Operating System Concepts
Ch. 11: File System Implementation

Silberschatz, Galvin & Gagne

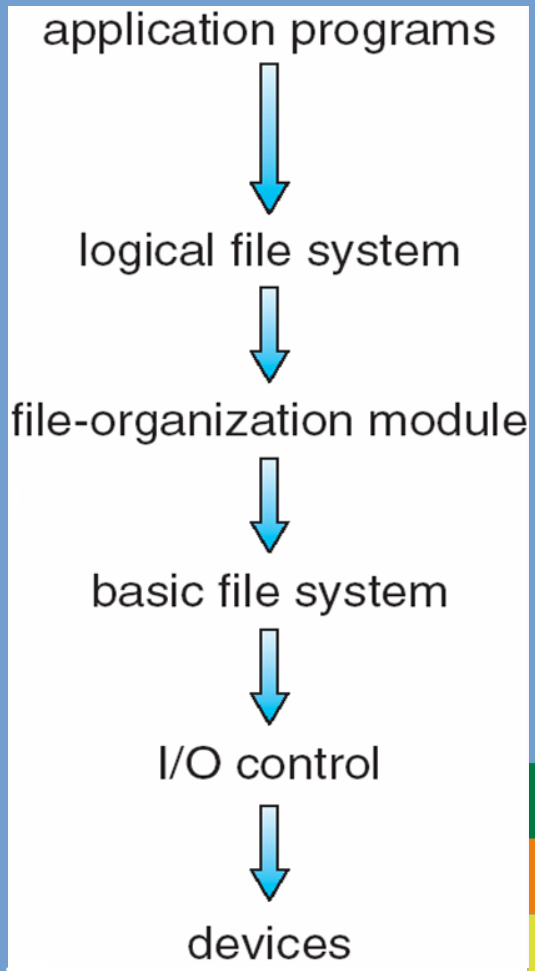
Universiteit Leiden
The Netherlands

Introduction

- When thinking about file system implementation in Operating Systems, it is important to realize the following:
  - The majority of file systems all work in terms of files and directories.
  - An OS typically supports multiple file systems.
  - A file system is independent of the (type of) disk it is stored on.
    - The same file system could be used on hard disks, USB sticks, SD cards, and so on.
- So, there is the potential to share a lot of code. File system implementations are therefore almost always layered.

Layered File System Implementation

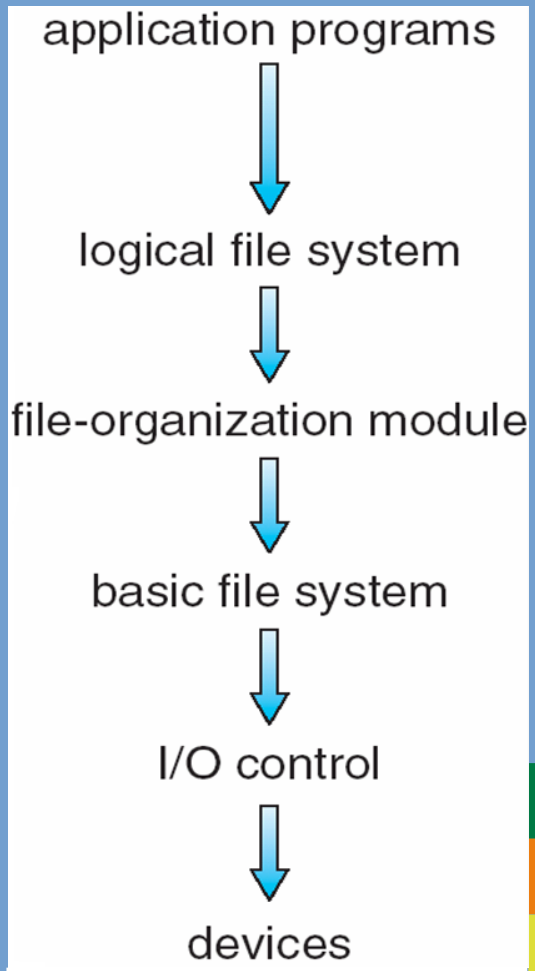
- **Logical File System:** common implementation of the concept of files and directories.
  - Directory management.
  - Metadata management.
  - Enforcement of file protection.
  - In fact, provides programming interface to application programs.
- **File organization module:** translates logical address spaces of files to physical disk addresses.
  - Free space management.
  - Disk block allocation.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Layered File System Implementation (2)

- **Basic File System:** I/O device abstraction.
  - Routes commands “retrieve block X” to the appropriate device driver.
  - Manages in-memory buffer cache of disk blocks. So ensures a memory buffer is allocated before the device driver is set to work.
- **I/O control & devices:** device driver implementation.
  - Translate abstract commands “retrieve block X” to actual low-level commands to the hardware devices.
  - Block numbers are translated to physical disk addresses if applicable.



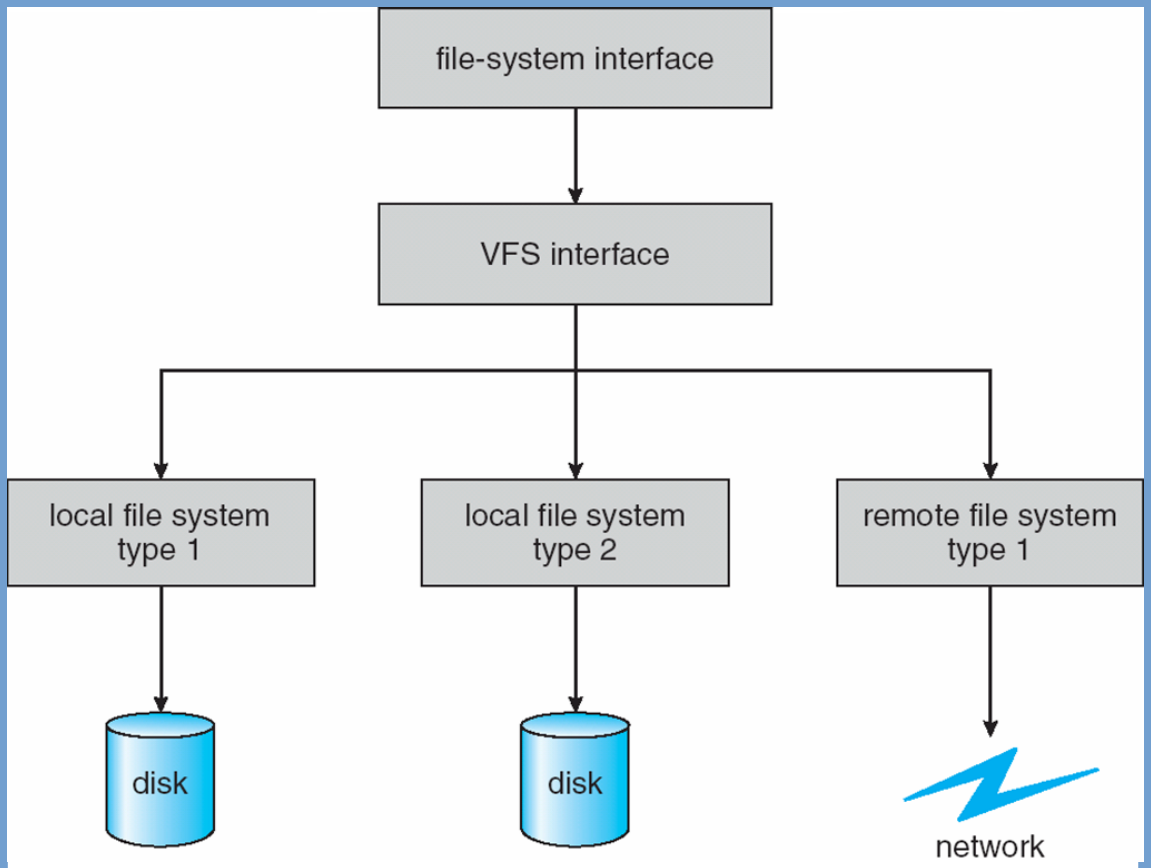
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Virtual File System (VFS)

- The Logical File System layer is often structured as a Virtual File System (VFS).
- This is an object-oriented way to implement file systems.
  - For instance, we have a “File” base class that specifies the methods to be implemented.
    - Applications (system calls) are programmed against these generic calls.
  - Different file system implementations subclass this base class and implement the methods specific to that file system.
    - Also consider network file systems here, the file might have to be transferred over the network.
  - We can now have a single tree of files, but the files may be located on different file systems.
    - Through polymorphism the correct functions are called and executed when a particular file is accessed.

Virtual File System (2)

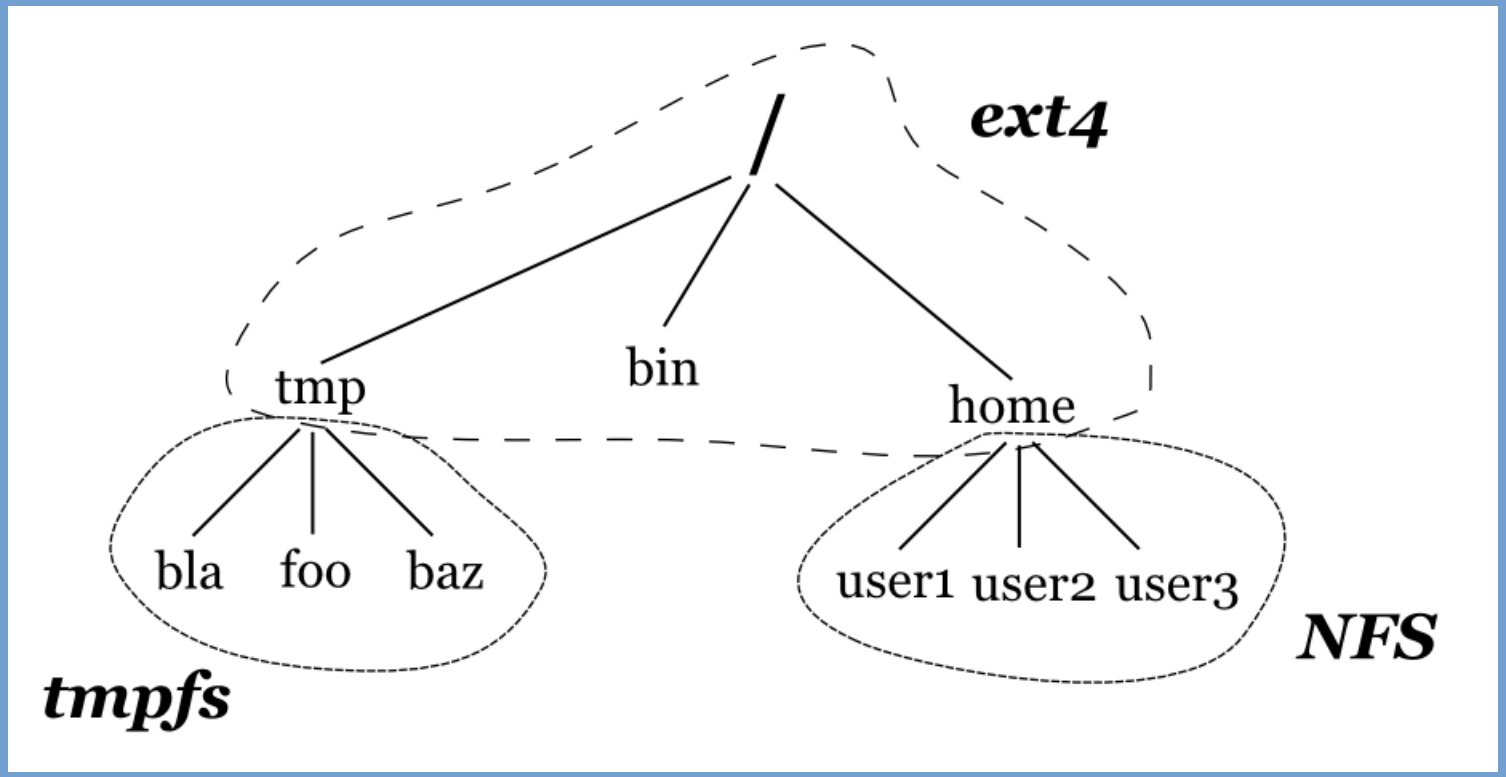
- The system calls are programmed against a generic file system interface. Through polymorphism the correct implementation of the call is found at run-time.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Virtual File System (3)

- This allows us to create a single large directory hierarchy in which multiple, different file systems may be attached (mounted):



Virtual File System (4)

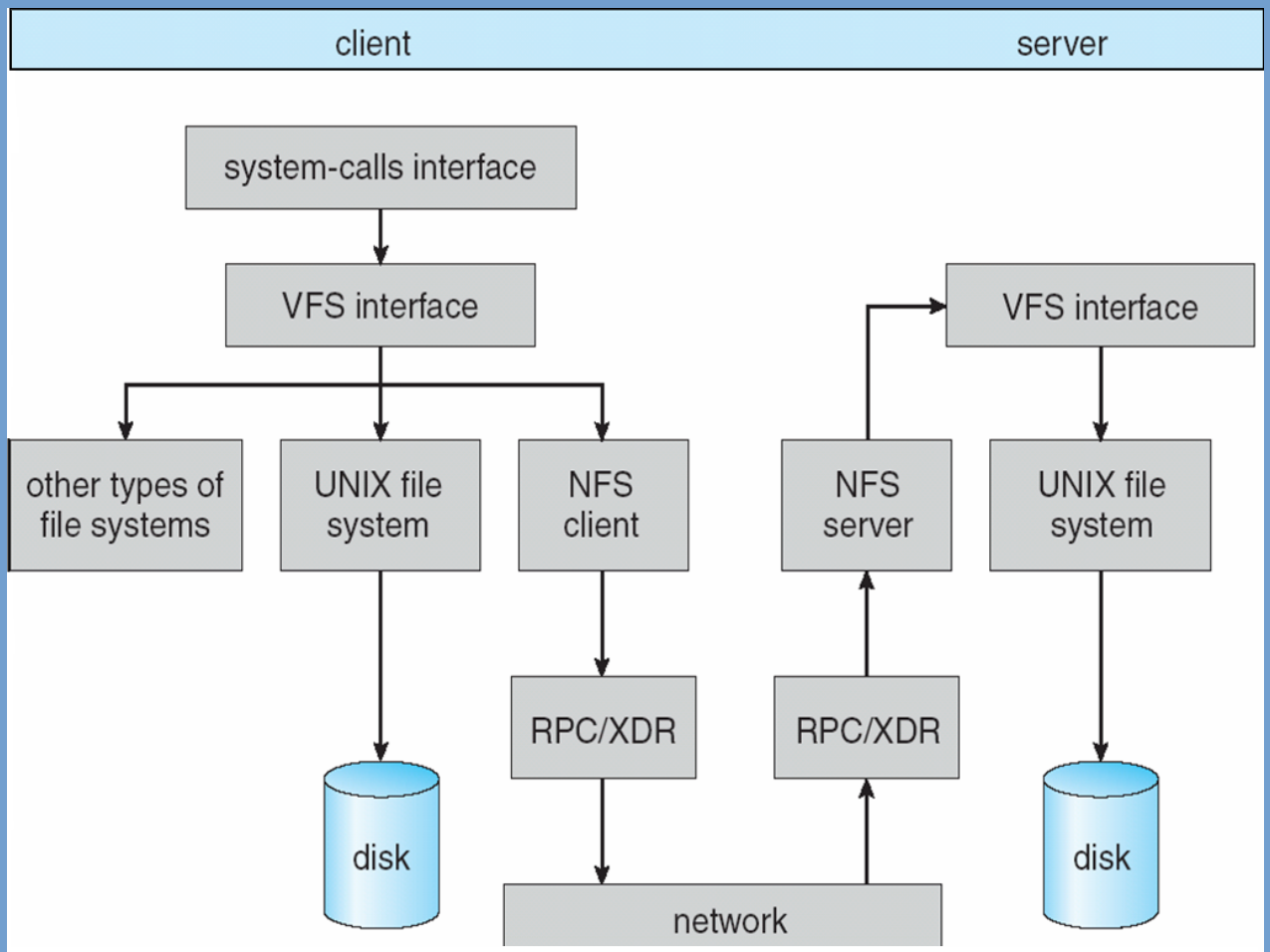
- But what about Linux? Wasn't Linux written in C and not C++??
- This is solved using plain C structure that contain function pointers.

```
From linux/include/fs.h:

struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        // ...
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
         // ...
```

The magic of abstraction …

- We can now understand (and appreciate) this schematic overview of the implementation of a client-server NFS system:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Brief tour of File System Implementation

- It appears so easy: store files and directories on a disk.
  - But there are a lot of details to get right.
  - And multiple ways to do certain things (with different implications for performance).
- We will now have a look at some common issues in file system implementations and their solutions.

Identifying and Opening File Systems

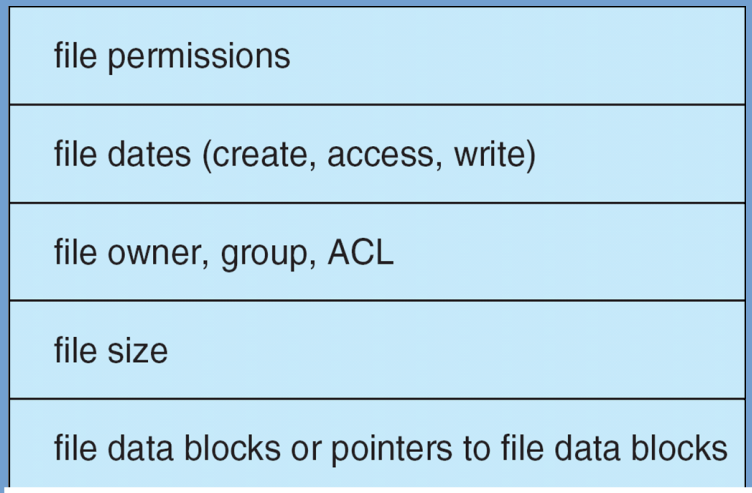
- (In the case of UNIX systems) a file system is “opened” when it is mounted
  - Only a device file and mount point are specified as part of the mount call.
  - How does the OS know that the given device does actually contain a file system of the given type?
    - File systems often have some special (“magic”) signature, this is first checked.
    - Auto-detection can be performed using a database of signatures.
    - Sometimes the file system type is explicitly provided with the mount call.
    - Then the metadata of the filesystem, its *superblock*, is read and validated.

Superblock

- Data typically in a superblock:
  - Size of the file system (in blocks).
  - The size of a block (does not have to equal physical disk block size).
  - Amount of blocks used / free.
  - Whether the file system was cleanly unmounted last time (clean system shutdown).
  - Some file system identifier.
  - A pointer to the root directory.
  - (Sometimes) a pointer to a table of file attributes (*inode table*).
- A superblock is sometimes also referred to as *volume control block*.
- Some filesystems store multiple copies of the superblock spread over the disk for the sake of redundancy.

Directory Structure & File Attributes

- The files are organized by structuring them in directories.
- To store a directory, we simply need to store a list of files & subdirectories (can be done in different ways).
  - Note that directories can be nested.
- File attributes are stored in a data structure that is generically referred to as a File Control Block (FCB).
  - Note that filename is not included, see next slide.



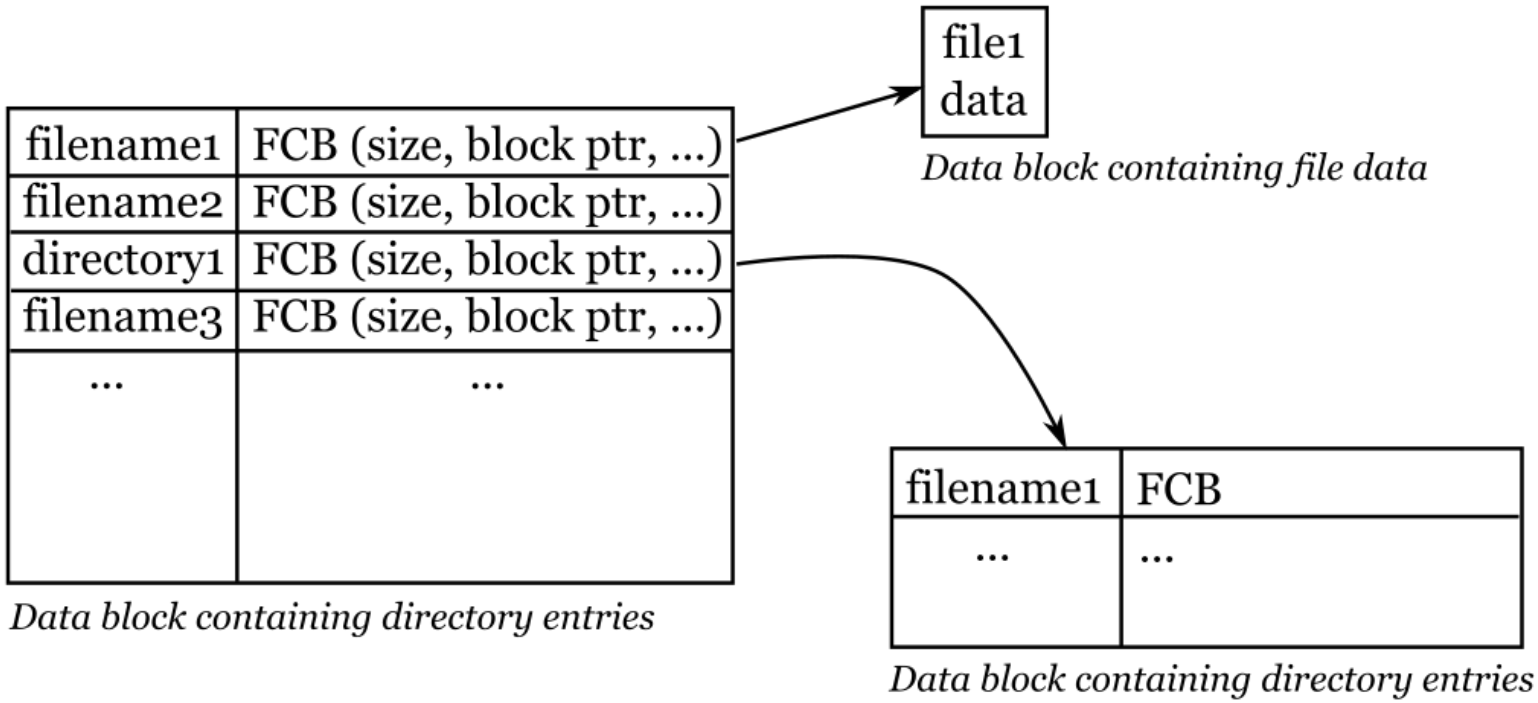
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Directory Structure & File Attributes (2)

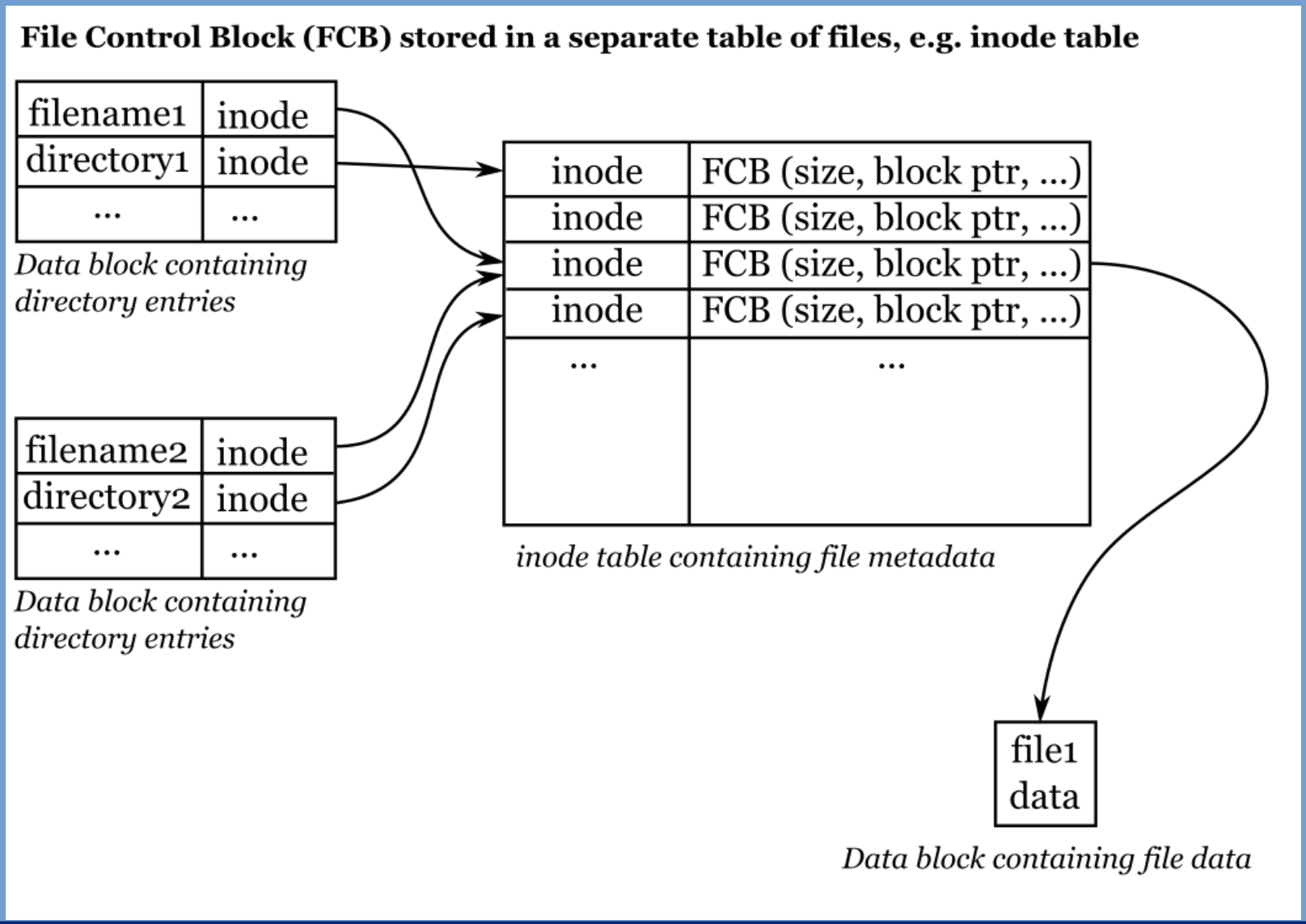
- Where is the FCB stored?
- Generally, two options:
  - (1) The FCB is stored as part of an entry in a directory's list of files. So in this list the filename and associated FCB are stored.
  - (2) The FCB is stored separately in a table.
    - Each file is identified by a number (often *inode*).
    - Directories store pairs of filename and inode. The inode then points at the FCB in the table.
    - Multiple filenames (in different directories) may point at the same inode number (and thus at the same attributes & file contents).

FCB in directory entries

File Control Block (FCB) stored as part of directory entries



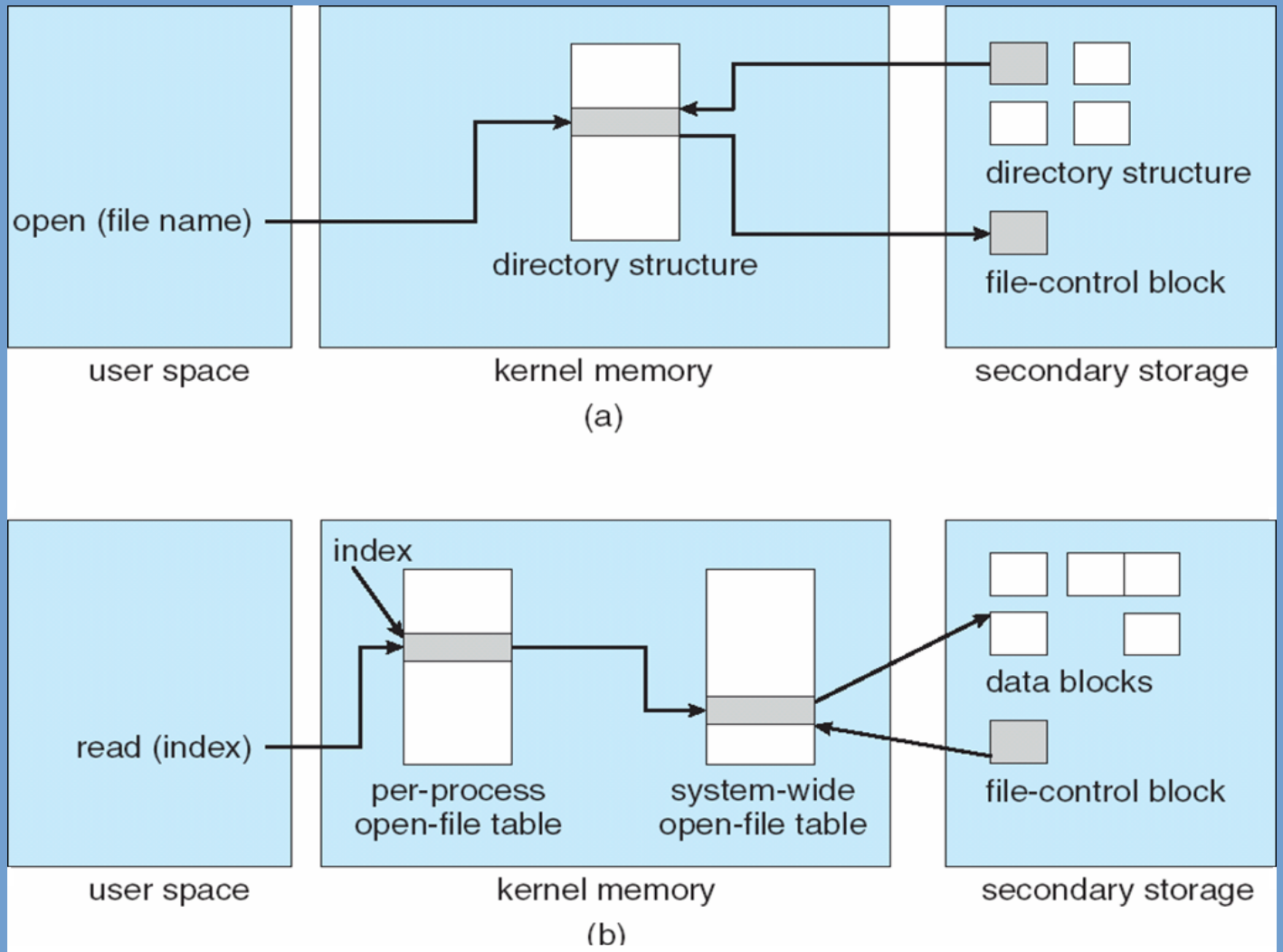
FCB in inode table



Directory Structure & File Attributes (3)

- Directory structure and file attributes are gradually brought into memory as more files are accessed.
  - The file system tree is cached in memory for performance reasons.
  - When a file is opened, its FCB is transferred to memory. The FCB is updated on disk when the file is closed.
  - FCB is stored in a system-wide table of open files. There is also a table of open files **per** process. This latter table includes the file R/W pointer.
    - Naturally, processes have individual file R/W pointers.

Directory Structure & File Attributes (4)



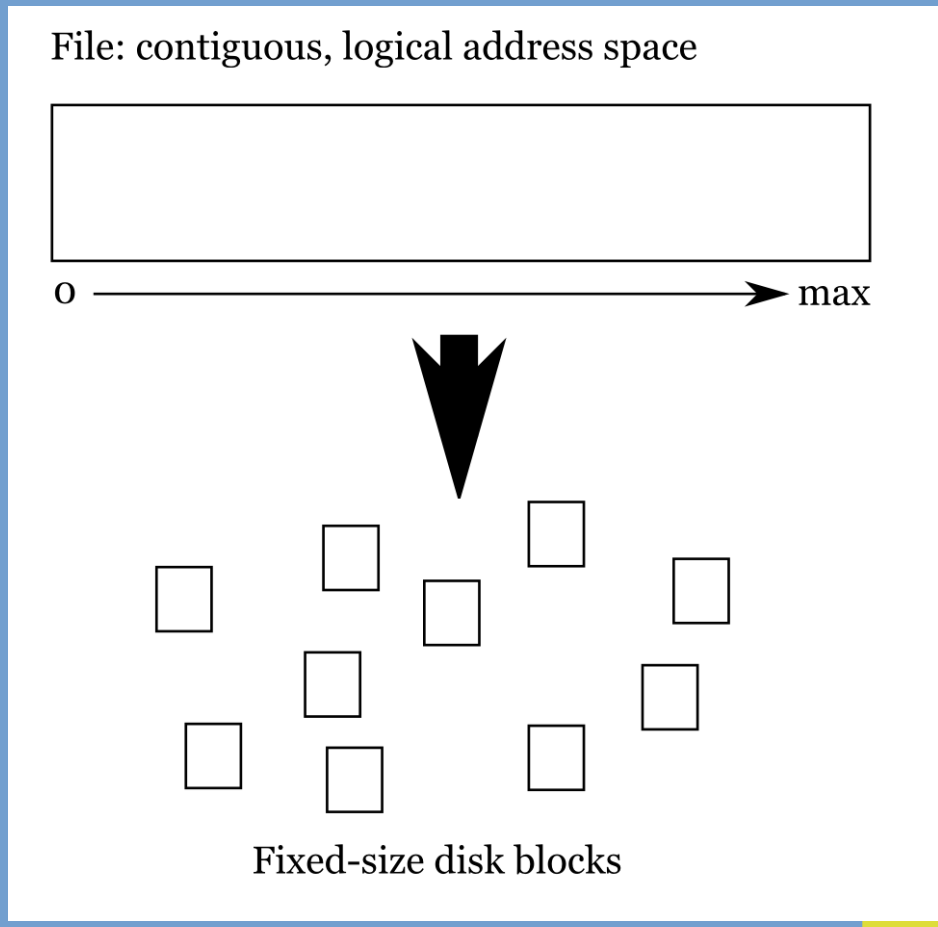
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Directory Implementation

- Now how to store the contents of a directory?
- We already saw one option, a simple linear list.
  - Advantage: easy to implement, read & manage.
  - Disadvantage: search (a common operation) requires linear scan. Does not scale to (very) large directories.
    - Could cache directory contents in memory (VFS tree) and maintain tree-based index.
- Another option: use hash tables to reduce seek time. We need to be able to deal with collisions however.
- Some file systems (example: XFS) implement B+ trees on disk.

Disk Block Allocation

- Our next problem concerns the storage of the mapping of the logical address space of a file to physical disk blocks.
  - Physical disk blocks typically 512 or 4096 bytes in size.
- The logical address space is contiguous, but the data does not have to be contiguously stored on physical disk.
- How do we register what is stored where?
  - (We cannot keep this in memory, we must store it on disk as well.)

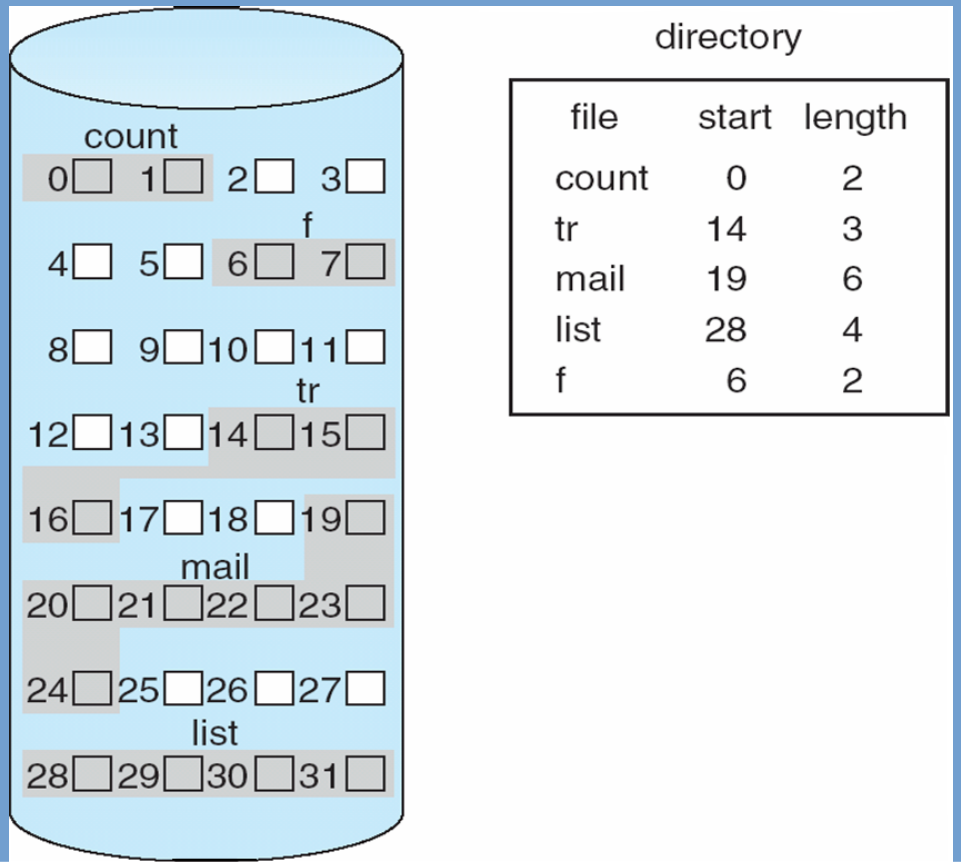


Contiguous Allocation

- Of course, we can choose to require the physical storage to be contiguous.
- In this case, we only have to register start block and size (in number of blocks) in the file's attributes.
  - Trivial implementation.
- The problems start to arise when data is appended to files in the future. There may be no space the grow.
  - Find another hole that is large enough. Rewrite the entire file there.
  - Searching for holes may become time consuming.
  - As with contiguous memory allocation: external fragmentation will occur. Compaction is expensive and disk contents need to be moved to other disk areas. Can be done in the background.

Contiguous Allocation

- In an image:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

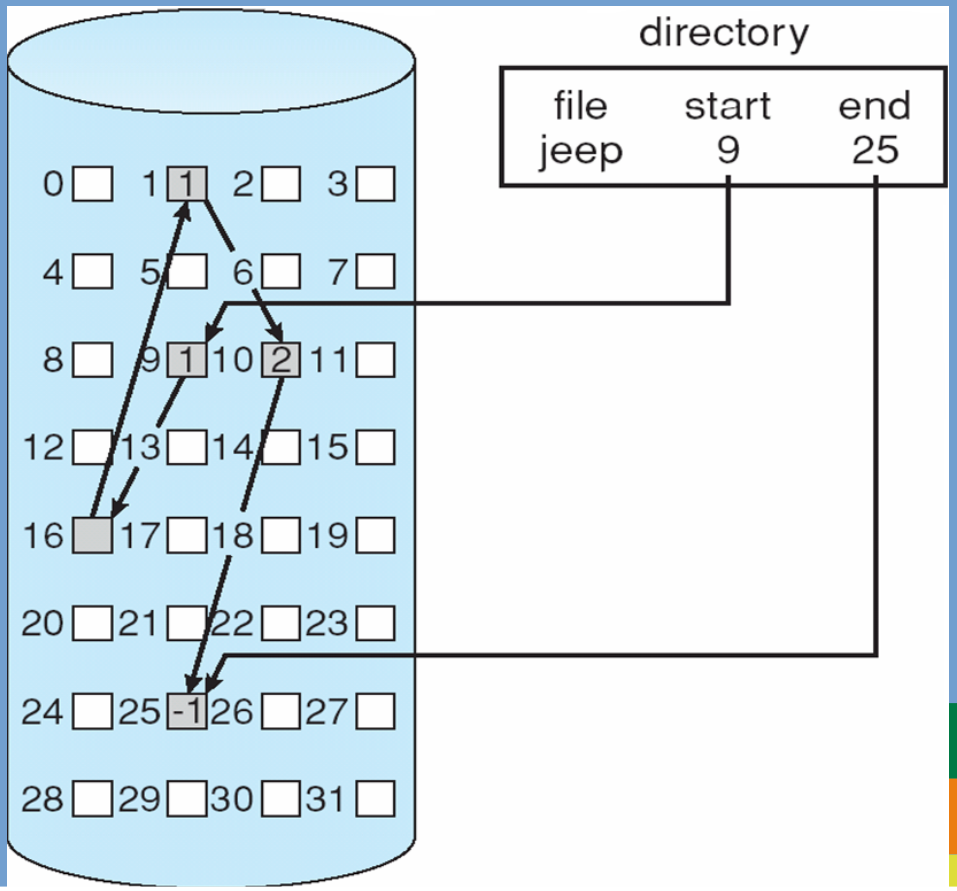
Linked Allocation

- Avoid contiguous allocation of disk areas, instead allocate non-contiguous blocks and stitch them together.
  - This eliminates the problem of external fragmentation.
  - And finding a block to extend a file is easy: any free block will suffice (when ignoring performance).
- A linked list is used to create a sequence of blocks that contain the contents of the file.
  - The file ends at the NULL (or EOF) pointer.
- Sometimes clusters of blocks are used to increase efficiency (less pointer seeks necessary).
  - The file system then actually operates on top of “virtual blocks” with a larger block size than the physical blocks.
  - This does increase internal fragmentation (cf. paging & page size).

Linked Allocation (2)

Where to store the pointers to the next block?

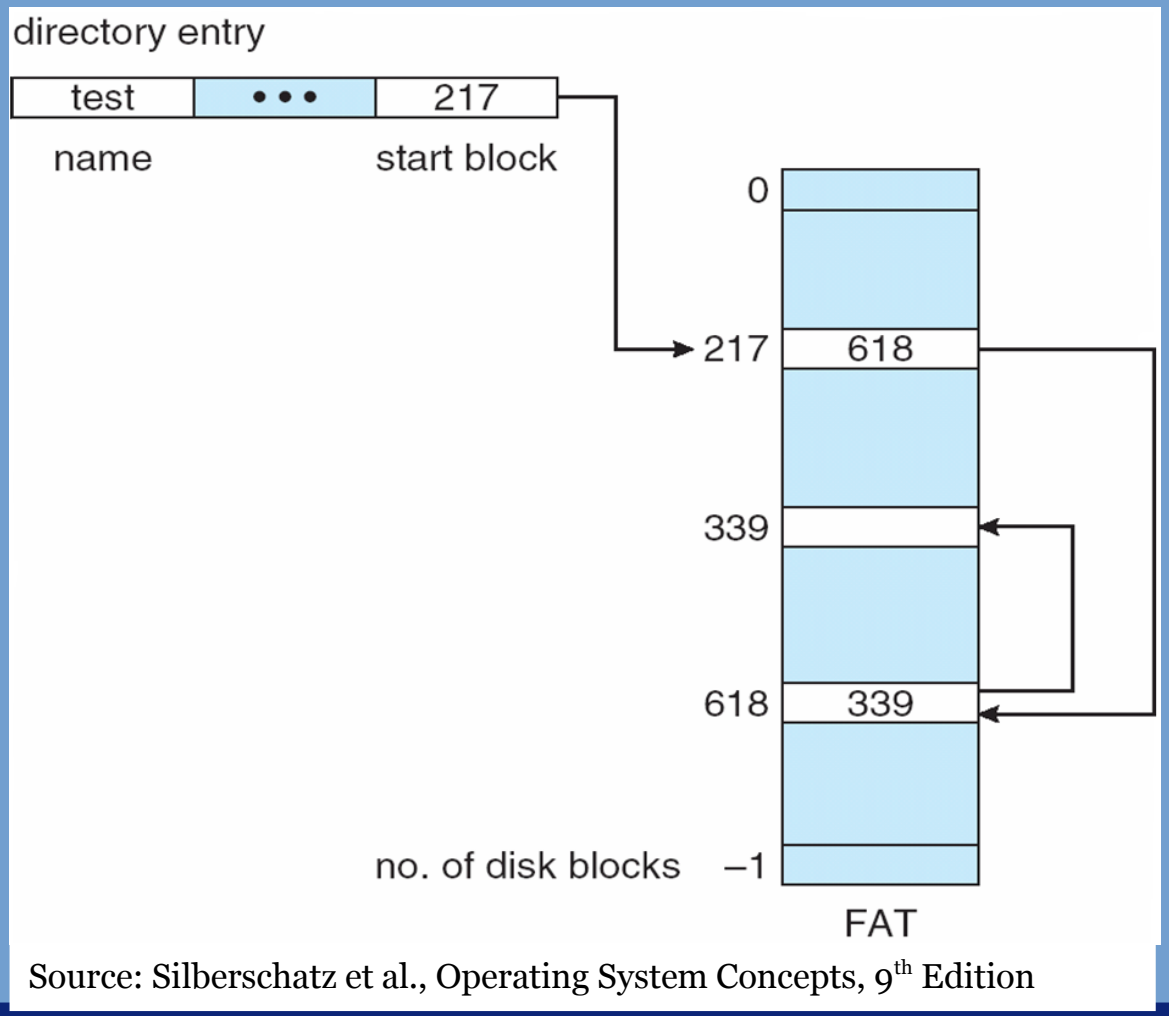
- This can be done within the blocks themselves, for example in the first or last 4 bytes.
- Or, in a separate table (example on next slide).
  - FAT (File Allocation Table) model.
  - Big advantage: all pointers are stored on disk contiguously and this area can be easily cached when file system is mounted.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Linked Allocation (3)

- Example of File Allocation Table model:



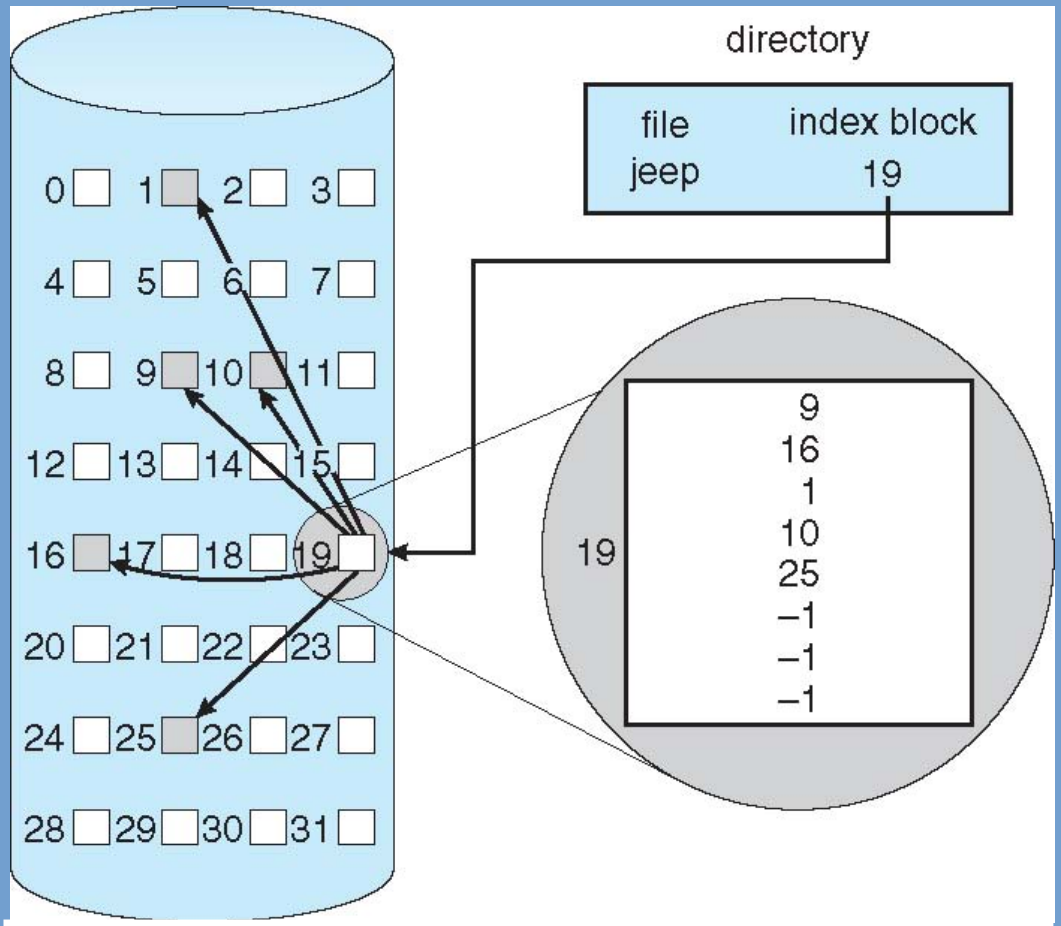
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Indexed Allocation

- A problem that remains with linked allocation is random access. If you want to access the middle of a file, you need to follow all the pointers.
- Indexed allocation tries to alleviate this.
  - In this model, each file has one or more *index blocks* that contain pointers to the actual data blocks.
  - You could store multiple pointers to index blocks in the FCB.
  - You could create a linked list of index blocks (but this results in the random access problem again).

Simple Indexed Allocation

- An example of indexed allocation with a single index block:



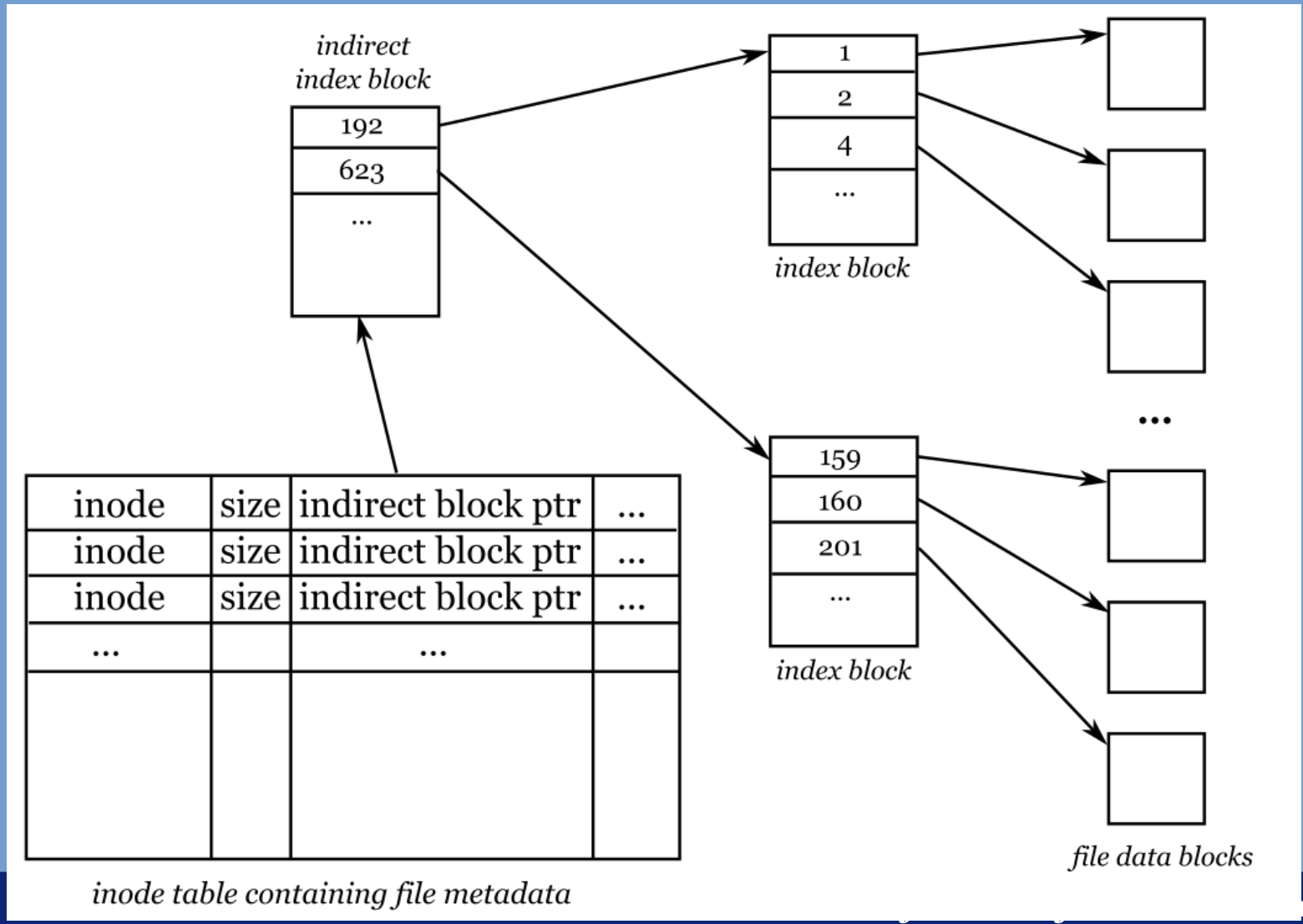
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Extending Indexed Allocation

- We cannot store a long list of index block pointers within the FCB, this would unnecessarily enlarge the FCB (in particular for small files).
- A linked list is also not an option.
- A solution that is in wide-spread use is the multi-level index (again cf. page tables).
  - The first index block contains pointers to other index blocks.
  - In turn, these index blocks contain pointers to data blocks.
    - (Or again to index blocks, the implementor decides the depth of the hierarchy and as such the maximum supported file size).

Extending Indexed Allocation (2)

- Schematic of example multi-level index to file contents.

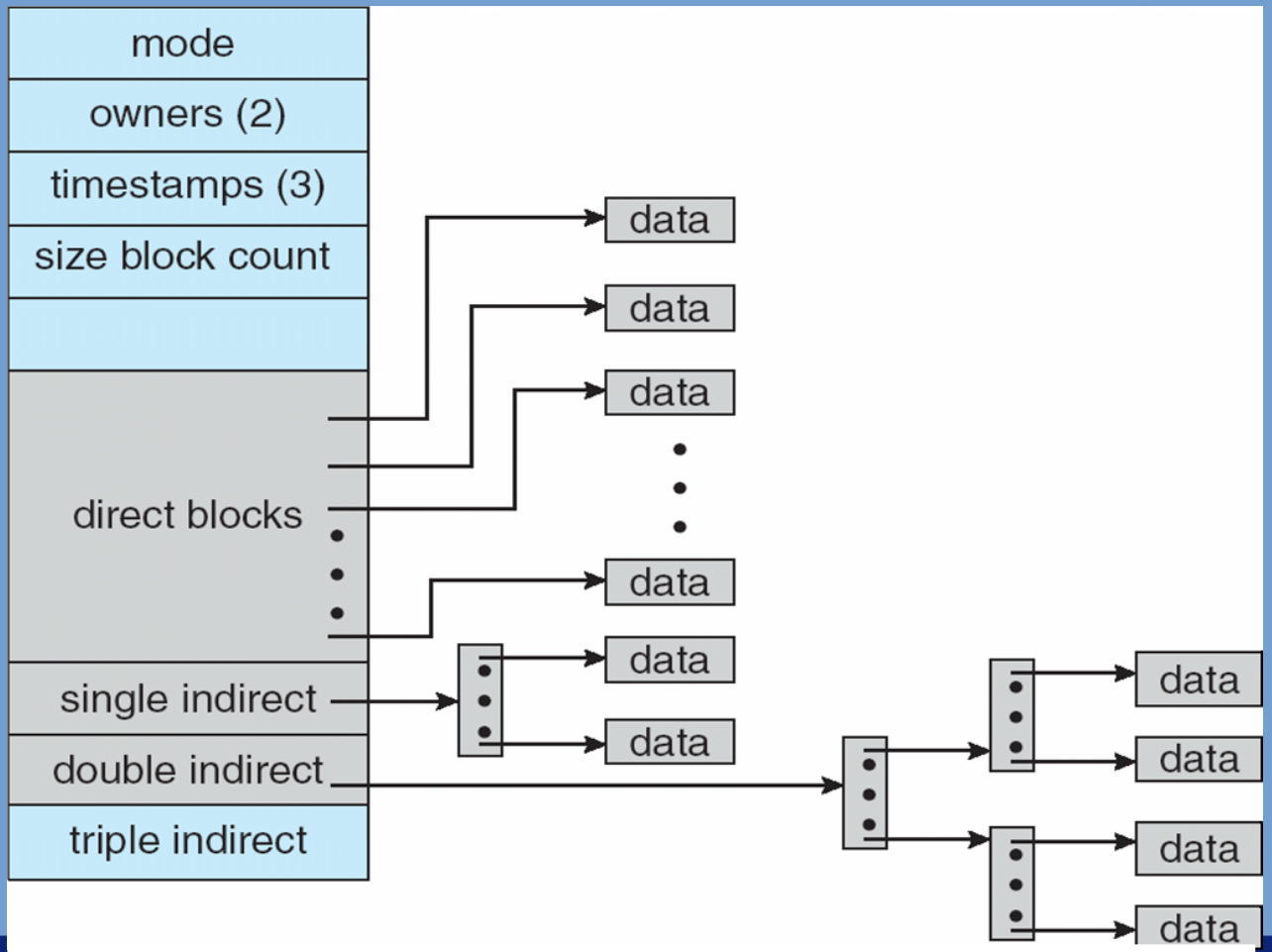


Extending Indexed Allocation (3)

- Disadvantage: also for small files multiple seeks through the hierarchy are needed.
- Solution: Combined Scheme.
  - Store different pointers in the FCB:
    - 2-3 pointers to data blocks (*direct blocks*): great for small files.
    - 1 pointer to an index block, which in turn contains pointers to data blocks (*single indirect block*).
    - 1+ pointers to index blocks, which contain pointers to index blocks, which in turn contain pointers to data blocks (*double indirect block*).
    - Could even implement triple indirect block and further ...
    - A picture helps, see next slide.

Extending Indexed Allocation (4)

- Example FCB (inode) of UNIX UFS, which uses the combined scheme of indexed allocation:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Performance of Allocation Methods

- Different allocation methods show different performance characteristics.
  - We already reasoned that random access performance with linked allocation is problematic.
- Contiguous allocation works well for sequential and random access.
  - Less disk seeks.
- A problem with indexed allocation is that it may require multiple disk reads to locate the data block that we are interested in.
  - Cf. page table walk.
  - The different index blocks may be scattered over the disk.

Free Space Management

- On to our next problem: how do we know what blocks of the disk are still available?
  - It is clear that scanning 4+ TB disks for “empty” blocks is not an option.
  - On the other hand, what is an “empty” block? A block filled with solely zeroes could be allocated to a file!
  - We need a separate data structure in which blocks must be marked as taken or available.
    - *Free-space list.*

Free Space Management (2)

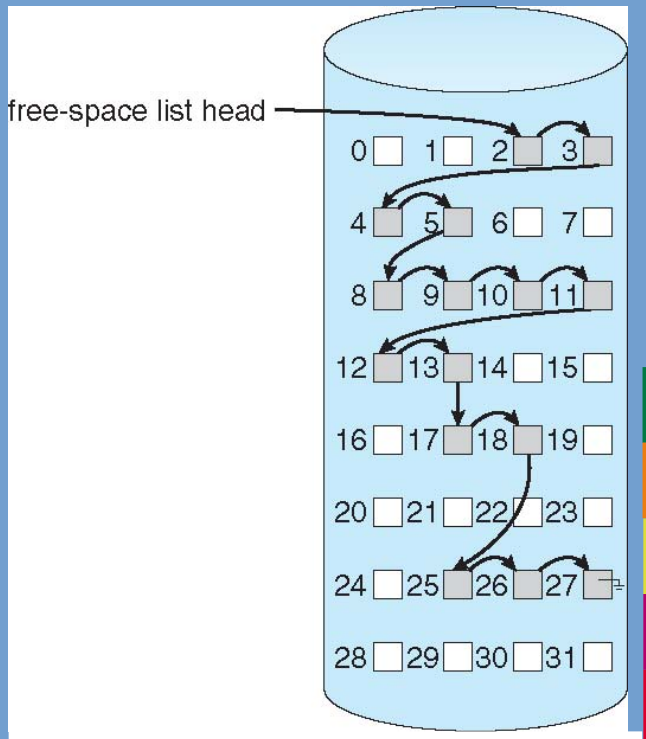
- Two simple schemes:
  - *bit vector* or *bitmap*. We store a bitmap on disk that covers all blocks on the disk. 1 or 0 signifies allocated or free.

    For example, Linux ext{2,3,4} file systems use block bitmaps.

    Becomes expensive for large disks with small blocks.

  - *Linked list*. Maintain a linked list of free blocks (depicted on the right).

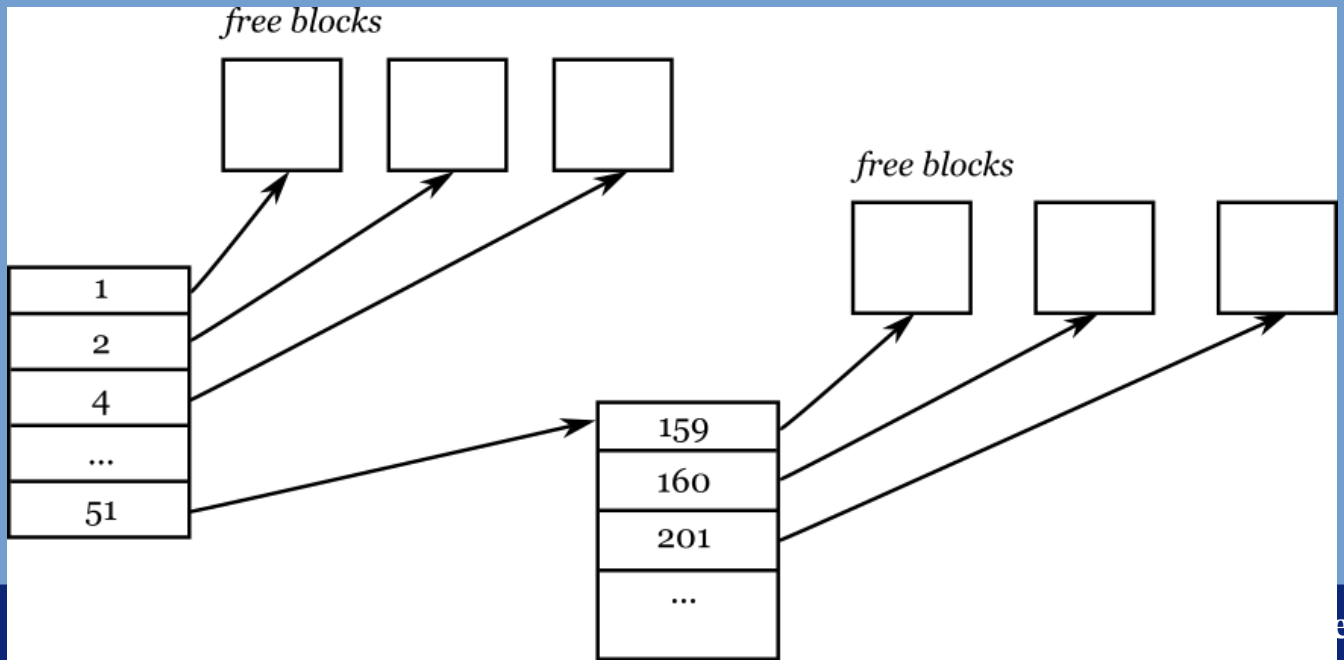
    Expensive to search for sequences of contiguous blocks.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Free Space Management (3)

- Modifications of free-list approach:
  - Grouping
    - Consider *n* free blocks, in the first free block store the addresses of these *n* free blocks.
    - The last of these *n* addresses is in fact the address of a block that again contains *n* addresses of free blocks (the next group).
    - In this scheme, the addresses of free blocks are grouped, allowing for addresses of a larger number of free blocks to be found more rapidly.



Free Space Management (4)

- Modifications of free-list approach:
  - Counting
    - Often contiguous blocks are allocated/released, in particular when clustering is used.
    - So why maintain addresses of individual free blocks?
    - With the counting method, tuples (*block address*, *n*) are maintained with the address of the first free block of a group of *n* contiguous blocks.

File System Performance

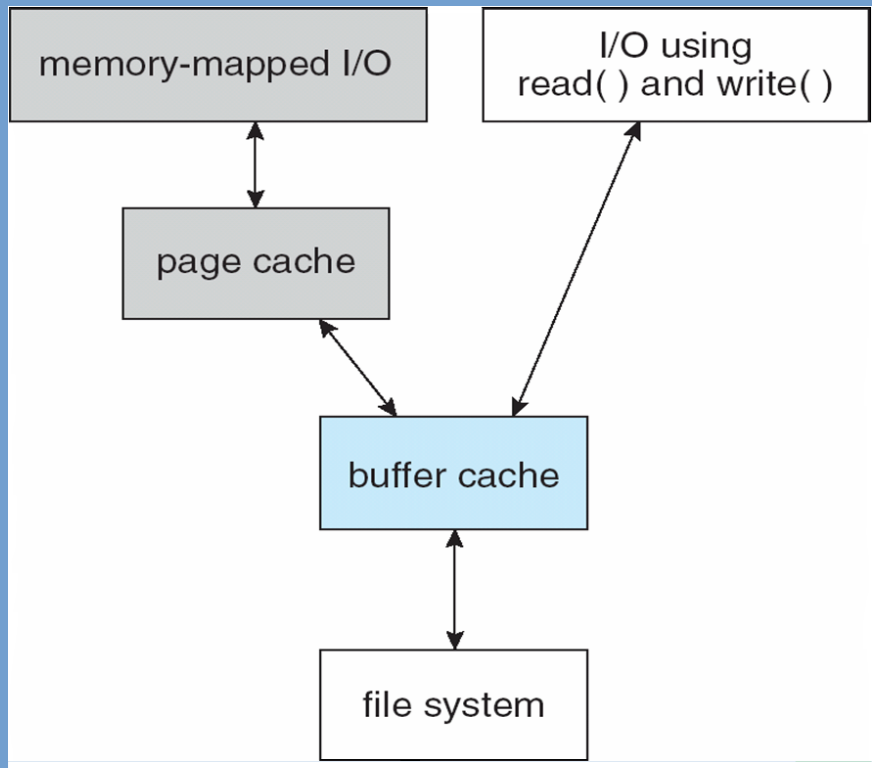
- Several design decisions in the implementation of file systems directly influence the file system's performance:
  - Disk allocation algorithm that is used
  - On-disk data structure for maintaining directories
  - How much data is stored in a file entry and the time necessary to maintain this
    - Some systems maintain access time entries. Expensive! Often disabled.
  - Method of allocation metadata data structures.
  - Are the data structures used of fixed-size or varying in size?
    - For fixed-size data structures the faster pool allocation can be used.

Disk Subsystem Performance

- Some techniques used to improve disk subsystem performance:
  - Buffering of disk blocks, keep frequently used disk blocks in main memory. *Buffer cache* or *page cache* (see also later).
  - *Asynchronous writes*. Tell the application a write has completed when this is actually not the case. Buffer the written data and write to disk later in time.
    - Some applications do require *synchronous* writes. More expensive (in time), but safer. When the write system call returns, application is guaranteed data was written to disk.
  - *Read-ahead*. When OS detects that an application simply reads through a file in sequential order, read next blocks from disk before they are explicitly requested.
    - Similarly, blocks that have been read already can immediately be purged from the buffer cache (*free-behind*).
  - What would be faster? Reads or writes? Why?

Page Cache

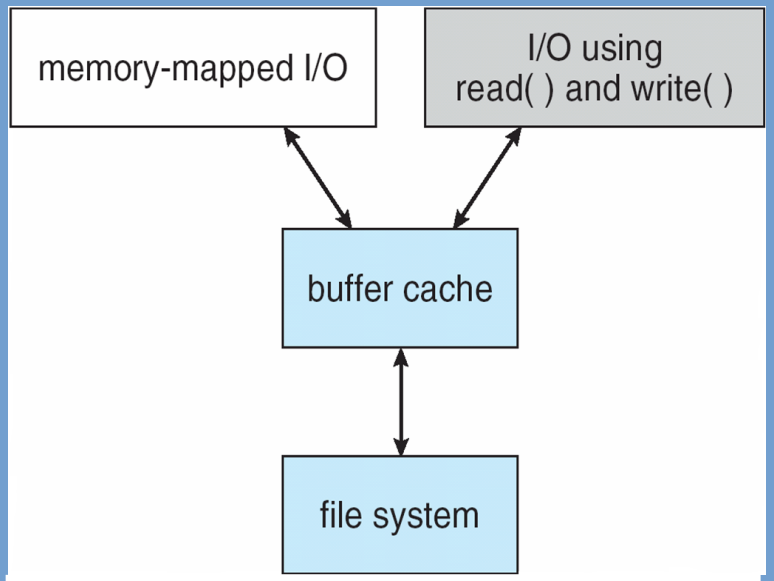
- Buffer cache was originally based on the caching of individual disk blocks (512 bytes in case).
- Recall our discussion on memory-mapped I/O. Here we need to cache pages (4K or larger) in memory.
- A naive implementation will read a “page of blocks” from disk and cache this data both in buffer cache and in memory-map cache.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Page Cache (2)

- To alleviate this a unified buffer cache was proposed and is now implemented by most systems.
  - The buffer cache is eliminated and all disk block caching is done in groups of blocks equal to the page size.
  - Ordinary file system I/O is routed through the unified buffer cache.
  - Double buffer is eliminated.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

- Need to be careful with page replacement algorithms.
  - High I/O rates may take away pages from ordinary processes to expand the page cache.
  - These ordinary processes might now run out of memory and start swapping/paging (I/O load further increases).

File System Corruption

- File systems data structures (or file contents) may be corrupted.
  - Bit flip due to broken DRAM, or disk fault.
  - Problems cannot be detected in all cases. If you want to detect this for file contents, you need to store checksums of disk blocks within file system metadata (done by ZFS).
- Data structures can be checked by a consistency checker.
  - Or file system checker (often abbreviated “fsck”).
  - Such a checker visits all file system metadata (traverses entire directory structure) and checks whether this is sound.
  - Detected errors can in many cases be fixed.

File System Corruption (2)

- Pitfall of file system checkers: they may fail to repair detected errors.
  - The file system is now left in a partly inconsistent state. Data may be lost!
- How to protect against this?
- *Back up* data to another storage device, such as magnetic tape, USB drive, external hard drive.
  - Never copy file system one-on-one (so at partition level), this will also copy potentially corrupted file system data structures.
  - So, back ups are not just done to protect against failed hard drives!
- Given a back up, lost files and/or disks can be restored using a restore or recovery procedure.

File System Corruption (3)

- Another problem of file system checkers: they are REALLY slow.
  - It may already take quite some time for 32 GB hard drives, imagine for 4+ TB drives.
  - A system crash meant a mandatory file system check to detect and repair inconsistencies: long boot time.
- Early 2000's a solution was proposed: journaling file systems (or log-structured file systems).
  - Main idea: maintain a log of modifications done to file system metadata.

File System Corruption (4)

Journaling File Systems:

- Each metadata update becomes a transaction stored in a log file.
- A system call will only return after the log entry is guaranteed on disk.
- At some later point in time, the actual file system data structure will be modified.
  - When completed, the entry in the log is marked as processed.
- In case of a system crash, at the next boot the log is checked ("journal replay").
  - All non-processed transactions are now carried out.
  - Smaller chance of inconsistencies after crash, no mandatory consistency check on boot after crash so much faster recovery time.
  - However, a consistency check is still strongly recommended to be performed once or twice a year.

End of Chapter 11.