Course
TÖL401G: Stýrikerfi / Operating Systems
11. File-System Implementation

Chapter Objectives

- Describe the details of implementing local file systems and directory structures.
- Discuss block allocation methods and free-space management.
- Explore file system efficiency and performance issues.
- Look at recovery from file system failures.

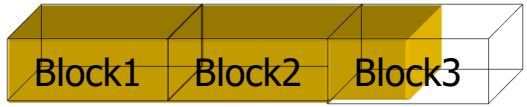
Contents

1. Introduction: File-System Structure
2. File-System Implementation
3. Allocation Methods
4. Free-Space Management
5. Directory Implementation
6. Efficiency and Performance
7. Recovery
8. Summary

Note for users of the Silberschatz et al. book: these slides differ slightly from their chapter 11

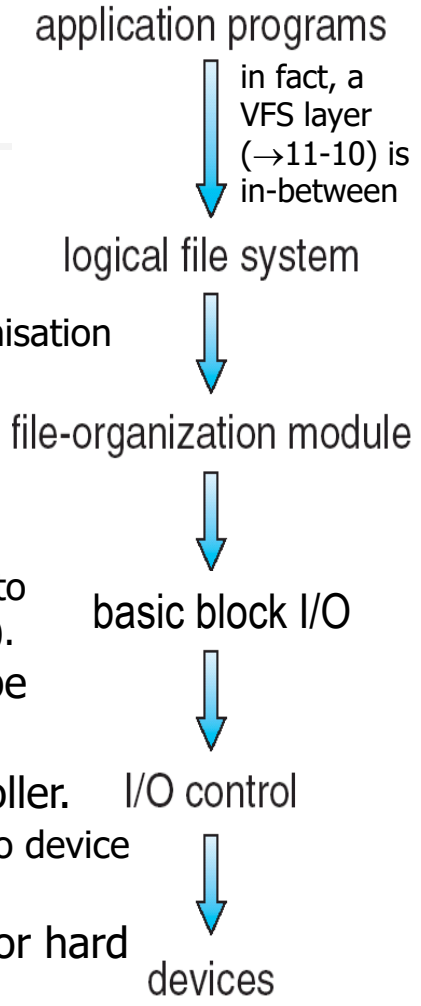
11.1 Introduction: File-System Structure

- File system store files (incl. directories) on secondary storage devices:
  - Mainly: Hard disk & flash memory/solid state drive (SSD), etc.
- Storage area of file system is divided into logical storage units:
  - Unix: blocks (typical size: 4KB), MS Windows: clusters (different sizes used).
    - In the following, the term "block" may also refer to "clusters".
  - If file is longer than a block, multiple of them are needed for storing the file contents.
  - If file size is not a multiple of block size, the last block is only partially used (internal fragmentation).



- In addition, some metadata (= data in in addition to actual file contents) is required and stored as part of the file system on the storage device: information about
  - which blocks on the storage device are allocated/free (→free space management).
  - in which blocks is the contents of each file stored (→allocation methods).

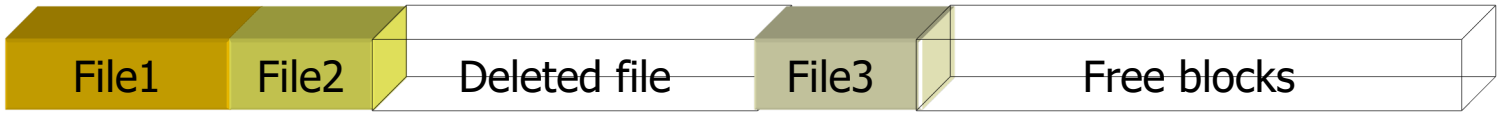
File-System Layers



- Logical file system: manages file pointer, metadata (e.g. owner, permissions, directory contents), symbolic links.
  - As directories are just some special kind of file, the underlying file-organisation module can be used to read/write content of a directory as a file.
- File-organisation module: manages in which blocks of the device the file contents is stored (→allocation methods) and where free blocks can be found on the device (→free space management).
  - Translates request from logical file system ("read first block of file x") into requests for basic block I/O and I/O control ("load block 123 from disk").
- Basic block I/O: e.g. handles buffers for blocks that are waiting to be transferred to/from storage device, caches blocks.
- I/O control: device driver that is specific to underlying device controller.
  - Translates command from block I/O layer ("load block number 123") into device specific hardware commands.
- Device: actual I/O device and controller (e.g. Serial ATA controller for hard disk, USB controller for flash memory key).
- Well defined interfaces between the layers allow to easily exchange, e.g., Logical file system, I/O device.
  - E.g. the same file-system may be used on different devices or different file-systems may be used on the same type of device.

11.3 Allocation Methods: Contiguous Allocation

- How to store file contents in the blocks of a file system?
- Naïve approach: allocate file contents in contiguous blocks.



  - Problem: If file needs to grow (e.g. File1 in the above example), subsequent blocks may already be allocated by other files.
  - Possible solution: extents (→next slide).

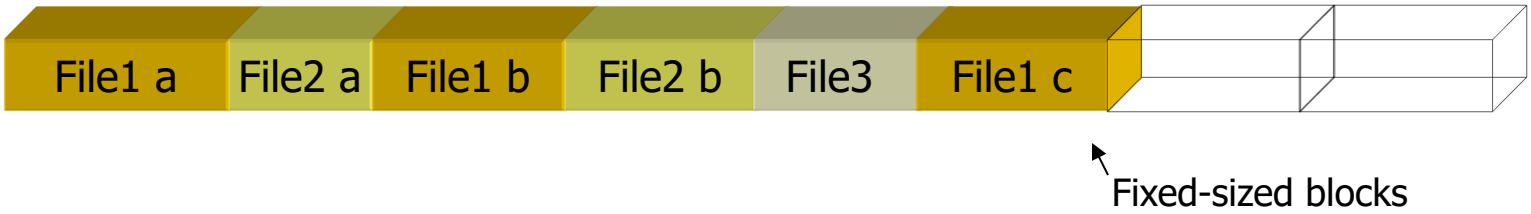
Fragmented Allocation: Extents

- Modification of contiguous allocation scheme:
  - If initial chunk of contiguous blocks is full, use additional contiguous chunks (an extent) that start at a new location and can be added to the already existing extents of a file.
    - Start and length of current extent needs to be stored together with information about where to find information about the following extent.
      - Such additional file allocation information is called (allocation-)metadata.

    - Extents have an arbitrary size (in contrast to the one-block-per-allocation linked & indexed approaches covered in the remainder)
      - But still, size of an extent is a multiple of a block.
  - As with main memory management, using arbitrary-sized extents may result in different sized holes of free storage areas.
    - Having many small free holes leads to a significant fragmentation.

Fragmented Allocation: Linked and Indexed Allocation

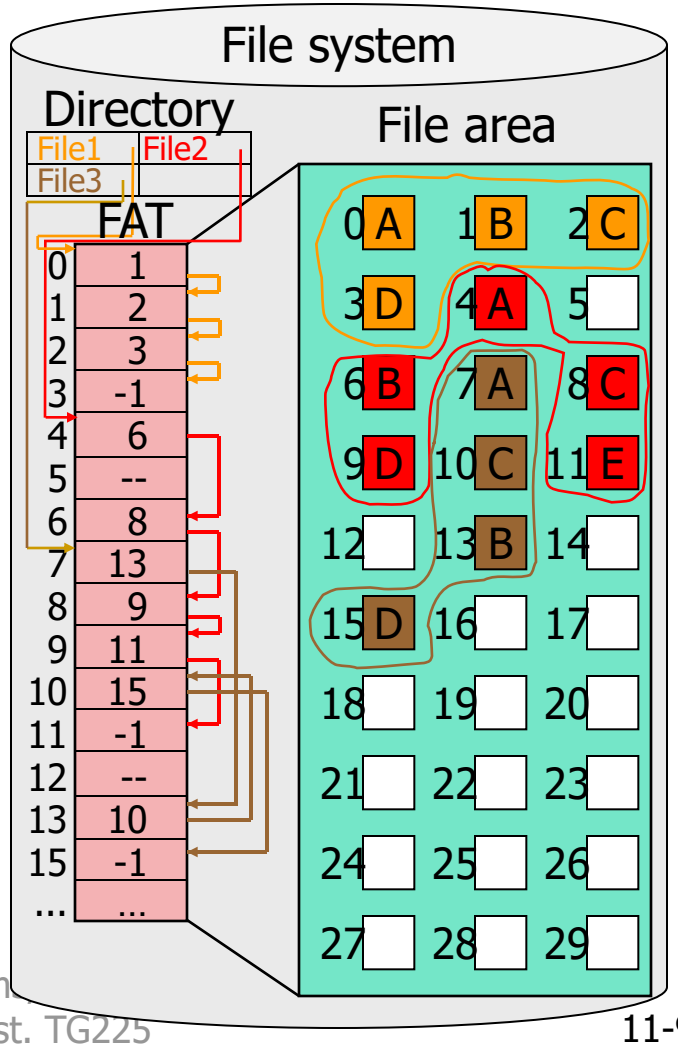
- Approaches that support fragmented storage by design (i.e. storing in a fragmented way does not require more allocation metadata than storing in a contiguous way):
  - For each individual fixed-sized block of a file, it is always (even if these blocks are contiguous) stored in the metadata where it is located in the file system.



- Typically used approaches:
  - Linked allocation: File Allocation Table (FAT) or
  - Indexed allocation: Index table (I-Nodes).
  - Also, Extents approach becomes popular again.

Linked Allocation: File Allocation Table (FAT)

- FAT=File Allocation Table (e.g. MS-DOS, USB keys, SD card).
- FAT is sorted with respect to blocks of file system:
  - FAT is a map of all blocks of the file system, i.e. for each block, one entry in the FAT exists.
  - The FAT entry of a block contains the number of the succeeding block of a file: can both be used to load that block and to identify next FAT entry (comparable to the pointer of a linked list).
    - -1 entry specifies the end of the linked list.
    - Directory entry points to first block of a file.
    - Depending on the number of blocks of the file system, a FAT entry has 12, 16 or 32 Bit (FAT12, FAT16, FAT32).

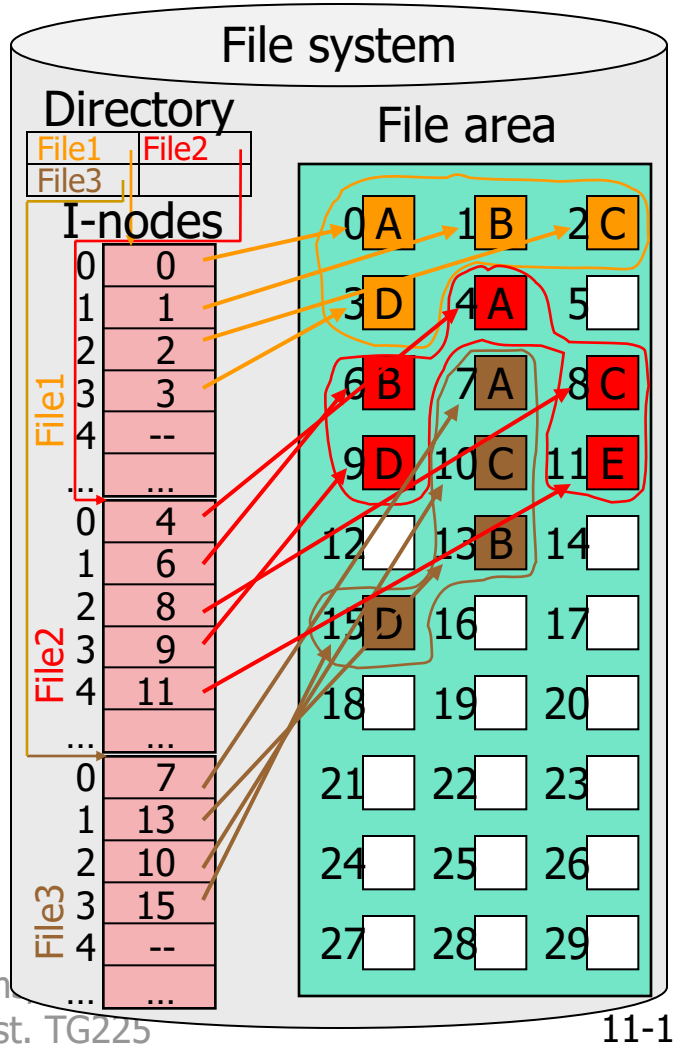


File Allocation Table: Disadvantages

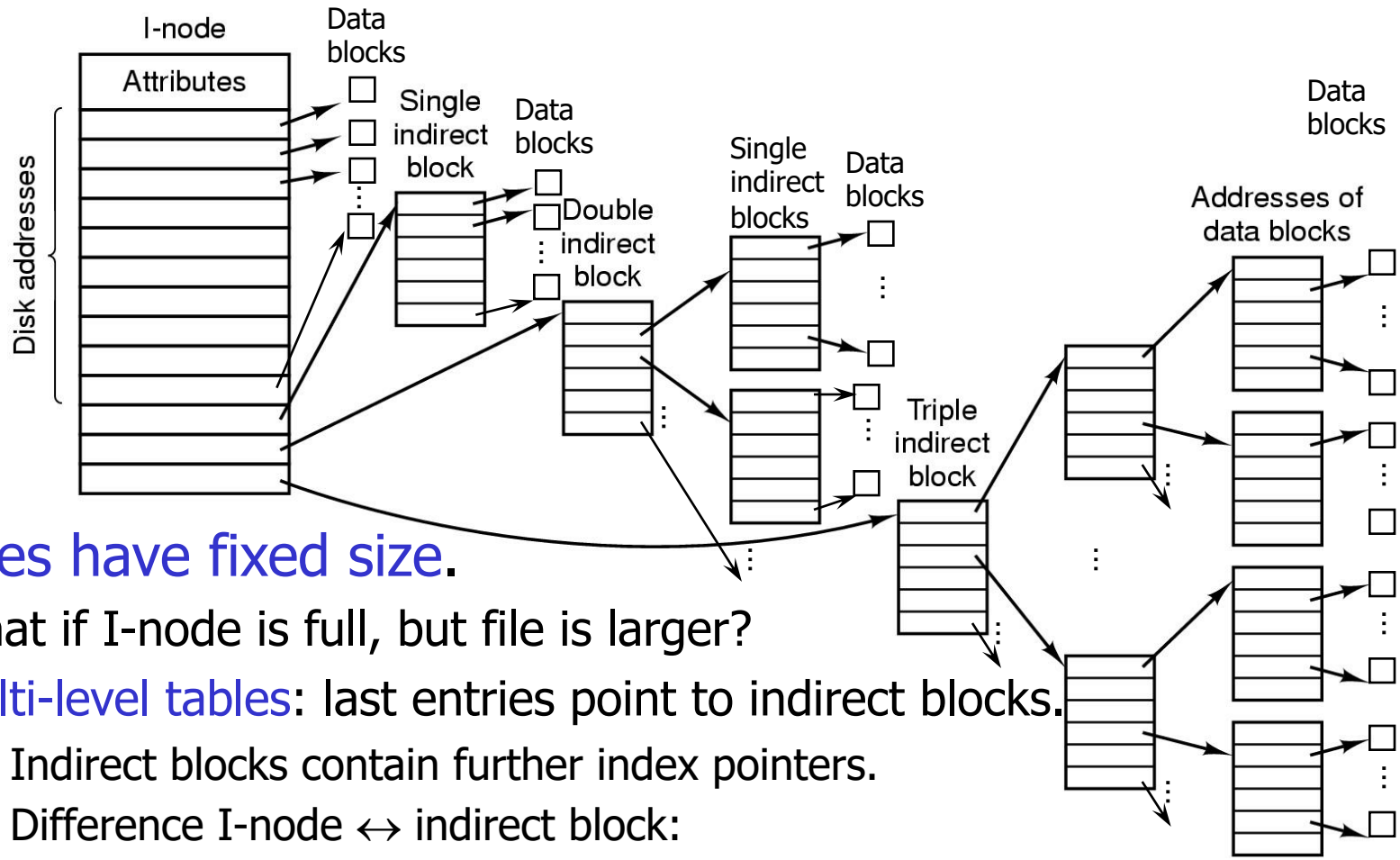
- If a file is stored fragmented, the corresponding FAT entries are as well scattered in the same way in the FAT.
  - In the worst case, for reading a file from start to end, huge parts of the FAT have to be traversed, i.e. the blocks of the FAT need to be read from the secondary storage device.
  - To speed up traversal: keep complete FAT in main memory.
    - Size of a FAT: e.g. 200 GB file system, 512 Byte block size $\Rightarrow 4 \cdot 10^8$ FAT entries, 4 Byte per entry$\Rightarrow$1.6 GB for FAT :-(
      - $\Rightarrow$Approach to reduce size of FAT: User clusters of blocks (up to 32KB clusters), i.e. FAT entries refer to cluster numbers.
      - Disadvantage of huge cluster sizes: larger internal fragmentation!
- To access the n^{th} cluster of a file (i.e. to find out in which cluster it is stored), $n-1$ FAT entries need to be traversed.
  - Complexity to determine n^{th} block/cluster: $O(n)$

Indexed Allocation: I-Nodes

- I(ndex)-nodes (most Unix file systems).
  - Alternative spelling: inode.
- Instead of having a file system-wide table, each file has its own table: the I-node.
  - For speeding up file access, it is sufficient to keep just the I-nodes of the currently opened files in main memory.
  - Directory entry points to I-node of a file.
- Each I-node is sorted with respect to blocks of a file:
  - I-node of a file contains for each block of that file a pointer to the block in the file system. (E.g.: storage location of 2nd block of file can be found in 2nd entry of I-node.)
  - Direct access to nth block possible: $\mathcal{O}(1)$



Indexed Allocation: Multi-level Index



- I-nodes have fixed size.
  - What if I-node is full, but file is larger?

  ⇒ Multi-level tables: last entries point to indirect blocks.
  - Indirect blocks contain further index pointers.
  - Difference I-node ↔ indirect block:
    I-node contains file attributes (except file name: stored in directory).

11.4 Free-Space Management

- To create a new file/extend an existing file, OS needs to know which blocks/clusters are free:
  - FAT: Special cluster number (typically -2) in FAT marks free cluster.
    - Linked list FAT data structure serves at the same time free-space management:
      - To find a free cluster: just search in FAT for a cluster with that special number. Complexity: $O(n)$ (n=number of clusters managed by FAT).
  - I-node: only deals with existing files, i.e. keeps only track of allocated blocks, not of free blocks :-(
    - Use an additional free space-bitmap where one bit indicates whether the corresponding block is allocated or free.
      - To find a free block: just search bitmap for a bit that indicates a free block. Complexity: $O(n)$ (n=number of bits in bitmap)
      - Size of free-space bitmap: as many bits as the file system has blocks.

Free-Space Management: Counting

- Problem with free-space bitmaps needed by I-node file-system: For a 1 TB file system, a bitmap where each 4 KB block is represented by 1 bit has a size of 32 MB that needs to be searched/updated (=slow).
  - Solution (used by more recent I-node file-systems): As free blocks are often contiguous, count number of free blocks and just store for each range of contiguous free blocks: [*start of contiguous range of free blocks*, *number of contiguous free blocks in that range*].
    - To find a free block: just search list for first entry (=first fit).
    - Complexity to find a free block in such a list: $O(n)$ (n=number of entries in list).

Free-Space Management: Choosing free blocks

- Which free block/cluster to choose?
  - Same memory allocation strategies as for main memory: e.g. First Fit.
  - When extending an existing file: chose location near to existing location (e.g. First Fit starting at current end of file) to minimise mechanical head movements of hard disk.
    - (Not relevant for electronic disks (SSDs) that have no moving mechanical parts.)

Free-Space Management: SSD TRIM

- Problem of electronic disks (SSDs):
  - Each write to a flash-memory cell wears that cell out (=after a finite number of writes, cell "dies").
  - An SSD is not aware of the file system used by the OS, hence it does not know that a free-space block of the file-system stored on the SSD is in fact currently not used for storing data.
  - A file-system implementation can send the SSD a so-called TRIM command to tell the SSD that a block is not used.
  - $\Rightarrow$ SSD can then internally map heavily used file-system blocks to unused blocks and thus achieve wear leveling of all flash-memory cells.
    - (See also forthcoming ch. 12.)

Free-Space Management (De-)Fragmentation/Compaction

- If a file system has been in use for some time and some files have been deleted & many files added, it may be possible that no contiguous blocks can be found to store a new file or extend and existing file: external fragmentation.
  - Contents of a file is stored at various locations scattered throughout the file system.
  - On a mechanical hard disk ($\rightarrow$ch. 12), this slows down sequential reading of a file.

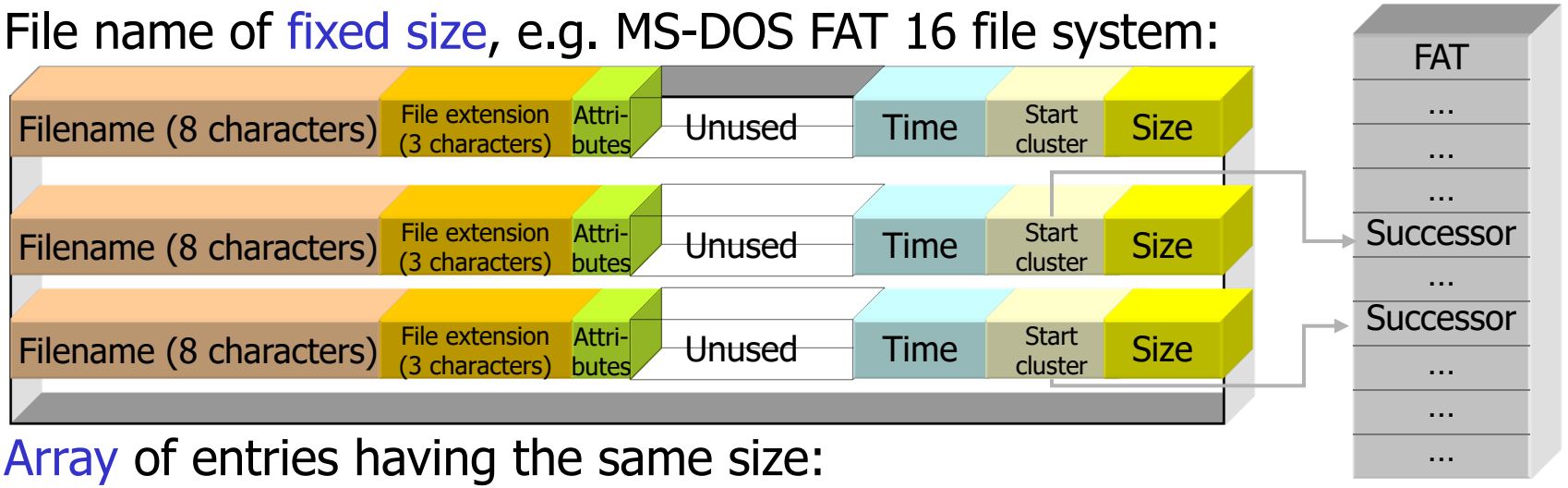
- Defragmentation/Compaction: Move blocks/cluster (by copying them)
  - to remove external fragmentation of files (i.e. store blocks/cluster of a file contiguous) and
  - to create contiguous free space for future files.

Free-Space Management Fragmentation on I-node systems

- Note: It is often claimed that I-node based file systems do not suffer from fragmentation: this is not true!
- Fragmentation occurs, but maybe less severe:
  - Most I-node based file systems typically divide the file system into sections, each of them having their own "pre-allocated" set of I-nodes. New blocks are preferably chosen from the same section as the corresponding I-node (=less far-away scattered).
    - ⇒More likely that all blocks of a file are located in the same section, less likely that a file "steals" free blocks from other sections (only if current section is full), thus leaving free blocks of another section for I-nodes of that other section.
    - ⇒Blocks of a file may still be fragmented, but are typically closer together (=access with a mechanical hard disk not slowed down that heavily).

11.5 Directory Implementation: Entries with fixed size

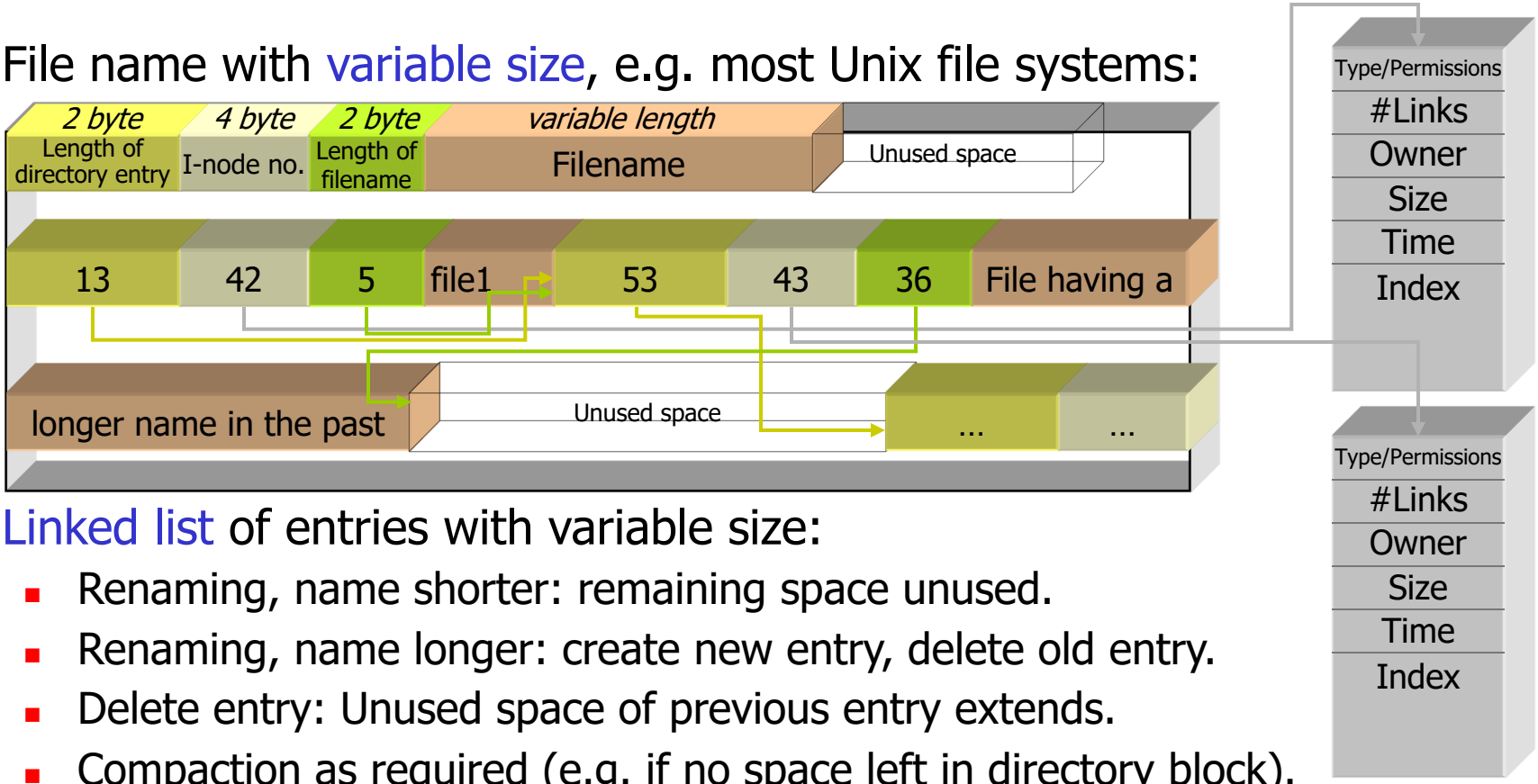
- File name of fixed size, e.g. MS-DOS FAT 16 file system:



- Array of entries having the same size:
  - Deleting an entry: Mark entry as empty (set first character to 0xE5).
  - Creating new entry: Find first empty entry.
- Entries unsorted in order of creation ⇒ Searching an entry: $O(n)$. (Number of directory entries)
- All file attributes in directory entry ⇒Hard links not possible.
- Longer file names (since Win95, backwards compatible to MS-DOS):
  - Entry with invalid combination of attributes contains in further elements of the entry the long name that refers to the subsequent entry with 8+3 name.

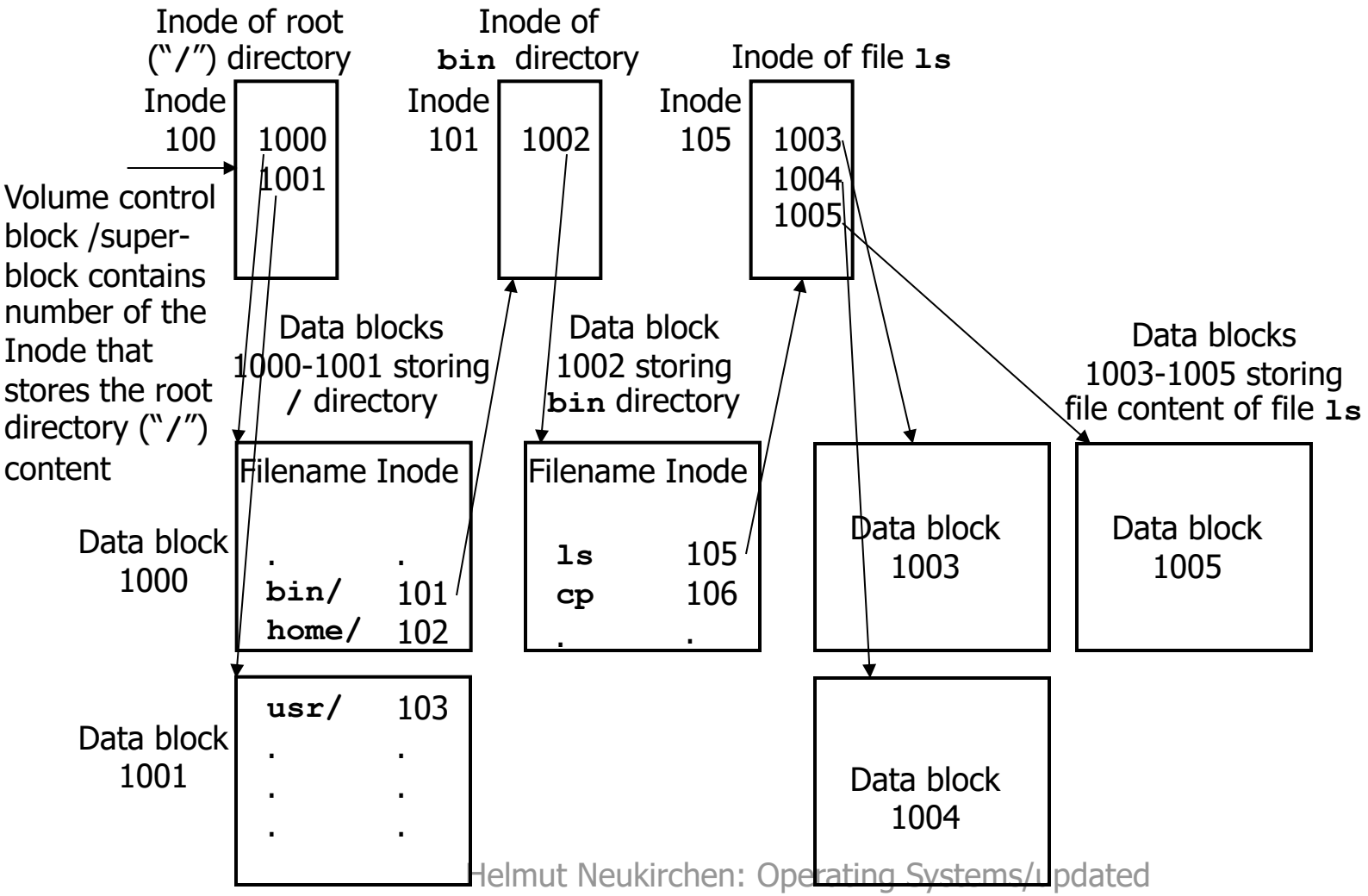
Directory Implementation: Entries with variable size

- File name with variable size, e.g. most Unix file systems:



- Linked list of entries with variable size:
  - Renaming, name shorter: remaining space unused.
  - Renaming, name longer: create new entry, delete old entry.
  - Delete entry: Unused space of previous entry extends.
  - Compaction as required (e.g. if no space left in directory block).
- Entries unsorted in order of creation ⇒ Searching an entry: $O(n)$.
- Attributes (except name) in I-node ⇒ Hard links possible.

Example Inode file system with directory and file (Inode 100 is entry point into file system)



Directory Implementation: Hash Tables/Sorted Trees

- The fixed and variable sized directory implementation assume a linear list-based directory data structure.
  - The associated complexity of $O(n)$ of searching an entry in a linear list-based directory data structure with n entries becomes a problem if a directory contains thousands of files.
  - Solution: Faster (but less easy to implement) data structure for directory entries.
    - Sorted trees: search complexity $O(\log n)$,
    - Hash tables (í. tætitafla): search complexity $O(1)$.
      - File name mapped on a single number (using a hash function, e.g. one that calculates a checksum). Number used as index into a table that contains directory entries (collisions possible if different file names map on the same hash value: hash algorithm needs to deal with this).

Sparse Files and Short Files

- Sparse files: if a file contains blocks that consist of only entries with value 0:
  - Do not waste a data block for storing these zeros, but just use a special marker (e.g. -1) in the I-node entry for that block.
  - Should later that block not be anymore 0 only, just start using a normal data block and let I-node point to it.
- Short files: If a file has a size of just a few bytes (typical example: a symlink containing a short path name):
  - Do not waste a data block for storing short file, rather use the I-nodes themselves (i.e. the areas otherwise containing the block pointers) for storing the short file contents.
  - Should later the file grow in size, still possible to use normal data block and let I-node point to it.

11.7 Recovery

- Files store valuable data, so being able to recover from file system problems is important.
  - Hard disk crashes, bugs in file system implementations, but also OS crashes or power outage during file system updates may lead to data loss.
- Example: Steps when appending to a file (I-node based scenario):
  - Select block locations to be used based on bitmap of free blocks,
  - Mark blocks as allocated in bitmap of free blocks,
  - Add additional block pointers to indirect blocks (if involved) & I-node,
  - Write actual file contents to selected blocks locations.
  - Update file size attribute stored in I-node.
- As soon as any of this steps (except the first) fails (power outage/ system crash), the file system becomes inconsistent, e.g.:
  - Bitmap of free blocks and actual block allocation do not match,
  - Indexes in I-nodes/indirect blocks point to wrong locations, …
- (The probability increases when write accesses to disk are cached.)

Consistency Checking

- Regularly check file system consistency to prevent that a possible corruption affects even further data.
  - Often done at the next system boot after a system crashed (=has not been properly shut down, i.e. a flag in the file system on disk is set by OS when mounting, but has not been reset by proper unmounting).
- Tools for checking the consistency of file systems: **`fsck`** on Unix, **`chkdsk`** on MS Windows.
  - Scan a whole file system (all the metadata) and collect information about file systems.
  - Compare whether this information fits together.
    - If not: report inconsistency.
    - Tools may even be able to repair some inconsistencies.
      - Typically, lost data cannot be reconstructed, but the metadata can be repaired.

Journaling/Log-based Transaction-oriented File Systems

- Consistency checks and file system repairs are performed after an inconsistency has occurred.
- Journaling (or log-based transaction-oriented) file systems can avoid inconsistencies to occur at all.
  - (At least, inconsistencies due to power outage/system crash while writing data can be prevented – but still data that has not been written yet, can get lost).
  - Journaling file systems are state of the art in all modern file systems e.g. (Microsoft NTFS, all modern POSIX/Unix-like file systems, but not FAT).

- Based on the concept of transactions: Either write all data or no data! (I.e. either old file system state or new file system state.)
  - Three step approach: announce action, commit action, acknowledge action. (→next slide…)

Journaling/Log-Based File Systems: Approach

1. Write information about intended changes to an intermediate buffer („Journal" or „Log") on file system. Format: [header, changes, trailer].
   - If crash during this step: actual storage location has not been changed, yet. After restart, journal entry will get discarded due to missing trailer.

2. If journal full: Write changes logged in journal to actual storage locations in file system.
   - If crash (i.e. changes only partially written, journal entry not deleted yet): After restart, write again changes still stored in journal (steps 2 & 3).

3. Delete entry from journal.
   - If crash during this step: Restart with step 2 (if journal entry not yet deleted) or do nothing (journal entry deleted).

Journaling/Log-Based File Systems: Discussion

- Write accesses get slower, as all data is written twice: first to journal, then to actual storage location.
  - But: journal uses contiguous blocks, thus write (typically using synchronous write) is sequential (=fast)
    - If no journaling would be used, writes would be made to different locations (I-nodes, free space-bitmap, directory, data blocks) in a random-access style = slow!
  - Writing the changes logged in journal to actual storage locations in file system is done in the background using asynchronous write, i.e. may occur when the hard disk is anyway idle.
    - In the optimal case, journaling may even be faster than no journaling.
- Faster variant: use journaling only for metadata (data structures, no file content).
  - File content may get inconsistent, file system structure will be still consistent.

11.8 Summary

- Today's operating systems support fragmented storage of files.
  - Information about location of file contents:
    - Link-based (FAT),
    - Index-based (I-nodes).
    - Recent file-system take up again the extents-based approach for speed reasons.
  - Information about free space: like management of free main memory.
- Directories may support fixed or variable size file names.
  - Storing all file attributes (other than file name) in I-node enables hard-links.
- Journaling file systems guarantee consistency of file system data structures.
  - Still important: backup!
- Only basic concepts presented. Not covered:
  - Network file systems (covered in TÖL503M Distributed Systems course).
  - Microsoft NTFS file system, recent Linux and Mac OS file systems.