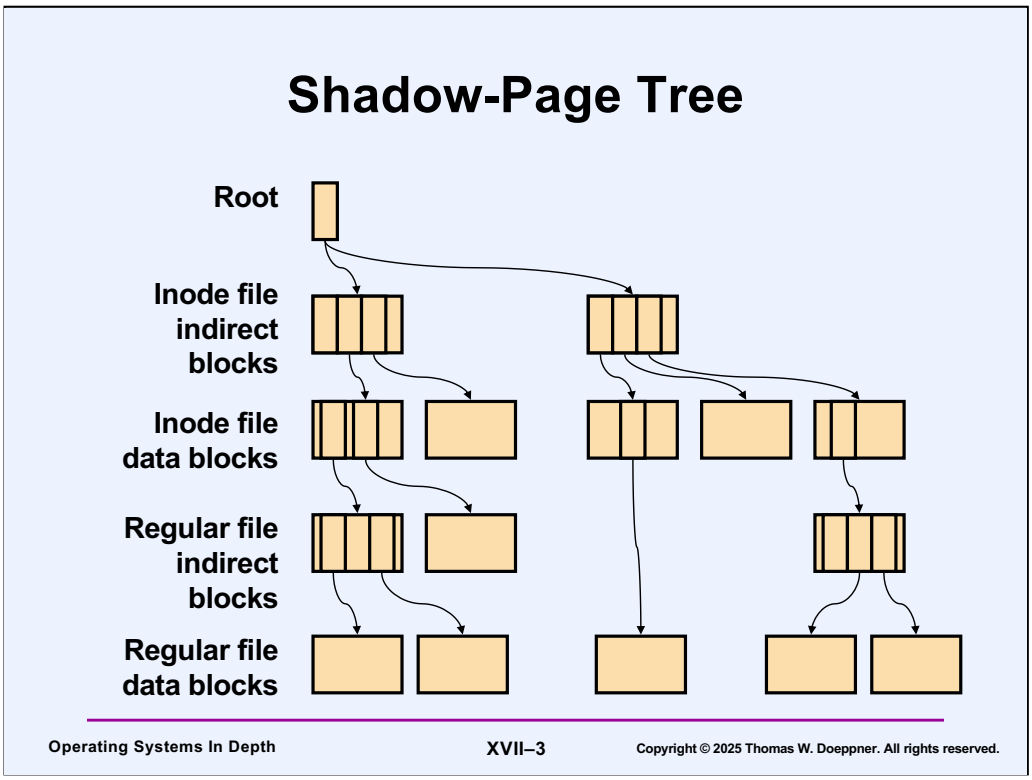
File Systems Part 4

Shadow Paging

- Refreshingly simple
- Provides historical snapshots
- Examples
  - WAFL (Network Appliance)
  - ZFS (Sun)

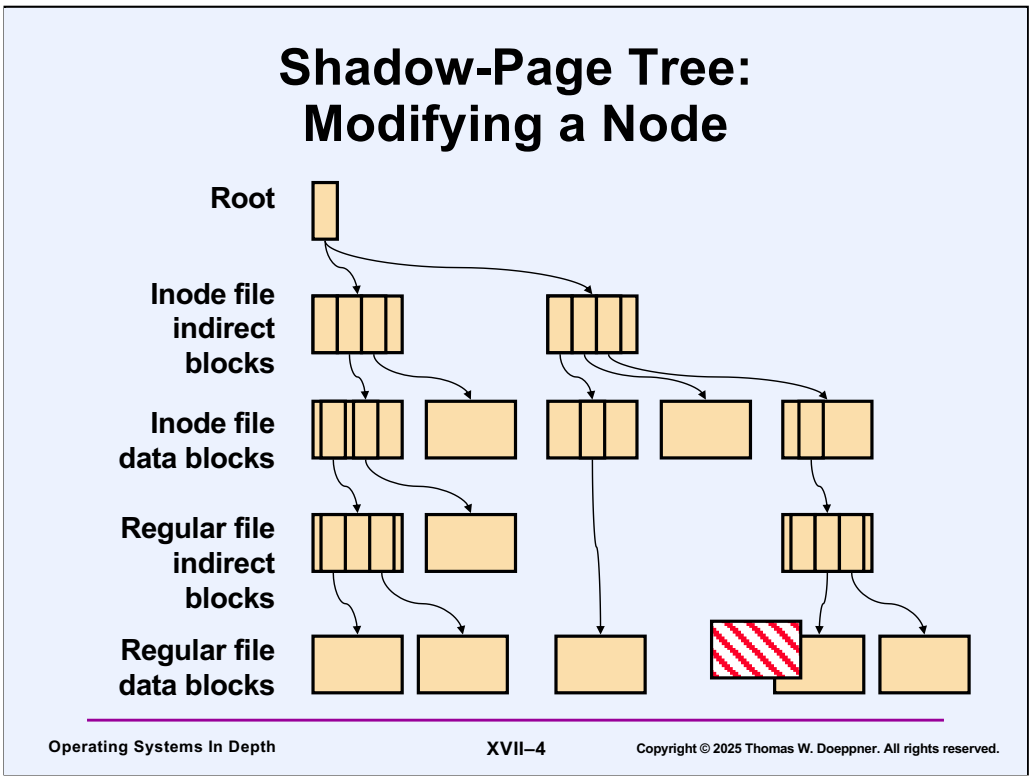


This and the next few slides are based on a description of the WAFL (Write-Anywhere File Layout) file system of Network Appliance Corporation, which is from:

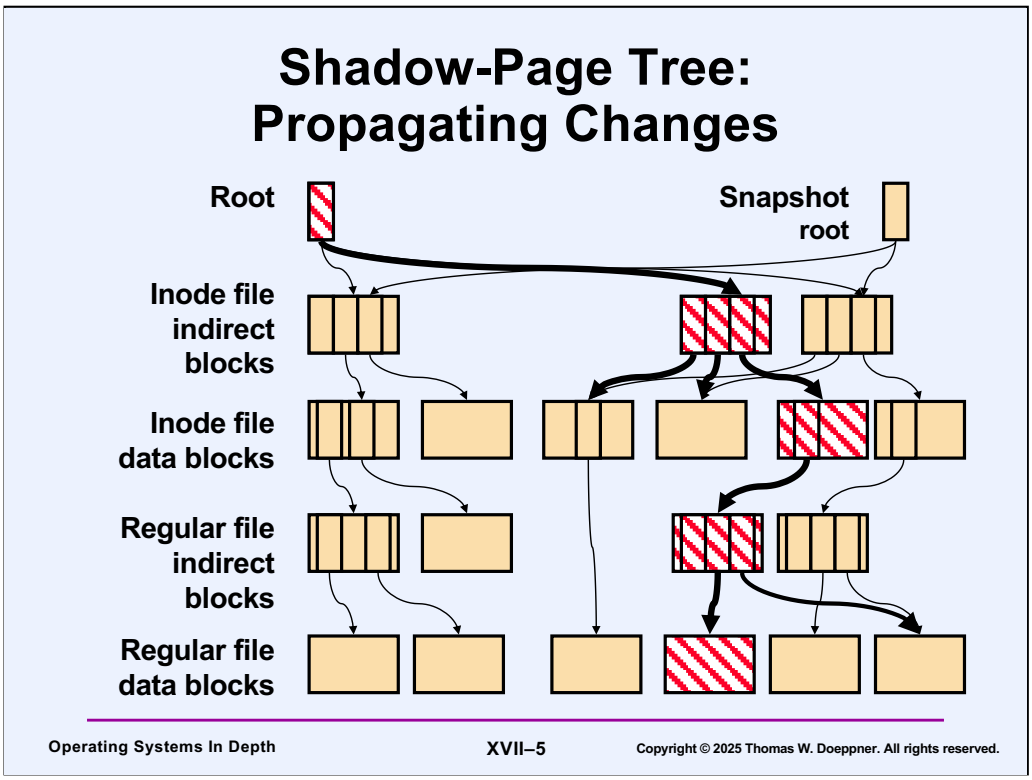
Hitz, D., J. Lau, M. Malcolm (1994). File System Design for an NFS File Server Appliance. Proceedings of USENIX Winter 1994 Technical Conference.

A file system is represented as a tree. Let's assume that the array of inodes is represented as a file, much as things are done in NTFS with its master file table (MFT). Thus we can look up an inode number in this inode file, obtaining the corresponding inode. From an inode we can find the blocks of the corresponding file. Let's assume that files (inode files and regular files) are represented via a disk map consisting of a number of indirect blocks referring to data blocks. In the diagram above, the root node points to the indirect blocks of the inode file. Each of these indirect blocks point to data blocks of the inode file (containing the actual inodes). The inodes contain disk maps for the files they represent, which are indirect blocks referring to data blocks.

Thus the leaves of the tree are the data blocks of all the files in the file system. As we'll see in the next two slides, links between levels of this file-system tree go in both directions (up as well as down).



Step1: A leaf node in a shadow-page tree is modified. A copy is made of the node, and the copy is modified.



Step 2: Copies are made of the leaf node's ancestors all the way up to the root, each modified so as to point to the copied node below it. The original root (snapshot root) points to the tree as it was before the leaf was modified.

Step 3: The last thing done is to modify the original root so as to point to the modified tree. Before the root is modified, it refers to the original tree. After it's modified, it points to the new tree. Thus a single write to disk makes the change happen.

Benefits

- Each update results in a new shadow-page tree (having much in common with the previous one)
- The current root identifies the current tree
- If the system crashes after an update has been made, but before changes are propagated to the new root, the update is lost
  - a single write (to the root) effectively serves as a commit
- Older roots refer to previous states of the file system – snapshots

In practice, new snapshots aren't maintained after each update, but after a collection of updates – perhaps hourly.

Quiz 1

When the shadow-page tree is updated:

a) all file-system data must be written to disk synchronously: writes may be cached for the sake of reads, but write system calls may not return until the data is on disk

b) file-system data may be cached (and written asynchronously) only if all lower parts of the tree are written to disk before upper parts

c) file-system data may be cached (and written asynchronously) as long as the root is written last

Desirable Properties of Directories

- No restrictions on names
- Fast
- Space-efficient

Implementation Strategies

- Fixed-sized entries
- Variable-sized entries

- Sequential search
- Hashing
- Balanced search trees

S5FS Directories

Component Name	Inode Number

directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

An S5FS directory consists of an array of pairs of **component name** and **inode number**, where the latter identifies the target file's **inode** to the operating system. Note that every directory contains two special entries, "." and "..". The former refers to the directory itself, the latter to the directory's parent (in the case of the slide, the directory is the root directory and has no parent, thus its ".." entry is a special case that refers to the directory itself).

S5FS directories are searched sequentially, from beginning to end. Entries are not placed in alphabetical order (or any particular order), thus other search algorithms are not feasible.

What greatly simplifies searching as well as allocation and deletion of entries is that the component-name field of each entry is of a fixed width (28 bytes in the Weenix version).

Quiz 2

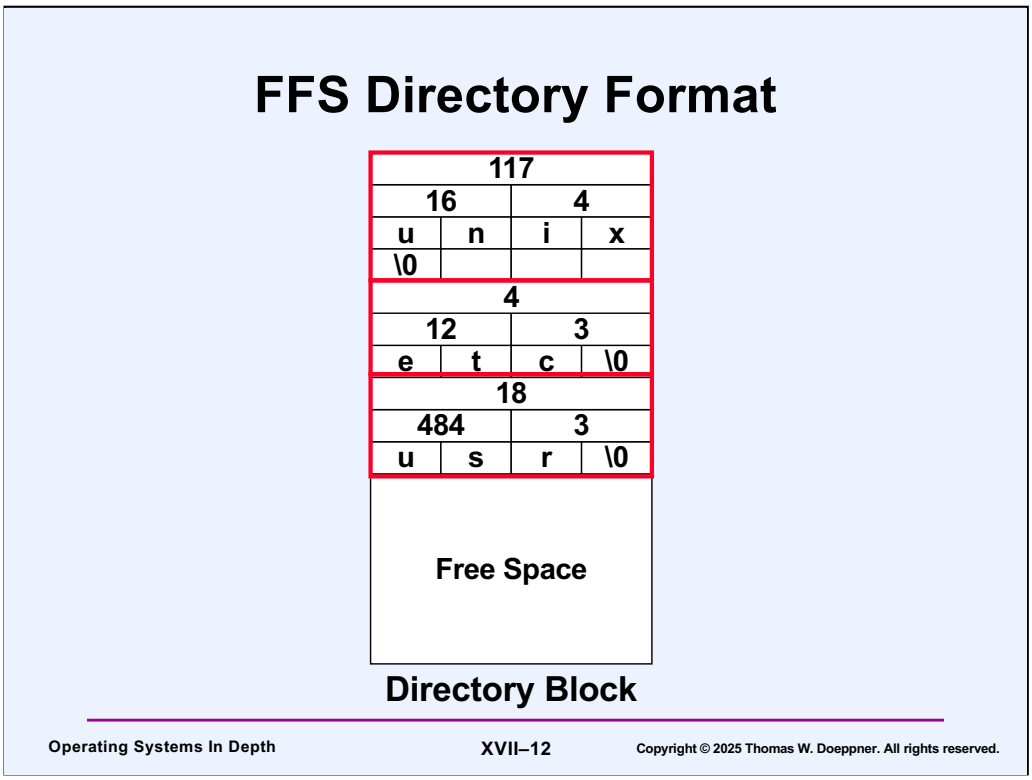
Deleting an S5FS directory entry entails simply zeroing out the entry. No compaction of entries is done, even if a great number are deleted. Why not?

a) The implementer was lazy

b) It's really difficult to do right

c) Deletion is so rare that it's not worth worrying about

d) It never occurred to them to do it



FFS allows component names of directory path names to be up to 255 characters long, thereby necessitating a variable-length field for components. Directories are composed of 512-byte blocks and entries must not cross block boundaries. This design adds a degree of atomicity to directory updates. It should take exactly one disk write to update a directory entry (512 bytes was chosen as the smallest conceivable disk sector size). If two disk writes are necessary to modify a directory entry, then clearly the disk will crash between the two!

Like the S5FS directory entry, the FFS directory entry contains the inode number and the component name. Since the component name is of variable length, there is also a string length field (the component name includes a null byte at the end; the string length does not include the null byte). In addition to the string length, there is also a record length, which is the length of the entire entry (and must be a multiple of four to ensure that each entry starts on a four-byte boundary). The purpose of the record length field is to represent free space within a directory block. Any free space is considered a part of the entry that precedes it, and thus a record length longer than necessary indicates that free space follows. If a directory entry is free, then its record length is added to that of the preceding entry. However, if the first entry in a directory block is free, then this free space is represented by setting the inode number to zero and leaving the record length as is.

Compressing directories is really difficult. Free space within a directory is made available for representing new entries, but is not returned to the file system. However, if there is free space at the end of the directory, the directory may be truncated to a directory-block boundary.

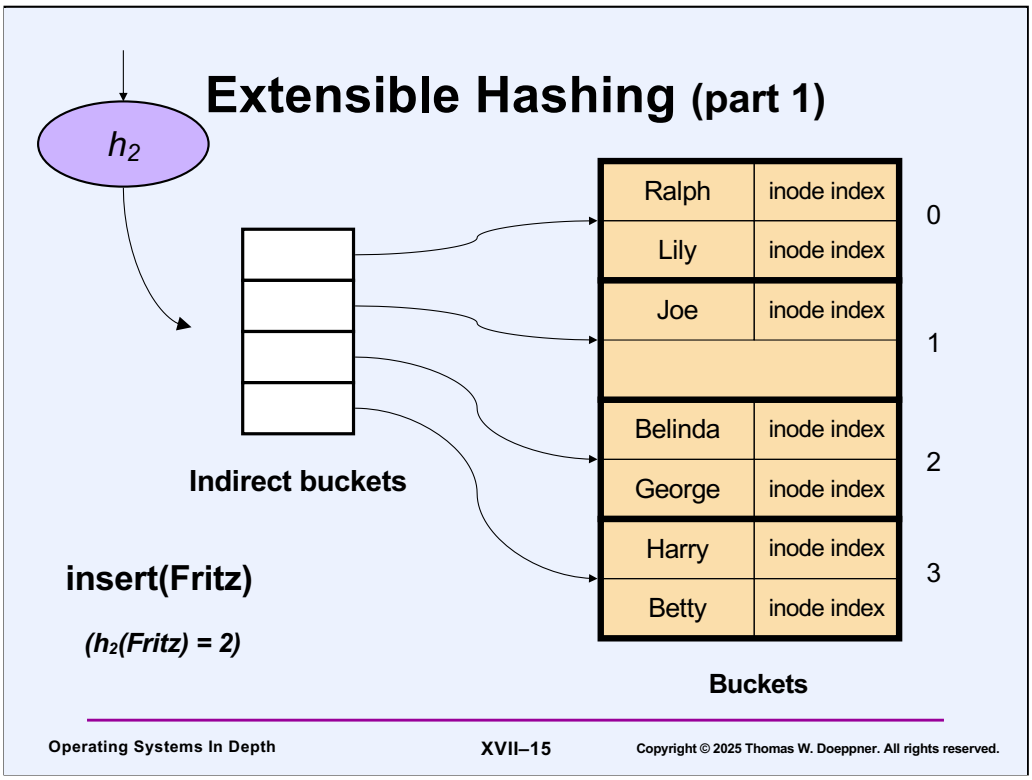
FFS directories were searched sequentially. The results of recent searches were cached, with cache lookup done via hashing.

Hashed Directories

- Implement directory using a hash table
  - h = hash function
  - h(name) yields either directory entry or not present
  - how large is the hash table?
    - can it increase in size if directory grows?
  - use *extensible hashing*
    - as directory grows, hash table grows and new hash functions are employed

Extensible Hash Functions

- Sequence of functions h_0, h_1, h_2, ...
  - h_i maps strings into 2^i indirect buckets
  - i low-order bits of h_{i+1}(component) are the same as the i bits of h_i(component)
    - h_2("Adam") is 01
    - h_3("Adam") is 001 or 101

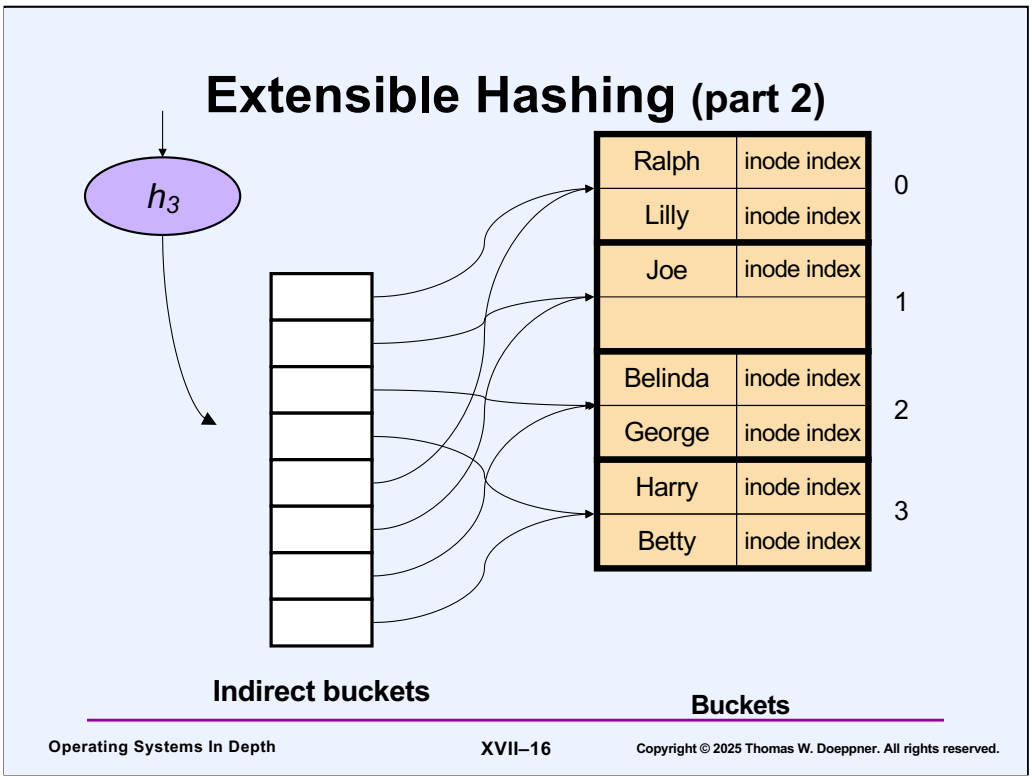


See the text, Section 6.3.1.1, which describes the use of extensible hashing for directories. h_i is a hash function that maps its arguments into 2^i buckets. In this example, each bucket is of size 2.

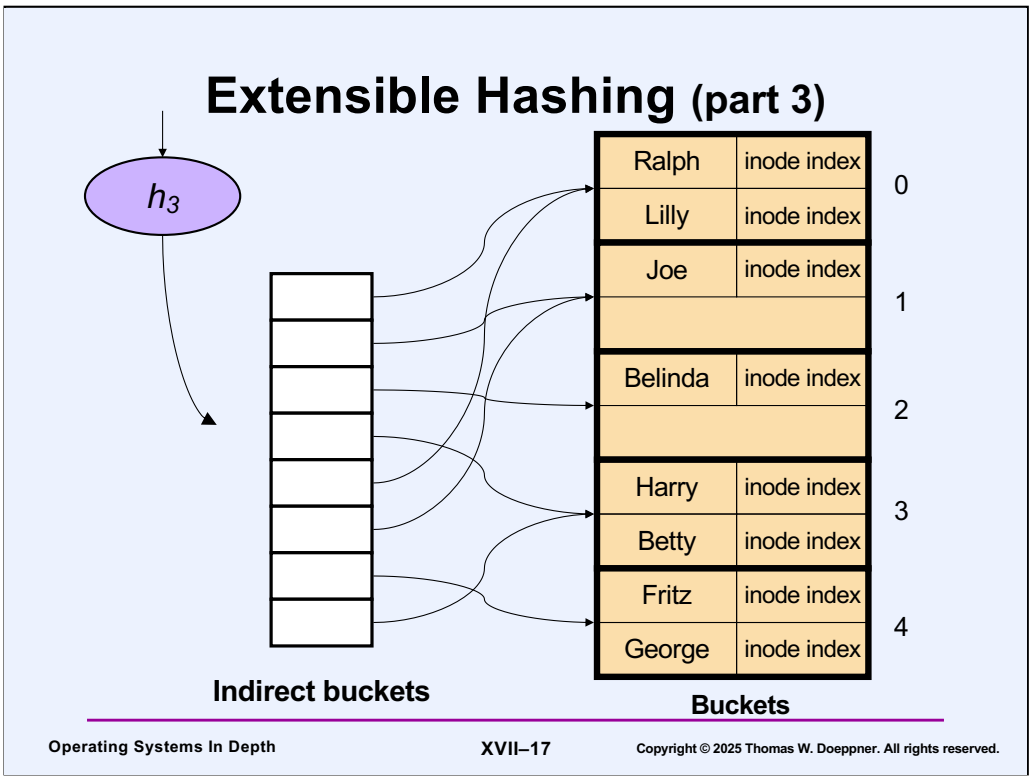
Here we have four buckets and hence are using h_2, which computes indexes into the array of indirect buckets, which, in turn, lead to the appropriate bucket. Each of our buckets holds two items. We are about to add an entry for Fritz. However, h_2(Fritz) is 2 and the bucket it leads to is already full.

The indirect buckets and buckets are maintained on disk and brought into memory when needed, as are any other files. Thus a directory is represented effectively as two files: the indirect buckets and the buckets.

Sun's ZFS file system uses extensible hashing.



Here we've switched to h_3, which maps names into eight buckets. However, rather than double the number of buckets and rehash the contents of all the old buckets, we take advantage of the array of indirect buckets. We double their number, but, initially, the new ones point to the same buckets as the old ones do: indirect buckets 0 and 4 point to bucket 0, indirect buckets 1 and 5 point to bucket 1, and so forth.



For the sake of Fritz we add a new (direct) bucket which we label 4. We rehash Fritz and the prior contents of bucket 2 under h_3, with the result that Fritz and George end up in the bucket referred to by indirect bucket 6, while Belinda stays in the bucket referred to by indirect bucket 2. Thus, we set indirect bucket 6 to refer to the new bucket 4. If, for example, we add another name that would go into bucket 0, we'd have to add another bucket to hold it and rehash the current contents of bucket 0.

Quiz 3

The number of disk reads required to search for an entry in an S5FS directory of size n is O(n).

How many disk reads are required to search for an entry in a directory implemented using extensible hashing? Assume the directory is contained within a file.

a) O(1)

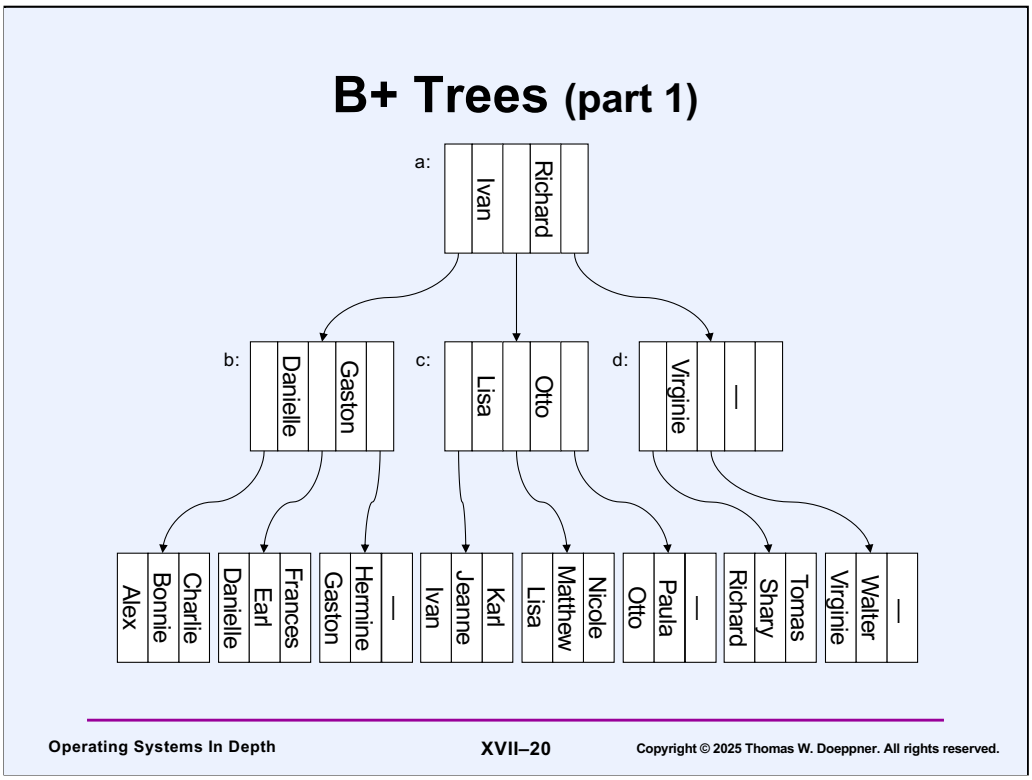
b) O(log n)

c) O(n)

d) O(n·log n)

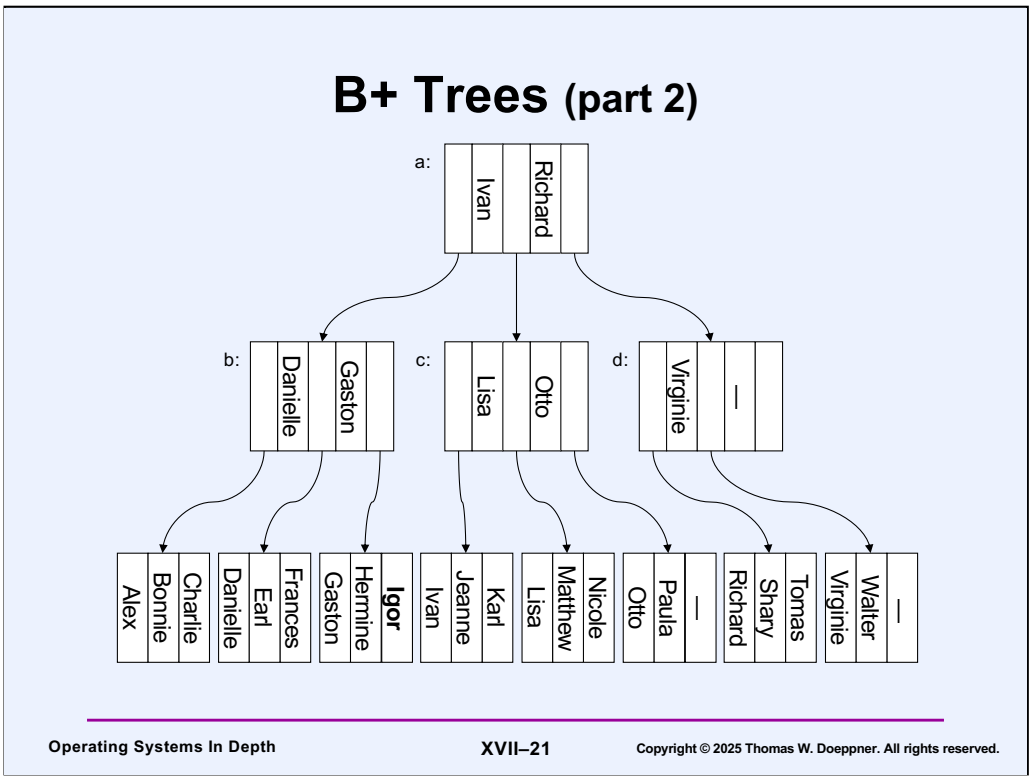
Balanced Search Trees

- Implement directory as a balanced search tree
  - easily grown to accommodate larger directories, but still log n searches
  - a common operation is to list the directory in alphabetical order
    - should be cheap
  - use B+ trees

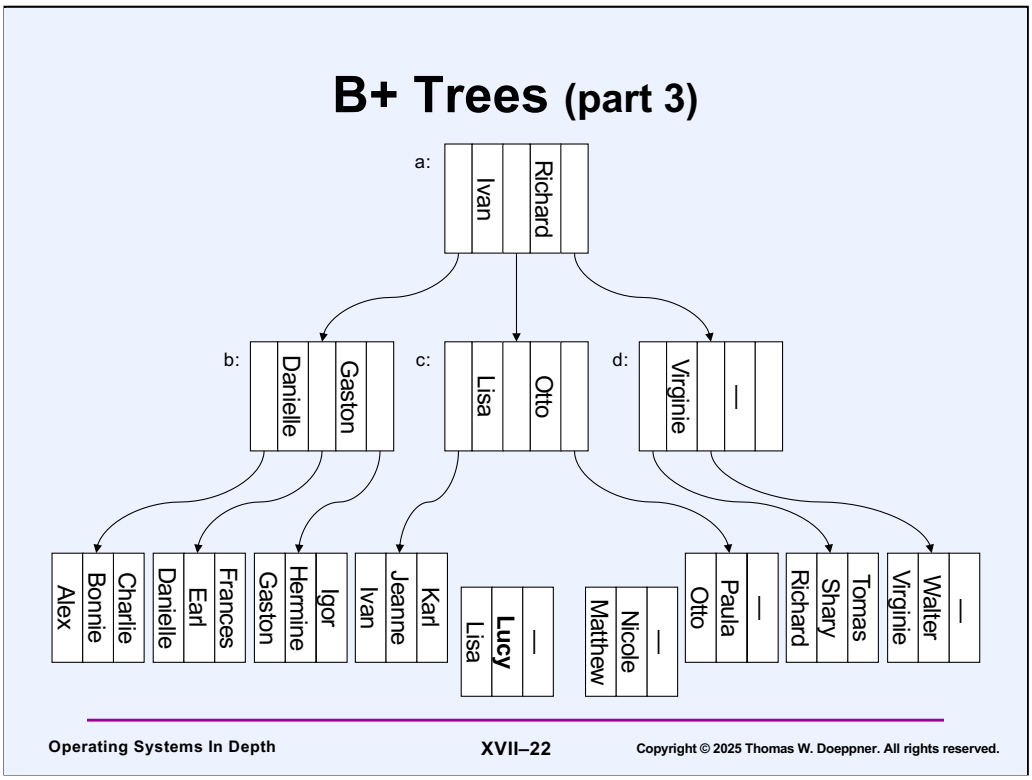


A B+ tree representing a directory. To keep the figure relatively simple, all entries occupy the same amount of space. The non-leaf nodes contain two or three pointers to other nodes, as well as two or three separator values. Every value in the subtree pointed to by a pointer is less than the value of the separator that follows the pointer. The separator appears as the smallest entry in the tree pointed to by the pointer following it. Leaf nodes contain two or three values.

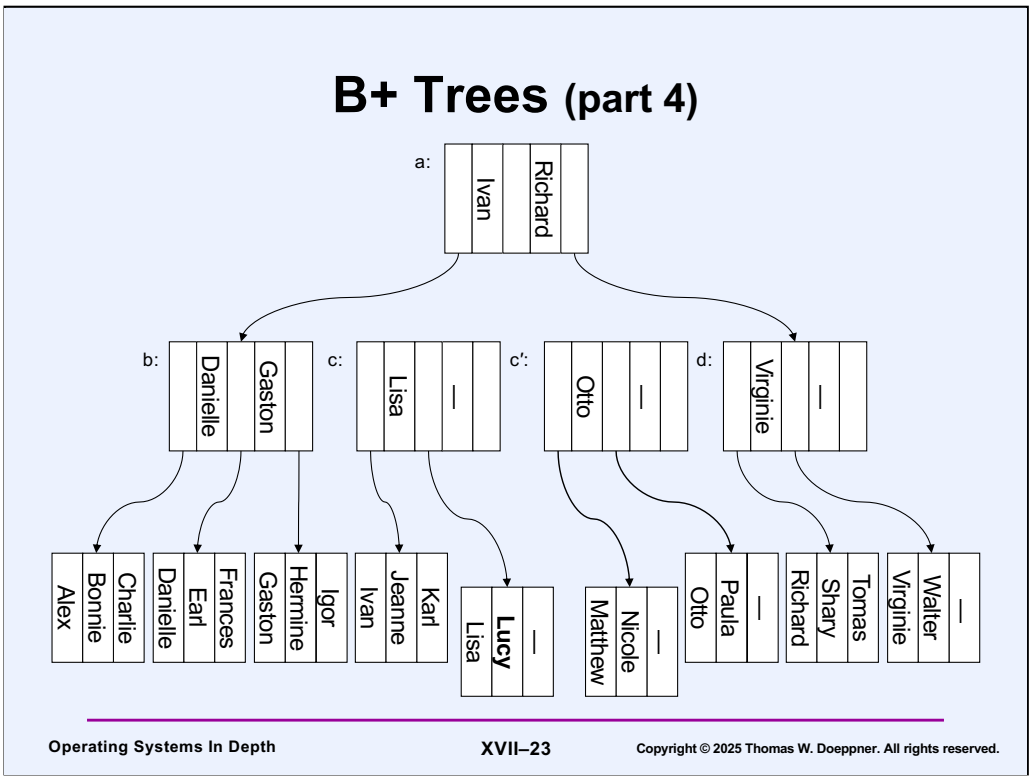
Microsoft's NTFS uses B+ trees.



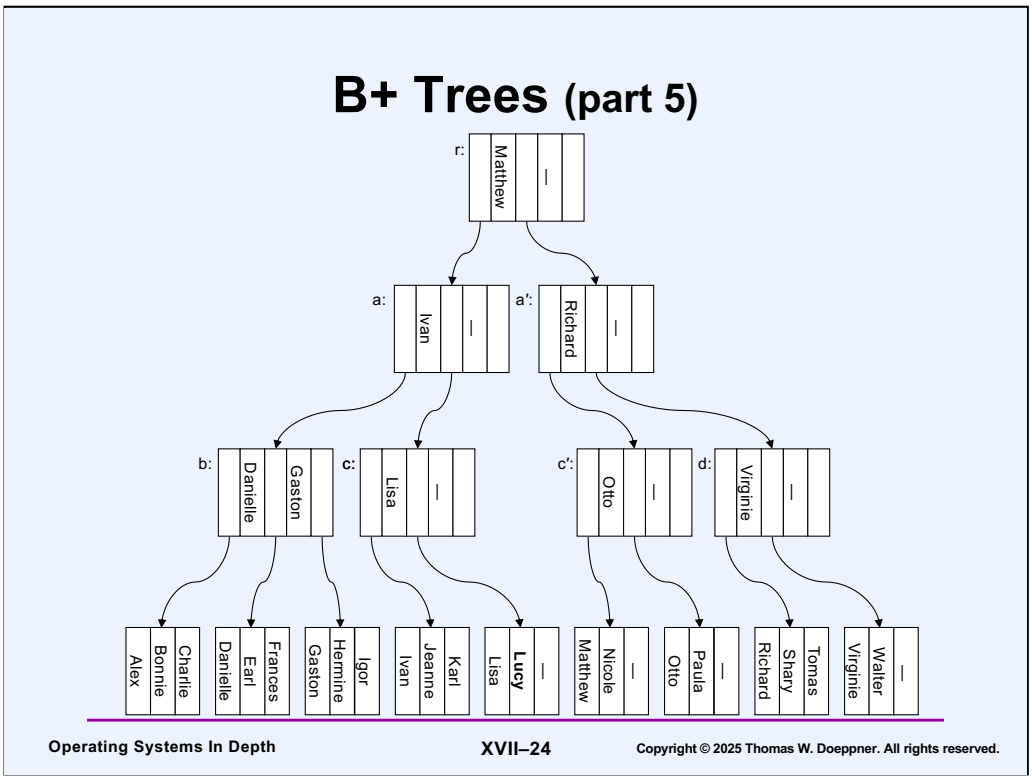
Here we've added Igor.



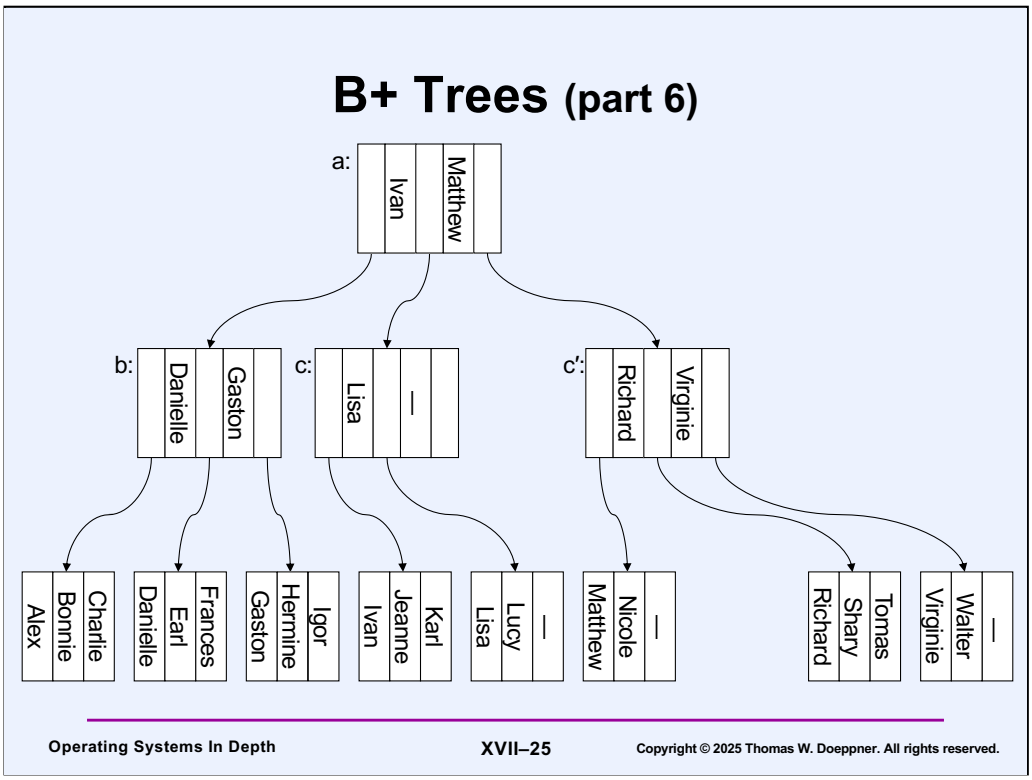
Here we try to add Lucy, but the node it should go into is full, so we have to create a new node and split the contents of the original node into it and the new node.



To fit the new node into the tree, we create a new non-leaf node, c', and move half of node c into it.



We have to distribute the original root over two nodes and create a new root to refer to them.



The directory after removing Paula and Otto.

Quiz 4

For what size directories does the B+ tree approach use fewer disk accesses for a lookup than does the extensible hashing approach? Assume each B+ tree node is in a separate block.

a) really small directories

b) really large directories

c) it always uses fewer disk accesses

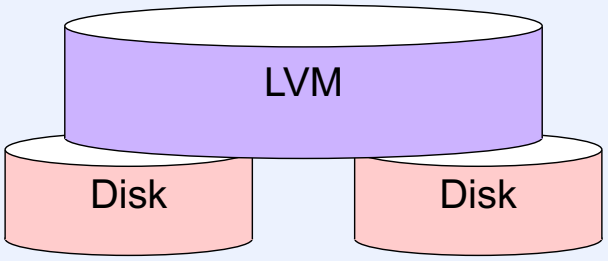
d) it never uses fewer disk accesses

You may assume that the pointer from one B+ tree node to another is an absolute disk address (and thus not an offset relative to the beginning of the file).

Benefits of Multiple Disks

- They hold more data than one disk does
- Data can be stored redundantly so that if one disk fails, they can be found on another
- Data can be spread across multiple drives, allowing parallel access

Logical Volume Manager



- **Spanning**
  - **two real disks appear to file system as one large disk**
- **Mirroring**
  - **file system writes redundantly to both disks**
  - **reads from one**

A logical volume manager appears to the rest of the OS kernel as if it were a disk driver. Thus no other kernel changes are required to support an LVM.

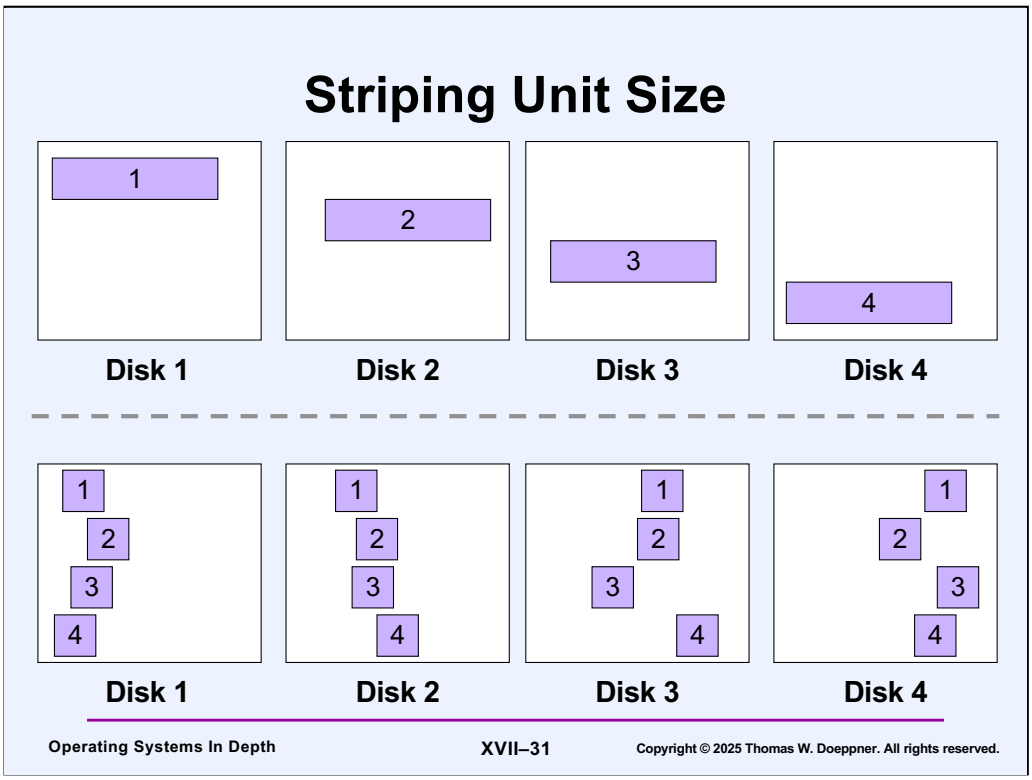
Striping

	Disk 1	Disk 2	Disk 3	Disk 4
Stripe 1	Unit 1	Unit 2	Unit 3	Unit 4
Stripe 2	Unit 5	Unit 6	Unit 7	Unit 8
Stripe 3	Unit 9	Unit 10	Unit 11	Unit 12
Stripe 4	Unit 13	Unit 14	Unit 15	Unit 16
Stripe 5	Unit 17	Unit 18	Unit 19	Unit 20

Disk striping: each stripe is written across all disks at once. The size of a "unit" may be anywhere from a bit to multiple tracks. If it's less than a sector in size, then multiple stripes are transferred at once so that the amount of data per transfer per disk is an integer multiple of a sector.

Concurrency Factor

- How many requests are available to be executed at once?
  - one request in queue at a time
    - concurrency factor = 1
    - e.g., one single-threaded application placing one request at a time
  - many requests in queue
    - concurrency factor > 1
    - e.g., multiple threads placing file-system requests



If we have only one waiting disk request at a time (a **concurrency factor** of 1), then a smaller striping unit is typically better than a larger one, since we can spread one request across lots of disks. But with more than one waiting disk request (a larger **concurrency factor**), it may make sense to have larger striping units, as the slide illustrates.

The top half of the slide shows four disks with a four-sector striping unit. Suppose we have concurrent requests for the four data areas shown, each four sectors in length. The four requests can be handled in roughly the time it takes to handle one, since the positioning for each of the requests can be done simultaneously as can the data transfer.

The bottom half of the slide shows four disks with a one-sector striping unit. We have the four concurrent requests for the same data as before, but in this case each item is spread across all four disks, one sector per disk. Handling each request requires first positioning the heads on all four disks for the first, then positioning the heads on all four disks for the second, and so forth. Thus the total positioning delays are four times that of the top half of the figure, which has a larger striping unit.

Quiz 5

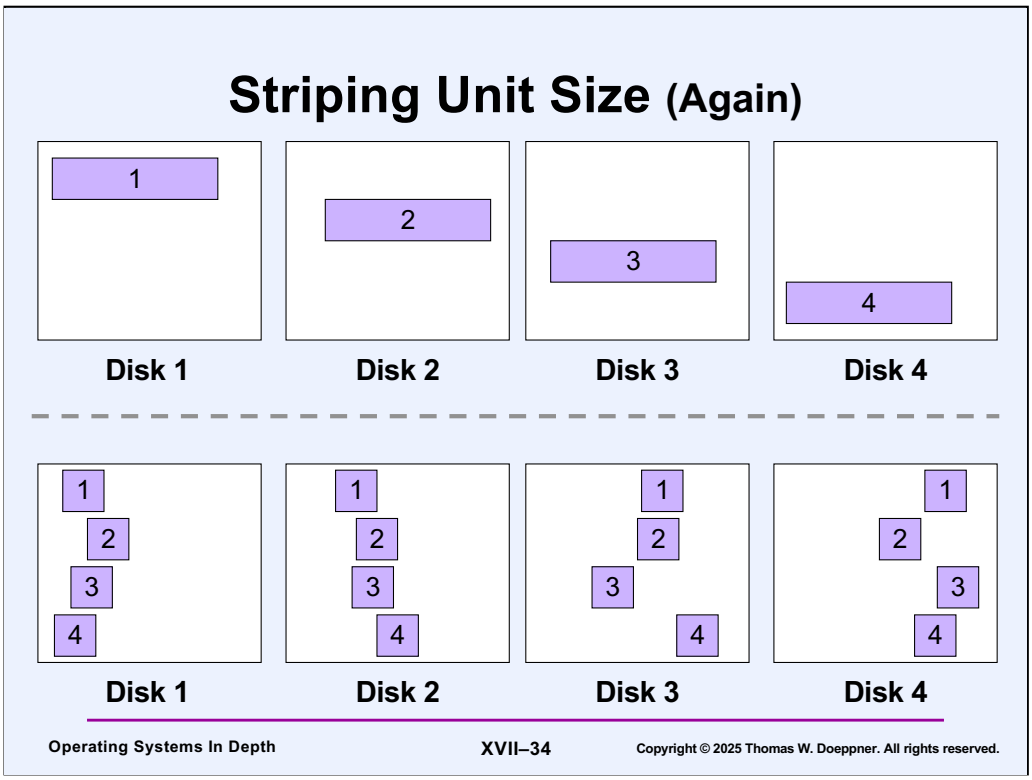
In the previous slide, suppose we have four threads concurrently reading from the disks. The first is reading all the sectors labeled one, the second is reading all the sectors labeled two, the third three, and the fourth four. With which layout would they complete all the transfers the quickest?

a) top
b) bottom
c) roughly equal

Quiz 6

Now suppose we have one thread that first reads the sectors labeled one, then those labeled two, then three, then four. With which layout would it complete the transfers the quickest?

a) top
b) bottom
c) roughly equal



Getting back to the case of just one waiting disk request (a concurrency factor of 1), suppose we have one thread that's first reading the data labelled one, then two, etc. In the layout of the top part of the slide, it would, for each disk in turn, wait for positioning delays and then transfer four sectors. Thus the total time required would be, roughly, four times the typical seek and rotational delays plus the transfer time for 16 total sectors.

For the layout of the bottom half of the slide, The thread would read the areas labelled one from all four disks, then the areas labelled two from all four disks, etc. Since the thread is reading the areas sequentially (first all of one, then all of two, etc.), it would, for each area, have to wait the maximum of the seek and rotational delays for the four disks, then the time to transfer one sector (since the four sectors for each area are transferred in parallel). This results in a total time of four positioning delays, plus the time required to transfer four sectors (totaling 16 sectors since four are transferred at a time). This will generally be considerably faster than for the non-striped case, under the assumption that the positioning delays don't completely dominate the transfer times.

Striping: The Effective Disk

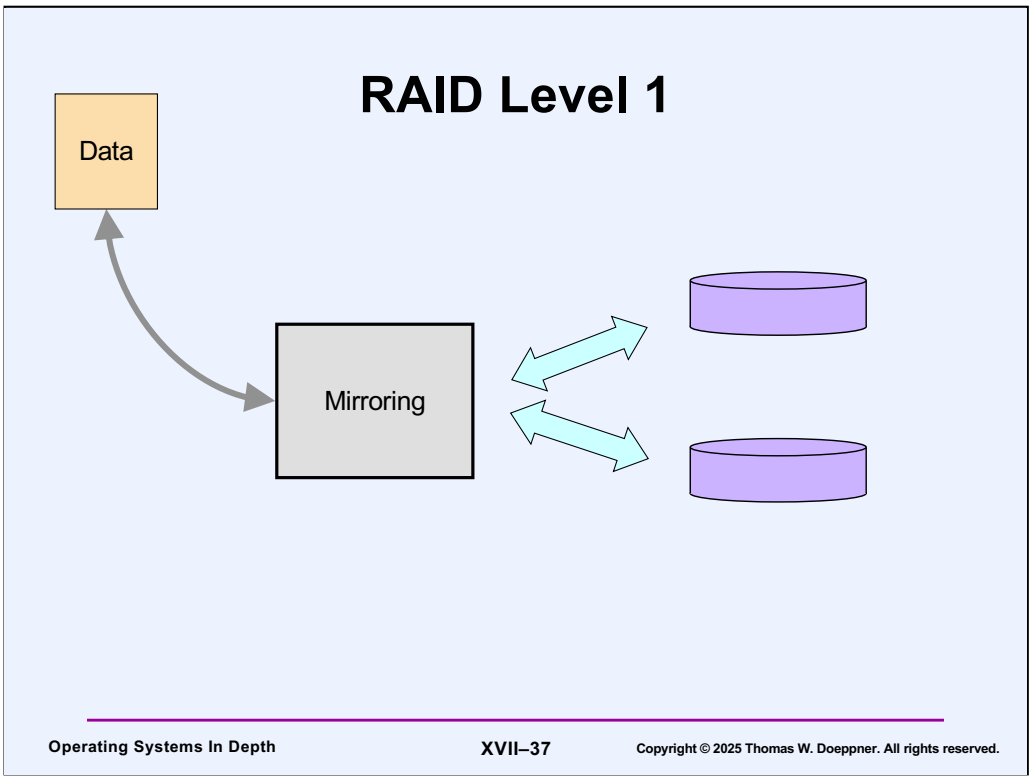
- Improved effective transfer speed
  - parallelism
- No improvement in seek and rotational delays
  - sometimes worse
- A system depending on N disks is much more likely to fail than one depending on one disk
  - if probability of one disk's failing is *f*
  - probability of N-disk system's failing is $(1-(1-f)^N)$
  - (assumes failures are IID, which is probably wrong …)

IID = independent and identically distributed.

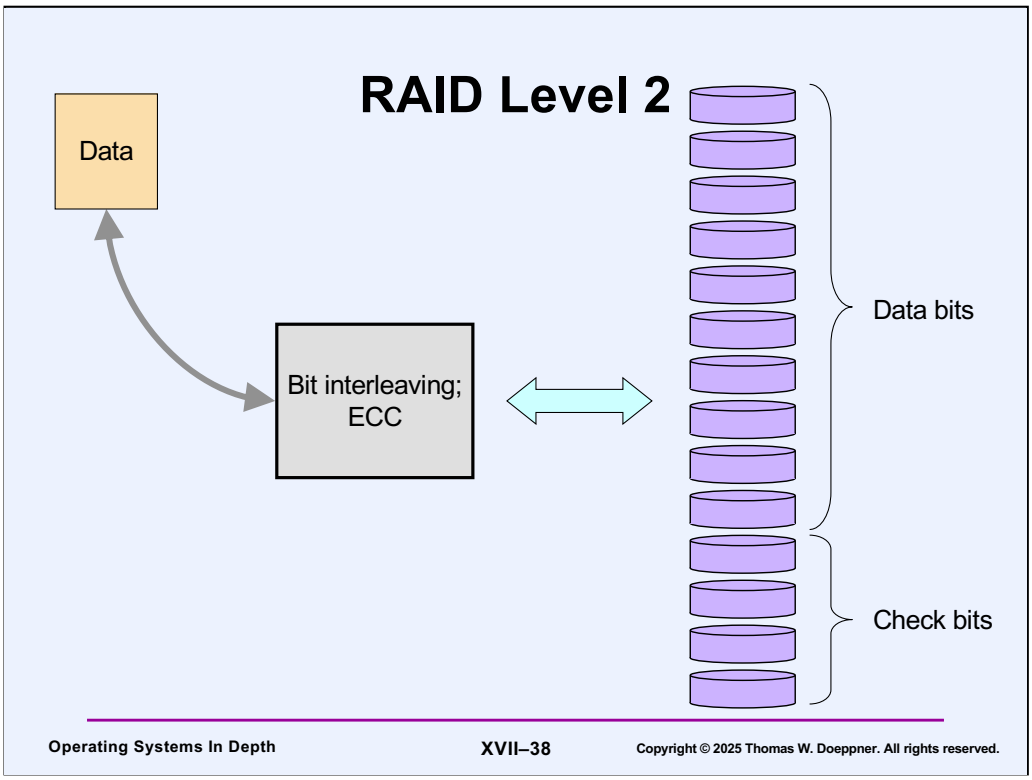
If the probability of one disk's failing is f, the probability of its not failing is 1-f. If we have two disks, the probability of their both not failing is $(1-f)^2$. Thus the probability that none of N disks fail is $(1-f)^N$. And thus the probability that at least one of them fails is $1-(1-f)^N$.

RAID to the Rescue

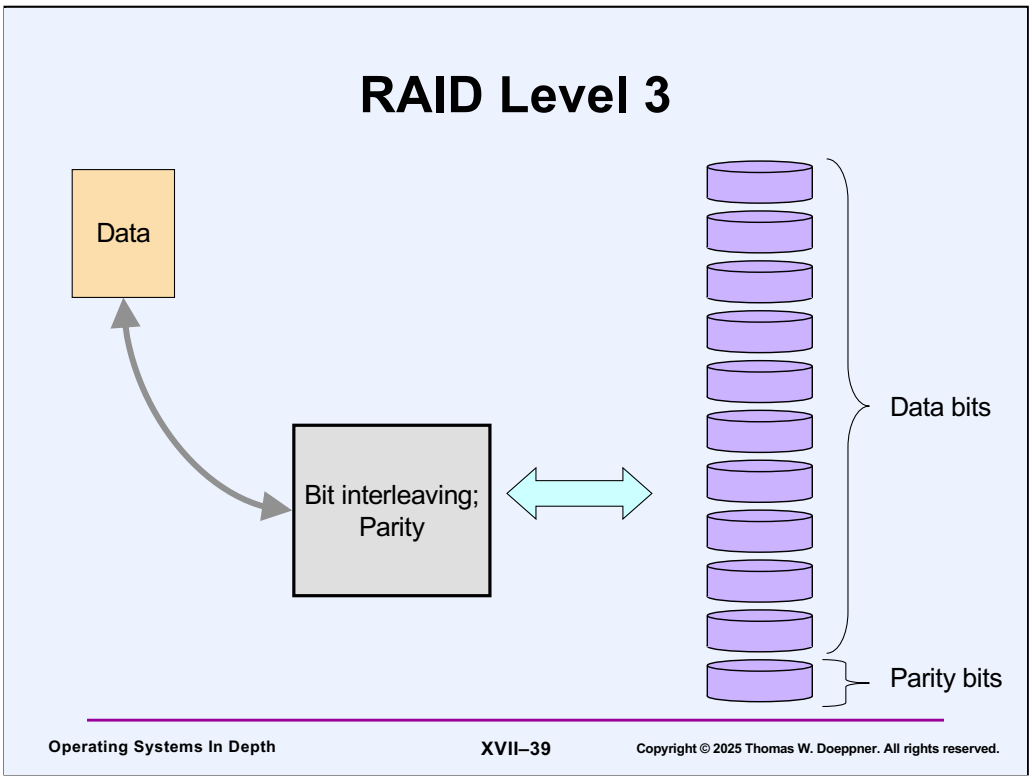
- Redundant Array of Inexpensive Disks
  - (as opposed to Single Large Expensive Disk: SLED)
  - combine striping with mirroring
  - 5 different variations originally defined
    - RAID level 1 through RAID level 5
      - RAID level 0: pure striping
        - numbering extended later
      - RAID level 1: pure mirroring



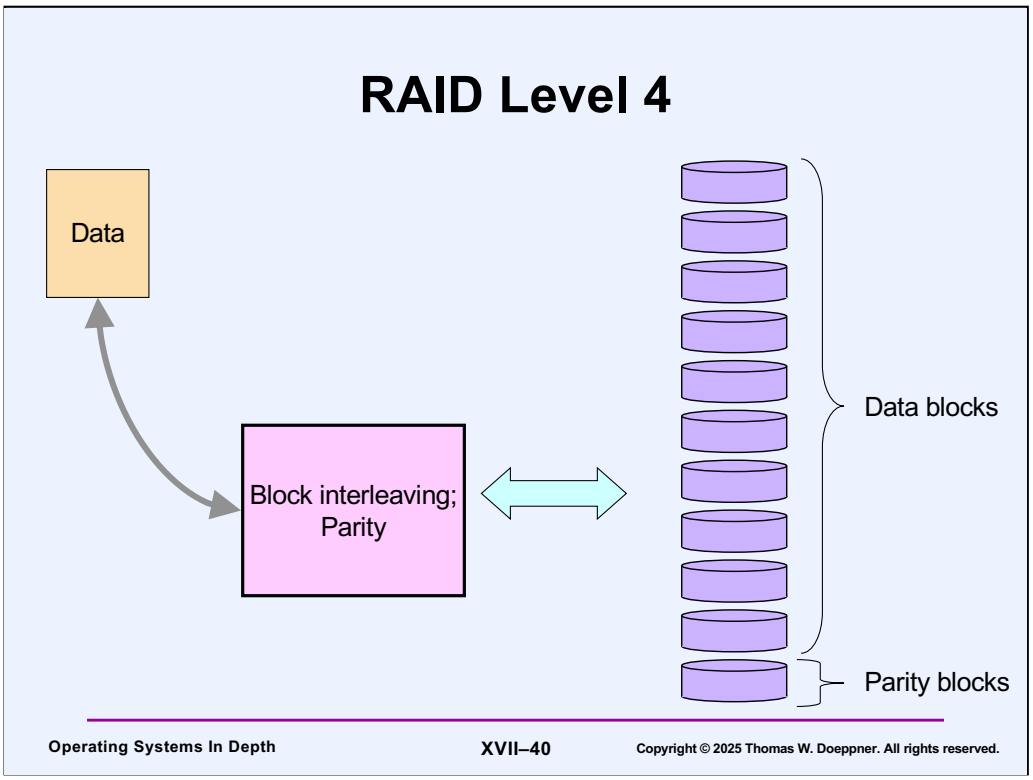
RAID level 1: mirroring.



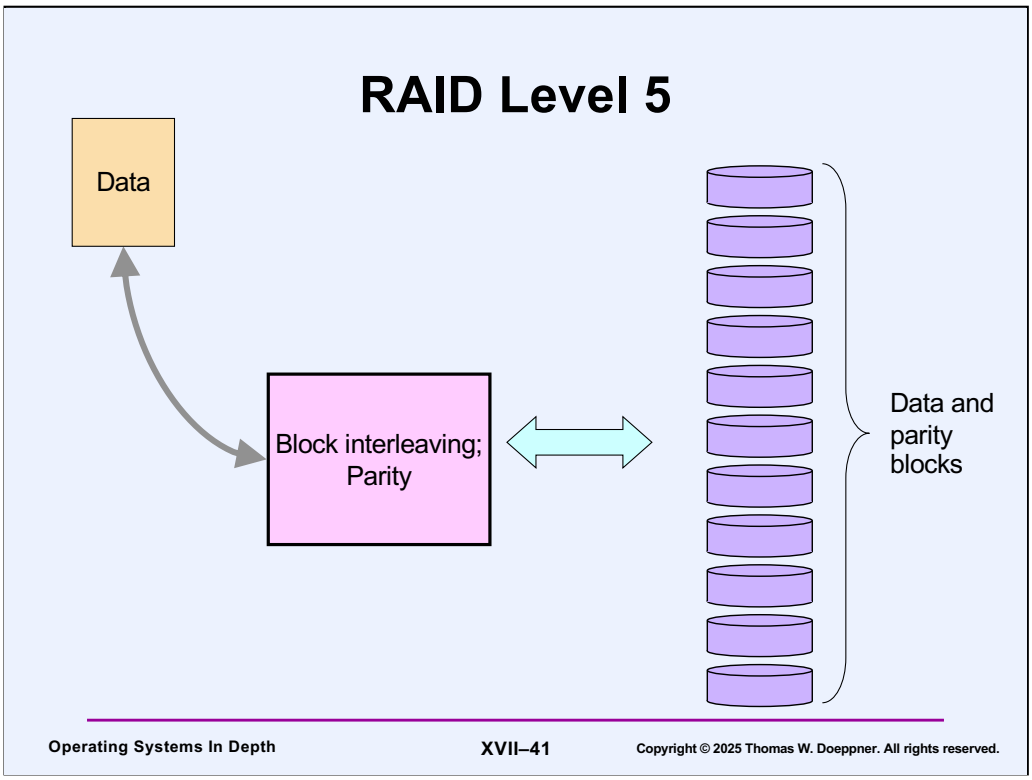
RAID level 2: bit interleaving with an error-correcting code. An error-correcting code can detect and fix n-bit errors, by supplying redundant information.



RAID level 3: bit interleaving with parity bits. A parity bit can detect a single-bit error, but not fix it (it wouldn't be known which bit is wrong). For example, the parity bit might be the exclusive or of the other bits.



RAID level 4: block interleaving with parity blocks. Note that an update to any of the disks holding data blocks requires an update to the disk holding the parity blocks, causing it to be a bottleneck.



RAID level 5: block interleaving with parity blocks. Rather than dedicating one disk to hold all the parity blocks, the parity blocks are distributed among all the disks. For stripe 1, the parity block might be on disk 1; for stripe 2 it would be on disk 2, and so forth. If we have eleven disks, then for stripe 12 the parity block would be back on disk 1.

RAID 4 vs. RAID 5

- Lots of small writes
  - RAID 5 is best
- Mostly large writes
  - multiples of stripes
  - either is fine