File Systems

Dr. Yingwu Zhu

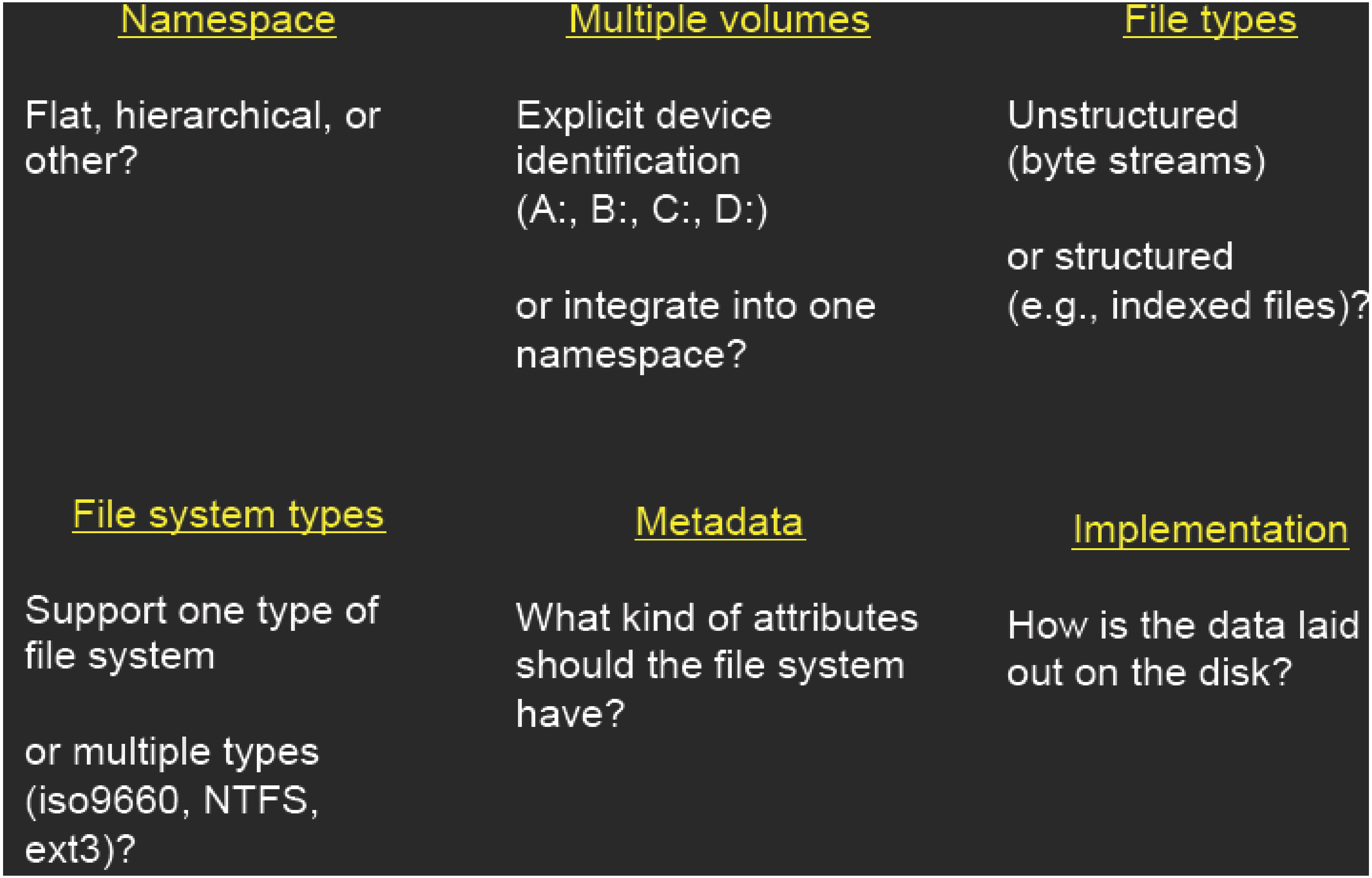
What is a file system?

- Organization of data and metadata
- What’s metadata?
  - Data of data
  - Attributes; things that describe the data
  - Name, length, type of file, creation/modification/access times, permissions, owner, location of data
- File systems usually interact with block devices

Standard Interfaces to Devices

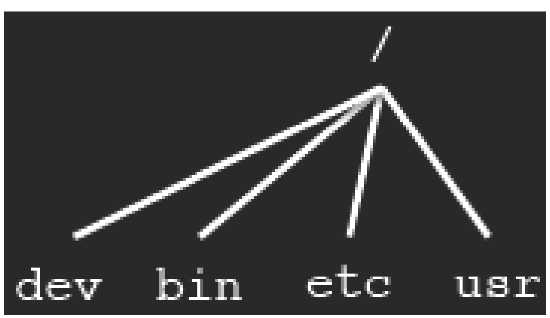
- Block Devices: *e.g.* disk drives, tape drives, Cdrom
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- Network Devices: *e.g.* Ethernet, Wireless, Bluetooth
  - different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - Separates network protocol from network operation
    - Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

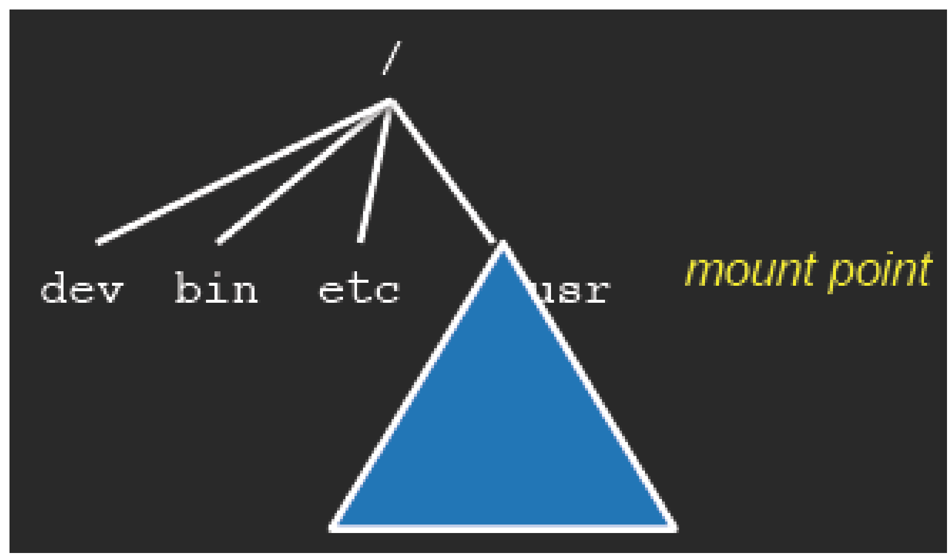
FS design choices



Mounting

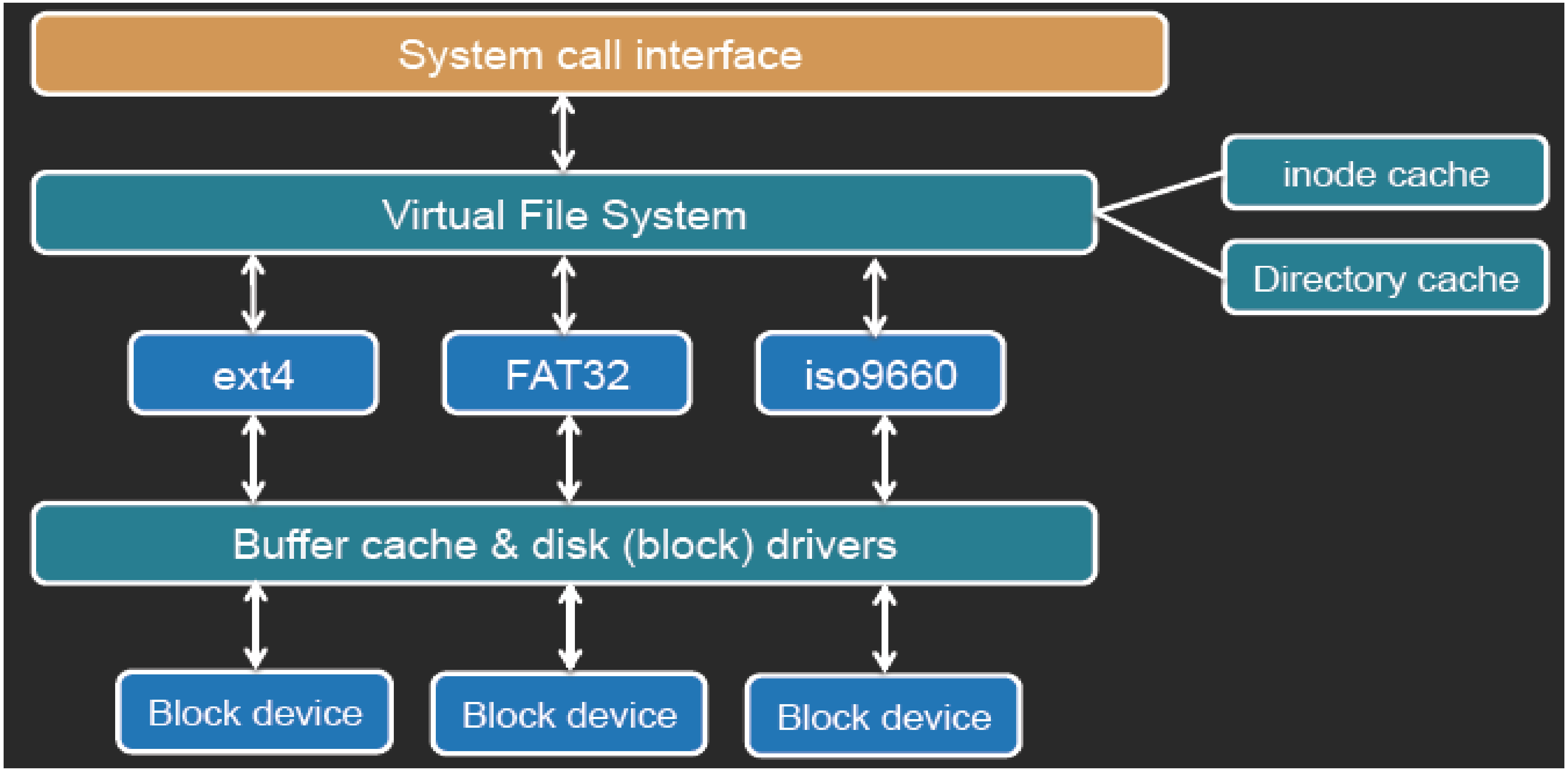
- A file system must be *mounted before it can be used by the operating* system
- The *mount system call is given the file system type, block device & mount* point
- The mounted file system overlays anything under that mount point
- Looking up a pathname may involve traversing multiple mount points





Virtual File System (VFS) Interface

- Abstract interface for a file system object
- Each *real file system interface exports a common interface*



VFS and Other Components

- System call interface: APIs for user programs
- VFS: manages the namespace, keeps track of open files, reference counts, file system types, mount points, pathname traversal.
- File system module: understands how the file system is implemented on the disk. Can fetch and store metadata and data for a file, get directory contents, create and delete files and directories
- Buffer cache: no understanding of the file system; takes read and write requests for blocks or parts of a block and caches frequently used blocks.
- Device drivers: the components that actually know how to read and write data to the disk.

Keeping track of file system types

- Like drivers, file systems can be built into the kernel or compiled as loadable modules (loaded at mount)
- Each file system registers itself with VFS
- Kernel maintains a list of file systems

```
struct file_system_type {
        const char *name;    name of file system type
        int fs_flags;       requires device, fs handles moves, kernel-only mount, …
        struct super_block *(*get_sb)(struct file_system_type *,
                int, char *, void *, struct vfsmount *); set up superblock
        void (*kill_sb) (struct super_block *); clean up at unmount
        struct module *owner; module that owns this
        struct file_system_type *next; next file system type in list
        struct list_head fs_supers; list of all superblocks of this type
        struct lock_class_key s_lock_key; used for lock validation
        struct lock_class_key s_umount_key; used for lock validation
};
```

Keeping track of mounted file systems

- Before mounting a file system, first check if we know the file system type: look through the file_systems list
  - If not found, the kernel daemon will load the file system module
    - /lib/modules/2.6.38-8-server/kernel/fs/ntfs/ntfs.ko!
    - /lib/modules/2.6.38-11-server/kernel/fs/jffs2/jffs2.ko!
    - /lib/modules/2.6.38-11-server/kernel/fs/minix/minix.ko!
- The kernel keeps a linked list of mounted file systems:

  current->namespace->list
- Check that the mount point is a directory and nothing is already mounted there

VFS: Common set of objects

- Superblock: Describes the file system
  - Block size, max file size, mount point
  - One per mounted file system
- inode: represents a single file
  - Unique identifier for every object (file) in a specific file system
  - File systems have methods to translate a name to an inode
  - VFS inode defines all the operations possible on it
- dentry: directory entries & contents
  - Name of file/directory, inode, a pointer to the parent dentry
  - Directory entries: name to inode mappings
- file: represents an open file
  - VFS keeps state: mode, read/write offset, etc.
  - Per-process view

VFS Superblock

- Structure that represents info about the file system
- Includes
  - File system name
  - Size
  - State (clean or dirty)
  - Reference to the block device
  - List of operations for managing inodes within the file system:
    - *alloc_inode, destroy_inode, read_inode, write_inode, sync_fs, …*

VFS Superblock

```
struct super_operations {
        struct inode *(*alloc_inode) (struct super_block *sb);
        void (*destroy_inode) (struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs) (struct super_block *, int);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*show_options) (struct seq_file *, struct vfsmount *);
};
```

inode

- Uniquely identifies a file in a file system
- Access metadata (attributes) of the file (except name)

```
struct inode {
        unsigned long i_ino;
        umode_t i_mode;
        uid_t i_uid;
        gid_t i_gid;
        kdev_t i_rdev;
        loff_t i_size;
        struct timespec i_atime;
        struct timespec i_ctime;
        struct timespec i_mtime;
        struct super_block *i_sb;
        struct inode_operations *i_op;      inode operations
        struct address_space *i_mapping;
        struct list_head i_dentry;
        ...
}
```

inode operations

- Functions that operate on file & directory *names and attributes*

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *, int);
        struct dentry * (*lookup) (struct inode *, struct dentry *);
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        int (*unlink) (struct inode *, struct dentry *);
        int (*symlink) (struct inode *, struct dentry *, const char *);
        int (*mkdir) (struct inode *, struct dentry *, int);
        int (*rmdir) (struct inode *, struct dentry *);
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *,int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

File operations

- Functions that operate on file & directory *data*

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                        unsigned long, unsigned long);
```

File operations

- Not all functions need to be implemented!

```
struct file_operations mydriver_fops = {
    .owner = MYDRIVER_MODULE;
    .open = mydriver_open;          /* allocate resources */
    .read = mydriver_read;
    .write = mydriver_write;
    .ioctl = mydriver_ioctl;
    .release = mydriver_release;  /* release resources */
    /* llseek, readdir, poll, mmap, readv, etc. not implemented */
};

register_chrdev(MYDRIVER_MAJOR_NUM, "mydriver", &mydriver_fops)
```

File System Implementation

Some Terminology

- Disk
  - Non-volatile block-addressable storage.
- Disk Block = sector
  - Smallest chunk of I/O on a disk
  - Most disks have 512-byte blocks
  - LBA: a unique number known as a **logical block address**
- Partition
  - Subset of all blocks on a disk. A disk has ≥ 1 partitions
- Volume
  - Disk, disks, or partition that contains a file system
  - A volume may span disks

More Terms

- Superblock
  - Area on the volume that contains key file system information
- Metadata
  - Attributes of a file, not the file contents (data)
  - E.g., modification time, length, permissions, owner
- inode
  - A structure that stores a file's metadata and location of its data

Files

- Contents (Data)
  - Unstructured (byte stream) or structured (records)
  - Stored in data blocks
  - Find a way of allocating and tracking the blocks that a file uses
- Metadata
  - Usually stored in an inode … sometimes in a directory entry
  - Except for the name, which is stored in a directory

Directories

- A directory is just a file containing names & references
  - Name → (metadata, data) *Unix (UFS) approach*
  - (Name, metadata) → data *MS-DOS (FAT) approach*
- Linear list
  - Search can be slow for large directories.
  - Cache frequently-used entries
- Hash table
  - Linear list but with hash structure
  - Hash(name)
- More exotic structures: B-Tree, HTree

Lay out file data on disks

Block Allocation: Contiguous

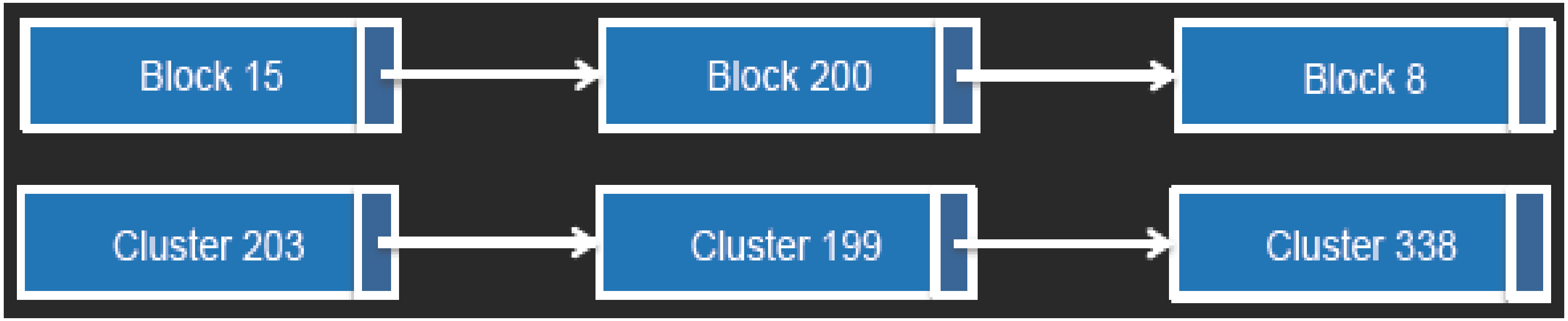
- Each file occupies a set of adjacent blocks
- You just need to know the starting block & file length
- We'd love to have contiguous storage for files!
  - Minimize disk seeks when accessing a file

Problems with contiguous allocation

- Storage allocation is a pain
  - External fragmentation: free blocks of space scattered throughout
  - vs. Internal fragmentation: unused space within a block (allocation unit)
  - Periodic defragmentation: move files
- Concurrent file creation: how much space do you need?
- Compromise solution: extents
  - Allocate a contiguous chunk of space
  - If the file needs more space, allocate another chunk (extent)
  - Need to keep track of all extents
  - Not all extents will be the same size: it depends how much contiguous space you can allocate

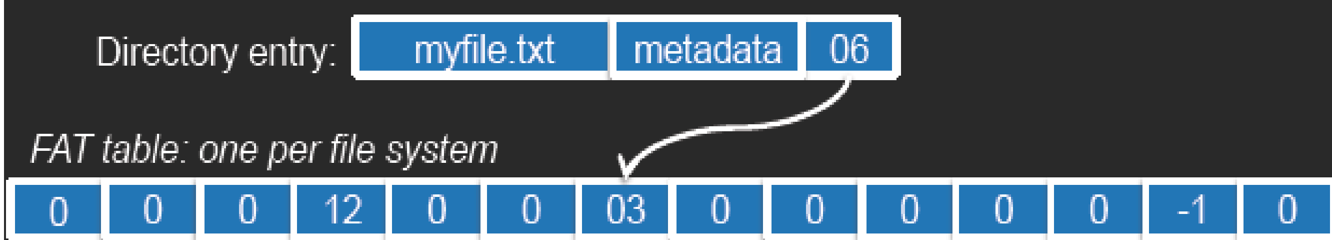
Block allocation: Linked Allocation

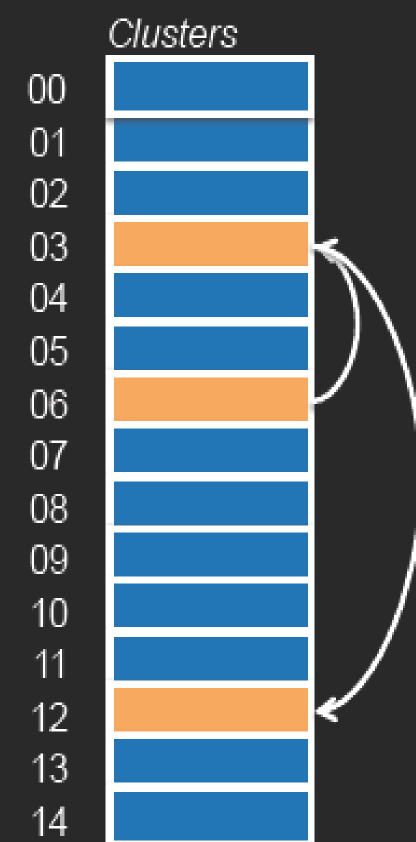
- A file's data is a linked list of disk blocks
  - Directory contains a pointer to the first block of the file
  - Each block contains a pointer to the next block
- Problems
  - Only good for sequential access
  - Each block uses space for the pointer to the next block
- Clusters
  - Multiples of blocks: reduce overhead for block pointer & improve throughput
  - *A cluster is the smallest amount of disk space that can be allocated to a file*
  - Penalty: increased internal fragmentation



File Allocation Table (DOS/Windows FAT)

- Variation of Linked Allocation
- Section of disk at beginning of the volume contains a file allocation table
- The table has one entry per block. Contents contain the next logical block (cluster) in the file.





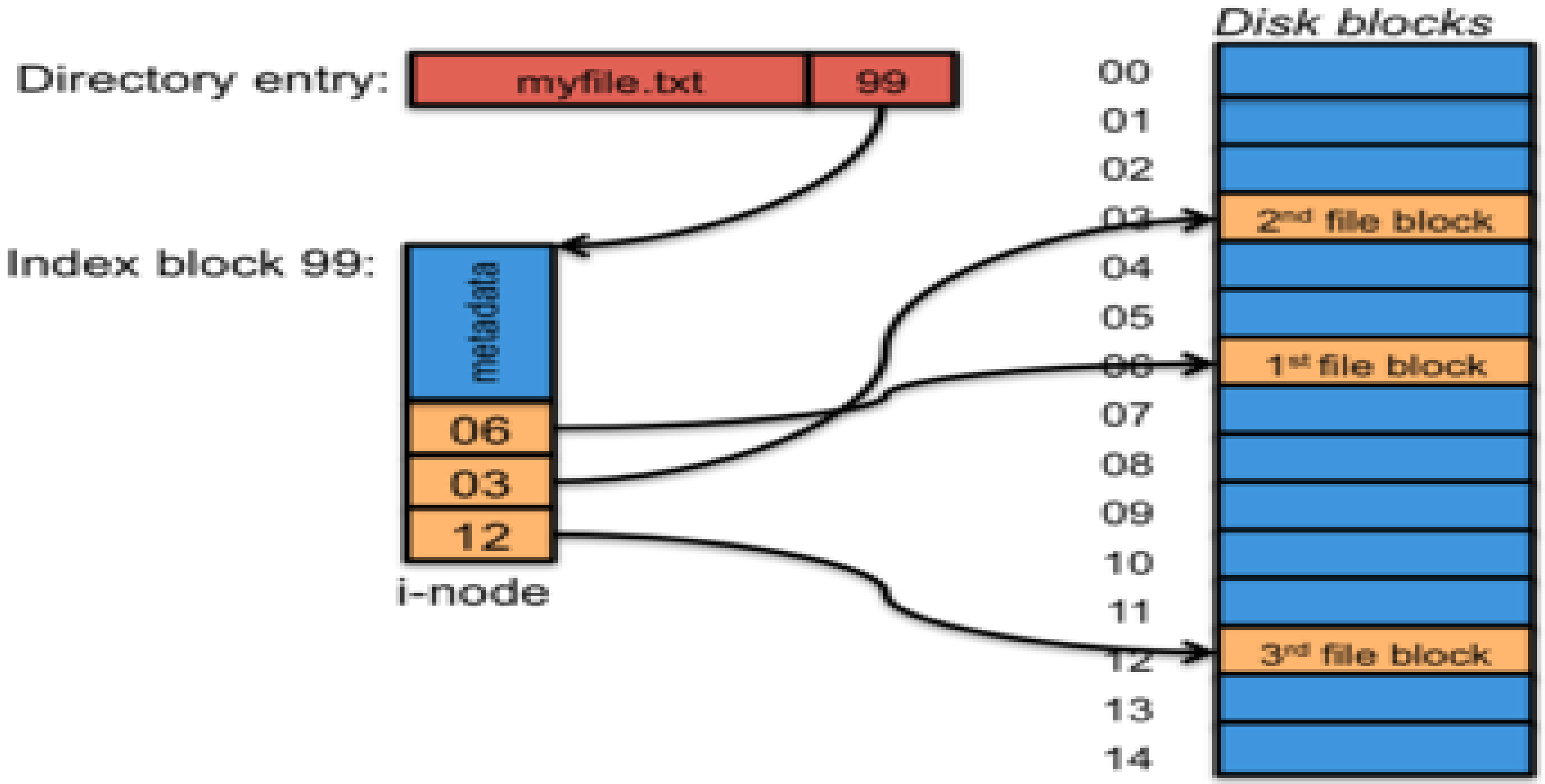
- FAT-16: 16-bit block pointers
  - 16-bit cluster numbers; up to 64 sectors/cluster
  - Max file system size = 2 GB (with 512 byte sectors)
- FAT-32: 32-bit block pointers
  - 32-bit cluster numbers; up to 64 sectors/cluster
  - Max file system size = 8 TB (with 512 byte sectors)
  - Max file size = 4 GB

Indexed Allocation

- Linked allocation is not efficient for random access
- FAT requires storing the *entire table in memory for* efficient access
- Indexed allocation:
  - Store the entire list of block pointers for a file in one place: the index block (inode)
  - One inode per file
  - We can read this into memory when we open the file

Indexed Allocation

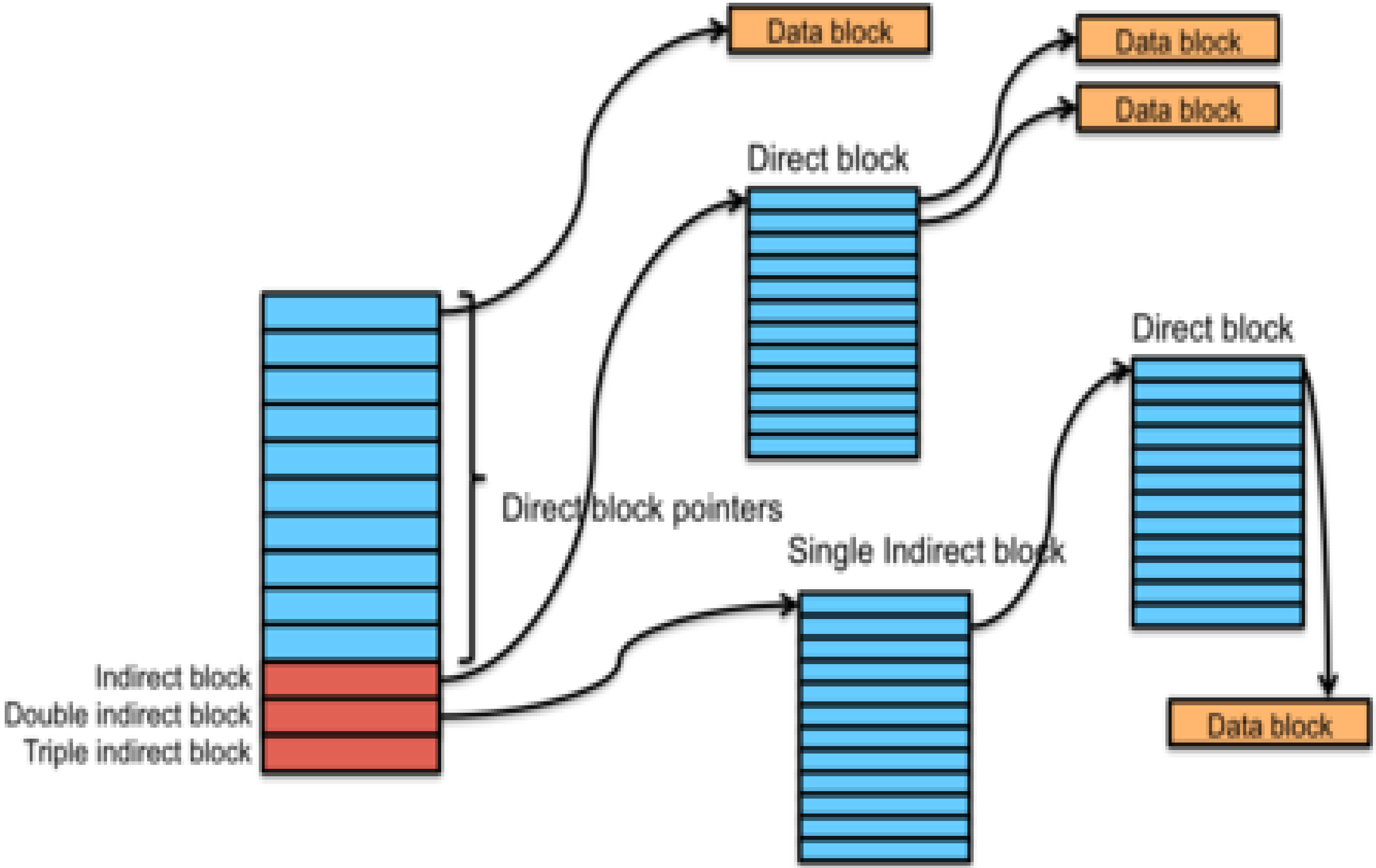
- Directory entry contains name and inode number
- inode contains file metadata (length, timestamps, owner, etc.) *and a block map*
- On file *open, read the inode to get the index map*



Combined Indexing (Unix File Systems)

- We want inodes to be a fixed size
  - Easy to allocate and reuse
  - Easy to locate inodes on disks
- Large files get
  - Single indirect block
  - Double indirect block
  - Triple indirect block
- 1024-byte blocks, 32-bit block pointers

Combined Indexing (Unix File Systems)



Unix File Example

- Unix File System
  - 1024-byte blocks, 32-bit block pointers
  - inode contains
    - 10 direct blocks, 1 indirect, 1 double-indirect, 1 triple indirect
- Capacity
  - Direct blocks will address : 1K x 10 blocks = 10240 bytes
  - 1 level of indirect block: (1K / 4) x 1K = 256K bytes
  - 1 double indirect block: (1K/4)x(1K/4)x1K = 64MB
  - 1 triple indirect block: (1K/4)x(1K/4)x(1K/4)x1K = 16GB

Extent Lists

- Extents: Instead of listing block addresses
  - Each address represents a range of blocks
  - Contiguous set of blocks
  - E.g., 48-bit block # + 2-byte length (total = 64 bits)
- Why are they attractive?
  - Less block numbers to store if we have lots of contiguous allocation
- Problem: file seek operations
  - Locating a specific location requires traversing a list
  - Extra painful with indirect blocks

Implementing File Operations

Initialization

- Low-level formatting (file system independent)
  - Define blocks (sectors) on a track
  - Create spare sectors
  - Identify and remap bad blocks
- High-level formatting (file system specific)
  - Define the file system structure
  - Initialize the free block map
  - Initialize sizes of inode and journal areas
  - Create a top-level (root) directory

File Open

- Two-step process
  - Pathname Lookup (*namei function in kernel)*
    - Traverse directory structure based on the pathname to find file
    - Return the associated inode
    - (cache frequently-used directory entries)
  - Verify access permissions
    - If OK, allocate in-memory structure to maintain state about access
    - (e.g., that file is open read-only)

File Writes

- A *write either overwrites data in a file or adds data to the file, causing it to* grow
  - Allocate disk blocks to hold data
  - Add the blocks to the list of blocks owned by the file
    - Update free block bitmap, the inode, and possibly indirect blocks
    - Write the file data
    - Modify inode metadata (file length)
    - Change current file offset in kernel

Deleting Files

- Remove name from the directory
  - Prevent future access
- If there are no more links to the inode (disk references)
  - mark the file for deletion
- … and if there are no more programs with open handles to the file (in-memory references)
  - Release the resources used by the file
    - Return data blocks to the free block map
    - Return inode to the free inode list
- Example:
  - Open temp file, delete it, continue to access it
  - OS cleans up the data when the process exits

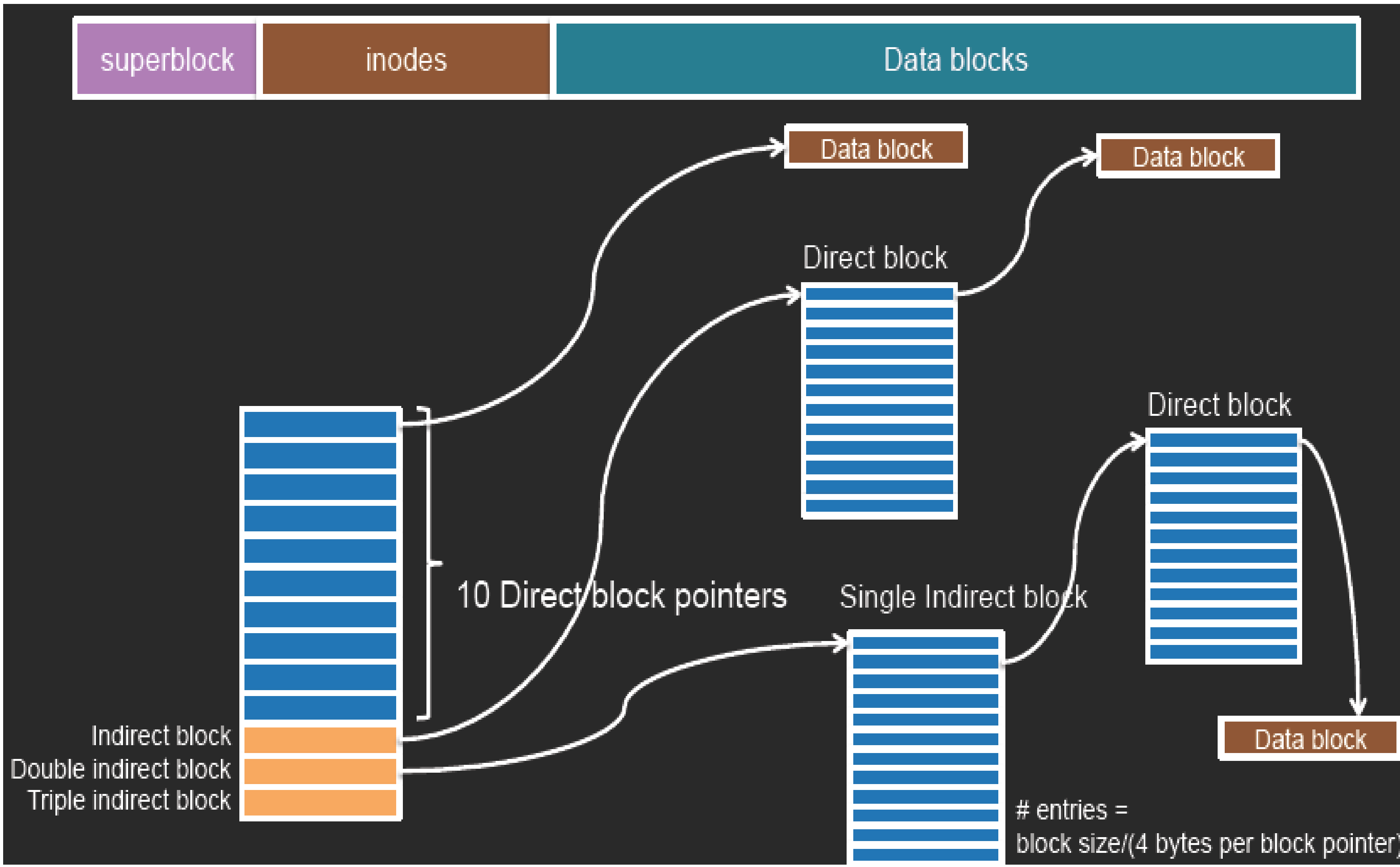
Additional File System Operations

- Hard links (aliases)
  - Multiple directory entries (file names) that refer to the same inode
  - inode contains reference count to handle deletion
- Symbolic links
  - File data contains a path name
  - Underlying file can disappear or change
- Access control lists (ACLs)
  - Classic UNIX approach: user, group, world permissions
  - ACL: enumerated list of users and permissions
    - Variable size

Additional File System Operations

- Extended attributes (NTFS, HFS+, XFS, etc.)
  - E.g., store URL from downloaded web/ftp content, app creator, icons
- Indexing
  - Create a database for fast file searches
- Journaling
  - Batch groups of changes. Commit them at once to a transaction log

UFS (Unix File System)



UFS

- Superblock contains:
  - Size of file system
  - # of free blocks
  - list of free blocks (+ pointer to free block lists)
  - index of the next free block in the free block list
  - Size of the inode list
  - Number of free inodes in the file system
  - Index of the next free inode in the free inode list
  - Modified flag (clean/dirty)

UFS

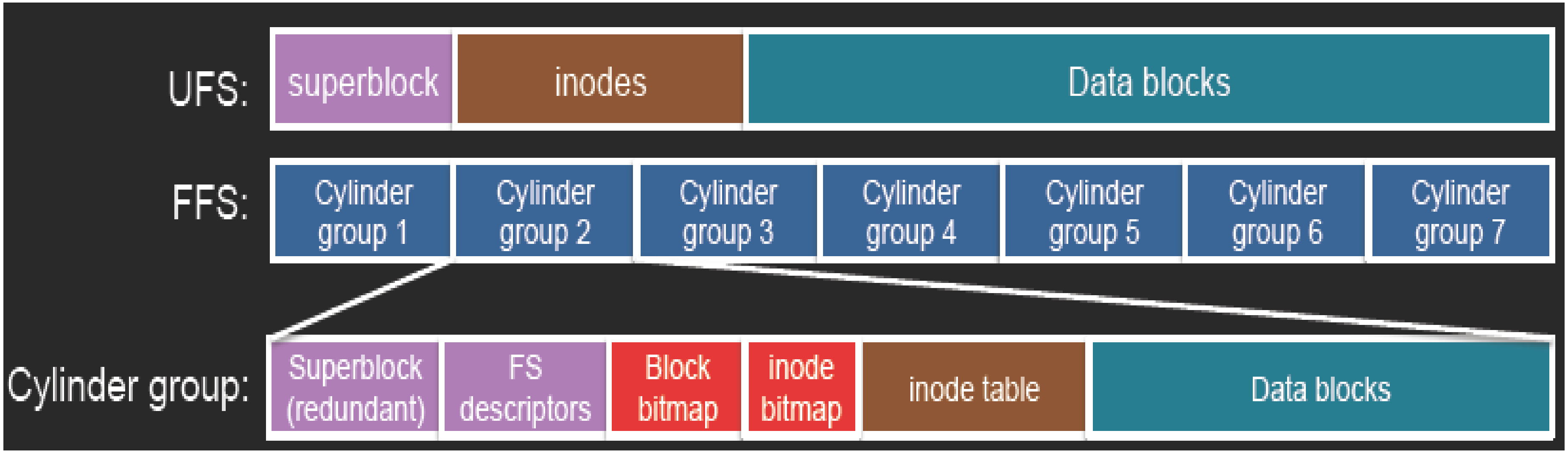
- Free space managed as a linked list of blocks
  - Eventually this list becomes random
  - Every disk block access will require a seek!
- Fragmentation is a big problem
- Typical performance was often:
  - 2–4% of raw disk bandwidth!

BSD FFS (Fast File System)

- Try to improve UFS
- Improvement #1: Use larger blocks
  - ≥ 4096 bytes instead of UFS’s 512-byte or 1024-byte blocks
    - Block size is recorded in the superblock
  - Just doubling the block size resulted in > 2x performance!
  - 4 KB blocks let you have 4 GB files with only two levels of indirection
  - Problem: increased internal fragmentation
    - Lots of files were small
  - Solution: Manage fragments within a block (down to 512 bytes)
    - A file is 0 or more full blocks and possibly one fragmented block
    - Free space bitmap stores fragment data
    - As a file grows, fragments are copied to larger fragments and then to a full block
    - Allow user programs to find the optimal block size
      - Standard I/O library and others use this
  - Also, avoid extra writes by caching in the system buffer cache

FFS

- Improvement #2: Minimize head movement (reduce seek time)
  - Seek latency is usually much higher than rotational latency
  - Keep file data close to its inode to minimize seek time to fetch data
  - Keep related files & directories together
  - Cylinder: collection of all blocks on the same track on all heads of a disk
  - Cylinder group: Collection of blocks on one or more consecutive cylinders



How to find inodes?

- UFS was easy:
  - inodes_per_block = sizeof(block)/sizeof(inode)
  - inode_block = inode / inodes_per_block
  - block_offset = (inode % inodes_per_block) * sizeof(inode)
- FFS
  - We need to know how big each chunk of inodes in a cylinder group is: keep a table

FFS

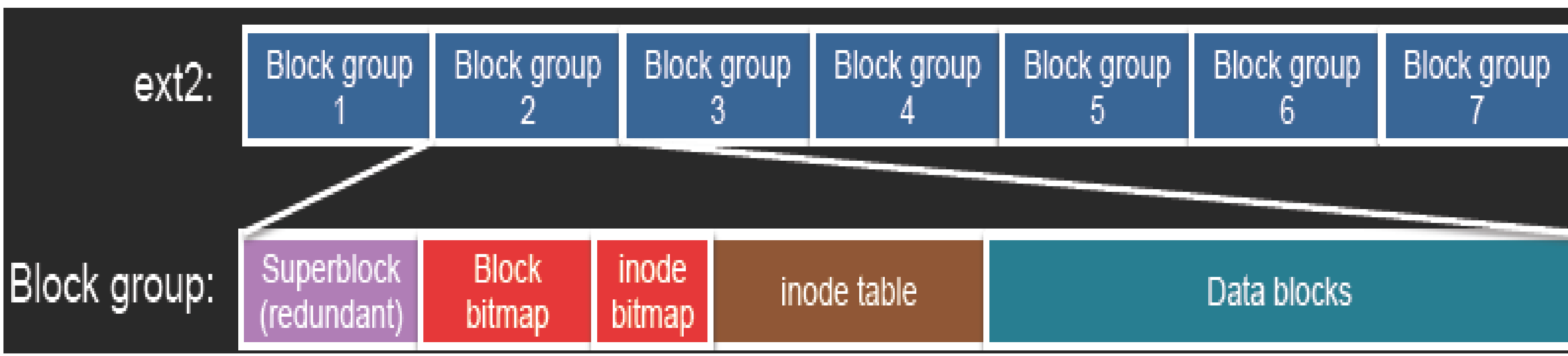
- Optimize for sequential access
- Allocate data close together
  - Pre-allocate up to 8 adjacent blocks when allocating a block
  - Achieves good performance under heavy loads
  - Speeds sequential reads
- Prefetch
  - If 2 or more logically sequential blocks are read
    - Assume sequential read
    - and request one large I/O on the entire range of sequential blocks
  - Otherwise, schedule a read-ahead (for the next disk block in the file)

FFS

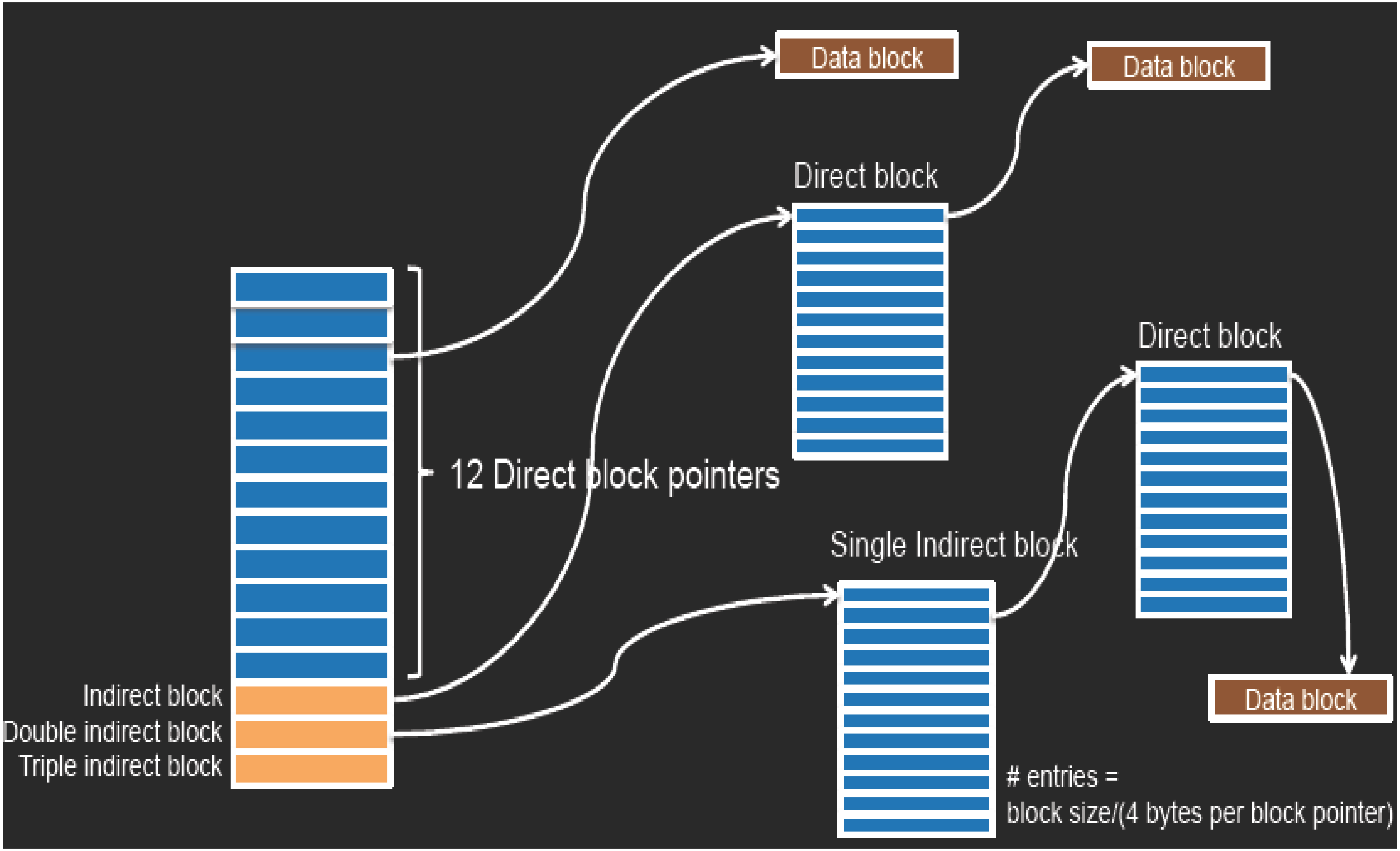
- Improve fault tolerance
  - Strict ordering of writes of file system metadata
  - *fsck still requires up to five passes to repair*
  - All metadata writes are synchronous (not buffered)
  - This limits the max # of I/O operations (thruput)
- • Directories
  - Max filename length = 256 bytes (vs. 12 bytes of UFS)
- Symbolic links introduced
  - Hard links could not point to directories (avoid namespace cycles) and worked only within the FS
- Performance:
  - 14-47% of raw disk bandwidth
  - Better than the 2-5% of UFS

Linux ext2

- Similar to BSD FFS
- No fragments
  - No need to worry about wasted space in modern disks
- No cylinder groups (not useful in modern disks) – block groups
- Divides disk into fixed-size block groups
  - Like FFS, somewhat fault tolerant: recover chunks of disk even if some parts are not accessible



Linux ex2


Data block
Data block
Direct block
Direct block
12 Direct block pointers
Single Indirect block
Indirect block
Double indirect block
Triple indirect block
Data block
entries =
block size/(4 bytes per block pointer)

Linux ex2

- Improve performance via aggressive caching
  - Reduce fault tolerance because of no synchronous writes
  - Almost all operations are done in memory until the buffer cache gets flushed
- Unlike FFS:
  - No guarantees about the consistency of the file system
    - Don’t know the order of operations to the disk: risky if they don’t all complete
  - No guarantee on whether a write was written to the disk when a system call completes
- In most cases, ext2 is *much faster than FFS*

Journaling

File system inconsistencies

Example:

- Writing a block to a file may require:
  - inode is
    - updated with a new block pointer
    - Updated with a new file size
  - Data free block bitmap is updated
  - Data block contents written to disk
- If *all of these are not written, we have a file system inconsistency*
- Consistent update problem

Journaling

- Journaling = write-ahead logging
- Keep a transaction-oriented journal of changes
  - Record what you are about to do (*along with the data!*)

```
Transaction-begin
  New inode 779
  New block bitmap, group 4
  New data block 24120
Transaction-end
```

  - Once this has committed to the disk then overwrite the real data
  - If all goes well, we don't need this transaction entry
  - If a crash happens any time after the log was committed
    - Replay the log on reboot (redo logging)
- This is called *full data journaling*

Writing the journal

- Writing the journal all at once would be great but is risky
  - We don’t know what order the disk will schedule the block writes
  - Don’t want to risk having a “transaction-end” written while the contents of the transaction have not been written yet
  - Write all blocks *except transaction-end*
  - Then write transaction-end
- If the log is replayed and a transaction-end is missing, ignore the log entry

Cost of journaling

- We're writing everything twice
    - …and constantly seeking to the journal area of the disk
- Optimization
    - Do not write user data to the journal
    - Metadata journaling (also called ordered journaling)

```
Transaction-begin
  New inode 779
  New block bitmap, group 4
Transaction-end
```

- What about the data?
    - Write it to the disk first (not in the journal)
    - Write transaction w/o without marking the end
    - Only after all previous ops are committed to the disk, then mark the end of the transaction
    - This prevents pointing to garbage after a crash and journal replay

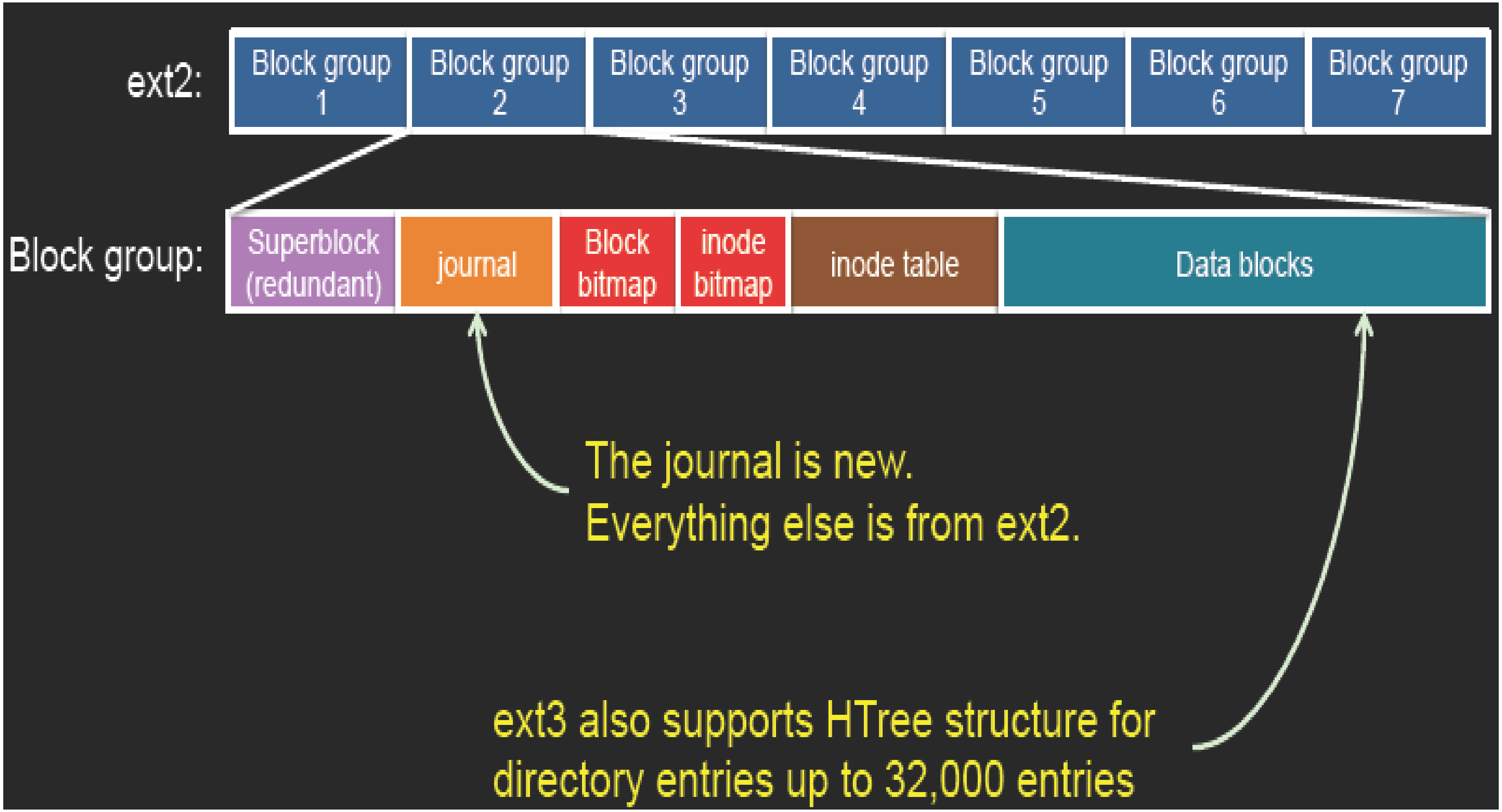
Linux ex3

- ext3 = ext2 + journaling (mostly)
- Goal: improved fault recovery
  - Reduce the time spent in checking file system consistency & repairing the file system by journaling

ext3 journaling options

- journal
  - full data + metadata journaling
  - [slowest]
- ordered
  - Data blocks written first, then metadata journaling
  - Write a transaction-end only when the other writes have completed
- writeback
  - Metadata journaling with no ordering of data blocks
  - Recent files can get corrupted after a crash
  - [fastest]

ex3 layout


ext2:
Block group 1
Block group 2
Block group 3
Block group 4
Block group 5
Block group 6
Block group 7
Block group:
Superblock (redundant)
journal
Block bitmap
inode bitmap
inode table
Data blocks
The journal is new.
Everything else is from ext2.
ext3 also supports HTree structure for directory entries up to 32,000 entries

Linux ext4

- Large file system support
  - 1 exabyte (10^18 bytes); file sizes to 16 TB
- Extents used instead of block maps
  - Range of contiguous blocks
  - 1 extent can map up to 12 MB of space (4 KB block size)
  - 4 extents per inode. Additional ones are stored in an HTree (constant-depth tree similar to a B-tree)
- Ability to pre-allocate space for files
  - Increase chance that it will be contiguous
- Delayed allocation
  - Allocate on flush – only when data is written to disk
  - Improve block allocation decisions because we know the size

Linux ex4

- Over 64,000 directory entries (vs. 32,000 in ext3)
  - HTree structure
- Journal checksums
  - Monitor journal corruption
- Faster file system checking
  - Ignore unallocated block groups
- Interface for multiple-block allocations
  - Increase contiguous storage
- Timestamps in nanoseconds
  - Timestamps in an inode (last modified time, last accessed time, created time)
  - one-second granularity in ext3

Acknowledgement

- Some slides are adapted from Dr. Paul Krzyzanowski