Operating System Concepts

Lecture 32: File System Reliability & Examples

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Free space management

- disk space is limited so file system has to allocate the space released after deleting files to new files
    - must be able to find free space quickly and release disk space quickly

Free space management

- disk space is limited so file system has to allocate the space released after deleting files to new files
  - must be able to find free space quickly and release disk space quickly
- to keep track of free disk blocks, file system maintains a free-space list
  - when blocks are allocated to a file, they are removed from this list
  - when a file is deleted, freed blocks are added to this list

Free space management

- disk space is limited so file system has to allocate the space released after deleting files to new files
  - must be able to find free space quickly and release disk space quickly
- to keep track of free disk blocks, file system maintains a free-space list
  - when blocks are allocated to a file, they are removed from this list
  - when a file is deleted, freed blocks are added to this list
- the free-space list may not be implemented as a list!

Free space management

- disk space is limited so file system has to allocate the space released after deleting files to new files
  - must be able to find free space quickly and release disk space quickly
- to keep track of free disk blocks, file system maintains a free-space list
  - when blocks are allocated to a file, they are removed from this list
  - when a file is deleted, freed blocks are added to this list
- the free-space list may not be implemented as a list!
- need to maintain a free-inode list too

Bit vector

- each disk block is represented by one bit
  - the block is free if the corresponding bit is set

00001001011 …

Bit vector

- each disk block is represented by one bit
  - the block is free if the corresponding bit is set

    00001001011 …

- marking a block as free is simple because the block number is the index into the bitmap

Bit vector

- each disk block is represented by one bit
  - the block is free if the corresponding bit is set

    000010010011 …

- marking a block as free is simple because the block number is the index into the bitmap
- to find a free block, the file system sequentially checks each word in the bitmap with zero
  - the first non-zero word is scanned for the first 1 bit then
  - expensive to find a free block if most blocks are used!

Bit vector

- each disk block is represented by one bit
  - the block is free if the corresponding bit is set

00001001011 …

- marking a block as free is simple because the block number is the index into the bitmap
- to find a free block, the file system sequentially checks each word in the bitmap with zero
  - the first non-zero word is scanned for the first 1 bit then
  - expensive to find a free block if most blocks are used!
- problem? the bitmap might be too big to keep in memory

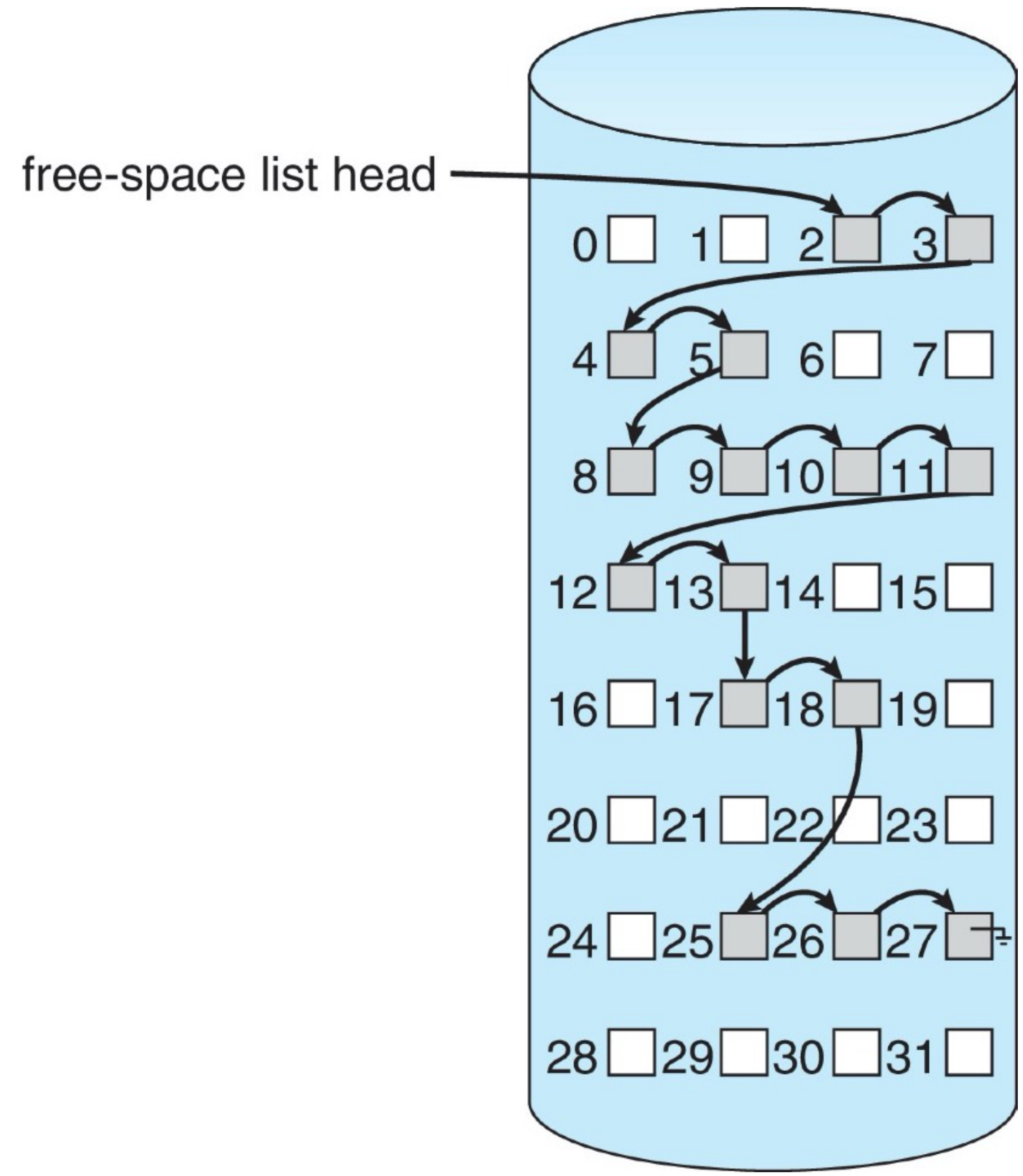
block size = 512 bytes

disk size = 2GB

n = $2^{31}/2^{9} = 2^{22}$ blocks -> 2^{22} bits (**512KB!**)

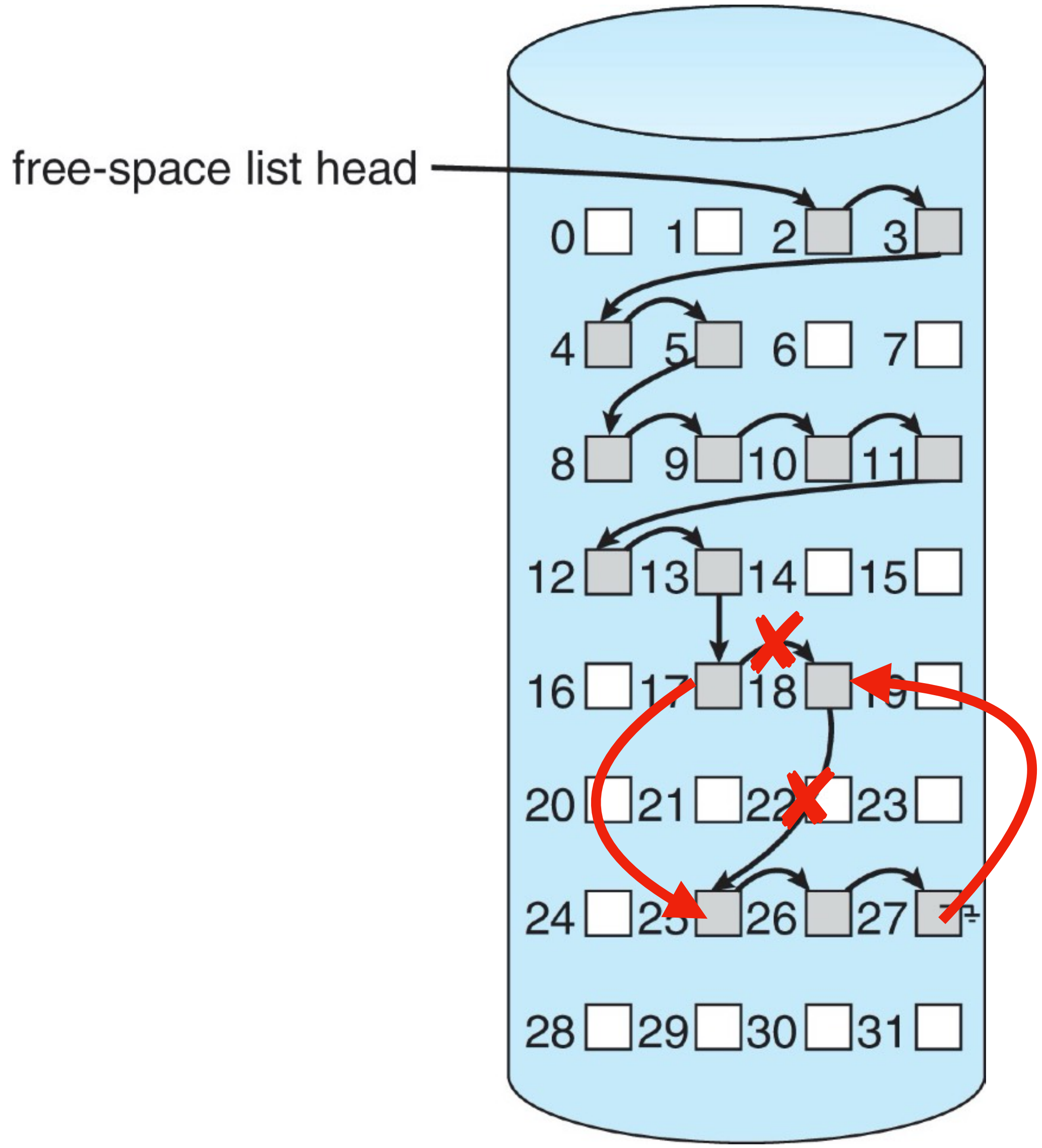
Linked free-space list

- link together all free disk blocks; each block contains a pointer to the next free block (saving space)
- the head of the list is cached in memory
- no need to traverse the entire list (if the number of free blocks is recorded)



Linked free-space list

- link together all free disk blocks; each block contains a pointer to the next free block (saving space)
- the head of the list is cached in memory
- no need to traverse the entire list (if the number of free blocks is recorded)
- problems?
  - linked list gets disorganized over time
  - cannot get contiguous space easily



Extensions

- grouping
  - modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this block)

Extensions

- grouping
  - modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this block)
- counting
  - space is contiguously used and freed frequently so
    - keep the address of the first free block and the number of following free blocks
    - free-space list has entries containing addresses and counts

Summary

- many of the concerns, tradeoffs, and design decisions discussed for file system are similar to those of virtual memory (and disk as we will see later)
  - contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow
  - indexed allocation is very similar to page tables
    - a mapping from logical file blocks to physical disk blocks
  - free space can be managed using a bitmap or a linked list

Today's class

- How to make sure that data and metadata previously stored can be retrieved regardless of software crashes and hardware failures?
  - data consistency
  - metadata consistency
- How real file systems are implemented?
  - FFS
  - NTFS

File System Reliability

File system reliability problem

- a single file system operation can involve updates to multiple physical disk blocks (inode, data blocks, bitmap, etc.)
  - to move a file between directories, it has to be deleted from the old directory and added to the new directory
  - to create a new file, file system must allocate space on disk for file's data, write inode to disk, add the file number and name to the directory containing the file

File system reliability problem

- a single file system operation can involve updates to multiple physical disk blocks (inode, data blocks, bitmap, etc.)
  - to move a file between directories, it has to be deleted from the old directory and added to the new directory
  - to create a new file, file system must allocate space on disk for file's data, write inode to disk, add the file number and name to the directory containing the file
- problem 1: physical disk blocks are updated one at a time
  - crash may occur between multiple updates causing inconsistencies among file-system data structures

File system reliability problem

- a single file system operation can involve updates to multiple physical disk blocks (inode, data blocks, bitmap, etc.)
  - to move a file between directories, it has to be deleted from the old directory and added to the new directory
  - to create a new file, file system must allocate space on disk for file's data, write inode to disk, add the file number and name to the directory containing the file
- problem 1: physical disk blocks are updated one at a time
  - crash may occur between multiple updates causing inconsistencies among file-system data structures
- problem 2: **write-through** (i.e., to write modified data back to disk synchronously) is really slow, so changes are cached (written to disk when convenient)
  - crash may occur before disk I/O causing loss of data

File system reliability problem

- a single file system operation can involve updates to multiple physical disk blocks (inode, data blocks, bitmap, etc.)
  - to move a file between directories, it has to be deleted from the old directory and added to the new directory
  - to create a new file, file system must allocate space on disk for file’s data, write inode to disk, add the file number and name to the directory containing the file
- problem 1: physical disk blocks are updated one at a time
  - crash may occur between multiple updates causing inconsistencies among file-system data structures
- problem 2: **write-through** (i.e., to write modified data back to disk synchronously) is really slow, so changes are cached (written to disk when convenient)
  - crash may occur before disk I/O causing loss of data
- **Goal:** guarantee consistency regardless of when crash occurs

Data consistency

- UNIX file system uses the write-back strategy (i.e., delaying writing the modified data back to disk) with periodic forced writes to disk (e.g., every 30 seconds)
  - so other processes read data from cache instead of disk
  - potential for loss of 30 seconds worth of cached changes
  - user can use the `sync` system call to flush buffer cache to disk immediately

Data consistency

- UNIX file system uses the write-back strategy (i.e., delaying writing the modified data back to disk) with periodic forced writes to disk (e.g., every 30 seconds)
  - so other processes read data from cache instead of disk
  - potential for loss of 30 seconds worth of cached changes
  - user can use the `sync` system call to flush buffer cache to disk immediately
- how to write changes to a file onto disk?
  - naive approach:
    - delete old version; create new version
  - correct approach:
    - write new version in temp file; move old version to another temp file; move new version into real file; unlink old version
    - on a crash look at temp area; if there is any files out there notify user that there might be a problem

Metadata consistency

- if multiple updates to metadata is needed, perform them in a specific order which allows for an operation to be safely interrupted

Metadata consistency

- if multiple updates to metadata is needed, perform them in a specific order which allows for an operation to be safely interrupted
- recover after crash by checking if there were any in-progress operations and undo them
  - run a consistency checker (called `fsck` in UNIX) which scans the entire disk for internal consistency
  - this approach is used in FAT and UNIX file system

Metadata consistency

- if multiple updates to metadata is needed, perform them in a specific order which allows for an operation to be safely interrupted
- recover after crash by checking if there were any in-progress operations and undo them
  - run a consistency checker (called `fsck` in UNIX) which scans the entire disk for internal consistency
  - this approach is used in FAT and UNIX file system
- disadvantages:
  - post-crash recovery is time consuming (can take minutes or hours) and may not be possible
  - difficult to reduce every operation to a safely interruptible sequence of writes
  - difficult to achieve consistency when multiple operations occur concurrently

Example

- let’s say you want to extend a given file by one block; the necessary operations are as follows

  1. find a free block
  2. write block bitmap
  3. write inode with pointer to free block and new file size
  4. write data

Example

- let's say you want to extend a given file by one block; the necessary operations are as follows

  1. find a free block
  2. write block bitmap
  3. write inode with pointer to free block and new file size
  4. write data

- so in the case of a crash
  - if a bit is set in the bitmap but a pointer to this block is not added to any inode, writing inode must have been in progress when system crashed
  - if a bit is set in the bitmap and a pointer to this block is added to an inode but file data is not written to that block, writing data must have been in progress when system crashed

Transaction concept

- **Definition**: a group of operations that are atomic and durable
  - atomic: happen as a group or not at all
  - durable: future failures do not affect/corrupt previously committed transactions

Transaction concept

- **Definition**: a group of operations that are atomic and durable
  - atomic: happen as a group or not at all
  - durable: future failures do not affect/corrupt previously committed transactions
- **Basic idea**: do a set of metadata updates tentatively; if you don’t get to commit (due to crash or failure), then roll-back the operations as if the transaction never happened
  - commit makes transaction durable by writing a single sector on disk (we assume this happens atomically)

Implementation

- write operations to be performed **sequentially** and synchronously in an on-disk data structure (known as log, journal, or intention list)
  - ignore changes if crash occurs in the middle of these operations

Implementation

- write operations to be performed **sequentially** and synchronously in an on-disk data structure (known as log, journal, or intention list)
  - ignore changes if crash occurs in the middle of these operations
- commit transaction when all changes are on log

Implementation

- write operations to be performed **sequentially** and synchronously in an on-disk data structure (known as log, journal, or intention list)
  - ignore changes if crash occurs in the middle of these operations
- commit transaction when all changes are on log
- write the changes asynchronously to appropriate blocks of disk (random-access)
  - if crash occurs after commit, replay the log (from a pointer) to make sure updates get to disk

Implementation

- write operations to be performed **sequentially** and synchronously in an on-disk data structure (known as log, journal, or intention list)
  - ignore changes if crash occurs in the middle of these operations
- commit transaction when all changes are on log
- write the changes asynchronously to appropriate blocks of disk (random-access)
  - if crash occurs after commit, replay the log (from a pointer) to make sure updates get to disk
- **Note**: sequential I/O is faster than random I/O, and therefore, can be done synchronously

Log-based transaction-oriented file system

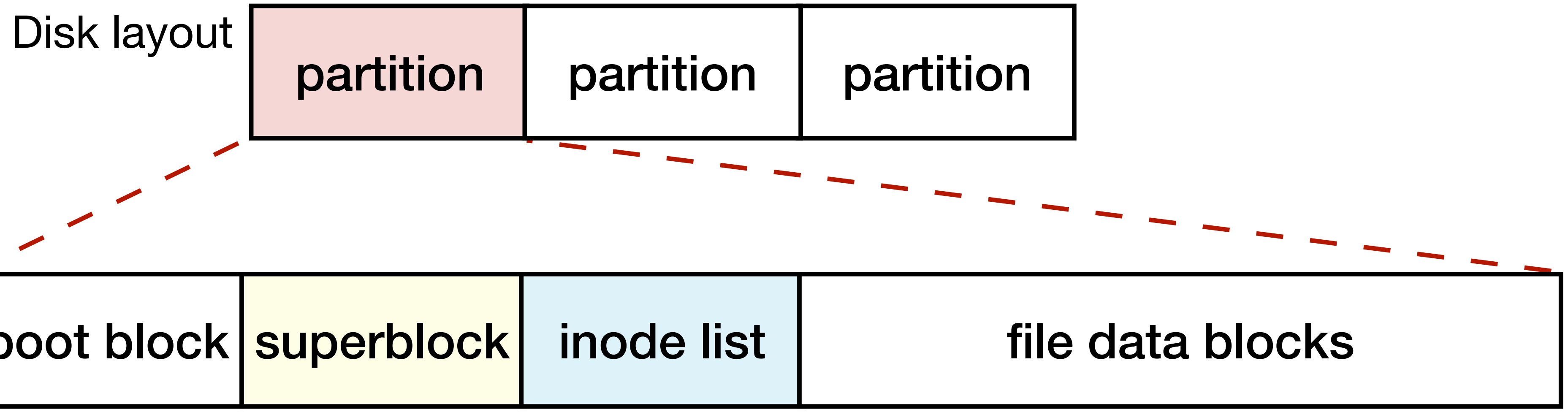
- almost all new file systems such as NTFS, ext3, and ext4 use **journaling** (or **write-ahead logging**)
    - write all changes in a transaction to log (update directory, allocate block, etc.) before sending any changes to disk
- journaling eliminates the need for doing post-crash file system consistency check (e.g., via `fsck`)

Log-based transaction-oriented file system

- almost all new file systems such as NTFS, ext3, and ext4 use **journaling** (or **write-ahead logging**)
  - write all changes in a transaction to log (update directory, allocate block, etc.) before sending any changes to disk
- journaling eliminates the need for doing post-crash file system consistency check (e.g., via `fsck`)
- advantage: a general solution to reliability problem
- disadvantage: data is written twice!

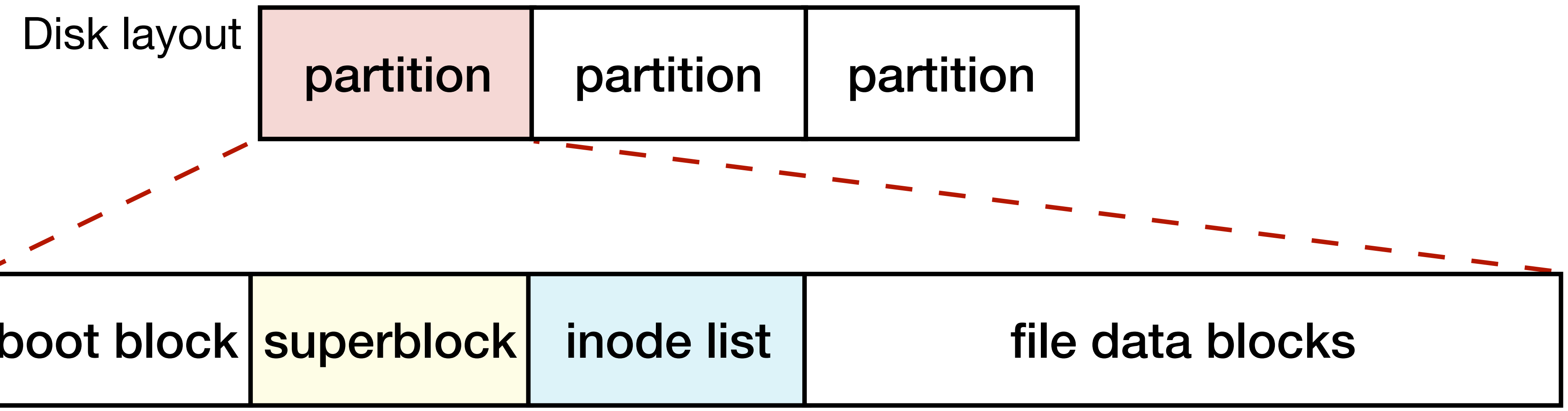
File System Examples

Traditional UNIX file system



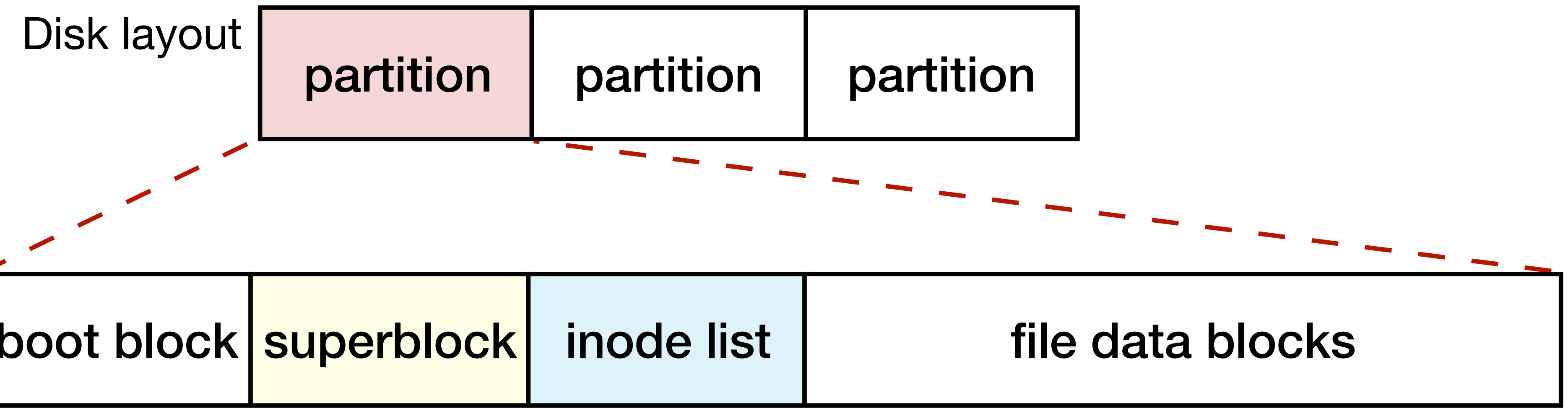
- boot block contains the bootstrap program that runs at boot time to load the OS from that partition

Traditional UNIX file system



- boot block contains the bootstrap program that runs at boot time to load the OS from that partition
- superblock defines a file system (there is only one per file system)
  - size of the file system and size of the inode list
  - list of free disk blocks (and index of the next free block)
  - list of free inodes (and index of the next free inode)
  - location of the inode of the root directory (inode #2)

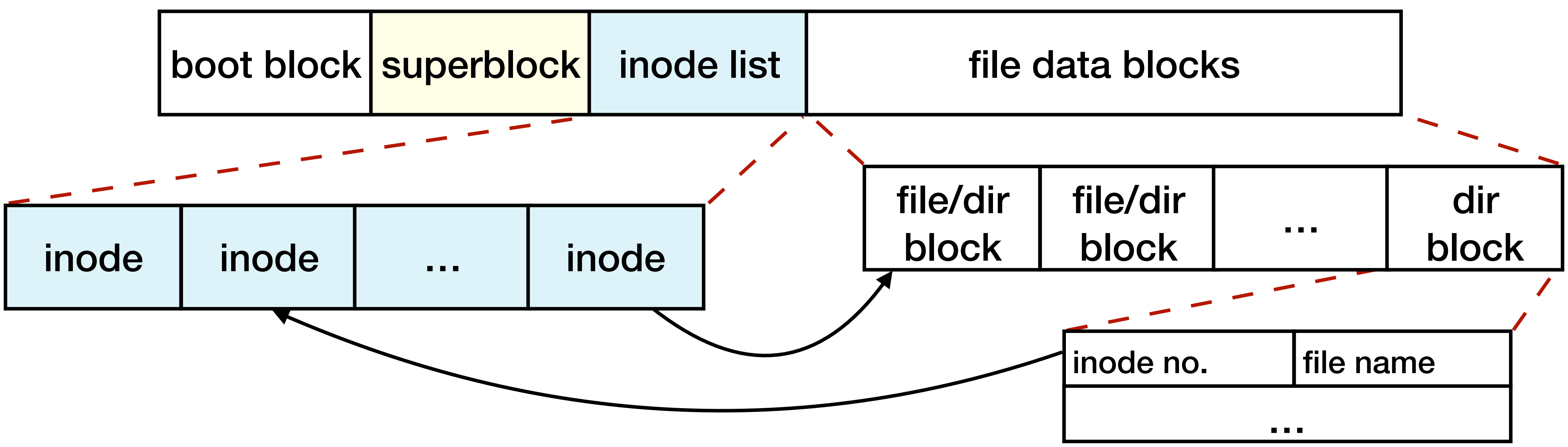
Traditional UNIX file system



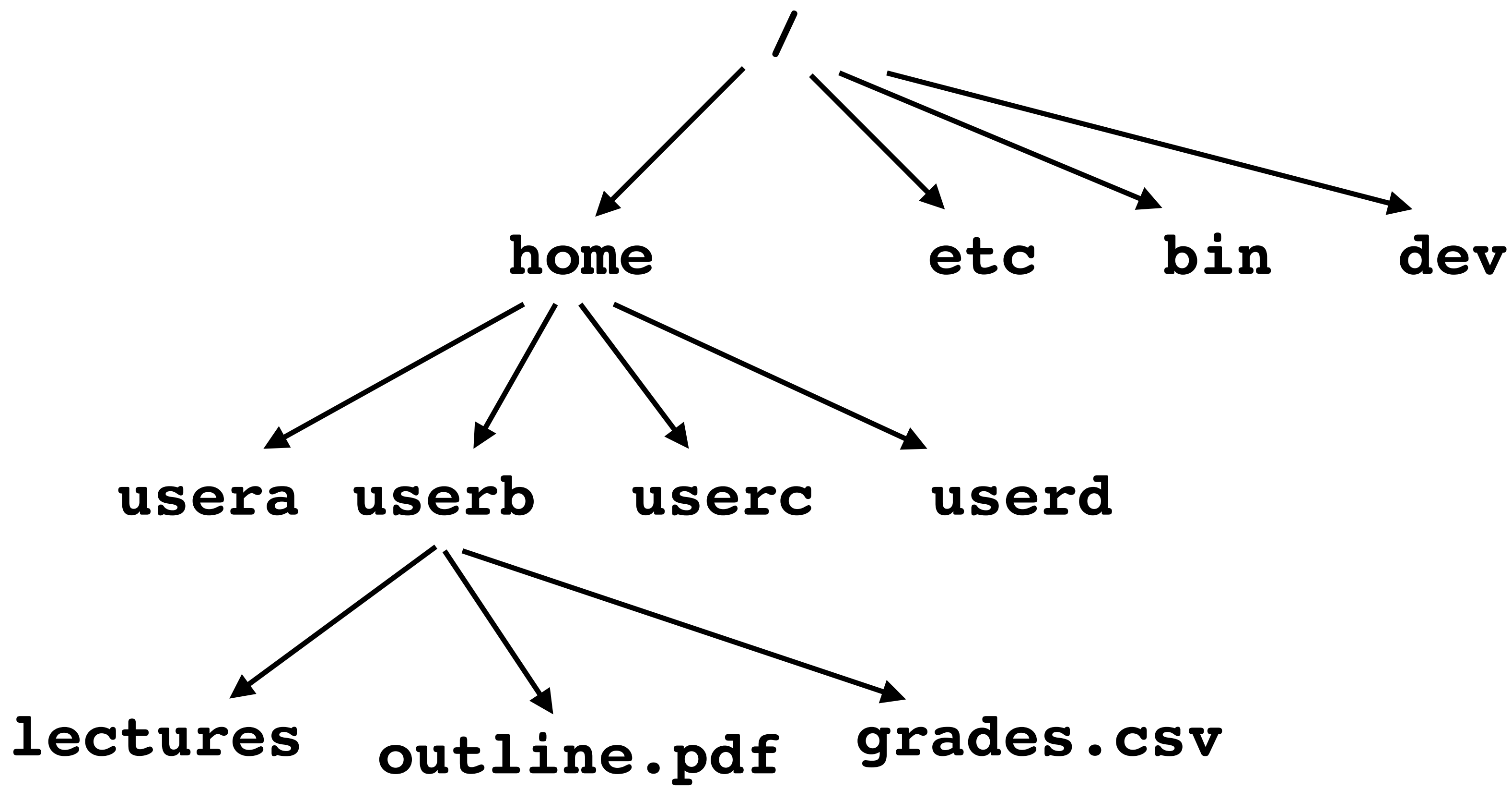
- boot block contains the bootstrap program that runs at boot time to load the OS from that partition
- superblock defines a file system (there is only one per file system)
  - size of the file system and size of the inode list
  - list of free disk blocks (and index of the next free block)
  - list of free inodes (and index of the next free inode)
  - location of the inode of the root directory (inode #2)
- inodes contain file metadata; each inode is identified by a nonnegative integer (i.e., the inode number)
  - can translate the inode number to a location on the disk (inode contains pointers to file data blocks)
  - inodes are either put together as one group (in the inode list) or spread across the disk

Naming and directory structure

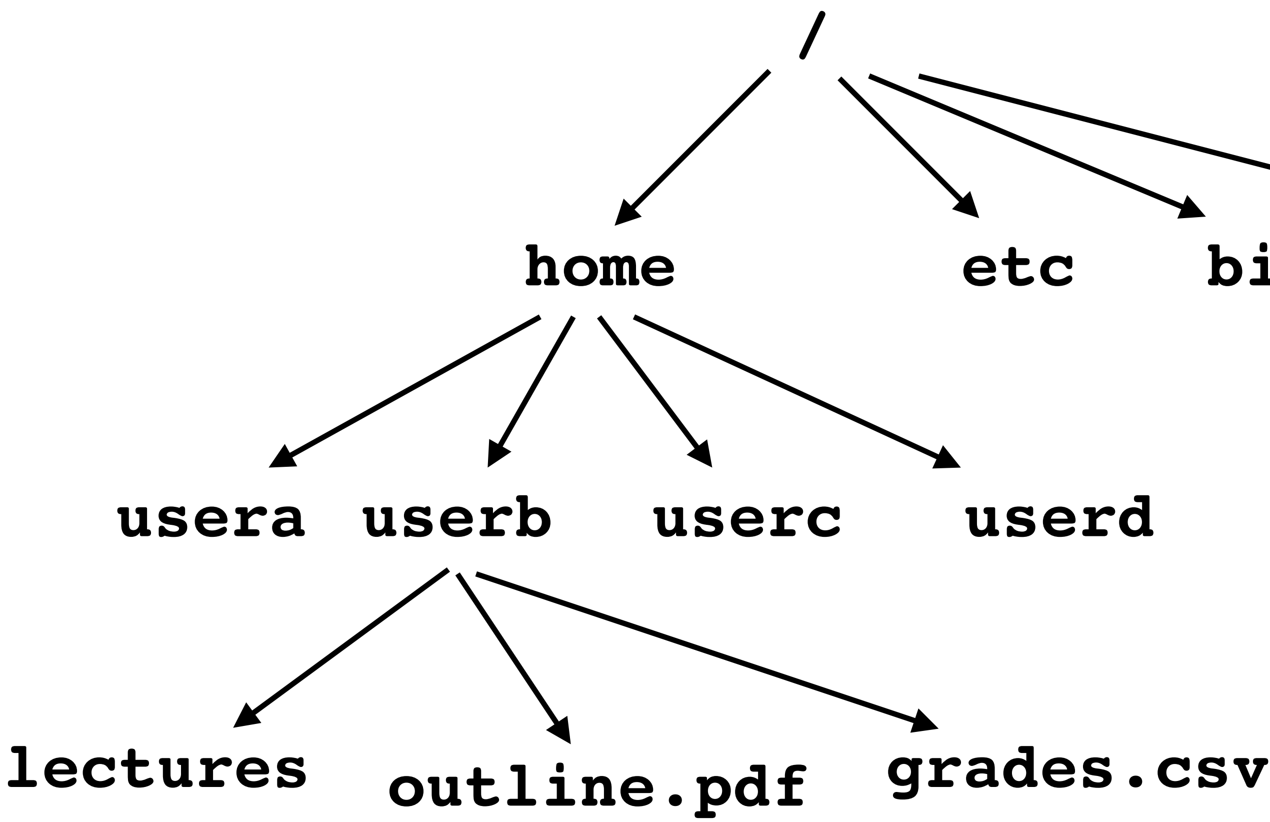
- a directory entry (dentry) is a collection of (name, inode number) pairs for files and directories therein
  - is stored as a regular file (its inode has a special flag bit set)
  - only OS can modify directory; users can just read them
  - . and .. are stored as ordinary file names with inode numbers pointing to the inodes of the same directory and the parent directory respectively



A tree-structured hierarchy



A tree-structured hierarchy



home's dentry contains

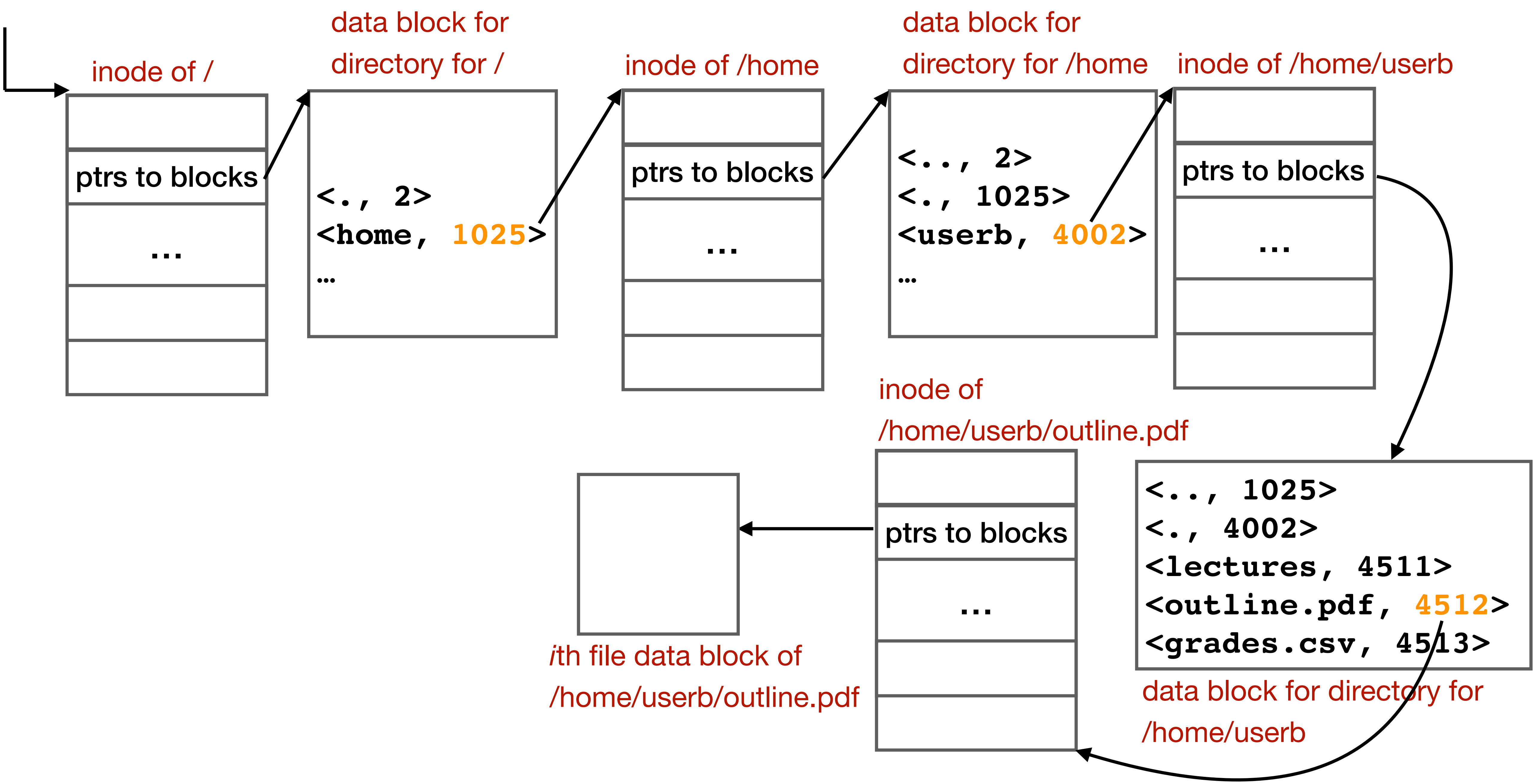
<.., 2>
<., 1025>
<usera, 4001>
<userb, 4002>
<userc, 4003>
<userd, 4004>

userb's dentry contains

<.., 1025>
<., 4002>
<lectures, 4511>
<outline.pdf, 4512>
<grades.csv, 4513>

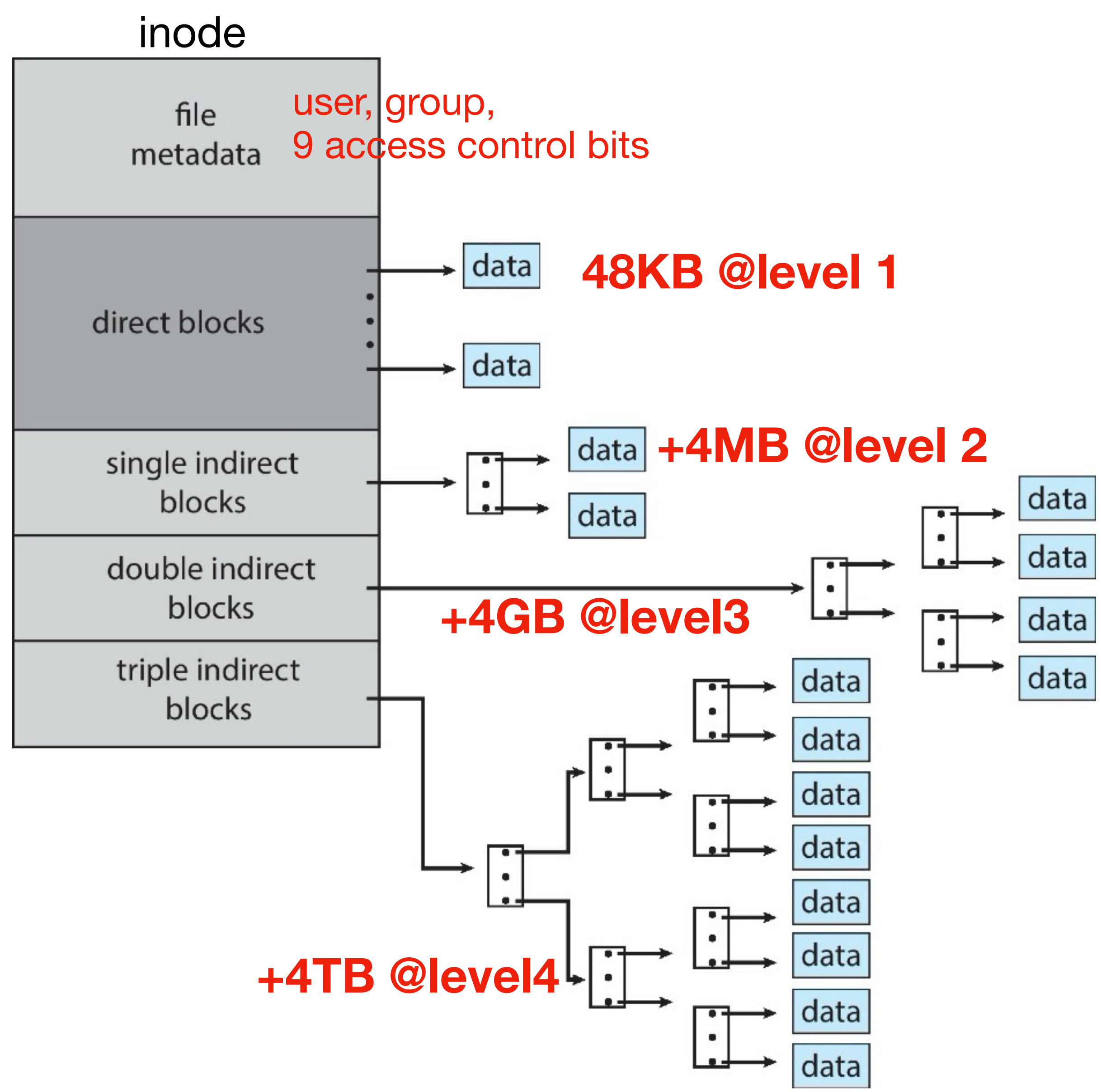
Resolving file location

Where are data blocks of `/home/userb/outline.pdf`?



Asymmetric tree

- original inode format appeared in BSD 4.1 (and later in BSD 4.2)
- each inode contains 15 block pointers (with 32-bit file pointer)
    - the first 12 pointers point to data blocks
        - small files (up to 48KB) use direct blocks only
    - the 13th pointer points to a block of 1024 pointers to 4KB data blocks (1-level indirection)
    - the 14th pointer points to a block of pointers to indirect blocks (2-level indirection)
    - the 15th pointer points to a block of pointers to indirect blocks (3-level indirection)
- shallow tree for small files
    - efficient storage for small files
- deep tree for large files
    - efficient lookup for random access in large files



Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks

Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)

Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)
  - a directory's contents are scattered over the whole disk;
    so listing a directory is slow

Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)
  - a directory's contents are scattered over the whole disk;
    so listing a directory is slow
    - store files from same directory near each other

Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)
  - a directory's contents are scattered over the whole disk;
    so listing a directory is slow
    - store files from same directory near each other
  - free-block list becomes disorganized over time (since blocks that are released are randomly placed in the free-block list) and blocks allocated to file get scattered;
    so sequential access becomes slow as it needs many seeks

Problems

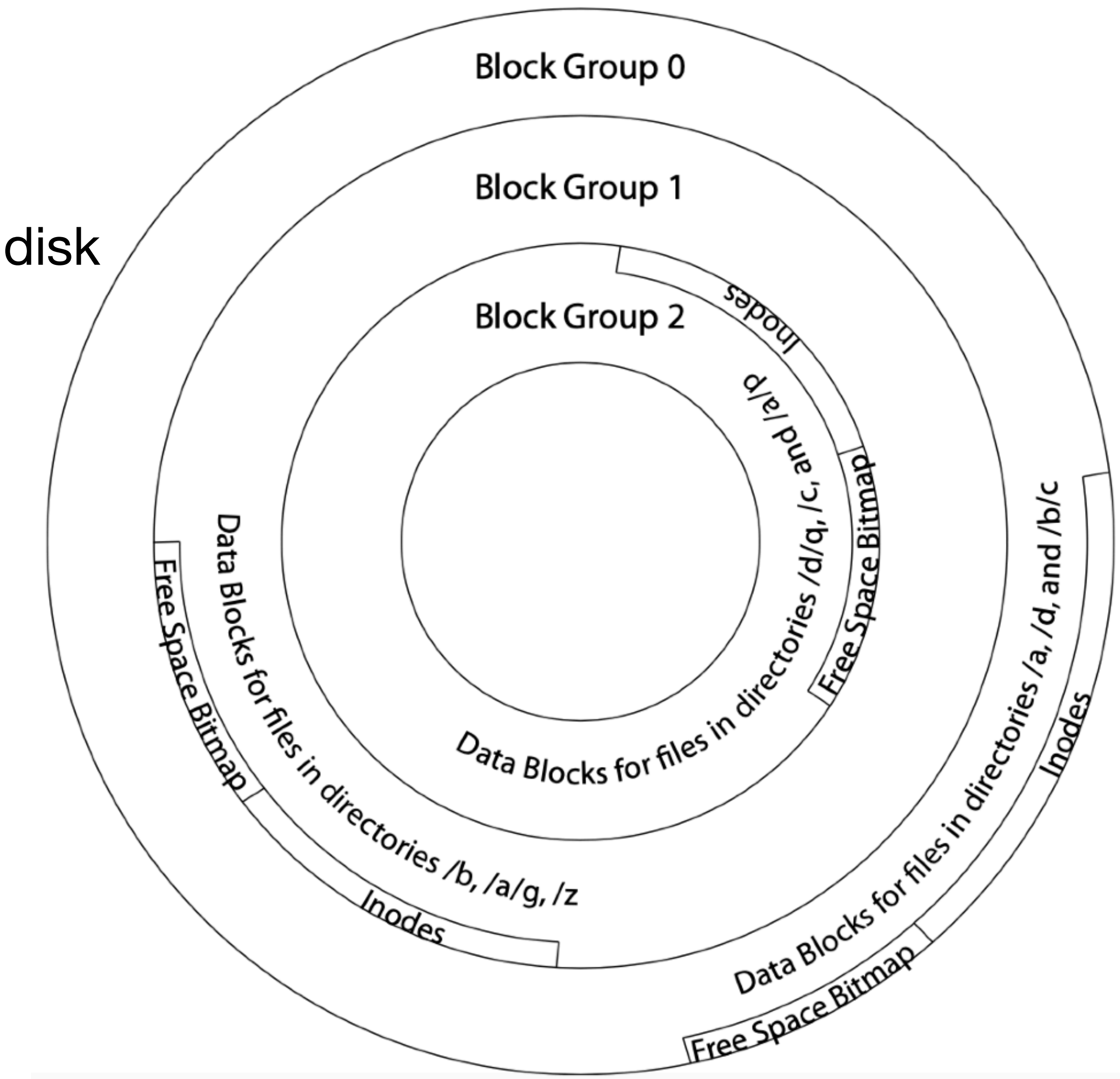
- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)
  - a directory's contents are scattered over the whole disk;
    so listing a directory is slow
    - store files from same directory near each other
  - free-block list becomes disorganized over time (since blocks that are released are randomly placed in the free-block list) and blocks allocated to file get scattered;
    so sequential access becomes slow as it needs many seeks
    - use a bitmap and allocate files contiguously if possible

Problems

- but early versions of UNIX file system had a couple of problems
  - all inodes are put together as one group far from file data blocks;
    so resolving file address requires multiple disk seeks
    - put file data and inode in the same cylinder (hence no disk seeks)
  - a directory's contents are scattered over the whole disk;
    so listing a directory is slow
    - store files from same directory near each other
  - free-block list becomes disorganized over time (since blocks that are released are randomly placed in the free-block list) and blocks allocated to file get scattered;
    so sequential access becomes slow as it needs many seeks
    - use a bitmap and allocate files contiguously if possible
- Fast File System (FFS) proposed allocation and placement policies for BSD 4.2 addressing these problems

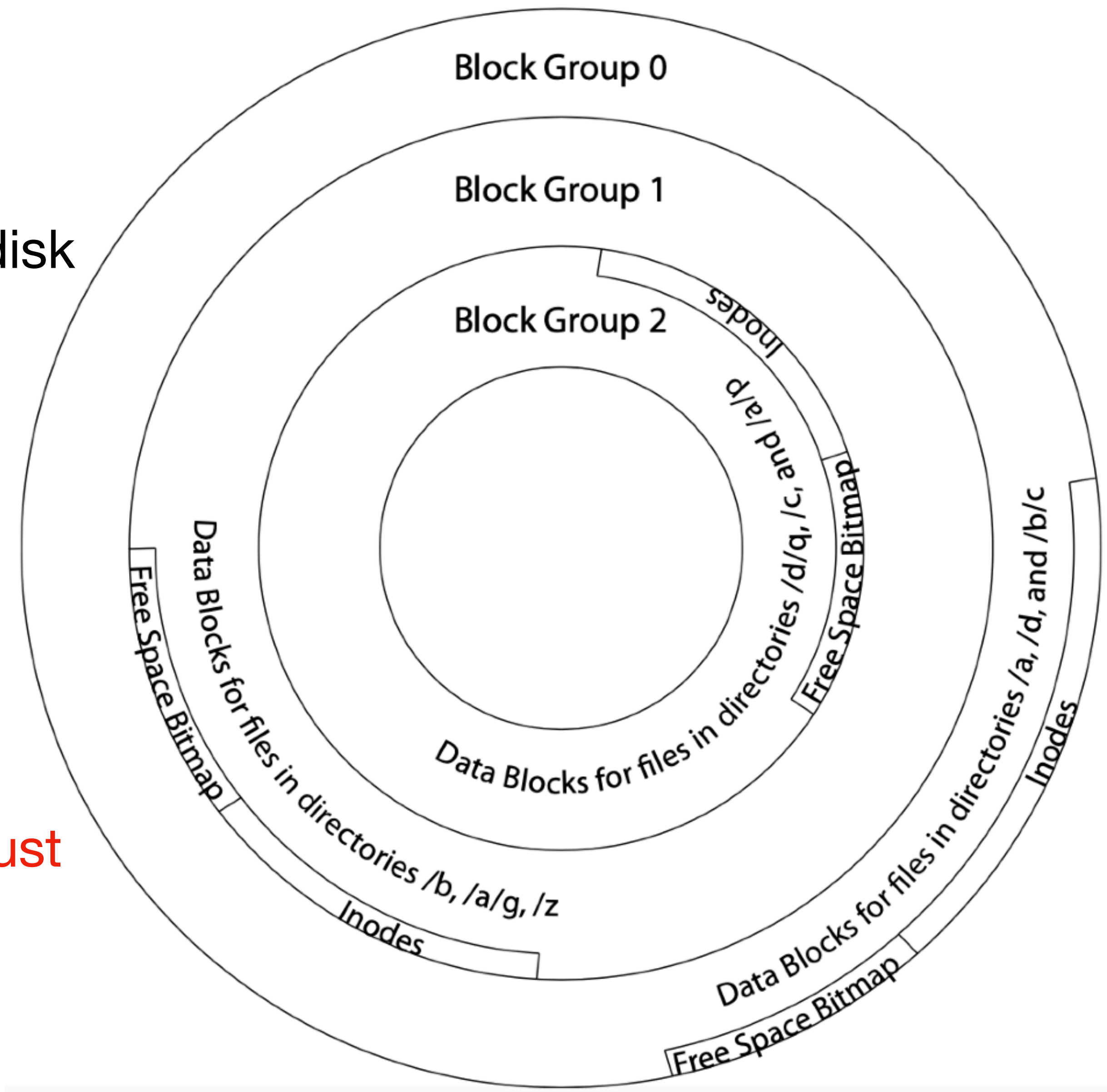
Achieving data locality in FFS

- block group is a set of nearby cylinders
  - fast seek between cylinders in same block group
- block group allocation
  - files in same directory are located in same group
  - subdirectories are located in different block groups
- inode list and bitmap are spread throughout disk
  - near file blocks



Achieving data locality in FFS

- block group is a set of nearby cylinders
  - fast seek between cylinders in same block group
- block group allocation
  - files in same directory are located in same group
  - subdirectories are located in different block groups
- inode list and bitmap are spread throughout disk
  - near file blocks



to allocate according to cylinder groups, disk must have free space scattered across cylinders (10% of disk space is reserved for this purpose)

FFS first-fit block allocation

- when allocating space to a new file, search for a free block from the start of the block group
- when extending a file, search for a free block from the last block allocated to that file
  - if there is not enough free blocks in that range, allocate the remaining blocks from a new range
- there will be few little holes at start and a big hole at end of a block group
  - so small files will be fragmented, and large files will be mostly contiguous

start of block group

FFS first-fit block allocation

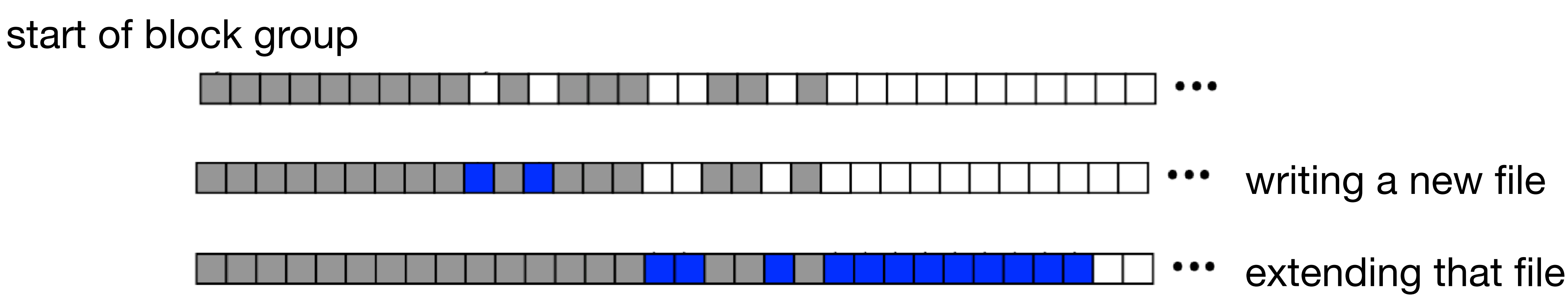
- when allocating space to a new file, search for a free block from the start of the block group
- when extending a file, search for a free block from the last block allocated to that file
  - if there is not enough free blocks in that range, allocate the remaining blocks from a new range
- there will be few little holes at start and a big hole at end of a block group
  - so small files will be fragmented, and large files will be mostly contiguous

start of block group

writing a new file

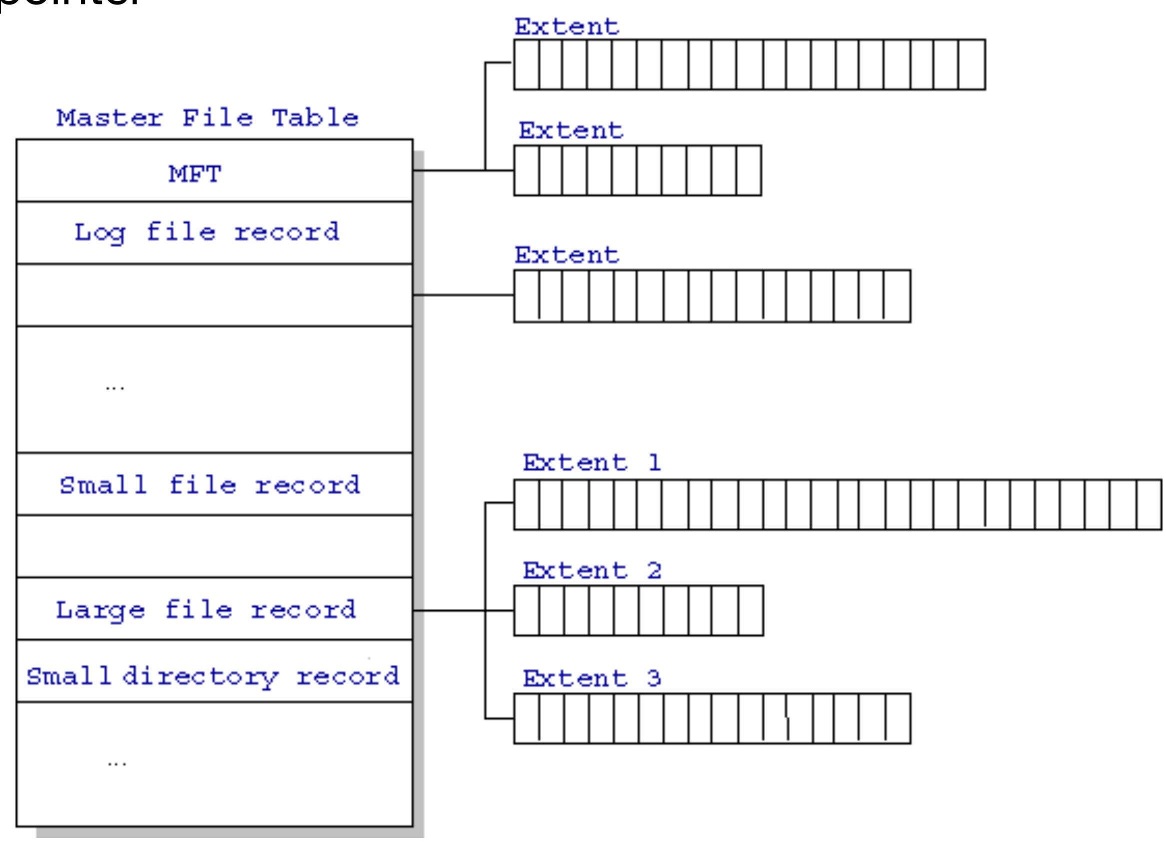
FFS first-fit block allocation

- when allocating space to a new file, search for a free block from the start of the block group
- when extending a file, search for a free block from the last block allocated to that file
  - if there is not enough free blocks in that range, allocate the remaining blocks from a new range
- there will be few little holes at start and a big hole at end of a block group
  - so small files will be fragmented, and large files will be mostly contiguous



NTFS

- master file table is a flexible (1KB to 4KB) storage for metadata and data
- directories organized using B-trees
- index structure is a variable-depth tree
  - mixes direct and indirect block pointers
- variable-length extents (i.e., adjacent disk blocks)
  - have to store start and length of each extent in block pointer
  - user can provide hint as to size of file being created
- journalling for consistency checking
  - all metadata changes are written sequentially to a log



to learn more visit: http://ntfs.com/ntfs_basics.htm