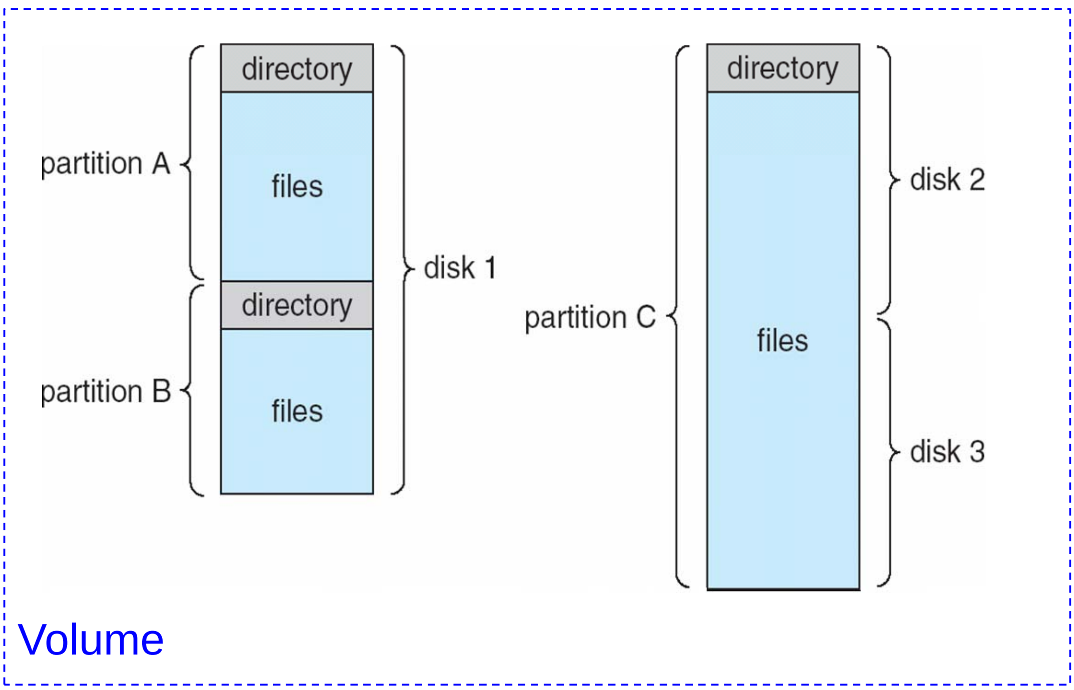
Chapter 5.11: Implementing File Systems

- File-Systems Structure
- File Implementation
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
- Directory Implementation
- Buffering
- Log-Structured Files Systems

A Typical File-system Organization



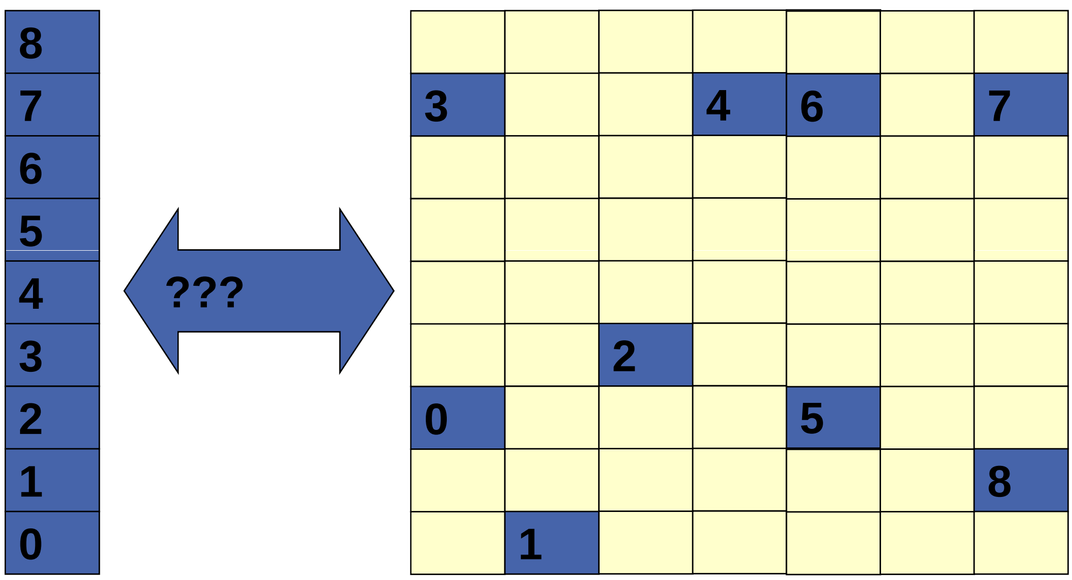
© 2009 Karlsruhe Institute of Technology, System Architecture Group
Frank Bellosa, Gerd Liefländer, Silberschatz et al.

Disk Structure

- Disk can be subdivided into partitions
- Disks, partitions[1] can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system (FS)
- Entity containing a FS known as a volume
- Each volume containing a FS also tracks that FS's info in device directory or volume table of contents
- As well as general-purpose FSs there are many special-purpose FSs, frequently all within the same operating system or computer

[1]Partitions also known as minidisks, slices

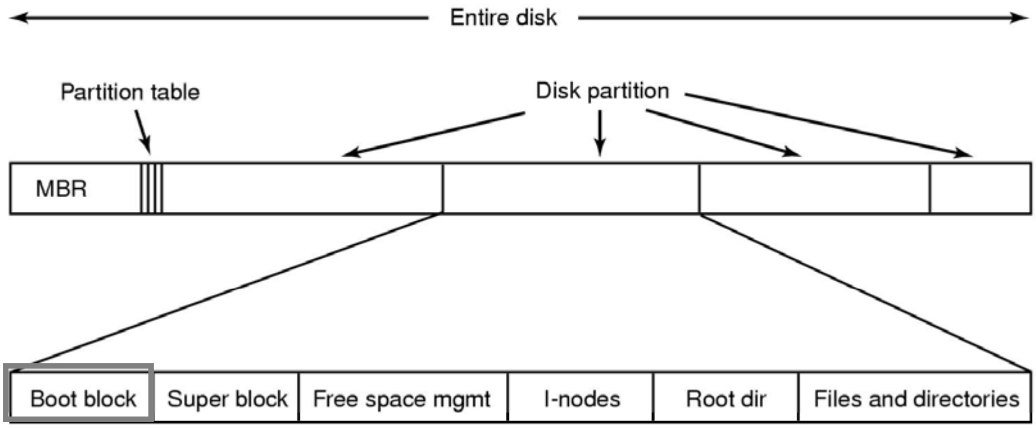
Implementing Files



File with a set of logical file blocks (records)

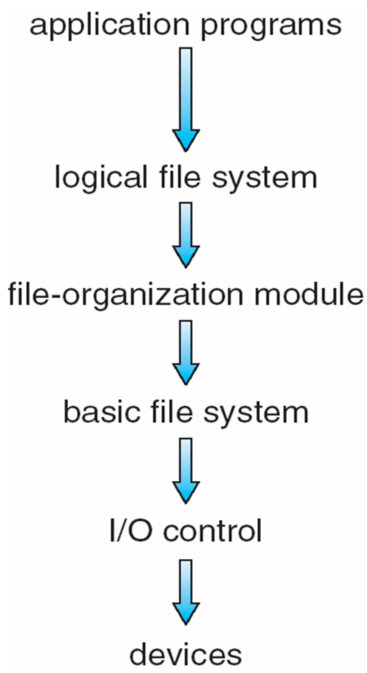
Disk with allocated and free physical disk blocks

Implementing a FS on Disk



- Possible FS layout per partition
- *Sector 0* of disk = MBR
  - Boot info (if PC is booting, BIOS reads in and executes MBR)
  - Disk partition info
- Sector 0 of partition is volumen boot record

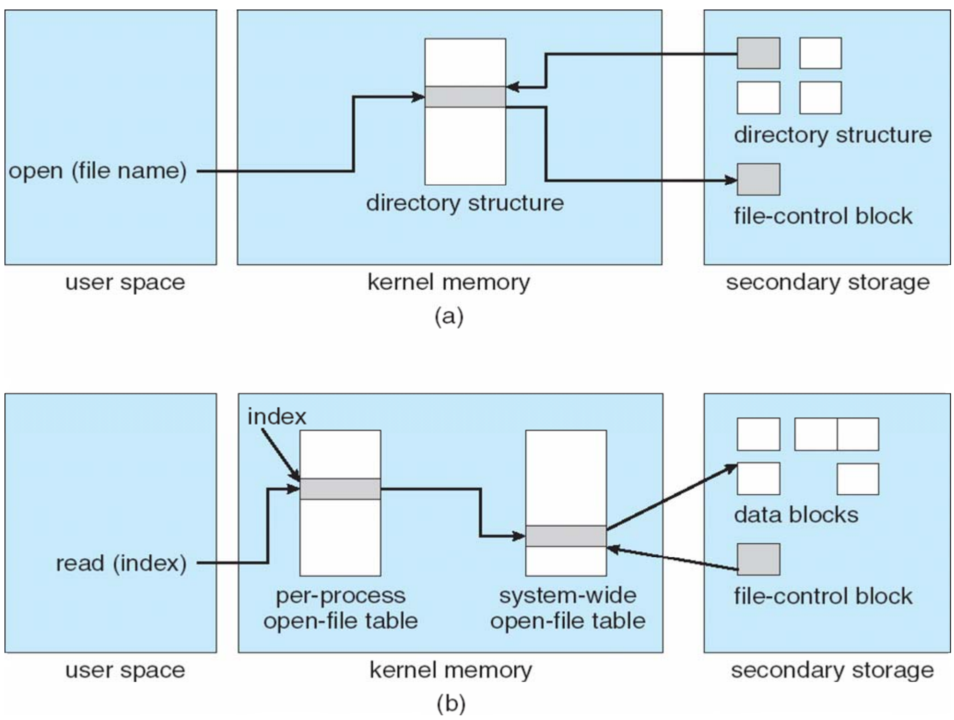
Layered File System



A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

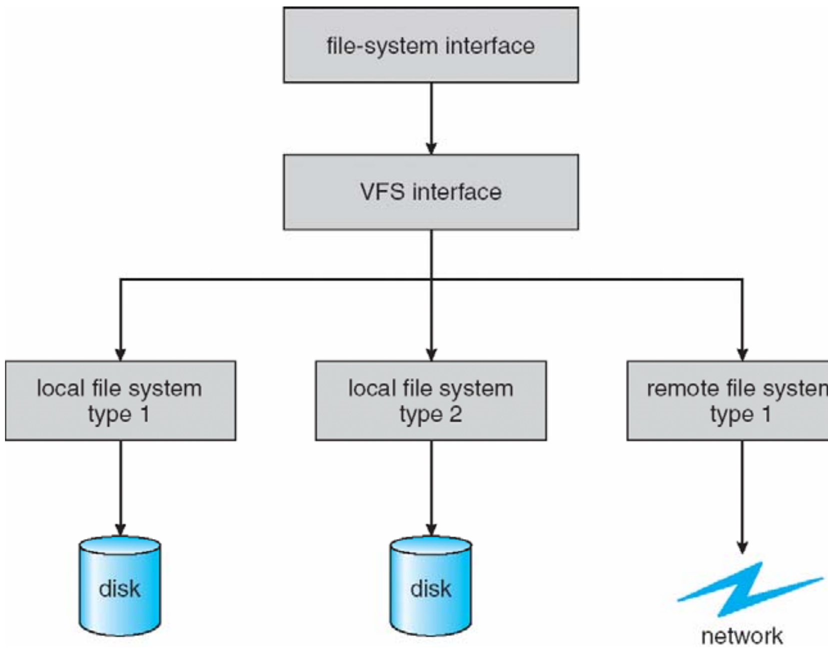
In-Memory File System Structures



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



© 2009 Karlsruhe Institute of Technology, System Architecture Group
Frank Bellosa, Gerd Liefländer, Silberschatz et al.
Betriebssysteme WS 09/10
5 Storage Management

Implementing Files

- FS must keep track of some meta data
  - *Which logical block belongs to which file?*
  - *In what order are the blocks form the file?*
  - *Which blocks are free for the next allocation?*
- Given a logical region of a file, the FS must identify the corresponding block(s) on disk
  - Needed meta data stored in
    - File allocation table (FAT)
    - Directory
    - Inode
- Creating (and updating) files might imply allocating new blocks (and modifying old blocks) on the disk

Allocation Policies

- Preallocation:
  - Need to know maximum size of a file at creation time (in some cases no problem, e.g. file copy etc.)
  - Difficult to reliably estimate maximum size of a file
  - Users tend to overestimate file size, just to avoid running out of space
- Dynamic allocation:
  - Allocate in pieces as needed

Fragment Size *

- Extremes:
  - Fragment size = length of file
  - Fragment size = smallest disk block size (sector size)
- Tradeoffs:
  - Contiguity ⇒ speedup for sequential accesses
  - Many small fragments ⇒ larger tables needed to manage free storage management as well as to support access to files
  - Larger fragments help to improve data transfer
  - Fixed-size fragments simplify reallocation of space
  - Variable-size fragments minimize internal fragmentation, but can lead to external fragmentation

*see page size discussion

Implementing Files

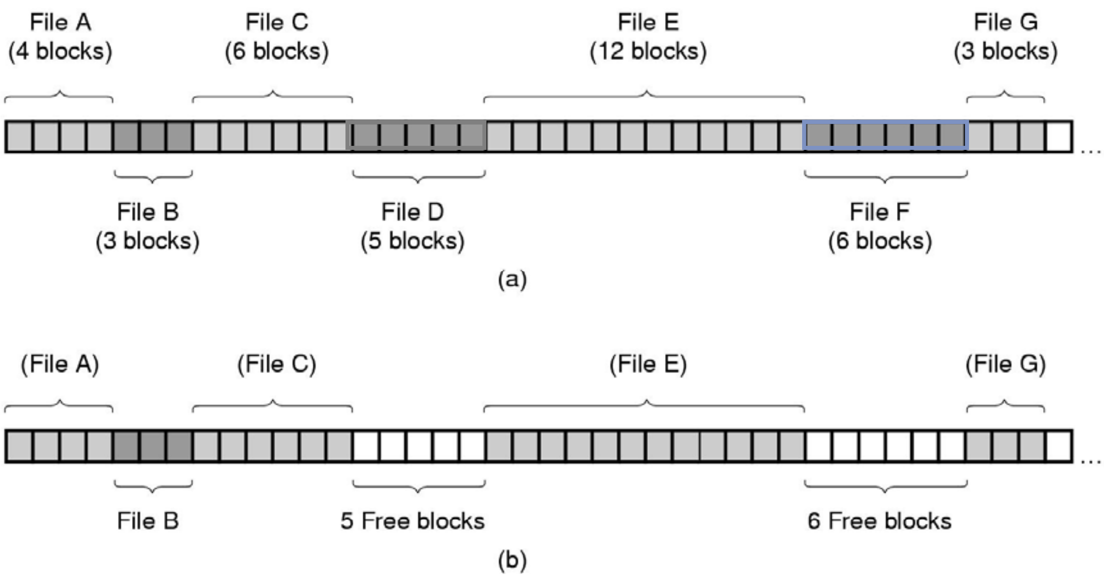
- 3 ways of allocating space for files:
  - contiguous
  - chained
  - indexed
    - fixed block fragments
    - variable block fragments

Contiguous Allocation

- Array of N contiguous logical blocks reserved per file (to be created)
- Minimum meta data per entry in FAT/directory
  - Starting block address
  - N
- *What is a good default value for N?*
- *What to do with an application that needs more than N blocks?*
- Discussion similar to ideal page size
  - Internal fragmentation
  - External fragmentation

⇒ scattered disk

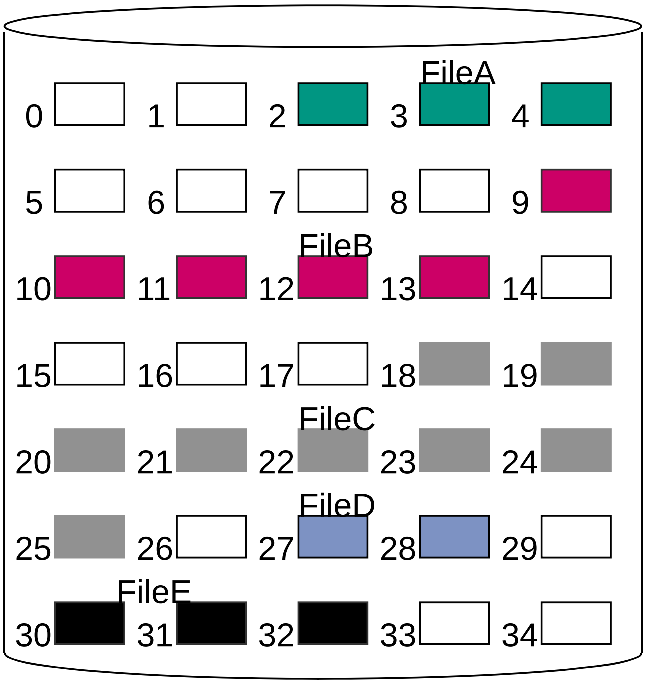
Scattered Disk



(a) Contiguous allocation of disk space for 7 files

(b) State of the disk after files D and F have been removed

Contiguous File Allocation



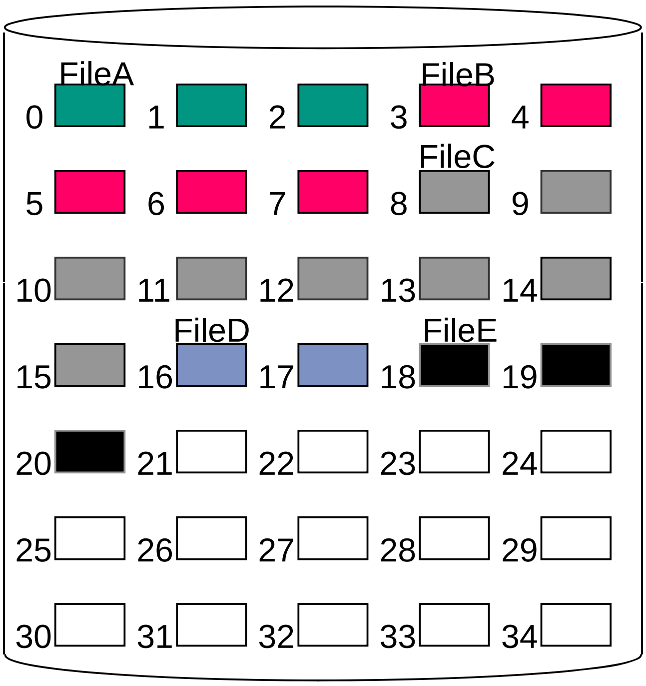
File Allocation Table

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	27	2
FileE	30	3

Remark: To overcome external fragmentation

⇒ periodic compaction

Contiguous File Allocation (After Compaction)



File Allocation Table

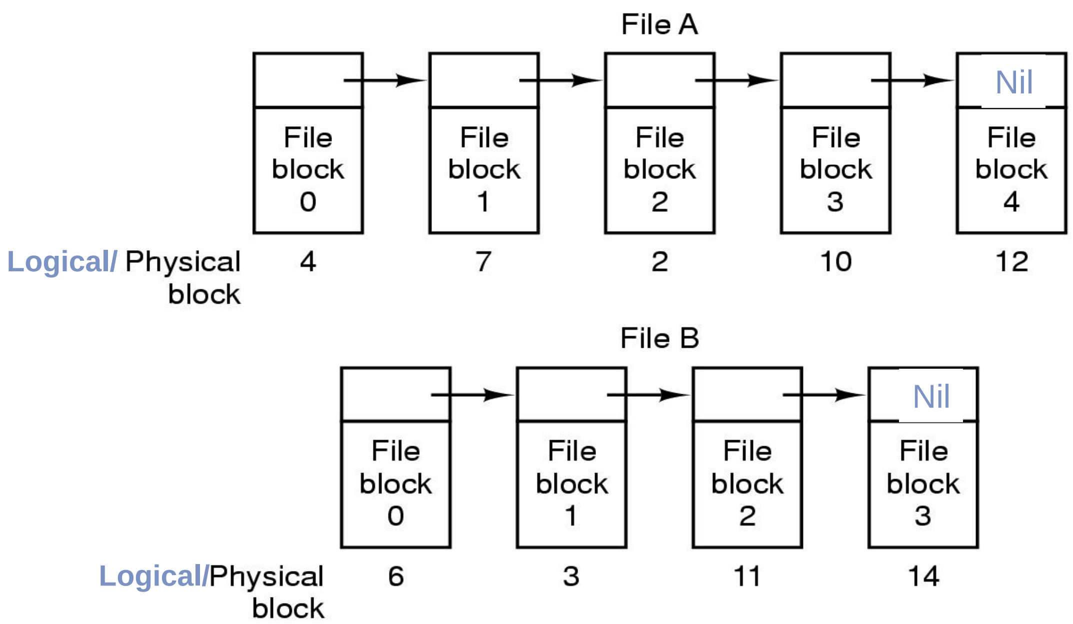
File Name	Start Block	Length
FileA	**0**	3
FileB	**3**	5
FileC	**8**	8
FileD	**16**	2
FileE	**18**	3

Chained Allocation (Linked List)

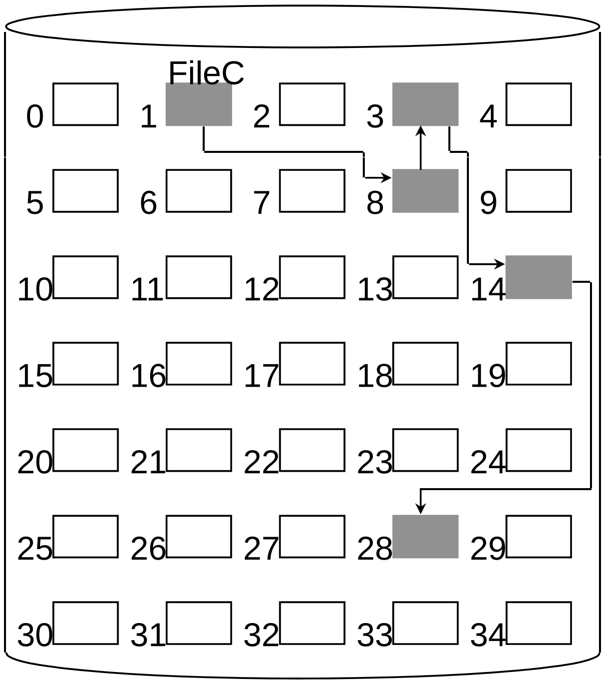
- Per file a linked list of logical file blocks, i.e.
  - Each file block contains a pointer to next file block, i.e. the amount of data space per block is no longer a power of two, ⇒ Consequences?
  - Last block contains a NIL-pointer (e.g. -1)
- FAT or directory contains address of first file block
- No external fragmentation
  - Any free block can be added to the chain
- Only suitable for sequential files
- No accommodation of the principle of disk locality
  - File blocks will end up scattered across the disk
  - Run a defragmentation utility to improve situation

Chained Allocation (2)

Storing a file as a linked list of disk blocks



Chained Allocation (3)



File Allocation Table

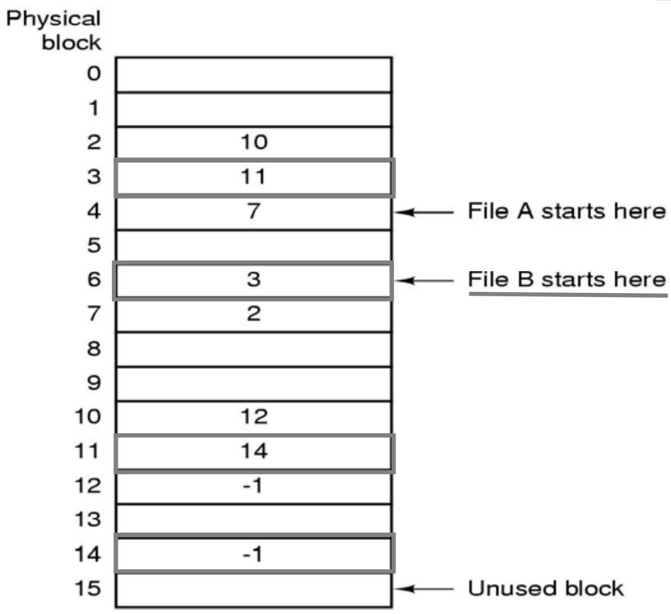
File Name	Start Block	Length
...	...	...
FileC	1	5
...	...	...

Remark:
If you only access sequentially this implementation is quite suited.

However requesting an individual record requires tracing through the chained block, i.e. far too many disk accesses in general.

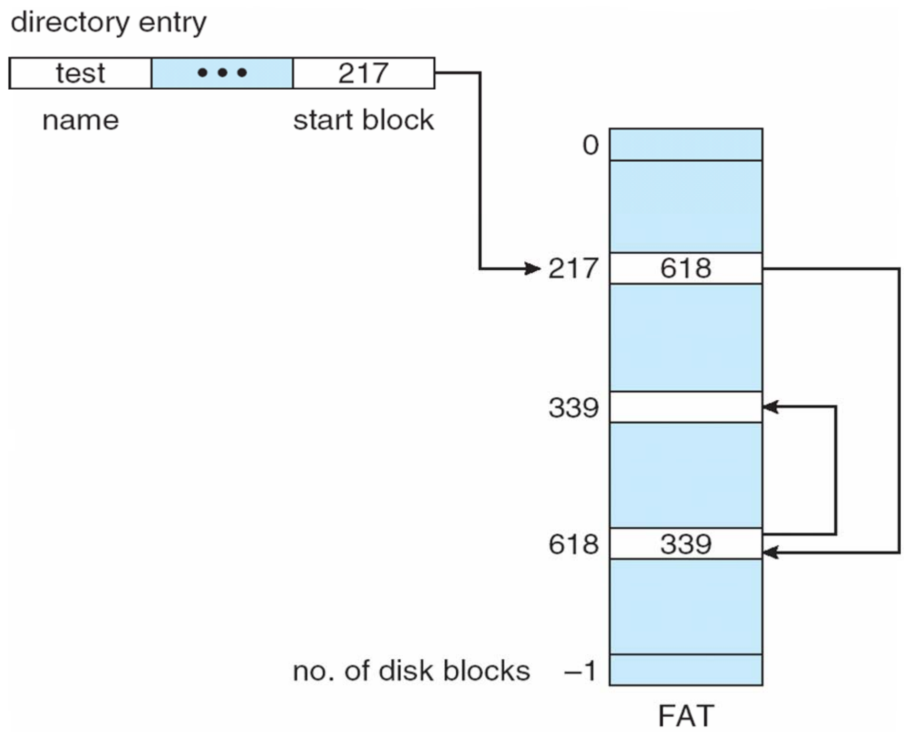
Linked List Allocation within RAM

- Each file block only used for storing file data
- Linked list allocation with FAT in RAM
  - Avoids disk accesses when searching for a block
  - Entire block is available for data
  - Table gets far too large for modern disks, ⇒
    - Can cache only, but still consumes significant RAM
    - Used in MS-DOS, OS/2



Similar to an inverted page table, one entry per disk block

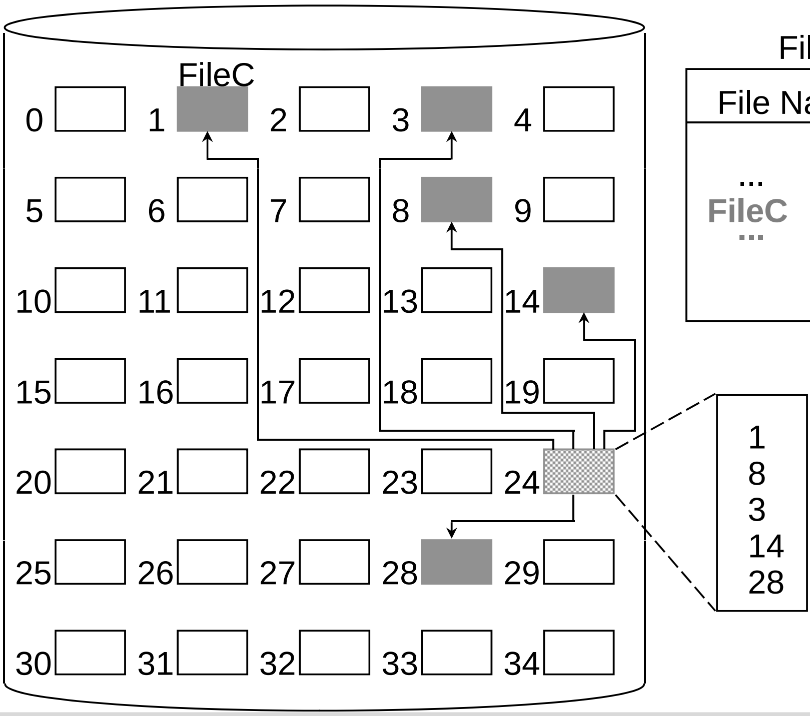
File-Allocation Table



Indexed Allocation (1)

- Indexed allocation
  - FAT (or special inode table) contains a one-level index table per file
    - Generalization n-level-index table
  - Index has one entry for allocated file block
  - FAT contains block number for the index

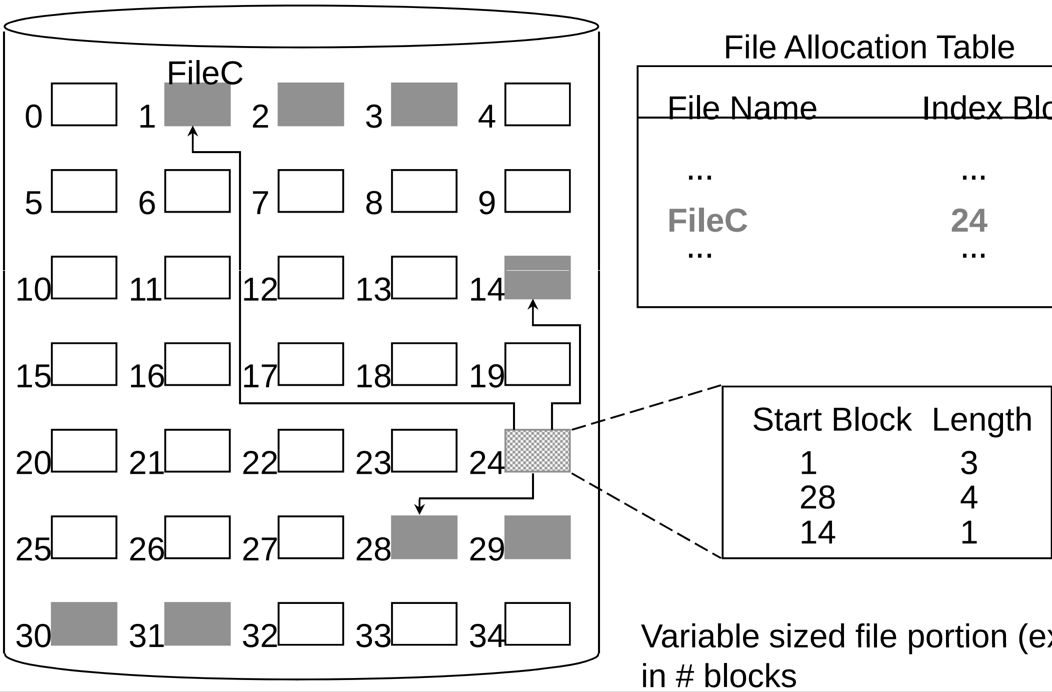
Indexed Allocation (2)



File Allocation Table

File Name	Index Block
...	...
FileC	**24**
...	...

Indexed Allocation (3)



File Allocation Table

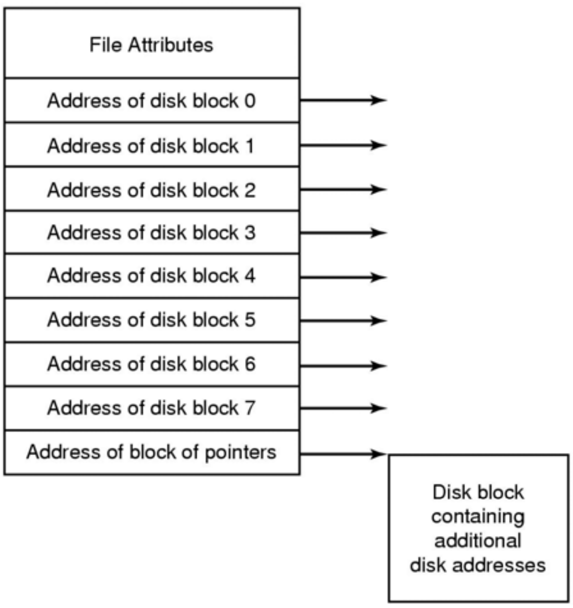
File Name	Index Block
...	...
FileC	**24**
...	...

Variable sized file portion (extent) in # blocks

Analysis of Indexed Allocation

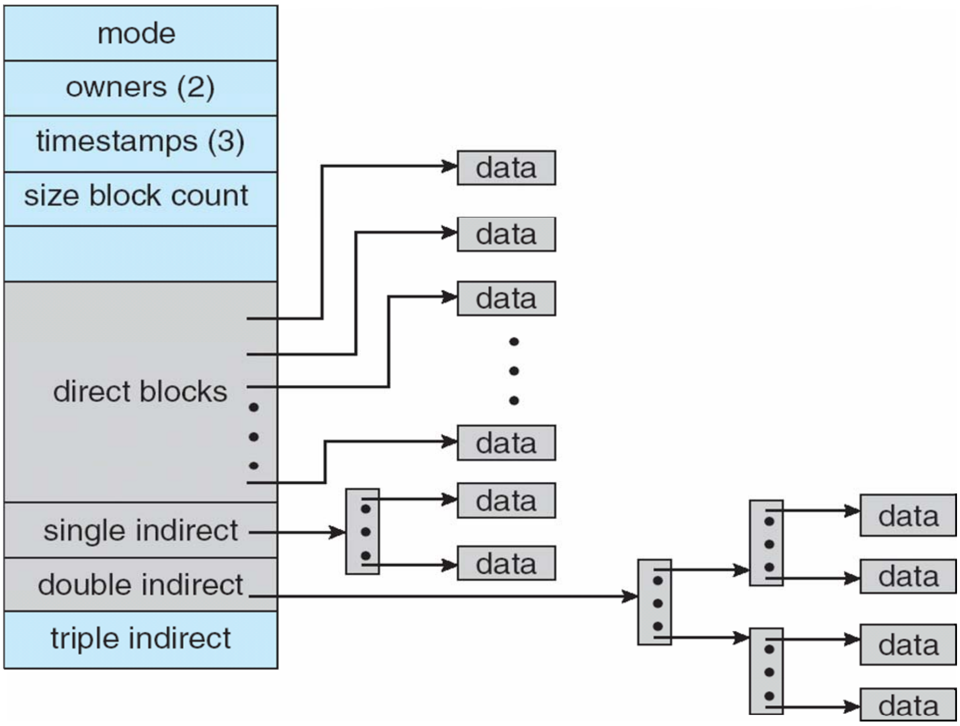
- Supports sequential and random access to a file
- Fragments
  - Block sized
    - Eliminates external fragmentation
  - Variable sized
    - Improves contiguity
    - Reduces index size

Indexed Allocation (5)



An example i-node

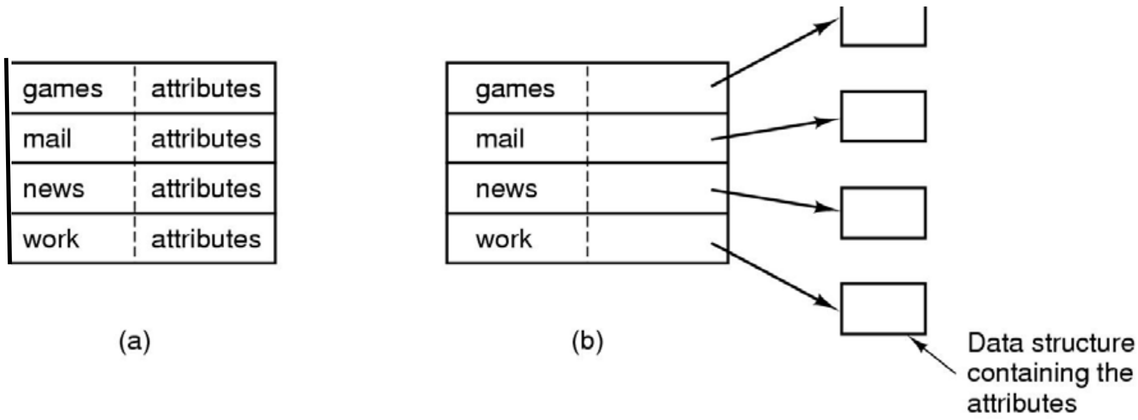
Example: UNIX (4K bytes per block)



Summary: File Allocation Methods

characteristic	contiguous	chained	indexed	
preallocation?	necessary	possible	possible	
fixed or variable size fragment?	variable	fixed	fixed	variable
fragment size	large	small	small	medium
allocation frequency	once	low to high	high	low
time to allocate	medium	long	short	medium
file allocation table size	one entry	one entry	large	medium

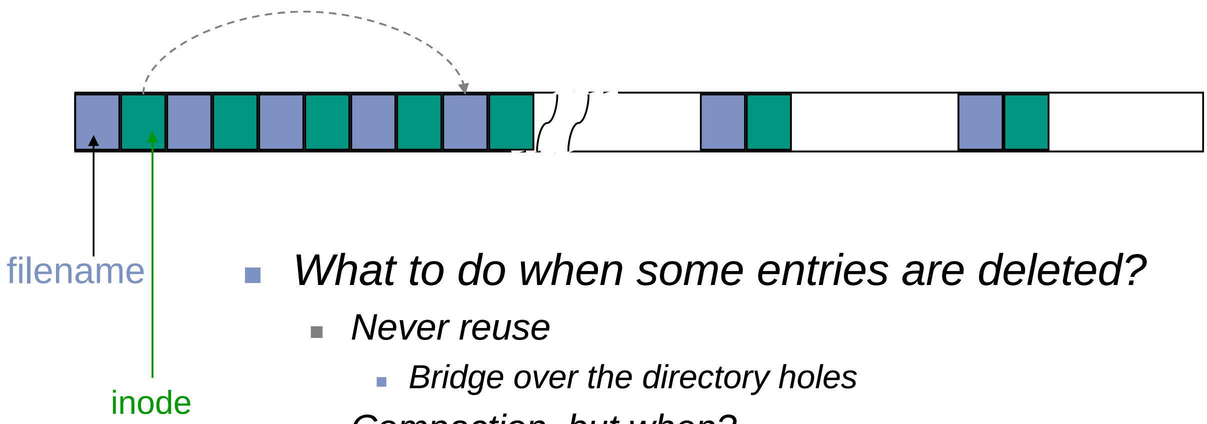
Implementing Directories



- (a) A simple directory (MS-DOS)
  - fixed size entries
  - disk addresses and attributes in directory entry
- (b) Directory in which each entry just refers to an i-node (Unix)

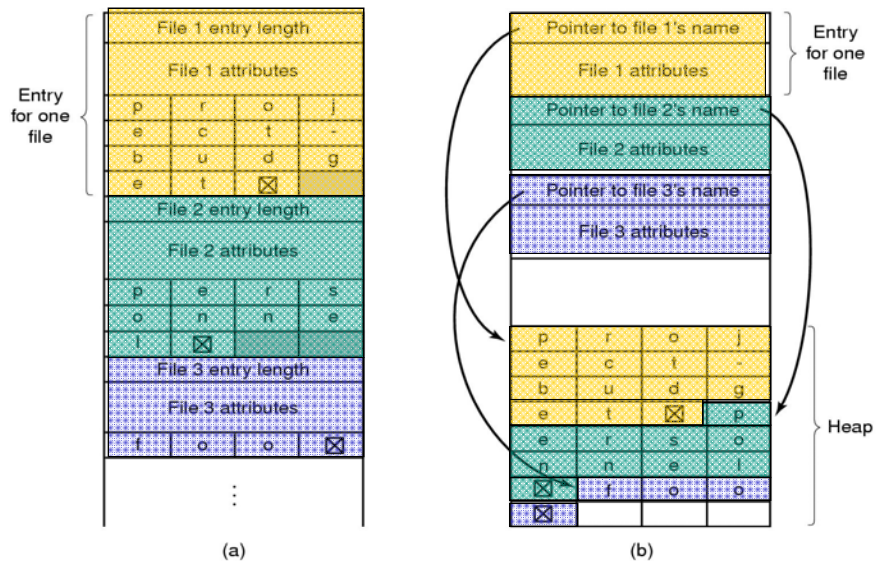
Implementing Directories

- *How to implement a Unix-like directory?*



- *What to do when some entries are deleted?*
  - *Never reuse*
    - *Bridge over the directory holes*
  - *Compaction, but when?*
    - *eager or*
    - *lazy*

Directory Entries & Long Filenames



- Two ways of handling long file names in directory
  - (a) In-line
  - (b) In a heap

Analysis: Linear Directory Lookup

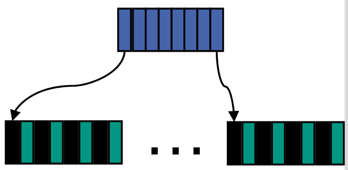
- Linear search ⇒ for big directories not efficient
- Space efficient as long as we do compaction
  - Either eagerly after entry deletion or
  - Lazily (but when?)
- With variable file names ⇒ deal with fragmentation
- Alternatives
  - (e.g., extensible) hashing
  - (e.g., B-) Tree structures

Hashing a Directory Lookup

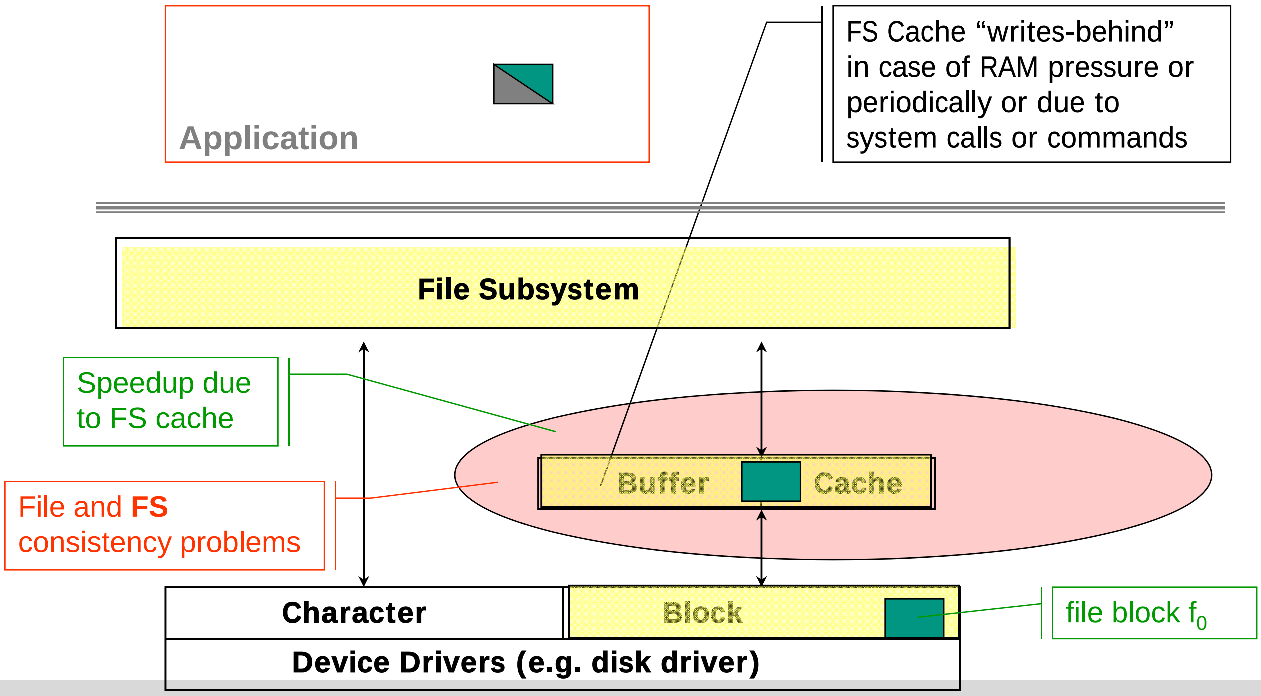
- Method:
  - Hashing a file name to an inode
  - Space for filename and meta data is variable sized
  - Create/delete will trigger space allocation and clearing
- Advantages:
  - Fast lookup and relatively simple
- Disadvantages:
  - Might be not as efficient as trees for very large directories

Tree Structure for a Directory

- Method
  - Sort files by name
  - Store directory entries in a B-tree like structure
  - Create/delete/search in that B-tree
- Advantages:
  - Efficient for a large number of files per directory
- Disadvantages:
  - Complex
  - Not that efficient for a small number of files
  - More space

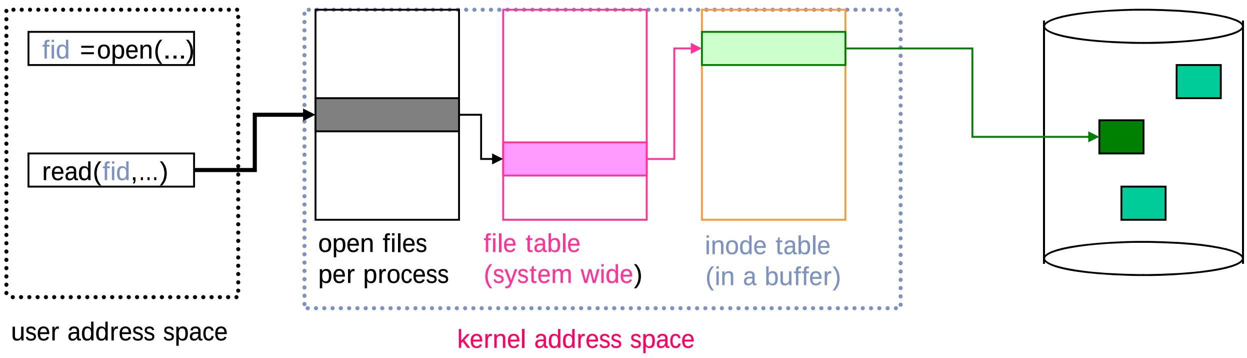


UNIX File System Structure



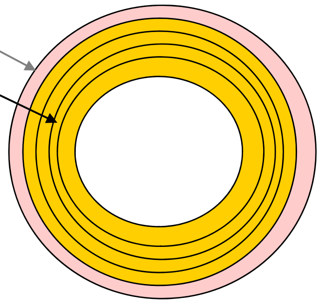
Using a Unix File

- Opening a file creates a file descriptor fid
- Used as an index into a process-specific table of open files
- The corresponding table entry points to a system-wide file table
- Via buffered inode table, you finally get the data blocks



Original Unix File System

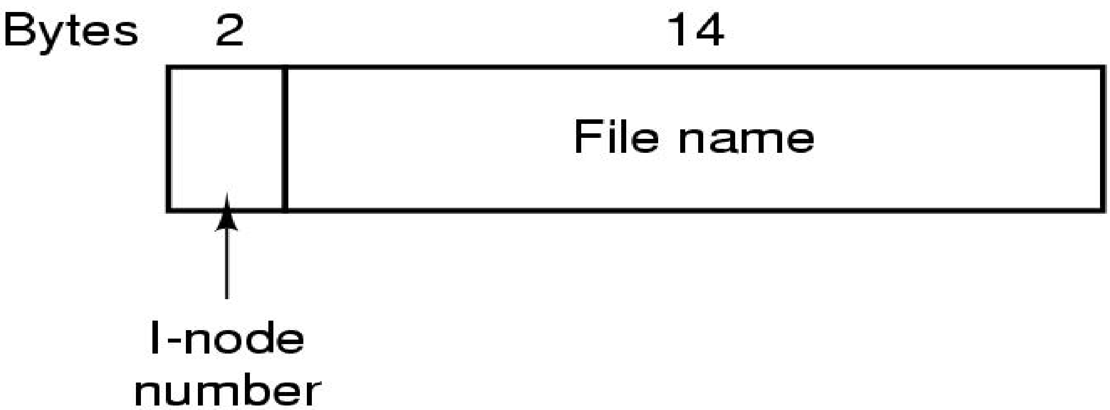
- Simple disk layout
  - Block size = sector size (512 bytes)
  - Inodes on outermost cylinders[1]
  - Data blocks on the inner cylinders
  - Freelist as a linked list
- Issues
  - Index is large
  - Fixed number of files
  - Inodes far away from data blocks
  - Inodes for directory not close together
  - Consecutive file blocks can be anywhere
  - Poor bandwidth for sequential access



[1] in very early Unix FSs inode table in the midst of the cylinders

Unix File Names

- Historically (Version 7) only 14 characters



- System V up to 255 ASCII characters

  <filename> . <extension>

BSD FFS

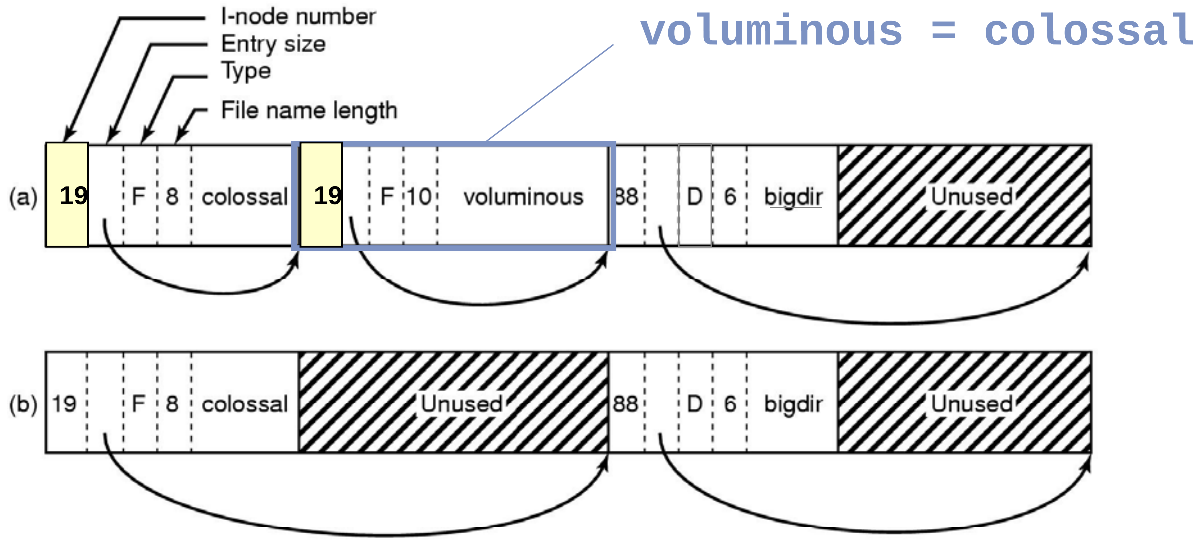
- Use a larger block size: 4 KB or 8 KB
  - Allow large blocks to be chopped into 2,4 or 8 fragments
  - Used for little files and pieces at the ends of files
- Use *bitmap* instead of a free list
  - Try to allocate more contiguously
  - 10% reserved disk space

BSD FFS Directory

- Directory entry needs three elements:
  - length of dir-entry (variable length of file names)
  - file name (up to 255 characters)
  - inode number (index to a table of inodes)
- Each directory contains at least two entries:
  - .. = link to the parent directory (forming the directory tree)
  - . = link to itself
- FFS offers a "tree-like structure" (like Multics), supporting human preference, ordering hierarchically

© 2009 Karlsruhe Institute of Technology, System Architecture Group
Frank Bellosa, Gerd Liefländer, Silberschatz et al.

Unix BSD FFS Directory (2)

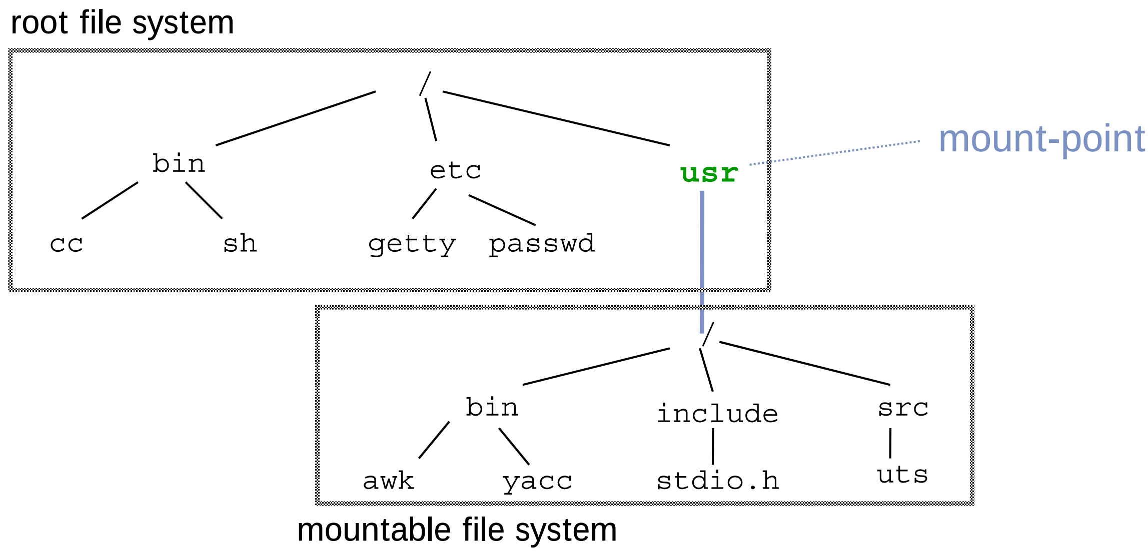


- BSD directory three entries (voluminous = hardlink to colossal)
- Same directory after file `voluminous` has been removed

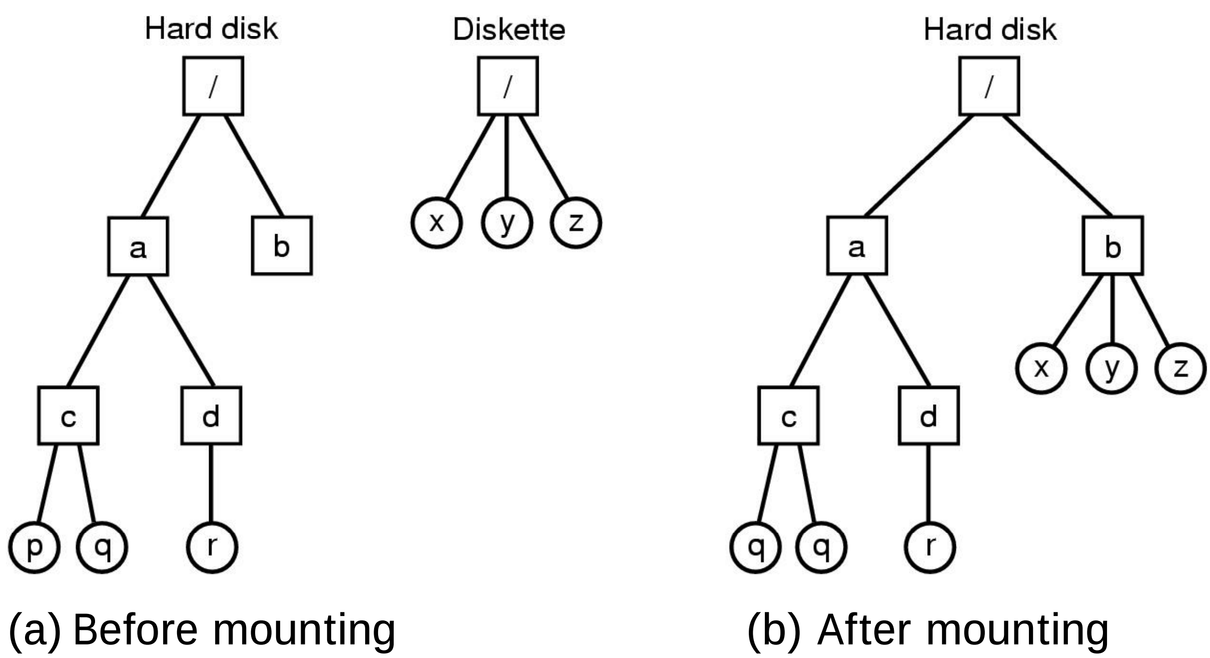
Unix Directories

- Multiple directory entries may point to same inode (hard link)
- Pathnames are used to identify files
  `/etc/passwd` an absolute pathname
  `../home/lief/examination` a relative pathname
- Pathnames are resolved from left to right
- As long as it's not the last component of the pathname, the component name must be a directory
- With symbolic links you can address files and directories with different names. You can even define a symbolic link to a file currently not mounted (or even that never existed); i.e. a symbolic link is a file containing a pathname

Logical and Physical File System



Mounting a File System

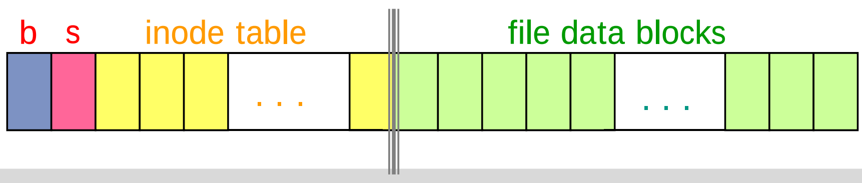


Logical and Physical File System

- A logical file system can consist of different physical file systems
- A file system can be mounted at any place within another file system
- When accessing the “local root” of a mounted file system, a bit in its inode identifies this directory as a so-called mount point
- Using mount respectively umount the OS manages a so called mount table supporting the resolution of path names crossing file systems
- The only file system that has to be resident is the root file system (in general on a partition of a hard disk)

Layout of a Logical Disk

- Each physical file system is placed within a logical disk partition. A physical disk may contain several logical partitions (or logical disks)
- Each partition contains space for the boot block, a super block, the inode table, and the data blocks
- Only the root partition contains a real boot block
- Border between inodes and data blocks region can be set, thus supporting better usage of the file system
  - with either few large files or
  - with many small files



Hard Links ↔ Symbolic Links

Hard link is another *file name*, i.e. ∃ another directory entry pointing to a specific file; its inode-field is the same in all hard links. Hard links are bound to the logical device (partition).

Each new hard link increases the *link counter* in file's i-node. As long as link counter ≠ 0, file remains existing after a *rm.* In all cases, a remove decreases link counter.

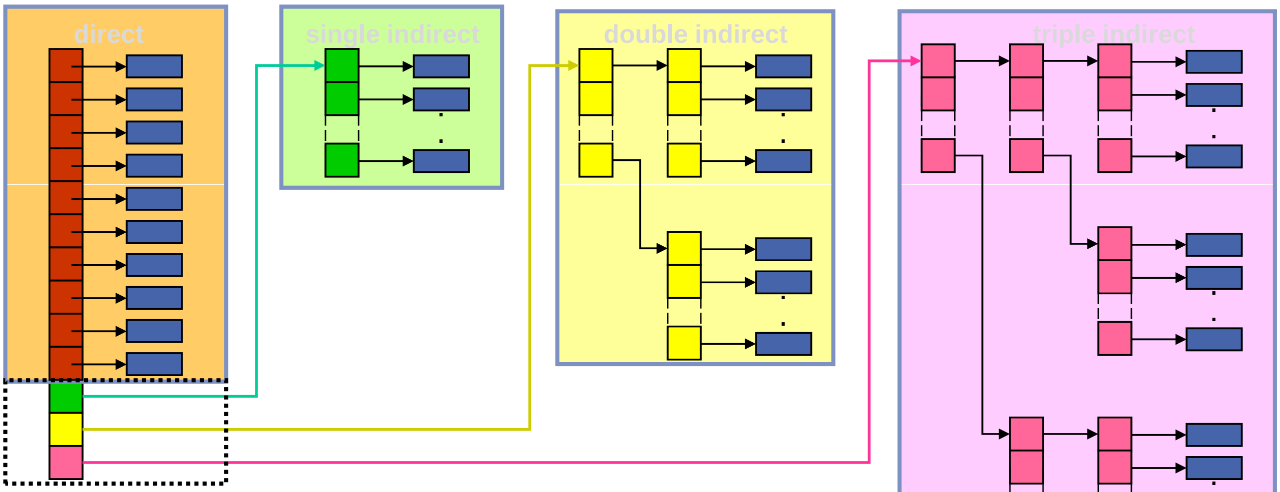
Symbolic link is a *new file* containing a pathname pointing to a file or to a directory. Symbolic links are evaluated per access. If file or directory is removed the symbolic link points to *nirwana*.

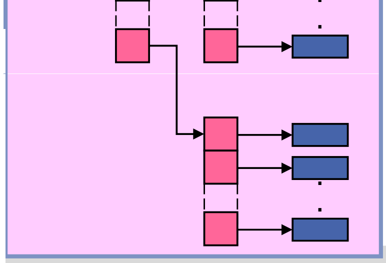
You may even specify a symbolic link to a file or to a directory currently *not present* or even currently *not existent*.

Unix Inode

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Access Structure

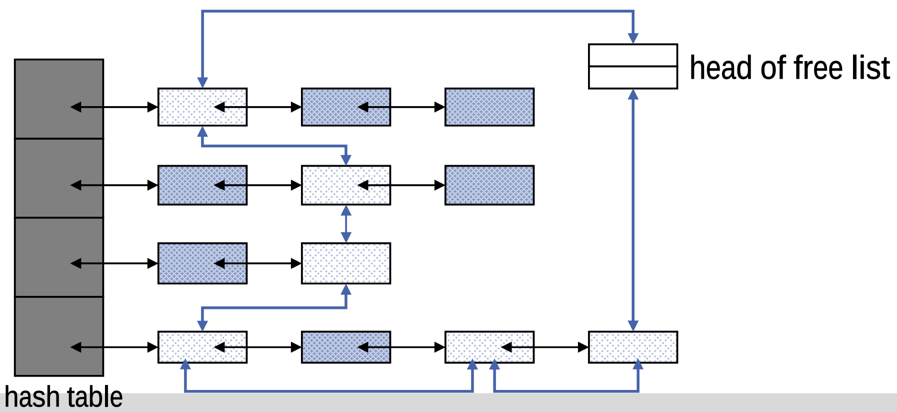




Remark:
Depending on the block size (e.g. 512 Bytes, ...) and on the pointer length (e.g. 4 Bytes) maximum file size is greater than 2 MB.
“Small” files are favored concerning access speed.

Buffering

- Disk blocks are buffered in main memory. Access to buffers is done via a hash table.
- Blocks with the same hash value are chained together
- Buffer replacement policy = LRU
- Free buffer management is done via a double-linked list.



UNIX Block Header

Dirty block,
Locked block etc.

Data block in memory

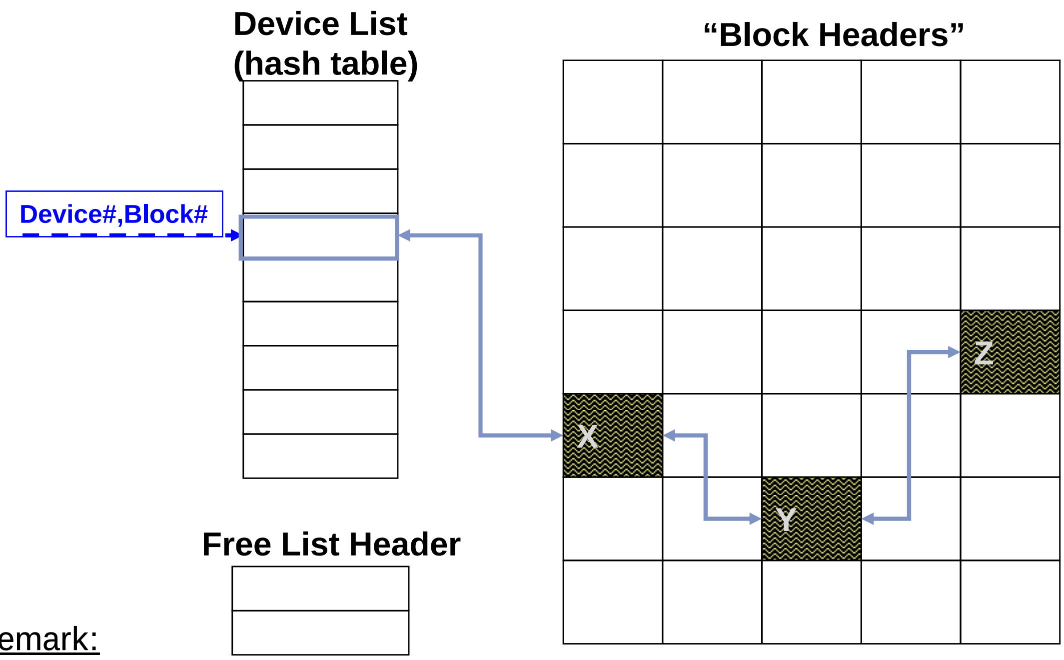
block status flags
pointer to cached block
device number
block number on device
I/O error status
bytes left to transfer
pair of pointers in case of hash collisions
pair of pointers for the free list

Used for hashing

Used by disk I/O driver

Between other block headers

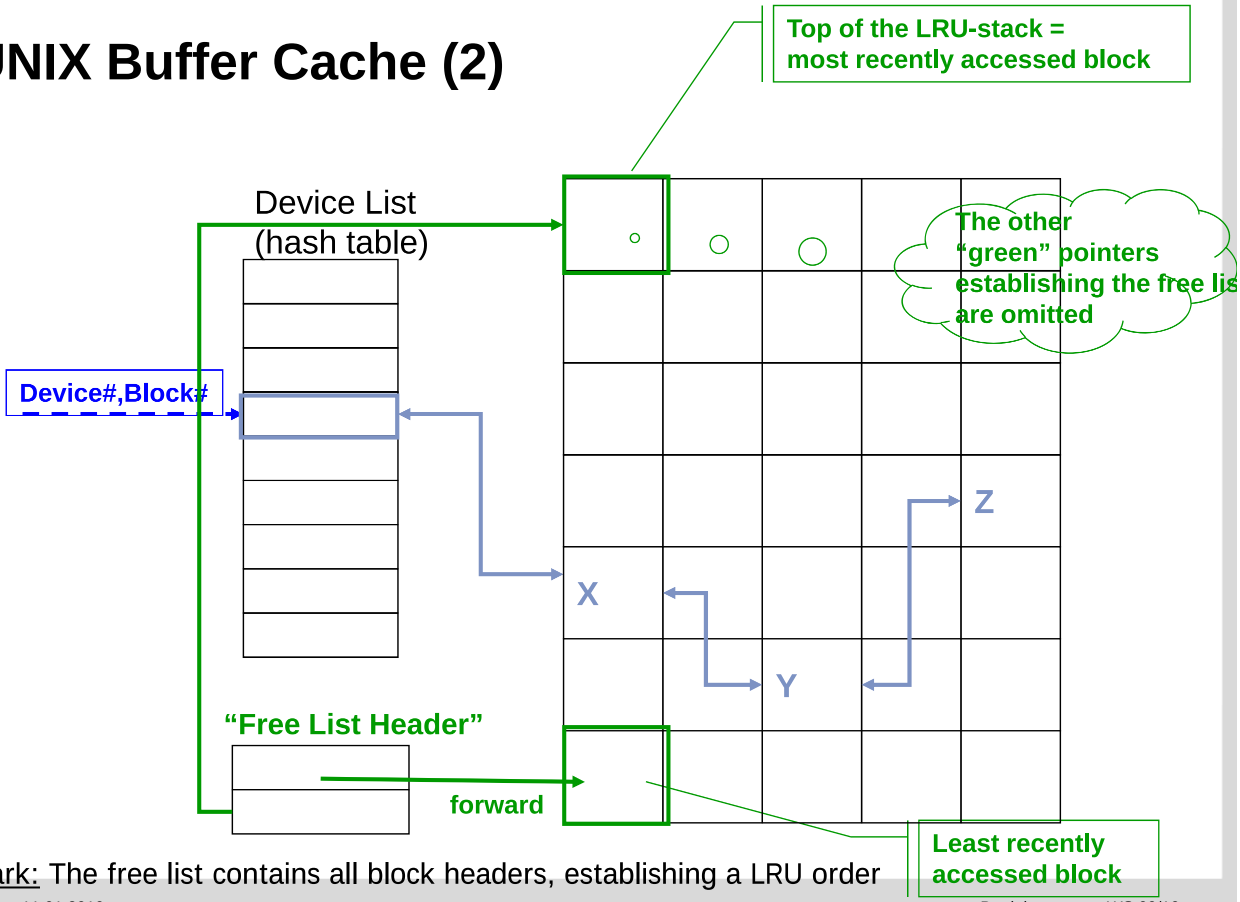
UNIX Buffer Cache (1)



Remark:

X,Y, and Z are block headers of blocks mapped into the same hash table entry

UNIX Buffer Cache (2)



Remark: The free list contains all block headers, establishing a LRU order

UNIX Buffer Cache (3)

Advantages:

- reduces disk traffic
- "well-tuned" buffer has hit rates up to 90% (according to Ousterhout 10.th SOSP 1985)
- ~ 10% of main memory for the buffer cache (recommendation for *old configurations*)

UNIX Buffer Cache (4)

Disadvantages:

- Write-behind policy might lead to
  - data losses in case of system crash and/or
  - inconsistent state of the FS

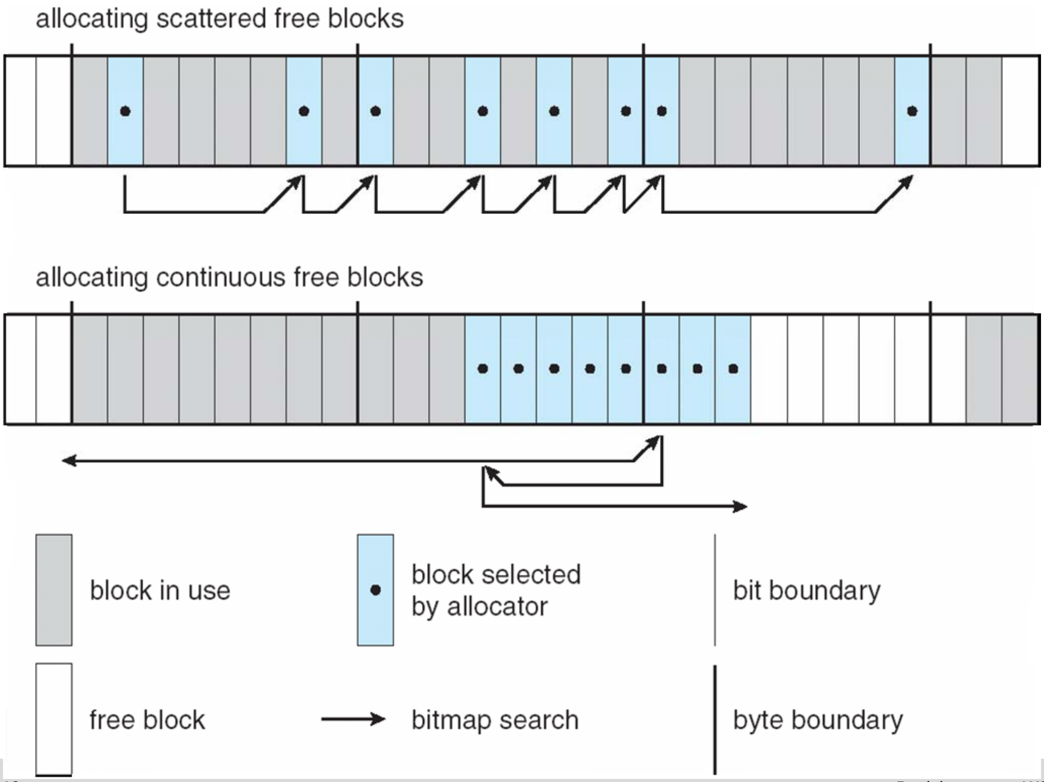
$\Rightarrow$ rebooting system might take some time due to fsck, i.e. *checking all directories and files* of FS

- Always *two copies* involved
  - from disk to buffer cache (in kernel space)
  - from buffer to user address space
- *FS Cache wiping* if sequentially reading a very large file from end to end and not accessing it again

The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file
- The main differences between ext2fs and ffs concern their disk allocation policies
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
  - Ext2fs does not use fragments; it performs its allocations in smaller units
    - The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation

Ext2fs Block-Allocation Policies



Journaling File Systems

- Journaling file systems record each update to the file system as a **transaction**
- All transactions are written to a **log**
  - A transaction is considered **committed** once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

Log-Structured File Systems

- Log-structured FS: use disk as a circular buffer:
- Write all updates, including inodes, meta data and data to end of log
  - have all writes initially buffered in memory
  - periodically write these within 1 segment (1 MB)
  - when file opened, locate i-node, then find blocks
- From the other end, clear all data, no longer used