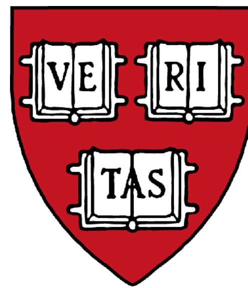
CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 14: Filesystem Organization
April 5, 2007

© 2007 Matt Welsh – Harvard University

Topics for today

File system design overview

- Basic structures: inodes, files, and directories
- Buffer cache: How do we avoid going to disk every time we need to read or write a file?

© 2007 Matt Welsh – Harvard University

Filesystems

A *filesystem* provides a high-level application access to disk

- *As well as CD, DVD, tape, floppy, etc...*
- Masks the details of low-level sector-based I/O operations
- Provides structured access to data (files and directories)
- Caches recently-accessed data in memory

Hierarchical filesystems: Most common type

- Organized as a tree of directories and files

Byte-oriented vs. record-oriented files

- UNIX, Windows, etc. all provide byte-oriented file access
    - *May read and write files a byte at a time*
- Many older OS's provided only record-oriented files
    - *File composed of a set of records; may only read and write a record at a time*

Versioning filesystems

- Keep track of older versions of files
- e.g., VMS filesystem: Could refer to specific file versions: `foo.txt;1, foo.txt;2`

© 2007 Matt Welsh – Harvard University

Filesystem Operations

Filesystems provide a standard interface to files and directories:

- Create a file or directory
- Delete a file or directory
- Open a file or directory – allows subsequent access
- Read, write, append to file contents
- Add or remove directory entries
- Close a file or directory – terminates access

What other features do filesystems provide?

- **Accounting and quotas** – prevent your classmates from hogging the disks
- **Backup** – some filesystems have a "$HOME/.backup" containing automatic snapshots
- **Indexing and search capabilities**
- **File versioning**
- **Encryption**
- **Automatic compression** of infrequently-used files

Should this functionality be part of the filesystem or built on top?

- Classic OS community debate: Where is the best place to put functionality?

© 2007 Matt Welsh – Harvard University

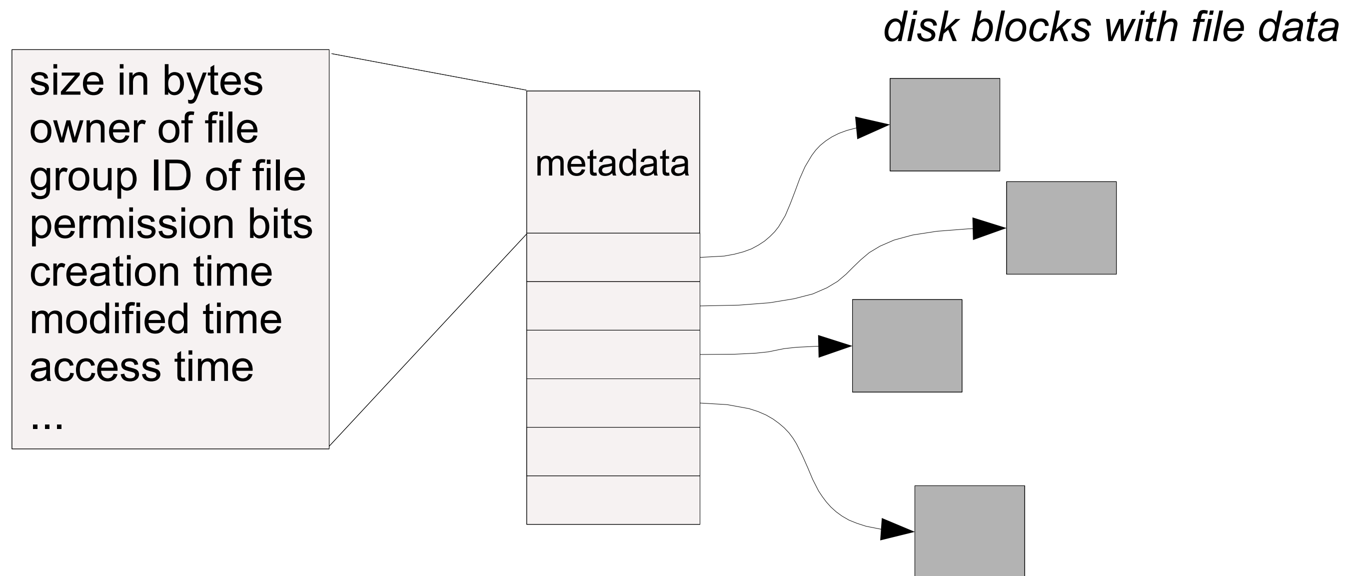
Basic Filesystem Structures

Every file and directory is represented by an **inode**

- Stands for “index node”

Contains two kinds of information:

- 1) Metadata describing the file's owner, access rights, etc.
- 2) Location of the file's blocks on disk



What's one obvious thing missing from the inode metadata?

© 2007 Matt Welsh – Harvard University

A word on blocks vs. sectors...

Filesystems generally access data on disk in terms of **blocks**

But, recall the disk can only be accessed one **sector** at a time

Generally, the FS wants to access multiple sectors at once ...

- Why??

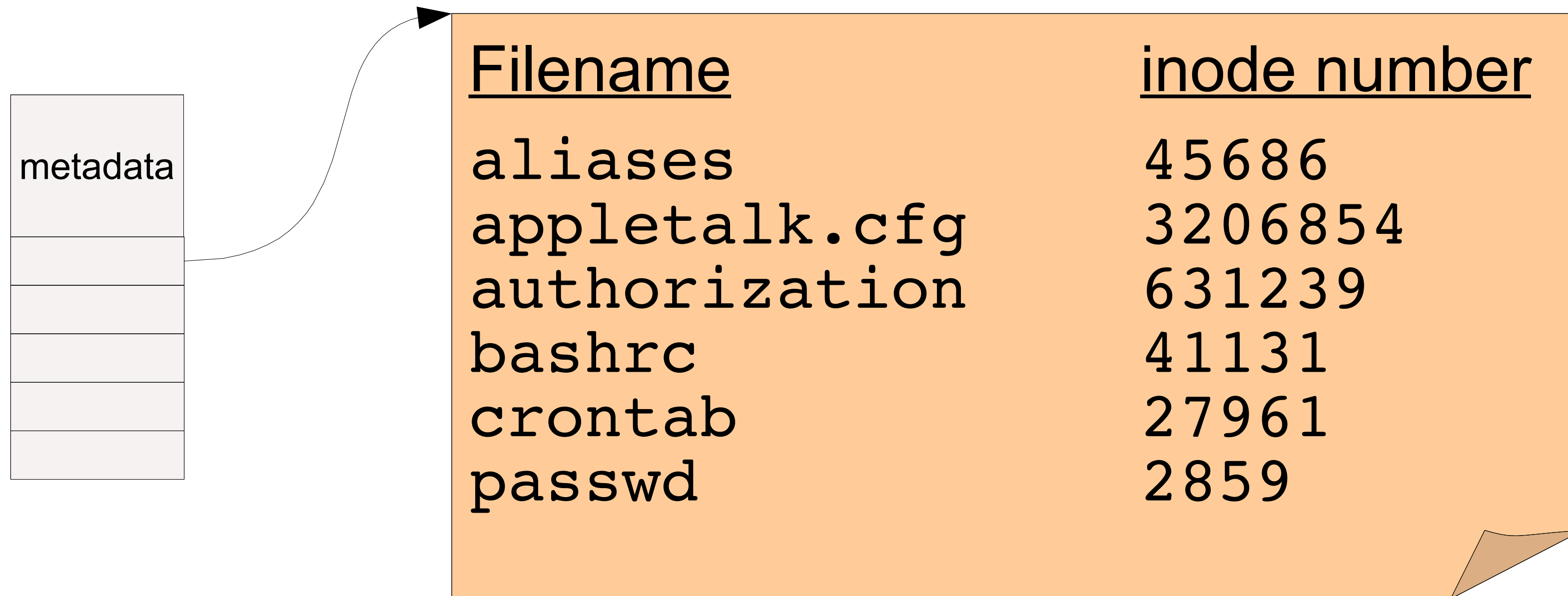
Say sector size is 512 bytes, but filesystem block size is 4 KB.

- This means the block consists of 8 *contiguous* sectors on disk
- Translating from block ID to set of sector IDs is pretty trivial:
    - *sectors(block_id) = { block_id*8 , (block_id*8)+1, ... (block_id*8)+7 }*

© 2007 Matt Welsh – Harvard University

Directories

A directory is a special kind of file that contains a list of *(filename, inode number)* pairs



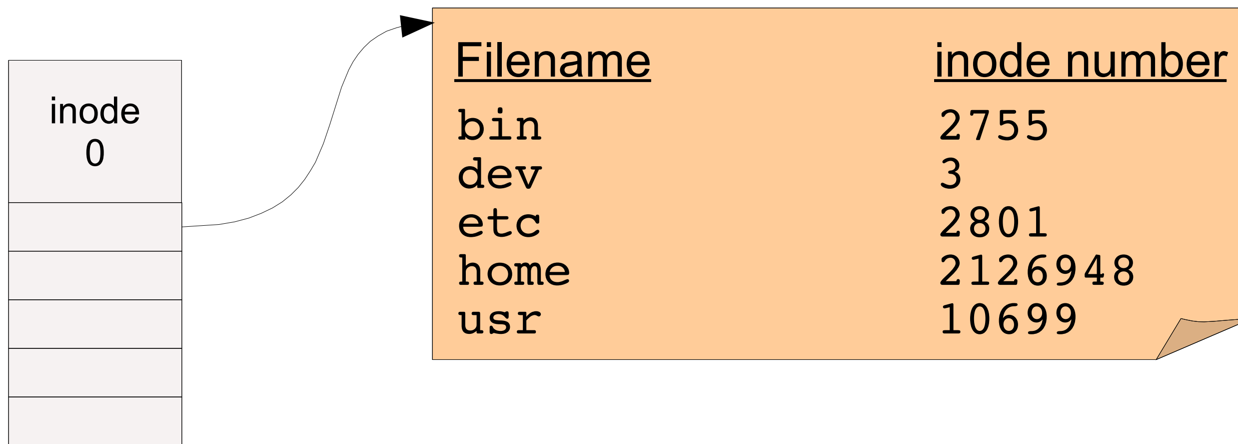
- These are the **contents** of the directory “file data” itself – NOT the directory's inode!
- Filenames (in UNIX) are not stored in the inode at all!

Two open questions:

- How do we find the root directory (“ / “ on UNIX systems)?
- How do we get from an inode number to the location of the inode on disk?

Pathname resolution

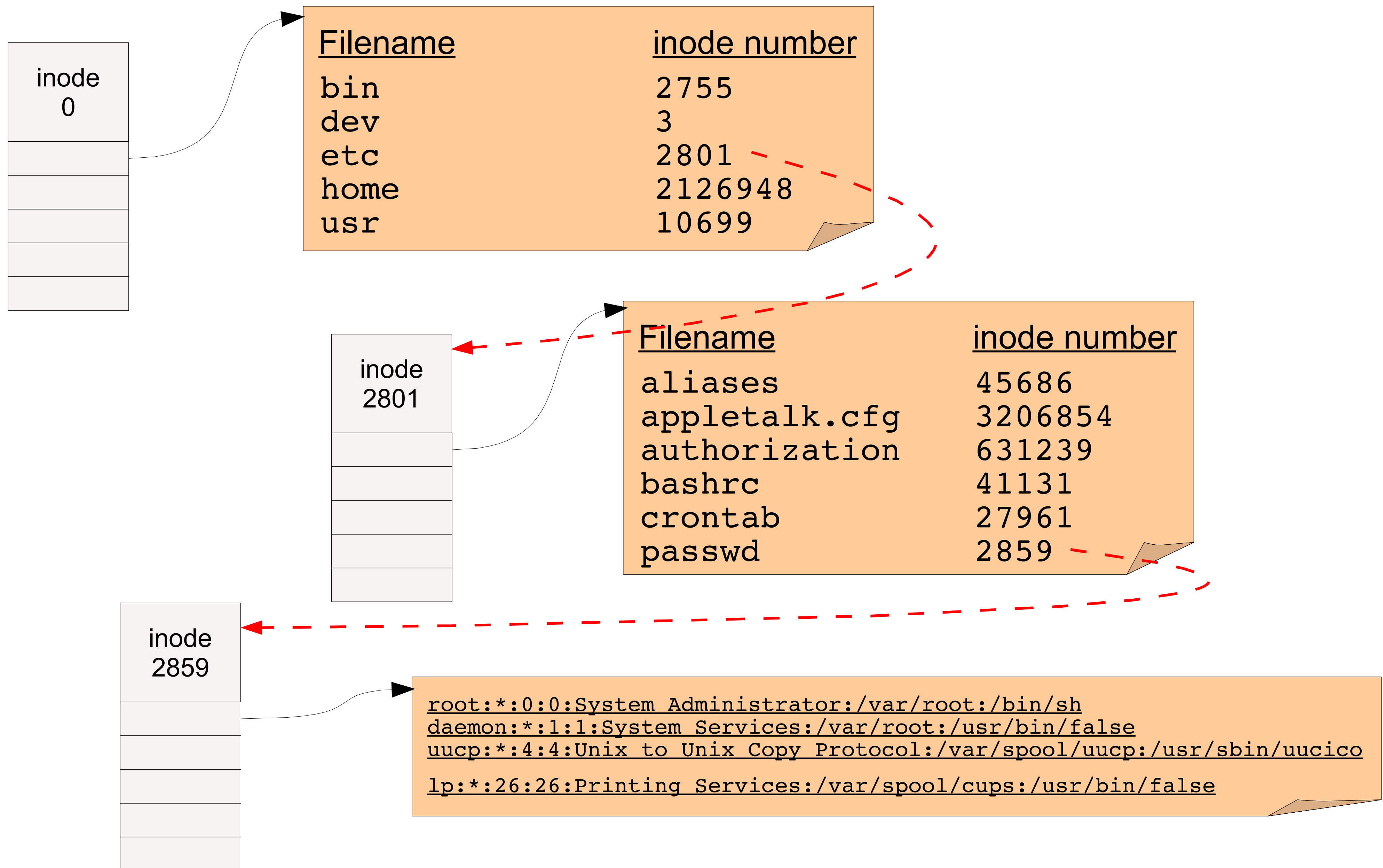
- The root directory is a special inode (usually numbered 0 or 1)



© 2007 Matt Welsh – Harvard University

Pathname resolution

- To look up a pathname "/etc/passwd", start at root directory and walk down chain of inodes...

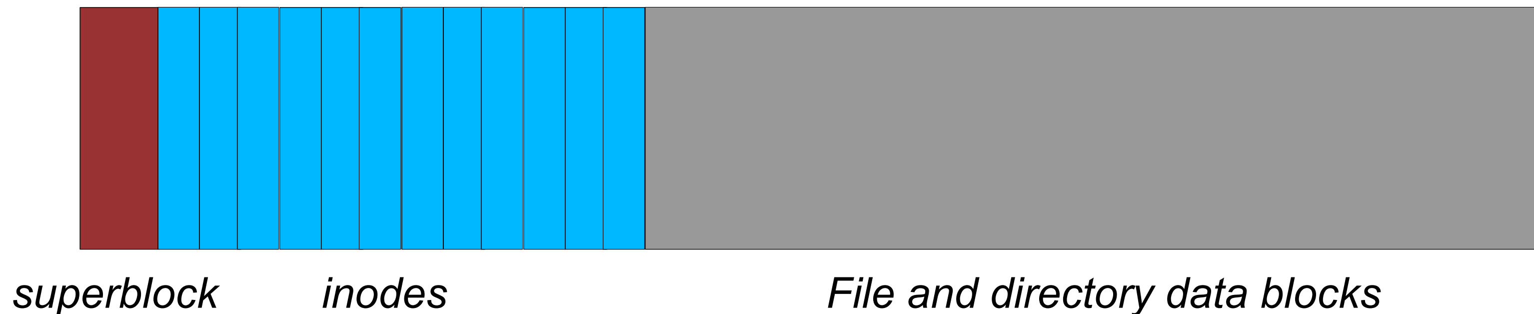


© 2007 Matt Welsh – Harvard University

Locating inodes on disk

All right, so directories tell us the *inode number* of a file.
How the heck do we find the inode itself on disk?

Basic idea: Top part of filesystem contains *all* of the inodes!



- inode number is just the "index" of the inode
- Easy to compute the block address of a given inode:
  - *block_addr(inode_num) = block_offset_of_first_inode + (inode_num * inode_size)*
- This implies that a filesystem has a *fixed* number of potential inodes
  - *This number is generally set when the filesystem is created*

- The **superblock** stores important metadata on filesystem layout, list of free blocks, etc.

© 2007 Matt Welsh – Harvard University

Stupid directory tricks

Directories map filenames to inode numbers. What does this imply?

We can create multiple pointers to the same inode in *different* directories

- Or even the same directory with different filenames

In UNIX this is called a "hard link" and can be done using "ln"

```
bash$ ls -i /home/foo
287663 /home/foo
```
(This is the inode number of "foo")

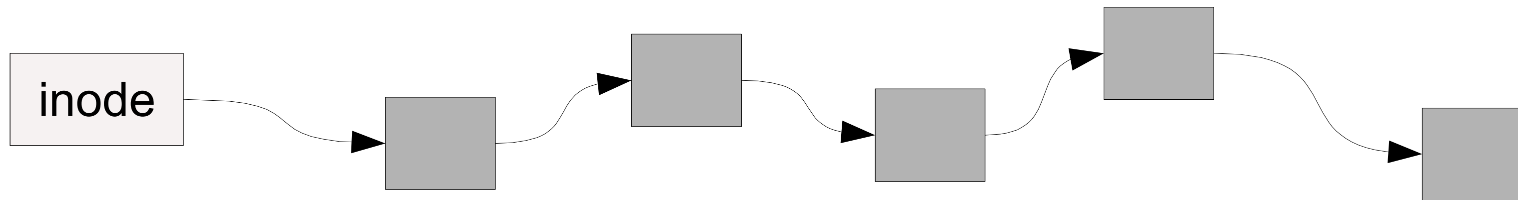
```
bash$ ln /home/foo /tmp/foo

bash$ ls -i /home/foo /tmp/foo
287663 /home/foo
287663 /tmp/foo
```

- "/home/foo" and "/tmp/foo" now refer to the **same file on disk**
  - *Not a copy! You will always see identical data no matter which filename you use to read or write the file.*
- Note: This is not the same as a "symbolic link", which only links one **filename** to another.

How should we organize blocks on disk?

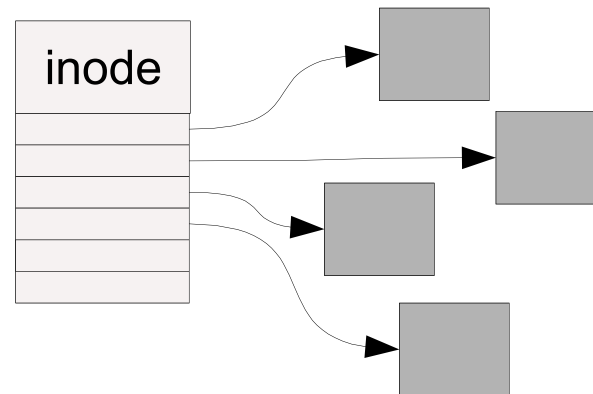
Very simple policy: A file consists of linked blocks

- inode points to the first block of the file
- Each block points to the next block in the file (just a linked list on disk)
    - *What are the advantages and disadvantages??*



Indexed files

- inode contains a list of block numbers containing the file
- Array is allocated when the file is created
    - *What are the advantages and disadvantages??*



© 2007 Matt Welsh – Harvard University

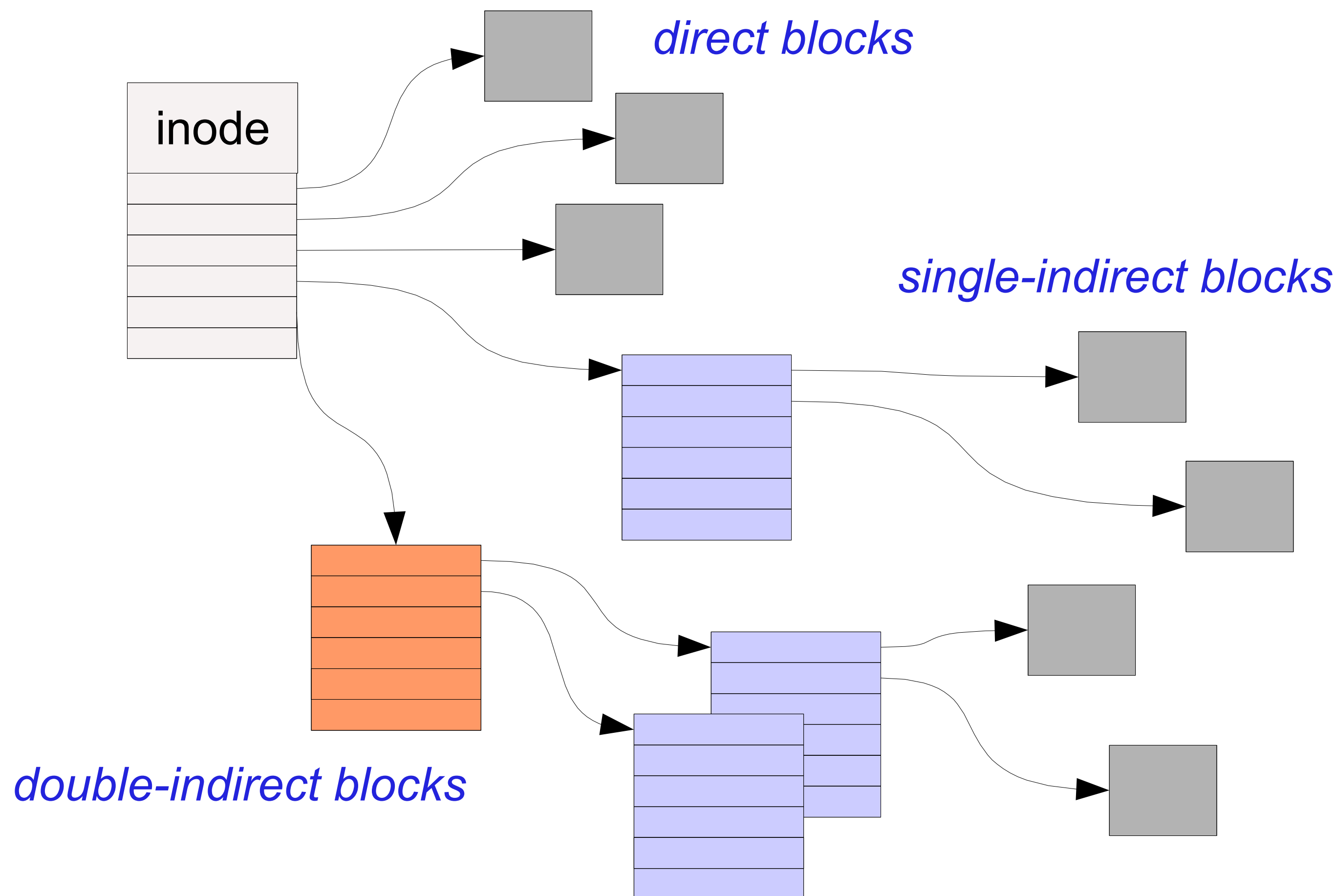
Multilevel Indexed Files

inode contains a list of 10-15 *direct block pointers*

- First few blocks of file can be referred to by the inode itself

inode also contains a pointer to a *single indirect*, *double indirect*, and *triple indirect* blocks

- Allows file to grow to be incredibly large!!!



© 2007 Matt Welsh – Harvard University

File System Caching

Most filesystems cache significant amounts of disk in memory

- e.g., Linux tries to use all "free" physical memory as a giant cache
- Avoids huge overhead for going to disk for every I/O

Application

User

Kernel

Filesystem

Provides illusion of files and and directories

Buffer cache

Maintains memory cache of recently accessed disk blocks

Low-level disk I/O

Performs low-level access to raw disk blocks

© 2007 Matt Welsh – Harvard University

Caching issues

Where should the cache go?

- Below the filesystem layer: Cache individual disk blocks
- Above the filesystem layer: Cache entire files and directories
  - *Which is better??*

Application

User

Kernel

Filesystem — *Provides illusion of files and and directories*

Buffer cache — *Maintains memory cache of recently accessed disk blocks*

Low-level disk I/O — *Performs low-level access to raw disk blocks*

© 2007 Matt Welsh – Harvard University

Caching issues

Where should the cache go?

- Below the filesystem layer: Cache individual disk blocks
- Above the filesystem layer: Cache entire files and directories
  - *Which is better??*

Application

User

Kernel

Filesystem cache — *Maintains memory cache of recently accessed files*

Filesystem — *Provides illusion of files and and directories*

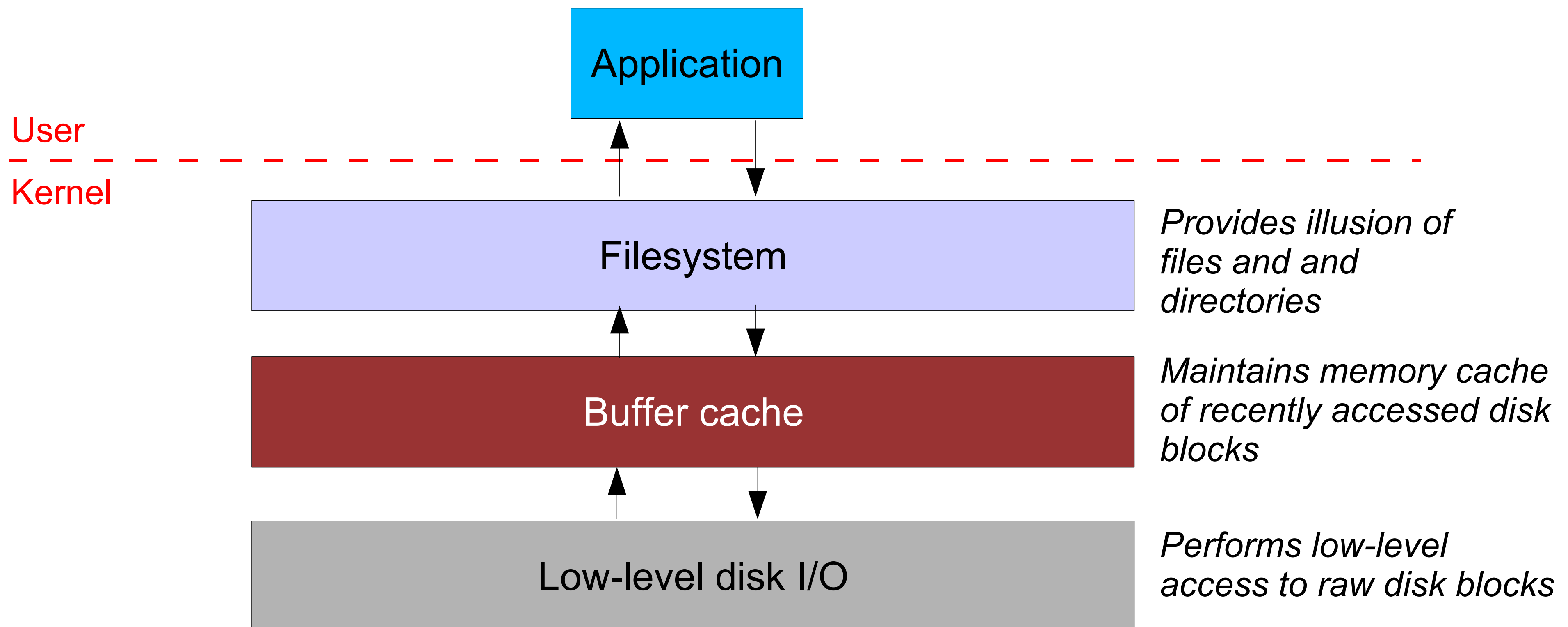
Low-level disk I/O — *Performs low-level access to raw disk blocks*

© 2007 Matt Welsh – Harvard University

Caching issues (2)

Reliability issues

- What happens when you write to the cache but the system crashes?
- What if you update some of the blocks on disk but not others?
  - *Example: Update the inode on disk but not the data blocks?*
- **Write-through cache:** All writes immediately sent to disk
- **Write-back cache:** Cache writes stored in memory until evicted (then written to disk)
  - *Which is better for performance? For reliability?*



© 2007 Matt Welsh – Harvard University

Caching issues (2)

“Syncing” a filesystem writes back any dirty cache blocks to disk

- UNIX “sync” command achieves this.
- Can also use fsync() system call to sync any blocks for a given file.
  - *Warning – not all UNIX systems guarantee that after sync returns that the data has really been written to the disk!*
  - *This is also complicated by memory caching on the disk itself.*

Crash recovery

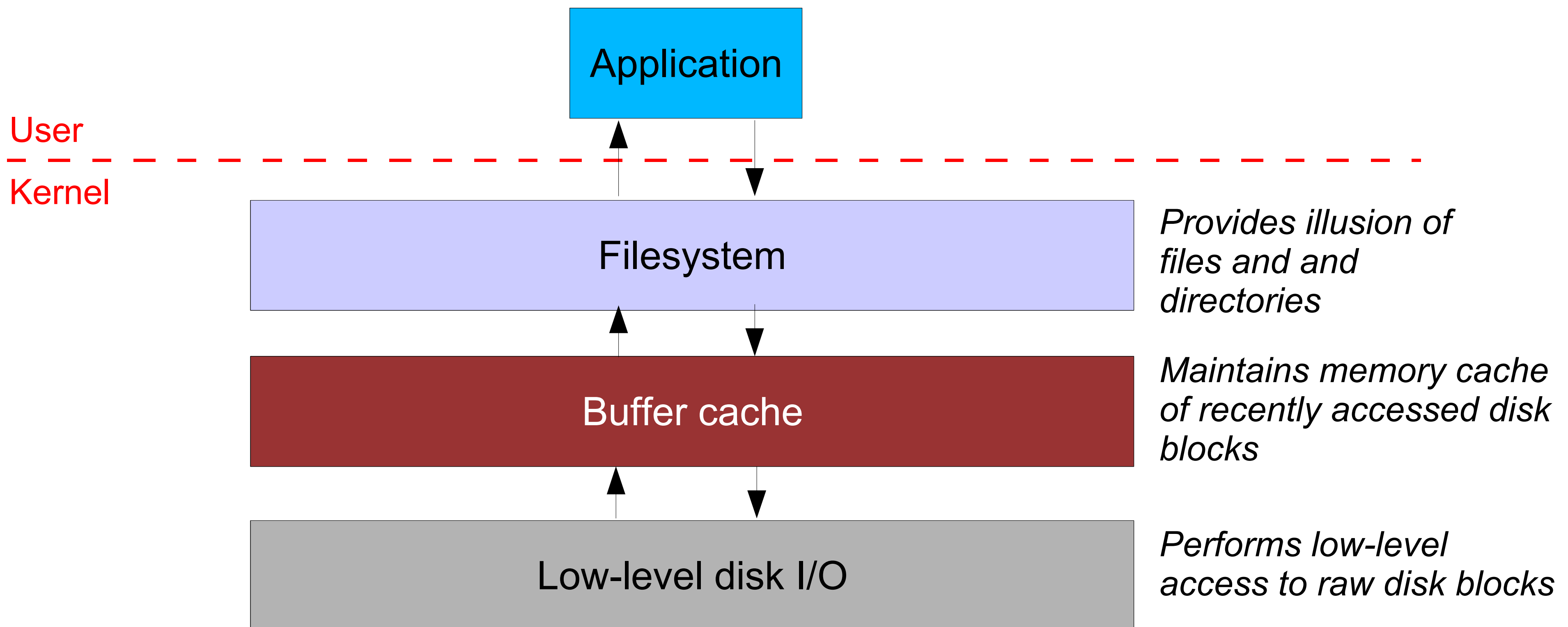
- If system crashes before sync occurs, “fsck” checks the filesystem for errors
- Example: an inode pointing to a block that is marked as free in the free block list
- Another example: An inode with no directory entry pointing to it
  - *These usually get linked into a “lost+found” directory*
  - *inode does not contain the filename so need the sysadmin to look at the file data and guess where it might belong!*

© 2007 Matt Welsh – Harvard University

Caching issues (3)

Read ahead

- Recall: Seek time dominates overhead of disk I/O
- So, would ideally like to read **multiple** blocks into memory when you have a cache miss
  - *Amortize the cost of the seek for multiple reads*
- Useful if file data is laid out in contiguous blocks on disk
  - *Especially if the application is performing sequential access to the file*



© 2007 Matt Welsh – Harvard University