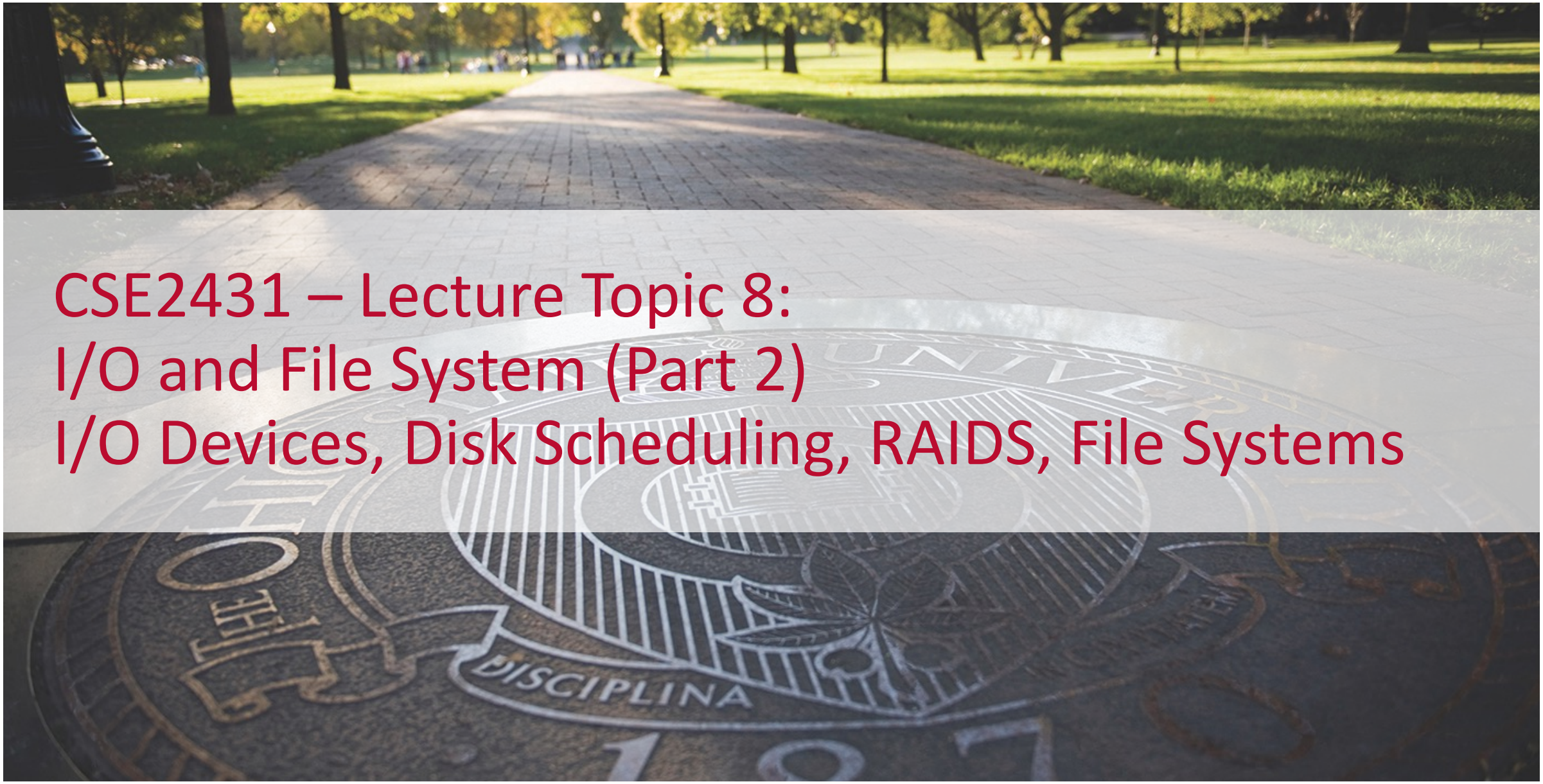


CSE2431 – Lecture Topic 8:
I/O and File System (Part 2)
I/O Devices, Disk Scheduling, RAIDS, File Systems

THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

I/O and File System (Part 2)

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: **Chapter 37, 38, 39 [OSTEP]**

Outline: I/O and File System (part 2)

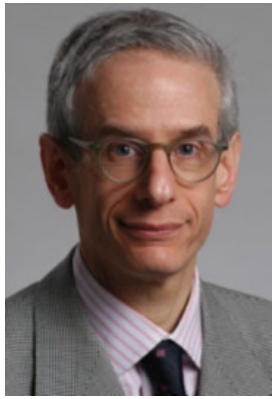
- I/O Systems
- Disk performance
- Disk Scheduling Algorithms
- **RAIDS (Redundant Arrays of Inexpensive Disks)**
  - RAID-0
  - RAID-1
  - RAID-4
- Files and Directories [Chapter 39, OSTEP]
  - Simple File System Implementation [Chapter 40, OSTEP]

Why RAIDs?

- People are greedy
- We want faster, bigger, more reliable, and affordable storage.
- Hard disks have been improved in all dimensions in last few decades
  - Improvement of speed is not significant (compared to CPU speed)
  - Improvement of capacity is quite significant
  - Highly-reliable (enterprise-level) disks are usually more expensive
- However, if even the best disk cannot meet your demand, there is a simple solution --- use an array of disks

Inventors of RAID

David Patterson: professor at UC Berkeley. Also known for his contribution to RISC and NOW. Turing Award 2017.



Randy Katz: professor at UC Berkeley.

Garth Gibson: professor at CMU.

Organization of RAID

- **Externally**, a RAID is just like a **normal disk**.
  - A user can use it like using a normal disk

- **Internally**, it contains **multiple disks** and **a controller**
  - Hardware RAID: the controller is some specific hardware
  - Software RAID: the controller is a special device driver running on OS

Fault Model

- RAID is working under the fail-stop model: a disk is either working correctly or fully failed (not responding to any requests).

- There are other more complex faults:
  - Partial failure: some sectors are not accessible while others are still good.
  - Silent corruption: some bits may be flipped without being detected
  - RAID cannot tolerate these failures. We will discuss them later.

Metrics to evaluate a RAID

- Capacity
- Reliability: how many disk failures can it tolerate
- Performance:
  - Latency: how long does it take to process one I/O?
  - Throughput: how many I/Os can it process per second?

RAID-0: Striping (Simple)

- Divide space into multiple chunks and distribute chunks to disks in round-robin order

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- Chunk size: usually quite large (e.g., 64KB)

Address mapping

- Assume we want to access chunk N (each chunk has s sectors), which disk and which sectors should we access?
- Chunk N is on disk N (mod number_of_disks)
- Sector = N / number_of_disks * s

RAID-0: Evaluation

- Capacity: N * disk size
- Reliability: can tolerate 0 disk failures
- Performance:
  - Latency: for single-chunk or smaller I/Os, latency does not change. For larger ones, latency reduces.
  - Throughput: N times of a single disk because all disks can work in parallel

RAID-1: Mirroring

- Basic idea: store each copy of data on two disks
  - We can tolerate 1 disk failure.

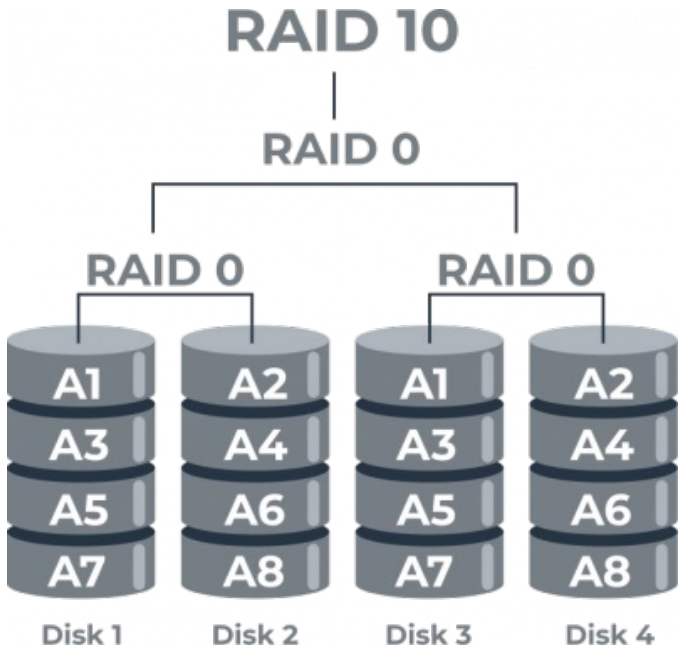
- RAID-1 can be used in combination with RAID-0. RAID-10 or RAID-01

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

RAID-1: Mirroring

- Basic idea: store each copy of data on two disks
  - We can tolerate 1 disk failure.

- RAID-1 can be used in combination with RAID-0. RAID-10 or RAID-01



RAID-1: Evaluation

- Capacity: N * disk_size / 2
- Reliability: can tolerate at least 1 disk failure, and up to N/2 if we are lucky
- Performance: writes go to 2 disks and read can be still done on 1.
  - Latency: read latency does not change. Write latency slightly increases (why?)
    - Reason: latency is random. With two writes, the latency is determined by the slower one.
  - Throughput: write throughput = N/2 * one disk. Random read throughput = N * one disk. Sequential read is the tricky one (you can get N * one disk if you do it correctly).
    - In the previous example, let disk 0 read chunks 0 and 2; disk 1 for 4 and 6; disk 2 for 1 and 3; disk 4 for 5 and 7.

RAID-4: Saving Space with Parity

- Revise XOR: 0 xor 0 = 0; 0 xor 1 = 1; 1 xor 0 = 1; 1 xor 1 = 0
  - If two bits are the same, xor result is 0. If different, result is 1.

- XOR parity: suppose we have a number of bits b_1, b_2, … , b_n, their parity bit is p = $b_1$xor b_2 xor … xor b_n

- XOR parity has a number of useful properties that make it popular in fault tolerance.

Properties of XOR parity

- Property 1: if there are **even** number of "1" bits, parity is 0. If odd, parity is 1. (even parity)
- Property 2: if one bit is **lost**, we can re-construct the bit by **xor-ing** all remaining bits and the parity bit.
- Property 3: if one bit is **flipped**, **parity** bit is also **flipped**.
  - In general, if one bit b is updated (either flipped or remain unchanged), then $p_{new} = b_{old}$ xor b_{new} xor p_{old}

RAID-4: Saving Space with Parity

- If we have a number of data chunks B_1, B_2, … , B_n, we can compute their parity chunk by combining the parity bits of each bit in these chunks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

- Parity can achieve reliability with a relatively low cost (compared to mirroring): if one disk fails, we can reconstruct it (property 2)

RAID-4: Saving Space with Parity

- How to read a chunk?
  - Do the address mapping
  - And then do a normal read

- How to write a chunk? Note we need to update the parity chunk.
  - Solution 1: read all data chunks, compute the parity chunk, and then write
  - Solution 2: use property 3. Read old target chunk C' and old parity chunk P'. The new parity chunk P = C xor C' xor P'.
  - Solution 2 is used in practice. Even with that, a write is turned into two reads and two writes.

RAID-4: Evaluation

- Capacity: (N-1)*disk_size
- Reliability: can tolerate 1 disk failure
- Performance:
  - Latency: read latency does not change. Write latency depends.
  - Throughput
    - Read is easy: (N-1) * one disk.
    - Sequential write: can be optimized to (N-1) * one disk. Random write: ½ * one disk (parity disk is more heavily used than others. This is bad).

RAID-5: Rotating Parity

- Problem of RAID 4: a write will incur two writes to parity chunk. As a result, parity disk is more heavily used than others.
- Basic idea of RAID-5: rotate the parity chunks across different disks

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5: Evaluation

- Capacity and reliability are the same as those of RAID-4
- Performance:
  - Throughput:
    - Sequential read throughput remains at (N-1) * one disk
    - Random read throughput increases to N * one disk.
    - Sequential write throughput remains at (N-1) * one disk.
    - Random write throughput increases to N/4 * one disk. This means we can add more disks to get more throughput, which we cannot do with RAID-4.

RAID Comparison

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

RAID in practice

- RAID-0, RAID-1, and RAID-5 are all widely used.
  - RAID-0 gives best performance and capacity.
  - RAID-1 gives best reliability.
  - RAID-5 provides a good balance.

- RAID-6 is introduced later: it can tolerate 2 disk failures.
- Hot spare: we often reserve a few disks to replace failed disks.
- Keep in mind that RAID can only tolerate fail-stop failure
  - It cannot tolerate silent disk corruptions.
  - It also cannot tolerate human errors, like deleting a file by mistake.

Why not Raid-2 and Raid-3?

- Raid-2 and Raid-3 both exist conceptually, however, they are rarely used in practice due to their inefficiencies, hardware demands and the availability of better alternative (Raid-4, Raid-5)
- Raid-2: **Bit-Level Striping** with **Hamming Code** (stripes data at bit level across drives and uses Hamming Code for error correction). Bit-level Striping itself is difficult to implement and inefficient for most workloads. Also, it requires specialized hardware for synchronization and error correction (Hamming Code)
- Raid-3: Similar to Raid-2, but **Byte-level Striping** with **Dedicated Parity** (Improved by Raid-4 and Raid-5 already with block-level striping (instead of byte-level)

Outline: I/O and File System (part 2)

- I/O Systems
- Disk performance
- Disk Scheduling Algorithms
- RAIDS (Redundant Arrays of Inexpensive Disks)
  - RAID-0
  - RAID-1
  - RAID-4
- **Files and Directories [Chapter 39, OSTEP]**
  - **Simple File System Implementation [Chapter 40, OSTEP]**

Storage abstraction

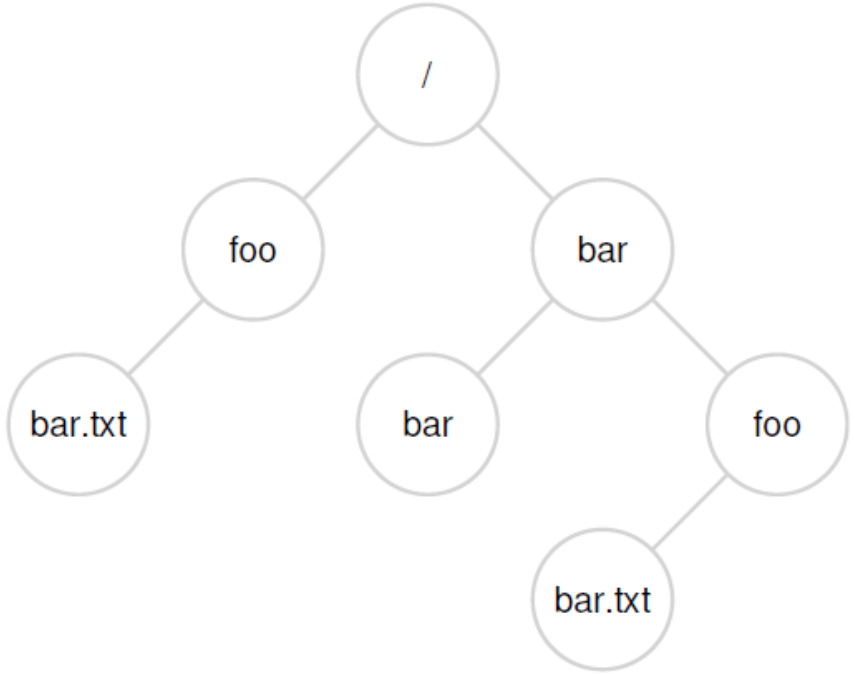
- Disks provide a fixed number of sectors, but this is hard to use
- OS provides files and directories to users
  - File: has a name; stores some bytes; size is variable
  - Directory: has a name; contains multiple files and/or directories
- A file system implements file and directory abstraction over disk abstraction. But first, let us take a brief look of the abstraction

File

- Inode: all metadata information OS maintains for a file
  - E.g. owner, creation time, disk sectors, ...
  - Each file has an **inode**
  - Each inode has a unique inode number
  - Users cannot see inode. Only OS can see it.
  - Did you notice any similarity? (i.e. PCB, TCB)
- Name: a file can have one or more user-readable names
  - OS maps these names to inodes.
- Content: OS views a file as a sequence of raw bytes. It does not interpret the content of files.
  - File can have an extension (e.g. doc, txt, html) to identify its type. But for OS, there is no difference between a doc file and a txt file.

Directory

- Each directory also has a unique inode.
- Each directory contains a map of <name, inode number>
  - Each entry in the map is called a dentry.
- Directory tree: a directory can have subdirectories
  - More accurately, it is a directed acyclic graph (DAG).



Directory

- Root directory: / on linux
- Two special directories: “.” means current dir; “..” means parent dir.
- Absolute path: path from root
  - E.g. /foo/bar.txt
- Current directory: use the “pwd” command
- Relative path: path from the current directory
  - E.g. if current directory is /foo2, then relative path is ../foo/bar.txt

File system interface

- Create (open)
- Read/write
- Seek
- Fsync
- Rename
- Get information
- Remove
- Make directories
- Read directories
- Delete directories
- Hard/Soft links
- Make and mount a file system

Creating Files

- Corresponding system call: open

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- Argument 1: name
  - If you are using relative path, be careful of the current dir
- Argument 2: mode
  - E.g. create/write only/read only/truncate (set size to 0)
- Argument 3: permission. Who can read/write this file?
- Return value: file descriptor (simply an integer)

Reading/Writing Files

- Corresponding system calls: read; write
- Argument 1: file descriptor
- Argument 2: data buffer
- Argument 3: number of bytes to read/write
- Return value:
  - Negative: failure
  - Positive: the number of bytes actually read/write (it may not be the same as argument 3).
  - 0: for read, it means read has reached end of file.

Reading/Writing Files

- Let's look at a real example "cat foo"
  - We use the strace tool to trace all its system calls.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

File descriptor 0 is for standard input (keyboard); 1 is for standard output (screen); 2 is for error output (screen), so the first file you create gets ID 3.

Reading/Writing Files

- Let's look at a real example "cat foo"
  - We use the strace tool to trace all its system calls.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

We often don't know how much data to read. So we create a buffer and try. In this example, the program creates a 4096-byte buffer and tries to read. It gets 6 bytes.

Reading/Writing Files

- Let's look at a real example "cat foo"
  - We use the strace tool to trace all its system calls.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

Write to standard output. You can see all bytes have been written successfully.

Reading/Writing Files

- Let's look at a real example "cat foo"
  - We use the strace tool to trace all its system calls.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

Because we don't know how much data to read, we can read until the return value is 0.

Reading/Writing Files

- Let's look at a real example "cat foo"
  - We use the strace tool to trace all its system calls.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

Remember to close the file when you are done. Otherwise, you may run out of file descriptors.

Reading/Writing Files

- File size can be big, so you cannot expect to create a memory buffer to hold the whole file.

- The following pattern is common:

```
fd = open(…)
byte[] buffer = malloc(size)
while(read(fd, buffer, size)!=0){
     handle buffer;
}
```

fsync

- Fact: when write returns, data might be in OS buffer. It may not be written to disk yet.
  - Why? Buffering allows OS to better schedule writes (remember Elevator's algorithm?).
- Sometimes you hope to make sure data is on disk (e.g. when you save a document).
- Corresponding system call: fsync
- Note: fsync can be slow. If you do (write,fsync,write,fsync, …), then its performance is close to random writes, even if you write to consecutive sectors.

Renaming Files

- Linux command: mv (e.g. mv foo bar)
- Corresponding system call: rename
  - rename(char *old, char *new)

- Note: OS guarantees that rename is atomic
  - If there is a system crash during a rename, the file is either named to the old name or the new name.
  - This is very useful if your program needs atomic update to a file.
  - Example: when saving a file, the application usually doesn't write to the original file directly. Instead, it writes to a new file and renames it.

Removing Files

- Linux command: “rm”
- Corresponding system call: unlink()
  - Hmmm. This name is weird. We will discuss why later.

- Note: Linux does not provide any mechanism to restore a removed file (no Trash Can or Recycle Bin like on Mac or Windows).
- So think twice before you type rm, then look three times before you press “Enter”

Hard Links vs. Soft Links

- Hard link: Linux allows a physical file to have multiple path names.
  - Why? For convenience.
  - Are you bored of typing “cd hadoop/src/hdfs/server/datanode” every time when you logged in? You can create another name “dn” for it, then you only need to type “cd dn”.

- Usage: when you create a file, it will have one path name. Then you can link it to a new path name.
- Linux command: ln
- Corresponding system call: link

Hard Links example

- Example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

- In principle, multiple dentries can map to the same inode

Hard Links fact

- All files point to the same inode
  - Their contents are the same
  - Modifying one will affect the other ones

- If you remove (unlink) one of them, the others are still there
  - The file is actually removed when all are unlinked. Now you see why “rm” calls unlink.
  - OS maintains a counter for each inode to record the number of its hard links. If the counter drops to 0, the OS may remove the inode.

Soft/Symbolic Links

- Limitation of hard link: cannot link directories; cannot link to other disk partitions.
- Soft links allow you to do those things (but with its own limitation)
- Linux command: ln **–s**

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

Soft Links fact

- A soft link is a different file: it has a different inode.
- Modifying one will affect the others. (This is the same as hard links)
- Deleting the link file does not affect the original file.
- Deleting the original file will actually delete the file.
  - The link becomes a dangling reference.
  - Soft links do not increase the counter in inode.

Making and Mounting a File System

- After installing a new disk, you need some preparation before using it.

- Making a file system (called Format on Windows) [mkfs]
  - You need to specify which file system you want to use with this disk
  - The file system may need to write some information to the disk

- Mounting: you need to put the new disk into the existing directory tree by linking it to one existing directory (mount point). [mount]

Simple File System Implementation

- Disk: a fixed number of fixed-sized sectors

- OS: a variable number of variable-sized files
  - Organized in directories
  - Users can create/read/write/delete/… them

- A file system implements the file interface upon the disk interface
  - This looks a bit like memory management, right?
  - Yes. File system and memory management do share similar goals and solutions, but there are also significant differences.

Review of Memory management

- Provide a virtual address space for each process (0 to MAX)
- Divide memory space into pages and frames
- Page table of each process: map a virtual page to a physical frame
- Free space management

Overview of file system implementation

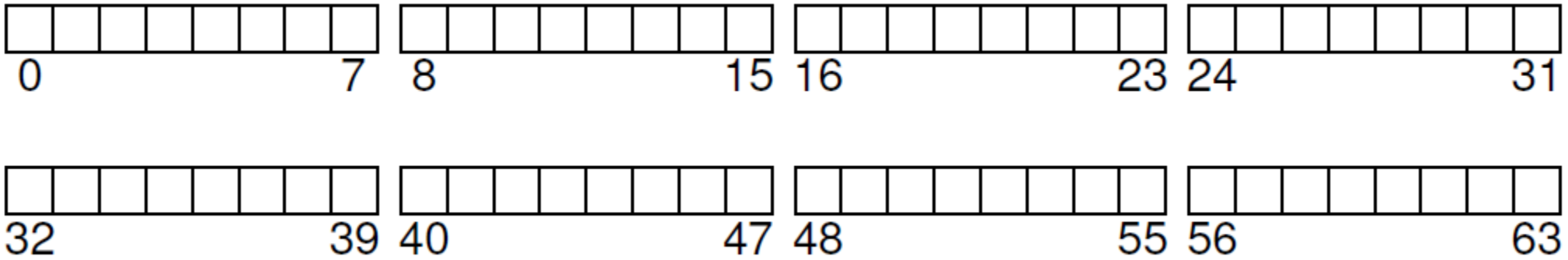
- Provide a virtual address space for each ~~process~~ file (0 to MAX)
- Divide memory disk space into ~~pages and frames~~ blocks
  - A block is usually 4KB, same as page size
- ~~Page table~~ Inode of ~~each process~~ file: map a ~~virtual page~~ block to a physical ~~frame~~ block
- Free space management
- Directory management
- Support for restart/crash recovery
- Optimization for disk characteristics (sequential faster than random)

Popular file systems in practice

- **Unix/Linux:** ext series (ext2, ext3, ext4), ZFS by SUN, ….
- **DOS/Windows:** FAT, FAT32, NTFS, …
- **Mac:** HFS, APFS, …
- **Networked File System:** NFS, AFS, Google File System, …

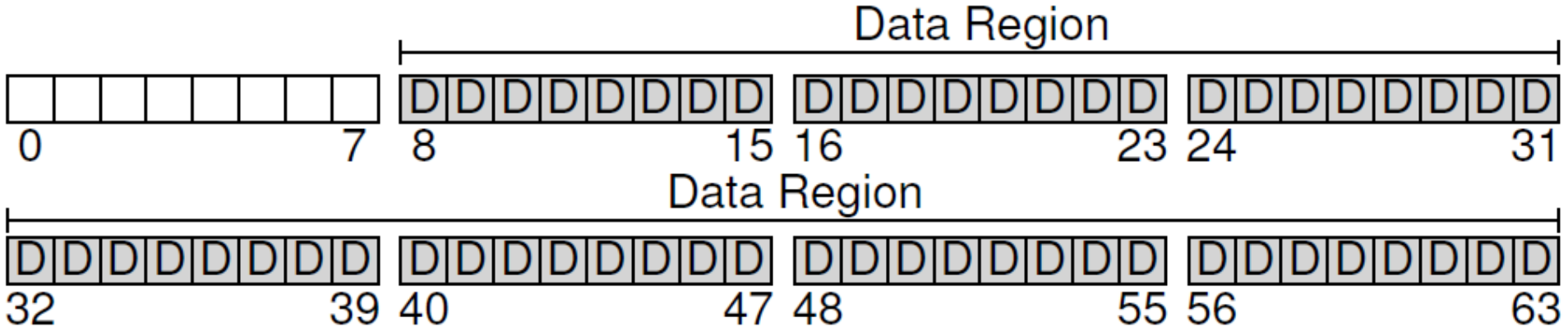
Let’s start from the simplest one

- Divide disk into multiple blocks



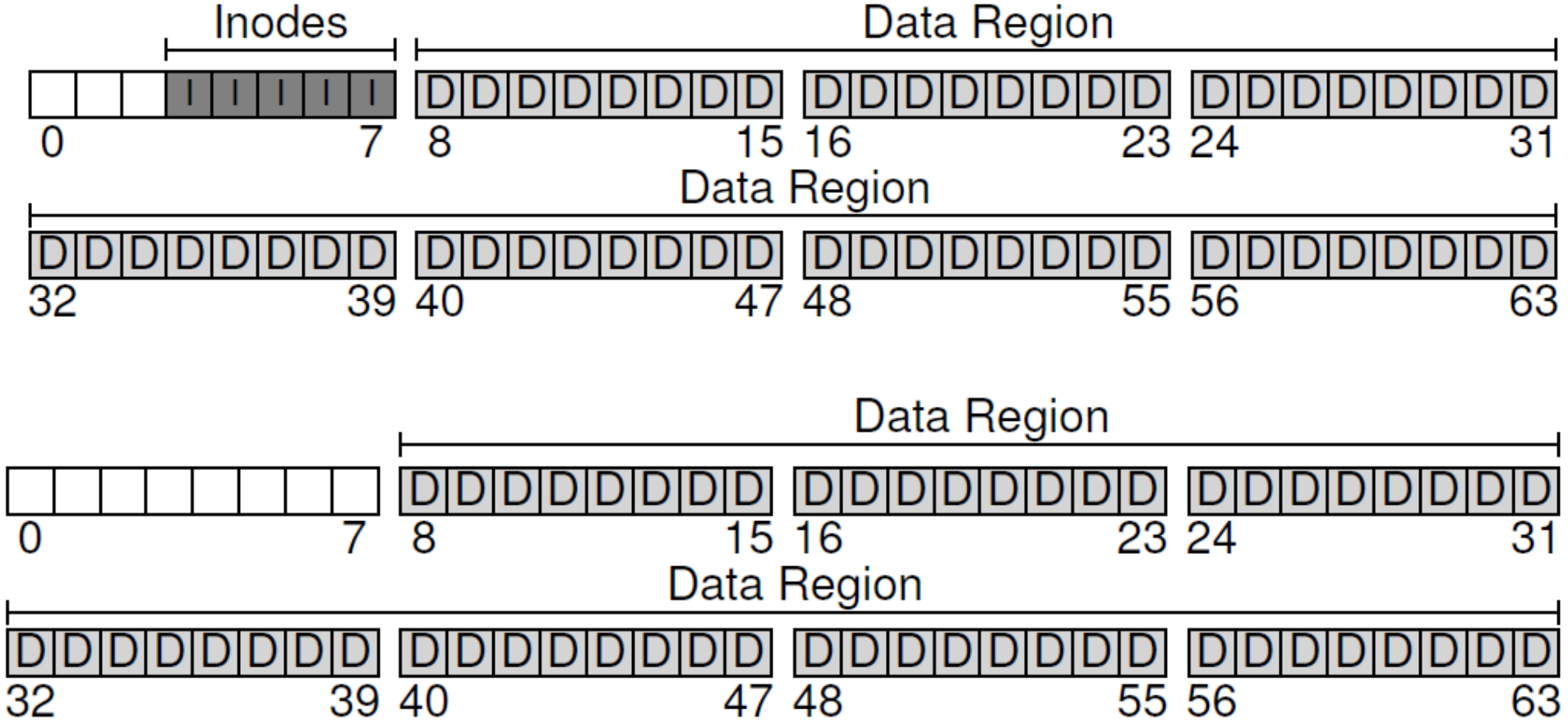
Static partition between data and metadata

- Usually we call user's data "data"
  - File contents are data
- Additional information maintained by a file system is "metadata"
  - Inode, free space management info, …
- First, let's assume we statically assign some disk blocks for data



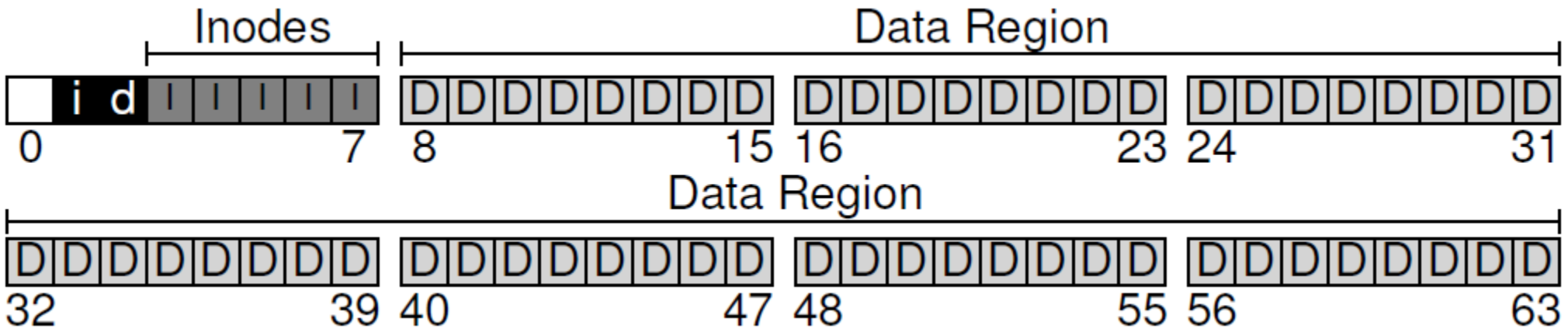
Reserve space for inodes

- Inode records metadata of a file/directory



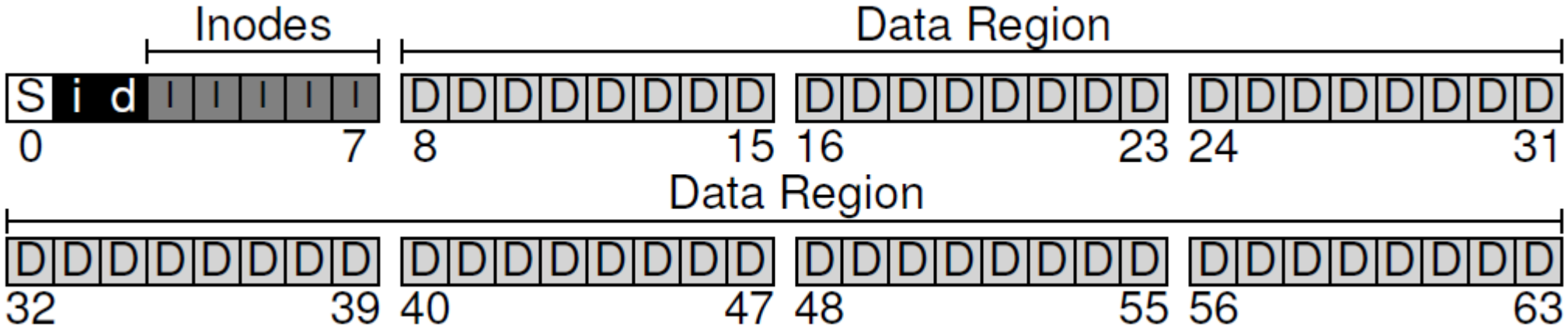
Free space management

- We need to record which blocks are free/used
- A simple solution is a bitmap (free list is an alternative)
  - Bitmap is an array of bits: bits[i] = true/false means entity i is used/free.
  - One for data region and one for inodes



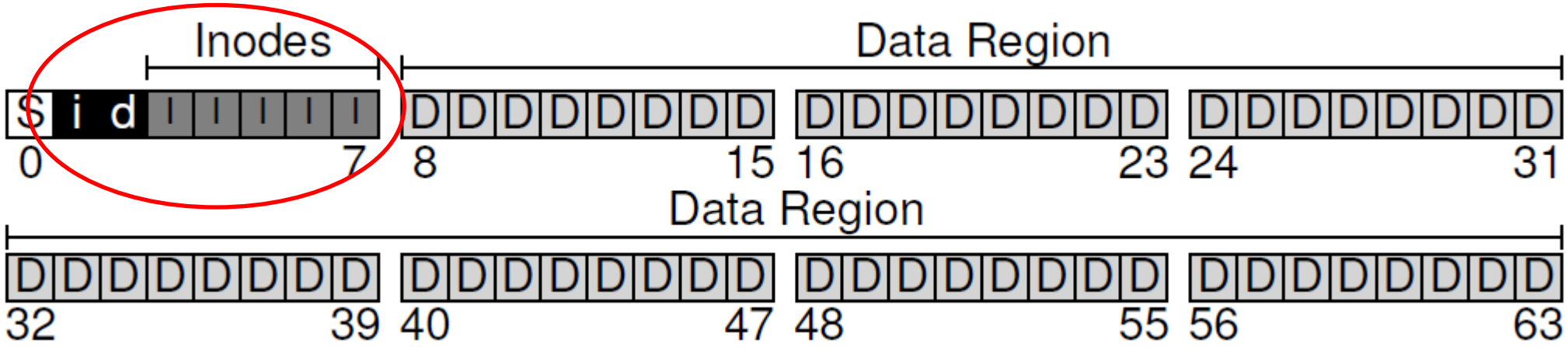
Superblock

- Superblock contains information about this particular file system
  - E.g. file system type, number of inodes and data blocks, location of inode tables, …



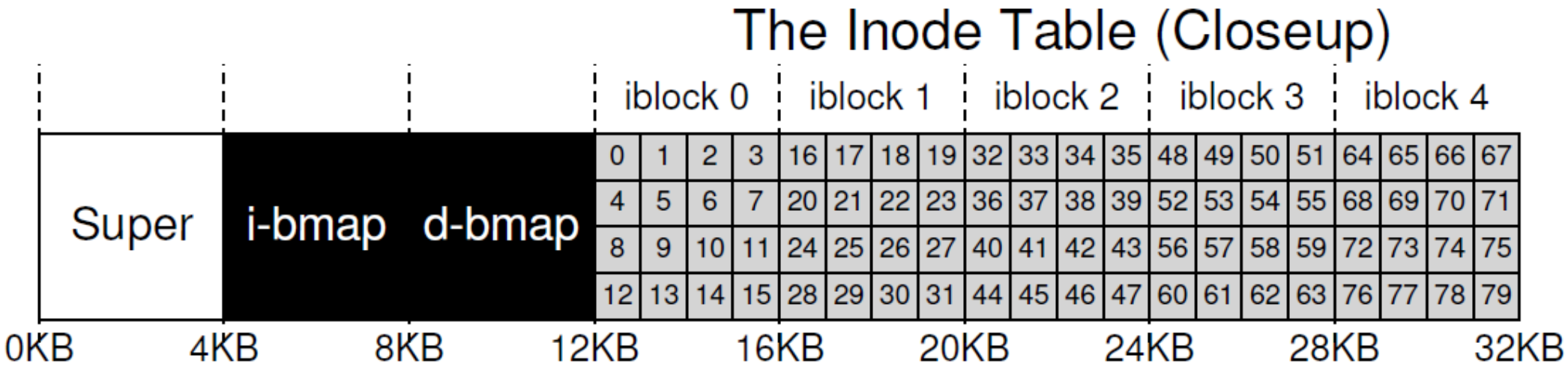
iNode

- Let’s investigagte inodes in more details



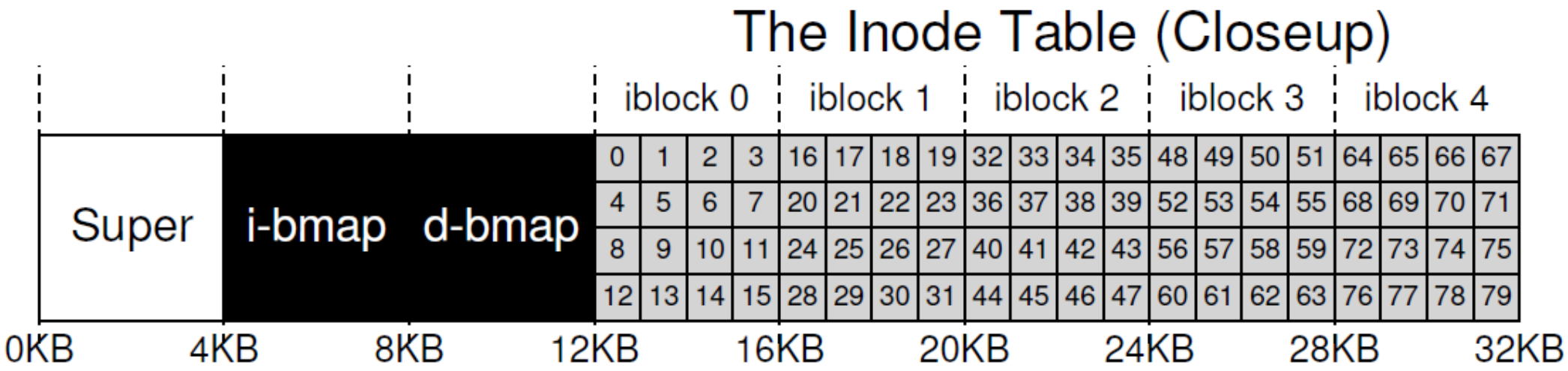
iNode

- Each inode has an ID
  - OS should be able to find an inode based on its ID
- Suppose each inode is 256 bytes, inode table starts at 12KB and has 5 blocks, how to find an inode based on its ID?



iNode

- The address of inode i = 12KB + i*256 bytes
- Remember a disk's basic I/O unit is a sector
  - So read sector (12KB + i*256 bytes)/sector size



Information in the inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: **Simplified Ext2 Inode**

Used for address mapping

iNode: Mapping V.A. to P.A.

- Similarly as paging, file system needs to map VA to PA

- Review of virtual memory and paging:
  - Each process has separate virtual spaces from 0 to MAX
  - Divide virtual space and physical space into pages/frames
  - Maintain a page table for each process to convert page number into frame number
  - Procedure: given an VA, compute page number and offset, map page number into frame number, and concatenate frame number with offset

iNode: Mapping V.A. to P.A.

- File system uses a mechanism similar to paging
  - Each file has separate virtual spaces from 0 to MAX
  - Divide virtual space and physical space into virtual/physical blocks
  - Each file's inode stores information to convert virtual block number into physical block number
  - Procedure: given a VA, compute virtual block number and offset, map virtual block number to physical block number, and concatenate

- Inode stores an array of block pointers for such mapping (these pointers are called direct pointers)
  - blocks[i] stores the physical block number of virtual block i.
  - This is essentially the same as single-array page table.

iNode: Mapping V.A. to P.A.

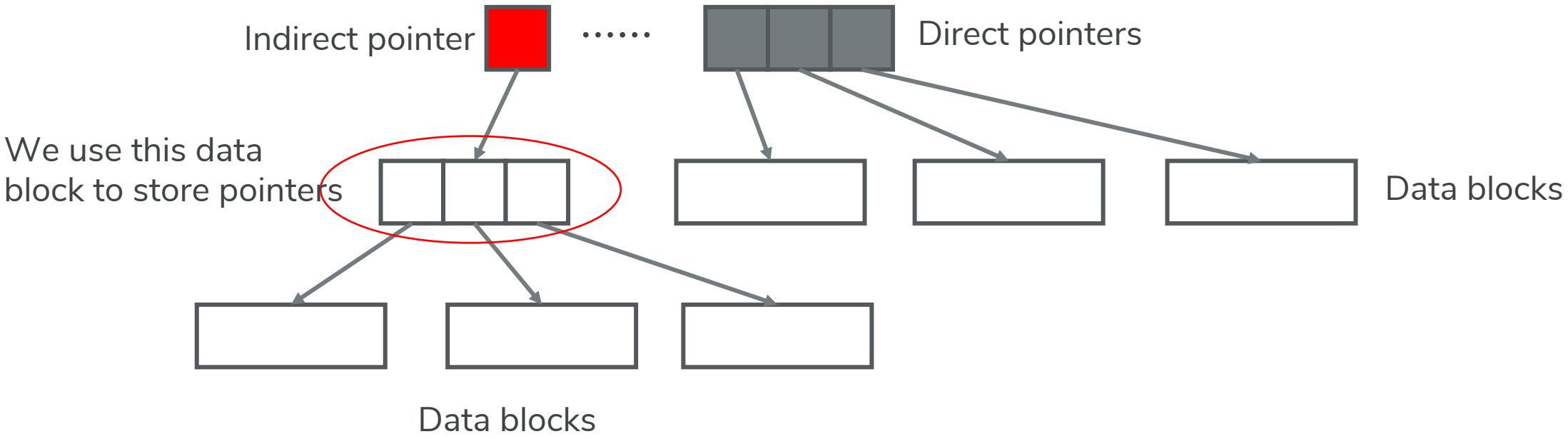
- What is the max size of a file?

- Since the size of inode is limited, we cannot have many such pointers (in the previous example, 15 in total)
  - Max file size = 15 * 4KB = 60KB (too small)

- Can you think of a solution? (Hint: we have learned this in memory management)

iNode: Mapping V.A. to P.A.

- Two-level index: we let one or a few pointers point to a data block
  - The data block stores more pointers to real data blocks
  - Such pointers in inodes are called "indirect pointers"



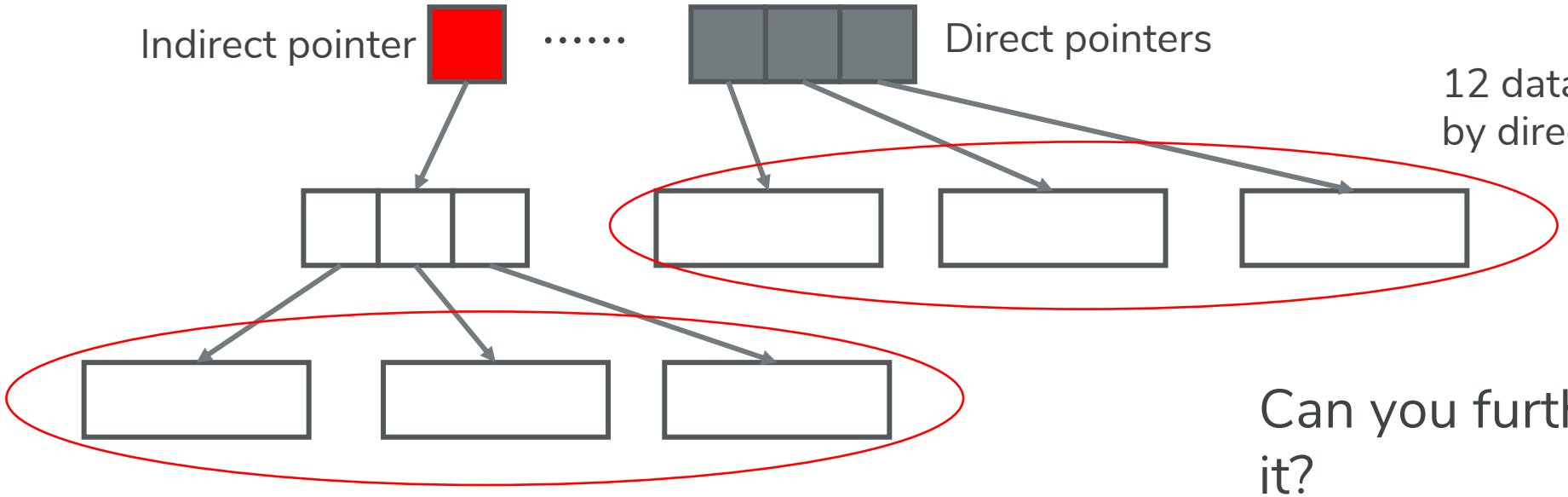
iNode: Mapping V.A. to P.A.

- Two-level index: we let one or a few pointers point to a data block
  - The data block stores more pointers to real data blocks
  - Such pointers in inodes are called "indirect pointers"

- What is the max file size now? (Assume 12 direct pointers and 1 indirect pointer)

iNode: Mapping V.A. to P.A.

- What is the max file size now? (Assume 12 direct pointers and 1 indirect pointer)



Can you further improve it?

How many data blocks are referenced by the indirect pointer?

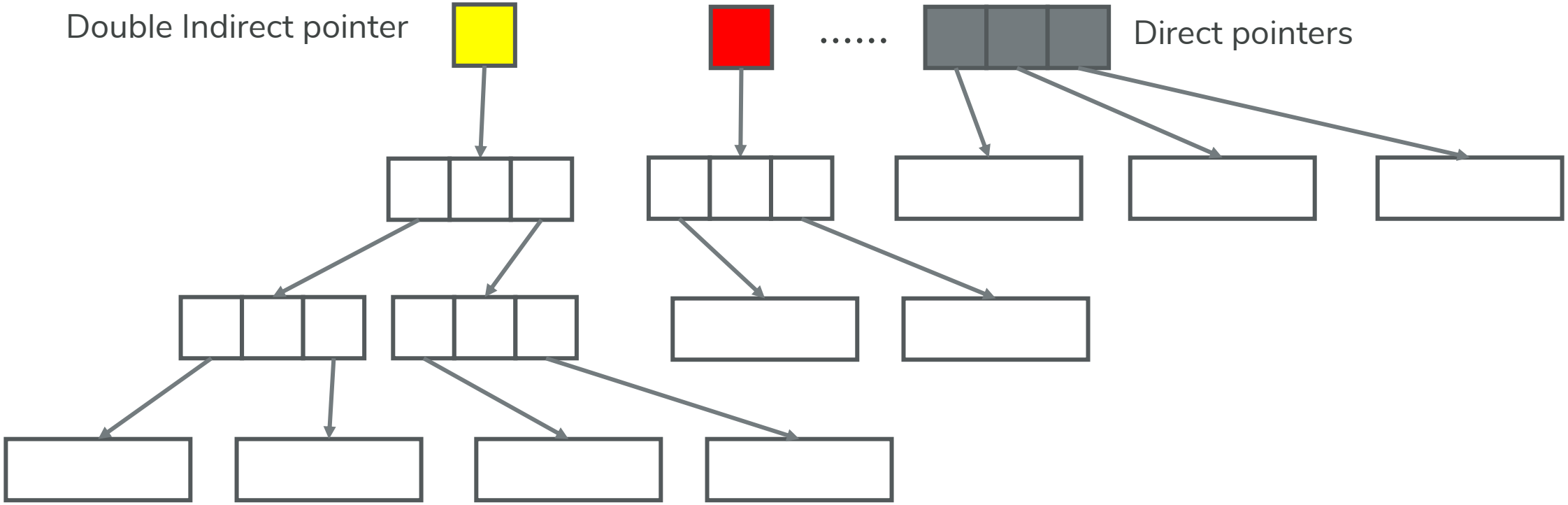
4KB (data block size) / 4B (pointer size) = 1024

Max number of blocks of a file = 12 + 1*1024 = 1036

Max file size = 1036 * 4KB = 4144KB (better, but still not enough)

iNode: Mapping V.A. to P.A.

- Double indirect pointer (Three-level index)



iNode: Mapping V.A. to P.A.

- Double indirect pointer (Three-level index)
  - Assume 12 direct pointers, 1 indirect point, and 1 double indirect pointer, what is max size of a file?
  - (12 + 1024 + 1024*1024) * 4KB = slightly more than 4GB

- To allow even larger files, use **triple indirect pointer** (**Four-level index**)
  - Do the math by yourself

- Such multi-level index idea is widely used in computer science.
  - Paging, file system, distributed system, …

iNode: Mapping V.A. to P.A.

- Why not use all 15 pointers as triple indirect pointers? We can support an even larger file.
- Fact: most files are small
- For small files, direct pointers are enough and are faster

iNode: Mapping V.A. to P.A.

- Given a virtual block number of a file, how to find its physical block number?

- If the block number is in [0, 12), it is referenced by a direct pointer.
- If the block number is in [12, 12 + 1024), it is referenced by the indirect pointer.
- ……

Keep track of blocks of a file

- Similar as in memory management, there are other solutions
- Contiguous allocation (extents): simple to keep track of blocks; external fragmentation
  - We can allow multiple extents in a file (like the segmentation approach in memory management)
- Linked list
  - Let each data block has a pointer to the next one
  - Random access becomes very inefficient

Directory Organization

- Abstractly a directory contains several children, each with an inode and a name

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

- Physically a directory can be implemented as a special file.
  - Reading the above information is like reading a file.

Free Space Management

- Need to remember which inodes/data blocks are used/unused
- Use one bitmap for each
  - Bitmap is an array of bits: bits[i] = true means entity i is used.
- When creating a file: search the bitmap for an unused inode/data block, and then mark them as used in the bitmap