CS162:
Operating Systems and Systems Programming

Lecture 24:
Filesystems: FAT, FFS, NTFS

30 July 2015
Charles Reiss
https://cs162.eecs.berkeley.edu/

Building a File System

File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

File System Components

- Disk Management: collecting disk blocks into files
- Naming: Interface to find files by name, not by blocks
- Protection: Layers to keep data secure
- Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc

Recall: Building a File System (2)

User vs. System View of a File

- User's view:
  - Durable Data Structures
- System's view (system call interface):
  - Collection of Bytes (UNIX)
  - Doesn't matter to system what kind of data structures you want to store on disk!
- System's view (inside OS):
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size >= sector size; in UNIX, block size is 4KB

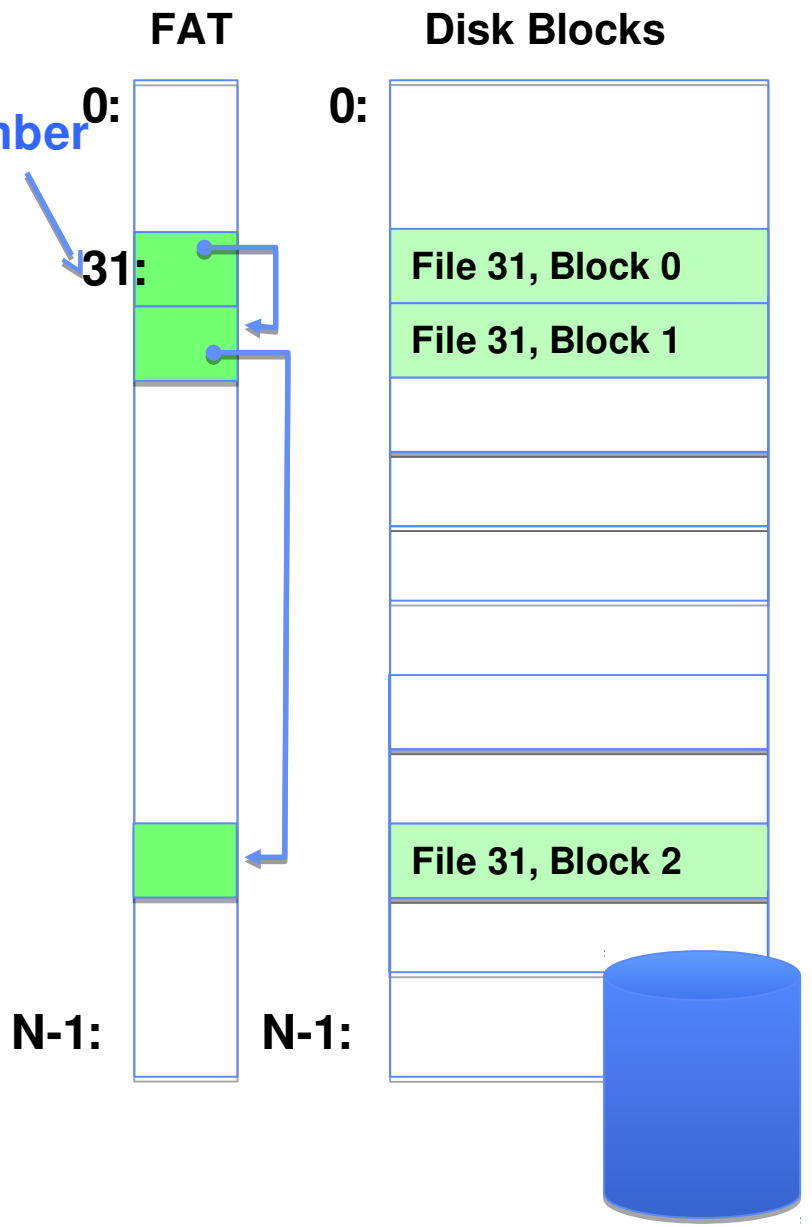
FAT (File Allocation Table)

Simple way to store blocks of a file: **linked list**

File number is the first block

FAT contains pointers to **the next block** for each block

- One entry for each data block

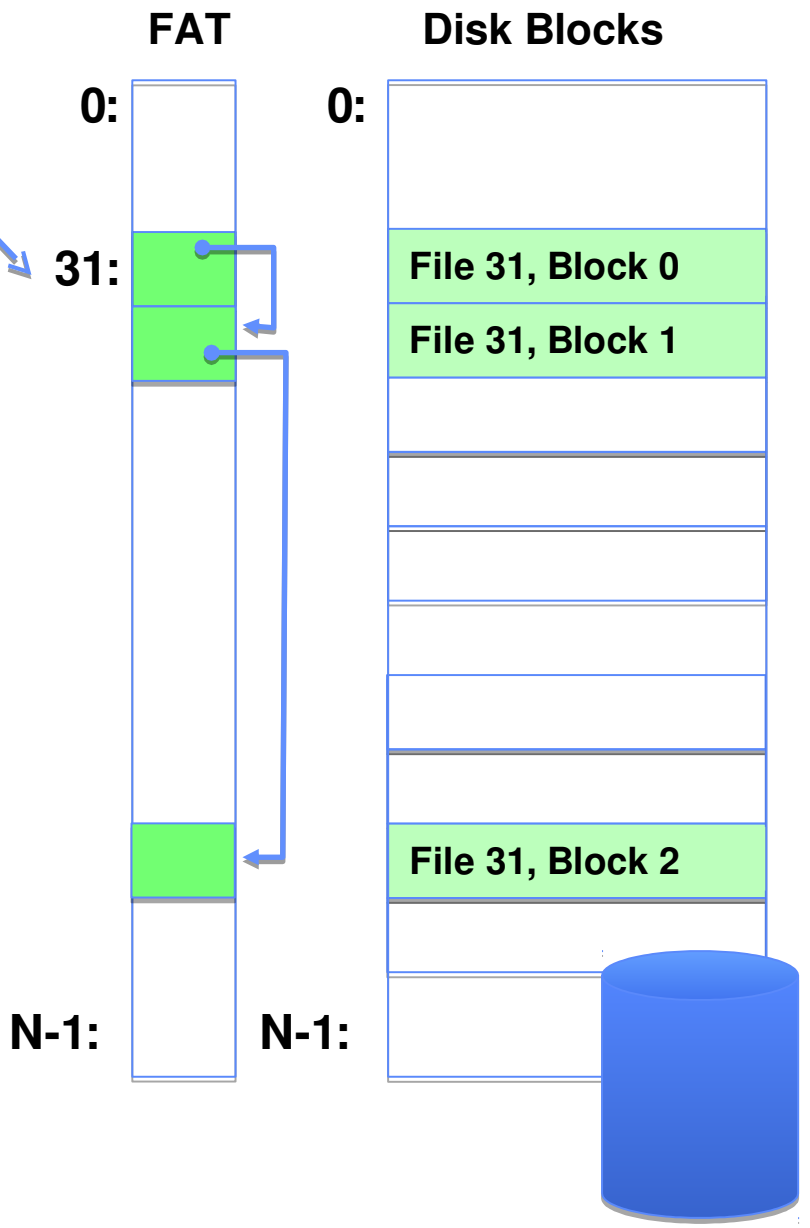


FAT (File Allocation Table)

Assume (for now) we have a way to translate a path to a "file number"

Example: file_read 31, < 2, x>

- Index into FAT with file number
- Follow linked list to block 2 of the file
- Read the block from disk into mem



FAT Properties

File is collection of disk blocks

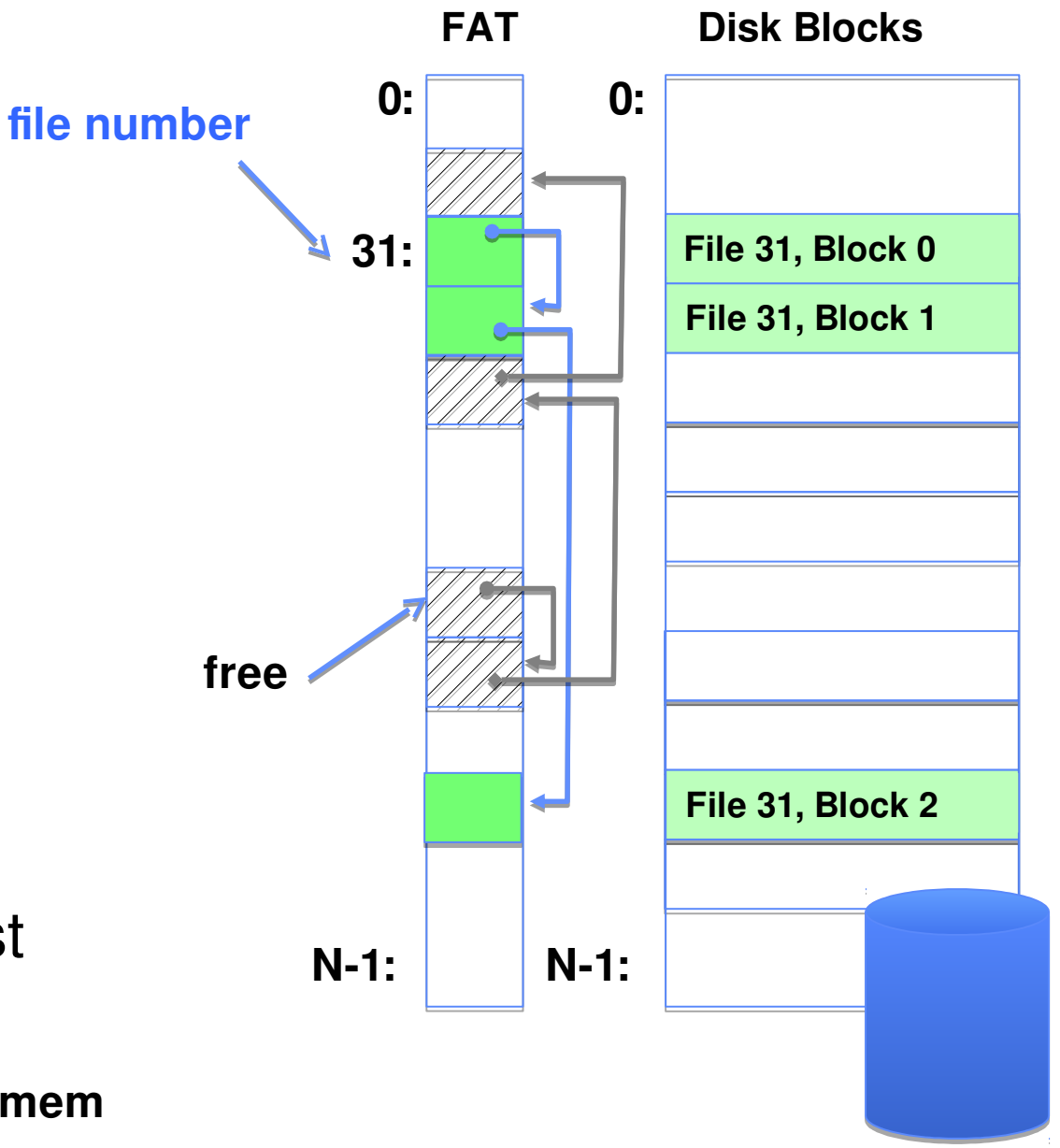
FAT is linked list 1-1 with blocks

File Number is index of first block list for the file

File offset (o = B:x)

Follow list to get block #

Unused blocks = FAT free list



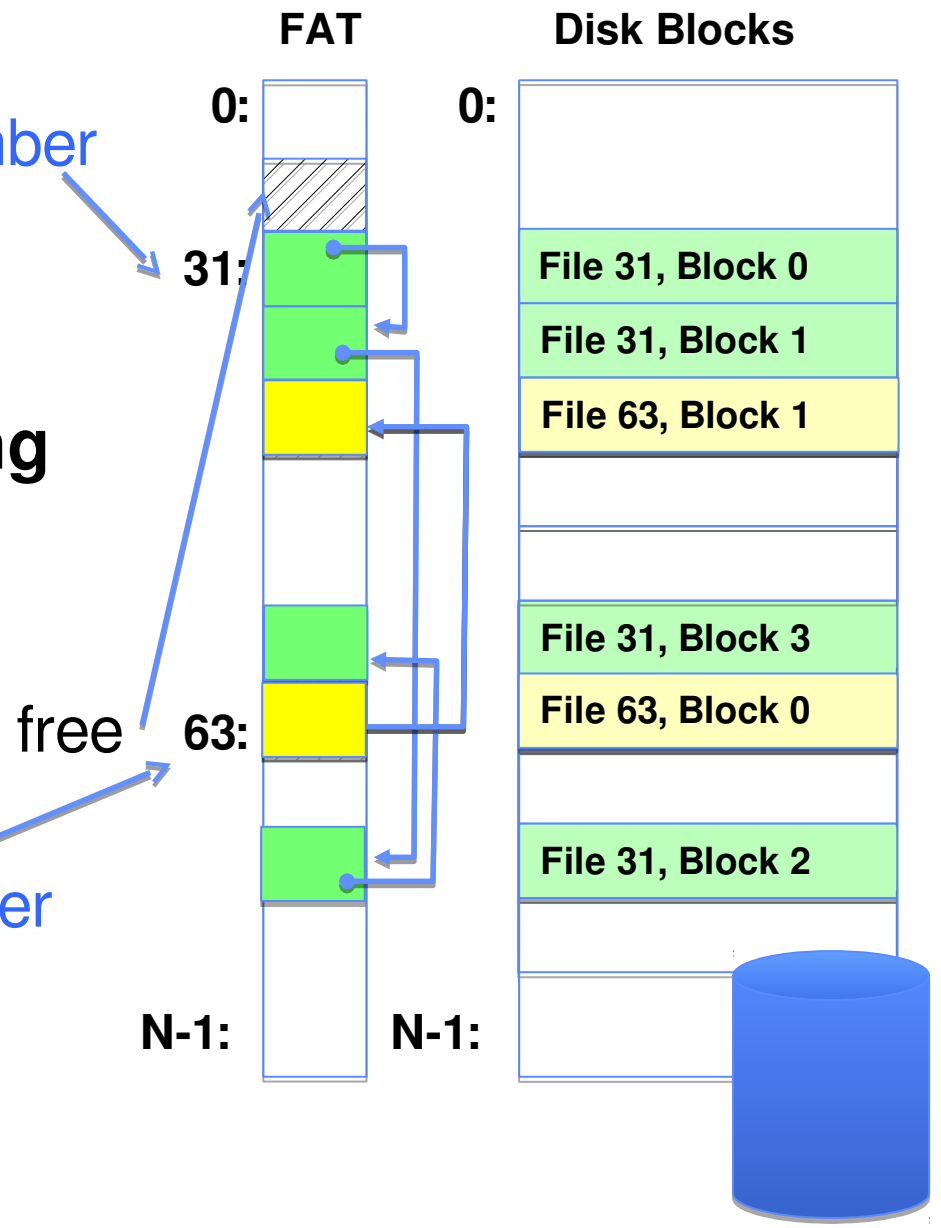
Storing the FAT

On disk when system is shutdown

- Fixed, well-known location

Copied in memory when running

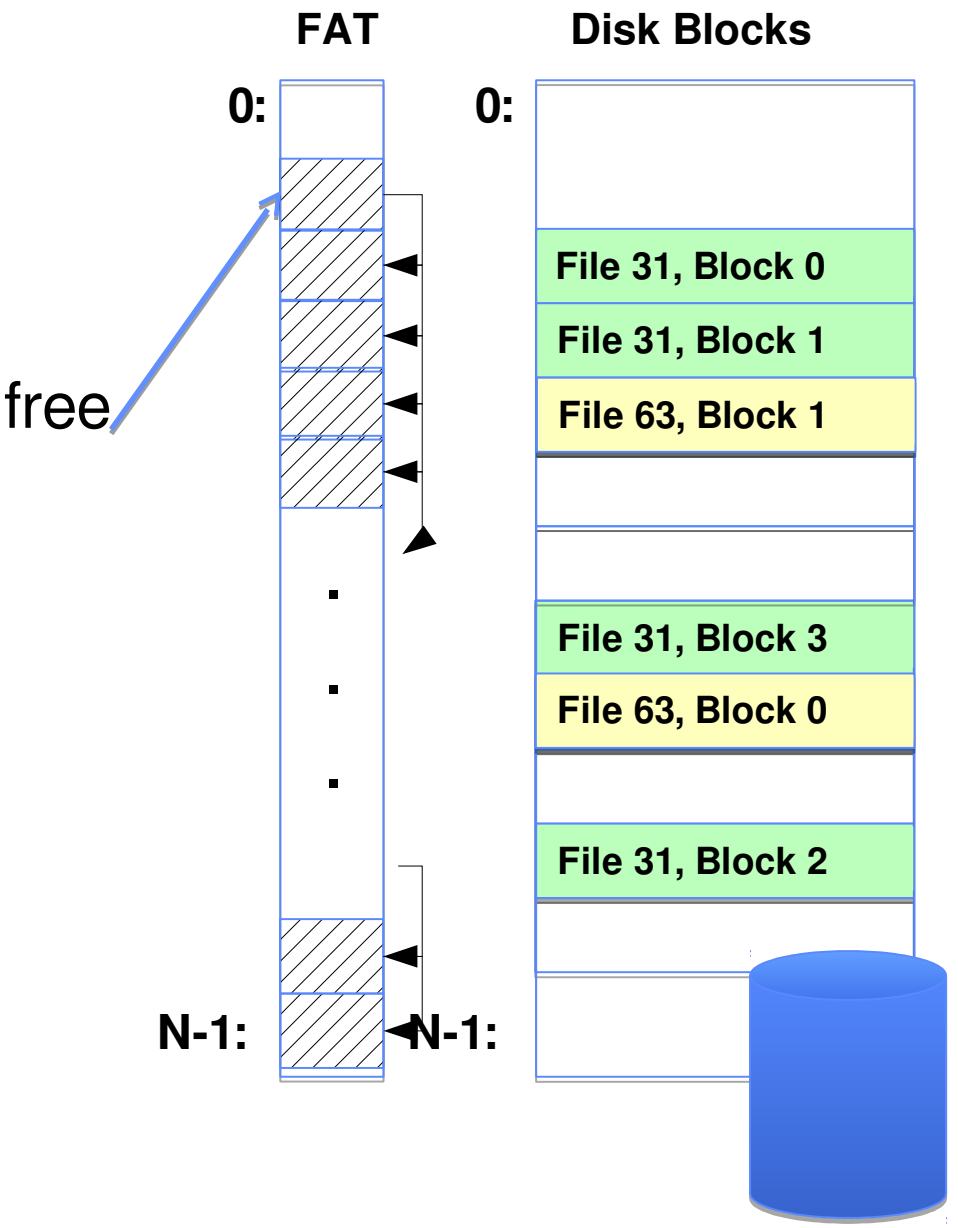
- Makes accesses, updates fast
- Otherwise *lots of disk seeking* to locate the blocks of a file



FAT Setup

Format a new FAT drive?

Link up the free list



FAT Assessment

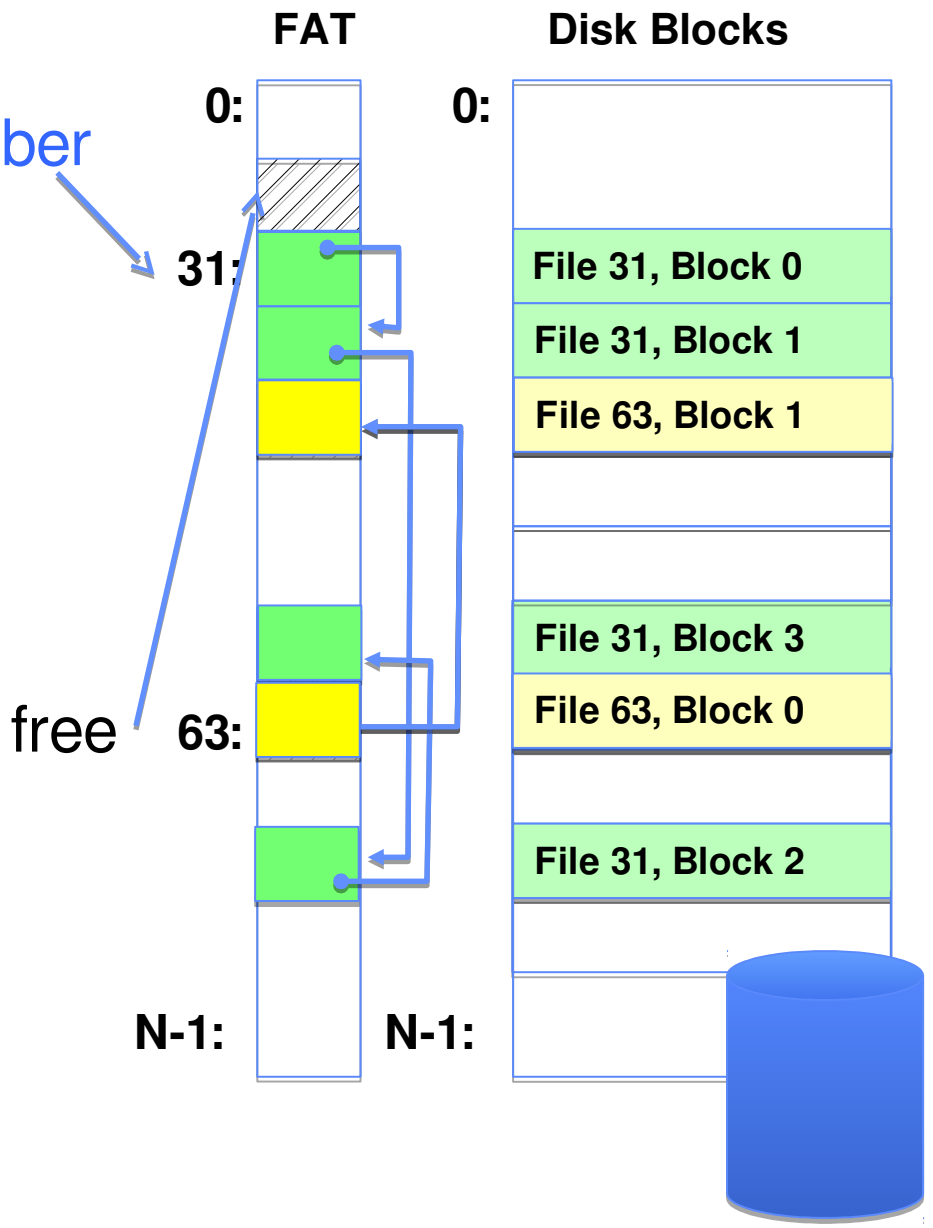
Used *all over the place*

- DOS
- Windows (sometimes)
- Thumb Drives

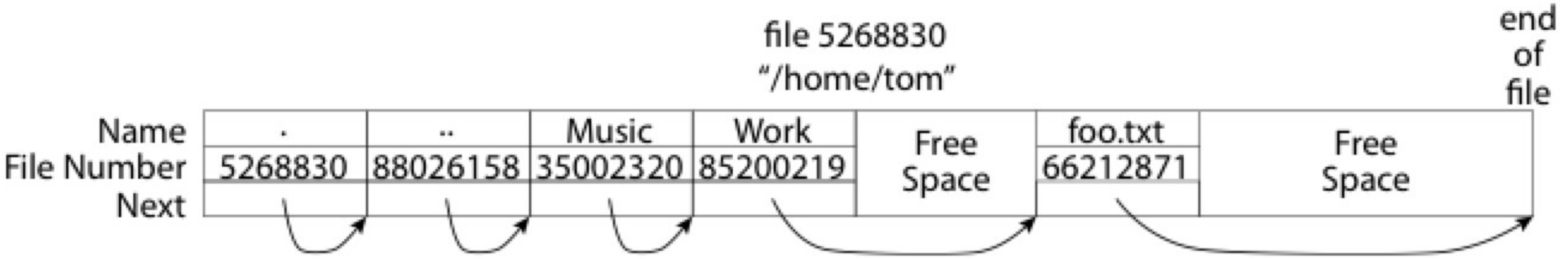
Really **simple**

Not much else in its favor...

- Random access slow
- File creation slow



What about the Directory?



File containing <file_name: file_number> mappings

Free space for new entries

In FAT: attributes kept in directory (!!!)

Each directory a **linked list** of entries

Where do you find root directory ("/")?

Directory Structure (Con't)

How many disk accesses to resolve "/my/book/count"?

- Read in file header for root (fixed spot on disk)
- Read in first data block for root
  - Table of file name/index pairs. Search linearly – ok since directories *typically* very small
- Read in file header for "my"
- Read in first data block for "my"; search for "book"
- Read in file header for "book"
- Read in first data block for "book"; search for "count"
- Read in file header for "count"

Current working directory: Per-address-space pointer to a directory (file #) used for resolving file names

- Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

Big FAT security holes

FAT has no access rights

FAT has no header in the file blocks

Just gives and index into the FAT

- (file number = block number)

Designing Better Filesystems

Question: What will they be used for?

Empirical Characteristics of Files

Most files are small

Most of the space is occupied by the rare big ones

A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

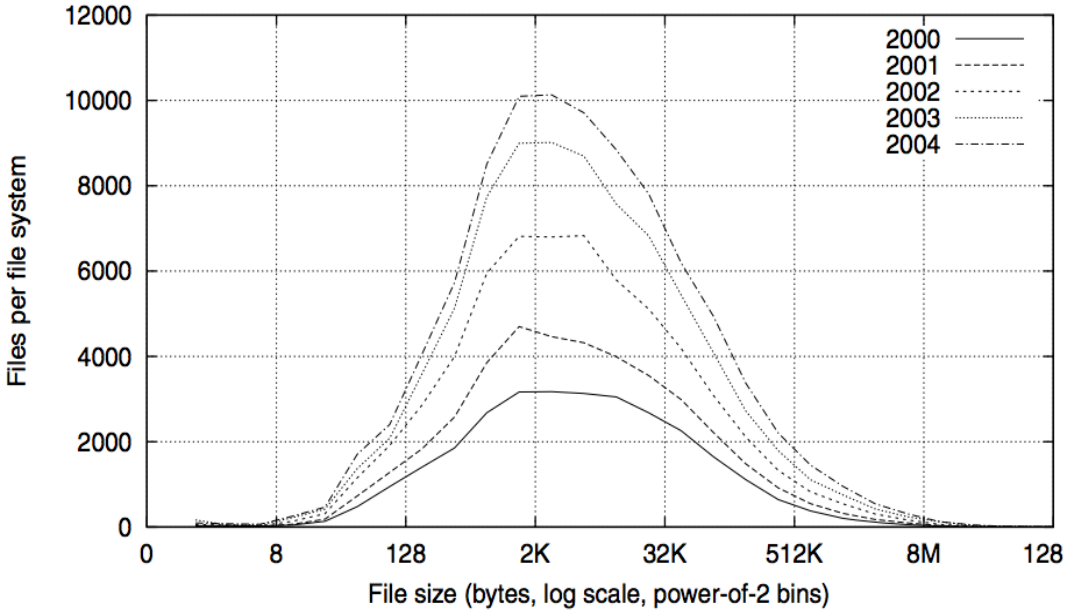


Fig. 2. Histograms of files by size.

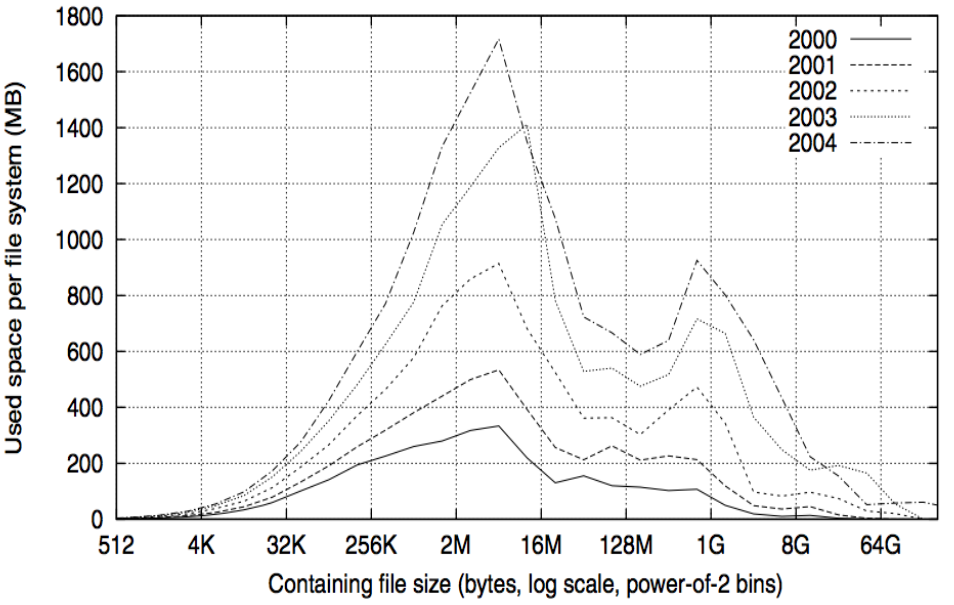
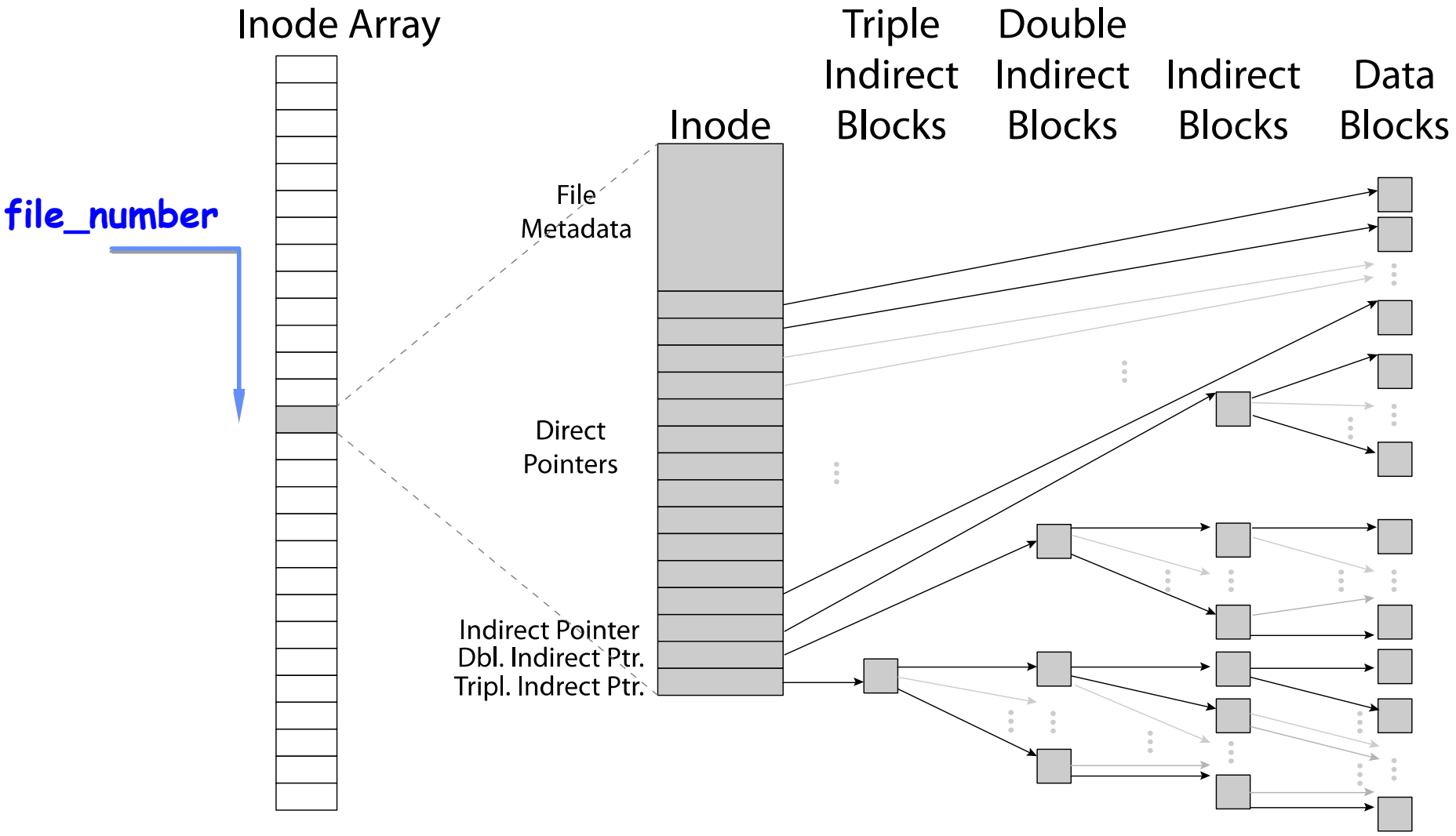


Fig. 4. Histograms of bytes by containing file size.

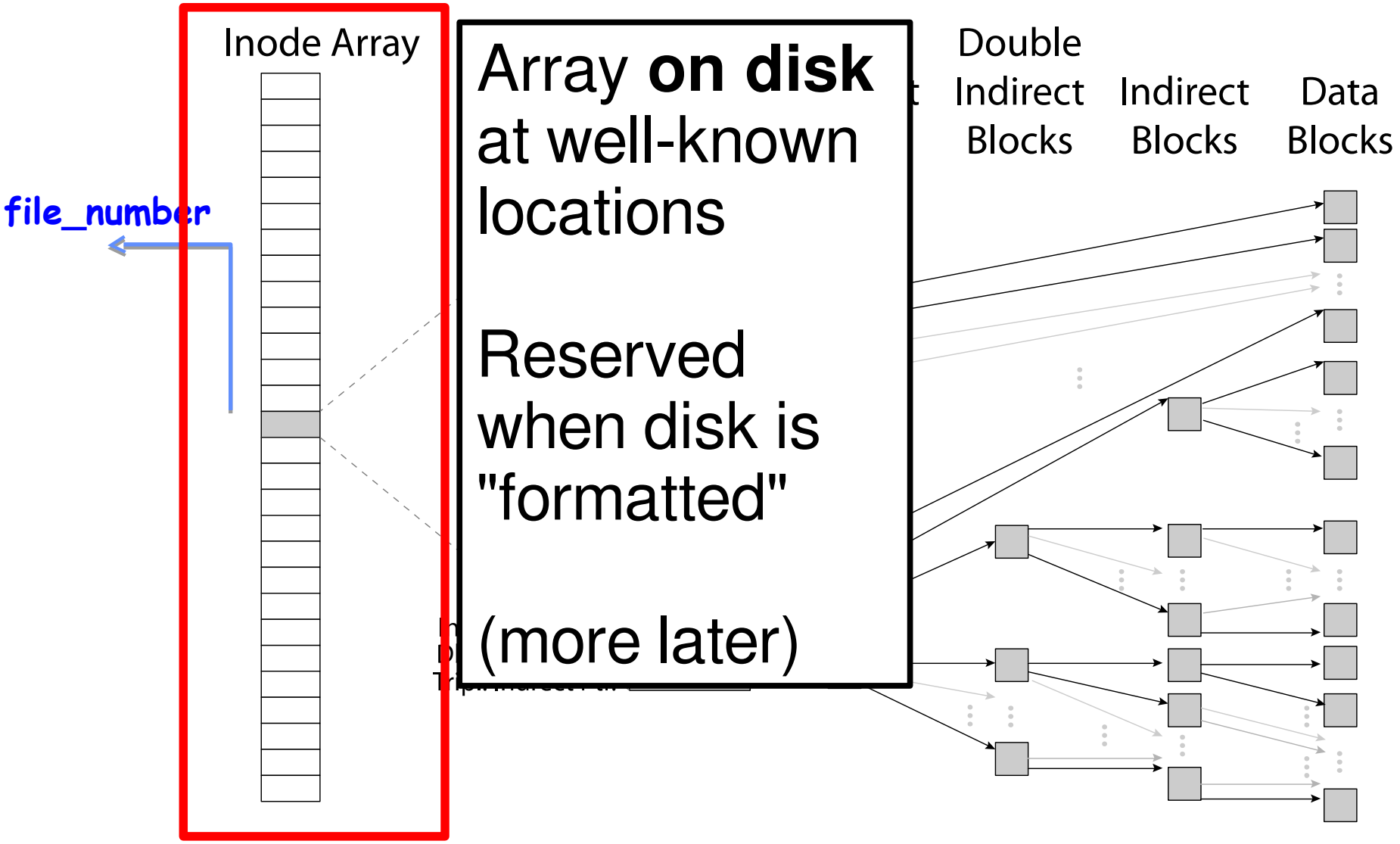
So what about a “real” file system

Meet the inode



So what about a “real” file system

Meet the inode



BSD Fast File System (1)

Original inode format appeared in BSD 4.1

- Berkeley Standard Distribution Unix

File Number is index into **inode arrays**

Multi-level index structure

- Great for little to large files
- Asymmetric tree with fixed sized blocks

BSD Fast File System (2)

Metadata associated with the file

- Rather than in the directory that points to it

UNIX FFS: BSD 4.2: Locality Heuristics

- Attempt to allocate files contiguously
- Block group placement
- Reserve space

Scalable directory structure

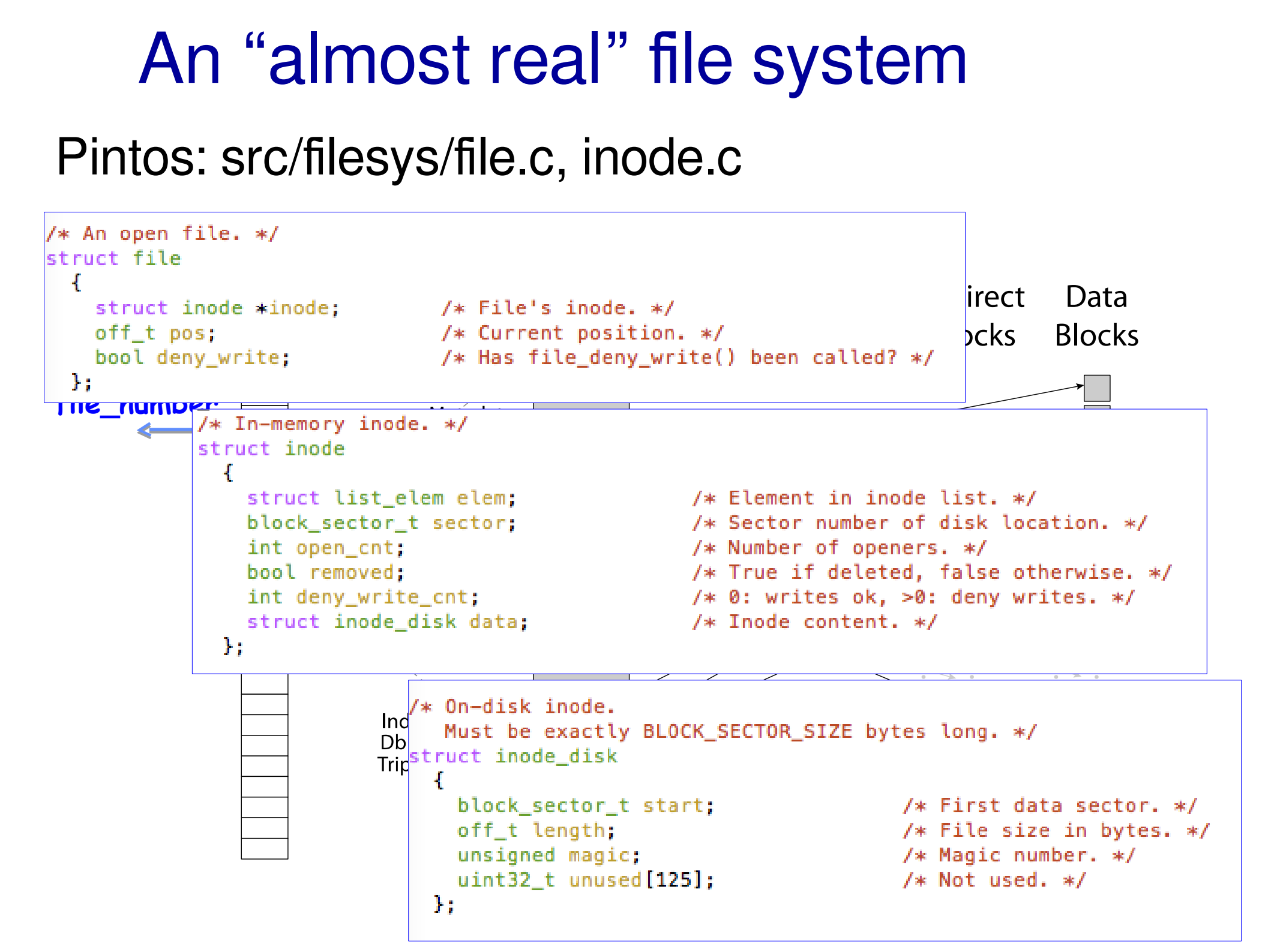
An “almost real” file system

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

```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? */
  };
```

```c
/* 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. */
  };
```

```c
/* 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. */
  };
```



How do you find an inode on disk?

Inode is, say, 128 bytes

Inode #1000 is **128 * 1000 bytes** into inode array

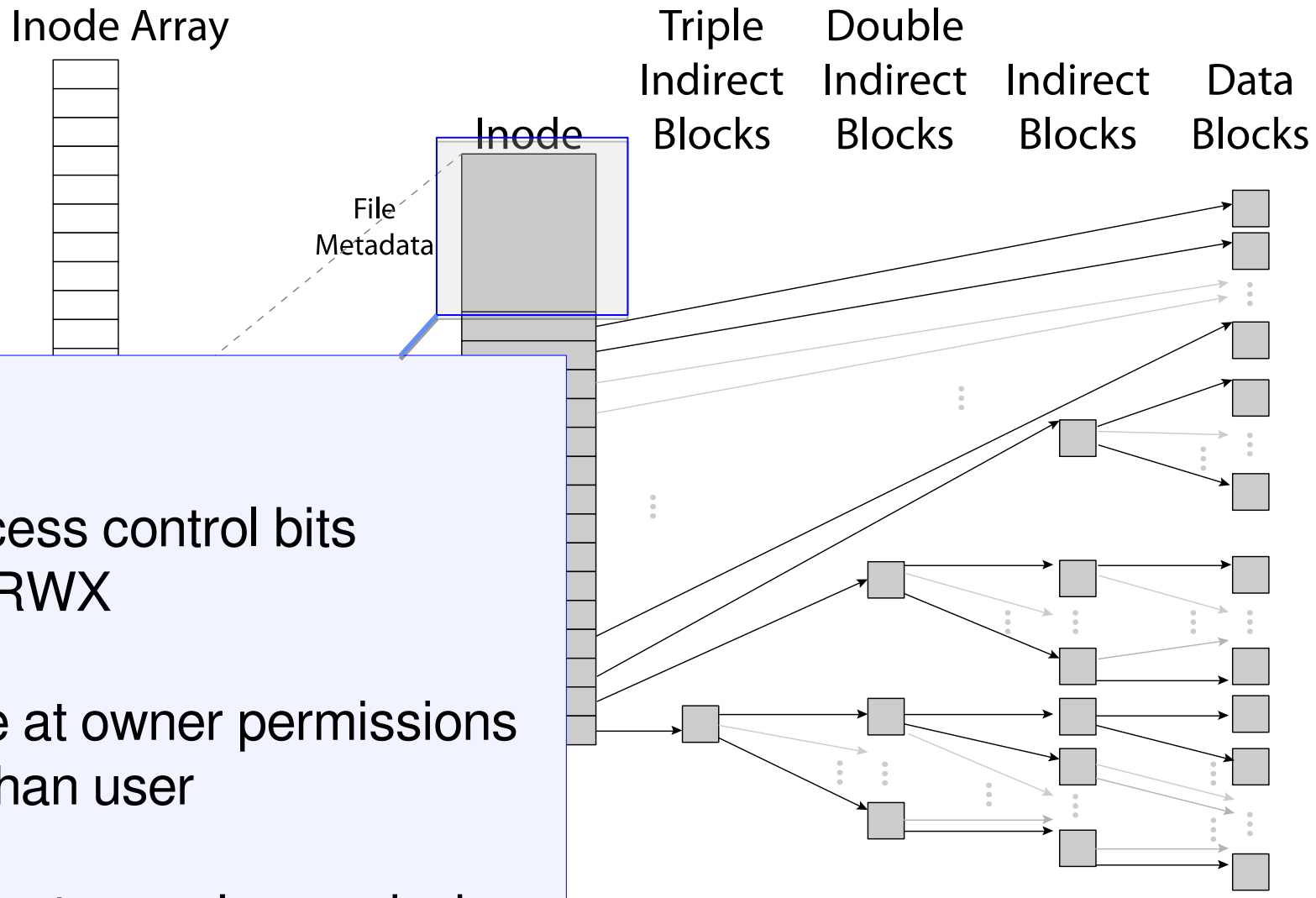
Inode array is **fixed location on disk**

- E.g. starts at block #2

Blocks are 4KB? Inode #1000 is in the middle of block #33 (= 2 + 128 * 1000 / 4096)

FFS: File Attributes

Inode metadata – stored **within** inode



User
Group
9 basic access control bits
- UGO x RWX

Setuid bit
- execute at owner permissions
- rather than user

Getgid bit
- execute at group's permissions

FFS: Data Storage

Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks: sufficient for files up to 48KB

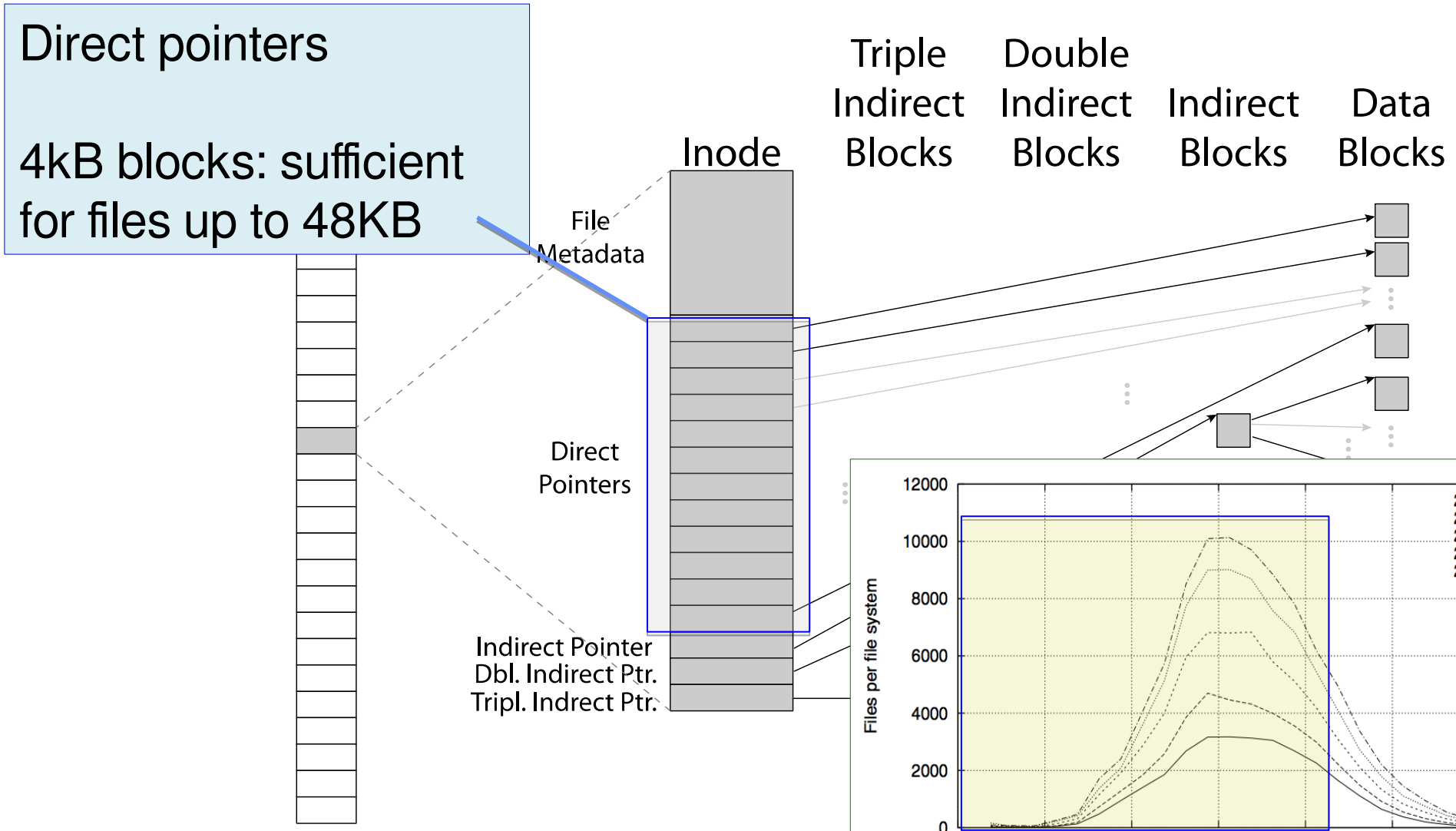


Fig. 2. Histograms of files by size.

FFS: Freespace Management

Bit vector with a bit per storage block

Stored at a **fixed location** on disk

Where are inodes stored? (1)

In early UNIX and DOS/Windows' FAT file system, headers/inodes stored in special array in **outermost cylinders**

- Outermost because fastest to read
- Fixed size, **set when disk is formatted.**

Where are inodes stored? (1)

In early UNIX and DOS/Windows' FAT file system, headers/inodes stored in special array in **outermost cylinders**

- Outermost because fastest to read
- Fixed size, **set when disk is formatted.**

How does OS know the size/location when booting up?

Written in *fixed location on every filesystem*
FFS: Called the "***superblock***"
(Sometimes backup copies in case of failure)

Where are inodes stored? (2)

In early UNIX and DOS/Windows' FAT file system, headers/inodes stored in special array in **outermost cylinders**

Problem: How do you read a small file?

- Read its inode (outermost cylinders)
- Read its data – probably far away
- **Lots of seek time**

Logistics

Project 3 Design Reviews Tomorrow

- You should have already submitted your design doc

HW3 out

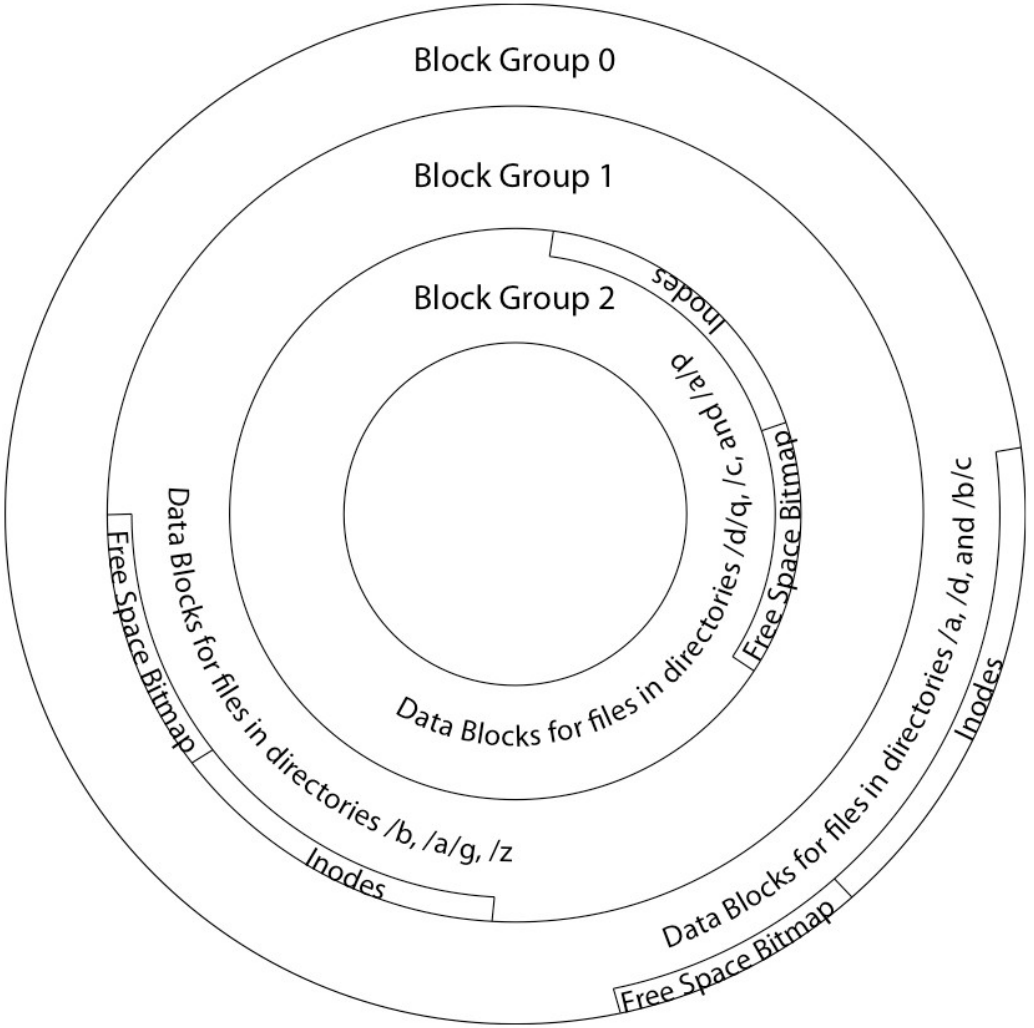
Break

Locality: Block Groups

File system volume is divided into a set of block groups

- **Close set of tracks**

Idea: Low seek times between inode/directories for a file and blocks of a file

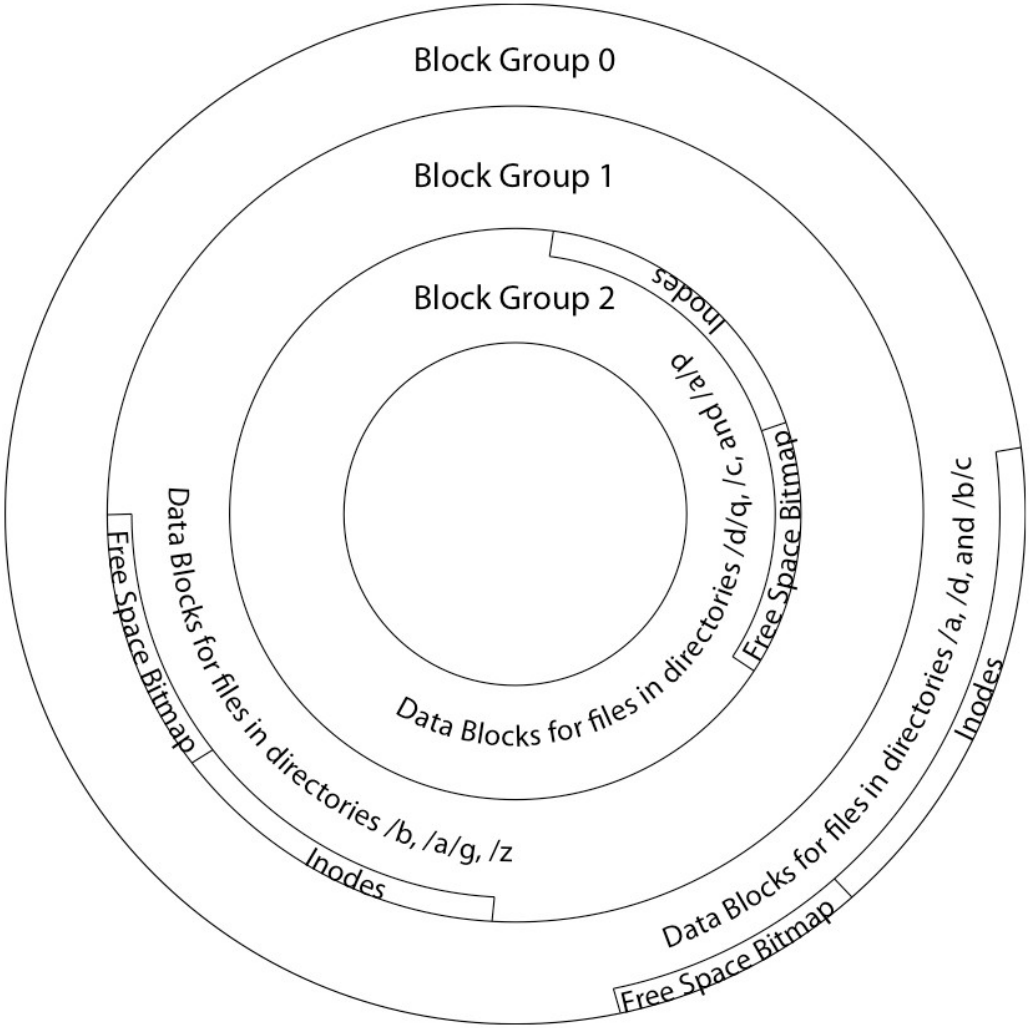


Locality: Block Groups

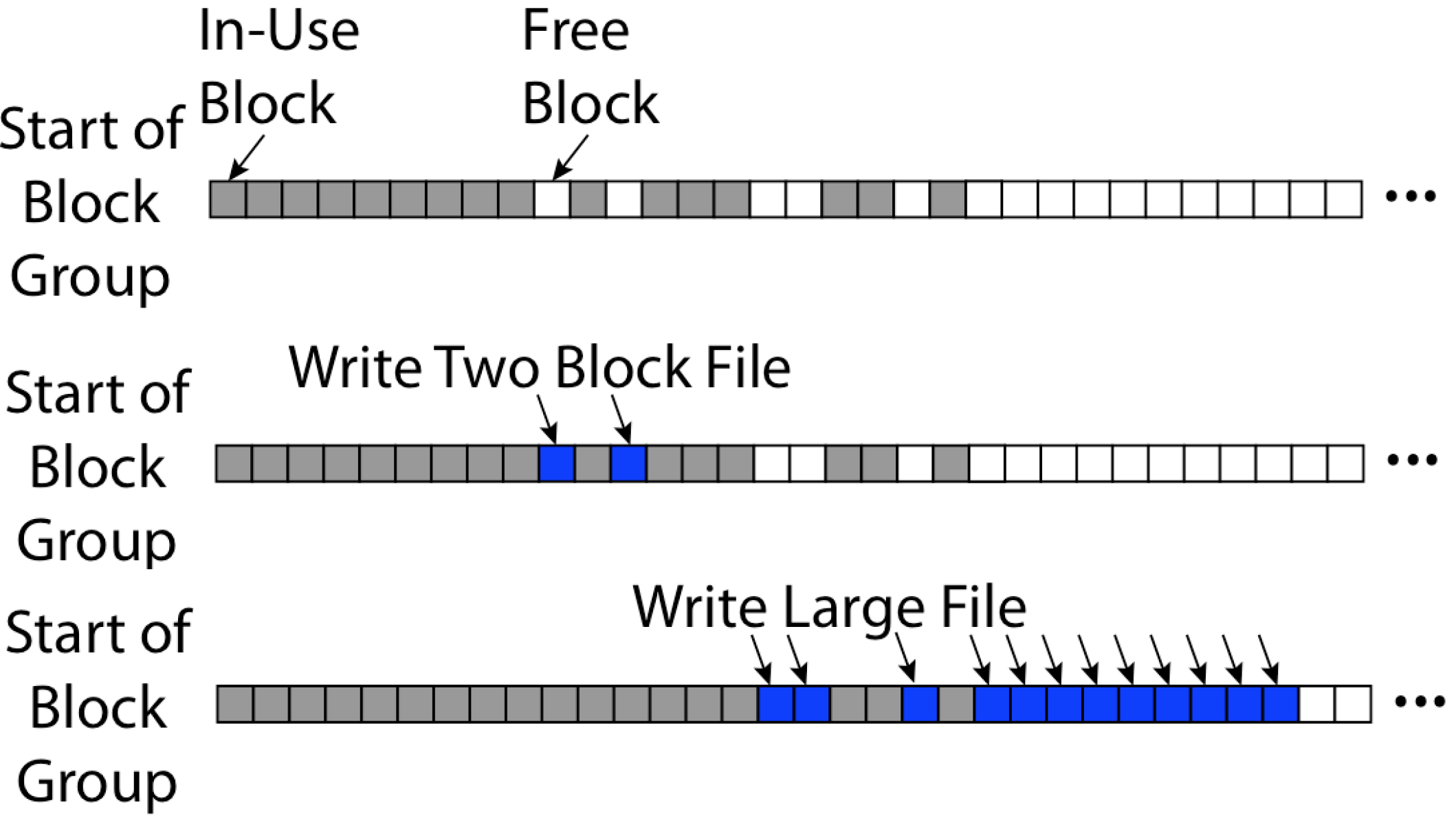
File data blocks, metadata, and free space are interleaved within block group

- No huge seeks

Put directory and its files in common block group



FFS First Fit Block Allocation



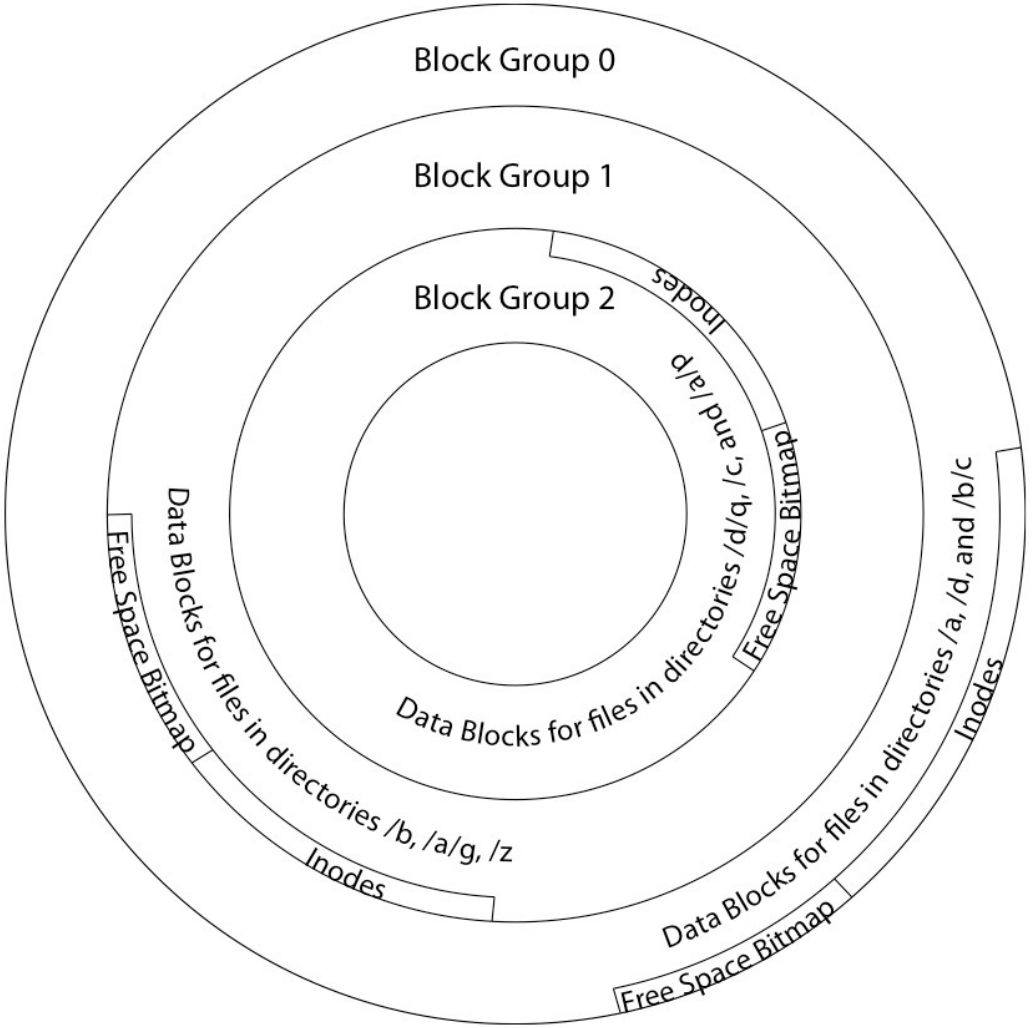
Locality: Block Groups

First-free allocation of new file block

- Few little holes at start, big sequential runs at end of group
- Sequential layout for big files

Reserve space in the BG

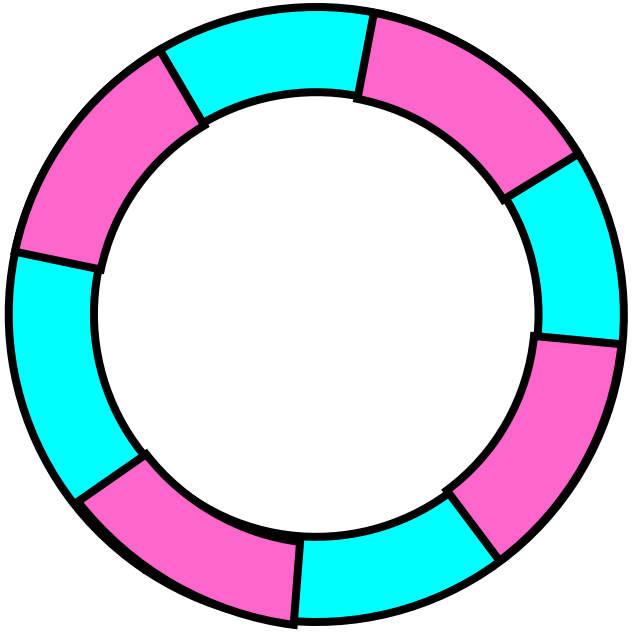
- 10%
- Makes sure there's sequential holes for big files
- Lets "first fit" be fast – likely to find something quickly



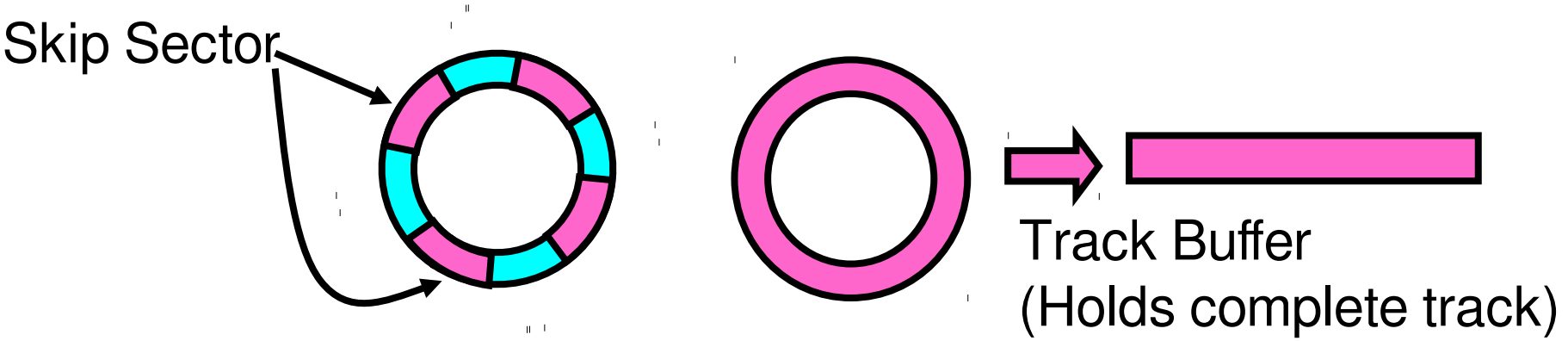
Attack of the Rotational Delay

Problem: Missing blocks due to rotational delay

- Read one block, do processing, and read next block.
- In meantime, disk has continued turning: missed next block!
- Need 1 revolution/block!



Attack of the Rotational Delay



Solution 1: Skip sector positioning ("interleaving")

- Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
- FFS did this

Solution 2: Read ahead: read next block right after first – **speculate** that it will be needed

- By OS (make larger request) – requires RAM to hold result of read
- By disk (*track buffers*) – requires RAM *in the controller*
- Most modern disk controllers do this

BSD Fast File System

Pros

- Efficient storage for both **small** and **large** files
- **Locality** for both small and large files
- **Locality** for metadata and data

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 (no way to say "blocks 1026-4085" – need to write out each block number)
- Need to reserve 10-20% of free space to prevent **fragmentation**

BSD Fast File System

Pros

- Efficient storage for both **small** and **large** files
- **Locality** for both small and large files
- **Locality** for metadata and data

Cons

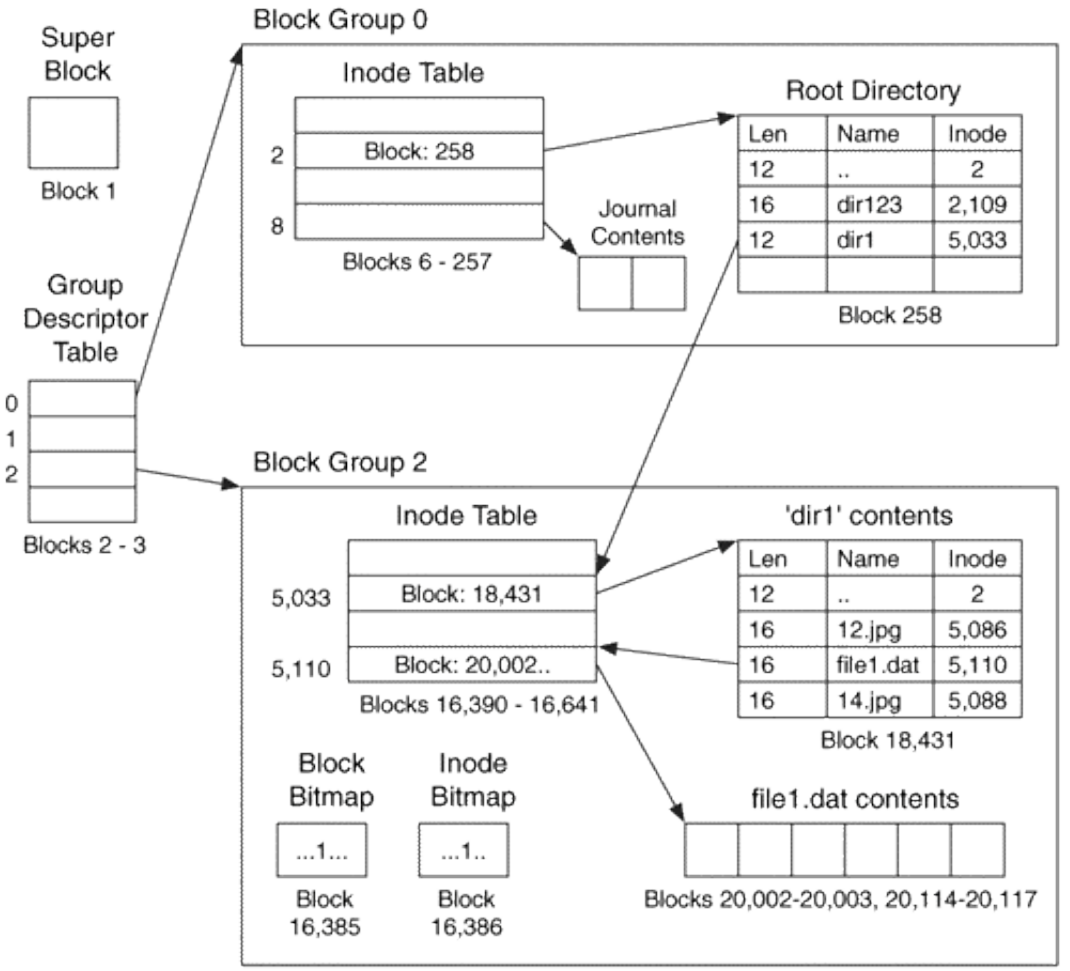
- Inefficient for **tiny files** (a 1 byte file requires both an inode and a data block)
- Inefficient encoding when file is mostly contiguous on disk (no way to say "blocks 1026-4085" – need to write out each block number)
- Need to reserve 10-20% of free space to prevent **fragmentation**

Linux Example: Ext2/3/4 Disk Layout

Disk divided into block groups
- Like FFS, provides locality
- Each group has two block-sized bitmaps (free blocks/inodes)

Actual Inode structure similar to 4.2BSD FFS
- with 12 direct pointers

Ext3: Ext2 w/Journaling
- Better durability – more on that later



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

A bit more on directories

Unix: Directories are ***just files***

Contents: **list of (file name, file number) pairs**

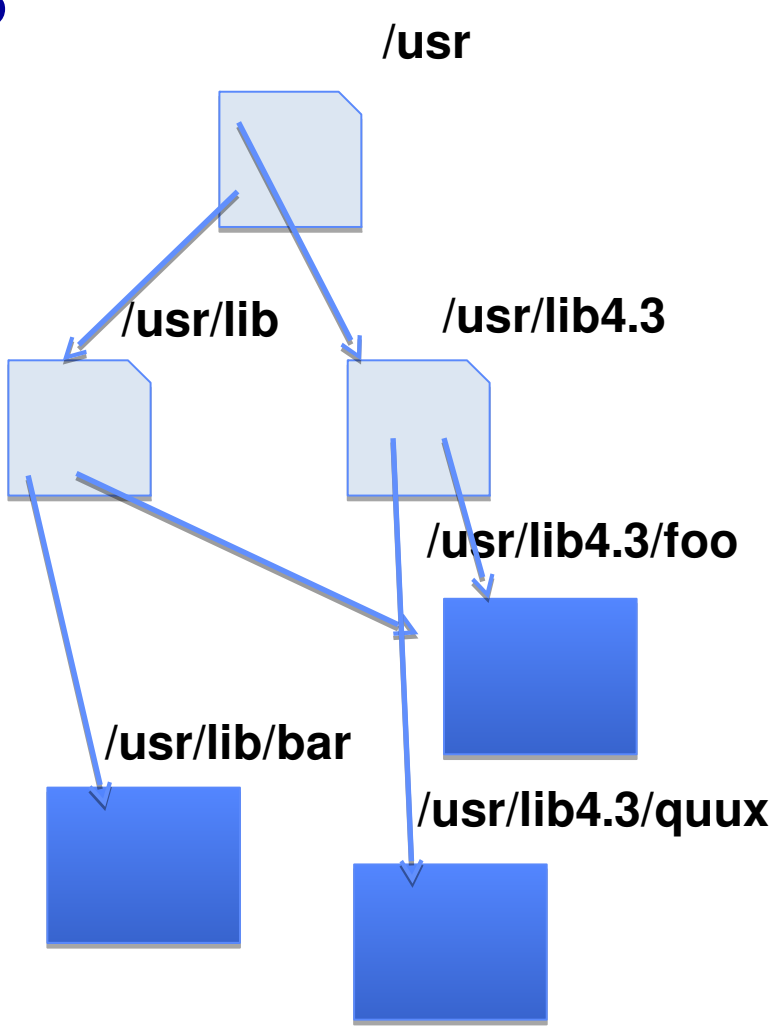
FFS: Linked list like FAT

Read/written through special system calls

- sometimes can read directly
- open/creat/mkdir/rmdir/link/unlink
- all system calls know how to traverse directories

What about attributes?

- Recall: Unix design – part of inode



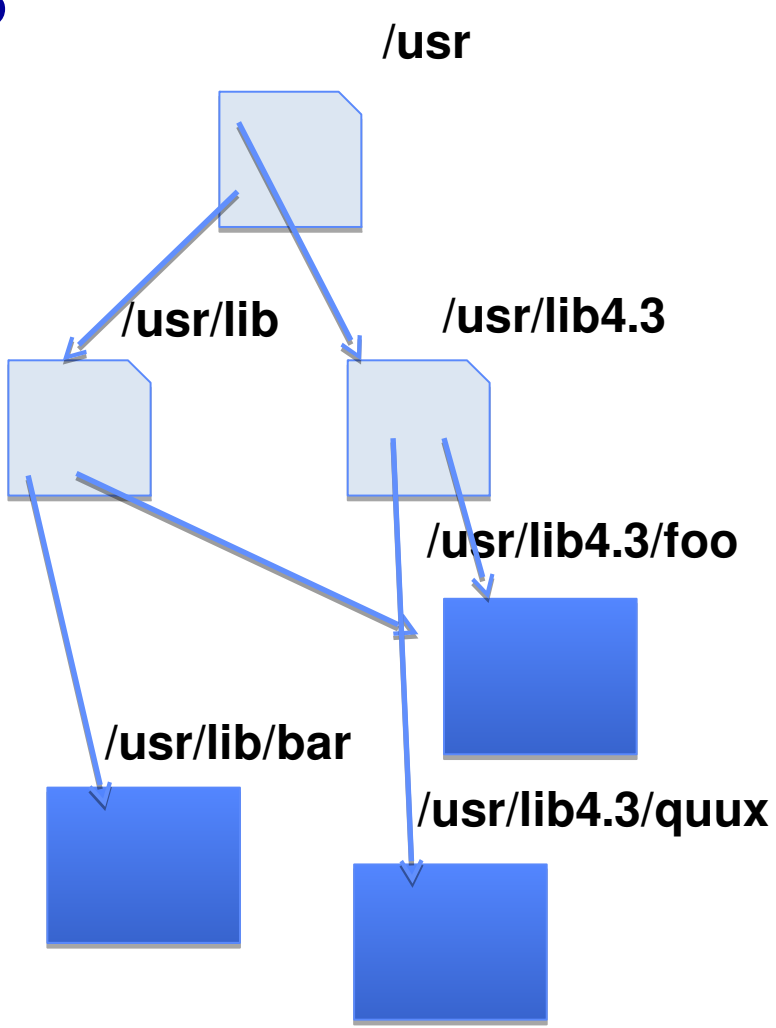
A bit more on directories

Unix: Directories are ***just files***

Contents: **list of (file name, file number) pairs**

libc support

- DIR * opendir (const char *dirname)
- int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)



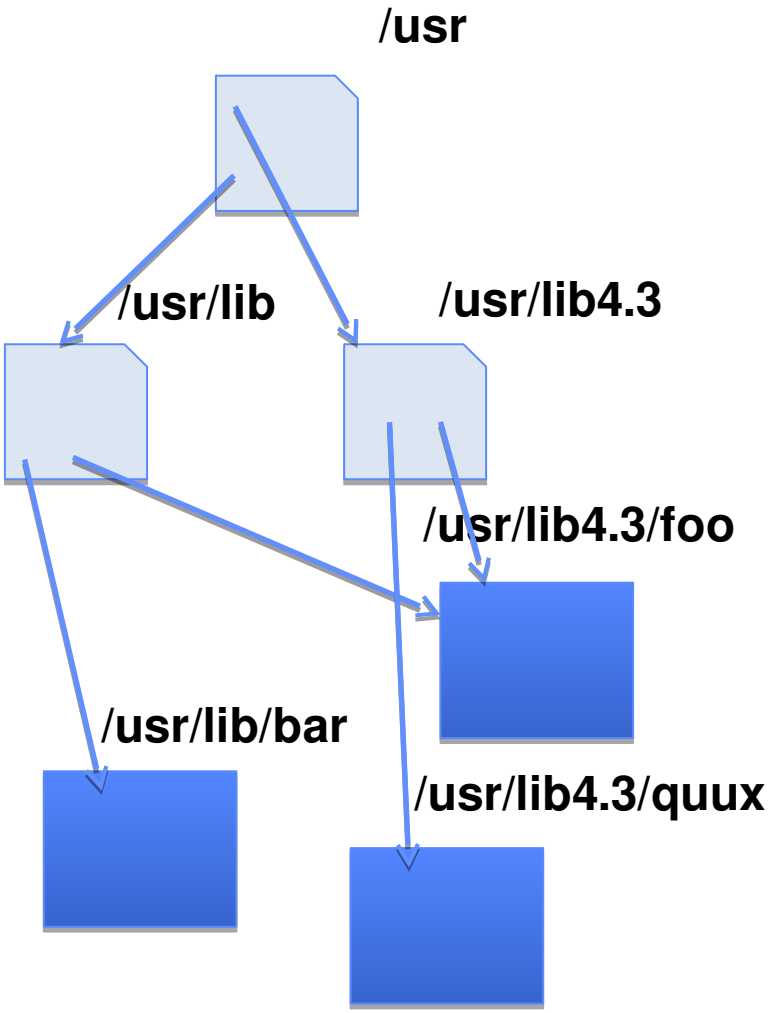
Hard Links (1)

Can we put the same file number in two directories?

- Unix: Yes!

```
/usr/lib:                 /usr/lib4.3:
   (bar, #1236)              (quux, #1236)
   (foo, #7823)              (foo, #7823)
   ...                       ...
```

Each "pointer" to a file is called a **hard link**



Hard Links (2)

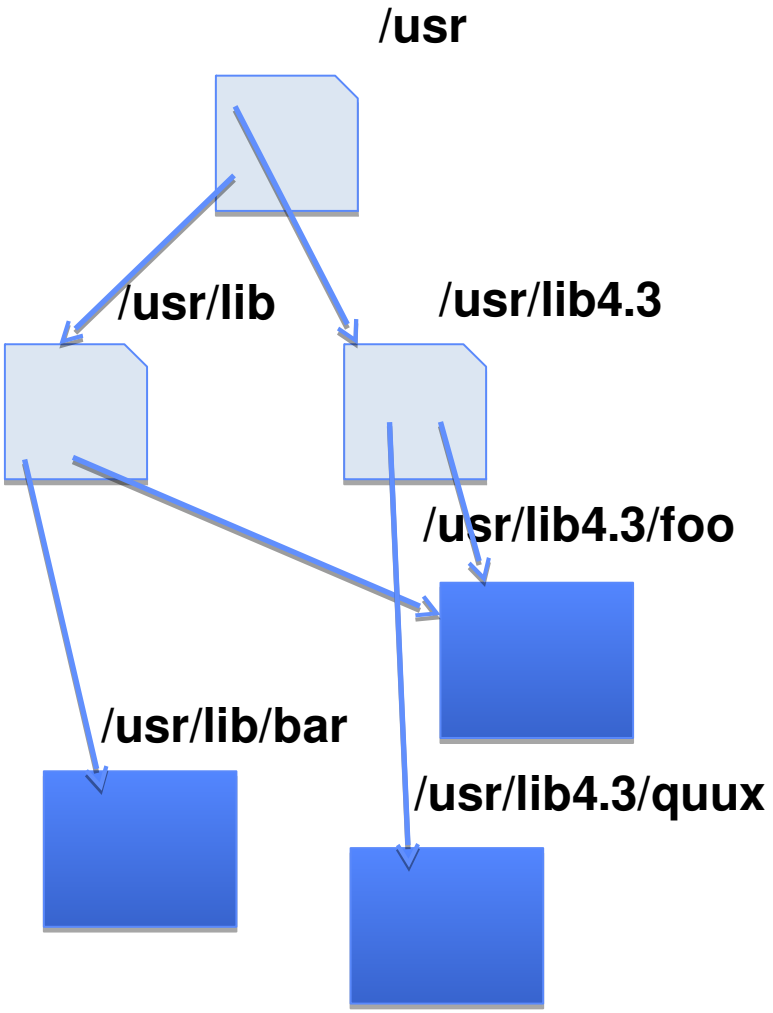
Create first hard link with open/creat

Unix: Create *extra* hard links with link() call:

- `link(`
  `"/usr/lib4.3/foo",`
  `"/usr/lib/foo"`
  `)`

Remove links with `unlink()`

- Metadata in inode: number of links
- When last link is removed, inode and blocks are actually freed



Soft Links AKA Symbolic Links

Different type of directory entry. Instead of

(source filename, file #)

have

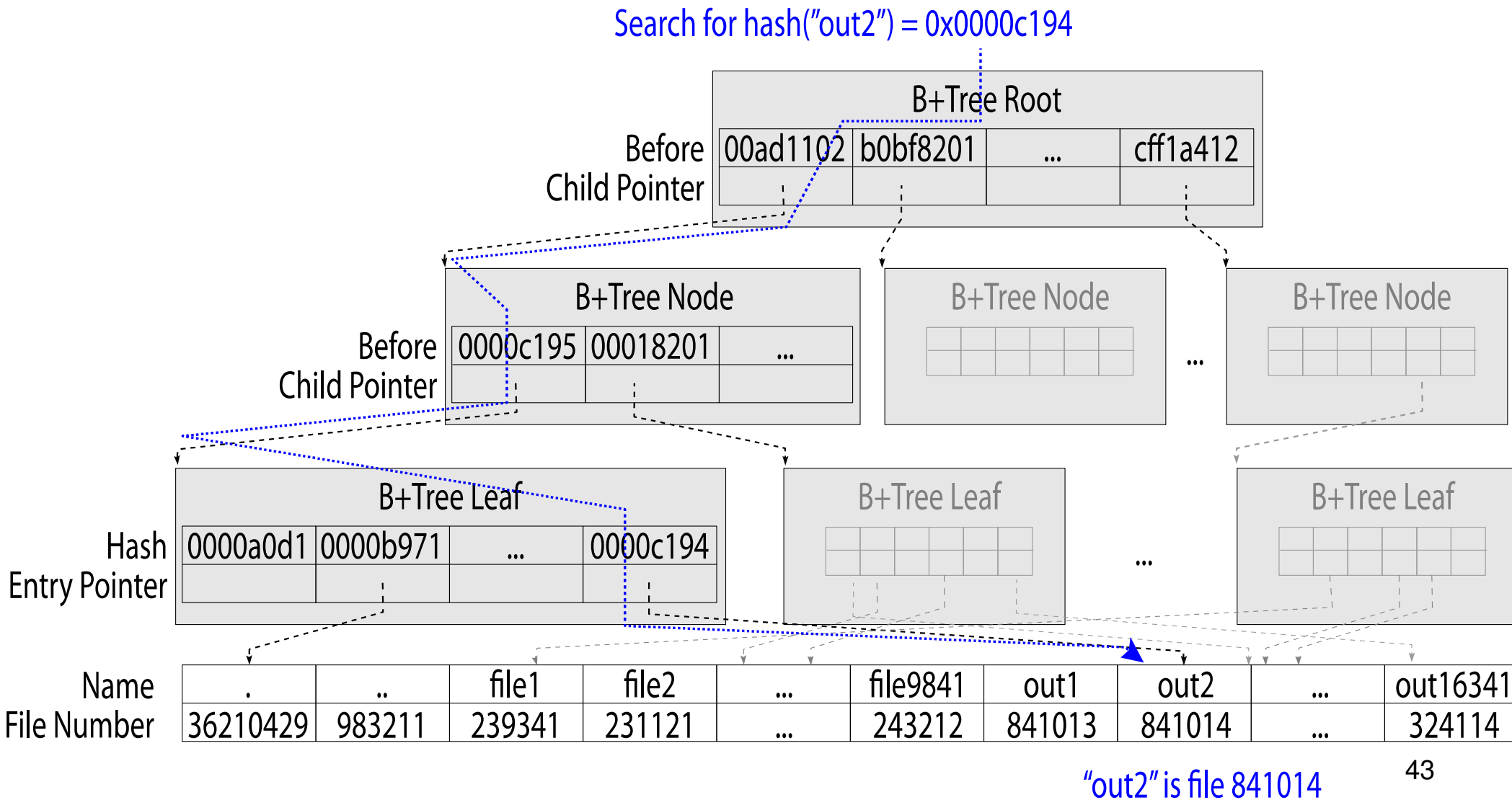
(source filename, *destination filename*)

OS looks up *destination filename* **each time** program accesses *source filename*

- Lookup can fail – open()/etc. returns error
- Lookup result can change without source directory changing
- Lookup can find a file on a different disk

Unix: create soft links with `symlink()`

Large Directories: B-Trees (dirhash)



B-Trees

Balanced trees suitable for storing on disk

Like balanced binary tree but not binary

- Why? Want big reads/writes – # children = size of disk read
- Sorted list of child nodes for each internal node

NTFS

New Technology File System (NTFS)

- Common on Microsoft Windows systems

NTFS: Master File Table

Analogous to inode array in FFS

Each 1KB entry stores attribute:value pairs

MFT entry for file has metadata plus

- the file's data directly (for small files) or
- a list of extents (start block, size) for the data or
- a list of extents and pointers to other MFT entries with more list of extents

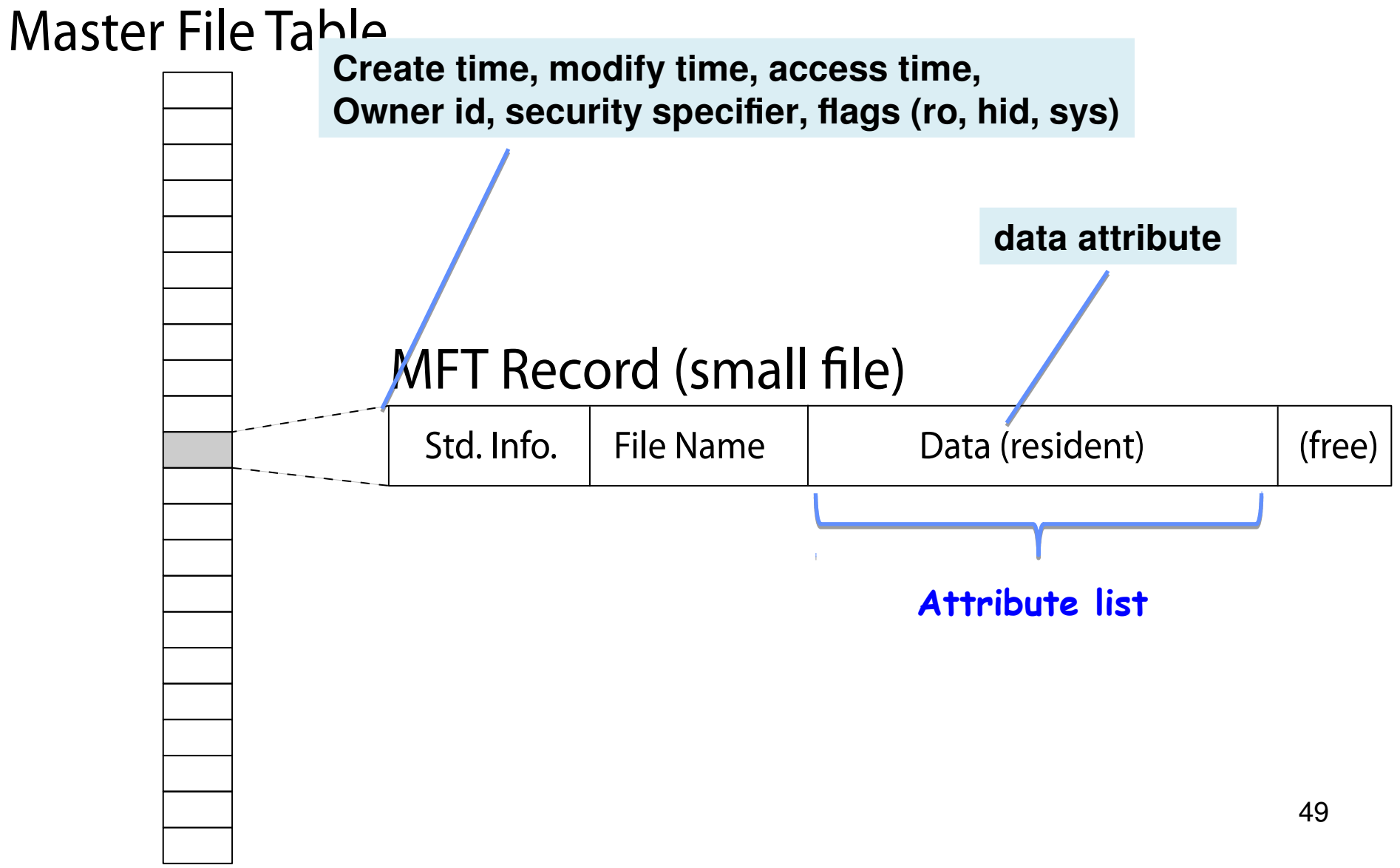
NTFS: Master File Table

Analogous to inode array in FFS

Each 1KB MFT entry stores metadata and

- **the file's data directly (for small files) or**
- a list of extents (start block, size) for the data or
- a list of extents and pointers to other MFT entries with more list of extents

NTFS Small File



NTFS: Master File Table

Analogous to inode array in FFS

Each 1KB MFT entry stores metadata and

- the file's data directly (for small files) or
- **a list of extents (start block, size) for the data or**
- a list of extents and pointers to other MFT entries with more list of extents

Extents

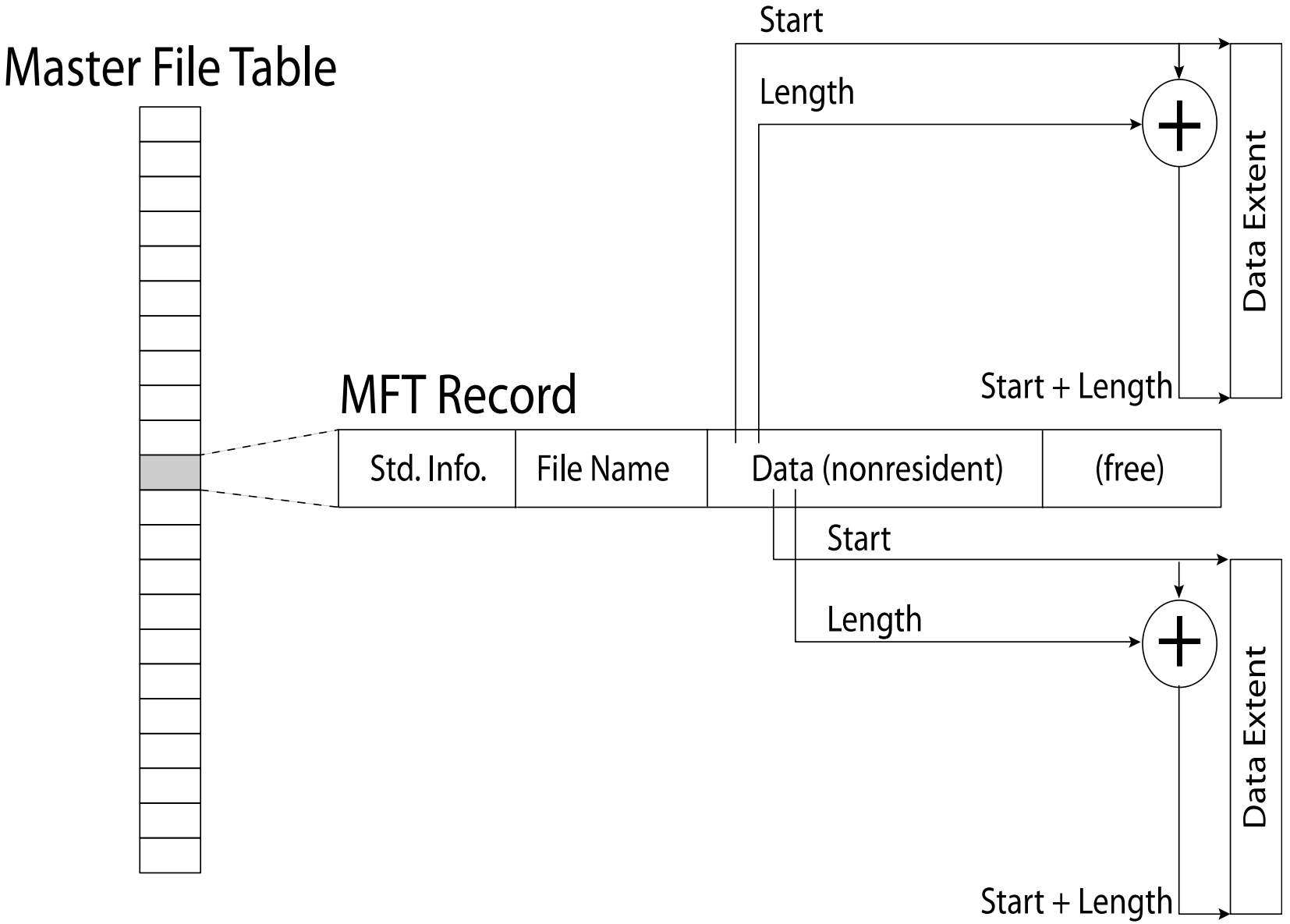
FFS: List of *fixed sized blocks*

For larger files, want to have file be contiguous anyways

Idea: Store start block + size

- 1000 block file? 1 metadata entry instead of 1000 (including indirect pointer!)

NTFS Medium File



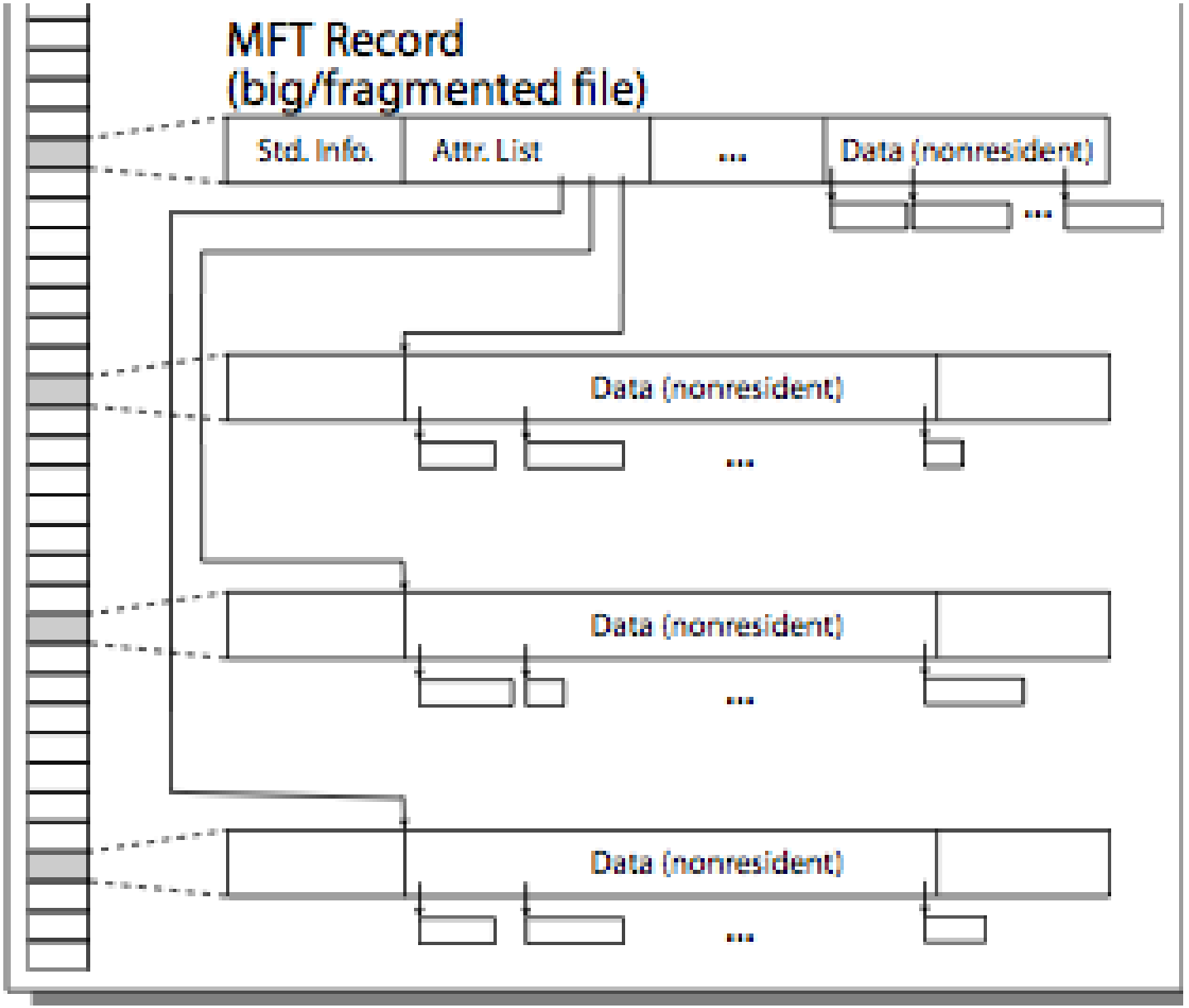
NTFS: Master File Table

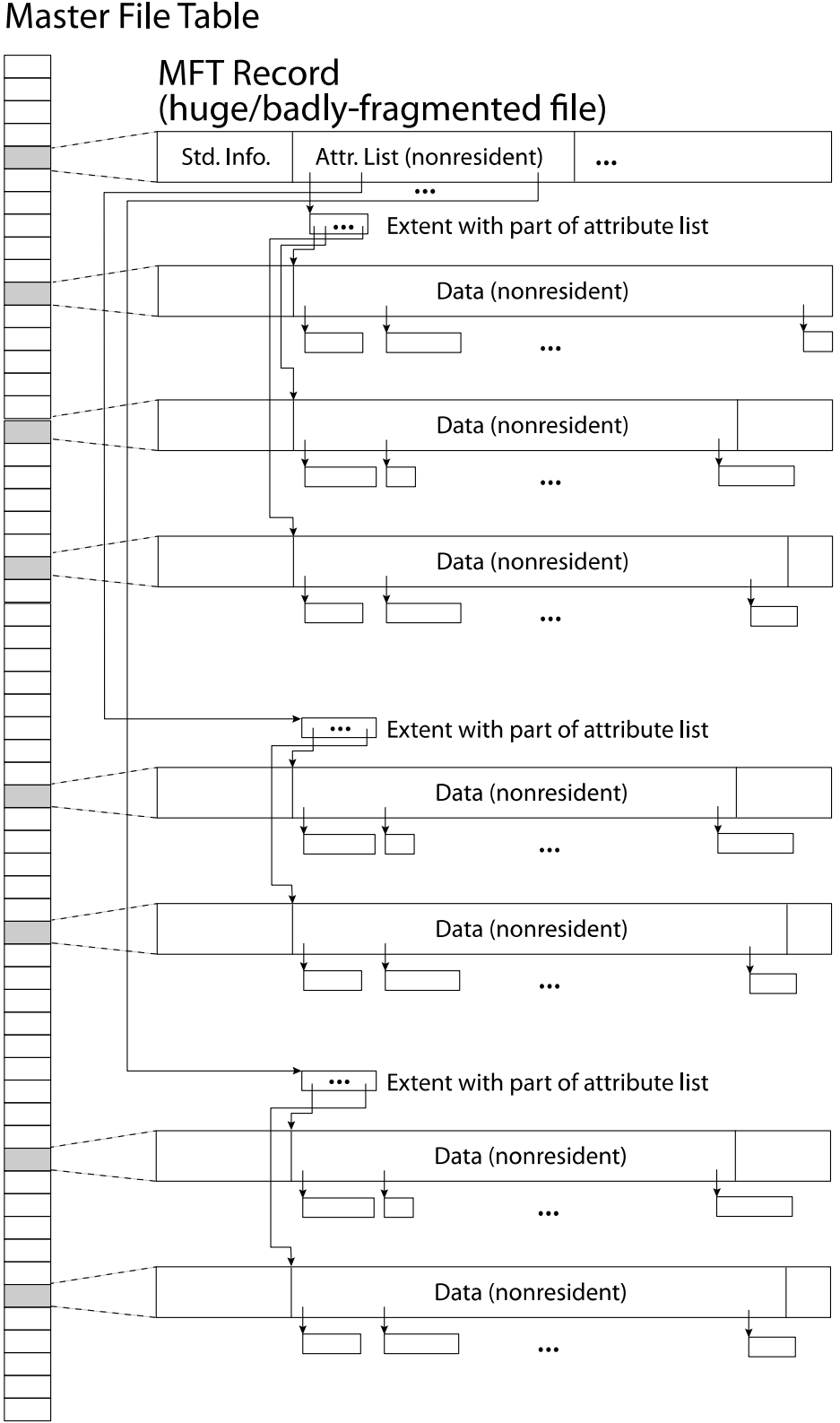
Analogous to inode array in FFS

Each 1KB MFT entry stores metadata and

- the file's data directly (for small files) or
- a list of extents (start block, size) for the data or
- **a list of extents and pointers to other MFT entries with more lists of extents**

NTFS Multiple Indirect Blocks




Master File Table
MFT Record
(huge/badly-fragmented file)
Std. Info.
Attr. List (nonresident)
...
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)

NTFS and Filenames

Recall: Unix – filenames only in directories

NTFS: FIlename

File System Summary (1)

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”

File System Summary (2)

Naming: act of translating from user-visible names to actual system 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 uses variable extents, rather than fixed blocks, and tiny files data is in the header

File System Summary (3)

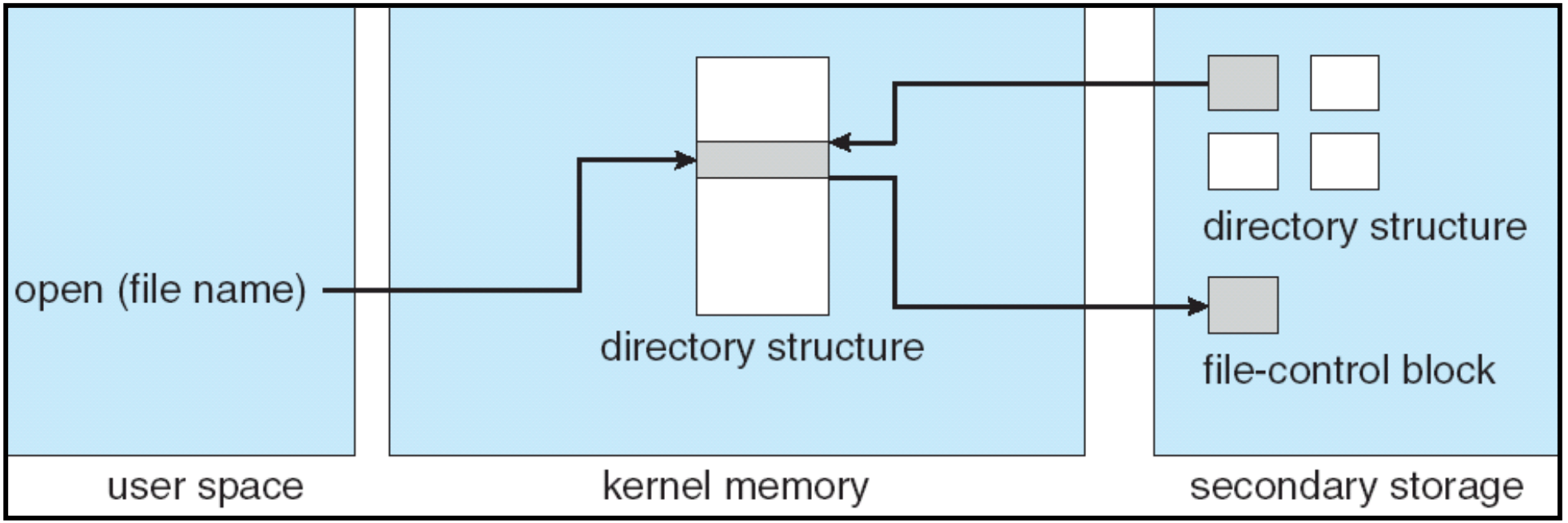
File layout driven by freespace management

- Integrate freespace, inode table, file blocks and directories into block group

Deep interactions between memory management, file system, and sharing

Filesystems in Memory

In-Memory File System Structures



Open system call:

- Resolves file name, finds file control block (inode)
- Makes entries in per-process and system-wide tables
- Returns index (called “file descriptor”) in open-file table

Memory Mapped Files

Traditional I/O involves explicit transfers between buffers in process address space to regions of a file

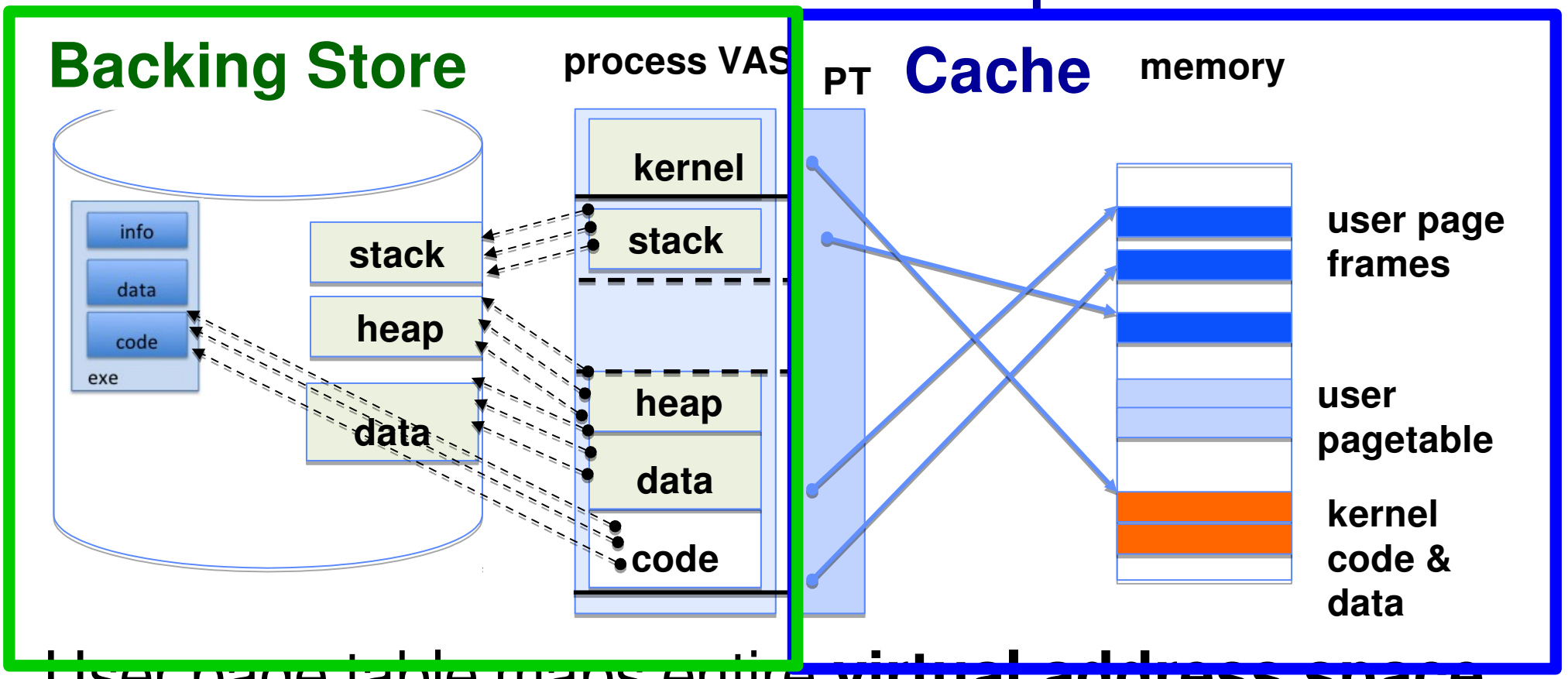
- This involves multiple copies into caches in memory, plus system calls

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 file is treated this way when we exec the process!!

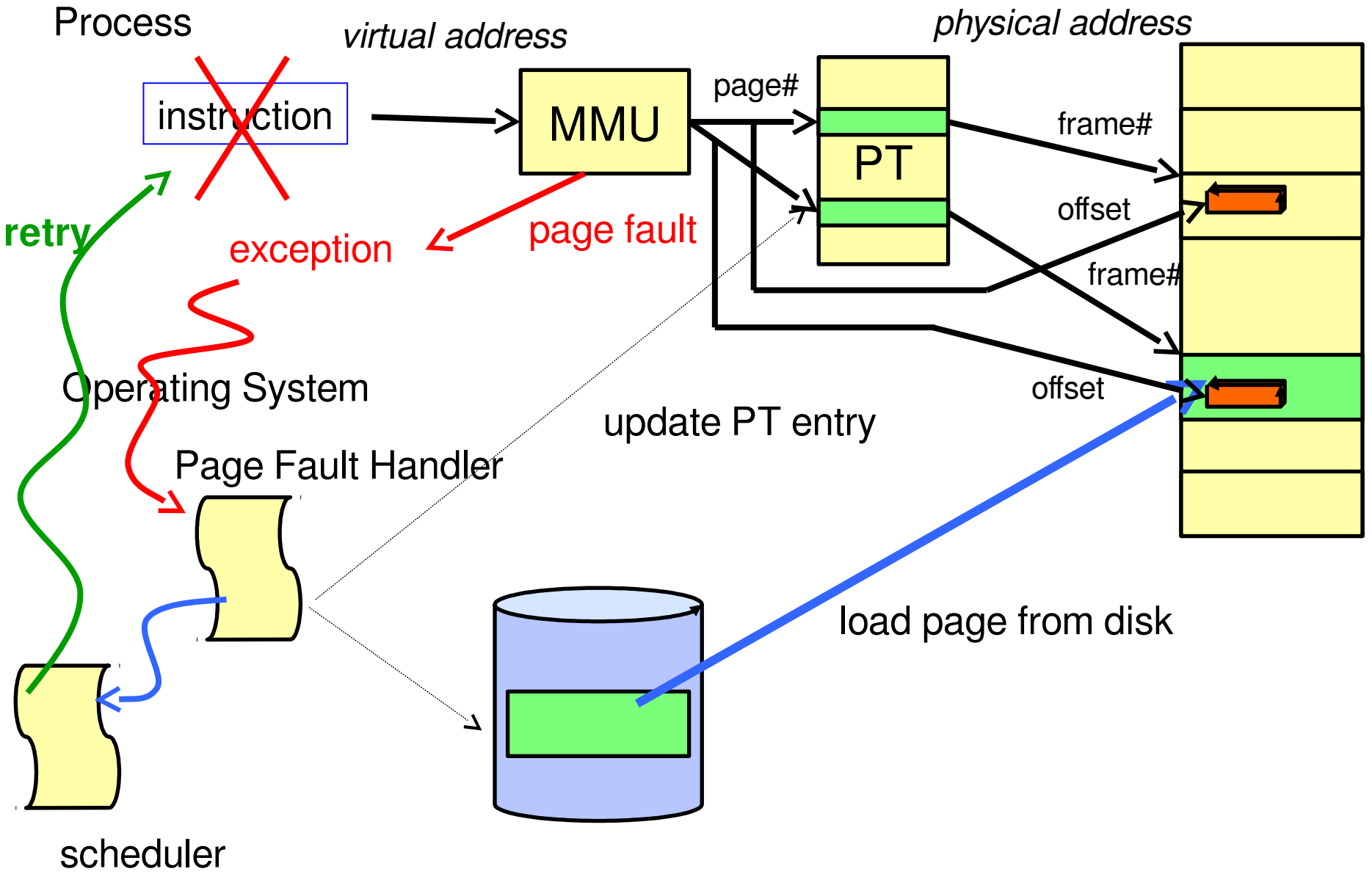
Recall: Create Address Space



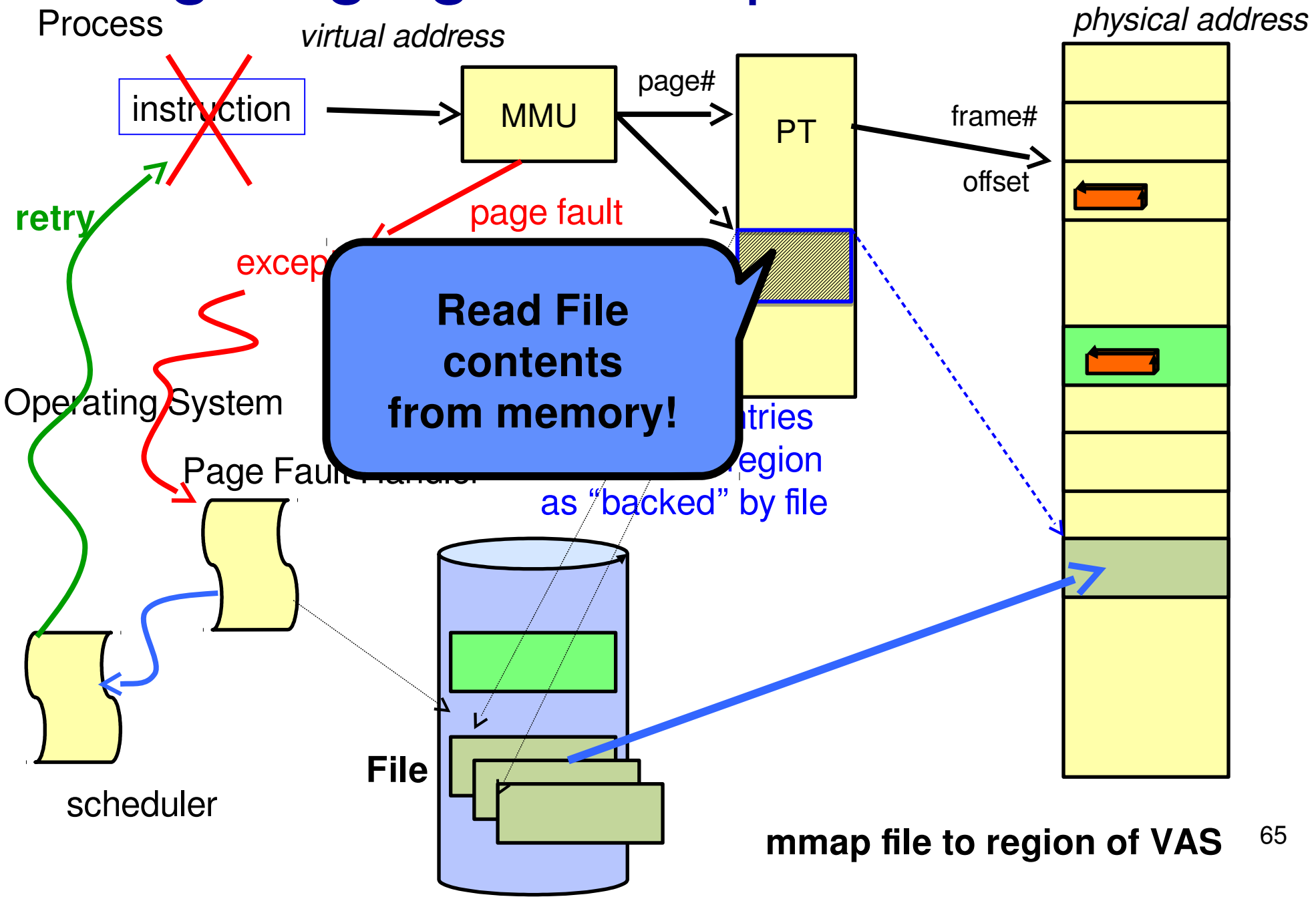
User page table maps entire **virtual address space**

- Only **resident** pages present
- One-to-one correspondence to OS's mapping

Recall: Demand Paging



Using Paging to mmap files



mmap file to region of VAS

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 *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
```

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 example

```
#include <sys/mman.h>

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

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CREATE);
  if (myfd < 0) { perror((“open failed!”);exit(1); }

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED,
               myfd, 0);
  if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

  printf("mmap at : %p\n", mfile);

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

Sharing through Mapped Files