds

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

The slide lists the characteristics of a hypothetical disk drive.



S5FS on CS167
(A Marketing Disaster …)

- CS167's maximum transfer speed?
  - 63.9 MB/sec
- S5FS's average transfer speed on CS167?
  - average seek time:
    - < 4 milliseconds (say 2)
  - average rotational latency:
    - ~3 milliseconds
  - per-sector transfer time:
    - negligible
  - time/sector: 5 milliseconds
  - transfer time: 102.4 KB/sec (.16% of maximum)

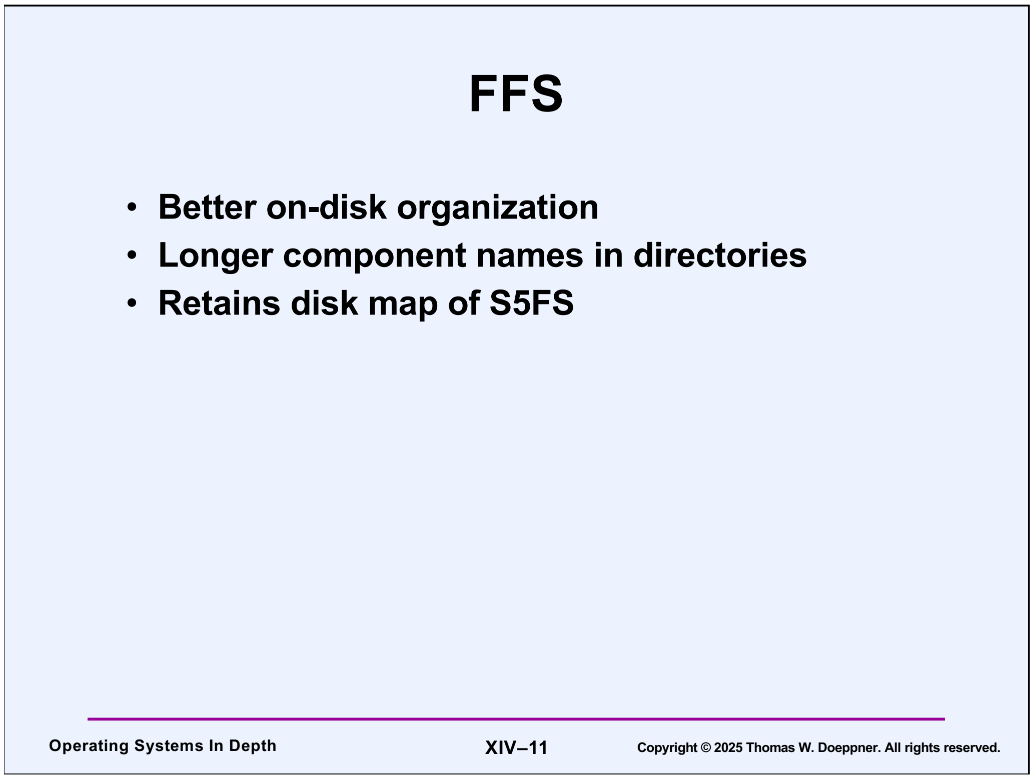
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Recall that, due to how the block free list is organized, over time the blocks of a typical file will be randomly scattered on the disk.

What to Do About It?

- Hardware
  - employ pre-fetch buffer
    - filled by hardware with what's underneath head
    - helps reads; doesn't help writes
- Software
  - better on-disk data structures
    - increase block size
    - minimize seek time
    - reduce rotational latency

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

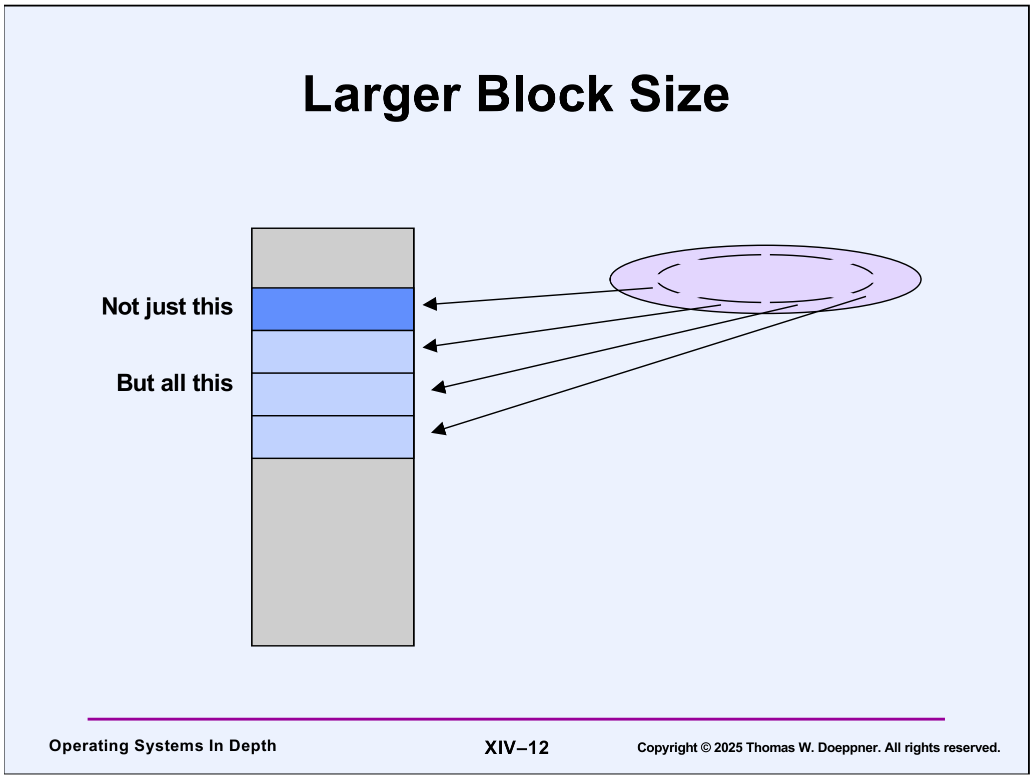


FFS

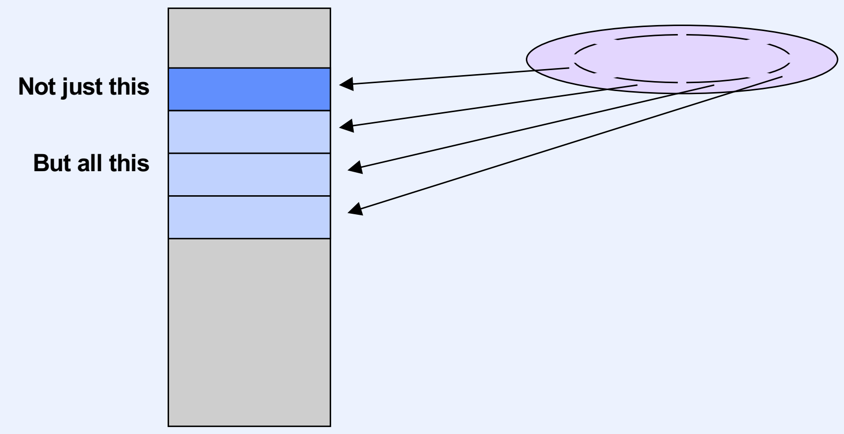
- Better on-disk organization
- Longer component names in directories
- Retains disk map of S5FS

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

FFS (Fast File System) we developed at UC Berkeley in the early 80s as part of BSD Unix.

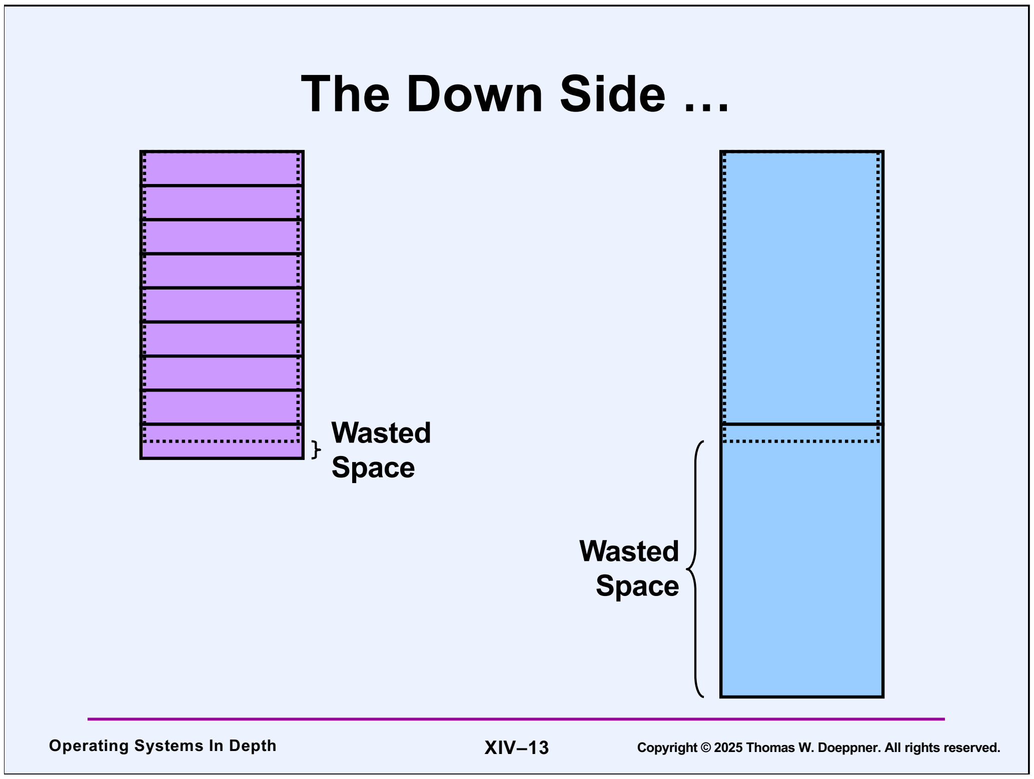


Larger Block Size

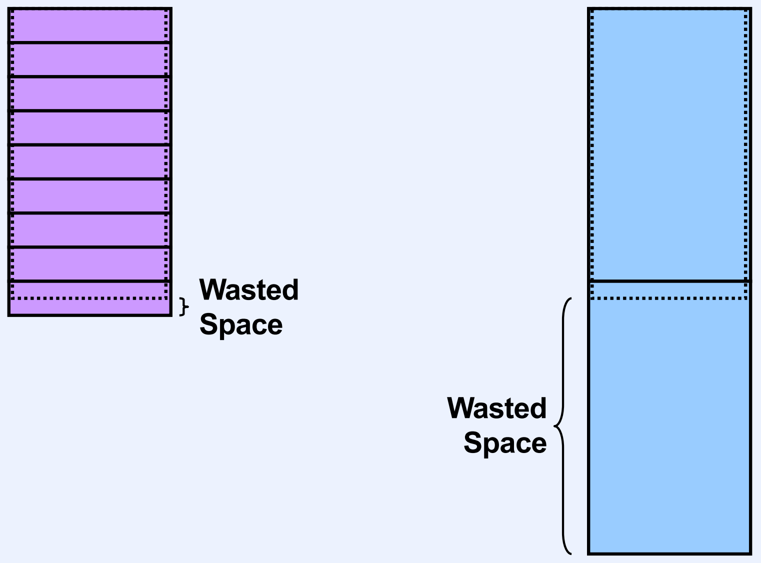


Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Increasing the block size is a rather obvious improvement to a file system. The major cost of a disk operation is the initial setup: positioning the disk head. The time it takes to transfer data is small in comparison. Thus, the best strategy would seem to be to transfer as much data as possible with each operation.



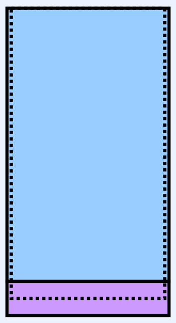
The Down Side …



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Of course, with a larger block size there is a greater wastage of disk space: on average one-half block per file. For large files this is no big deal, but if a system has a large number of small files, this is a big deal. This sort of wasted space is known as **internal fragmentation**.

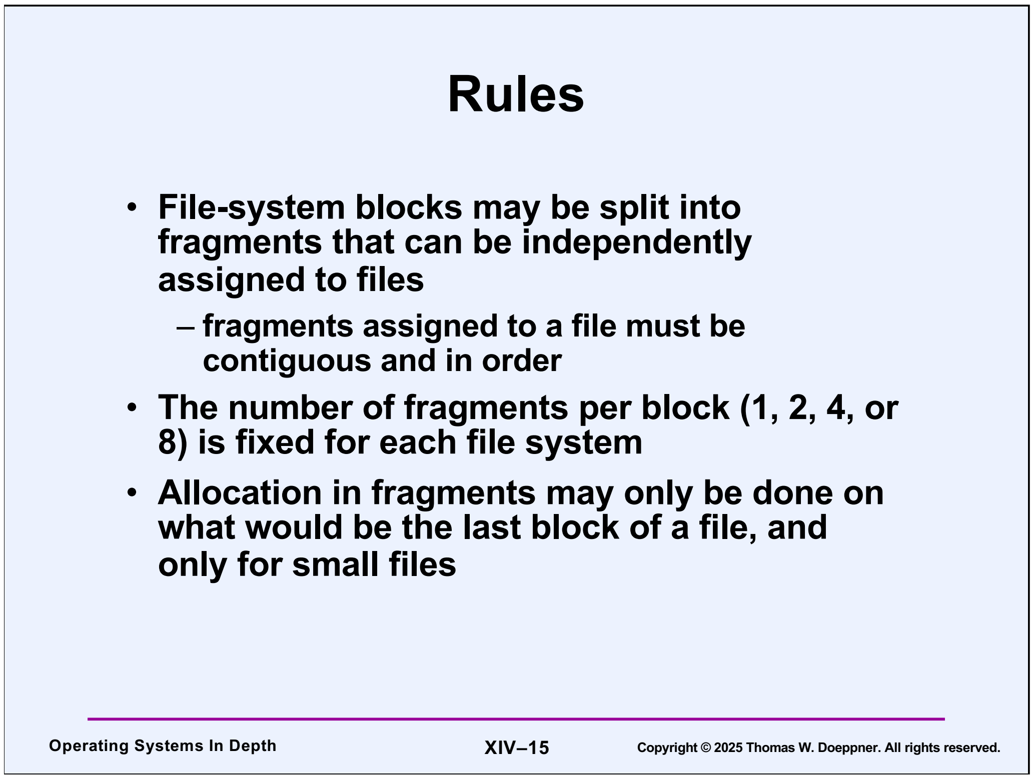
Two Block Sizes …



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

One approach to dealing with this fragmentation problem is to use two block sizes: use large blocks for most of the file, then finish off with small blocks. Thus, for most of the file we get the benefits of the large block size, but we deal with the internal fragmentation issue with the use of small blocks in what would have been the last block of the file.

In FFS, these small blocks are known as **fragments**.



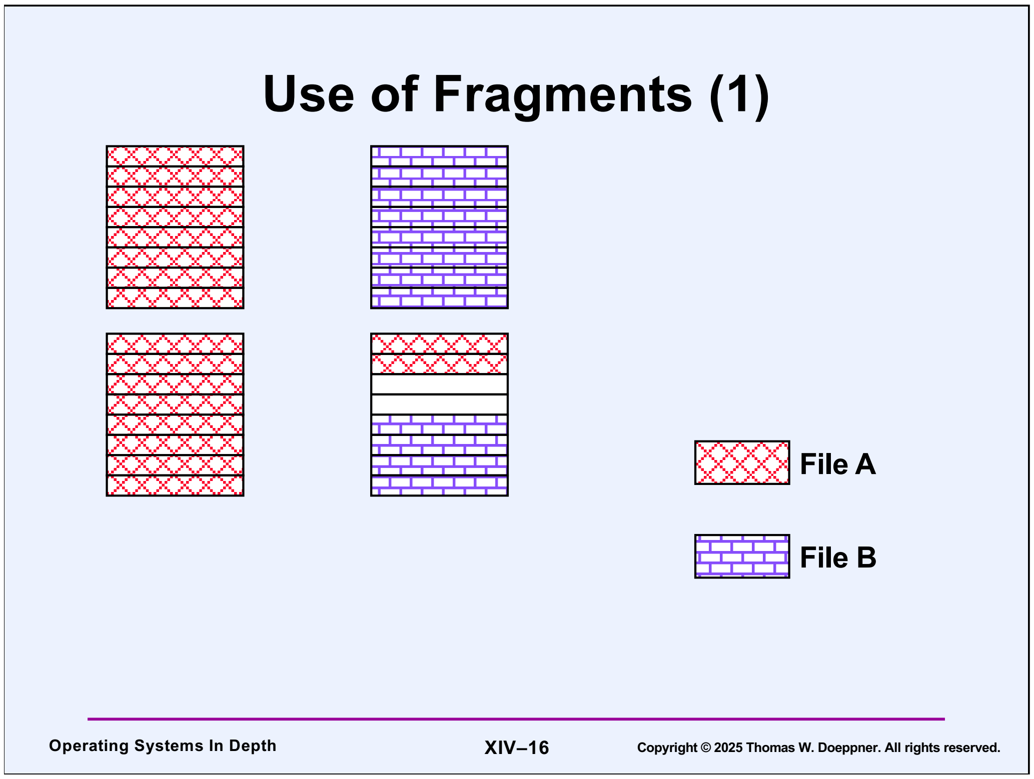
Rules

- File-system blocks may be split into fragments that can be independently assigned to files
  - fragments assigned to a file must be contiguous and in order
- The number of fragments per block (1, 2, 4, or 8) is fixed for each file system
- Allocation in fragments may only be done on what would be the last block of a file, and only for small files

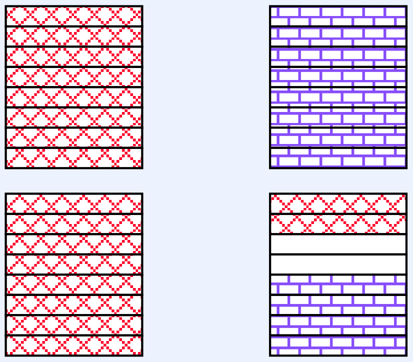
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Fragmentation is what the technique is called in FFS for reducing disk space wastage due to file sizes not being an integral multiple of the block size. Files are normally allocated in units of blocks, since this allows the system to transfer data in relatively large, block-size units. But this causes space problems if we have lots of small files, where the average amount of space wasted per file (half the block size) is an appreciable fraction of the size of the file (the wastage can be far greater than the size of the file for very small files). The ideal solution might be to reduce the block size for small files, but this could cause other problems; e.g., small files might grow to be large files.

The FFS solution is to allocate files in units of blocks, except what would be the last block of a file may be allocated in smaller units, called **fragments**, with 2, 4, or 8 fragments per blocked, the number being fixed for each individual file system. To simplify file representation, the fragments assigned to any one file must be contiguous and in order. Furthermore, since internal fragmentation is an issue only for small files, such fragments are used only with small files. A small file is defined to be one that contains no indirect blocks.



Use of Fragments (1)



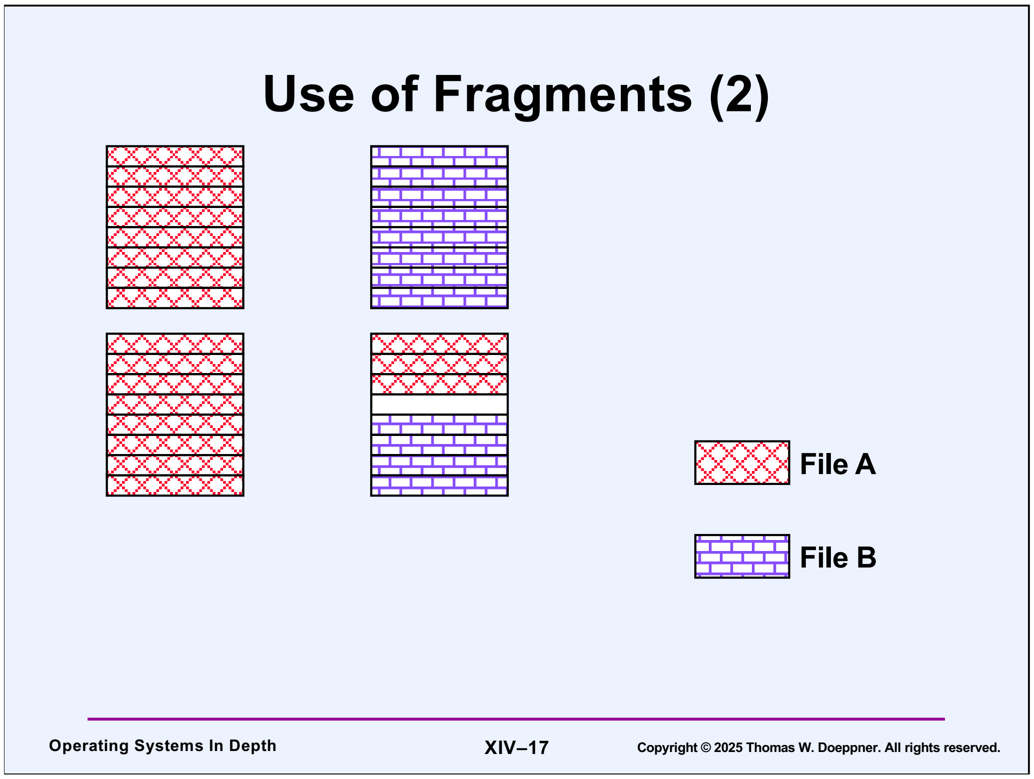
▨ File A

▦ File B

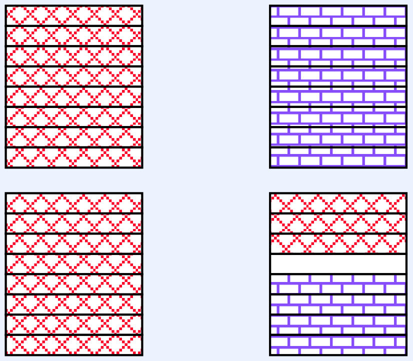
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

This example illustrates a difficulty associated with the use of fragments. The file system must preserve the invariant that fragments assigned to a file must be contiguous and in order, and that allocation of fragments may be done only on what would be the last block of the file. In the picture, the direction of growth is downwards. Thus, file A may easily grow by up to two fragments, but file B cannot easily grow within this block.

In the picture, file A is 18 fragments in length, file B is 12 fragments in length.



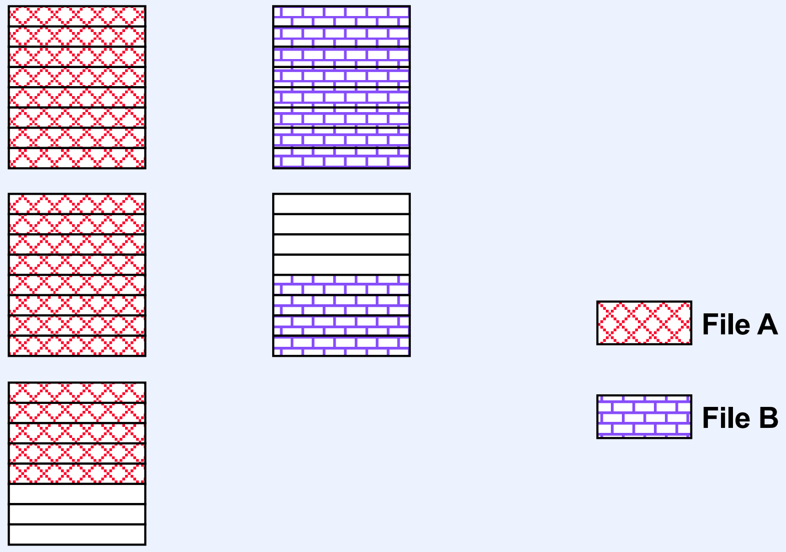
Use of Fragments (2)



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

File A grows by one fragment.

Use of Fragments (3)



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

File A grows by two more fragments, but since there is no space for it, the file system allocates another block and copies file A's fragments into it. How much space should be available in the newly allocated block? If the newly allocated block is entirely free, i.e., none of its fragments are used by other files, then further growth by file A will be very cheap. However, if the file system uses this approach all the time, then we do not get the space-saving benefits of fragmentation. An alternative approach is to use a "best-fit" policy: find a block that contains exactly the number of free fragments needed by file A, or if such a block is not available, find a block containing the smallest number of contiguous free fragments that will satisfy file A's needs.

Which approach is taken depends upon the degree to which the file system is fragmented. If disk space is relatively unfragmented, then the first approach is taken ("optimize for time"). Otherwise, i.e., when disk space is fragmented, the file system takes the second approach ("optimize for space").

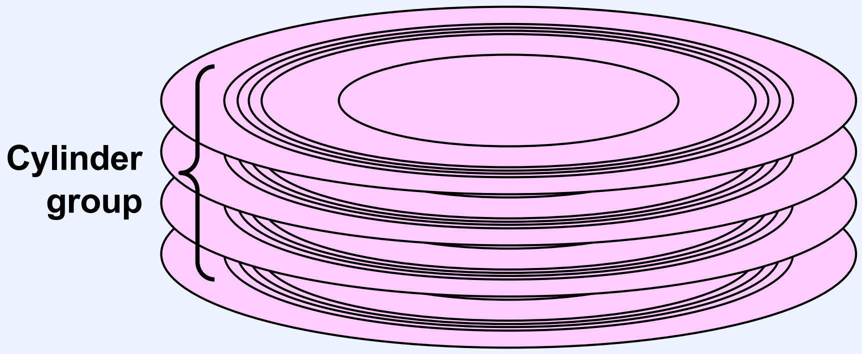
The points at which the system switches between the two policies is parameterized in the superblock: a certain percentage of the disk space, by default 10%, is reserved for superuser. (Disk-allocation techniques need a reasonable chance of finding free disk space in each cylinder group in order to optimize the layout of files.) If the total amount of fragmented free disk space (i.e., the total amount of free disk space not counting that portion consisting of whole blocks) increases to 8% of the size of the file system (or, more generally, increases to 2% less than the reserve), then further allocation is done using the best-fit approach. Once this approach is being used, if the total amount of fragmented free disk space drops below 5% (or half of the reserve), then further allocation is done using the whole-block technique.

Minimizing Seek Time

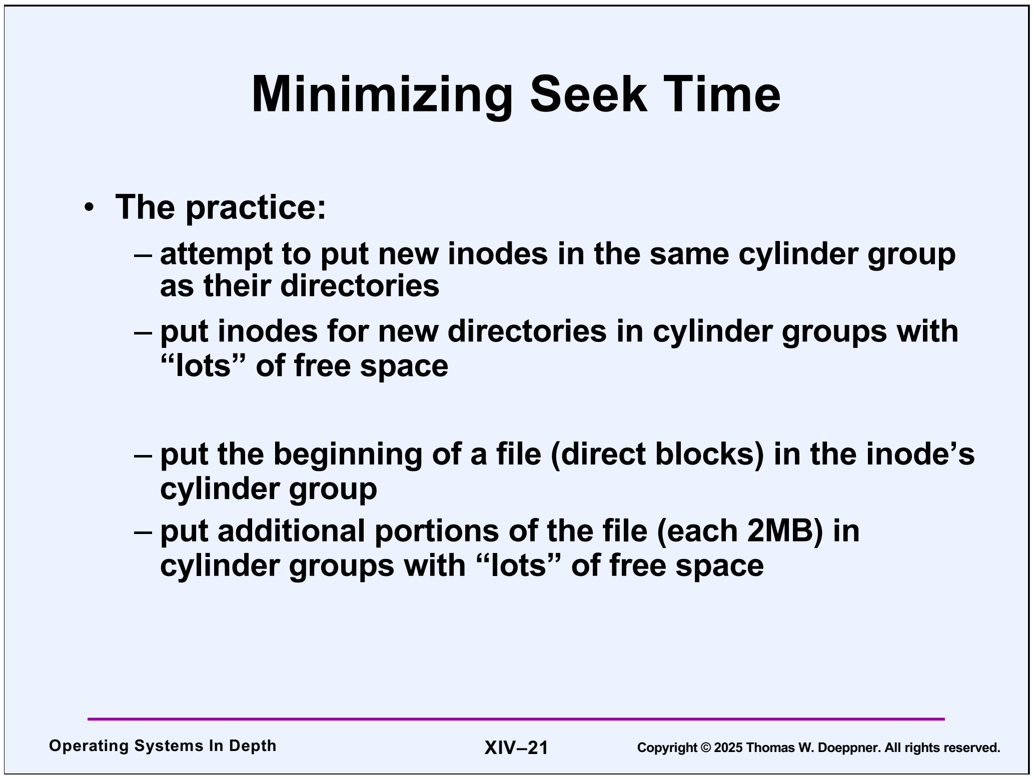
- Keep related things close to one another
- Separate unrelated things

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Cylinder Groups



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

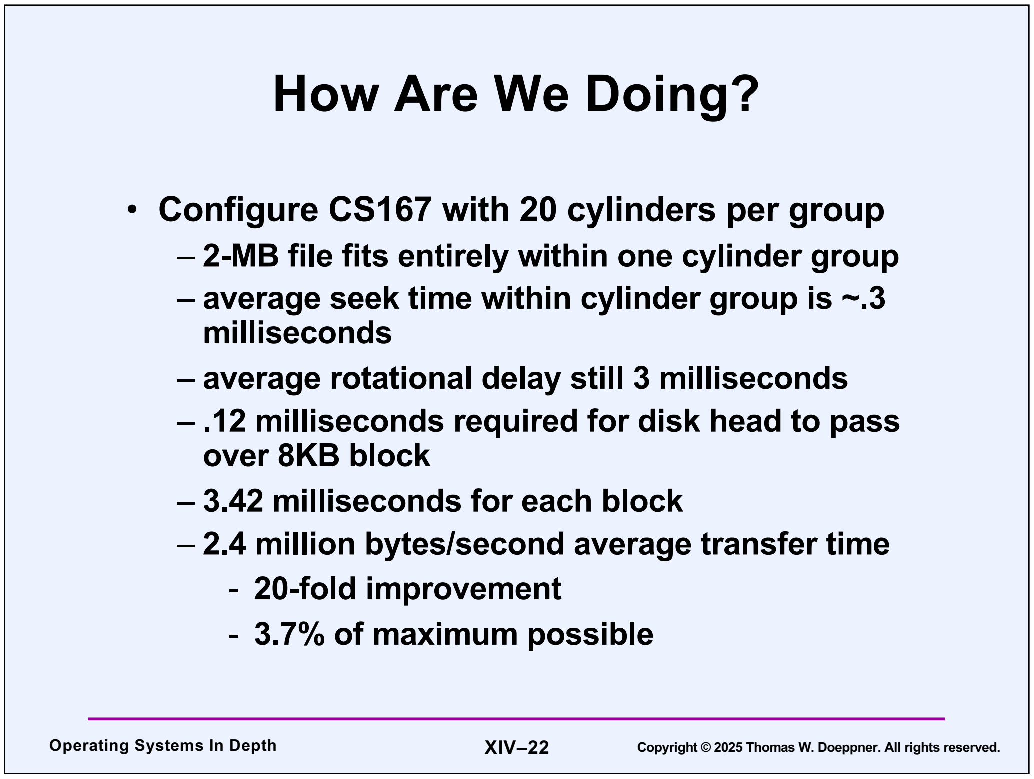


Minimizing Seek Time

- The practice:
  - attempt to put new inodes in the same cylinder group as their directories
  - put inodes for new directories in cylinder groups with “lots” of free space
  - put the beginning of a file (direct blocks) in the inode’s cylinder group
  - put additional portions of the file (each 2MB) in cylinder groups with “lots” of free space

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

One of the major components (in terms of time) of a disk I/O operation is the positioning of the disk head. In S5FS we didn’t worry about this, but in FFS we would like to lay out files on disk so as to minimize the time required to position the disk head. If we know exactly what the contents of an entire file system will be when we create it, then, in principle, we could lay files out optimally. But we don’t have this sort of knowledge, so, in FFS, a reasonable effort is made to lay files out “pretty well.”



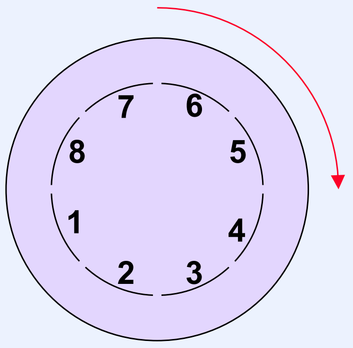
How Are We Doing?

- Configure CS167 with 20 cylinders per group
  - 2-MB file fits entirely within one cylinder group
  - average seek time within cylinder group is ~.3 milliseconds
  - average rotational delay still 3 milliseconds
  - .12 milliseconds required for disk head to pass over 8KB block
  - 3.42 milliseconds for each block
  - 2.4 million bytes/second average transfer time
    - 20-fold improvement
    - 3.7% of maximum possible

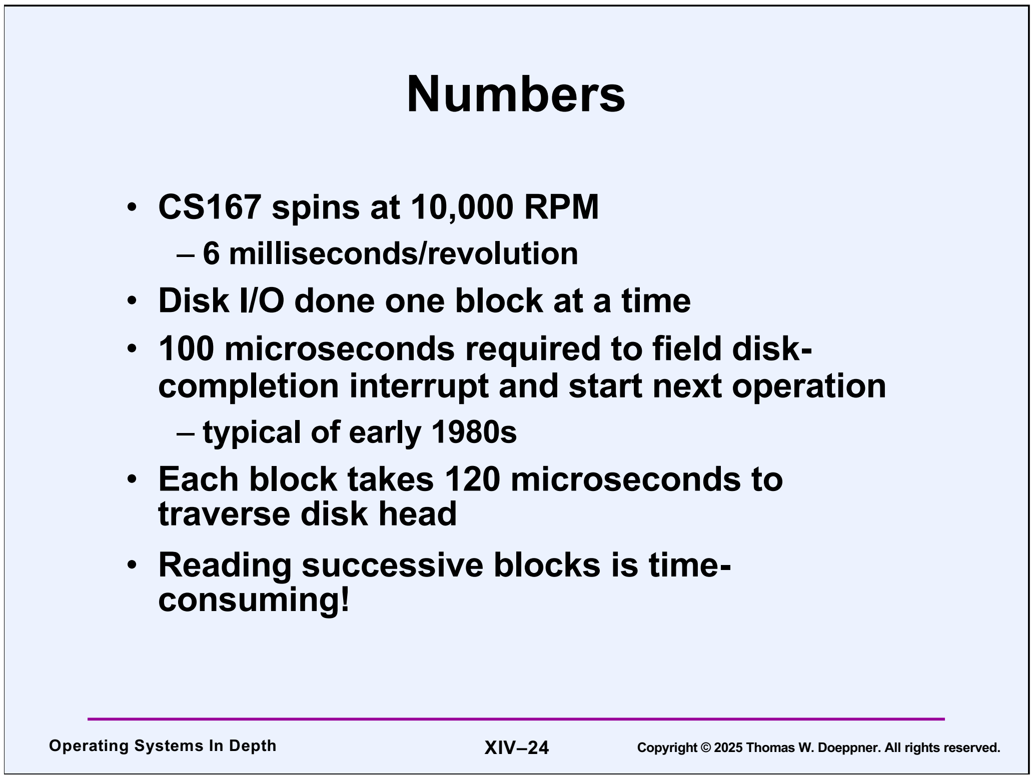
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

To make our numbers look as good as possible, we assume the disk heads are already positioned in the correct cylinder group.

Minimizing Latency (1)



Copyright © 2025 Thomas W. Doeppner. All rights reserved.



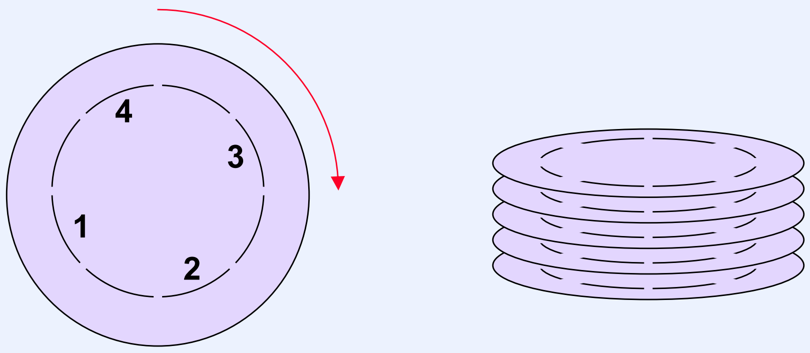
Numbers

- CS167 spins at 10,000 RPM
  - 6 milliseconds/revolution
- Disk I/O done one block at a time
- 100 microseconds required to field disk-completion interrupt and start next operation
  - typical of early 1980s
- Each block takes 120 microseconds to traverse disk head
- Reading successive blocks is time-consuming!

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

UNIX systems of the early 80s were designed to read just one block at a time: the kernel buffers into which blocks were read were just big enough to hold single blocks (a fair number of such buffers were available in the kernel, but each held just one block). Thus, for example, reading two consecutive blocks from the disk required the processor to issue a read request to the disk controller for the first block, then, after the controller interrupted the processor to indicate that the read had completed, the processor would issue a read request for the second block. Unfortunately, by the time the processor could start the second disk request, the disk block had already rotated beyond the disk head and thus the controller had to wait an entire disk rotation before it could start the read of the next block.

Minimizing Latency (2)



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

How’re We Doing Now? (part 1)

- Time to read successive blocks (two-way interleaving):
  - after request for second block is issued, must wait 20 microseconds for the beginning of the block to rotate under disk head
    - factor of 300 improvement!

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

How’re We Doing Now? (part 2)

- Same setup as before
  - 2-MB file within one cylinder group
  - actually fits in one cylinder
  - block interleaving employed: every other block is skipped
  - .3-millisecond seek to that cylinder
  - 3-millisecond rotational delay for first block
  - 50 blocks/track, but 25 read in each revolution
  - 10.24 revolutions required to read all of file
  - 32.4 MB/second (50% of maximum possible)

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Quiz 2

If file access is one (8KB) block at a time and we employ 2-way block interleaving, can we do better than the 50% of maximum transfer speed achieved by FFS?

a) yes – we can get arbitrarily close to 100%

b) yes, but the limit is somewhere between 50% and 100%

c) no – we've reached the limit

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Further Improvements?

- S5FS: 0.16% of capacity
- FFS without block interleaving: 3.8% of capacity
- FFS with block interleaving: 50% of capacity
- What next?

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Larger Transfer Units

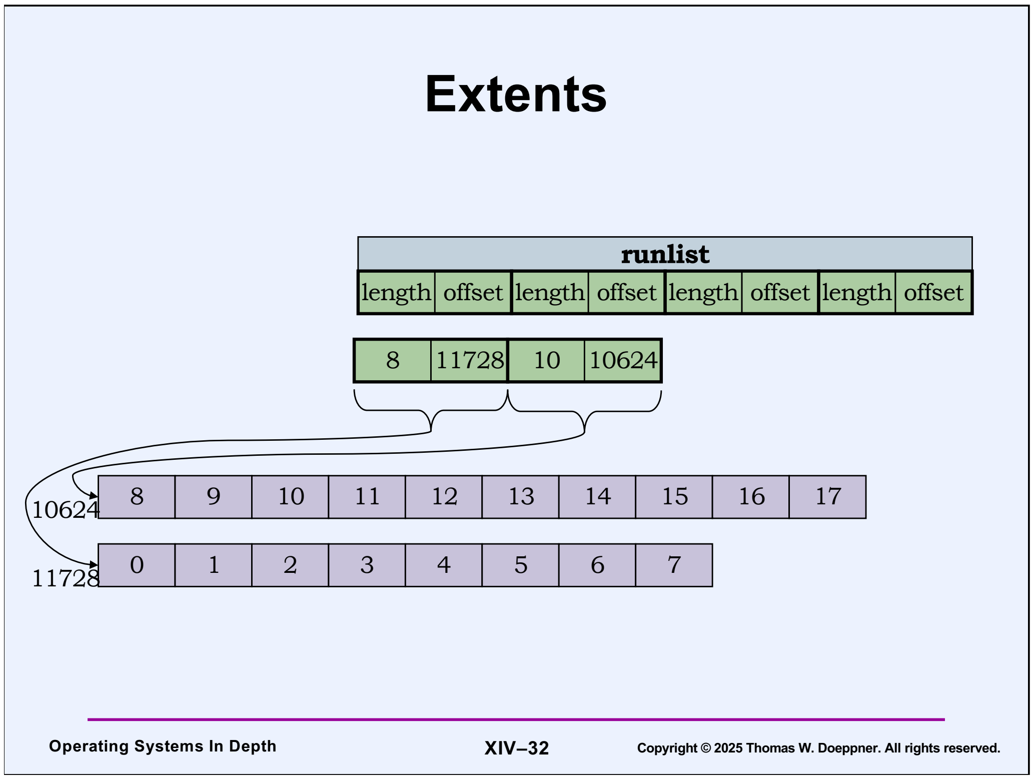
- Allocate in whole tracks or cylinders
  - too much wasted space
- Allocate in blocks, but group them together
  - transfer many at once
  - wasted space, but not as bad

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

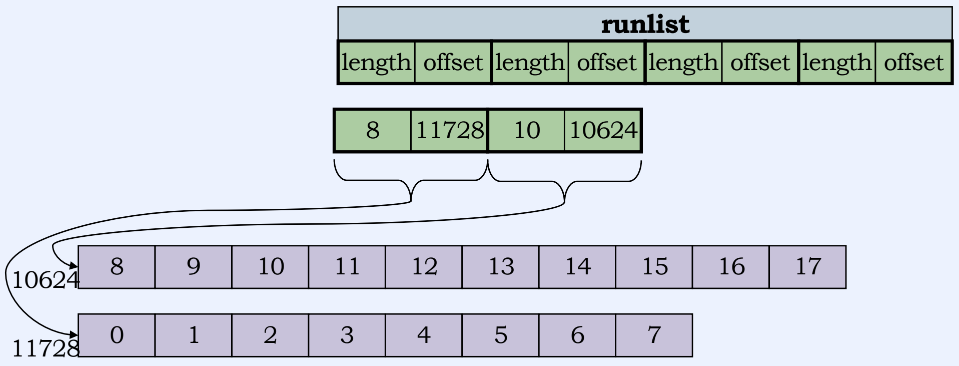
Block Clustering

- Allocate space in blocks, eight at a time
- Linux's Ext2 (an FFS clone):
  - allocate eight blocks at a time
  - extra space is available to other files if there is a shortage of space
- FFS on Solaris (~1990)
  - delay disk-space allocation until:
    - 8 blocks are ready to be written
    - or the file is closed

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



Extents



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

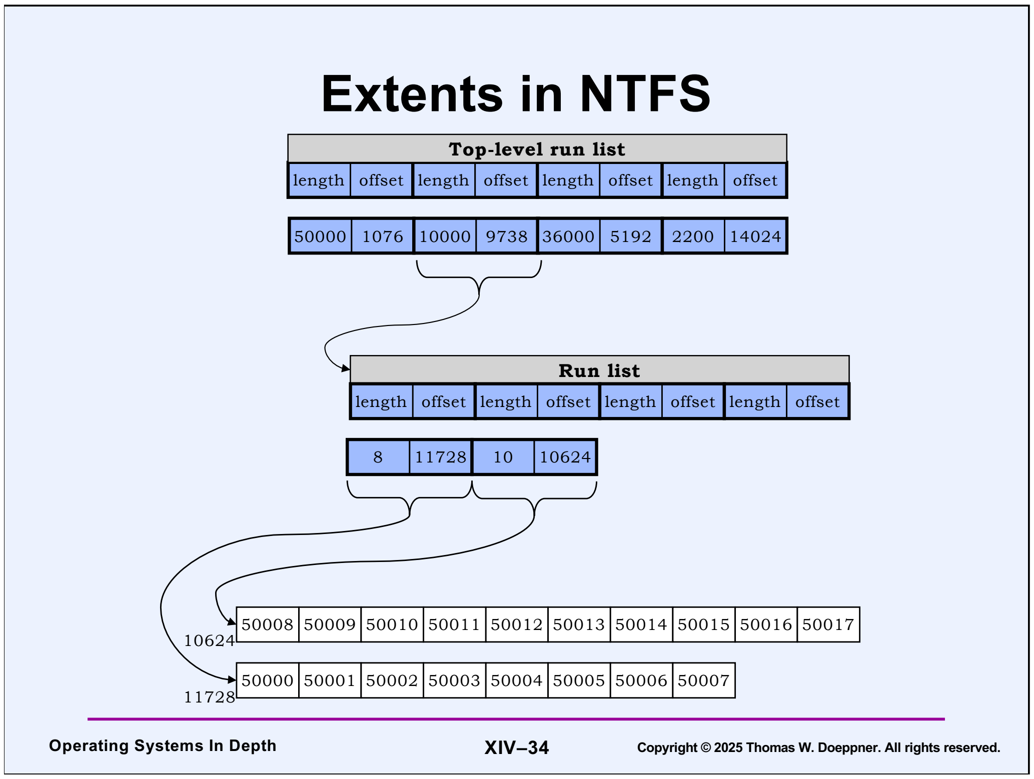
Extents is a technique for allocating space on disk for files in variable-length units. As shown on the slide, we keep track of where a file's extents are by use of a runlist data structure that's stored on disk and brought into kernel memory when needed.

Note that we'll need a different approach than used for S5FS and FFS for managing buffer space in the kernel, since we'll be reading in variable-sized chunks of files from disk.

Problems with Extents

- Could result in highly fragmented disk space
  - lots of small areas of free space
  - solution: use a *defragmenter*
- Random access
  - linear search through a long list of extents
  - solution: multiple levels

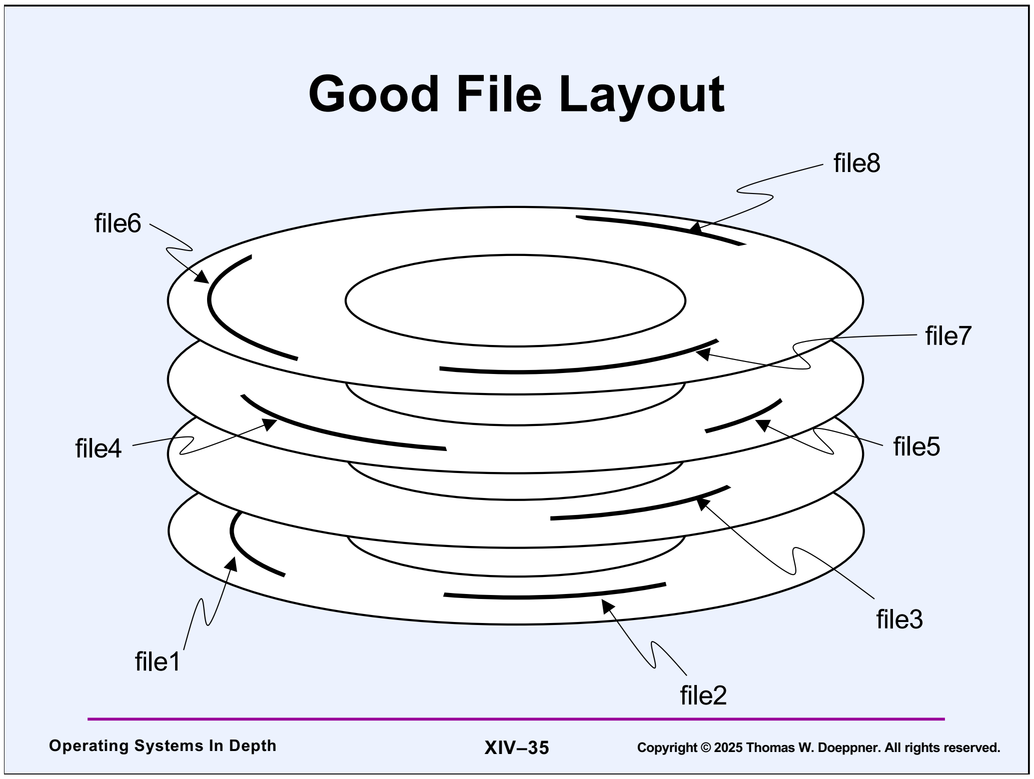
Copyright © 2025 Thomas W. Doeppner. All rights reserved.



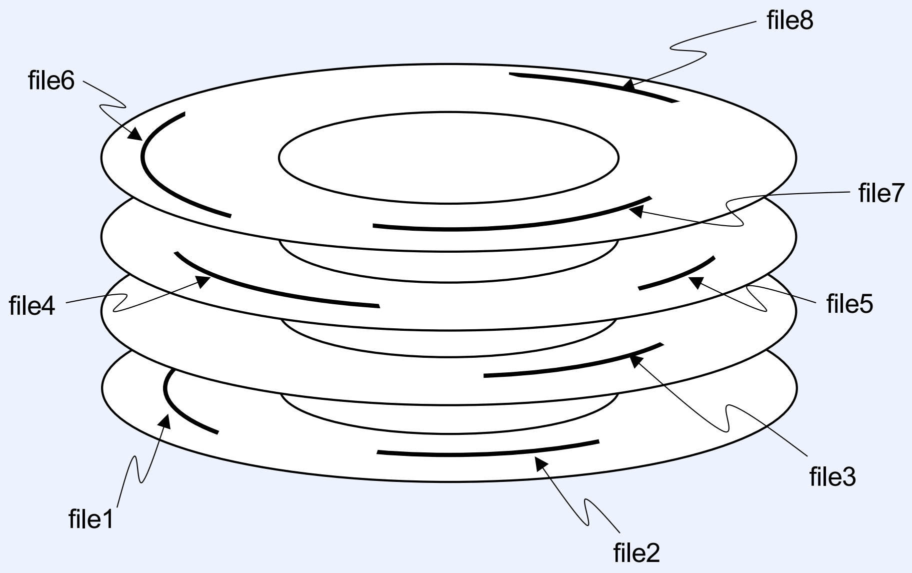
Extents in NTFS

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

The slide shows a two-level run list, as used in in NTFS (used in Microsoft Windows). To find block 50011, we search the top-level run list for a pointer to the file record containing the run list that refers to this block. In this case it's pointed to by the second entry, which contains a pointer to the run list that refers to extents containing file blocks starting with 50,000 and continuing for 10,000 blocks. We show the first two references to extents in that run list. The block we're after is in the second one.



Good File Layout



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

The slide shows a number of files, each stored in a single extent. This allows fast, perhaps optimal read access for each file, once the disk is positioned for the beginning of that file. Of course, files requiring multiple extents aren't read quite so quickly. However, we seem to be optimizing for (sequential) read access. Is this the right strategy?

Quiz 3

So far we've been attempting to optimize *logical locality* in file systems: bytes that are logically near one another within a file are physically near one another on disk.

a) This is important on both single-user systems and busy (multi-client) file servers

b) This is important on single-user systems, but of marginal importance on busy (multi-client) file servers

c) This is of marginal importance on single-user systems

Copyright © 2025 Thomas W. Doeppner. All rights reserved.