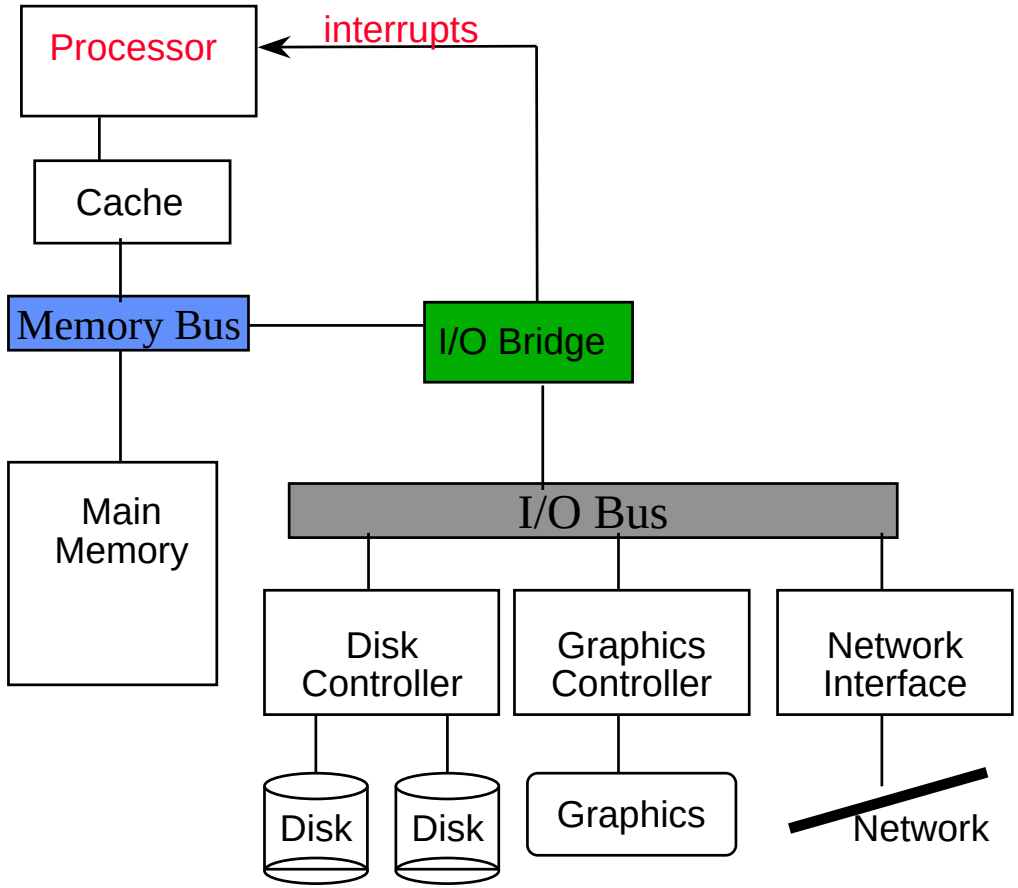
# File Systems and Disk Layout

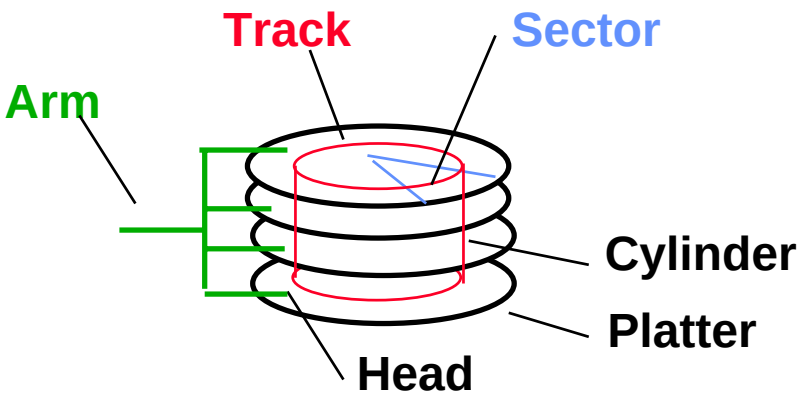
Dr. Daniel Andresen

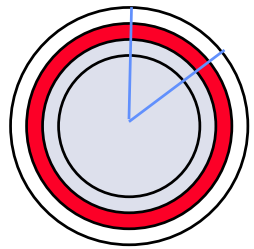
CIS520 – Operating Systems

# I/O: The Big Picture



# Rotational Media



Access time = seek time + rotational delay + transfer time

***seek time*** = 5-15 milliseconds to move the disk arm and settle on a cylinder
***rotational delay*** = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
***transfer time*** = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

# Disks and Drivers

Disk hardware and driver software provide basic facilities for nonvolatile secondary storage (*block devices*).

1. OS views the block devices as a collection of *volumes*.

   A logical volume may be a *partition* of a single disk or a *concatenation* of multiple physical disks (e.g., RAID).

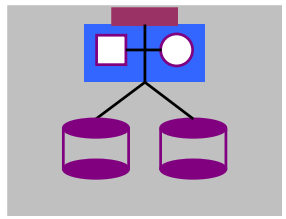
2. OS accesses each volume as an array of fixed-size *sectors*.

   Identify sector (or block) by unique *(volumeID, sector ID)*.

   *Read/write* operations DMA data to/from physical memory.

3. Device interrupts OS on I/O completion.

   ISR wakes up process, updates internal records, etc.



# Using Disk Storage

Typical operating systems use disks in three different ways:

1. System calls allow user programs to access a “raw” disk.

   Unix: special *device file* identifies volume directly.

   Any process that can *open* the device file can read or write any specific sector in the disk volume.

2. OS uses disk as *backing storage* for virtual memory.

   OS manages volume transparently as an “overflow area” for VM contents that do not “fit” in physical memory.

3. OS provides syscalls to create/access *files* residing on disk.

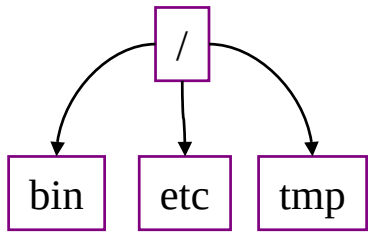
   OS *file system* modules virtualize physical disk storage as a collection of logical files.

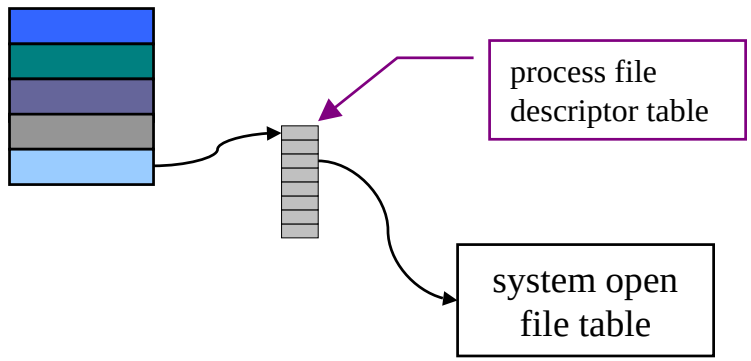
# Unix File Syscalls

```
int fd;                    /* file descriptor */
fd = open("/bin/sh", O_RDONLY, 0);
fd = creat("/tmp/zot", 0777);
unlink("/tmp/zot");

char data[bufsize];
bytes = read(fd, data, count);
bytes = write(fd, data, count);
lseek(fd, 50, SEEK_SET);

mkdir("/tmp/dir", 0777);
rmdir("/tmp/dir");
```



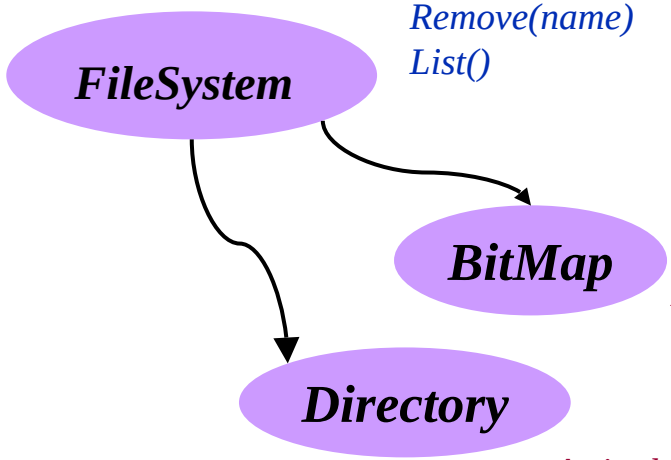
# Nachos File Syscalls/Operations

```
Create("zot");

OpenFileId fd;
fd = Open("zot");
Close(fd);

char data[bufsize];
Write(data, count, fd);
Read(data, count, fd);
```

*FileSystem class internal methods:*
*Create(name, size)*
*OpenFile = Open(name)*
*Remove(name)*
*List()*




*Bitmap indicates whether each disk block is in-use or free.*

*A single 10-entry directory stores names and disk locations for all currently existing files.*

Limitations:

1. small, fixed-size files and directories
2. single disk with a single directory
3. stream files only: no seek syscall
4. file size is specified at creation time
5. no access control, etc.

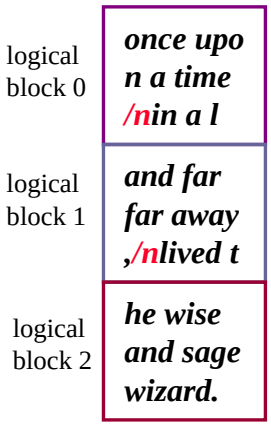
FileSystem data structures reside on-disk, but file system code always operates on a cached *copy* in memory (read/modify/write).

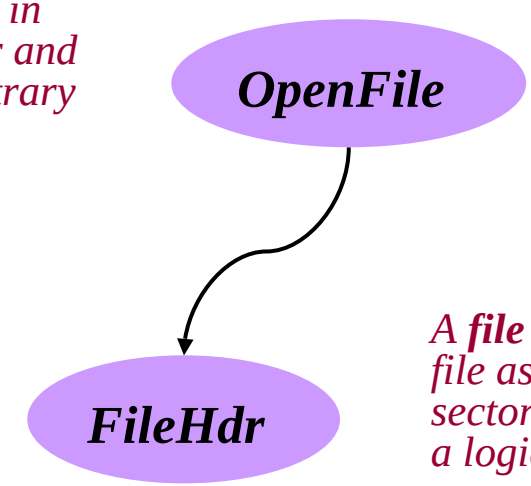
# Preview of Issues for File Systems

1. Buffering disk data for access from the processor.
   - block I/O (DMA) must use aligned, physically resident buffers
   - block update is a read-modify-write
2. Creating/representing/destroying independent files.
   - disk block allocation, file block map structures
   - directories and symbolic naming
3. Masking the high seek/rotational latency of disk access.
   - smart block allocation on disk
   - block caching, read-ahead (prefetching), and write-behind
4. Reliability and the handling of updates.

# Representing a File On-Disk in Nachos

*An **OpenFile** represents a file in active use, with a seek pointer and read/write primitives for arbitrary byte ranges.*

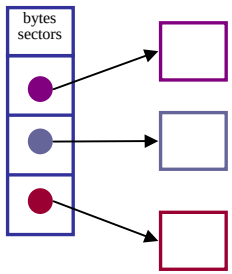



*OpenFile(sector)*
*Seek(offset)*
*Read(char* data, bytes)*
*Write(char* data, bytes)*

*A **file header** describes an on-disk file as an ordered sequence of sectors with a length, mapped by a logical-to-physical block map.*

*OpenFile* ofd = filesys->Open("tale");*
*ofd->Read(data, 10) gives 'once upon '*
*ofd->Read(data, 10) gives 'a time/nin '*




*Allocate(...,filesize)*
*length = FileLength()*
*sector = ByteToSector(offset)*

# File Metadata

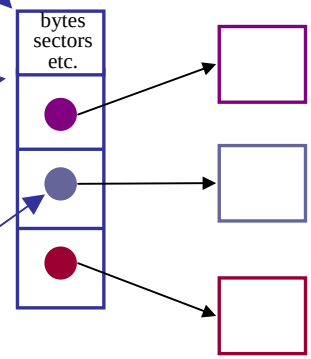
On disk, each file is represented by a *FileHdr* structure.

The *FileHdr* object is an in-memory *copy* of this structure.

*file attributes*: may include owner, access control, time of create/modify/access, etc.

logical-physical block map (like a translation table)

physical block pointers in the block map are sector IDs




The *FileHdr* is a file system "bookeeping" structure that supplements the file data itself: these kinds of structures are called filesystem *metadata*.

A Nachos *FileHdr* occupies exactly one disk sector.

To operate on the file (e.g., to open it), the *FileHdr* must be read into memory.

Any changes to the attributes or block map must be written back to the disk to make them permanent.

```
FileHdr* hdr = new FileHdr();
hdr->FetchFrom(sector)
hdr->WriteBack(sector)
```

# Representing Large Files

The Nachos *FileHdr* occupies exactly one disk sector, limiting the maximum file size.

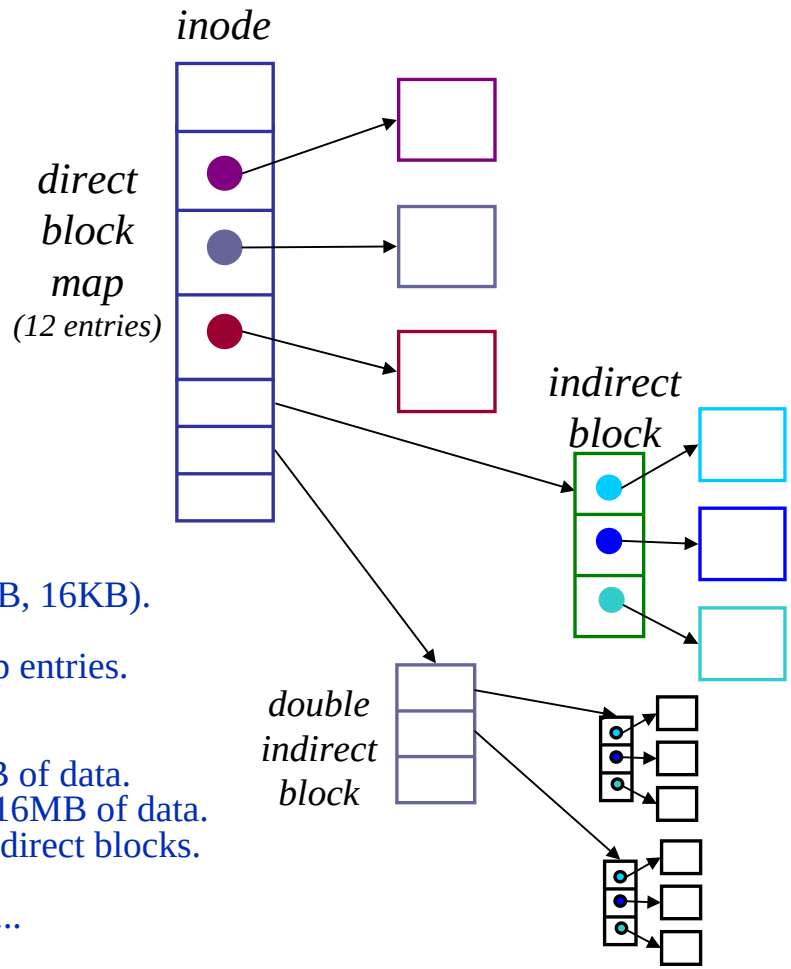
sector size = 128 bytes
120 bytes of block map = 30 entries
each entry maps a 128-byte sector
max file size = 3840 bytes

In Unix, the *FileHdr* (called an index-node or *inode*) represents large files using a hierarchical block map.

Each file system block is a clump of sectors (4KB, 8KB, 16KB).
Inodes are 128 bytes, packed into blocks.
Each inode has 68 bytes of attributes and 15 block map entries.

suppose block size = 8KB
12 direct block map entries in the inode can map 96KB of data.
One indirect block (referenced by the inode) can map 16MB of data.
One double indirect block pointer in inode maps 2K indirect blocks.

maximum file size is 96KB + 16MB + (2K*16MB) + ...



# Representing Small Files

Internal fragmentation in the file system blocks can waste significant space for small files.

E.g., 1KB files waste 87% of disk space (and bandwidth) in a naive file system with an 8KB block size.

Most files are small: one study [Irlam93] shows a median of 22KB.

FFS solution: optimize small files for space efficiency.

- Subdivide blocks into 2/4/8 *fragments* (or just *frags*).
- Free block maps contain one bit for each fragment.

  To determine if a block is free, examine bits for all its fragments.
- The last block of a small file is stored on fragment(s).

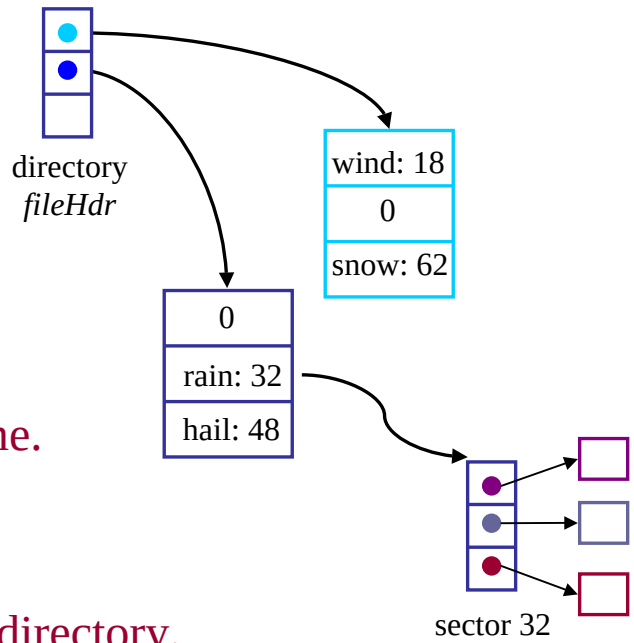
  If multiple fragments they must be contiguous.

# Basics of Directories

A *directory* is a set of file names, supporting lookup by symbolic name.

In Nachos, each directory is a file containing a set of mappings from name->*FileHdr*.

*Directory(entries)*
*sector = Find(name)*
*Add(name, sector)*
*Remove(name)*




Each directory entry is a fixed-size slot with space for a *FileNameMaxLen* byte name.

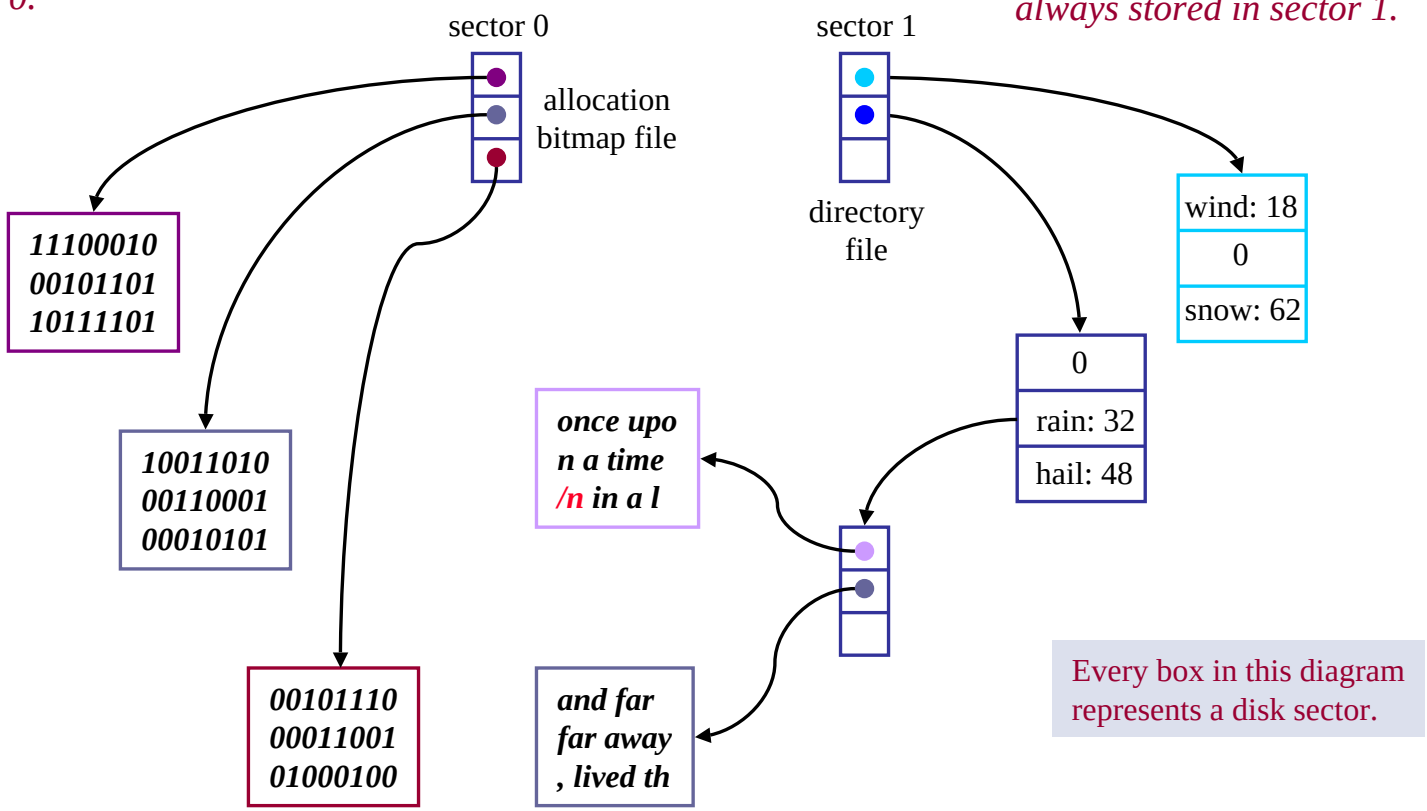
*Entries or slots are found by a linear scan.*

A directory entry may hold a pointer to another directory, forming a hierarchical name space.

# A Nachos Filesystem On Disk

*An allocation bitmap file maintains free/allocated state of each physical block; its **FileHdr** is always stored in sector 0.*

*A directory maintains the name->FileHdr mappings for all existing files; its **FileHdr** is always stored in sector 1.*



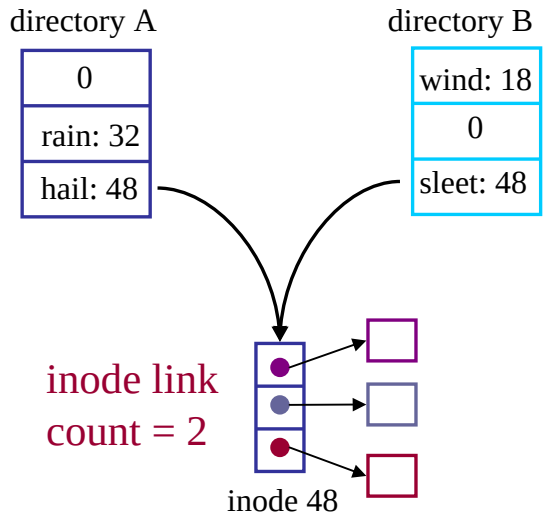

Every box in this diagram represents a disk sector.

# Unix File Naming (Hard Links)

A Unix file may have multiple names.

Each directory entry naming the file is called a *hard link*.

Each inode contains a *reference count* showing how many hard links name it.



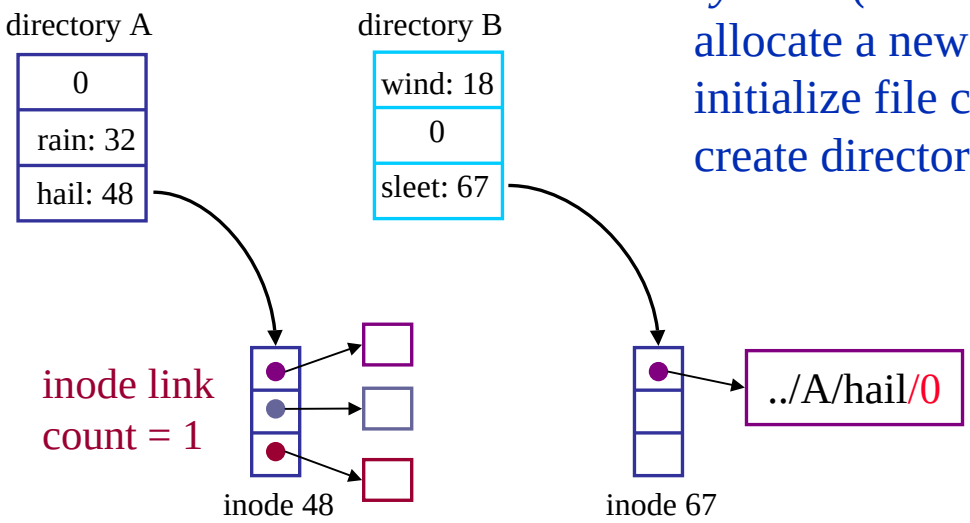

*link* system call
*link (existing name, new name)*
create a new name for an existing file
increment inode link count

*unlink* system call ("remove")
*unlink(name)*
destroy directory entry
decrement inode link count
if count = 0 and file is not in active use
free blocks (recursively) and on-disk inode

# Unix Symbolic (Soft) Links

Unix files may also be named by *symbolic (soft) links*.

- A soft link is a file containing a pathname of some other file.




*symlink* system call
*symlink (existing name, new name)*
allocate a new file (inode) with type symlink
initialize file contents with *existing name*
create directory entry for new file with *new name*

The target of the link may be removed at any time, leaving a dangling reference.

How should the kernel handle recursive soft links?

# The Problem of Disk Layout

The level of indirection in the file block maps allows flexibility in file layout.

"File system design is 99% block allocation." [McVoy]

Competing goals for block allocation:

- *allocation cost*
- *bandwidth* for high-volume transfers
- *stamina*
- *efficient directory operations*

**Goal:** reduce disk arm movement and seek overhead.

metric of merit: *bandwidth utilization*

# FFS and LFS

We will study two different approaches to block allocation:

- Cylinder groups in the Fast File System (FFS) [McKusick81]

  clustering enhancements [McVoy91], and improved cluster allocation [McKusick: Smith/Seltzer96]

  FFS can also be extended with metadata logging [e.g., Episode]

- Log-Structured File System (LFS)

  proposed in [Douglis/Ousterhout90]

  implemented/studied in [Rosenblum91]

  BSD port, sort of maybe: [Seltzer93]

  extended with self-tuning methods [Neefe/Anderson97]

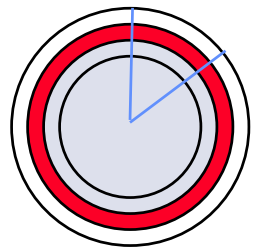
- Other approach: *extent-based* file systems

# FFS Cylinder Groups

FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.

- typical: thousands of cylinders, dozens of groups
- Strategy: place "related" data blocks in the same cylinder group whenever possible.

  seek latency is proportional to seek distance



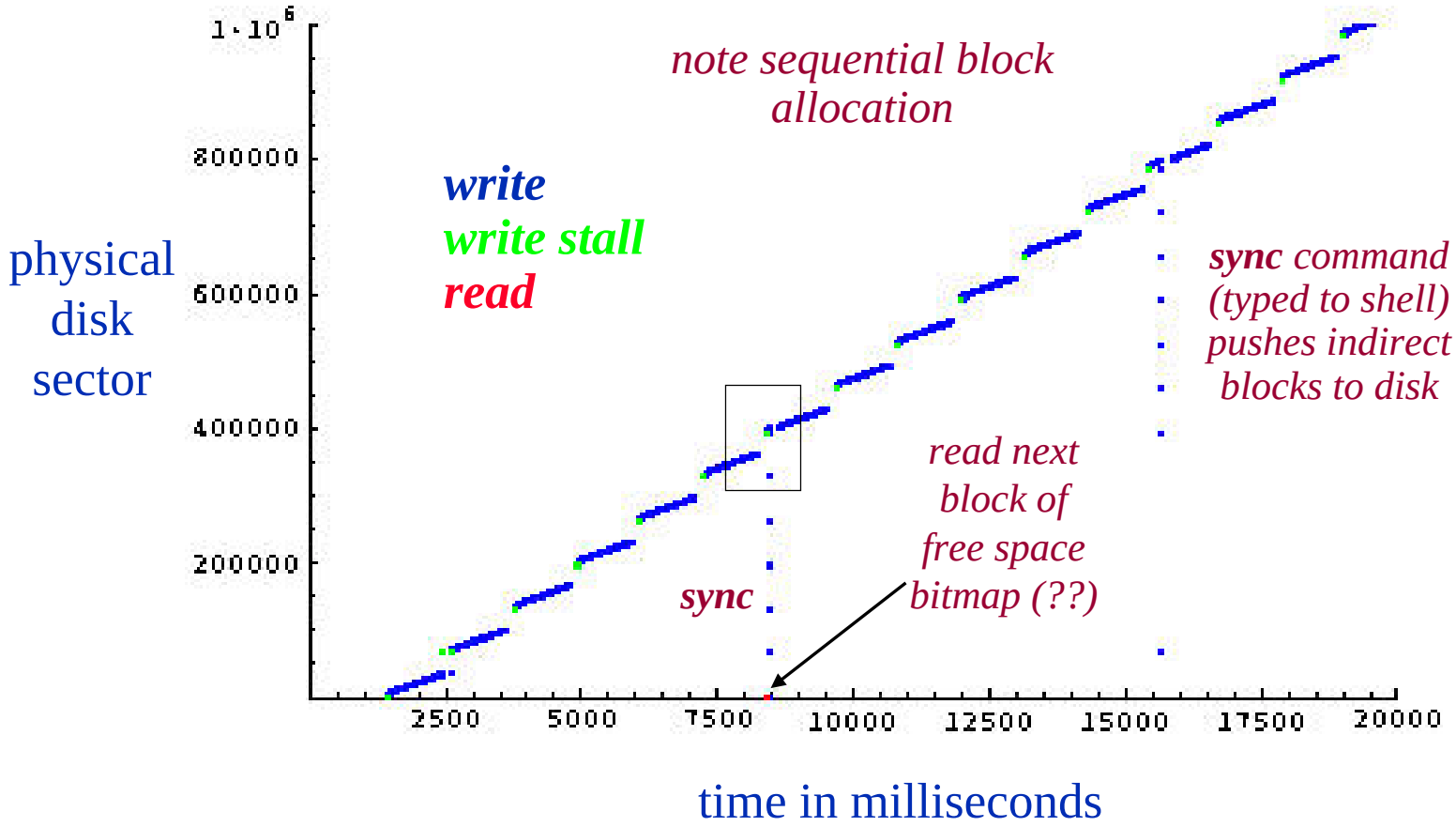
- Smear large files across groups:

  Place a run of contiguous blocks in each group.

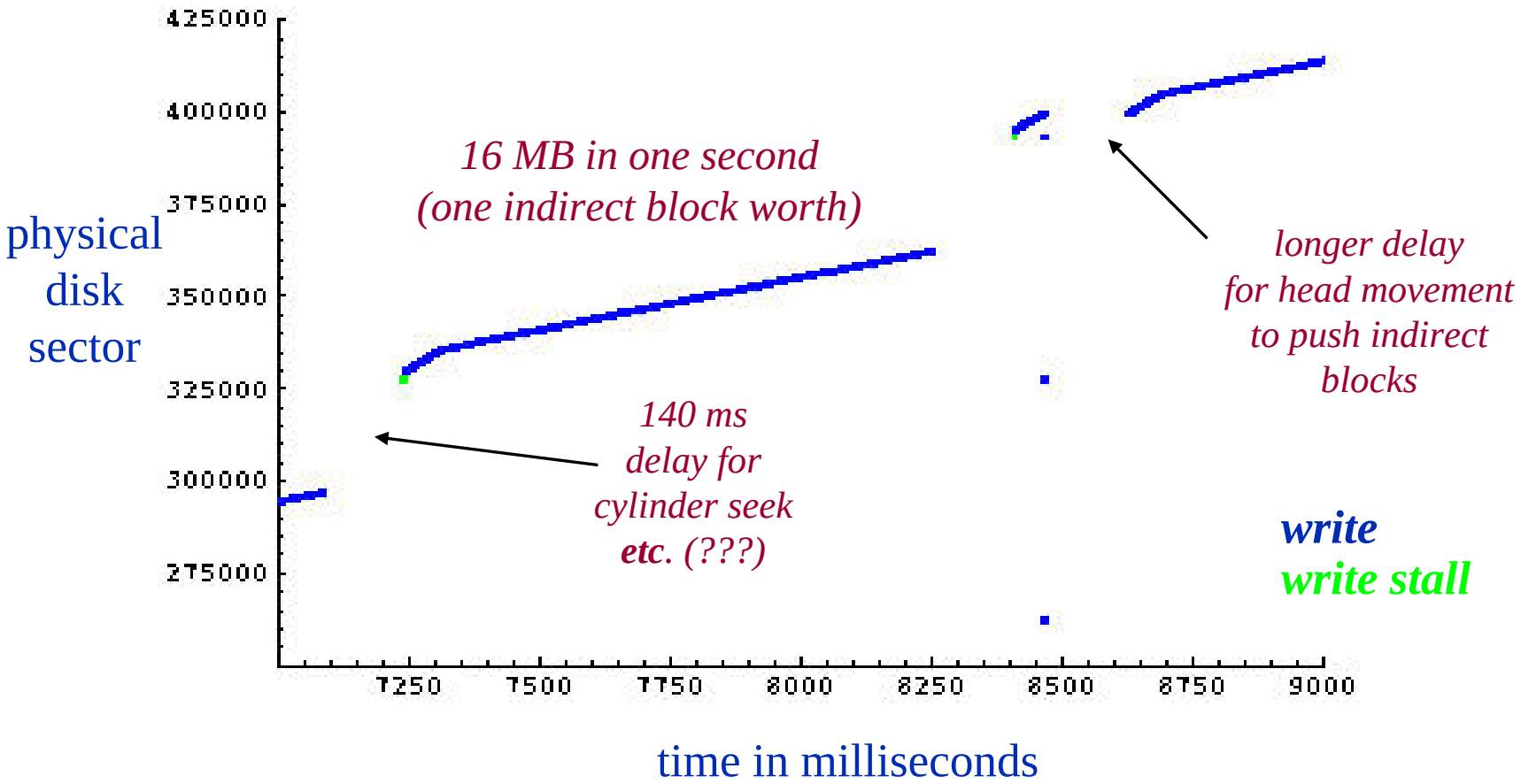
- Reserve inode blocks in each cylinder group.

  This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).

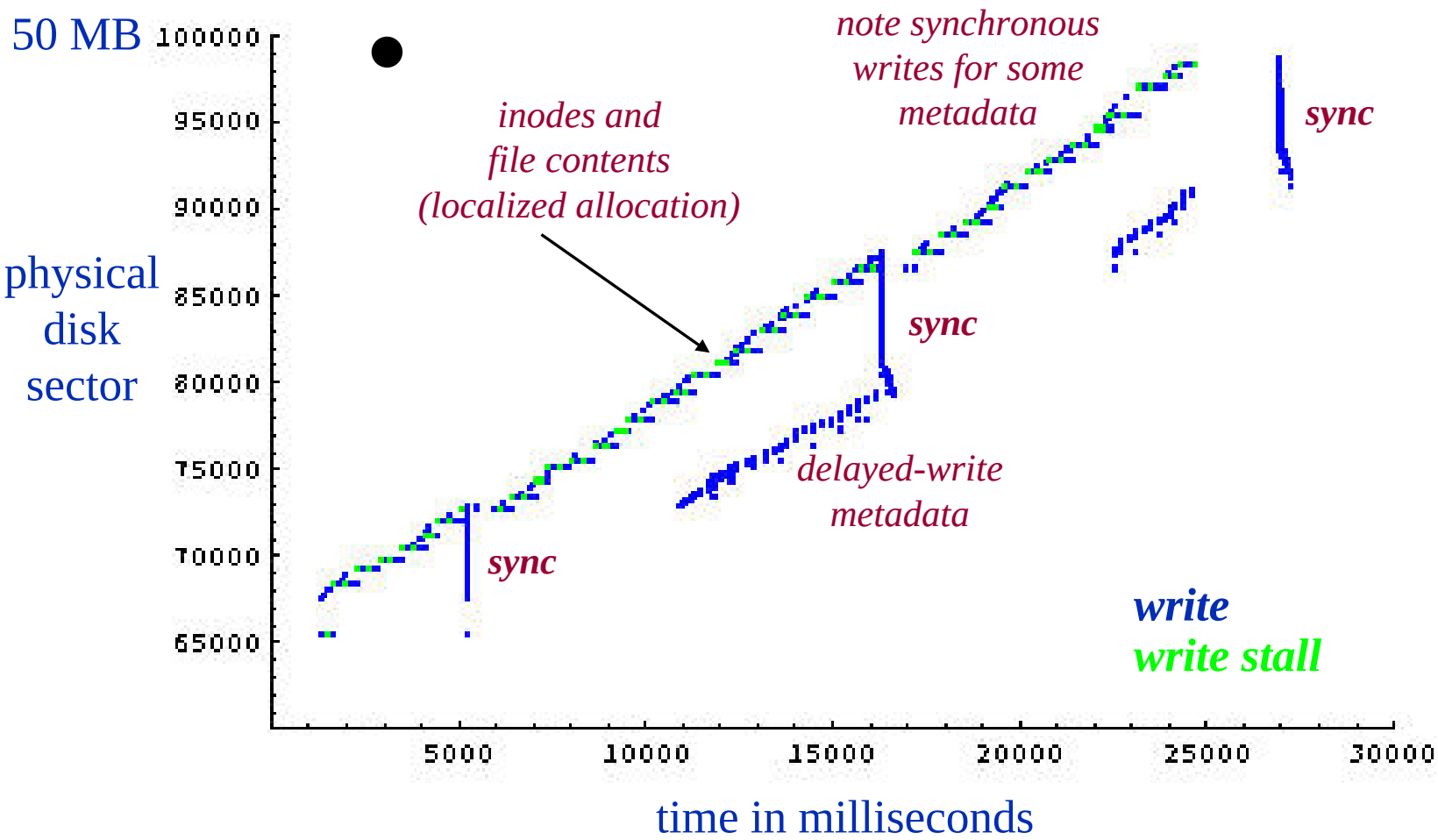
# Sequential File Write



# Sequential Writes: A Closer Look



# Small-File Create Storm



# Alternative Structure: DOS FAT