Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

February 16, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

Definition (**file system**)

- 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.

Notes:

- Some examples of where the file system is used as an interface with the kernel include
  - configuration (e.g., `/proc`),
  - device drivers (e.g., `/dev/sda`), or
  - pseudo-files (e.g., `/dev/random`).

Definition (**file system**)

A **file system** provides a mapping

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

plus a mechanism to manage (concurrent) manipulation of both

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

Notes:

- Some examples of where the file system is used as an interface with the kernel include
  - configuration (e.g., `/proc`),
  - device drivers (e.g., `/dev/sda`), or
  - pseudo-files (e.g., `/dev/random`).

Daniel Page ⟨Daniel.Page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15
University of BRISTOL

Definition (**file system**)

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?

Notes:

- Some examples of where the file system is used as an interface with the kernel include
  - configuration (e.g., `/proc`),
  - device drivers (e.g., `/dev/sda`), or
  - pseudo-files (e.g., `/dev/random`).

Definition (**file system**)

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.

Notes:

- Some examples of where the file system is used as an interface with the kernel include
  - configuration (e.g., `/proc`),
  - device drivers (e.g., `/dev/sda`), or
  - pseudo-files (e.g., `/dev/random`).

- Challenge(s): 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

  noting the former necessarily has a nod to user-friendliness.

Notes:

Concept (1)

- 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
└── ...
```

Notes:

- A path is used to identify entries in the file system: if the file system is hierarchical, the identifier is basically a sequence of path components (themselves identifiers in essence) separated by a distinguished character. For example, the path

  `a/foo.txt`

  is basically the 2-element sequence

  $\langle$`a`, `foo.txt`$\rangle$

  where the character '/' acts as a separator.
  Although we are consistent wrt. this definition here, use of the term could vary a little elsewhere: path is sometimes (ab)used to mean the sequence of directories that need to be traversed to eventually identify an entry. Put another way, path is sometimes (ab)used to refer to the directory name (i.e., those path components upto and including the last separator) with the term file name used to refer to the rest (i.e., after the last separator).
  Either way, various special-purpose path components are commonly permitted, with examples including `.` (the current directory), `..` (the parent directory, i.e., one level above the current directory) and ~ (the home directory of the current user).
- The fact the path may be implicit is meant to capture the following: if we omit the path, for example in `foo.txt`, we typically mean `./foo.txt`, i.e., the path is implicitly taken to mean the current directory.
- An example where a hierarchy is identified by (i.e., the root *is*) a volume identifier would be Windows: in this case, a (physical) device identifier such as `A:` or `C:` is used
- The difference between hard and soft (or symbolic) links is subtle, but quite important: conceptually you could view them as low- and high-level links (i.e., wrt. the concrete file system, or the logical file system as presented abstractly to the user). A hard link is basically an association between some identifier and some *underlying* data. This makes them indistinguishable from a normal entry, in the sense that any entry is a hard link to some data. However, they allow multiple identifiers to hard link to the *same* data; if one hard link is moved (or deleted) within the file system, other hard links to the same data are unaffected (since the link is to the data, not identifier). In contrast, a soft link is basically a reference using one identifier to some other identifier. These references can span file systems, and even reference non-existent targets; moving (or deleting) the target now invalidates the source.
  Note that the acyclic restriction on the resulting hierarchy depends on the inability to make hard links between directories: this restriction is useful in so far as it prevents numerous special, odd cases (e.g., allowing a given directory to be the parent directory of itself).

Concept (1)

- 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
│   └── ...
└── ...
```

Notes:

- A path is used to identify entries in the file system: if the file system is hierarchical, the identifier is basically a sequence of path components (themselves identifiers in essence) separated by a distinguished character. For example, the path

  `a/foo.txt`

  is basically the 2-element sequence

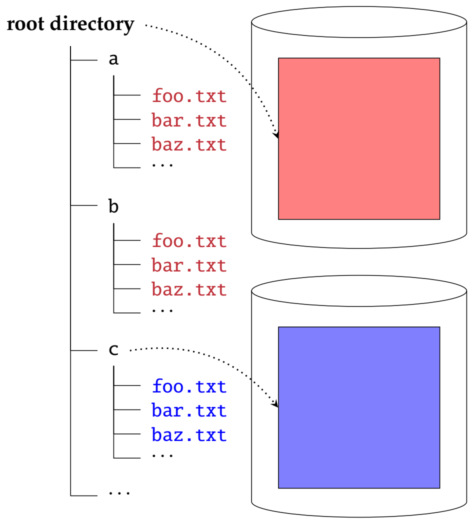
  $\langle$`a`, `foo.txt`$\rangle$

  where the character '/' acts as a separator.
  Although we are consistent wrt. this definition here, use of the term could vary a little elsewhere: path is sometimes (ab)used to mean the sequence of directories that need to be traversed to eventually identify an entry. Put another way, path is sometimes (ab)used to refer to the directory name (i.e., those path components upto and including the last separator) with the term file name used to refer to the rest (i.e., after the last separator).
  Either way, various special-purpose path components are commonly permitted, with examples including `.` (the current directory), `..` (the parent directory, i.e., one level above the current directory) and ~ (the home directory of the current user).
- The fact the path may be implicit is meant to capture the following: if we omit the path, for example in `foo.txt`, we typically mean `./foo.txt`, i.e., the path is implicitly taken to mean the current directory.
- An example where a hierarchy is identified by (i.e., the root *is*) a volume identifier would be Windows: in this case, a (physical) device identifier such as `A:` or `C:` is used
- The difference between hard and soft (or symbolic) links is subtle, but quite important: conceptually you could view them as low- and high-level links (i.e., wrt. the concrete file system, or the logical file system as presented abstractly to the user). A hard link is basically an association between some identifier and some *underlying* data. This makes them indistinguishable from a normal entry, in the sense that any entry is a hard link to some data. However, they allow multiple identifiers to hard link to the *same* data; if one hard link is moved (or deleted) within the file system, other hard links to the same data are unaffected (since the link is to the data, not identifier).
  In contrast, a soft link is basically a reference using one identifier to some other identifier. These references can span file systems, and even reference non-existent targets; moving (or deleting) the target now invalidates the source.
  Note that the acyclic restriction on the resulting hierarchy depends on the inability to make hard links between directories: this restriction is useful in so far as it prevents numerous special, odd cases (e.g., allowing a given directory to be the parent directory of itself).

Concept (1)

- 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**.



Notes:

- A path is used to identify entries in the file system: if the file system is hierarchical, the identifier is basically a sequence of path components (themselves identifiers in essence) separated by a distinguished character. For example, the path

  `a/foo.txt`

  is basically the 2-element sequence

  $\langle$`a`, `foo.txt`$\rangle$

  where the character '/' acts as a separator.
  Although we are consistent wrt. this definition here, use of the term could vary a little elsewhere: path is sometimes (ab)used to mean the sequence of directories that need to be traversed to eventually identify an entry. Put another way, path is sometimes (ab)used to refer to the directory name (i.e., those path components upto and including the last separator) with the term file name used to refer to the rest (i.e., after the last separator).
  Either way, various special-purpose path components are commonly permitted, with examples including `.` (the current directory), `..` (the parent directory, i.e., one level above the current directory) and ~ (the home directory of the current user).
- The fact the path may be implicit is meant to capture the following: if we omit the path, for example in `foo.txt`, we typically mean `./foo.txt`, i.e., the path is implicitly taken to mean the current directory.
- An example where a hierarchy is identified by (i.e., the root *is*) a volume identifier would be Windows: in this case, a (physical) device identifier such as `A:` or `C:` is used
- The difference between hard and soft (or symbolic) links is subtle, but quite important: conceptually you could view them as low- and high-level links (i.e., wrt. the concrete file system, or the logical file system as presented abstractly to the user). A hard link is basically an association between some identifier and some *underlying* data. This makes them indistinguishable from a normal entry, in the sense that any entry is a hard link to some data. However, they allow multiple identifiers to hard link to the *same* data; if one hard link is moved (or deleted) within the file system, other hard links to the same data are unaffected (since the link is to the data, not identifier).
  In contrast, a soft link is basically a reference using one identifier to some other identifier. These references can span file systems, and even reference non-existent targets; moving (or deleting) the target now invalidates the source.
  Note that the acyclic restriction on the resulting hierarchy depends on the inability to make hard links between directories: this restriction is useful in so far as it prevents numerous special, odd cases (e.g., allowing a given directory to be the parent directory of itself).

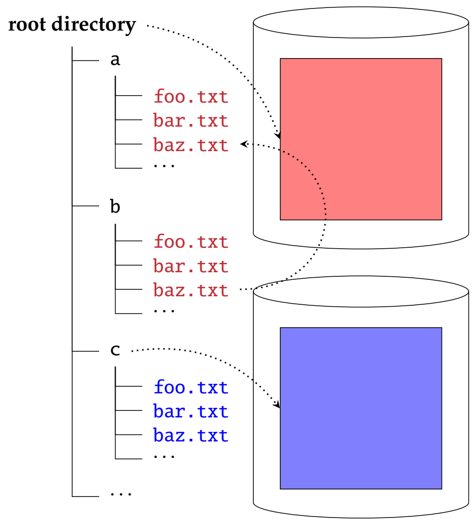
Concept (1)

- 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).



Notes:

- A path is used to identify entries in the file system: if the file system is hierarchical, the identifier is basically a sequence of path components (themselves identifiers in essence) separated by a distinguished character. For example, the path

  `a/foo.txt`

  is basically the 2-element sequence

  $\langle \texttt{a}, \texttt{foo.txt} \rangle$

  where the character '/' acts as a separator.
  Although we are consistent wrt. this definition here, use of the term could vary a little elsewhere: path is sometimes (ab)used to mean the sequence of directories that need to be traversed to eventually identify an entry. Put another way, path is sometimes (ab)used to refer to the directory name (i.e., those path components upto and including the last separator) with the term file name used to refer to the rest (i.e., after the last separator).
  Either way, various special-purpose path components are commonly permitted, with examples including `.` (the current directory), `..` (the parent directory, i.e., one level above the current directory) and ~ (the home directory of the current user).
- The fact the path may be implicit is meant to capture the following: if we omit the path, for example in `foo.txt`, we typically mean `./foo.txt`, i.e., the path is implicitly taken to mean the current directory.
- An example where a hierarchy is identified by (i.e., the root *is*) a volume identifier would be Windows: in this case, a (physical) device identifier such as `A:` or `C:` is used
- The difference between hard and soft (or symbolic) links is subtle, but quite important: conceptually you could view them as low- and high-level links (i.e., wrt. the concrete file system, or the logical file system as presented abstractly to the user). A hard link is basically an association between some identifier and some *underlying* data. This makes them indistinguishable from a normal entry, in the sense that any entry is a hard link to some data. However, they allow multiple identifiers to hard link to the *same* data; if one hard link is moved (or deleted) within the file system, other hard links to the same data are unaffected (since the link is to the data, not identifier). In contrast, a soft link is basically a reference using one identifier to some other identifier. These references can span file systems, and even reference non-existent targets; moving (or deleting) the target now invalidates the source.
  Note that the acyclic restriction on the resulting hierarchy depends on the inability to make hard links between directories: this restriction is useful in so far as it prevents numerous special, odd cases (e.g., allowing a given directory to be the parent directory of itself).

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

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

$$\text{identifier} \mapsto (\text{meta-data}, \text{data}) = (\text{meta-data}, \boxed{d_0} \boxed{d_1} \boxed{\cdots} \boxed{d_{l-1}})$$

read/write pointer

that supports

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

Notes:

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,

Notes:

- Exactly what an FCB constitutes depends on the underlying file system: we will see later that for UNIX-centric examples, the inode basically captures the concept. That said, the fact an FCB is used internally suggests a separation between a) an abstract, human-readable identifier (i.e., the path) used by the user, and b) a concrete, machine-readable identifier (e.g., an inode number) used by the kernel.
- In Linux, the file descriptor table is exposed by `procfs` st. each entry appears in

```
/proc/${PID}/fs
```

  for a process whose PID is `${PID}`.
- The table of POSIX system calls presented is far from exhaustive. For example, one might reasonably supplement it with

Function	Reference	Purpose
`mkdir`	[3, Page 1289]	create a directory
`opendir`	[3, Page 1391]	open a directory
`closedir`	[3, Page 680]	close a directory
`rmdir`	[3, Page 1790]	delete a directory
`readdir`	[3, Page 1744]	read from a directory
`stat`	[3, Page 1979]	get file status
`mknod`	[3, Page 1298]	create a device node
`link`	[3, Page 1216]	create a hard link
`symlink`	[3, Page 2057]	create a soft link

  to cope with directories (as a special-case of files) and also various auxiliary and niche operations.
- In Linux, each file descriptor used to index into the file descriptor table is essentially just an integer: you can see this by inspecting the declaration of associated system calls. For example,

```
int open( const char* identifier, int flags );
```

  is used to open a file and return the file descriptor, while

```
ssize_t write( int fd, const void* x, size_t n );
```

  is used to write data via a specific file descriptor. It is important to also note that 3 *standard* file descriptors exist, on a per process basis, relating to `stdin` (0), `stdout` (1) and `stderr` (2).

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,

Notes:

- Exactly what an FCB constitutes depends on the underlying file system: we will see later that for UNIX-centric examples, the inode basically captures the concept. That said, the fact an FCB is used internally suggests a separation between a) an abstract, human-readable identifier (i.e., the path) used by the user, and b) a concrete, machine-readable identifier (e.g., an inode number) used by the kernel.
- In Linux, the file descriptor table is exposed by `procfs` st. each entry appears in

```
/proc/${PID}/fs
```

  for a process whose PID is `${PID}`.
- The table of POSIX system calls presented is far from exhaustive. For example, one might reasonably supplement it with

Function	Reference	Purpose
`mkdir`	[3, Page 1289]	create a directory
`opendir`	[3, Page 1391]	open a directory
`closedir`	[3, Page 680]	close a directory
`rmdir`	[3, Page 1790]	delete a directory
`readdir`	[3, Page 1744]	read from a directory
`stat`	[3, Page 1979]	get file status
`mknod`	[3, Page 1298]	create a device node
`link`	[3, Page 1216]	create a hard link
`symlink`	[3, Page 2057]	create a soft link

  to cope with directories (as a special-case of files) and also various auxiliary and niche operations.
- In Linux, each file descriptor used to index into the file descriptor table is essentially just an integer: you can see this by inspecting the declaration of associated system calls. For example,

```
int open( const char* identifier, int flags );
```

  is used to open a file and return the file descriptor, while

```
ssize_t write( int fd, const void* x, size_t n );
```

  is used to write data via a specific file descriptor. It is important to also note that 3 *standard* file descriptors exist, on a per process basis, relating to `stdin` (0), `stdout` (1) and `stderr` (2).

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`	[3, Page 702]	create a file
`open`	[3, Page 1379]	open a file
`close`	[3, Page 676]	close a file
`unlink`	[3, Page 2154]	delete a file
`write`	[3, Page 2263]	write to a file
`read`	[3, Page 1737]	read from a file
`lseek`	[3, Page 1265]	move read/write pointer

which are ...

Notes:

- Exactly what an FCB constitutes depends on the underlying file system: we will see later that for UNIX-centric examples, the inode basically captures the concept. That said, the fact an FCB is used internally suggests a separation between a) an abstract, human-readable identifier (i.e., the path) used by the user, and b) a concrete, machine-readable identifier (e.g., an inode number) used by the kernel.
- In Linux, the file descriptor table is exposed by `procfs` st. each entry appears in

  `/proc/${PID}/fs`

  for a process whose PID is `${PID}`.
- The table of POSIX system calls presented is far from exhaustive. For example, one might reasonably supplement it with

Function	Reference	Purpose
`mkdir`	[3, Page 1289]	create a directory
`opendir`	[3, Page 1391]	open a directory
`closedir`	[3, Page 680]	close a directory
`rmdir`	[3, Page 1790]	delete a directory
`readdir`	[3, Page 1744]	read from a directory
`stat`	[3, Page 1979]	get file status
`mknod`	[3, Page 1298]	create a device node
`link`	[3, Page 1216]	create a hard link
`symlink`	[3, Page 2057]	create a soft link

  to cope with directories (as a special-case of files) and also various auxiliary and niche operations.
- In Linux, each file descriptor used to index into the file descriptor table is essentially just an integer: you can see this by inspecting the declaration of associated system calls. For example,

```
int open( const char* identifier, int flags );
```

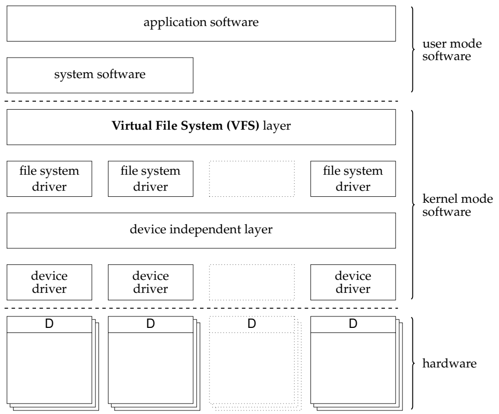
  is used to open a file and return the file descriptor, while

```
ssize_t write( int fd, const void* x, size_t n );
```

  is used to write data via a specific file descriptor. It is important to also note that 3 *standard* file descriptors exist, on a per process basis, relating to `stdin` (0), `stdout` (1) and `stderr` (2).

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

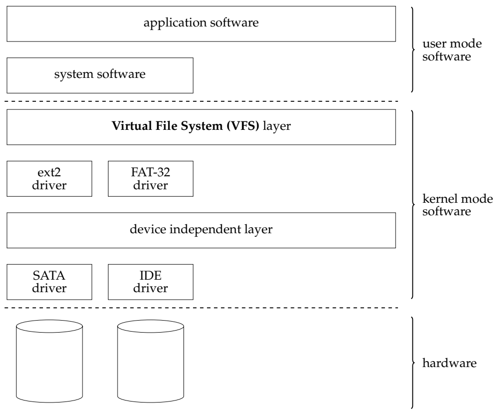
   plus
2. various optimisation and translation operations.

Notes:

- The VFS layer is an ideal place to locate additional caches. For example, it may make sense to cache FCBs st. subsequent accesses to the same entry are more efficient.

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

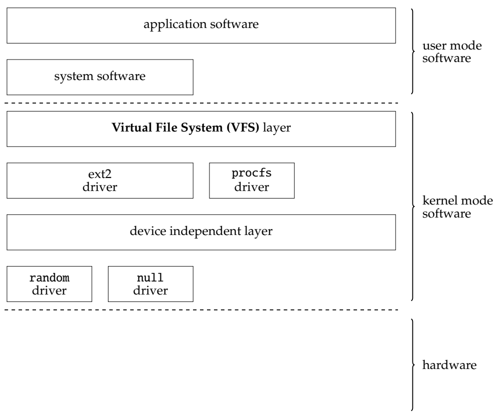
   plus
2. various optimisation and translation operations.

Notes:

- The VFS layer is an ideal place to locate additional caches. For example, it may make sense to cache FCBs st. subsequent accesses to the same entry are more efficient.

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

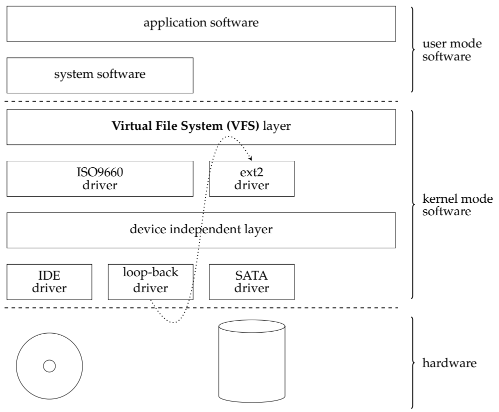
   plus
2. various optimisation and translation operations.

Notes:

- The VFS layer is an ideal place to locate additional caches. For example, it may make sense to cache FCBs st. subsequent accesses to the same entry are more efficient.

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

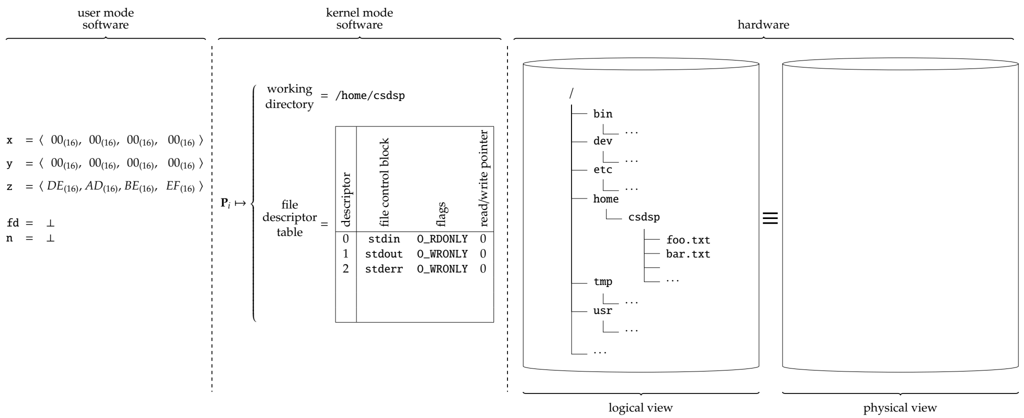
   plus
2. various optimisation and translation operations.

Notes:

- The VFS layer is an ideal place to locate additional caches. For example, it may make sense to cache FCBs st. subsequent accesses to the same entry are more efficient.

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

- Example:

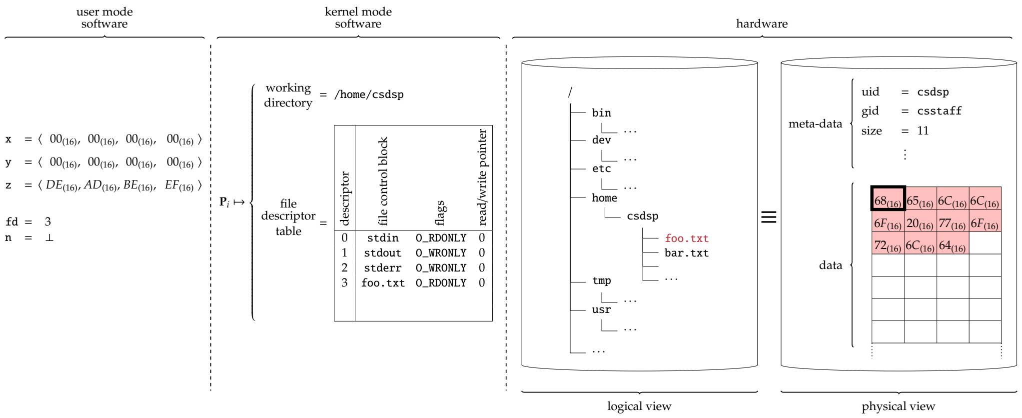


Notes:

- Example: if the user mode process executes

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

then the result is described by

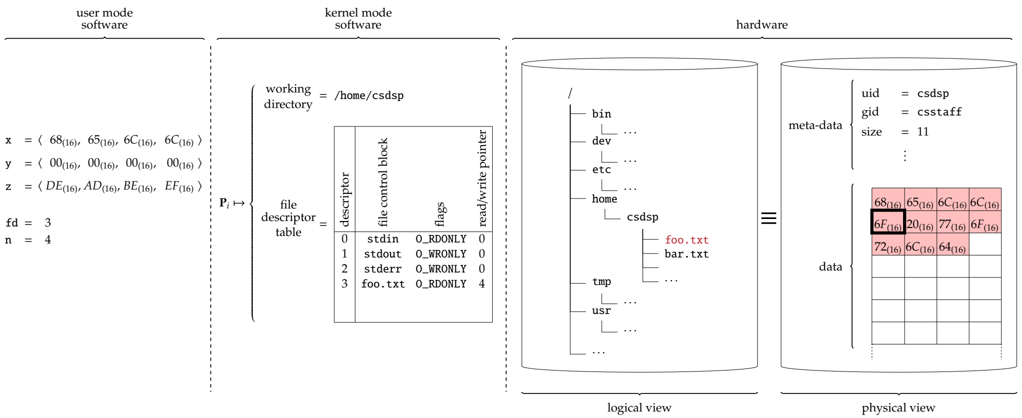


Notes:

- Example: if the user mode process executes

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

then the result is described by

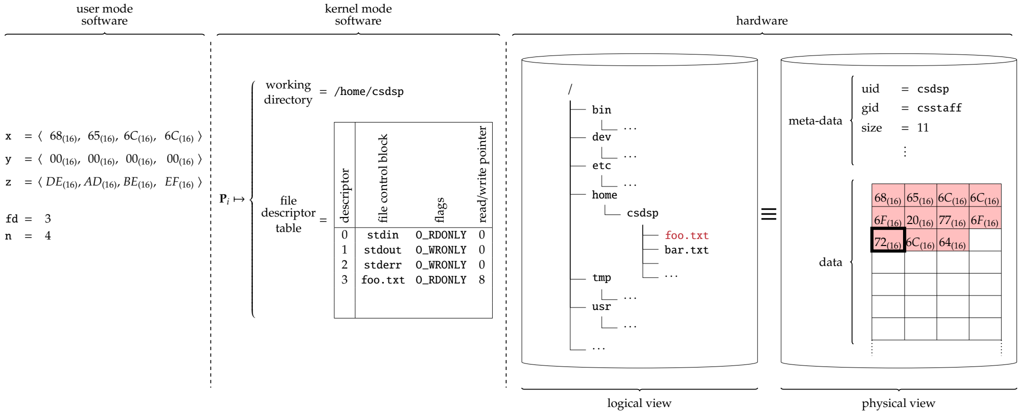


Notes:

- Example: if the user mode process executes

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

then the result is described by



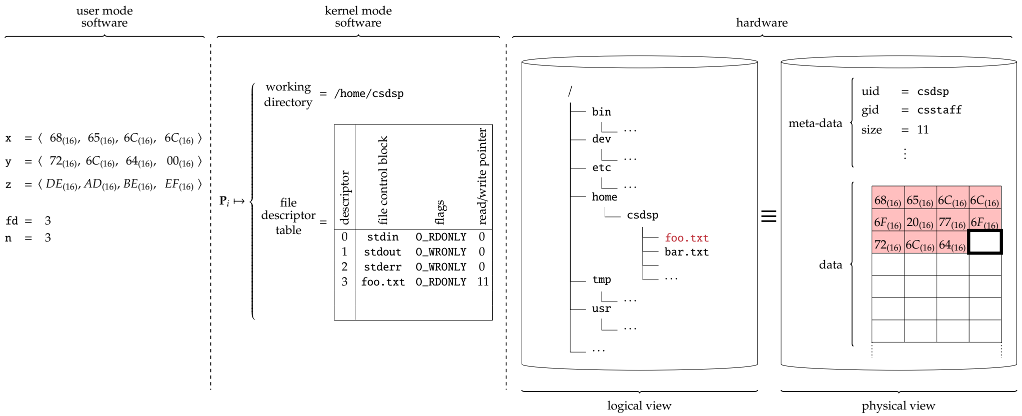
Notes:

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



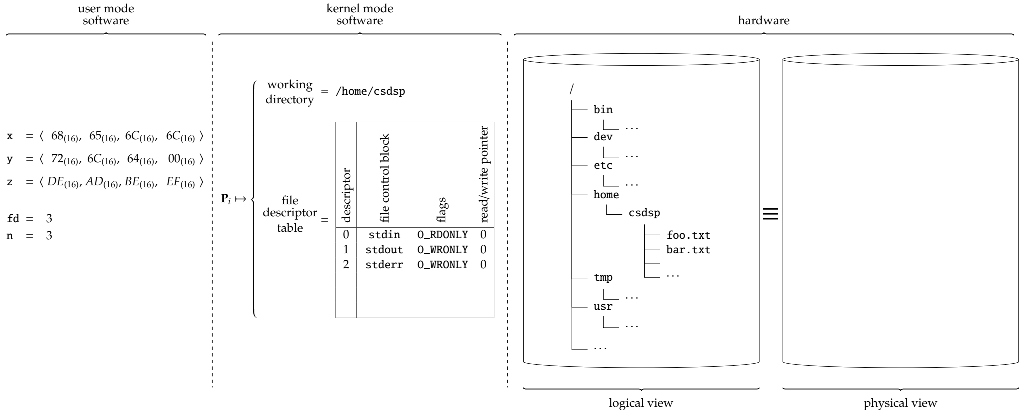
Notes:

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

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by



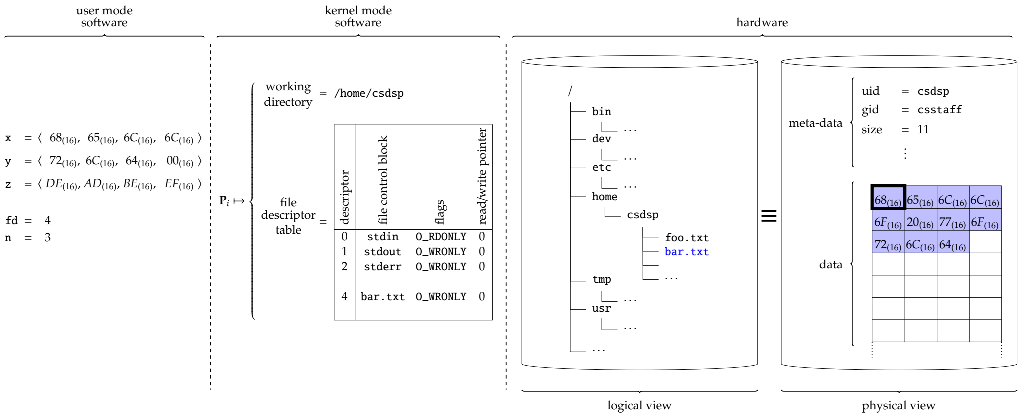
Notes:

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



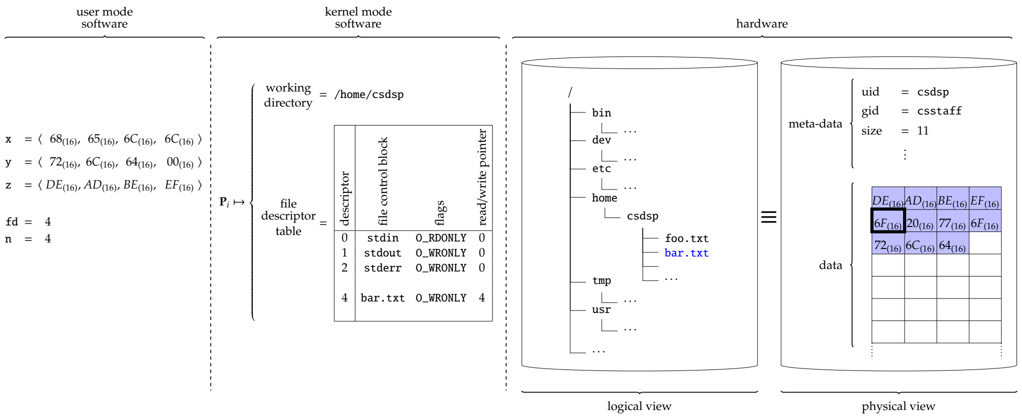
Notes:

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



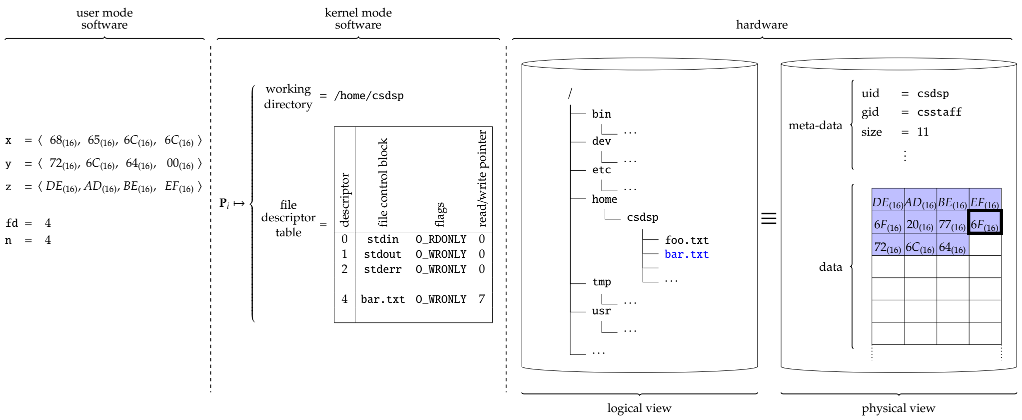
Notes:

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



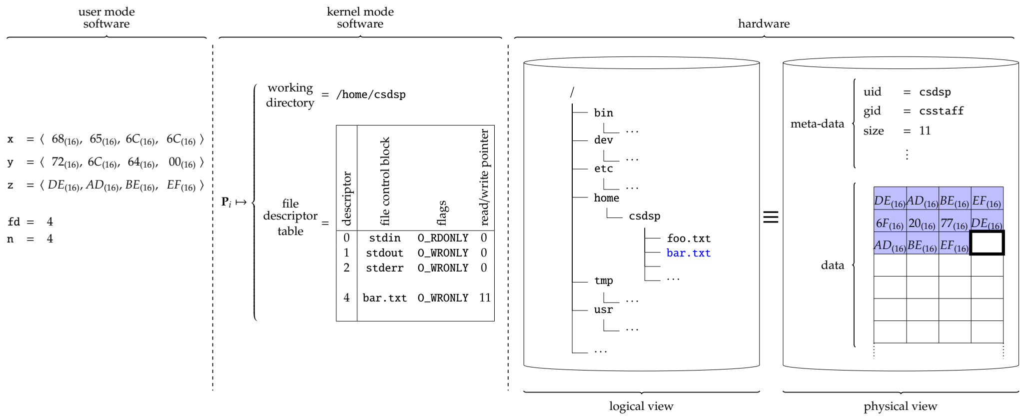
Notes:

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



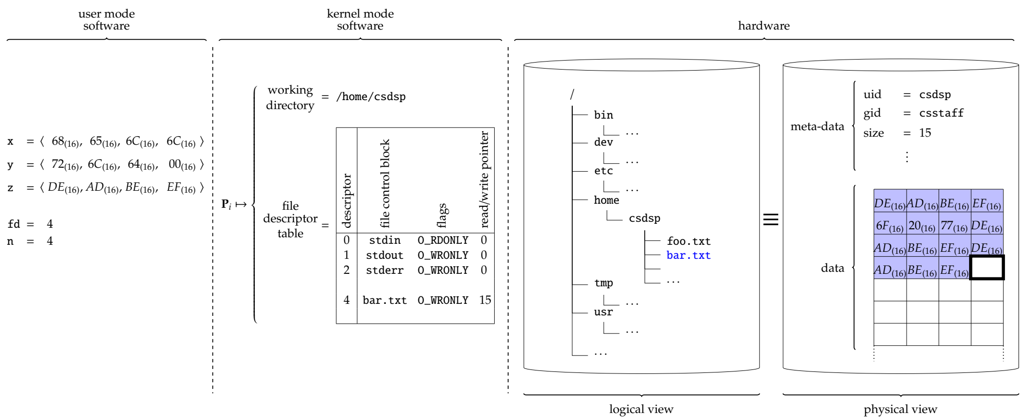
Notes:

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



Daniel Page (Daniel.Page@bristol.ac.uk)
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15
University of BRISTOL

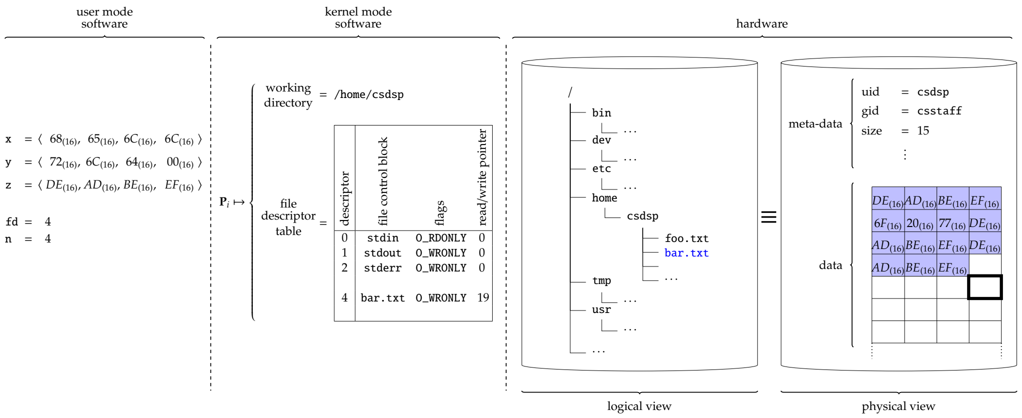
Notes:

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



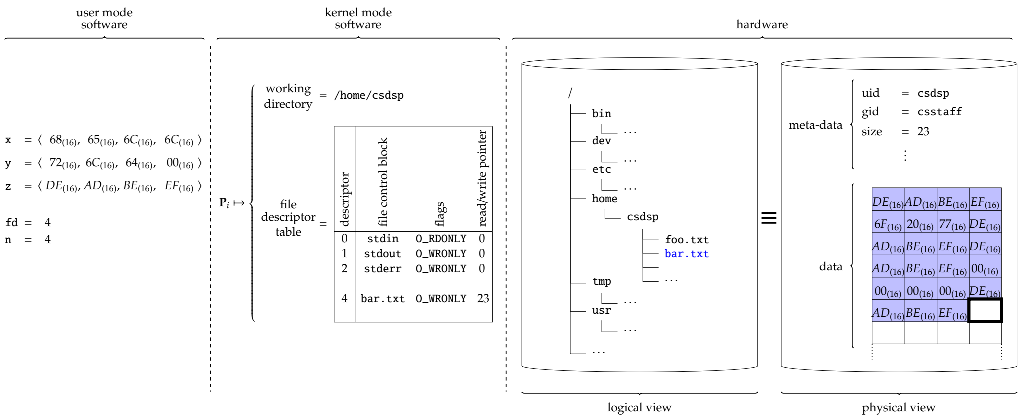
Notes:

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



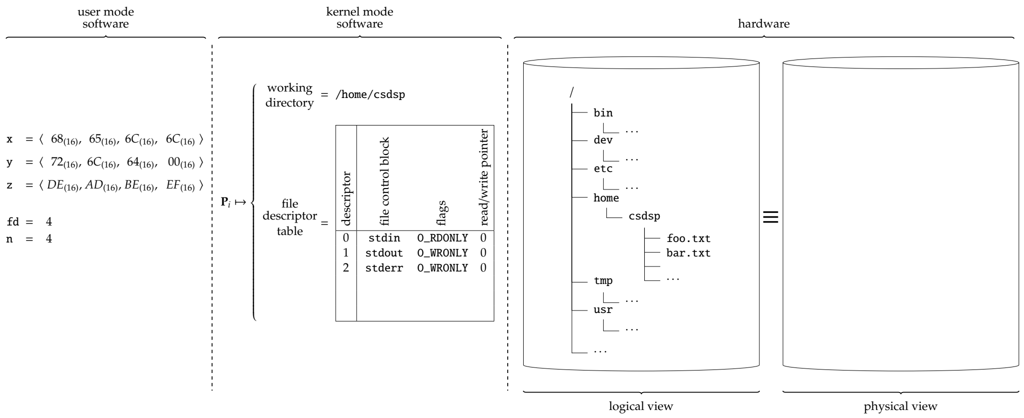
Notes:

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

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by



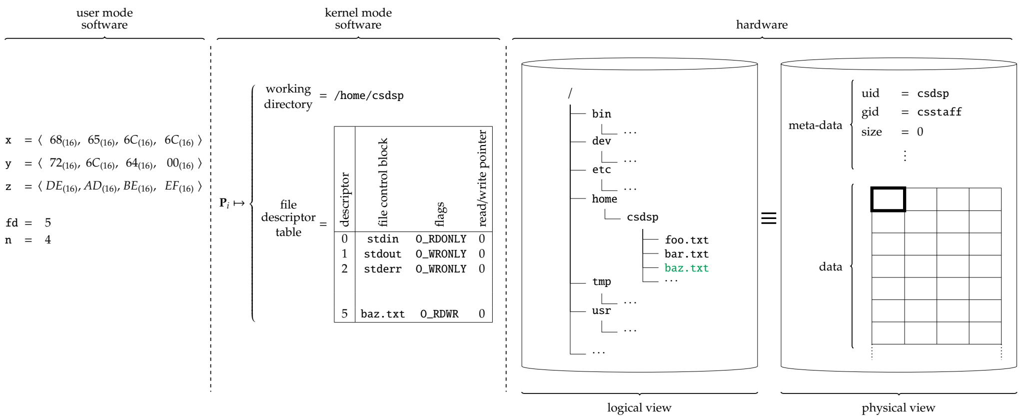
Notes:

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

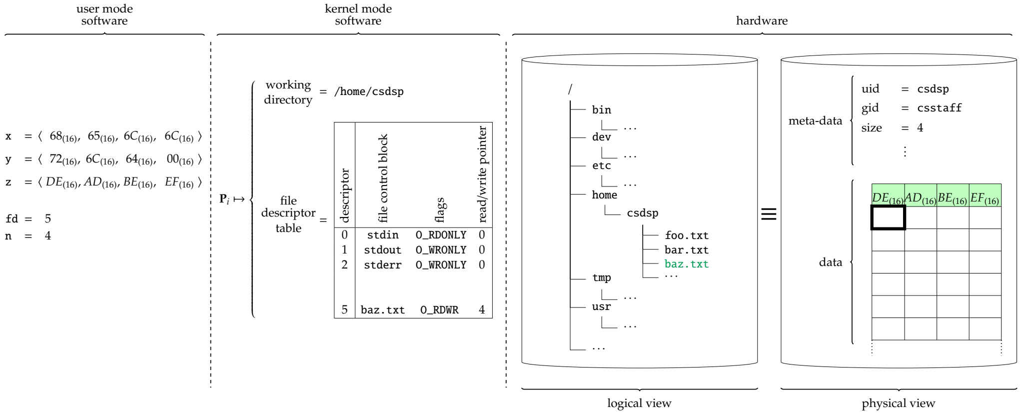


Notes:

- Example: if the user mode process executes

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

then the result is described by

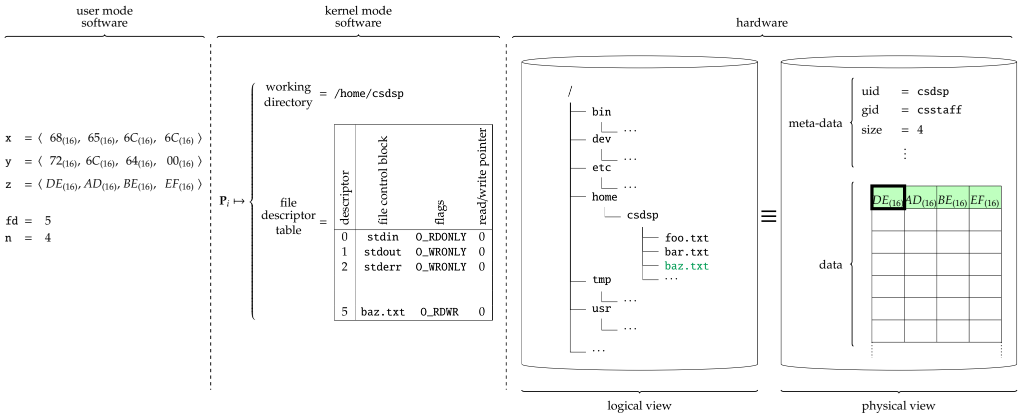


Notes:

- Example: if the user mode process executes

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

then the result is described by



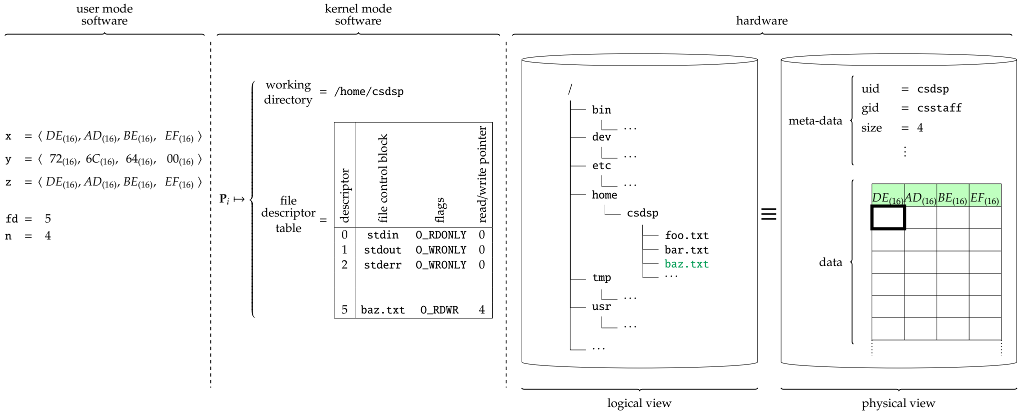
Notes:

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



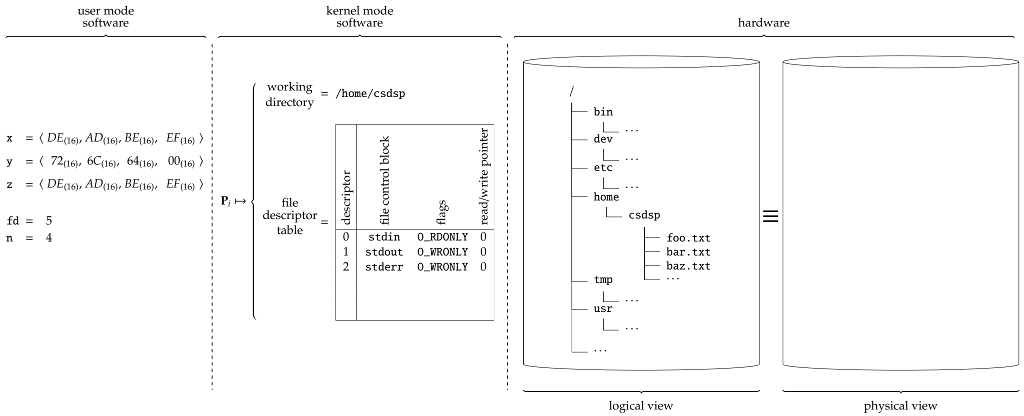
Notes:

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

- Example: if the user mode process executes

```
close( fd )
```

then the result is described by



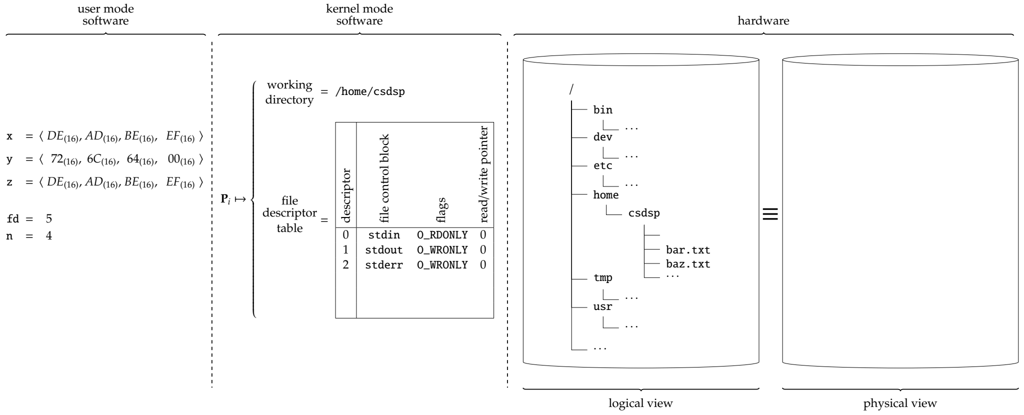
Notes:

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

- Example: if the user mode process executes

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

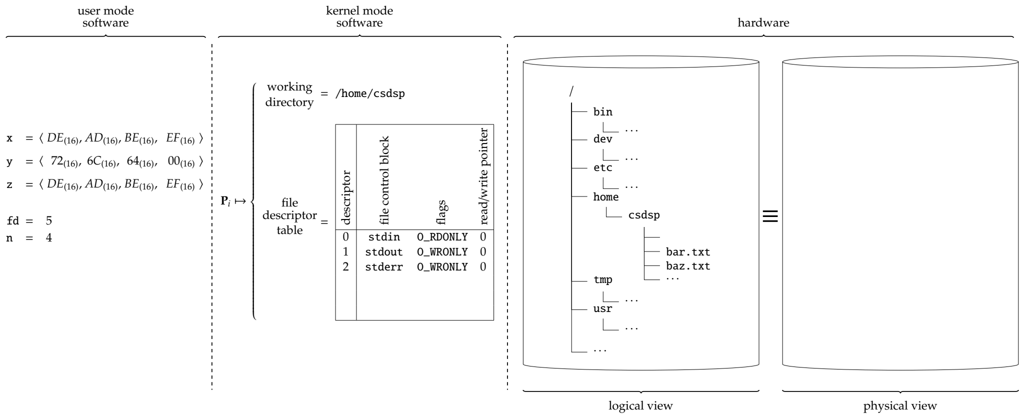
then the result is described by



Notes:

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

- Example:



Notes:

Continued in next lecture ...

Notes:

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

[2] Wikipedia: Path.
https://en.wikipedia.org/wiki/Path_(computing).

[3] Standard for information technology - portable operating system interface (POSIX).
Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008.
http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[4] M. Gorman.
Chapter 11: Swap management.
In *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.
http://www.kernel.org/doc/gorman/.

[5] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 10: File system.
In *Operating System Concepts*. Wiley, 9th edition, 2014.

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

Notes: