

CIS 657 – Principles of Operating Systems

Topic: Persistence – File System Implementation

Endadul Hoque

Acknowledgement

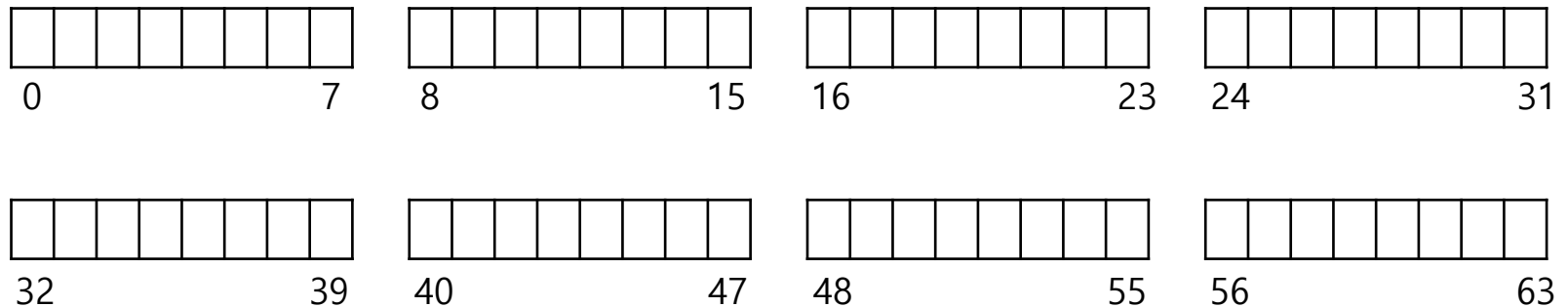
- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

The Way To Think

- There are two different aspects to implement file system
 - **Data structures**
 - What types of **on-disk structures** are utilized by the file system to organize its data and metadata?
 - **Access methods**
 - How does it map the calls made by a process as `open()`, `read()`, `write()`, etc.?
 - Which structures are read during the execution of a particular system call?

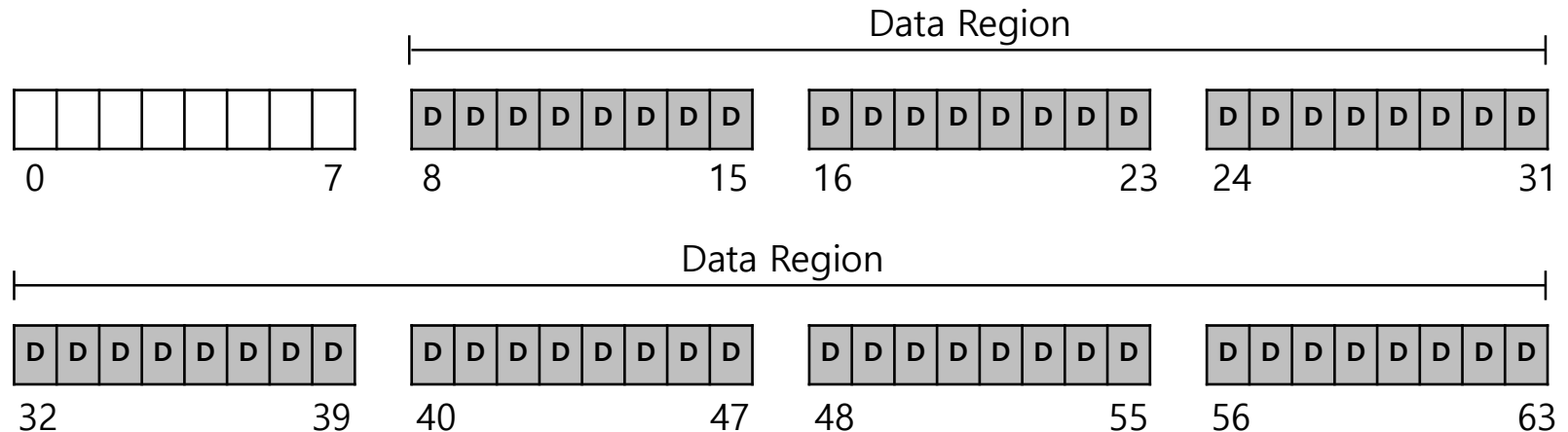
Overall Organization

- Let's develop the overall on-disk organization of the file system data structure.
- **vsfs** (very simple file system)
- Divide the disk into **blocks**.
 - Block size is 4 KB.
 - The blocks are addressed from 0 to $N - 1$.



Data region in file system

- Reserve **data region** to store user data (i.e., file content)

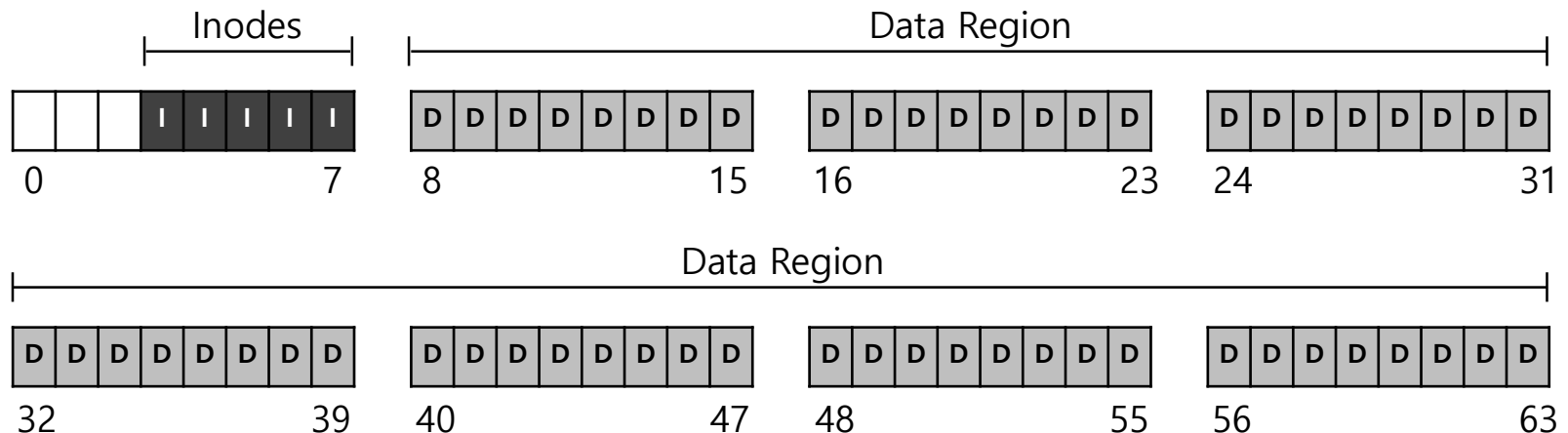


- File system has to track which data blocks comprise a file and its metadata (the size of the file, its owner, etc.)
 - It uses an **inode** (index node) structure

**How can we store
the *inodes* in the file
system?**

Inode table in file system

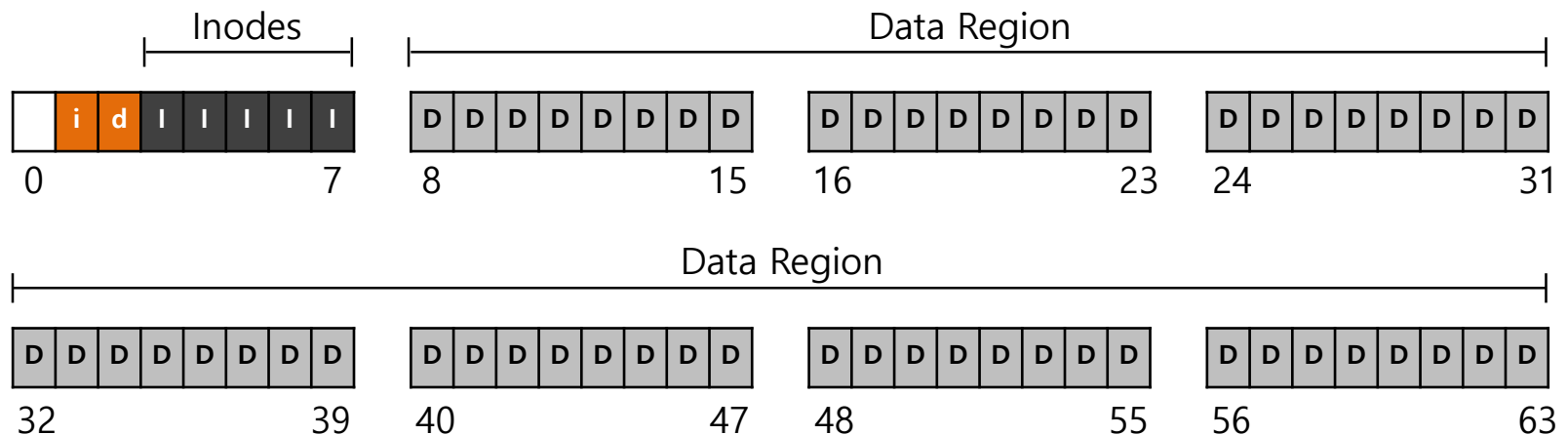
- Reserve some space for **inode table**
 - This holds an array of on-disk inodes.
 - Ex) inode table: 5 blocks (from 3 to 7) and inode size : **256 bytes**
 - Each 4-KB block can hold 16 inodes.
 - The filesystem supports total **80 inodes**. (maximum number of files)



**How does the file system
keep track whether
inodes or data blocks are
free or allocated?**

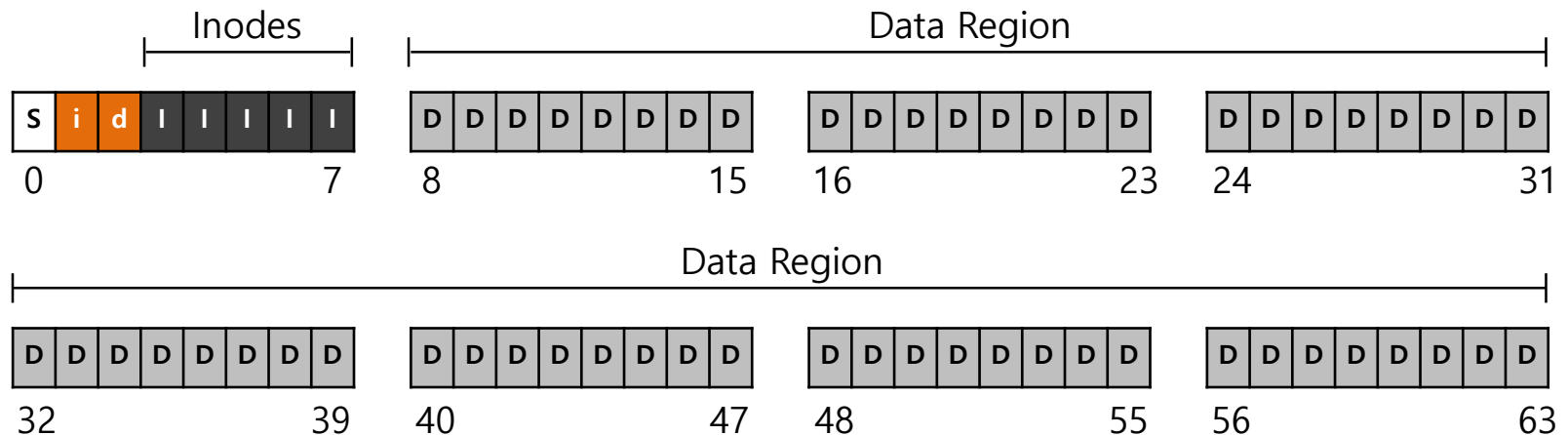
Allocation structures

- These structures are to track whether inodes or data blocks are free or allocated.
- Use **bitmap**, each bit indicates **free(0)** or **in-use(1)**
 - **data bitmap**: for data region
 - **inode bitmap**: for inode table



Superblock

- Super block (**S**) contains the **information** for **particular file system**
 - Ex) The number of inodes, starting location of inode table. etc



- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by the inode number.
 - Given an inode number, File system can calculate the location of the inode on the disk.
 - Ex) inode number: **32**
 - Calculate the offset into the inode region = inode number \times sizeof(inode) = 32×256 bytes = 8192 bytes = 8 KB
 - Actual location = start address of the inode table + the offset = 12 KB + 8 KB = 20 KB

The Inode table

				iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
Super	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67	
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71	
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75	
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79	
0KB	4KB	8KB	12KB	16KB				20KB				24KB				28KB				32KB			

File Organization: The inode

- Disk are ***not byte addressable***, **sector addressable**.
- Disk consists of a large number of addressable sectors (e.g., 512 bytes each)
 - Ex) Fetch the desired inode block for inode number 32

The sector of the inode block can be calculated as follows:

```
blk = (inumber * sizeof(inode)) / blocksize = 2
```

```
sector = ((blk * blocksize)+inodeStratAddr)/sectorsize = 40
```

The Inode table

				iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
Super	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67	
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71	
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75	
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79	

0KB

4KB

8KB

12KB

16KB

20KB

24KB

28KB

32KB

File Organization: The inode

- `inode` have all of the information (**metadata**) about a file
 - File type (regular file, directory, etc.),
 - Size, the number of blocks allocated to it.
 - Protection information(who owns the file, who can access, etc).
 - Time information.
 - Etc.
- inode needs to refer to where data blocks of this file are
- One simple approach: use **direct pointers** (say, 12 pointers)
 - Each pointer refers to one disk block that belongs to the file
 - **Limitation:** A file can grow to be **48 KB** ($= 4\text{KB} \times 12$)

File Organization: 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?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	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
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

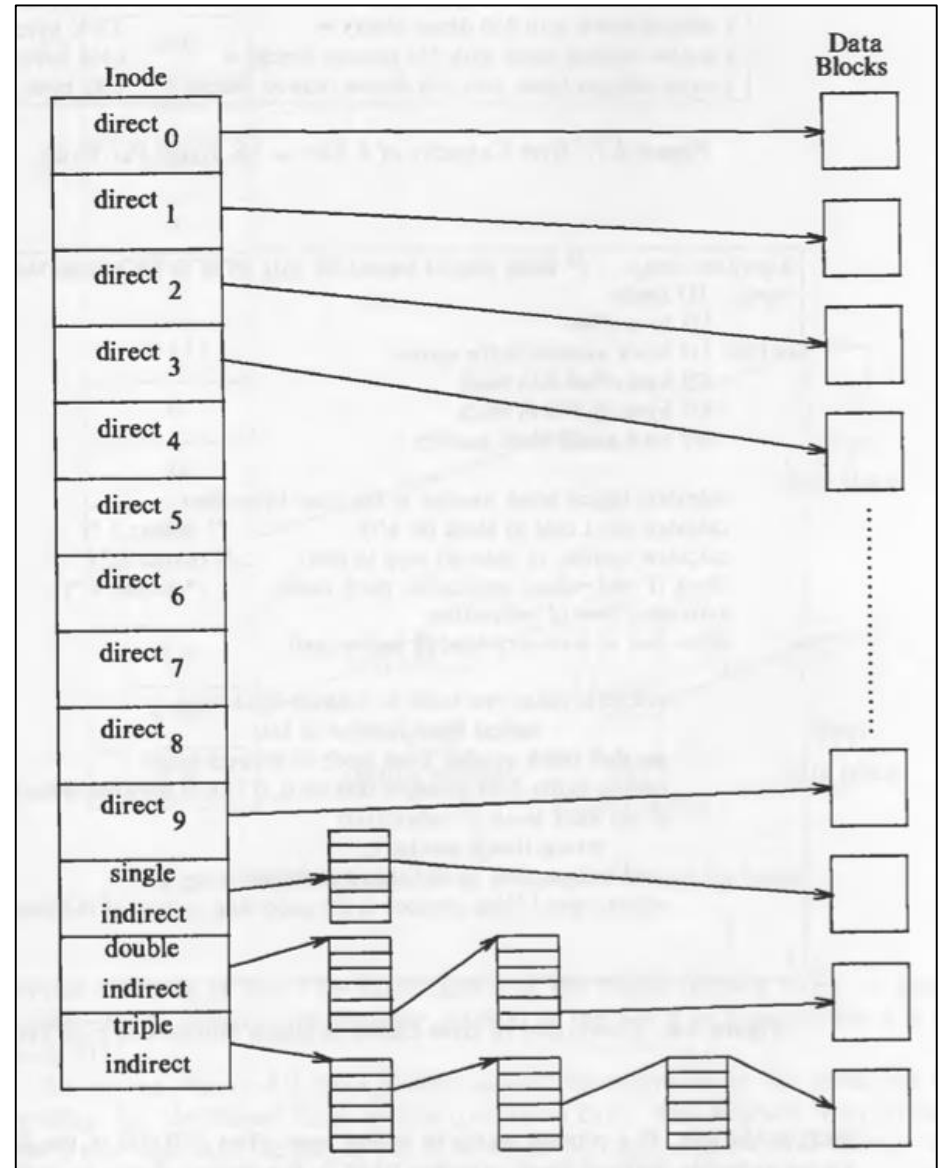
The EXT2 Inode

The Multi-Level Index

- To support bigger files, we use multi-level index.
- **Indirect pointer** points to a disk block that contains more pointers.

The Multi-Level Index

- To support bigger files, we use multi-level index.
- **Indirect pointer** points to a disk block that contains more pointers.



src: "The Design of the UNIX Operating System" by Maurice J. Bach. 1986.

The Multi-Level Index

- To support bigger files, we use multi-level index.
- **Indirect pointer** points to a disk block that contains more pointers.
 - Ex) inode have fixed number of **direct pointers** (12) and a single indirect pointer.
 - If a file grows large enough, an indirect block is allocated (from the data-block region), and inode's slot for an indirect pointer is set to point to it
 - A file can grow to be $(12 + 1024) \times 4\text{ K}$ or 4144 KB

The Multi-Level Index

- **Double indirect pointer** points to a block that contains pointers to indirect blocks.
 - Allow file to grow with an additional 1024×1024 or 1 million 4KB blocks.
- **Triple indirect pointer** points to a block that contains pointers to double indirect blocks.
- Multi-Level Index approach to pointing to file blocks.
 - Ex) 12 direct pointers, a single and a double indirect block.
 - over 4GB in size, because $(12 + 1024 + 1024^2) \times 4\text{KB}$
- Many file system use a multi-level index.
 - Linux EXT2, EXT3, NetApp's WAFL, Unix file system.
 - Linux EXT4 use **extents** instead of simple pointers. (Extent = a pointer + a length in blocks ... similar to variable-length memory segments)

The Multi-Level Index

Most files are small

Average file size is growing

Most bytes are stored in large files

File systems contains lots of files

File systems are roughly half full

Directories are typically small

Roughly 2K is the most common size

Almost 200K is the average

A few big files use most of the space

Almost 100K on average

Even as disks grow, file system remain ~50% full

Many have few entries; most have 20 or fewer

File System Measurement Summary

Directory Organization


- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files **.** (dot) for current directory and **..** (dot-dot) for parent directory
 - For example, `dir` has three files (`foo`, `bar`, `foobar`)

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

on-disk dir content

Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files **.** (dot) for current directory and **..** (dot-dot) for parent directory
 - For example, `dir` has three files (`foo`, `bar`, `foobar`)



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

on-disk dir content

Total bytes for the name + any left over space (because a new entry may reuse an old, bigger entry)

Free Space Management

- File system tracks which inodes and data blocks are free or not.
- In order to manage free space, we have two simple bitmaps.
 - When a file is newly created, it allocates inode by **searching** the inode bitmap and **update** on-disk bitmap.
 - Pre-allocation policy is commonly used for allocate contiguous data blocks.

Access Paths: Reading a File From Disk

- Issue an `open ("/foo/bar", O_RDONLY),`
 - Traverse the pathname and thus locate the desired inode.
 - Begin at the root of the file system (`/`)
 - In most Unix file systems, the root inode number is 2
 - Filesystem reads in the block that contains inode number 2.
 - Look inside of it to find pointer to data blocks (contents of the root).
 - By reading in one or more directory data blocks, it will find "foo" directory.
 - Traverse recursively the path name until the desired inode ("bar")
 - Check final permissions, allocate a file descriptor for this process and returns file descriptor to user.

Access Paths: Reading a File From Disk

- Issue `read()` to read from the file.
 - Read in the first block of the file, consulting the inode to find the location of such a block.
 - Update the inode with a new last accessed time.
 - Update in-memory open file table for file descriptor, the file offset.
- When file is closed:
 - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No disk I/Os take place.

Access Paths: Reading a File From Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read	read	read	read	read			
read()					read			read		
read()					read				read	
read()					read					read

File Read Timeline (Time Increasing Downward)

Access Paths: Writing to Disk

- Issue `write()` to update the file with new contents.
- File may allocate a block (unless the block is being overwritten).
 - Need to update data block, data bitmap.
 - It generates five I/Os:
 - one to read the data bitmap
 - one to write the bitmap (to reflect its new state to disk)
 - two more to read and then write the inode
 - one to write the actual block itself.
 - To create file, it also allocates space for directory, causing high I/O traffic.

Access Paths: Writing to Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read		read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

File Creation Timeline (Time Increasing Downward)

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os.
 - For example, long pathname(/1/2/3/.../100/file.txt)
 - One read for the inode of the directory and at least one read for its data.
 - Literally perform hundreds of reads just to open the file.
- In order to reduce I/O traffic, file systems aggressively use system memory (DRAM) to cache.
 - Early file system use fixed-size cache to hold popular blocks.
 - Static partitioning of memory can be wasteful;
 - Modern systems use **dynamic partitioning approach, unified page cache.**
- Read I/O can be avoided by large cache.

Caching and Buffering

- Write traffic has to go to disk for persistent. Thus, cache does not reduce write I/Os.
- File system use **write buffering** for write performance benefits.
 - **Delaying writes** (file system **batch** some updates into a smaller set of I/Os).
 - By **buffering** a number of writes in memory, the file system can then schedule the subsequent I/Os.
 - In some cases, it can avoid some writes completely
- Some applications force flush data to disk by calling `fsync()` or calling direct I/O functions.

Reading Material

- **Chapter 40** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)
<http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf>

Questions?