Chapter 10

File System

10.1 Motivation

- We need the possibility to store large amounts of data reliably and permanently.
- **Files** (logical resources)
- A file is a collection of logical data entities, the so-called **records**.
- Depending on the application area a file system supports
  - Records of constant or variable length
  - Files of dynamically changing size
  - Modifiable internal structure of files
  - Variable number of files on a volume
  - Oversized files spanning across more than one volume

Storage media

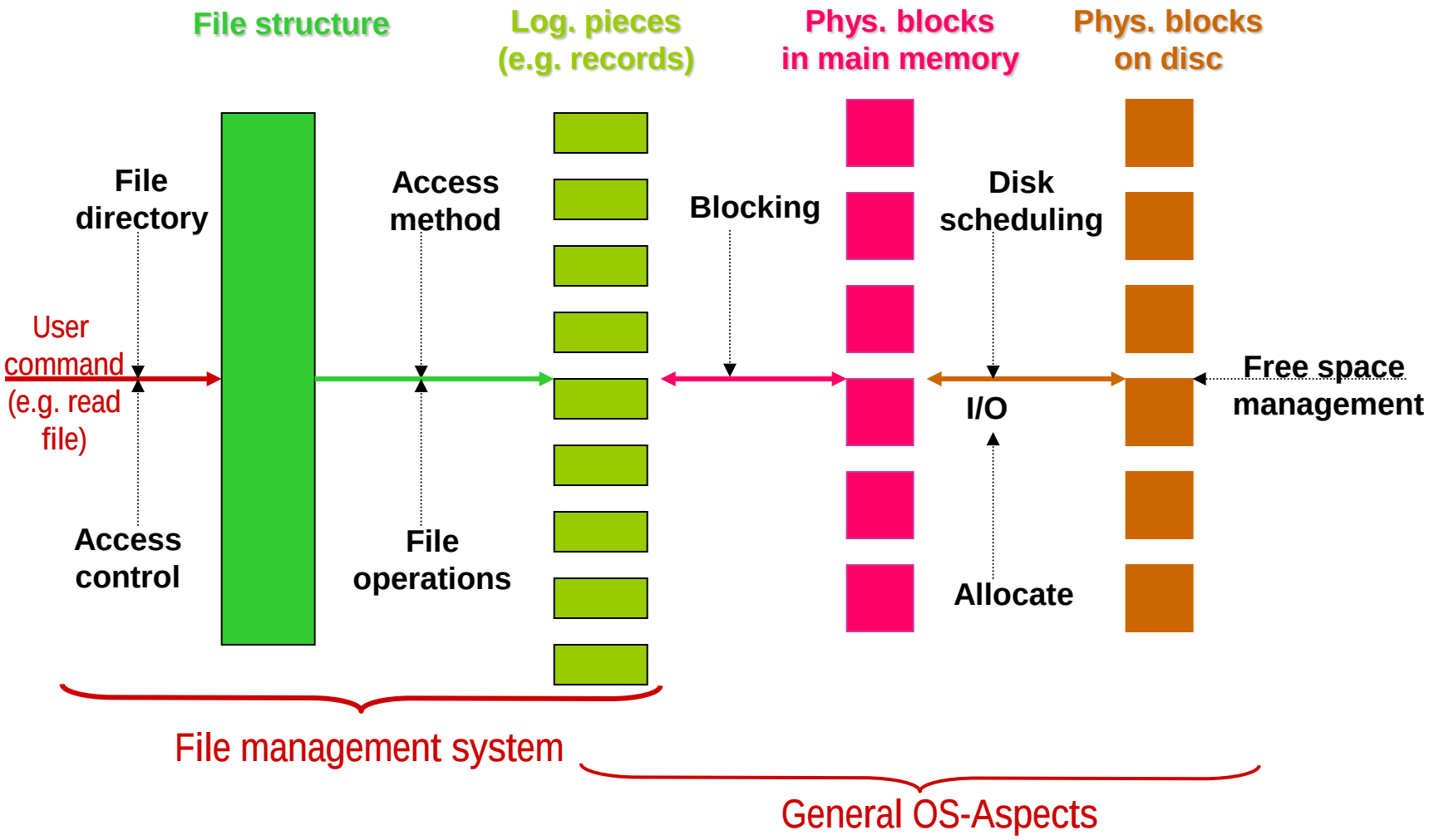
- Files should be stored
  - on non-volatile media with relatively
  - short access times at
  - low cost
- and allow
  - reading and writing access.

- The currently still best suited medium is the magnetic disk
  - For smaller volumes also: Flash Memory (SSD)
  - For larger amounts: Tape libraries
  - For archiving or read-only access: CD-ROM, DVD, Blueray

Storage media

- Following, we will focus on disk drives as main medium for file storage.
- DVDs and CD-ROMs have a similar structure.
- SSDs also have a similar structure, but certain specialties have to be obeyed (addressed separately in the lecture on I/O).
- Only tapes are fundamentally different, since they allow only sequential access due to their one-dimensional linear structure.

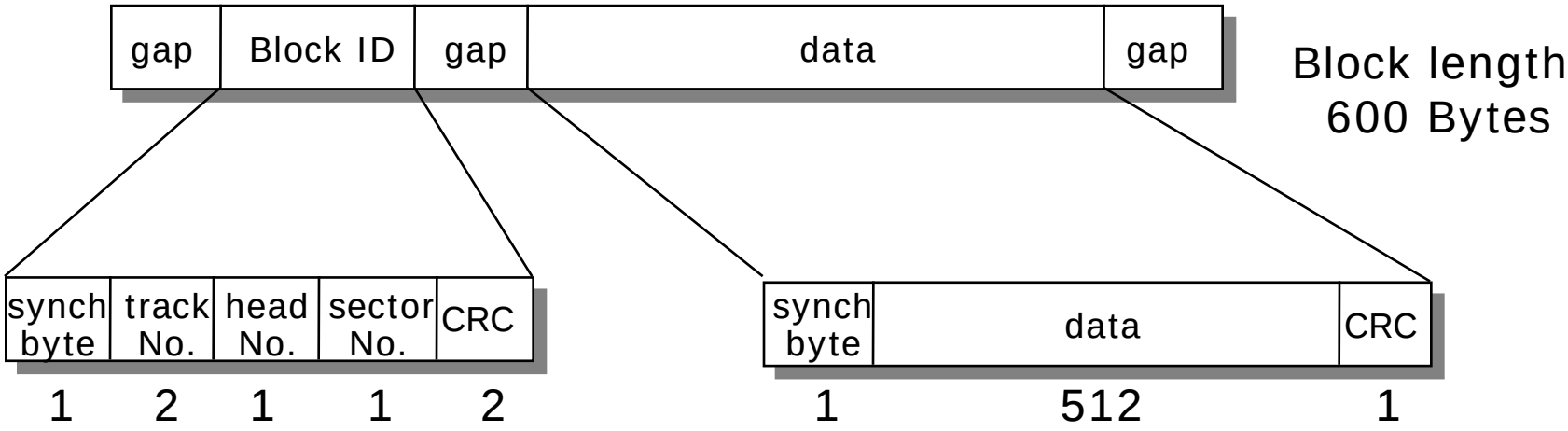
Interplay


File structure
Log. pieces (e.g. records)
Phys. blocks in main memory
Phys. blocks on disc
File directory
Access method
Blocking
Disk scheduling
User command (e.g. read file)
I/O
Free space management
Access control
File operations
Allocate
File management system
General OS-Aspects

Structure of blocks

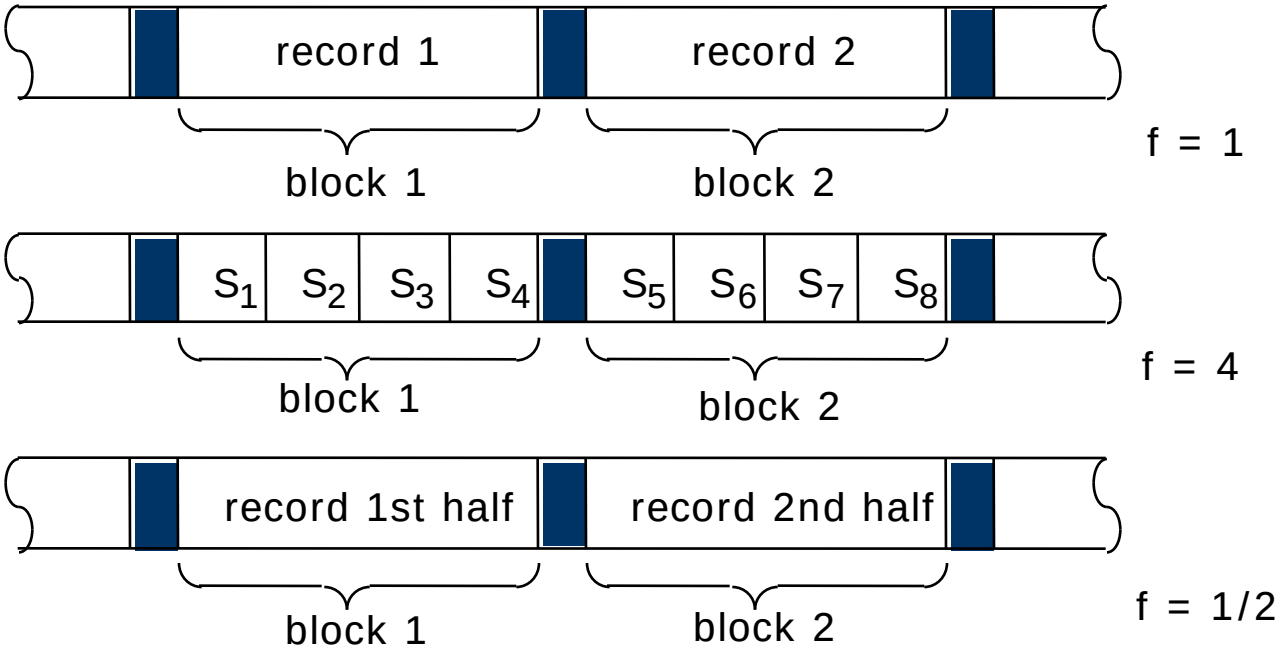
- **Blocks** (or sectors) are the smallest addressable units of a disk.
- They are spatially separated by so called block gaps.
- Each individual block contains (in addition to the data)
  - A block identifier which is its physical location in the simplest case
  - Check fields for error detection (CRC, cyclic redundancy check).

- Example:



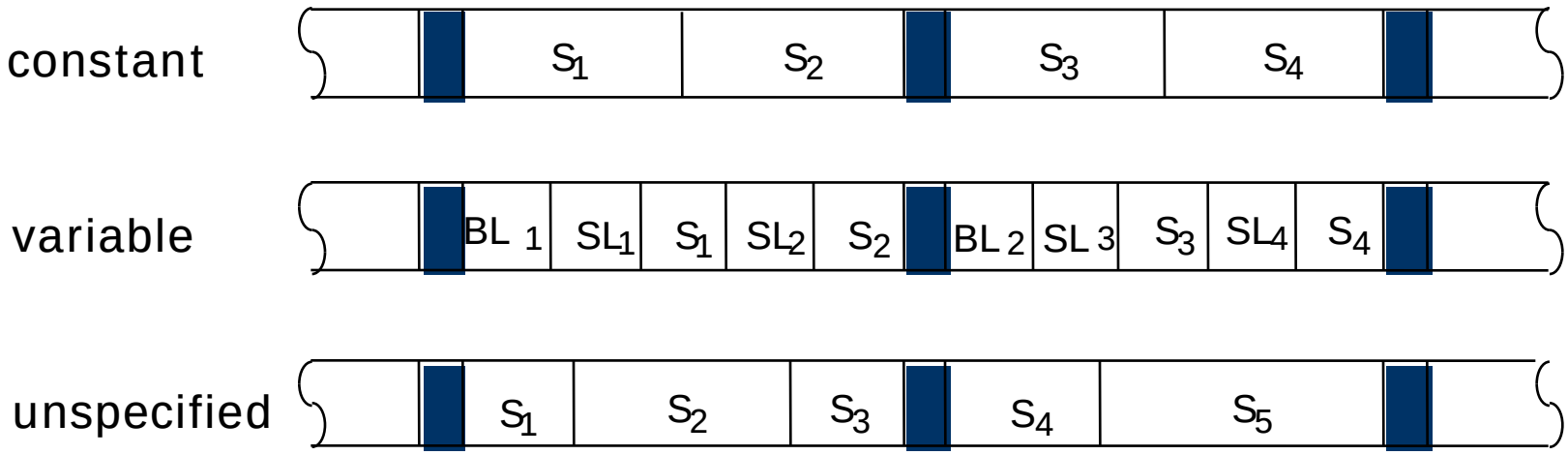
Blocks and records

- **Blocks** are the elementary units of a disk (physical units), while **records** are the elementary units of a file (logical units).
- If records are required to be of arbitrary length, we need a flexible assignment of records to blocks.
- For constant record length the ratio of block length to record length is called **blocking factor *f***.



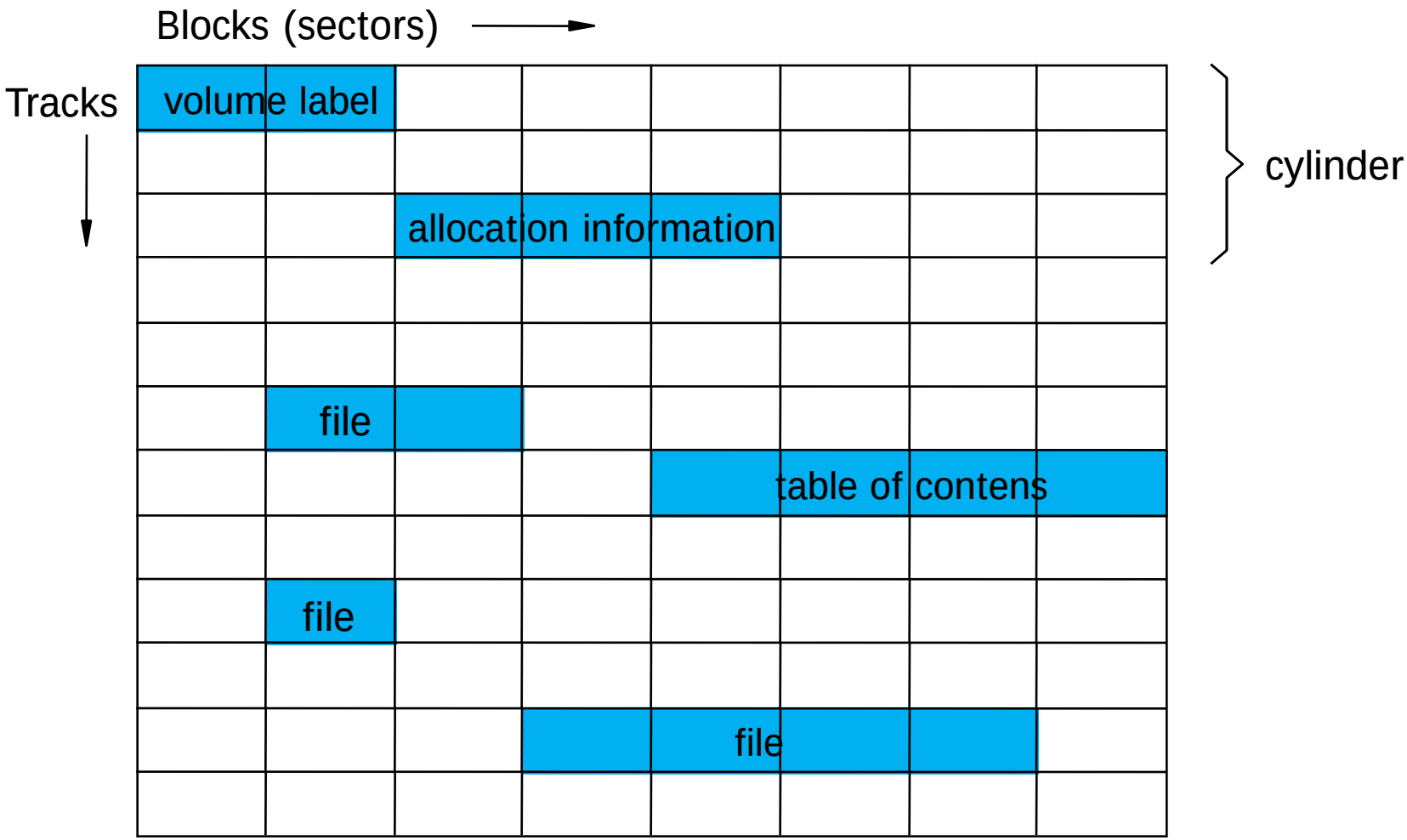
Blocks and records

- Records do not need to have constant length.
- The following record formats are possible:



- It is also possible that there is no record structure at all, i.e. the file is an unstructured sequence of bytes (e.g. Unix).

Allocation of a disk storage



Volume label

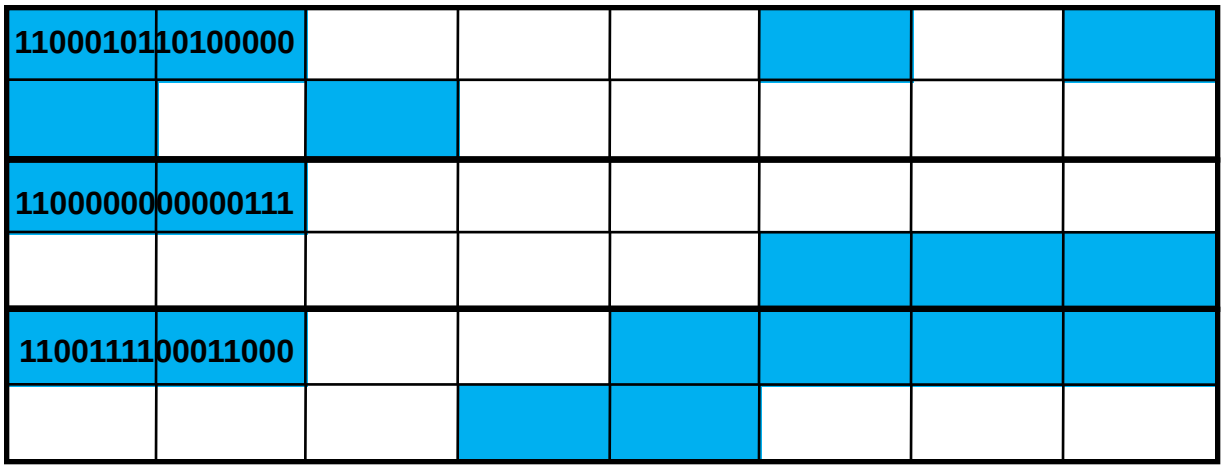
- Identifier of volume
- Date of activation
- Capacity
- Physical layout
- Bad blocks
- Link to allocation information (or allocation information itself)
- Link to table of contents (or table of contents itself)

- The volume label is placed at a well-defined position (e.g. first block) and is created at activation (i.e. formatting) time.

Allocation information (free and allocated blocks)

- Vector- or list-based
- contiguous or scattered

- Example:
  - Vector (bit map) for allocated and free blocks, separate for each area (minimizing head movements).



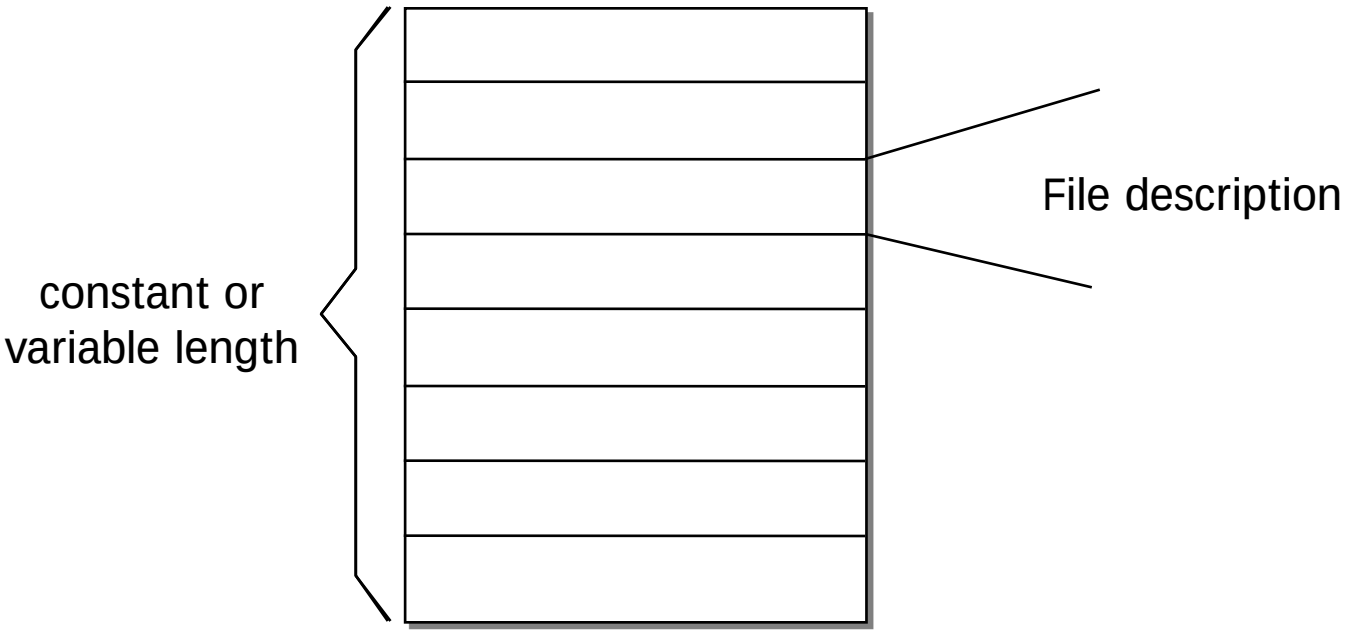
Free space list as a separate table

adress (block no.)	length
3	16
22	9
32	10
44	9
57	8

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

10.3 Table of contents (file directory)

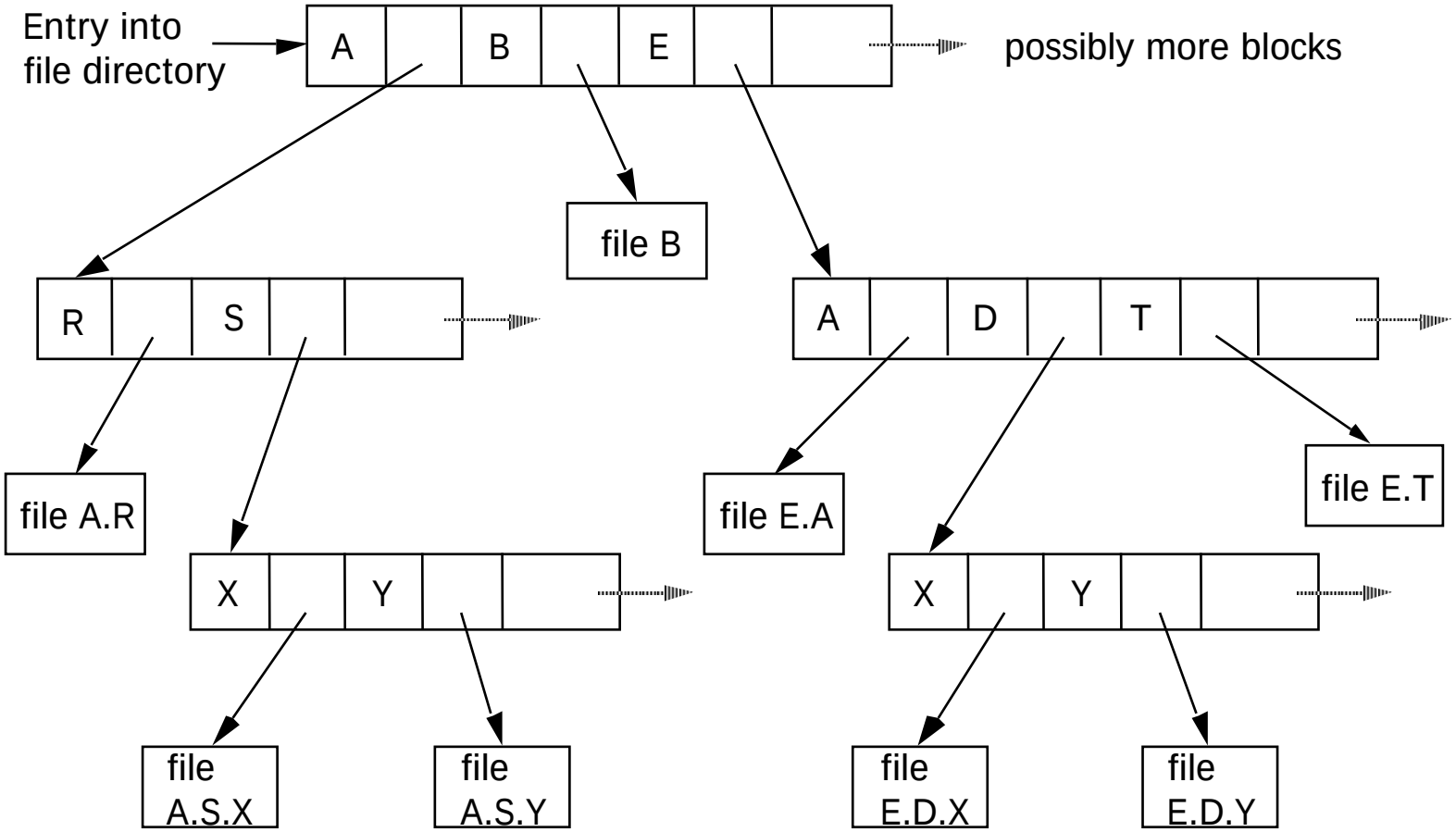
- The file directory contains the list of the descriptions of all files and is stored on the volume.
- Flat directory structure
  - In the simplest case it consists of a one-dimensional table



- For large volumes and many files the flat structure is awkward and unhandy (for the human user as well as for accessing programs).

File directory

- Structured directories



File description

- The file description contains all necessary information concerning the particular file:
  - file name
  - Organization
  - creation date
  - Owner
  - access rights
  - date of last access
  - date of last change (write access)
  - position of file (or its parts)
  - Size
  - ....

Access rights

- Access rights are determined by the owner, who usually is also the creator of the file.
- If read (r) and write (w) are offered as basic rights, access rights could be specified as follows:

	file 1	file 2	file 3	file 4
user(group) A	r, w	w		
user(group) B	r	r, w	r	r, w
user(group) C		r	r	
user(group) D		r		

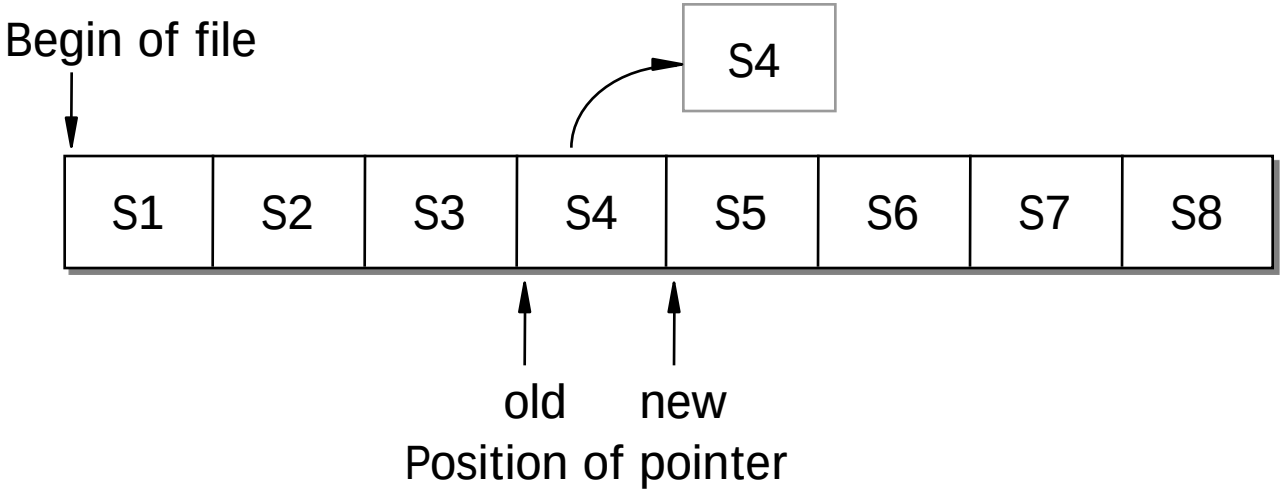
- Advanced differentiation of access rights
  - Execute (for program files)
  - Change access rights (reserved to owner)
  - Write differentiated between „update“ or „append“
  - Delete
  - ...

10.4 File organization

- File organization concerns the internal structure of the file.
- The organization specifies in which way access to individual records is possible.
- We distinguish:
  - **Sequential file organization**
    - The records are accessed sequentially, i.e. one after the other.
  - **Direct file organization**
    - Random access to arbitrary records
  - **Index-sequential file organization**
    - Sequential as well as random access
- More than one form of organization may be offered simultaneously and mapped to a basic form of organization.

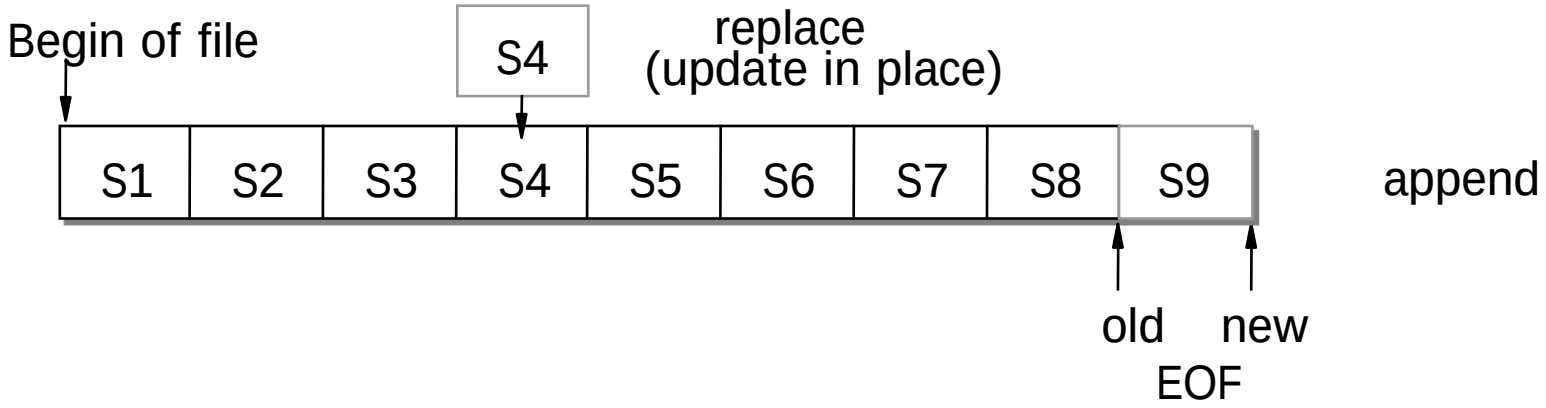
10.4.1 Sequential file organisation

- There is an order of the records that determines the access.
- It is the mandatory organization for files on magnetic tapes.
- It could also be applied for disk storage devices.
- There is a pointer which identifies the current record and can be moved explicitly or implicitly by special operations.
- The access (e.g. read) always relates to the current position of the pointer:



Sequential file organization

- Writing to a file is usually only possible at the end of the file.
- Only if a record can be replaced by another record of the same length, writing within the file is possible.



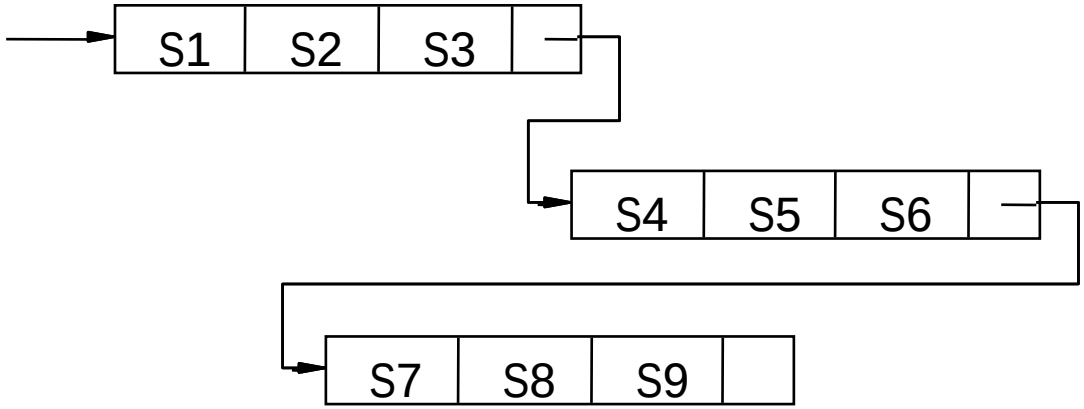
- Often, there are explicit operations to move the pointer:
  - next advance the pointer by one
  - previous set back by one (*sometimes not offered)*
  - reset set back to begin

Sequential files on disk

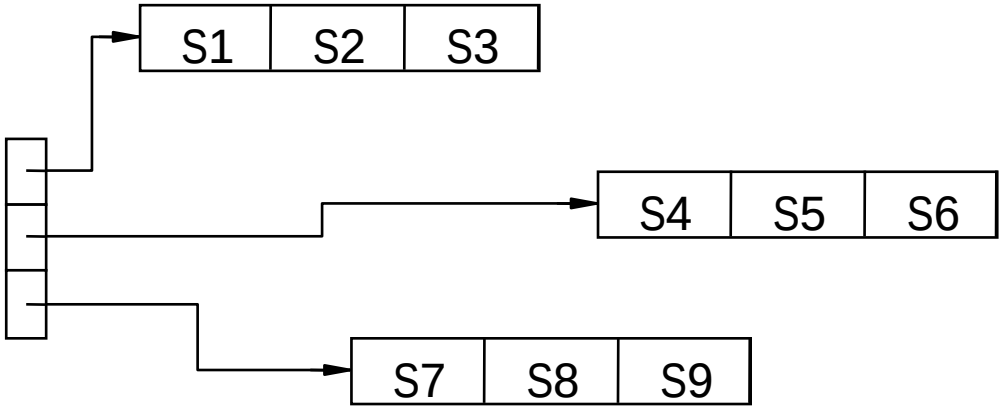
- Disk devices offer some choices for storing sequential files:
  - Contiguous allocation
    - The file occupies consecutive blocks on the disk.
  - Noncontiguous allocation
    - The file occupies arbitrary blocks on the disk.
    - The ordering of the blocks can be achieved in two different ways:
      - Linked allocation (chaining)
        - direct (integrated) chaining of blocks
        - separate chaining in a table (e.g. FAT in MS-DOS / Windows)
      - Indexed allocation

Sequential files on disk

- blocks directly chained

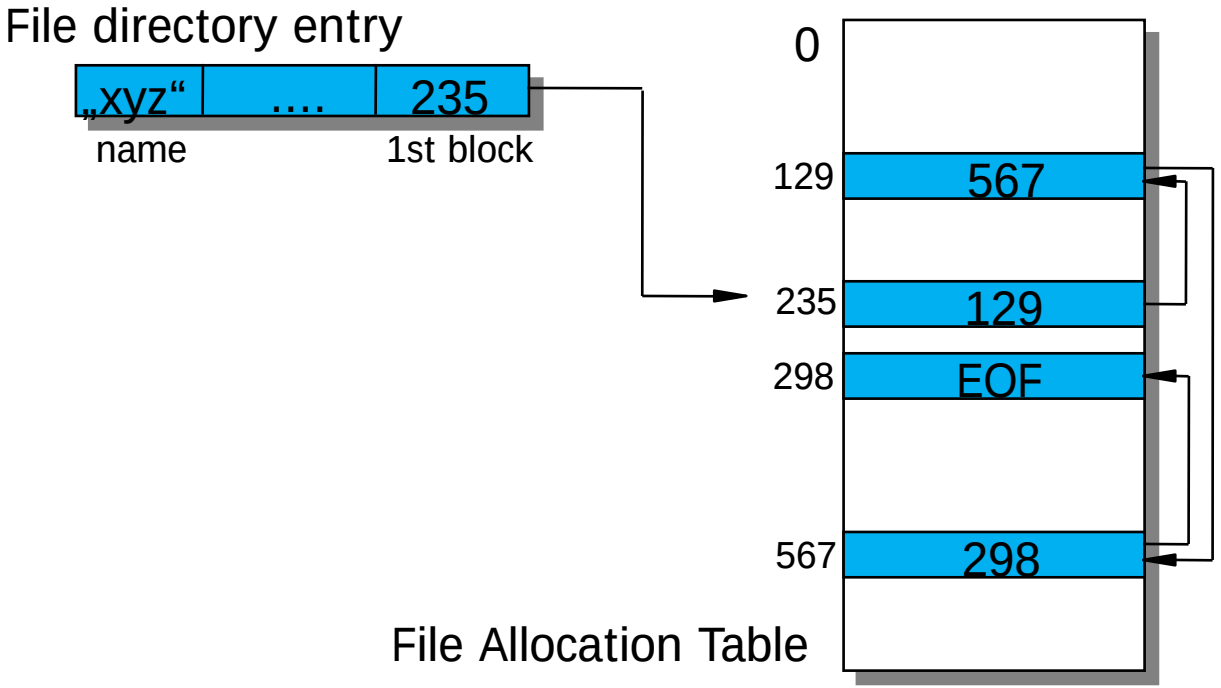


- blocks managed by usage of an index block



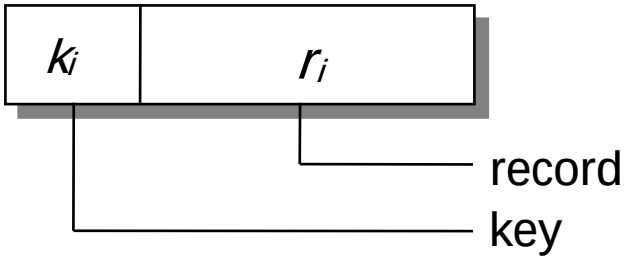
Example

- The FAT file system of MS-DOS uses separate chaining.
- The chaining is done in a File Allocation Table (FAT) that provides an entry for each block.
- For performance reasons it must be kept in main memory permanently.



10.4.2 Direct file organization

- Direct access to a record is organized using a **key.**

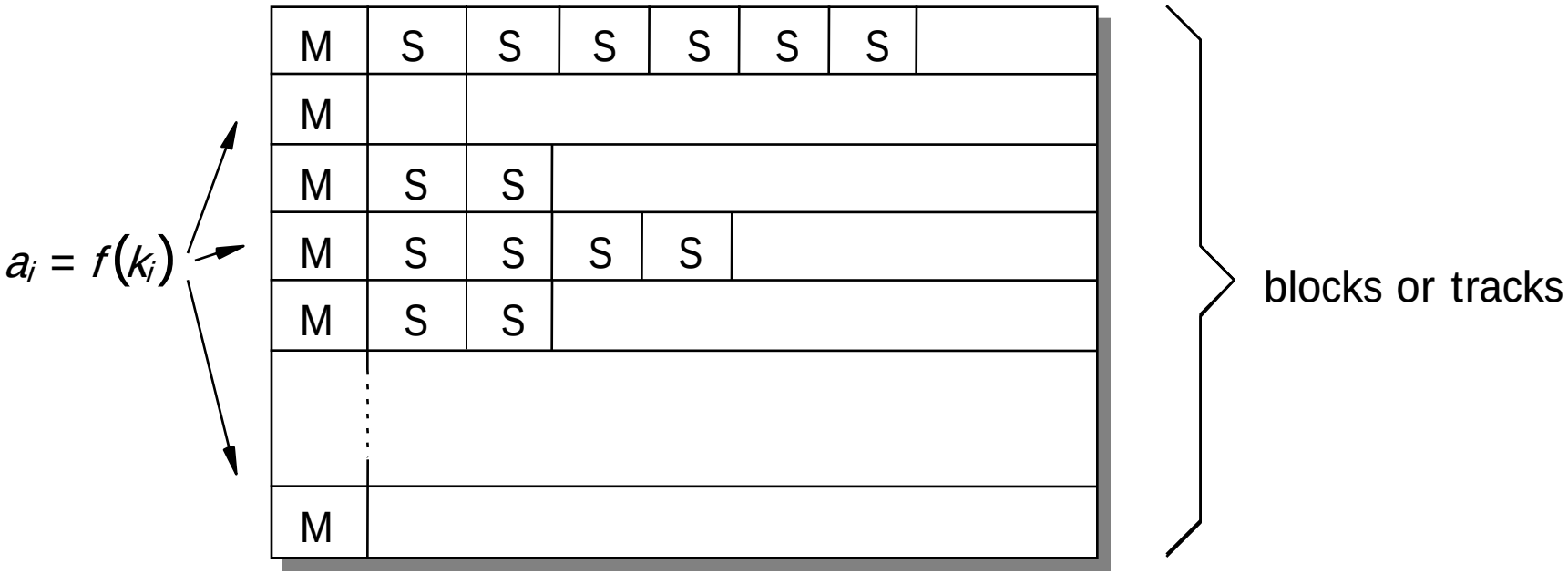


- The address (block- or track number) is calculated based on the key value.

  ⇒ Hash function $a_i = f(k_i)$, *e.g.* $a_i = k_i \bmod n$

- The address calculated (block number) is not necessarily the physical block number. An additional intermediate mapping is possible.
- Blocks or tracks serve as containers for several records, i.e. for all those that are mapped to the same hash address.
- Only in case of an overflow we need to resolve the hash collision.

Direct file organization



blocks or tracks

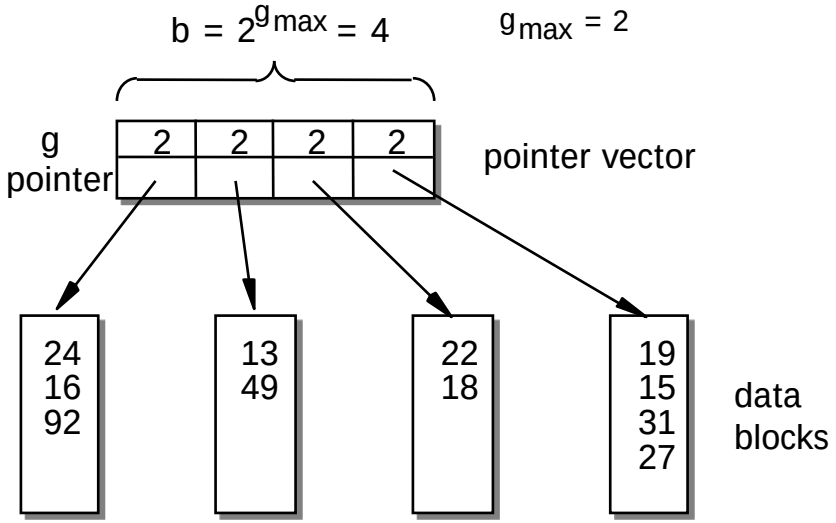
- Collision resolution, e.g. linearly by $a_{i+1} = (a_i + d) \bmod n$

Direct file organization

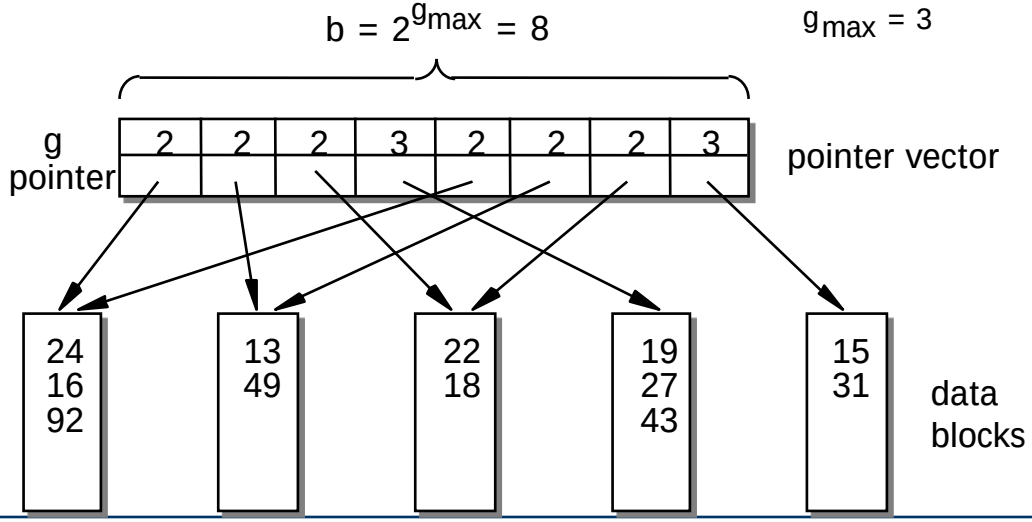
- If files grow arbitrarily, the hash table will overflow at some time.
- Then we need a costly reorganization (rehashing).
- To avoid that, we can employ the ***extendible hashing.***
- It allows an incremental extension of the hash table without a complete reorganization.
- For that we need an additional stage of indirection, e.g. the hash function leads first into a component of a vector of pointers.
- As hash function we use $a_i = k_i \bmod 2^g$, *e.g. the keys are discriminated according to their last g* binary digits.
- If there is an overflow of one of these hash buckets, we create a new bucket and the content is distributed to the two buckets (previous and new) according to the „refined“ hash function.
- To ensure correct addressing, we increase *g by one (length of pointer vector doubles), and the* pointers need to be translated accordingly.

Example

- before extension
  (key 43 is inserted)

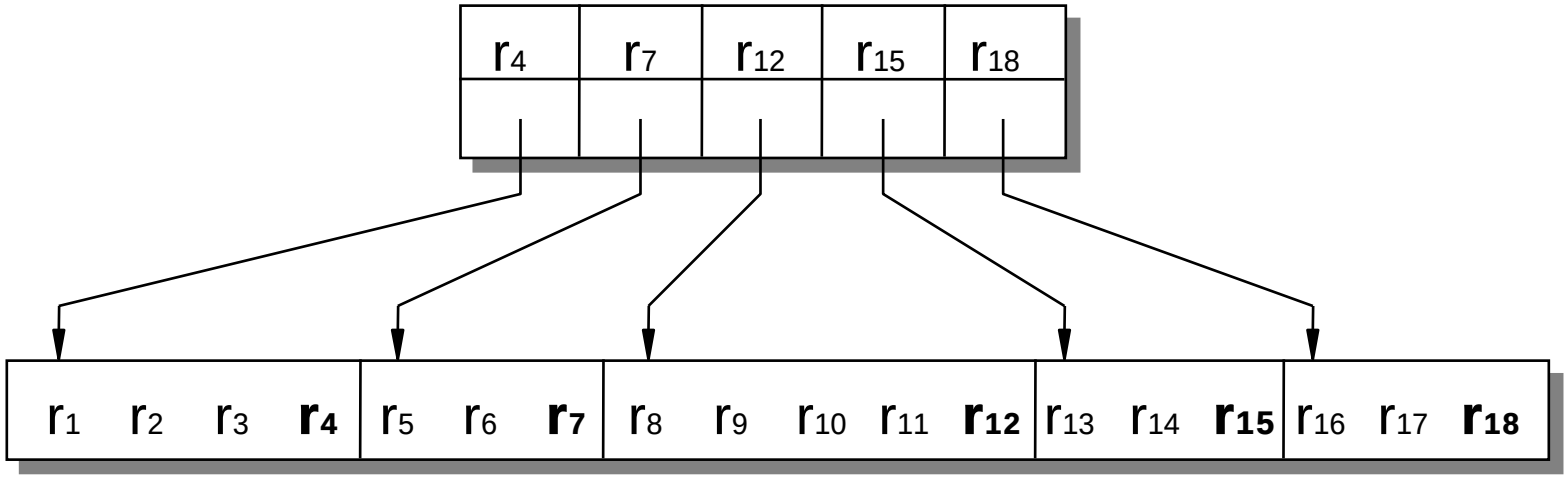


- after extension



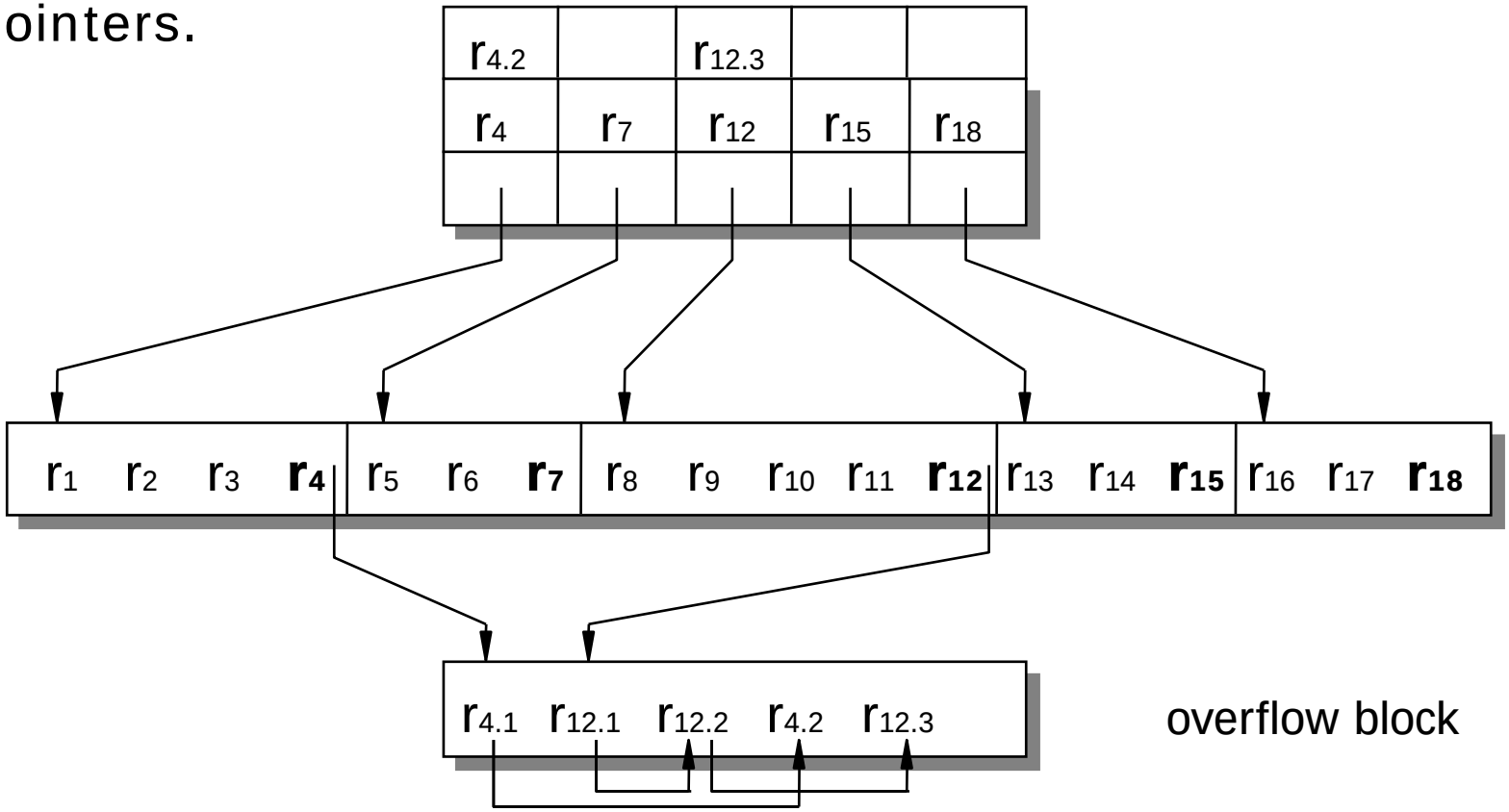
10.4.3 Index-sequential file organization

- Some data sets require both sequential and direct processing (at different times).
- That leads to a mixture of sequential and direct (indexed) organization, the **index-sequential file** organization.
- Although the records of the file are sequentially stored on the storage media, there is additionally direct access supported by appropriate data structures.
- In its original form, there is exactly one index stage which holds the largest key of each block.



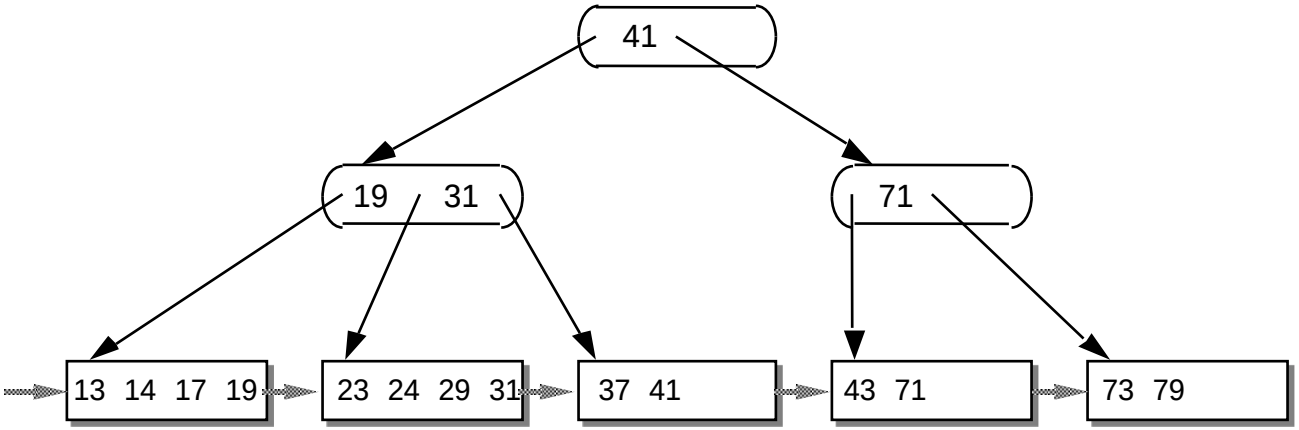
Index-sequential Organization

- With dynamic operations (insertion and deletion of records) blocks may overflow.
- Then we need to provide overflow blocks to store the records that don‘t fit in and have to insert appropriate reference pointers.



B*-Trees

- The usage of pointers with overflow blocks can significantly increase access times to individual records.
- Better are data structures that by their very nature support growing and shrinking.
- The *B*-Tree is a variant of the B-Tree. It contains records only in the leaves.*
- The internal nodes only contain keys and serve for the speed-up of the access.
- With regard to the degree of filling and shape conservation, the B*-tree corresponds to the B-tree.



Properties of B*-Tree

- The nodes correspond to the blocks on the disk.
- Each node (block) is at least half full.
- Let be
  - c_i *the number of keys in an internal node i*
  - *m the minimal degree of filling for internal nodes (min. number of* keys)
  - c_i^* *the number of records in a leaf node i*
  - *m* the minimal degree of filling for leaves (min. number of records)*
- Then for all internal nodes *i (except the root):*

  $$m \leq c_i \leq 2m$$

  and for all leaf nodes *i:*

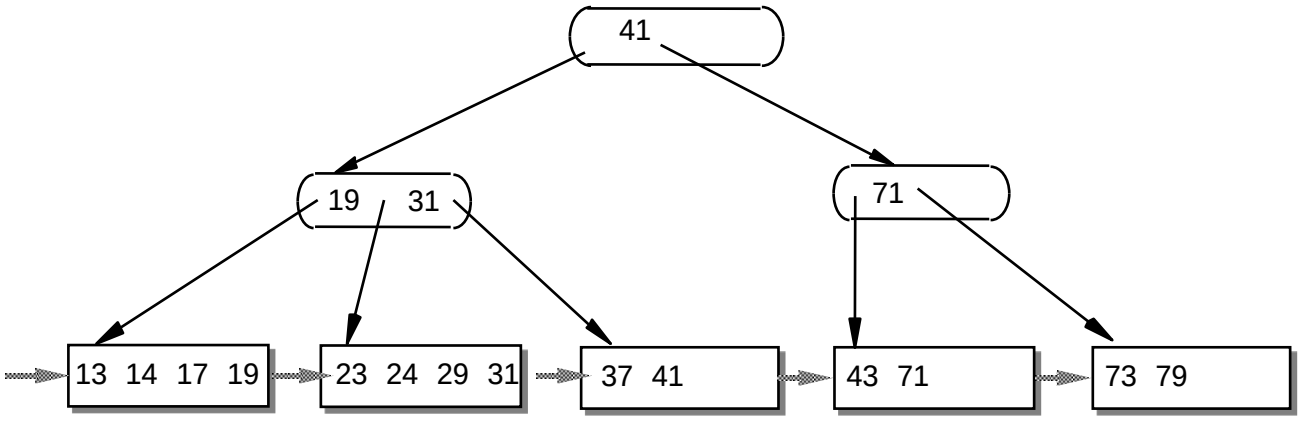
  $$m^* \leq c_i^* \leq 2m^*$$

  (For the example on the preceding slides we have *m = 1, m* = 2.)*

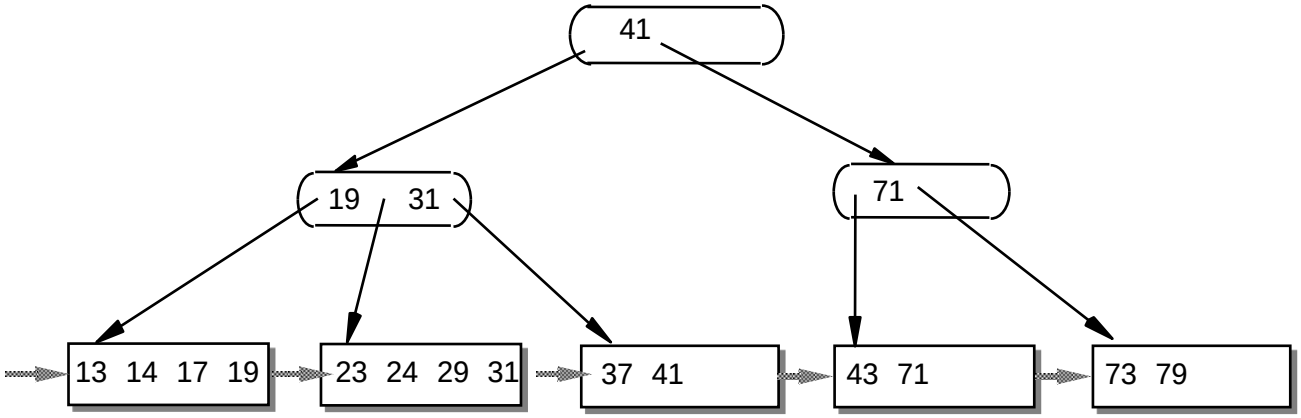
Insertion in B*-Tree

- Normal case: still free space in node
- Overflow case:
  - neighbor has enough space: load balancing with neighbor
  - both neighbors full: split node (allocate new block)

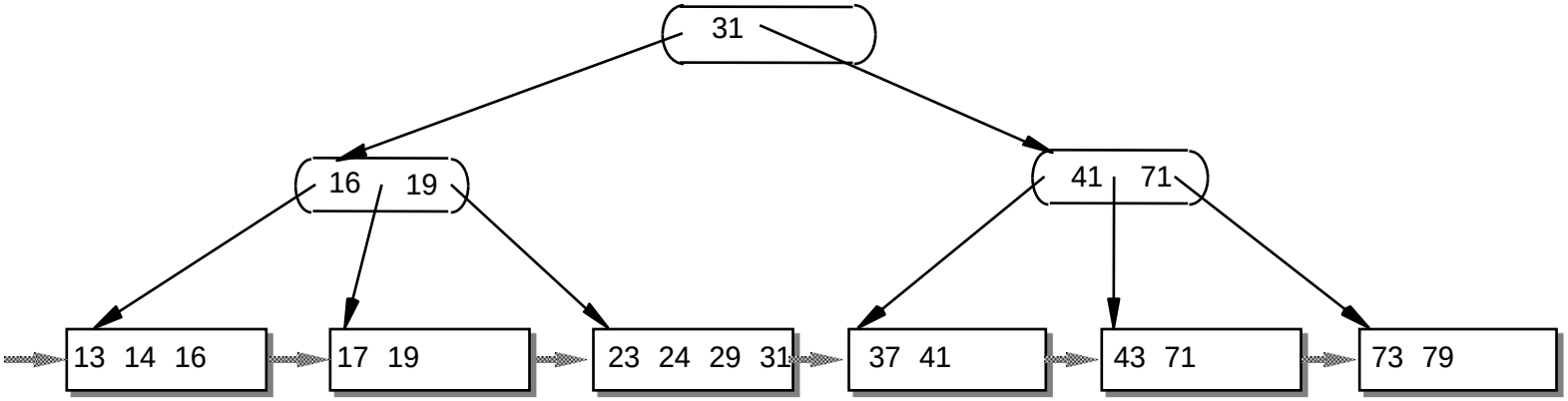
- B*-tree after insertion of a record with key 16?



Insertion in B*-Tree



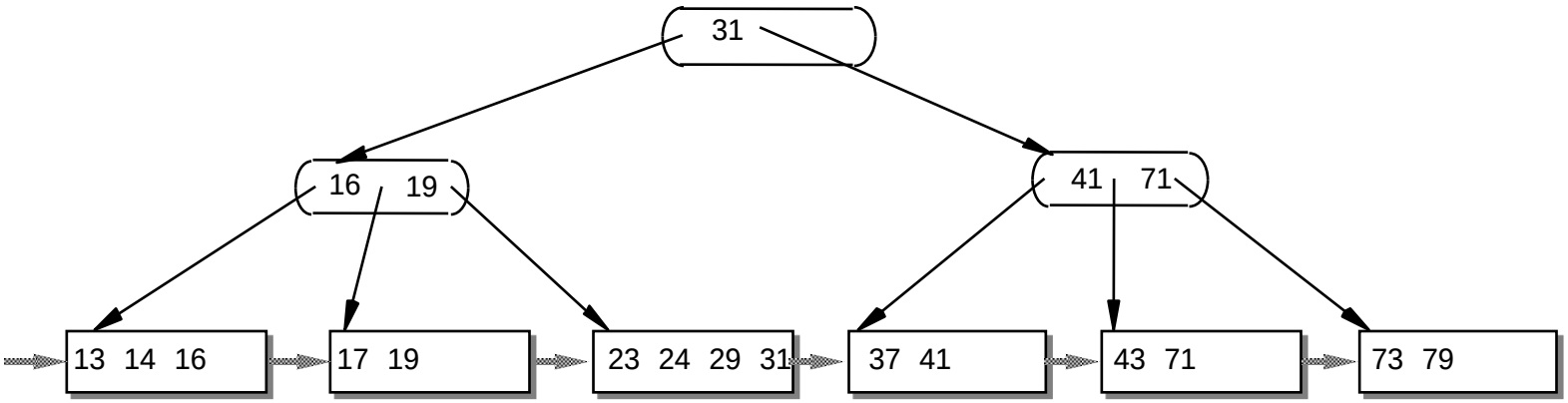
- B*-tree after insertion of a record with key 16 (node splitting at leaf level, load balancing at level above)



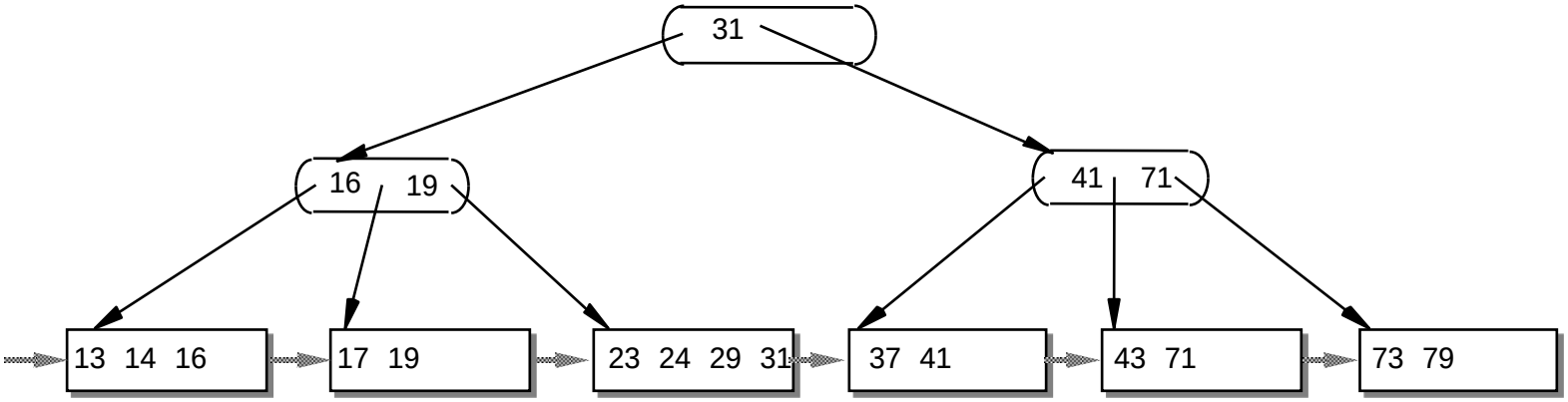
Deletion in B*-Tree

- Normal case: node remains at least half full
- Reconfiguration case (node falls below 50% filling):
  - neighbor more than half full: load balancing with neighbor
  - both neighbors exactly half full:
    merger with one the neighbors (release block)

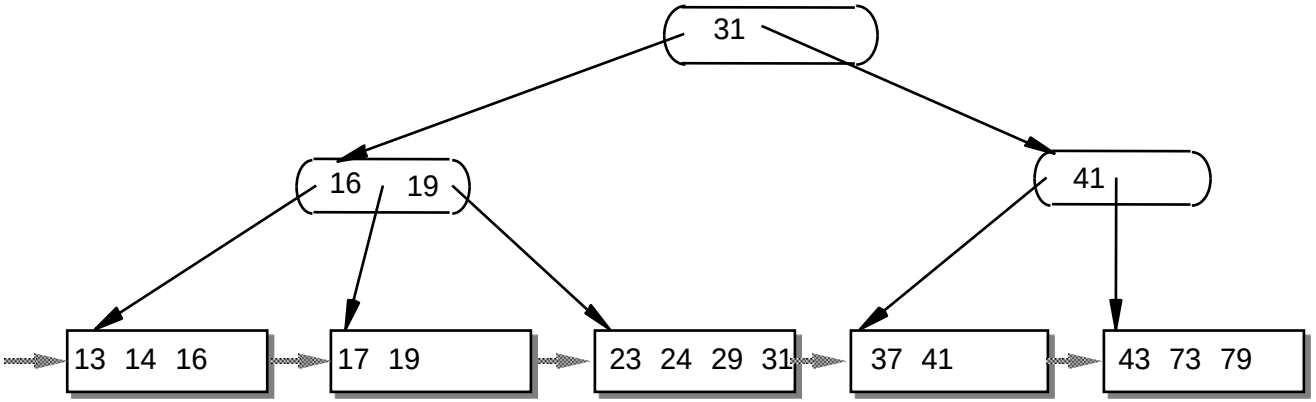
- B*-tree after deletion of record with key 71?



Deletion in B*-Tree

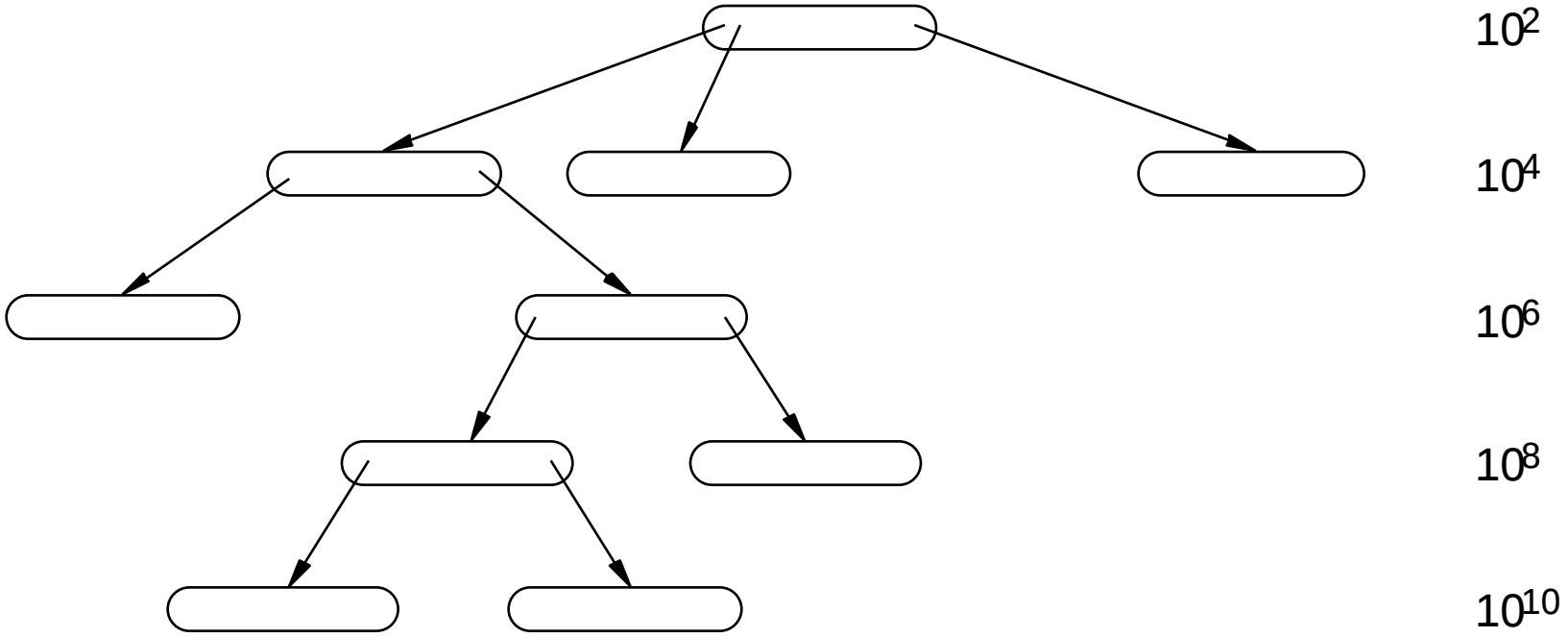


- B*-tree after deletion of record with key 71 (node merger at leaf level)



How tall do B*-Trees get?

- e.g. social security in China with ca. 10^9 entries
- With 40 bytes per entry (key and pointer) and a block size of 4096 bytes we get a fan out degree of
  t = 4096/40 ≈ 100 (no. of keys per node)



- A B*-tree of height 4 is sufficient!

- Operations on files
  - Create
    - open
      - read
      - write
      - reset
      - lock
      - ...
    - close
    - read parameter
    - set parameter (e.g. access rights)
  - Delete

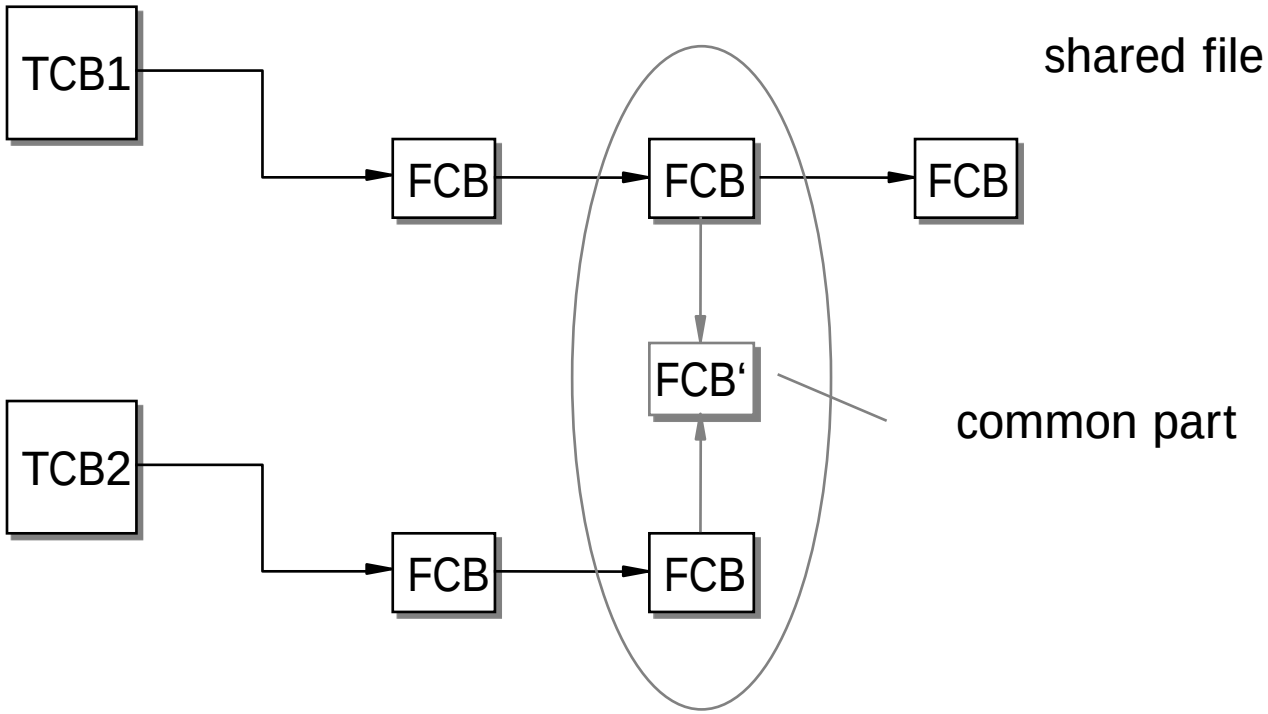
File control block

- Operation of a file requires some management information:
  - position pointer
  - current block address
  - references to buffer (in main memory)
  - degree of filling of buffer
  - lock information

- These data are stored in the **file control block** (FCB).
- The FCB is a data structure that is created at opening time of a file and deleted after closing.
- The thread or process control block contains references to the file control blocks of files that this process has currently opened.

Shared usage of files

- A file can be used by several threads at the same time.
- Since the FCB contains data that refers to the file as such, and data that is related to the respective user, we can split those two parts:



Buffering

- Since data are often accessed more than once, e.g. index blocks, it pays off to keep copies of those disk blocks in main memory (disk cache).
- Some operating systems use the whole otherwise unused main memory as disk cache e.g. Linux even swaps out program memory in favor of disk cache (if sysctl parameter vm.swappiness > 0; default is 60).
- (In addition, disk controllers usually have another, internal and transparent cache)
- At each access to a disk block the OS checks the cache whether the block is already present in main memory.

Buffering

- As replacement strategy at shortages we can employ the same algorithms we know from the virtual storage. (LRU, FIFO, ...).
  - Performance?
- If a modified block is written back to the disk only as part of a replacement, there is some danger of loss.
  - Why?
- Important blocks on which the consistency of the file system depends (directory blocks, index blocks) should be saved immediately after writing.
  - How?
- Sequential access can be exploited: *Read-Ahead* and *Free-Behind*.

10.6 Example: Unix File System

- **Hierarchic**
  - tree structure
  - file directories as internal nodes
  - files as leaves
  - no restrictions with regard to breadth or depth
- **Unified**
  - Almost all system objects are represented as files and accessed by using the file interface (files, catalogues, communication objects, devices).
  - syntactically equal treatment of all types, semantically as far as possible.
  - therefore programs are independent of the object type
- **Simple**
  - only few, but flexible file operations
  - simple file structure

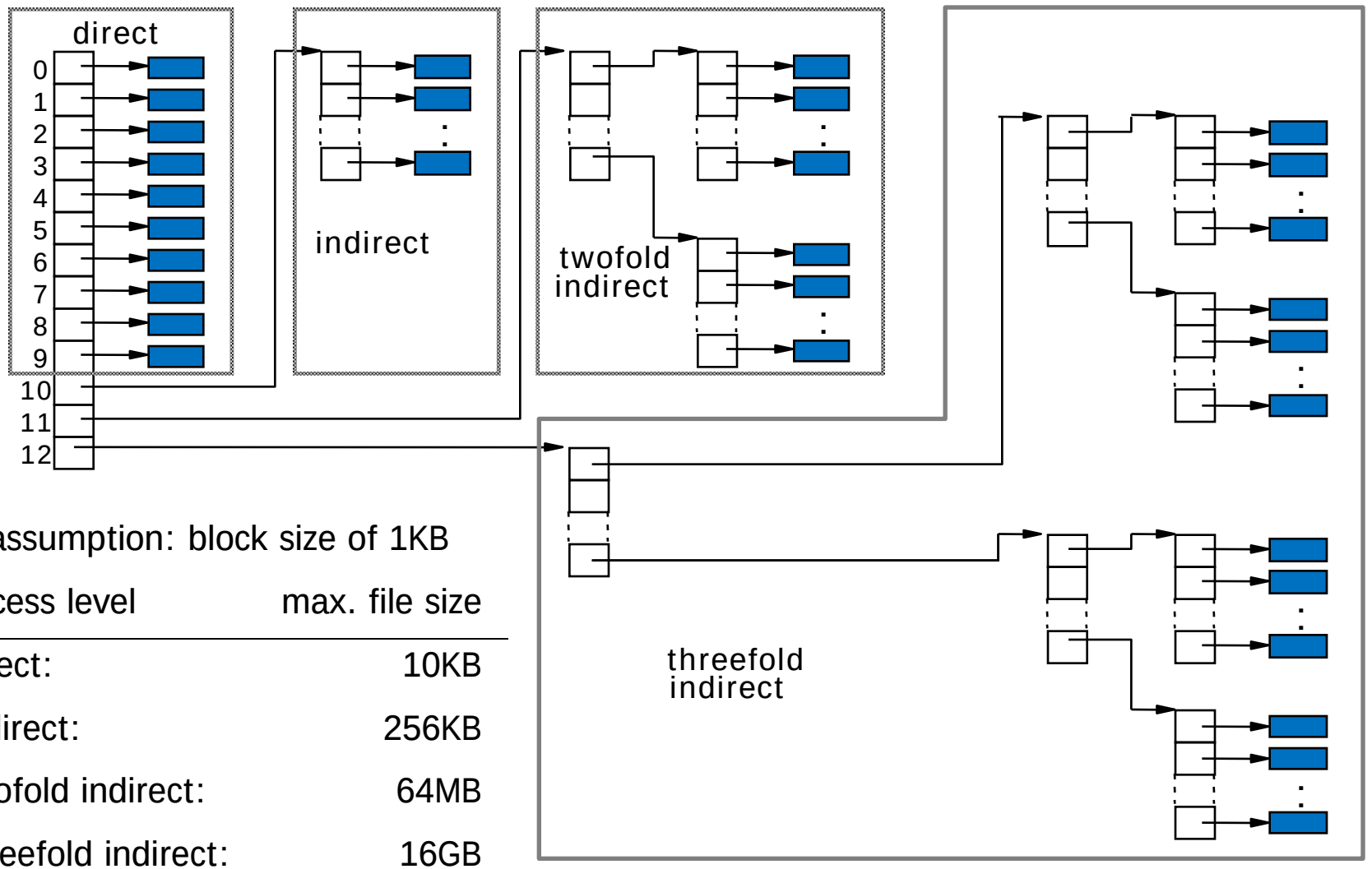
Files in Unix

- byte string
- arbitrarily addressable
- content without property and structure
- form and content completely defined by user
- maximum file size depending on system (typically GB or TB range, originally GB range)
- limited to one logical volume
- protection by access rights
  - r *read*
  - w *write*
  - e *execute*
- specifically for user, group, world

The Inode (Index-node)

- Each file is described by a so-called *I-node*. It represents the file.
- It contains
  - owner (UID, GID)
  - rights
  - date of creation
  - date of last change
  - size
  - type (file, directory, device, pipe,...)
  - array for tree-like access structure (references to data blocks)
- I-nodes are kept in a table for each file system.

Unix file organization



assumption: block size of 1KB

access level	max. file size
direct:	10KB
indirect:	256KB
twofold indirect:	64MB
threefold indirect:	16GB

Directories (catalogues)

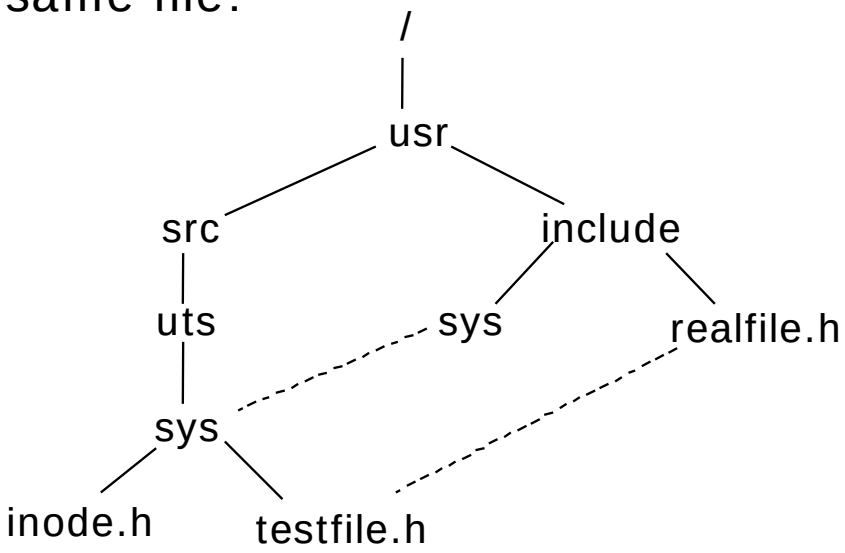
- Directories are managed as normal files. Only an entry in the type field indicates that it is actually a directory.
- A directory entry contains:
  - length of entry
  - name (variable length up to 255 characters)
  - I-node number
- More than one directory entry can point to the same I-node (*hard link*).
- Users identify files by a *path name* (series of identifiers with “/” as separator which is translated into an I-node number.
  - If the path starts with “/”, then it is an *absolute path name* that starts at the *root-directory*.
  - If a path starts with a character other than “/”, it is a *relative path name*, that refers to the current directory.

Directories

- Each directory starts with an entry ".", that indicates the I-node of the current directory.
- The second entry is ".." and refers to the parent directory.
- The path name is being resolved from left to right and at every resolution the respective name is searched in the directory.
- As long as it is not the last name of the path, it must be a directory. If not, the search is aborted with an error message.

Symbolic Links

- Unix offers the opportunity to access files and directories using different names. That also helps to facilitate shared usage.
- by symlink(old_name, new_name) an additional name is generated.
- Example:
  - with symlink("/usr/src/uts/sys","/usr/include/sys")
  - and symlink("/usr/include/realfile.h","/usr/src/uts/sys/testfile.h")
  - we have three path names to the same file:
    - /usr/src/uts/sys/testfile.h
    - /usr/include/sys/testfile.h
    - /usr/include/realfile.h



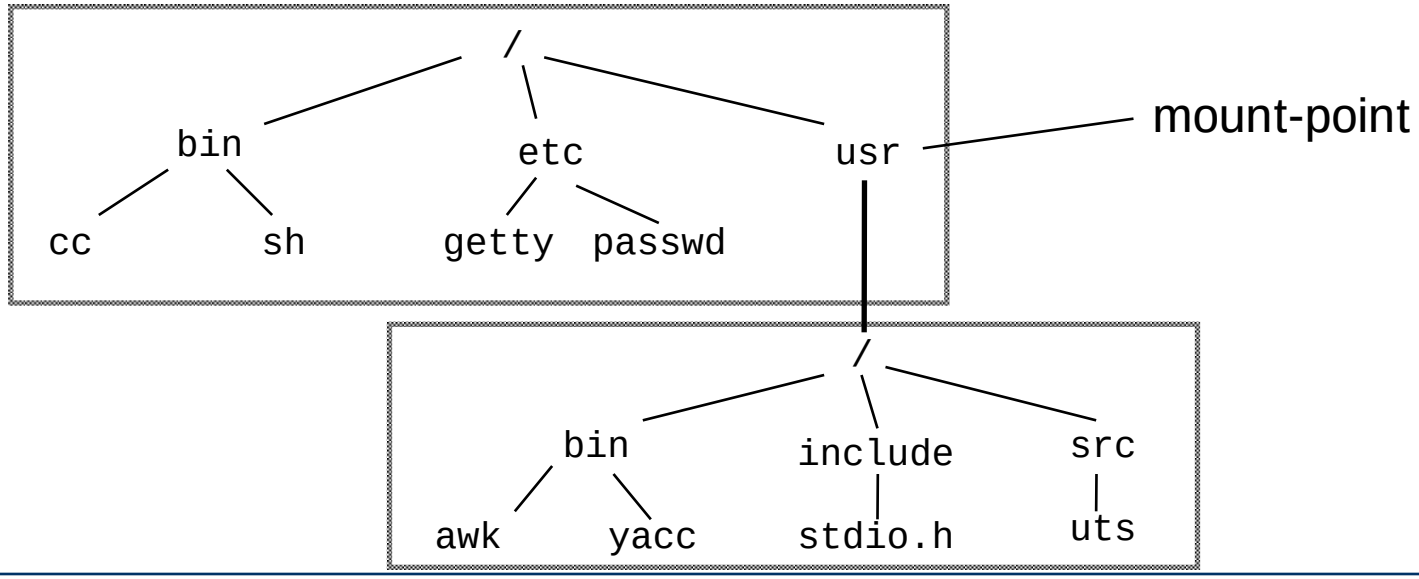
Hard and Symbolic Links

- A **hard link** is only another file name.
  - There is another directory entry with a pointer to the same file.
  - The I-node entry is the same for all *hard links*.
  - Each new *hard link* increases the *link counter* in the I-node of the file.
  - As long as *link counter* ≠ 0, the file will not be deleted after a *remove().*
  - A *remove()* only decrements the *link counter*.

- A **symbolic link** (*soft link*) is a file that contains a path name for a directory or a file.
  - Symbolic Links will be evaluated at each access.
  - If the file is deleted, the path name does no longer refer to an existing file, e.g. it gets invalid.
  - Symbolic links to files or directories can be created even if the file or directory does not yet exist.

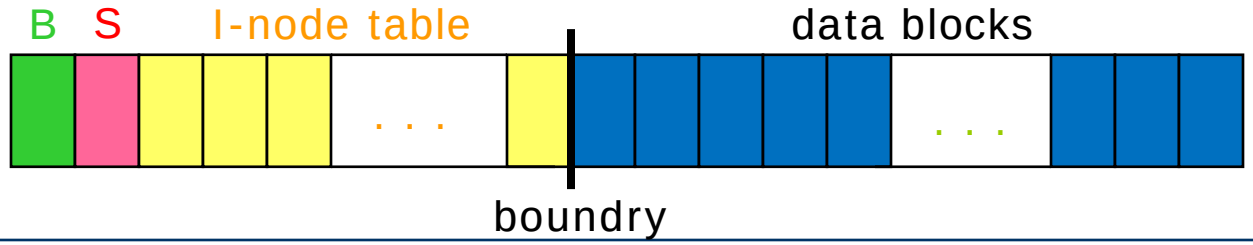
Logical and physical file system

- A logical file system can consist of more than one physical file system.
- A file system can be inserted into another file system at an arbitrary place by using the “mount”-command and removed again by using “umount”.
- At the access of a mounted directory a special bit in the I-node indicates that it is a “mount-point”.
- A “mount-table” is established, which is managed by the OS as a connection between the I-node of the mount point and the root directory of the mounted file system.



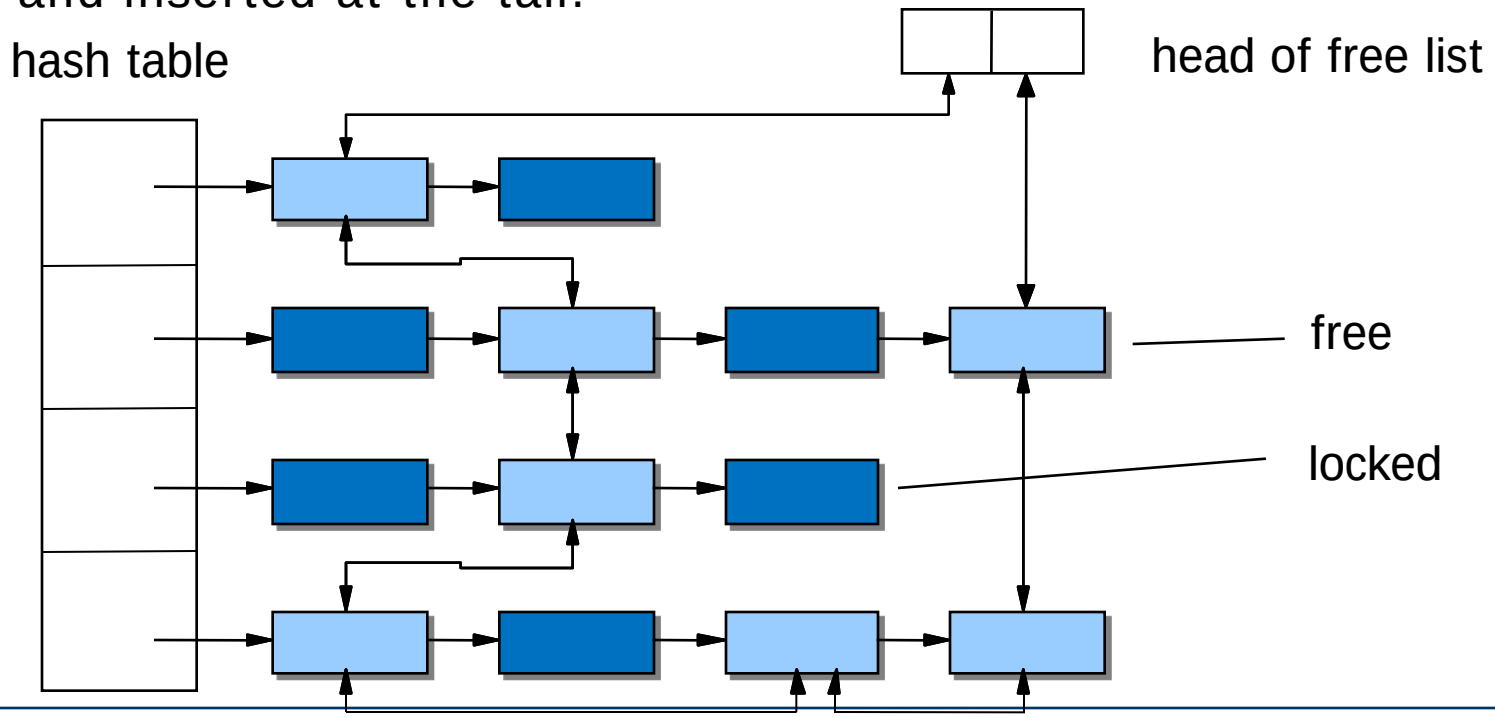
Disk structure

- Each physical file system typically resides on a logical device (*partition* of a physical device or *logical volume* managed by a *logical volume manager*). Many of those partitions can be existent on a physical device.
- Each logical device (file system) contains, behind the **boot block,** a so-called **super block** with the following content:
  - size of file system
  - list of free blocks
  - list of free I-nodes
- Following the super block there is the list of I-nodes of this file system.
- Newer Unix file systems use the concept of a cylinder group which provides this structure (super block, Inodes) for each cylinder group separately. (Reduction of arm movement)



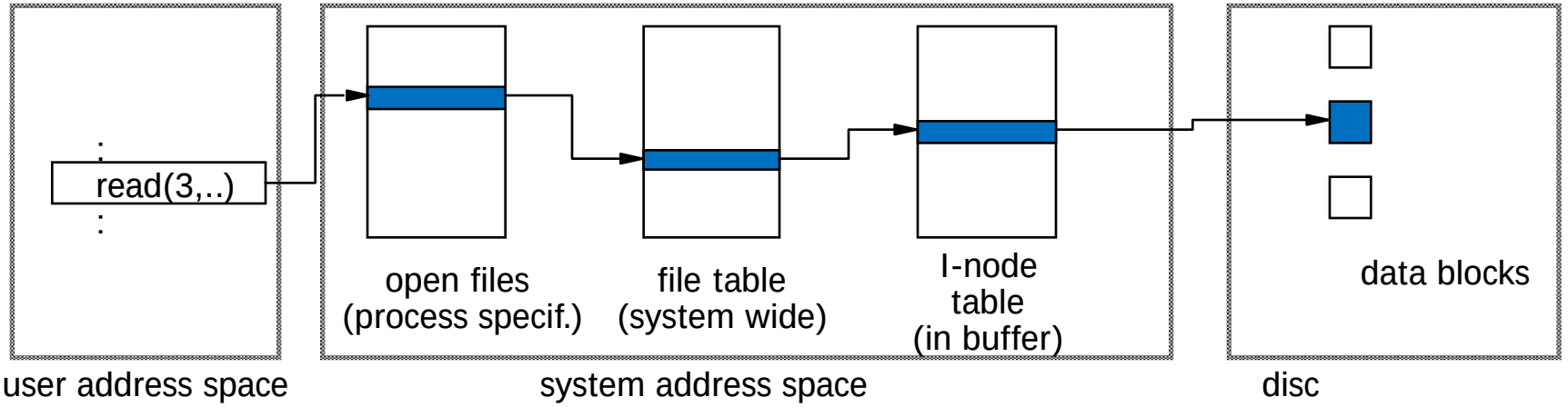
Buffering

- Disk blocks are buffered in main memory. For fast access a hash table is used. Blocks with identical hash value are kept in a linked list.
- The management of the buffer blocks (replacement strategy) follows the LRU principle. A free list is maintained, in which the free blocks are double linked in a circular way. Free blocks are removed at the head and inserted at the tail.



File operation

- Upon opening a file, a file descriptor (integer) is created.
- It is used as an index to a process specific table of open files.
- From that table entry a reference leads to an entry in a system wide file table.
- From there we get to the I-node of the file.



File operation

- *Read* and *write* do not contain any position information, only the number of bytes to be transferred.
- The current position pointer is kept in the file table and updated after each operation.
- A process can have more than one descriptor to the same file.
- Different processes can access the same file.
  - Problem?
- The copy of the I-node in the buffer contains a counter which indicates how many entries in the file table are pointing to it.
- Locks are possible, but are not enforcing (*advisory locks*).
- They can be applied to entire files or to parts of files.

10.7 Windows NTFS

- Windows NT supports older file systems like the FAT system of DOS and Windows95 or HPFS of OS/2.
- The usual one is NTFS (NT File System) with following properties:
- Fault tolerance
  - Transaction concept (all-or-nothing)
  - Automatic restart at system crash during operation.
  - Support of RAID-1 (mirroring) and RAID-5 (block-level-parity, server only)
- Security
  - Discretionary access control
- Support of large files and large disks
- Unicode-Names
- Posix support
- Encryption
- Compression

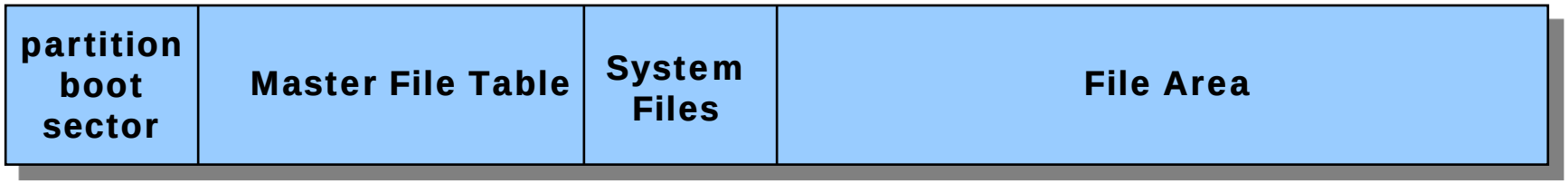
Volume structure

- Sector: smallest addressable unit on a disk
- Cluster: Collection of several contiguous sectors on the same track (smallest unit of allocation in NTFS)
- Volume: logical partition
  - at least 1 Cluster
  - can stretch across several disks (< 264 bytes)
  - Raid 5 supported

volume size	sectors per cluster	cluster size in KB
< 512 MB	1	0.5
512 MB – 1 GB	2	1
1 GB – 2 GB	4	2
2 GB – 4 GB	8	4
4 GB – 8 GB	16	8
8 GB – 16 GB	32	16
16 GB – 32 GB	64	32
> 32 GB	128	64

- Access to the cluster by Cluster Numbers:
  - Numbering of cluster of a volume: Logical CN
  - Numbering of cluster of a file: Virtual CN
  - Mapping of VCN to LCN in file description

Volume structure



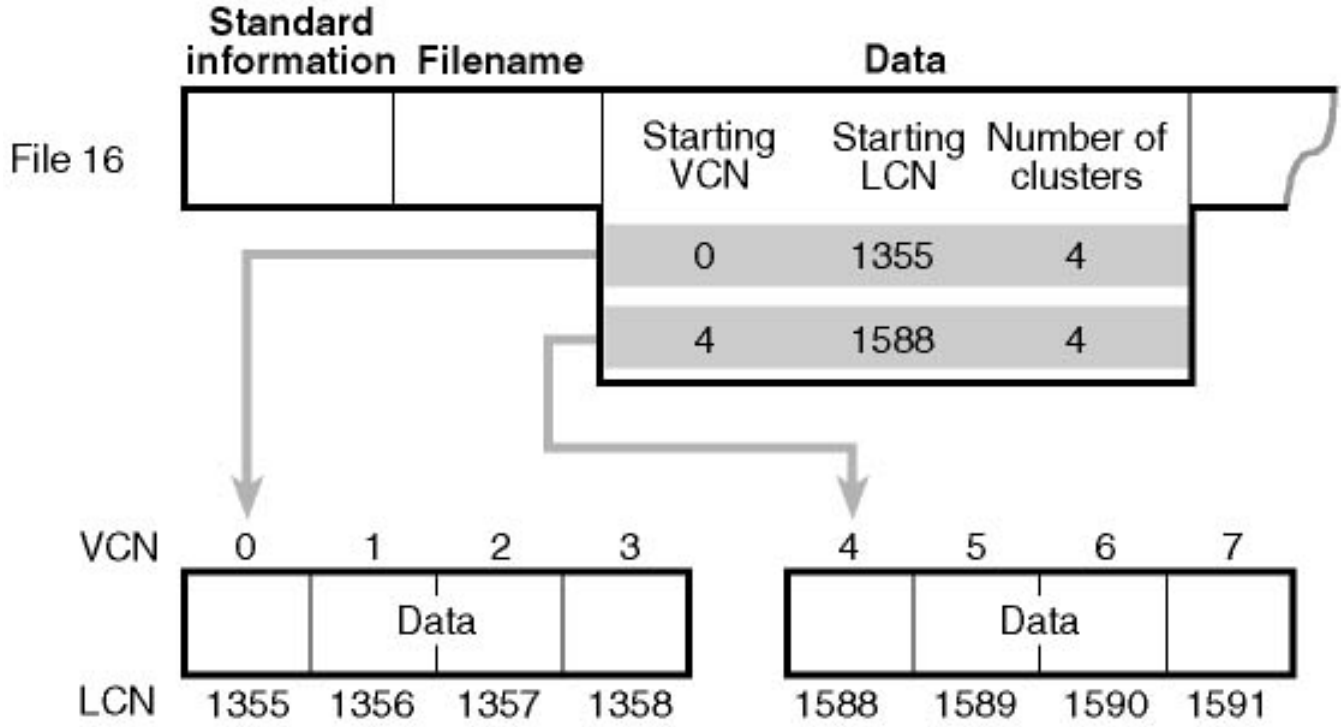
- MFT (corresponds to Inode-Table in Unix)
  - Each row (1KB) describes a file or a directory and contains Information about the location of the data blocks. If the file is very small it will be stored exactly there.
- System files
  - MFT2: Mirroring of the first 3 rows of the MFT
  - Log file: List of transaction steps
  - Cluster bit map: Representation of occupied and free Clusters
  - Attribute definition table: Supported file attributes on this volume
  - Bad cluster file

File attributes

Freie Universität Berlin

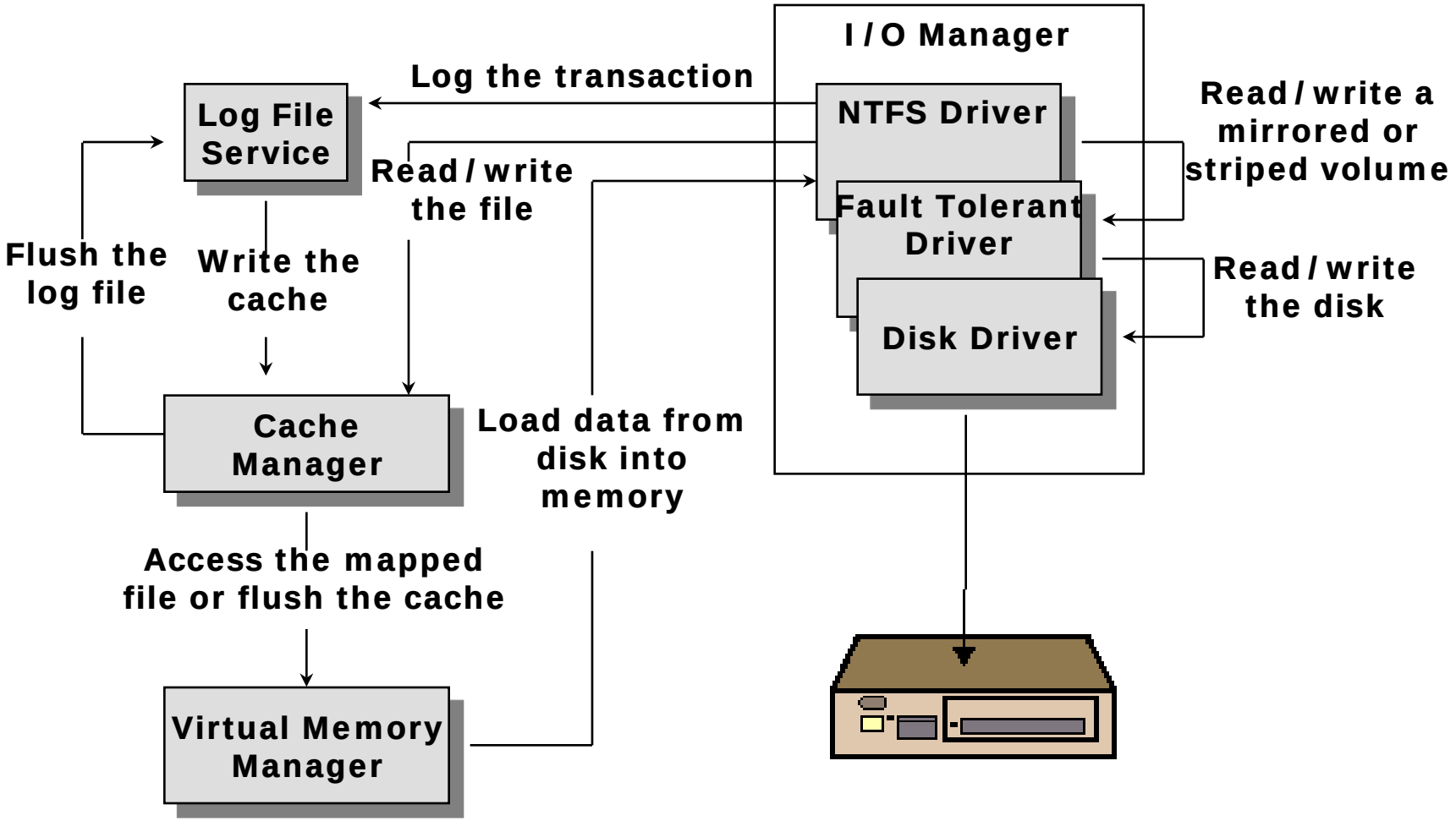
Attribute type	Description
Standard Information	file attributes (Read-only, read/write, etc.) time stamp (creation date etc.) Link counter etc.
Attribute list	For those attributes that do not fit into one Row of the MFT (1KB)
File name	At least one name (up to 255 Unicode characters)
Security descriptor	owner and list of authorized users
Data	file content
Index root	For directory files
Index allocation	For directory files
Volume information	Version number and volume name
Bitmap	For directory files

Mapping of Virtual to Logical Numbers



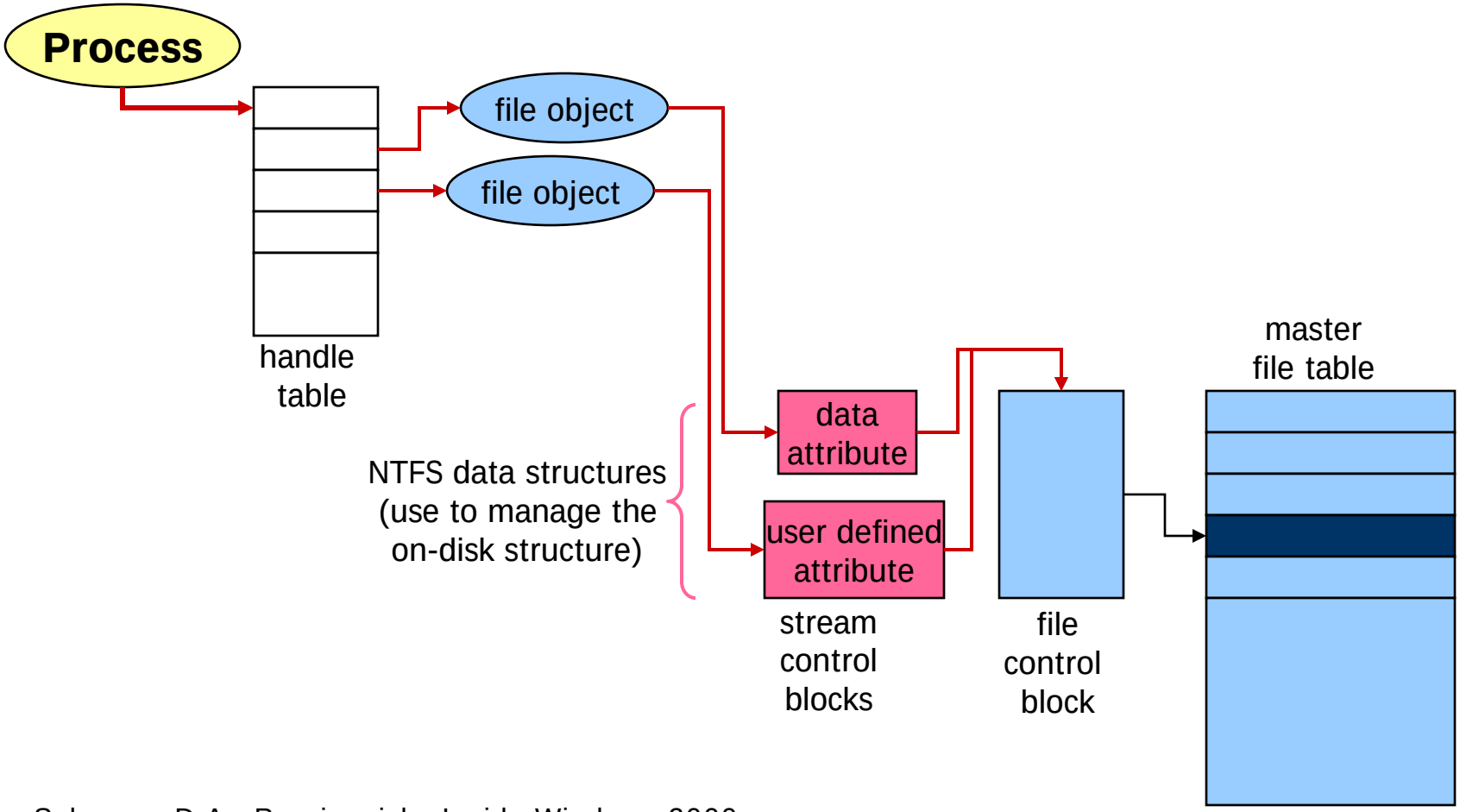
Source: Solomon, D.A., Russinovich: Inside Windows 2000

NTFS in the context of other services



Source: Solomon, D.A., Russinovich: Inside Windows 2000

NTFS Data structure



Source: Solomon, D.A., Russinovich: Inside Windows 2000

10.8 On-disk Consistency of File Systems

- Traditional file systems
  - Require complete check after crash
  - Traverse everything, check for consistency, fix problems
  - (Possibly hidden) data loss likely
- Newer approaches try to avoid this check
  - Journaling file systems (aka logging file systems)
  - Log-structured file systems
  - Copy-on-write file systems (aka shadow paging file systems)
  - File systems with soft updates
- No replacement for regular consistency checks and backups!
  - File system bugs
  - Silent data corruption
  - User errors

Journaling File Systems

- Journaling file system = Traditional file system + Journal
- Every modification (or set of modifications) is written as a transaction to the journal first.
- Finished transactions are discarded from the journal.
- After a crash, only the journal has to be replayed
  - Incompletely written transactions are recognized by wrong checksums.
- Drawback: low performance, everything has to be written twice
- Trade-off: journal only metadata
- Other issues: read-only access and boot-up after a crash problematic due to pending log replay; log replay often not that well tested

Log-structured File Systems

- Log-structured file system
  - = Journaling file system – Traditional file system
  - = Journal
- Just the Log itself remains
- Requires extensive caching
  - Which is done today anyway
- In-memory file system is constructed by replaying the log
  - Regularly written checkpoints contain enough metadata, so that not everything has to be replayed
- All writes are sequential, good for HDDs and SSDs
- Drawback: garbage collection necessary, low performance when nearly full and too much I/O to do garbage collection in background

Copy-on-Write File Systems

- Copy-on-Write file system
  - = Traditional file system – In-place updates (strictly speaking, log-structured file systems are also COW)
- No in-place updates, therefore no inconsistencies
- If a block must be updated, a copy of that block is updated and then the reference to it is replaced
  - Possibly recursively up to the superblock
  - (remember our B* Tree example: How many copy operation are necessary?)
- Allows easy realization of snapshots
- Drawbacks: COW operations can be costly
  - Trade-off: increase commit latency for better performance
  - “Not enough free space to delete file”?

File Systems with Soft Updates

- Soft update file system
  - = Traditional file system + Enforced order of writes
- Writes are ordered to avoid inconsistencies
- On-disk file system is always semi-consistent
  - Not all inconsistencies can be avoided, but
  - Nothing bad will happen after a crash
- Still needs a complete file system check after a crash
  - Can be done concurrently in the background
- Drawback: difficult to implement, reduced performance due to frequently used I/O barriers, conflicts with I/O scheduler

10.9 ZFS: “The Last Word on File Systems”

Freie Universität Berlin

- Originally: “Zettabyte File System”
- Goal: Sufficiently large file system (256 quadrillion ZB, 1ZB = 2^{70}B)

- Features:
  - Storage pools
  - End-to-end data integrity by checksums
  - Copy on write transactional model
  - Caching
  - Snapshots
  - Deduplication
  - Encryption
  - …many more

ZFS: Architecture

- Three layers:
- ZFS POSIX layer (ZPL)
  - Issues transactions atomically
  - Changes are reverted if it does not complete atomically
  - Offers the classical POSIX interface to the user
- Data management unit (DMU)
  - Manages copy on write behavior
  - Commits at the end of a transaction
- Storage pool allocation (SPA)
  - Manages storage pools including
    - Compression
    - Checksums and verification
    - Snapshots
    - ...

ZFS: Storage Pools

- Classical way: One filesystem per partition or using volume manager to combine devices into logical partitions
- ZFS: Devices contribute to a storage pool that can be extended at run-time
- Support for different replication strategies (similar to RAID)
- Storage pool is dynamically used to offer
  - File systems
  - Block devices (e.g., for virtual machines)
  - Snapshots

ZFS: Caching

- ZFS supports caching at different levels
- RAM cache
  - Caches read accesses
  - Can use large amounts of RAM in an efficient manner
- SSD cache (optional)
  - Read cache (L2ARC) automatically filled during operation to speed up read operations. No damage if lost because data is always on hard disk. Speeds up deduplication significantly.
  - Write cache caches synchronous writes as asynchronous writes that are later committed to the storage pool. Used only in case of crash. Device failure may result in losing latest writes.

References

- Stallings,W.: *Operating Systems 5th ed.*, Prentice Hall, 2005, Chapter 12
- Tanenbaum, A.: *Moderne Betriebssysteme*, 2.Aufl., Hanser, 1995, Kapitel 4+7
- Bacon,J.: *Concurrent Systems*, Addison Wesley, 1997, Chapter 7
- Nehmer,J.; Sturm,P.: *Systemsoftware*, dpunkt-Verlag, 2001, Kapitel 9
- Solomon, D.A., Russinovich: *Inside Windows 2000*, MS Press, 1998, Chapter 9
- Fagin,R.; Nievergelt,J.; Pippenger,N. and Strong,H.R.: *Extendible Hashing - A Fast Access Method for Dynamic Files*, ACM Transactions on Database Systems, 4(3):315-344, 1979
- Kumar, V.: *Concurrent Operations on Extendible Hashing and its Performance*. Commun. ACM 33(6): 681-694(1990)