File system

- Non–volatile long–term storage is needed.
- A computer system must have a semi–infinite storage capacity.
- Concurrent access to long–term storage is a must.
- The data stored in long–term storage must be identifiable in some mnemonic fashion. It must be possible to group data stored in long–term storage according to some grouping rules (by project, by owner, by date, etc.).
- Data kept in long–term storage must be reasonably protected from malicious damage (accidental damage is a different thing).

Anyone who agrees with most of the above will see a need for the **file system** as an integral part of any OS.

File

The basic unit of data kept in long–term storage is a **file**. The set of all the files plus the utilities provided for file manipulation form the **file system**.

An OS may have one or (more likely) more than one file system. File systems that coexist inside one OS usually are incompatible and reside on different storage media. What makes them file systems is that they allow to treat their components as files (examples: /dev, /cdrom, nfs).

The main property of a file is its **name** which must be unique in any specific context.

Naming

A file name is not defined in exactly the same term by every OS:

Extension: Microsoft systems consider the file extension to be meaningful. A specific program is associated with each "registered" extension and only that program will touch a file with the extension. By contrast, **UNIX** treats the extension as a user convention which has no binding meaning (it has a lot to do with the point–and–click vs. command–line interfaces).

Special characters: systems differ in the range of allowed characters in a file name. Some go to extreme (**NTFS** allows any Unicode character in a name) while other limit the name to letters (ignoring the case).

Version: some systems performed an implicit backup by keeping obsoleted versions of files, so a file could be identified by its name and version (always relative to the current version #1).

File structure

While a file is an indivisible entity for file management purposes, it is made of smaller sub–units, which could be:

Fixed–length records: a file is made of blocks that are identified by a sequence number.

Variable–length records: a file is made of blocks that are identified by a key or, in more elaborate cases, a combination of keys.

A structure of records such as a **tree** where variable–length records are linked together by pointers representing some relevant relationship, such as parent–child–sibling.

Underneath every structure comes a raw representation of a file as a **stream of bytes** (could be viewed as fixed–size records of size 1). We will ignore all file types other than **byte streams**.

The file system

The original purpose of a file system was to provide a utility for manipulating data stored in files.

This utility gave the ability to:

- Create/delete a file including giving the file a **unique** name and subsequently changing it.
- Protect a file from unsolicited use.
- Reading/writing/repositioning/truncating a file.
- Grouping files by theme (the origin of directories).

Clever programmers quickly saw the potential of using the file system for other purposes, mainly as a registry for unique names and for incorporating the never ending procession of new devices into an existing OS. As a result, a contemporary file system is much more than what the name implies.

File types

Files can represent a large variety of different objects disguised as files for the purpose of naming.

Regular file: is exactly that.

Directory file: is a table containing information about files (of any type).

Symbolic link: is a alternative entry for another file in the file system.

Character special files: **slow** devices pretending to be files so that a programmer can use standard i/o operations to access them.

Block special files: **fast** devices pretending to be files.

Sockets: pseudo–files for network communication.

Swap file: a disk partition used for virtual memory support.

Other possibilities: consider **/dev/random**

Character device files

Originally there was a terminal made of a keyboard and a 24x80 alphanumeric screen. Reading and writing was not buffered (although the
n character played a special role).

The slow devices are typically pretending to be character special (device) files. This allows to use character–oriented system calls (getchar(), putchar(), etc.) to be used.

The devices do not have to be slow, but this type of interface triggers an interrupt for every character (or for every end–of–line) or mouse click (etc.), so the overhead of character–oriented i/o is enormous.

Block device files

When fast input–output operations are needed, data transfer has to be done in (large) chunks of data, called **blocks**.

An interrupt is triggered when the transfer of a whole block is finished, reducing the system overhead proportionally to the size of a block.

Different devices have different block sizes, with 512B to 8192B being the norm. while actual input–output operations moved whole blocks of data, this is normally hidden from the applications by **buffering**. Character i/o can thus be used, albeit it could make an application horribly inefficient.

File contents

Files can be **text** files or **binary** files. This distinction means little in practice, since text is just a pre–agreed upon binary representation of something.

The main reason for the text/binary label is to distinguish between two kinds of file contents:

- Free–format files made of lines or paragraphs (forms of records). They could be plain text in ASCII or fairly complex files, such as **.html** or **.doc** Word files.
- Fixed–format files made of units which correspond to imaginary objects of an imaginary class (say, integers are stored as 4–byte quantities, not sequences of digits).

It is important to realise that block devices treat everything they handle as **binary**.

An exec file (a.out)

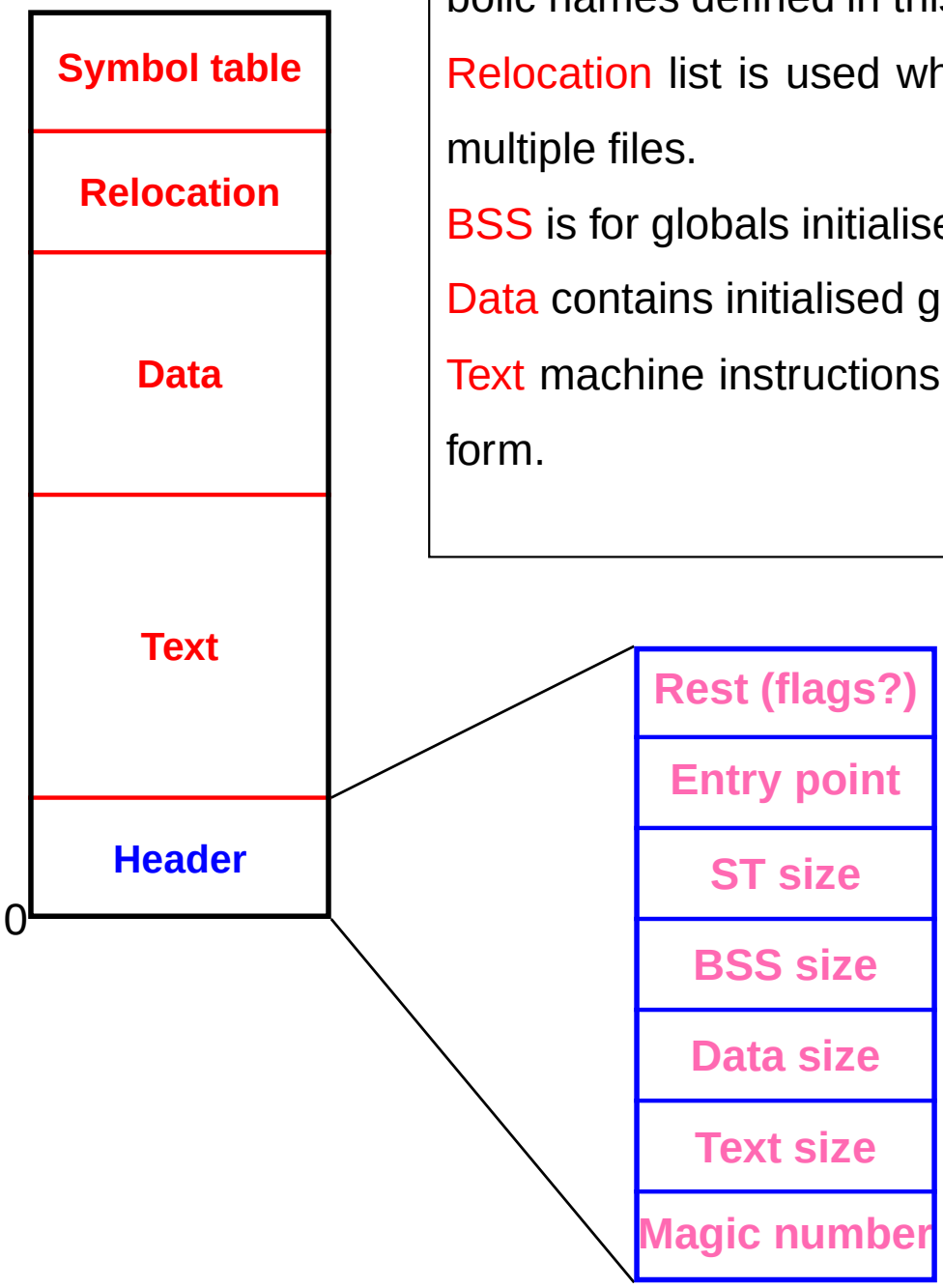
Symbol Table contains the global symbolic names defined in this module.

Relocation list is used when combining multiple files.

BSS is for globals initialised to 0.

Data contains initialised globals.

Text machine instructions in relocatable form.



A simple block copy

```
int id = open( argv[1] , O_RDONLY ) ;
int od = creat( argv[2] , 0600 ) ;
if( id < 0 || od < 0 )
    exit( id+od ) ;
while( (rc = read( id , Buf , BSize )) > 0 )
    if( (wc = write( od , , Buf , rc )) != rc ) {
        printf( "write %d bytes short\n" , rc-wc ) ;
close( id ) ;
close( od ) ;
exit( rc ) ;
```

The file system

A typical OS divides a disk into **partitions**. A partition is usually created statically (using **fdisk**) and is perceived by the OS as a separate storage device.

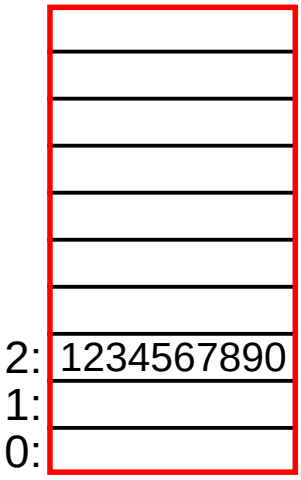
A file system occupies one partition. It is normal that a system has more than one partition, hence more than one file system and that each file system is different from the other.

A file system may look like this:

Each block group is a sequence of disk blocks and looks more or less like this:

I–node

A file is made of a number of disk blocks. The locations of these blocks are stored in a table called **index**. The index can be viewed as an array:



Assuming a 1024B disk block, the correct entry for byte **b** of a file is the table entry

$$\lfloor b/1024 \rfloor$$

and **b** is byte

b % 1024

in the block given in the entry.

The disk block which contains the 2500th byte of the file described by the index above is block 1234567890 (and it is byte 452 of the block).

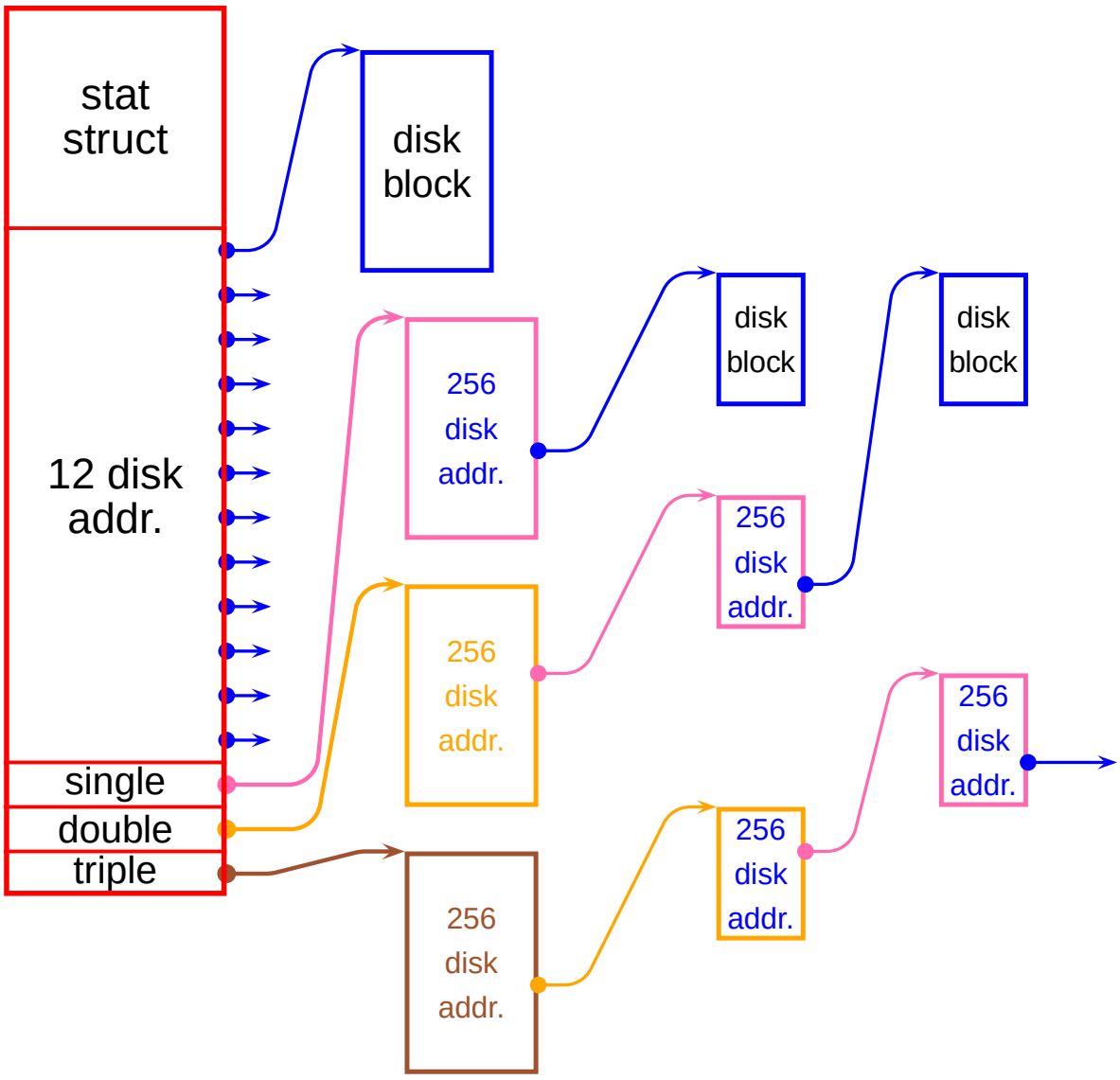
I–node

It is possible to store a complete index for each file but not easy to implement if files grow (they must, since every file starts empty).

An **I–node** (index node is one way to store the index of a file. There is exactly one I–node per file; it is an entry in the I–node table residing in a block group. An I–node must have a fixed format; traditionally, this format used to be:

- A header.
- An index of the first 12 blocks of the file.
- The disk address of a block containing more disk addresses (single indirect).
- The disk address of a block containing addresses of blocks containing disk addresses (double indirect).
- The disk address of a block containing addresses of blocks containing addresses of blocks containing disk addresses (triple indirect).

I–node in detail



How many disk addresses fit into a disk block depends on the size of an address and on the amount of additional information stored. The number shown, 256 is a common case.

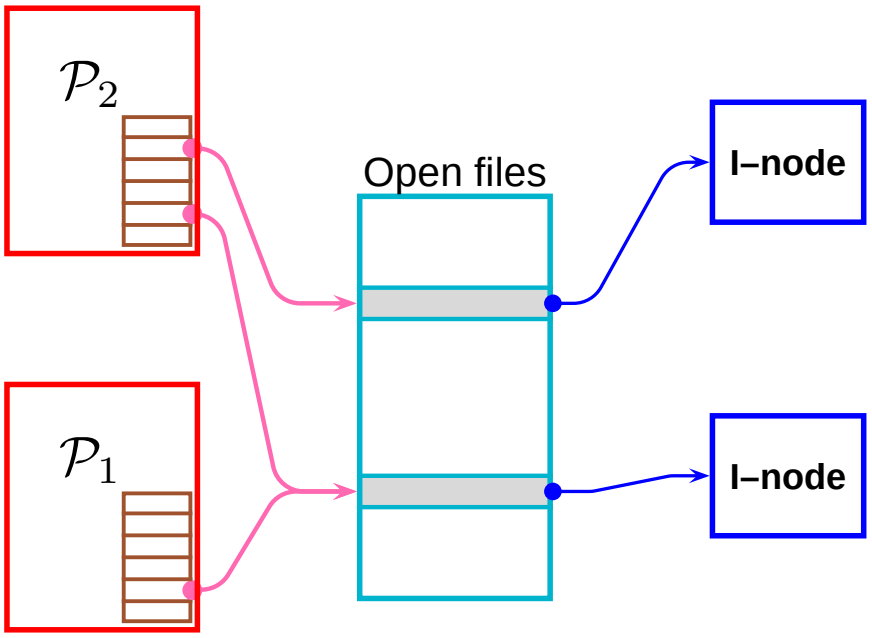
The format of a block group and I–node shown are for the standard linux file system **ext2** (also used in **ext3**).

The format of an I–node is undergoing changes as file systems move from 32–bit disk addresses (good for up to 2TB) to 64–bit disk addresses.

Open files

Processes do not have the ability to access files using disk addresses. They have to use the kernel as intermediary; the kernel also facilitates concurrent access—when several processes access the same file.

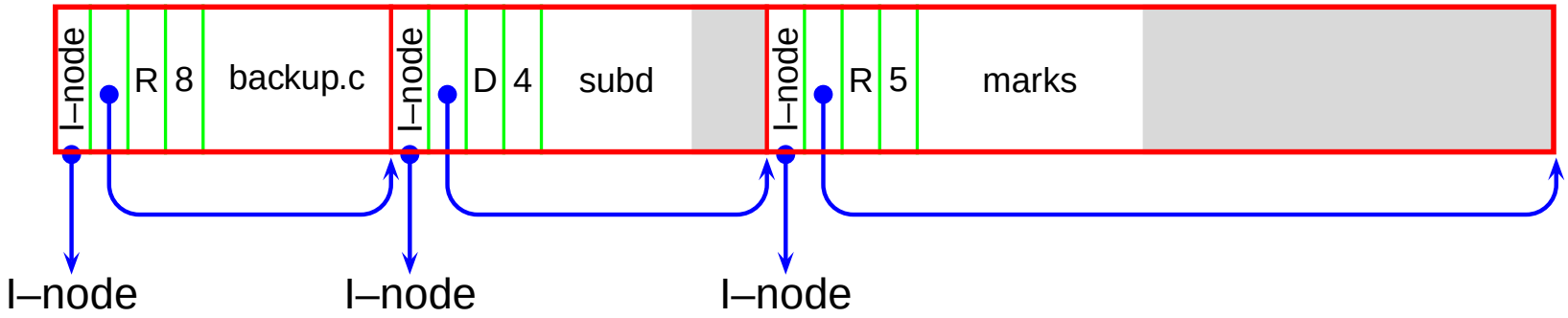
The kernel keeps one global table of open files storing in it the information needed for concurrent access. It also keeps a table of file descriptors for each process.



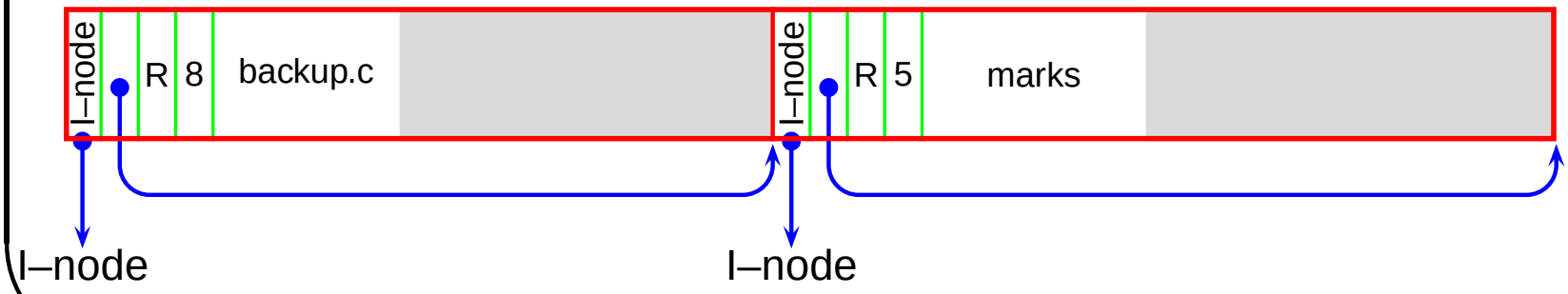
The current **seek** position is stored in the per–process file descriptor table.

Directory

A **directory** is a file that has no contents other than the description of other files–the files residing inside this directory.



After the directory **subd** is deleted:



Regular files

Although contiguous allocation still is used here and there, the accepted approach to storing files is that they are stored on a disk in a way similar to pages in physical memory: the blocks forming a file are scattered across the whole partition (but within one parition only).

The hierarchical approach to storing the file index has its merits (no memory wasted for short files and the file size limit is very generous), but it makes the **append** operation very slow for large files (this is particularly true if a process adds a character at a time to a file that is many megabytes long).

The inefficiency of working with large files is partially solved by using two mechanisms:

Buffer cache: all disk operations are performed on copies of disk blocks (stored in main memory) and not on the directly on the blocks. Periodicaly, the copies are written out to make the disk contents up to date. mamin memory

Journaling: the i/o operations are not performed but just recorded. They are physically performed only when needed or at some point in time.

These two mechanisms result in having two versions of each file: a and up–to–date version stored in main memory and an outdated version stored on disk. The volatile nature of main memory requires that these two versions be **synchronised** from time to time. This is done by the kernel (and by savvy users) by issuing the **sync** command.