- **Problem**:
  - memory represents short-term (transient, volatile) storage, so
  - we need explicit control over long-term (permanent, non-volatile) storage, st.

1.	**persistence**	:	decouple lifetime of data from any given process
2.	**organisation**	:	unambiguously and conveniently identify data
3.	**efficiency**	:	low-latency, random access to data
4.	**robustness**	:	accessing or duplicating data is error-free

- **Solution**: we need to consider
  - what an appropriate system call interface should be, *and*
  - how to map the semantics of said interface onto one or more concrete storage devices.

Concept (1)

Definition

- A **file** is a logical unit of (stored) data.
- A **file system** is an abstraction mechanism: it allows logical manipulation of files, without knowledge of their physical representation.

Concept (1)

Definition

A **file system** provides a mapping

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

plus a mechanism to manage (concurrent) manipulation of both

$$\begin{array}{rcl} \text{data} & \simeq & \text{content} \\ \text{meta-data} & \simeq & \text{structure} \end{array}$$

Concept (1)

Definition

A **file system** provides a mapping

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

plus a mechanism to manage (concurrent) manipulation of both

$$\begin{array}{rcl} \text{data} & \simeq & \text{content} \\ \text{meta-data} & \simeq & \text{structure} \end{array}$$

- Question: why be so abstract?

Concept (1)

Definition

A **file system** provides a mapping

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

plus a mechanism to manage (concurrent) manipulation of both

$$\begin{array}{rcl} \text{data} & \simeq & \text{content} \\ \text{meta-data} & \simeq & \text{structure} \end{array}$$

- Question: why be so abstract?
- Answer: file systems support multiple use-cases, e.g.,
  1. general-purpose data storage,
  2. special-purpose data storage (e.g., swap space [4]), or
  3. interface with kernel

  so saying "file" rather than "data" *may* be artificially limiting.

Concept (2)

- Option: the mapping can range between

  $l = 1$ ⟷ $l = \infty$

  flat ⟷ hierarchical

  with $l > 1$ implying

  - entries may be **directories**,
  - the identifier specifying an entry includes a (potentially implicit) **path**,
  - paths can be

    - **absolute** (from *root* directory) or
    - **relative** (from *some* directory)

    and hence needn't be unique.

root directory

```
├── foo.txt
├── bar.txt
├── baz.txt
└── ...
```

Concept (2)

- Option: the mapping can range between

  $$l = 1 \longleftrightarrow l = \infty$$

  flat ⟷ hierarchical

  with $l > 1$ implying

  - entries may be **directories**,
  - the identifier specifying an entry includes a (potentially implicit) **path**,
  - paths can be
    - **absolute** (from *root* directory) or
    - **relative** (from *some* directory)

    and hence needn't be unique.

root directory

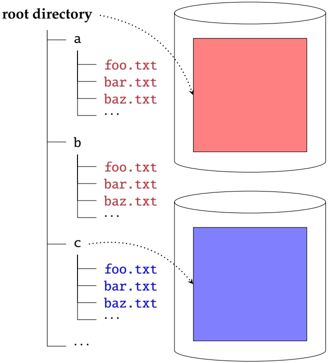
```
├── a
│   ├── foo.txt
│   ├── bar.txt
│   ├── baz.txt
│   └── ...
├── b
│   ├── foo.txt
│   ├── bar.txt
│   ├── baz.txt
│   └── ...
└── ...
```

Concept (2)

- Option: the root can be
  1. a **volume** identifier, or
  2. a directory

  where

  - the former implies multiple, segregated hierarchies,
  - the latter implies one, unified hierarchy ...
  - ... we **mount** a volume at a **mount point**.



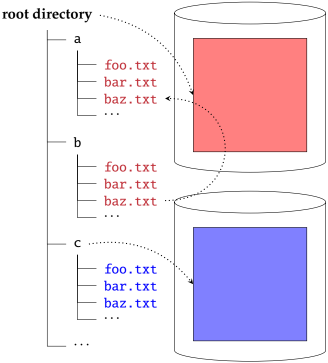
Concept (2)

- Option: entries in the hierarchy include
  - a file,
  - a directory,
  - a **symbolic link**, and
  - a **device node**

  where

  - entry types may be differentiated via meta-data or embedded magic numbers,
  - links are often categorised as either
    - **hard**, or
    - **soft**

    but, either way, imply the hierarchy is now a (acyclic) *graph* (vs. a tree).



Mechanism: POSIX(ish) system call interface (1)

- Assumption: underlying file system allows us to view data as a byte sequence, i.e.,

$$\text{identifier} \mapsto (\text{meta-data}, \text{data}) = (\text{meta-data}, \begin{array}{|c|c|c|c|}\hline d_0 & d_1 & \cdots & d_{n-1} \\ \hline \end{array})$$

read/write pointer

that supports

1. automatic extensibility, and
2. random access (via **seek** operations).

Mechanism: POSIX(ish) system call interface (2)

- Based on this assumption, the kernel must (at least)
  1. maintain a global **mount table** that captures the hierarchy,

Mechanism: POSIX(ish) system call interface (2)

- Based on this assumption, the kernel must (at least)

2. maintain a per process **file descriptor table** that captures
   - a **File Control Block (FCB)** of physical addressing information,
   - the mode the entry was opened in (e.g., read, write, or read/write),
   - the current read/write pointer

   that indexes into a global **file table** tracking open entries,

Mechanism: POSIX(ish) system call interface (2)

- Based on this assumption, the kernel must (at least)

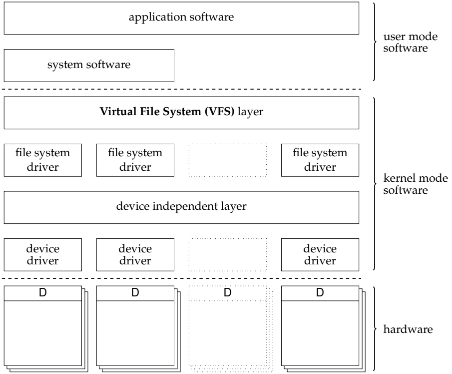
3. support a suite of system calls, e.g.,

Function	Reference	Purpose
`creat`	[11, Page 702]	create a file
`open`	[11, Page 1379]	open a file
`close`	[11, Page 676]	close a file
`unlink`	[11, Page 2154]	delete a file
`write`	[11, Page 2263]	write to a file
`read`	[11, Page 1737]	read from a file
`lseek`	[11, Page 1265]	move read/write pointer

which are ...

Mechanism: POSIX(ish) system call interface (3)

- ... (typically) exposed via a **Virtual File System (VFS)** layer



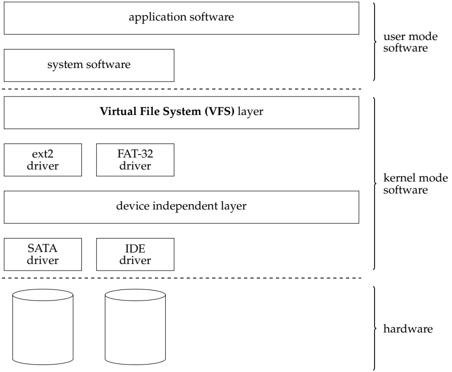
offering

1. a uniform interface to
   - multiple heterogeneous concrete file systems, and
   - "device-less" pseudo-files

   plus
2. various optimisation and translation operations.

Mechanism: POSIX(ish) system call interface (3)

- ... (typically) exposed via a **Virtual File System (VFS)** layer



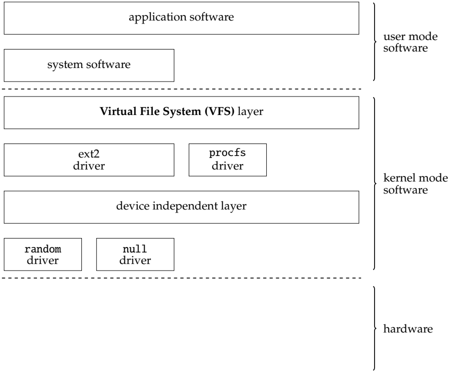
offering

1. a uniform interface to
   - multiple heterogeneous concrete file systems, and
   - "device-less" pseudo-files

   plus
2. various optimisation and translation operations.

Mechanism: POSIX(ish) system call interface (3)

- ... (typically) exposed via a **Virtual File System (VFS)** layer



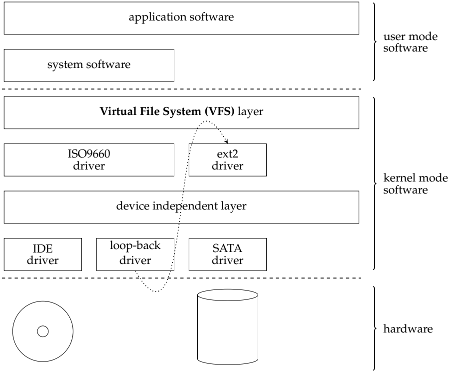
offering

1. a uniform interface to
   - multiple heterogeneous concrete file systems, and
   - “device-less” pseudo-files

   plus
2. various optimisation and translation operations.

Mechanism: POSIX(ish) system call interface (3)

- ... (typically) exposed via a **Virtual File System (VFS)** layer



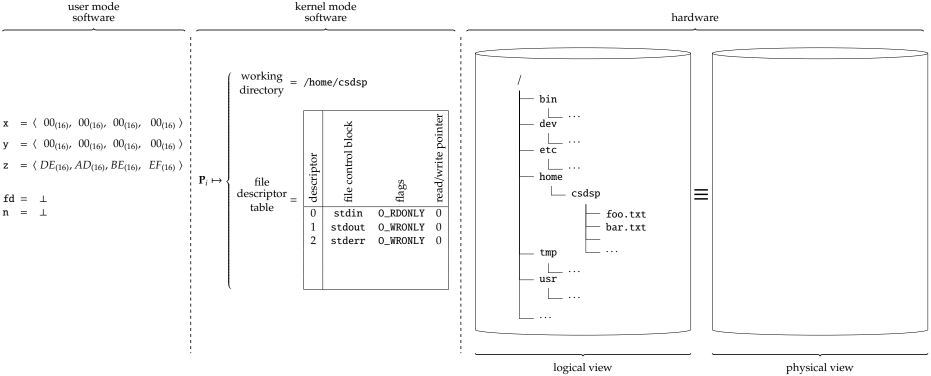
offering

1. a uniform interface to
   - multiple heterogeneous concrete file systems, and
   - "device-less" pseudo-files

   plus
2. various optimisation and translation operations.

Mechanism: POSIX(ish) system call interface (4)

- Example:

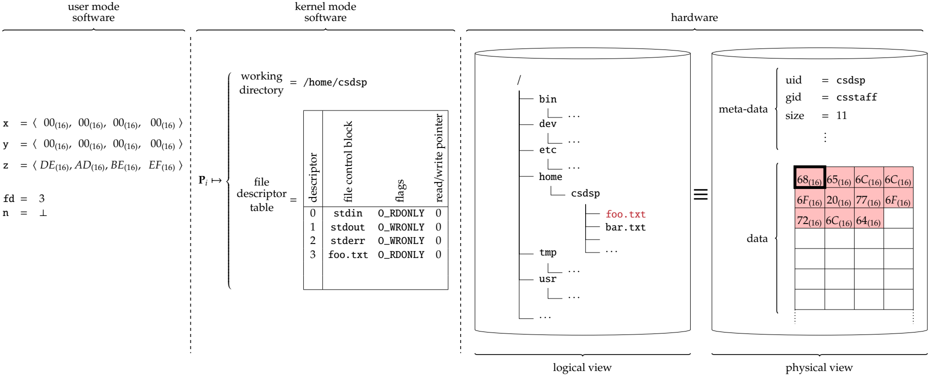


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
fd = open( "foo.txt", O_RDONLY )
```

then the result is described by

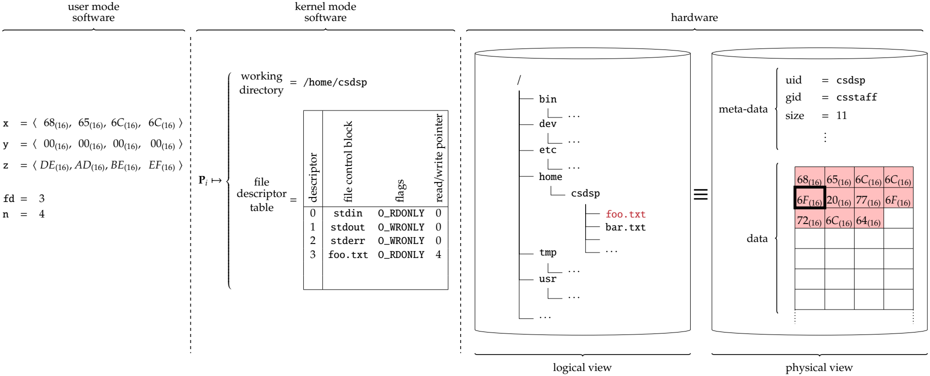


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = read( fd, x, 4 )
```

then the result is described by

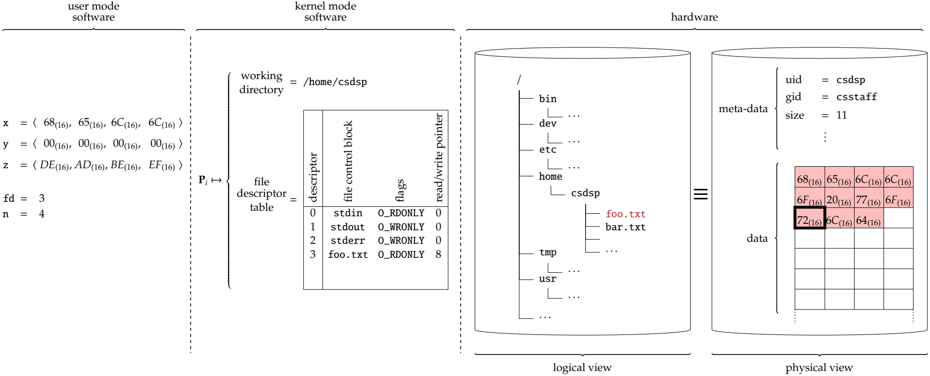


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
lseek( fd, +8, SEEK_SET )
```

then the result is described by

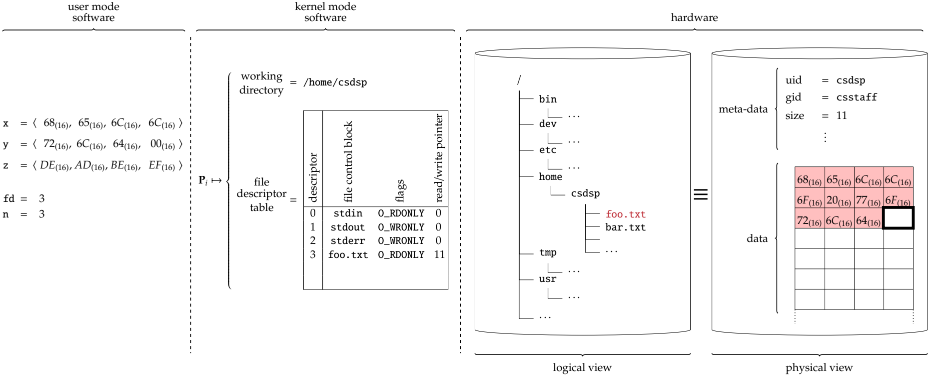


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = read( fd, y, 4 )
```

then the result is described by

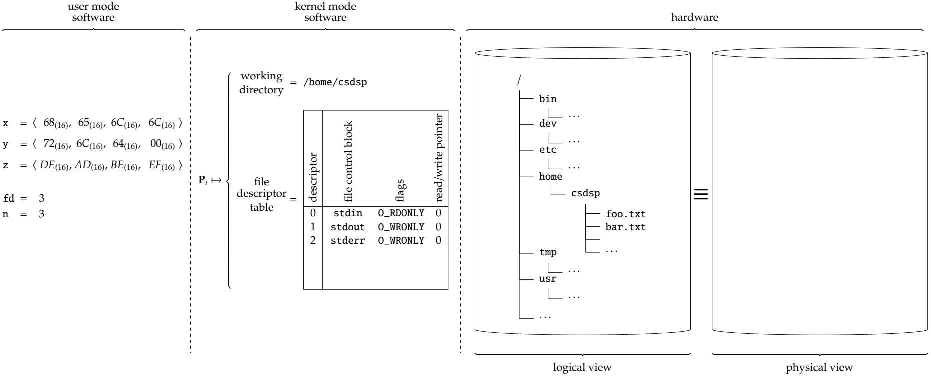


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by

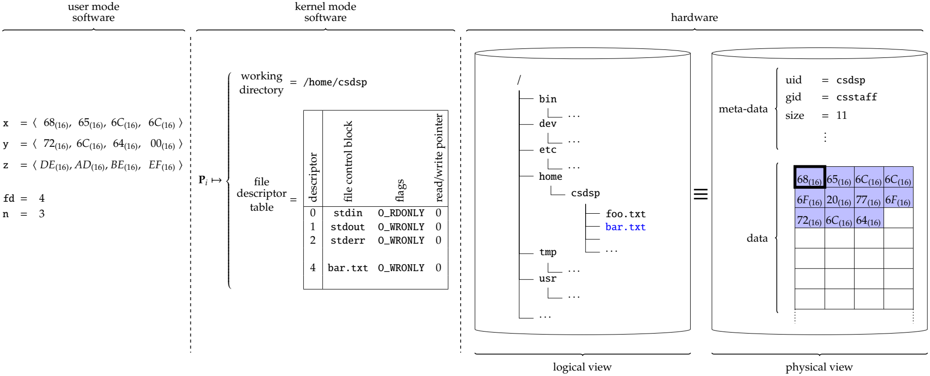


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
fd = open( "bar.txt", O_WRONLY )
```

then the result is described by

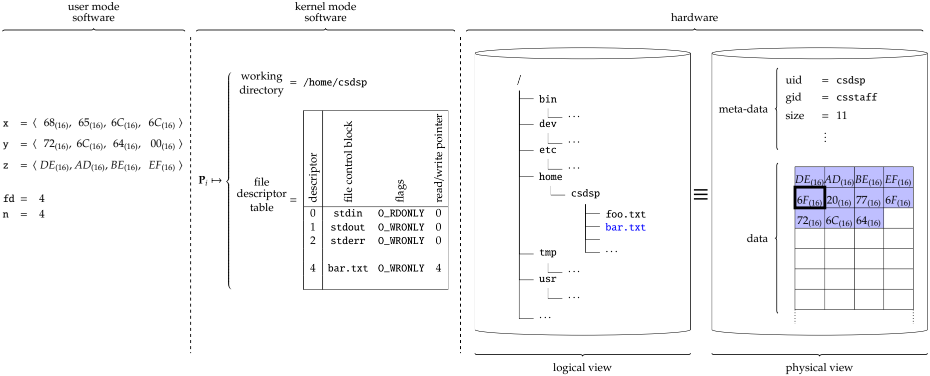


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = write( fd, z, 4 )
```

then the result is described by

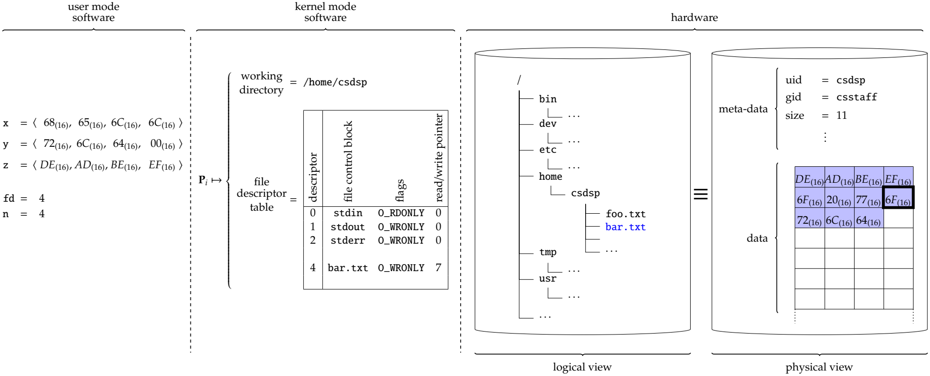


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
lseek( fd, -4, SEEK_END )
```

then the result is described by

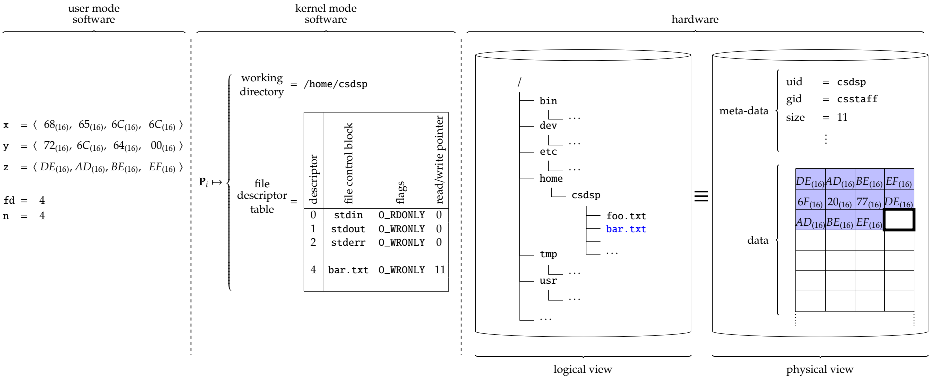


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = write( fd, z, 4 )
```

then the result is described by

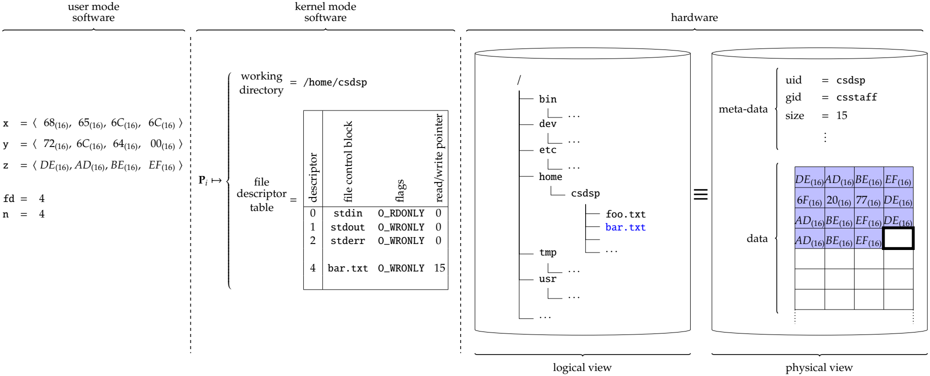


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = write( fd, z, 4 )
```

then the result is described by

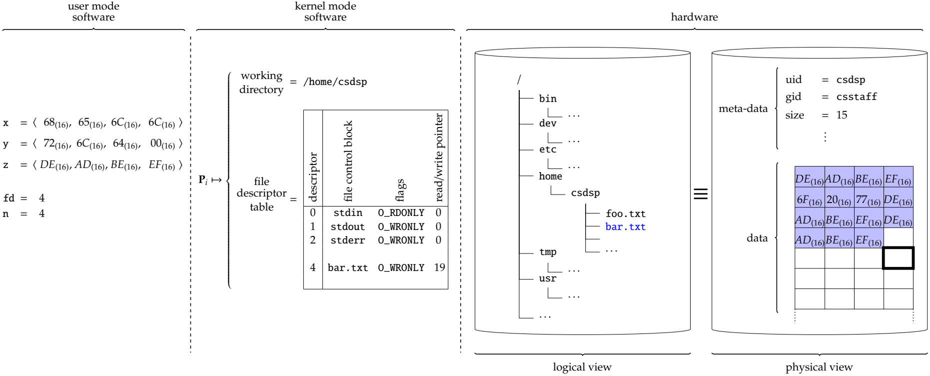


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
lseek( fd, +4, SEEK_END )
```

then the result is described by

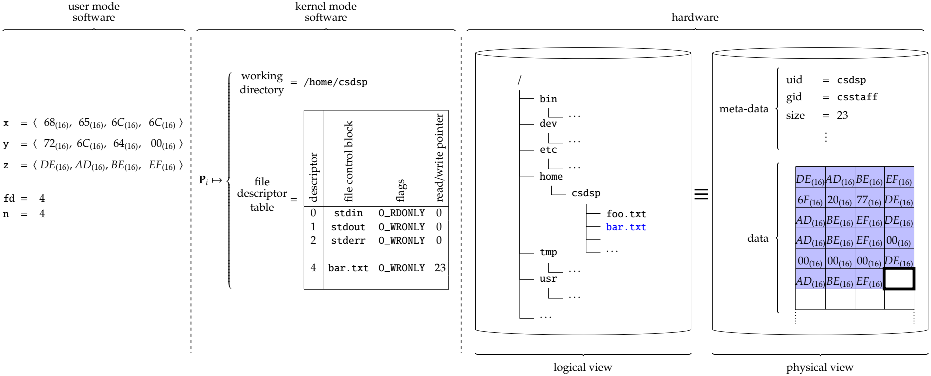


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = write( fd, z, 4 )
```

then the result is described by

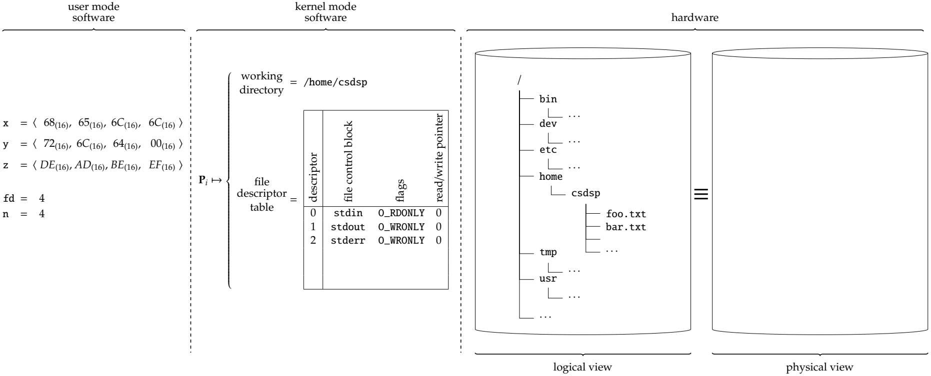


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by

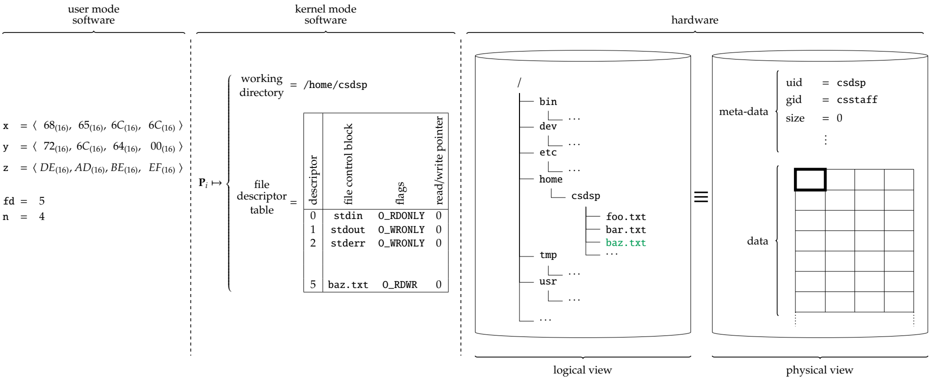


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
fd = creat( "baz.txt", O_WRONLY )
```

then the result is described by

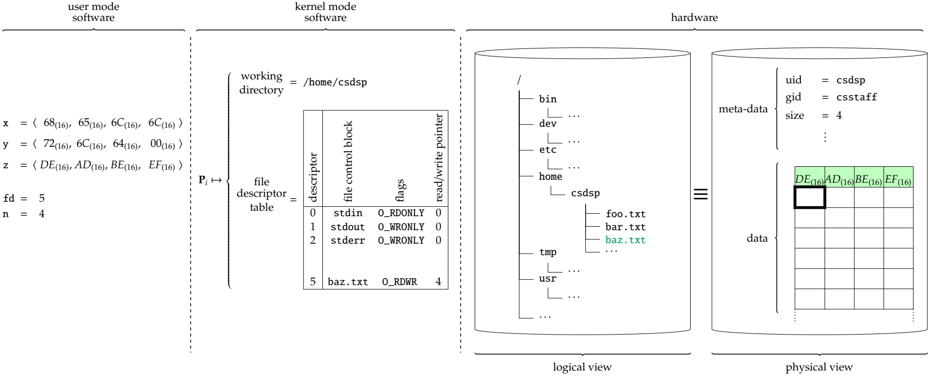


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = write( fd, z, 4 )
```

then the result is described by

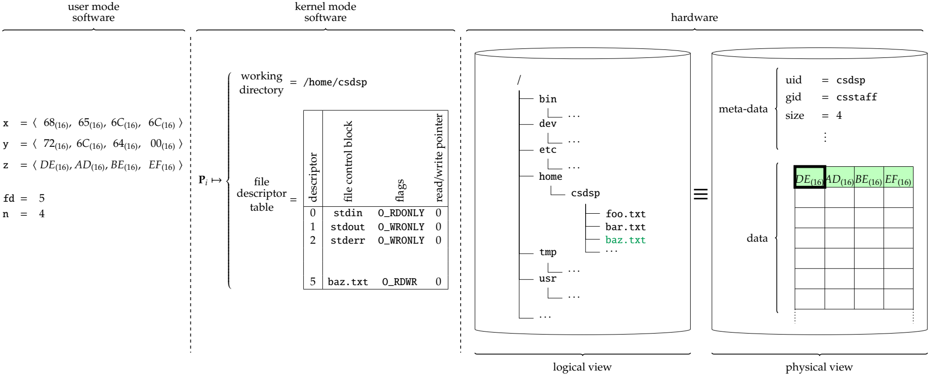


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
lseek( fd, +0, SEEK_SET )
```

then the result is described by

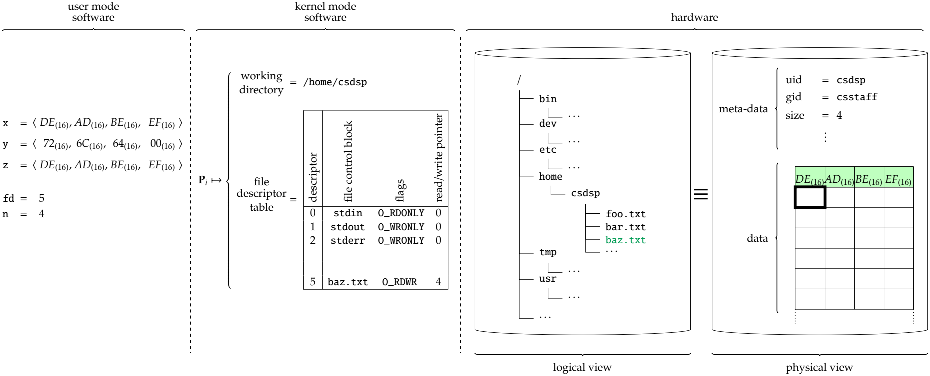


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
n = read( fd, x, 4 )
```

then the result is described by

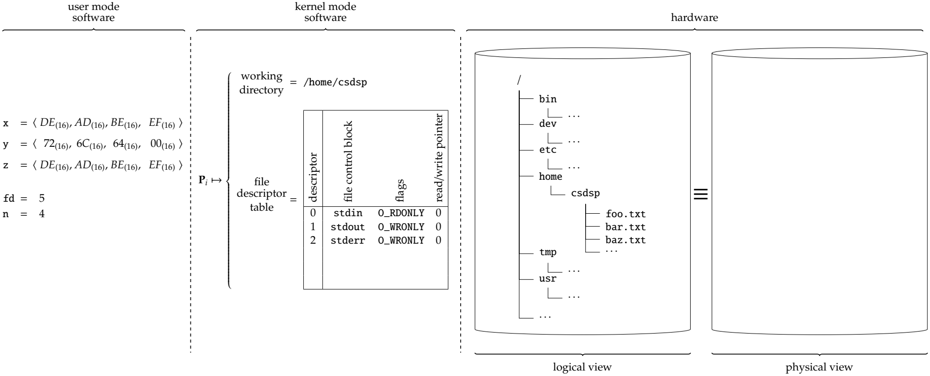


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by

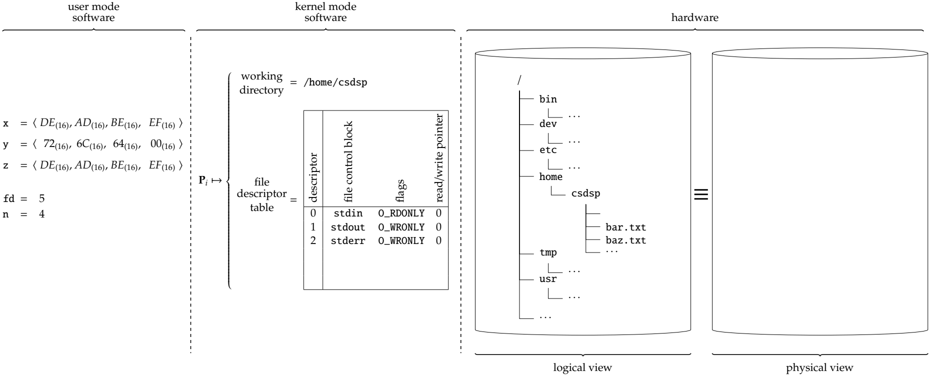


Mechanism: POSIX(ish) system call interface (4)

- Example: if the user mode process executes

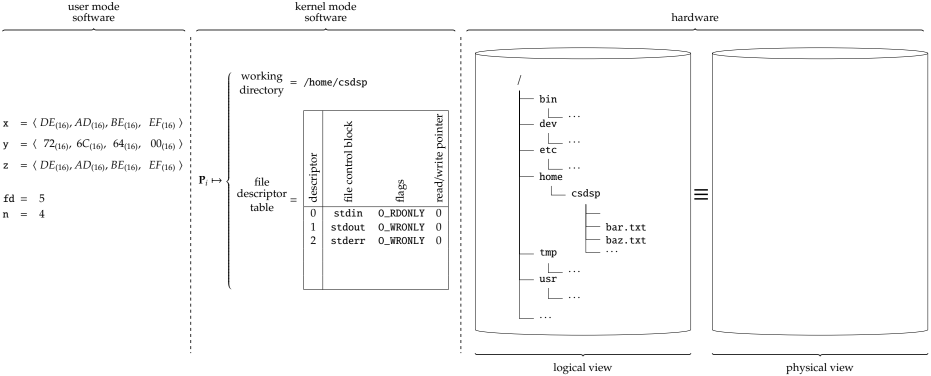
```
unlink( "foo.txt" )
```

then the result is described by



Mechanism: POSIX(ish) system call interface (4)

- Example:



Concept (1)

- ... 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.

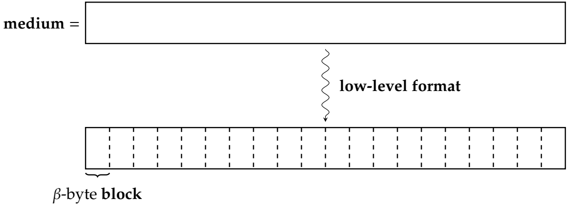
Concept (2): (mass storage) devices $\leadsto$ blocks

- We add structure to the medium via several steps

medium = $\boxed{}$

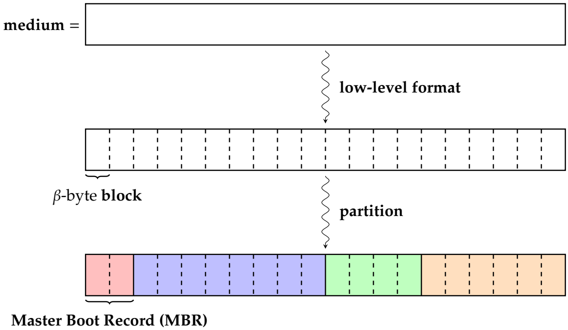
Concept (2): (mass storage) devices $\leadsto$ blocks

- We add structure to the medium via several steps



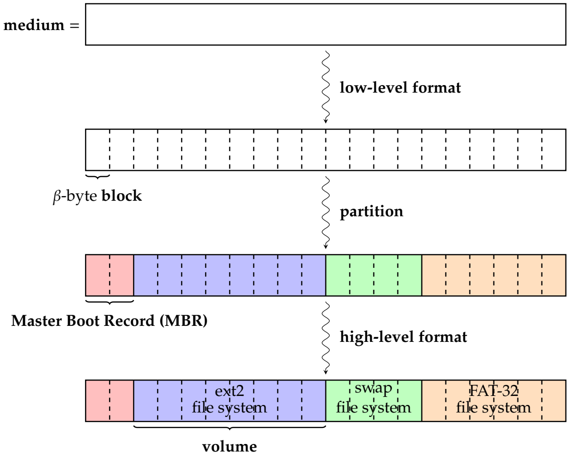
Concept (2): (mass storage) devices $\rightsquigarrow$ blocks

- We add structure to the medium via several steps



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

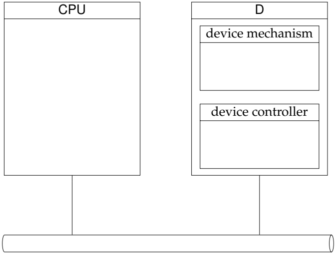
- We add structure to the medium via several steps



then ...

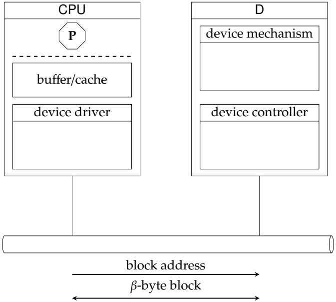
Concept (3): (mass storage) devices $\rightsquigarrow$ blocks

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



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

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



but, since efficiency is crucial we will (typically)

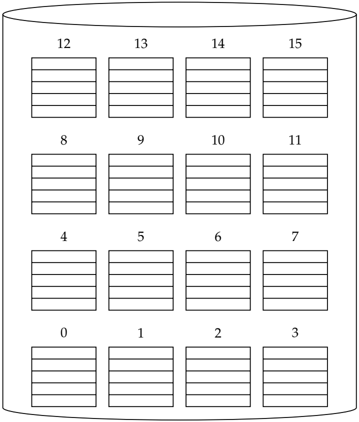
- amortise overhead by fixing transferring β-byte blocks, and
- buffer and/or cache accesses.

Concept (4): blocks ⇝ files

- 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.

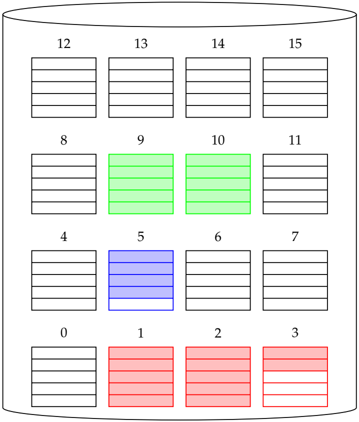


Concept (4): blocks $\rightsquigarrow$ files

- 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.

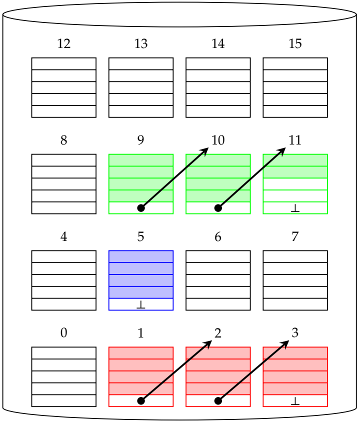


Concept (4): blocks $\rightsquigarrow$ files

- 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.



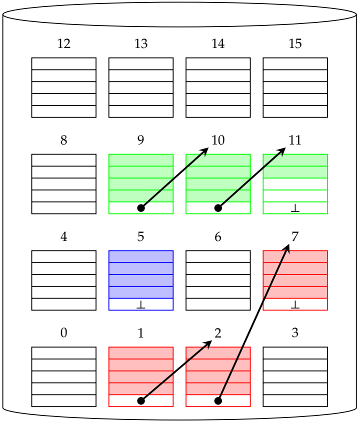
Concept (4): blocks $\rightsquigarrow$ files

- 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.

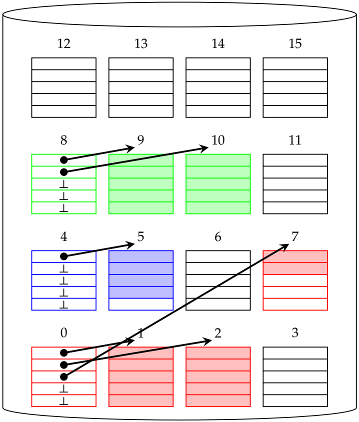


Concept (4): blocks ⇝ files

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

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

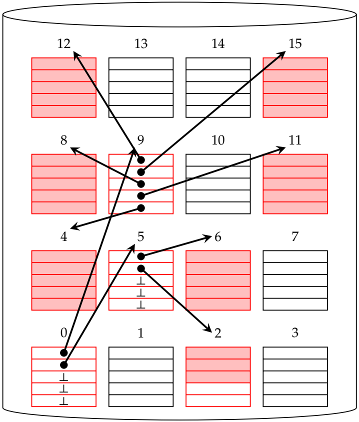
  of
  - variable number of files, and
  - variable sized files.
- Idea: **indexed allocation**.
  - \+ allocation is less challenging,
  - \+ sequential *and* random access is efficient,
  - \+ internal fragmentation only,
  - − some storage overhead due to pointers.



- 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) ...



Concept (5): blocks $\rightsquigarrow$ files

- 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

       logical block group address =

$w-1$ … t	$t-1$ … 0
offset	$00\ldots0$

     - 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

       logical block group address =

$w-1$ … t	$t-1$ … 0
offset	length

     - 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.

Concept (6): blocks $\rightsquigarrow$ files

- 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.

Concept (7): blocks $\rightsquigarrow$ files

- 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!

Concept (7): blocks $\rightsquigarrow$ files

- 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. ...

Concept (7): blocks $\rightsquigarrow$ files

- 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

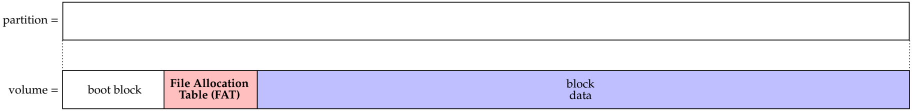
1. options for where meta-data should reside, and
2. the fact a file might not *itself* have an identifier!

- 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.

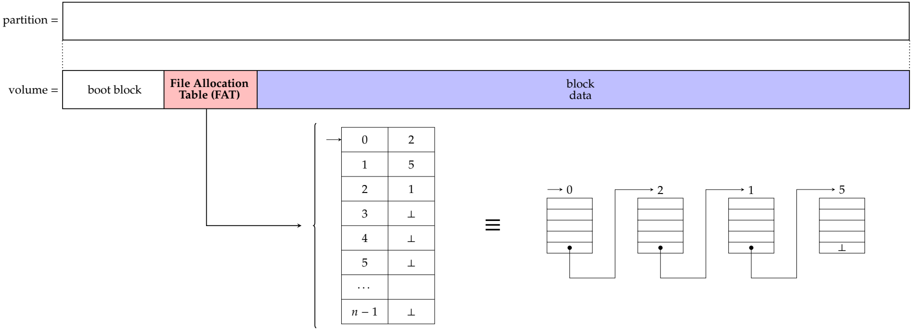
- Idea: **File Allocation Table (FAT)** $\simeq$ fancy linked allocation.



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

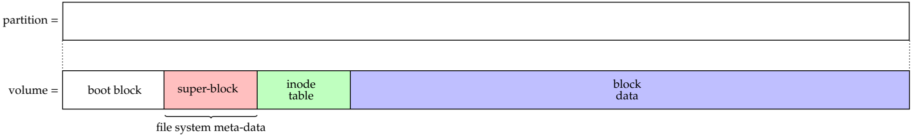


0	2
1	5
2	1
3	$\perp$
4	$\perp$
5	$\perp$
...	
$n-1$	$\perp$

noting that

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

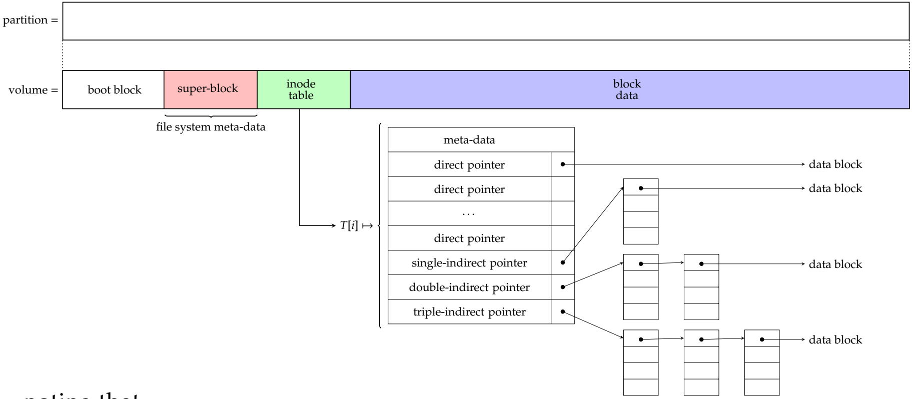
- Idea: **Unix File System (UFS)** [14, Section 4] ≃ fancy indexed allocation.



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).

- **Idea**: **Unix File System (UFS)** [14, Section 4] $\simeq$ fancy indexed allocation.

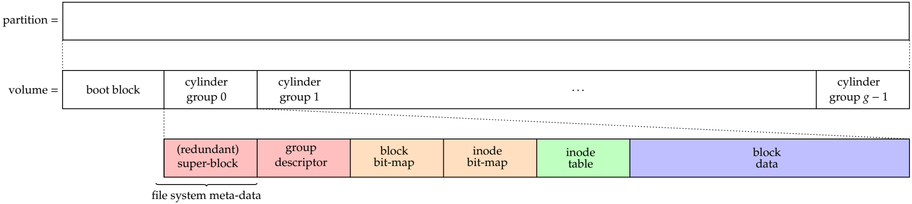


noting that

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

Implementation (3): devices ⇝ file systems
UNIX-centric: FFS

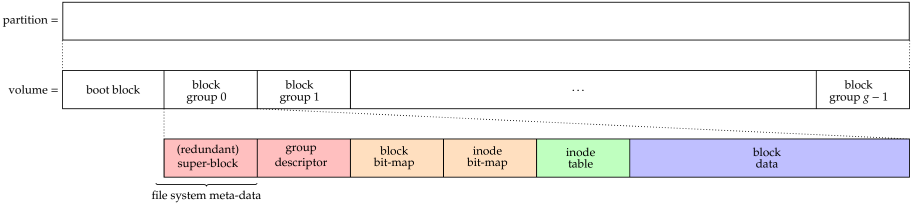
- Idea: **Fast File System (FFS)** [12] ≃ UFS + larger block size (≥ 4KiB).



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)** [3] $\simeq$ FFS + caching.



noting that

+ improved directory representation via hash tables [3, 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.

Additional Reading

- *Wikipedia: File system.* URL: http://en.wikipedia.org/wiki/File_system.
- R. Love. "Chapter 2: File I/O". . In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.
- R. Love. "Chapter 8: File and directory management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 10: File system". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 11: Implementing file systems". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 12: Mass storage structure". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A.S. Tanenbaum and H. Bos. "Chapter 4: File systems". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.

References

[1] *Wikipedia: File system*. URL: http://en.wikipedia.org/wiki/File_system (see p. 64).

[2] *Wikipedia: HTree*. URL: http://en.wikipedia.org/wiki/HTree.

[3] D. Poirier. *Second Extended File System*. URL: http://www.nongnu.org/ext2-doc (see p. 62).

[4] M. Gorman. "Chapter 11: Swap management". In: *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004. URL: http://www.kernel.org/doc/gorman/ (see pp. 2–5).

[5] R. Love. "Chapter 2: File I/O". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 64).

[6] R. Love. "Chapter 8: File and directory management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 64).

[7] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 10: File system". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 64).

[8] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 11: Implementing file systems". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 64).

[9] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 12: Mass storage structure". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 64).

[10] A.S. Tanenbaum and H. Bos. "Chapter 4: File systems". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 64).

[11] *Standard for Information Technology - Portable Operating System Interface (POSIX)*. Institute of Electrical and Electronics Engineers (IEEE) 1003.1-2008. 2008. URL: http://standards.ieee.org (see pp. 11–13).

[12] M.K. McKusick et al. "A Fast File System for UNIX". In: *ACM Transactions on Computer Systems (TOCS)* 2.3 (1984), pp. 181–197 (see p. 61).

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

[14] K. Thompson. "UNIX Implementation". In: *Bell System Technical Journal* 57.6 (1978), pp. 1931–1946 (see pp. 59, 60).

[15] S.C. Tweedie. "Journaling the Linux ext2fs Filesystem". In: *LinuxExpo*. 1998.