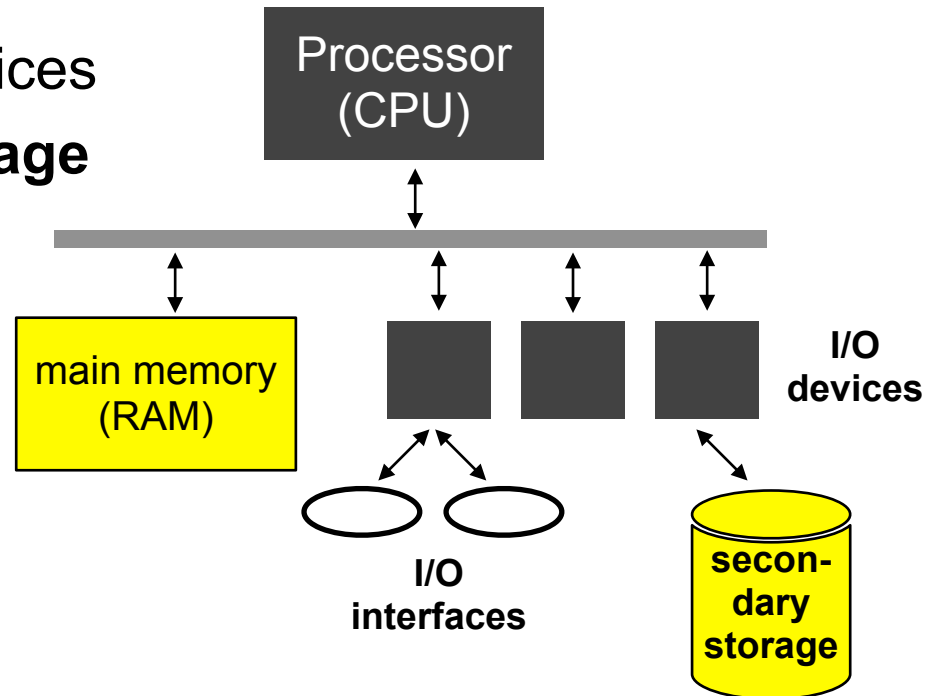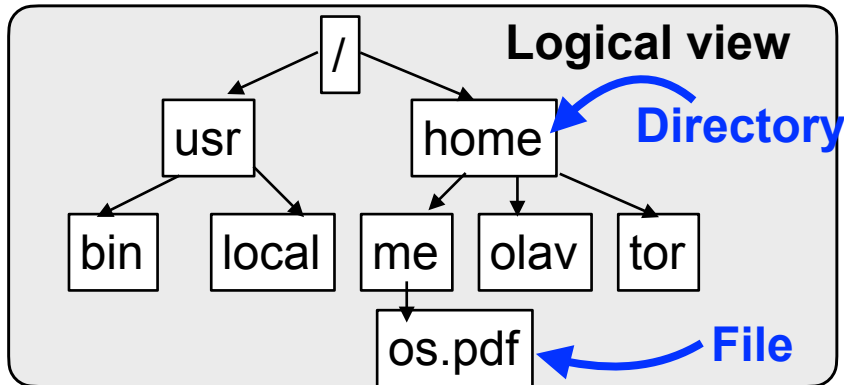# Operating Systems

Lecture 15: File systems

Michael Engel

Resources again

- So far we considered:
  - CPU
  - main memory
  - I/O devices
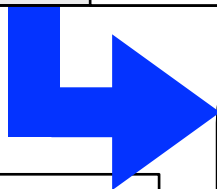   - block-oriented devices
- **Today: background storage**

Processor (CPU)

main memory (RAM)

I/O devices

I/O interfaces

secon-dary storage

Background storage

Logical view

/
 ├── usr
 │   ├── bin
 │   └── local
 └── home
      ├── me
      │   └── os.pdf
      ├── olav
      └── tor

Directory

File

mapping
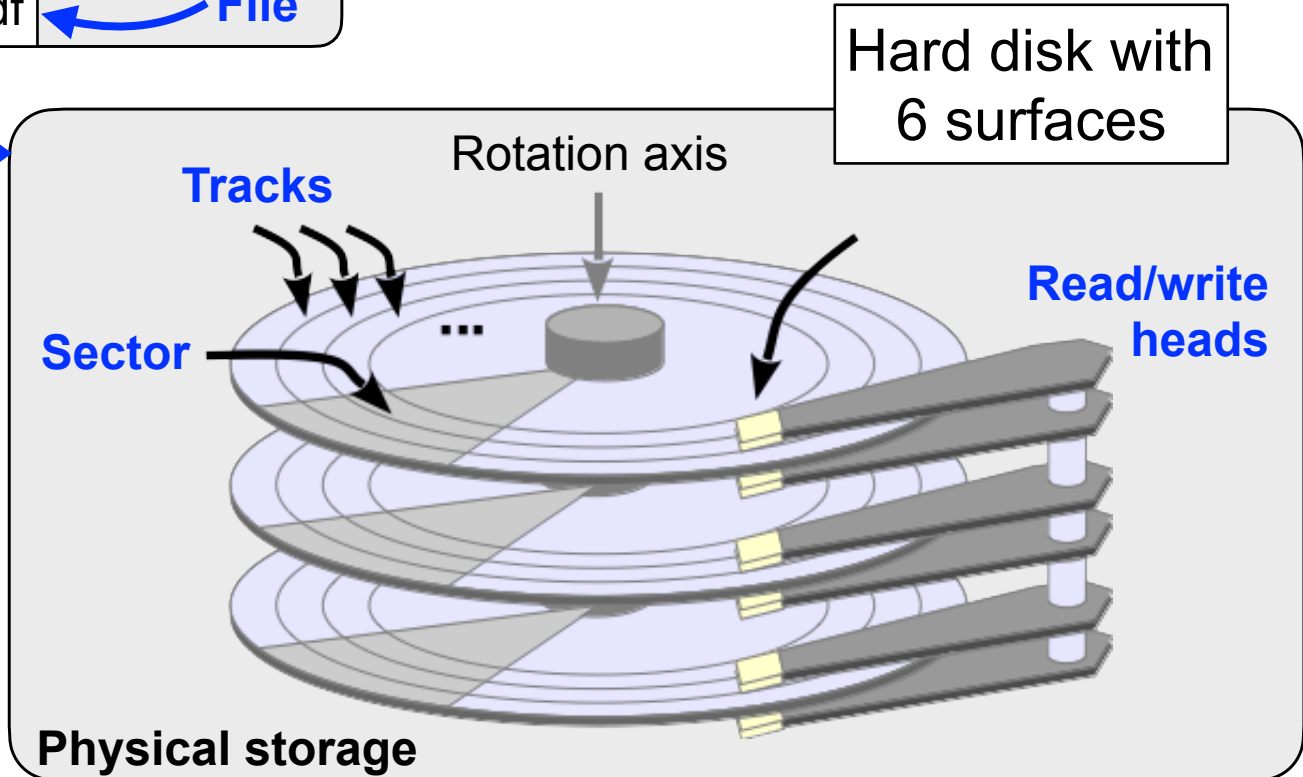
File systems enable permanent storage of large amounts of data

The operating system provides a logical view to applications and has to implement this efficiently

Hard disk with 6 surfaces

Tracks

Rotation axis

Sector

...

Read/write heads

Physical storage

Norwegian University of Science and Technology
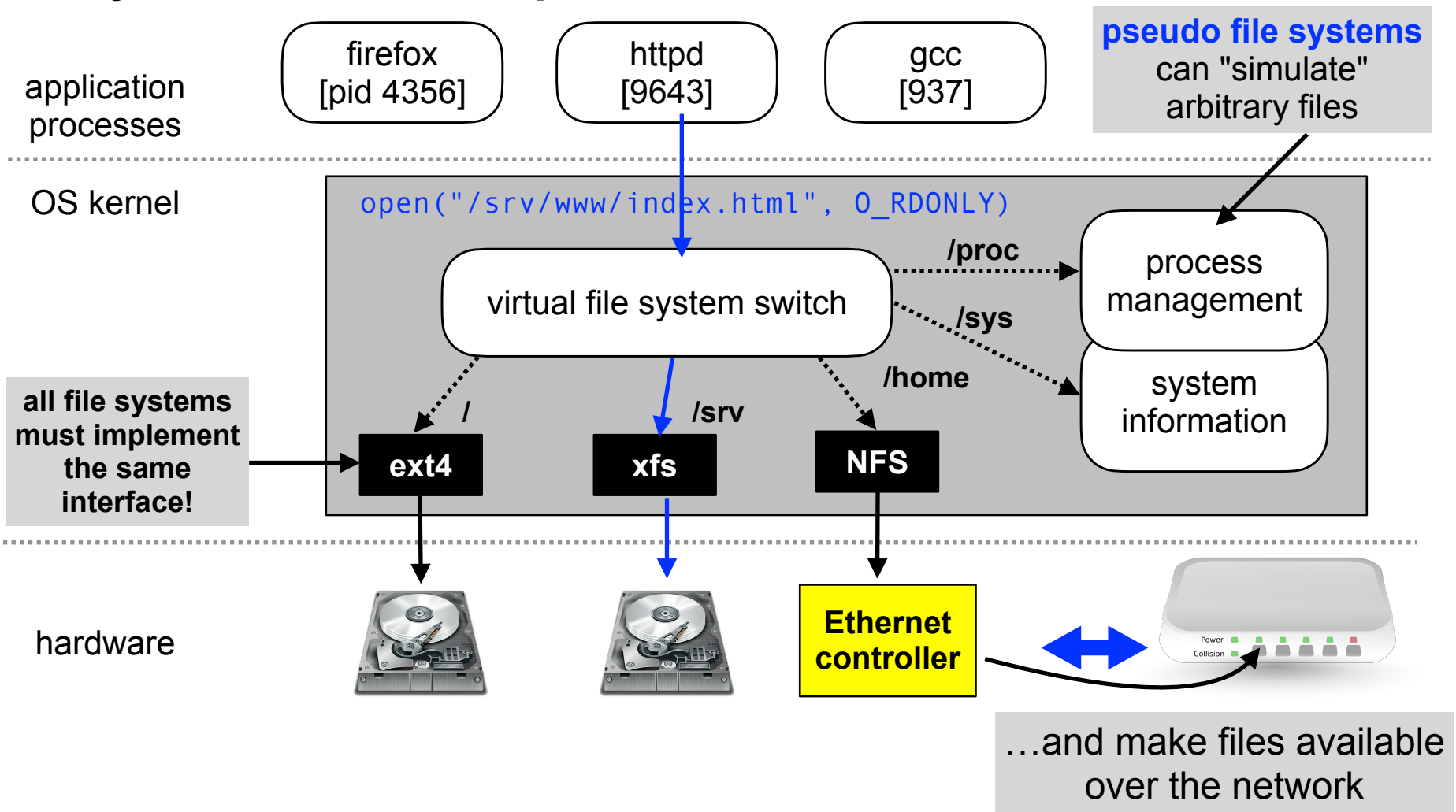
The file abstraction

- Unix principle: "***everything is a file***"
    - more precisely: every resource in the system can be accessed using a ***name*** mapped into a directory hierarchy
    - access to the resource takes place using the standard Unix system calls for file access
    - file permissions are used to control access to the resource
- Examples:
    - regular files and directories
    - special files for devices, symbolic links, named pipes
    - virtual files for process and system information
- Not completely consistent in Unix, but e.g. in the Plan 9 OS:
    - network connections and protocols
    - access to the graphics frame buffer

NTNU | Norwegian University of Science and Technology

Accessing files

- Files are identified by per process *file descriptors* in the OS
  - positive integer number, can be reassigned
- The Unix file access API consists of four simple system calls:
- `int open(const char *path, int oflag, ...);`
  - Attempts to open the file with the given path name and options for accessing (read only, read/write etc.)
  - Returns a **file descriptor** (fd) refering to the file on success
- `ssize_t read(int fd, void *buf, size_t nbyte);`
- `ssize_t write(int fd, const void *buf, size_t nbyte);`
  - Read (write) `nbyte` bytes from (to) file `fd` into (from) the memory starting at user space memory address `buf`
- `int close(int fildes);`
  - Closes the file: flushes buffers and invalidates file descriptor

The Unix virtual file system (VFS) switch

- **System-wide name space** for files

application processes

firefox [pid 4356]

httpd [9643]

gcc [937]

pseudo file systems can "simulate" arbitrary files

OS kernel

`open("/srv/www/index.html", O_RDONLY)`

virtual file system switch

/proc

/sys

/home

process management

system information

all file systems must implement the same interface!

ext4 /

xfs /srv

NFS

hardware

Ethernet controller

Power
Collision

…and make files available over the network

Norwegian University of Science and Technology
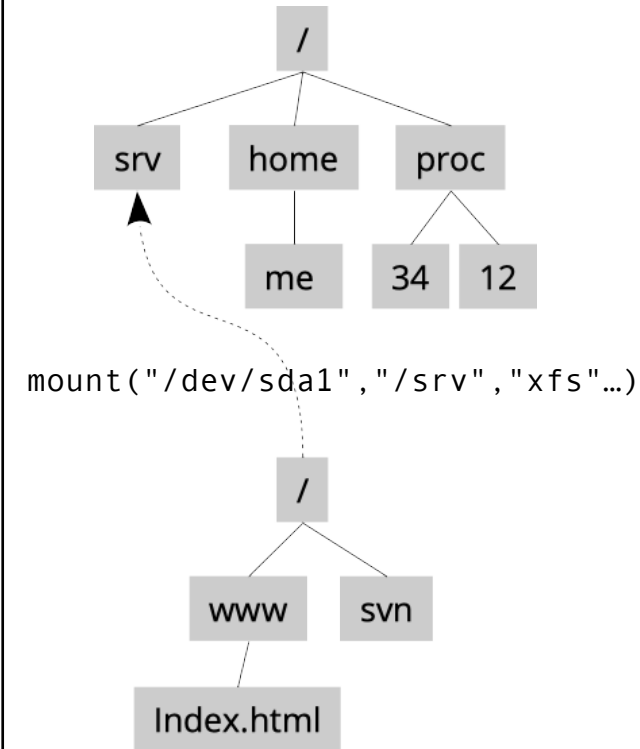
Virtual file system: mounting

System Call:

```
int mount(const char *source, const char
  *target, const char *filesystemtype,
  unsigned long mountflags, const void *data);
```

Attaches ("mounts") a file system to the given directory in the global directory tree

System Call:

```
int umount(const char *target);
```

Removes the attachment. *Note:* **umount**, not unmount!
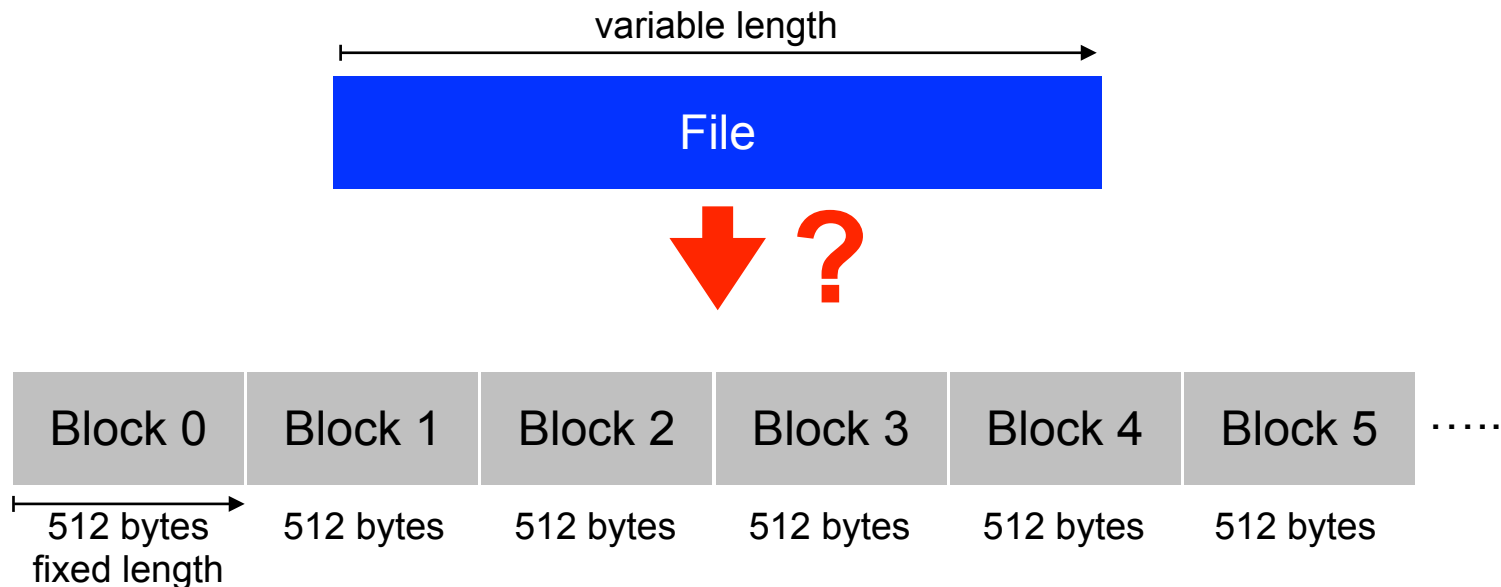


```
mount("/dev/sda1","/srv","xfs"…)
```

Using both system calls requires **system administrator privileges!**

When the system is booted, all filesystems listed in **/etc/fstab** are automatically mounted

File storage

- In most cases, files require multiple blocks of storage on disk
  - We simply view a disk as a large array of blocks
    - Each block has an identical small size, e.g. 512 bytes
    - This is already an *abstraction* from the heads, tracks and sectors of a real disk drive
  - Which of the blocks are used to store a file?

variable length

File

?

Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	⋯⋯
512 bytes fixed length	512 bytes	512 bytes	512 bytes	512 bytes	512 bytes	

Norwegian University of Science and Technology

Contiguous storage

- A file is stored in blocks with increasing block numbers
  - requires to store information about the first block and the number of subsequent blocks, e.g. Start: block 2, length: 3

| Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6

- Advantage:
  - Access to all blocks with minimal delay due to disk arm positioning
  - Fast direct access to a given file offset position
  - Used for read-only file systems, e.g. CD-ROM/DVD

Contiguous storage problems

- **Finding free space** on the disk
  - required: sufficiently large set of contiguous free disk blocks
- **Fragmentation**
  - free blocks that cannot be used since they are too small for the given file
  - cf. main memory management
- The **size of new files** is usually not known in advance
  - **Extending** (growing the size of) a file is problematic: what if the following blocks are already allocated?
  - Requires **copying of data** if insufficient free following blocks are available

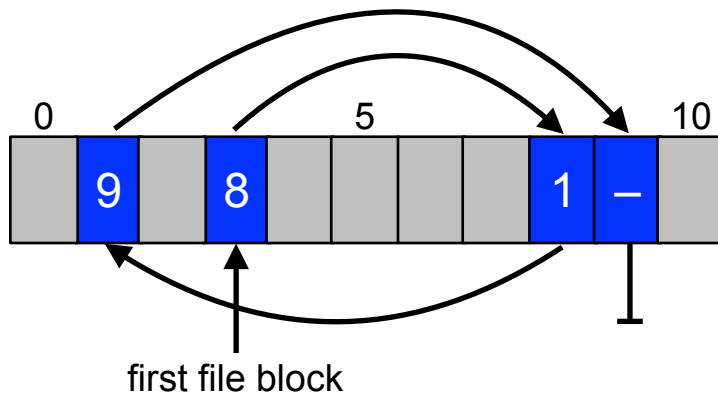# Linked list storage

- Blocks of a file are linked
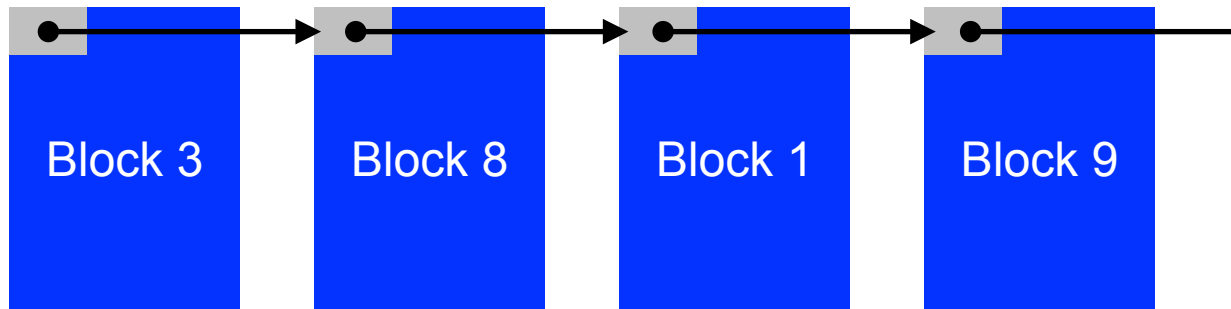
- e.g. used on Commodore disk drives (C64 etc.)
  - block size 256 bytes
  - first two bytes: track and sector nr. of following block
  - if track number = 0 ➤ final block
  - 254 byte payload data
- Files can be extended and shrunk
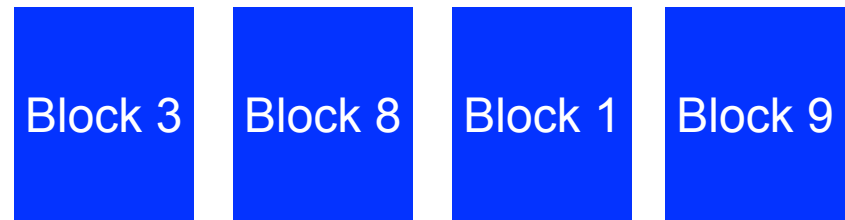
Linked list storage problems

- Available storage is reduced by amount of **memory used for pointers**
  - Problematic when using paging:
    a page would always require parts of two disk blocks

- **Error prone**
  - a file cannot be completely restored if the pointer information contains errors

- **Direct access** to arbitrary file positions is difficult

- Requires **frequent positioning** of the disk head when the data blocks are spread over the disk

Norwegian University of Science and Technology

Linked list storage: FAT

- Links are stored in separate disk blocks
  - FAT: file allocation table (first used in MS-DOS)

Blocks of the file: 3, 8, 1, 9

first file block

| Block 3 | Block 8 | Block 1 | Block 9 |

- Advantages:
  - complete content of data block is usable
  - redundant storage of the FAT is possible
    - useful in case of an error

Norwegian University of Science and Technology
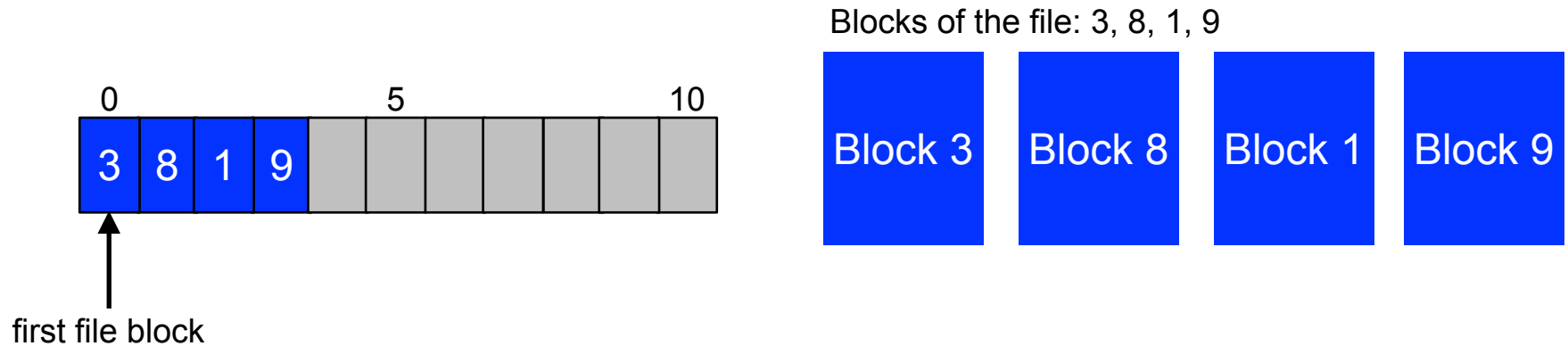
Linked list storage problems (2)

- **Additional loading** of at least one block required
  - it is possible to cache the FAT to increase efficiency

- **Unused information is loaded**
  - FAT contains links for *all files*

- **Search overhead** for the data block containing information at a given offset inside a file

- **Frequent positioning** of the disk head when data blocks are scattered over the disk

Discussion: chunks/extents/clusters

- Variation
  - Split a file into sequences of blocks stored contiguously (called *chunk*, *extent* or *cluster*)
  - Reduces the number of positioning actions
  - Improves the speed to search for a block linear
    - depending on the chunk size
- Problems:
  - additional information required for managing chunks
  - fragmentation
    - fixed size: inside of a **sequence** (internal fragmentation)
    - variable size: outside of the sequences (external fragm.)

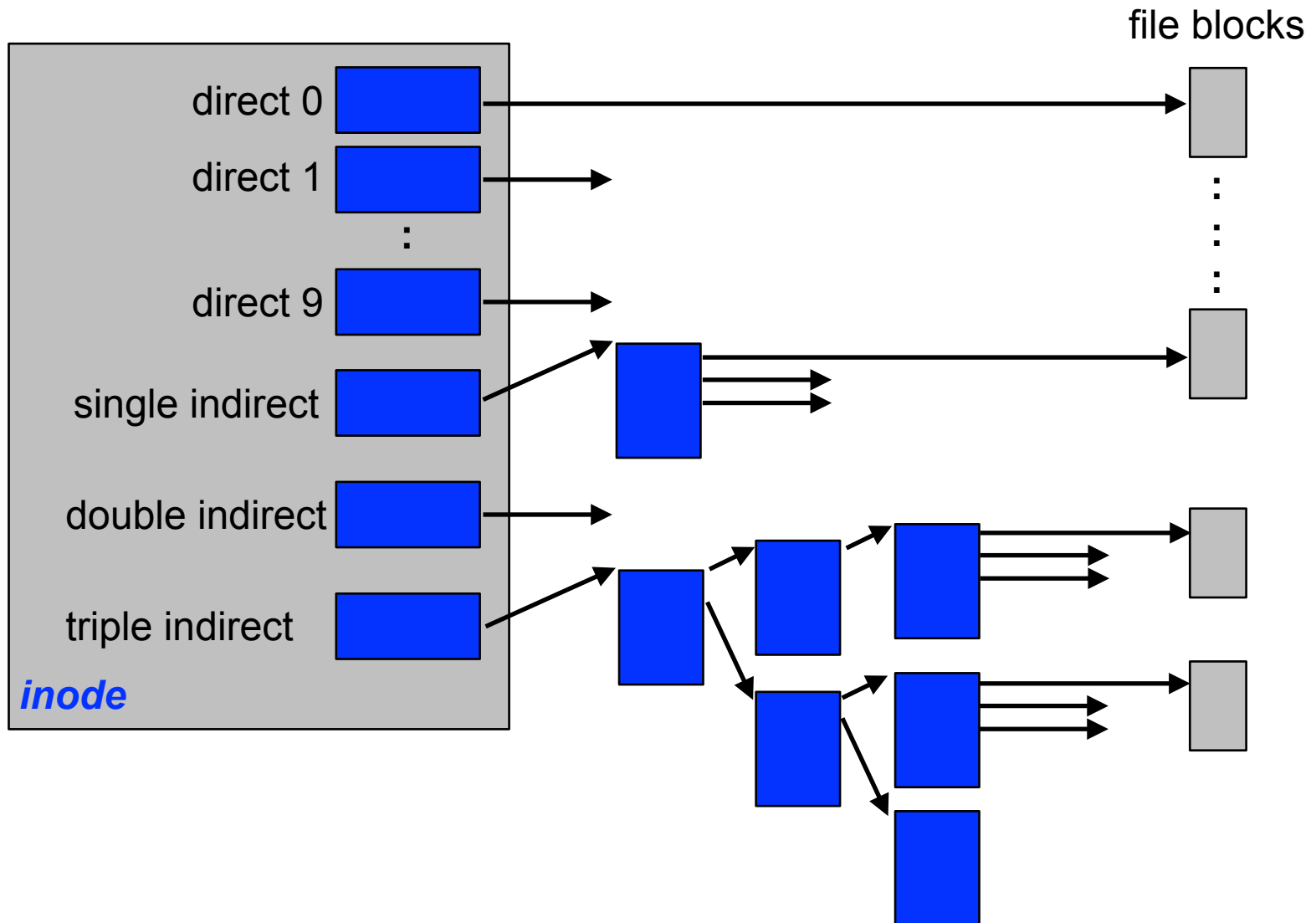- Is used in practice, but does not have significant advantages

Indexed storage

- A special disk block contains block numbers of the data blocks of a file

Blocks of the file: 3, 8, 1, 9

0				5					10
3	8	1	9						

↑
first file block

Block 3	Block 8	Block 1	Block 9

- Problem
  - Fixed number of blocks that can be referenced in the index block
    - Fragmentation for small files
    - Extensions required for large files

Norwegian University of Science and Technology

Indexed storage: Unix inodes

file blocks

direct 0

direct 1

:

direct 9

single indirect

double indirect

triple indirect

inode

NTNU | Norwegian University of Science and Technology

Indexed storage: discussion

- Use of multiple indexing levels

  - inodes require a block on the disk in any case (fragmentation is not a problem for small files)

  - multiple levels of indexing enable the addressing of large files

- Disadvantage:

  - multiple blocks have to be loaded (only for large files)

Tree sequential storage

- Used in databases to efficiently find records using a search key
  - Key space can be sparsely populated
- Can also be used to find *chunks* of files with a certain file offset, e.g. in NTFS, btrfs, IBM JFS2, Apple HFS+ (B+ tree)
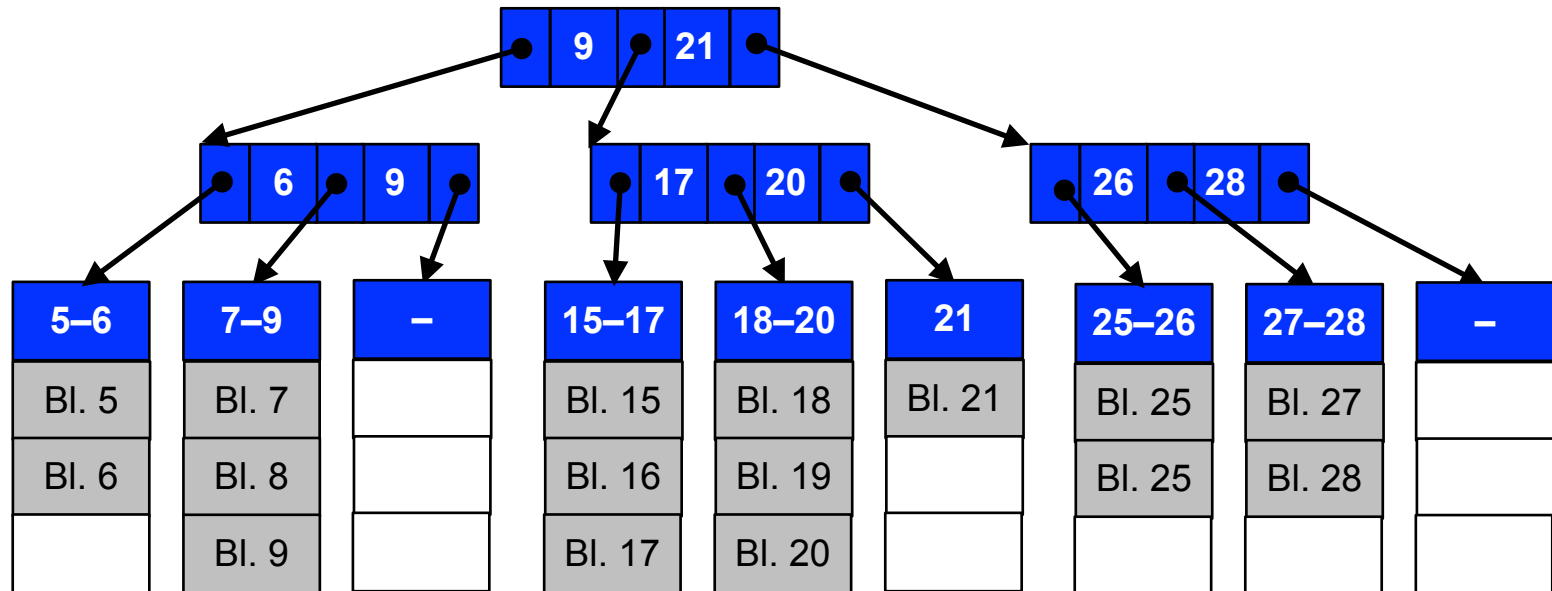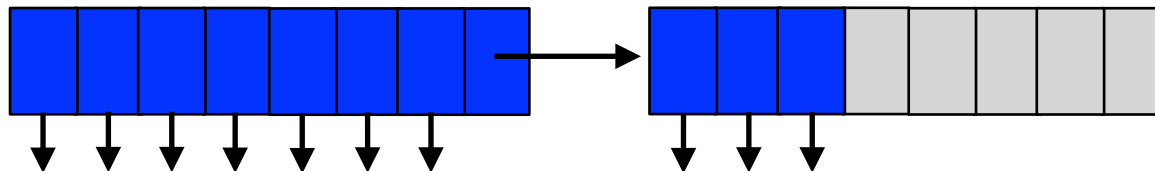
Free space management

Similar to free main memory management

- **Bit vectors** indicate for each block if it is used or not

- **Linked Lists** represent free blocks

  - linking information can be stored in the free blocks

  - Optimization: information on contiguous block is not stored separately but in one single piece

  - Optimization: one free block contains many block numbers of additional free blocks and possibly also the block number of an additional block containing the numbers of free blocks

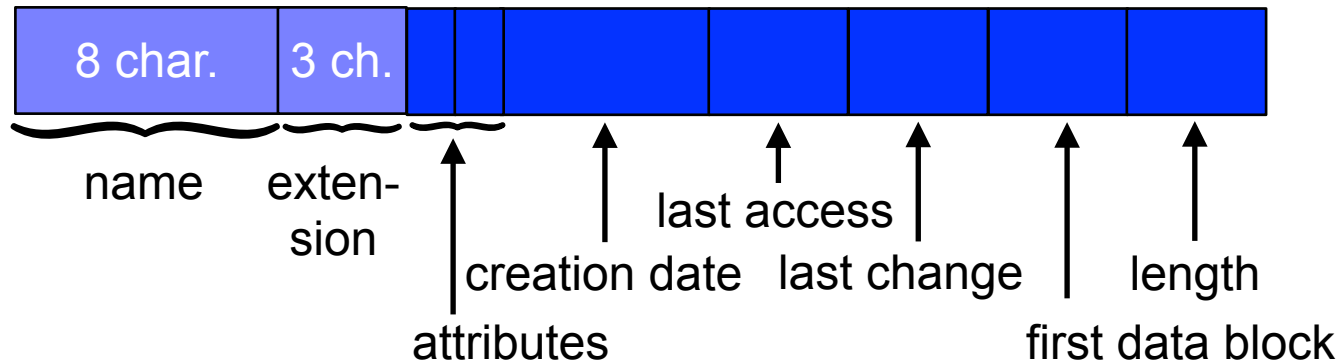free block list with references

free blocks

Free space management (2)

- **Tree sequential** storage of free block sequences
  - Enables faster search for a free sequence of blocks of a given size
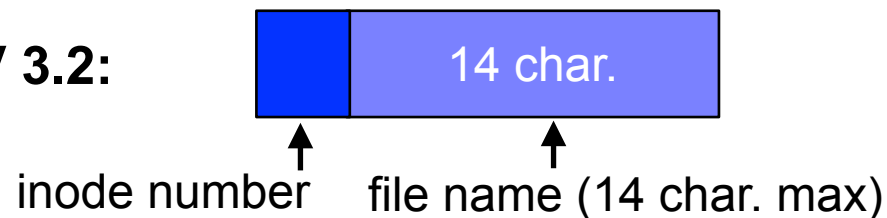 - Used e.g. in the SGI XFS file system

Norwegian University of Science and Technology

Directory management: lists

- Entries of identical length stored one after the other in a list, e.g.

  FAT32:

  | 8 char. | 3 ch. | | | | | | | |

  name   exten-sion   attributes   creation date   last access   last change   first data block   length

  *long file names in **VFAT** use multiple directory entries*

 **Unix System V 3.2:**

  | | 14 char. |

  inode number   file name (14 char. max)

- Problems:
  - Linear search for a given entry required
  - When sorting the list: fast search, insertion overhead

Norwegian University of Science and Technology

Using hash functions

- Function maps file name to index in directory list
- Enables faster access to the entry
  - no linear search required
- Simple (but bad…) example:
  ($\sum$ character values) mod N

index   directory entries

0

file name

N-1

hash function
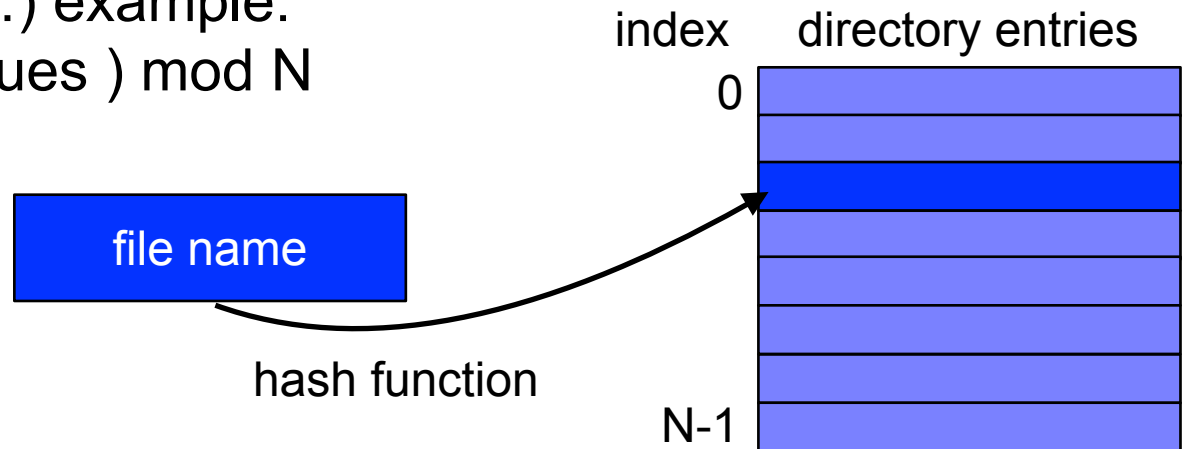
- Problems:
  - Collisions (multiple file names mapped to the same entry)
  - Adaptation of the list size required if list is full

List elements with variable length

- Example: used in 4.2 BSD, System V Rel. 4, etc.

inode number

name length

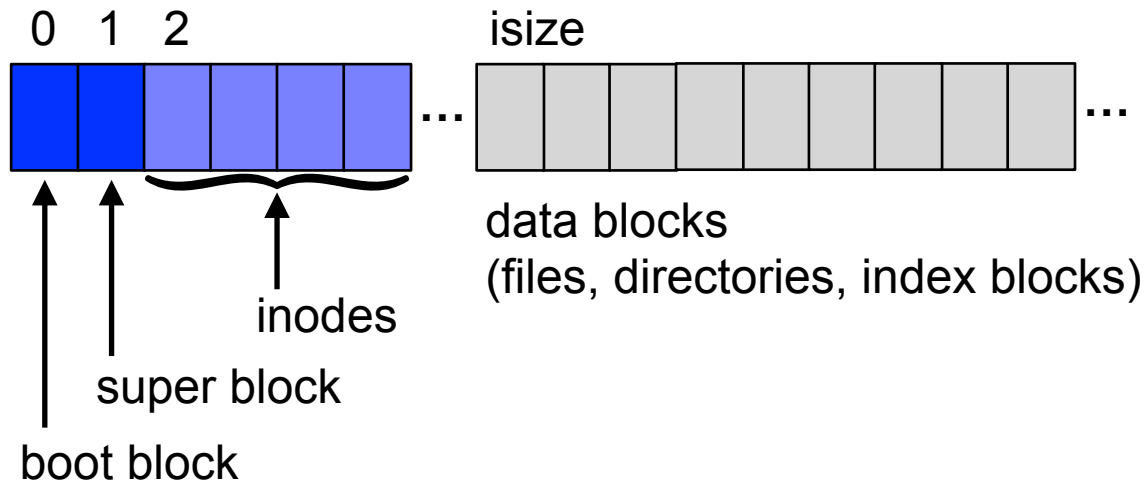offset to next valid entry

name
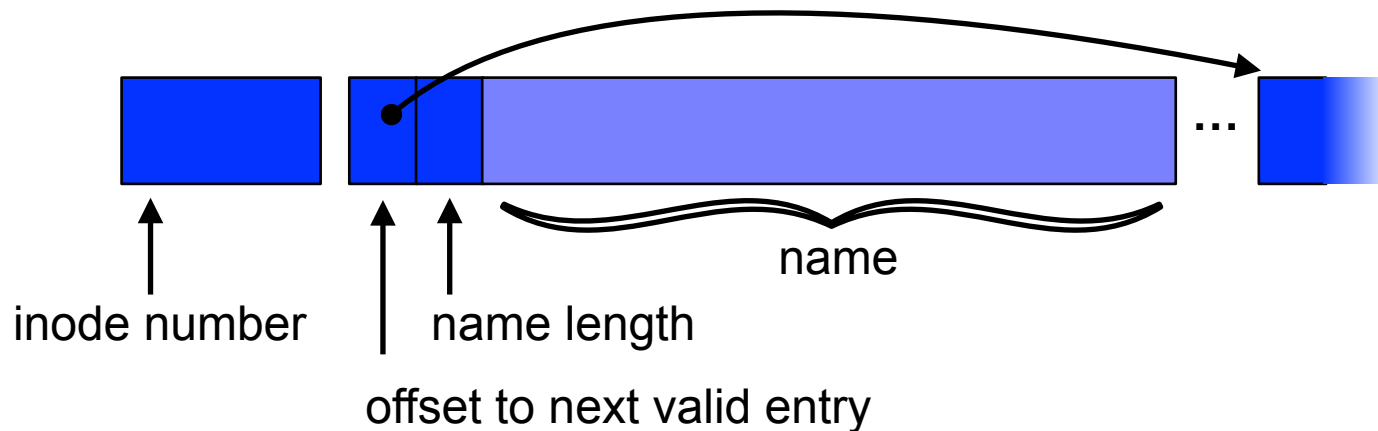
- Problems:
  - management of free entries in the list
  - fragmentation (need for compaction etc.)

Unix example: System V file system

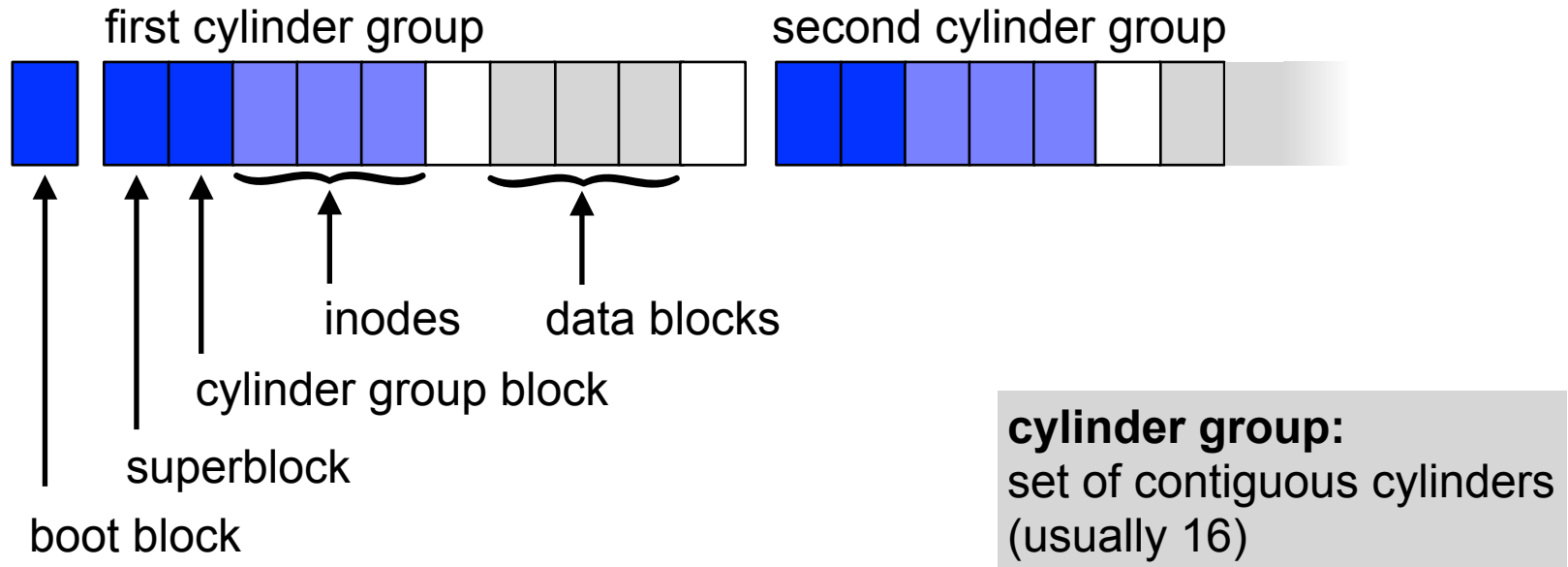- Block organization


```
0   1   2                    isize
┌───┬───┬───┬───┬───┬───┐      ┌───┬───┬───┬───┬───┬───┬───┬───┬───┐
│   │   │   │   │   │   │ ...  │   │   │   │   │   │   │   │   │   │ ...
└───┴───┴───┴───┴───┴───┘      └───┴───┴───┴───┴───┴───┴───┴───┴───┘
  ↑   ↑       ↑                data blocks
  │   │       │                (files, directories, index blocks)
  │   │     inodes
  │  super block
boot block
```

- **Boot block** contains information used to load the OS

- **Superblock** contains management data for a file system:
  - number of blocks and ***inodes***
  - number and list of free blocks and inodes
  - attributes (e.g. flag indicating the file system was modified)
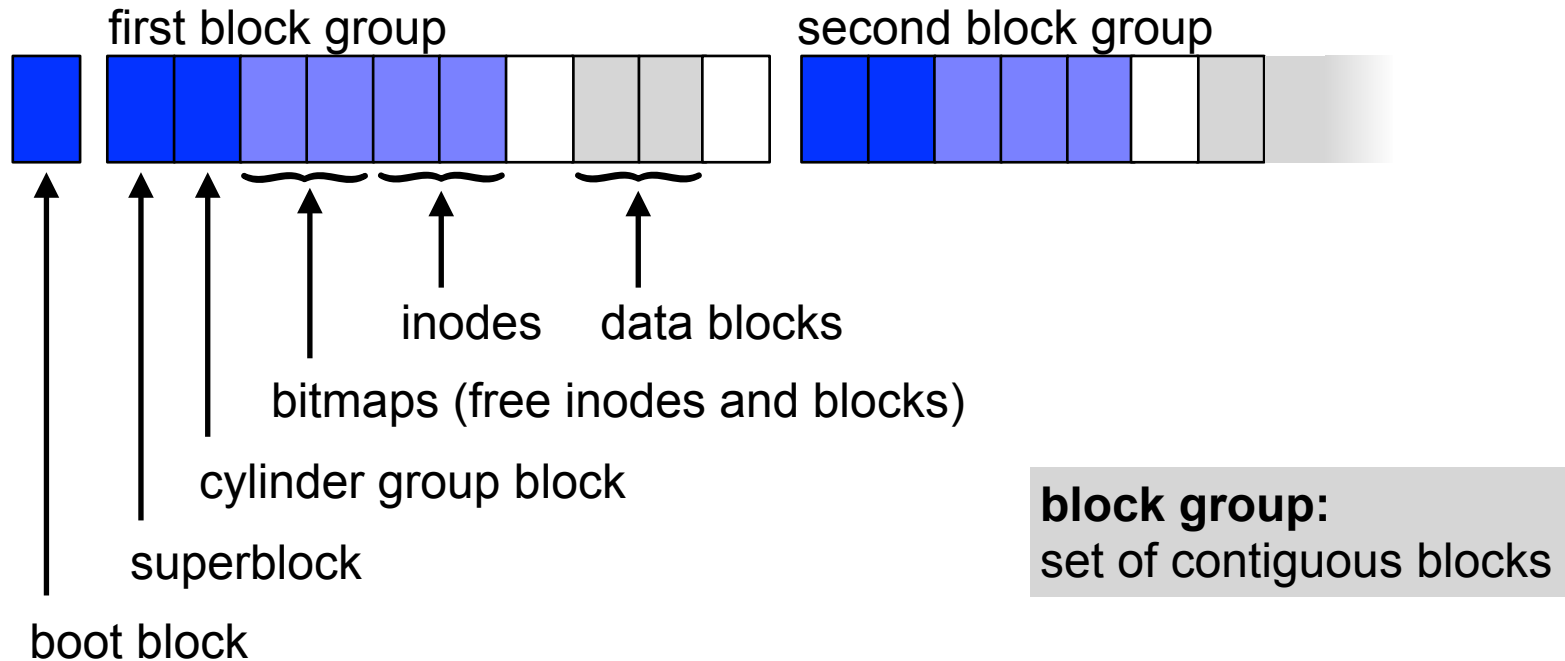
Unix example: Berkeley Fast File System

- Block organization (used from 4.2 BSD Unix onwards)

first cylinder group   second cylinder group

inodes   data blocks

cylinder group block

superblock

boot block

cylinder group:
set of contiguous cylinders
(usually 16)

- Copy of the superblock is stored in every cylinder group
- One file is stored inside a single cylinder group if possible
- Directories are distributed, files of a directory are stored together

- Advantage: reduced positioning time

Unix example: Linux ext2/3/4 file system

- Block organization

first block group   second block group

inodes   data blocks

bitmaps (free inodes and blocks)

cylinder group block

superblock

boot block

block group:
set of contiguous blocks

- Similar layout to BSD FFS
- Block groups are independent of cylinders

Norwegian University of Science and Technology

Conclusion

File systems are an **operating system abstraction**

- Logically related information is represented and stored as a file
- Often uses a hierarchical directory structure to organize data

... are **influenced by the hardware**

- Minimization of positioning times for disks
- Wear leveling for Flash memories

... are **influenced by the application profile**

- Block size
    - too small → management data structures can lead to *performance loss*
    - too big → fragmentation wastes disk space
- Structure of directories
    - no hash function → long search
    - using a hash function → more administrative overhead

References

[1] Karels, M.; M. K. McKusick (September 1986).
Towards a Compatible File System Interface.
Proceedings of the European UNIX Users Group Meeting,
Manchester, England: EUUG. pp. 481–496

[2] Apple Inc. Inside Macintosh Volume 2: Files.

[3] Maurice J. Bach.
Design of the UNIX Operating System.
Prentice Hall 1986, ISBN 978-0132017992

[4] Marshall Kirk McKusick, William Joy, Samuel Leffler and Robert
Fabry (1984).
A Fast File System for UNIX" (PDF)
ACM Transactions on Computer Systems. 2 (3): 181–197

[5] R. Card, T. Y. Ts'o, and S. Tweedie.
Design and implementation of the second extended filesystem
Proceedings of the 1994 Amsterdam Linux Conference, 1994.

NTNU | Norwegian University of Science and Technology