CS162
Operating Systems and Systems Programming
Lecture 19

File Systems (Con't), MMAP, Buffer Cache

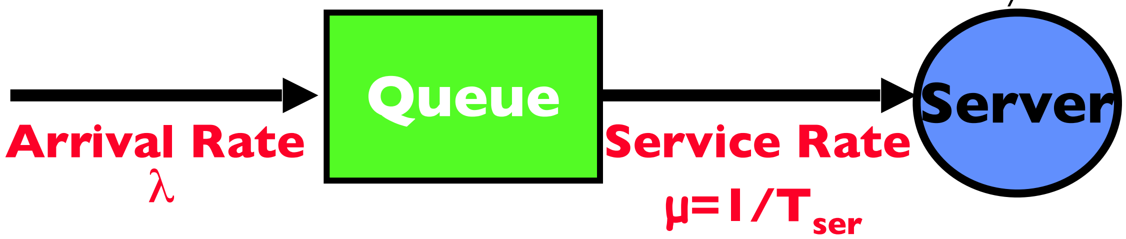
April 7th, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. *When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

Recall: A Little Queuing Theory: Some Results

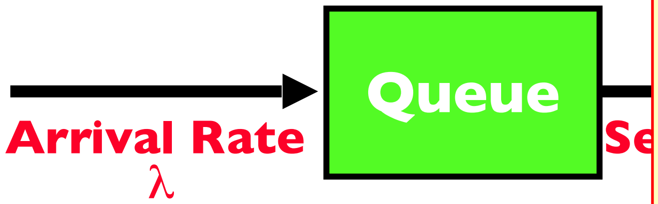
- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution (C = 1): (an "M/M/1 queue"):
    - » $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server (an "M/G/1 queue"):
    - » $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$

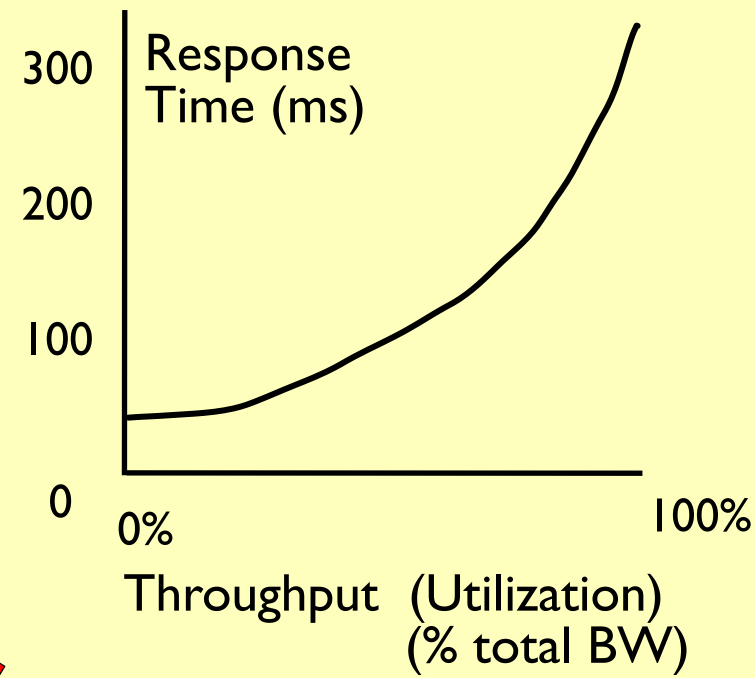
Recall: A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the q
  - Time between successive arrivals is ranc

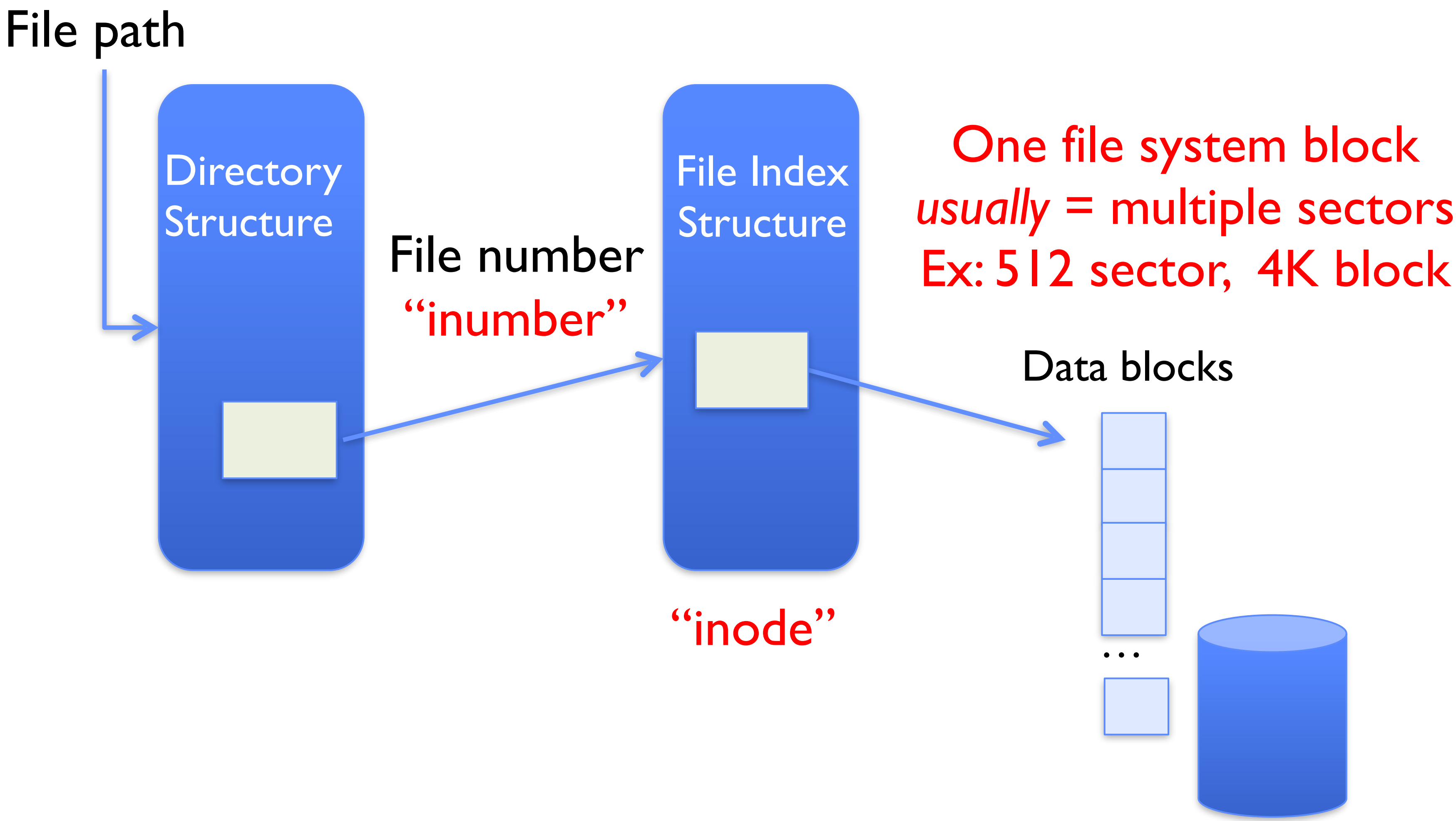


- Parameters that describe our system:
  - λ: mean number of arriving custo
  - T_{ser}: mean time to service a custor
  - C: squared coefficient of variance
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): u =
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by
- Results:
  - Memoryless service distribution (C = 1): (an "M/M/1 queue"):
    - » $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server (an "M/G/1 queue"):
    - » $T_q = T_{ser} \times ½(1+C) \times u/(1 - u)$

Why does response/queueing delay grow unboundedly even though the utilization is < 1 ?



Components of a File System



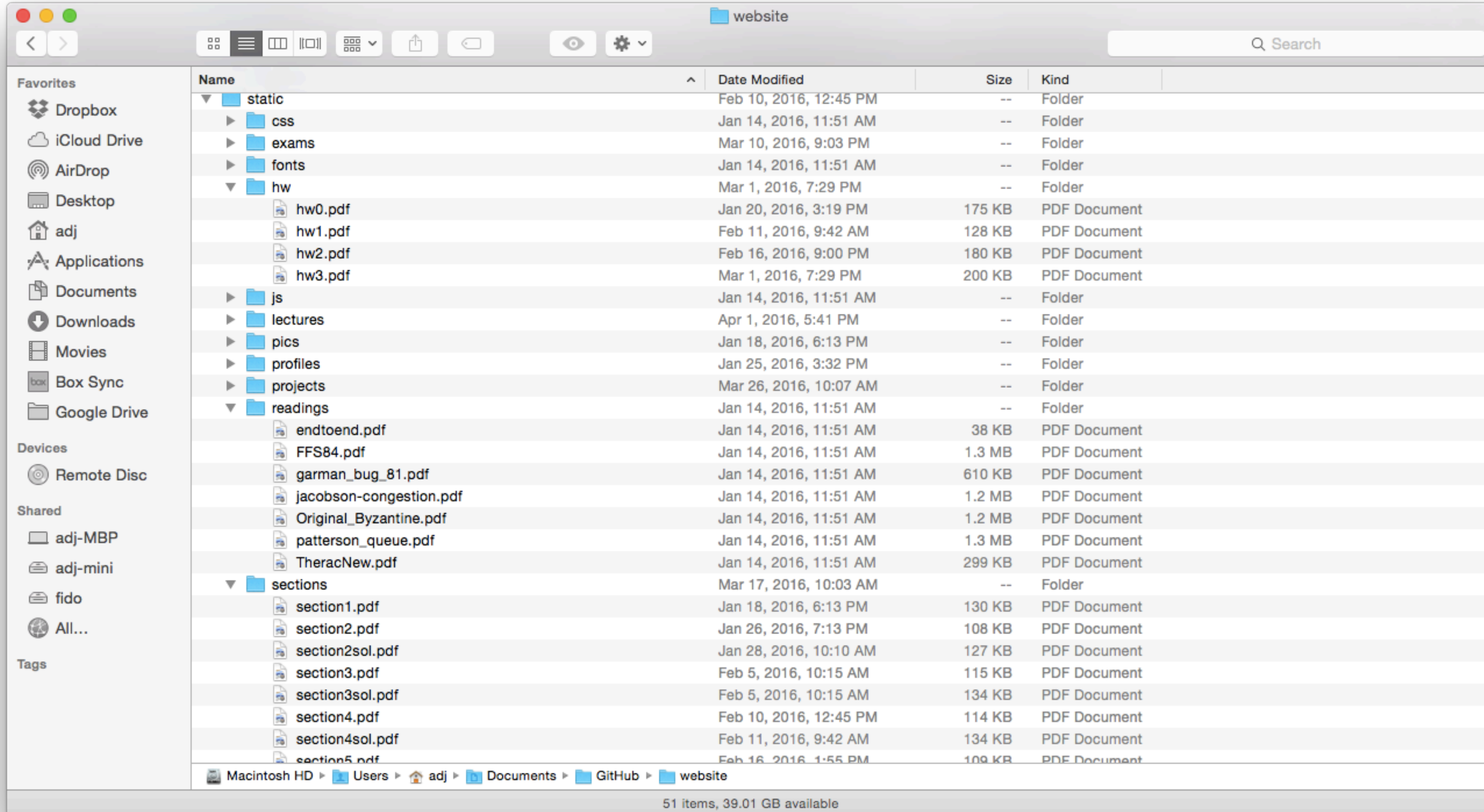
Components of a file system

file name offset → directory → *file number offset* → Index structure → *Storage block*

- Open performs *Name Resolution*
  - Translates pathname into a "file number"
    - » Used as an "index" to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a "handle" (another integer) to user process

- Read, Write, Seek, and Sync operate on handle
  - Mapped to file descriptor and to blocks

Directories



Directory

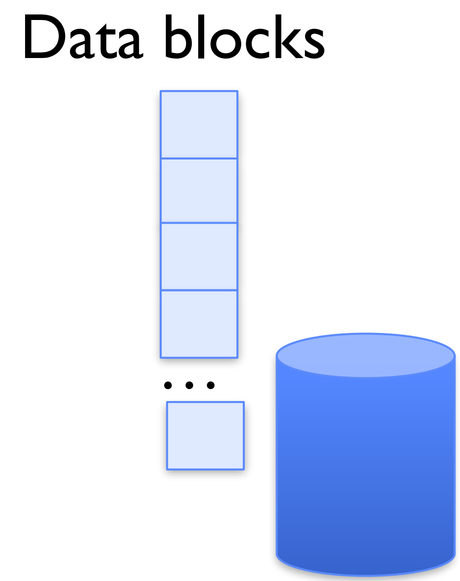
- Basically a hierarchical structure
- Each directory entry is a collection of
  - Files
  - Directories
    - » A link to another entries
- Each has a name and attributes
  - Files have data
- Links (hard links) make it a DAG, not just a tree
  - Softlinks (aliases) are another name for an entry

Directory Structure

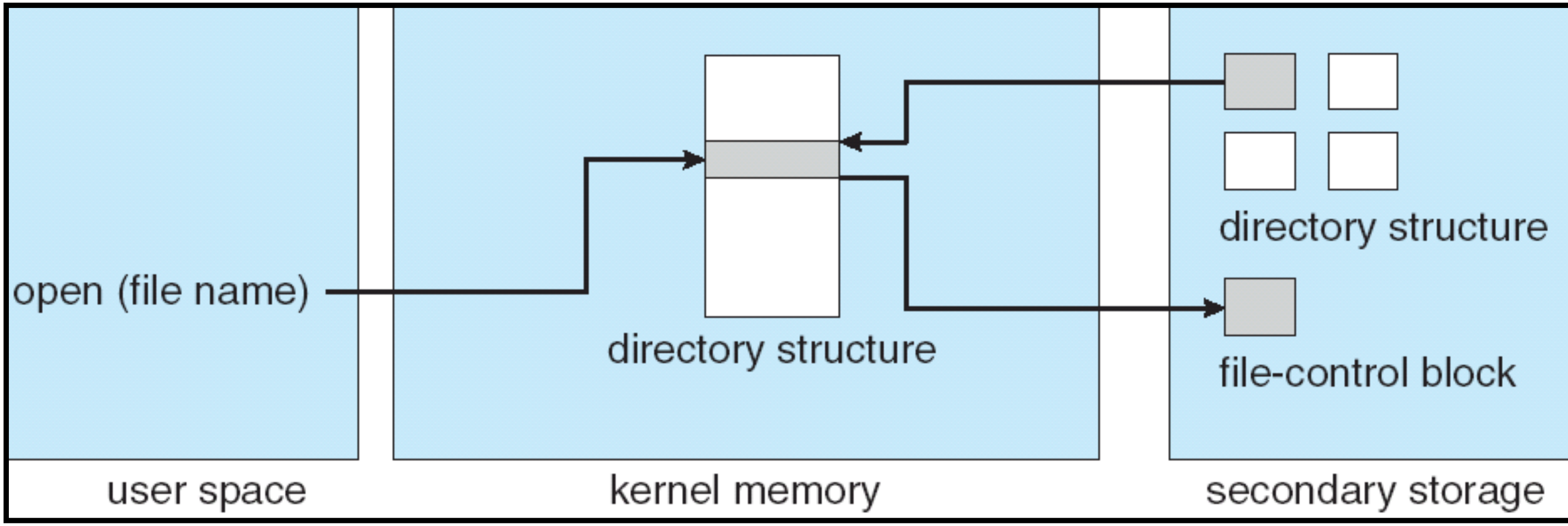
- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data block for root
    - » Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data 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 (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

File

- Named permanent storage
- Contains
  – Data
    » Blocks on disk somewhere
  – Metadata (Attributes)
    » Owner, size, last opened, …
    » Access rights
      • R, W, X
      • Owner, Group, Other (in Unix systems)
      • Access control list in Windows system

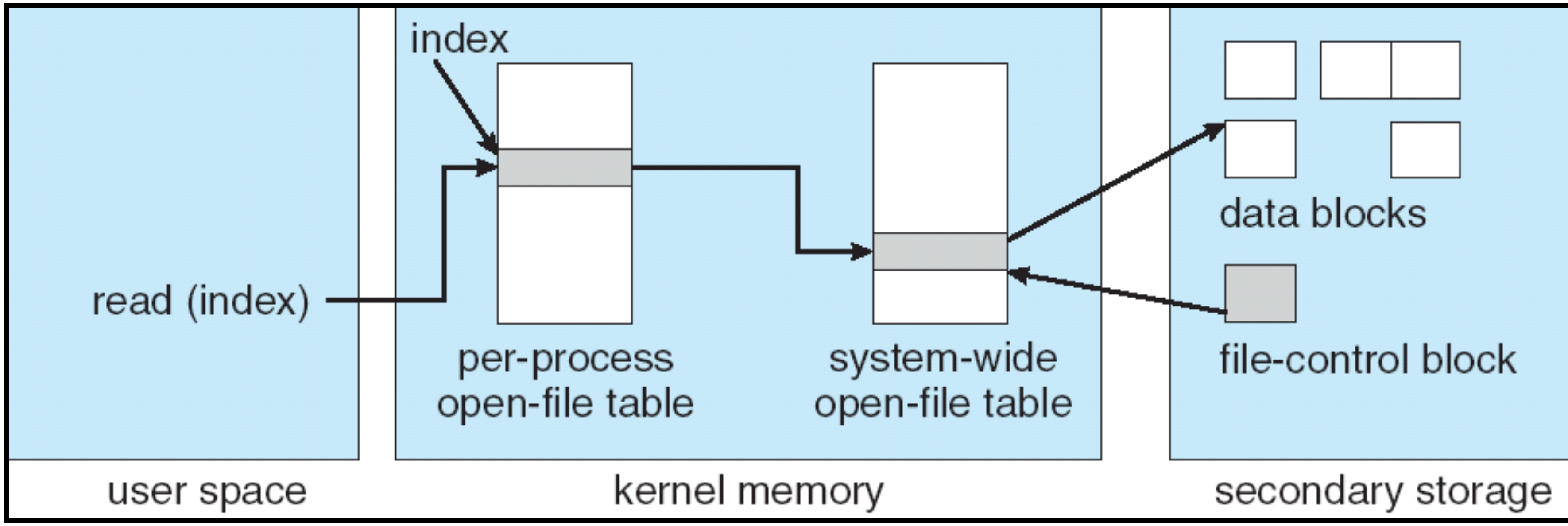


In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table

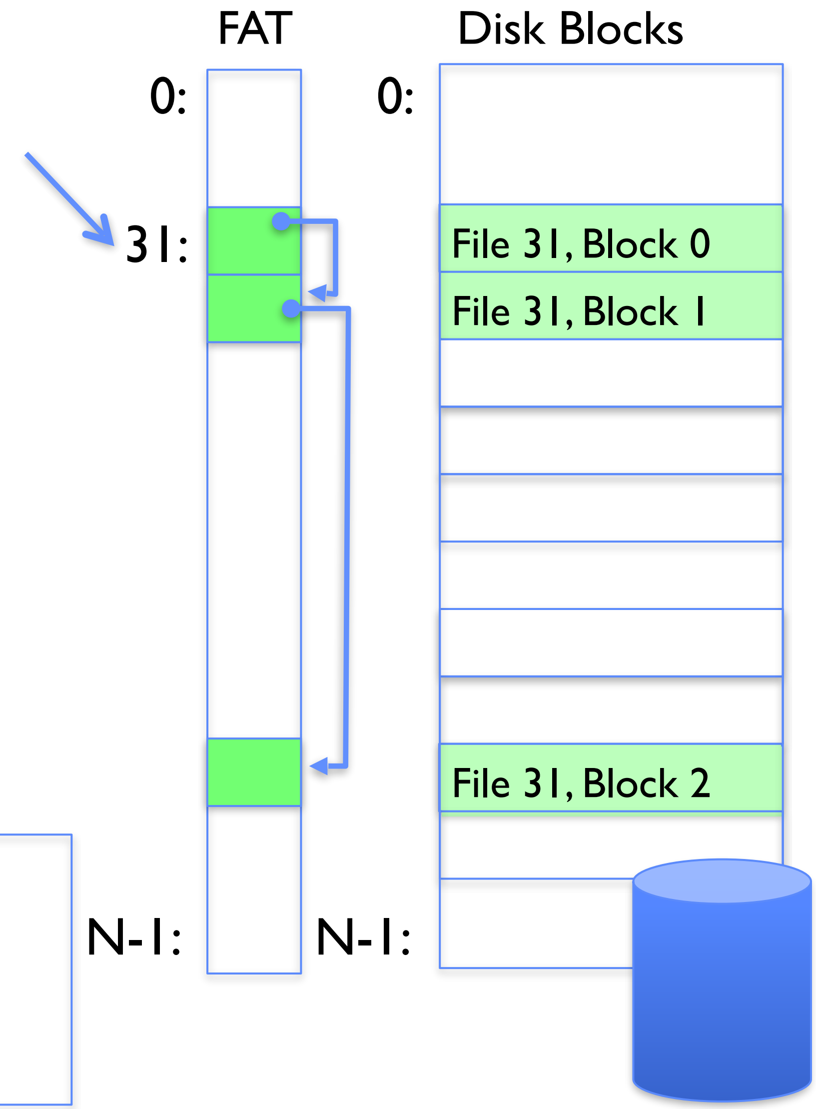
In-Memory File System Structures



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

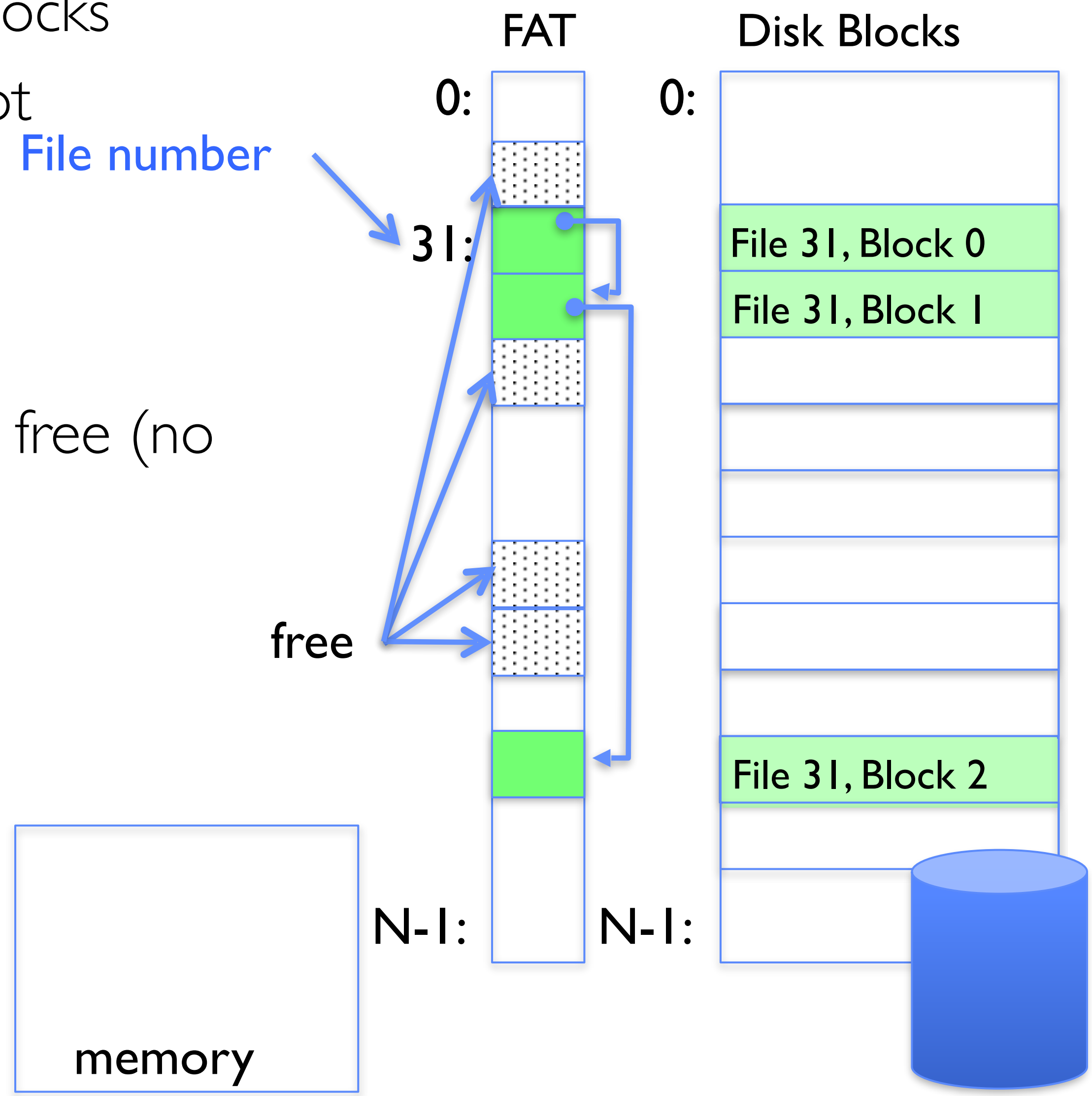
Our first filesystem: FAT (File Allocation Table)

- The most commonly used filesystem in the world!
- Assume (for now) we have a way to translate a path to a "file number"
  - i.e., a directory structure
- Disk Storage is a collection of Blocks
  - Just hold file data (offset o = < B, x >)
- Example: file_read 31, < 2, x >
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory



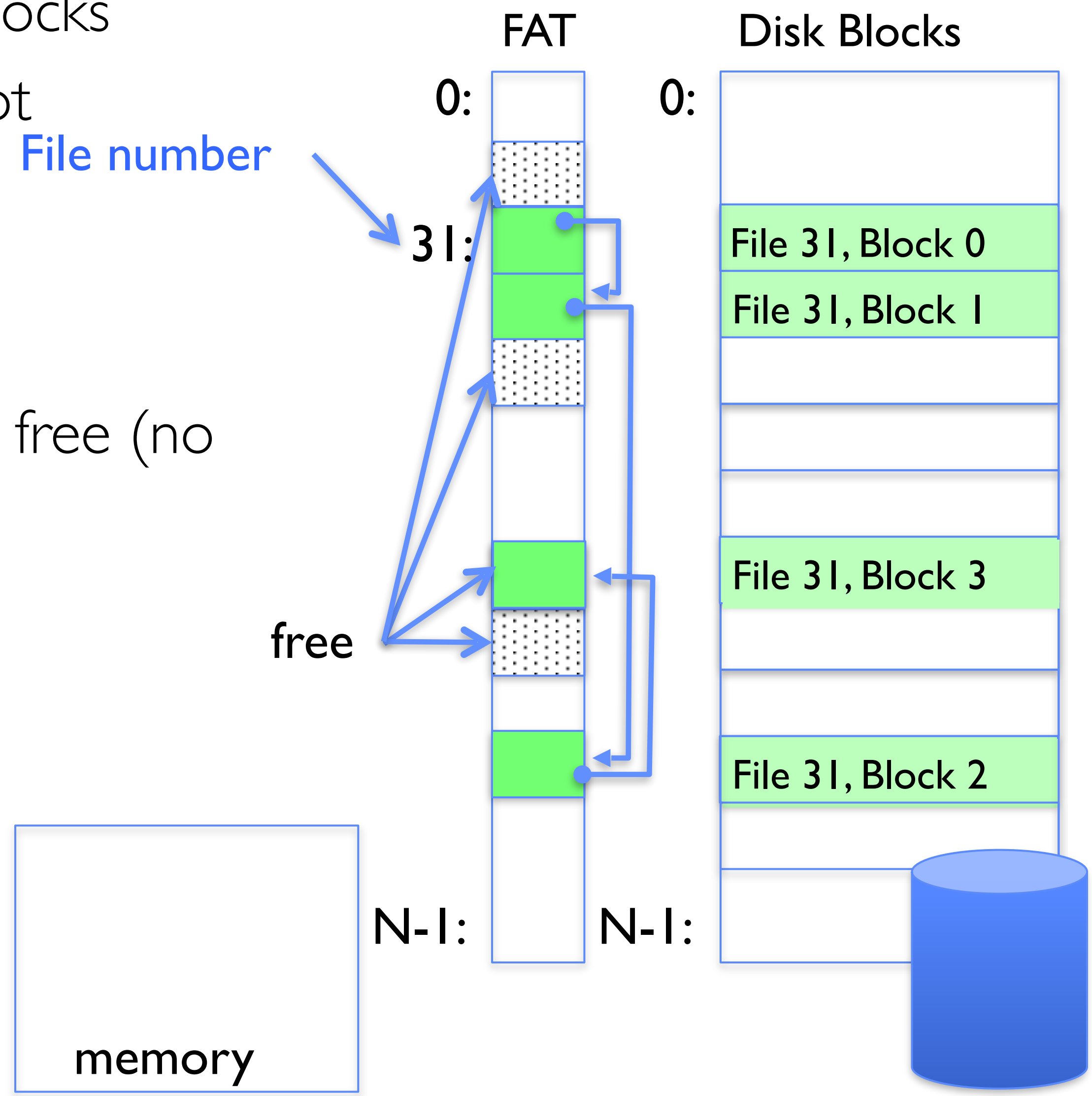
FAT Properties

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = < B, x >)
- Follow list to get block #
- Unused blocks ⇔ Marked free (no ordering, must scan to find)



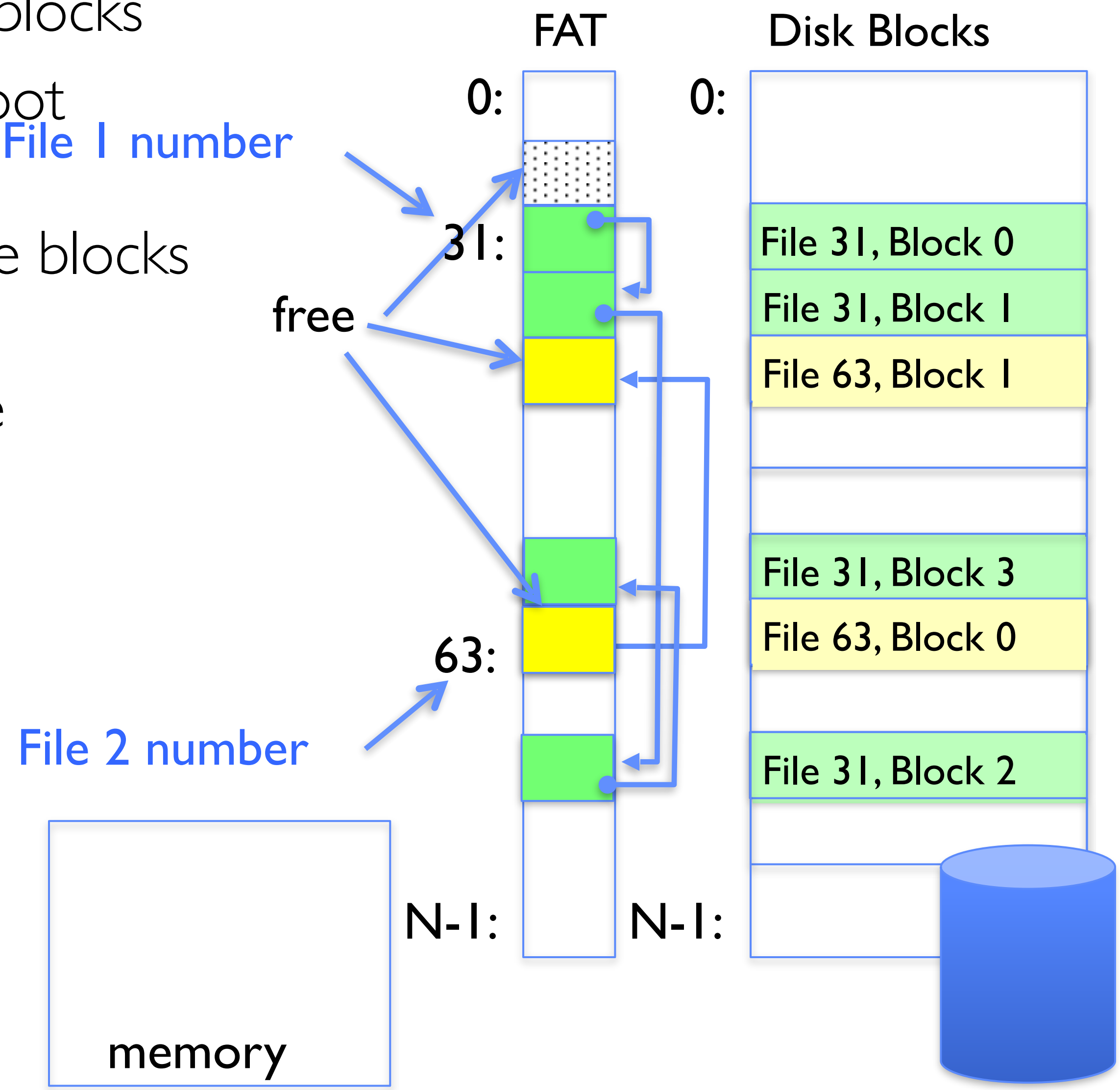
FAT Properties

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = < B, x >)
- Follow list to get block #
- Unused blocks ⇔ Marked free (no ordering, must scan to find)
- Ex: file_write(31, < 3, y >)
  – Grab free block
  – Linking them into file



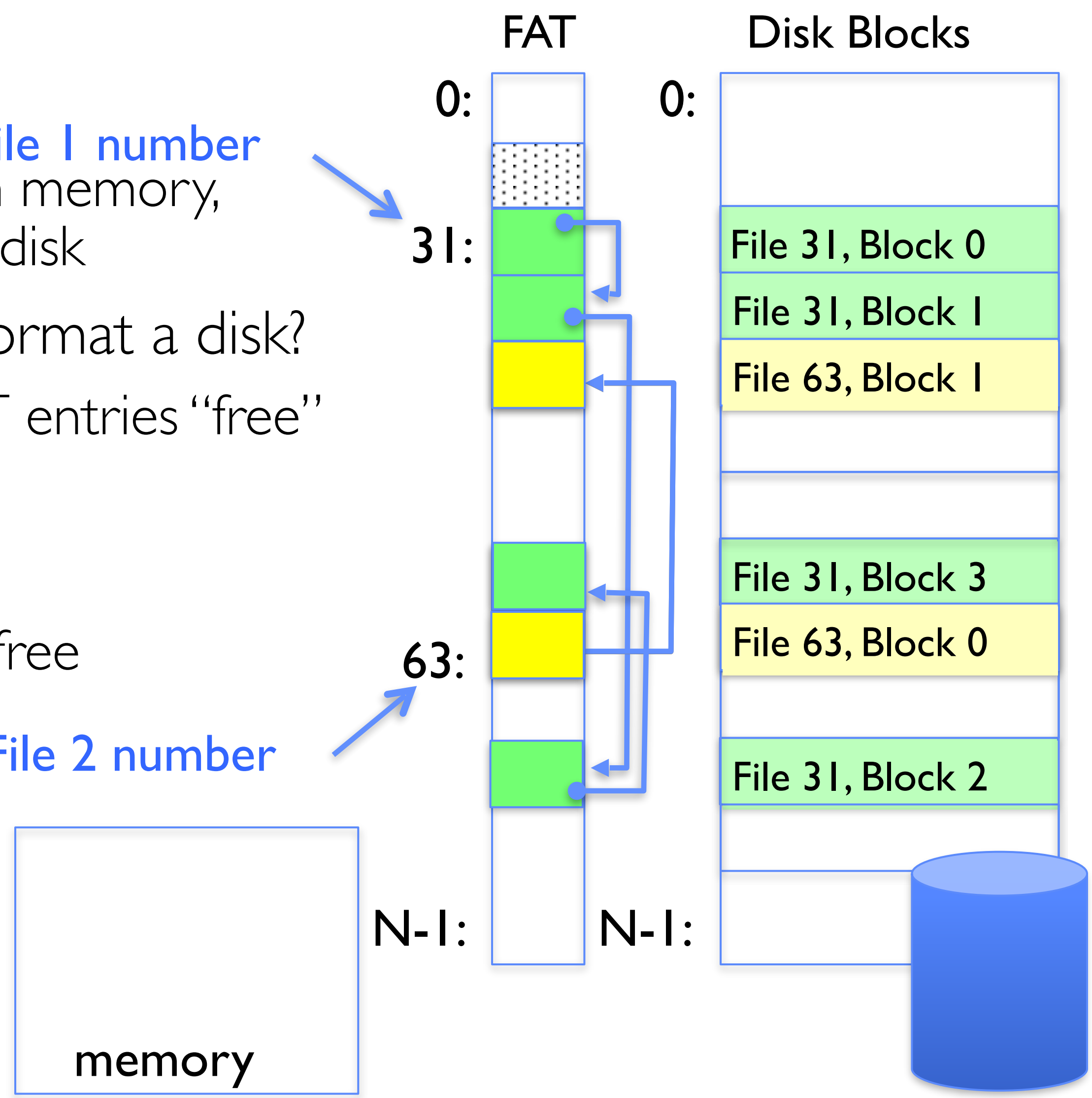
FAT Properties

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- Grow file by allocating free blocks and linking them in
- Ex: Create file, write, write



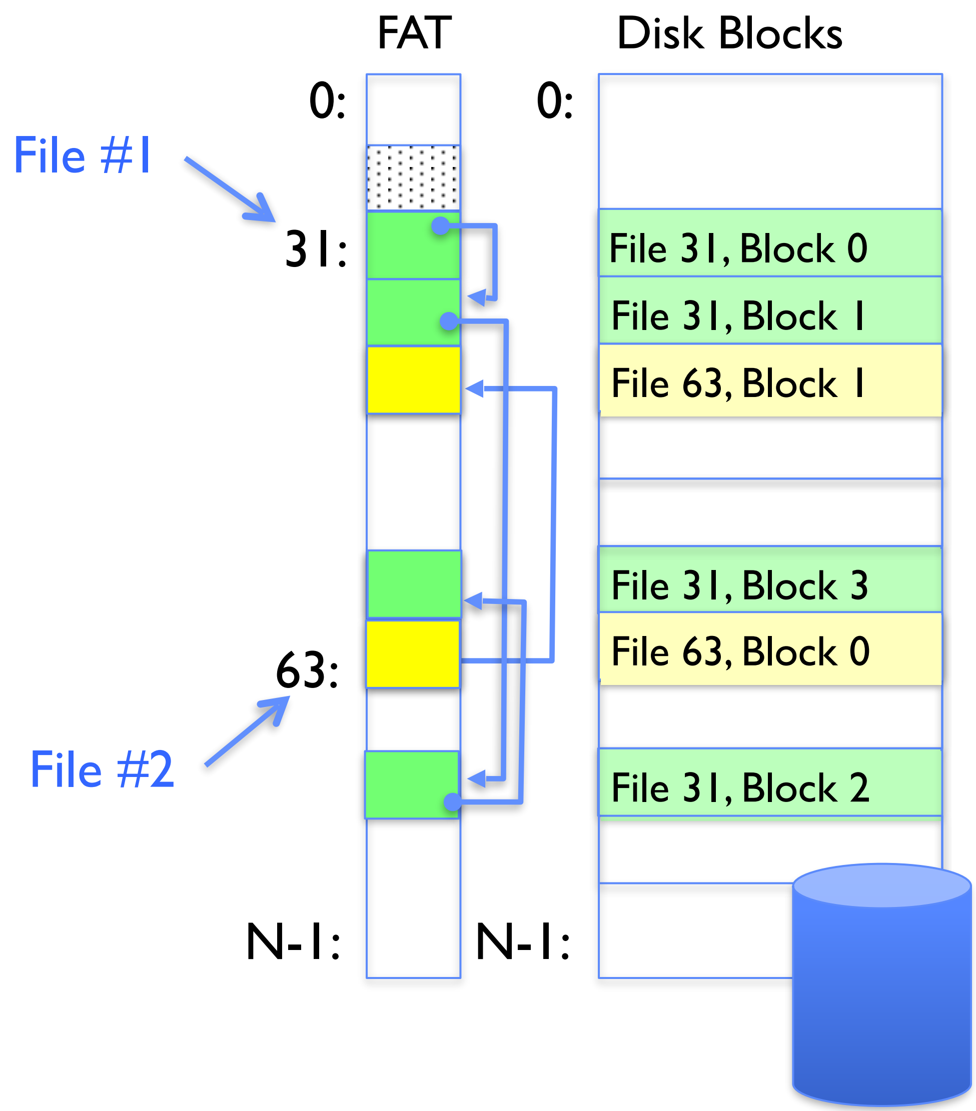
FAT Assessment

- *FAT32 (32 instead of 12 bits) used in Windows, USB drives, SD cards, …*
- Where is FAT stored?
  - On Disk, on boot cache in memory, second (backup) copy on disk
- What happens when you format a disk?
  - Zero the blocks, Mark FAT entries "free"
- What happens when you quick format a disk?
  - Mark all entries in FAT as free
- *Simple*
  - Can implement in device firmware

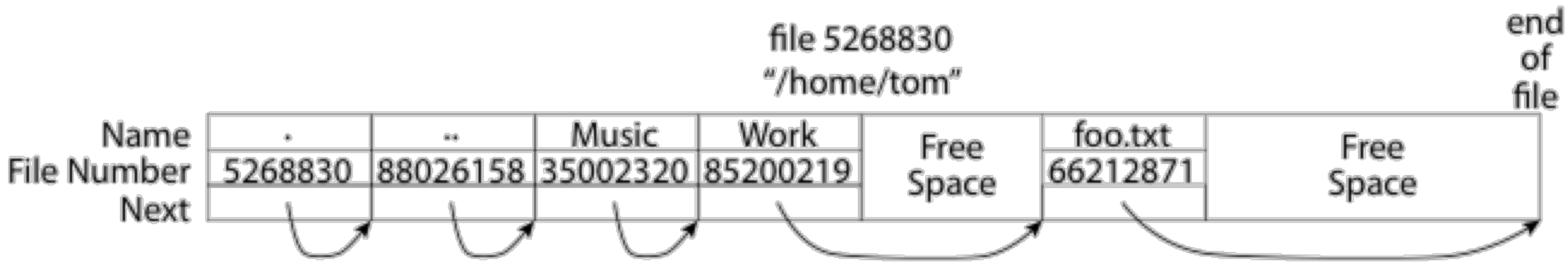


FAT Assessment – Issues

- Time to find block (large files) ??
- Block layout for file ???
- Sequential Access ???
- Random Access ???
- Fragmentation ???
  - MSDOS defrag tool
- Small files ???
- Big files ???



What about FAT directories?



- Directory *is* a file containing <file_name: file_number> mappings
  - Free space for new/deleted entries
  - In FAT: file attributes are kept in directory (!!!)
  - Each directory is a linked list of entries
- Where do you find root directory ("/")?
  - At well-defined place on disk
  - For FAT, this is at block 2 (there are no blocks 0 or 1)
  - Remaining directories are accessed via their file_number

Many Huge FAT Security Holes!

- FAT has no access rights
  - No way, even in principle, to track ownership of data
- FAT has no header in the file blocks
  - No way to enforce control over data, since all processes have access of FAT table
  - Just follow pointer to disk blocks
- Just gives an index into the FAT to read data
  - (file number = block number)
  - Could start in middle of file or access deleted data

Characteristics of Files

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

9:9

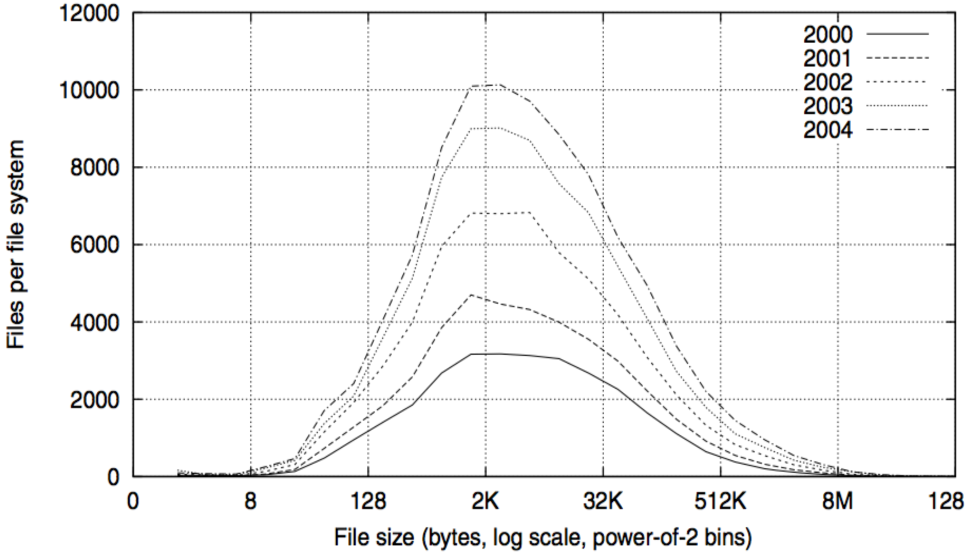


Fig. 2. Histograms of files by size.

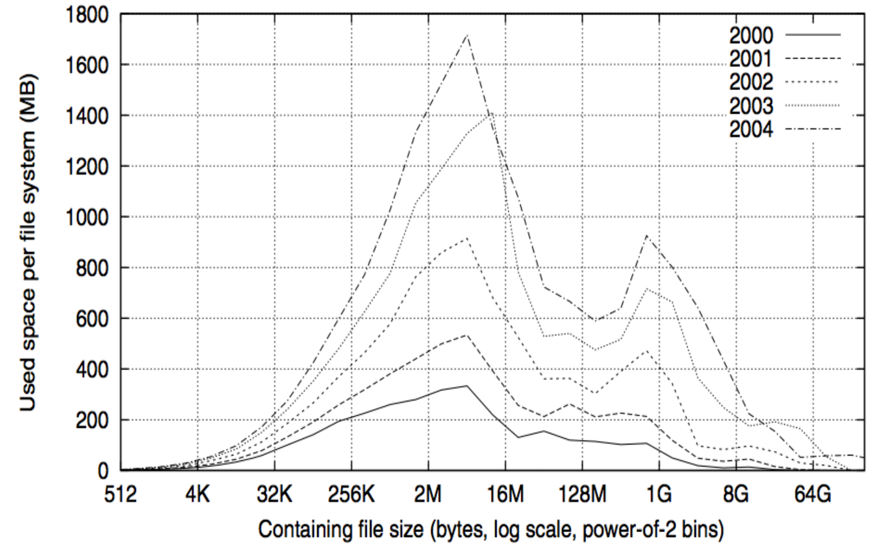


Fig. 4. Histograms of bytes by containing file size.

Unix File System (1/2)

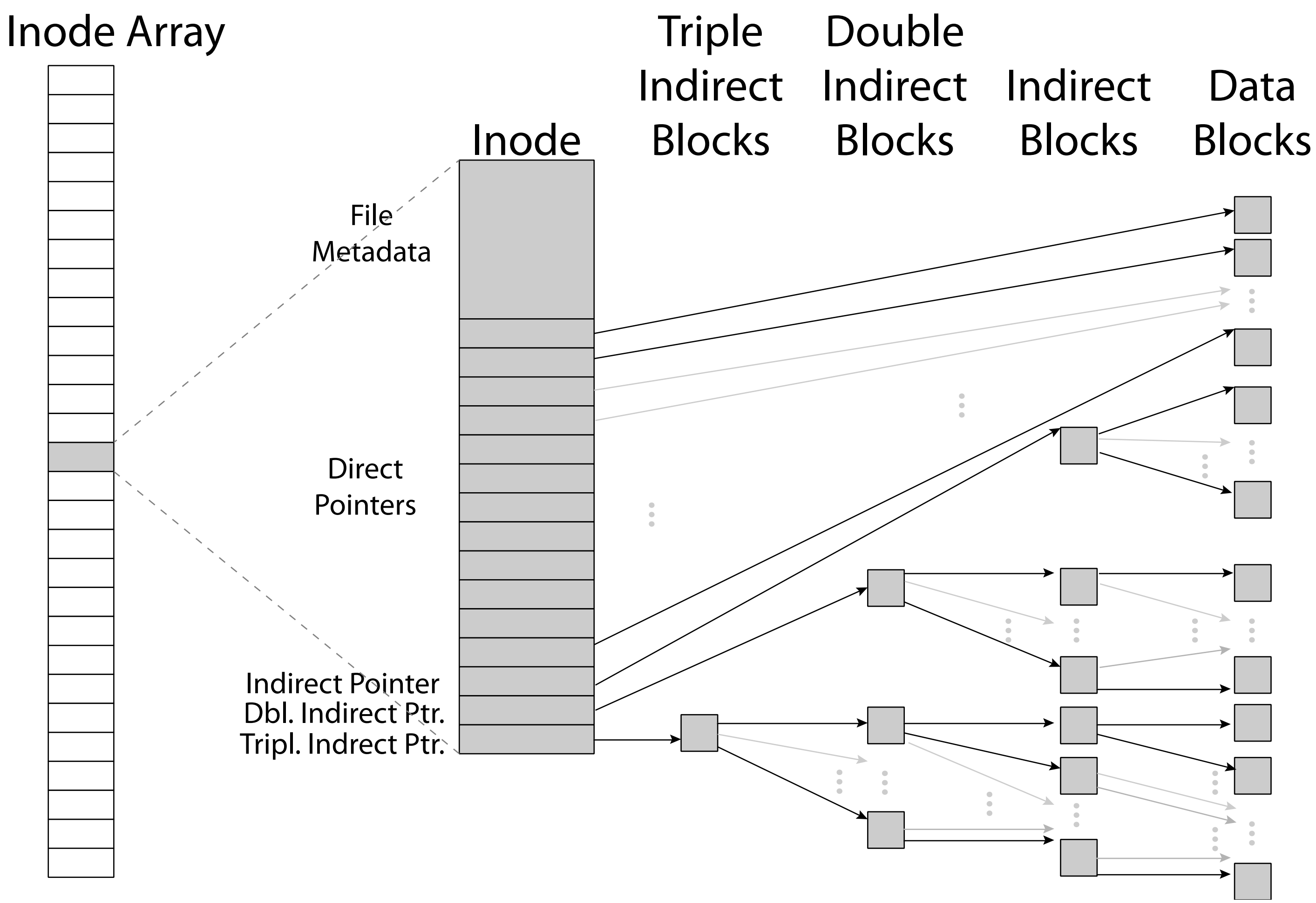
- Original inode format appeared in BSD 4.1
  - Berkeley Standard Distribution Unix
  - Part of your heritage [if you are at Berkley]!
  - Similar structure for Linux Ext2/3
- File Number is index into inode arrays
- Multi-level index structure
  - Great for little and large files
  - Asymmetric tree 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

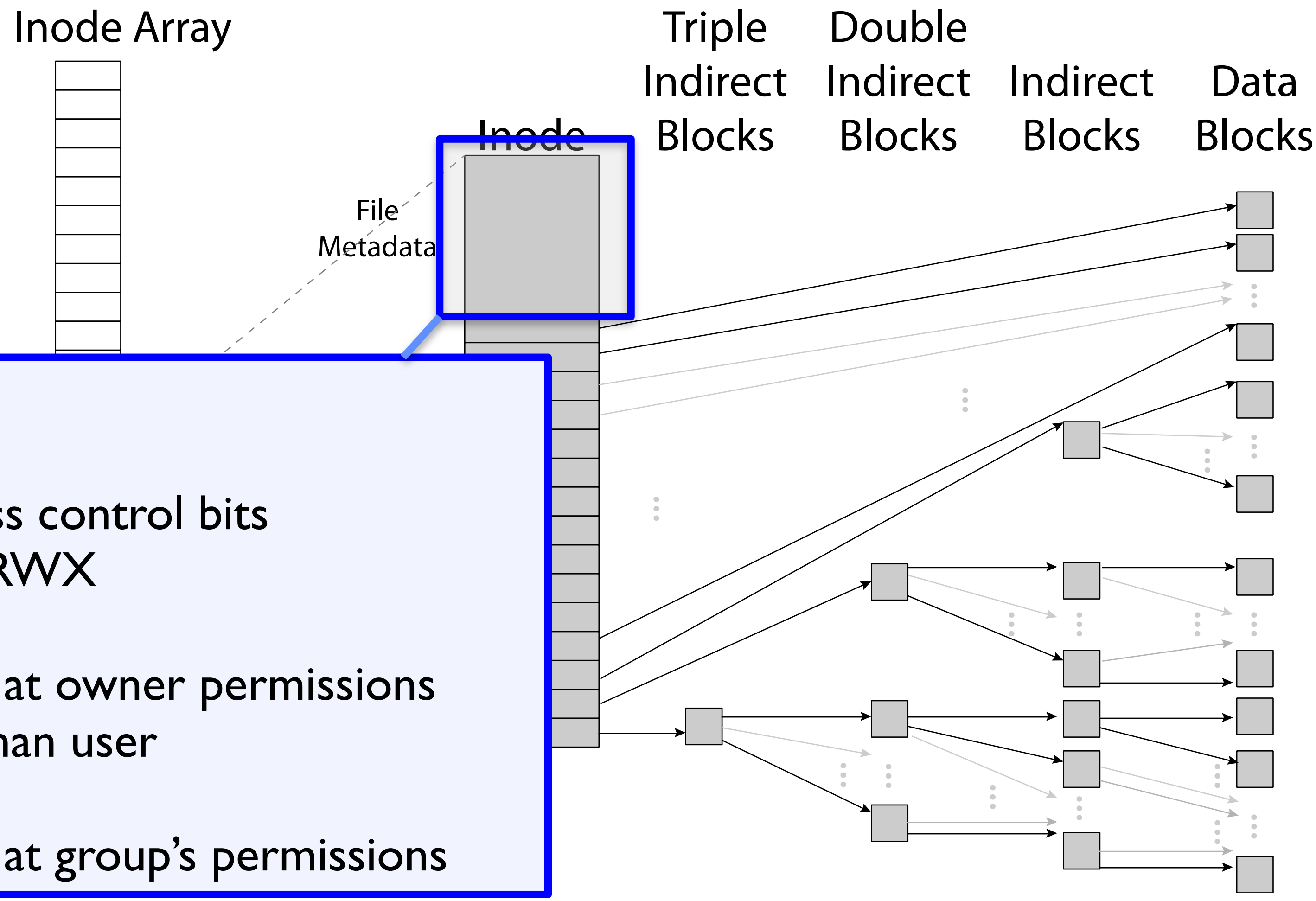
Inode Structure

- inode metadata



File Attributes

- inode metadata



User
Group
9 basic access control bits
- UGO x RWX

Setuid bit
- execute at owner permissions rather than user

Setgid bit
- execute at group's permissions

Data Storage

- Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

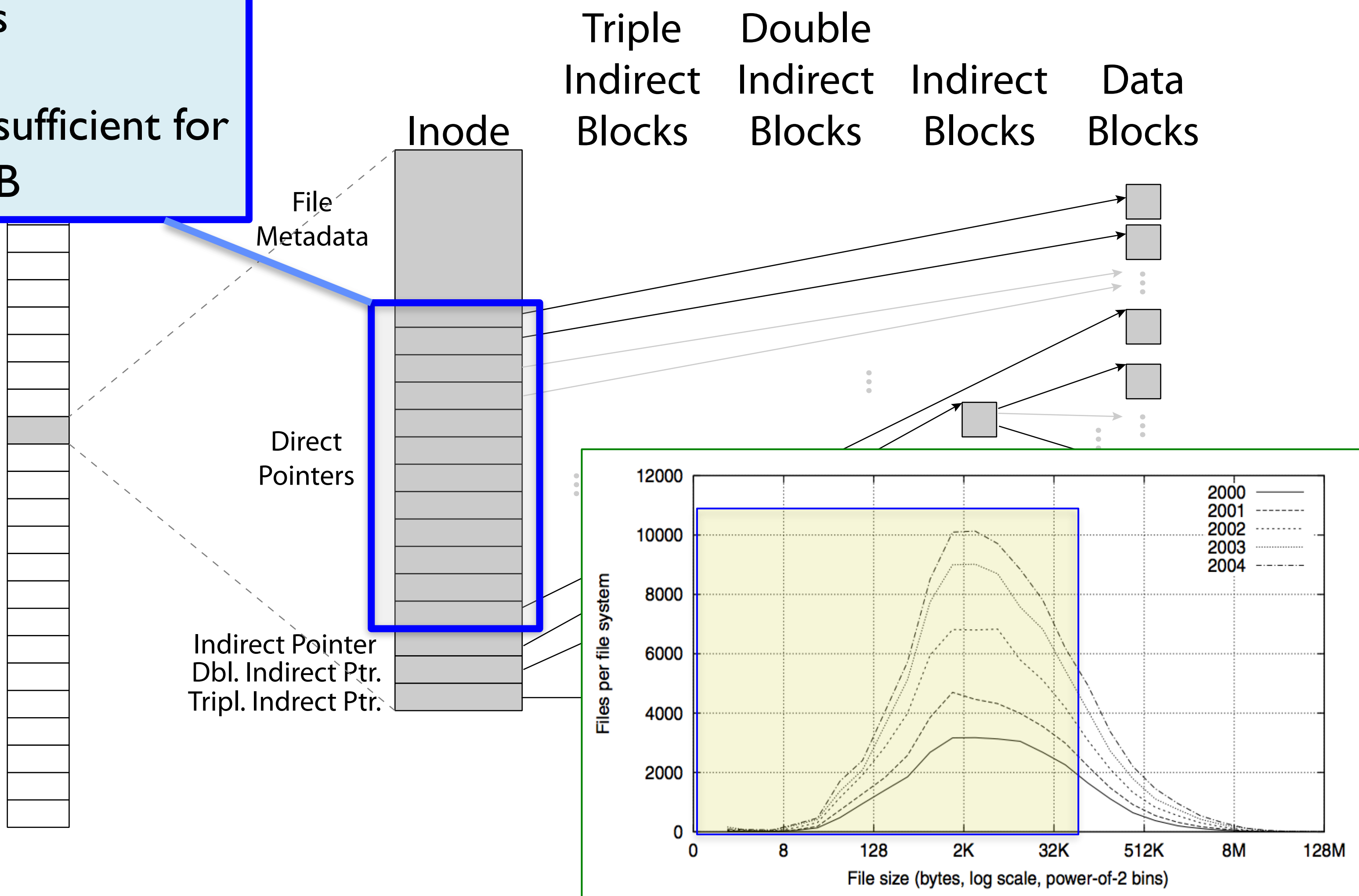
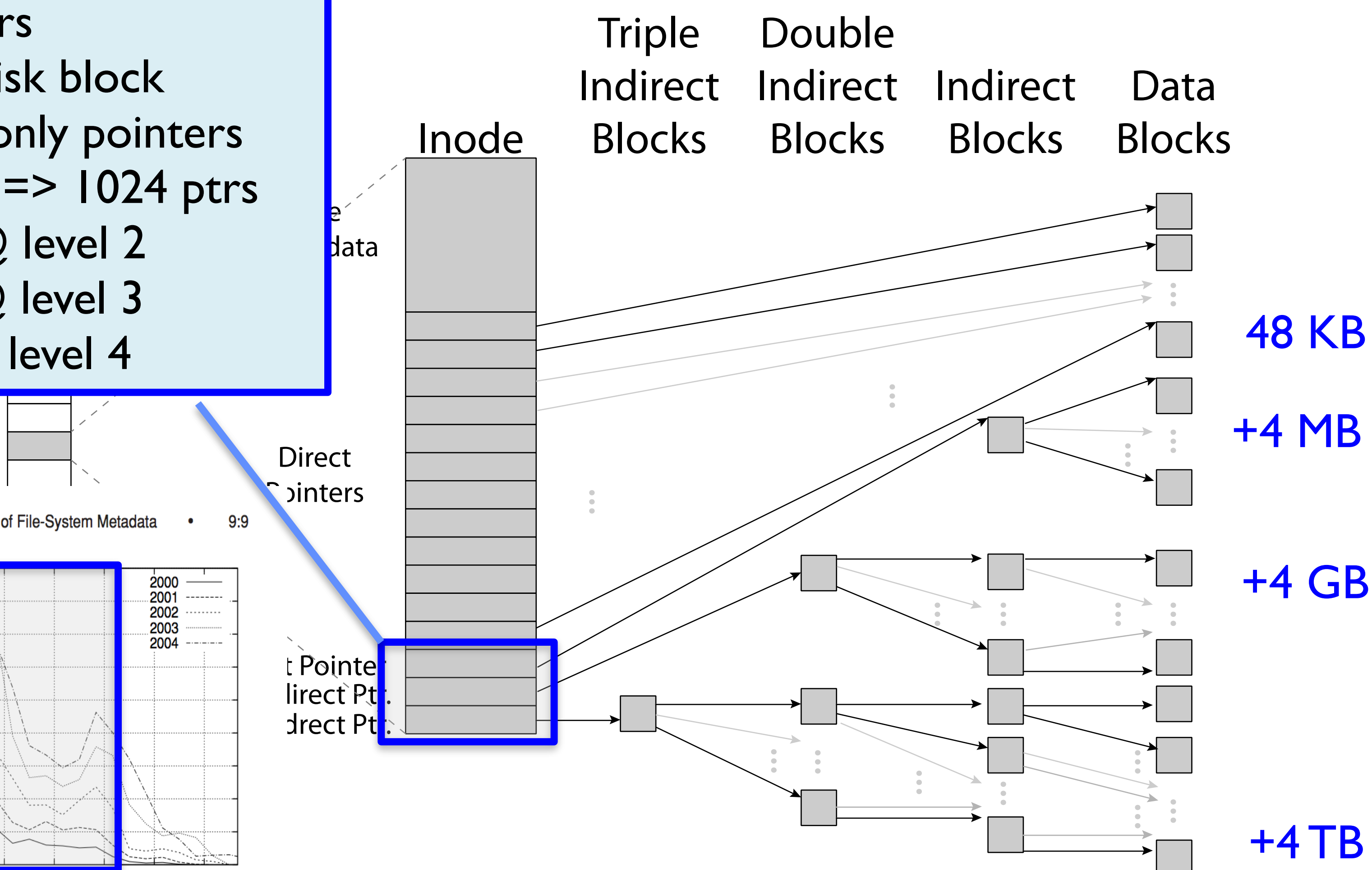


Fig. 2. Histograms of files by size.

Data Storage

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

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



A Five-Year Study of File-System Metadata • 9:9

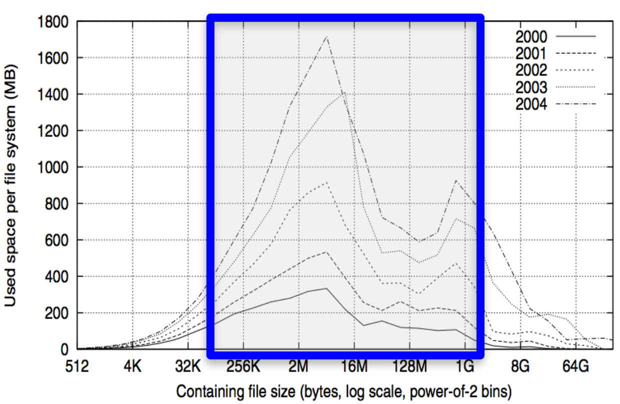


Fig. 4. Histograms of bytes by containing file size.

UNIX BSD 4.2 (1984) (1/2)

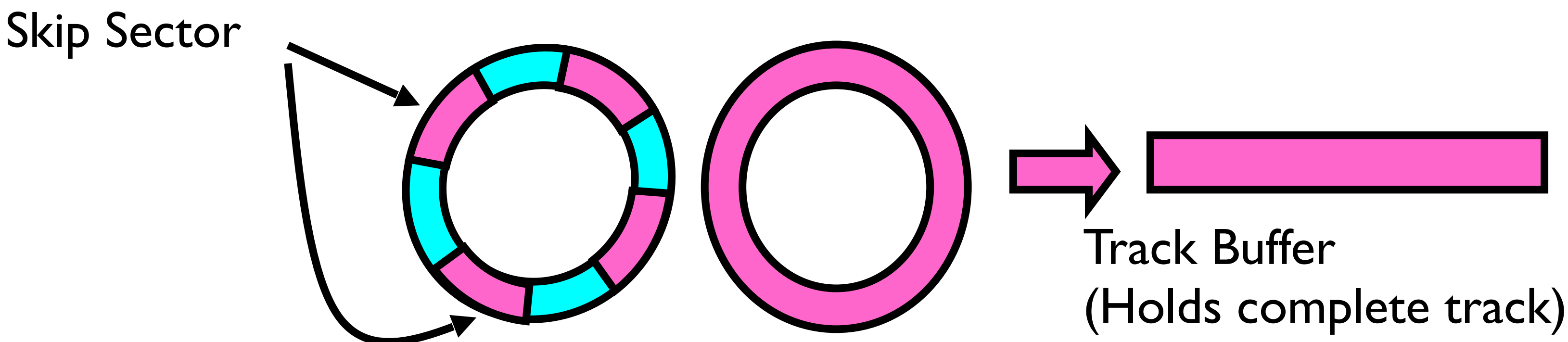
- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from Cray Operating System:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned later)

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!



  - Solution1: 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
  - Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it yet
    - » 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
- Modern disks + controllers do many things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

Where are inodes Stored?

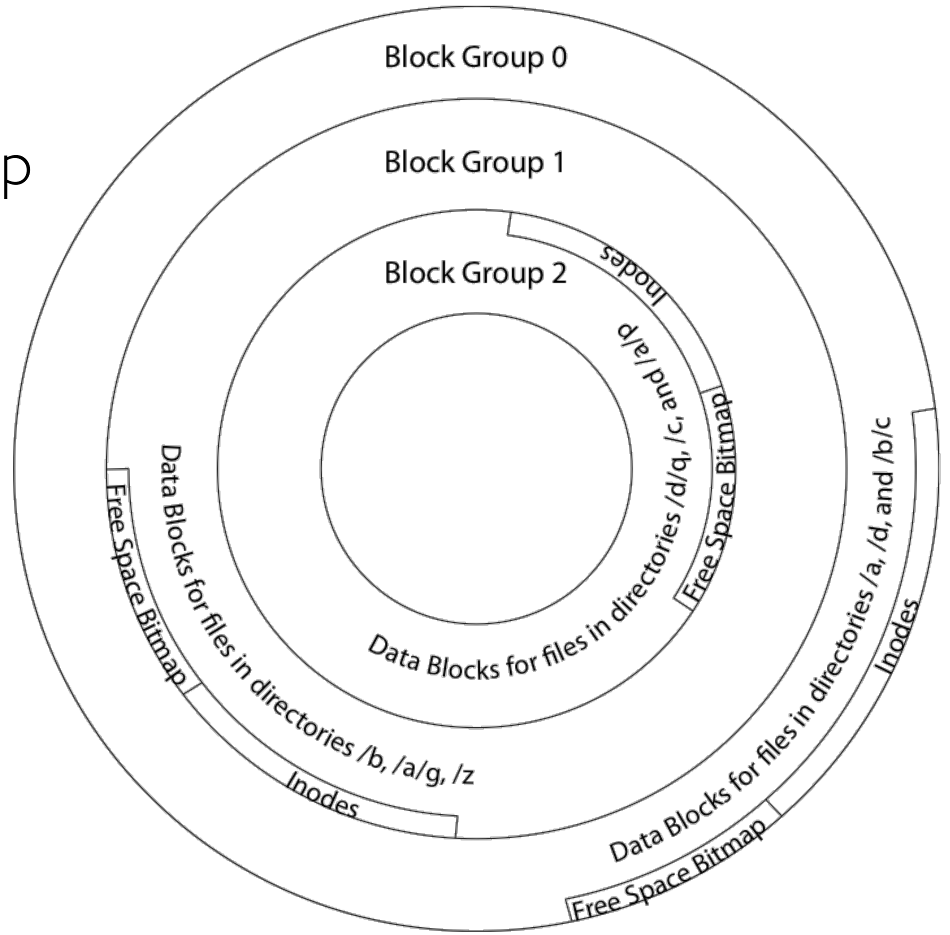
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
- Header not stored anywhere near the data blocks
  – To read a small file, seek to get header, seek back to data
- Fixed size, set when disk is formatted
  – At formatting time, a fixed number of 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 bits of file header array on many cylinders
  - For small directories, can fit all data, file headers, etc. in same cylinder ⇒ 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

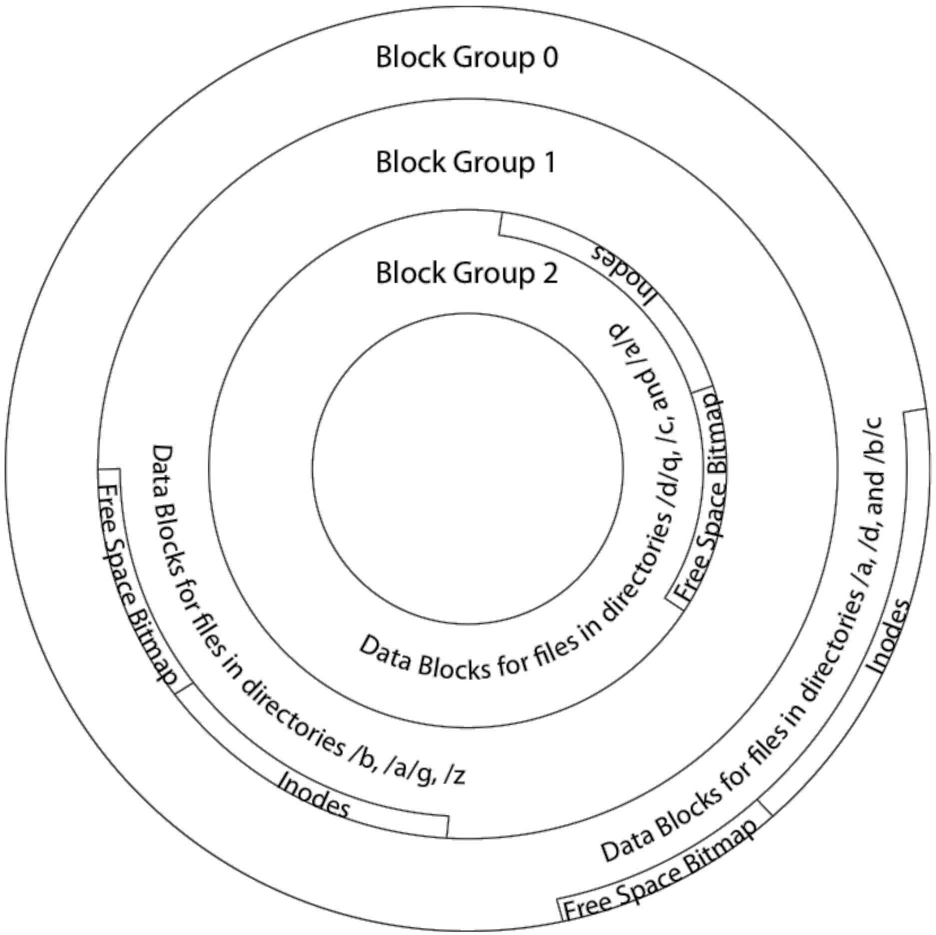
4.2 BSD Locality: Block Groups

- 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

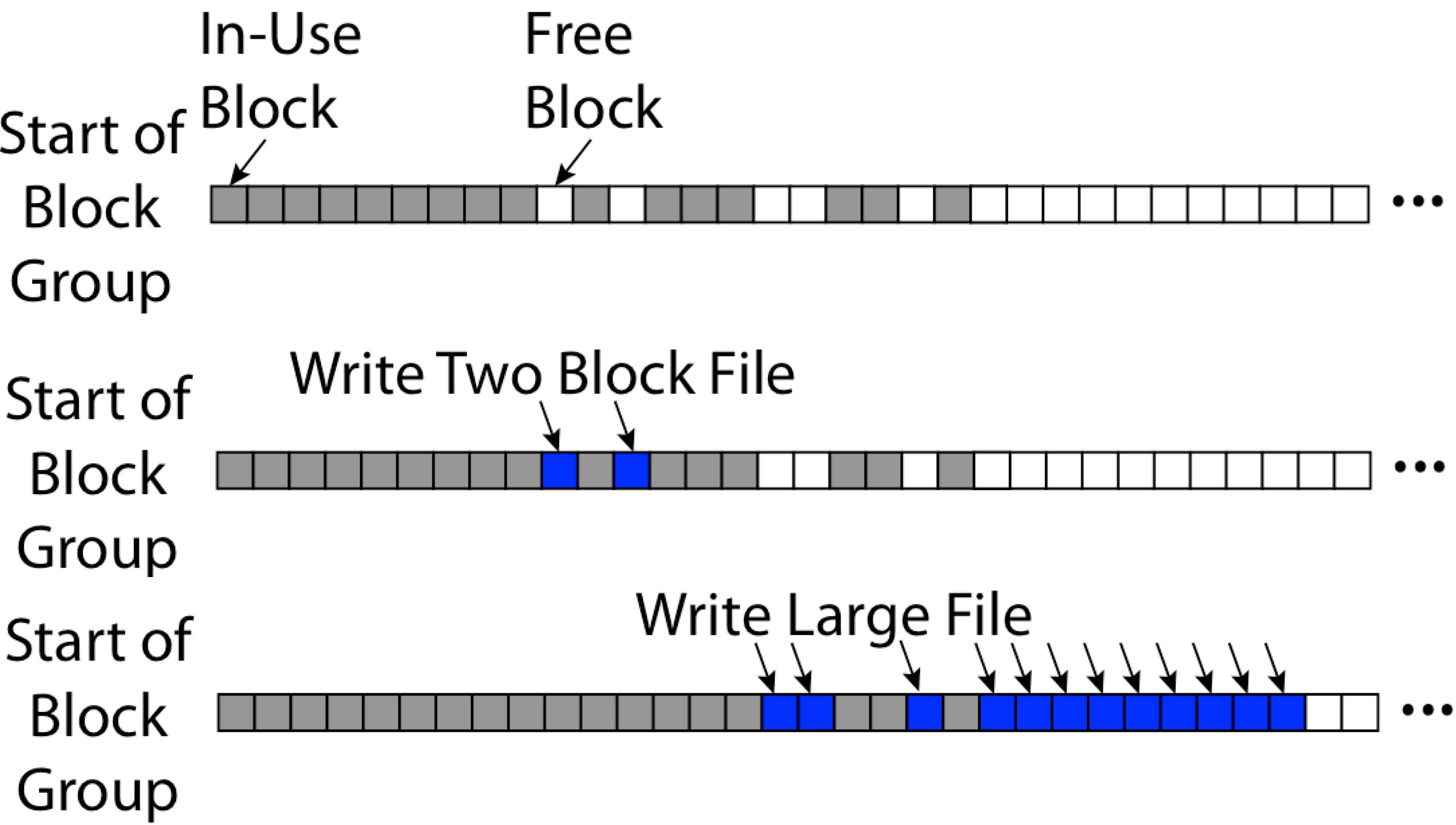


4.2 BSD Locality: Block Groups

- 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



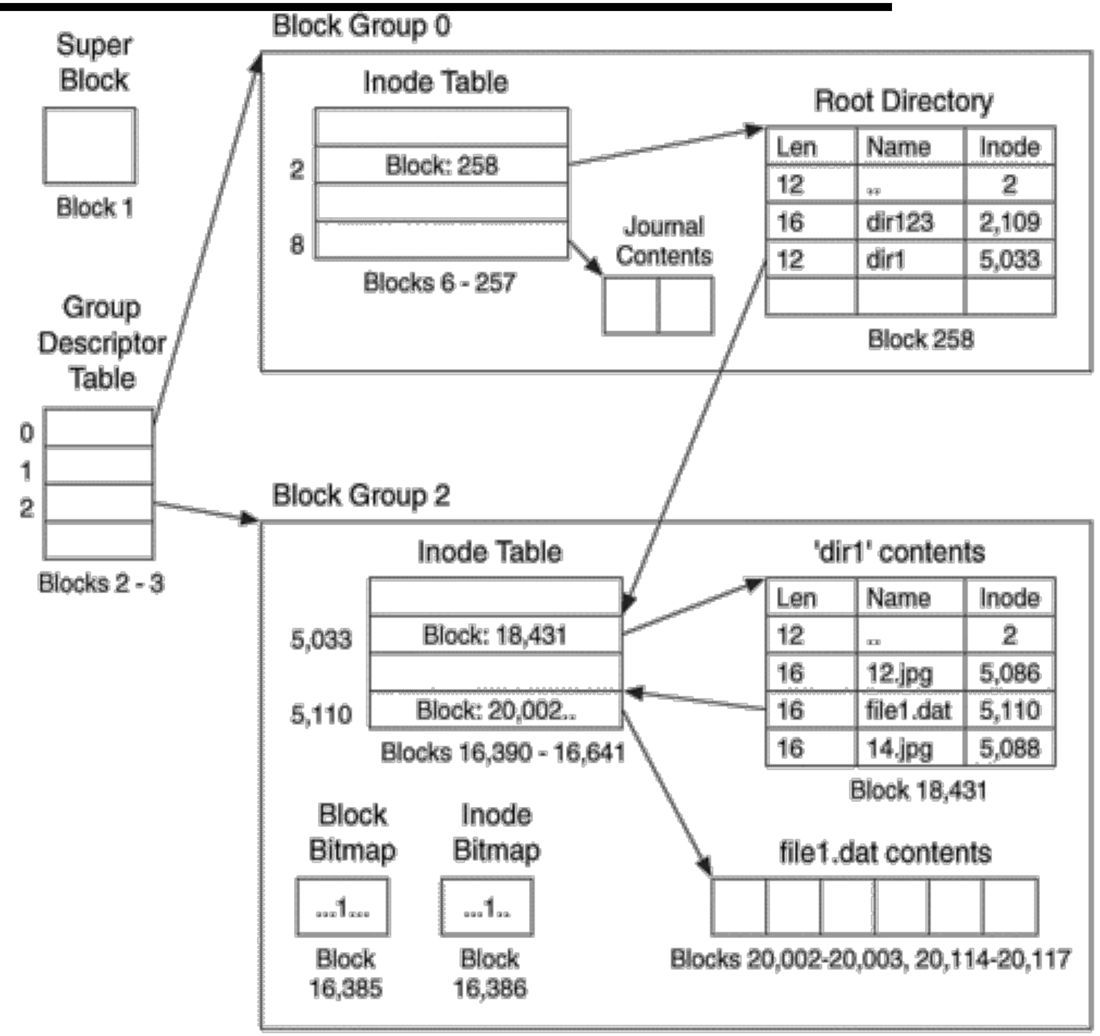
UNIX 4.2 BSD FFS

- 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 (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

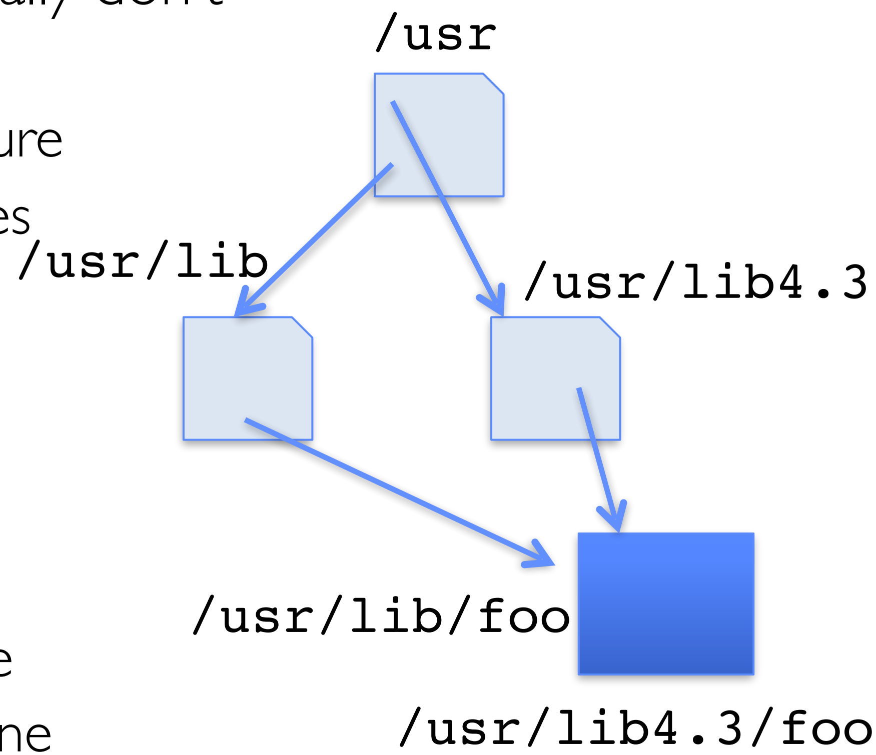
Linux Example: Ext2/3 Disk Layout

- 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)`

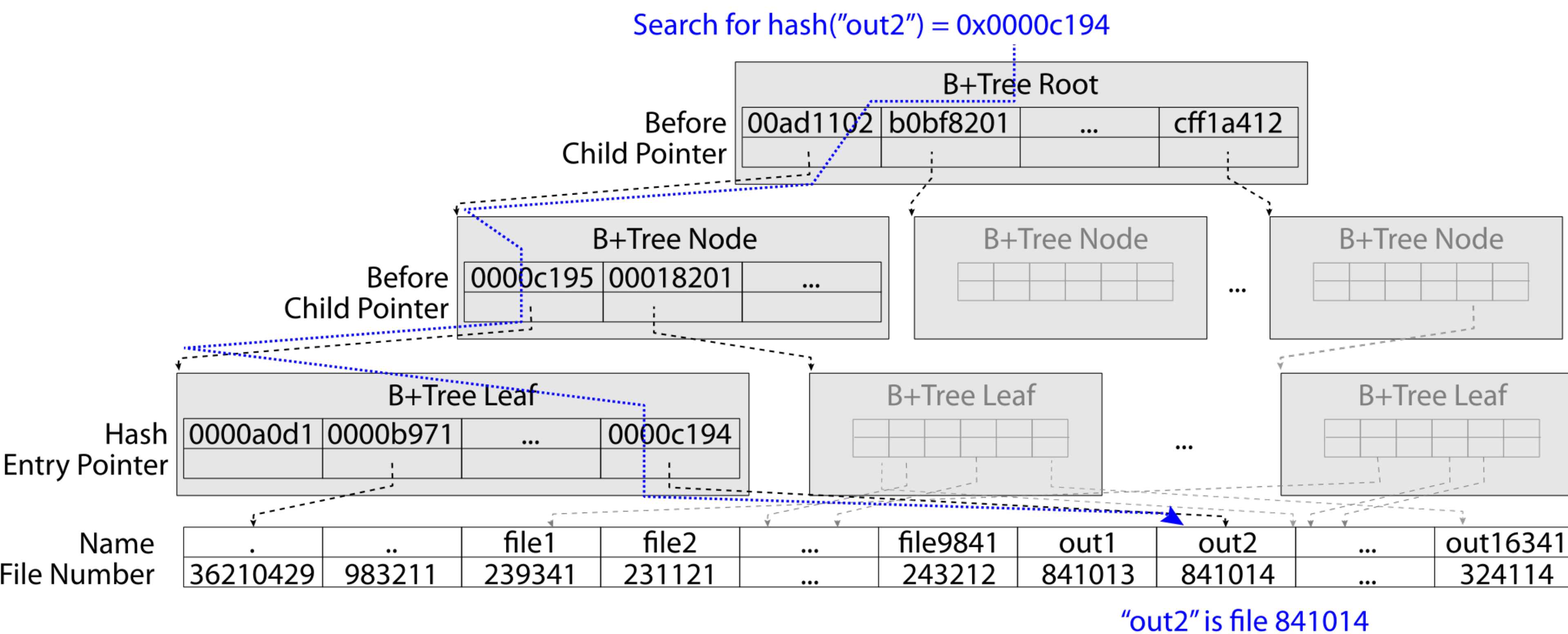
Links

- 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



NTFS

- New Technology File System (NTFS)
  – Default on Microsoft Windows systems

- Variable length extents
  – Rather than fixed blocks

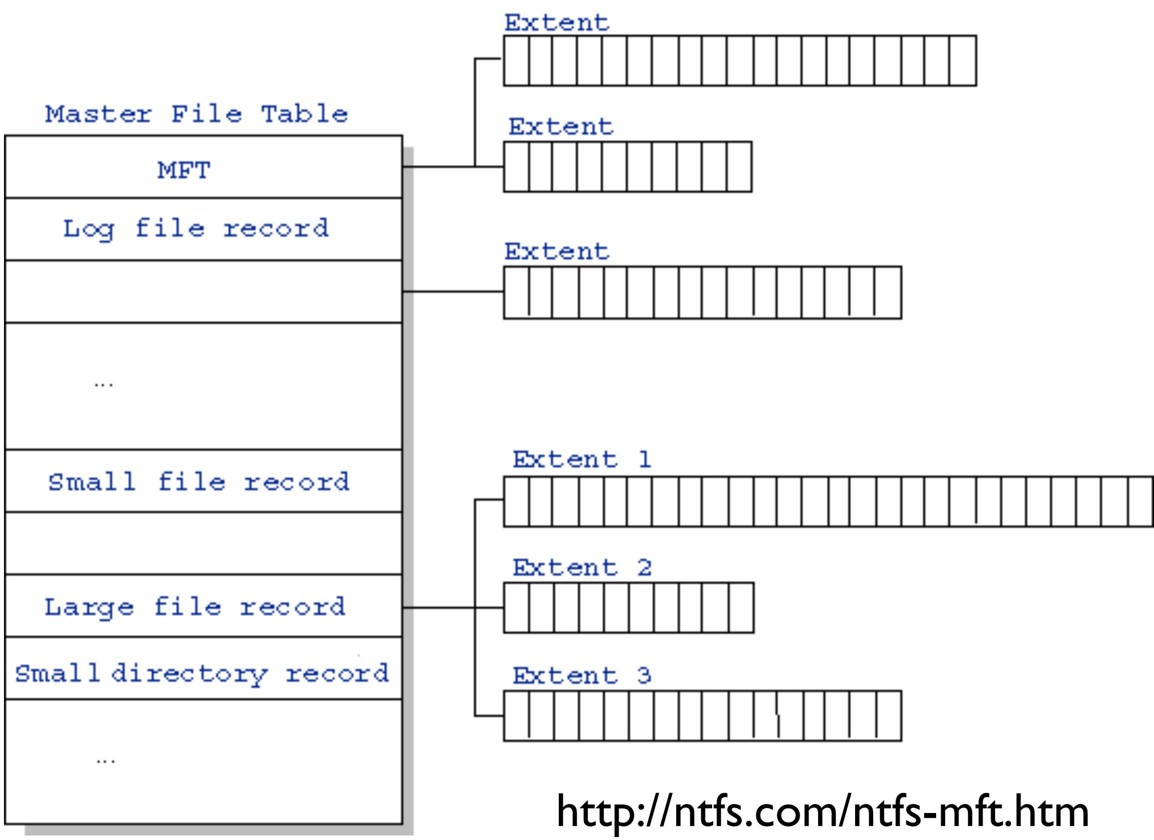
- Everything (almost) is a sequence of <attribute:value> pairs
  – Meta-data and data

- Mix direct and indirect freely

- Directories organized in B-tree structure by default

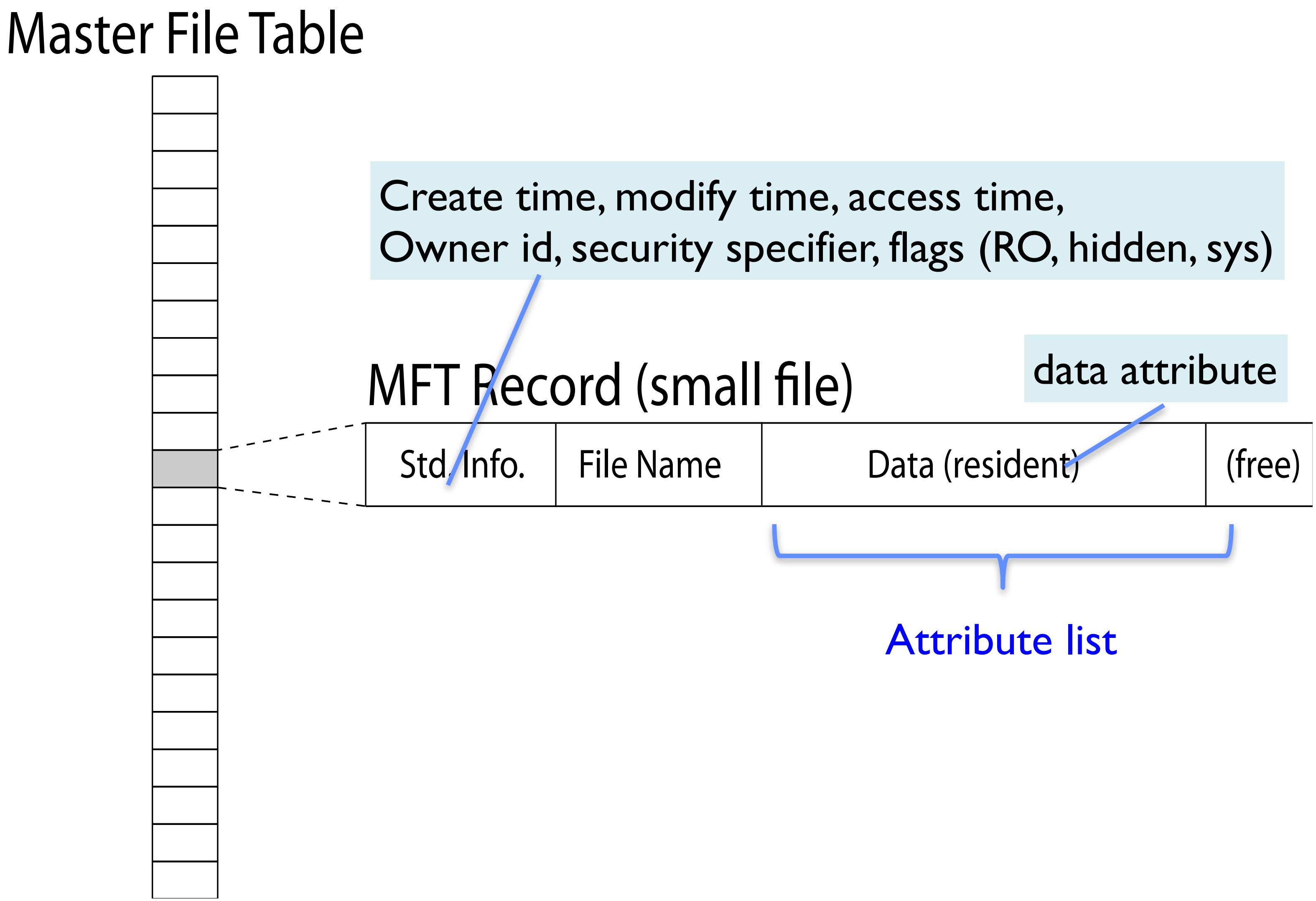
NTFS

- 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

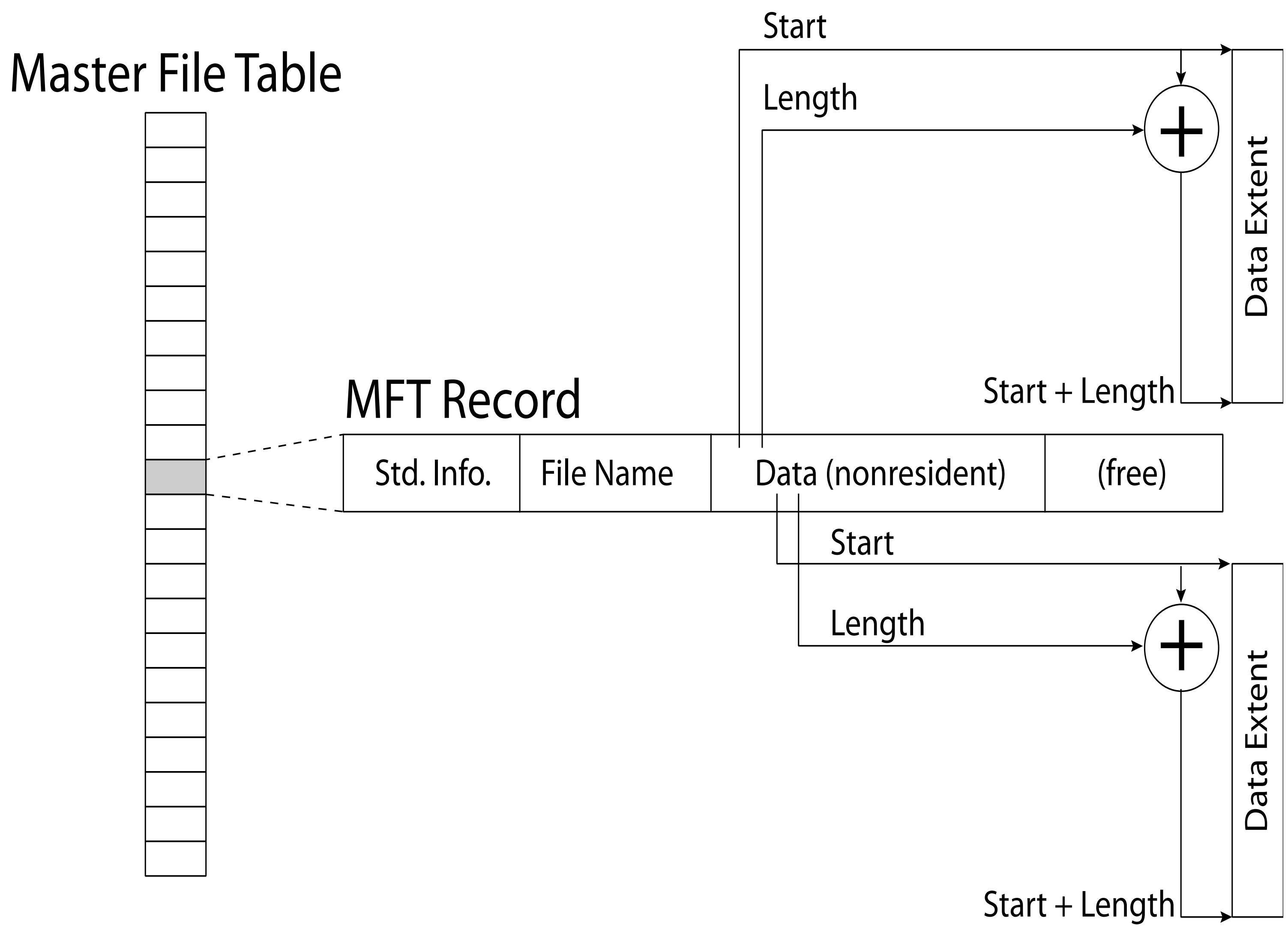


http://ntfs.com/ntfs-mft.htm

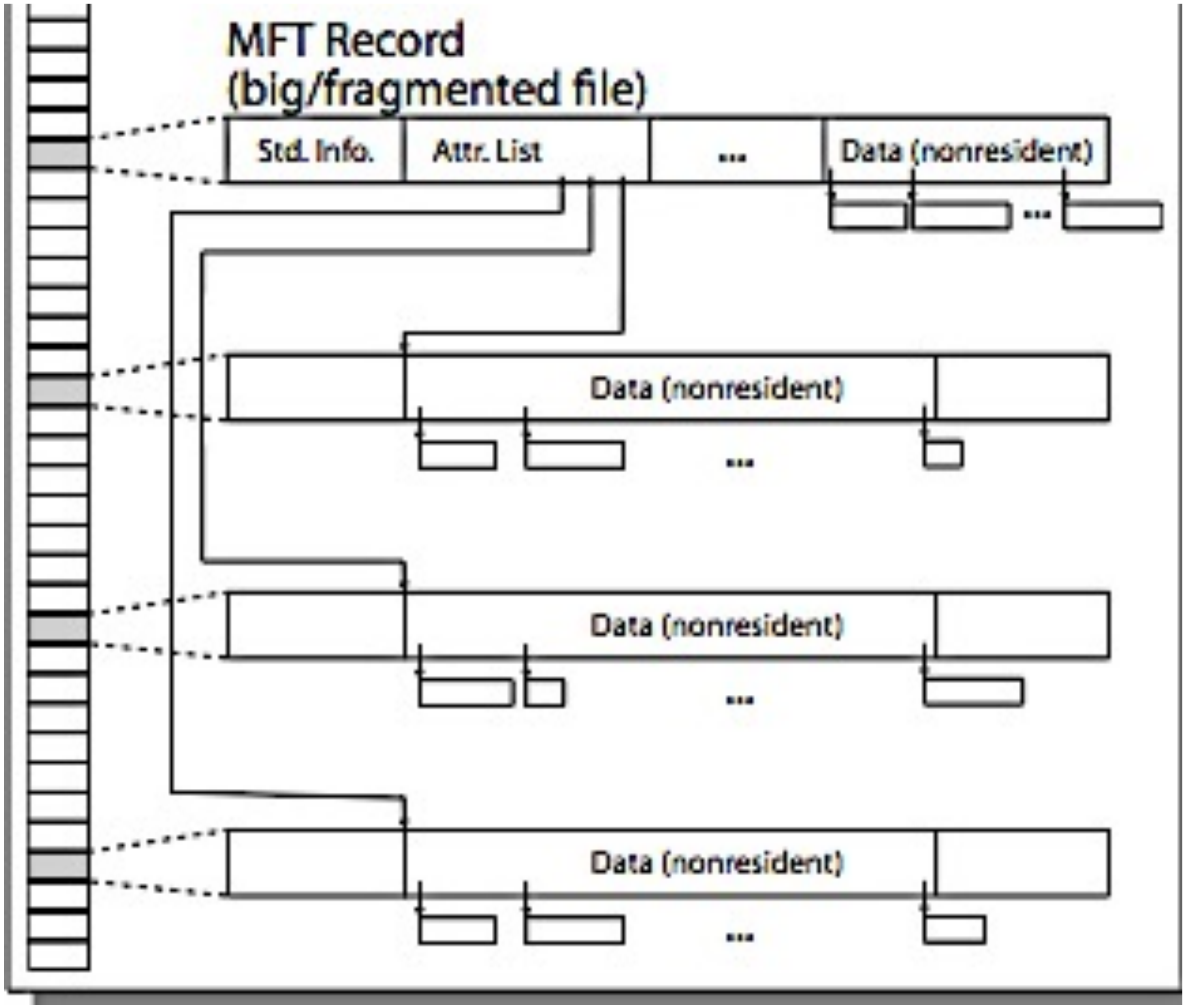
NTFS Small File



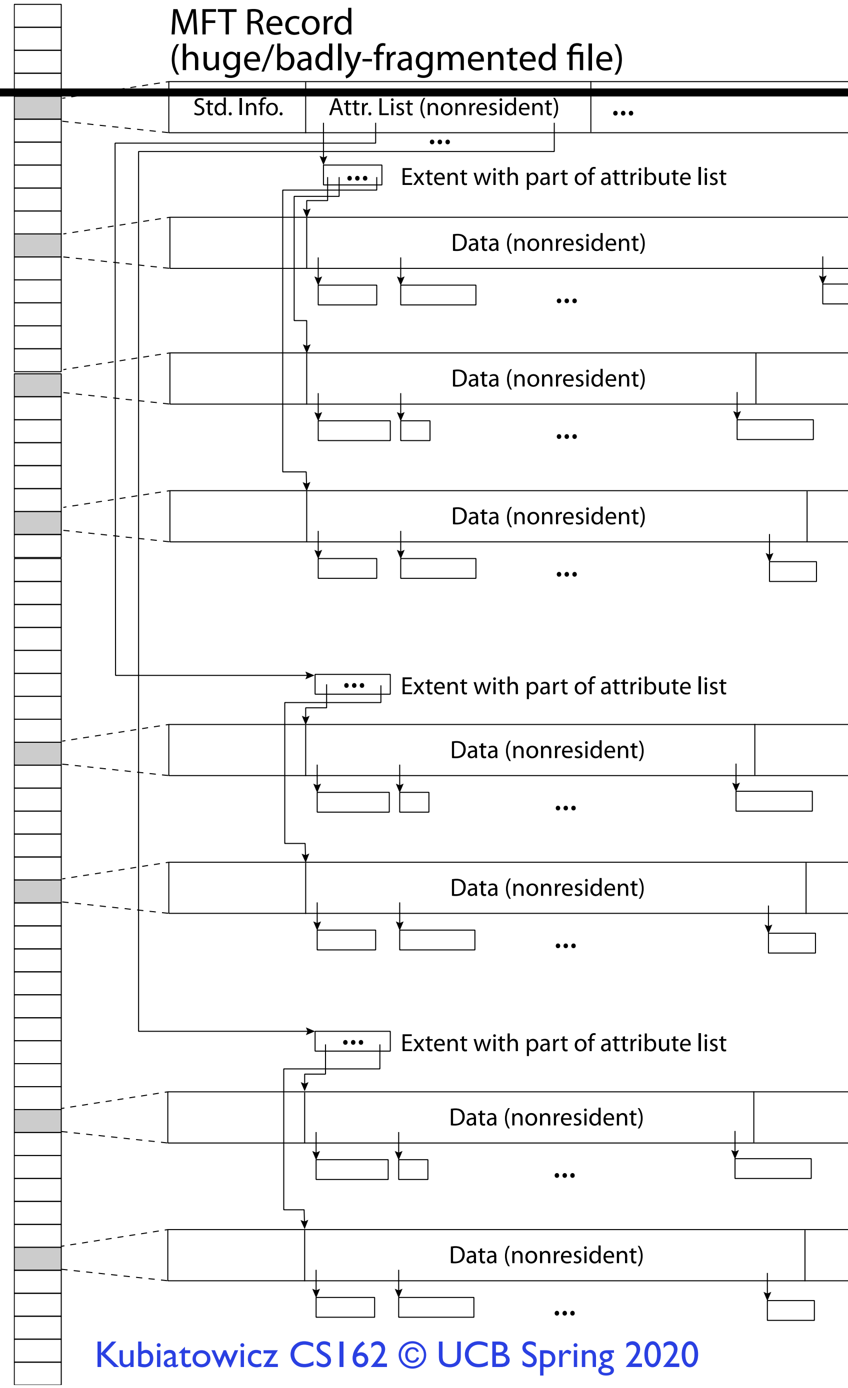
NTFS Medium File



NTFS Multiple Indirect Blocks



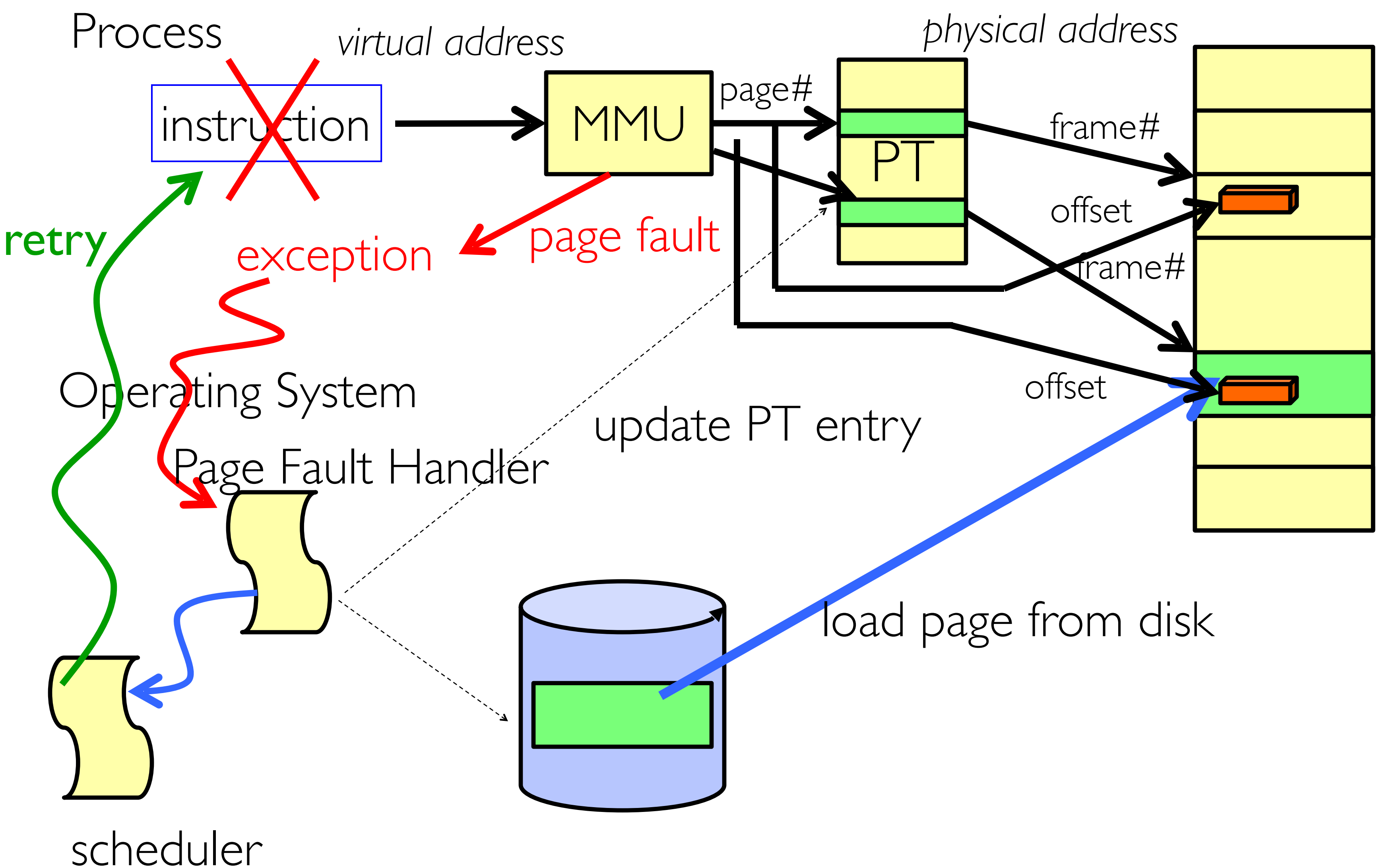
Master File Table



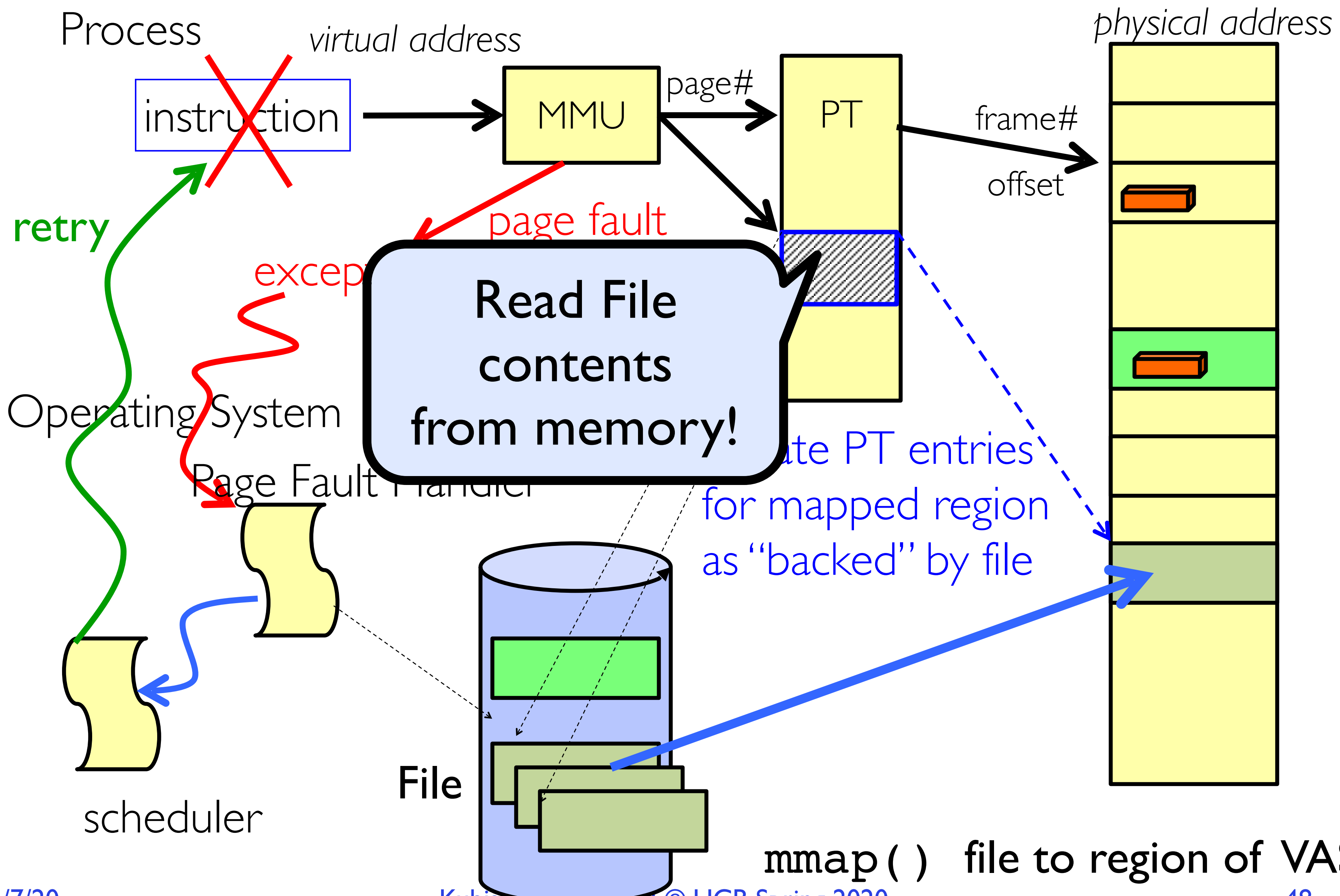
Memory Mapped Files

- 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()` 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
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- 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 unsigned int) &something);
  printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CREAT);
  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 : %16lx\n", (long unsigned int) mfile);

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

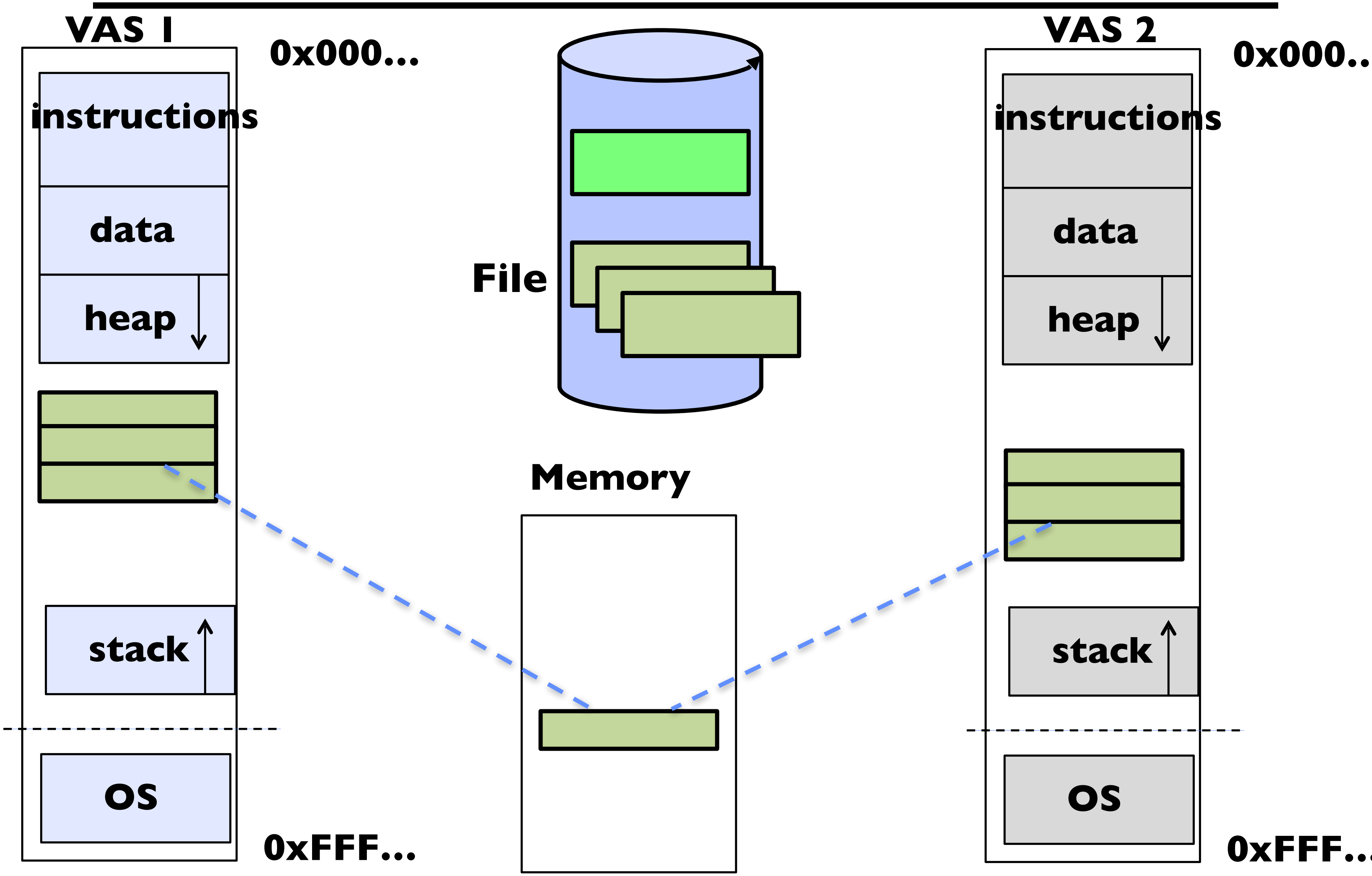
An mmap( ) Example

```
#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", (lo
  printf("Heap at : %16lx\n", (lo
  printf("Stack at: %16lx\n", (lo

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_
  if (myfd < 0) { perror("open fail

  /* map the file */
  mfile = mmap(0, 10000, PROT_REA
  if (mfile == MAP_FAILED) {perro

  printf("mmap at : %16lx\n", (lo

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

```
$ cat test
This is line one
This is line two
This is line three
This is line four
$ ./mmap test
Data  at:        105d63058
Heap at :     7f8a33c04b70
Stack at:     7fff59e9db10
mmap at :        105d97000
```

```
$ cat test
This is line one
ThiLet's write over its line three
This is line four
```

Sharing through Mapped Files



- Also: anonymous memory between parents and children
  - no file backing – just swap space

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

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
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  – Can contain "dirty" blocks (blocks yet on disk)