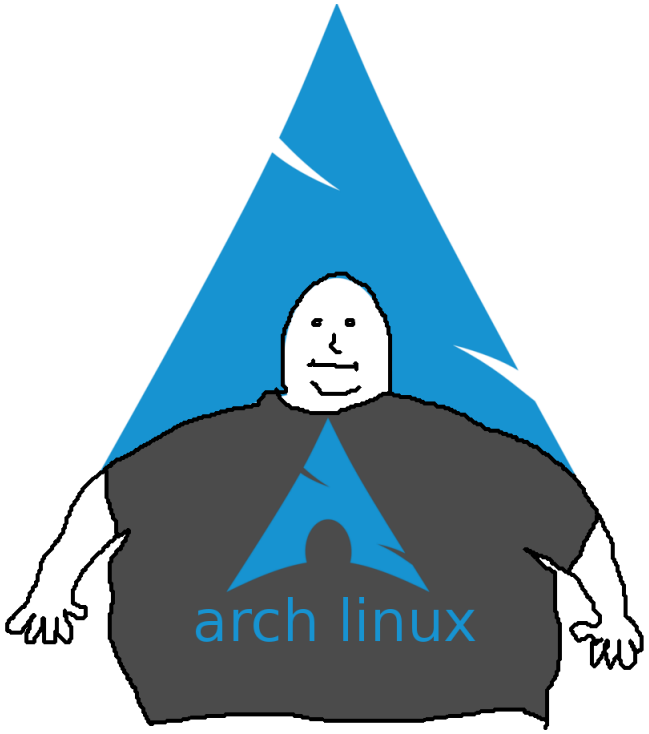
Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Operating Systems

Résumé TE02

Auteur :
Marc Roten

Professeur :
Jacques Supcik



11 janvier 2019

Table des matières

1 Introduction

Résumé pour la deuxième inter d'OS. Spécial dédicace à ma mère, pour la fête des mères.

2 Chapitre_6 La Gestion de la Mémoire

2.1 TLB toussa toussa

TODO

2.2 Page Replacement Algorythm

TODO

3 Chapitre_7 Systèmes de fichiers

DOS, FAT12, FAT16	MS-DOS (1977)
FAT32	Windows 95 OSR 2 (1996)
NTFS	Windows NT 3.1 (1993)
UFS	Unix File System (1983)
EXT2, EXT3, EXT4	Linux (1992, 1993, 2006)
S5FS	System V File System (1969)
ZFS[1]	Open Solaris (2005)
Btrfs	Linux (2009[2])
UBIFS	Unsorted Block Image Filesystem[3] (2008)
F2FS	Samsung, Flash-Friendly File System (2012)

FIGURE 1 – Problèmes avec la RAM

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small. A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process. A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process. Thus, we have three essential requirements for long-term information storage :

- It must be possible to store a very large amount of information.
- The information must survive the termination of the process using it.
- Multiple processes must be able to access the information at once.

for the moment, it is sufficient to think of a disk as a linear sequence of fixed-size blocks and supporting two operations :

- Read Block k
- Write Block k

Questions who pops out at this point in this chapter :

- How do we find the information ?

- How do we avoid user from reading others user's Data ?
- How do we know if a block is free ?

3.1 New Abstraction : the File

The system file used to take care of the files is names **File System**.A file is an abstraction mechanism. It provides a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Windows 95 and Windows 98 both used the MS-DOS file system, called FAT-16, and thus inherit many of its properties, such as how file names are constructed. Windows 98 introduced some extensions to FAT -16, leading to FAT-32, but these two are quite similar. In addition,Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7, and Windows 8 all still support both FAT file systems, which are really obsolete now.

3.1.1 File Name

Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the file extension and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may ev en hav e two or more extensions, as in homepage.html.zip, where .html indicates a Web page in HTML and .zip indicates that the file (homepage.html) has been compressed using the zip program.

3.1.2 File system Examples

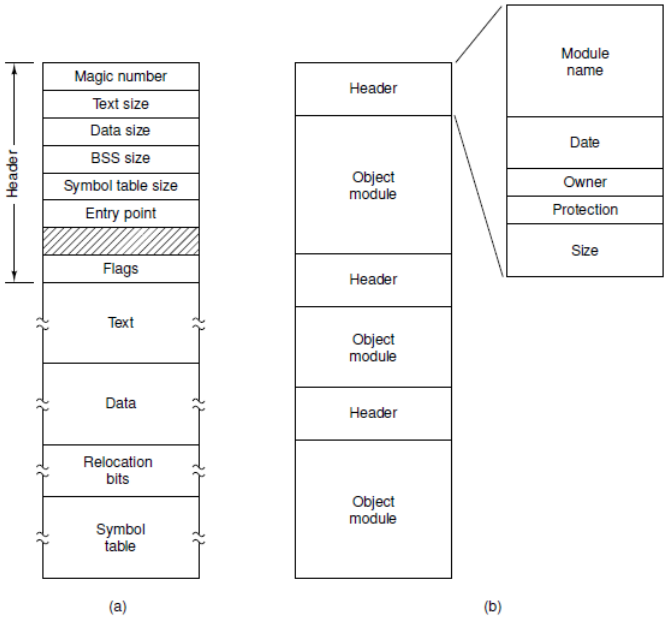


FIGURE 2 – File systems examples

3.1.3 File extensions

Carriage Return	CR	chr(13)	^M	\r	\x0D
Line Feed	LF	chr(10)	^J	\n	\x0A

Mac OS ($\leq$ 9), Apple II	$\rightarrow$	CR
UNIX, Mac OS X	$\rightarrow$	LF
DOS, Windows	$\rightarrow$	CR + LF

FIGURE 3 – Most common extensions

3.1.4 File Structure

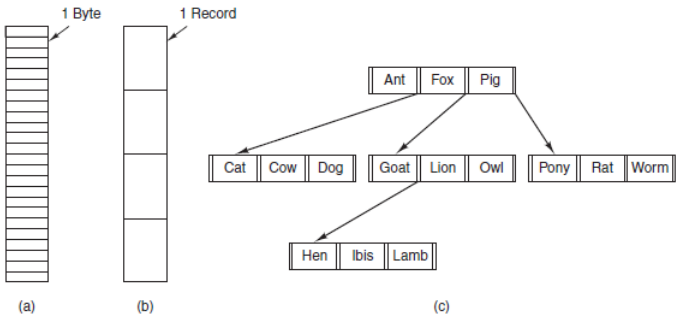


FIGURE 4 – File structure

The figure 4a, is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

The figure 4b, In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.

The figure 4c, The third kind of file structure is shown in Fig. 4-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key.

3.1.5 File type

Fichier Jpeg	`0xFF 0xD8`
PDF	`%PDF`
DOS exécutable	`0x4D 0x5A` («MZ» en ASCII / Mark Zbikowski[4])
MS Office	`0xD0 0xCF 0x11 0xE0` (`D0CF11E0` / DocFile)
ZIP	`0x50 0x4B` («PK» en ASCII / Phil Katz[5])

FIGURE 5 – FileName

Attributs	Signification
Protection	Qui peut accéder au fichier et de quelle manière
Mot de passe	Mot de passe nécessaire pour accéder au fichier
Créateur	Créateur du fichier
Propriétaire	Propriétaire actuel du fichier
Indicateur lecture seule	0 pour la lecture/écriture, 1 pour la lecture seule
Indicateur fichier caché	0 pour un fichier normal, 1 un pour fichier caché
Indicateur fichier système	0 pour un fichier normal, 1 pour un fichier système
Indicateur d'archivage	0 si le fichier a été archivé, 1 s'il doit être archivé
Indicateur fichier ASCII/binaire	0 pour un fichier ASCII, 1 pour un fichier binaire
Indicateur fichier accès aléatoire	0 pour un accès séquentiel, 1 pour un accès aléatoire
Indicateur fichier temporaire	0 pour un fichier normal, 1 pour supprimer le fichier lorsque le processus se termine
Indicateur de verrouillage	0 pour un fichier non verrouillé, 1 pour un fichier verrouillé
Longueur d'enregistrement	Nombre d'octets dans l'enregistrement
Position de la clé	Position de la clé dans chaque enregistrement
Longueur de la clé	Nombre d'octets du champ clé
Date de création	Date et heure de création du fichier
Date du dernier accès	Date et heure du dernier accès au fichier
Date de modification	Date et heure de la dernière modification
Taille courante	Nombre d'octets du fichier
Taille maximale	Taille maximale autorisée pour le fichier

FIGURE 6 – FileName

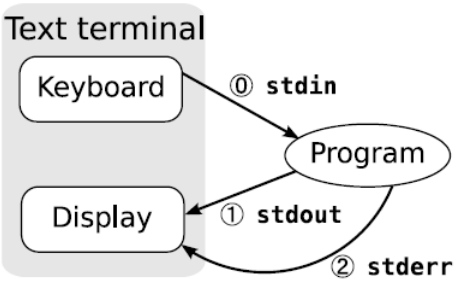


FIGURE 7 – FileName

Définition (Wikipedia)

«En informatique, une **extension de nom de fichier** (ou simplement **extension de fichier**, voire **extension**) est un suffixe ajouté au nom d'un fichier pour identifier son format. Ainsi, on dira qu'un fichier nommé `exemple.txt` a l'extension `txt` ou `.txt`».

- Pour UNIX, l'extension est juste une convention et n'est pas imposée par le système.
- Dans UNIX encore, on peut avoir plusieurs extensions (par exemple : `archive.tar.gz`).
- Pour Windows, les extensions sont associées au programme qui peut traiter les fichiers correspondants.

FIGURE 8 – FileName

3.2 Ordinary file

3.2.1 Text

3.2.2 Binary

3.3 File Access

Early operating systems provided only one kind of file access : sequential access. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position. Files whose bytes or records can be read in any order are called random-access files. They are required by many applications.

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods can be used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position. The latter method is used in UNIX and Windows.

3.4 File Operations

- **Create.** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
- Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
- Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
- Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
- Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
- .Wr ite. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
- Append. This call is a restricted form of wr ite. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
- Seek. For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

- Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

- Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

- Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Extension	Signification
fichier.bak	Fichier de sauvegarde
fichier.c	Fichier source d'un programme C
fichier.gif	Fichier image de format GIF (Graphical Interchange Format)
fichier.hlp	Fichier d'aide
fichier.html	Fichier document en langage HTML (HyperText Markup Language)
fichier.jpg	Fichier image de format standard JPEG
fichier.mp3	Fichier de musique codée en MPEG de niveau 3
fichier.mpg	Fichier de vidéo codée en MPEG
fichier.o	Fichier objet (source compilée, non encore liée)
fichier.pdf	Fichier document au format PDF (Portable Document File)
fichier.ps	Fichier document au format PostScript
fichier.tex	Fichier document au format TEX
fichier.txt	Fichier document au format texte
fichier.zip	Fichier archive compressé

FIGURE 9 – A simple program to copy a file

3.5 File attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's attributes. Some people call them metadata. The list of attributes varies considerably from system to system. The table of Fig. 4-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up recently. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

Attributs	Signification
Protection	Qui peut accéder au fichier et de quelle manière
Mot de passe	Mot de passe nécessaire pour accéder au fichier
Créateur	Créateur du fichier
Propriétaire	Propriétaire actuel du fichier
Indicateur lecture seule	0 pour la lecture/écriture, 1 pour la lecture seule
Indicateur fichier caché	0 pour un fichier normal, 1 un pour fichier caché
Indicateur fichier système	0 pour un fichier normal, 1 pour un fichier système
Indicateur d'archivage	0 si le fichier a été archivé, 1 s'il doit être archivé
Indicateur fichier ASCII/binaire	0 pour un fichier ASCII, 1 pour un fichier binaire
Indicateur fichier accès aléatoire	0 pour un accès séquentiel, 1 pour un accès aléatoire
Indicateur fichier temporaire	0 pour un fichier normal, 1 pour supprimer le fichier lorsque le processus se termine
Indicateur de verrouillage	0 pour un fichier non verrouillé, 1 pour un fichier verrouillé
Longueur d'enregistrement	Nombre d'octets dans l'enregistrement
Position de la clé	Position de la clé dans chaque enregistrement
Longueur de la clé	Nombre d'octets du champ clé
Date de création	Date et heure de création du fichier
Date du dernier accès	Date et heure du dernier accès au fichier
Date de modification	Date et heure de la dernière modification
Taille courante	Nombre d'octets du fichier
Taille maximale	Taille maximale autorisée pour le fichier

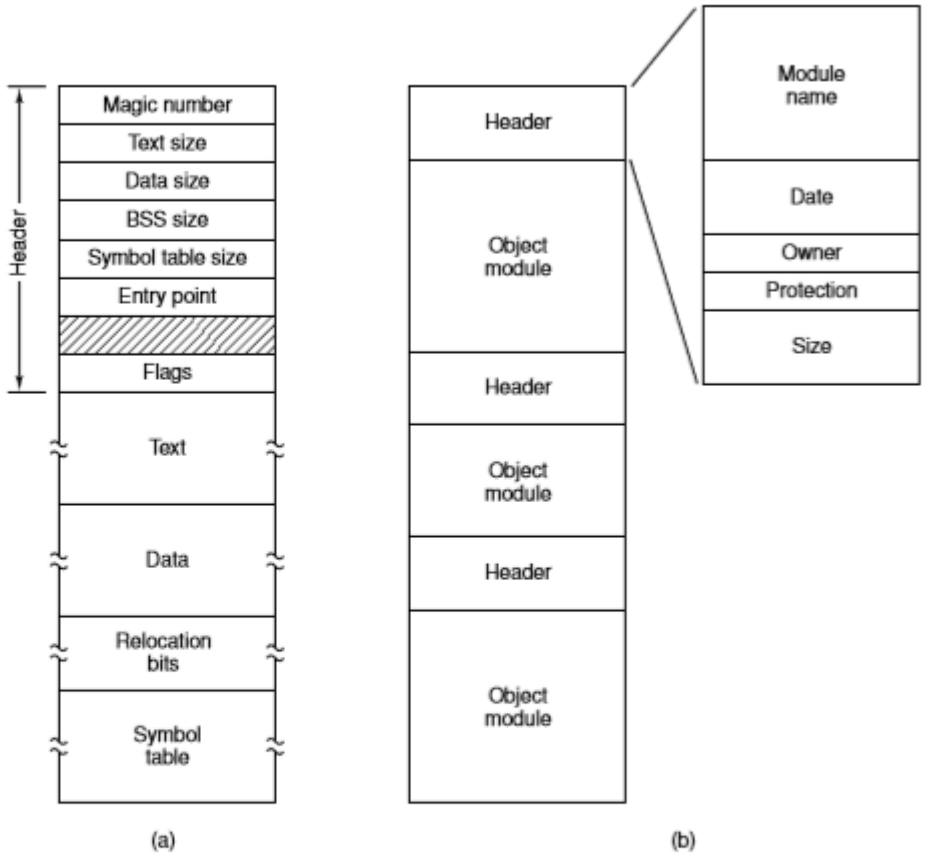
FIGURE 10 – Non exhaustive list of file attributes

3.6 To go further

Many operating systems support several types of files. UNIX (again, including OS X) and Windows, for example, have regular files and directories. UNIX also has character and block special files. Regular files are the ones that contain user information. We will study directories below. Character special files are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks. Block special files are used to model disks. In this chapter we will be primarily interested in regular files. Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., Windows) use both. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.) Other files are binary, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they hav e some internal structure known to programs that use them. For example, in Fig. 11(a) we see a

simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will execute a file only if it has the proper format. It has fiv e sections : header, text, data, relocation bits, and symbol table. The header starts with a so-called magic number, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging. Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish. Every operating system must recognize at least one file type : its own executable file ; some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw whether the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the make program had been built into the shell. The file extensions were mandatory, so it could tell which binary program was derived from which source. Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension .dat (data files). If a user writes a program formatter that reads a .c file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type .dat. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy file.dat to file.c will be rejected by the system as invalid (to protect the user against mistakes). While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they hav e to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.



(a) Un fichier exécutable.

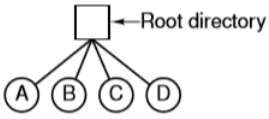
(b) Un fichier d'archive.

FIGURE 11 – a) an executable b) an archive

4 Chapitre_8 Systèmes de Fichiers / Répertoires

4.1 Hierarchical Directory Systems

The single level is adequate for very simple dedicated applications (and was ev en used on the first personal computers), but for modern users with thousands of files, it would be impossible to find anything if all files were in a single directory.



Système de répertoire à un seul niveau contenant quatre fichiers.

- Il n'y a que le «root directory» (pas de sous-répertoire).
- Utilisé par des systèmes tels que le *CDC 6600* ou les premiers PC *(TRS 80)*.

FIGURE 12 – A single-level directory system containing four files.

Consequently, a way is needed to group related files together. A professor, for example, might have a collection of files that together form a book that he is writing, a second collection containing student programs submitted for another course, a third group containing the code of an advanced compiler-writing system he is building, a fourth group containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. What is needed is a hierarchy (i.e., a tree of directories). With this approach, there can be as many directories as are needed to group the files in natural ways. Furthermore, if multiple users share a common file server, as is the case on many company networks, each user can have a private root directory for his or her own hierarchy. This approach is shown in Fig. 4-7. Here, the directories A, B, and C contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

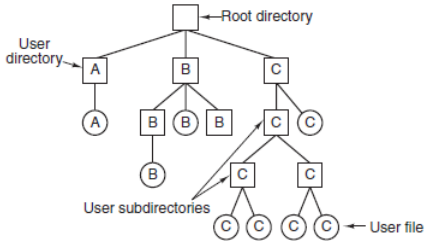


FIGURE 13 – A hierarchical directory system.

4.2 PathName

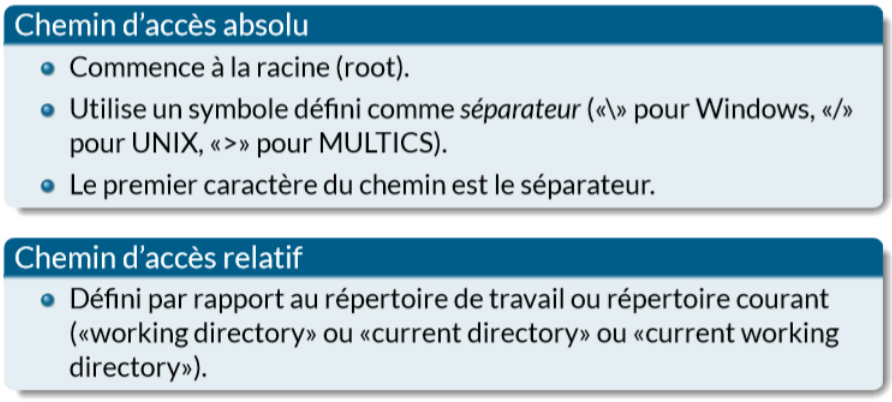


FIGURE 14 – Les chemins d'accès(pathnames)

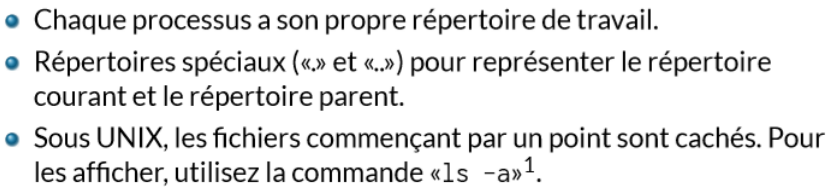


FIGURE 15 – Les chemins d'accès(pathnames)

4.3 Directories Operations

- Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the mkdir program).
- Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.
- Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
- Closedir. When a directory has been read, it should be closed to free up internal table space.
- Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
- Rename. In many respects, directories are just like files and can be renamed the same way files can be.

- Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a hard link.
- Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

4.4 Directory architecture

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR (Master Boot Record)** and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the boot block, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every

partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future. Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. Often the file system will contain some of the items shown in Fig. 16. The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.

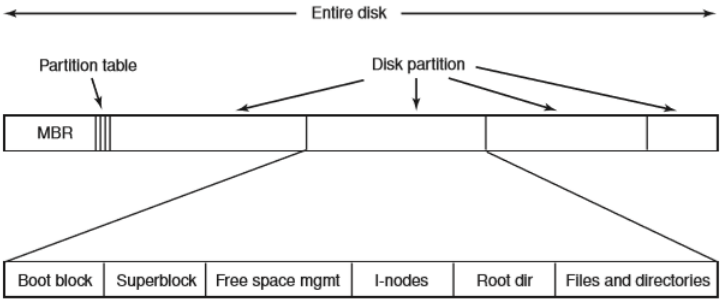
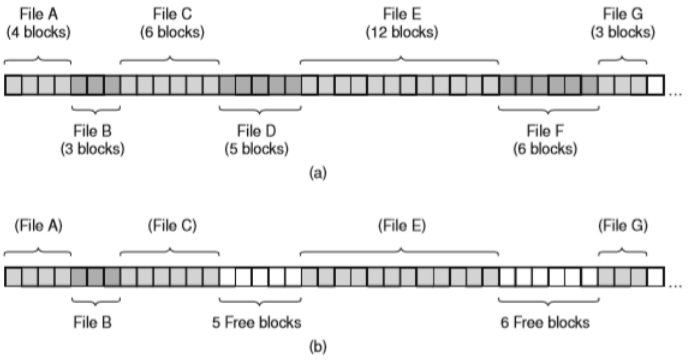


FIGURE 16 – A possible file-system layout.

4.4.1 Contiguous allocation



(a) Allocation contiguë de l'espace disque pour 7 fichiers.
(b) L'état du disque après la suppression des fichiers *D* et *F*.

FIGURE 17 – (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files D and F have been removed.

Avantages are :

- **Ez to inplement :** it is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers : the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
- **Good read performance :** Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed, so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Bad points :

- **Fragmentation :** over the course of time, the disk becomes fragmented. to see it look at fig 17b Here two files, D and F, hav e been removed.When a file is removed, its blocks are naturally freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole, since that would involve copying all the blocks following the hole, potentially millions of blocks, which would take hours or even days with large disks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.
- **Scalability**We don't know exactly before the implementation the exact size needed, but it is an important condition to use this architecture

4.4.2 Linked List

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig 18. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

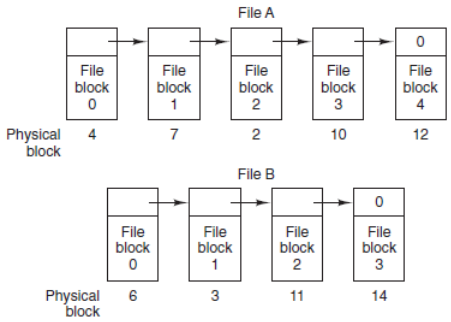


FIGURE 18 – Storing a file as a linked list of disk blocks

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there. On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the n-1 blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow. Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Avantages

- Pas de perte de capacité, pas de gaspillage (pas de «trous»).
- Lecture séquentielle facile.

Désavantages

- Lecture aléatoire difficile («seek»). Pour lire un bloc, n on doit lire les $n-1$ blocs précédents.
- La taille des blocs de données n'est plus une puissance de 2, car le pointeur fait partie du bloc.

FIGURE 19 – Avantages and inconvenients

4.4.3 Linked list with RAM table

Let's consider a linked list as explained previously, but to counter the inconvenients, we will add a little detail. See Fig 20.

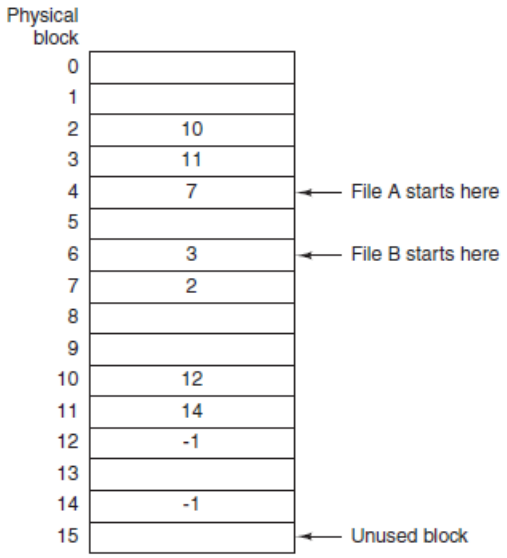


FIGURE 20 – Linked list with RAM table

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 21 shows what the table looks like for the example of Fig. 18. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 21, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker. Such a table in main memory is called a FAT(File Allocation Table).

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks. It was the original MS-DOS file system and is still fully supported by all versions of Windows though.

Avantages

- Accès aléatoire plus performant, car la table est en mémoire.
- Les blocs peuvent à nouveau avoir des tailles avec des puissances de 2.

Désavantages

- La table utilise de la RAM.
- La table doit être régulièrement copiée sur le disque.
- Problème avec les très gros disques.

Ce système est utilisé par MS-DOS (FAT = «File Allocation Table»)

FIGURE 21 – Avantages and inconvenients

4.4.4 I-nodes

————notes de cours———— pour accéder aux niveaux 2-3, celà prend plus de temps. les blocs ne sont pasforcément stockés de manière contigue.

les blocs directs sont stockés en RAM

les blocs indirects, double indirects sont chargé en ram au moment ou il devient utile.

petit exercice :

architecture avec 12 poiinteurs direct et a indirect double et triple indirect Avec des blocs de 2 kByte et des pointeurs sur 32 bit on a une taille maximum pour les Fichiers de :

512 car :**combien de blocs 4Byte dans 2kByte**.

$(12 + 512 + 512^2 + 512^{13}) * 2048 = +- 256GByte$

Avantages

- Seul les tables des **fichiers ouverts** sont en RAM (et non la table du disque entier).

FIGURE 22 – Avantages

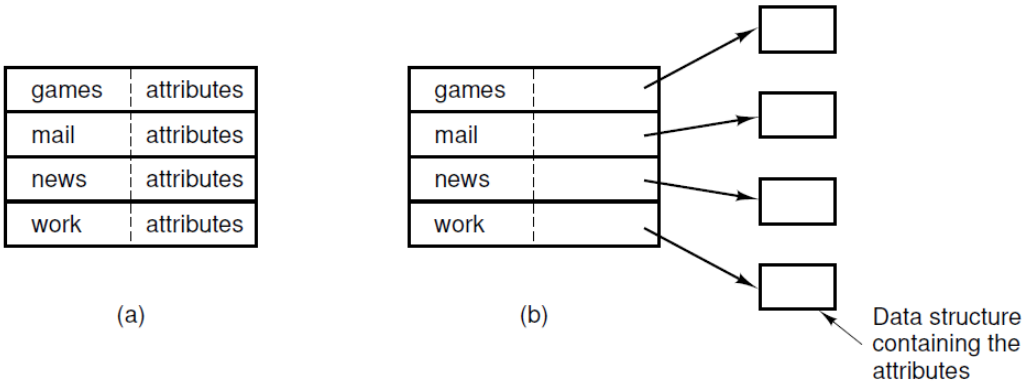
4.5 Directory implementation

Things to stock on each directories's entries.

- FileName
- Adress (on contiguous allocation), or adress of the first inode
- acces restriction, creation date, owner

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked- list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data. A closely related issue is where the attributes should be stored. Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that. This option is shown in Fig. 24. In this simple design, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

Où stocker les attributs?



(a) Répertoire simple contenant des entrées de taille fixe avec les adresses disque et les attributs.

(b) Répertoire dans lequel chaque entrée fait référence à un i-node.

FIGURE 23 – (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

One alternative is to giv e up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 4-15(a) in big-endian format (e.g., SPARC). In this example we have three files, project-budget, personnel, and foo. Each file name is terminated by a special character (usually 0), which is represented in the figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

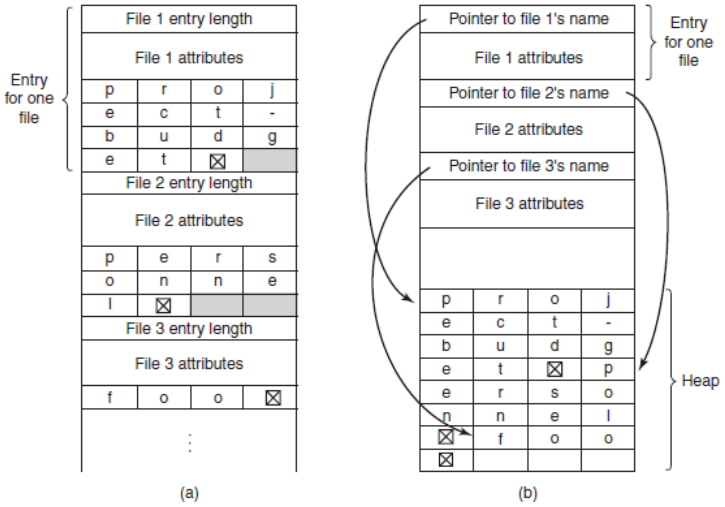


FIGURE 24 – Two ways of handling long file names in a directory. (a) In-line.(b) In a heap.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is essentially the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name. Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 24(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 24(b) as they are in Fig. 24(a). In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and n - 1, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar. Either way, the table entry corresponding to the hash code is inspected. If it is unused, a pointer is

placed there to the file entry. File entries follow the hash table. If that slot is already in use, a linked list is constructed, headed at the table entry and threading through all entries with the same hash value. Looking up a file follows the same procedure. The file name is hashed to select a hash-table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory. Using a hash table has the advantage of much faster lookup, but the disadvantage of more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files. A different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located immediately. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

- En-ligne : la suppression d'un fichier reste assez difficile («trous» de taille variable dans la structure).
- Dans les deux cas, la recherche d'un fichier peut prendre beaucoup de temps (recherche séquentielle).
- On peut améliorer la vitesse de la recherche avec des techniques telles que «hash table», «balanced tree», «cache».

FIGURE 25 – Longueur des noms de fichiers de variables

5 Chapitre_9 Disques / Systèmes de Fichiers

————————————-notes de cours————————————-

6 Conclusion

Si vous avez aimé mon résumé, faites un git clone de mon Git. Suivez moi sur gitlab.forge.heia-fr.ch github and iLoveFreeSoftware.com.