Introduction to File Systems

CS 161: Lecture 12

3/23/17

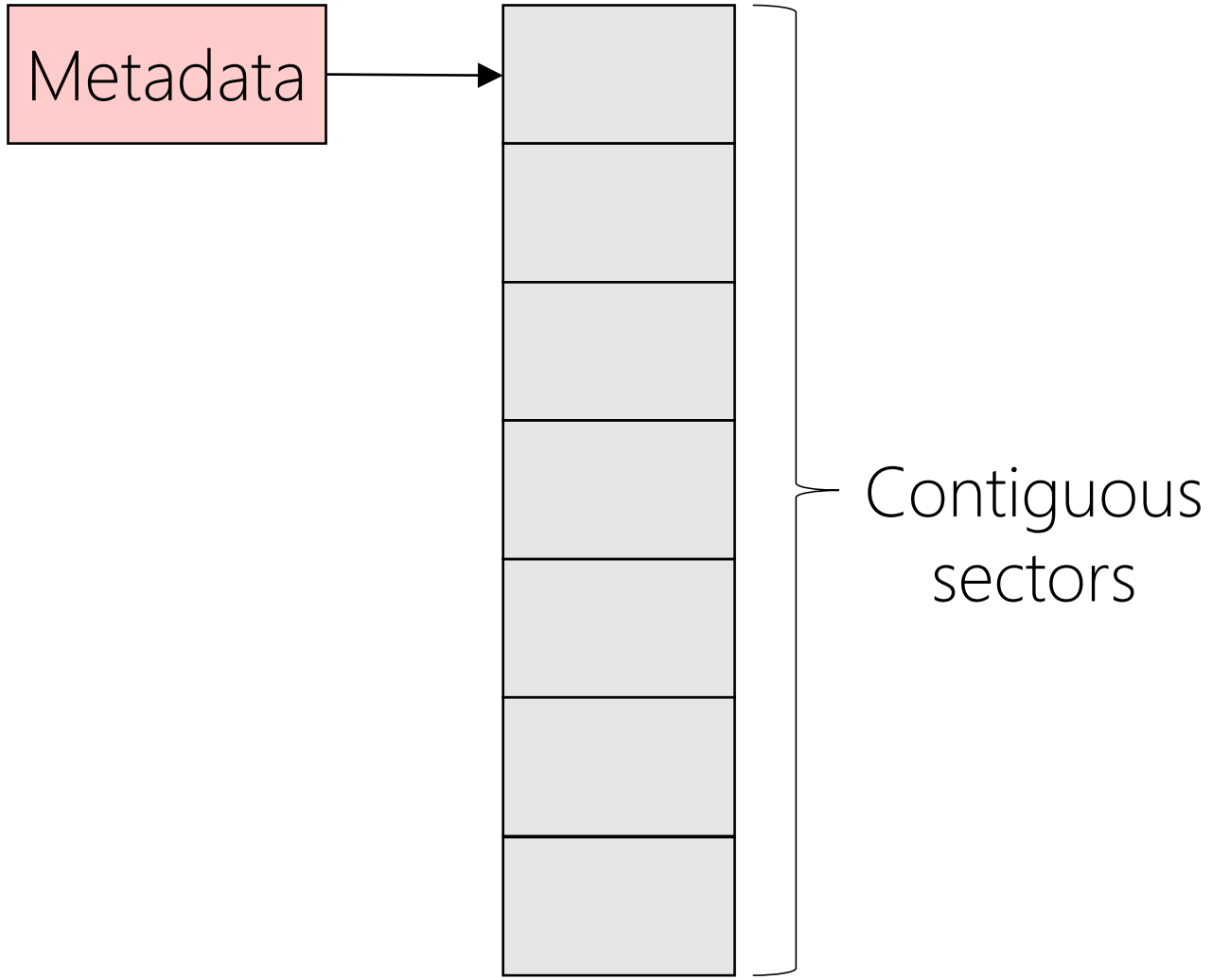
Why Do File Systems Exist?

- From the perspective of an OS, a storage device is a big, linear array of bytes
  - Sector: Smallest amount of data that the device can read or write in a single operation
- Most applications want a higher-level storage abstraction
  - String-based naming of application data (e.g., “photos/koala.jpg” instead of “the bytes between sectors 12,802 and 12,837”)
  - Automatic management of free and allocated sectors as files are created and deleted
  - Performance optimizations like:
    - Caching of recently read/written data
    - Prefetching of preexisting data that is likely to be read in the future
  - Some notion of reliability in the presence of application crashes, OS crashes, and unexpected power failures

Core File System Abstractions: Files and Directories

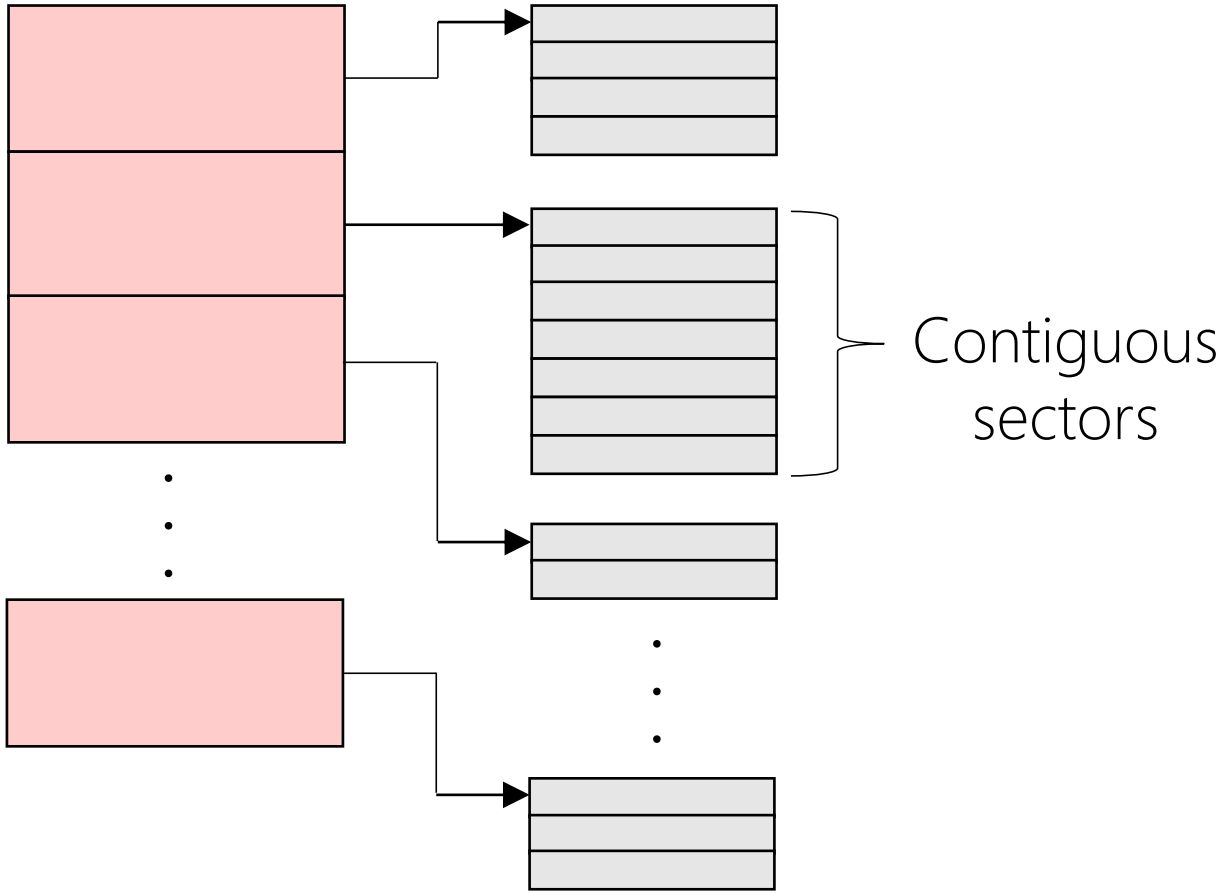
- A file is a named, linear region of bytes that can grow and shrink
  - Associated with metadata like:
    - a user-visible name (e.g., “koala.jpg”)
    - a size in bytes
    - access permissions (read/write/execute)
    - statistics like last modification time
    - a seek position if open
  - File also has a low-level name (e.g., Linux inode number) that the file system uses to locate the file data on a storage device
  - File systems are typically agnostic about the contents of the file (i.e., applications decide how to interpret file bytes)
- A directory is a container for other files and directories
  - Typically contains pairs of <user-visible-name, low-level-name>
  - Nested directories create hierarchical trees (e.g., “/home/todd/photos/koala.jpg”)

Files as Single Extents (1960s File Systems)



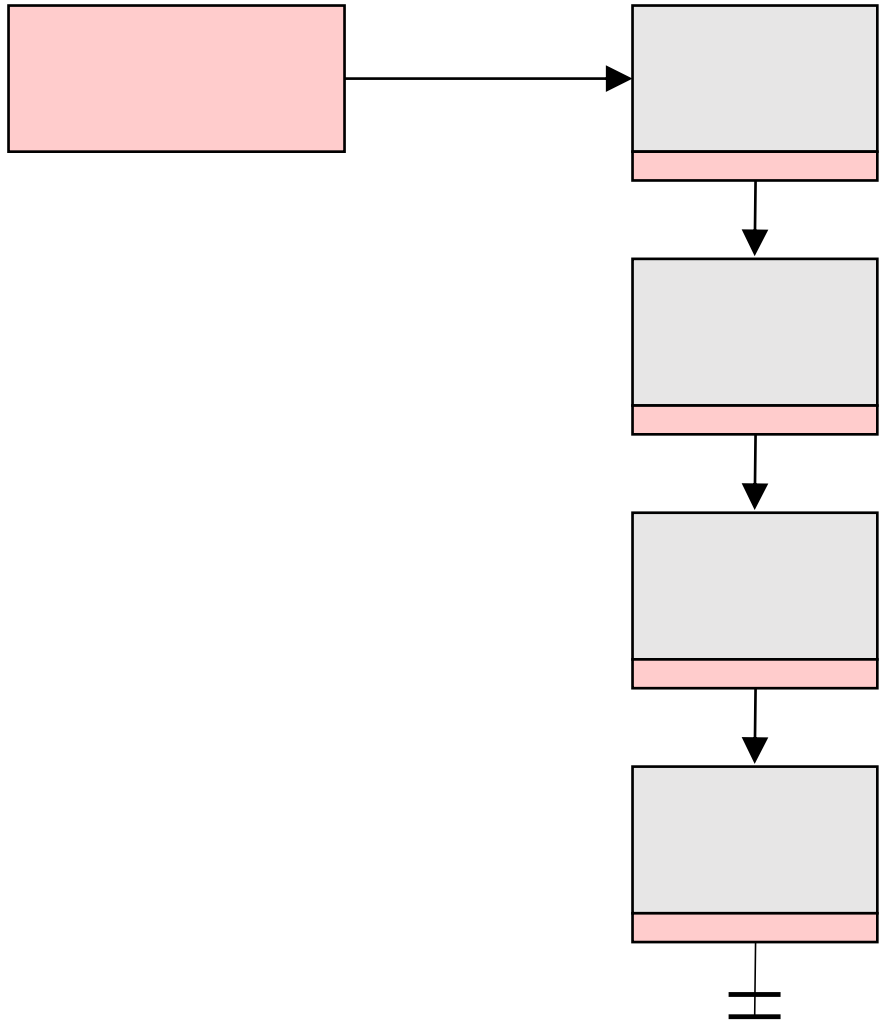
- A file's metadata consists of:
  - A starting sector for the file data
  - The number of bytes in the file (note that the last sector may not be completely full)
- Advantages:
  - Simple!
  - Metadata is small
  - Efficiently supports sequential and random IO
- Disadvantages:
  - For a new file, how much space should be preallocated?
  - What happens if the file needs to grow beyond its allocation, or shrink?
  - External fragmentation

Files as Collections of Extents (IBM OS360, ext4)



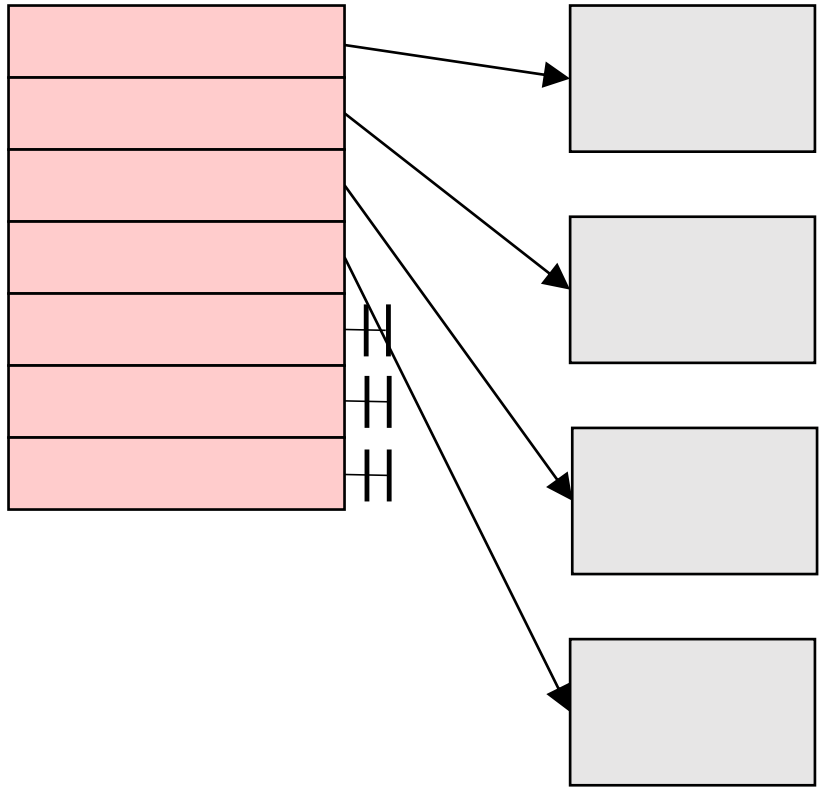
- Advantages:
  - If extents are large, sequential IO almost as fast as with a single extent
  - Random IO almost as efficient as with single extent (offset calculations a little trickier)
  - Metadata is small
- Disadvantages: Most of the single-extent design challenges are multiplied!
  - How large should a file's initial extents be?
  - What happens if a file needs to grow or shrink?
  - External fragmentation not as bad, but depending on design, may have internal fragmentation inside each extent

Files as Linked Lists (FAT: MSDOS, SSD memory cards)



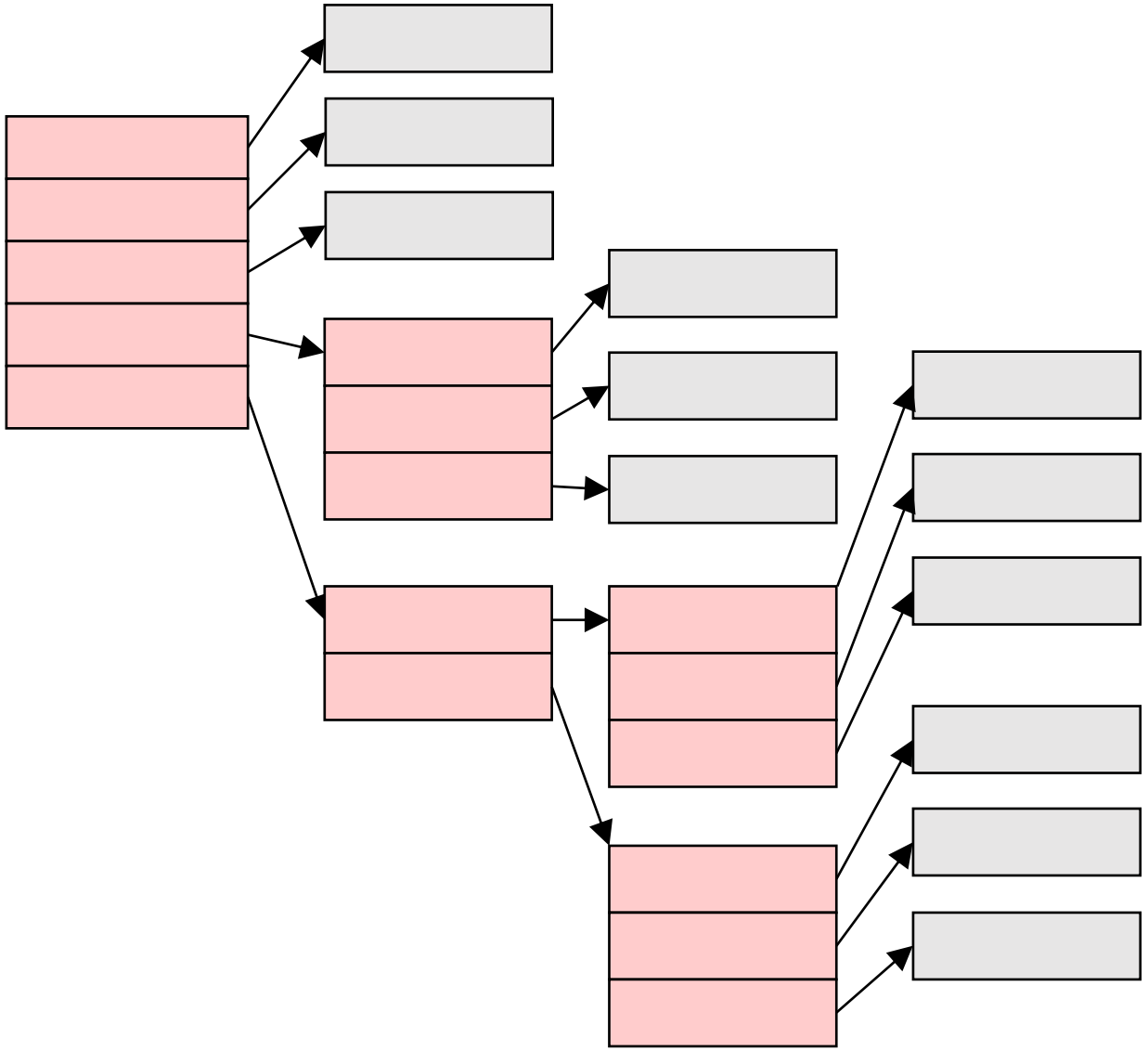
- Advantages
  - Easy to shrink and grow files
  - Low internal and external fragmentation
  - Calculating sector offsets for sequential IO is easy
- Disadvantages
  - Sequential IO requires a lot of seeks
  - Random IO is difficult—where does the target sector live?
  - Must store pointer information at the end of each data block

Files as Flat Indices



- Advantages
  - Easy to calculate offsets for both sequential and random IO
  - Low internal and external fragmentation
- Disadvantages
  - Maximum file size is fixed by the number of entries in an index
  - Sequential IO requires a lot of seeks

Files as Hybrid Indices (FFS, ext2, ext3)



- Top-level index contains direct pointers, indirect pointers, doubly-indirect pointers, etc.
- Advantages:
  - Efficient for small files: don't need to materialize indirect blocks
  - There is still a maximum file size, but it's really big
  - Low internal and external fragmentation
- Disadvantages:
  - Reading or writing a single data block may require multiple disk accesses to fetch indirection info
  - Even if indirection info is cached, reading or writing adjacent blocks may require extra seeks if those blocks are not physically adjacent on disk

Free Space Management

- Fixed-sized blocks: File systems typically use a bitmap to indicate which blocks are in use
  - Allocation metadata is very compact
  - Finding a single free block is straightforward . . .
  - . . . but finding a *contiguous* region of N free blocks is tedious without auxiliary data structures
- Extents: File system can implement "on-disk malloc"
  - File system breaks the disk into discrete-sized extents (e.g., 4KB, 8KB, 12KB,...,4MB), or arbitrarily-sized extents sorted by size
  - File system maintains a list of unallocated extents
  - To allocate an extent for a request of size N bytes, file system uses a policy (e.g., best-fit, worst-fit, first-fit, etc., each of which has trade-offs between internal fragmentation, external fragmentation, and speed of finding a match)

kern/include/kern/sfs.h

```
/* File types for sfi_type */
#define SFS_TYPE_INVAL    0       /* Should not appear on disk */
#define SFS_TYPE_FILE     1
#define SFS_TYPE_DIR      2

/* On-disk inode */
struct sfs_dinode {
    uint32_t sfi_size;                  /* Size of this file (bytes) */
    uint16_t sfi_type;                  /* One of SFS_TYPE_* above */
    uint16_t sfi_linkcount;             /* # hard links to this file */
    uint32_t sfi_direct[SFS_NDIRECT];   /* Direct blocks */
    uint32_t sfi_indirect;              /* Indirect block */
    uint32_t sfi_dindirect;             /* Double indirect block */
    uint32_t sfi_tindirect;             /* Triple indirect block */
    uint32_t sfi_waste[128-5-SFS_NDIRECT]; /* pad to 512 bytes */
};
```

SFS: Managing Free Space

- In SFS, the block size is 512 bytes

```
/* In-memory info for a file system */
struct sfs_fs {
    struct bitmap *sfs_freemap;     /* blocks in use are marked 1 */
    bool sfs_freemapdirty;          /* true if freemap modified */
    struct lock *sfs_freemaplock;   /* lock for freemap/superblock */
    /* Other fields . . . */
};
```

- In SFS, sizeof(struct sfs_dinode) is 512 bytes, which is the block size!
  - So, SFS allows blocks to live anywhere on disk—to allocate/deallocate an inode, SFS manipulates the block bitmap
  - The struct sfs_direntry::sfd_ino field contains a block number (the root directory is always inode 1)
  - SFS differs from most file systems, which place inodes in specific regions of the disk (e.g., inodes blocks should be close to the corresponding file data blocks)

Walking A Directory Path /p0/p1/p2/…

- The inode for the root directory has a fixed, known value
- So, to traverse a directory path, we first set:
  - curr_inode to the root directory's inode
  - pathIndex to 0
  - curr_path to path_components[pathIndex]
- Then, we iteratively:
  - Read the data associated with curr_inode
  - Find the directory entry (i.e., the <path, inode> pair) for which dir_entry.path = curr_path
  - Set curr_inode to dir_entry.inode, and set curr_path to path_components[++i]

More Fun With Directories

- Using full path names can be tedious, so most file systems associate a "current working directory" with each process
  - The current working directory is stored in the struct proc
  - When a process requires the OS to resolve a path, the OS checks whether the first character in the path is "/"
    - If so, start resolving at the root directory
    - If not, start resolving at the current working directory
- Most file systems support the special directory entries "." and ".."
  - "." refers to the directory itself (e.g., "./foo.txt" refers to a file in the directory)
  - ".." refers to the parent directory (e.g., "../foo.txt" refers to a file in the parent directory)

Multiple Directory Entries Can Point To The Same File!

- A “soft link” or “symbolic link” is a file that contains the name of another file
- When the OS encounters a symbolic link, it continues pathname resolution using the path name in the link

```
$ echo “hello” > /target/file
$ ln –s /target/file /path/of/symlink
$ ls –l /target/file /path/of/symlink
  -rw-rw-r--  1 jane admins  6 Mar  22 22:01 /target/file
  lrwxrwxrwx  1 jane admins  6 Mar  22 22:01 /path/of/symlink -> /target/file
```

Multiple Directory Entries Can Point To The Same File!

- A “hard link” directly references an inode number
  - The file system maintains a reference count for each file
  - When you hard link to a file, you increment the ref count
  - When you delete a hard link, you remove the link from its directory, and decrement the ref count for the file
  - A file’s data is only deleted when its ref count drops to zero

```
$ echo “hello” > /target/file
$ ln /target/file /path/of/hardlink
$ ls –l /target/file /path/of/hardlink
  -rw-rw-r--  2 jane admins  6 Mar  22 22:01 /target/file
  -rw-rw-r--  2 jane admins  6 Mar  22 22:01 /path/of/hardlink
```

The Virtual File System (VFS) Interface

- In The Olden Days, a particular OS could only use a single, baked-in file system
- A VFS defines an abstract, generic interface that a file system should present to the OS
  - A particular file system implements the abstract VFS methods, and the OS only interacts with the file system through those VFS methods
  - In principle, the core OS doesn't need to know anything about the internal implementation of the file system!
- A VFS makes it easy for a single OS to run one (or more!) file systems of the user's choice
  - Ex: A Linux machine might simultaneously use ext3 for locally storing files, and NFS for storing files on remote servers

OS161's VFS: kern/include/vfs.h

```
/* Abstract low-level file. */
struct vnode {
  int vn_refcount;                /* Reference count */
  struct spinlock vn_countlock;   /* Lock for vn_refcount */
  struct fs *vn_fs;               /* Filesystem vnode belongs to */
  void *vn_data;                  /* Filesystem-specific data */
  const struct vnode_ops *vn_ops; /* Functions on this vnode */
};

struct vnode_ops {
  int (*vop_read)(struct vnode *file, struct uio *uio);
  int (*vop_write)(struct vnode *file, struct uio *uio);
  int (*vop_stat)(struct vnode *object, struct stat *statbuf);
  //...other vops...
};
```