File System Internals

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Today's Topics

- **File system implementation**
  - File descriptor table, File table
  - Virtual file system

- **File system design issues**
  - Directory implementation: filename → metadata
  - Allocation: metadata → a set of data blocks

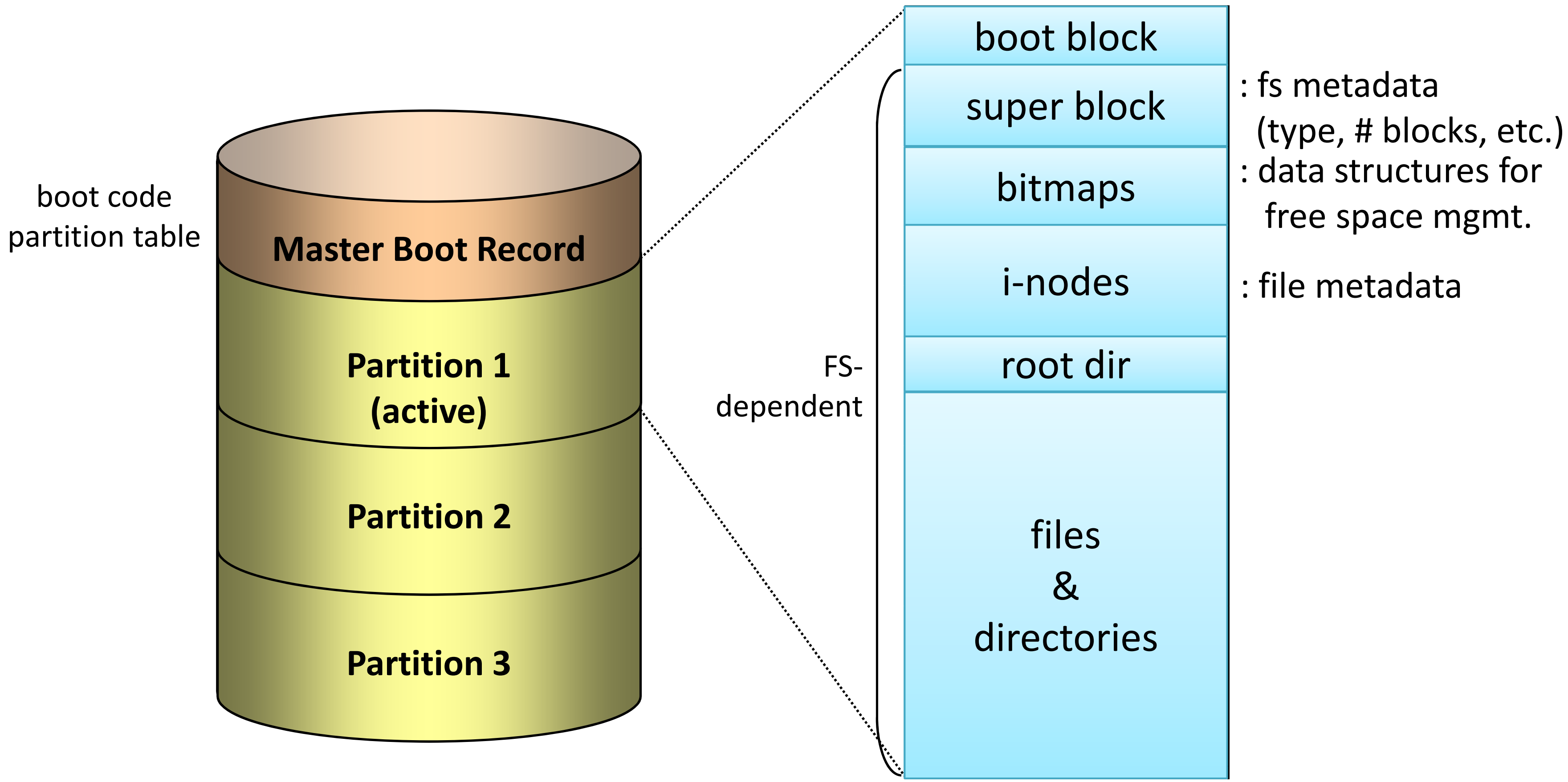
  - Reliability issues
  - Performance issues

Overview

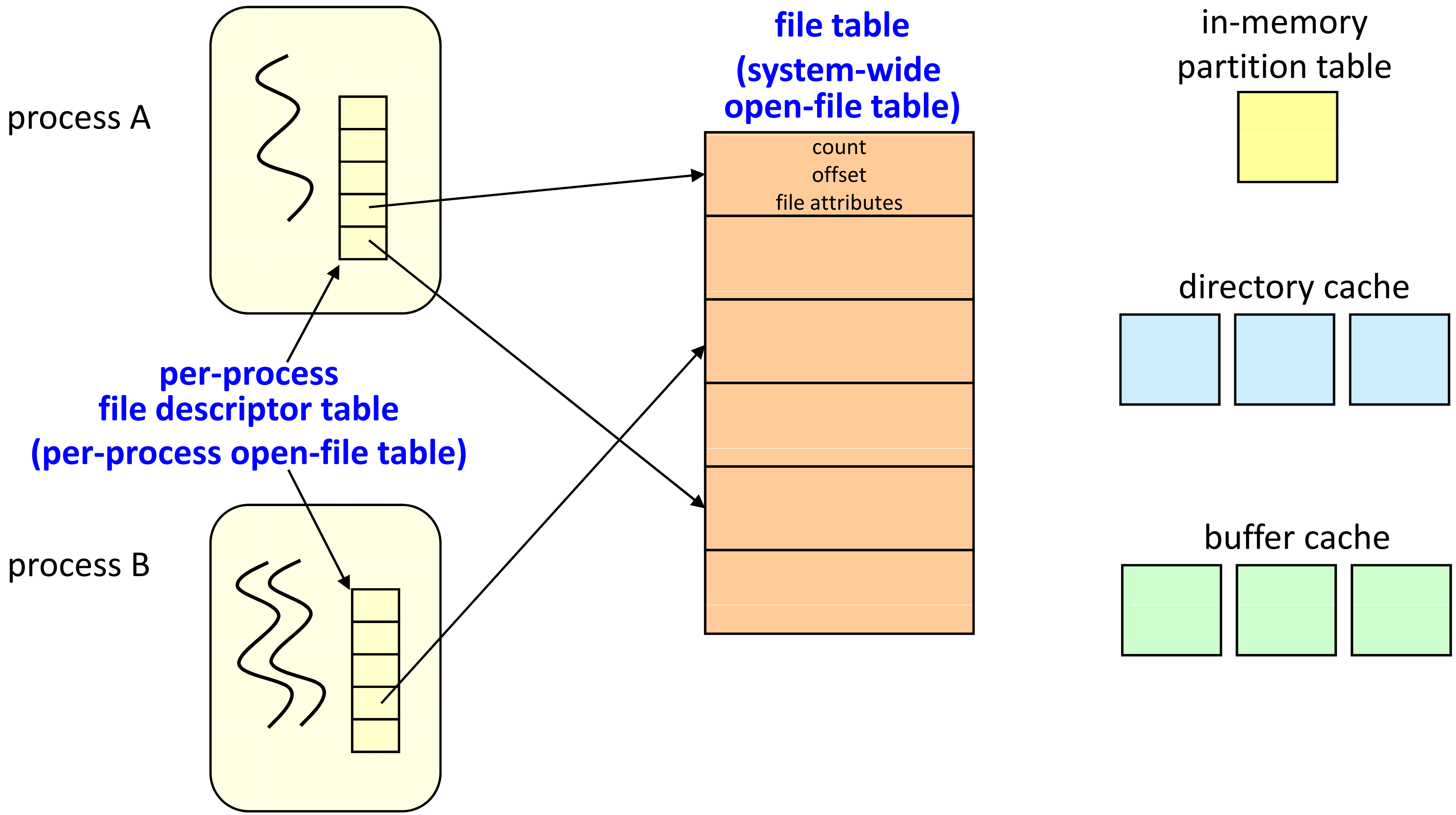
- **User's view on file systems:**
  - How files are named?
  - What operations are allowed on them?
  - What the directory tree looks like?

- **Implementor's view on file systems:**
  - How files and directories are stored?
  - How disk space is managed?
  - How to make everything work efficiently and reliably?

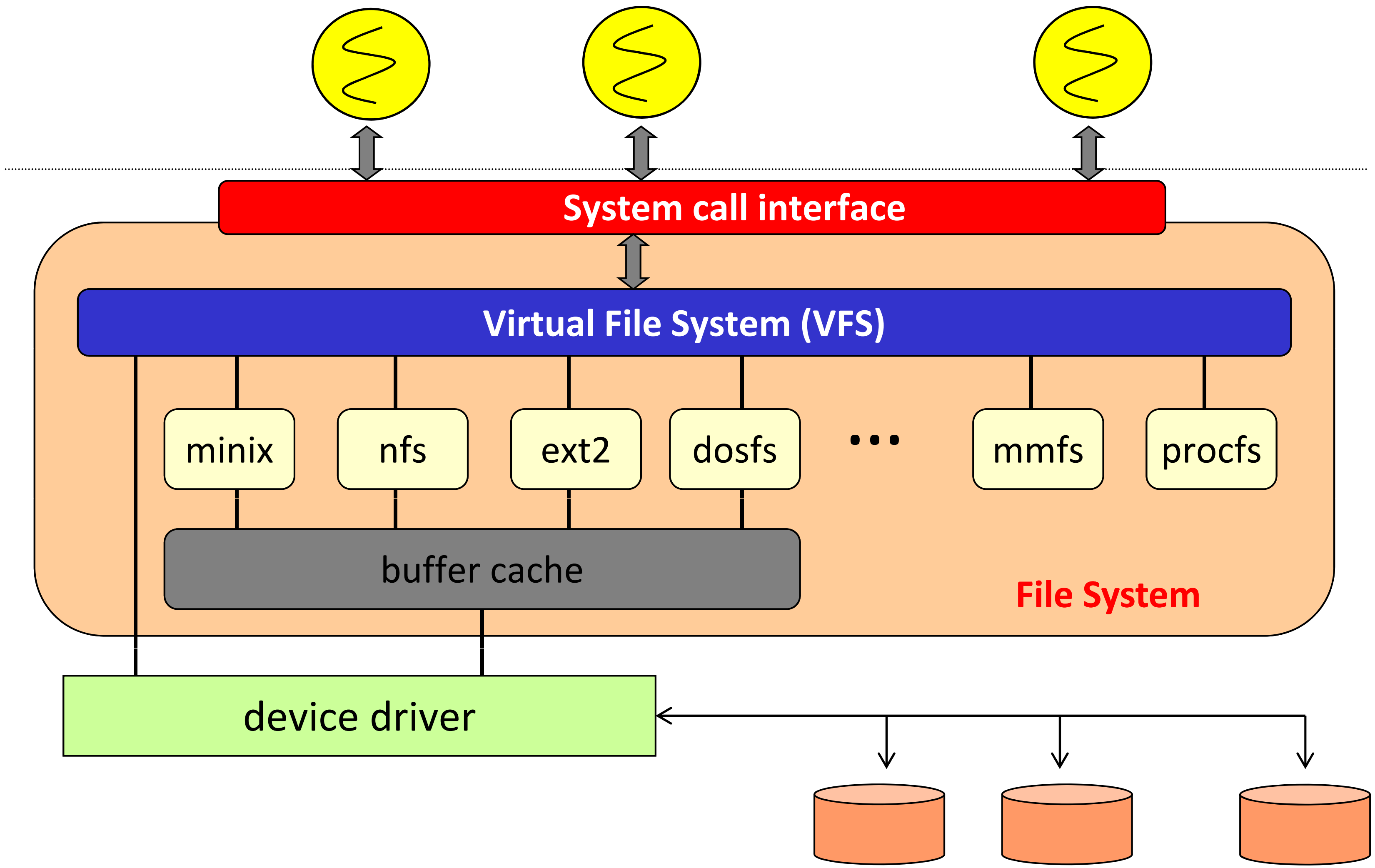
Disk Layout


boot code
partition table
Master Boot Record
Partition 1
(active)
Partition 2
Partition 3
FS-
dependent
boot block
super block
bitmaps
i-nodes
root dir
files
&
directories
: fs metadata
(type, # blocks, etc.)
: data structures for
free space mgmt.
: file metadata

In-memory Structures



File System Internals



VFS (1)

- **Virtual File System**
  - Manages kernel-level file abstractions in one format for all file systems.
  - Receives system call requests from user-level (e.g., open, write, stat, etc.)
  - Interacts with a specific file system based on mount point traversal.
  - Receives requests from other parts of the kernel, mostly from memory management.
  - Translates file descriptors to VFS data structures (such as vnode).

VFS (2)

- **Linux: VFS common file model**
  - The superblock object
    - stores information concerning a mounted file system.
  - The inode object
    - stores general information about a specific file.
  - The file object
    - stores information about the interaction between an open file and a process.
  - The dentry object
    - stores information about the linking of a directory entry with the corresponding file.
  - In order to stick to the VFS common file model, in-kernel structures may be constructed on the fly.

Directory Implementation (1)

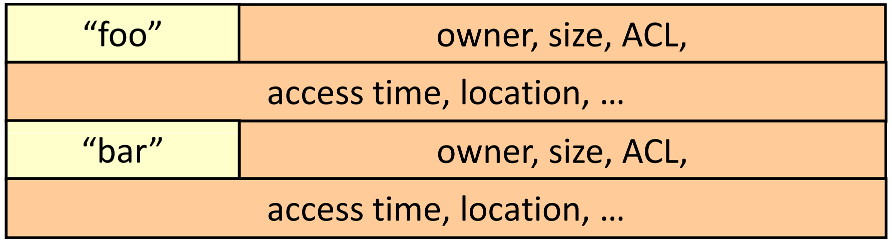
- **Directory structure**
  - Table (fixed length entries)
  - Linear list
    - Simple to program, but time-consuming.
    - Requires a linear search to find an entry.
    - Entries may be sorted to decrease the average search time and to produce a sorted directory listing easily (e.g., using B-tree).
  - Hash table
    - Decreases the directory search time.
    - A hash table is generally fixed size and the hash function depends on that size. (need mechanisms for collisions)
    - The number of files can be large:

      (1) enlarge the hash table and remap.

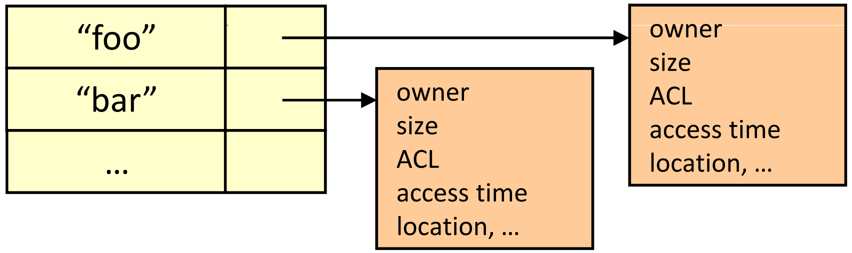
      (2) use a chained-overflow hash table.

Directory Implementation (2)

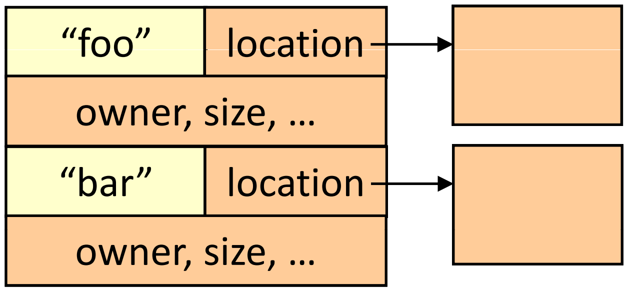
- **The location of metadata**
  - In the directory entry



  - In the separate data structure (e.g., i-node)

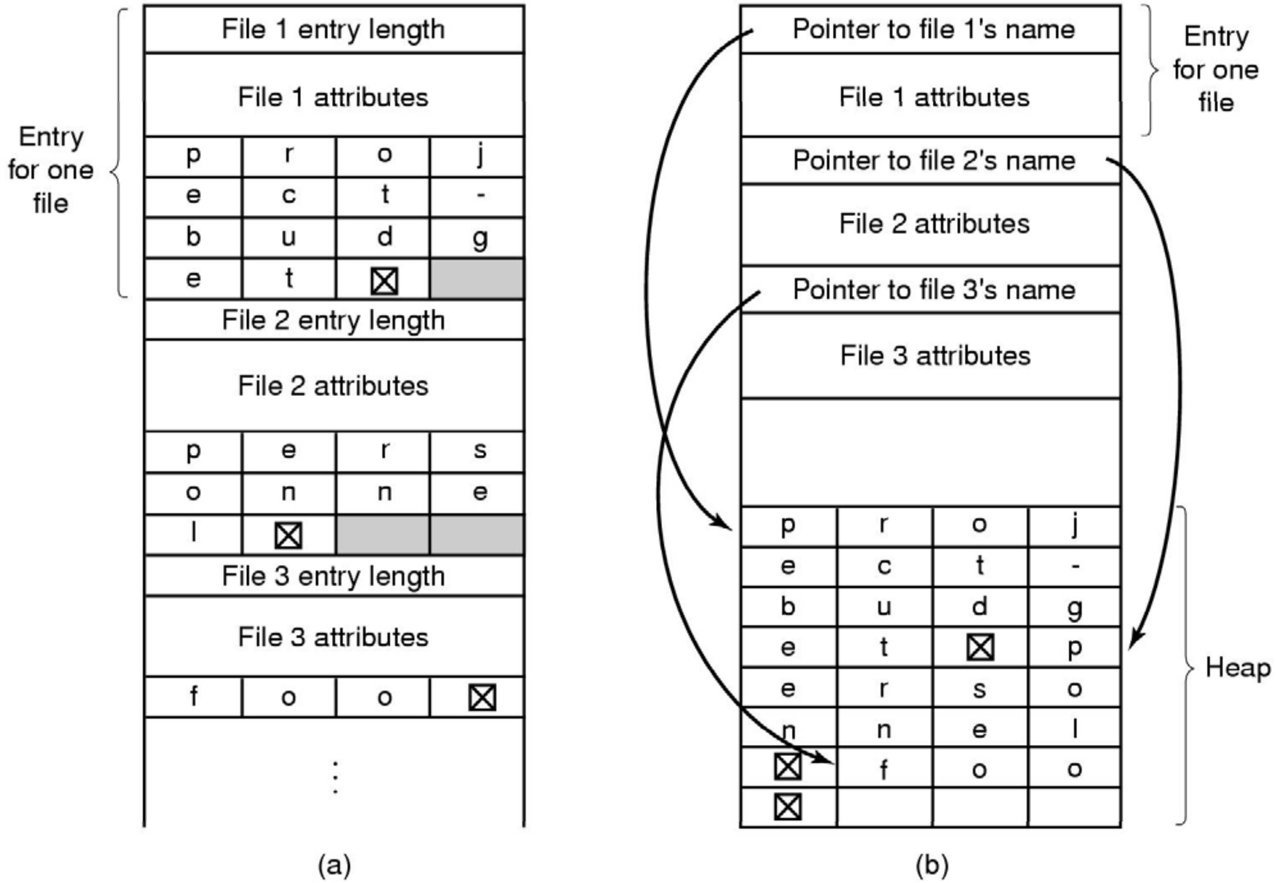


  - A hybrid approach



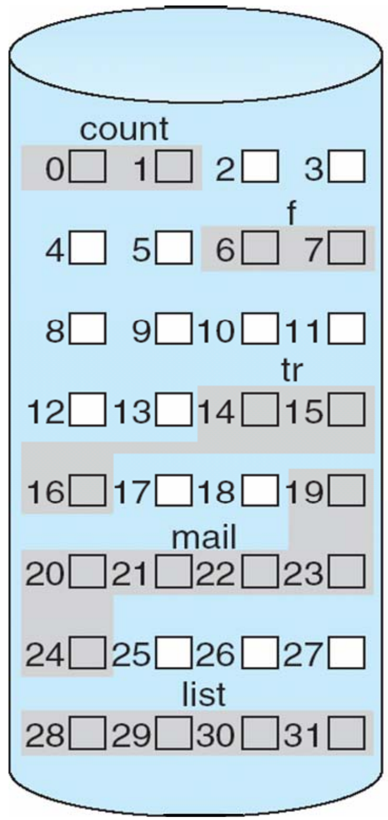
Directory Implementation (3)

- **Supporting long file names**



Allocation (1)

- **Contiguous allocation**
  - A file occupies a set of contiguous blocks on the disk.
  - Used by IBM VM/CMS



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation (2)

- **Contiguous allocation (cont'd)**
  - Advantages
    - The number of disk seeks is minimal.
    - Directory entries can be simple:
      <file name, starting disk block, length, etc.>
  - Disadvantages
    - Requires a dynamic storage allocation: First / best fit.
    - External fragmentation: may require a compaction.
    - The file size is hard to predict and varying over time.
  - Feasible and widely used for CD-ROMS
    - All the file sizes are known in advance.
    - Files will never change during subsequent use.

Allocation (3)

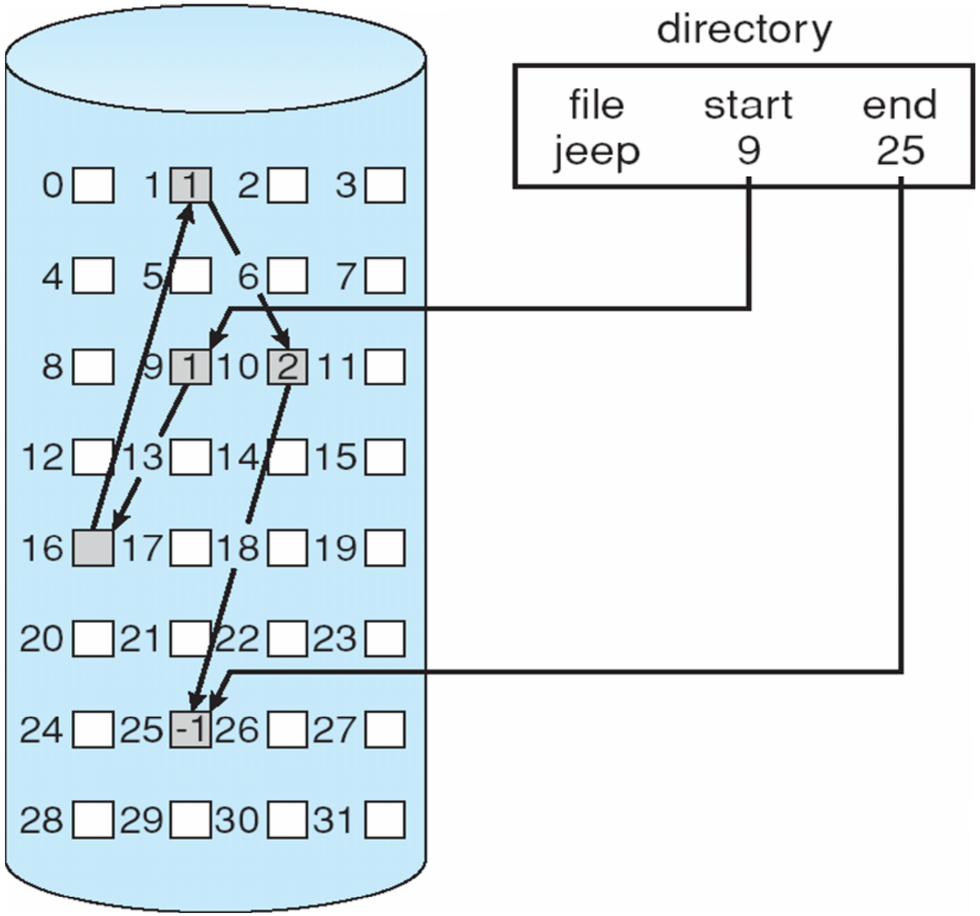
▪ Modified contiguous allocation

- A contiguous chunk of space is allocated initially.
  - When the amount is not large enough, another chunk of a contiguous space (an **extent**) is added.
- Advantages
  - Still the directory entry can be simple.

    <name, starting disk block, length, link to the extent>
- Disadvantages
  - Internal fragmentation: if the extents are too large.
  - External fragmentation: if we allow varying-sized extents.
- Used by Veritas File System (VxFS).

Allocation (4)

- **Linked allocation**
  - Each file is a linked list of disk blocks.



Allocation (5)

▪ Linked allocation (cont'd)

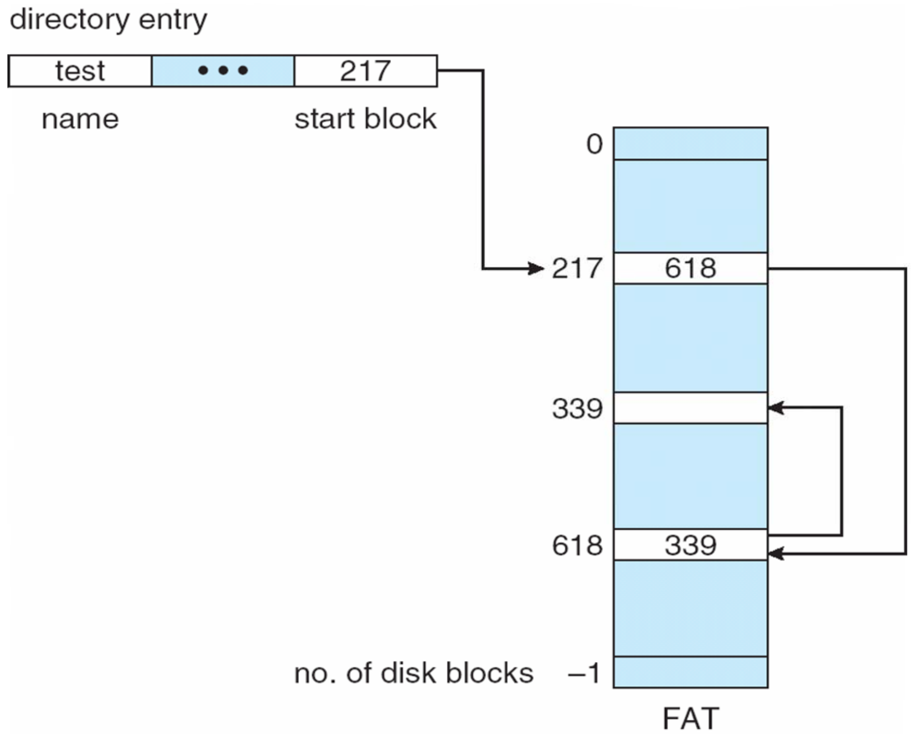
- Advantages
  - Directory entries are simple:
    <file name, starting block, ending block, etc.>
  - No external fragmentation: the disk blocks may be scattered anywhere on the disk.
  - A file can continue to grow as long as free blocks are available.
- Disadvantages
  - It can be used only for sequentially accessed files.
  - Space overhead for maintaining pointers to the next disk block.
  - The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes.
  - Fragile: a pointer can be lost or damaged.

Allocation (6)

- **Linked allocation using clusters**
  - Collect blocks into multiples (clusters) and allocate the clusters to files.
    - e.g., 4 blocks / 1 cluster
  - Advantages
    - The logical-to-physical block mapping remains simple.
    - Improves disk throughput (fewer disk seeks)
    - Reduced space overhead for pointers.
  - Disadvantages
    - Internal fragmentation

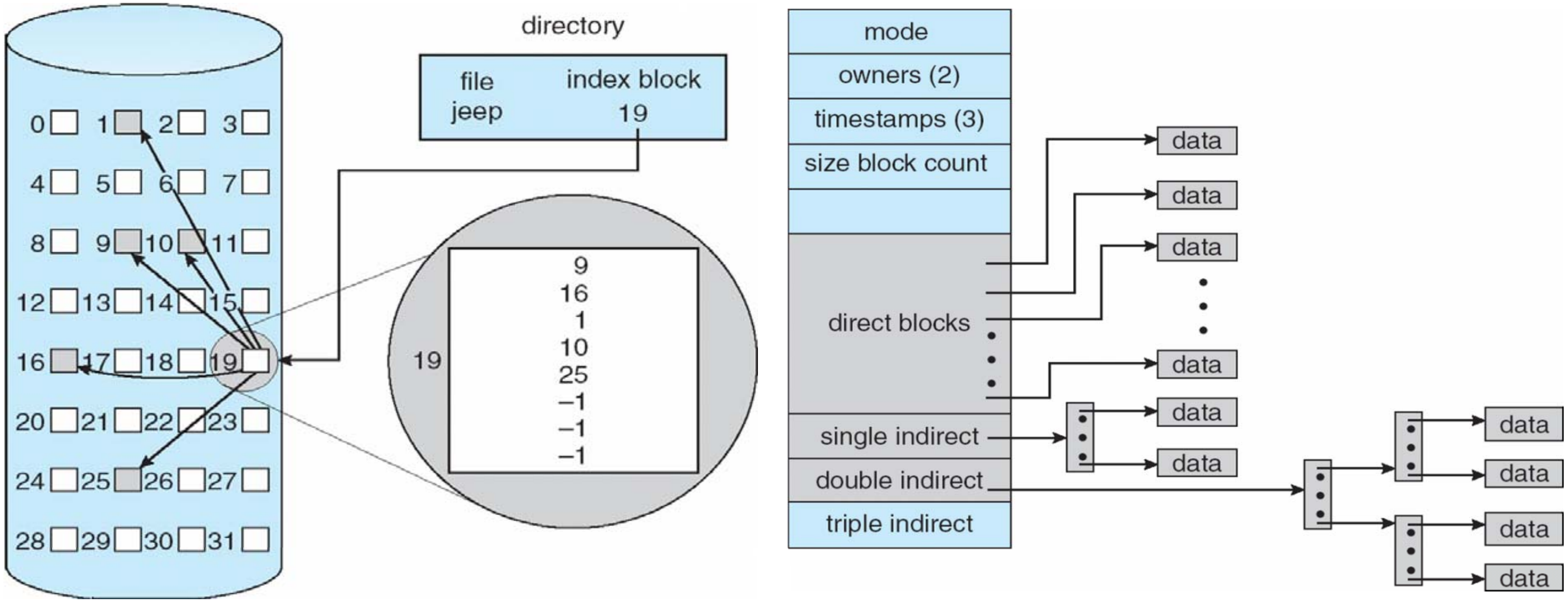
Allocation (7)

- **Linked allocation using a FAT**
  - A section of disk at the beginning of each partition is set aside to contain a file allocation table (FAT).
  - FAT should be cached to minimize disk seeks.
    - Space overhead can be substantial.
  - Random access time is improved.
  - Used by MS-DOS, OS/2
    - cf. FAT-16: 2GB limitation with 32KB block size



Allocation (8)

- **Indexed allocation**
  - Bring all the pointers together into one location (**index block** or **i-node**)
  - Each file has its own index block.



Allocation (9)

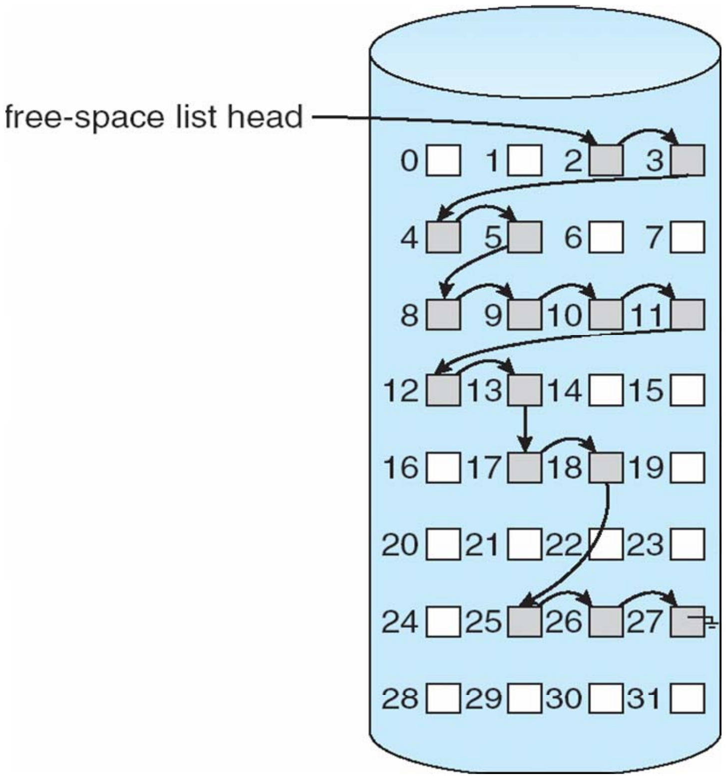
- **Indexed allocation (cont'd)**
  - Advantages
    – Supports direct access, without suffering from external fragmentation.
    – I-node need only be in memory when the corresponding file is open.
  - Disadvantages
    – Space overhead for indexes:
      (1) Linked scheme: link several index blocks
      (2) Multilevel index blocks
      (3) Combined scheme: UNIX
        - 12 direct blocks, single indirect block,
          double indirect block, triple indirect block

Free Space Management (1)

- **Bitmap or bit vector**
  - Each block is represented by 1 bit.
    - 1 = free, 0 = allocated
  - Simple and efficient in finding the first free block.
    - May be accelerated by CPU's bit-manipulation instructions.
  - Inefficient unless the entire vector is kept in main memory.
    - Clustering reduces the size of bitmaps.

Free Space Management (2)

- **Linked list**
  - Link together all the free disk blocks, keeping a pointer to the first free blocks.
  - To traverse the list, we must read each block, but it's not a frequent action.
  - The FAT method incorporates free-block accounting into the allocation data structure.



Free Space Management (3)

- **Grouping**
  - Store the addresses of *n* free blocks in the first free block.
  - The addresses of a large number of free blocks can be found quickly.
- **Counting**
  - Keep the address of the free block and the number of free contiguous blocks.
  - The length of the list becomes shorter and the count is generally greater than 1.
    - Several contiguous blocks may be allocated or freed simultaneously.

Reliability (1)

- **File system consistency**
  - File system can be left in an inconsistent state if cached blocks are not written out due to the system crash.
  - It is especially critical if some of those blocks are i-node blocks, directory blocks, or blocks containing the free list.
  - Most systems have a utility program that checks file system consistency
    - Windows: scandisk
    - UNIX: fsck

Reliability (2)

- **fsck: checking blocks**
  - Reads all the i-nodes and mark used blocks.
  - Examines the free list and mark free blocks.

Consistent

	0	1	2	3	4	5	6	7
Blocks in use	1	1	0	1	0	1	1	1
Free blocks	0	0	1	0	1	0	0	0

Missing block
-- add it to the free list

	0	1	2	3	4	5	6	7
Blocks in use	1	1	0	1	0	1	1	1
Free blocks	0	0	0	0	1	0	0	0

Duplicated free block
-- rebuild the free list

	0	1	2	3	4	5	6	7
Blocks in use	1	1	0	1	0	1	1	1
Free blocks	0	0	1	0	2	0	0	0

Duplicated data block
-- allocate a new block and copy

	0	1	2	3	4	5	6	7
Blocks in use	1	1	0	1	0	2	1	1
Free blocks	0	0	1	0	1	0	0	0

Reliability (3)

- **fsck: checking directories**
  - Recursively descends the tree from the root directory, counting the number of links for each file.
  - Compare these numbers with the link counts stored in the i-nodes.
  - Force the link count in the i-node to the actual number of directory entries.

i-node	count
1	1
5	2
12	4
…	…

i-node #5
…
count=3
…

i-node #12
…
count=2
…

Reliability (4)

- **Journaling file systems**
  - Fsck'ing takes a long time, which makes the file system restart slow in the event of system crash.
  - Record a log, or journal, of changes made to files and directories to a separate location. (preferably a separate disk).
  - If a crash occurs, the journal can be used to undo any partially completed tasks that would leave the file system in an inconsistent state.
  - IBM JFS for AIX, Linux

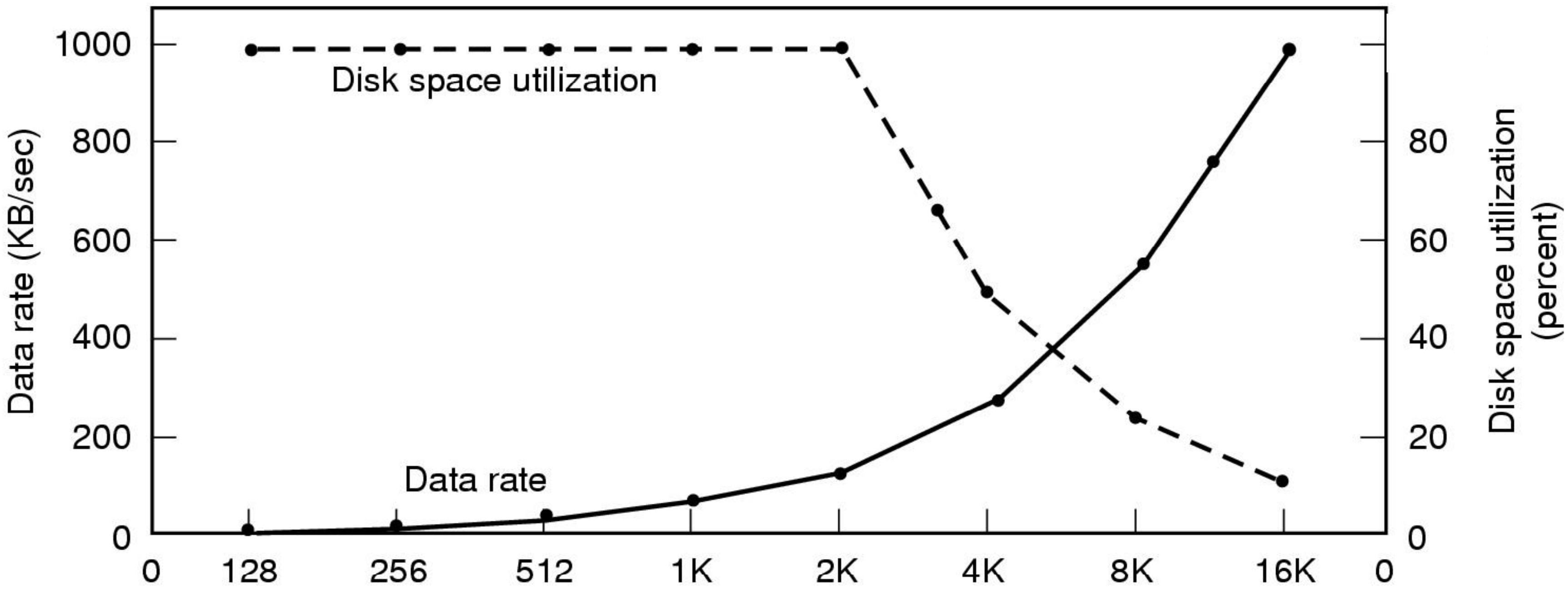
    Veritas VxFS for Solaris, HP-UX, Unixware, etc.

    SGI XFS for IRIX, Linux

    Reiserfs, ext3 for Linux

Performance (1)

- **Block size**
  - Disk block size vs. file system block size
  - The median file size in UNIX is about 1KB.



Performance (2)

- **Buffer cache**
  - Applications exhibit significant locality for reading and writing files.
  - Idea: cache file blocks in memory to capture locality in buffer cache (or buffer cache).
    - Cache is system wide, used and shared by all processes.
    - Reading from the cache makes a disk perform like memory.
    - Even a 4MB cache can be very effective.
  - Issues
    - The buffer cache competes with VM.
    - Live VM, it has limited size.
    - Need replacement algorithms again.

      (References are relatively infrequent, so it is feasible to keep all the blocks in exact LRU order)

Performance (3)

- **Read ahead**
  - File system predicts that the process will request next block.
    - File system goes ahead and requests it from the disk.
    - This can happen while the process is computing on previous block, overlapping I/O with execution.
    - When the process requests block, it will be in cache.
  - Compliments the disk cache, which also is doing read ahead.
  - Very effective for sequentially accessed files.
  - File systems try to prevent blocks from being scattered across the disk during allocation or by restructuring periodically.