File systems

Johan Montelius

KTH

2021

The bigger picture

A file system is the user space implementation of *persistent storage*.

The bigger picture

A file system is the user space implementation of *persistent storage*.

- a *file* is persistent i.e. it survives the termination of a process
- a *file* can be access by several processes i.e. a shared resource
- a *file* can be located given a *path* name

What's a file

- a sequence of bytes

What's a file

- a sequence of bytes
- attributes, associated meta-data

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions
  - author

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions
  - author
  - created, last written

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions
  - author
  - created, last written
  - icons, fonts, presentation....

What's a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions
  - author
  - created, last written
  - icons, fonts, presentation....

User space operations

We need functionality to:

User space operations

We need functionality to:

- create and delete a file

User space operations

We need functionality to:

- create and delete a file
- find a file

User space operations

We need functionality to:

- create and delete a file
- find a file
- read and write the content of the file

User space operations

We need functionality to:

- create and delete a file
- find a file
- read and write the content of the file
- control authorization - who is allowed to do what

User space operations

We need functionality to:

- create and delete a file
- find a file
- read and write the content of the file
- control authorization - who is allowed to do what

Implementation

directory

map from name to identifier

Implementation

directory	map from name to identifier
file module	locate file

Implementation

directory	map from name to identifier
file module	locate file
access control	interacts with authentication system

Implementation

directory	map from name to identifier
file module	locate file
access control	interacts with authentication system
file operations	read and write operations

Implementation

directory	map from name to identifier
file module	locate file
access control	interacts with authentication system
file operations	read and write operations
block operations	which blocks on which devices

Implementation

directory	map from name to identifier
file module	locate file
access control	interacts with authentication system
file operations	read and write operations
block operations	which blocks on which devices
device operations	operations on physical drive

Implementation

directory	map from name to identifier
file module	locate file
access control	interacts with authentication system
file operations	read and write operations
block operations	which blocks on which devices
device operations	operations on physical drive

Name service

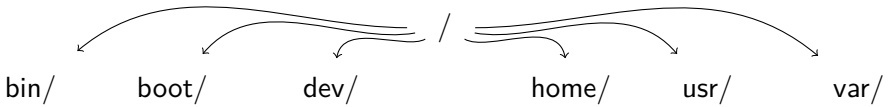
File service

/

Name service

File service

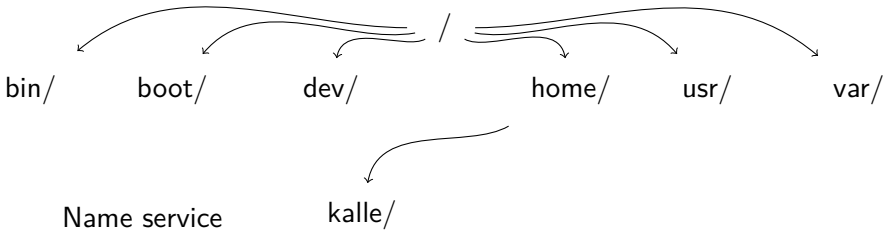
the name service



Name service

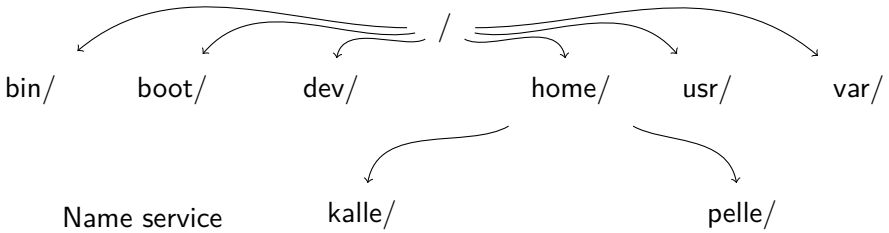
File service

the name service



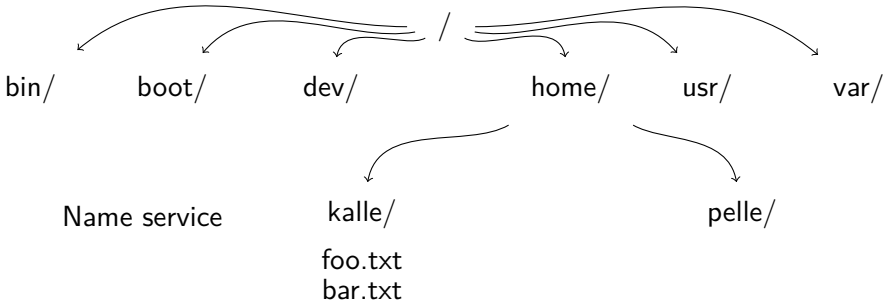
File service

the name service



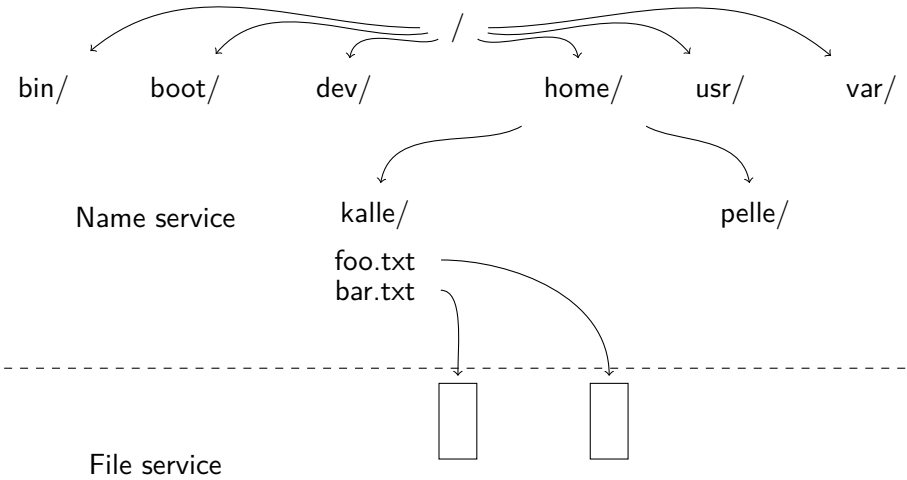
File service

the name service



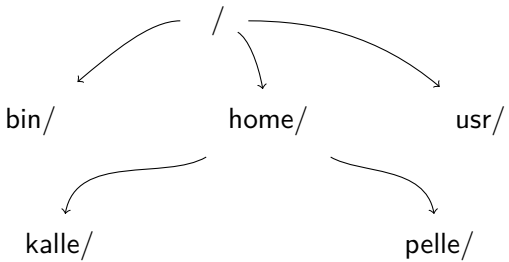
File service

the name service



Looking only at hard links:

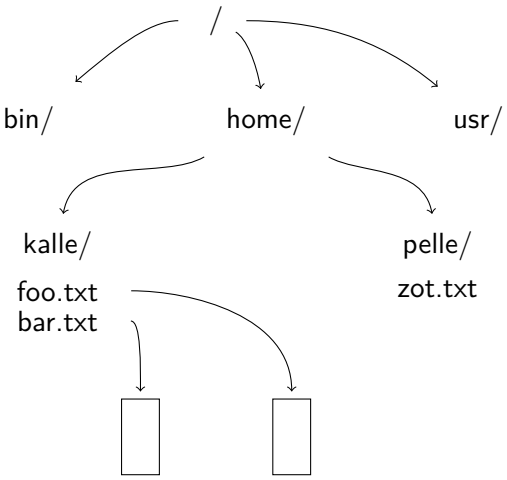
- The directory graph is a tree of directories.



links and the directory tree

Looking only at hard links:

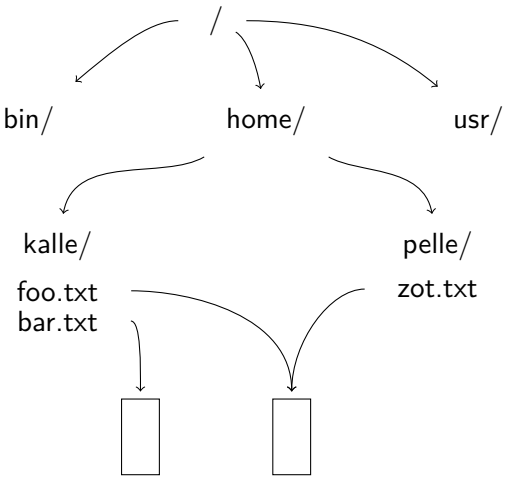
- The directory graph is a tree of directories.
- Directories contain sub directories and files.



links and the directory tree

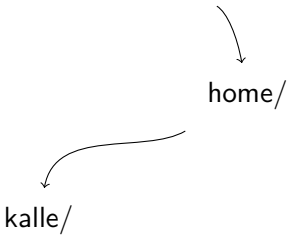
Looking only at hard links:

- The directory graph is a tree of directories.
- Directories contain sub directories and files.
- A file can be linked to from many directories.



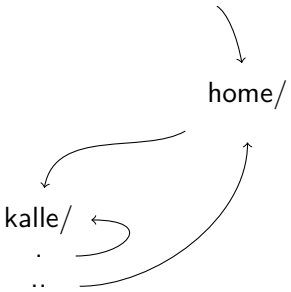
Looking only at hard links:

- Special directory links to:
  - . self



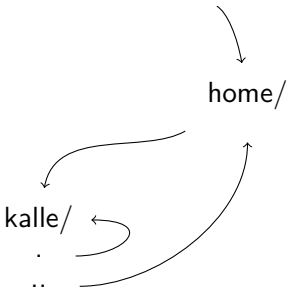
Looking only at hard links:

- Special directory links to:
  - . self
  - .. parent



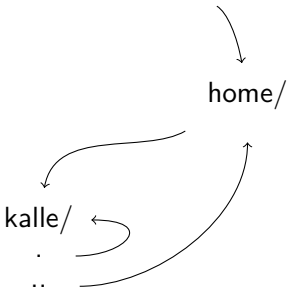
Looking only at hard links:

- Special directory links to:
  - . self
  - .. parent
- There is no explicit way of creating hard links to directories.



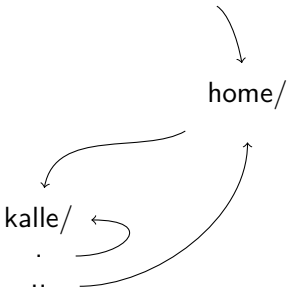
Looking only at hard links:

- Special directory links to:
  - . self
  - .. parent
- There is no explicit way of creating hard links to directories.
- Hard links to directories are created when directories are created or moved.



Looking only at hard links:

- Special directory links to:
  - . self
  - .. parent
- There is no explicit way of creating hard links to directories.
- Hard links to directories are created when directories are created or moved.
- Why?



symbolic links a.k.a soft links

Symbolic links are in-directions in the directory.

symbolic links a.k.a soft links

Symbolic links are in-directions in the directory.

symbolic links a.k.a soft links

Symbolic links are in-directions in the directory.

```
$ ls -l ./
total 16
drwxrwxr-x 2 johanmon johanmon 4096 nov 14 16:43 bar
drwxrwxr-x 2 johanmon johanmon 4096 nov 14 16:41 foo
-rw-rw-r-- 1 johanmon johanmon    8 nov 14 16:11 zat.txt
-rw-rw-r-- 1 johanmon johanmon    8 nov 14 16:11 zot.txt
```

symbolic links a.k.a soft links

Symbolic links are in-directions in the directory.

```
$ ls -l ./
total 16
drwxrwxr-x 2 johanmon johanmon 4096 nov 14 16:43 bar
drwxrwxr-x 2 johanmon johanmon 4096 nov 14 16:41 foo
-rw-rw-r-- 1 johanmon johanmon    8 nov 14 16:11 zat.txt
-rw-rw-r-- 1 johanmon johanmon    8 nov 14 16:11 zot.txt

$ ls -l foo
total 0
lrwxrwxrwx 1 johanmon johanmon 6 nov 14 16:41 gurka -> ../bar
```

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link
- `int symlink(const char *target, const char *linkpath)`
  - a soft link

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link
- `int symlink(const char *target, const char *linkpath)`
  - a soft link

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link
- `int symlink(const char *target, const char *linkpath)`
  - a soft link
- `int stat(const char *pathname, struct stat *buf)`
  - reads the meta-data of the file

File name operations

- `int creat(const char *pathname, mode_t mode)`
  - hardly ever used
- `int unlink(const char *pathname)`
  - when last link is removed the file is deleted
- `int link(const char *oldpath, const char *newpath)`
  - a hard link
- `int symlink(const char *target, const char *linkpath)`
  - a soft link
- `int stat(const char *pathname, struct stat *buf)`
  - reads the meta-data of the file

The process state

The kernel keeps a record of all *open files* of a process.

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`
- `int close(int fd)`
  - closes a file given a file descriptor

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`
- `int close(int fd)`
  - closes a file given a file descriptor

The process state

The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`
- `int close(int fd)`
  - closes a file given a file descriptor

By default a process has three open files: 0 *standard input*, 1 *standard output* and 2 *standard error*.

The process state

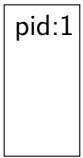
The kernel keeps a record of all *open files* of a process.

- `int open(const char *pathname, int flags)`
  - returns a *file descriptor*
- `int open(const char *pathname, int flags, mode_t mode)`
  - will create the file if it does not exist i.e. same as `creat()` and `open()`
- `int close(int fd)`
  - closes a file given a file descriptor

By default a process has three open files: 0 *standard input*, 1 *standard output* and 2 *standard error*.

You can not choose descriptor, the operating system will choose the lowest available.

The file tables



The file tables

processes

The file tables

processes

The file tables

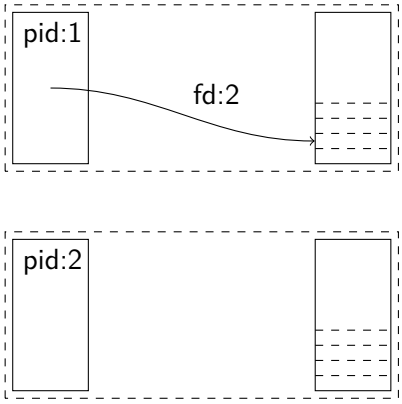
processes

file descriptor tables

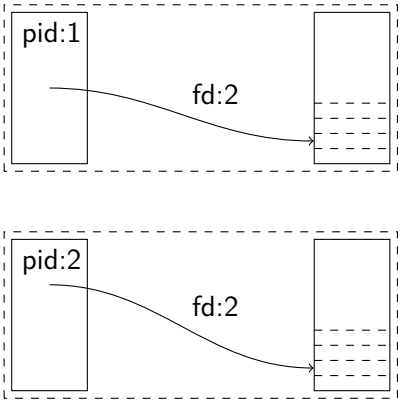
The file tables

The file tables

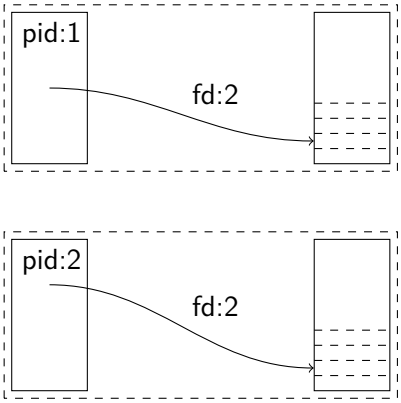
The file tables



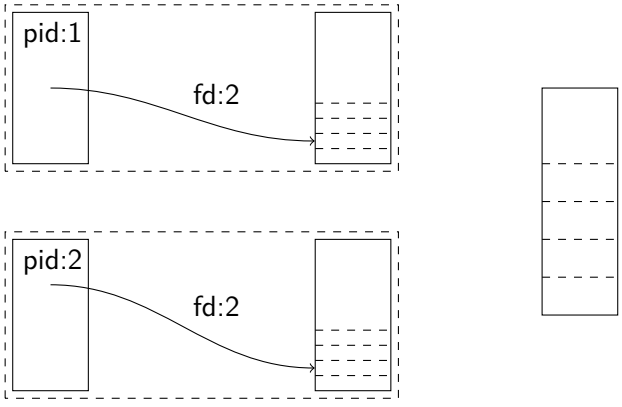
The file tables



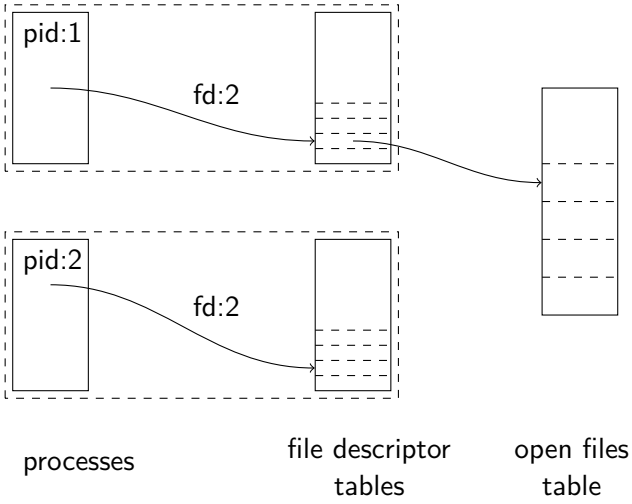
The file tables



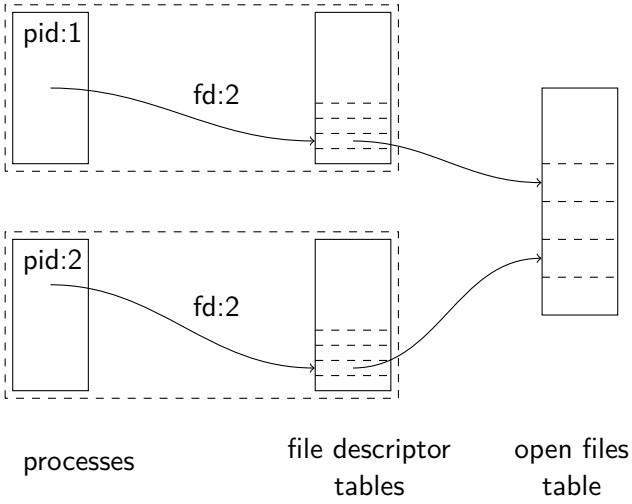
The file tables



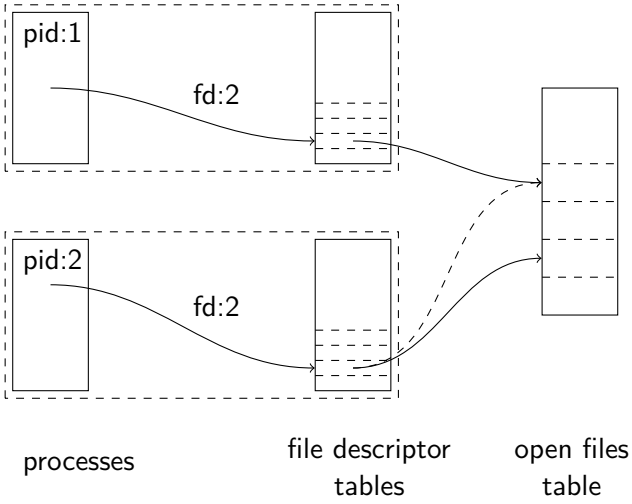
The file tables



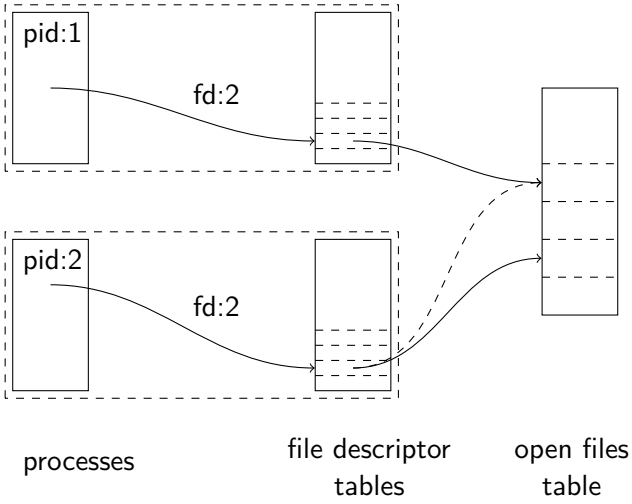
The file tables



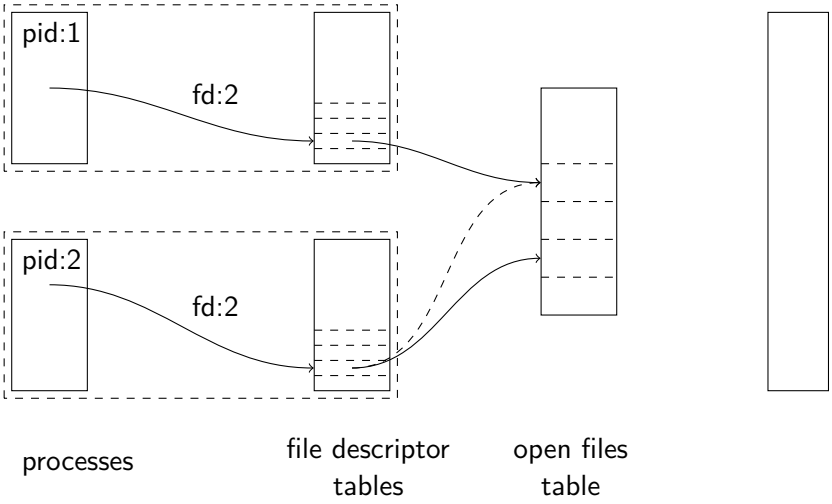
The file tables



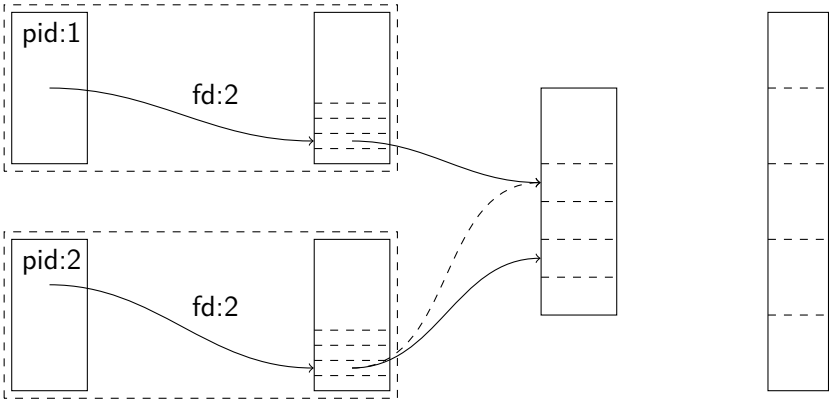
The file tables



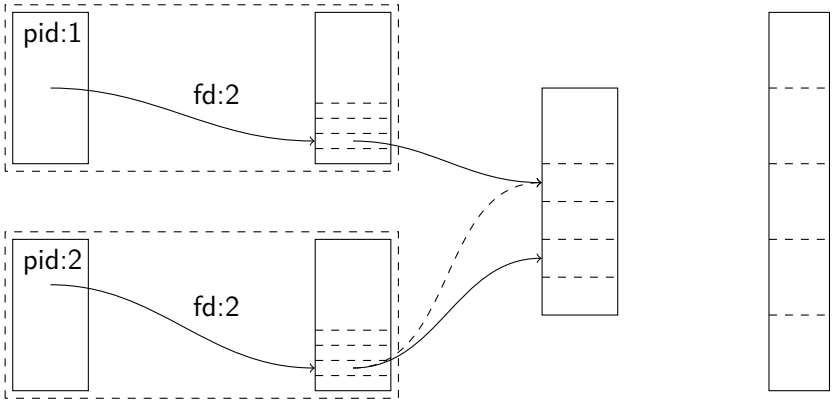
The file tables



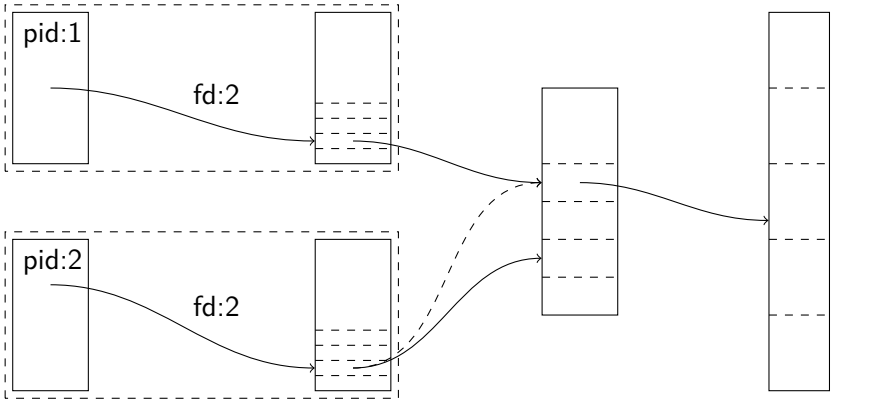
The file tables



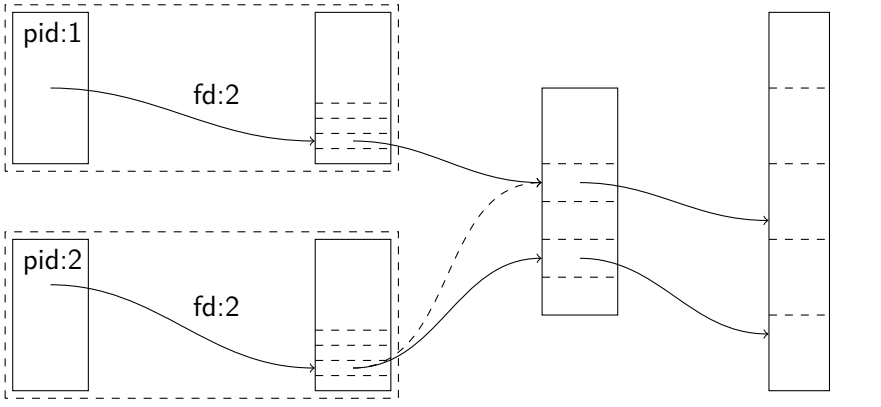
The file tables



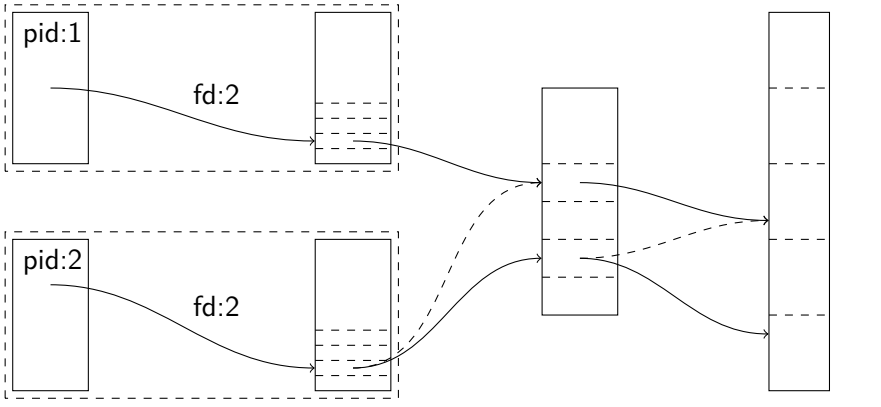
The file tables



The file tables



The file tables



The file tables

- descriptor table: one table per process, copied when process is forked

The file tables

- descriptor table: one table per process, copied when process is forked
- open files table: table is global, one entry per open operation

The file tables

- descriptor table: one table per process, copied when process is forked
- open files table: table is global, one entry per open operation
- inode table: one table per file system, one entry per *file object*

The file tables

- descriptor table: one table per process, copied when process is forked
- open files table: table is global, one entry per open operation
- inode table: one table per file system, one entry per *file object*

The file tables

- descriptor table: one table per process, copied when process is forked
- open files table: table is global, one entry per open operation
- inode table: one table per file system, one entry per *file object*

Note - a forked process can share file table entry with mother.

The file table entry

The file table entry holds:

- reference to inode
- reference counter, when it reaches 0 the file is closed and the entry is removed

The file table entry

The file table entry holds:

- reference to inode
- reference counter, when it reaches 0 the file is closed and the entry is removed
- the current *position* in the file

The file table entry

The file table entry holds:

- reference to inode
- reference counter, when it reaches 0 the file is closed and the entry is removed
- the current *position* in the file
- read and write access rights, determined when opened

The file table entry

The file table entry holds:

- reference to inode
- reference counter, when it reaches 0 the file is closed and the entry is removed
- the current *position* in the file
- read and write access rights, determined when opened

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read
- `int write(int fd, const void *buf, size_t count)`
  - returns the number of bytes actually written

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read
- `int write(int fd, const void *buf, size_t count)`
  - returns the number of bytes actually written

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read
- `int write(int fd, const void *buf, size_t count)`
  - returns the number of bytes actually written
- `lseek(int fd, off_t offset, int whence)`
  - sets the current position in the file

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read
- `int write(int fd, const void *buf, size_t count)`
  - returns the number of bytes actually written
- `lseek(int fd, off_t offset, int whence)`
  - sets the current position in the file

File operations

- `int read(int fd, void *buf, size_t count)`
  - returns the number of bytes actually read
- `int write(int fd, const void *buf, size_t count)`
  - returns the number of bytes actually written
- `lseek(int fd, off_t offset, int whence)`
  - sets the current position in the file

lseek() will only modify the file tabels entry i.e. it will not read anything from disk.

The inode

- mode: access rights
- number of links: when zero the file is deleted
- user id: the owner of the file
- group id: the associated group
- size: file size in bytes
- blocks: how many data blocks have been allocated
- identifiers: block identifiers (more on this later)
- generation: incremented when inode is reused
- access time: last time read
- modify time: last time modified
- change time: the time the inode was changed

The file system

The *file system* determines how files and directory structures are organized on a disk.

The file system

The *file system* determines how files and directory structures are organized on a disk.

An operating system can *mount* different file systems, all accessible from the same directory structure.

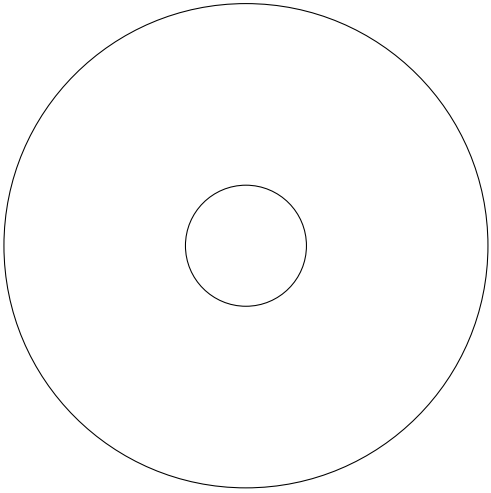
The file system

The *file system* determines how files and directory structures are organized on a disk.

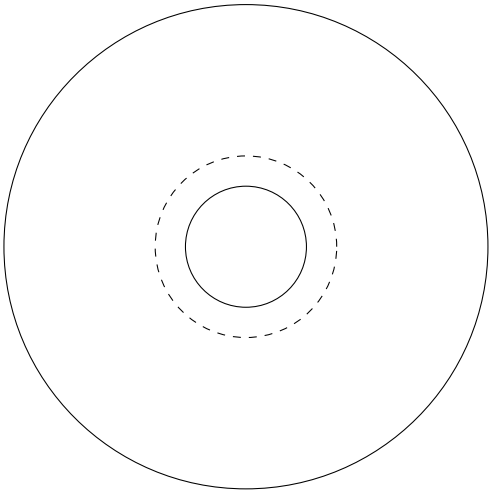
An operating system can *mount* different file systems, all accessible from the same directory structure.

Do `mount`, to see which file systems you have mounted.

Anatomy of a HDD

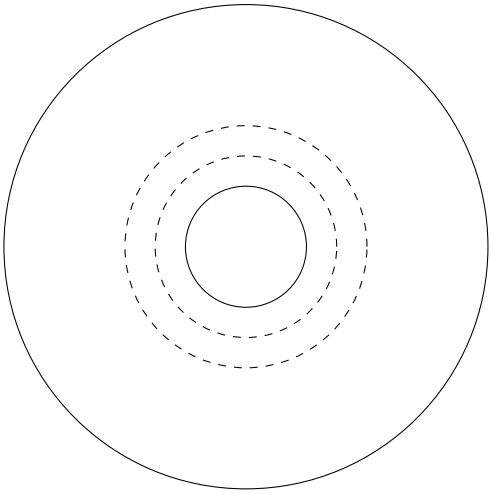


Anatomy of a HDD



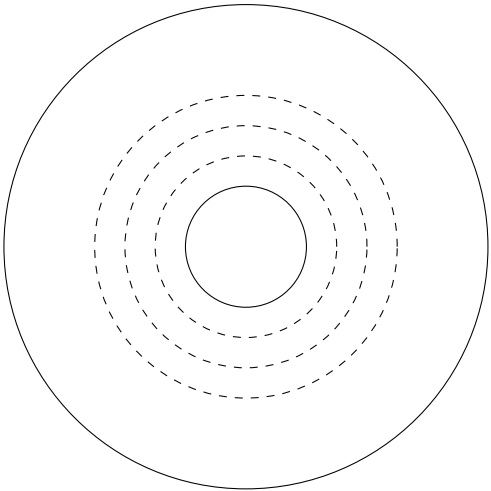
How do we store a file system in the sectors of a hard disk drive.

Anatomy of a HDD



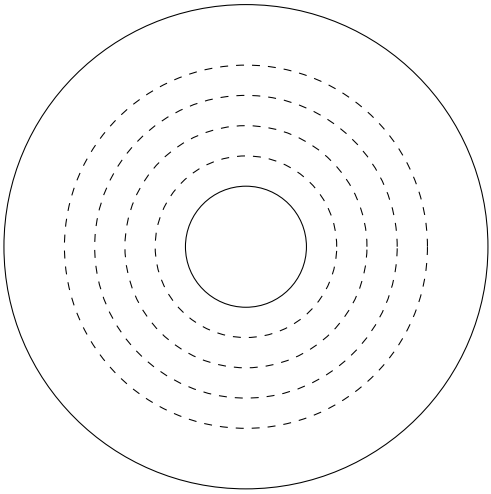
How do we store a file system in the sectors of a hard disk drive.

Anatomy of a HDD



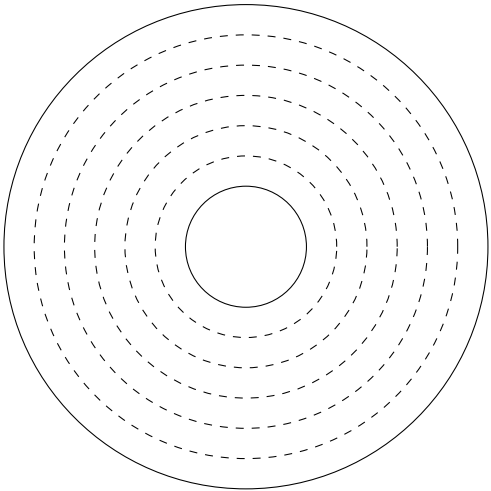
How do we store a file system in the sectors of a hard disk drive.

Anatomy of a HDD



How do we store a file system in the sectors of a hard disk drive.

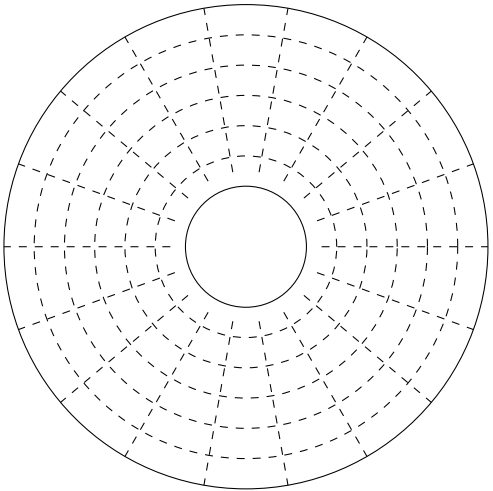
Anatomy of a HDD



- track/cylinder

How do we store a file system in the sectors of a hard disk drive.

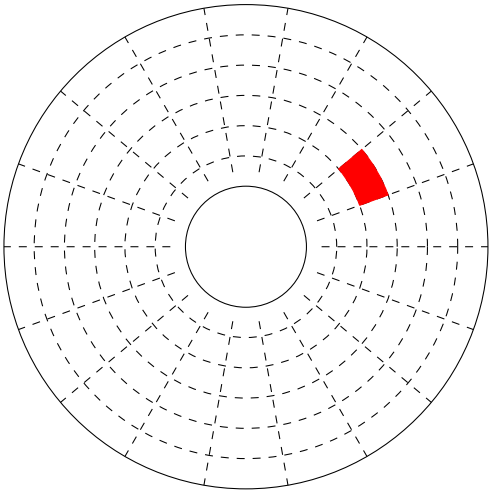
Anatomy of a HDD



- track/cylinder
- sectors per track varies

How do we store a file system in the sectors of a hard disk drive.

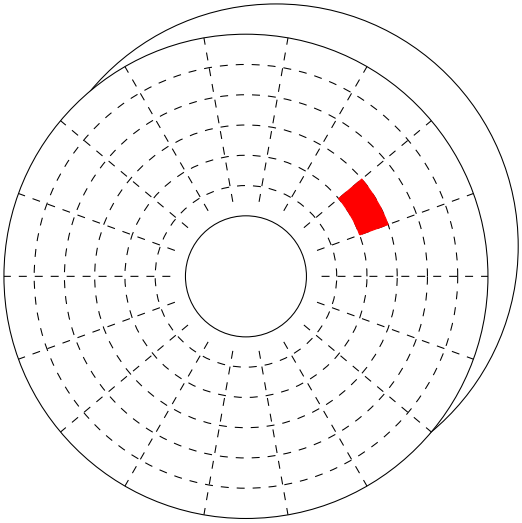
Anatomy of a HDD



- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes

How do we store a file system in the sectors of a hard disk drive.

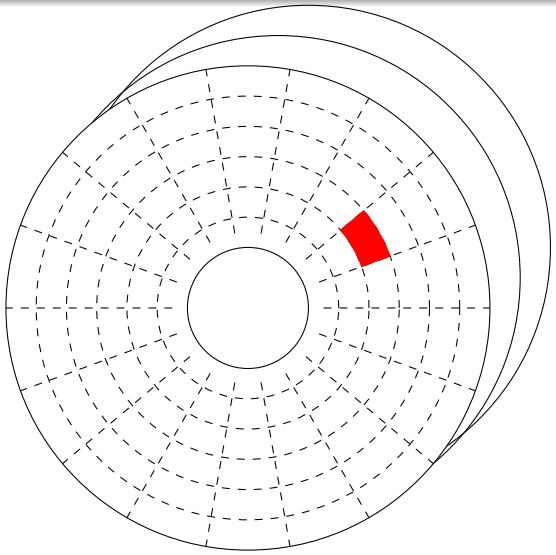
Anatomy of a HDD



- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes
- platters: 1 to 6
- heads: one side or two sides

How do we store a file system in the sectors of a hard disk drive.

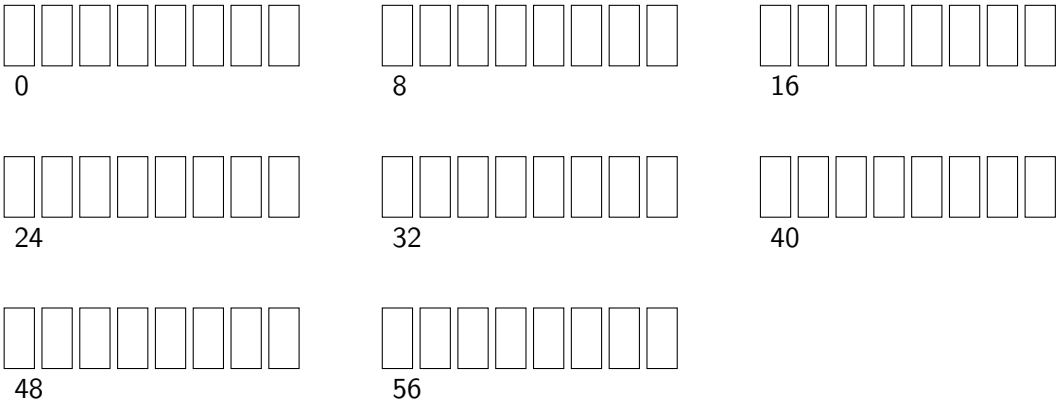
Anatomy of a HDD



- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes
- platters: 1 to 6
- heads: one side or two sides

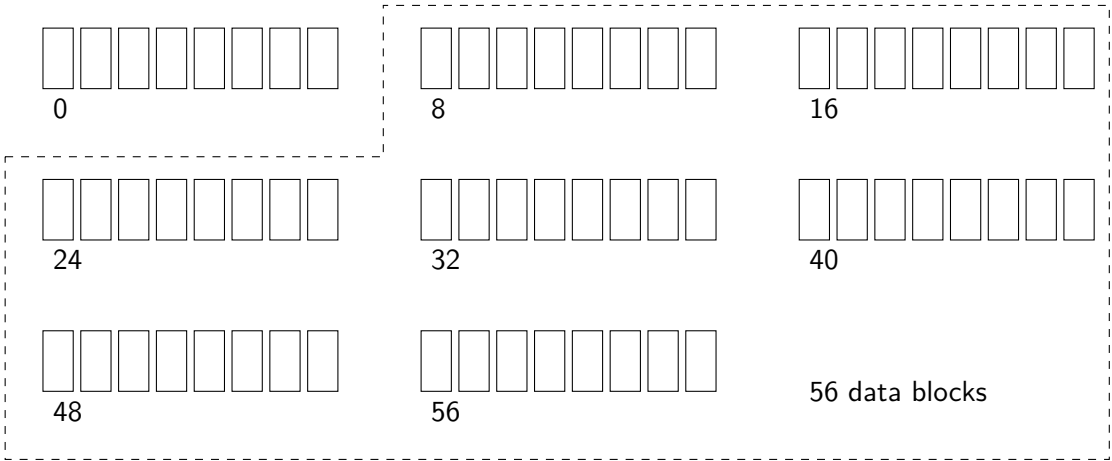
How do we store a file system in the sectors of a hard disk drive.

The Very Simple File System



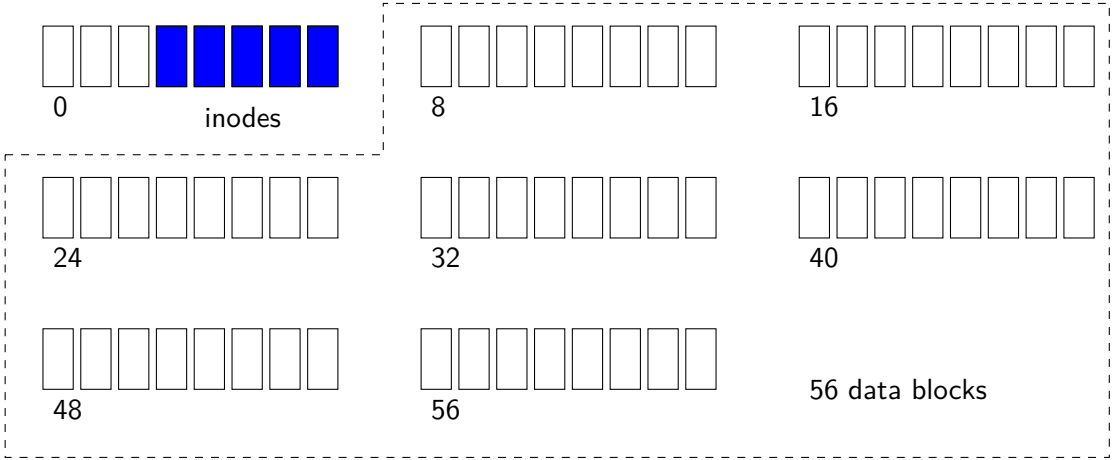
Each block is 4Kbytes, an inode 256 bytes.

The Very Simple File System



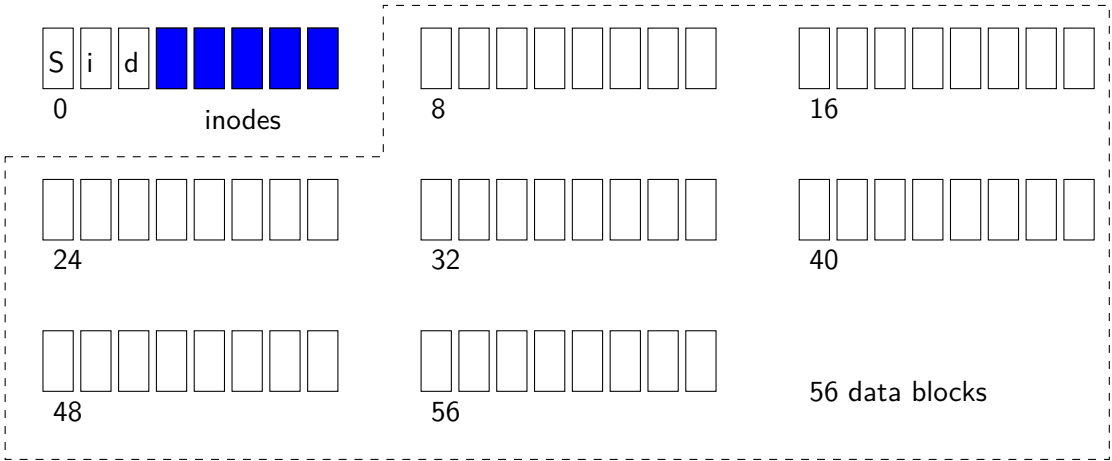
Each block is 4Kbytes, an inode 256 bytes.

The Very Simple File System



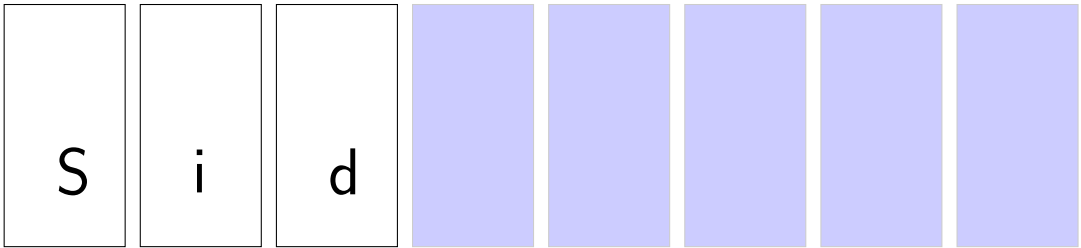
Each block is 4Kbytes, an inode 256 bytes.

The Very Simple File System

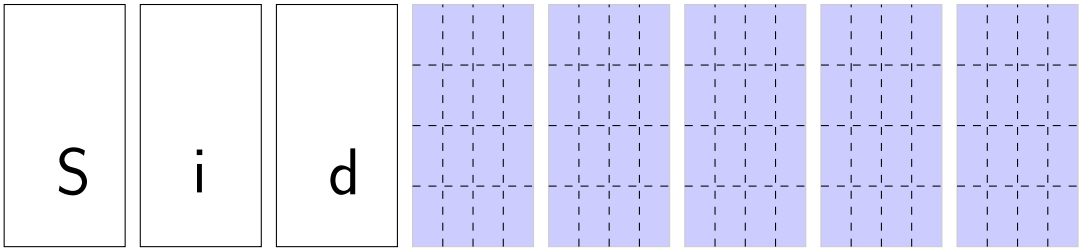


Each block is 4Kbytes, an inode 256 bytes.

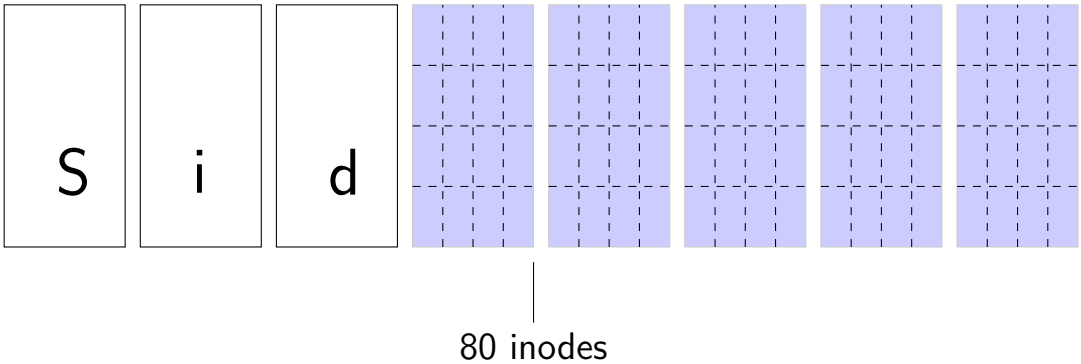
The super block, bitmaps and inodes



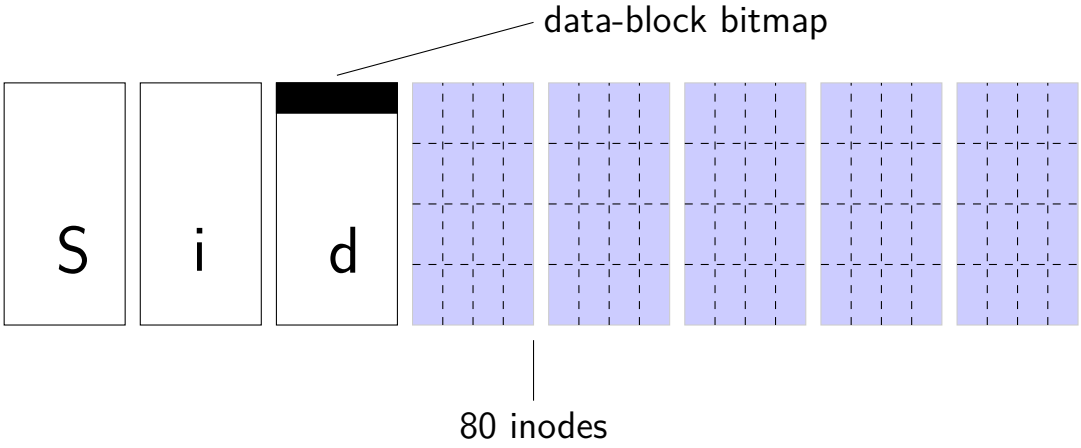
The super block, bitmaps and inodes



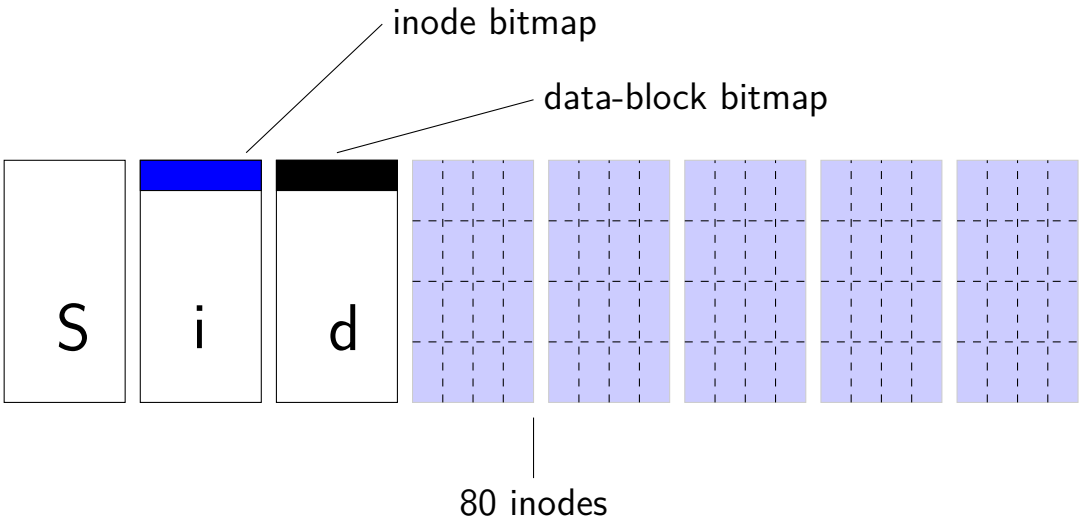
The super block, bitmaps and inodes



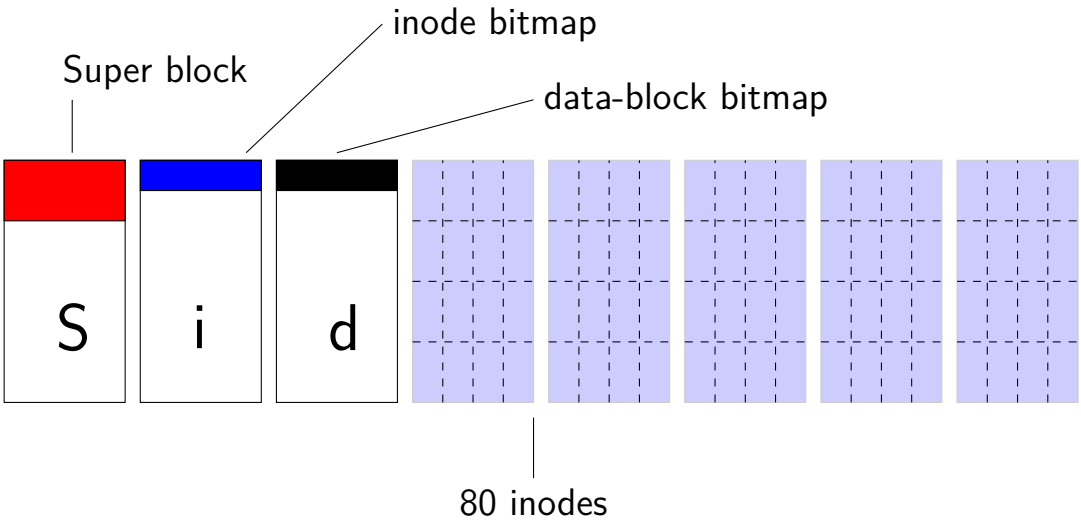
The super block, bitmaps and inodes



The super block, bitmaps and inodes



The super block, bitmaps and inodes



The super block, bitmaps and the inodes

The super block

- describes the file system
- how many inodes, where are they located
- how many data blocks, where are they located
- where are bitmaps

The super block, bitmaps and the inodes

The super block

- describes the file system
- how many inodes, where are they located
- how many data blocks, where are they located
- where are bitmaps

The bitmaps

- which inode blocks are available
- which data blocks are available

The super block, bitmaps and the inodes

The super block

- describes the file system
- how many inodes, where are they located
- how many data blocks, where are they located
- where are bitmaps

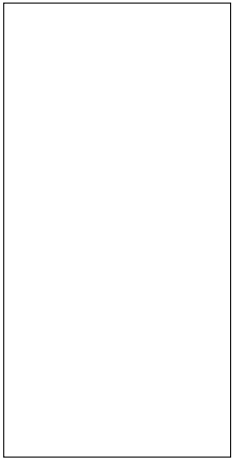
The bitmaps

- which inode blocks are available
- which data blocks are available

The inodes

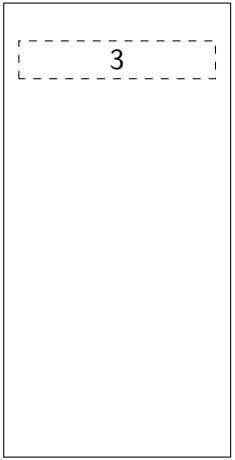
- all meta-data of the file
- which data blocks are used

the inode revisited



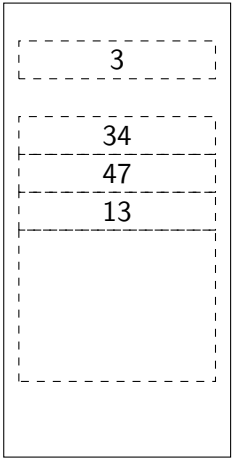
the inode revisited

- blocks: how many data blocks have been allocated



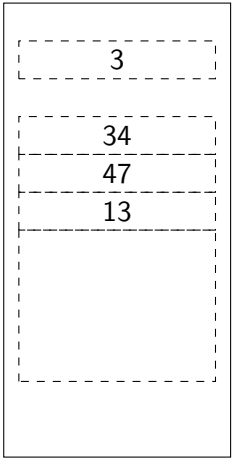
the inode revisited

- blocks: how many data blocks have been allocated
- identifiers: block identifiers



the inode revisited

- blocks: how many data blocks have been allocated
- identifiers: block identifiers



If the inode has room for 15 block identifiers ..... what is the maximum file size?

Multi-level index

- If the first fourteen entries are block identifiers,

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

- If the first thirteen entries are block identifiers,

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

- If the first thirteen entries are block identifiers,
- the fourteenth entry is a reference to a block holding block identifiers,

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

- If the first thirteen entries are block identifiers,
- the fourteenth entry is a reference to a block holding block identifiers,
- and the fifteenth entry is a reference to
  a block holding identifiers to blocks holding block identifiers.

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

- If the first thirteen entries are block identifiers,
- the fourteenth entry is a reference to a block holding block identifiers,
- and the fifteenth entry is a reference to
  a block holding identifiers to blocks holding block identifiers.
- What is the maximum size of a file?

Multi-level index

- If the first fourteen entries are block identifiers,
- and the fifteenth entry is a reference to a block holding block identifiers.
- What is the maximum size of a file?

- If the first thirteen entries are block identifiers,
- the fourteenth entry is a reference to a block holding block identifiers,
- and the fifteenth entry is a reference to
  a block holding identifiers to blocks holding block identifiers.
- What is the maximum size of a file?

Why not have a balanced tree?

where are the directories?

where are the directories?

A directory is stored as a file using an inode and (almost always) one data block.

where are the directories?

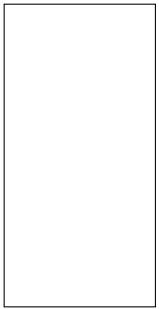
A directory is stored as a file using an inode and (almost always) one data block.

where are the directories?

A directory is stored as a file using an inode and (almost always) one data block.

The data block contains a mapping from names of files and directories to inode numbers.

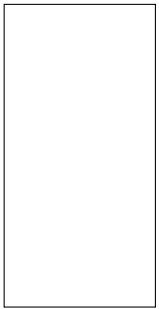
a directory



.

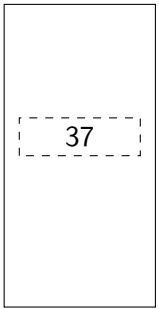
a directory

inode 2



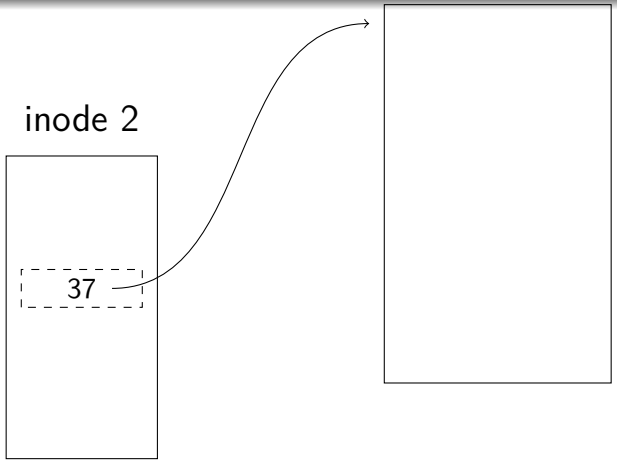
.

inode 2



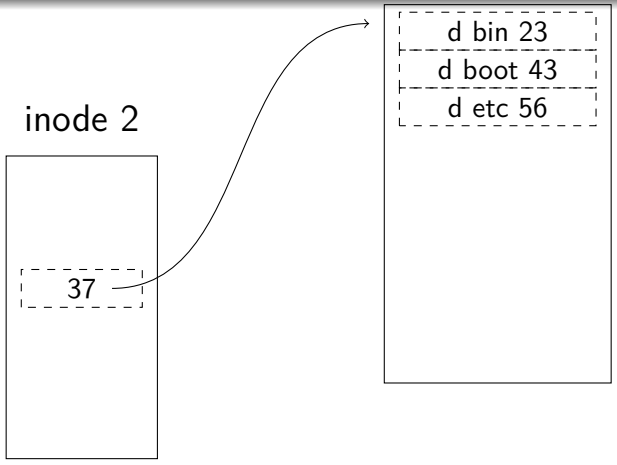
.

a directory



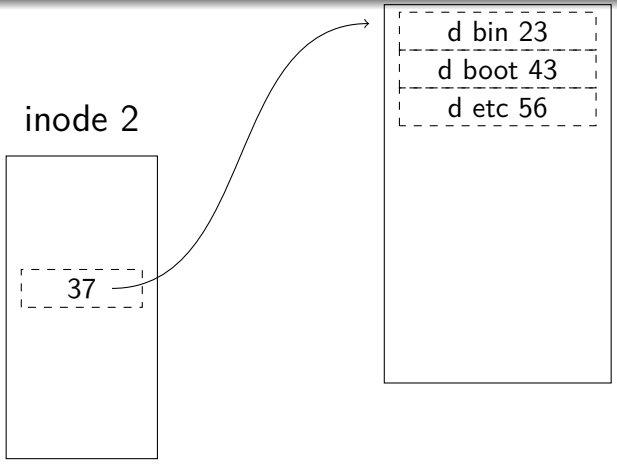
.

a directory



.

a directory



The inode of the root directory is specified by the file system (possibly in the super block).

try this

- `ls -ila ./`
  - list the current directory, show inode numbers

try this

- `ls -ila ./`
  - list the current directory, show inode numbers

try this

- `ls -ila ./`
  - list the current directory, show inode numbers
- `stat <file name>`
  - inspect an inode, try directories and regular files

try this

- `ls -ila ./`
  - list the current directory, show inode numbers
- `stat <file name>`
  - inspect an inode, try directories and regular files

try this

- `ls -ila ./`
  - list the current directory, show inode numbers
- `stat <file name>`
  - inspect an inode, try directories and regular files
- `istat <device> <inode>`
  - shows information about a particular inode

try this

- `ls -ila ./`
  - list the current directory, show inode numbers
- `stat <file name>`
  - inspect an inode, try directories and regular files
- `istat <device> <inode>`
  - shows information about a particular inode
- `sudo dd if=/dev/sda1 bs=4096 skip=<block> count=1`
  - inspect a data block

try this

- `ls -ila ./`
  - list the current directory, show inode numbers
- `stat <file name>`
  - inspect an inode, try directories and regular files
- `istat <device> <inode>`
  - shows information about a particular inode
- `sudo dd if=/dev/sda1 bs=4096 skip=<block> count=1`
  - inspect a data block

to use istat, install: sudo apt install sleuthkit

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`
- create a file table entry

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`
- create a file table entry
- create a file descriptor entry

Read from the file:

- read the inode of `bar.txt`

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`
- create a file table entry
- create a file descriptor entry

Read from the file:

- read the inode of `bar.txt`
- read the first 100 bytes from the first block

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`
- create a file table entry
- create a file descriptor entry

Read from the file:

- read the inode of `bar.txt`
- read the first 100 bytes from the first block
- update the inode of `bar.txt` - why?

reading a file

Assume we want to read the first 100 bytes of `/foo/bar.txt`.

Open the file:

- read inode of / (the root dir) - why?
- read the data block of / - why?
- read the inode of `foo`
- read the data block of `foo`
- read the inode of `bar.txt`
- create a file table entry
- create a file descriptor entry

Read from the file:

- read the inode of `bar.txt`
- read the first 100 bytes from the first block
- update the inode of `bar.txt` - why?

Think about this the next time you complain about your computer being slow.

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of `/`
- read the data block of `/`
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of `/`
- read the data block of `/`
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of `/`
- read the data block of `/`
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of `/`
- read the data block of `/`
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`
- read the data block bitmap - find a free block

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`
- read the data block bitmap - find a free block
- update the data block bitmap

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`
- read the data block bitmap - find a free block
- update the data block bitmap
- write to the new block

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`
- read the data block bitmap - find a free block
- update the data block bitmap
- write to the new block
- write to the inode of `bar.txt`

creating a file

Assume we want to create and write to a new file `/foo/bar.txt`.

Create the file:

- read inode of /
- read the data block of /
- read inode of `foo`
- read the inode bitmap - find a free inode
- update the inode bitmap
- write the data block of `foo`
- write the `bar.txt` inode
- write the inode of `foo`

Write to the file:

- read to the inode of `bar.txt`
- read the data block bitmap - find a free block
- update the data block bitmap
- write to the new block
- write to the inode of `bar.txt`

Caching and buffering

Caching: keep frequently used inodes and data blocks in memory.

Caching and buffering

Caching: keep frequently used inodes and data blocks in memory.

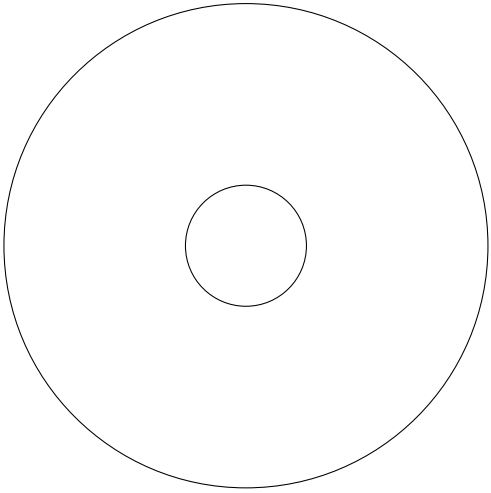
Buffering: perform updates in memory, batch operations to disk.

Caching and buffering

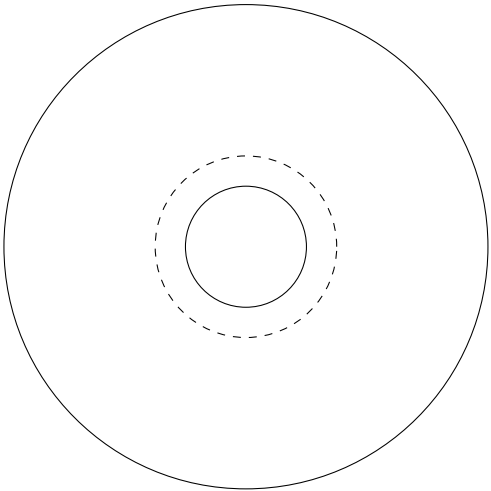
Caching: keep frequently used inodes and data blocks in memory.

Buffering: perform updates in memory, batch operations to disk.

Anatomy of a HDD

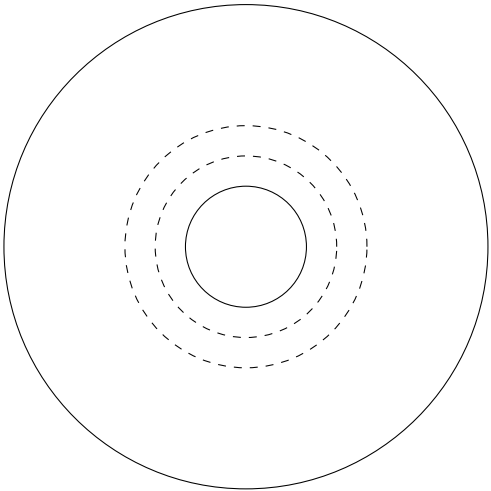


Anatomy of a HDD



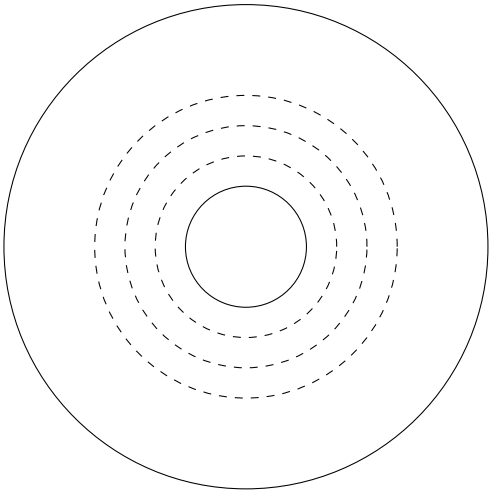
How do we store a file system in the sectors of a hard disk drive?

Anatomy of a HDD



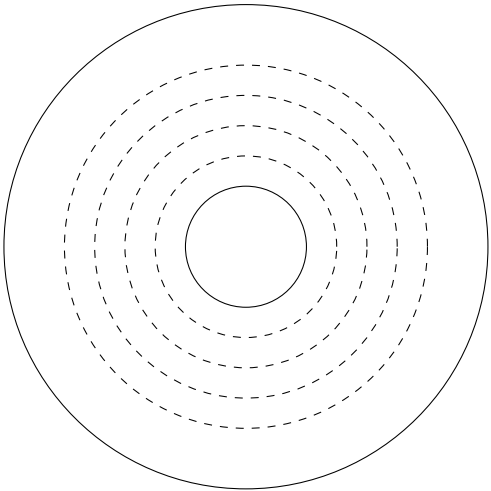
How do we store a file system in the sectors of a hard disk drive?

Anatomy of a HDD



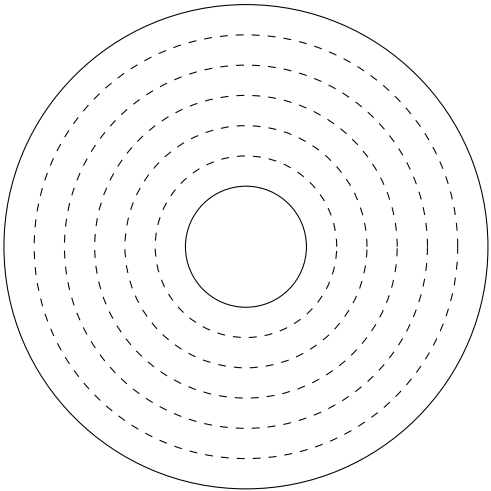
How do we store a file system in the sectors of a hard disk drive?

Anatomy of a HDD



How do we store a file system in the sectors of a hard disk drive?

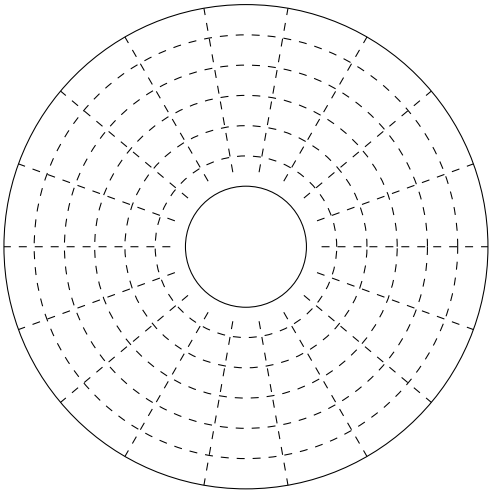
Anatomy of a HDD



- track/cylinder

How do we store a file system in the sectors of a hard disk drive?

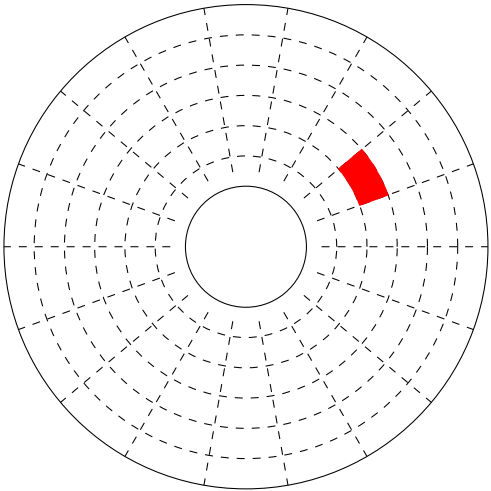
Anatomy of a HDD



- track/cylinder
- sectors per track varies

How do we store a file system in the sectors of a hard disk drive?

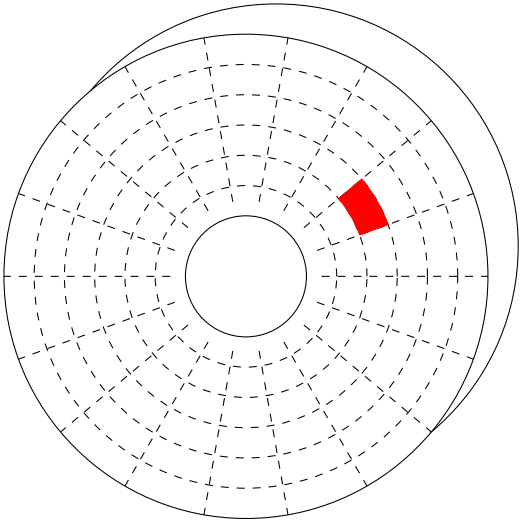
Anatomy of a HDD



- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes

How do we store a file system in the sectors of a hard disk drive?

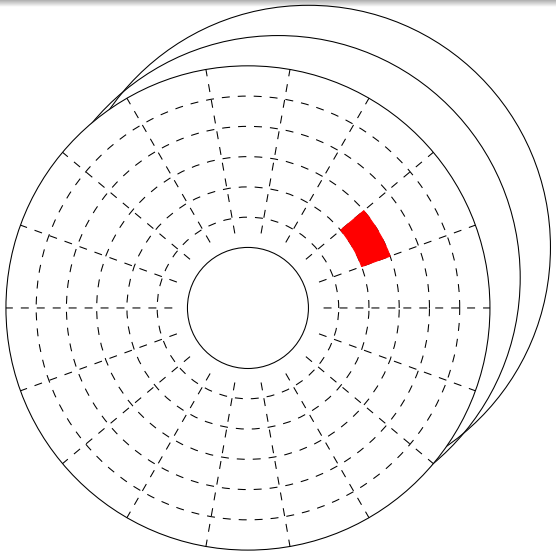
Anatomy of a HDD



- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes
- platters: 1 to 6
- heads: one side or two sides

How do we store a file system in the sectors of a hard disk drive?

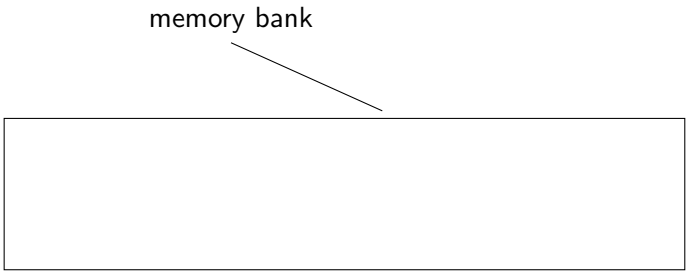
Anatomy of a HDD



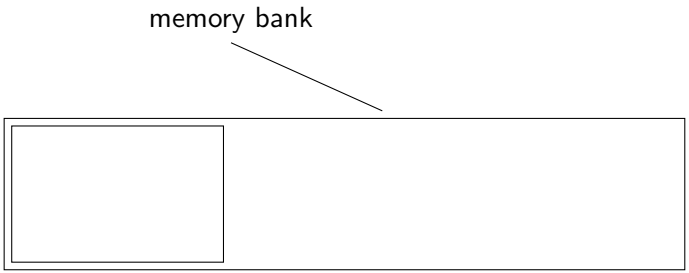
- track/cylinder
- sectors per track varies
- sector size: 4K or 512 bytes
- platters: 1 to 6
- heads: one side or two sides

How do we store a file system in the sectors of a hard disk drive?

Anatomy of a SSD

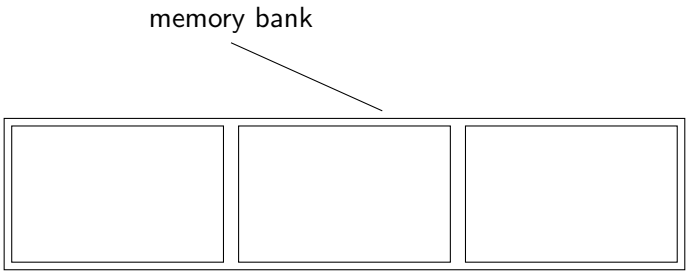


Anatomy of a SSD



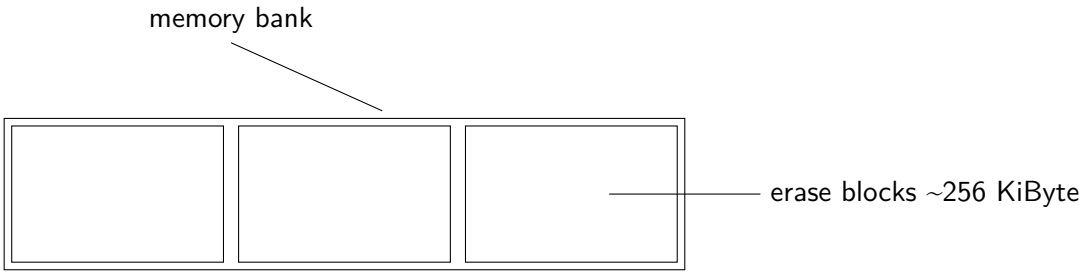
How do we store a file system in the sectors of a SSD?

Anatomy of a SSD



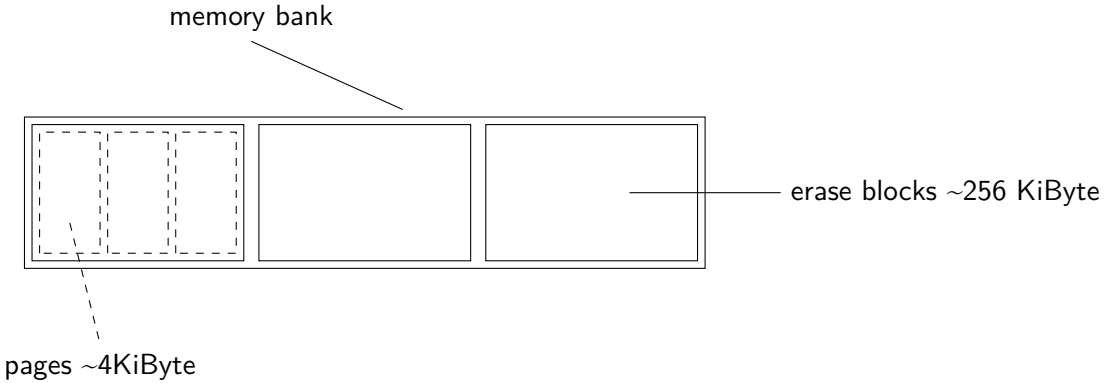
How do we store a file system in the sectors of a SSD?

Anatomy of a SSD



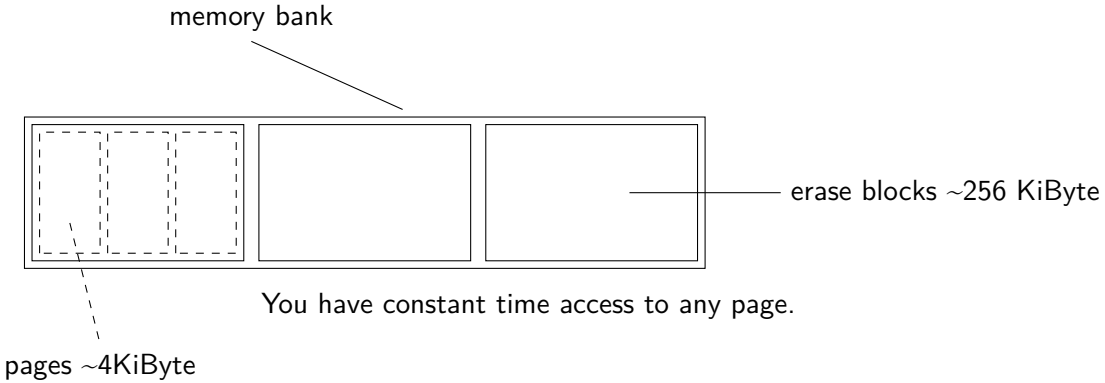
How do we store a file system in the sectors of a SSD?

Anatomy of a SSD



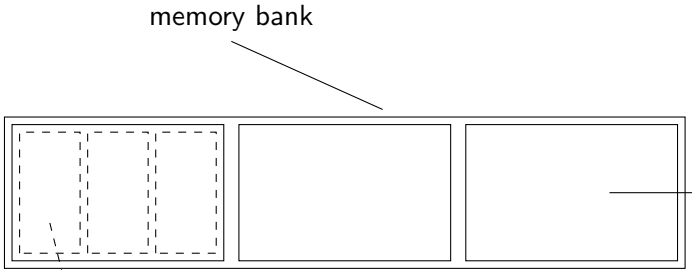
How do we store a file system in the sectors of a SSD?

Anatomy of a SSD



How do we store a file system in the sectors of a SSD?

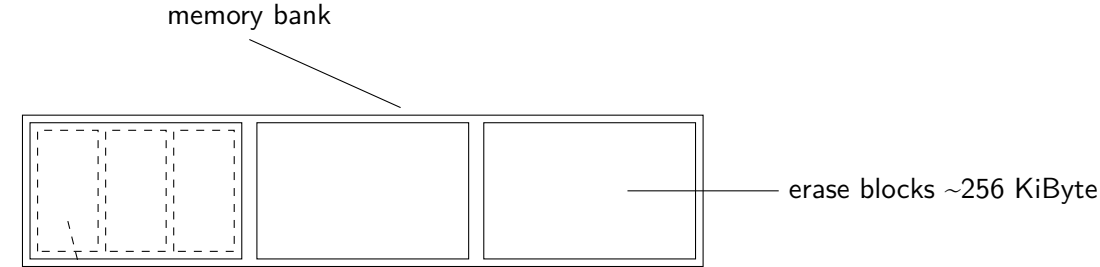
Anatomy of a SSD



You have constant time access to any page.
You can only write to (or program) an erased page.

How do we store a file system in the sectors of a SSD?

Anatomy of a SSD



How do we store a file system in the sectors of a SSD?

tmpfs

Reserve one part of main memory to store a file system.

tmpfs

Reserve one part of main memory to store a file system.

- Do we provide persistent storage?

tmpfs

Reserve one part of main memory to store a file system.

- Do we provide persistent storage?
- Is the storage much larger or cheaper than main memory?

tmpfs

Reserve one part of main memory to store a file system.

- Do we provide persistent storage?
- Is the storage much larger or cheaper than main memory?
- Can processes use the file system to share data?

man mount

All files accessible in a Unix system are arranged in one big tree, the file hierarchy, rooted at `/`. These files can be spread out over several devices. The mount command serves to attach the filesystem found on some device to the big file tree. Conversely, the umount(8) command will detach it again.

The standard form of the mount command is:

```
mount -t type device dir
```

This tells the kernel to attach the filesystem found on `device` (which is of type `type`) at the directory `dir`. The previous contents (if any) and owner and mode of dir become invisible, and as long as this filesystem remains mounted, the pathname `dir` refers to the root of the filesystem on device.

Some file systems

- `ext4` : the default file system used by most Linux distributions

Some file systems

- `ext4` : the default file system used by most Linux distributions
- `HFS+` : used by OSX

Some file systems

- `ext4` : the default file system used by most Linux distributions
- `HFS+` : used by OSX
- `NTFS` : default Windows file system

Some file systems

- `ext4` : the default file system used by most Linux distributions
- `HFS+` : used by OSX
- `NTFS` : default Windows file system
- `FAT32` : older file system from Microsoft, used if you want maximal portability

Summary

- separate directory service from file service

Summary

- separate directory service from file service
- directory as tree structure

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data
- file system, structure on disk

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data
- file system, structure on disk
- performance, caching, buffering

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data
- file system, structure on disk
- performance, caching, buffering
- take device anatomy into account

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data
- file system, structure on disk
- performance, caching, buffering
- take device anatomy into account
- mount several file systems in one tree

Summary

- separate directory service from file service
- directory as tree structure
- multiple hard links to files (not to directories)
- symbolic links are re-directions
- inodes, the data structure of file meta-data
- file system, structure on disk
- performance, caching, buffering
- take device anatomy into account
- mount several file systems in one tree