CS162: Operating Systems and Systems Programming

Lecture 23: Queuing Theory / Filesystems (Intro)

29 July 2015

Charles Reiss

https://cs162.eecs.berkeley.edu/

Recall: Storage Devices

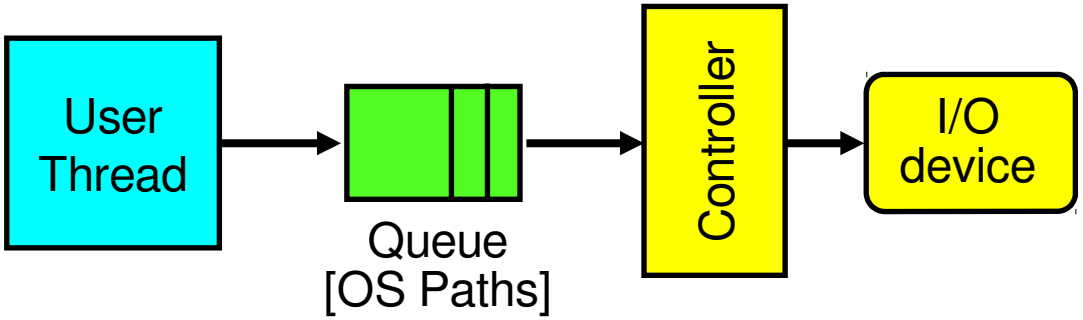
Magnetic disks

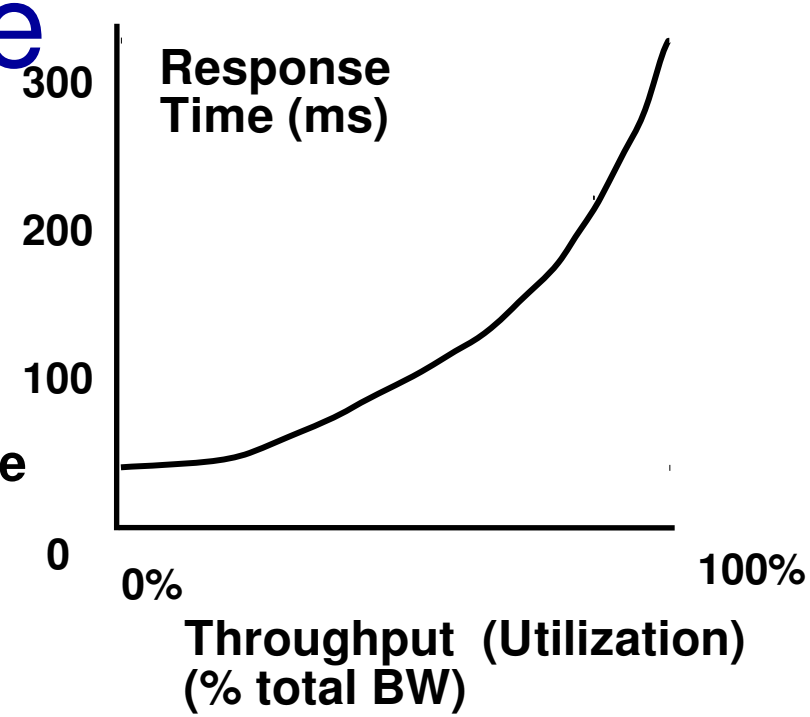
- Storage that rarely becomes corrupted
- Large capacity at low cost
- Block level random access
- Slow performance for random access
- Better performance for streaming access

Flash memory

- Storage that rarely becomes corrupted
- Capacity at intermediate cost (50x disk ???)
- Block level random access
- Good performance for reads; worse for random writes
- Erasure requirement in large blocks
- Wear patterns

Recall: I/O Performance





Response Time = Queue + I/O device service time

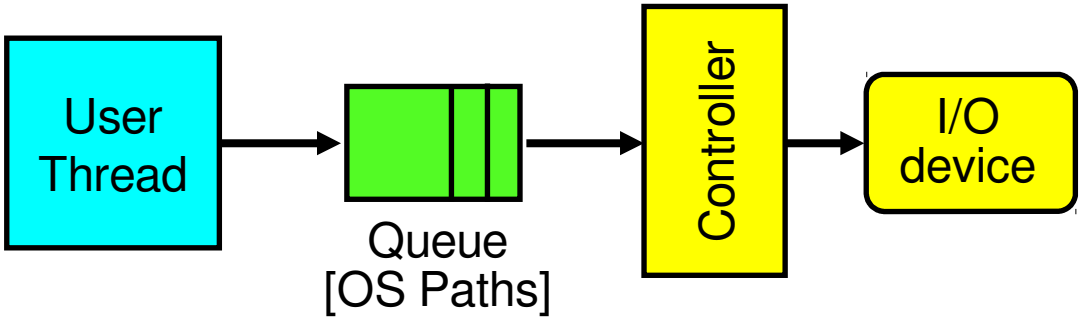
Latency:

- **Wait time** – Software paths – model as *queue*
- Controller time – constant? queue?
- Actual **service time**

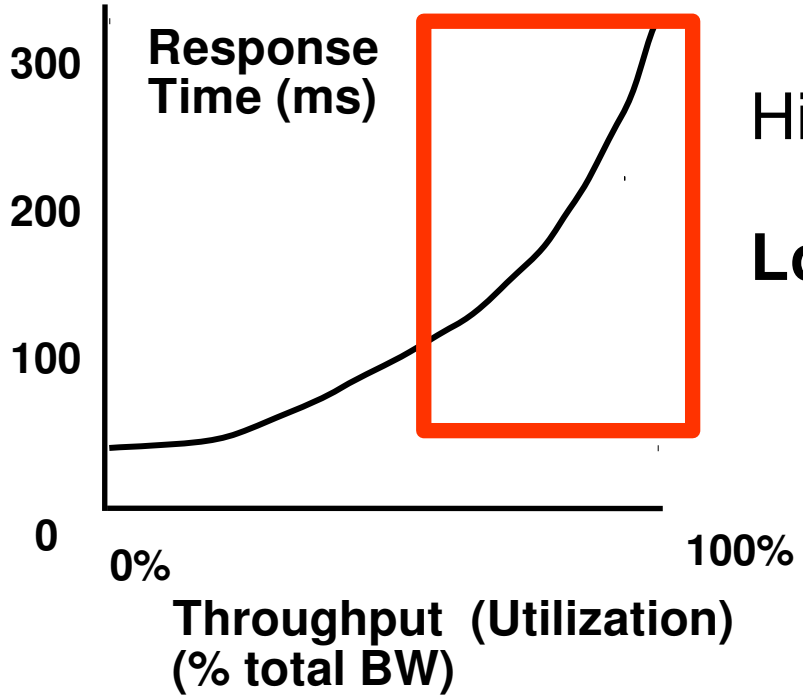
Queuing behavior:

- Can lead to big increases of latency as utilization increases
- Solutions?

Recall: I/O Performance



Response Time = Queue + I/O device service time



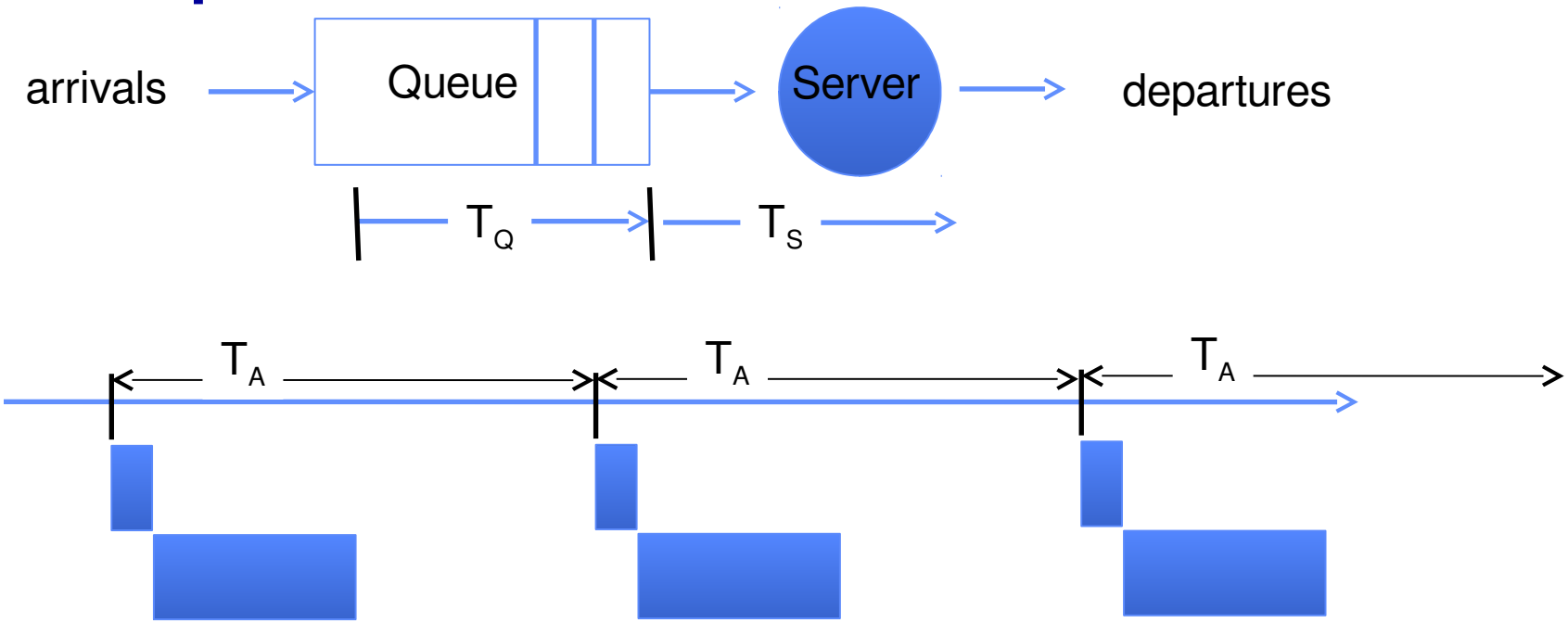
Long queues → more wait time

High throughput *overall*

Low effective throughput *per operation*

Effective BW = *size* / *response time*

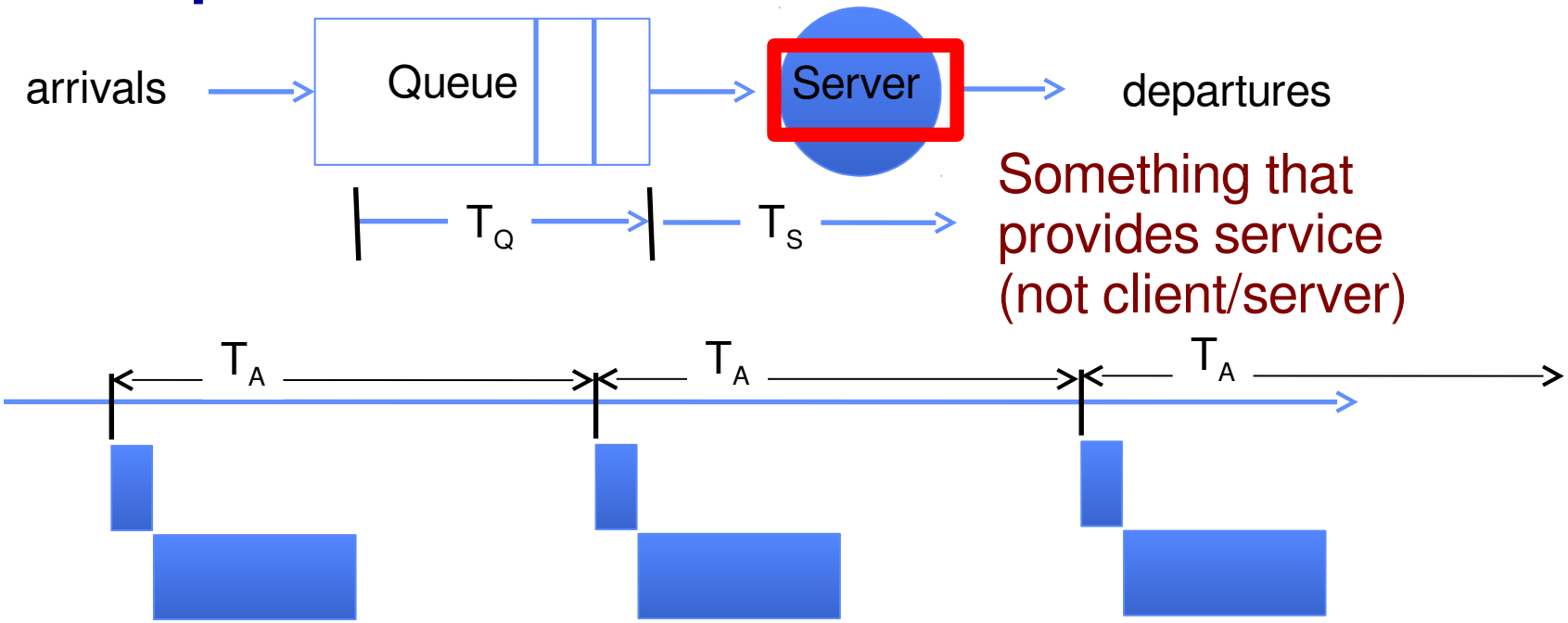
A Simple Deterministic World



Requets arrive at *fixed intervals* and take *constant amount of time to service*

Service Rate	$\mu = 1/T_S$	operations per second
Arrival Rate	$\lambda = 1/T_A$	requests per second
Utilization	$U = \lambda/\mu$	%

A Simple Deterministic World

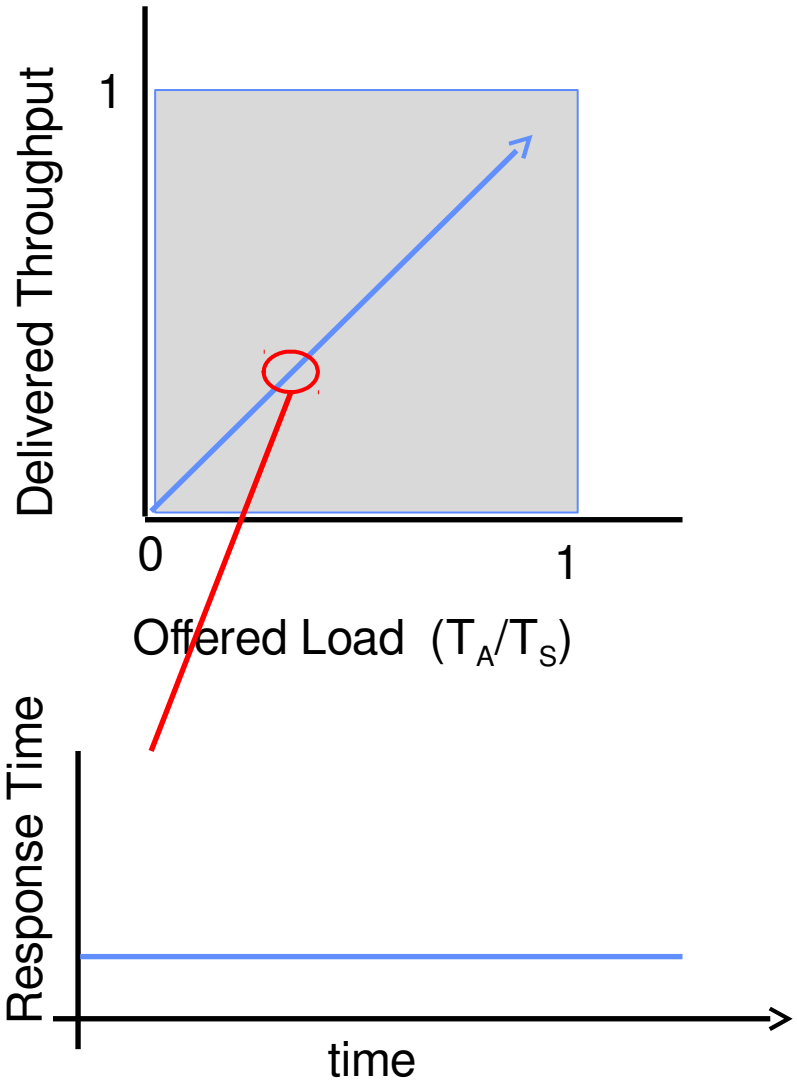


Requets arrive at *fixed intervals* and take *constant amount of time to service*

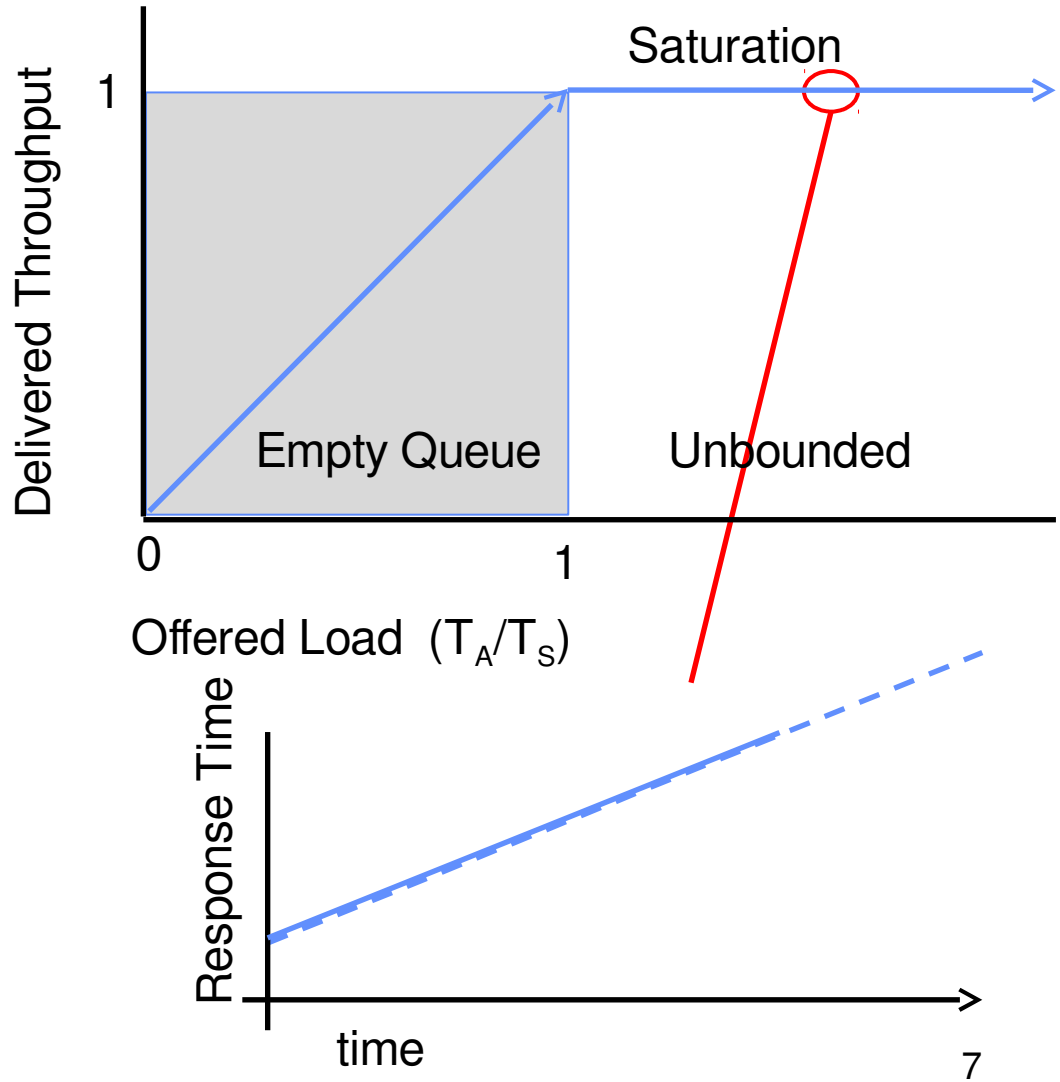
Service Rate	$\mu = 1/T_S$	operations per second
Arrival Rate	$\lambda = 1/T_A$	requests per second
Utilization	$U = \lambda/\mu$	%
Offered Load	T_A/T_S	%

A Ideal Linear World

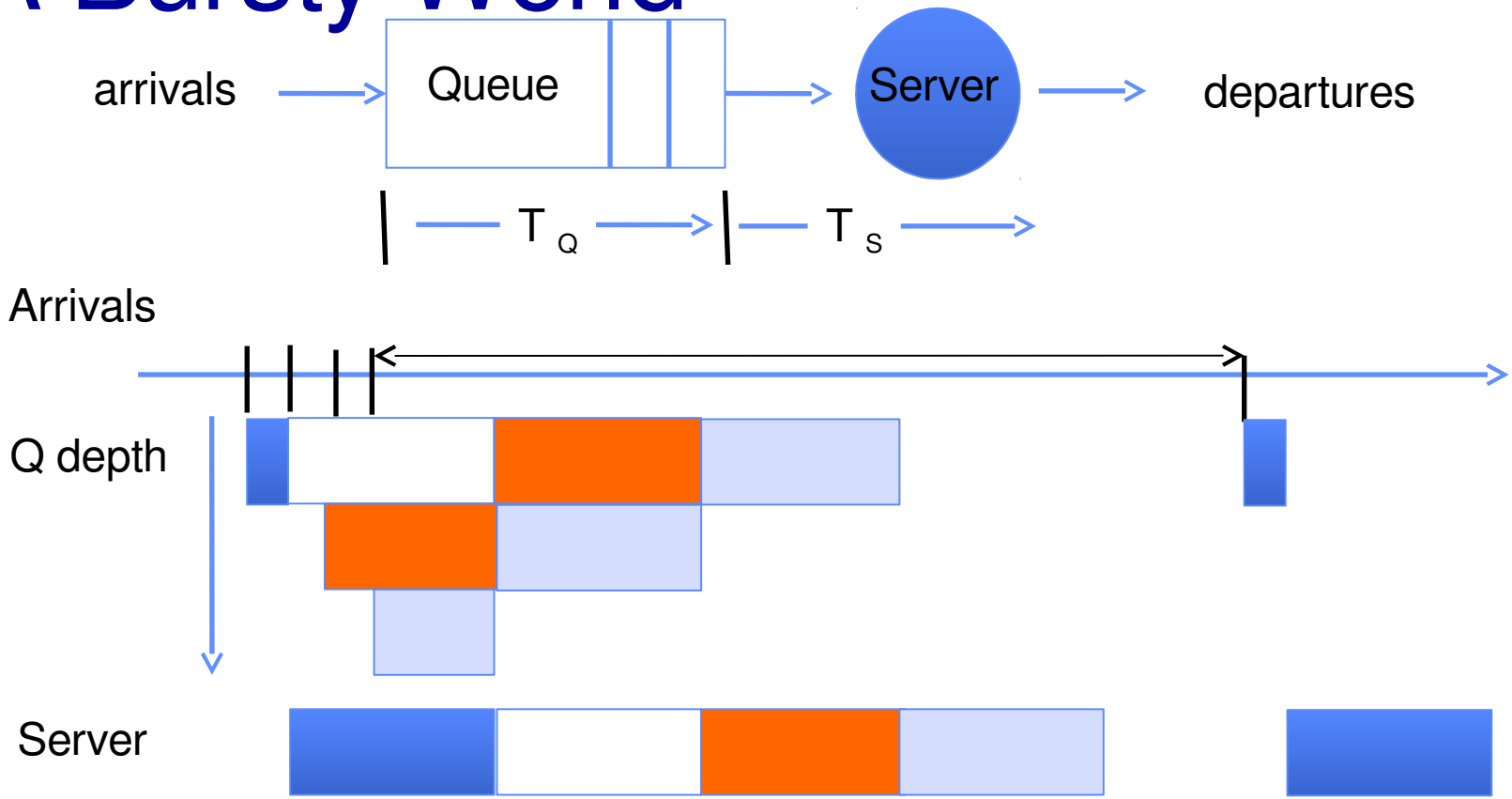
Offered Load < 100%



Offered Load > 100%



A Bursty World



Same **average arrival time**, but almost all of the requests experience **large queue delays**

Even though average utilization is low

Modeling Uneven Arrivals

Simplest assumption: arrivals are equally likely at any time

- chance of 1 arrival between 1pm and 2pm = chance of 1 arrival between 2pm and 3pm
- chance of 1 arrival between 1:00pm and 1:01pm = chance of 1 arrival between 2:00pm and 2:01pm

Memoryless

- Doesn't matter when last arrival happened

Modeling Uneven Arrivals

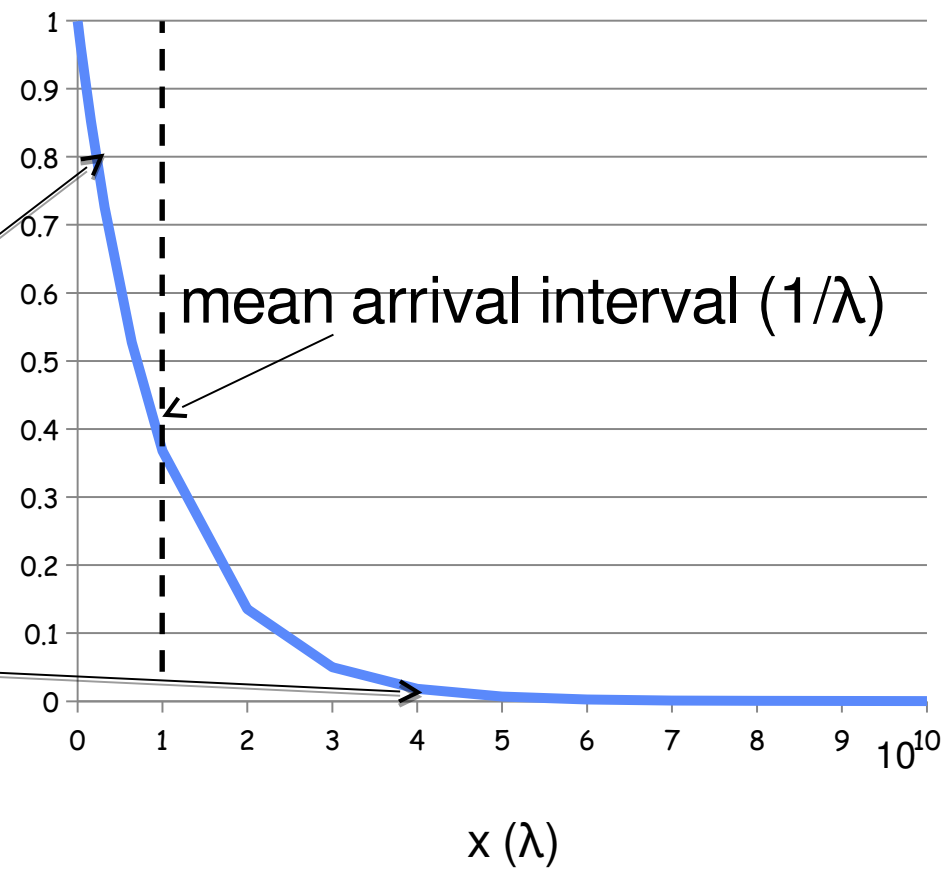
Memoryless property implies time between arrivals follows **exponential distribution**

- probability distribution function $f(x) = \lambda e^{-\lambda x}$
- mean $1/\lambda$

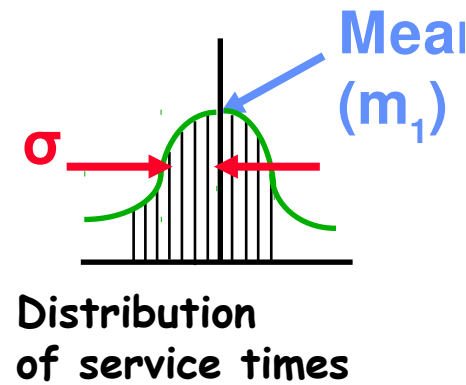
Likelihood of an event occuring is independent of how long we've been waiting

Lots of short arrival intervals (i.e., high instantaneous rate)

Few long gaps (i.e., low instantaneous rate)

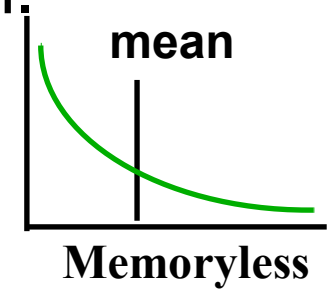


Other distribution



Generally, some probability distribution:

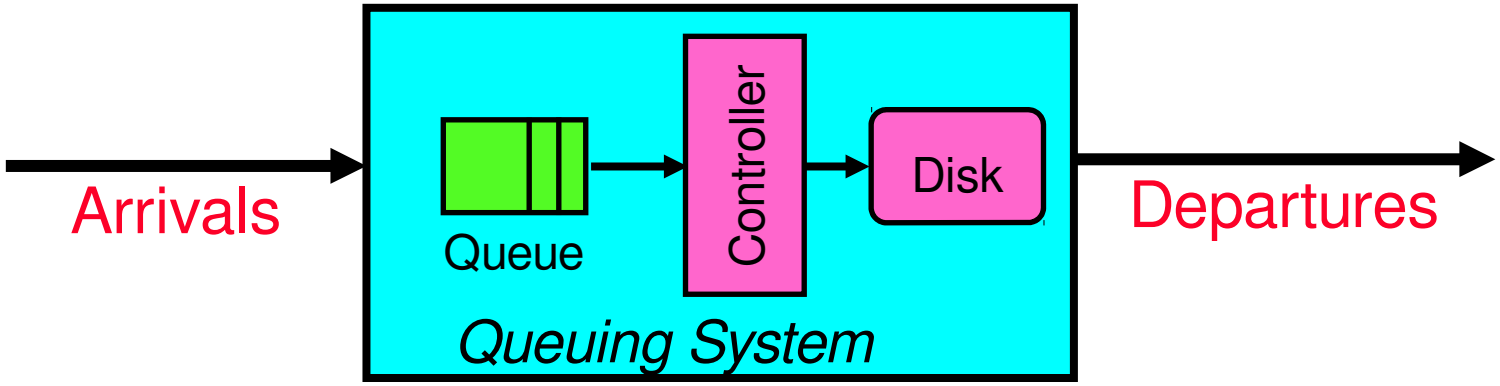
- Mean (Average) $m_1 = \Sigma p(T) \bullet T$
- Variance $\sigma^2 = \Sigma p(T) \bullet (T - m_1)^2 = \Sigma p(T) \bullet (T^2 - m_1^2)$
- **Squared coefficient of variance**: $C = \sigma^2 / m_1^2$
  Aggregate description of the distribution.



Important values of C:

- No variance or **deterministic**: C=0
- "**memoryless**" or exponential: C=1
- Disk response times C ≈ 1.5 (majority seeks < avg)

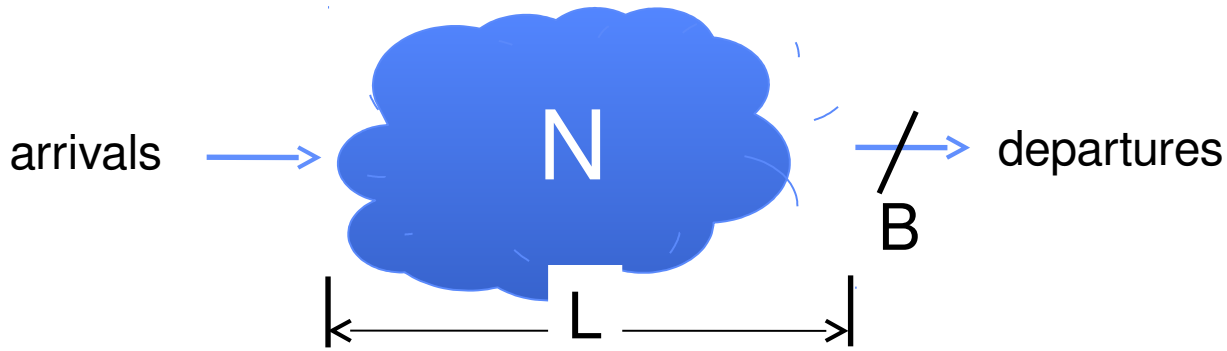
Queuing Theory



Queuing Theory: studies long term, steady state behavior of *queuing systems*

Based on queue behavior, and *probability distributions of arrival, service times*

Little's Law:
What goes in must come out (1)

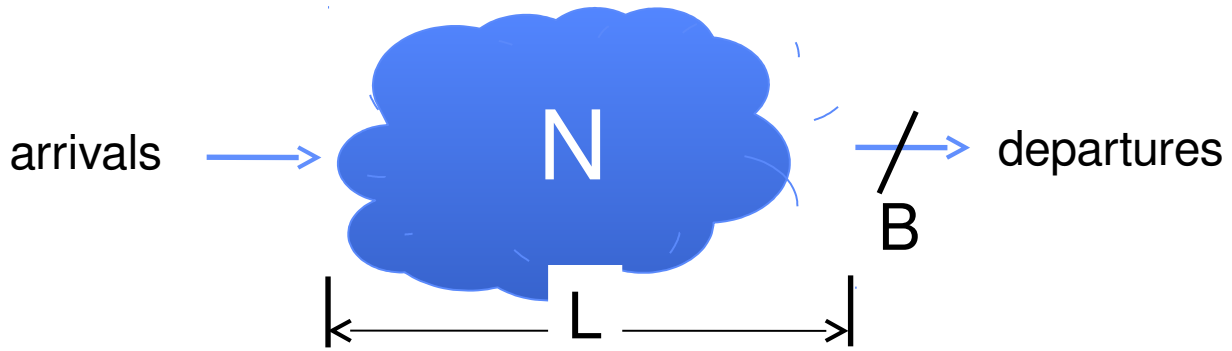


Avg arrival rate = Avg departure rate = Throughput/Bandwidth *B*

- or queue is growing to infinity

Avg latency = L

Little's Law: What goes in must come out (2)

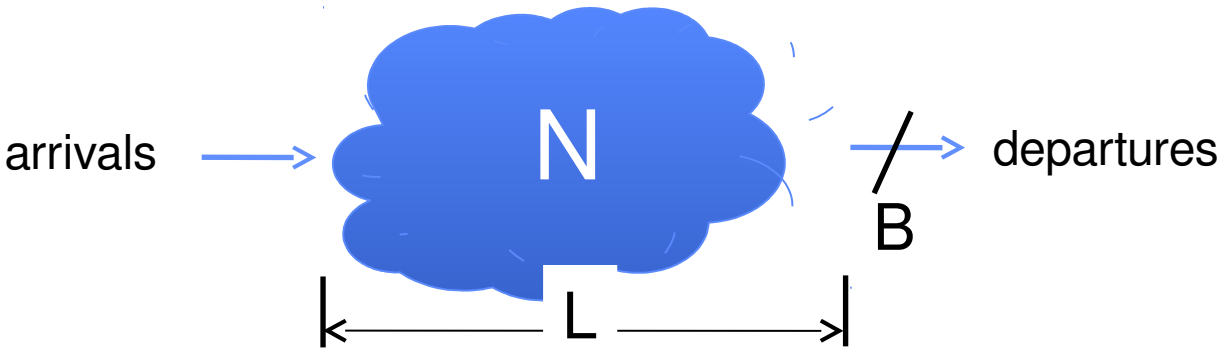


How many items are in the system?

- **B** items come in per unit time
- each averages **L** units in the system

→ **N** (# requests in the system) = **B** (ops/s) x **L** (s)

Little's Law: What goes in must come out (3)



N (# requests in the system) = **B** (ops/s) x **L** (s)

Called **Little's Law**

About *averages*

Works *regardless of arrival, service distribution*

- (assuming system is stable)

Naming Queuing Systems

X/X/N

# of requests serviced at a time (usually 1)

Distribution of Service Times

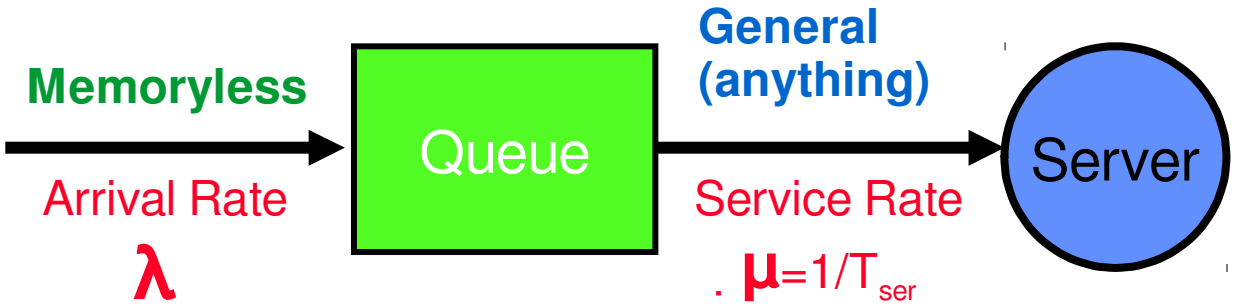
Distribution of Arrival Times

Distributions

M – memoryless (exponential)
G – general (anything)
D – deterministic (constant)

Results for **Memoryless** Arrivals

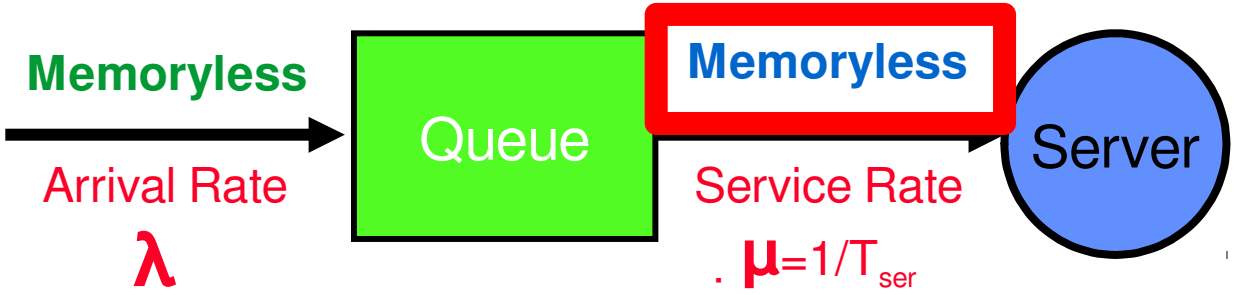
M/**G**/1



λ	mean number of requests/second
T_{ser}	mean time to service a customer (m_1)
C	squared coeff. variance of service time (σ^2/m_1^2)
μ	service rate = $1/T_{ser}$
U	server utilization ($0 \le U \le 1$): $U = \lambda/\mu = \lambda \times T_{ser}$
T_q	**Time in queue = $T_{ser} \times U/(1-U) \times (1+C)/2$**
L_q	**Length of queue** = $\lambda \times T_q$ (from Little's Law)

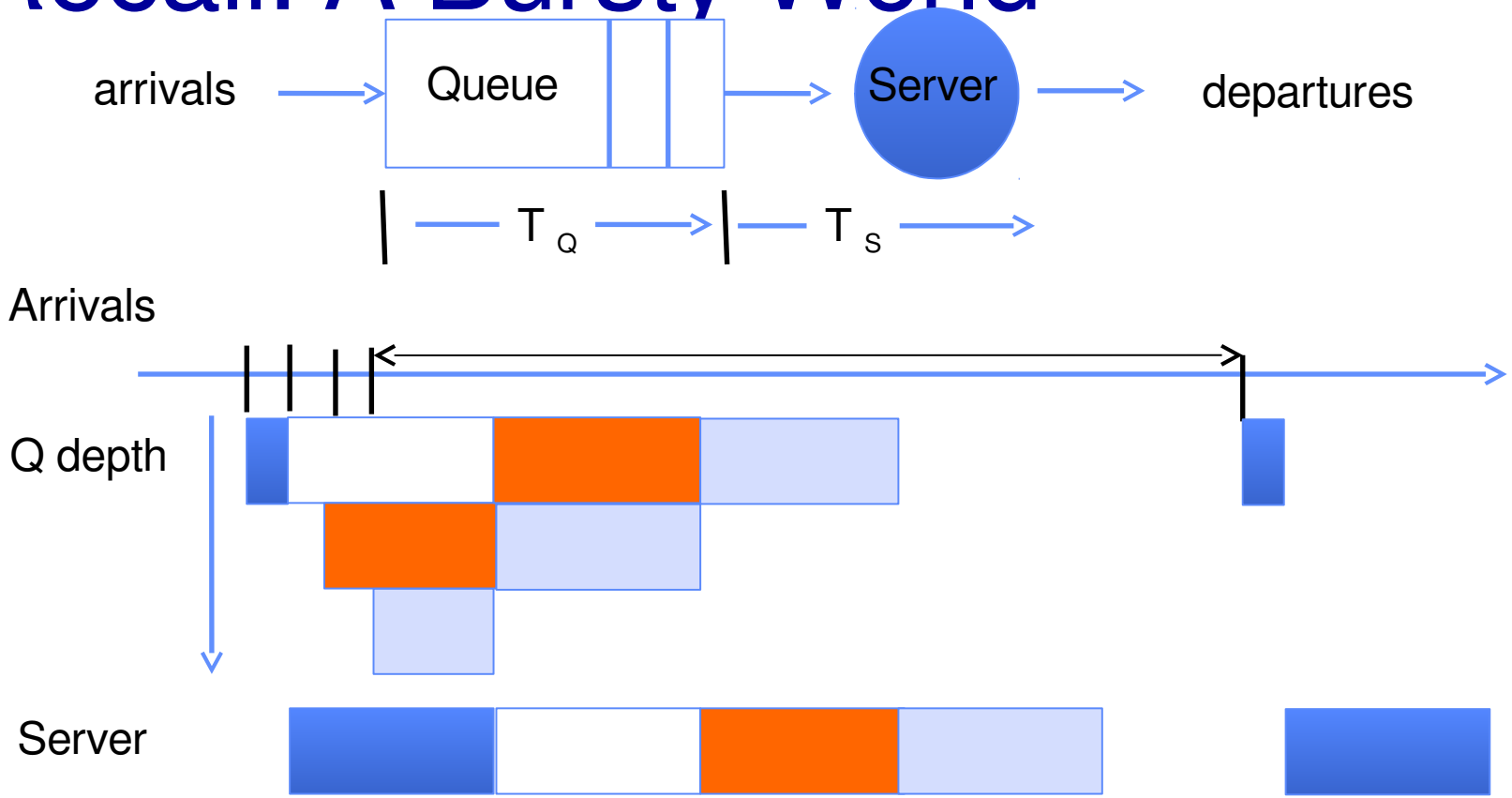
... for **Memoryless** Arrival + Service

M/**M**/1



λ	mean number of requests/second
T_{ser}	mean time to service a customer (m_1)
C **= 1**	quared coeff. variance of service time (σ^2/m_1^2)
μ	service rate = $1/T_{ser}$
U	server utilization ($0 \le U \le 1$): $U = \lambda/\mu = \lambda \times T_{ser}$
T_q	**Time in queue = T_{ser} x U/(1-U)** ~~**x (1+C)/2**~~
L_q	**Length of queue** = $\lambda \times T_q$ (from Little's Law)

Recall: A Bursty World



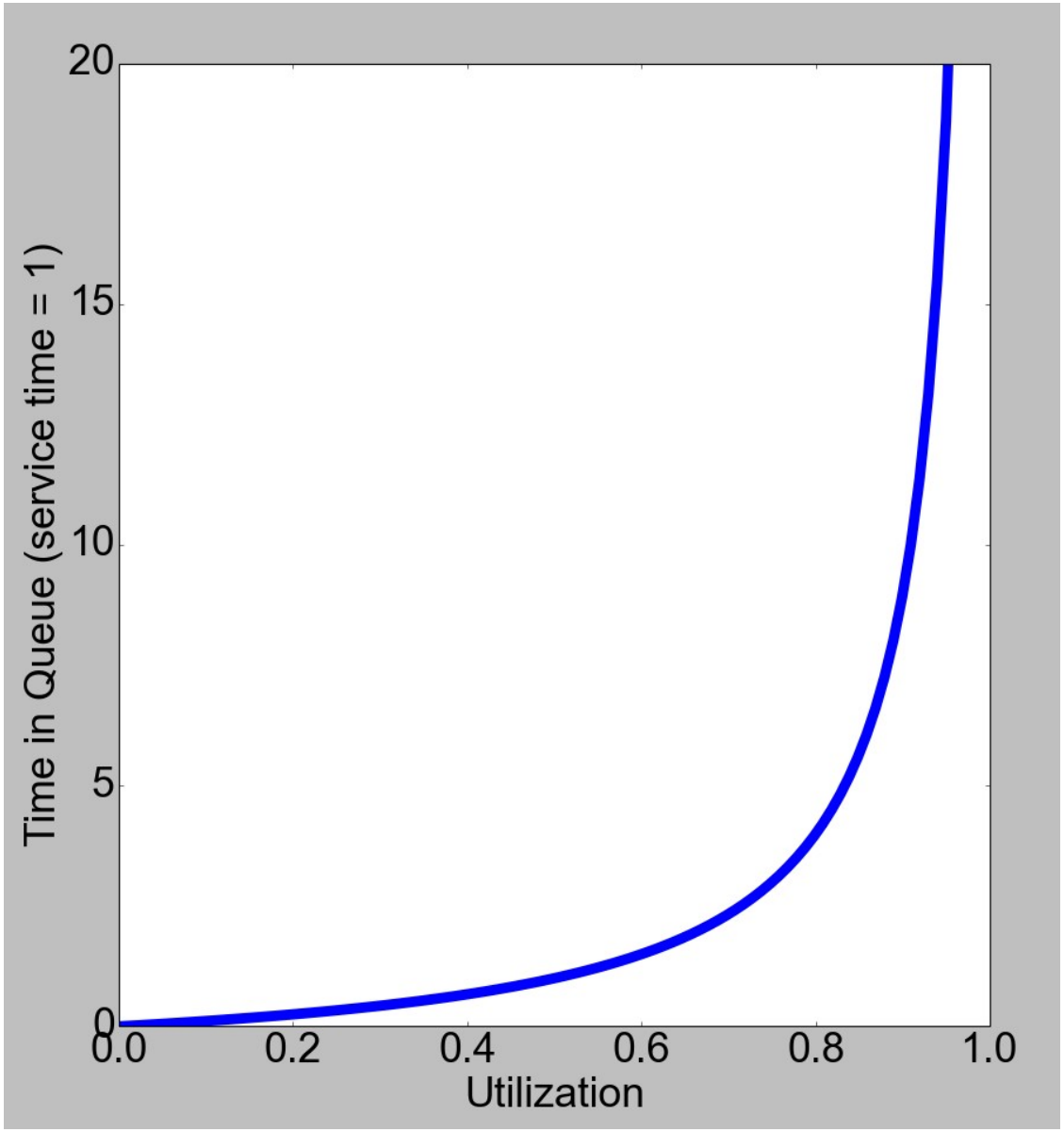
Same **average arrival time**, but almost all of the requests experience **large queue delays**

Even though average utilization is low

The Utilization Curve

Time in queue =
$T_{ser} \times U/(1-U)$

Cannot get 100% utilization



Queuing Theory Example

- Ten 8KB disk I/Os per second (λ = 10/s)
- Memoryless arrival + service (M/M/1 queue)
- Average service time = T_{ser}= 20 ms (avg controller + seek + rotational delay + transfer time)

Questions

- Utilization?
  - $U = \lambda T_{ser}$ = 10/s x 0.02 s = **20%**
- Average time spent in queue?
  - $T_q = T_{ser} U/(1-U)$ = 0.02 s x 0.2 / 0.8 = **5 ms**
- Average number waiting requests?
  - $L_q = \lambda T_q$ = 10/s x 0.005 s = **0.05 reqs**
- Average response time?
  - $T_{sys} = T_q + T_{ser}$ = 5 ms + 20 ms = **25 ms**

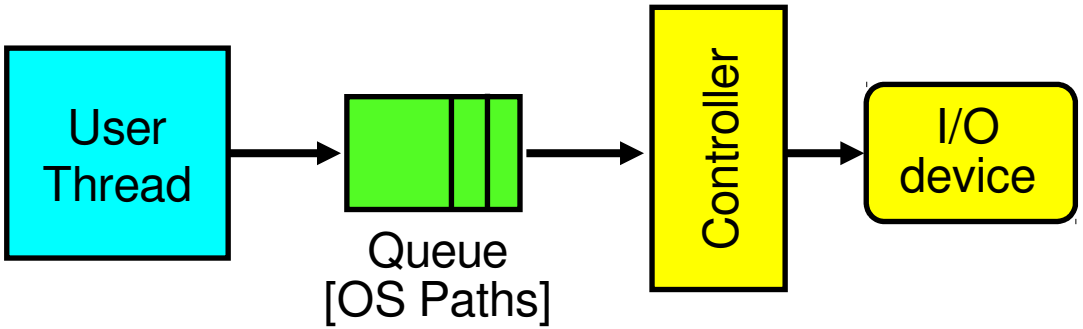
Queuing Theory Resources

Handouts page contains Queueing Theory Resources:

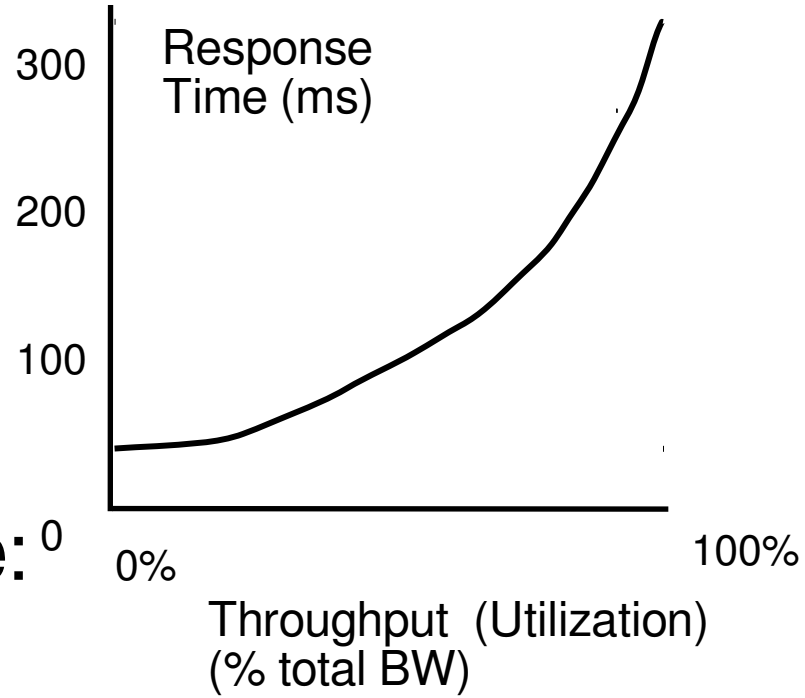
- Scanned pages from Patterson and Hennesey book that gives further discussion and simple proof for general eq.
- A complete website full of resources

Assume that Queueing theory is fair game for Final!

Optimize I/O Performance



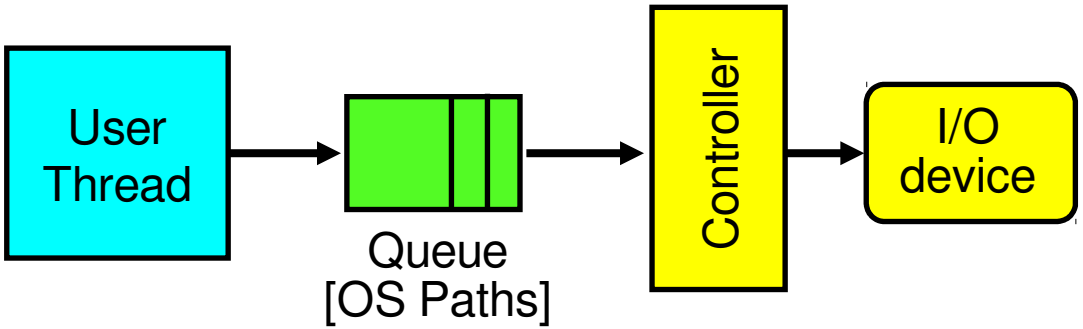
Response Time =
Queue + I/O device service time



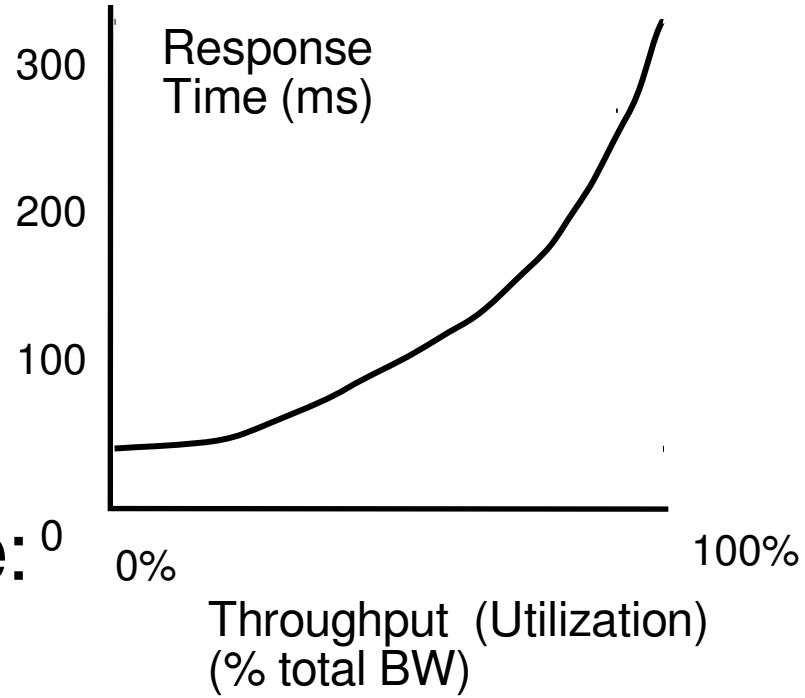
Options to improve performance:

- Improve service time
- Multiple servers – e.g. use two disks instead of one
- Do more useful work while waiting
- Admission control: don't allow too many threads
  - Response time over throughput

Optimize I/O Performance



Response Time =
Queue + I/O device service time

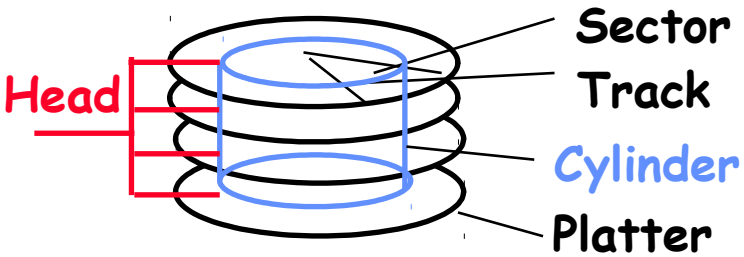


Options to improve performance:

- **Improve service time**
- Multiple servers – e.g. use two disks instead of one
- Do more useful work while waiting
- Admission control: don't allow too many threads
  - Response time over throughput

Recall: Reading and Writing

1. **Seek time** – move heads to the correct cylinder



avg **5-10 ms**
faster if reads adjacent

2. **Rotational latency** – wait for sector to come under heads

~**4-8 ms** (3600-7200 rpm, typical laptop/desktop)
~2-4 ms (15000 rpm; high-end server)
faster if reads adjacent

3. **Transfer time** – time to actually read the sectors

50-100 MB/sec

When does disk perform best?

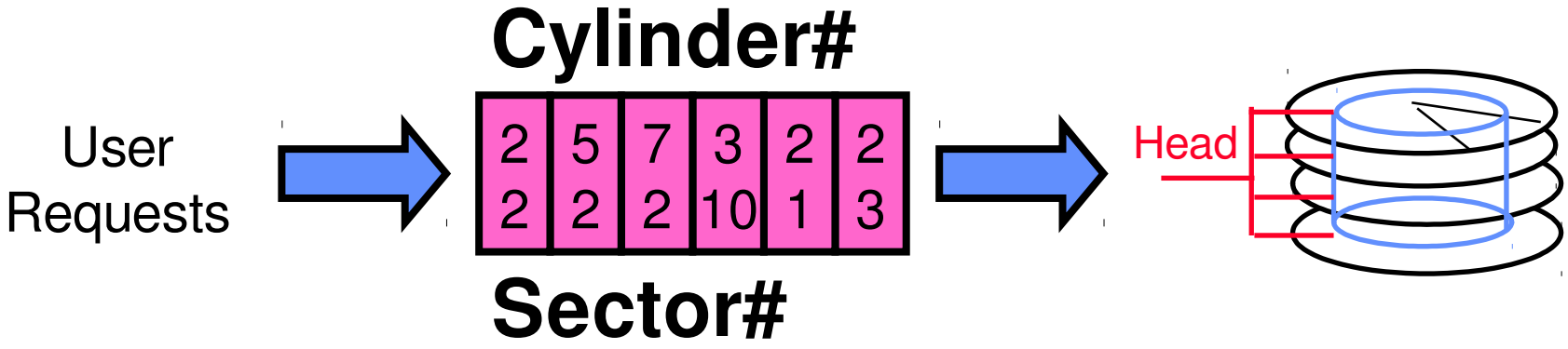
Big sequential reads

Reads are in order so there's no "backtracking" when seeking/rotating

Idea: sort disk queue to minimize seek times

- Needs multiple independent reads

Disk Scheduling



How do we pick from the queue?

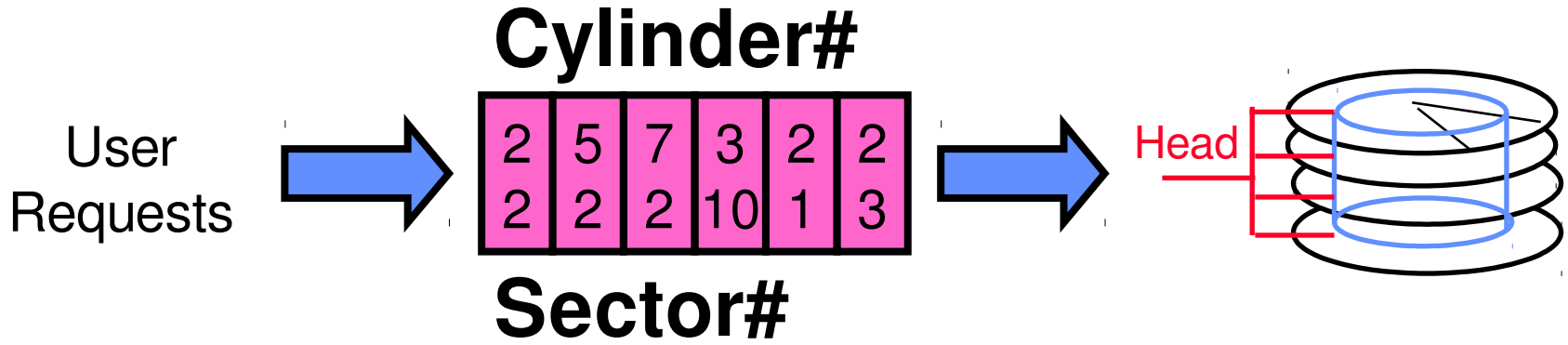
Go from cylinder 2 to cylinder 2?

- No seek time

Go from cylinder 3 to cylinder 7?

- Seek time

Disk Scheduling

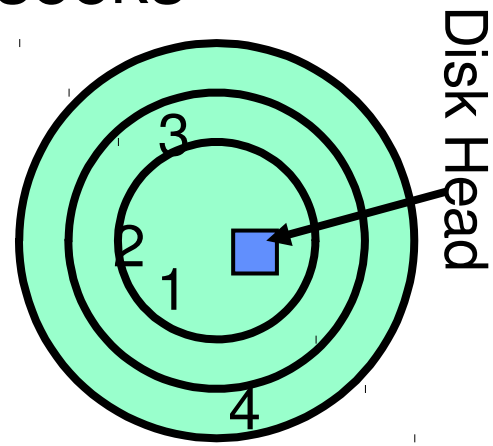


FIFO – First-In, First-Out

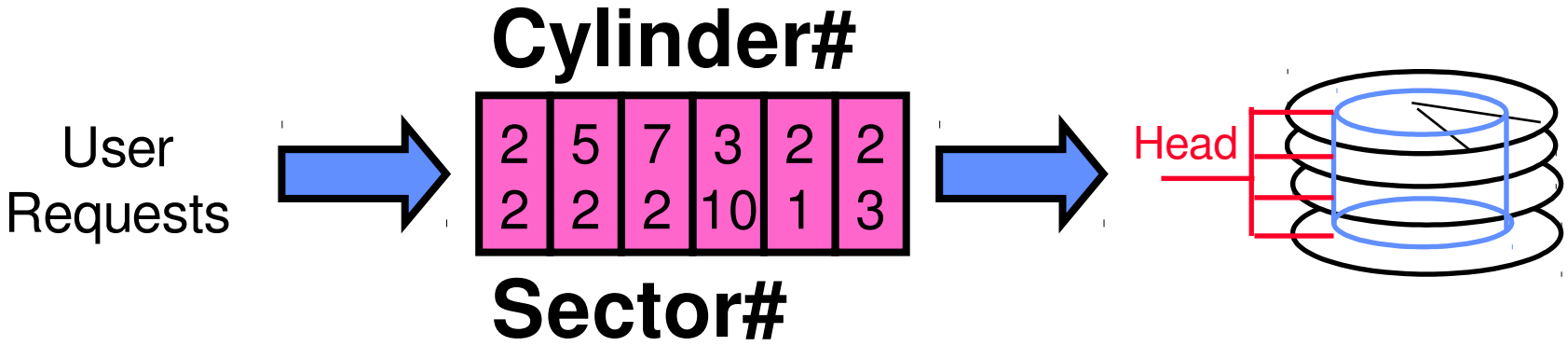
- Simplest, Fair, but encourages very long seeks

SSTF – Shortest Seek Time First

- Pick the closest request
- Problem: **Starvation** – never leaves "current" section of disk



Disk Scheduling

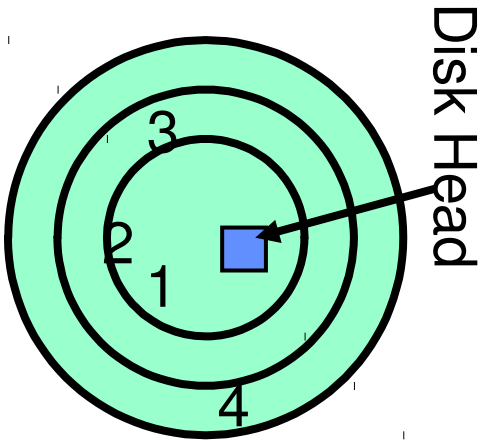


SCAN – "Elevator" Algorithm

- Go from inside to outside and then back
- Cylinder #1 → #N → #1 → #N → ...
- Favors middle cylinders of disk

C-SCAN – Circular Scan

- Like SCAN, but only one direction
- Cylinder #1 → N; #1 → N; ...
- Fairer than SCAN



Recall: Intelligence in the Controller

Does OS know what cylinder it's using?

Used to address sectors with cylinder #, head #, track #

Not anymore – opaque sector #s

- Numerically close sector # are close to each other

Track skewing

- Different cylinders have different numbers of sectors

Disk Scheduling

Also something controller does (disk controller has its own queue)

More criteria to optimize for, like process scheduling

- Fairness versus Throughput versus Response Time

Recall: When does disk perform best?

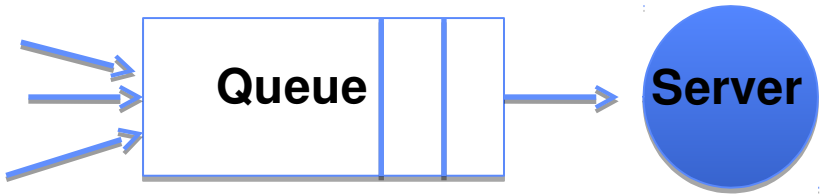
Big sequential reads

Reads are in order so there's no "backtracking" when seeking/rotating

Idea: sort disk queue to minimize seek times

- Needs multiple independent reads

Performance: multiple outstanding requests



Suppose each read takes 10 ms to service.

If a process works for 100 ms after each read, what is the utilization of the disk?

- U = 10 ms / 110ms = 9%

What it there are two such processes?

- U = (10 ms + 10 ms) / 110ms = 18%

What if each of those processes have two such threads?

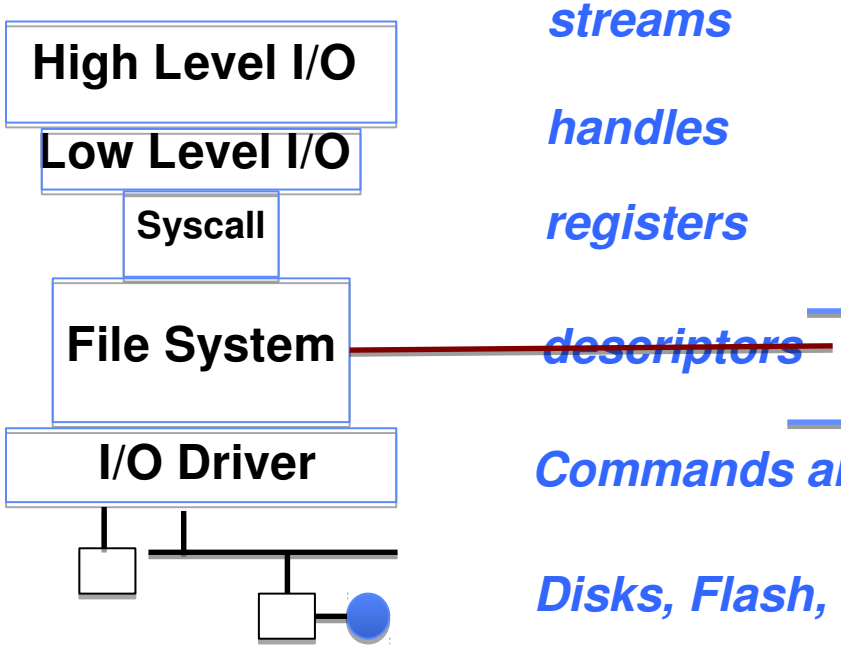
Logistics

Break

I/O & Storage Layers

Operations, Entities and Interface

Application / Service



streams

handles

registers

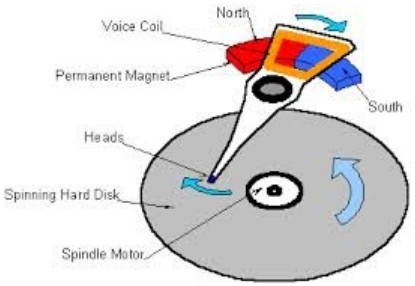
file_open, file_read, ... on struct file * & void *

descriptors

we are here …

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Recall: C Low level I/O

Operations on File Descriptors – as OS object representing the state of a file

- User has a "handle" on the descriptor

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:

- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:

- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

Recall: C Low Level Operations

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) - wait for i/o to finish
void sync (void) - wait for ALL to finish
```

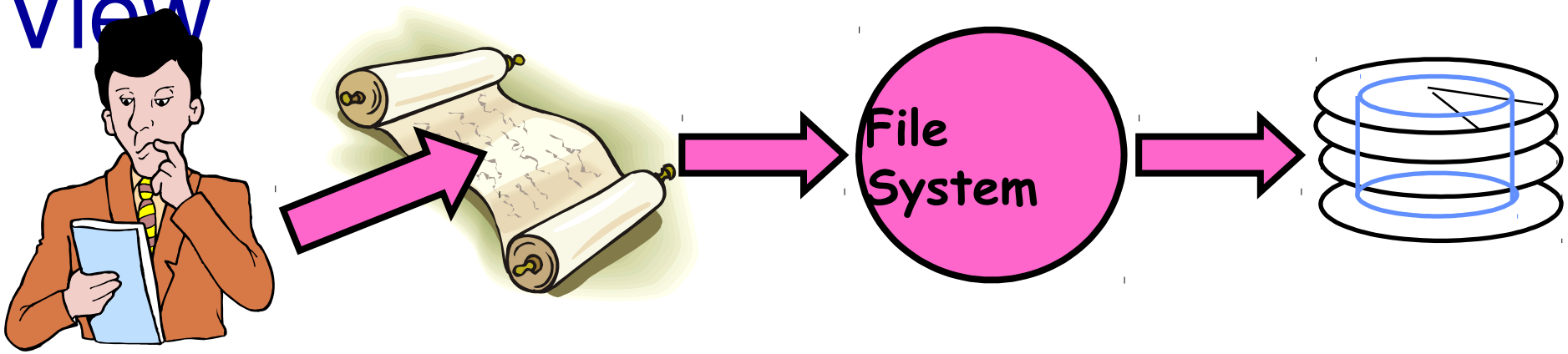
When write returns, data is on its way to disk and can be read, but it may not actually be permanent!

Building a File System

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 may not be sector size; in UNIX, block size is 4KB

Translating from User to System View



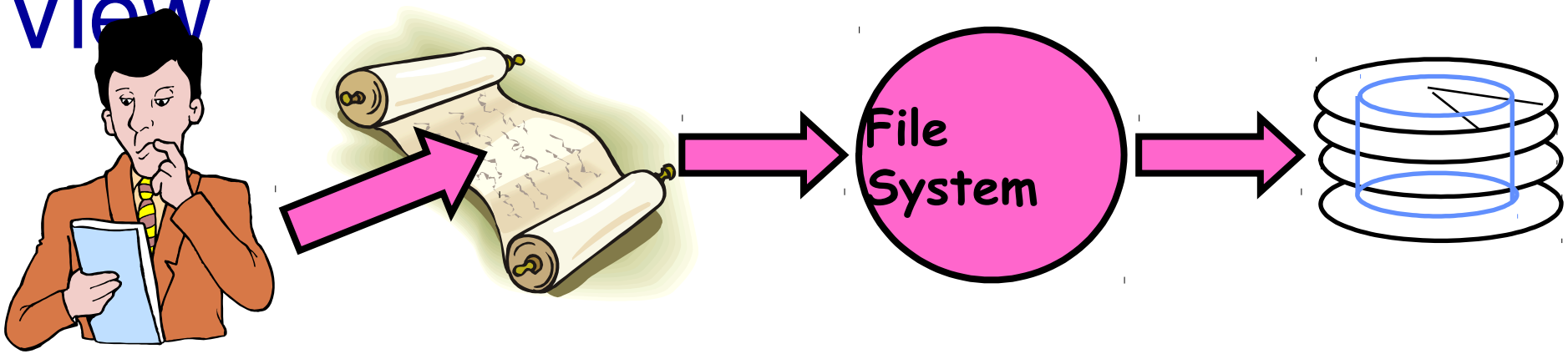
What happens if user says: give me bytes 2—12?

- Fetch block corresponding to those bytes
- Return just the correct portion of the block

What about: write bytes 2—12?

- Fetch block
- Modify portion
- Write out block

Translating from User to System View



Everything inside File System is in whole size blocks

- Actual disk I/O only happens in whole blocks
- read()/write() of less than blocks need to translate/buffer

From now on, file is a collection of blocks

Disk Management Policies

Basic entities on a disk:

- **File**: user-visible group of blocks arranged sequentially in logical space
- **Directory**: user-visible index mapping names to files

Access disk as linear array of sectors

- Controller translates from address = physical position
  - First case: OS/BIOS must deal with bad sectors
  - Second case: hardware shields OS from structure of disk

Things Our Filesystem Needs

Track free disk blocks

- Need to know where to put data that's written

Track blocks containing parts of files

- Need to know where to read a file from

Track files in a directory

- Need to find list of blocks given a name

Where do we track all of this?

- Somewhere on the disk

Things Our Filesystem Needs

Track free disk blocks

- Need to know where to put data that's written

Track blocks containing parts of files

- Need to know where to read a file from

Track files in a directory

- Need to find list of blocks given a name

Where do we track all of this?

- **Somewhere on the disk**

Data Structures on Disk

Not quite like data structures in memory

Access in blocks:

- Can't efficiently read or write a single word – have to read or write the whole block containing it
- Ideally want to read/write blocks **sequentially**

Durable:

- Need to make sure filesystem is in a sane state if power lost
- You may know that this doesn't always happen

Tracking Free Blocks: Option 1

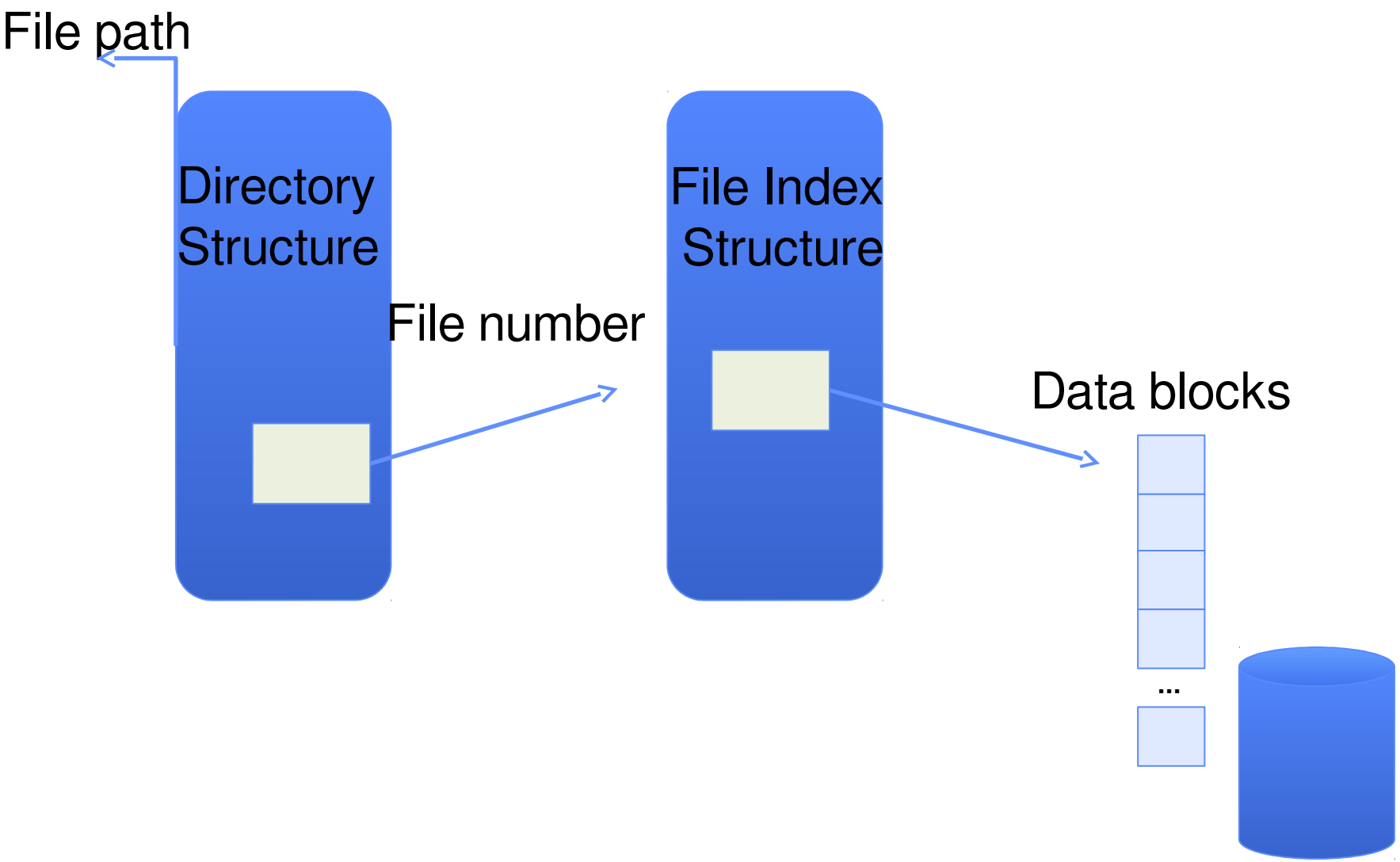
Bitmap on disk (at well-known block #s)

- Bit #i is 0 if block #i is free, 1 otherwise
- Scan sequentially until 1
- Rewrite whole block to update

Isn't this slow?

- Make faster with caching
- Finding free block is **sequential**

Components of a File System



Components of a file system

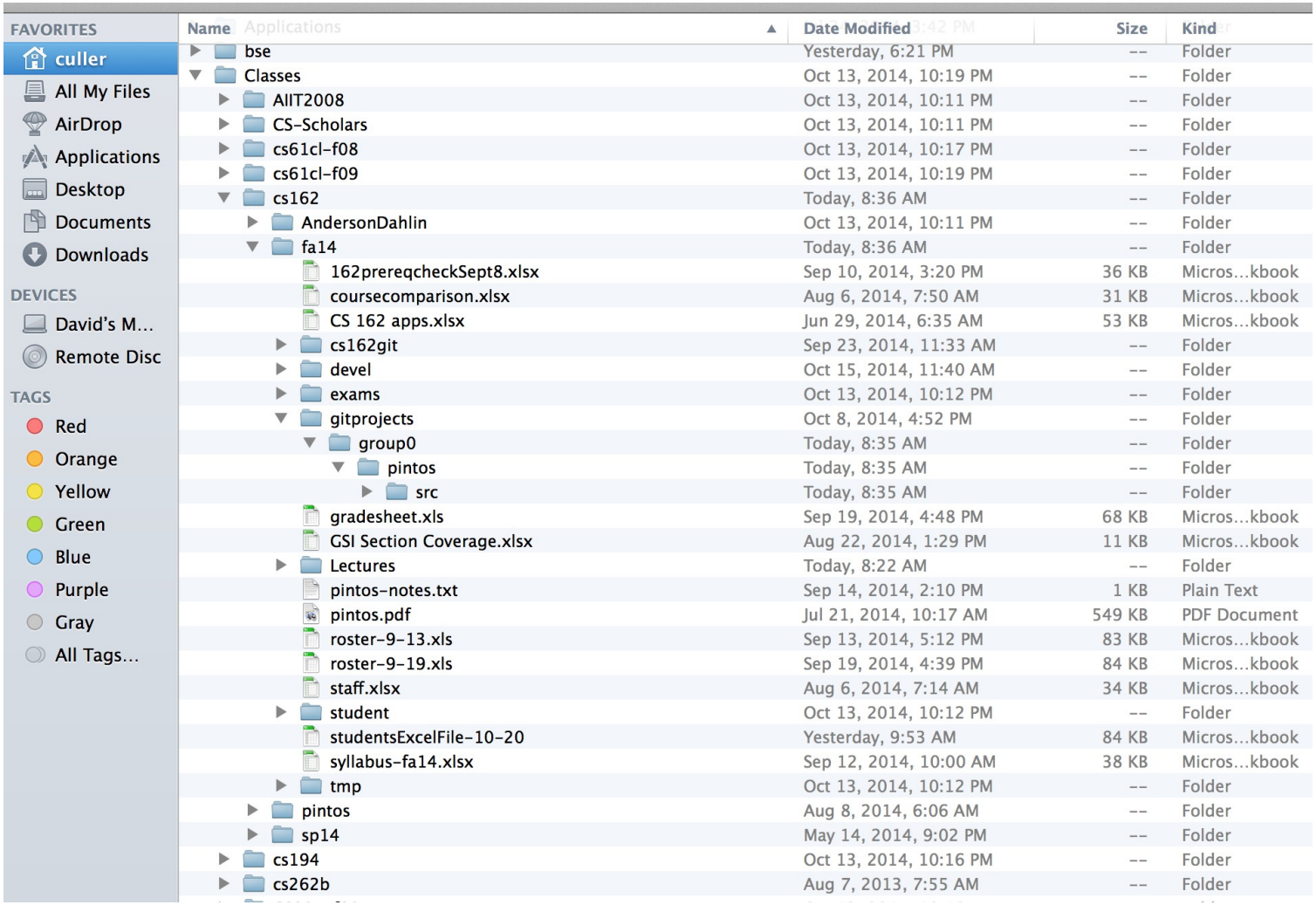
Open performs **name resolution**

- Translates pathname into a "**file number**"
  - Used as an "index" to locate the blocks
- Creates a open file description in PCB within kernel
- Returns a file descriptor (int) to user process

Read, **Write**, **Seek**, and **Sync** operate on handle

- Mapped to descriptor and to blocks

Directories



Directory

Basically a **hierarchical** structure

Each **directory entry** is a collection of

- Regular Files
- Directories
  - A link to another entries

Each has a **name** and **attributes**

- Files have data

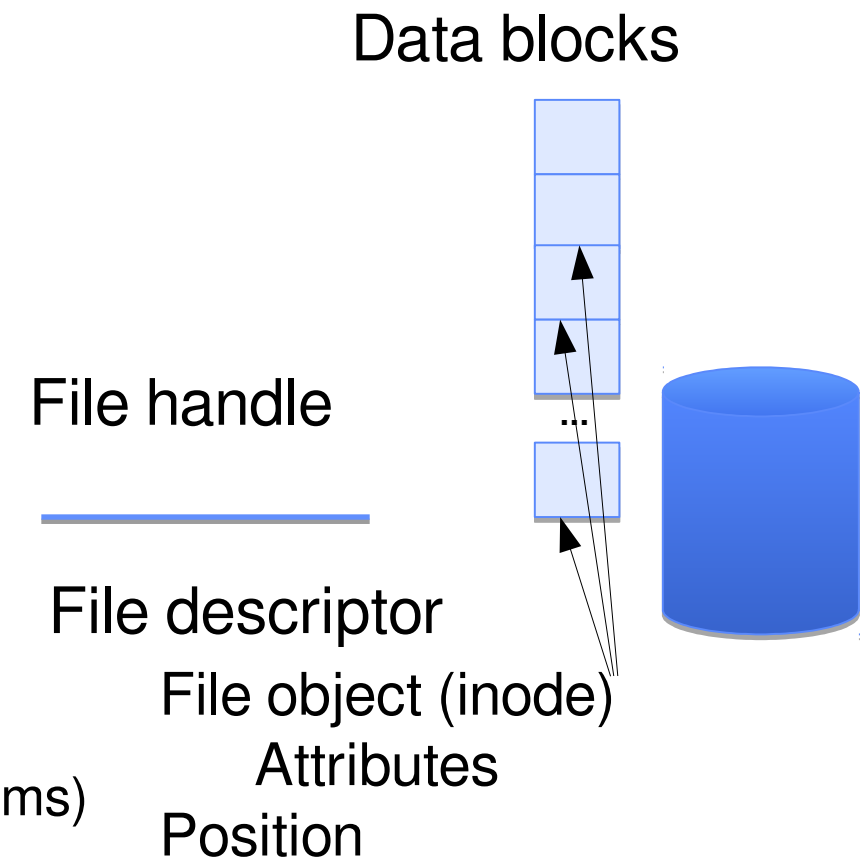
Links (hard links – tomorrow) make it a DAG, not just a tree

- **Softlinks** (aliases) are another name for an entry

File

Named permanent storage:

- **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



Performance Summary

Disk Performance:

- Queuing time + Controller + Seek + Rotational + Transfer
- Rotational latency: on average ½ rotation
- *Scheduling* to minimize seek + rotational time

Queuing Latency:

- M/M/1 and M/G/1 queues: simplest to analyze
- As utilization approaches 100%, latency → infinity
- time in queue = $T_{ser} \times U/(1-U) \times (1+C)/2$

File System Intro

File System:

- Transforms blocks into Files and Directories
- Optimize for access and usage patterns
- Maximize sequential access, allow efficient random access

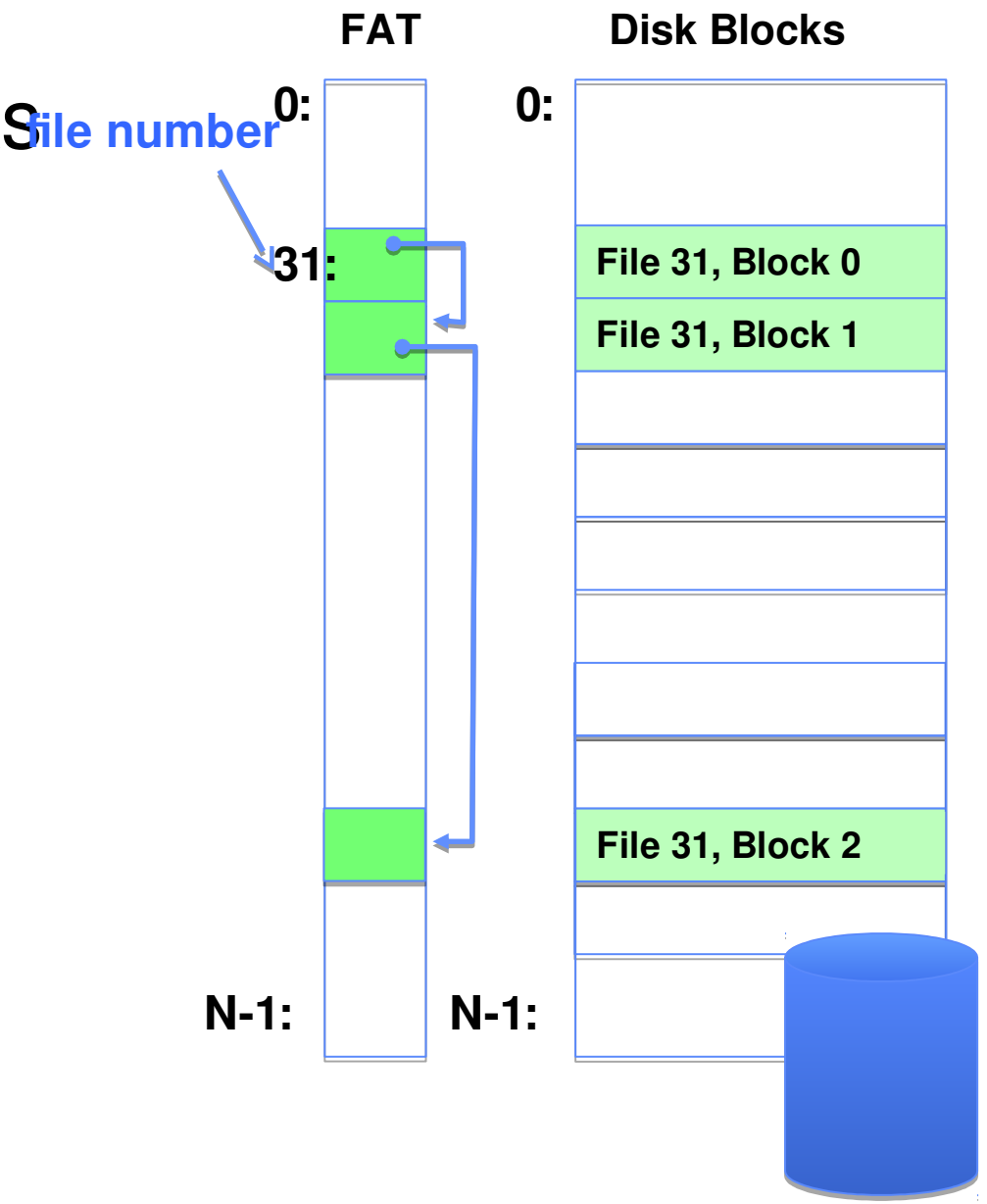
FAT (File Allocation Table)

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

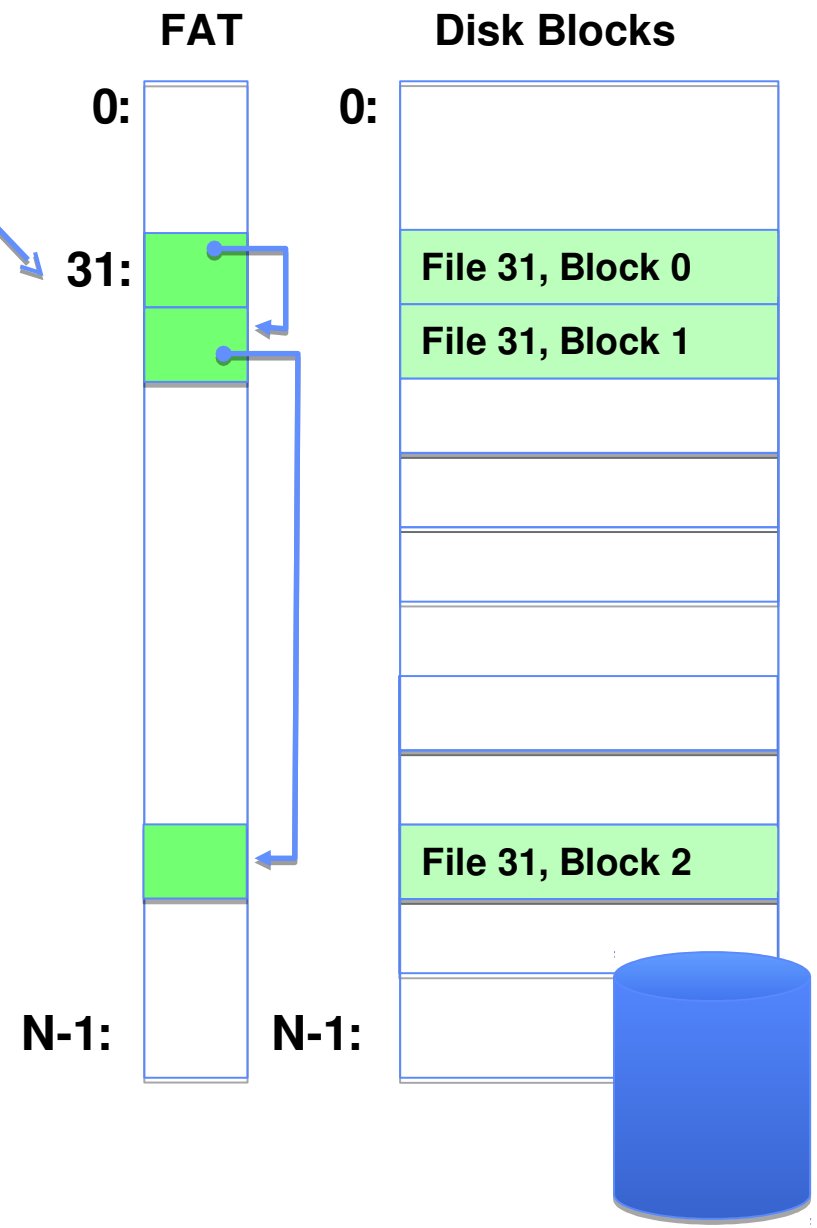


FAT (File Allocation Table)

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

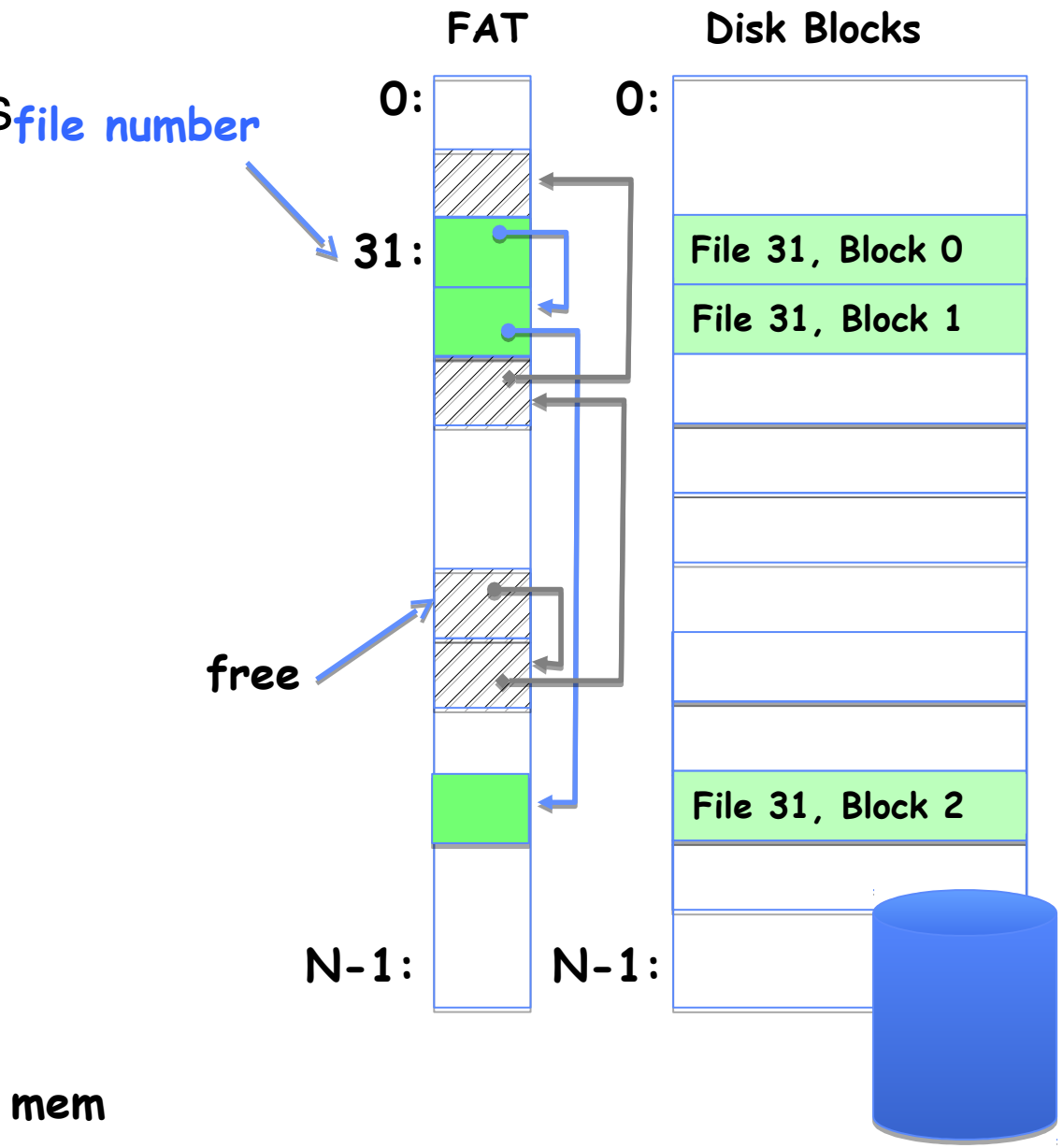
FAT is linked list 1-1 with blocks

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



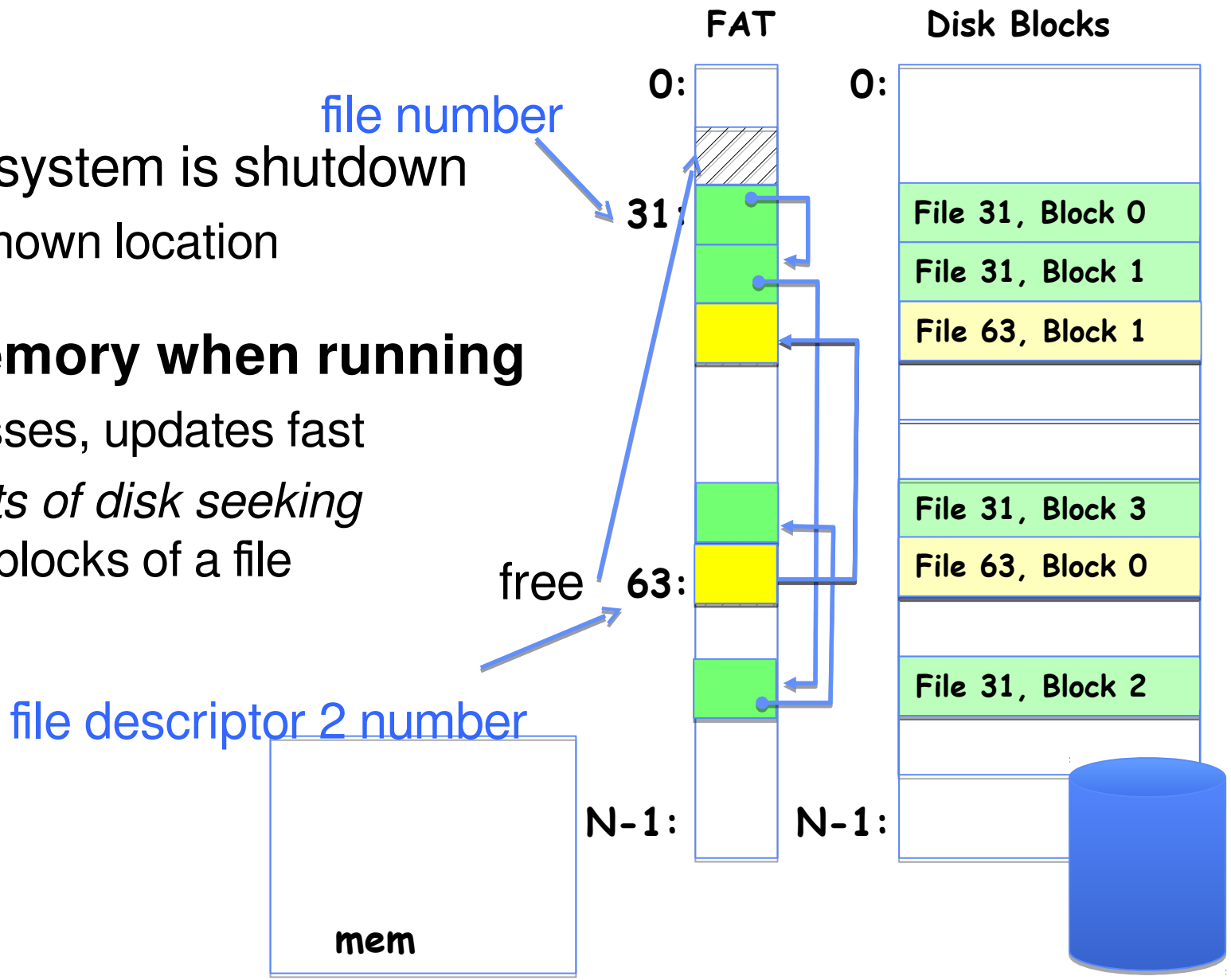
Storing the FAT

On disk when system is shutdown

- Fixed, well-known location

Copied in memory when running

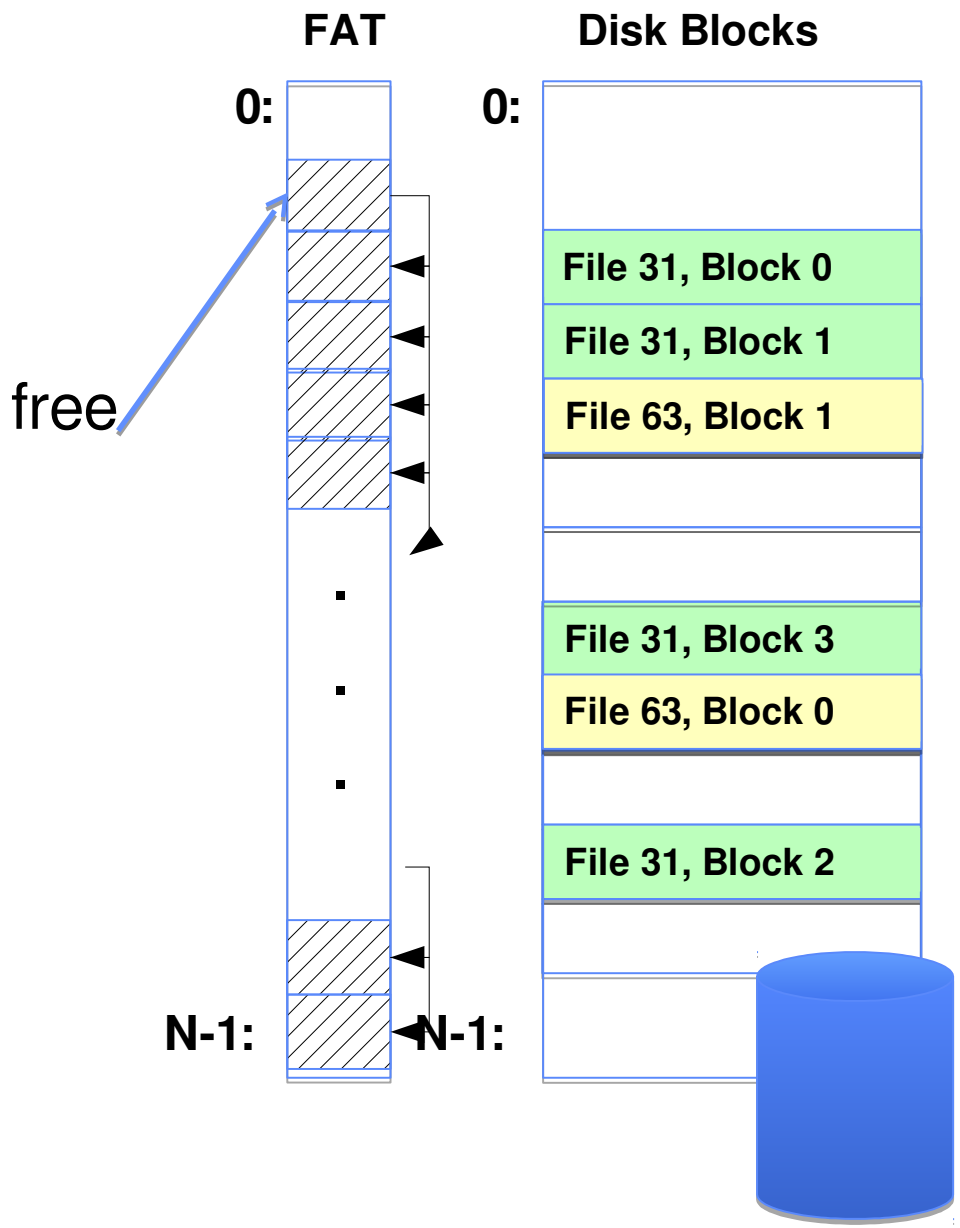
- Makes accesses, updates fast
- Otherwise *lots of disk seeking* to locate the blocks of a file



FAT Setup

Format a new FAT drive?

Link up the free list

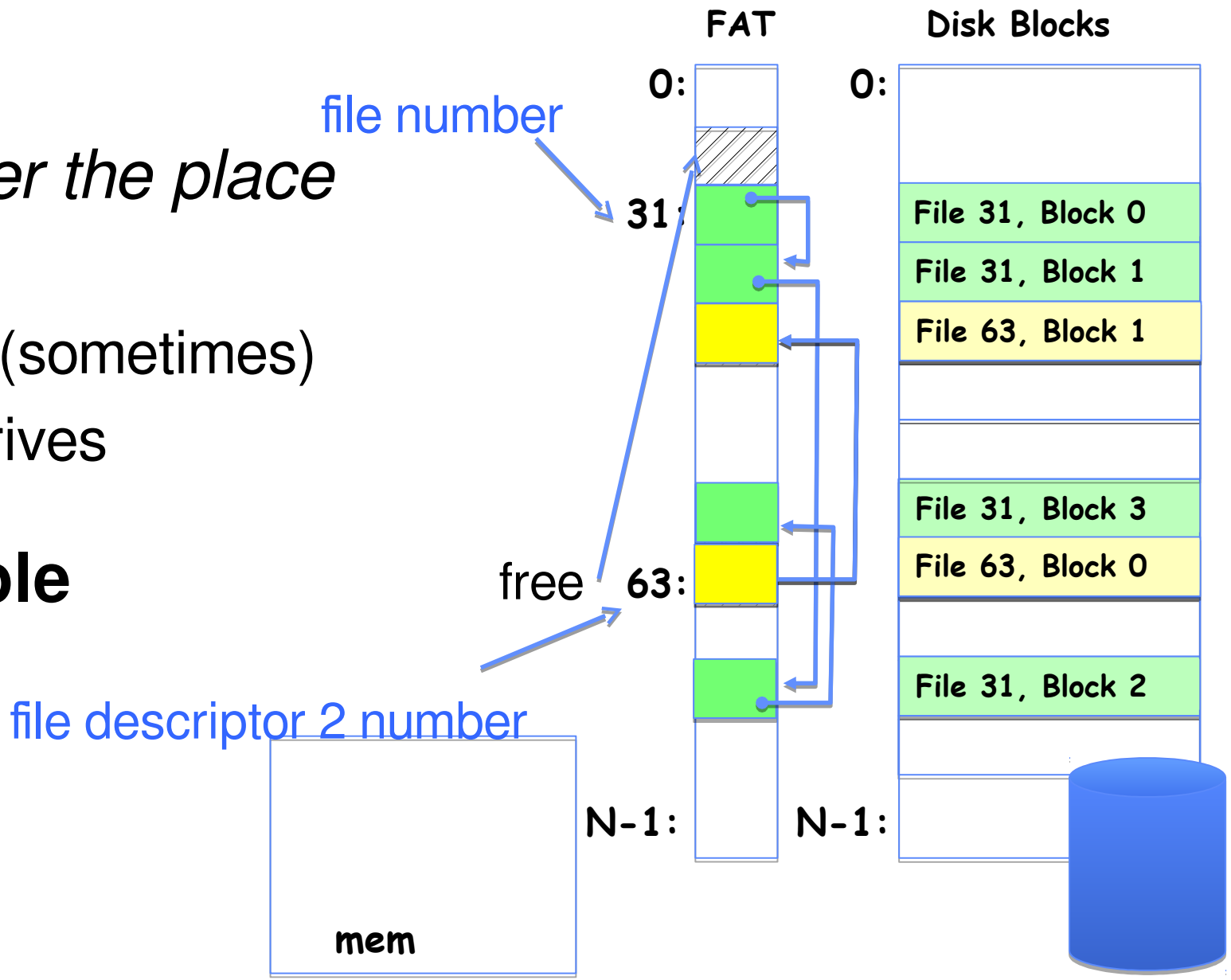


FAT Assessment

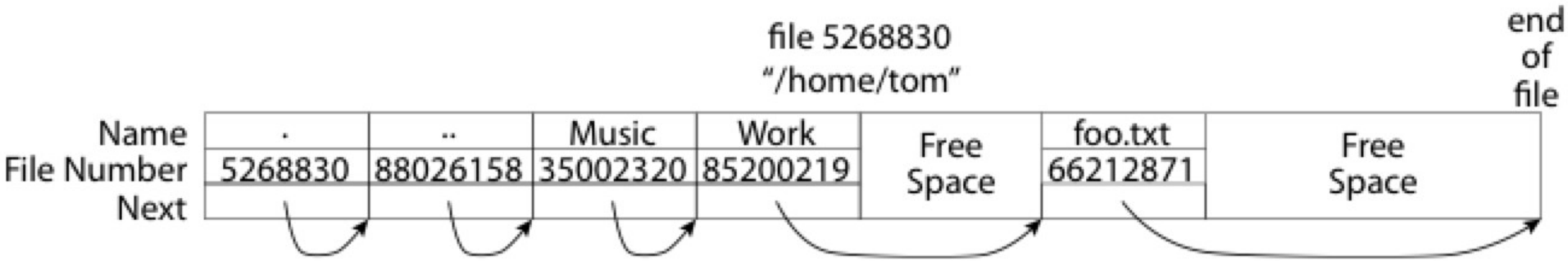
Used *all over the place*

- DOS
- Windows (sometimes)
- Thumb Drives

Really **simple**



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

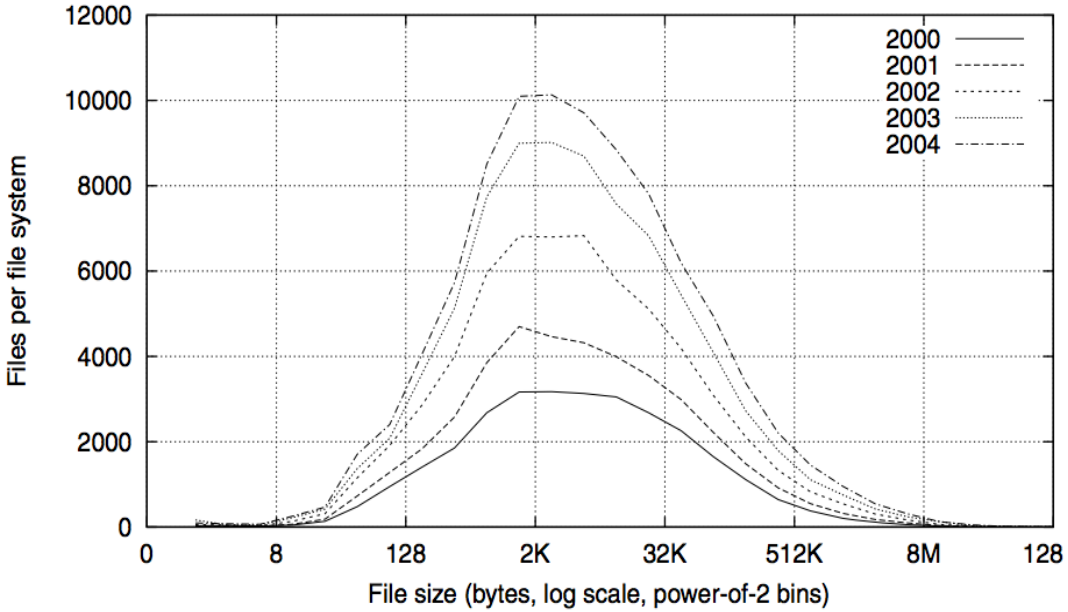


Fig. 2. Histograms of files by size.

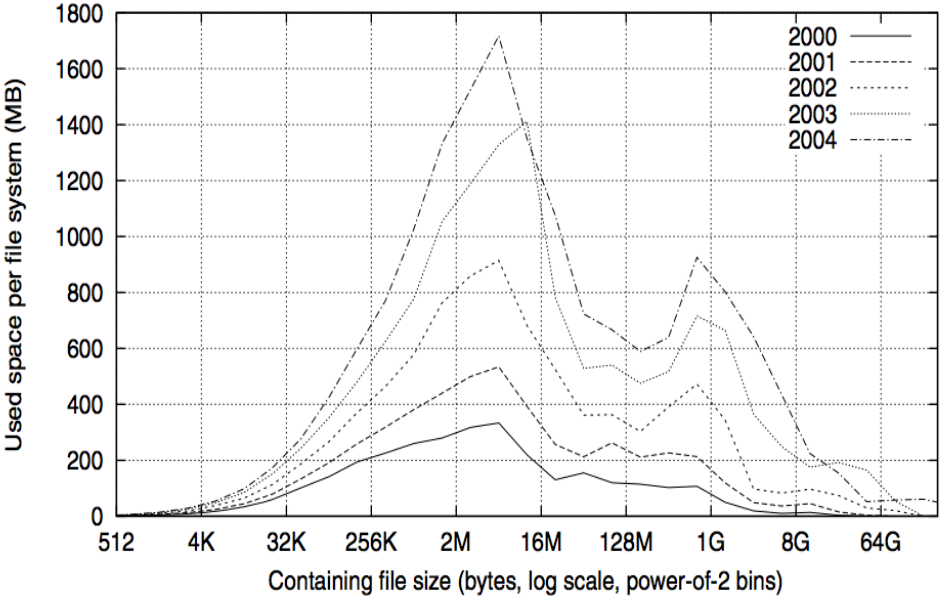
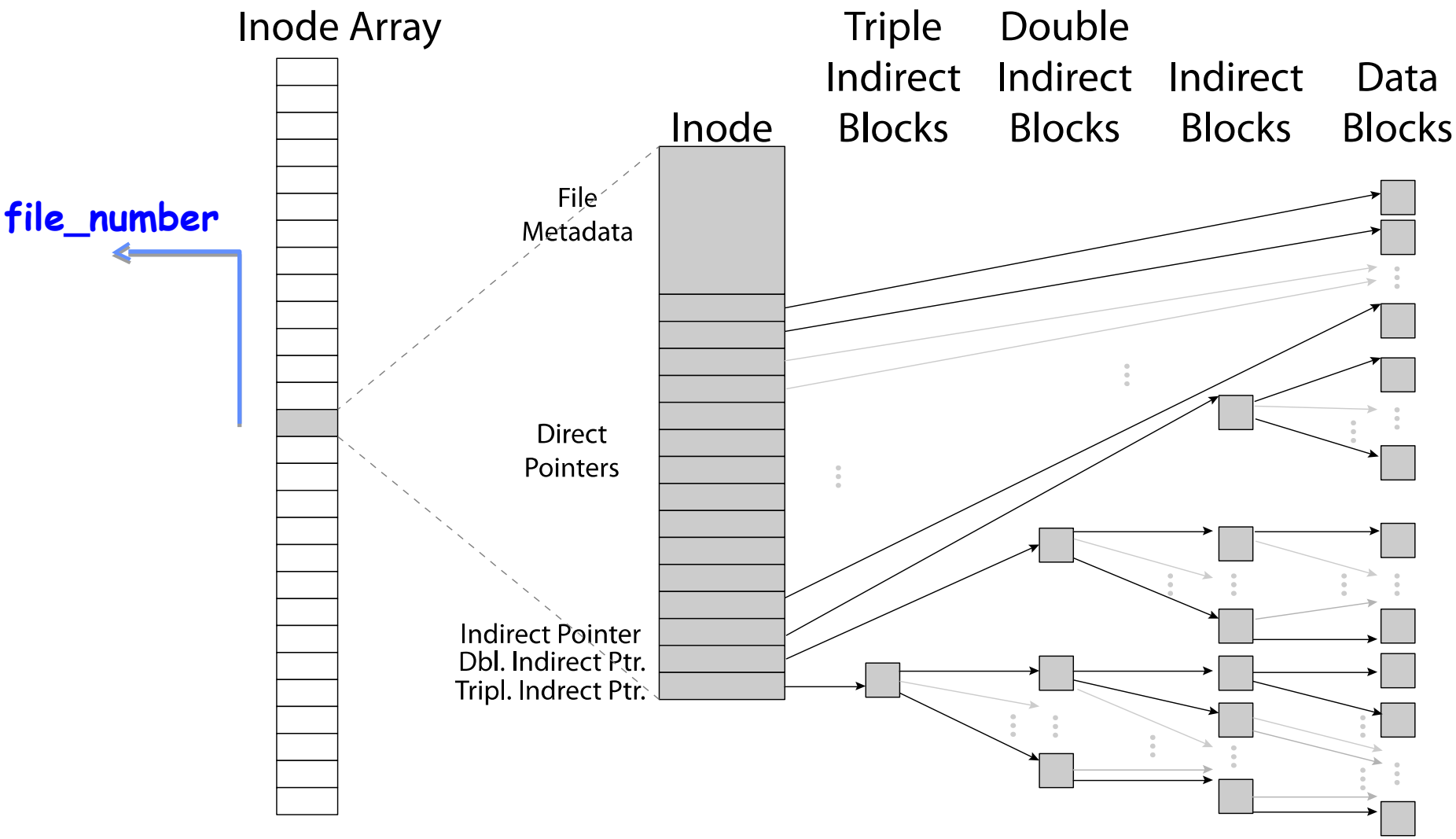


Fig. 4. Histograms of bytes by containing file size.

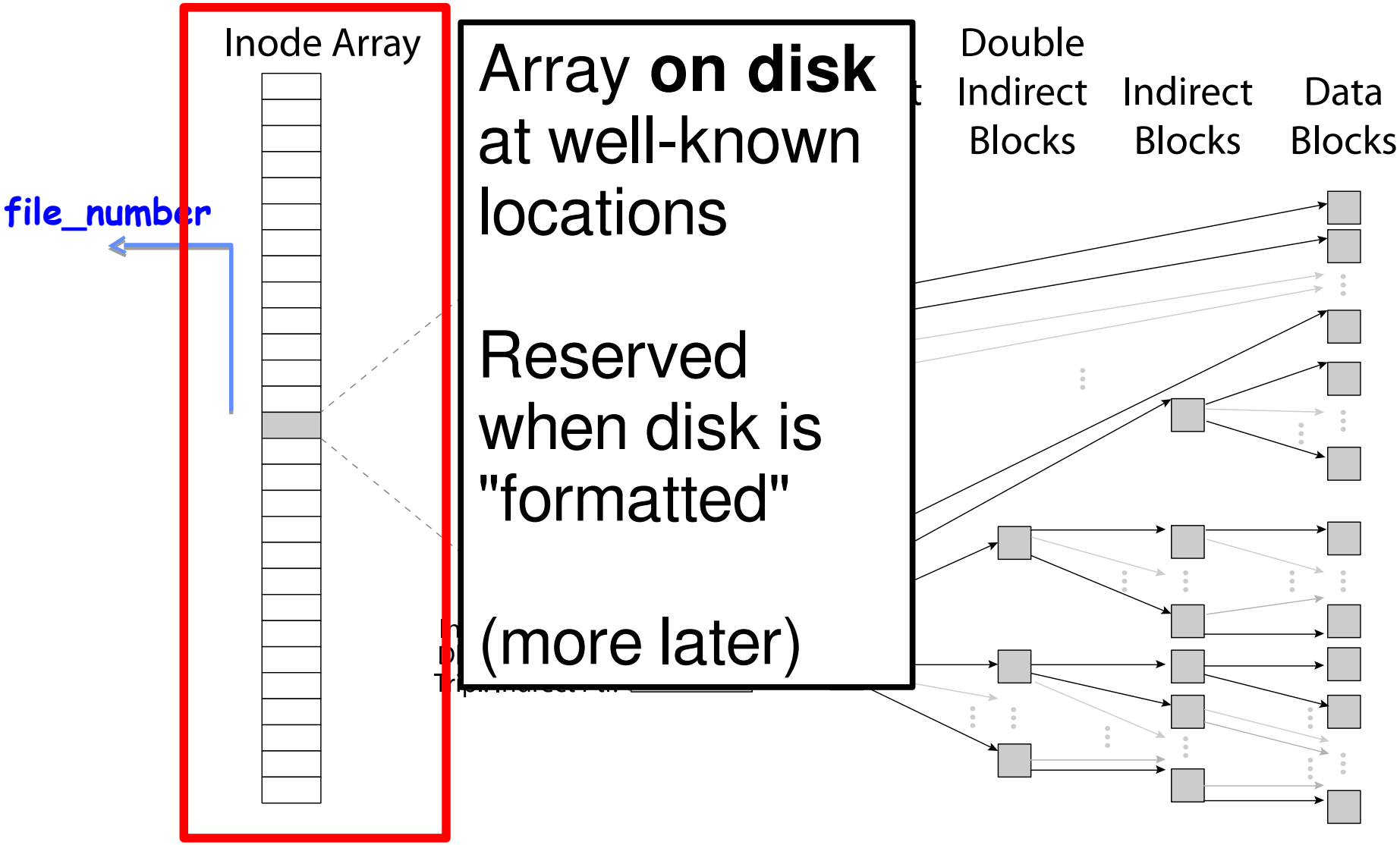
So what about a “real” file system

Meet the inode



So what about a “real” file system

Meet the inode



Array **on disk** at well-known locations

Reserved when disk is "formatted"

(more later)

Unix Fast File System (Optimization on Unix Filesystem)

Original inode format appeared in BSD 4.1

- Berkeley Standard Distribution Unix
- Part of your heritage!

File Number is index into **inode arrays**

Multi-level index structure

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

Metadata associated with the file

- Rather than in the directory that points to it

UNIX FFS: BSD 4.2: Locality Heuristics

- Block group placement
- Reserve space

Scalable directory structure

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

```
/* 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? */
  };
```

irect Data
ocks Blocks

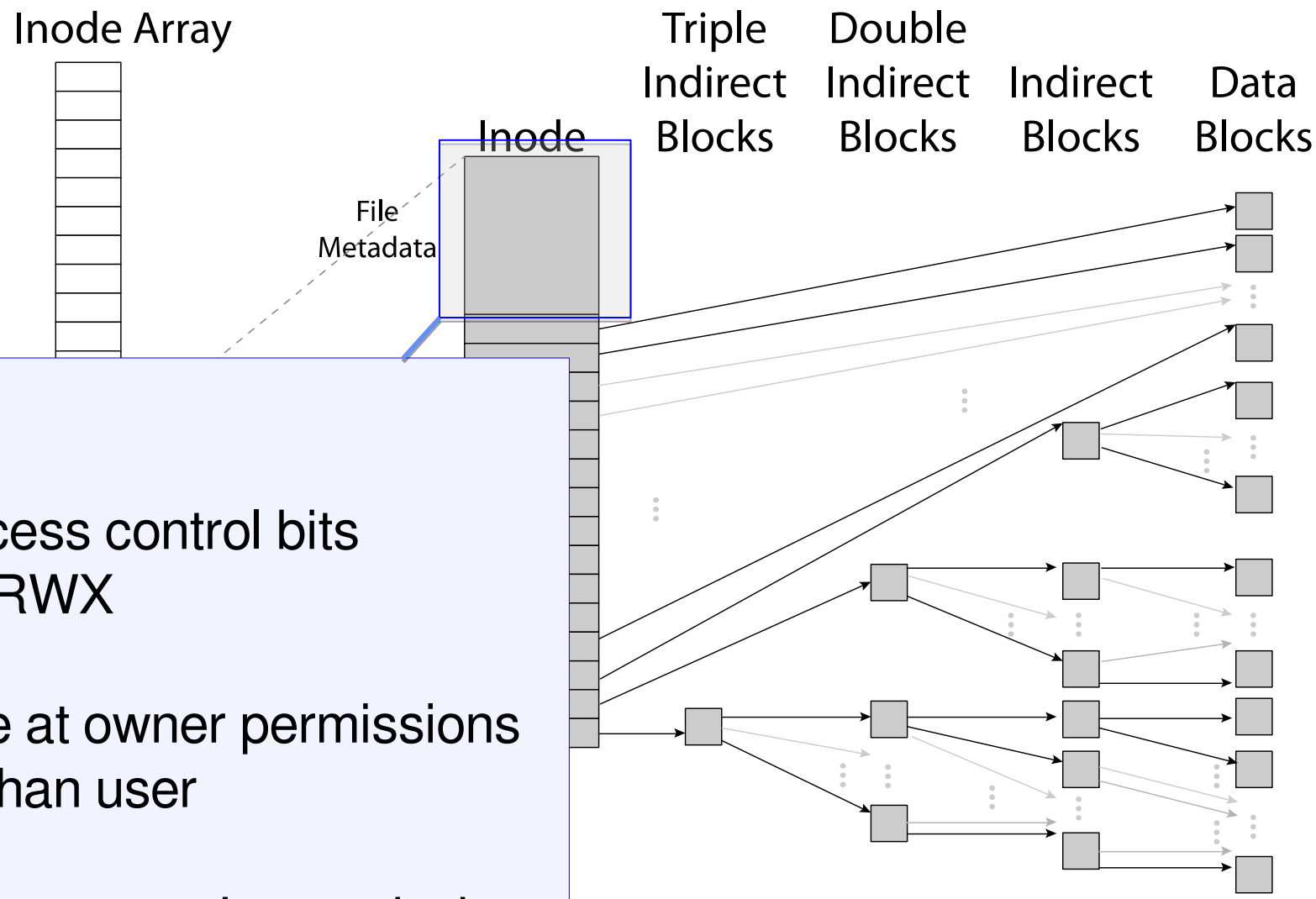
```
/* In-memory inode. */
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True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

Ind
Db
Trip

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

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

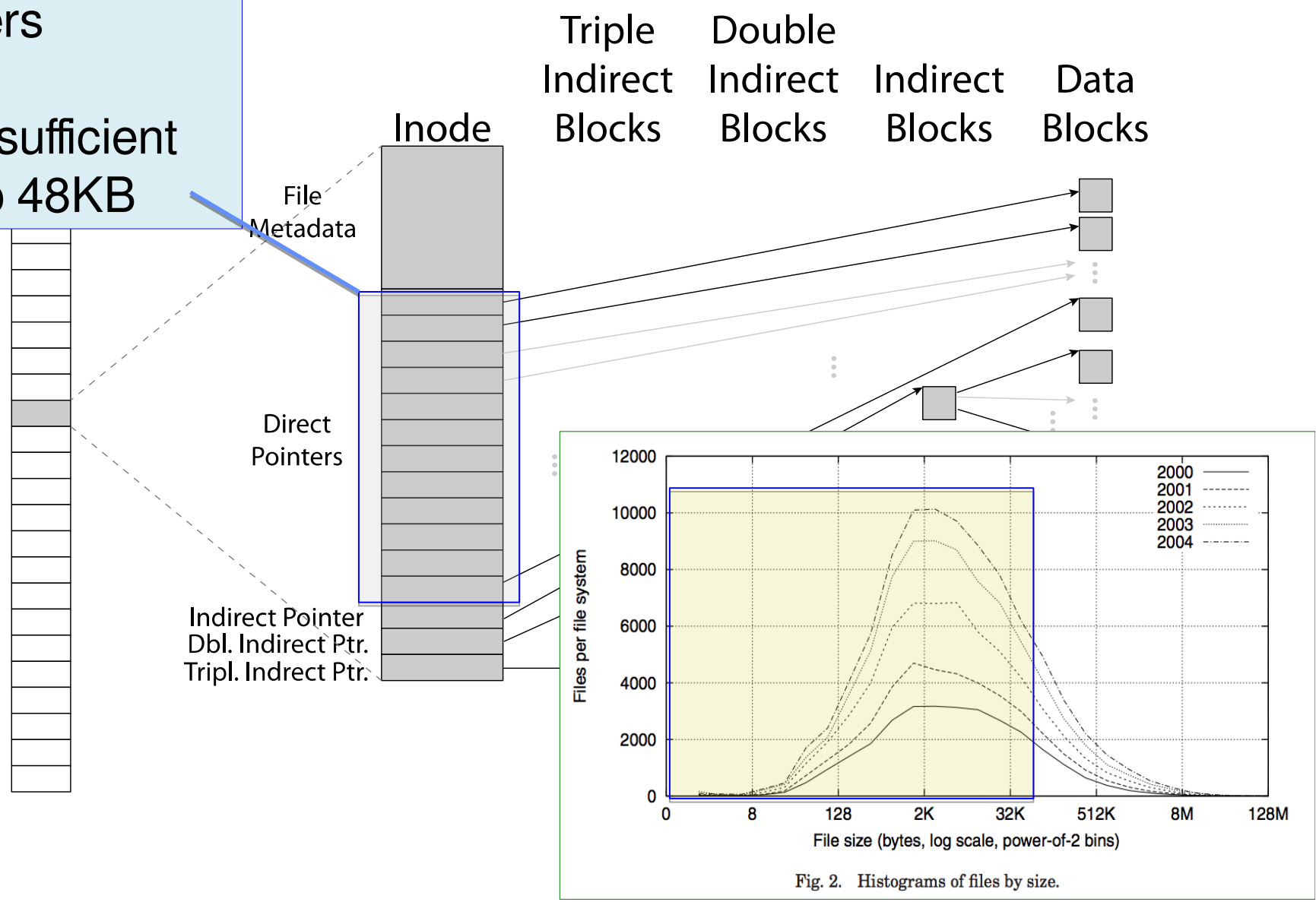


Fig. 2. Histograms of files by size.

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? (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?

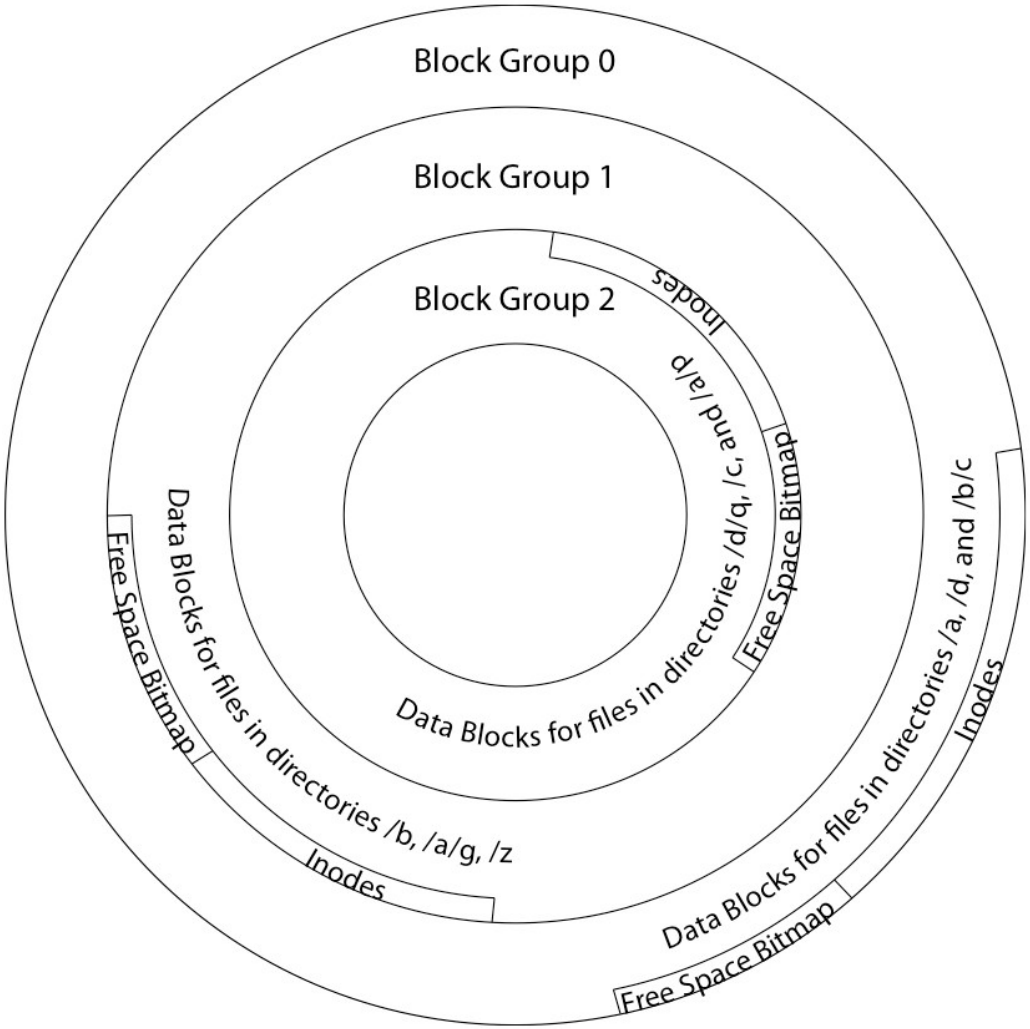
- Read its inode (outermost cylinders)
- Read its data – probably far away
- **Lots of seek time**

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

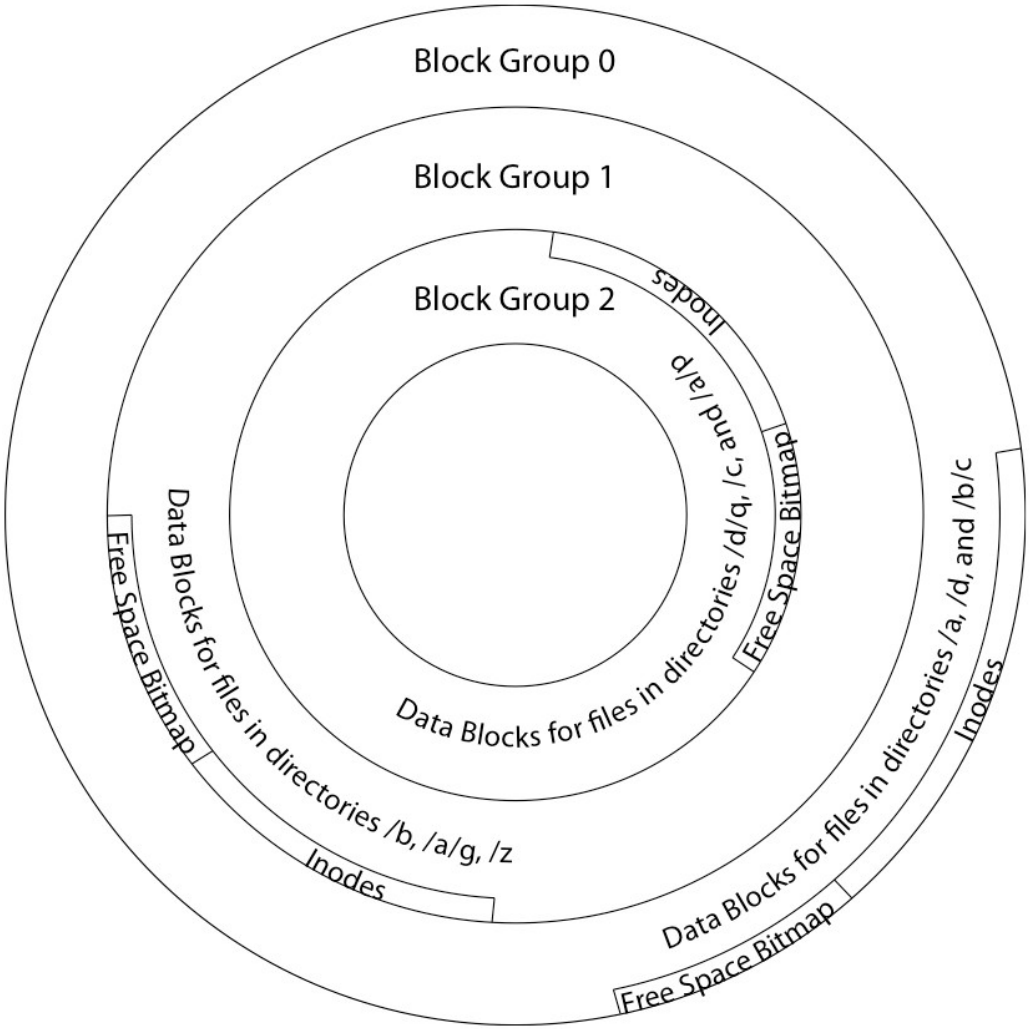


Locality: Block Groups

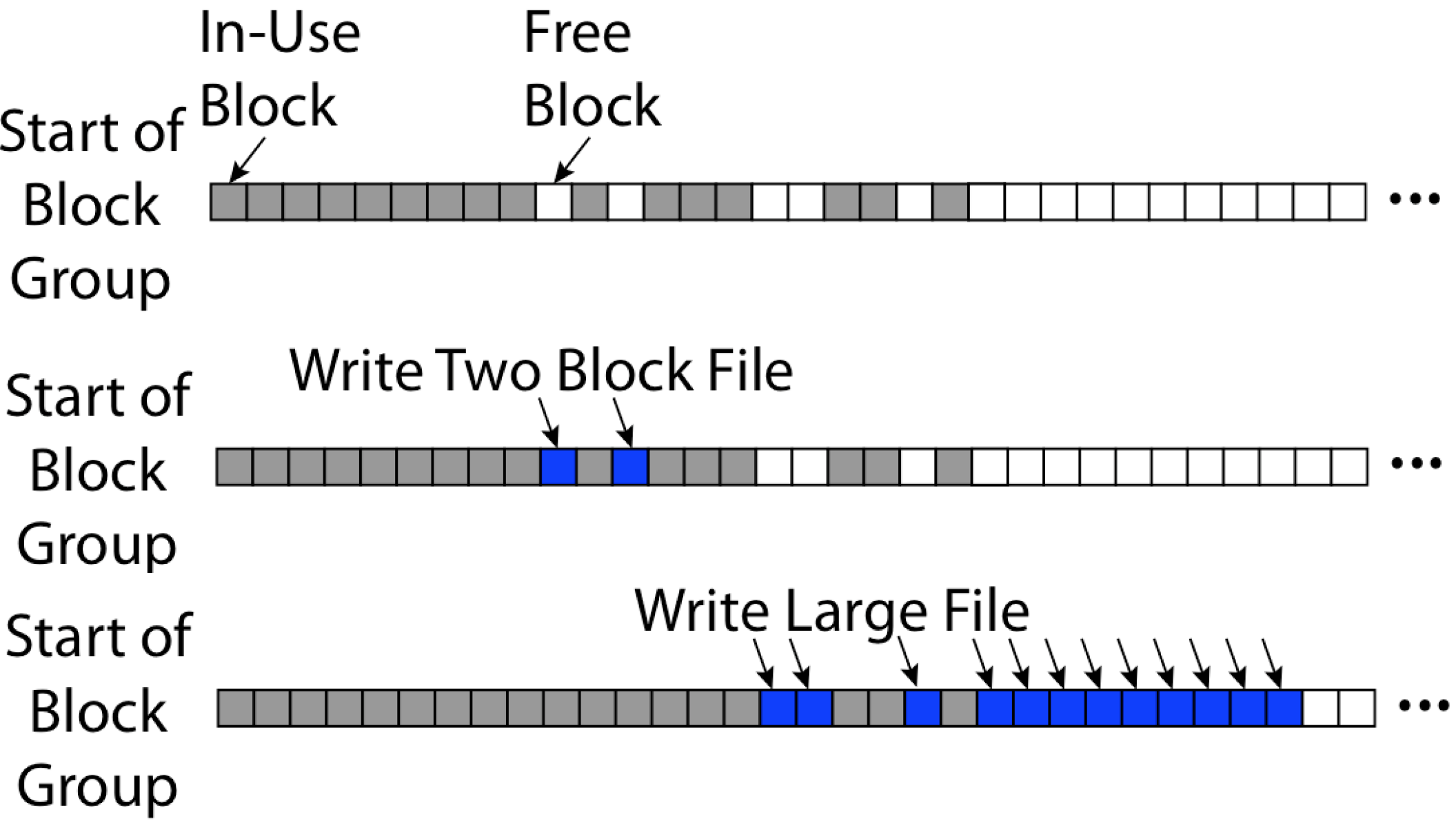
File data blocks, metadata, and free space are interleaved within block group

- No huge seeks

Put directory and its files in common block group



FFS First Fit Block Allocation



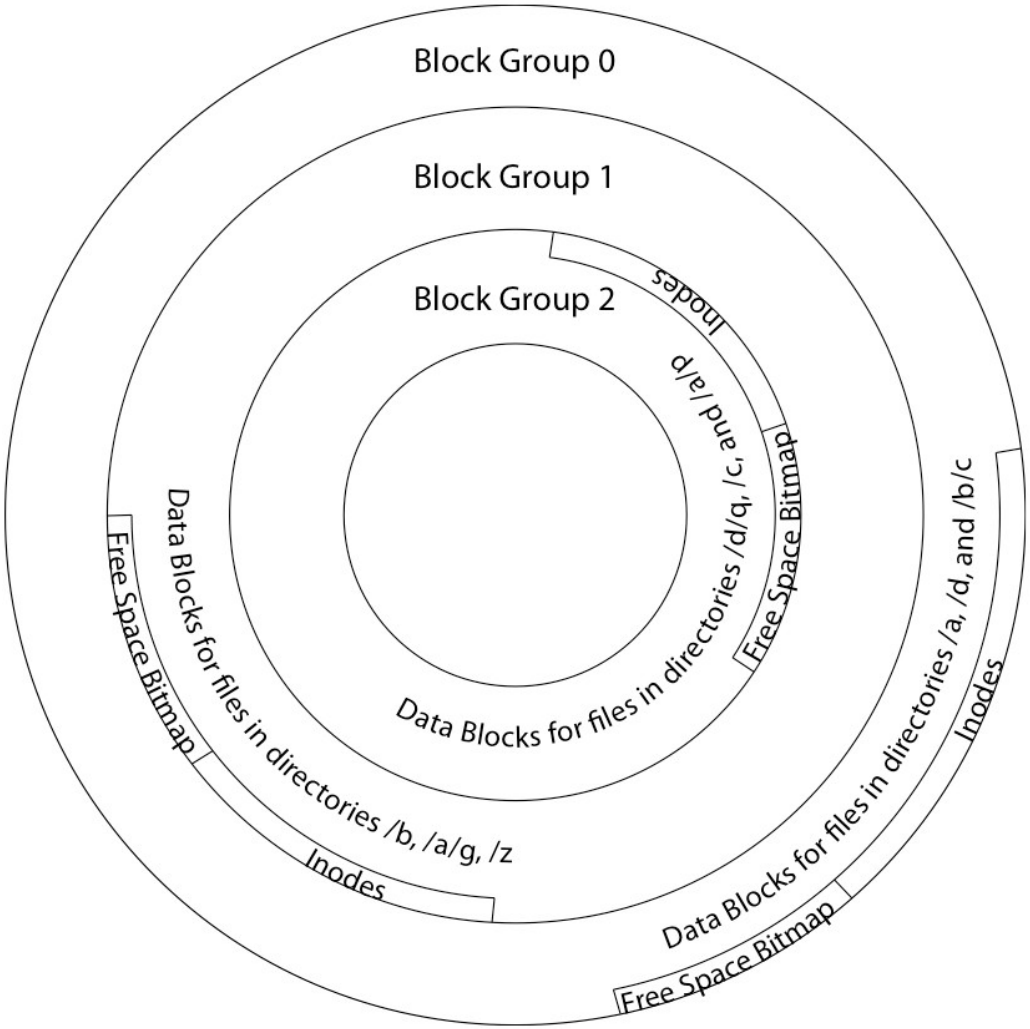
Locality: Block Groups

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

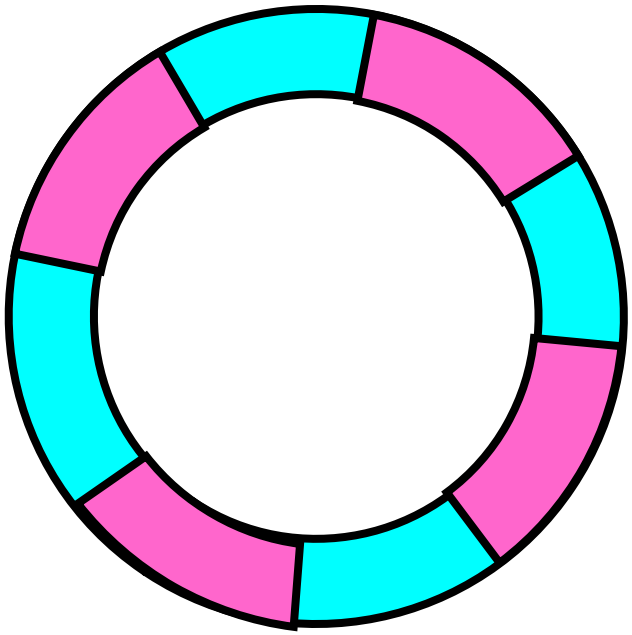
- 10%
- Makes sure there's sequential holes for big files
- Lets "first fit" be fast – likely to find something quickly



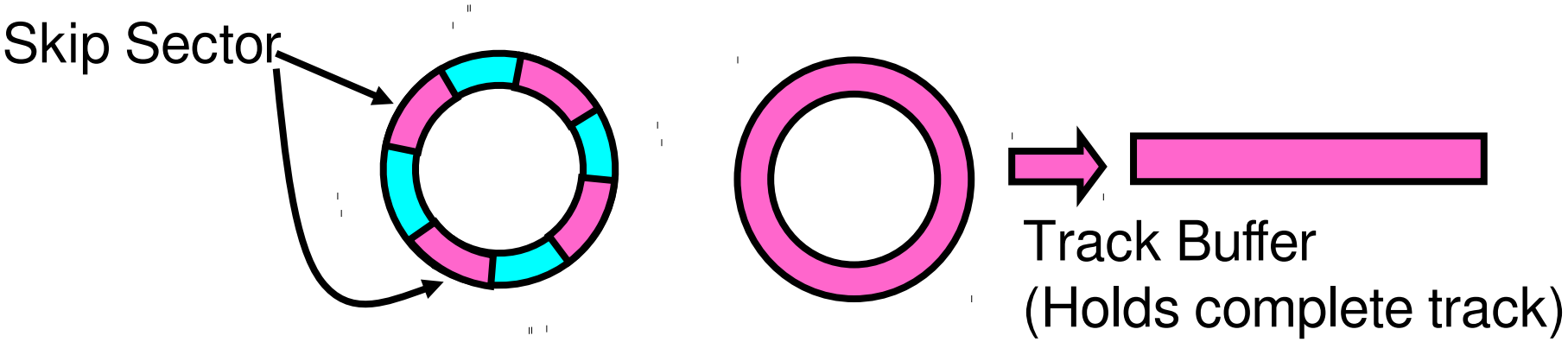
Attack of the Rotational Delay

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**

File System Summary

File System:

- Transforms blocks into Files and Directories
- Optimize for access and usage patterns
- Maximize sequential access, allow efficient random access

File (and directory) defined by header, called “inode”

Multilevel Indexed Scheme

- Inode contains file info, direct pointers to blocks,
- indirect blocks, doubly indirect, etc..

4.2 BSD Multilevel index files

- Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
- Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization

Naming: act of translating from user-visible names to actual system resources

- Directories used for naming for local file systems