Chapter 8

FILE SYSTEM

Table of contents:

8.1 INTRODUCTION 8-1

8.2 FILE ALLOCATION 8-3

8.2.1 Contiguous Allocation 8-4
8.2.2 Linked Allocation 8-5

8.2.3 Indexed Allocation 8-7
8.2.4 Multilevel Indexed Allocation 8-8
8.2.5 Combined Multilevel Indexed Scheme 8-10
8.2.6 Combined Indexed and Contiguous Allocation 8-11
8.2.7 Free Space Management 8-12

8.3 NAMING 8-14

8.3.1 Hierarchical File Names 8-14
8.3.2 Relative Path Names 8-16
8.3.3 Parent and Current Dierectory Names 8-16
8.3.4 Name Linking 8-17

8.4 DIRECTORIES 8-18

8.4.1 File Attributes 8-18
8.4.2 File Header 8-19
8.4.3 Directory Implementation 8-20
8.4.4 Implementation of Links 8-23

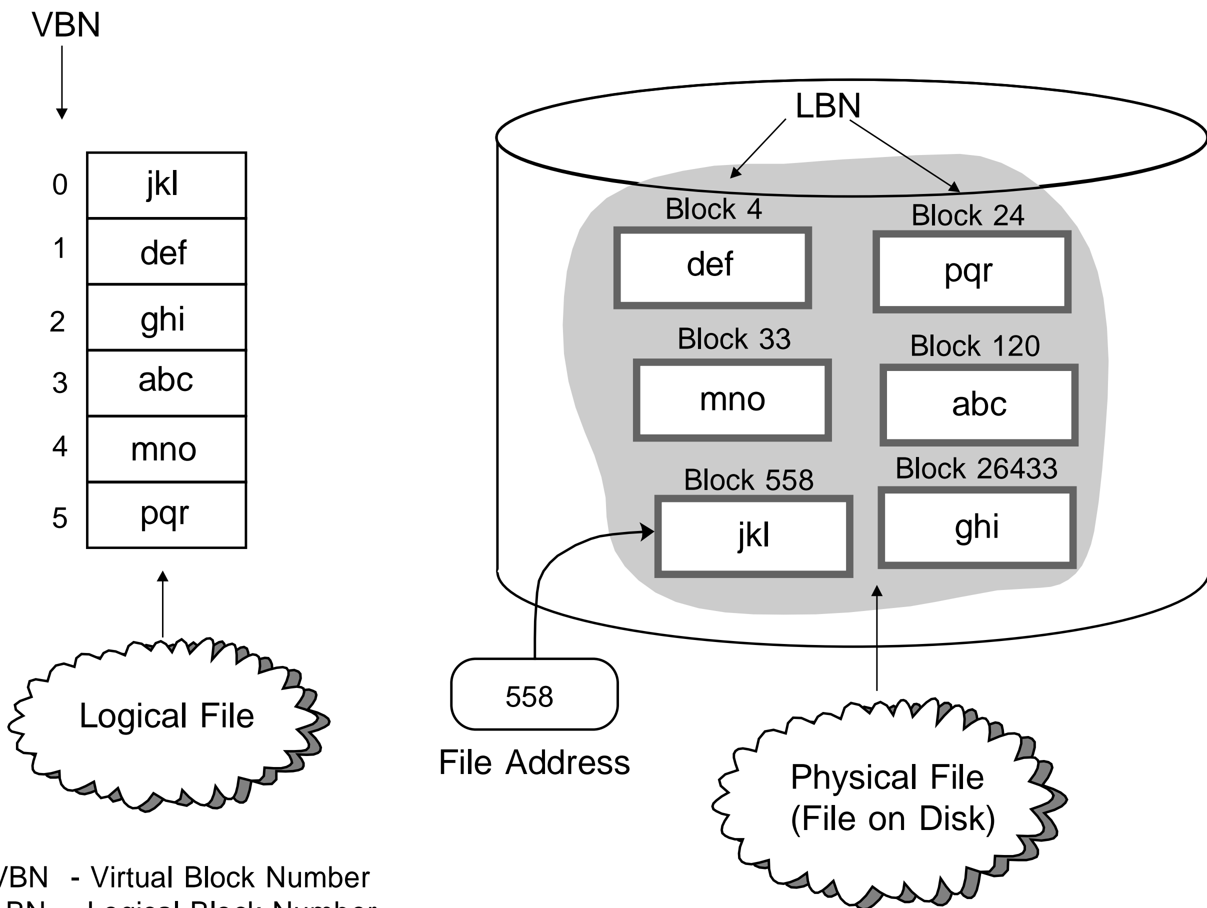
8.5 FILE SYSTEM 8-24

8.5.1 MS-DOS File System 8-24
8.5.2 UNIX File System 8-27

8.6 FILES AND PROCESSES 8-29

Copyright © 1998-2002 by Marko Vuskovic

INTRODUCTION



VBN - Virtual Block Number
LBN - Logical Block Number

User view of a file:

Give me three bytes starting with the 17705-th byte from the beginning of the file "xxx"
.

```
char p[3];                        // My buffer
int fd;                           // File descriptor
fd = open("xxx", O_RDONLY,0);     // Open file "xxx" for reading
lseek(fd,17705,0);                // Position the file to  byte 17705
read(fd,p,3);                     // Get 3 characters into p[].
```

System:

If the block size is 4 KB = 4096 bytes, then 17705/4096 = 4 + 1321/4096,
VBN = 4, block offset = 1321;
Map file name "xxx" and VBN into LBN;
Get the block LBN from the drive and place it into system buffer;
Move characters 1321, 1322, and 1323, from the system buffer into user's buffer p[].

Copyright © 1998-2002 by Marko Vuskovic

INTRODUCTION (Cont.)

Problems:

Problem	Area
(1) How to organize collection of disk blocks into a file? (2) How to map VBN into LBN? (3) How to manage free disk space?	**FILE ALLOCATION**
(4) How to map file names into file addresses?	**FILE DIRECTORIES**
(5) How to glue solutions for (1),(2),(3) and (4) together?	**FILE SYSTEM**
(6) How to make files available to processes and threads?	**FILE DESCRIPTORS**
(7) What can user do with files and how?	**SYSTEM CALLS**

Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION

File allocation must support creation of new files, deletion of existing files and accessing of files (writing, reading and modifying data in files).

There are two basic file accessing methods:

Sequential Access

Data are transferred from/to file as a consecutive sequence of logical file units (bytes or records). This is the most common method (loading images, loading memory pages, editors, compilers, copying files, displaying directories,...).

Direct Access (Relative Access)

A particular byte or record is read (it is specified by its logical sequence number, i.e. by its logical file offset).

Other access methods (for example index sequential access) are not very common and they are usually built on top of the file system (like DBMS - Data Base Management System).

File allocation approaches have to support both file access methods and they are evaluated through both of them.

There are several approaches for file allocation:

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation
- Combined Multilevel Indexed Allocation
- Combined Indexed and Contiguous Allocation

Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

Contiguous Allocation

Blocks on disk are contiguous - VBN is mapped directly onto LBN (+ base LBN). The base LBN is the file address. Each file has its own file address.

Advantages:

Good performance, high transfer rate, minimal number of disk seeks (head does only cylinder-to-cylinder moves)

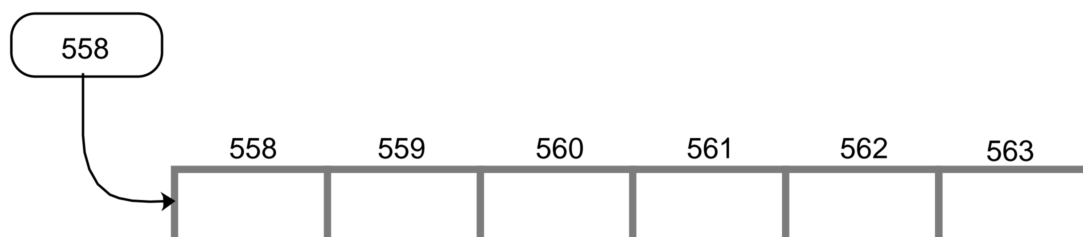
Simple VBN to LBN mapping. Does not require special data structures and consequently multiple file access.

Disadvantages:

Difficult to grow files (if a file becomes bigger, the new bigger file has to be created, and the old file has to be copied into the new file.)

If only appendages are planned, a conservative file size has to be specified at the file creation time. This introduces internal fragmentation.

Due to different file sizes file management is faced with the dynamic allocation problem (external fragmentation and compaction).



Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

Linked Allocation

Physical blocks are linked together (each block has the LBN of the next block, last block has NIL.) The base LBN is the file address.

Advantages:

Files can grow dynamically (no need to know the file size in advance).

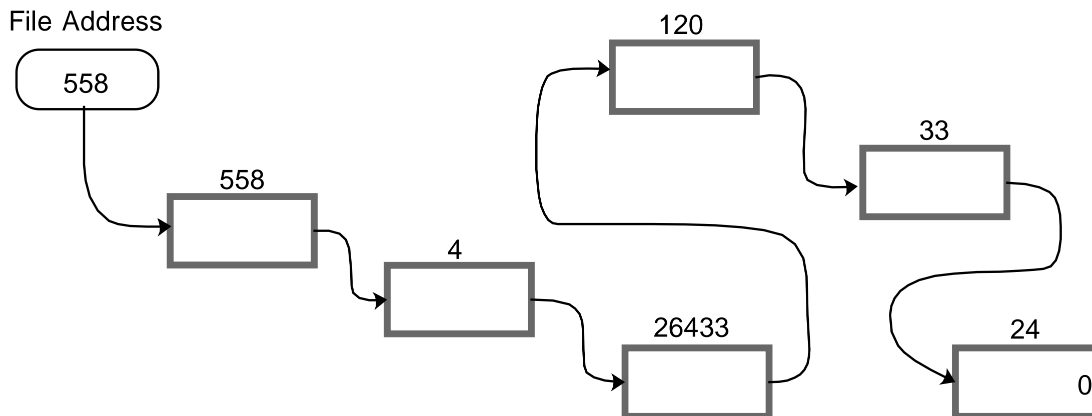
No external fragmentation, internal fragmentation same as in contiguous allocation.

Disadvantages:

Sequential access slower: each new block requires a new seek.

Random access very slow: in order to find a particular block in the linkage, all previous block have to be transferred from the disk, and its next block pointer (LBN) has to be read. This takes many full seeks and unnecessary data transfers/buffering.

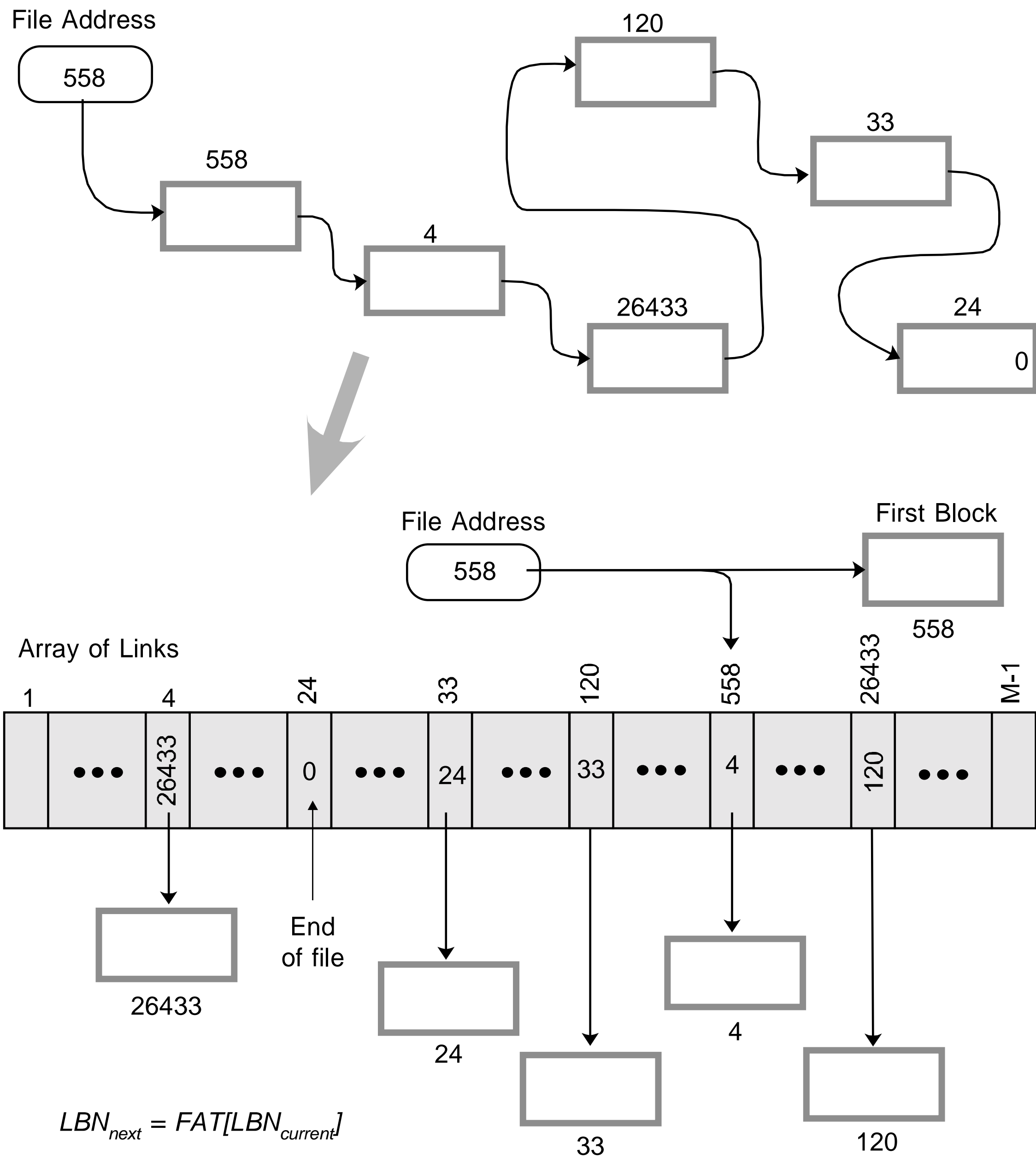
Bad reliability: one lost block - the rest of the file is lost.



Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

The linked allocation can be improved if the links (pointers to next block) are extracted into a separate array of links. The size of the array is equal to the total number of blocks on disk. The array can be kept in blocks which can be cached. This would reduce number of seeks in case of random access. This allocation method is used in MS-DOS (FAT - File Allocation Table).



Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

Indexed Allocation

An array of LBNs is kept in a physical block (the array fits the entire block). This block is called Index Block. The index block is initially filled with NILs, later it gets LBNs. The file header points to the index block.

Advantages:

Can easily grow (however maximal space is limited by the block size)

Fast random access (two seeks)

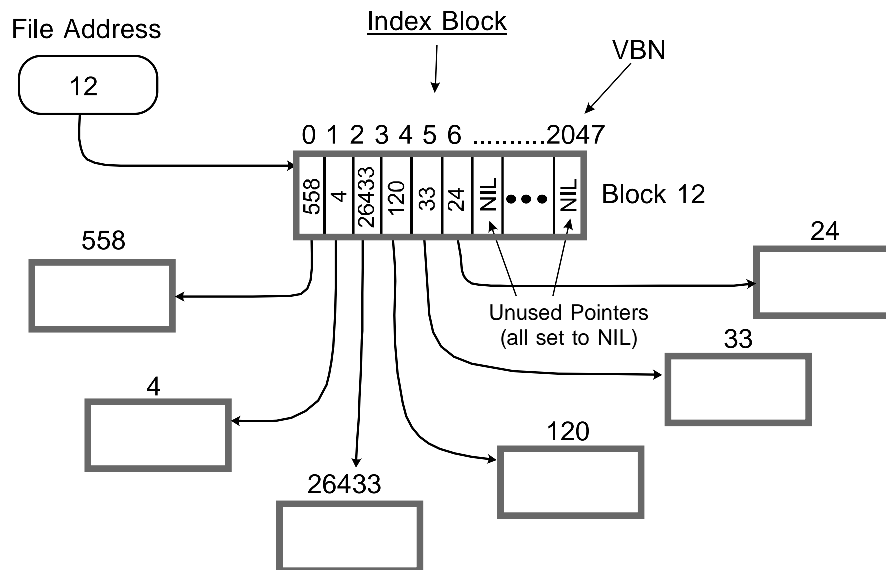
No external fragmentation.

Disadvantages:

Sequential access inefficient, but not as bad as linked allocation (requires a seek per each block).

Problem if the file size exceeds the index block.

Space overhead (NIL entries in index block are unused).



Example:

Block size is 4 KB; each entry has 2 bytes

-> Index block has $2^{12}/2$ = 2 K entries (LBNs), i.e it can point to 2048 blocks.

-> Maximal length of a file is 2 K * 4 KB = $2^{11+12} = 2^{23}$ bytes = 8 MB

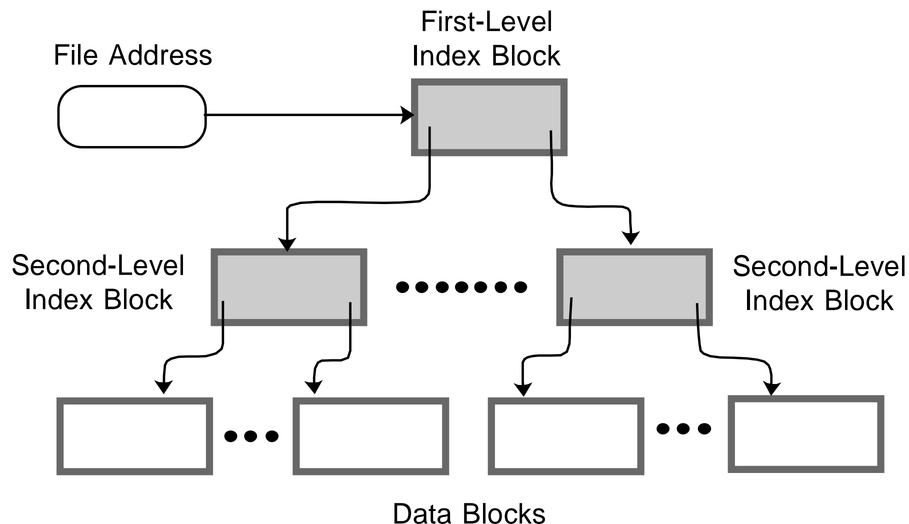
Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

Multilevel Indexed Allocation

The single-level indexed file allocation is conceptually similar to single-level paging in memory management: the index block like a page table does mapping of VBN (the index) into LBN (the entry value).

In order to increase the range of addressable blocks by using index blocks we construct two-level index blocks. The LBNs of the first-level index block will point to the second-level index blocks, which in turn have LBNs that point to data blocks. Apparently the size of index block entries must be greater than 2 bytes (4 bytes used in UNIX).



Example:

Block size is 4 KB, index entry is 4 bytes.
-> An index block has 1024 entries.
-> Total number of index blocks which point to data blocks: 1024.
-> Total number of data blocks: 1024 x 1024 = 1 M
-> Maximal file size: 1 M x 4 KB = 4 GB

How many disk accesses is required to access a random block?

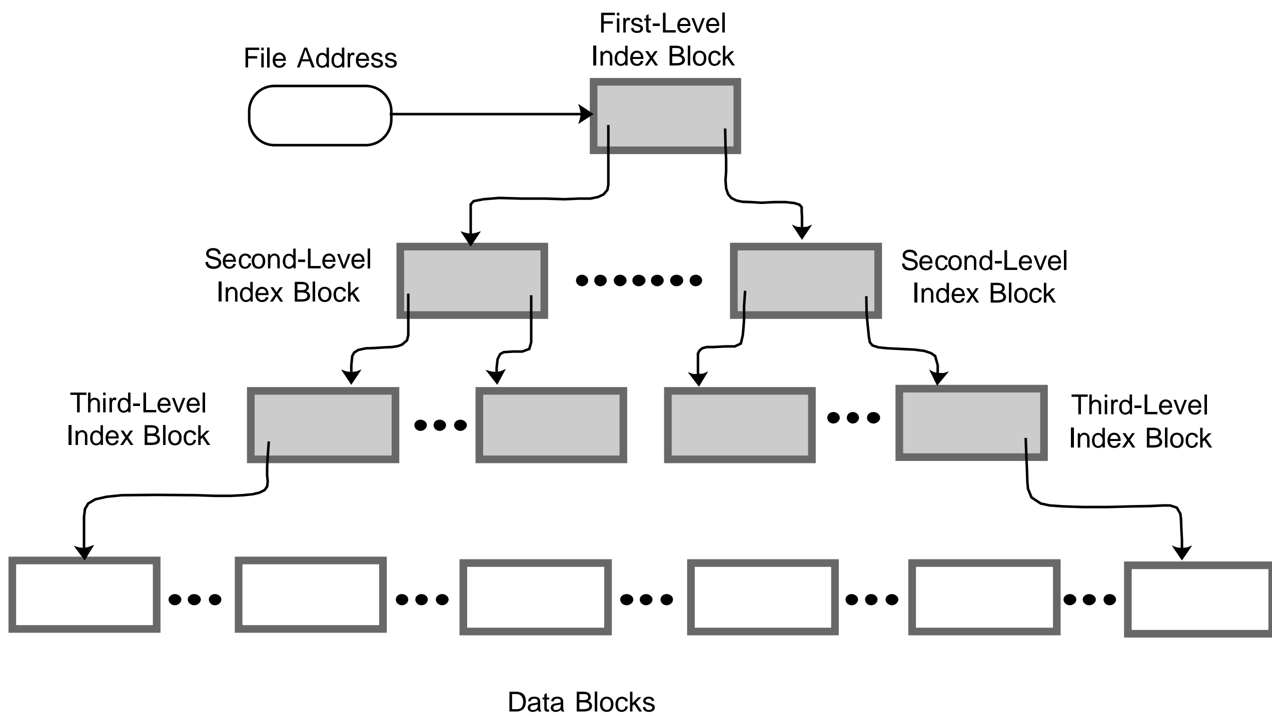
-> One access to get the first-level index block, one access to get the second-level index block, and one access to get the data block, total of three accesses.

Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

What if size of a file grows beyond 4 GB (multimedia, web)?

Then a three-level index blocks have to be used.



Example:

Block size is 4 KB, index entry is 4 bytes.
-> An index block has 1024 entries.
-> Total number of index blocks which point to data blocks: 1024 x 1024 = 1 M.
-> Total number of data blocks: 1024 x 1024 x 1024 = 1 G
-> Maximal file size: 1 G x 4 KB = 4096 GB = 4 TB
(hope will hold for a couple of years!)
-> Four disk accesses required to get a data block.

Copyright © 1998-2002 by Marko Vuskovic

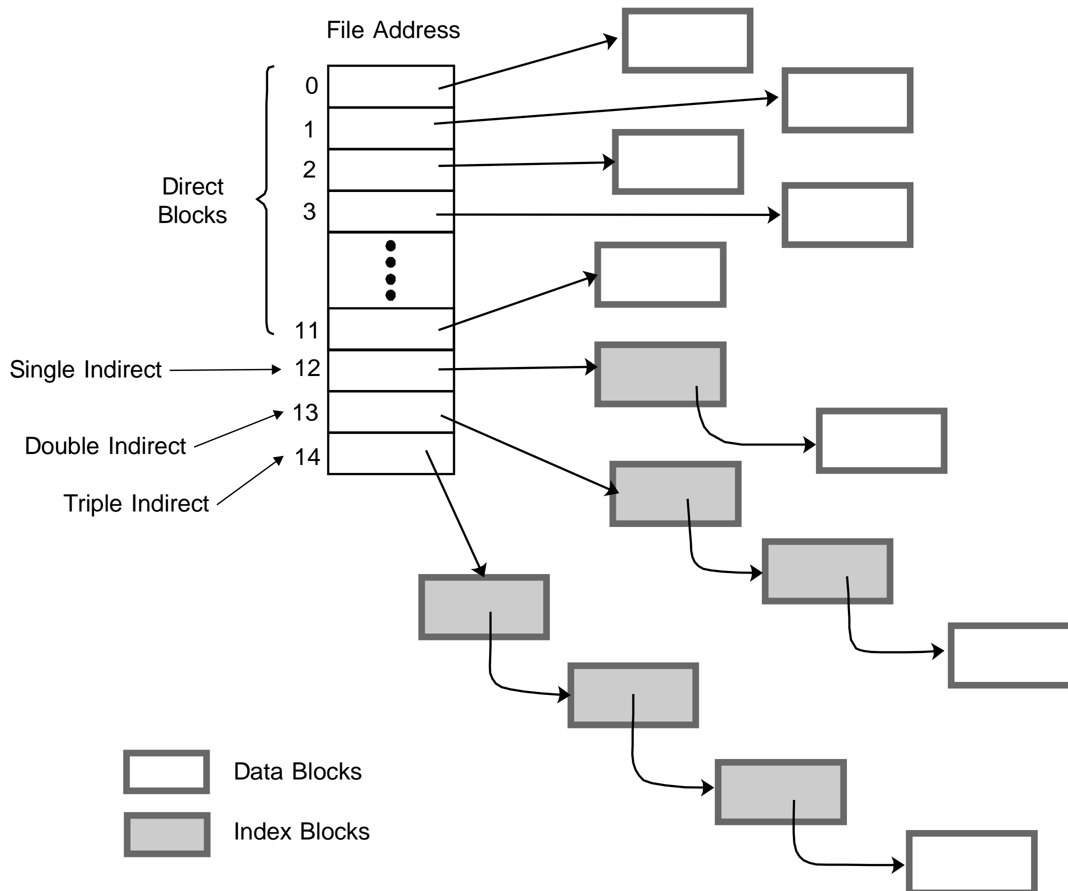
FILE ALLOCATION (Cont.)

Combined Multilevel Indexed Scheme

Multilevel indexing introduces a substantial overhead in disk space and in disk access time. Systems usually have a large number of small files that do not need multilevel indexing, or any block-based indexing at all. Therefore the schemes can be combined.

UNIX family of operating systems uses file addresses with several entries: first 12 entries (in some systems 10 entries) contain LBN of data blocks. Rest of the entries contain LBN of first-level index blocks for single level, double level and triple level indexing.

Very small files (up to 4 KB) take only one data block, i.e. one entry of the file address. Rest of the entries would be filled with NIL. Files up to 8 KB need two entries etc. Files that need more than 12 blocks (48 KB) have to use indirect indexed schemes based on index blocks.



Copyright © 1998-2002 by Marko Vuskovic

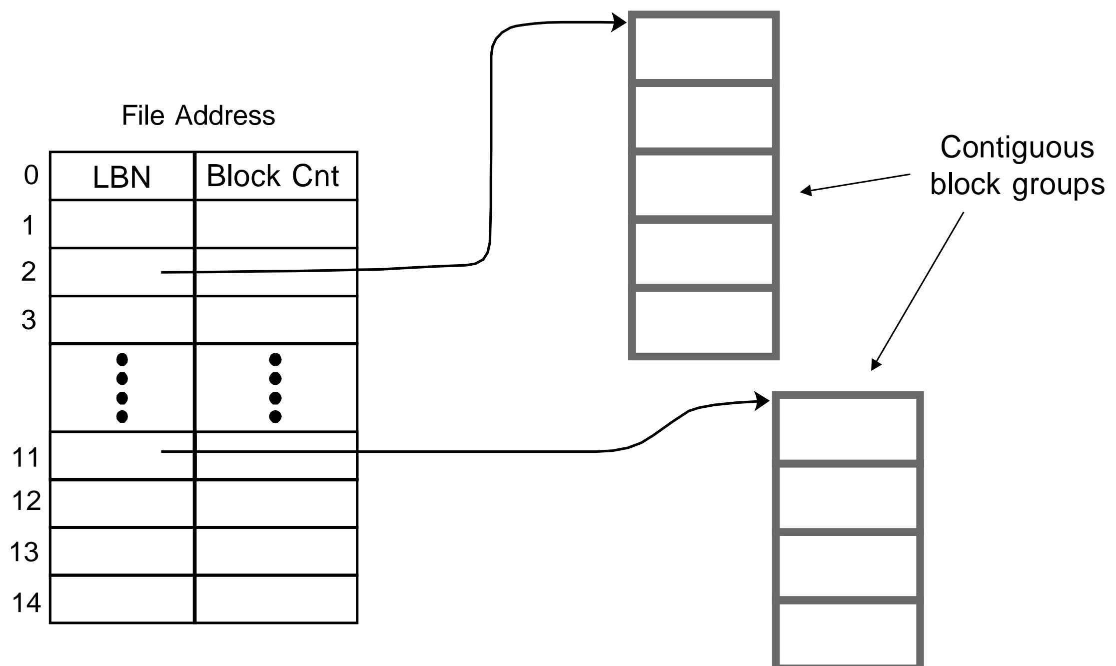
FILE ALLOCATION (Cont.)

Combined Indexed and Contiguous Allocation

Indexed and multilevel indexed allocation is efficient for random access but is not efficient for sequential file access as is the contiguous allocation. Therefore, the two approaches can be combined: an index entry (which normally has the LBN) gets a block count which defines the number of consecutive blocks that follow the first block pointed to by the LBN. Consecutive blocks will reduce seek time in sequential access.

If there is more space needed, the idea can be extended to multilevel indexing.

This approach is used in BSD 4.2 UNIX.



Copyright © 1998-2002 by Marko Vuskovic

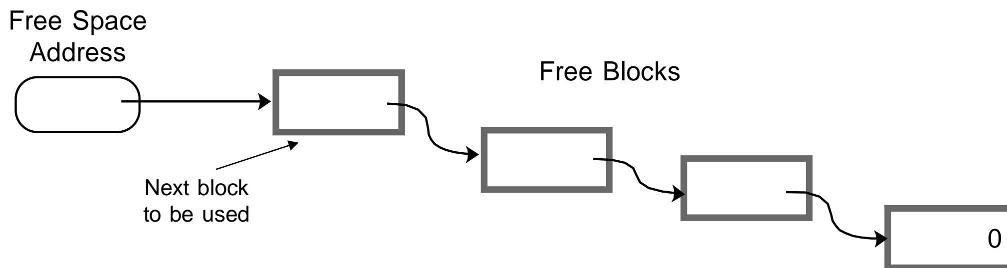
FILE ALLOCATION (Cont.)

Free Space Management

How to find free blocks for data and/or index blocks?
There are three common approaches:

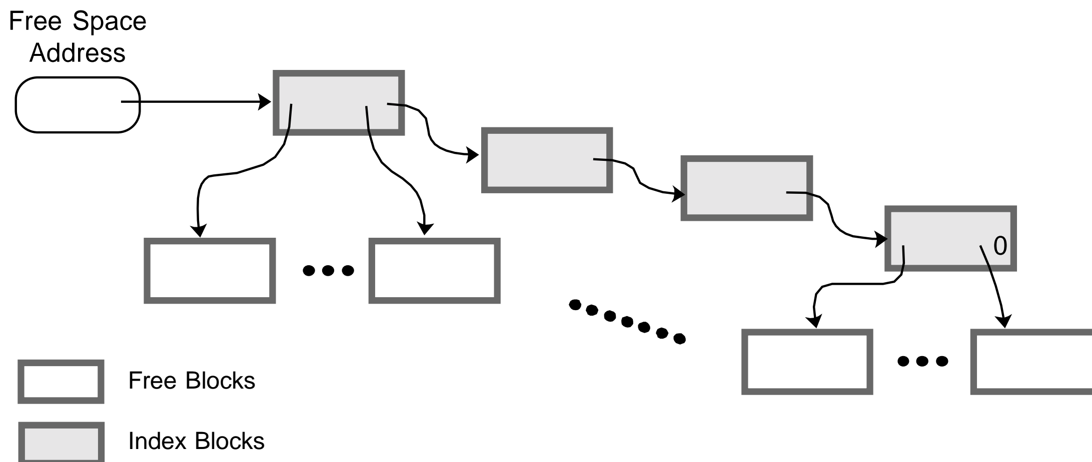
Linked List of Free Blocks

Free blocks are linked together. They are taken/put from/to the front of the list. Simple but inefficient: suppose the first block is allocated to a file - need to acces that block in order to get pointer to the next free block to update the free space address.



Indexing and Linking

Pointers to free blocks (LBNs) are kept in index blocks. Index blocks are linked together. A large number of free blocks can be found with only one disk access. This is specially efficient if one or more index blocks are cached.
One index block has 4 K/4 -1 = 1023 LBNs of free blocks, the last entry is LBN of the next index block. (Used by UNIX.)



Copyright © 1998-2002 by Marko Vuskovic

FILE ALLOCATION (Cont.)

Bit Map (Bit Vector)

Each block on a disk is assigned a bit (1 - free block, 0 - allocated block)

For 8 GB disk with 4 KB blocks the bit map takes $2^{33}/2^{12+3} = 2^{18}$ bytes = 256 KB of internal memory, or 256/4 = 64 consecutive blocks to hold the entire bit map. (Used by Macintosh OS).

Advantage: n consecutive free blocks can be easily found just by string manipulation: look for n consecutive 1s in the bit map. This is used in combined indexed and contiguous allocation.

Copyright © 1998-2002 by Marko Vuskovic

NAMING

So far the structure of a single file has been discussed. Regardless of the implementation of the file structure and the file allocation, the starting point is always the file address which contains the LBN of the first data or index block.

How we manage situation when several (many, or thousands) of files are in the system? In addition, how can we assign and maintain the file attributes (type, size, access rights, time of last update, etc.)? How about the file sharing (among different processes)?

Hierarchical File Names

A common method to handle the complexity of a large number of files is naming. Each file is given a symbolic name. Name is an ASCII string, determined by the user, which has different maximal lengths in different OS:

MS-DOS: 8+3 characters (name + file extension)
Windows NT: 255+3 characters
UNIX BSD 4.1: 14 characters
BSD 4.2, Solaris: 255 characters

In order to systemize the files, they are given several names, which are hierarchically structured. (Similarly is with humans: they (usually) have two names: family name and personal name. This kind of naming is also used in biology, and is called "binary nomenclature". Files use more complex n-nary nomenclature which has an arbitrary number of hierarchical levels. Examples:

UNIX: `/math/faculty/smith/courses/570/programs/a3/p3.c`
WNT: `C:\users\cs570\smith\programs\a3\p3.cpp`

These names can be represented graphically as trees (see next page).

The last name in a hierarchical name, i.e. the leaf node of the name tree is usually called file name, while the full string which starts with the root name and ends with the file name is called path name. Names which are non-leaf nodes of the name tree are called directory names (for short directories). The term "directory" comes from the implementation of the file naming (*see later*).

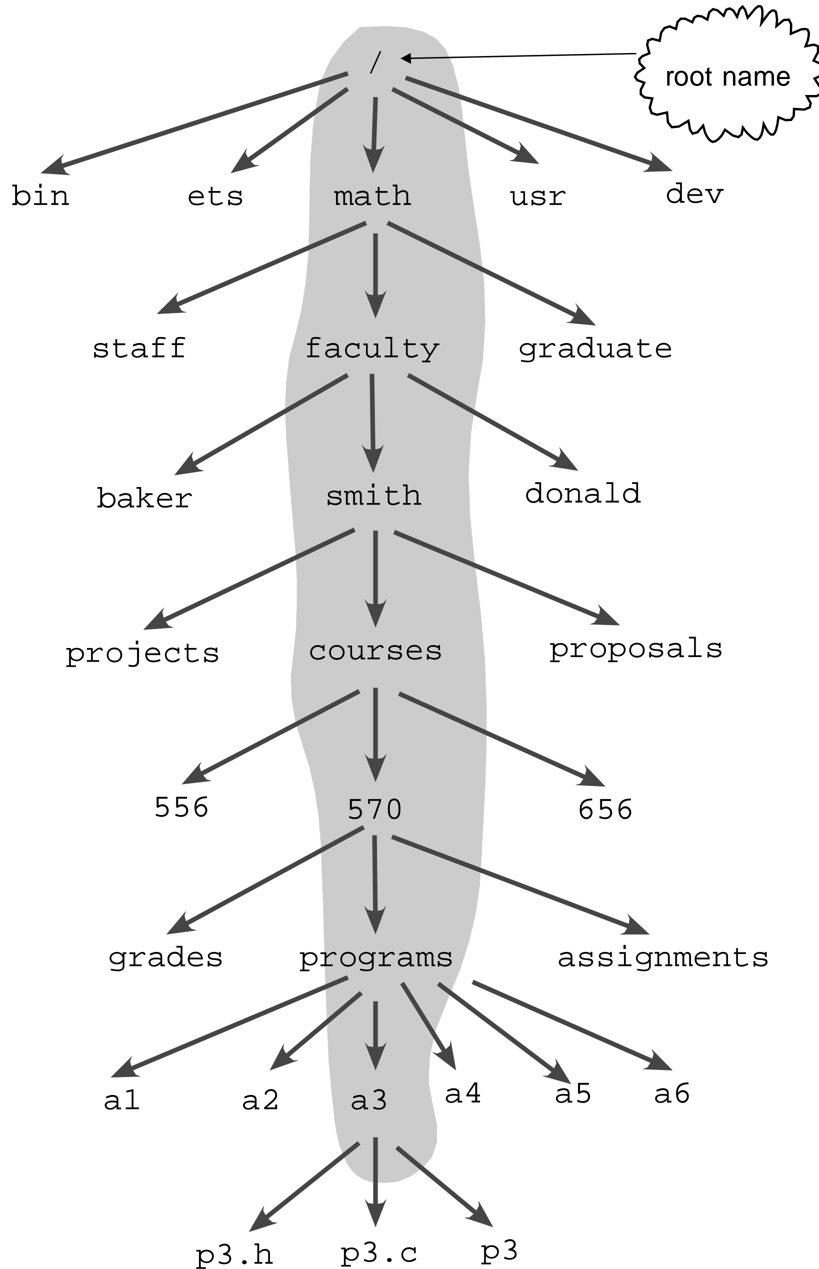
Each path name has to be unique. However, particular parts of the path name can have identical names. For example the following would be correct:

```
/math/faculty/smith/projects/programs/...
/math/faculty/baker/programs/p3.c
```

Directory name `programs` shows up three times, twice in subtree `smith` and once in subtree `baker`. Also, both `smith` and `baker` have files `p3.c`, but they are different files.

Copyright © 1998-2002 by Marko Vuskovic

NAMING (Cont.)



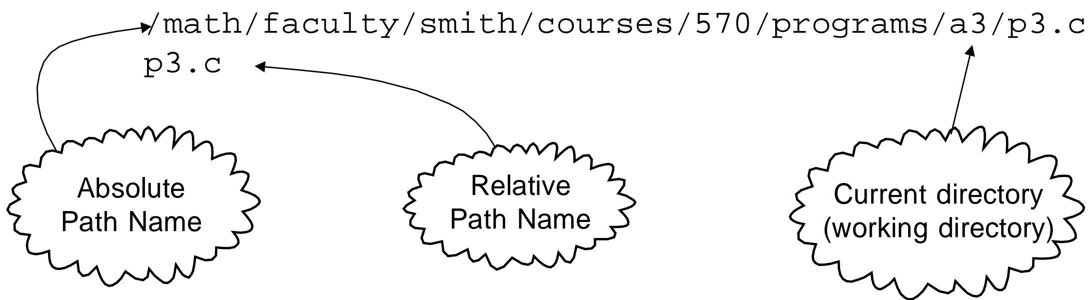
This is an incomplete name structure of a file system (not all names are shown, e.g. `bin`, `etc`, `usr`,...can have subtrees). The full name of the file "`p3.c`" includes all parent names, i.e. `/math/faculty/smith/courses/570/programs/a3/p3.c`.

Copyright © 1998-2002 by Marko Vuskovic

NAMING (Cont.)

Relative Path Names

Identifying a file with its full path name could be very cumbersome, specially in case of large trees. Therefore all OS support shortcuts for the name usage by maintaining a variable called current directory name, or working directory name. For example, suppose that the current directory is a3, then the following two file names are equivalent:



Absoulte path names always start with the root directory name (in UNIX "/"), while the relative path names start with a directory name or a file name which is not "/". Examples of relative path names:

`smith/courses` (can be referenced only if the current directory is `/math/faculty`)
`courses` (the current directory must be `/math/faculty/smith`)
`a3/p3.h` (the current directory must be `/math/faculty/smith/courses/570/programs`)

Parent and Current Directory Names

Most of the OS also have generic names for the current directory name (".") and for the parent directory name (".."). Suppose the working directory is `/math/faculty/smith`, then the following is valid:

`.`	is equivalent to	`/math/faculty/smith`
`..`	is quivalent to	`/math/faculty`
`../..`	is equivalent to	`/math`
`../../staff`	is equivalent to	`/math/staff`

Copyright © 1998-2002 by Marko Vuskovic

NAMING (Cont.)

Name Linking

Sometimes it is convenient that same file shows up in two different subtrees. For example `smith` and `baker` want to share file `p3.h`, and baker wants to access the file as

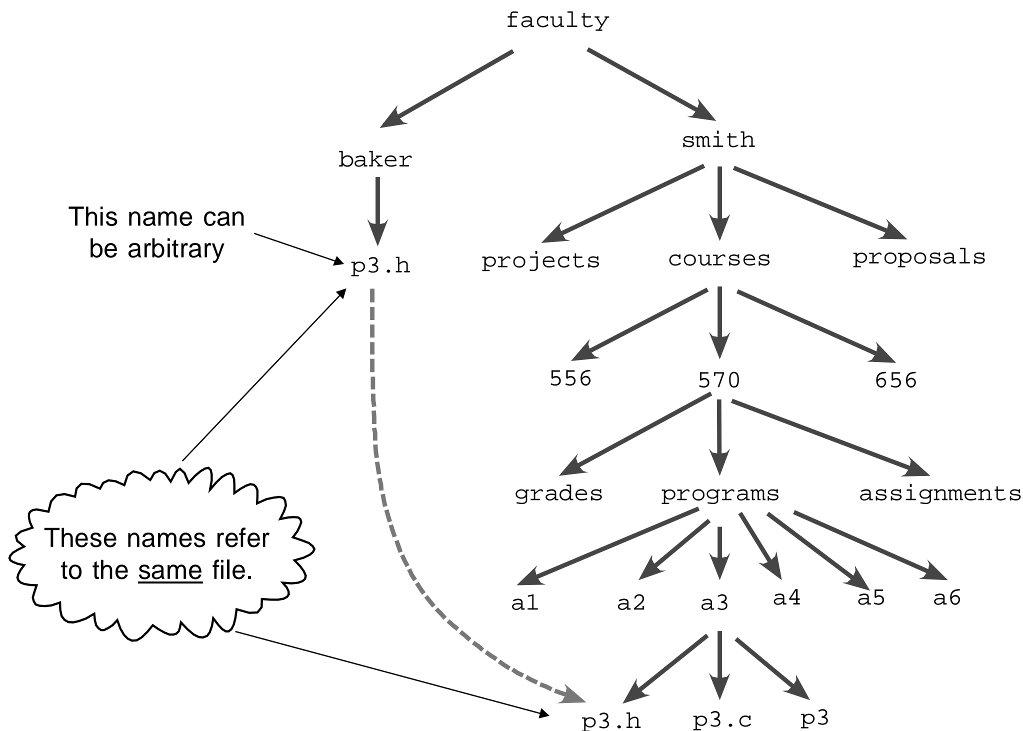
`p3.h` (Baker's current directory is `/math/faculty/baker`)

instead of:

`/math/faculty/smith/courses/570/programs/a3/p3.h`

This can be done by name linking. In UNIX:

```
ln /math/faculty/smith/courses/570/programs/a3/p3.h p3.h
```



After name linking the name graph is no longer tree, it becames a more general kind of graph called Directed Acyclic Graph (DAG). ("Acyclic" means there are no loops in graph, i.e. the same node can not have any of its ancestor nodes as its child node.)

Copyright © 1998-2002 by Marko Vuskovic

DIRECTORIES

File Attributes

Besides the file address and the file name, there are other pieces of information which are of interest for a given file, and which should be part of it as well as the file's primary content. These pieces of information are called file attributes. Here are typical (the most important) file attributes:

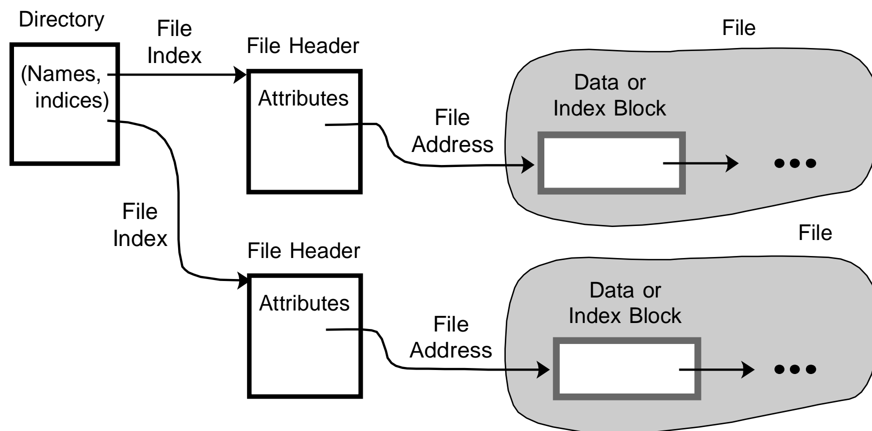
File Name (ASCII string of characters defined by the file creator)
File Type (regular file, directory, symbolic link (see later), char. device, block device, pipe, socket)
Count (Count of number of hard links to this file, *see later*)
Volume (device on which the file is stored)
File Address (A single LBN, or an array of LBNs that point to data/index blocks)
Size Used (current size of file in bytes or blocks)
Size Allocated (maximum size of the file)
Owner (user name of the user who has full control over the file)
Access Rights (r/w/x bits for owner, group and others)
Date Created (when the file was first placed in the system)
Date Last Read Access
Date Last Modified
Current Usage (open, locked, updated in memory but not rewritten onto disk)

The file attributes are usually kept in two separate data structures:

Directory (Maps file name into file index. One directory for several files)

File Header (Contains the rest of the file attributes. One file header for each file)

(Some, usually smaller file systems like FAT, keep all file attributes in directory.)

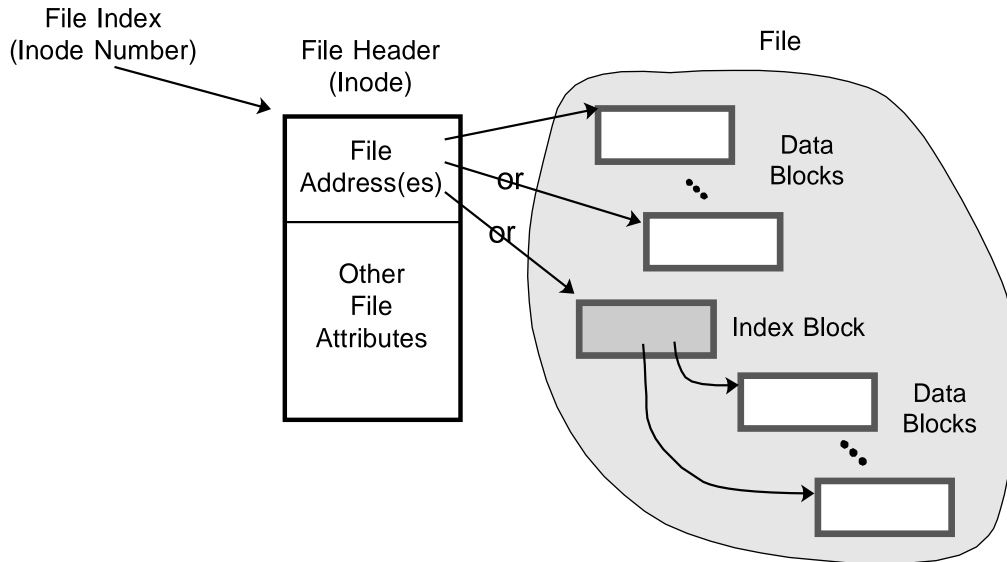


Copyright © 1998-2002 by Marko Vuskovic

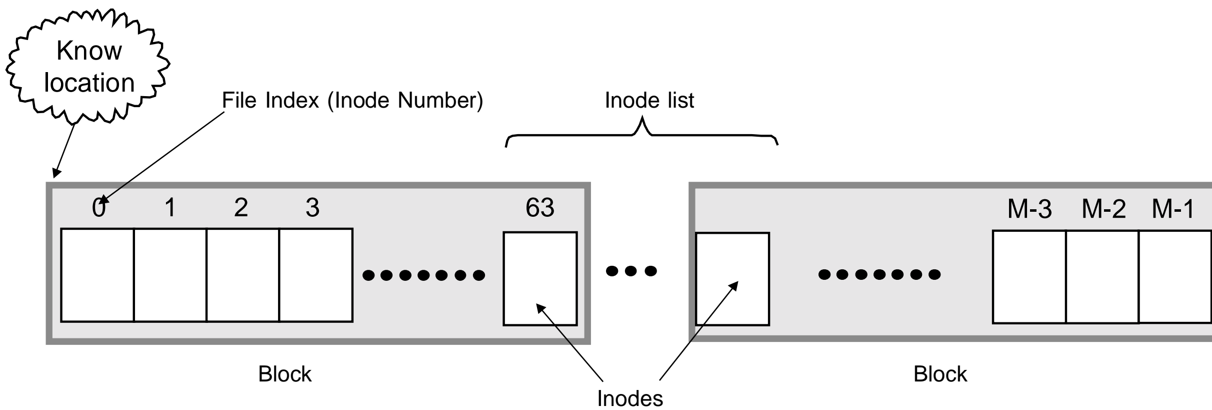
DIRECTORIES (Cont.)

File Header

File headers are also called inodes (index nodes, i-nodes) in UNIX world.
They contain file attributes (without the file name).



File headers are organized as an array of structures called file header list (or inode list), which can span several disk blocks, starting at some known place in disk space.
The index of the array is called file index (or inode number).



Example:

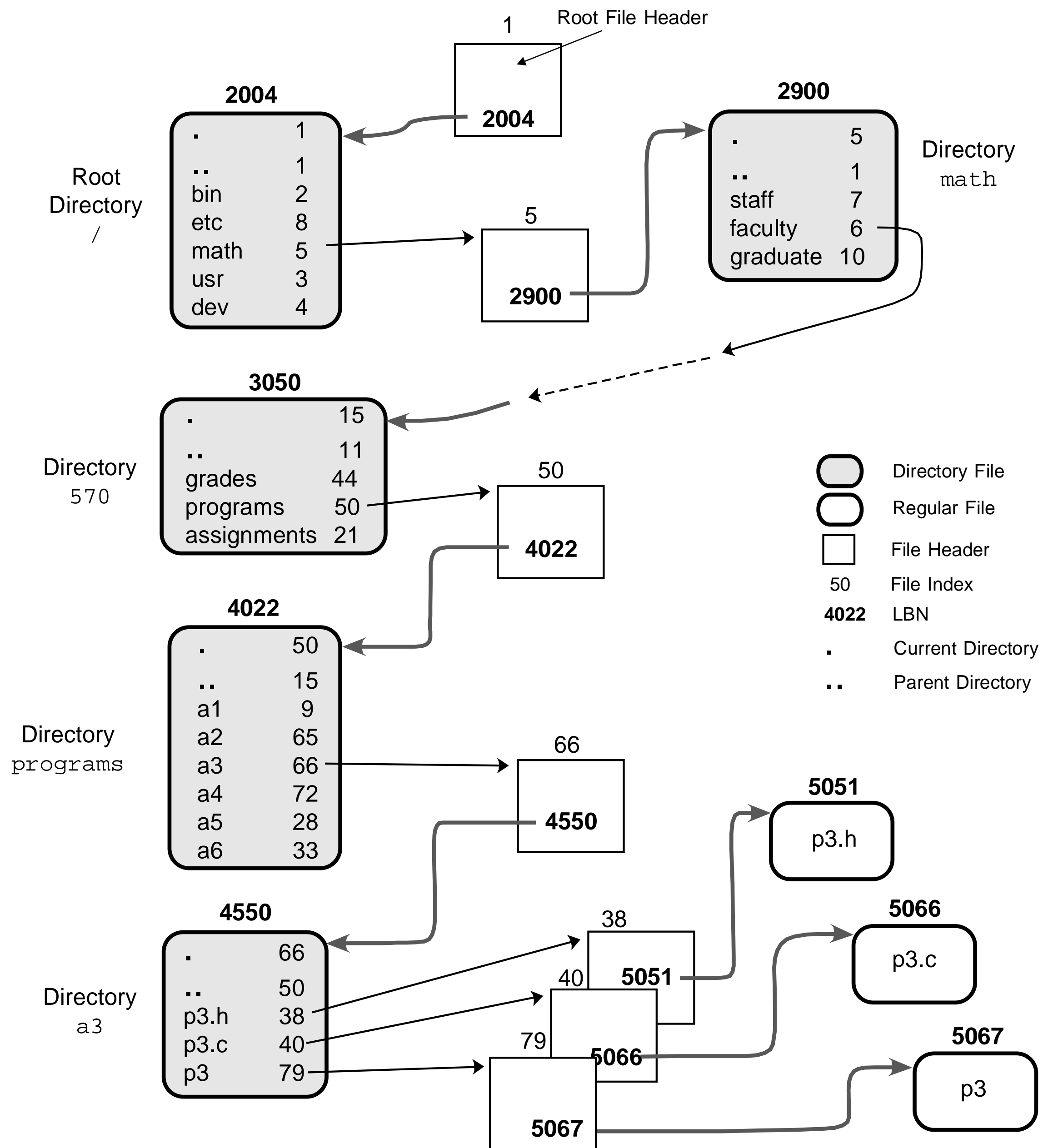
Suppose a file header size = 64 bytes (BSD 4.1) and disk blocks = 4KB.
Then the total of 1 M files in the file system would require 2^{20} x 64/(4 2^{10}) = 16 K blocks.
(An 8 GB disk has 2 M blocks.)

Copyright © 1998-2002 by Marko Vuskovic

DIRECTORIES (Cont.)

Directory Implementation

In most OS directory is a file which contains the list of pairs: <*file name, file index*>. The directory would be too complex and wastefull if it was a single file, and if it contained the full path name for each file. Instead, directories are hierarchically organized.



Copyright © 1998-2002 by Marko Vuskovic

DIRECTORIES (Cont.)

NOTICES:

Directories are files which are under control of the OS. An interactive user can only see some of the attributes by using commands `ls` (UNIX) or `dir` (MS-DOS).

There are several ways to implement directories:

Linear lists

Requires linear search (simple but time inefficient)
Adding new names: Check for name conflict, add name at the end fo list.
Deleting name: Mark used-bit.

Sorted lists

Allows binary search, complicates adding and deleting.

Hashing

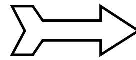
A hash table is added to the linear list of names.
Fast search. Must organize overflow linkages.

Copyright © 1998-2002 by Marko Vuskovic

DIRECTORIES (Cont.)

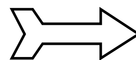
How many disk I/O are needed to access the first byte of the file p3.c, starting from the root directory?

root file header
root directory
file header for `math`
directory for `math`
file header for `faculty`
directory for `faculty`
file header for `smith`
directory for `smith`
file header for `courses`
directory for `courses`
file header for `570`
directory for `570`
file header for `programs`
directory for `programs`
file header for `a3`
directory for `a3`
file header for `p3.c`
first data block for `p3.c`



9 access to file headers
9 accesses to disk blocks
If the file headers were kept in memory there would be 9 seeks and disk transfers.

How many disk I/O would be needed if `a3` was the current directory?
(file header for a3 is in cahe, so is directory for a3)



file header for `p3.c`
first data block for `p3.c`

1 access to hile header
1 accesses to disk blocks

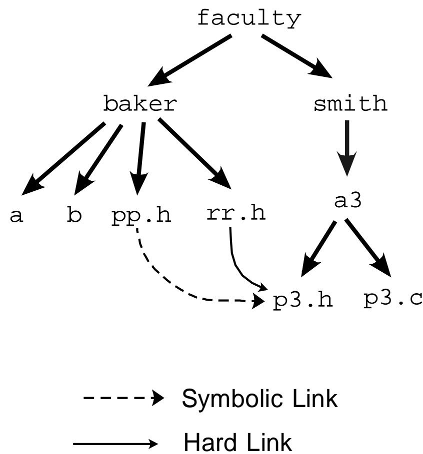
NOTICE: The efficiency of the file access can be very much improved by <u>caching</u>.
First, file headers of all open files are normally kept in memory.
Second, the most recently used disk blocks (directories, regular files) are kept in memory.

Copyright © 1998-2002 by Marko Vuskovic

DIRECTORIES (Cont.)

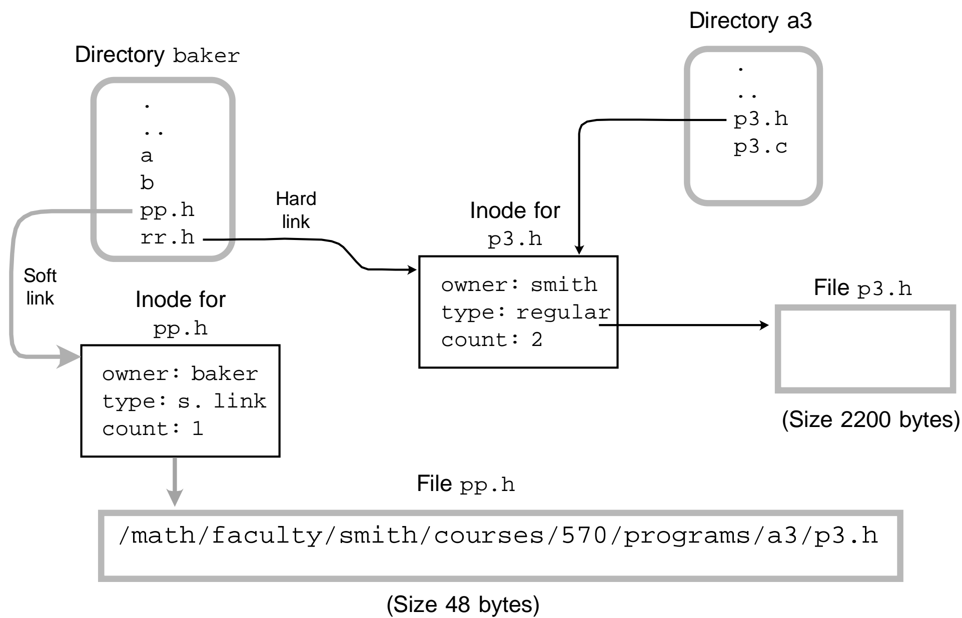
Implementation of Links

There are two kinds of links: hard links and symbolic links.



Hard link introduces a new directory entry which points to the same inode (that inode will increment the link count). Problem: if `smith` deletes the file `p3.h` the `baker`'s directory will continue to point to the inode which is owned by `smith`. Consequently, there will be a file owned by `smith`, which can be seen only by `baker`, and `baker` is the only one who can delete the file. In other words, `smith` has no way to delete the file which he owns!

Soft link introduces a new directory entry and a new file and inode. The file will contain the full path name of the smith's file `p3.h` and the file type will be link type. All programs that access the file `pp.h` will see that it is a link file and will search for the indicated file. If smith deletes `p3.h` baker will simply get the message that the file doesn't exist. Symbolic link introduces time and space overhead (more directory searches and more inodes.)



Example of creation of a hard link (default), and soft link (switch -s):

```
> pwd
> /math/faculty/baker
> ln /math/faculty/smith/courses/570/programs/a3/p3.h rr.h
> ln -s /math/faculty/smith/courses/570/programs/a3/p3.h pp.h
```

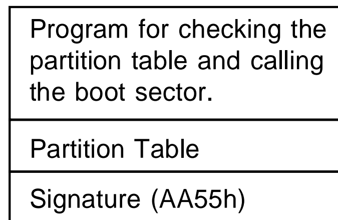
Copyright © 1998-2002 by Marko Vuskovic

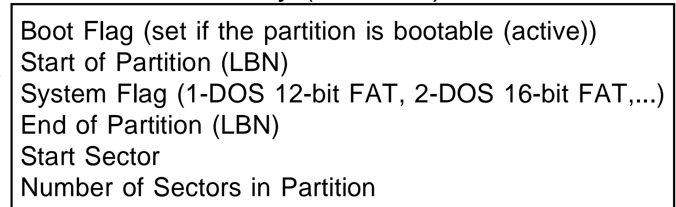
FILE SYSTEM

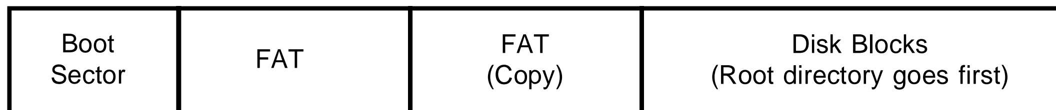
The integration of directories, file headers, files and free space into a file system is different in different OS. The most important file systems are MS-DOS (also called FAT - File Allocation Table), UNIX, and NTFS.

MS-DOS File System

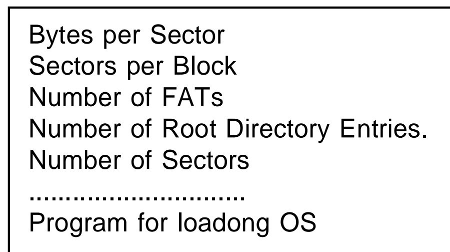
There can be 1,2,3, or 4 partitions on one disk. Partitions can be formatted for different file systems, for example one partition can have MS-DOS, while the other partition can have UNIX file system (dual boot)







Fixed location on disk



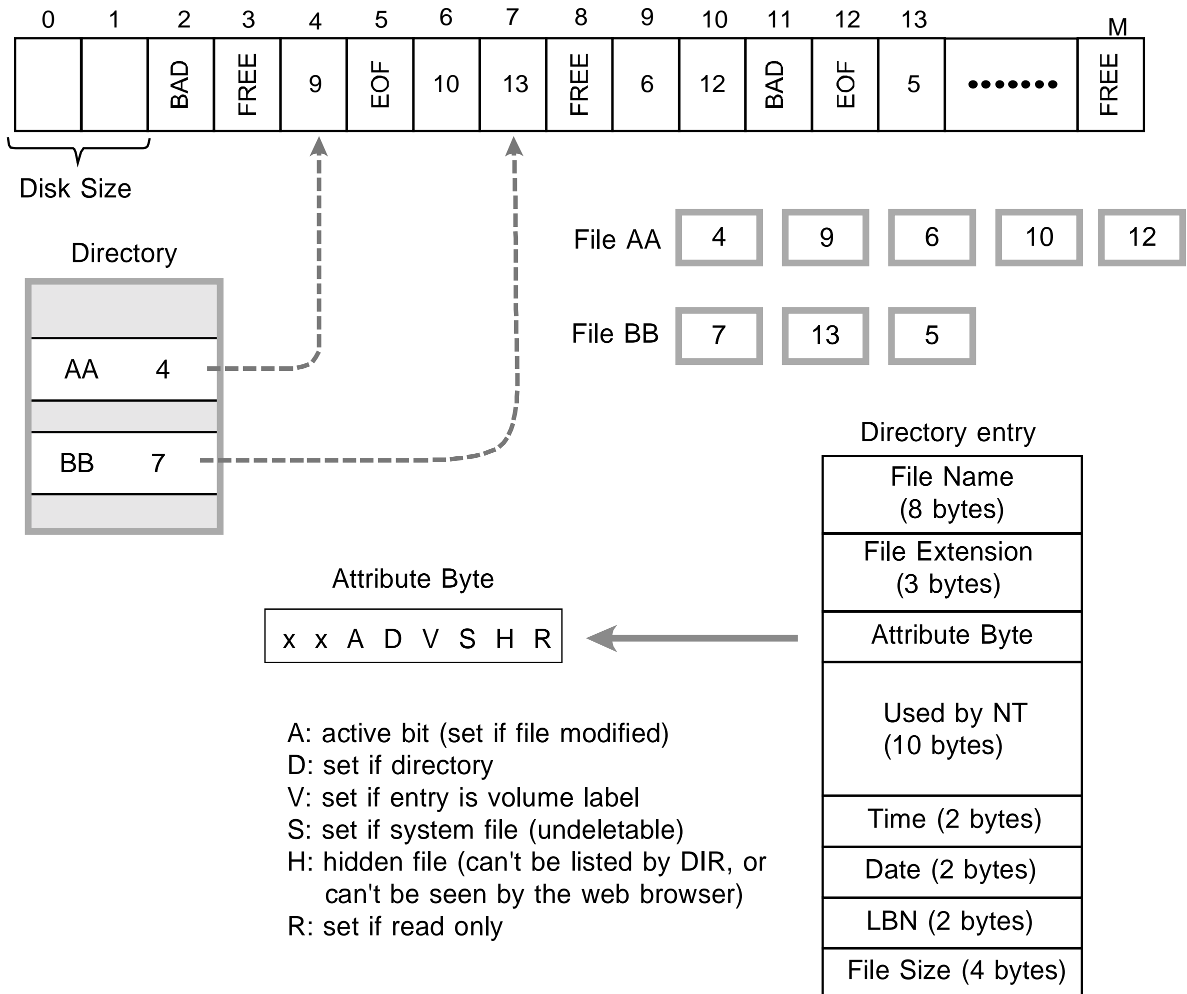
Copyright © 1998-2002 by Marko Vuskovic

FILE SYSTEM (Cont.)

File Allocation Table (FAT) implements a linked allocation with linkages kept in a separate array of links. The array has an entry for each disk block in the partition. Each entry is 16-bit long and it can have the following values:

0000 - free block
FFF0 - FFF6 - reserved
FFF8 - Bad block
xxxx - LBN of the next block.

Directories are files. First directory is the root directory which is always at a fixed location on disk. Directories in FAT contain file name, file start address and other file attributes (i.e. the directory and the file header are not separated in FAT file system.)



Copyright © 1998-2002 by Marko Vuskovic

FILE SYSTEM (Cont.)

NOTICES:

16-bit FAT can have maximum 2^{16} = 64 K entries. If blocks have size of one sector (512 bytes) then the maximal addressable disk kapacity is 64 K x 0.5 KB = 32 MB. ("The magic 32-MB-boundary" for DOS partitions.)

With 2 KB blocks (also called clusters in Microsoft terminology), the disk capacity is increased to 128 MB. Further increase of the block size would yeld larger internal fragmentation.

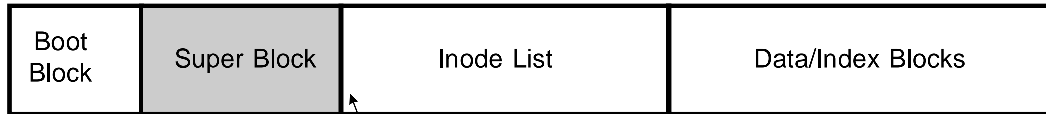
Windows NT uses 32-bit FAT which allows 2^{32} x 2 KB = 8 TB (tera bytes) of addressable disk space!

Copyright © 1998-2002 by Marko Vuskovic

FILE SYSTEM (Cont.)

UNIX File System

Disk



(Can be empty)

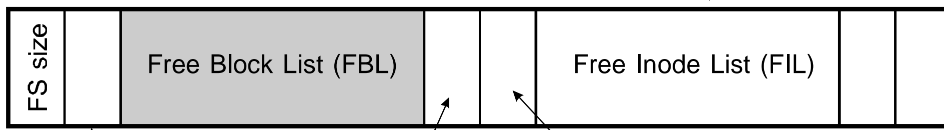
(Usually one block)

(Several blocks)

First inode is root inode

See next page

Super Block (Cached)



# of free blocks in FS

Size of Inode List

# of free inodes in FS

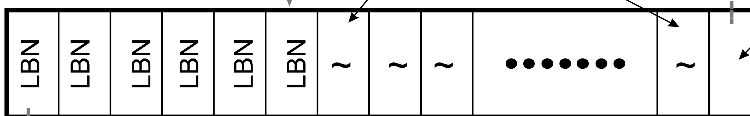
Lock Fields for FBL and FIL

Blocks are allocated from here

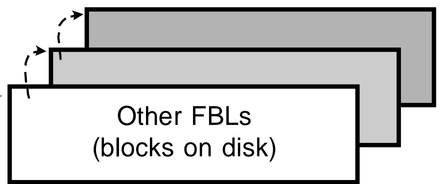
Free Block List (FBL)

Allocated blocks

Index to the next free block to be allocated



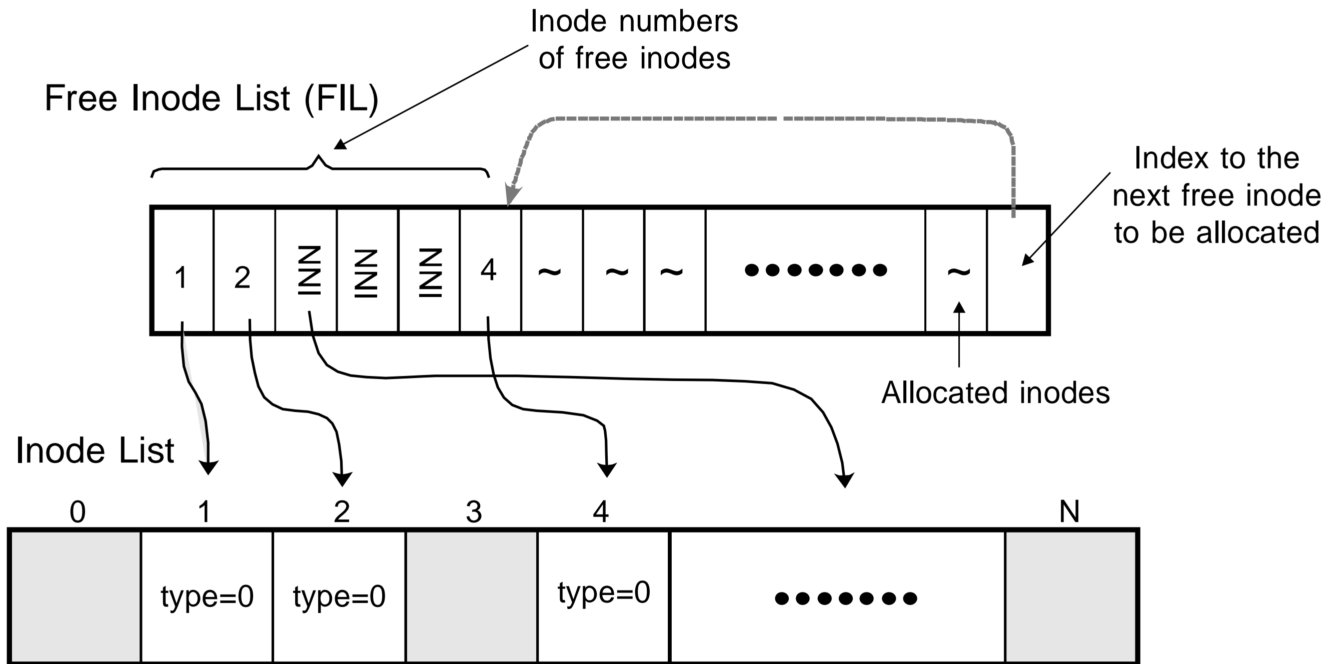
Pointer to the next FBL



If the free block list is empty, then the file system searches this part of the FBL block for the freed blocks and allocates them

Copyright © 1998-2002 by Marko Vuskovic

FILE SYSTEM (Cont.)

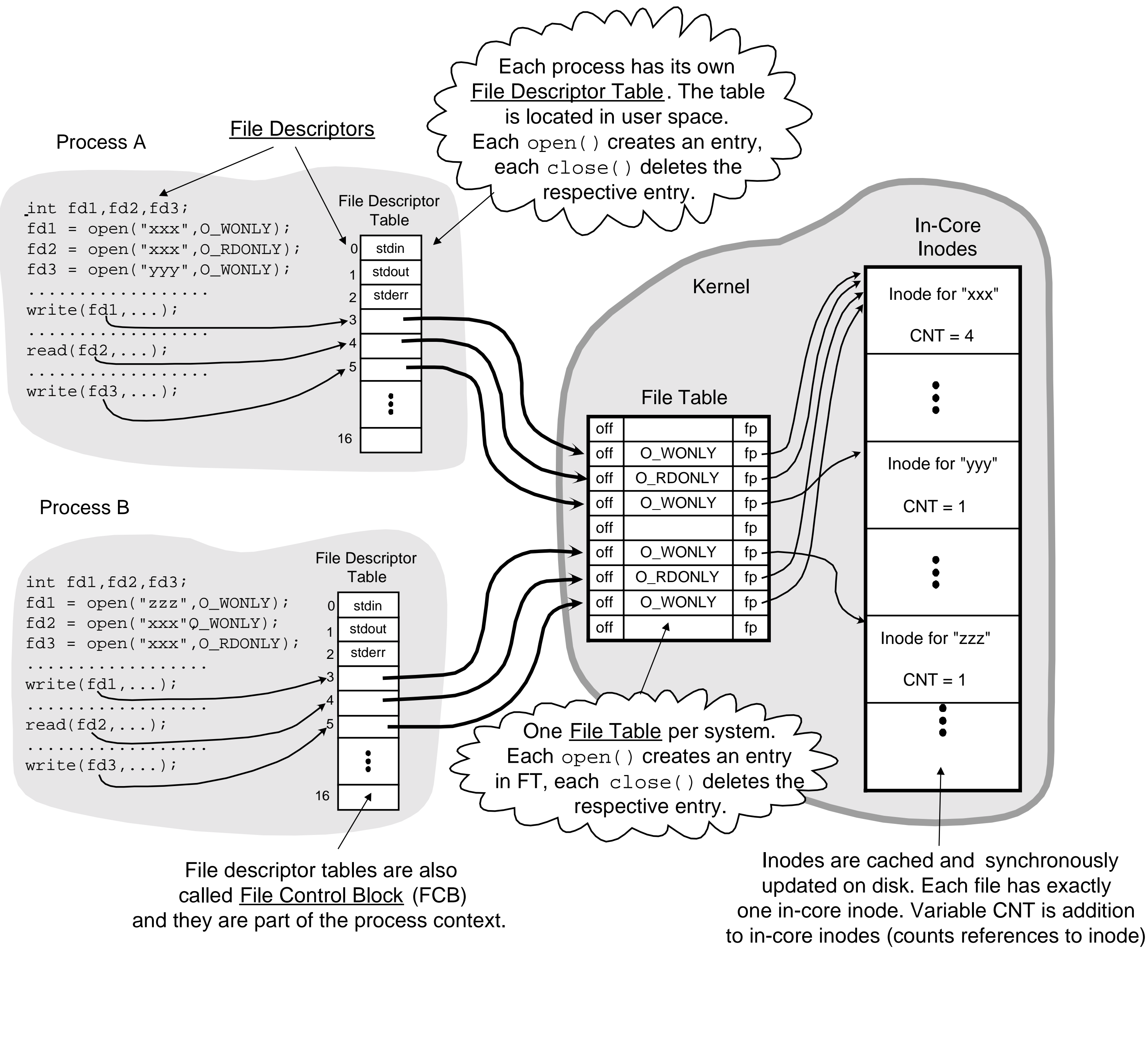


Inode list is not a part of the super block.
It spawns several disk blocks. Each inode takes 64 bytes. Free inodes are marked type=0.

Inodes are allocated from the FIL. If FIL is empty, OS searches inode list for free inodes (type=0) and places them into FIL, as much as possible.
If an inode is freed, its INN is placed into FIL if possible.

Copyright © 1998-2002 by Marko Vuskovic

FILES AND PROCESSES



Copyright © 1998-2002 by Marko Vuskovic

FILES AND PROCESSES (Cont.)

COMMENTS

(1) Why directory must be separated from the file header (inode)?

Helps file sharing and usage of links. The file attributes must be unique, while there can be several names (directories) pointing to the same file. Suppose Baker and Smith share the file `p3.h` (Smith is the owner). If Smith changes the file (appends more data) this would require update of the file attributes (time, file address etc.) If these parameters were in several places, how could the owner know where to make updates. Besides, this could cause unnecessary redundancy.

(2) What is the reason to have File Table?

File table has the byte offset for the file and the access method (O_RDONLY, O_WRONLY, O_RDWR). A file can be opened several times by the same process or by several processes. Each file opening should maintain separate state of that particular file operation, which is not relevant to the attributes.

(3) Why File Descriptor Table?

That table makes design cleaner. File descriptor table is a private data structure of the process, and as such constitutes a part of the process context (it is also called File Control Block (FCB)).

(4) What is file descriptor and what is file handle?

File descriptor is an index to the file descriptor table. Since the FD table entries contain pointers to the file table, which is a memory resident Kernel data structure, we can say that the file descriptor is essentially a double pointer.

File handle is the same concept used in Windows NT. There is some formal difference though. A file descriptor has type `int`, while a file handle has type `*void`. The concept of double pointing is used in Windows NT for many other structures, not only files. They include all structures that are named, protected and shared by several processes and managed by the operating system (processes, threads, files, semaphores, events, pipes, sockets,...). As will be shown later (see Chapter 12) all these structures are implemented as objects, which are referenced uniformly by the same variable type (`typedef void *HANDLE`) regardless of the object type.

Copyright © 1998-2002 by Marko Vuskovic