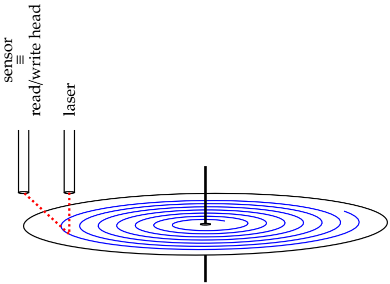
Continued from last lecture ...

- ... so far so good, *but* we need to explore
  1. how the file system supports our assumed access model, *and*
  2. how the underling **storage device** supports the file system.

Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (1)

- Example: **optical disks** (inc. CDs and DVDs).



noting that

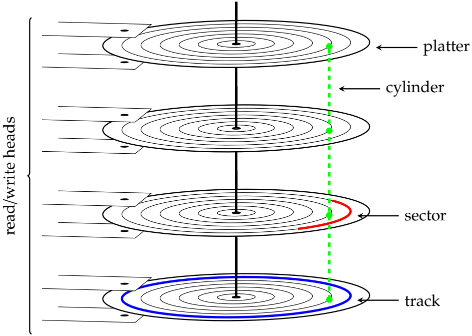
- one might attempt
  1. **random access**, and/or
  2. **sequential access**

  and
- efficiency of said access is limited by

$$\begin{array}{lcl} \text{transfer rate} & \propto & \text{read/write head performance} \\ \text{positioning latency} & \simeq & \textbf{seek latency} + \textbf{rotational latency} \end{array}$$

Mechanism: (mass storage) devices $\leadsto$ blocks (1)

- Example: **magnetic disks**.



noting that

- one might attempt
  1. **random access**, and/or
  2. **sequential access**

  and
- efficiency of said access is limited by

$$\begin{array}{lcl} \text{transfer rate} & \propto & \text{read/write head performance} \\ \text{positioning latency} & \simeq & \textbf{seek latency} + \textbf{rotational latency} \end{array}$$

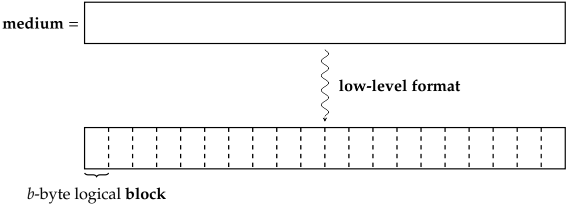
Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (2)

- We add structure to the medium via several steps

medium =

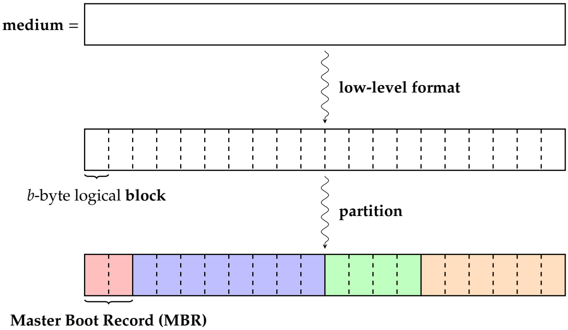
Mechanism: (mass storage) devices $\leadsto$ blocks (2)

- We add structure to the medium via several steps



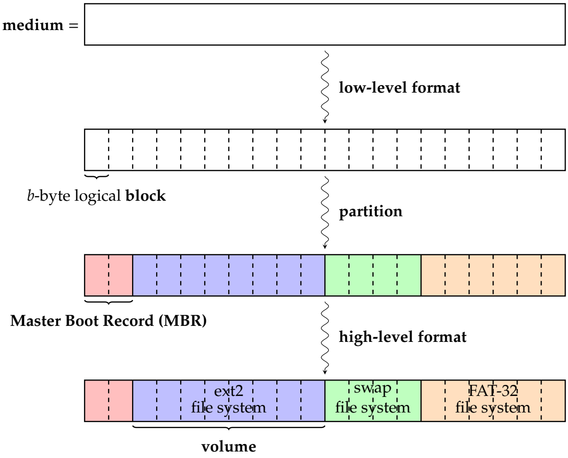
Mechanism: (mass storage) devices ⇝ blocks (2)

- We add structure to the medium via several steps



Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (2)

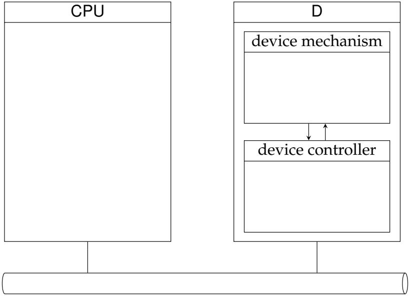
- We add structure to the medium via several steps



then ...

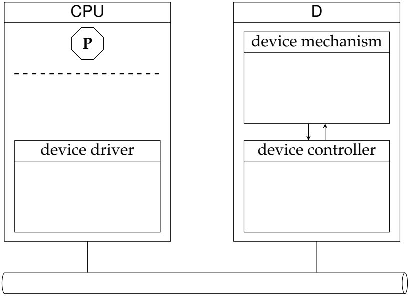
Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (3)

- ... assume an interface as previously described, e.g.



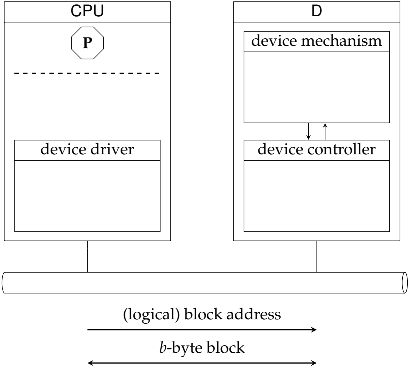
Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (3)

- ... assume an interface as previously described, e.g.



Mechanism: (mass storage) devices $\rightsquigarrow$ blocks (3)

- ... assume an interface as previously described, e.g.

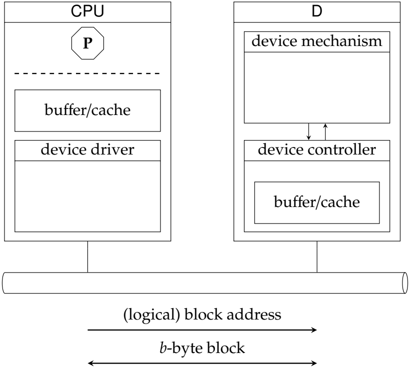


but, since efficiency is crucial we (typically) also

1. amortise overhead by fixing transferring b-byte blocks,
2. use **Logical Block Addressing (LBA)**, forcing translation by the device controller.

Mechanism: (mass storage) devices ⇝ blocks (3)

- ... assume an interface as previously described, e.g.



but, since efficiency is crucial we (typically) also

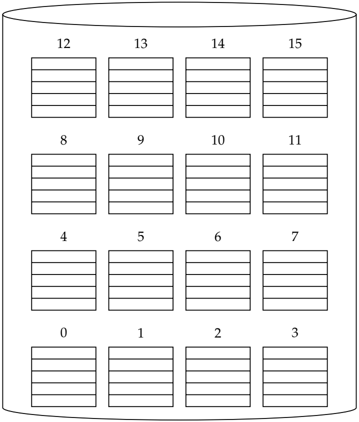
1. amortise overhead by fixing transferring b-byte blocks,
2. use **Logical Block Addressing (LBA)**, forcing translation by the device controller, and
3. buffer and/or cache accesses (in various layers).

Mechanism: blocks ⇝ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Solution: we need
  1. an allocation algorithm, and
  2. a data structure to capture the current allocation state.

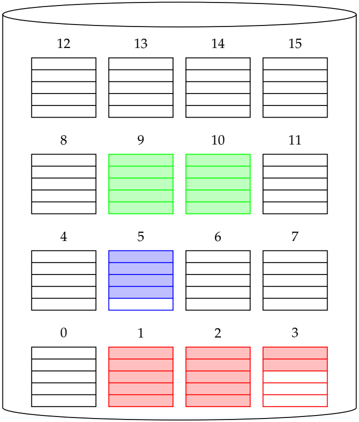


Mechanism: blocks $\rightsquigarrow$ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Idea: **contiguous allocation**.
  - – allocation is more challenging,
  - + sequential *and* random access is efficient,
  - – internal *and* external fragmentation,
  - + no storage overhead.

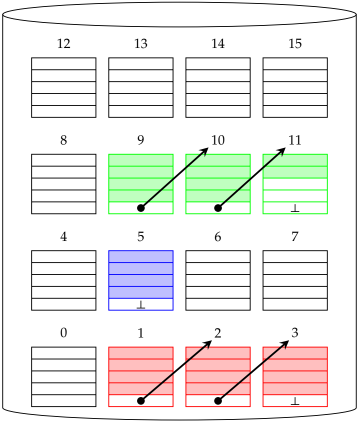


Mechanism: blocks $\rightsquigarrow$ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Idea: **linked allocation**.
  - \+ allocation is less challenging,
  - \+ sequential access is efficient,
  - − random access is inefficient,
  - \+ internal fragmentation only,
  - − some storage overhead due to pointers.

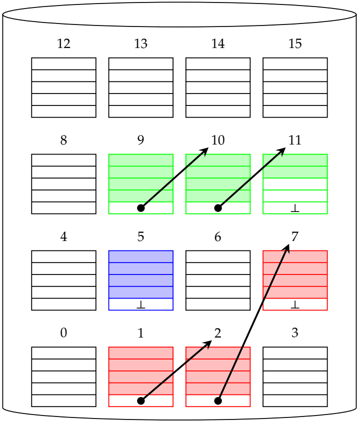


Mechanism: blocks $\rightsquigarrow$ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Idea: **linked allocation**.
  + allocation is less challenging,
  + sequential access is efficient,
  − random access is inefficient,
  + internal fragmentation only,
  − some storage overhead due to pointers.

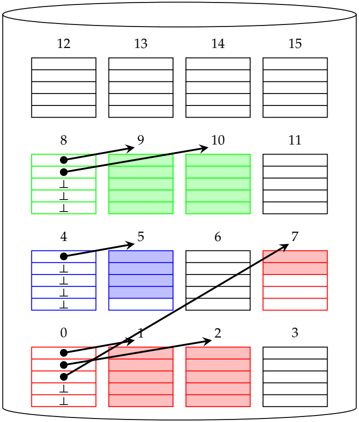


Mechanism: blocks $\rightsquigarrow$ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Idea: **indexed allocation**.
  - \+ allocation is less challenging,
  - \+ sequential *and* random access is efficient,
  - \+ internal fragmentation only,
  - − some storage overhead due to pointers.

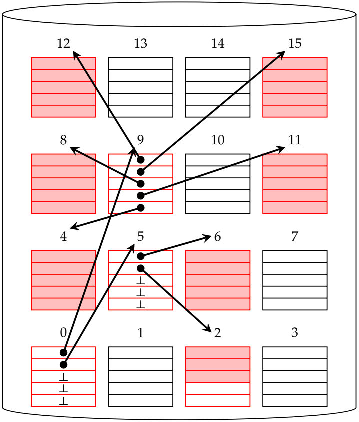


Mechanism: blocks $\rightsquigarrow$ files (1)

- Challenge: given a device with
  - fixed number of logical blocks and
  - fixed sized logical blocks,

  realise (hierarchical) file system supporting
  - representation and
  - manipulation

  of
  - variable number of files, and
  - variable sized files.
- Problem: what if the index block is full?
- Solution(s): use *multiple* index blocks, e.g., via
  1. linked list,
  2. linked tree, or
  3. various hybrid(s) ...



Mechanism: blocks $\rightsquigarrow$ files (2)

- **Problem**: larger storage capacity means more logical blocks, so
  1. larger logical block addresses,
  2. decreased access locality, and
  3. greater overhead (in time and space) wrt. allocation.
- **Solution**: use a hybrid part contiguous, part non-contiguous approach, e.g.,
  1. **cluster** $\simeq$ fixed size, contiguous group of logical blocks:
     - e.g., divide w-bit logical block address into two

       $$\text{logical block group address} = \underbrace{\boxed{\text{offset}}}_{w-1 \ldots t}\,\underbrace{\boxed{00\ldots0}}_{t-1 \ldots 0}$$

     - each offset now addresses a contiguous group of 2^t logical blocks.
  2. **extent** $\simeq$ variable size, contiguous group of logical blocks:
     - e.g., divide w-bit logical block address into two

       $$\text{logical block group address} = \underbrace{\boxed{\text{offset}}}_{w-1 \ldots t}\,\underbrace{\boxed{\text{length}}}_{t-1 \ldots 0}$$

     - each offset now addresses a contiguous group of logical blocks whose length is given by the t LSBs,
     - this is more flexible, *but* yields complications wrt. seeking and allocation.

  although from here on we ignore this option.

Mechanism: blocks $\leadsto$ files (3)

- Problem: each write requires one or more of

  1. update the allocation state,
  2. update the file meta-data, *and*
  3. update the file data

  which *must* be **atomic**: if not, the file system can become inconsistent.

- Solution:
  - describe update in write-ahead log (or journal),
  - commit update to file system iff. write to log is complete

Mechanism: blocks $\rightsquigarrow$ files (4)

- Question: what *is* a directory?
- Answer: a mapping, e.g.,

$$\text{identifier} \;\mapsto\; (\text{meta-data}, \text{data})$$

or

$$(\text{identifier}, \text{meta-data}) \;\mapsto\; \text{data}$$

which also hint at

1. options for where meta-data should reside, and
2. the fact a file might not *itself* have an identifier!

Mechanism: blocks $\rightsquigarrow$ files (4)

- Question: what *is* a directory?
- Answer: a mapping, e.g.,

$$\text{identifier} \mapsto (\text{meta-data}, \text{data})$$

or

$$(\text{identifier}, \text{meta-data}) \mapsto \text{data}$$

which also hint at

1. options for where meta-data should reside, and
2. the fact a file might not *itself* have an identifier!

- Problem: *how* should a directory be represented?
- Solution:
  1. list,
  2. tree,
  3. hash table,
  4. ...

Mechanism: blocks $\rightsquigarrow$ files (4)

- Question: what *is* a directory?
- Answer: a mapping, e.g.,

$$\text{identifier} \mapsto (\text{meta-data}, \text{data})$$

  or

$$(\text{identifier}, \text{meta-data}) \mapsto \text{data}$$

  which also hint at

  1. options for where meta-data should reside, and
  2. the fact a file might not *itself* have an identifier!

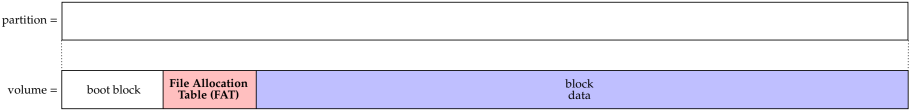
- Problem: *where* should a directory representation be stored?
- Solution:

  1. as a file, plus special-purpose rules for access,
  2. as a special-purpose structure,
  3. ...

  i.e., unified or segregated wrt. the rest of the file system.

Implementation: devices ⇝ file systems (1)
Windows-centric: FAT

- Idea: **File Allocation Table (FAT)** ≃ fancy linked allocation.



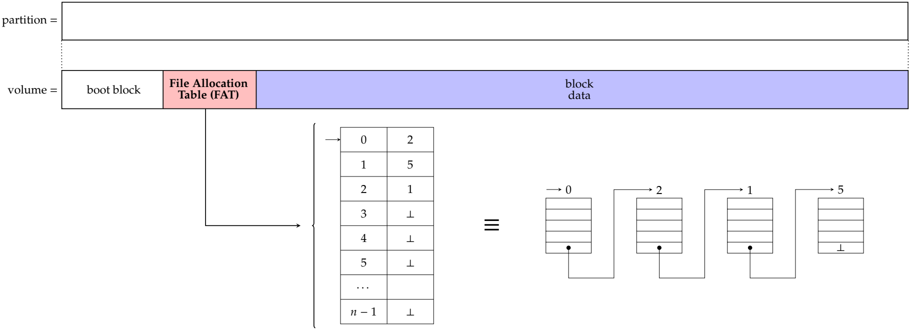
noting that

+ resolves issue of random access wrt. linked allocation,
– need to retain FAT in memory ... which, for n logical blocks, can be large!

Implementation: devices $\rightsquigarrow$ file systems (1)

Windows-centric: FAT

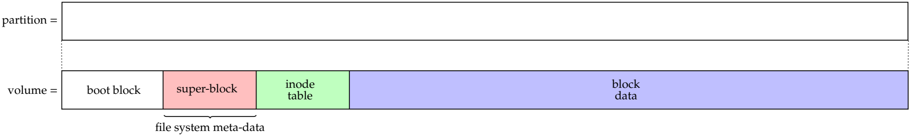
- Idea: **File Allocation Table (FAT)** $\simeq$ fancy linked allocation.



noting that

+ resolves issue of random access wrt. linked allocation,
− need to retain FAT in memory ... which, for n logical blocks, can be large!

- Idea: **Unix File System (UFS)** [15, Section 4] ≃ fancy indexed allocation.

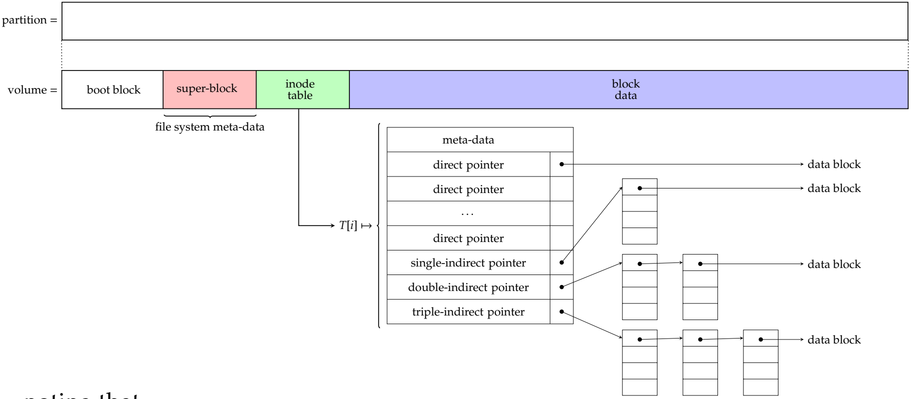


noting that

+ inodes are of (small) fixed size, so indexing into the inode table is efficient,
– linked representation of free space (for inodes *and* blocks).

Implementation: devices ⇝ file systems (2)
UNIX-centric: UFS

- Idea: **Unix File System (UFS)** [15, Section 4] ≃ fancy indexed allocation.



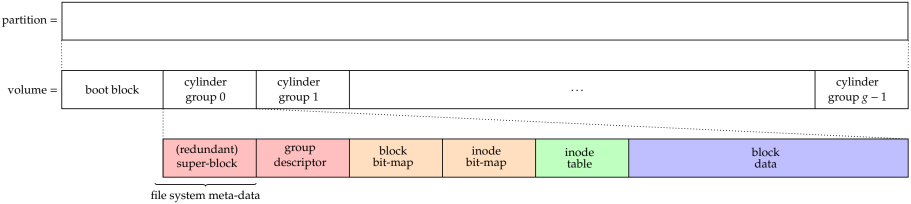
noting that

+ inodes are of (small) fixed size, so indexing into the inode table is efficient,
– linked representation of free space (for inodes *and* blocks).

Implementation: devices ⇝ file systems (3)

UNIX-centric: FFS

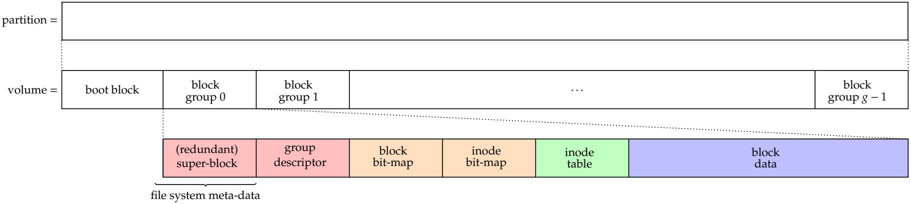
- **Idea**: **Fast File System (FFS)** [8] ≃ UFS + larger block size (≥ 4KiB).



noting that

+ bit-map representation of free space (for inodes *and* blocks),
+ redundant copy of super-block improves fault tolerance (for overhead of space),
+ cylinder groups increase access locality,
+ includes additional features such as soft links.

- Idea: **Second Extended File System (ext2)** [9] ≃ FFS + caching.



noting that

+ improved directory representation via hash tables [9, Section 4.2],
+ caching and asynchronous writes improve performance ...
− ... but with disadvantages wrt. coherence (viz. robustness).

Conclusions

- Take away points:
  - This is a broad and complex topic: it involves (at least)
    1. a hardware aspect:
       - an interrupt controller,
       - a block device
    2. a low(er)-level software aspect:
       - an interrupt handler,
       - a device driver,
       - a file system driver
    3. a high(er)-level software aspect:
       - some data structures (e.g., mount and file descriptor tables),
       - any relevant POSIX system calls (e.g., `write`)
  - Keep in mind that, even then,
    - we've excluded and/or simplified various (sub-)topics,
    - there are numerous trade-offs involved, meaning it is often hard to identify one ideal solution.

References

[1] Wikipedia: Disk formatting.
https://en.wikipedia.org/wiki/Disk_formatting.

[2] Wikipedia: Disk partitioning.
https://en.wikipedia.org/wiki/Disk_partitioning.

[3] Wikipedia: File system.
https://en.wikipedia.org/wiki/File_system.

[4] Wikipedia: HTree.
https://en.wikipedia.org/wiki/HTree.

[5] Wikipedia: Magnetic storage.
http://en.wikipedia.org/wiki/Magnetic_storage.

[6] Wikipedia: Master Boot Record (MRB).
https://en.wikipedia.org/wiki/Master_boot_record.

[7] Wikipedia: Optical disk.
http://en.wikipedia.org/wiki/Optical_disc.

[8] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry.
A fast file system for UNIX.
ACM Transactions on Computer Systems, 2(3):181–197, 1984.

References

[9] D. Poirier.
Second extended file system.
http://www.nongnu.org/ext2-doc/.

[10] V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau.
Analysis and evolution of journaling file systems.
In *USENIX Annual Technical Conference (ATEC)*, pages 105–120, 2005.

[11] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 11: Implementing file systems.
In *Operating System Concepts* [13].

[12] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 12: Mass storage structure.
In *Operating System Concepts* [13].

[13] A. Silberschatz, P.B. Galvin, and G. Gagne.
Operating System Concepts.
Wiley, 9th edition, 2014.

[14] A.S. Tanenbaum and H. Bos.
Chapter 4: File systems.
In *Modern Operating Systems*. Pearson, 4th edition, 2015.

References

[15] K. Thompson.
UNIX implementation.
Bell System Technical Journal, 57(6):1931–1946, 1978.

[16] S.C. Tweedie.
Journaling the Linux ext2fs filesystem.
In *LinuxExpo*, 1998.