BTI1341 – Operating Systems

Part 3: Persistence – 11) RAID, Files and Directories

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- RAID
- Files and Directories
- Appendix

RAID

Motivation

A Redundant Array of Independent Disks (RAID) [1] improves one or more of the following aspects, compared to a single HDD:

Performance: As seen last time, HDD I/O transfer rate is comparatively slow and often the bottleneck for performance-critical applications.

Capacity: Even though HDD capacity has constantly increased, often more space than available on a single disk is required.

Reliability: HDDs can (and do!) fail, leading to potential downtime and partial or complete data loss.

⚠ Attention: RAID is not a replacement for a working backup!

[1] Earlier: Redundant Array of *Inexpensive* Disks...

RAID Types I

A RAID system is a combination of *at least 2* HDDs, which can be accessed like a single HDD by the host computer. Implementation can be done in hard- or in software:

- Hardware RAID

  All HDDs of the RAID are attached to a RAID controller. Such a controller can be complex and include separate memory and backup power. Modern mainboards often also offer on-board RAID functionality. On the host side, the controller attaches transparently as a regular HDD.

- Software RAID

  HDDs are attached normally to the host computer. All RAID functionality is implemented in software; typically in the OS as a device driver, or as part of a volume manager or file system.

RAID Types II

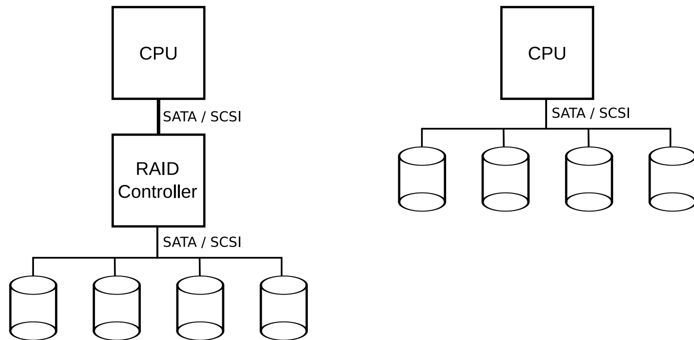

Figure: Hardware RAID (Left) and Software RAID

Logical Interface and Fault Model

In the following, any physical details of the RAID (interface, soft- / hardware RAID etc.) are ignored and *the same logical interface* as for HDDs is assumed:

- A RAID is just a *linear array of blocks*.
- Blocks can be addressed and read/written *individually*.

The RAID internally converts from logical- to physical I/O requests .
A single logical request may result in multiple physical requests.

For now, only fail-stop faults are assumed:

- Any RAID HDD can only be in two states: *working* (all blocks accessible) or *failed*.
- *Failure detection* is assumed to be easy and fast.

In the following, N is the number of disks and B the number of blocks.

RAID Level 0 (Striping)

RAID Level 0 or striping is not really a RAID level, as it provides *no redundancy*. Basic idea: Stripe blocks across all disks.

For performance and capacity, it is optimal: It has $N \cdot B$ *blocks* in total, N *can be accessed simultaneously*.

Example: The following table depicts a RAID 0 system with 4 disks and 16 blocks (4 per disk).

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Chunk Size

The example on the previous slide has a chunk size of 1, i.e. *one block per chunk*. For a chunk size of 2, it would look as follows:

Disk 0	Disk 1	Disk 2	Disk 3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

Chunk size affects performance:

- Small chunks → many files have to be striped across disks...
  - More reads and writes in parallel
  - Positioning time increases
- Big chunks reduce parallelism per file but also positioning time.
- Optimal size is workload dependent!

RAID Level 1 (Mirroring)

RAID Level 1 or mirroring provides *redundancy of* $N - 1$ *disks*, at the expense of *capacity reduced to* 1 (i.e. a single disk).

Performance heavily depends on implementation and ranges from 1 *to* $\sim N$ *for reads*. Write performance is 1 in general.

Basic idea: *Store a copy of every block on every disk.*

Example: The following table depicts a RAID 1 system with 2 disks and 4 blocks in total.

Disk 0	Disk 1
0	0
1	1
2	2
3	3

RAID 10 and RAID 01 I

Basic idea: *Combine RAID levels 0 and 1.*

RAID 10 combines two or more mirror pairs into a RAID 0:

Mirror 0,0	Mirror 0,1	Mirror 1,0	Mirror 1,1
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Capacity is $\frac{N}{M}$ (typical), where M is the number of mirror pairs.

Redundancy ranges from at least 1 up to $\frac{N}{2}$.

Performance can be assumed similar to RAID 0.

RAID 10 and RAID 01 II

On the other hand, a RAID 01 system is a mirror of *striped pairs:*

Stripe 0,0	Stripe 0,1	Stripe 1,0	Stripe 1,1
0	1	0	1
2	3	2	3
4	5	4	5
6	7	6	7

⚠ In general, this is not such a good idea: The failure of a single disk always leads to the loss of a complete stripe!

RAID Level 4 (Parity)

RAID Level 4 uses parity information to provide reliability, while improving overall capacity.

Basic idea: Have an *additional disk* storing parity information.

Example: The following table depicts a RAID 4 system with 5 disks (1 parity) and 16 blocks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Implementing Parity

To compute the parity, the XOR ($\oplus$) function can be used.

Example: Given 4 bits $\{0, 0, 1, 1\}$, the parity is $0 \oplus 0 \oplus 1 \oplus 1 = 0$, where $\oplus$ is the XOR function.

XOR parity is defined as an *even amount of 1's* including the parity bit. It can detect *single bit* errors; if the faulty bit is known, it can also repair them:

B_0	B_1	B_2	B_3	$Parity$
0	0	1	1	0
0	1	0	0	1

The failed bit can be restored by XORing the remaining bits:

0	X	1	1	0	Fix: $0 \oplus 1 \oplus 1 \oplus 0 \rightarrow X = 0$
0	1	0	0	X	Fix: $0 \oplus 1 \oplus 0 \oplus 0 \rightarrow X = 1$

Additive and Subtractive Parity I

When writing large, sequential chunks, full-stripe writes can be used. They synchronously *update a whole stripe*, including parity:

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	*1*	*2*	*3*	*P0*
...	...	...	...	...

However, with random writes, a single block and parity must be updated:

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	*2*	3	*P0*
...	...	...	...	...

Additive and Subtractive Parity II

There are two ways to compute a parity:

Additive parity : Read out *every block* of a stripe and compute a fresh parity, i.e. compute $0 \oplus 1 \oplus 2 \oplus 3 \oplus$ P0 for blocks 0-3.

Problem: scales with the *number of disks*. However, due to the properties of the XOR function, subtractive parity solves this:

$$P_{\text{new}} = B_{\text{old}} \oplus B_{\text{new}} \oplus P_{\text{old}}$$

Example:

- Old value: $\{\mathbf{0}, 0, 1, 1, \mathbf{0}\}$
- New value: $\{\mathbf{1}, 0, 1, 1, \mathbf{X}\}$
- Compute $0 \oplus 1 \oplus 0 = 1$, thus $\{\mathbf{1}, 0, 1, 1, \mathbf{1}\}$ is the new value to be written.

The Small-Write Problem

Small (and random) writes cause yet another issue: the small-write problem . Assume the following small writes:

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	*P1*
8	9	10	11	P2
12	*13*	14	15	*P3*

As blocks 4 and 13 are on different disks, they could be written in parallel. However, both writes need to modify parity, thus they depend on the parity disk. *The parity disk becomes the bottleneck.*

RAID Level 5 (Rotating Parity)

RAID Level 5 addresses the small write problem. Basic idea: *Distribute parity data* over all disks of the RAID.

Example: The following table depicts a RAID 5 system with 5 disks and 20 blocks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID Level 6 (“Double Parity”)

RAID Level 6 addresses more problems: rebuild time and failure probability .

With *large HDD capacities*, a rebuild may take several hours or even days. Chances that a *second disk fails* during this time have become relevant.

Basic idea: Have *redundant parity* information.[2]

Example: The following table depicts a RAID 6 system with 5 disks and 12 blocks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	P1	P2
3	4	P1	P2	5
6	P1	P2	7	8
P1	P2	9	10	11

[2] XOR cannot be used, requires different parity functions (not discussed).

RAID Level X, JBOD, SANs, ...

In practice, more RAID levels like 1E, 50, 60, 100 etc. are in use. As these are based on the principles discussed so far, they will not be further treated. Additionally, hot-spare (standby) disks may also be used to shorten rebuild time.

A different, non-RAID architecture is JBOD, which stands for “just a bunch of disks”. This is a (large) deployment of independent HDDs or also RAIDs, used by volume managers. They are often combined to larger volumes, called (linear) spans. Spans offer no redundancy but more flexibility, e.g. to combine different drive types.

Finally, Storage Area Networks (SANs) are (large) deployments of HDDs and other storage media in a network topology, exposing block devices. SANs allow for virtualization of storage and use by different nodes and accross different sites.

Files and Directories

One More Abstraction

Earlier, we have introduced processes and address spaces for virtualizing the CPU and memory. Together, they allow a program to run as if being alone on the machine (and to share the machine between multiple processes).

So far, in this final part of the course, we have introduced persistent storage using HDDs. As this is a *shared resource*, another abstraction is required. Furthermore, implementing direct, block-based I/O in every program requiring disk access would be highly unpractical.

Most abstractions of persistent storage are based on the basic concepts of files and directories, which will now be discussed.

Definitions

Definition

A **file** is a *linear array of bytes* which can be read and written. Each file has a low-level name, called its **inode number**. Besides content and inode number, every file also has some associated metadata. It is assumed that the OS has no concept of the file structure, i.e. cannot distinguish text files from pictures etc.

Definition

A **directory** is considered a special kind of file. It also has an inode number, but specific contents: a list of *(inode, name)-pairs*, which maps the low-level names (inodes) to user-readable names. Different files and directories can have the same name, as long as they are not in the same directory.

See Slide 39 for a graphical overview.

Everything is a File (-Descriptor)

The definitions on the previous slide correspond to a central aspect of the UNIX philosophy: everything is a file .[3]

Not only "real" files can be found in UNIX systems, but also devices (e.g. printers), named pipes, network connections etc., which are all managed as *byte streams* using the same API.

Furthermore, virtual filesystems provide access to OS internals (e.g. "`/proc`" and "`/sys`" in GNU/Linux).

Other OSes may differ completly or adhere in parts to this concept. For example, in NTFS file systems, data is not strictly organized in files.

[3] Sometimes also: everything is a file descriptor.

File API

Recap: The most relevant functions of the POSIX file API.

```c
// open or create a file, returns a file descriptor
int open(const char *pathname, int flags);

// read and write to a file descriptor
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);

// close a file descriptor
int close(int fd);
```

☞ See the corresponding man pages for details.

File Descriptors

A file descriptor (FD) is simply an integer which is used as a *handle* for an open file.

Each process has its own set of open file descriptors and their numbering always starts at 0. In a new process, the following file descriptors are open by default:

FD	Name	Usage
0	`stdin`	Reading input, e.g. from a shell.
1	`stdout`	Writing output.
2	`stderr`	Separate channel for error output.

New file descriptors are created by `open()` and various other syscalls, in general they return the next free fd number.

👉 On GNU/Linux systems, a list of open file descriptors of a given process can be found in `/proc/<PID>/fd`.

The Open File Table

For every process, the open file descriptors must be tracked by the OS. For this, a per-process *open file table* is typically used.

Example: In xv6, the open file table is part of `struct proc`:

```
// proc.h:38
// (NOFILE is 16, defined in param.h)
struct proc {
  ...
  struct file *ofile[NOFILE];  // Open files
  ...
}
```

xv6 also maintains a *global file table* (to see why, cf. Slide 32):

```
// file.c:14
// (NFILE is 100, defined in param.h)
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

File Offsets

Each file has a file offset, indicating its current position. Different operations use and/or modify the file offset, either

- Implicitly: E.g. `read()` and `write()` use and adjust the offset on every call.
- Explicitly: Direct adjustment, e.g. using `lseek()`.[4]

In *xv6*, the offset is maintained along with other attributes in a file descriptor table entry:

```
// file.h:1
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off; // <-- file offset
};
```

[4] Note that `lseek()` and HDD seeks are two different things!

File Offsets: Example 1

In this example, a file with a size of 300 bytes is opened and reads of 100 bytes each occur, until the whole file has been read.

The following table gives the return codes (RC) for each syscall, as well as the offset in the corresponding entry 10 of the open file table (OFT):

Syscalls	RC	OFT[10]
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

File Offsets: Example 2

In this example, a process opens the same file twice (file descriptors 3 and 4), resulting in two *different* OFT entries (10 and 11). It then reads 100 bytes from each descriptor and closes both files:

Syscalls	RC	OFT[10]	OFT[11]
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

File Offsets: Example 3

Finally, in this example, a file is opened and the file offset is initially adjusted to 200 using `lseek()`. Then, 50 bytes are read and the file is closed:

Syscalls	RC	OFT[10]
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Shared File Table Entries I

An entry in the open file table can be shared between processes or multiple file descriptors. Common cases are:

- *Open file descriptors are shared between parent and child* after a call to `fork()`! (☞ see "`man 2 fork`" for details)
- FDs *can be duplicated* using `dup()`, `dup2()` and `dup3()`, e.g. for implementing output redirection.
- Using `sendmsg()`, FDs can be *sent to another process* (basically, a pointer to the OFT entry is passed).

When a file table entry is shared, its *reference count* (see Slide 28) is incremented. It is only removed when reaching 0 again.

The figure on the following slide illustrates this.

Shared File Table Entries II

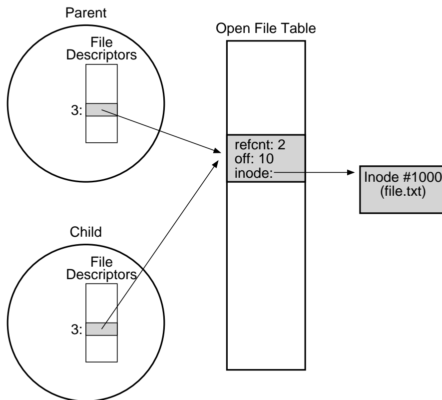

Figure: A Shared OFT Entry

Courtesy of [ADAD18]

Other Important File Syscalls

There are additional, important system calls for interacting with files, which will not be further detailed. Amongst them:

```
// force writing a file to disk
// (may be buffered by the file system!)
int fsync(int fd);

// rename a file
int rename(const char *oldpath, const char *newpath);

// retrieve file metadata
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);

// delete a file
int unlink(const char *pathname);
```

Analyze Syscalls: strace

Using **strace** (or similar tools on other OSes), all syscalls made by a process can be dumped.[5] This is an easy way to debug and understand how a program works.

Example: Analyzing file-related syscalls in "**cat**":

```
$ echo "hello, strace!" > file1
$ strace cat file1
openat(AT_FDCWD, "file1", O_RDONLY)     = 3
...
read(3, "hello, strace!\n", 131072)     = 15
write(1, "hello, strace!\n", 15)        = 15
read(3, "", 131072)                     = 0
...
close(3)                                = 0
...
```

[5] ☞ There is also `ltrace` for tracing library calls.

Directory API

Per definition, *a directory is also just a file* (cf. Slide 23). However, its contents are file system metadata (implementation specific) and *cannot be read or written directly*.

The most important functions for manipulating directories are:[6]

```
// create a directory
int mkdir(const char *pathname, mode_t mode);

// open a directory
DIR *opendir(const char *name);

// read directory contents
struct dirent *readdir(DIR *dirp);

// close a directory
int closedir(DIR *dirp);

// remove a directory (must be empty!)
int rmdir(const char *path);
```

[6] There are also directory-specific syscalls, not intended for direct usage.

Hard Links

When a new file is created, basically two things happen:

1. *A fresh inode is created*, which tracks all relevant information and the data (contents) of the new file.
2. A human readable *name is linked to the inode* in the corresponding directory.[7]

Using a hard link , an *additional name* can be linked to an existing inode.

Files keep a reference count , storing the number of hard links to them. When it reaches 0, the file is effectively deleted.

Example:

```
$ touch file1                   # create empty file
$ ln file1 file2                # create hard link
$ ls -i file1 file2             # ls with inode numbers
16384005 file1  16384005 file2
```

[7] Thus the name `unlink()` for the syscall to delete files...

Symbolic Links

Hard links have some limitations. For example, they do not work for directories or files on a different file system (e.g. different device).

Symbolic links can be used instead. They are an additional *file type* (besides regular files and directories), which serves as *some sort of pointer* to another file.

Example:

```
$ ln -s file1 file3                # create symbolic link
$ ls -i file1 file3
16384005 file1  16386513 file3
```

☞ Use the "`stat`" command to find out more about a given file!

Overview of Files and Directories

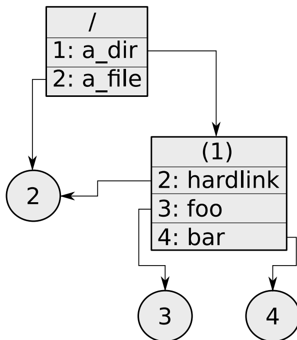

Figure: Files, Directories and Hard Links (Numbers are Inodes)

Valid paths:
`/a_dir`, `/a_file`, `/a_dir/hardlink`, `/a_dir/foo`, `/a_dir/bar`.

Access Control

Compared to the other abstractions, *files and directories are shared between multiple users and/or processes.* Thus, some additional access control mechanism is required!

On UNIX systems, permission bits are traditionally used for this.[8] They work as follows:

- There are three different permissions: read, write and execute.
- These apply either to the user (i.e. the owner), its group, or to all others.

NB: Read and write permissions do what the name implies. The execute permission works differently for files and directories:

- Files: Indicates that a file may be executed (e.g. programs).
- Directories: Gives permission to change into a directory.

[8] And also more advanced Access Control Lists (ACL), not discussed here.

Examples I

To change user and group of a file, `chown` and `chgrp` may be used:

```
$ ls -l file1
-rw-r--r-- 3 root root 15 May  1 22:01 file1
$ chown test file1
$ chgrp test file1
$ ls -l file1
-rw-r--r-- 3 test test 15 May  1 22:01 file1
```

Examples II

To change permissions, `chmod` is used:

```
$ ls -l file1
-rw-r--r-- 3 test test 15 May  1 22:01 file1
$ chmod u=rwx,g+wx,o-r,o+x file1
$ ls -l file1
-rwxrwx--x 3 test test 15 May  1 22:01 file1
```

Examples III

Permissions can also be given as octal number (`r`=4, `w`=2 and `x`=1):

```
$ chmod 644 file1
$ ls -l file1
-rw-r--r-- 3 test test 15 May  1 22:01 file1
$ chmod 771 file1
$ ls -l file1
-rwxrwx--x 3 test test 15 May  1 22:01 file1
```

☞ See the corresponding man pages for further details.

Appendix

Bibliography I

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.