File System Implementation

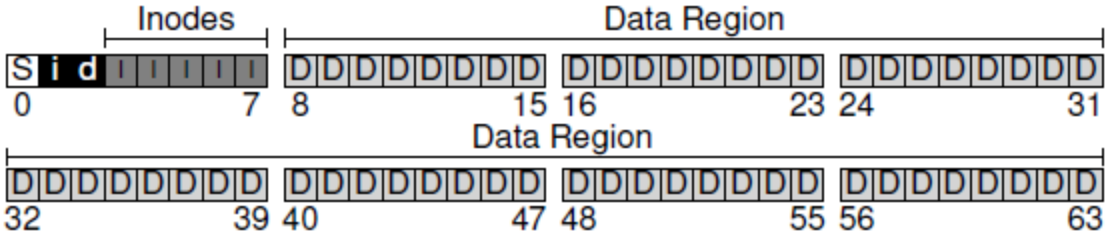
File System Implementation

- Data structures
  - What types of on-disk structures are utilized by the file system to organize its data and metadata?
  - Simple structures, like arrays of blocks or other objects
  - More complicated tree-based structures
- Access methods
  - How map the calls made by a process, such as open(), read(), write(), etc., onto its structures?
  - Which structures are read during the execution of a particular system call?
  - Which are written?
  - How efficiently are all of these steps performed?

- This class introduces a simple file system implementation, known as vsfs (the Very Simple File System).
  - a simplified version of a typical UNIX file system

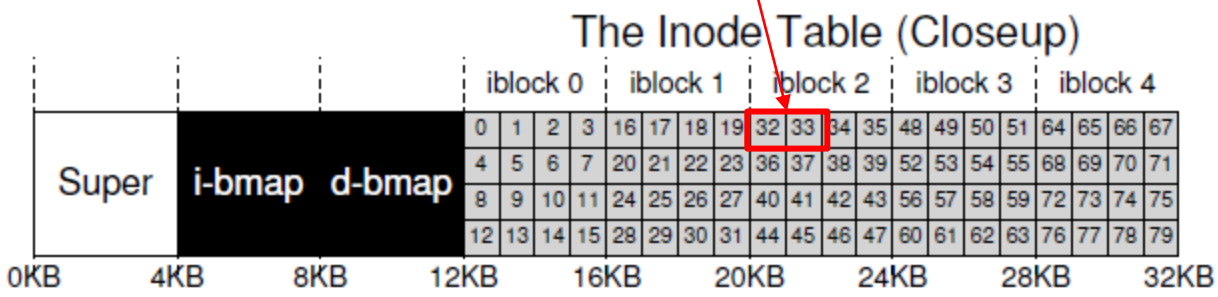
Overall Organization of vsfs

- Divide the disk into blocks (4KB)
- Data region (D): user data
- Metadata
  - Inode table (I)
    - Inode (128B or 256B): pointers to data blocks, file size, owner/access rights, access/modify times, etc.
    - a 4-KB block can hold 16 inodes (256B)
    - 5 blocks of inode table can contain 80 total inodes (=max # of files)
  - Allocation structures: data bitmap (d) and inode bitmap (i)
  - Superblock (S): contains information about file system
    - # of (max, allocated, free) inodes and data blocks in the file system
    - the start addresses of inode table, inode bitmap, data bitmap
    - a magic number to identify the file system type
    - Mount operation reads the superblock first to initialize various parameters



Inode (index node)

- Each inode is implicitly referred to by a number (i-number)
- Given an i-number, you can calculate where on the disk the corresponding inode is located
  - To read inode number 32,
  - Calculate offset: 32*sizeof(inode) = 8192
  - add it to the start address of the inode table on disk (inodeStartAddr = 12KB), then 20KB
  - Sector address: 20KB/512B = 40th sector



metadata

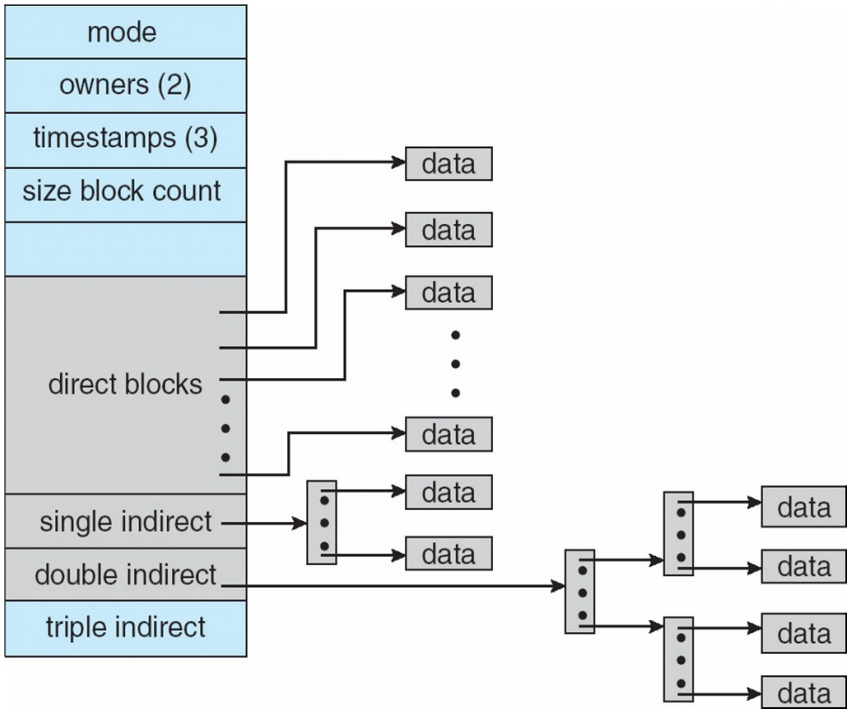
- Type (e.g., regular file, directory, etc.), size, the number of blocks allocated
- Protection (who owns the file, who can access it)
- Time (created, modified, or last accessed)
- Where its data blocks reside on disk (e.g., pointers of some kind)
- How it refers to where data blocks are is an important design decision
  - Direct or indirect pointers
  - Single block or multiple blocks

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**

Multi-Level Index

- **Indirect pointer**
  - Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data.
  - ext2/ext3, NetApp's WAFL, UNIX



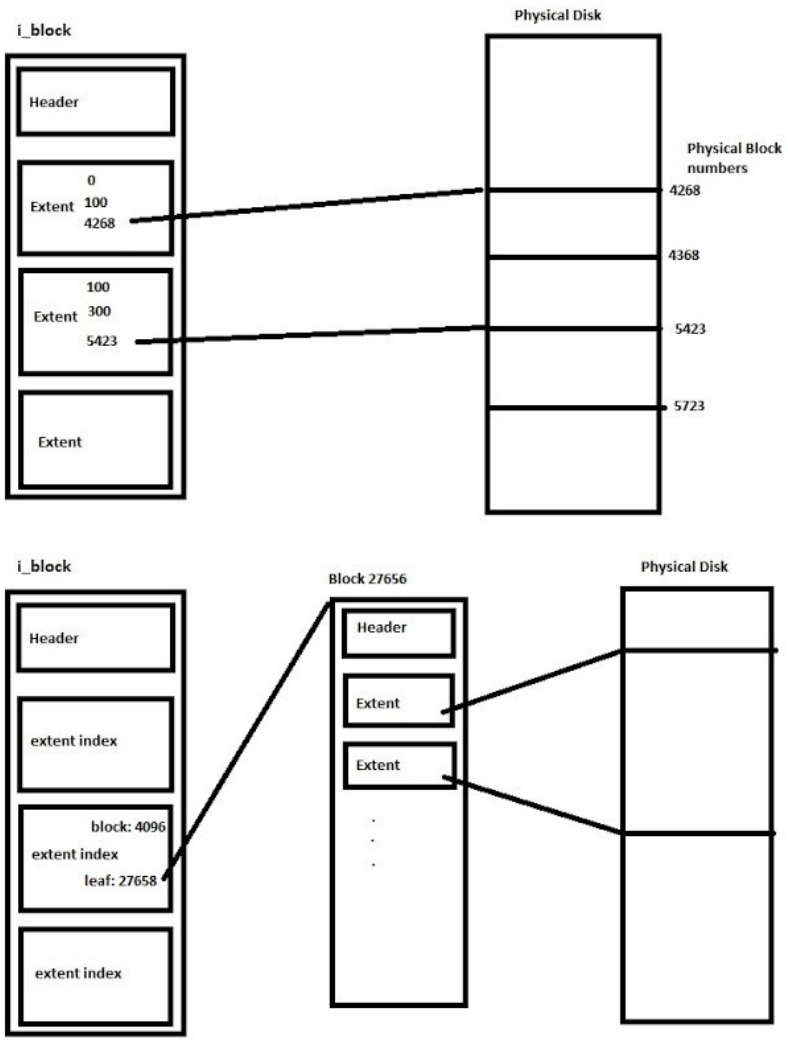
- **Ex.**
  - An inode may have 12 direct pointers and a single indirect pointer
  - 4-KB blocks and 4-byte disk addresses
  - Max file size
    - $(12+1024)\times 4K = 4144KB$
  - If you add a double indirect block
    - $(12+1024+1024^2)\times 4KB$ = over 4GB
- **Why imbalanced tree?**
  - most files are small

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Agrawal [FAST'07]

Extent Mapping

- Ext4, XFS
- An extent = a disk pointer plus a length
- Pointer-based approaches
  - flexible but use a large amount of metadata per file (particularly for large files).
- Extent-based approaches
  - less flexible but more compact
  - work well when there is enough free space on the disk and files can be laid out contiguously (not fragmented)



Directory Organization

- In vsfs (as in many file systems), a directory basically just contains a list of (entry name, inode number) pairs.
- each entry has
  - inode number
  - record length (the total bytes for the name plus any left over space)
  - string length (the actual length of the name)
  - the name of the entry
- File systems treat directories as a special type of file.
  - A directory has an inode, marked as "directory" instead of "regular file".
  - The directory has data blocks pointed to by the inode
- Simple linear list of directory entries is inefficient
  - XFS uses B-tree; faster file creation

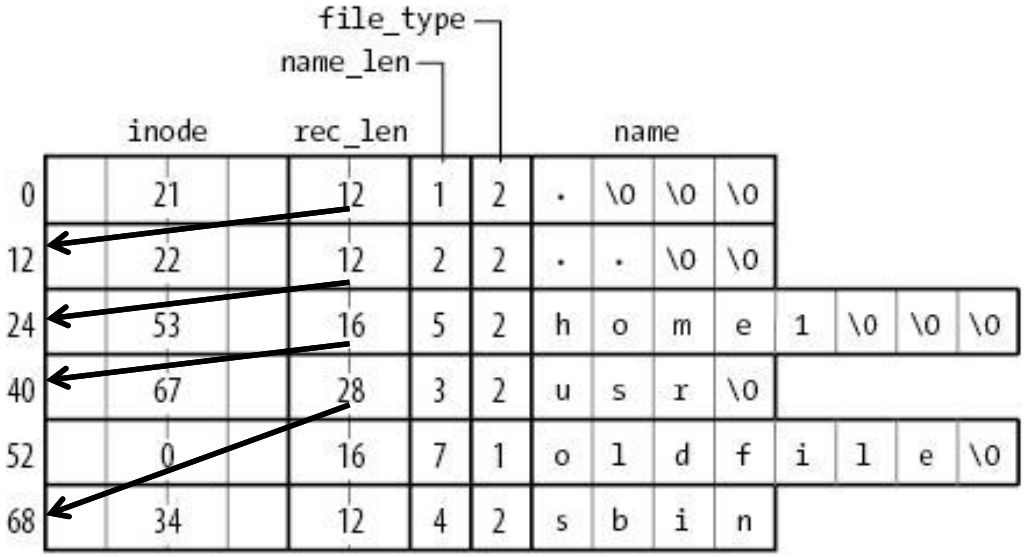
current directory
parent directory

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

Directory Entry (EXT4)

struct ext4_dir_entry_2

Type	Field	Description
__le32	inode	Inode number
__le16	rec_len	Directory entry length (pointer to next item)
__u8	name_len	Filename length (real)
__u8	file_type	File type
char [EXT4_NAME_LEN]	name	Filename (A multiple of 4)

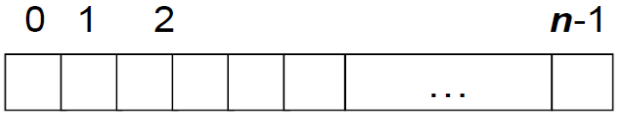


When Ext4 wants to delete a directory entry, it just increase the record length of the previous entry to the end to deleted entry.

→ Deleted

Free Space Management

- Must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it.
- Bitmap, free list, or B-tree
- Bitmap approach
  - Search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1)
  - A similar set of activities take place when a data block is allocated
- **Pre-allocation**
  - Contiguous block allocation, improve performance



$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

Bit map requires extra space
Example:
- block size = 4KB = 2^{12} bytes
- disk size = 2^{40} bytes (1 terabyte)
- $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)
- if clusters of 4 blocks -> 8MB of memory

Reading A File From Disk

- Open a file (e.g., /foo/bar), read it, and then close it.
- The file is just 4KB in size (i.e., 1 block)

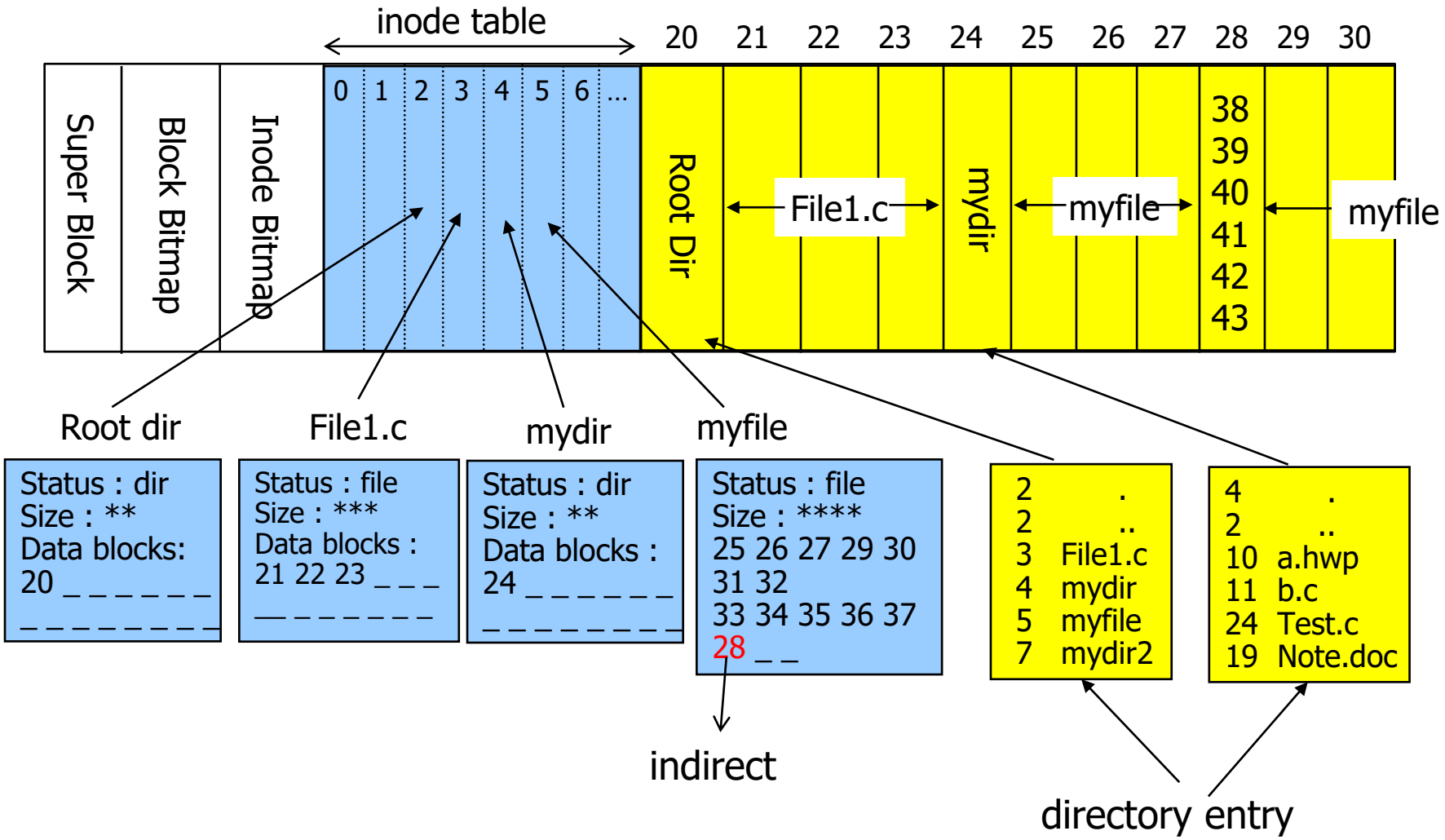
1. Open(): Find the inode for the file bar (traverse the pathname)
   (1) Read root inode (root i-number is pre-determined, e.g., 2)
   (2) Read the data block of / to find the i-number of /foo
   (3) Read the inode of /foo
   (4) Read the data block of /foo to find the i-number of /foo/bar
   (5) Read the inode of /foo/bar
2. Read bar's inode into memory; does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.
3. Read()
   (1) consult the inode to find the location of the target block
   (2) also update the inode with a new last accessed time.
   (3) update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the next file block
4. Close()
   (1) the file descriptor should be deallocated,
   (2) No disk I/Os

I/O is proportional to the length of the pathname. Should every open traverse the pathname?

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					
					write					
read()					read					
					write					
read()					read					
					write					

Inode & Directory



Writing to Disk

- Writing to a file may also **allocate** a block
  - Decide which block to allocate to the file
  - Update other structures of the disk accordingly (bitmap, inode)
  - Five IOs for each write()
    - read and then write the data bitmap to mark the newly-allocated block as used
    - read and then write the inode (which is updated with the new block's location)
    - write the actual block itself.
- File creation
  - allocate an inode & space within the directory containing the new file.
  - Six IOs
    - read the inode bitmap (to find a free inode)
    - write to the inode bitmap (to mark it allocated)
    - write to the new inode itself (to initialize it)
    - write to the data of the directory (to link the high-level name of the file to its inode number)
    - read and write to the directory inode to update it.
  - If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too.

Writing to Disk

The file /foo/bar is created, and three blocks are written to it.

	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 write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

10 I/Os to create a file

5 I/Os for allocating write

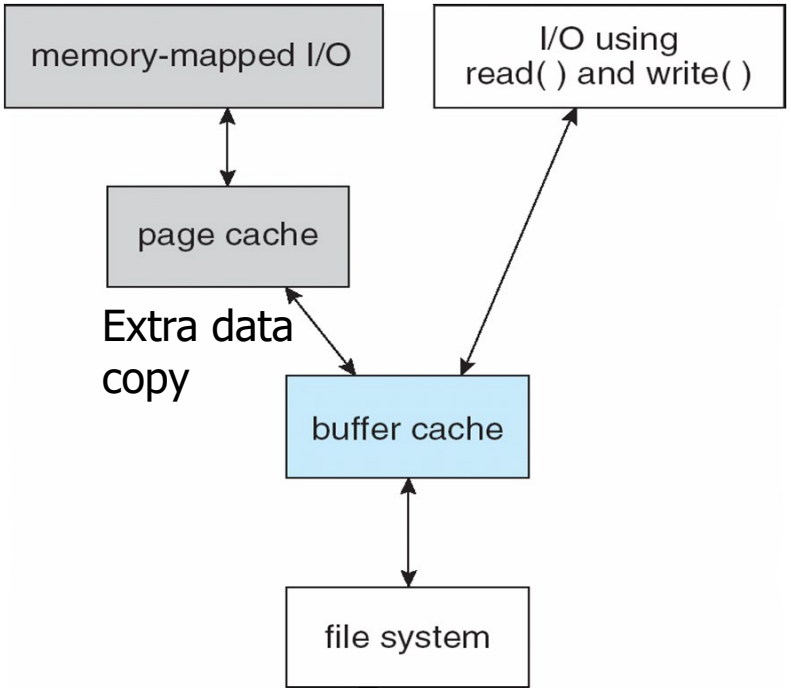
What can a file system do to reduce the high costs of doing so many I/Os?

Caching and Buffering

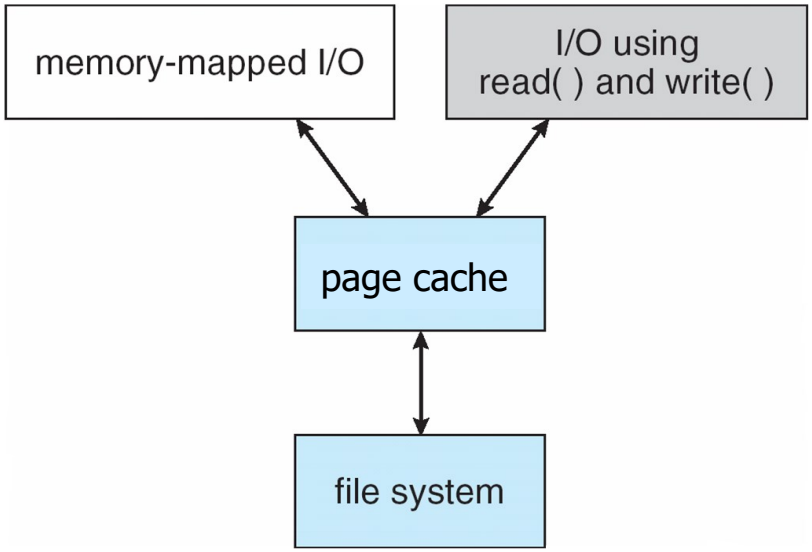
- Reading and writing files incur many I/Os to the disk.
- Most file systems aggressively use system memory (DRAM) to cache important blocks.
  - The first open may generate a lot of I/O traffic to read in directory inode and data
  - Subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O needed.
- Static partitioning
  - Fixed-size cache (e.g., 10% of total memory)
  - can be wasteful: unused pages in the file cache cannot be re-purposed for some other use.
- Dynamic partitioning
  - integrate virtual memory pages and file system pages into a unified page cache
  - More flexible

Caching and Buffering

- Buffer cache – separate section of main memory for frequently used blocks (block address)
- Page cache – file data (virtual address)



I/O Without a Unified Page Cache



I/O Using a Unified Page Cache

Caching and Buffering

- Write traffic has to go to disk in order to become persistent.
- Benefits of write buffering
  - by delaying writes, the file system can batch some updates into a smaller set of I/Os. (e.g., multiple updates on an inode bitmap block)
  - by buffering a number of writes in memory, the system can then schedule the subsequent I/Os and thus increase performance.
  - Some writes are avoided altogether by delaying them
- How long buffering?
  - durability/performance trade-off
    - If the system crashes before the updates have been propagated to disk, the updates are lost
    - By keeping writes in memory longer, performance can be improved
  - Typically 5~30s of buffering
- To avoid unexpected data loss due to write buffering,
  - Call fsync()
  - Direct I/O: work around the cache
  - Raw disk