Notes on the File System in Unix

Alex Nelson*†

26 August 2009

Contents

List of Algorithms

*Email pqnelson@gmail.com
†Taken from http://code.google.com/p/notebk/

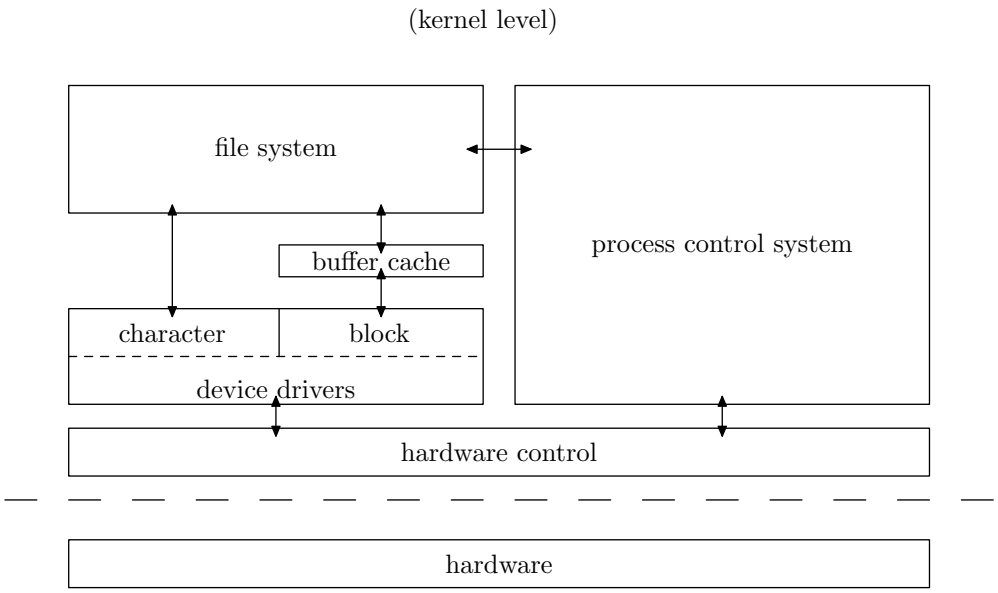

Figure 1: Diagram relating the file system to the other parts of the kernel and hardware.

1 Introduction

This is an introduction to the notion of a file system in a Unix like operating system. We'll try to set up the motivation for the file system and then perform a case study on `xv6` – a modern implementation of Sixth Edition Unix in ANSI C for the x86 architecture (i.e. 32 bit Intel/AMD type processors). It is currently in its second revision [1].

The file system is the heart and soul of the Unix-like operating system. Everything "is-a" file, after all. We will take care in motivating the existence of the file system with several problems.

We will then walk through the basic idea of what the file system does. More precisely, we will discuss its algorithms and data structures, and try to limit our concern to the file system. For the time being we will not delve too deeply into any other part of the operating system.

The second part of this introductory text will be a case study of the `xv6` file system, specifically in relation to the fairly generic algorithms and data structures we considered in the first part of this paper.

Remark. *We will name the algorithms via the English equivalent of the names used for the functions in the* `xv6` *source code. So instead of* `bget()` *(or the ancient equivalent* `getblk()`*) we will consider* GetBlock(). *Also we will make algorithm names used as a step in an algorithm a link. Consider* WriteBlock() *line 5 as an example.*

2 Motivating Problem

Consider the following problems:

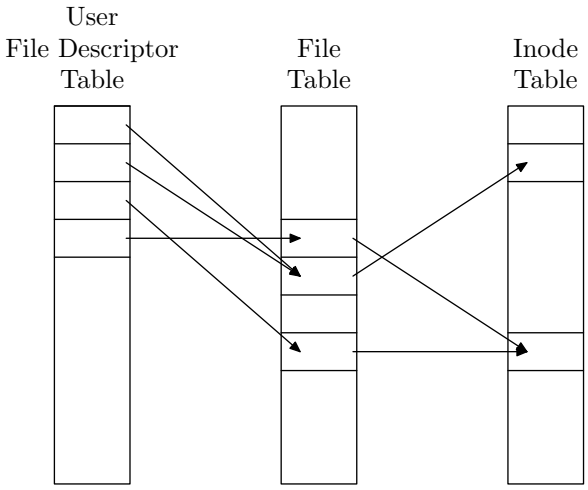

Figure 2: The relationship between the file descriptor table, the file table, and the inode table.

Problem. *We want to keep information after we shut down the computer, and recover it when we turn it back on.*

Problem. *We have the hard drive, which is basically a form of memory (that is, a large but finite collection of bits), we would like to organize the information written to the hard drive "somehow". More generally, we want some sort of interface between the user and the hard disk.*

We want to solve both of these problems at once, in one fell swoop.

2.1 A Note on Terminology

table = array

Note that the term "table" is used often in this text, and should be interpreted as "array" in the context of languages in the C family.

3 Basic Ideas of the File System

We organize information written to the hard drive in the form of some data structure which is typically called a "file". The internal representation of a file is a data structure is really, for unix like systems, an *inode*. The term "inode" is a contraction of the term "*index node*" and is commonly used in most Unix-like operating systems[1]. Every file has one inode (it lives in one place on the hard drive), but it can be given several different paths which point to that one single quantum of hard disk space. These sort of pointers are called a "*(symbolic) link*".

inode
index node
(symbolic) link

When a program (e.g. the user through a shell command, or some other program) refers to the full path, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. E.g. the line of code a process calls

```
open("/usr/local/bin/foobar",1);
```

[1] Most virtual file systems use the phrase as well.

the kernel retrieves the inode for `"/usr/local/bin/foobar"`. It first looks in `/` — the root directory — for the directory `usr`. If the user has the right permissions, the kernel looks in `/usr/` for the `local` directory, and so on until it either can't go further due to a lack of permissions or it finds `foobar`.

When a process creates a new file, the kernel assigns an unused inode to the file. Inodes are stored in the file system, but the kernel reads them into an inode table in-RAM when manipulating the files.

There are two other data structures worthy of note, namely the *file table* and the *user file descriptor table* . The file table is a global variable for the kernel. However, the user file descriptor table is allocated *per process*. Entries in the three structures (the user file descriptor table, the file table, and the inode table) maintain the state of the file and the process' access to it. The file table keeps track of the offset byte in the file where the process' next `read()` or `write()` will start. It is also responsible for the permissions for `open()`-ing the file. So to sum up, the file table:

file table, and user file descriptor table

1. is responsible for permissions;
2. keeps track of the offset byte, where a process will `read()` from or `write()` to next.

The user file descriptor table identifies all open files for a process. (The user file descriptor data structure is a glorified pointer to an entry in the file table really.) If one has experience with programming file input/output in C, the `FILE*` pointer returned by `fopen()` is precisely a file descriptor. The file descriptor is used for `read()` and `write()` system calls. The operating system uses the file descriptor to access the user file descriptor table, which points to the file table which then points to the inode table. From there it finds data in the file.

We can turn our attention now to the hard disk. For simplicity we won't worry about using solid state drives, or USB disks or anything of the sort. A disk is divided up into several *partitions*, which makes it easier to deal with the disk. At a higher level of abstraction, the kernel treats each partition as a "*logical device*" identified by a "*device number*". The conversion between the logical device (file system) addresses and physical device (disk) addresses is performed by the disk driver.

partitions
logical device
device number

A file system typically consists of a sequence of logical blocks (each containing 512, 1024, 2048, or more generally $(2^n)512$ bytes, depending on the implementation). Within a file system, all blocks are the same size. But be warned this is a system dependent size. The block size on your system may be different than the block size on mine. The conventional wisdom is that using large logical blocks increase the effective data transfer rate between disk and memory (because the kernel can transfer more data per disk operation, so fewer operations are needed). Consider the situation when one wants to read in 1024 bytes from a disk in one `read()` operation is faster than reading 512 bytes twice. One may be tempted, therefore, to use as large a block size as possible, but this temptation should be resisted! If the block size is too large, the effective storage capacity may drop.

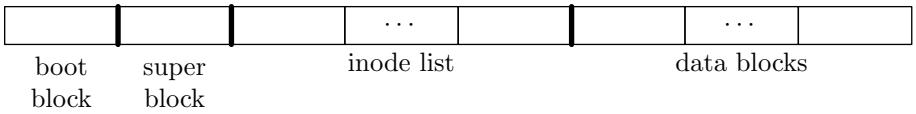

Figure 3: Structure of a disk in terms of blocks

As far as blocks are concerned, a generic file system has the following structure:

1. The **boot block** occupies the beginning of the file system, and its usually the first sector. Its main task is to boot (initialize) the operating system. Every file system has a boot block, although it may be empty. *boot block*
2. The **Super block** describes the state of the file system — viz. how large it is, how many files it can store, where to find the free space on the file system, and so on. *super block*
3. The **Inode List** is a list of inodes that follows the super block in the file system. *inode list*
4. The rest of the disk is the **Data Blocks** which consists of the data of the file. Each data block belongs to exactly one file in the file system. *Data Blocks*

4 Block Cache

Recall the second problem we were considering as motivation for studying the file system was:

Problem. *We have the hard drive, which is basically a form of memory (that is, a large but finite collection of bits), we would like to organize the information written to the hard drive "somehow". More generally, we want some sort of interface between the user and the hard disk.*

We would like to consider how to organize information on mediums like a disk. The basic idea is that when a process wants some information from a disk, the kernel loads the data into RAM where the process can examine it, alter it, and then request the modified data be saved to the disk.

We see in figure 1 that the buffer cache module in the kernel takes its place mediating between the file system and the block device drivers. The kernel tries to read from the block cache. If the data is not in the cache, it loads the desired data into the block cache while trying to save as much good data as possible. Similarly, the kernel writes to the block cache. This design is chosen to minimize the frequency of disk write operations, which are relatively slow.

4.1 Buffer Header

Algorithm 4.A GetBlock() Look through buffer cache for block on a given device

Input: file system number fn
Input: block number bn
Output: locked buffer that can now be used for block

```
1:  procedure GetBlock(fn, bn)
2:      while buffer not found do
3:          if block in hash queue then
4:              if buffer busy then
5:                  Sleep(until buffer becomes free) (then continue to the while loop)
6:              end if
7:              mark buffer busy
8:              remove buffer from free list
9:              return buffer
10:         else          (block not on hash queue)
11:             if there are no buffers on free list then
12:                 Sleep(until any buffer becomes free) (then continue while loop)
13:             end if
14:             remove buffer from free list
15:             if buffer marked for delayed write then
16:                 asynchronous write buffer to disk (then continue to the while loop)
17:             end if
18:             remove buffer from old hash queue
19:             put buffer onto new hash queue
20:             return buffer
21:         end if
22:     end while
23: end procedure
```

Algorithm 4.B ReleaseBlock() release the block buffer

Input: locked buffer
Output: none.

```
1:  procedure ReleaseBlock(b)
2:      if buffer contents valid and buffer not old then
3:          enqueue buffer at the end of the free list
4:      else
5:          enqueue buffer at beginning of free list
6:      end if
7:      lower processor execution level to allow interrupts
8:      Unlock(buffer)
9:  end procedure
```

Algorithm 4.C ReadBlock() reads the block buffer

Input: file system block number bn
Output: buffer containing data

```
1: procedure ReadBlock(b)
2:     GetBlock(file system number, bn)
3:     if buffer contents valid then
4:         return buffer
5:     end if
6:     initiate disk read
7:     Sleep(until disk read complete)
8:     return buffer
9: end procedure
```

Algorithm 4.D WriteBlock() writes the block buffer to disk

Input: buffer b
Output: none

```
1: procedure WriteBlock(b)
2:     initiate disk write
3:     if I/O synchronous then
4:         Sleep(until I/O complete)
5:         ReleaseBlock(b)
6:     else if buffer is marked for delayed write then
7:         mark buffer to be put at head of free list
8:     end if
9: end procedure
```

A Algorithms Syntax

The algorithms notation can be a little daunting at first. The symbols look familiar, its like an Ancient Roman trying to read Italian — the words are familiar, but the ordering is baffling. We will set the standard for algorithm syntax here.

Algorithm A.A Euclid's Algorithm

Euclid's algorithm to find the greatest common divisor of two positive integers m and n. That is, it returns the largest integer that divides both m and n.

1: [Find Remainder] Divide m by n and let r be the remainder (so we have $0 \leq r < n$.)
2: [Is it zero?] If $r = 0$, the algorithm terminates and n is the answer.
3: [Reduce] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step 1. ∎

Each step of the algorithm consists of:
(1) summary in brackets;
(2) a description of some action to be performed;

Each step of the algorithm begins with square brackets which summarizes as briefly as possible what the step does. We can put the algorithm in the form of a flow chart, using the bracketed phrase as the label of the nodes of the flow chart.

After the summarizing phrase comes a description in words and symbols of some *action* to be performed or some decision to be made. Any comments made are included as an explanation of that step, often indicating certain "invariant characteristics" of the variables or the current goals at that step.

Now the arrow ← in step 3 is the important *replacement operation*, sometimes called *assignment* or *substitution*: so "$a \leftarrow b$" means that the value of the variable m is to be replaced by the current value of variable n. An arrow has different meaning than equality: we will not say "Set $m = n$" but perhaps ask "Does $m = n$?" The "=" sign denotes a condition that can be tested, the "←" sign denotes an action that can be performed. In general, "variable ← formula" means that the formula is to be computed using the present values of any variables appearing within it; then the result should replace the (previous) value of the variable on the left hand side.

(3) $a \leftarrow b$ assigns value of b to variable a; and

Order matters with the assignment operator. So "Set $m \leftarrow n$, $n \leftarrow r$" is quite different from "Set $n \leftarrow r$, $m \leftarrow n$". The latter sets m to the current value of n, which is r. The former sets m to the current value of n, and n has its value be replaced by r.

The heavy vertical line "▮" appearing at the end of step 3 is used to indicate the end of an algorithm and the resumption of the text.

(4) Heavy verticle bar ▮ indicates end of algorithm.

One minor confusing point of notation. If we have a procedure defined, say FOO, it will be denoted by small capital letters. It will be called by the notation FOO(), having the open and close parentheses after the method name. If the procedure takes in arguments, they are listed in between the parentheses. **They are not a comment!** If the procedure's algorithm has been defined elsewhere, it will be in red.

References

[1] F. Kaashoek, R. Morris, and R. Cox, "The `XV6` Source Code." Online. http://pdos.csail.mit.edu/6.828/2008/src/xv6-rev2.tar.gz.

[2] R. Russell, D. Quinlan, and C. Yeoh, *Filesystem Hierarchy Standard*. Filesystem Hierarchy Standard Group, 2004. http://www.pathname.com/fhs/.

[3] M. J. Bach, *The design of the UNIX operating system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[4] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: Design and implementation*. Prentice-Hall, Upper Saddle River, NJ, USA, third ed., 2006.

[5] D. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly & Associates, 2003.

[6] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems (TOCS)* **2** (1984) no. 3, 181–197.

[7] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, first ed., 2004.

[8] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Professional, second ed., 1996.

[9] J. Lion, *A commentary on the Sixth Edition UNIX Operating System*. Peer-to-Peer Communications Inc., sixth ed., 1977. http://www.lemis.com/grog/Documentation/Lions/index.php.

[10] S. Kleiman, "Vnodes: An architecture for multiple file system types in Sun UNIX," in *USENIX Conference Proceedings*, pp. 238–247. 1986.

[11] M. McKusick, "The virtual filesystem interface in 4.4 BSD," *Computing Systems* **8** (1995) no. 1, 3–25. http://www.ittc.ku.edu/~niehaus/classes/s96-850/papers/bsd-vfile.ps.

[12] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network filesystem," in *Proceedings of the Summer 1985 USENIX Conference*, pp. 119–130. 1985.

[13] R. Sandberg, "The Sun network file system: Design, implementation and experience," in *in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*. 1986.