SUPSI

Filesystem

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Understand how a filesystem is implemented
- Study the issues related to reliable storage

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

Logical blocks vs physical blocks

Physical block

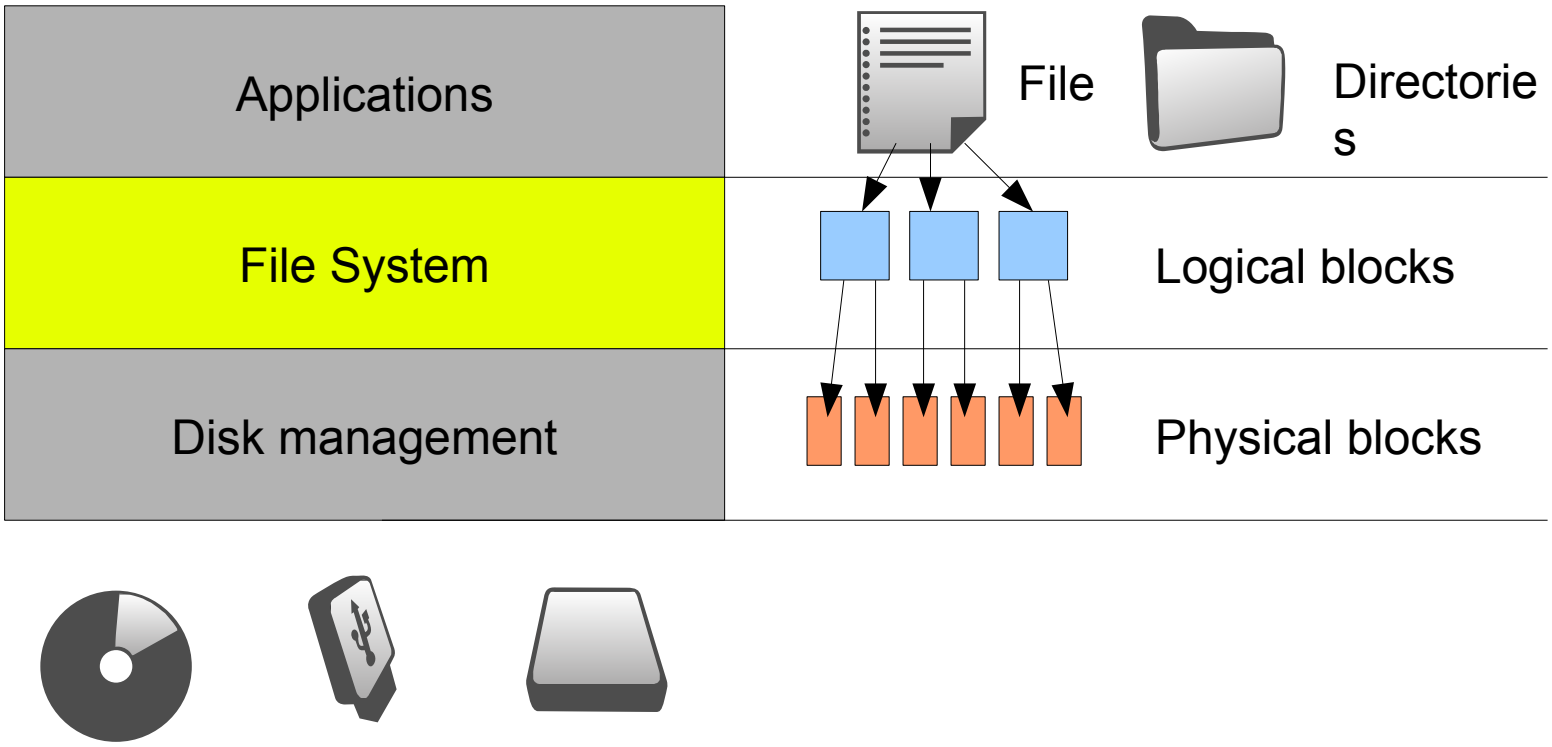
is the basic storage unit supported by the device (for example, a sector of a harddisk drive). Typically 512/4096 bytes

≠

Logical block

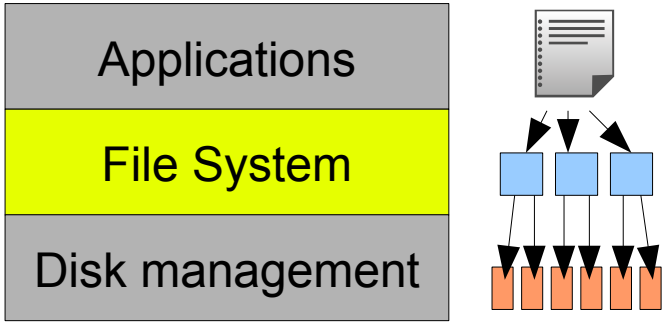
or *cluster*, is the smallest storage unit supported by the filesystem (can be a multiple of the physical block size, i.e. 512 bytes to 64 Kbytes)

Files, directories, logical blocks, physical blocks



The filesystem

- ... is an **abstraction** that presents block devices as a hierarchical structure of files and directories
- ... is a **component** of the operating system which implements and provide access (through system calls) to such an abstraction

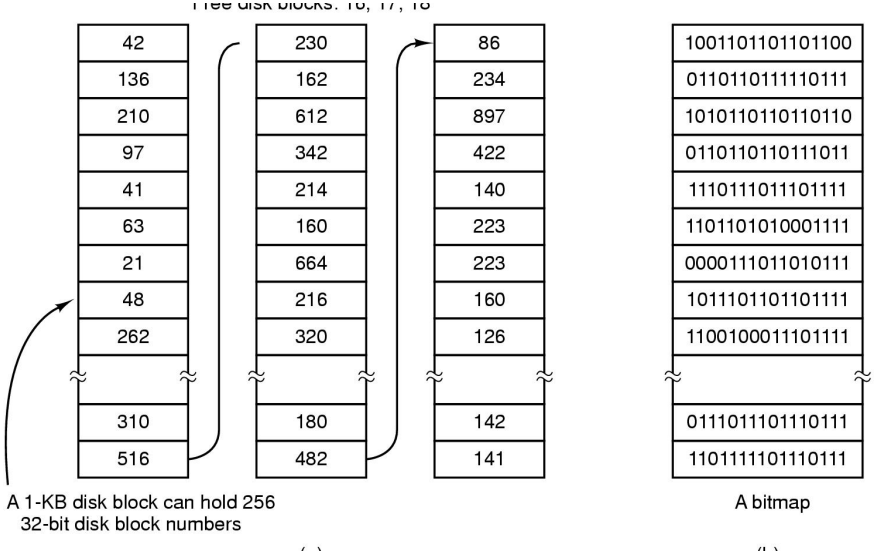



```
READ(2)                    Linux Programmer's Manual                    READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into
       the buffer starting at buf.

       On files that support seeking, the read operation commences at the cur-
       rent  file  offset, and the file offset is incremented by the number of
       bytes read.  If the current file offset is at or past the end of  file,
       no bytes are read, and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the
       absence of any errors, or if read() does not check for errors, a read()
       with a count of 0 returns zero and has no other effects.
```

Filesystem tasks

- Manage space
  - **free space** (blocks)
  - **allocated space** (blocks to files and directories)
- Provide a way to **access** and **manage data**
  - **filesystem hierarchy** (and a way to navigate it)
    - **naming**, **paths** (absolute / relative)
  - **system calls to deal with files** (open, read, write, ...)
- Provide **reliable** storage and **fast** operation

Manage free space (blocks)



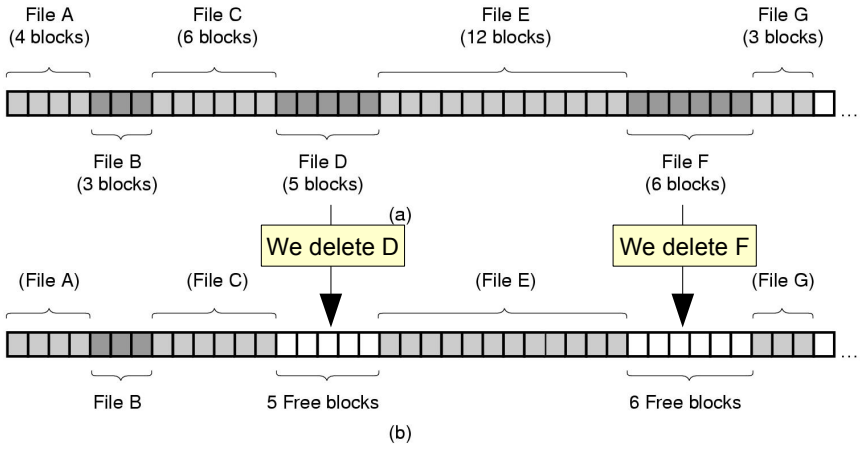
- Free space (blocks) can be managed using a **linked list** or a **bitmap**
  - To find the optimal set of contiguous blocks we can use a *-fit algorithm

Images from A. Tanenbaum, "Modern Operating Systems", 2nd Ed

Allocating and managing blocks

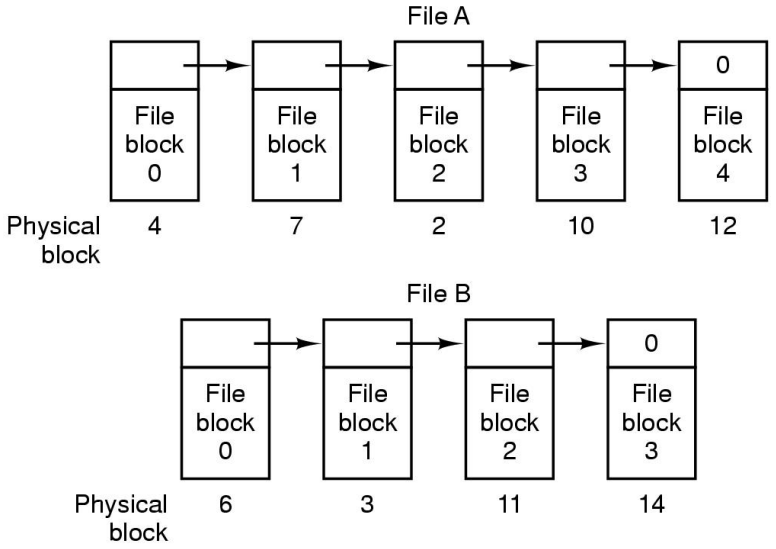
* we assume that there's an index somewhere that points to the beginning of a file

Contiguous allocation*



- **Simple**
- **Reading is efficient (we can read a file with few I/O operations)**
- **Fragmentation (on deletion)**

Linked-list*

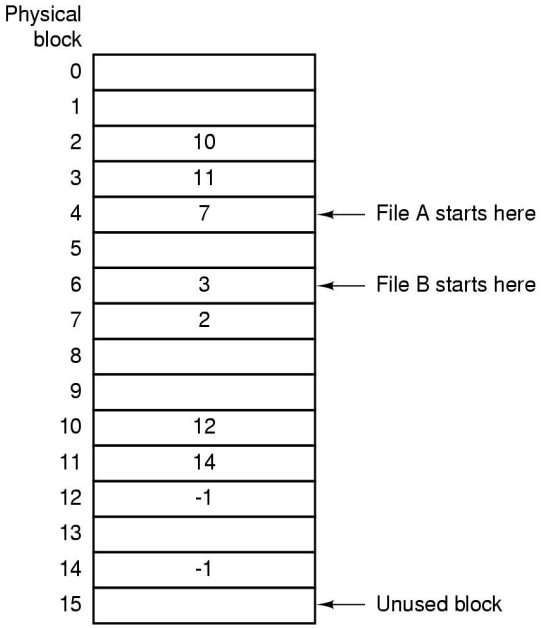


- **Supports non-contiguous blocks**
- **Each block must contain a pointer to the next block**
- **Slow sequential and random reading** (must read the entire chain)
- **Not reliable** (what if the chain is broken?)

Images from A. Tanenbaum, "Modern Operating Systems", 2nd Ed

Allocating and managing blocks

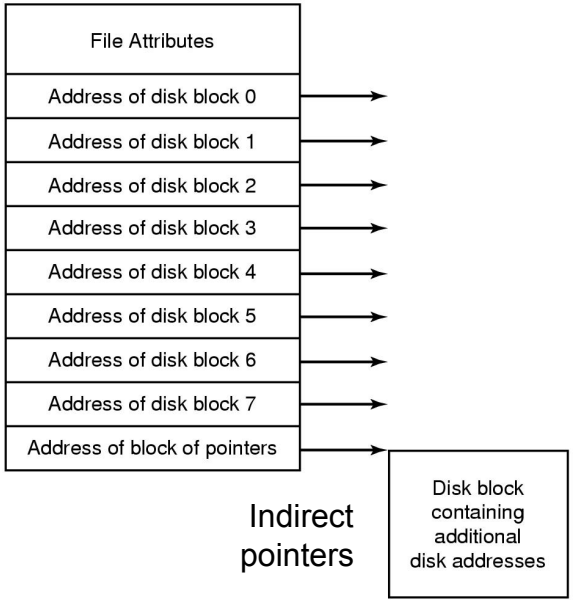
File Allocation Table (FAT)*



- **Easy to implement**
- **Quite fast random access** (if the table is kept in memory)
- **Table can be big on large disks**

* we assume that there's an index somewhere that points to the beginning of a file

Index Node (inode) **

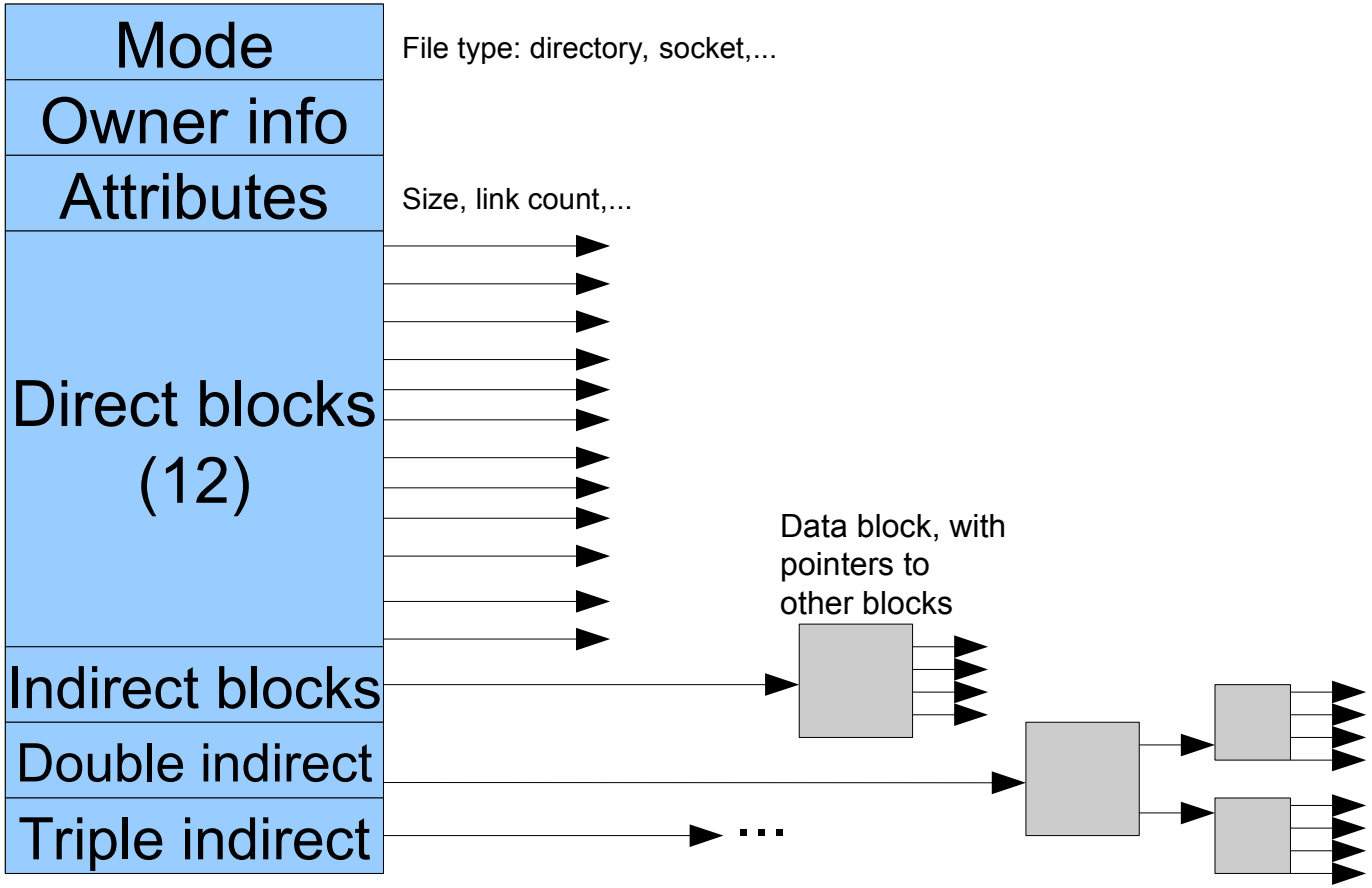


- **Quick random access**
- **Support for growing files** (with indirect pointers)

** we assume that there's an index somewhere that, for each file, points to the corresponding inode

Images from A. Tanenbaum, "Modern Operating Systems", 2nd Ed

Example inode on the ext2 filesystem (Linux)



Details: http://www.nongnu.org/ext2-doc/ext2.html

NTFS

- The core of NTFS is the **Master File Table** (MFT):
  - The MFT is like a database → everything's a file (the MFT itself is a file)
  - Each file is represented by a record in the MFT (1-4 Kbytes) which is contains a variable number of *key, attribute* pairs
    - The first 16 records of the table are reserved for special information

MFT Header	Standard information (access attributes)	File name	Security descriptor (who owns the file)	Data	Extended attributes

MFT record for a small file

  - Each MFT record is referenced by a 64 bit value:
    - a 48 bit MFT entry value (index in the MFT)
      - The first entry has address 0
    - a 16 bit sequence number that is incremented when the entry is allocated

NTFS: Master File Table

- The first **16 records** are reserved for special files, such as:
  - Record 0: correspondes to the $MFT file (a self-reference to the MFT)
  - Record 1: copy of first 4 records of the MFT
  - Record 2: Transactional log
  - Record 3: Serial number, creation time, dirty flag
  - Record 4: Continuation of $Mft
  - Record 5: root directory (\)
  - Record 6: volume cluster allocation file ($Bitmap)
  - Record 8: bad-cluster file ($BadClus)

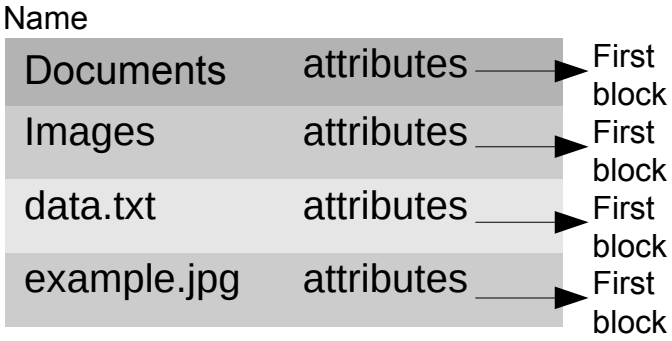
NTFS

- The attributes of a file are written inside the record: for optimization purposes, small files and directories (< 512 bytes) are entirely contained within the master file table record
  - Large directories are organized into B-trees, having records with pointers to external clusters
  - The data attribute points to runs of contiguous clusters called **extents**
    - each extent is described by a starting cluster number and a length
  - If attributes do not fit inside a record additional records can be used
    - those records will reference the base record using a specific field called base file record

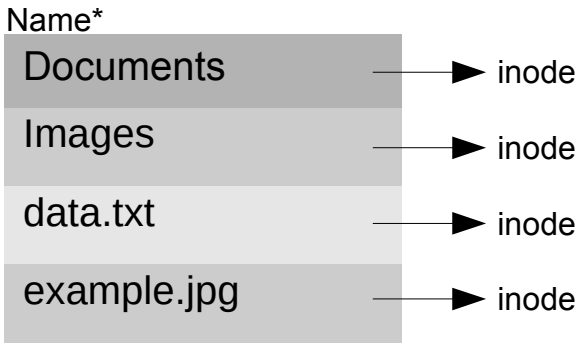
Reaching a file (or inode)

- **Directories** are special files which contain a list of files
  - these can be a simple arrays, linked lists or more complex structures (i.e. B-Trees)
  - a filesystem typically “starts” from a very specific directory: the root directory

Directory on FAT filesystems

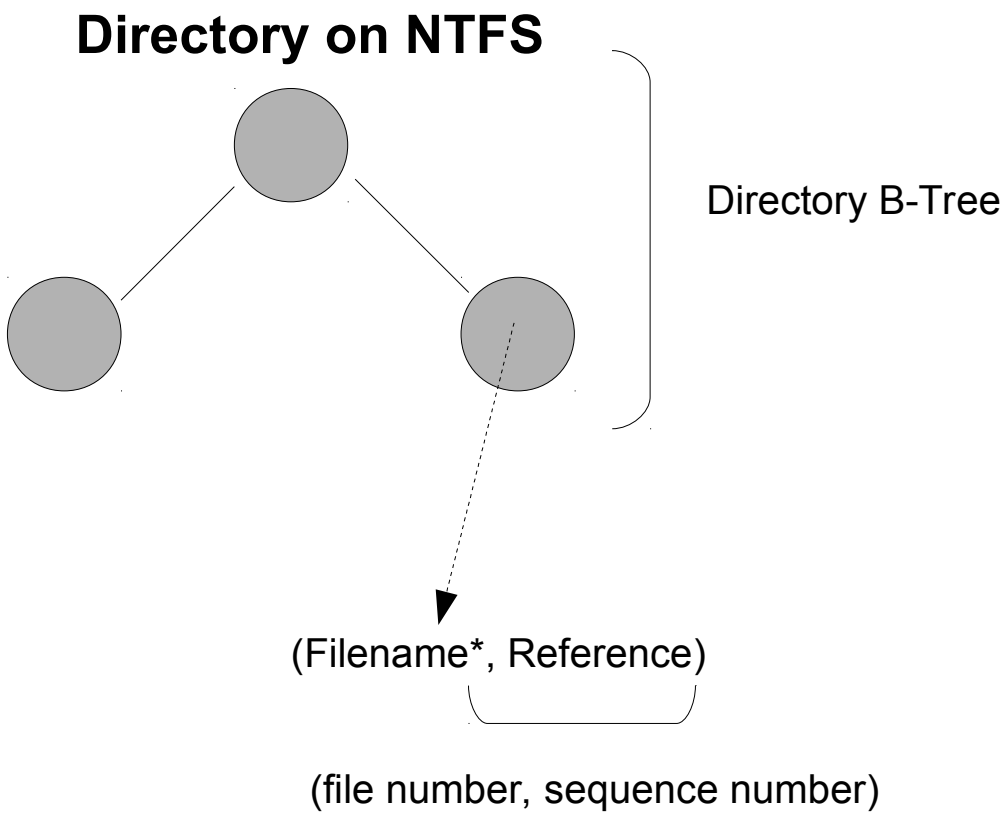


Directory on Index Node-based filesystems



* if more than one filename points to the same inode we have... **hard links**!

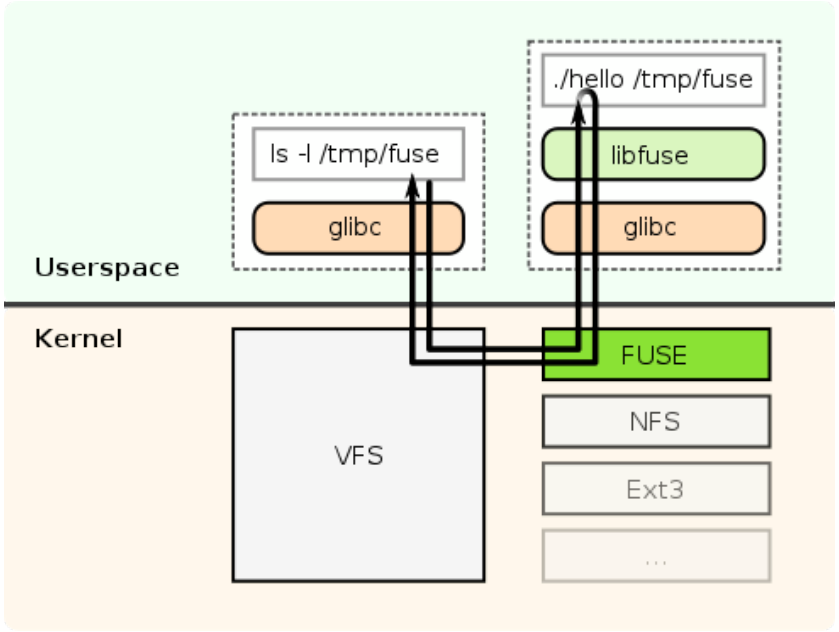
Directories



With NTFS the filename is stored both in the directory structure and in the file's MTF entry (filename attribute)

Filesystems in userspace (FUSE)

- Filesystems can also be implemented in user-mode (i.e. as user processes) using **FUSE** (Filesystem in USErspace)
  - https://github.com/libfuse/libfuse
  - There exist bindings for different programming languages:
    - Python https://github.com/libfuse/python-fuse
    - Java https://github.com/EtiennePerot/fuse-jna
  - and several filesystems implemented with it:
    - EncFs (Encrypted Filesystem) https://vgough.github.io/encfs/
    - NTFS-3g https://www.tuxera.com/community/open-source-ntfs-3g/
    - SSHfs https://github.com/libfuse/sshfs



(Image: https://en.wikipedia.org/wiki/Filesystem_in_Userspace#/media/File:FUSE_structure.svg License: This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license)

Fuse example

```
/*
  FUSE: Filesystem in Userspace
  Copyright (C) 2001-2007  Miklos Szeredi <miklos@szeredi.hu>
  This program can be distributed under the terms of the GNU GPL.
  See the file COPYING.
*/
#define FUSE_USE_VERSION 30
#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
static const char *hello_str = "Hello World!\n";
static const char *hello_path = "/hello";
```

compile with
gcc -Wall hello.c $(pkg-config fuse --cflags --libs) -o hello

Fuse example (2)

```
static int hello_getattr(const char *path, struct stat *stbuf)
{
        int res = 0;
        memset(stbuf, 0, sizeof(struct stat));
        if (strcmp(path, "/") == 0) {
                stbuf->st_mode = S_IFDIR | 0755;
                stbuf->st_nlink = 2;
        } else if (strcmp(path, hello_path) == 0) {
                stbuf->st_mode = S_IFREG | 0444;
                stbuf->st_nlink = 1;
                stbuf->st_size = strlen(hello_str);
        } else
                res = -ENOENT;
        return res;
}
```

Fuse example (3)

```
static int hello_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                         off_t offset, struct fuse_file_info *fi)
{
        (void) offset;
        (void) fi;
        if (strcmp(path, "/") != 0)
                return -ENOENT;
        filler(buf, ".", NULL, 0);
        filler(buf, "..", NULL, 0);
        filler(buf, hello_path + 1, NULL, 0);
        return 0;
}
static int hello_open(const char *path, struct fuse_file_info *fi)
{
        if (strcmp(path, hello_path) != 0)
                return -ENOENT;
        if ((fi->flags & 3) != O_RDONLY)
                return -EACCES;
        return 0;
}
```

Fuse example (4)

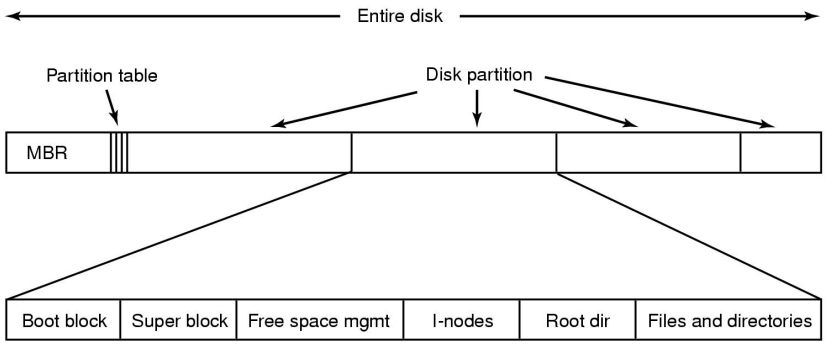
```
static int hello_read(const char *path, char *buf, size_t size, off_t offset,
                      struct fuse_file_info *fi)
{
        size_t len;
        (void) fi;
        if(strcmp(path, hello_path) != 0)
                return -ENOENT;
        len = strlen(hello_str);
        if (offset < len) {
                if (offset + size > len)
                        size = len - offset;
                memcpy(buf, hello_str + offset, size);
        } else
                size = 0;
        return size;
}
// fuse_operations hello_oper is redirecting function-calls to _our_ functions implemented above
static struct fuse_operations hello_oper = {
        .getattr        = hello_getattr,
        .readdir        = hello_readdir,
        .open           = hello_open,
        .read           = hello_read,
};
```

Fuse example (5)

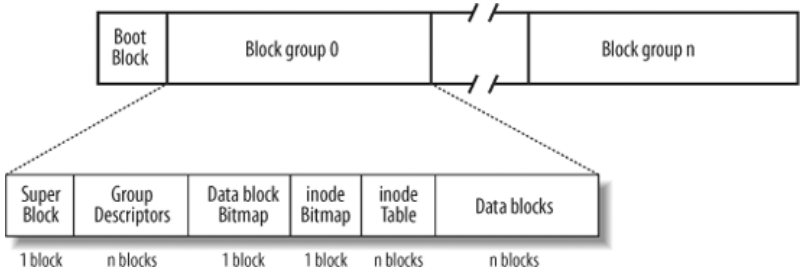
```
// in the main function we call the blocking fuse_main(..) function with &hello_oper
int main(int argc, char *argv[])
{
        return fuse_main(argc, argv, &hello_oper, NULL);
}
```

Where are all these structures stored: the filesystem layout

- Each filesystem uses a different **layout** to organize data (created upon formatting)
  - **super block** (contains information about the filesystem)
  - free space management (for example, bitmap)
  - inodes



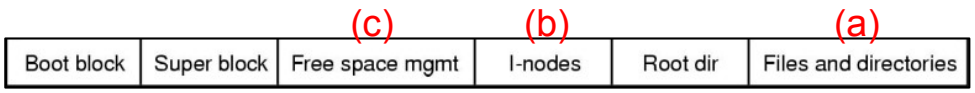
Example generic layout
(from A. Tanenbaum, Modern Operating Systems, 2nd ed)



Linux ext2/3 layout
(from Bovet, Cesati, Understanding the Linux Kernel, 3rd Edition)

Crash-consistency

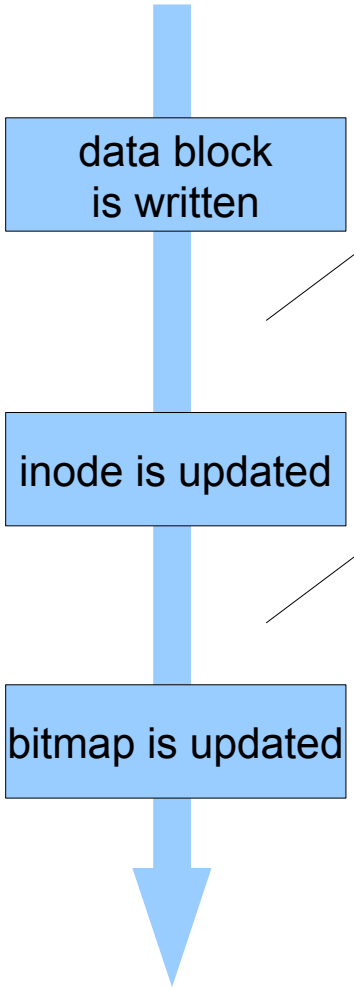
- What happens when we write a file? (in this example, on a inode based filesystem)
  - The **data block** is written (a)
  - The **inode** is updated (c)
  - The **bitmap table** is updated (c)



- These data structures reside in different blocks on disk and **are thus not updated atomically**
  - The update/write order might change
  - Moreover there might be caches, buffers to store that data (either at the operating system level, or in the hardware) so data is not guaranteed to be already on the disk.

- What happens when the system crashes during this operation? Because the crash can happen anytime not all updated data structures might have been written to disk...

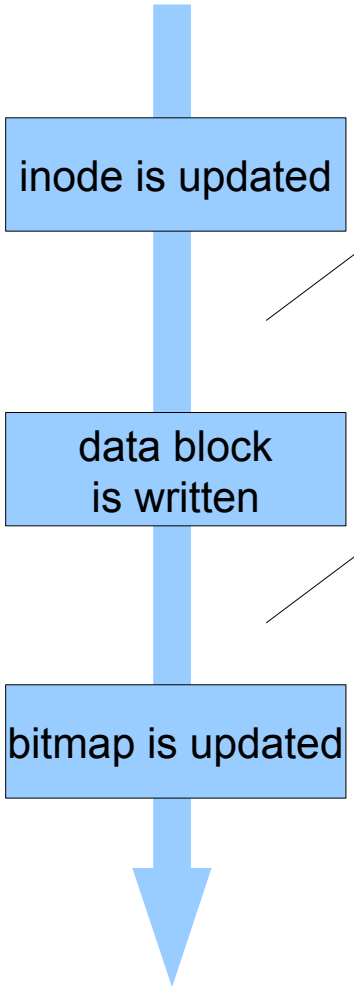
Possible scenario



Just the data block is written: not a problem from the filesystem point of view (but might be a problem for the application).
Data is unaccessible and corresponding data block is still marked as free

The data block and the inode are written: the node points to the correct data, but the bitmap still indicates the block as free

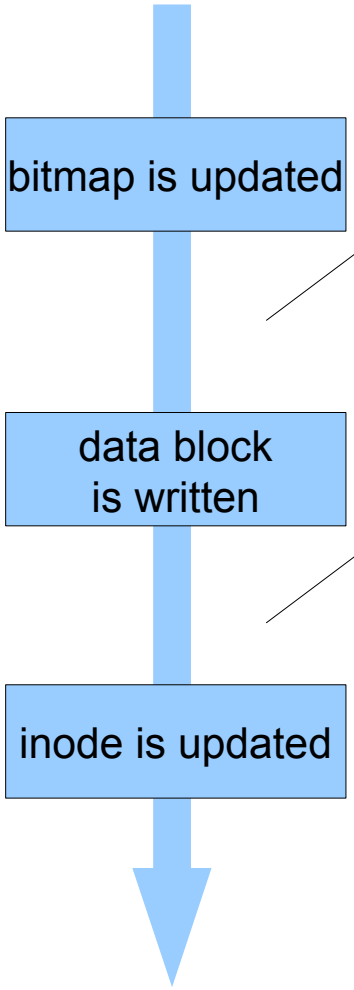
Possible scenario



Just the inode is updated: the inode might point to old data or garbage data. Furthermore the bitmap will tell us that the same datablock is still free, leading to a **file-system inconsistency**.

The data block and the inode are written: the node points to the correct data, but the bitmap still indicates the block as free

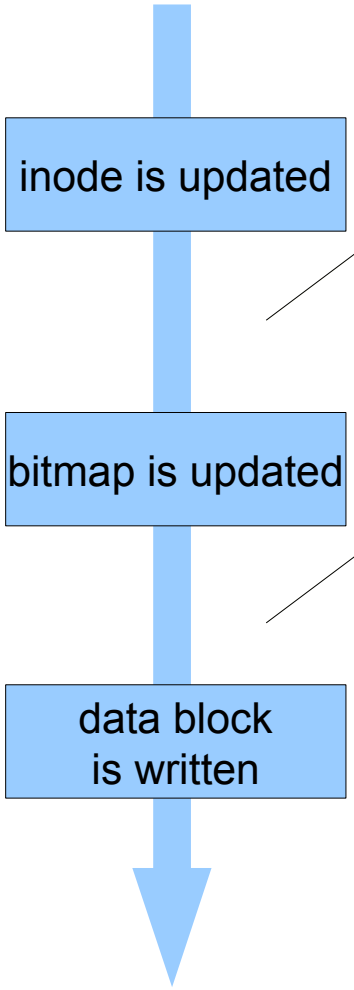
Possible scenario



Just the bitmap is updated: the bitmap might indicate that a block is used but there is no inode that points to it: we have a disk space **leak**.

The data block and the bitmap are written: we might not know which file the block belongs to.

Possible scenario



Just the inode is updated: the inode might point to old data or garbage data. Furthermore the bitmap will tell us that the same datablock is still free, leading to a **file-system inconsistency**.

The inode and the bitmap are written: we get old / garbage data when trying to read the file

Solution: Filesystem checking

- At boot, before mounting the filesystem do a filesystem check:
  - check the superblock
  - check allocated blocks by analyzing each inode, keep track of free blocks and see if they are marked as such in the bitmap
  - check inode state
  - check link count
  - check for inodes that point to the same blocks
  - check for block references outside the range
  - check directory structure

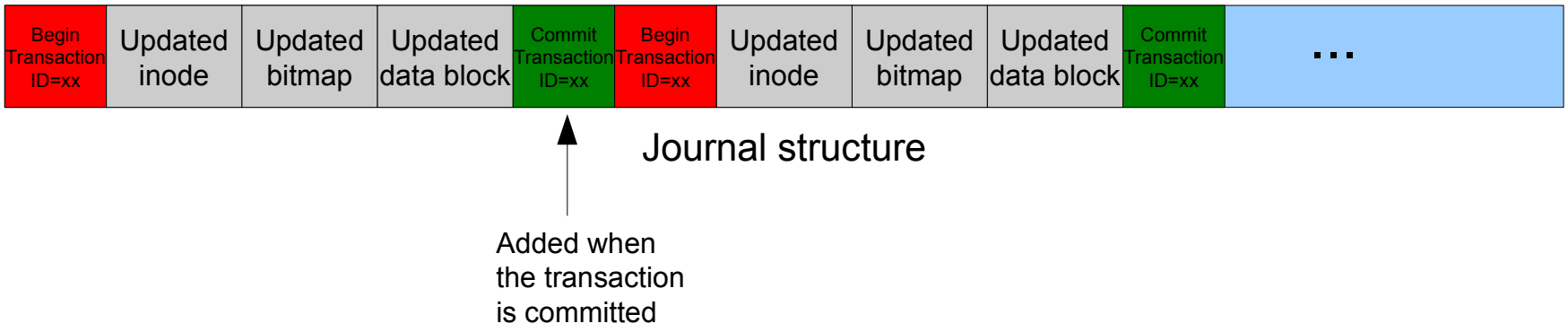
Solution: Journaling

- Use a journaling mechanism: the filesystem maintains a log of all the changes:
  - when updating the disk, before writing to the actual data structures, a log with the intended action is written
  - if the system crashes we can examine the log and reply the log

- Types of journaling
  - **Physical**: each update on disk (blocks) is logged
  - **Logical**: only changes to the metadata (i.e. inodes, bitmaps) are logged

Journaling on the Linux ext3 filesystem

- The ext3 journal is commonly stored as a file within the file system (inode 8)
  - Information about pending file system updates is written to the journal
    - Journal updates happens before updating the corresponding data structures (**write-ahead logging**)
    - The journal works as a circular buffer containing **transactions**
      - several updates can also be grouped into a single compound transaction that is periodically committed to disk

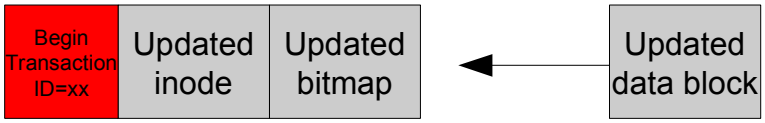


Journaling on the Linux ext3 filesystem

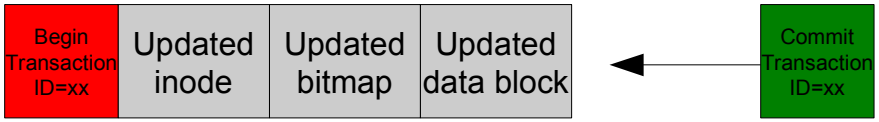
- ext3 supports three journaling modes (chosen at mount time)
  - **writeback**: only the filesystem metadata (*inode*, *bitmap, directories*) is journaled, data blocks are directly written directly to their fixed location → **weakest consistency, fastest operation**
  - **ordered**: only metadata updates are journaled, but the filesystem flushes data blocks updates to disk before updating the metadata
  - **full data journaling**: both metadata and data are logged into the journal. When a process writes a data block it is written first into journal, and then later to its location → **strongest consistency, slowest operation**

Journaling on the Linux ext3 filesystem

- With **full data journaling** updates are performed as follows:
  1) A **new transaction** is written into the journal; the filesystem waits for these writes to complete



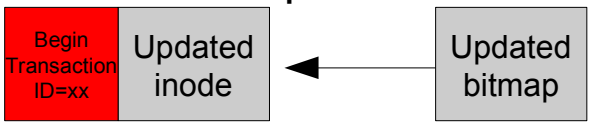
  2) The transaction is **committed**



  3) Transaction **checkpoint**: updates are performed on the filesystem structures and the data block
  4) (later, at some point) **Freeing**: the marked as free (as the journal cannot grow indefinely)

Journaling on the Linux ext3 filesystem

- With **writeback** or **ordered journaling** updates are performed as follows:
  1) The data block is written to disk; with **ordered journaling** wait for this write to complete to guarantee that the filesystem will never point to garbage data
  2) A **new transaction** is written into the journal; the filesystem waits for these writes to complete



  3) The transaction is **committed**

  4) Transaction **checkpoint**: updates are performed on the filesystem structures
  5) (later, at some point) **Freeing**: the marked as free (as the journal cannot grow indefinely)

Recovery

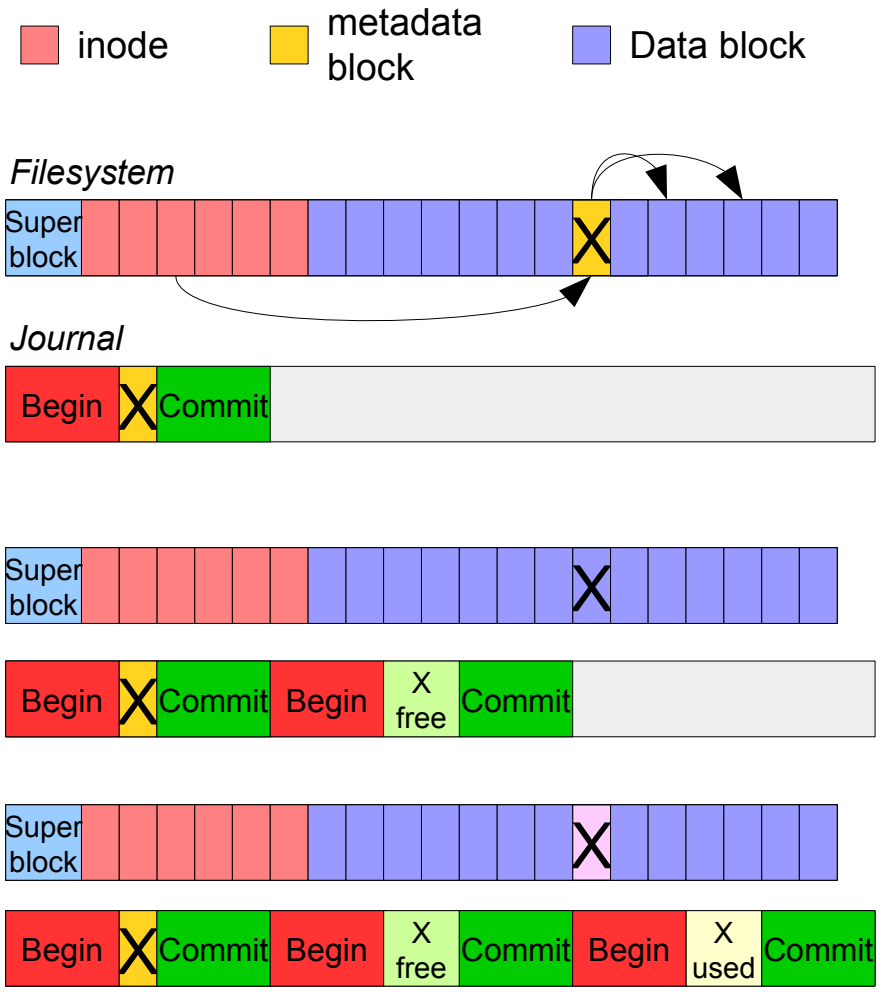
- In the event of a crash (which may happen at any time) we need to perform a **recovery** before mounting the filesystem

- For each transaction in the journal:
  - If the crash happened before the transaction has been committed the pending update is ignored
  - If the crash happened between the commit and the checkpoint the transaction is **replayed** (changes on disk are redone)

Block reuse issue

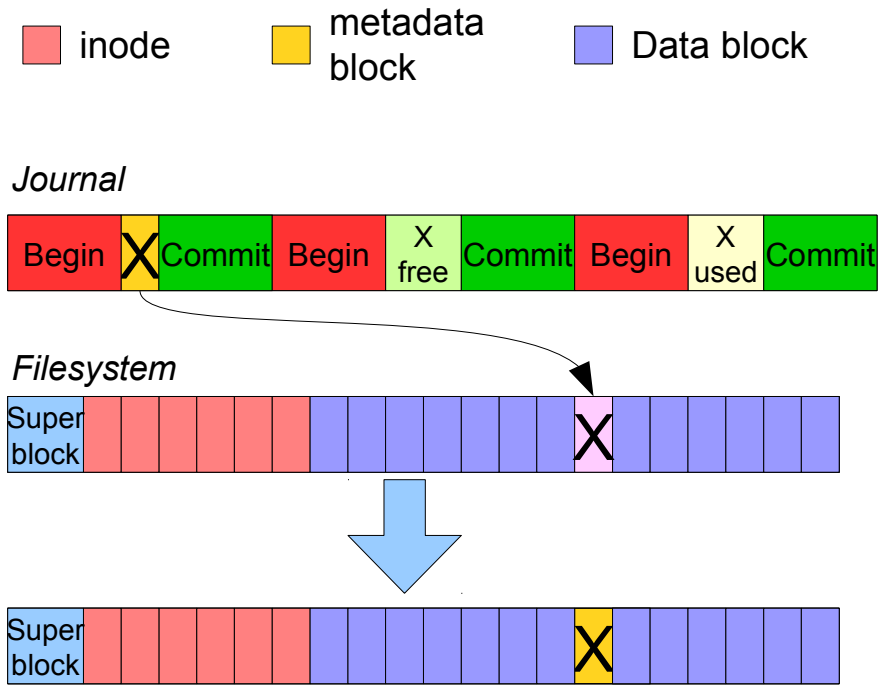
- Consider the following situation concerning metadata journaling:
  - A **metadata block X** is updated (for example, when using indirect blocks in the inode)
    - a new transaction is written into the journal
  - The same block **X** is, at some point, freed
  - Block **X** is **reused** to store some user information

THE SYSTEM CRASHES...



Recovery...

- The system reboots and the journal log is replayed...



We just overwrote some user data!

Revoke

- To avoid the aforementioned issue ext3 puts a **revoke** record into the journal when a metadata block is freed
  - When replaying the journal the operating system looks for revoke records and avoids replaying the associated data

Other uses of the file system: File locking

- The filesystem can also be used for synchronization purposes, using **file locking**
  - Locking can be **mandatory** (prevents and read/write on the locked file) or **advisory** (other processes can read/write but are *advised* to check if a lock exist)

- Not to be confused with lock files, which are useful to prevent multiple instances of a program running at the same time... and a little less useful when Firefox crashes and refuses to restart

File locking example

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[]) // filelock.c
{
    int fd;
    struct flock file_lock;
    fd = open ("mylockedfile", O_WRONLY | O_CREAT);
    file_lock.l_type   = F_WRLCK;
    file_lock.l_whence = SEEK_SET;
    file_lock.l_start  = 0;
    file_lock.l_len    = 0;
    /* Lock */
    fcntl (fd, F_SETLKW, &file_lock);
    printf("Run filelockcheck ... then press enter\n");
    getchar();
    /* Unlock.  */
    file_lock.l_type = F_UNLCK;
    fcntl (fd, F_SETLKW, &file_lock);
    close (fd);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[]) // filelockcheck.c
{
    int fd;
    struct flock file_lock;
    fd = open ("mylockedfile", O_WRONLY | O_CREAT);
    file_lock.l_type   = F_WRLCK;
    file_lock.l_whence = SEEK_SET;
    file_lock.l_start  = 0;
    file_lock.l_len    = 0;
    /* Check Lock */
    fcntl (fd, F_GETLK, &file_lock);
    if (file_lock.l_type != F_UNLCK) {
        /* Who has the lock? */
        printf("%d has the lock\n", file_lock.l_pid);
    } else { printf("Nobody has the lock\n"); }
    close (fd);
    return 0;
}
```