

CS162
Operating Systems and
Systems Programming
Lecture 19

File Systems (Con't),
MMAP

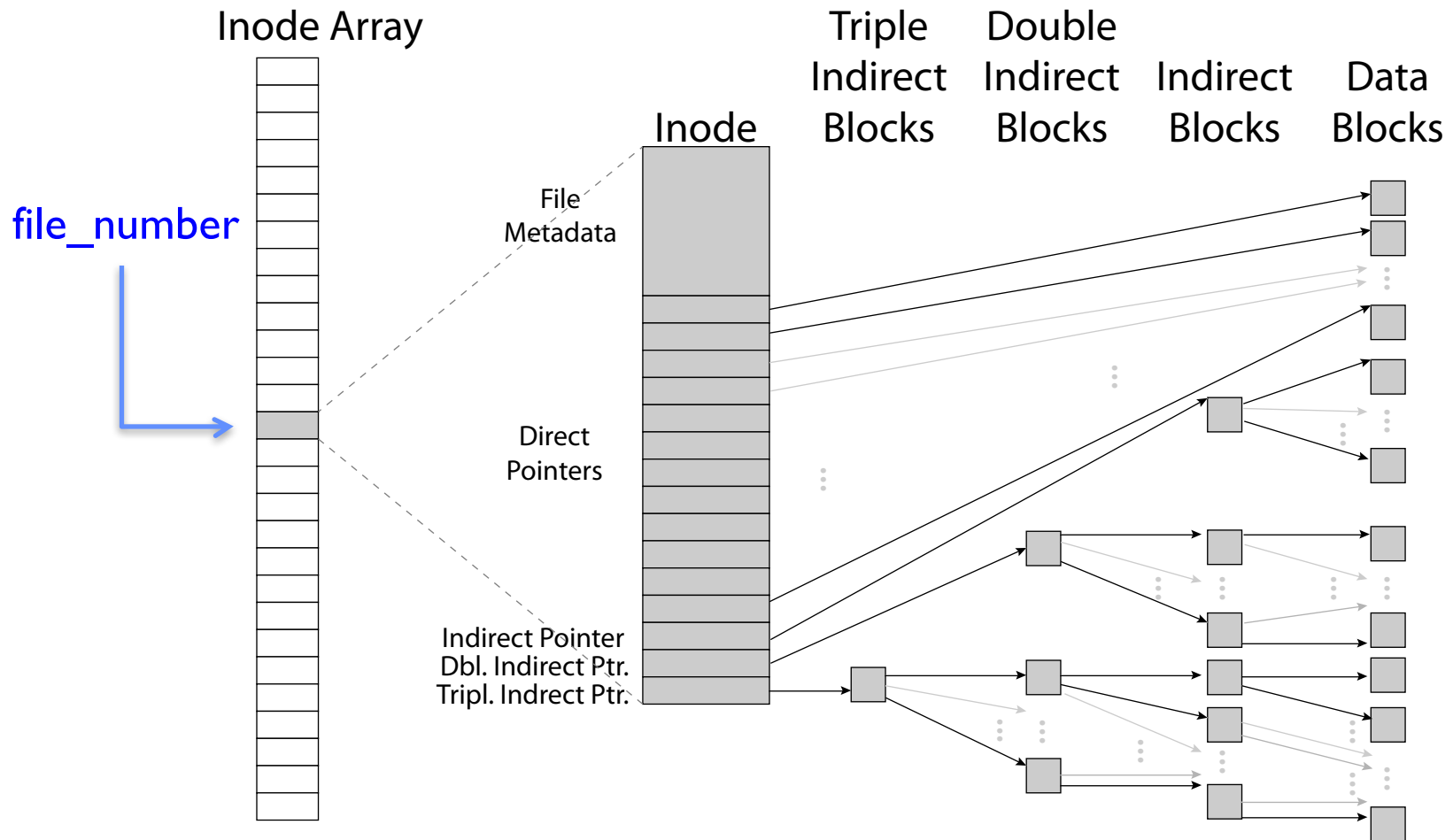
October 1st, 2017

Prof. Ion Stoica

<http://cs162.eecs.Berkeley.edu>

So What About a “Real” File System?

- Meet the inode:



An “Almost Real” File System

- Pintos: `src/filesys/file.c`, `inode.c`

```
/* An open file. */
struct file
{
    struct inode *inode;           /* File's inode. */
    off_t pos;                     /* Current position. */
    bool deny_write;               /* Has file_deny_write() been called? */
};
```

Direct Data
Blocks Blocks

me_num

```
/* In-memory inode. */
struct inode
{
    struct list_elem elem;         /* Element in inode list. */
    block_sector_t sector;         /* Sector number of disk location. */
    int open_cnt;                  /* Number of openers. */
    bool removed;                  /* True if deleted, false otherwise. */
    int deny_write_cnt;            /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;        /* Inode content. */
};
```

In
D
Tr

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
{
    block_sector_t start;          /* First data sector. */
    off_t length;                  /* File size in bytes. */
    unsigned magic;                /* Magic number. */
    uint32_t unused[125];          /* Not used. */
};
```

Unix File System (1/2)

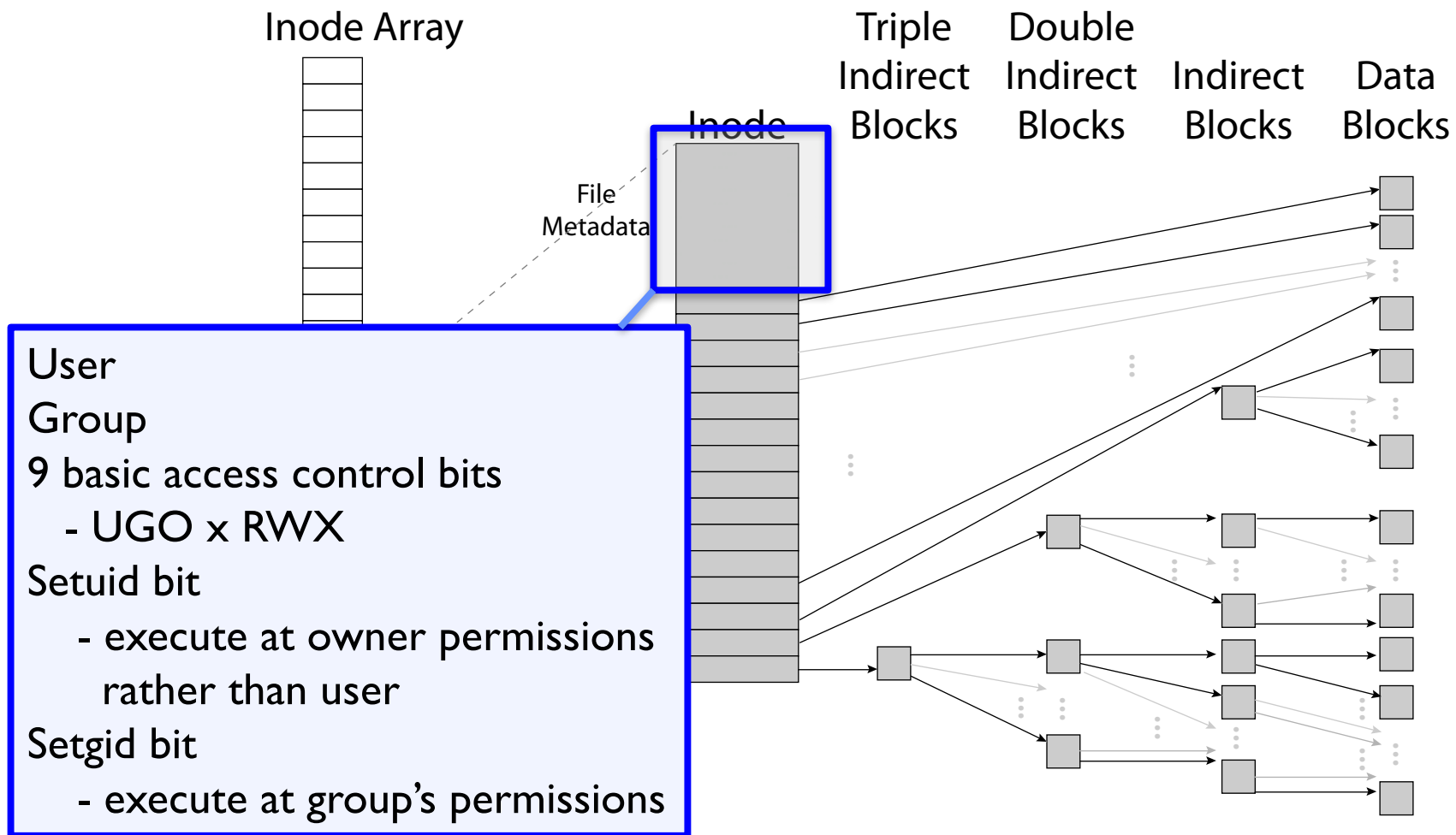
- Original inode format appeared in BSD 4.1
 - Berkeley Standard Distribution Unix
 - Part of your heritage!
 - Similar structure for Linux Ext2/3
- File Number is index into inode arrays
- Multi-level index structure
 - Great for little and large files
 - Asymmetric tree with fixed sized blocks

Unix File System (2/2)

- Metadata associated with the file
 - Rather than in the directory that points to it
- UNIX Fast File System (FFS) BSD 4.2 Locality Heuristics:
 - Block group placement
 - Reserve space
- Scalable directory structure

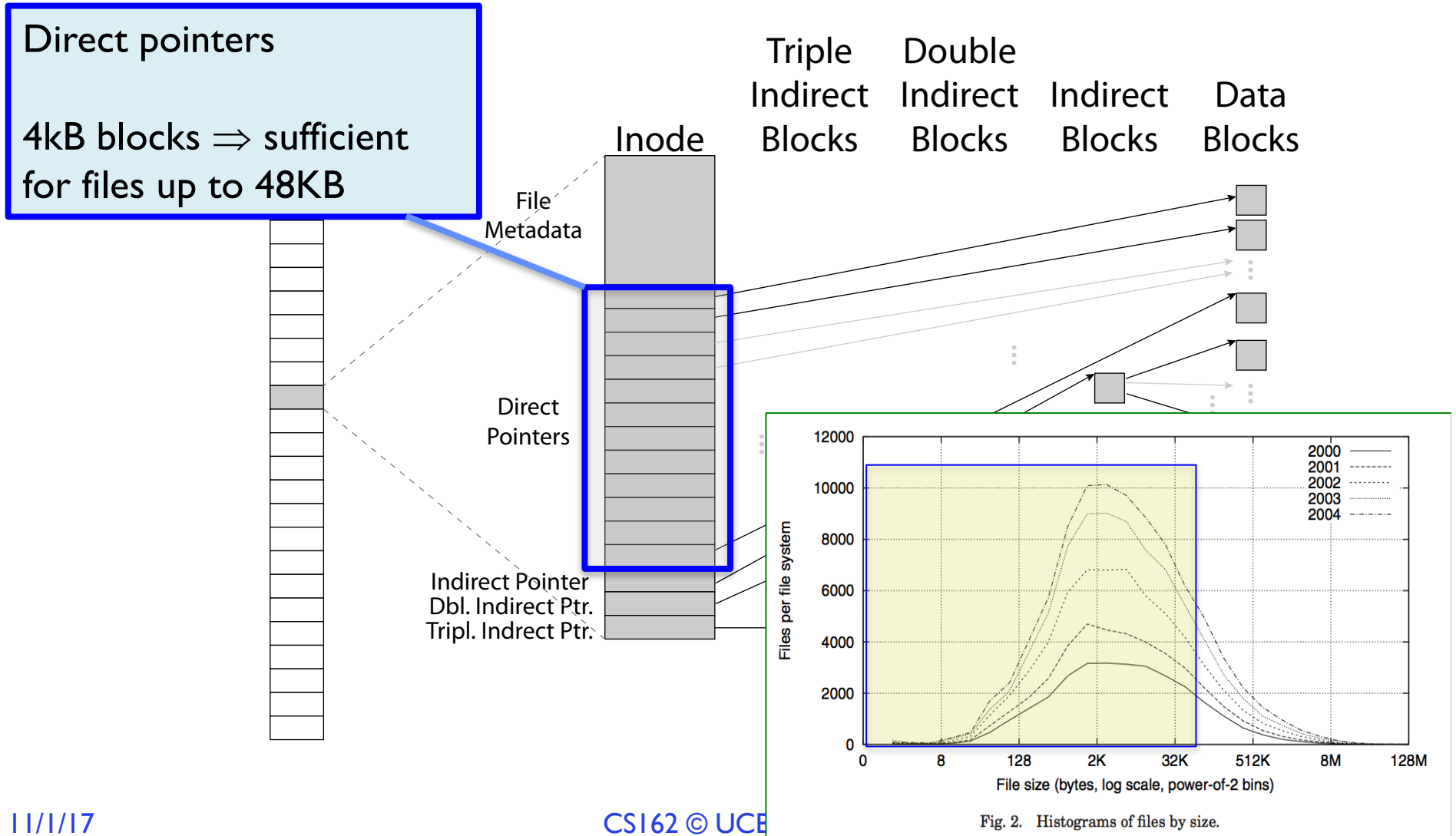
File Attributes

- inode metadata



Data Storage

- Small files: 12 pointers direct to data blocks



Data Storage

- Large files: 1,2,3 level indirect pointers

Indirect pointers

- point to a disk block containing only pointers
- 4 kB blocks \Rightarrow 1024 ptrs
- \Rightarrow 4 MB @ level 2
- \Rightarrow 4 GB @ level 3
- \Rightarrow 4 TB @ level 4

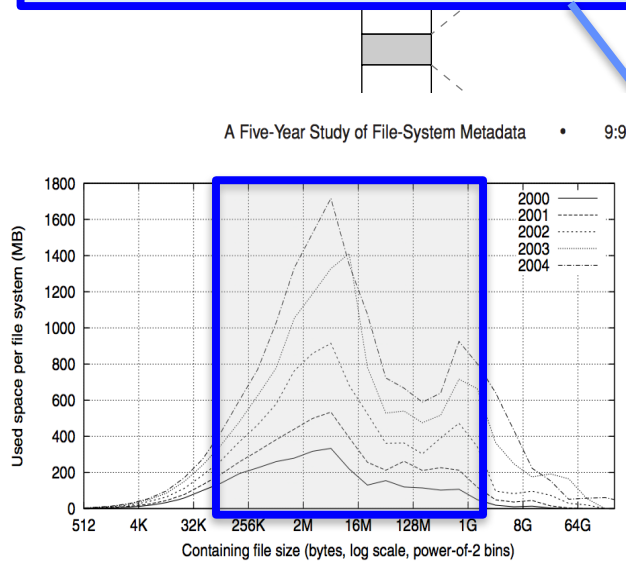
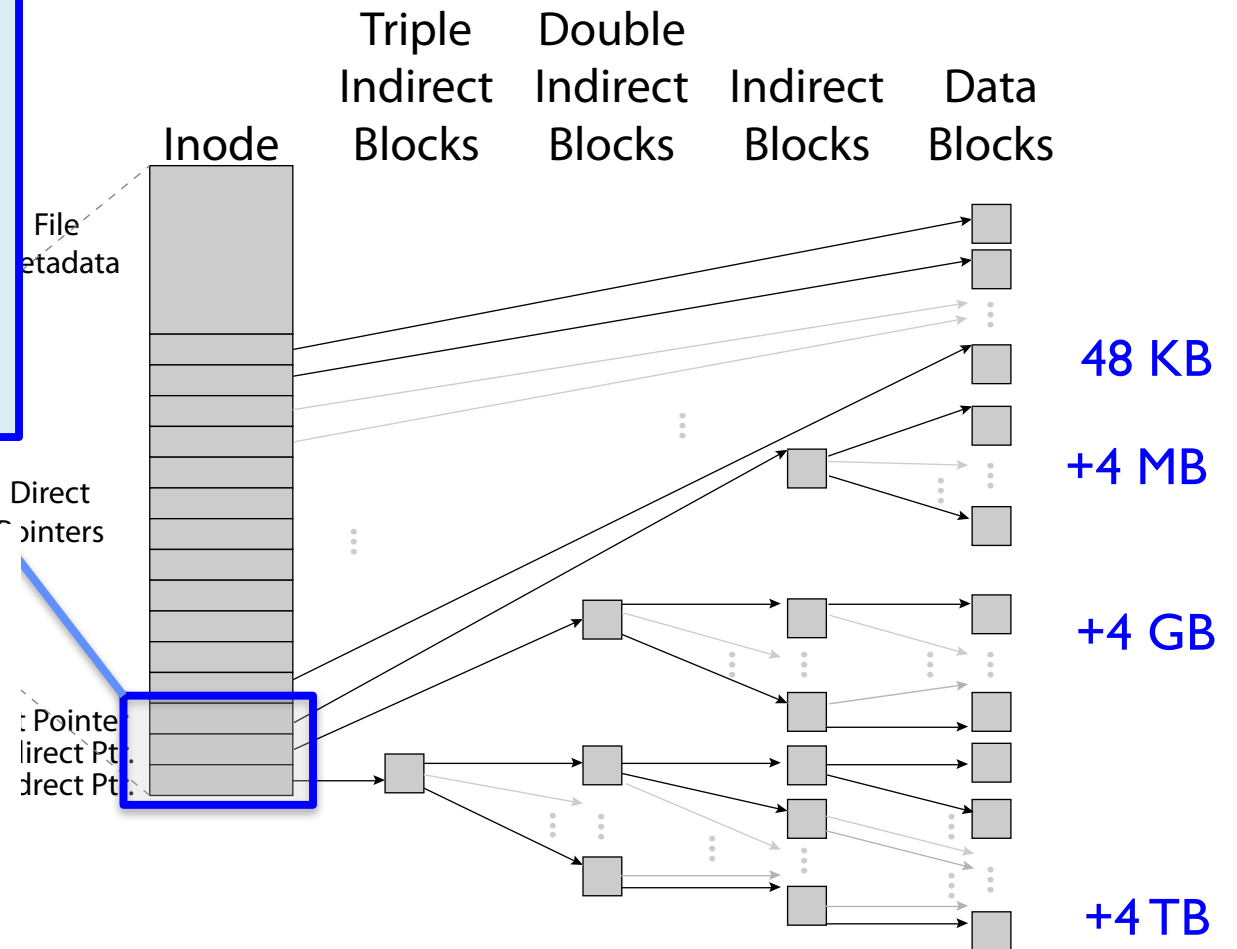


Fig. 4. Histograms of bytes by containing file size.



UNIX BSD 4.2 (1984) (1/2)

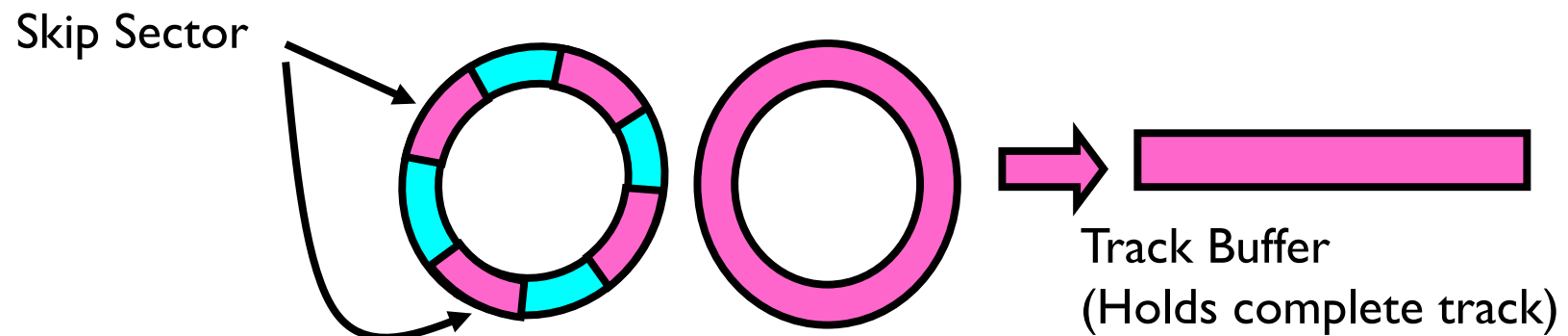
- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from Cray Operating System:
 - Uses bitmap allocation in place of freelist
 - Attempt to allocate files contiguously
 - 10% reserved disk space
 - Skip-sector positioning (mentioned later)

UNIX BSD 4.2 (1984) (2/2)

- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
 - How much contiguous space do you allocate for a file?
 - In BSD 4.2, just find some range of free blocks
 - » Put each new file at the front of different range
 - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
 - Also in BSD 4.2: store files from same directory near each other
- Fast File System (FFS)
 - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

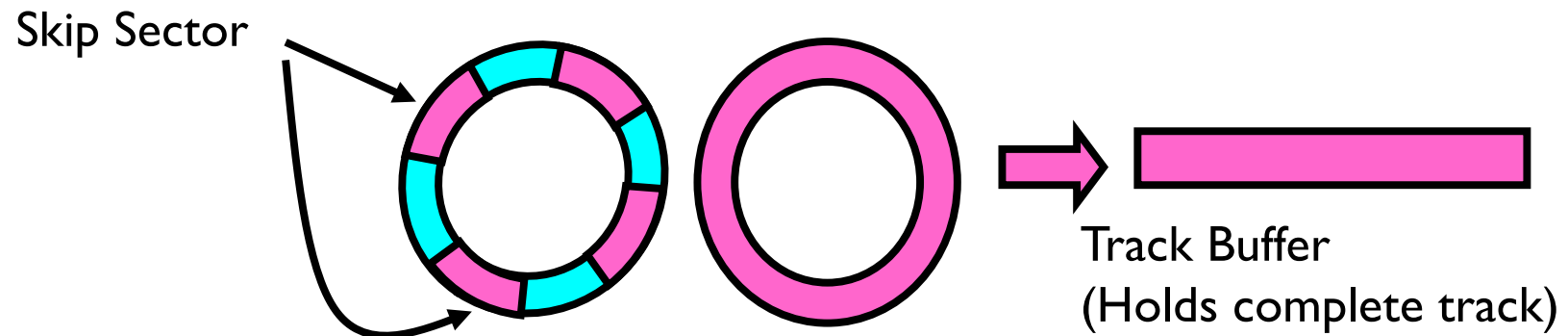
- Problem 2: Missing blocks due to rotational delay
 - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



- Solution I: Skip sector positioning (“interleaving”)
 - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
 - » Can be done by OS or in modern drives by the disk controller

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
 - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



- Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it
 - » This can be done either by OS (read ahead)
 - » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track
- Note: Modern disks + controllers do many things “under the covers”
 - Track buffers, elevator algorithms, bad block filtering

Where are inodes Stored?

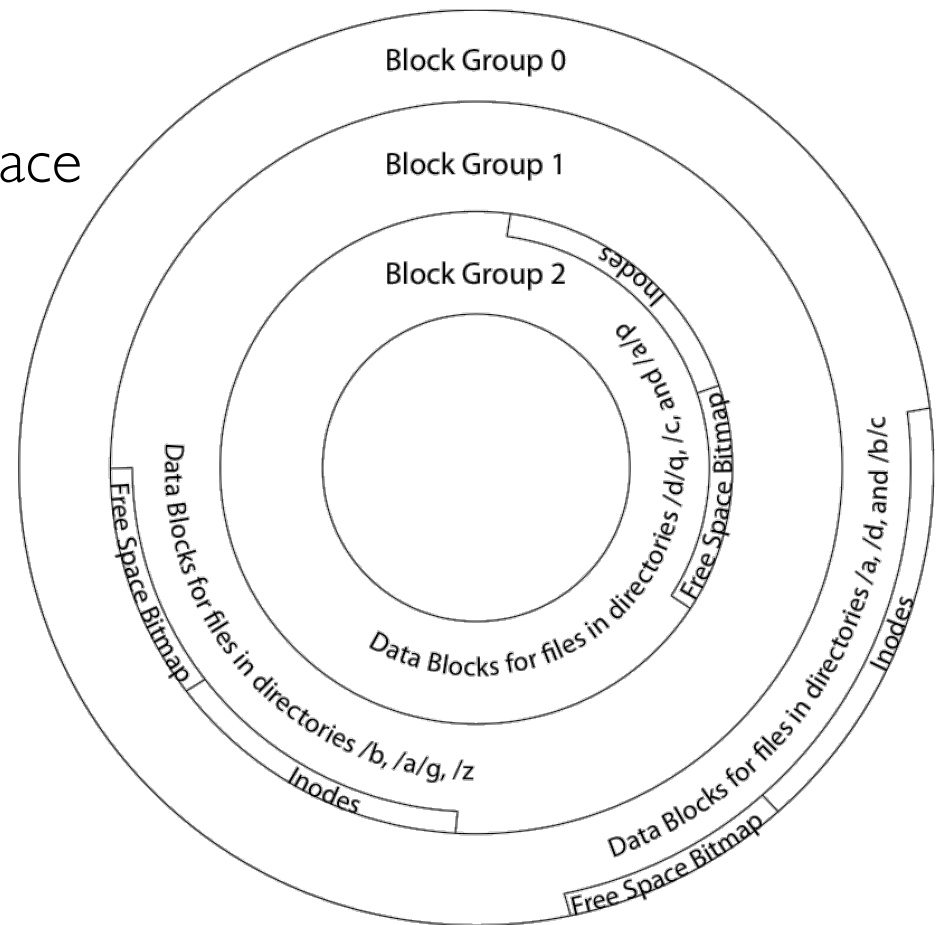
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
- Header not stored anywhere near the data blocks
 - To read a small file, seek to get header, seek back to data
- Fixed size, set when disk is formatted
 - At formatting time, a fixed number of inodes are created
 - Each is given a unique number, called an “inumber”

Where are inodes Stored?

- Later versions of UNIX moved the header information to be closer to the data blocks
 - Often, inode for file stored in same “cylinder group” as parent directory of the file (makes an **ls** of that directory run fast)
- Pros:
 - UNIX BSD 4.2 puts bit of file header array on many cylinders
 - For small directories, can fit all data, file headers, etc. in same cylinder \Rightarrow no seeks!
 - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
 - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
- Part of the Fast File System (FFS)
 - General optimization to avoid seeks

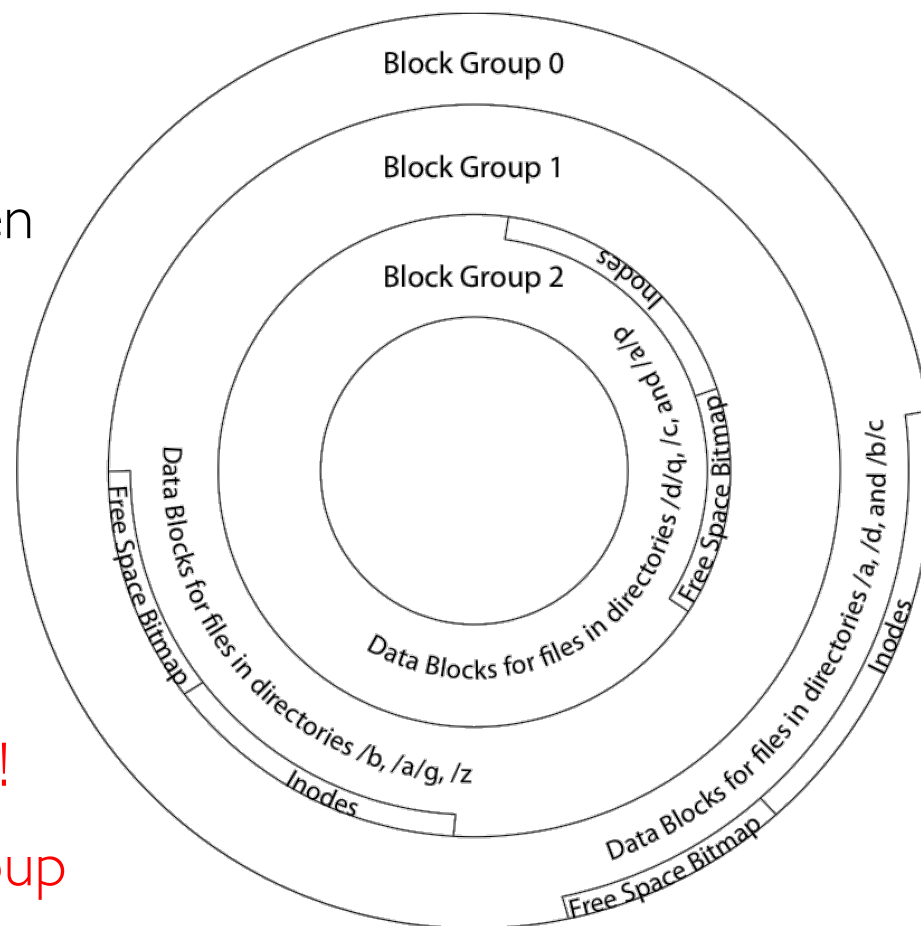
4.2 BSD Locality: Block Groups

- File system volume is divided into a set of block groups
 - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
 - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group

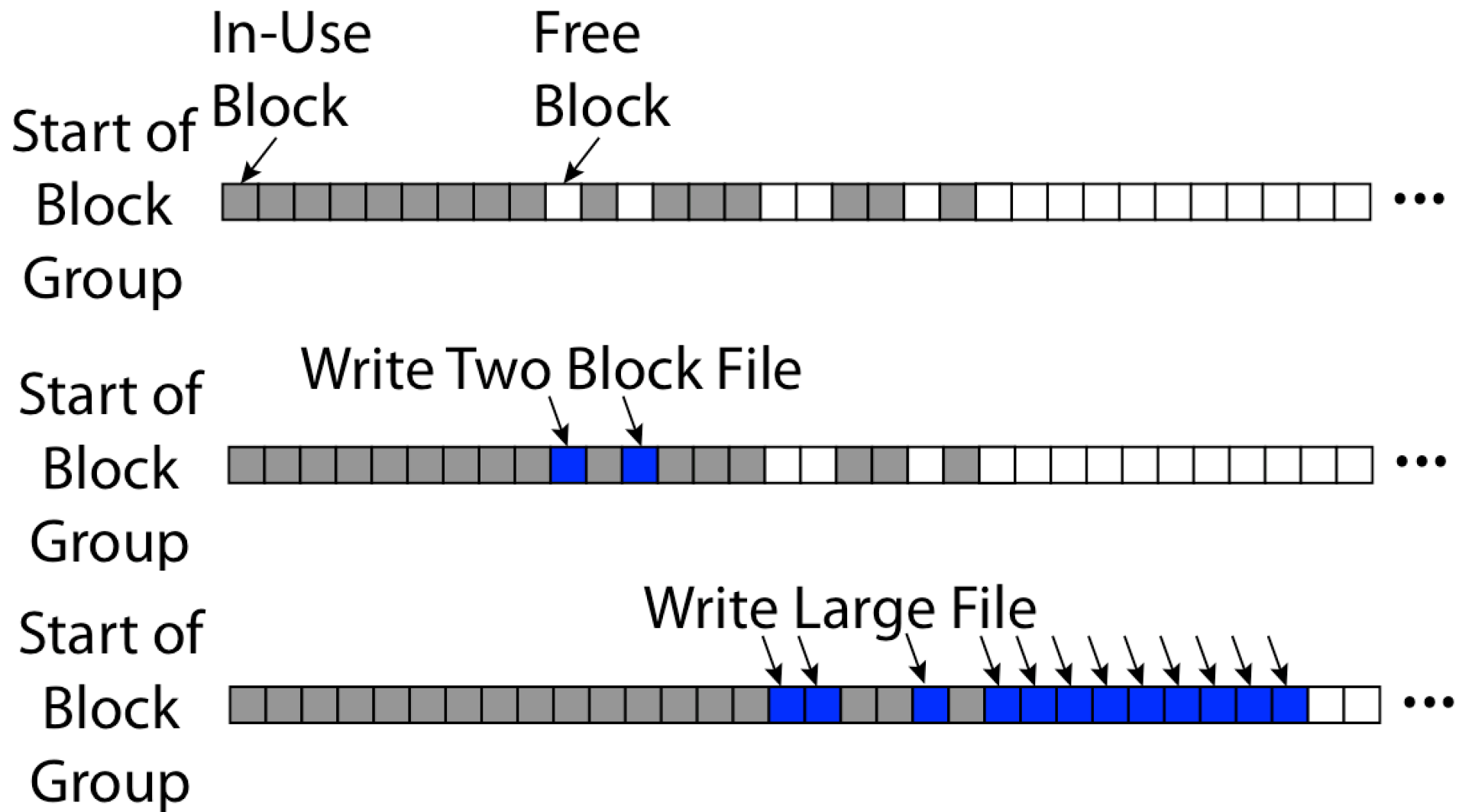


4.2 BSD Locality: Block Groups

- First-Free allocation of new file blocks
 - To expand file, first try successive blocks in bitmap, then choose new range of blocks
 - Few little holes at start, big sequential runs at end of group
 - Avoids fragmentation
 - Sequential layout for big files
- Important: keep 10% or more free!
 - Reserve space in the Block Group



UNIX 4.2 BSD FFS First Fit Block Allocation



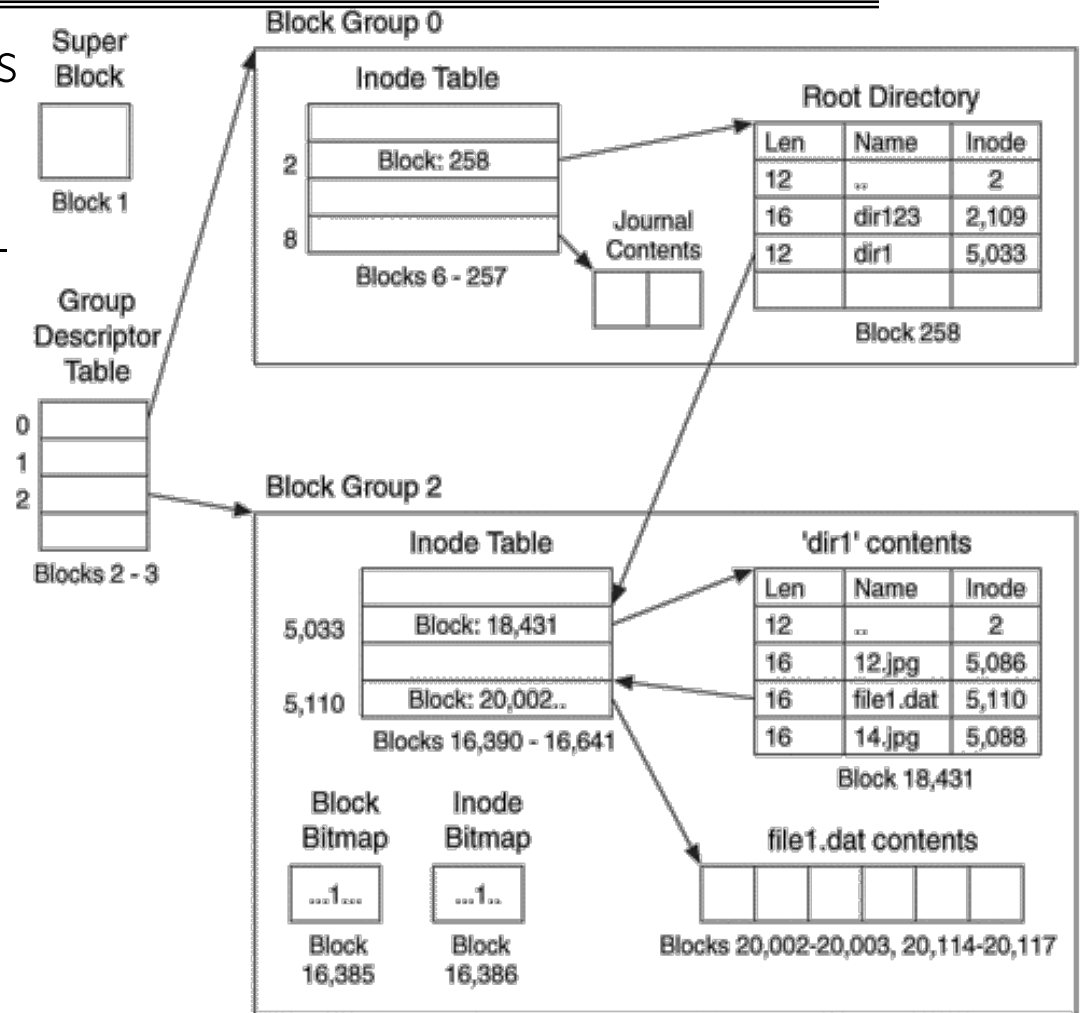
UNIX 4.2 BSD FFS

- Pros
 - Efficient storage for both small and large files
 - Locality for both small and large files
 - Locality for metadata and data
 - No defragmentation necessary!
- Cons
 - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
 - Inefficient encoding when file is mostly contiguous on disk
 - Need to reserve 10-20% of free space to prevent fragmentation

BREAK

Linux Example: Ext2/3 Disk Layout

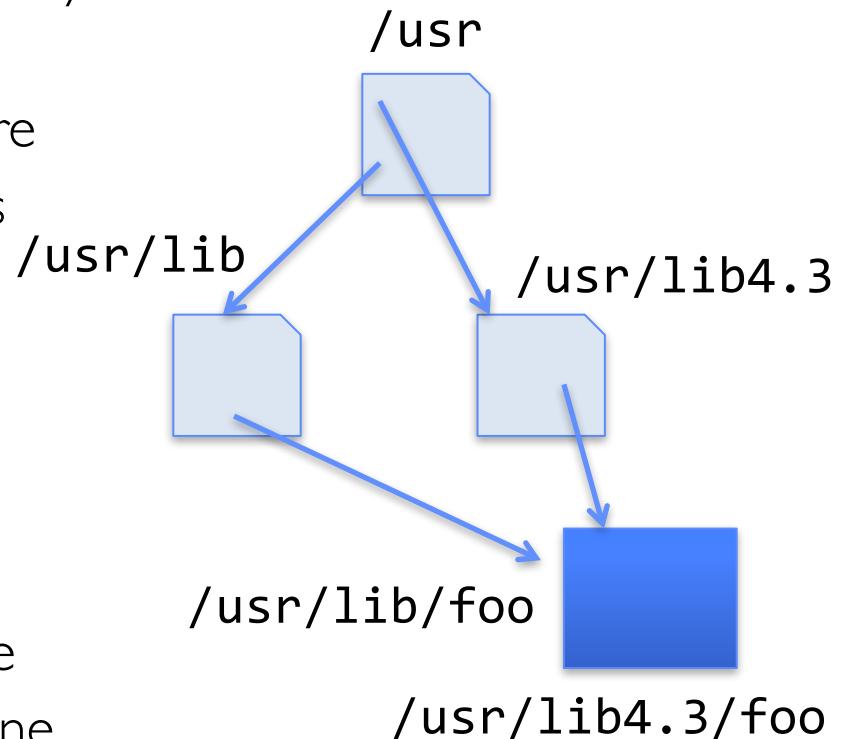
- Disk divided into block groups
 - Provides locality
 - Each group has two block-sized bitmaps (free blocks/inodes)
 - Block sizes settable at format time: 1K, 2K, 4K, 8K...
- Actual inode structure similar to 4.2 BSD
 - with 12 direct pointers
- Ext3: Ext2 with Journaling
 - Several degrees of protection with comparable overhead



- Example: create a file1.dat under /dir1/ in Ext3

A bit more on directories

- Stored in files, can be read, but typically don't
 - System calls to access directories
 - **open** / **creat** traverse the structure
 - **mkdir** / **rmdir** add/remove entries
 - **link** / **unlink** (**rm**)
 - » Link existing file to a directory
 - Not in FAT !
 - » Forms a DAG
- When can file be deleted?
 - Maintain ref-count of links to the file
 - Delete after the last reference is gone
- libc support
 - `DIR * opendir (const char *dirname)`
 - `struct dirent * readdir (DIR *dirstream)`
 - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`



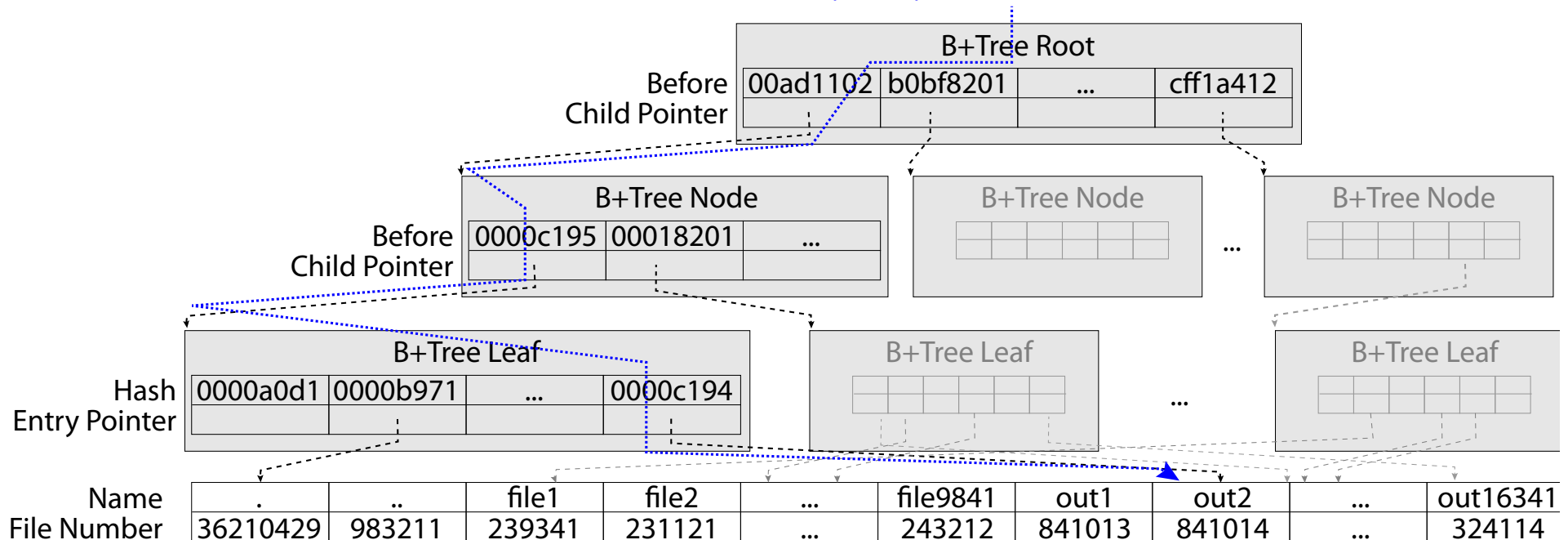
Links

- Hard link
 - Sets another directory entry to contain the file number for the file
 - Creates another name (path) for the file
 - Each is “first class”
- Soft link or Symbolic Link or Shortcut
 - Directory entry contains the path and name of the file
 - Map one name to another name

Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD

Search for hash("out2") = 0x0000c194



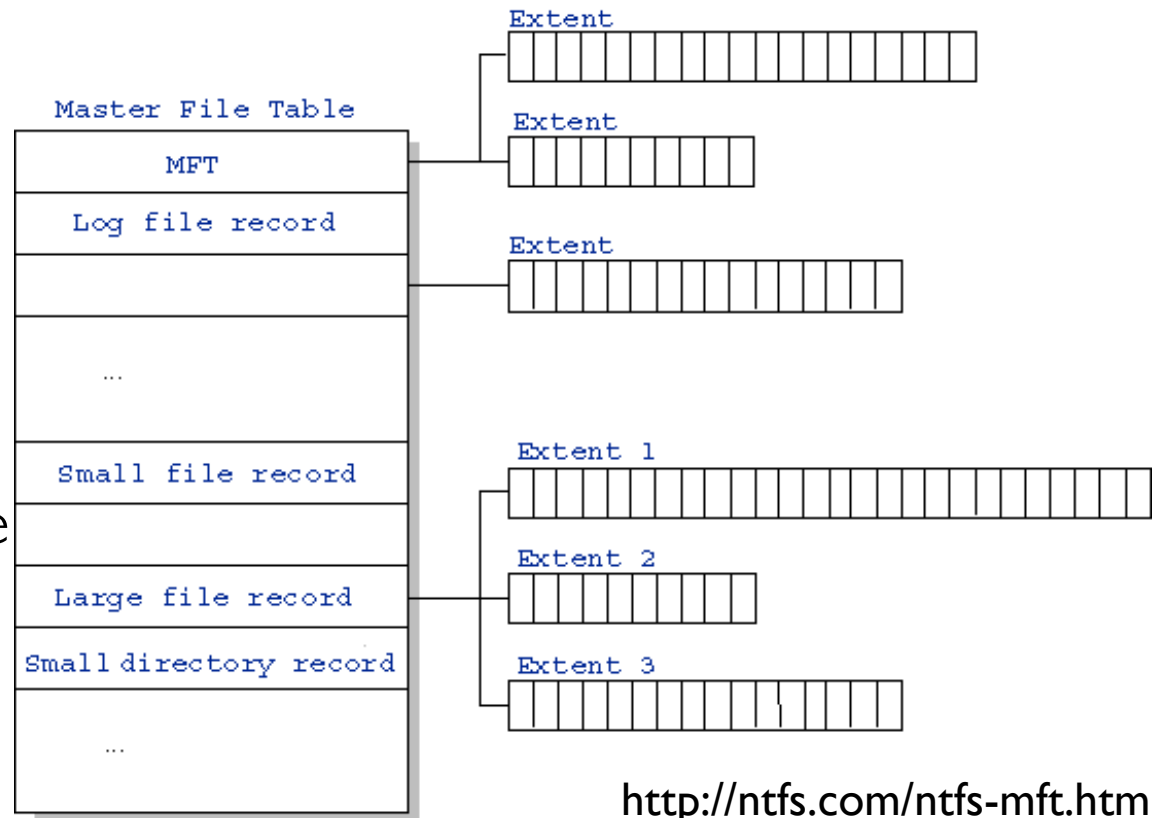
"out2" is file 841014

NTFS

- New Technology File System (NTFS)
 - Default on Microsoft Windows systems
- Variable length extents
 - Rather than fixed blocks
- Everything (almost) is a sequence of <attribute:value> pairs
 - Meta-data and data
- Mix direct and indirect freely
- Directories organized in B-tree structure by default

NTFS

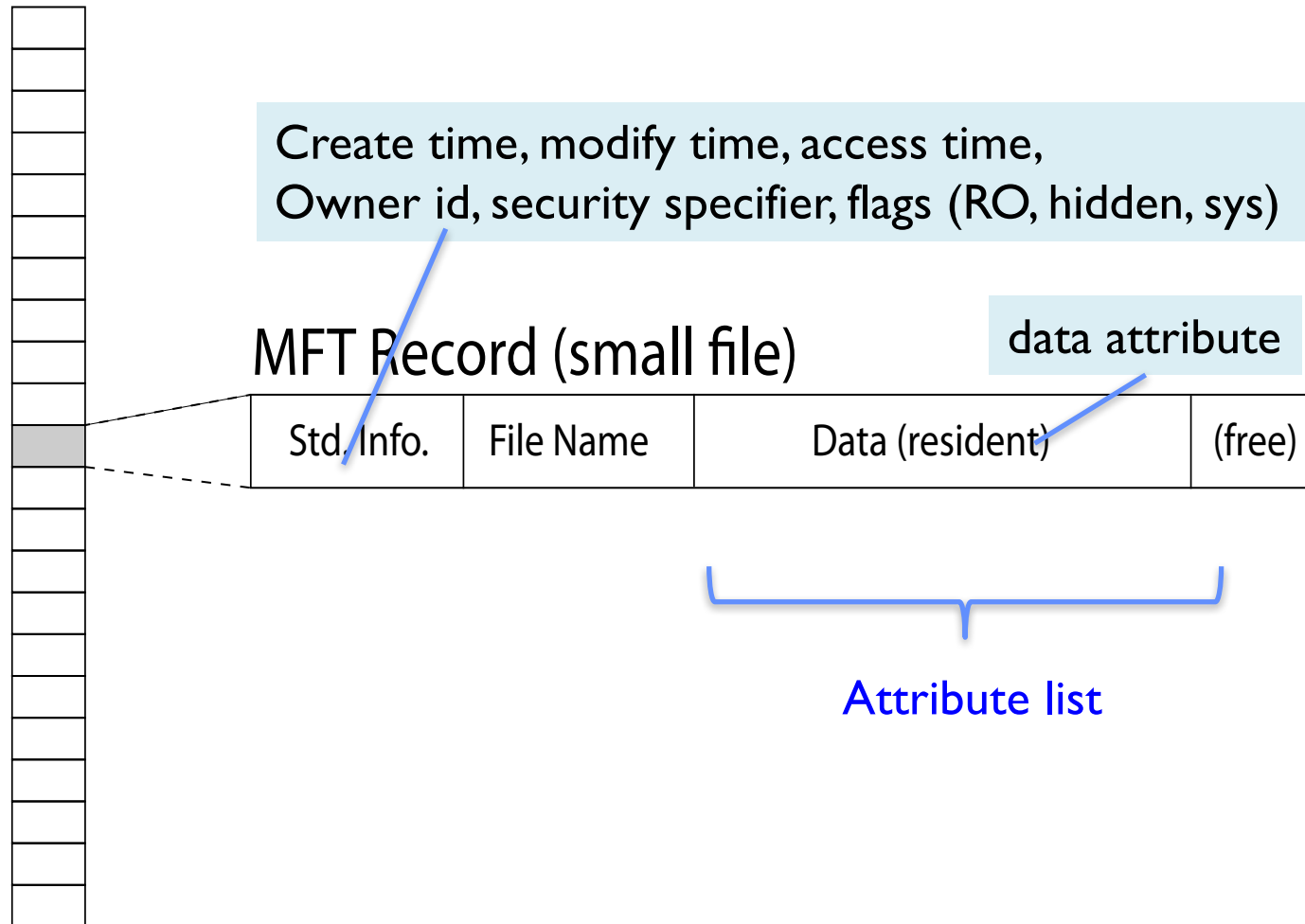
- Master File Table
 - Database with Flexible 1KB entries for metadata/data
 - Variable-sized attribute records (data or metadata)
 - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
 - Block pointers cover runs of blocks
 - Similar approach in Linux (ext4)
 - File create can provide hint as to size of file
- Journaling for reliability
 - Discussed later



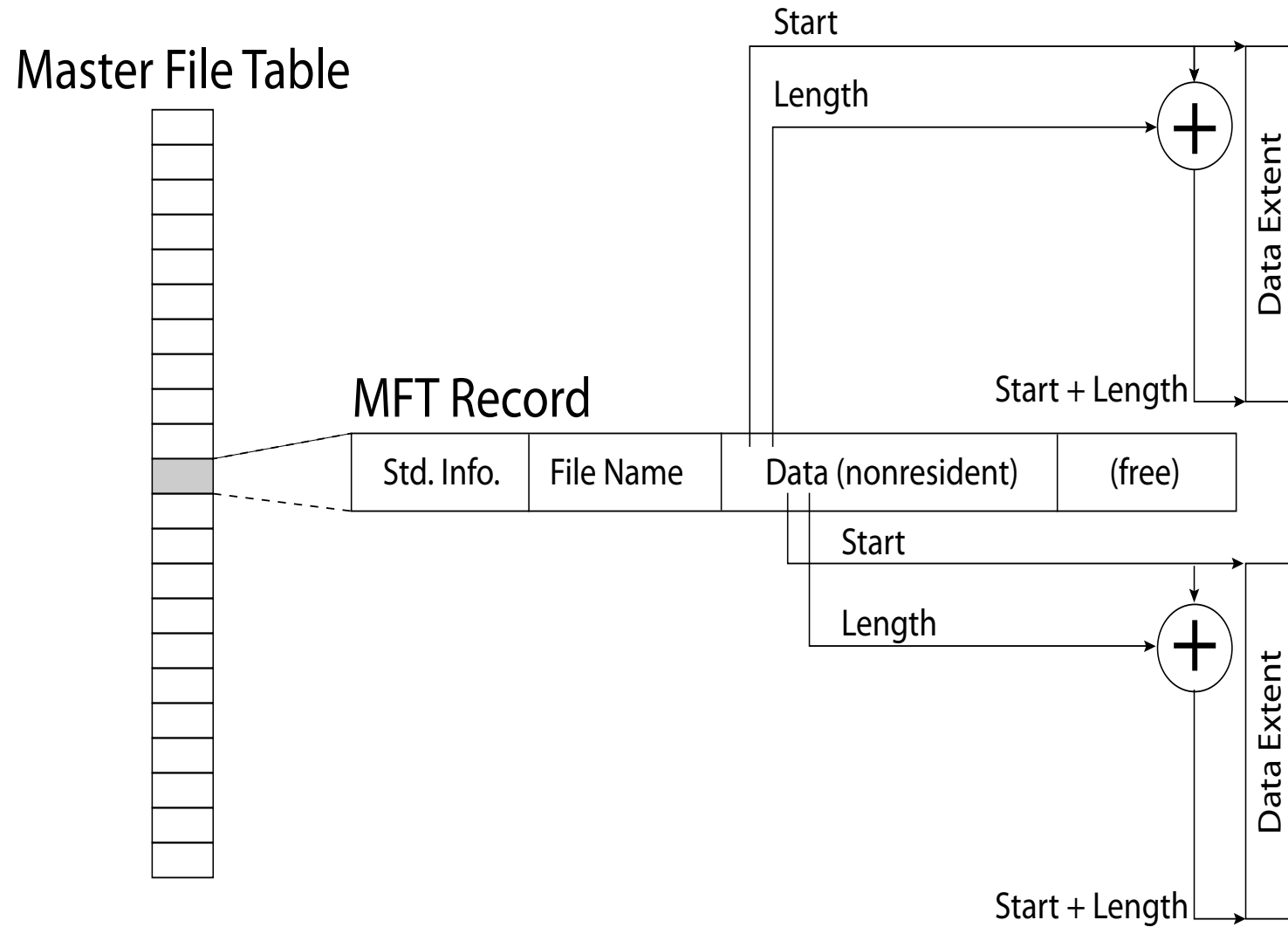
<http://ntfs.com/ntfs-mft.htm>

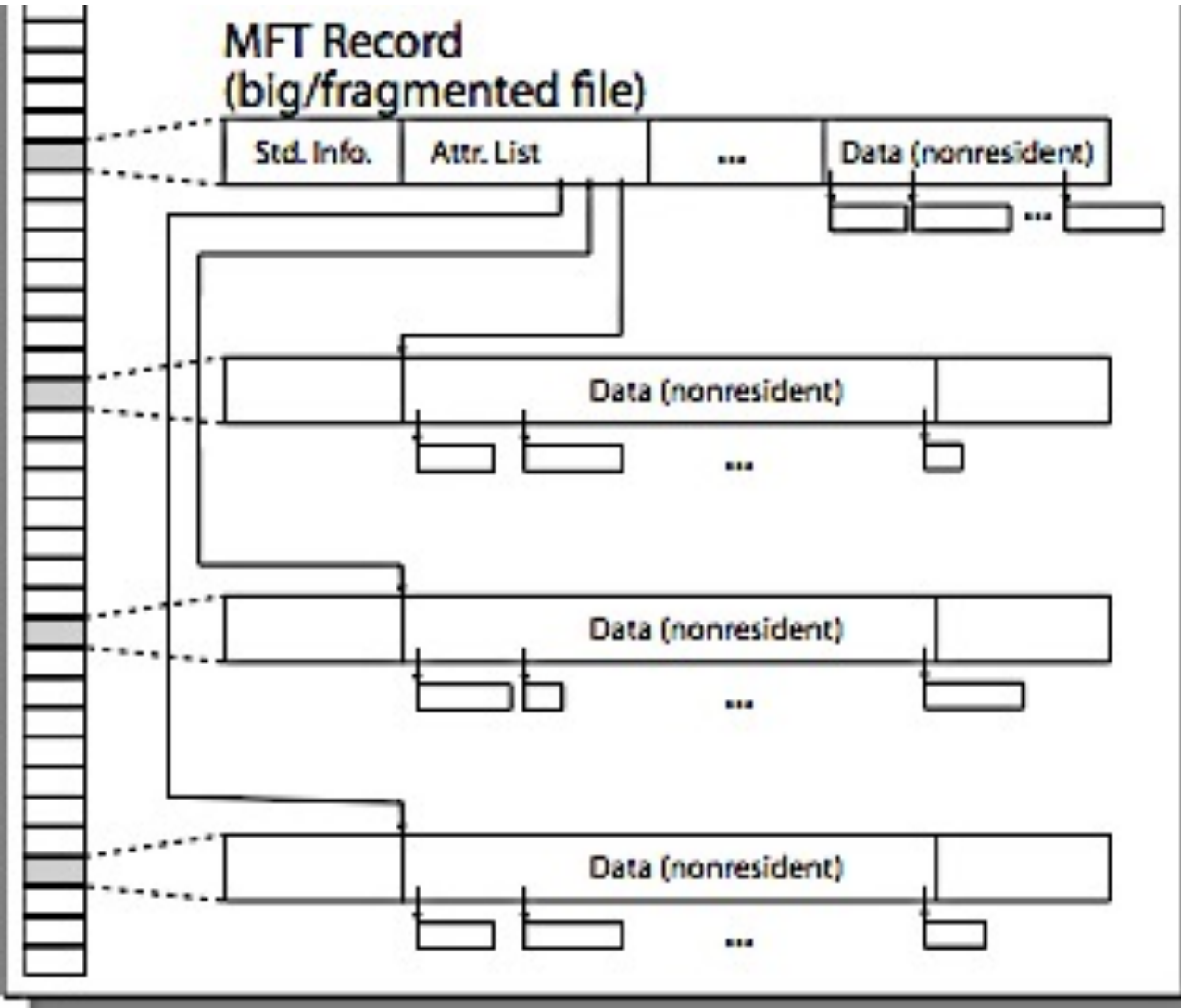
NTFS Small File

Master File Table

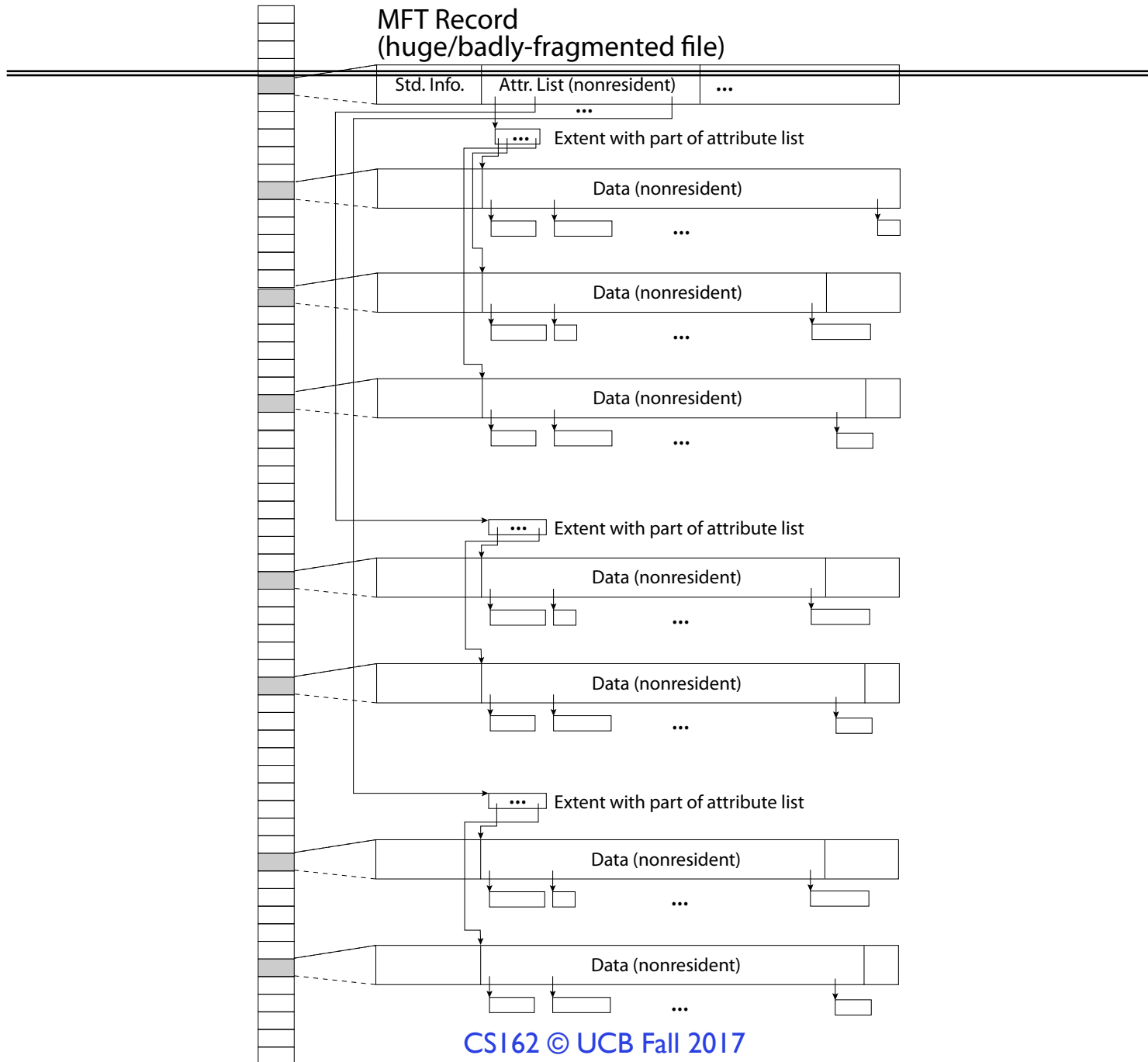


NTFS Medium File





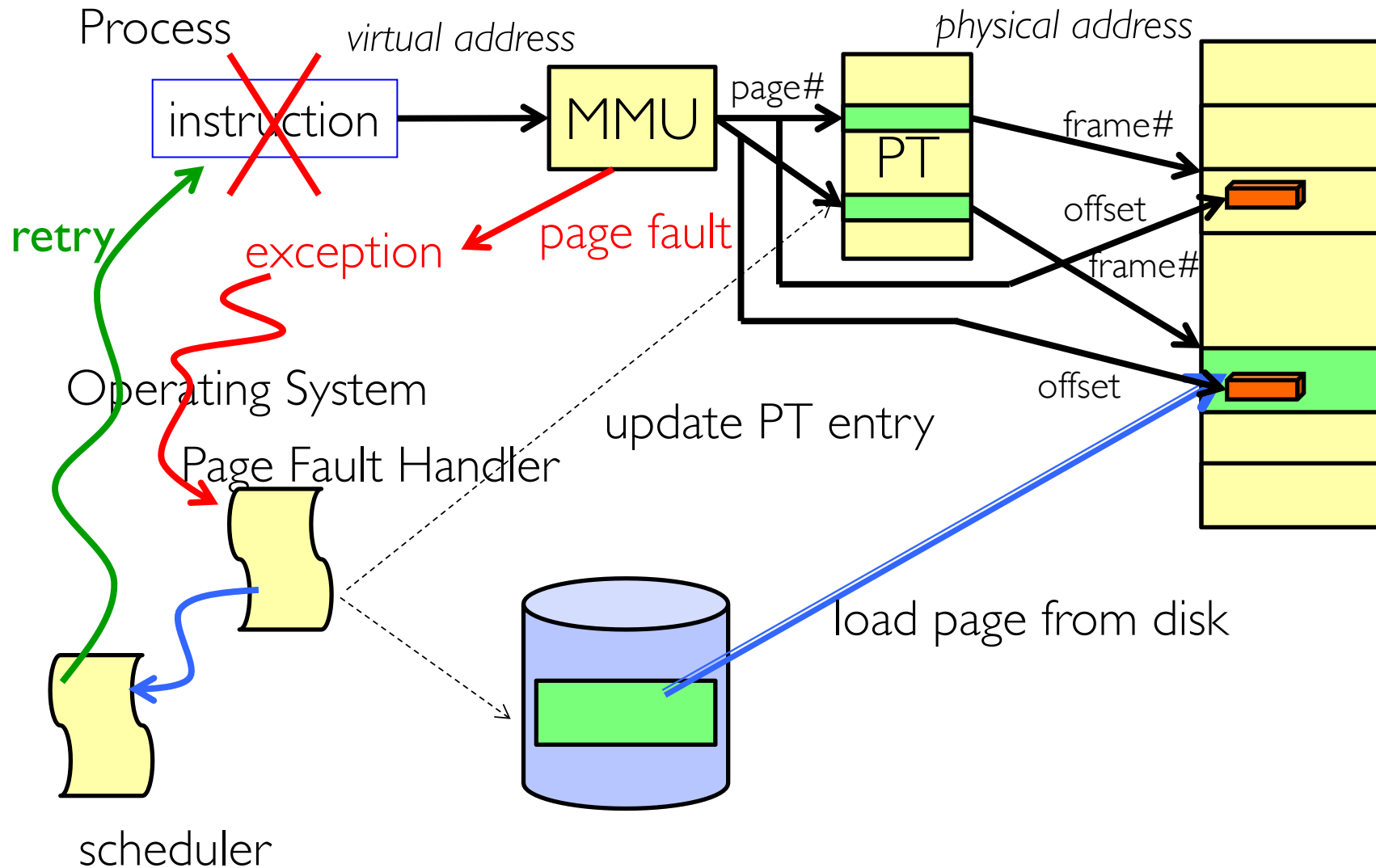
Master File Table



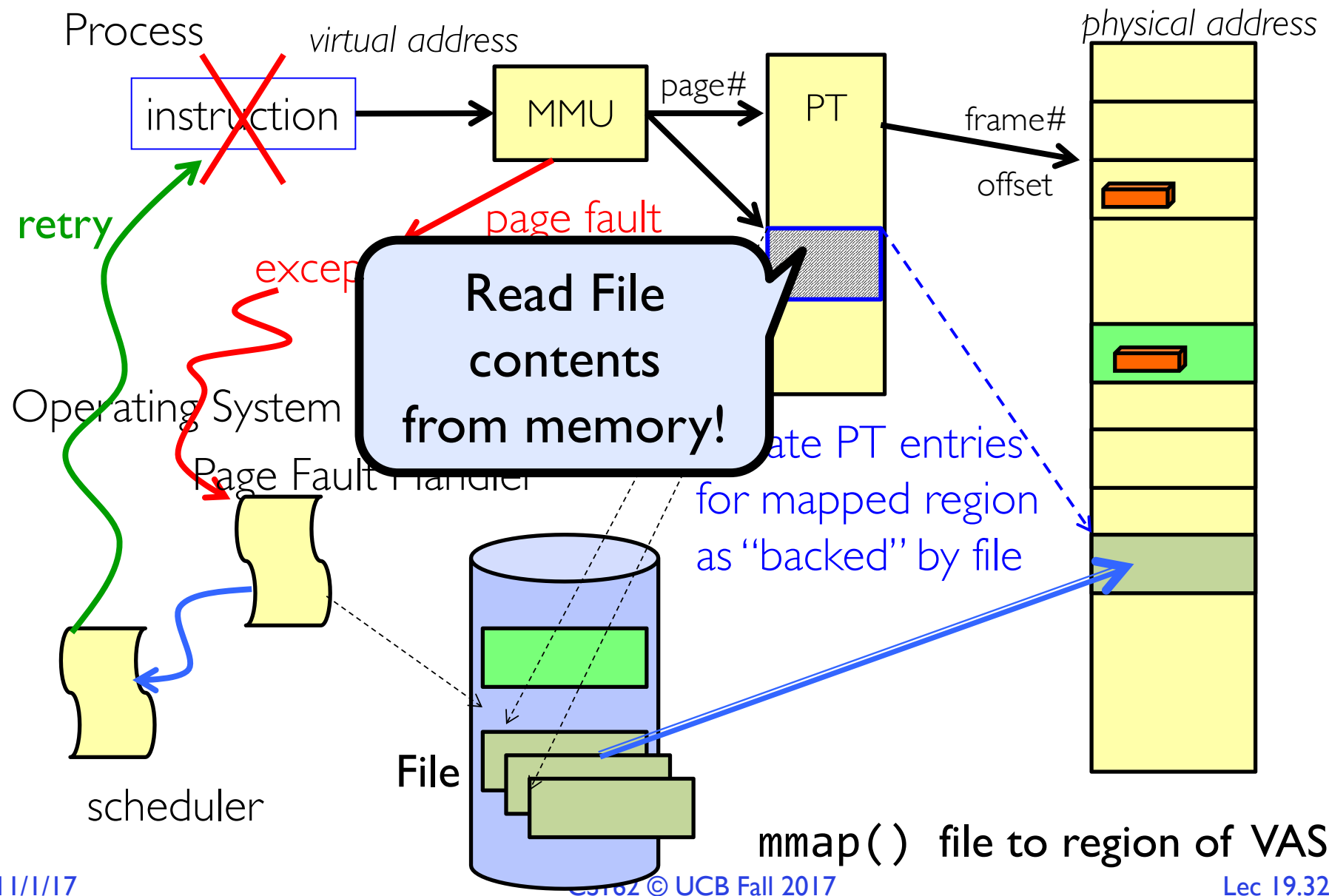
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
 - This involves multiple copies into caches in memory, plus system calls
- What if we could “map” the file directly into an empty region of our address space
 - Implicitly “page it in” when we read it
 - Write it and “eventually” page it out
- Executable files are treated this way when we exec the process!!

Recall: Who Does What, When?



Using Paging to `mmap()` Files



mmap () system call

MMAP(2)	BSD System Calls Manual	MMAP(2)
NAME		
mmap -- allocate memory, or map files or devices into memory		
LIBRARY		
Standard C Library (libc, -lc)		
SYNOPSIS		
#include <sys/mman.h>		
void *		
mmap (void * <u>addr</u> , <u>size_t</u> <u>len</u> , <u>int</u> <u>prot</u> , <u>int</u> <u>flags</u> , <u>int</u> <u>fd</u> ,		
<u>off_t</u> <u>offset</u>);		
DESCRIPTION		
The mmap () system call causes the pages starting at <u>addr</u> and continuing for at most <u>len</u> bytes to be mapped from the object described by <u>fd</u> , starting at byte offset <u>offset</u> . If <u>offset</u> or <u>len</u> is not a multiple of the page size, the mapped region may extend past the specified range.		

- May map a specific region or let the system find one for you
 - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

An mmap() Example

```
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h,fcntl.h,unistd.h */
```

```
int something = 162;
```

```
int main (int argc, char *argv[]) {  
    int myfd;  
    char *mfile;
```

```
    printf("Data at: %16lx\n", (long) something);  
    printf("Heap at : %16lx\n", (long) something);  
    printf("Stack at: %16lx\n", (long) something);
```

```
    /* Open the file */
```

```
    myfd = open(argv[1], O_RDWR | O_CREAT, 0666);  
    if (myfd < 0) { perror("open failed"); return 1; }
```

```
    /* map the file */
```

```
    mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_SHARED, myfd, 0);  
    if (mfile == MAP_FAILED) { perror("mmap failed"); return 1; }
```

```
    printf("mmap at : %16lx\n", (long) something);
```

```
    puts(mfile);  
    strcpy(mfile+20, "Let's write over it");  
    close(myfd);  
    return 0;
```

```
}
```

```
$ ./mmap test
```

```
Data at:          105d63058
```

```
Heap at :          7f8a33c04b70
```

```
Stack at:          7ffff59e9db10
```

```
mmap at :          105d97000
```

```
This is line one
```

```
This is line two
```

```
This is line three
```

```
This is line four
```

```
$ cat test
```

```
This is line one
```

```
This is line two  
Let's write over it
```

```
This is line four
```

File System Summary (1/2)

- File System:
 - Transforms blocks into Files and Directories
 - Optimize for size, access and usage patterns
 - Maximize sequential access, allow efficient random access
 - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called “inode”
- Naming: translating from user-visible names to actual sys resources
 - Directories used for naming for local file systems
 - Linked or tree structure stored in files
- Multilevel Indexed Scheme
 - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
 - NTFS: variable extents not fixed blocks, tiny files data is in header

File System Summary (2/2)

- 4.2 BSD Multilevel index files
 - Inode contains ptrs to actual blocks, indirect blocks, double indirect blocks, etc.
 - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
- File layout driven by freespace management
 - Integrate freespace, inode table, file blocks and dirs into block group
- Deep interactions between mem management, file system, sharing
 - **mmap()**: map file or anonymous segment to memory