

Chapter 4 - File Systems

Luis Tarrataca

`luis.tarrataca@gmail.com`

CEFET-RJ

1 Motivation

2 Files

File Naming

File Structure

File Types

File Access

File Attributes

File Operations

Example Program Using File-System Calls

3 Directories

Hierarchical Directory Systems

Path Names

Directory Operations

4 File System Implementation

File System Layout

Implementing the files

Implementing the files

Implementing the files

Contiguous Allocation

Linked-List Allocation

Linked-List Allocation Using a Table in Memory

I-Nodes

I-Nodes

Implementing Directories

5 File-system Management and Optimization

Disk-space management

Block Size

Keeping track of free blocks

Disk Quotas

File-system performance

Caching

Block read-ahead

Reducing Disk-Arm motion

Defragmenting Disks

6 References

Motivation

There are three essential requirements for long-term information storage:

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

How do you find information?

How do you keep one user from reading another user's data?

How do you know which blocks are free?

In your opinion what are some of the most important concepts OS?

In your opinion what are some of the most important concepts OS?

- Process? Threads?
- Physical memory? Virtual Memory?

In your opinion what are some of the most important concepts OS?

- Process? Threads?
- Physical memory? Virtual Memory?

Today we will learn a new abstraction. Can you guess what it is?

In your opinion what are some of the most important concepts OS?

- Process? Threads?
- Physical memory? Virtual Memory?

Today we will learn a new abstraction. Can you guess what it is?

- The file...

Files

First things first:

What is a file? Any ideas?

Files

First things first:

What is a file? Any ideas?

- **Files:** are logical units of information created by processes:
 - Processes / Threads can read existing files and create new ones;
 - Information stored in files must be persistent, *i.e.*:
 - not affected by process creation and termination.

Files are managed by the operating system. How they are

- structured...
- named...
- accessed...
- used...
- protected...
- implemented...
- and managed

are major topics in operating system design.

OS part dealing with files is known as the **file system**:

- The subject of this chapter =>

File Naming

Exact rules for file naming vary somewhat among OS:

- Current OS allow strings of various lengths as legal file names;
- OS typically support two-part file names: (filename, extension);

File Structure

Files can be structured in any of several ways (1/3):



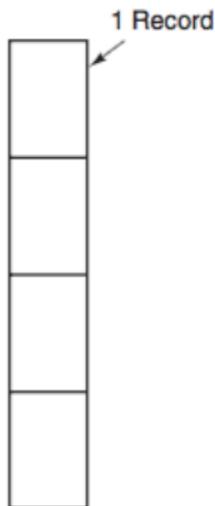
Files are merely **byte sequences**:

- Maximum flexibility;
- Unix, Linux, OS X and Windows use this model;

Figure: The memory hierarchy (Source: (Tanenbaum and Bos, 2015))

File Structure

Files can be structured in any of several ways (2/3):



File is a sequence of fixed-length **records**:

- Each record has a certain number of bytes;
- Read operation returns one record;
- Write operation overwrites or appends one record.

Figure: Record sequence file structure. (Source: (Tanenbaum and Bos, 2015))

File Structure

Files can be structured in any of several ways (3/3):

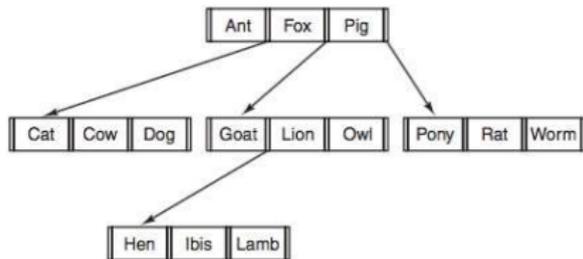


Figure: Tree file structure (Source: (Tanenbaum and Bos, 2015))

File consists of a **tree of records**:

- Not necessarily all the same length;
- Each record contains a key field in a fixed position in the record
- Tree is sorted on the key field:
 - Allowing rapid key search;

File Types

OS support several types of files:

- **Files:** containing user information:
 - Containing ASCII characters;
 - Or containing binary information:
 - Only readable by the computer;
 - All programs are binary files;
- **Directories:** system files for maintaining the structure of the file system;

File Access

When magnetic disks appeared 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**;

Two methods can be used for specifying where to start reading:

- 1st method: every read gives the position in the file to start reading at;
- 2nd method: **seek** operations sets current position:
 - After a seek, the file can be read sequentially from the now-current position;
 - Used in UNIX and Windows;

File Attributes

OS keep track of a wide range of information regarding a file:

Can you think of a few attributes that OS maintain regarding a file? Any ideas?

File Attributes

OS keep track of a wide range of information regarding a file:

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure: Some possible file attributes (Source: (Tanenbaum and Bos, 2015))

File Operations

What are the most common file operations made available by the OS?
Any ideas?

File Operations

Most common system calls relating to files (1/5):

- **Create:** file is created with no data;
- **Delete:** When the file is no longer needed, it has to be deleted to free up disk space
- **Open:** Before using a file, a process must open it in order to:
 - fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

File Operations

Most common system calls relating to files (2/5):

- **Close:** When all the accesses are finished:
 - attributes and disk addresses are no longer needed;
 - file should be closed to free up internal table space;
- **Read:** Data are read from file:
 - Bytes come from the current position;
 - Caller must specify how many bytes to read and buffer to place data;

Most common system calls relating to files (3/5):

- **Write:** Data are written to the file using 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;

Most common system calls relating to files (4/5):

- **Append:** restricted form of write. It can add data only to the end of the file;
- **Seek:** repositions file pointer to a specific place in the file:
 - After this call, data can be read from, or written to, that position

Most common system calls relating to files (5/5):

- **Get attributes:** read file attributes;
- **Set attributes:** set some of the attributes;
- **Rename:** changes the name of an existing file;

Example Program Using File-System Calls

Can you tell what the following program is doing? Any ideas?

```
#include <sys/types.h> /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]); /* ANSI prototype */

#define BUF_SIZE 4096 /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700 /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1); /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY); /* open the source file */
    if (in_fd < 0) exit(2); /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
    if (out_fd < 0) exit(3); /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break; /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0) /* no error on last read */
        exit(0);
    else
        exit(5); /* error on last read */
}
```

Can you tell what the following program is doing? Any ideas?

- Copies one file from its source file to a destination file;

Hierarchical Directory Systems

First things first:

What is a directory? Any ideas?

Directories

First things first:

What is a directory? Any ideas?

- File systems normally have directories or folders, which are themselves files:
 - Allows the file system to have a hierarchy of files;
 - Grouping related files together;
 - Tree of directories;

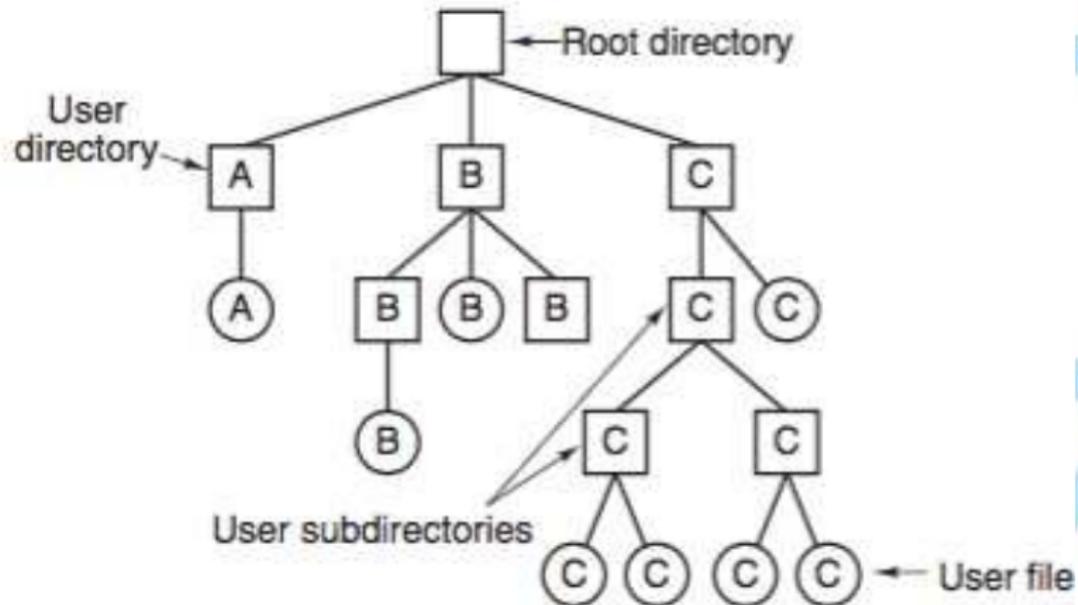


Figure: A hierarchical directory system. (Source: (Tanenbaum and Bos, 2015))

Path Names

When the file system is organized as a directory tree:

- Some way is needed for specifying file names;
- Usually there are two solutions:
 - **Absolute Pathname:** *E.g.:* `"/usr/ast/mailbox"`
 - **Relative Pathname:** makes use of the current directory:
 - *E.g.:* current directory is `"/usr/ast"` which can have file `"mailbox"`

Usually, OS also have two special directories:

- Directory `.` - represents the current directory;
- Directory `..` - represents the parent directory;

Directory Operations

What are the most common directory operations made available by the OS? Any ideas?

Directory Operations

What are the most common directory operations made available by the OS? Any ideas?

Don't forget that directories are files...

- Therefore the available system calls should be similar;

Directory Operations

Most common system calls relating to directories (1/3):

- **Create:** creates an empty directory;
- **Delete:** removes an empty directory;
- **Opendir:** to open a directory;

Directory Operations

Most common system calls relating to directories (2/3):

- **Closedir:** . When a directory has been read, it should be closed to free up internal table space.
- **Readdir:** to list the contents of a directory;
- **Rename:** to rename an existing directory;

Directory Operations

Most common system calls relating to directories (3/3):

- **Link:** creates a link for a file in a given directory;
- **Unlink:** removes a file present in the directory;

File System Implementation

Now that we know all the main file system concepts:

How are such concepts implemented in an OS? Any ideas?

- How are files and directories stored?
- How is disk space managed?
- How to make everything work efficiently and reliably?

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):
 - Used to boot the computer;
 - End of MBR contains the **partition table**

Partition Table contains:

- Starting and ending addresses of each partition;
- One of the partitions in the table is marked as active;
- When computer is booted:
 - BIOS reads in and executes the MBR program;
 - Active partition is located;
 - Active partition boot block is read and executed;
 - Boot block program loads OS;

Layout of a disk partition varies a lot from file system to file system:

- Usually it goes something like this:

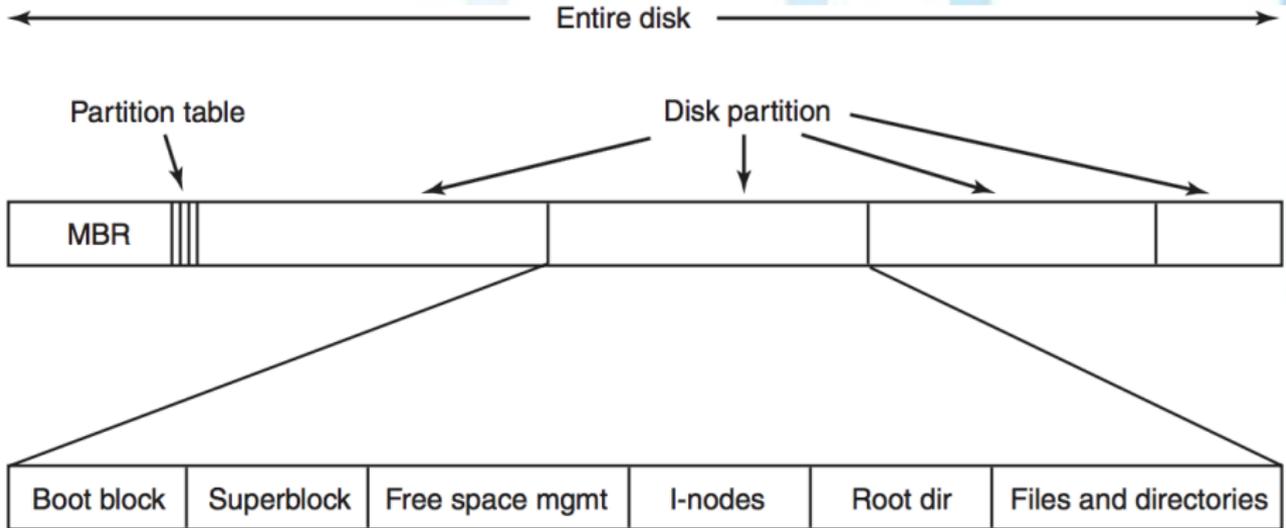


Figure: A possible file-system layout (Source: (Tanenbaum and Bos, 2015))

From the previous figure (1/2):

- **Superblock:** contains all the key parameters about the file system;
 - File-system type identification;
 - Number of blocks;
 - Etc...
- **Free space mgmt:** information about the file system free blocks;
 - *E.g.:* Bitmap or list of pointers

From the previous figure (2/2):

- **I-nodes:** array of data structure, one per file, detailing the file;
- **Root directory:** contains the top of the file-system-tree;
- **Files and directories:** containing all the real information;

Implementing the files

How can we implement a file?

Implementing the files

How can we implement a file?

How is a file represented?

Implementing the files

How can we implement a file?

How is a file represented?

Using a magnetic disk:

- Tracks;
- Sectors;
- New concept: **Block** which is a set of sectors;

Various methods are used in different operating systems:

- **Contiguous Allocation**
- **Linked List Allocation**
- **Linked-List Allocation Using a Table in Memory**

Guess what we will be seeing next? Any ideas? =P

Contiguous Allocation

Idea: Store each file as a contiguous run of disk blocks:

- *E.g.:* 50-KB file would be allocated to
 - 50 consecutive blocks using a disk with 1-KB blocks:
 - 25 consecutive blocks using a disk with 2-KB blocks:

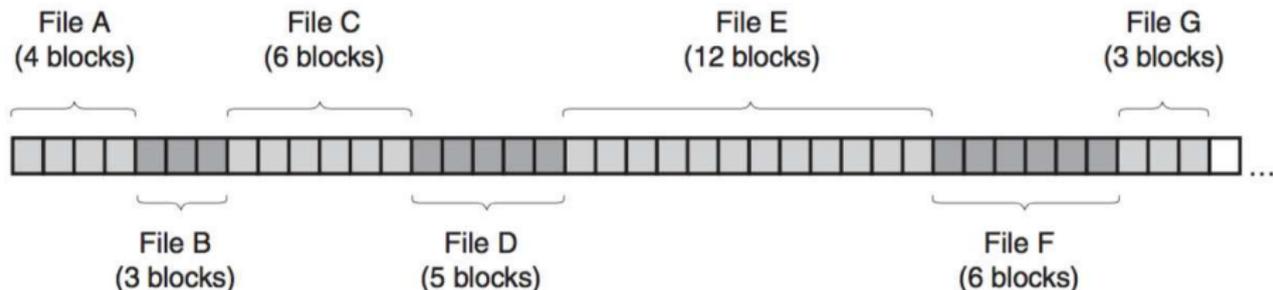


Figure: Contiguous allocation of disk space for seven files (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- First 40 disk blocks are shown;
- Initially, the disk was empty;
- Then a file A, of length four blocks, was written:
 - If file A was $3\frac{1}{2}$ blocks, some space is wasted at the end of the last block;
- After that a three-block file, B, was written;
- In the figure, a total of seven files are shown:
 - Each one starting at the block following the end of the previous one.

In your opinion what are the **advantages** of contiguous allocation? Any ideas?

In your opinion what are the **advantages** of contiguous allocation? Any ideas?

Advantage 1: Simple to implement:

- Keeping track of where a file's blocks are is reduced to:
 - remembering disk address of the first block and number of blocks in the file;

Can you see any other **advantage** of contiguous allocation? Any ideas?

Can you see any other **advantage** of contiguous allocation? Any ideas?

Advantage 2: Read performance:

- Entire file can be read from the disk in a single operation;
- Only one seek is needed for the first block;
- After that, no more seeks or rotational delays are needed:
 - data come in at the full bandwidth of the disk;

In your opinion what are the **disadvantages** of contiguous allocation?
Any ideas?

In your opinion what are the **disadvantages** of contiguous allocation?
Any ideas?

Major disadvantage: over time, disk becomes fragmented

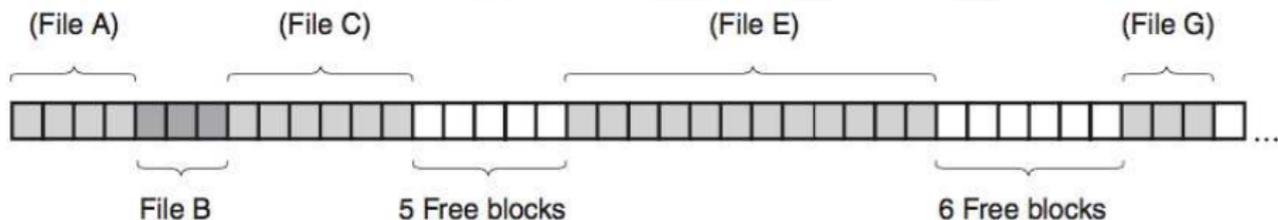


Figure: The state of the disk after files D and F have been removed. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- Files D and F were removed:
 - Respective blocks were then freed;
 - Leaving a run of free blocks on the disk;
- Disk would have to be compacted immediately:
 - Potentially millions of blocks to compact...
 - Disastrous performance;
- As a result: disk consists of files and holes;

Initially: fragmentation is not a problem:

- Each new file can be written at the end of disk:
 - following the previous one;
- However, eventually the disk will fill up, then two solutions exist:
 - Compact the disk: prohibitively expensive;
 - Reuse free space:
 - When a new file is created choose a hole big enough;
 - Requires maintaining a list of holes;

Can you see any other **disadvantages** of contiguous allocation? Any ideas?

Can you see any other **disadvantages** of contiguous allocation? Any ideas?

Major disadvantage: file size needs to be known at time of creation

- This is not always possible to know in advance:
 - File size may change with time...

Conclusion: contiguous allocation is problematic...

Can you think of any other type of method for implementing files?

Linked-List Allocation

Idea: keep each file as a linked list of disk blocks:

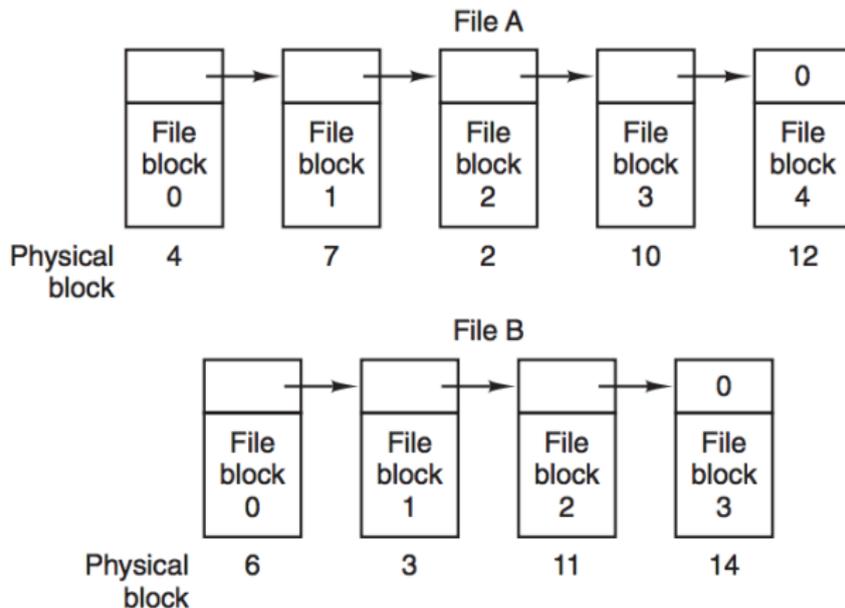


Figure: Storing a file as a linked list of disk blocks. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- First word of each block is used as a pointer to the next one:
 - Rest of the block is for data.
- Unlike contiguous allocation:
 - Every disk block can be used in this method;
 - No space is lost to disk fragmentation:
 - **This does not mean that fragmentation does not occur!**
- Directory entries merely need to store the disk address of the first block:
 - Rest can be found starting there.

Can you see any other **disadvantages** of Linked-List Allocation? Any ideas?

Can you see any other **disadvantages** of Linked-List Allocation? Any ideas?

- Contiguous allocation allows for sequentially file reads:
 - Very efficient =>
- Linked-list allocation implies random block accesses:
 - To get to block **n**, OS has to:
 - Start at the beginning and read **n - 1** blocks prior;
 - **Painfully slow** =(

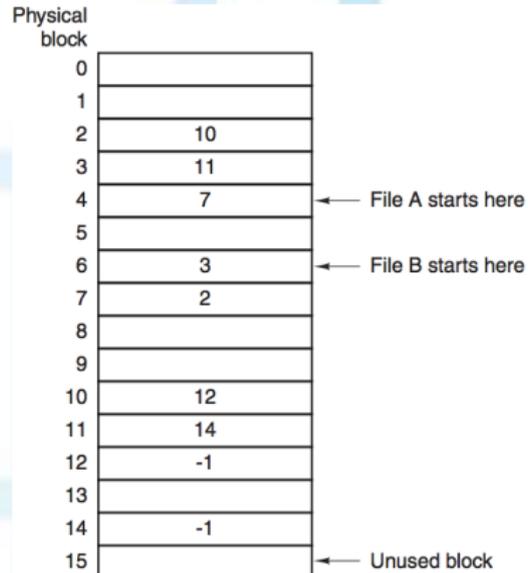
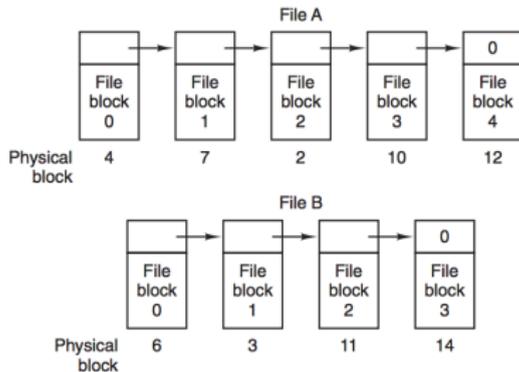
Linked-list file implementation is also problematic...

Can you think of any other methods for implementing a file? Any ideas?

Linked-List Allocation Using a Table in Memory

Disadvantages of the linked-list allocation can be eliminated by:

- Storing the pointer word from each disk block in a table in memory;



In the previous two figures we have two files:

- File A uses disk blocks 4, 7, 2, 10, and 12;
- File B uses disk blocks 6, 3, 11, and 14;
- Using the table:
 - File A: start with block 4 and follow the chain until the end;
 - Chain is terminated with a special marker (e.g., -1)
 - File B: start with block 6 and follow the chain until the end;
 - Chain is terminated with a special marker (e.g., -1)

Such a table in main memory is called a **FAT** (File Allocation Table);

Can you see any any **advantages** with linked-list allocation using a table in memory? Any ideas?

Can you see any **advantages** with linked-list allocation using a table in memory? Any ideas?

Random access is much easier, however chain:

- **Must still be followed to find a given offset within the file;**

Despite this the chain is entirely in memory:

- Can be followed without making any disk references;

Can you see any any **disadvantages** with linked-list allocation using a table in memory? Any ideas?

Can you see any any **disadvantages** with linked-list allocation using a table in memory? Any ideas?

- Entire table must be in memory all the time to make it work;
- Example: 1-TB disk and a 1-KB block size:
 - Table needs to be $2^{40}/2^{10} = 2^{30}$ entries;
 - one for each of the ≈ 1 billion disk blocks
 - Each entry needs a minimum of 30 bits:
 - In order to properly identify the block;
 - Thus the table requires a total of $2^{30} \times 30 \approx 3GB...$
 - **Conclusion:** FAT does not scale well to large disks

Can you see any any **disadvantages** with linked-list allocation using a table in memory? Any ideas?

- When computer is shut down:
 - Table must be stored in non-volatile memory;
 - This implies disk accesses and writes;
- Table is only used to get the block number:
 - Reading / Writing block still requires disk accesses and writes;

I-Nodes

The previous methods had some problems...

How can we keep track efficiently of which blocks belong to which file?
Any ideas?

I-Nodes

The previous methods had some problems...

How can we keep track efficiently of which blocks belong to which file?
Any ideas?

Our last method: **i-nodes**, short for index-node:

- Lists the attributes and disk addresses of the file's blocks
- Each i-node has a fixed position on the disk;

I-nodes, short for index-node:

- Lists the attributes and disk addresses of the file's blocks
- Given i-node, it is then possible to find all the blocks of the file:

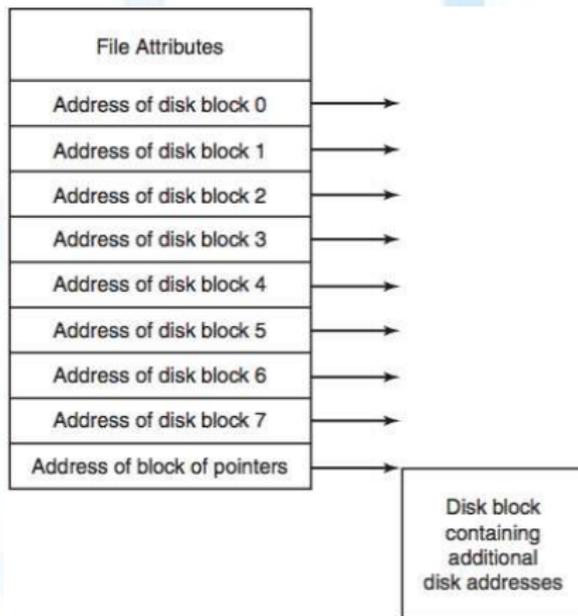


Figure: An example i-node. (Source: (Tanenbaum and Bos, 2015))

How does this scheme compare against linked files using an in-memory table? Any ideas?

How does this scheme compare against linked files using an in-memory table? Any ideas?

i-node needs be in memory only when the corresponding file is open:

- If each i-node occupies n bytes and k files may be open:
 - Array holding the i-nodes for the open files is only kn bytes;

I-node array is far smaller than space occupied by the file table approach:

Why do you think this happens? Any ideas?

I-node array:

- Usually far smaller than the space occupied by the file table approach;
- Reason is simple:
 - Table holding all disk blocks is proportional in size to the disk itself;
 - If the disk has n blocks, the table needs n entries;
 - As disks grow larger, this table grows linearly with them.
- In contrast, i-node scheme requires array size:
 - **Proportional to the maximum number of files that may be open at once**

Can you see any problem with the i-nodes approach? Any ideas?

Can you see any problem with the i-nodes approach? Any ideas?

If each i-nodes has room for a fixed number of disk addresses:

what happens when a file grows beyond this limit? Any ideas?

Can you see any problem with the i-nodes approach? Any ideas?

If each i-nodes has room for a fixed number of disk addresses:

What happens when a file grows beyond this limit? Any ideas?

One solution: reserve the last disk address not for a data block:

- But for the address of a block containing more disk-block addresses;

What happens when a file grows beyond this limit? Any ideas?

One solution: reserve the last disk address not for a data block:

- but for the address of a block containing more disk-block addresses;

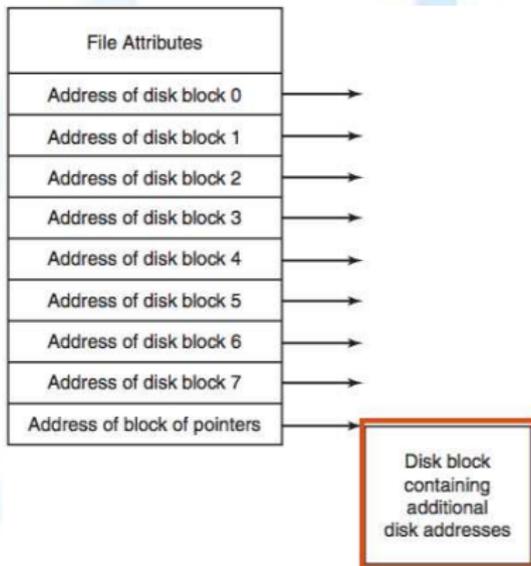


Figure: An example i-node. (Source: (Tanenbaum and Bos, 2015))

An even more advanced solution:

- 1 Two or more such blocks containing disk addresses;
- 2 Disk blocks pointing to other disk blocks full of addresses;

This is known as **indirection blocks**.

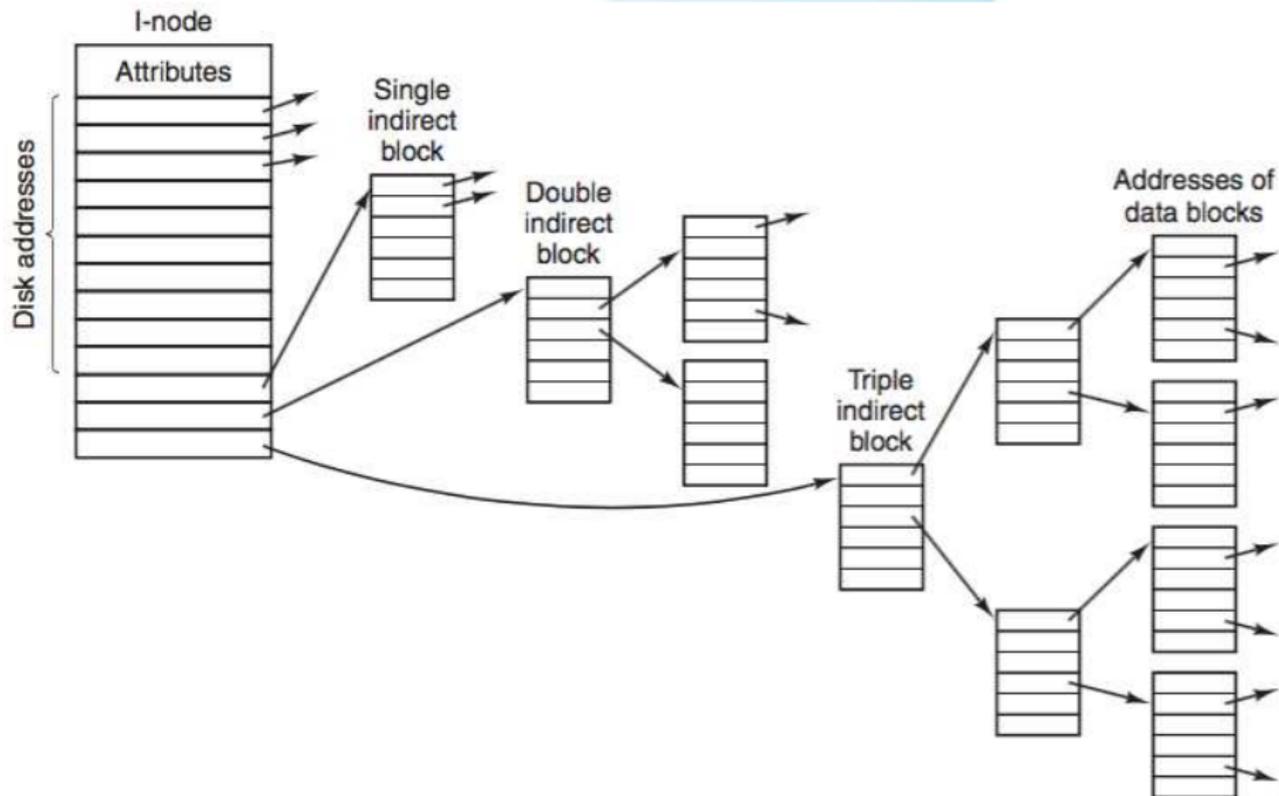


Figure: Indirection blocks example (Source: (Tanenbaum and Bos, 2015))

Implementing Directories

Before a file can be read:

- File must be opened;
- OS uses the path name to locate the directory entry on the disk:
 - Directory entry provides information needed to find the disk blocks:
 - Depending on the system may be:
 - Disk address of the entire file (with contiguous allocation);
 - Number of the first block (both linked-list schemes);
 - I-Node number;
- Main function of directory system is to:
 - Map file name onto the information needed to locate the data.

Every file system maintains various file attributes (1/3):

- *E.g.*: file's owner and creation time;
- These attributes need to be stored somewhere;
- **One possibility**: store them in the directory entry:

games	attributes
mail	attributes
news	attributes
work	attributes

Figure: Simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (Source: (Tanenbaum and Bos, 2015))

Every file system maintains various file attributes (2/3):

games	attributes
mail	attributes
news	attributes
work	attributes

Figure: Simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (Source: (Tanenbaum and Bos, 2015))

Directory consists of a **list of fixed-size entries**, one per file, containing:

- A (fixed-length) file name;
- Attributes:
 - Creator, Time, etc.
 - File disk blocks

Every file system maintains various file attributes (3/3):

- **Another possibility:** for systems that use i-nodes:
 - Store attributes in the i-nodes;
 - Each directory entry can be shorter: {File name, i-node number}

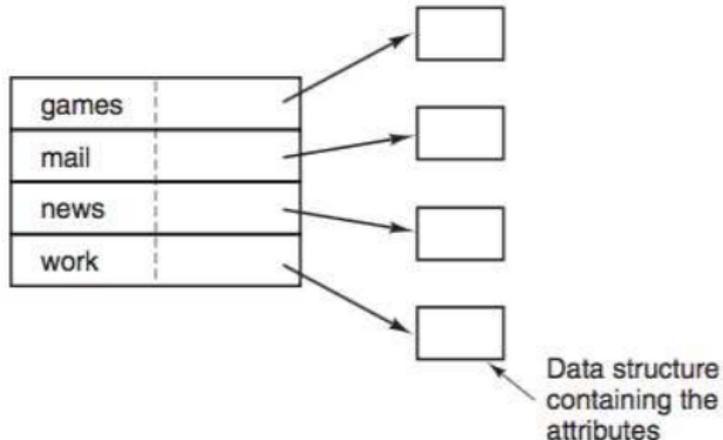


Figure: A directory in which each entry just refers to an i-node. (Source: (Tanenbaum and Bos, 2015))

Just for curiosity:

How can we determine the i-node of files and directories in Linux? Any ideas?

Just for curiosity:

How can we determine the i-node of files and directories in Linux? Any ideas?

- `ls -li =>`

```
takambp:Chapter 4 Taka$ ls -lhi
total 48792
30564577 -rw-r--r--@ 1 Taka  staff  1.2M Sep 22 18:14 Chapter4-FileSystems.pdf
30601907 -rwxr-xr-x@ 1 Taka  staff  2.1K Sep 27 17:09 CoreLatexElements.rtf
30346949 drwxr-xr-x  19 Taka  staff  646B Sep 27 17:10 images
30601652 -rw-r--r--  1 Taka  staff  7.9K Sep 27 17:26 presentation.aux
30354512 -rw-r--r--  1 Taka  staff  212B Sep 27 16:34 presentation.bbl
30347077 -rw-r--r--  1 Taka  staff  364K Sep 27 17:26 presentation.dvi
30601651 -rw-r--r--  1 Taka  staff   68K Sep 27 17:26 presentation.log
30347078 -rw-r--r--  1 Taka  staff  3.8K Sep 27 17:26 presentation.nav
30601653 -rw-r--r--  1 Taka  staff  180B Sep 27 17:26 presentation.out
30602520 -rw-r--r--  1 Taka  staff  485K Sep 27 17:26 presentation.pdf
30347082 -rw-r--r--  1 Taka  staff  22M Sep 27 17:26 presentation.ps
30347080 -rw-r--r--  1 Taka  staff    0B Sep 27 17:26 presentation.snm
30602679 -rwxr-xr-x@ 1 Taka  staff   58K Sep 27 17:38 presentation.tex
30601654 -rw-r--r--  1 Taka  staff  326B Sep 27 17:26 presentation.toc
takambp:Chapter 4 Taka$
```

Figure: ``ls -lhi'' output example

But what about the file names lengths impact on the directory structure?

But what about the file names lengths impact on the directory structure?

- **Simplest approach:** limit file-name length to, typically, 255 characters:
 - Approach is simple;
 - However: wastes a great deal of directory space:
 - Each directory entry needs to reserve 255 characters;
 - Few files have such long names.
 - For efficiency reasons: different structure is desirable.

But what about the file names lengths impact on the directory structure?

All modern operating systems support long variable-length file names:

- **Idea:** Give up the idea that all directory entries are the same size;

Idea: Give up the idea that all directory entries are the same size:

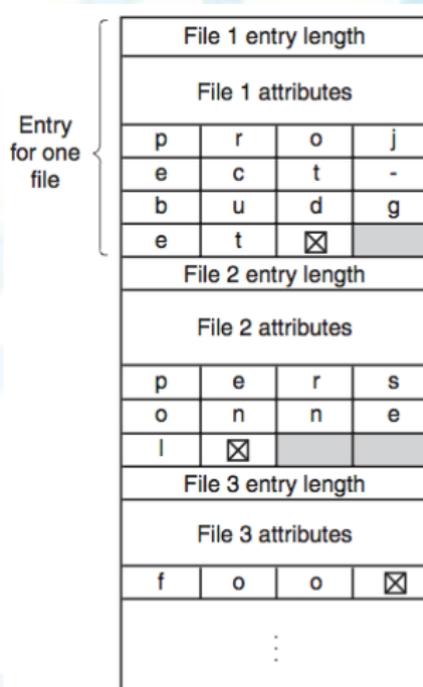


Figure: In-line handling of long file names (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- Each directory entry contains a fixed portion:
 - Starting with the length of the entry;
 - Followed by the attributes (fixed-length):
 - *E.g.*: Owner, creation time, protection information, etc...
 - Followed by the actual file name:
 - However long it may be...
- In this example we have three files:
 - project-budget, personnel, and foo.
 - Each file name is terminated by a special character:
 - Usually 0;
 - Represented in the figure by a box with a cross in it ☒

Can you see any problems with the previous approach? Any ideas?

Can you see any problems with the previous approach? Any ideas?

When a file is removed:

- a variable-sized gap is introduced into the directory:
 - Next file to be entered may not fit
- Problem is essentially the same one we saw with contiguous disk files,

Can you see any **other** problem with the previous approach? Any ideas?

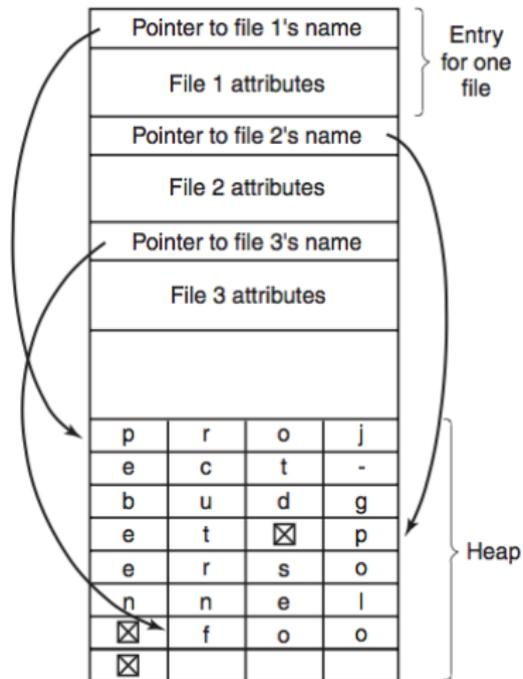
Can you see any **other** problem with the previous approach? Any ideas?

Single directory entry may span multiple pages:

- Page fault may occur while reading a file name;

Another way to handle variable-length names (1/2):

- Make directory entries themselves all fixed length;
- Keep the file names together in a heap at the end of the directory:



Can you see any advantage of using the heap method? Any ideas?

Can you see any advantage of using the heap method? Any ideas?

When an entry is removed:

- next file entered will always fit there;

Important:

- Heap must be managed;
- Page faults can still occur while processing file names;

There is one thing we still have not discussed:

How are directories to be searched? Any ideas?

There is one thing we still have not discussed:

How are directories to be searched? Any ideas?

Several possibilities:

- Search linearly from beginning to end for a filename;
 - Bad for extremely long directories;
- Search using a hash table in each directory:
 - Hashed based on the filename;
 - Faster lookup, but more complex administration.

Example

Consider the path: `/usr/ast/mbox`

How does the OS find the i-node information? Any ideas?

Example

Consider the path: `/usr/ast/mbox`

How does the OS find the i-node information? Any ideas?

- 1 Locate root directory i-node:
 - This is **always** a fixed place on disk;
 - Each directory entry contains: {filename, i-node}
- 2 I-node is read containing the info for `/user/`
 - Each directory entry contains: {filename, i-node}
- 3 Next i-node is read containing the info for `/user/ast/`;
 - Each directory entry contains: {filename, i-node}
- 4 Next i-node is read containing the info for `/user/ast/mbox`

File-system Management and Optimization

Making the file system work is one thing:

- Making it work efficiently and robustly in real life is something quite different;
- Guess what we will be seeing next ;)
 - Some of the issues involved in managing disks:
 - Disk-space management;
 - File-system performance;
 - Defragmenting disks;

Disk-space management

Files are normally stored on disk:

- Disk space management is a major concern to file-system designers.
- Lets have a look at some of the issues influencing file-system design:
 - Block Size;
 - Keeping Track of Free Blocks;
 - Disk quotas

Block Size

Two general strategies are possible for storing an n byte file:

- n consecutive bytes of disk space are allocated:
 - However, If a file grows, it may have to be moved on the disk;
 - Very slow operation...
- File is split up into a number of (not necessarily contiguous) blocks;
 - Most file systems chop files up into fixed-size **blocks**
 - Blocks need not to be adjacent;
 - File **fragmentation** may occur;

Blocks are an important part of the file system:

- They represent a fixed-length sequence of bytes;

But how can we choose an appropriate block size? Any ideas?

Blocks are an important part of the file system:

- They represent a fixed-length sequence of bytes;

But how can we choose an appropriate block size? Any ideas?

- **Large block size** means:
 - small files waste large amounts of disk space;
- **Small block size** means:
 - Most files will span multiple blocks;
 - Thus needing multiple seeks and rotation delays to read:
 - bad for performance;

In conclusion:

- If the allocation unit is too large we waste space;
- If the allocation unit is too small we waste time;

Again: But how can we choose an appropriate block size? Any ideas?

Studies have shown that making a good choice requires:

- having some information about the file-size distribution:

Studies have shown that making a good choice requires:

- having some information about the file-size distribution:

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Figure: Percentage of files smaller than a given size (in bytes) (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- For each power-of-two file size:
 - Each line lists % of all files \leq to it for three data sets;
- *E.g.:* in 2005: $\approx 59\%$ in the 2^{nd} data set were 4KB or smaller
- *E.g.:* in 2005: $\approx 90\%$ in the 2^{nd} data set were 64KB or smaller
- Median file size was 2475 bytes;

What conclusion can we draw from these data? Any ideas?

What conclusion can we draw from these data? Any ideas?

- With a 1KB block: $\approx 30\%$ – 50% of all files will fit
- With a 4KB block: $\approx 60\%$ – 70% of all files will fit

Example (1/2)

Consider a disk with:

- 1MB per track;
- Rotation time of 8.33 (7200 rpms);
- Average seek time of 5 msec;
- The time in milliseconds to read a block of k bytes is the sum of:
 - Seek time;
 - Rotational delay;
 - Transfer times;
 - *i.e.*: $5 + 4.165 + (k/2^{20}) \times 8.33$

Example (2/2)

Data rate for such a disk as function of block size:

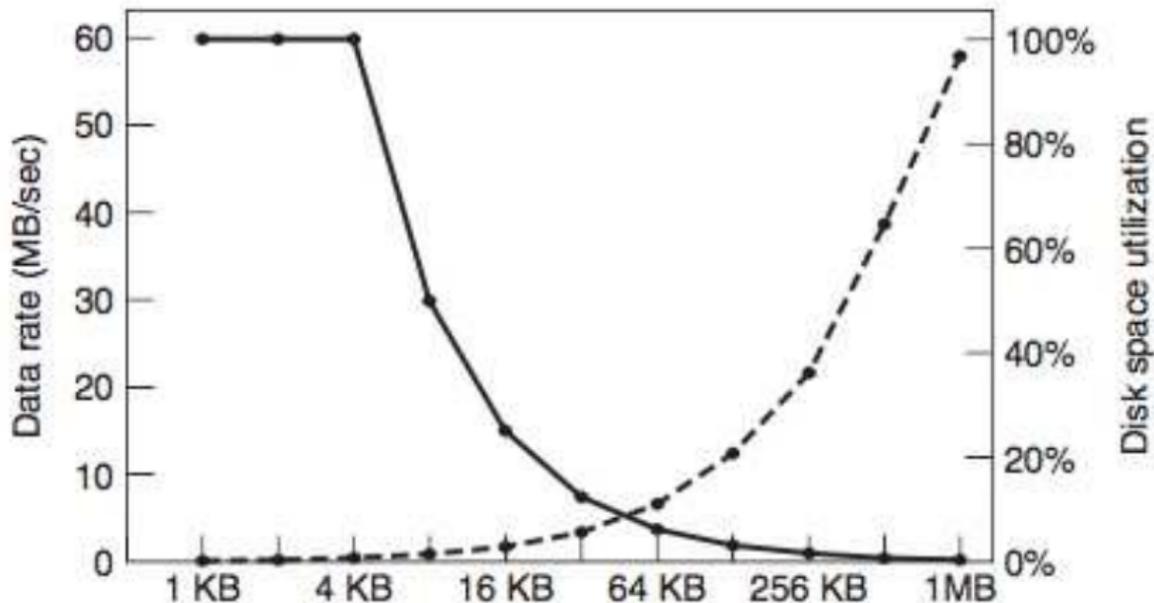


Figure: The dashed curve (left-hand scale) give the date rate of a disk. The solid curve (right-hand scale) gives the disk-space efficiency. All files are 4 KB (Source: (Tanenbaum and Bos, 2015))

From the previous figure (1/3):

- For simplicity: assume that all files are 4KB
- Solid curve shows the space efficiency as a function of block size;
- Dashed curve shows the data rate as a function of block size;

From the previous figure (2/3):

- **Dashed curve** can be understood as follows:
 - Block access time is dominated by the seek time and rotational delay;
 - *i.e.* $5 + 4.165 = 9.165$ msec are needed to access a block;
 - Therefore, the more data are fetched the better;
 - Hence, data rate goes up almost linearly with block size:
 - Until the transfers take so long that the transfer time begins to matter;

From the previous figure (3/3):

- **Solid curve** can be understood as follows:
 - With 4-KB files:
 - Four 1-KB blocks are used;
 - Two 2-KB blocks are used;
 - One 4-KB blocks are used;
 - Half 8-Kb blocks are used (50% efficiency)
 - Quarter 16-Kb blocks are used (25% efficiency)
 - In reality: some space is always wasted:
 - Not all files are an exact multiple of the disk block size;

What is the main conclusion you can draw from the previous figure? Any ideas?

What is the main conclusion you can draw from the previous figure? Any ideas?

Performance and space utilization are inherently in conflict:

- Small blocks are **bad** for performance, but **good** for disk utilization;
- For this reason: **No reasonable compromise is available!**
- Size closest to the two curves is 64 KB but:
 - Data rate is only 6.6 MB/sec;
 - Space efficiency is about 7%;
 - Neither of which is very good;

Historically:

- File systems have chosen sizes in the 1-KB to 4-KB range;
- But with disks now exceeding 1 TB:
 - Better to increase block size to 64 KB and accept wasted disk space;
 - Disk space is hardly in short supply any more;

Keeping track of free blocks

Once a block size has been chosen:

How does the OS keep track of free blocks? Any ideas?

Keeping track of free blocks

Once a block size has been chosen:

How does the OS keep track of free blocks? Any ideas?

Two methods are widely used:

- Linked-List of disk blocks;
- Bitmap of disk blocks;

Lets have a look at the **linked-list approach**:

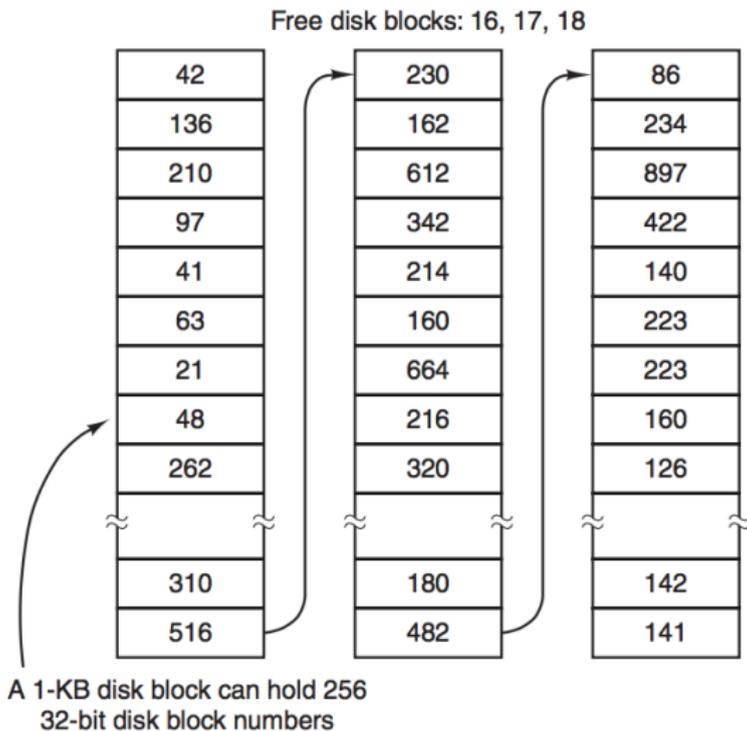


Figure: Storing the free list on a linked list. (Source: (Tanenbaum and Bos, 2015))

From the previous figure (1/2):

- Linked list of disk blocks:
 - Each block holding as many free disk block numbers as will fit;
 - Storage of the list requires three blocks: 16, 17 and 18
- *Example:* With a 1-KBytes block and a 32-bit disk block number:
 - Each list block holds numbers of $(2^{10} \times 8)/2^5 = 256$ free blocks.
 - One of these slots is required for the pointer to the next block:
 - **As a result:** only capable of describing 255 free blocks;

From the previous figure (2/2):

- Consider a 1-TB disk:
 - Then $2^{40}/2^{10} = 2^{30}$ blocks exist;
 - If each block in the list stores the addresses of 255 blocks
 - $2^{30}/(2^8 - 1) \approx 4$ million blocks will be required for the list;

Lets have a look at the **bitmap approach**:

1001101101101100
0110110111110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
⋮
0111011101110111
1101111101110111

Figure: Storing the free list on a bitmap. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- A disk with n blocks requires a bitmap with n bits:
 - Free blocks are represented by 1s in the map;
 - Allocated blocks by 0s (or vice versa)
- Consider a 1-TB disk with 1-KB blocks:
 - Then $2^{40}/2^{10} = 2^{30}$ bits are required;
 - These bits require around $2^{30}/(2^{10} * 8) \approx 130.000$ 1KB blocks to store;

Why does the value 8 appear in the previous calculation? Any ideas?

From the previous figure:

- A disk with n blocks requires a bitmap with n bits:
 - Free blocks are represented by 1s in the map;
 - Allocated blocks by 0s (or vice versa)
- Consider a 1-TB disk with 1-KB blocks:
 - Then $2^{40}/2^{10} = 2^{30}$ bits are required;
 - These bits require around $2^{30}/(2^{10} * 8) \approx 130.000$ 1KB blocks to store;

Why does the value 8 appear in the previous calculation? Any ideas?

- Each block has size 1-KB, *i.e.*: 1-K**Byte**
- **Conclusion:** bitmap requires less space than linked-list:
 - Uses 1-bit per blocks vs 32-bits...

But when is one approach better than the other?

But when is one approach better than the other?

- Both approaches require linear search so that is not it...

But when is one approach better than the other?

- Both approaches require linear search so that is not it....
- Bitmap approach always requires the same size:
 - regardless of how full the disk is...
- Linked-list approach will require less-size as the disk becomes full;

Disk Quotas

Multuser OS often provide a mechanism for enforcing disk quotas:

- Prevents people from monopolising too much space;
- **Idea:** System administrator assigns each user a maximum space (**quota**);
 - OS makes sure users do not exceed their quota;

How do you think an OS enforces user quotas? Any ideas?

How do you think an OS enforces user quotas? Any ideas?

Typical mechanism (1/2):

- When a user opens a file:
 - Any increases / decreases in file size will be charged to the owner's quota;
 - OS table contains the quota record for every user with a currently open file:
 - Even if the file was opened by someone else
- When a new entry is made in the open-file table:
 - Pointer to the owner's quota record is entered into it
 - Making it easy to find various limits;

How do you think an OS enforces user quotas? Any ideas?

Typical mechanism (2/2):

- Every time a block is added to a file:
 - Total number of blocks charged to the owner is incremented;
 - Check is made against both the hard and soft limits:
 - Soft limit may be exceeded, but the hard limit may not;
 - Appending to a file when the hard block limit has been reached:
 - Will result in an error.

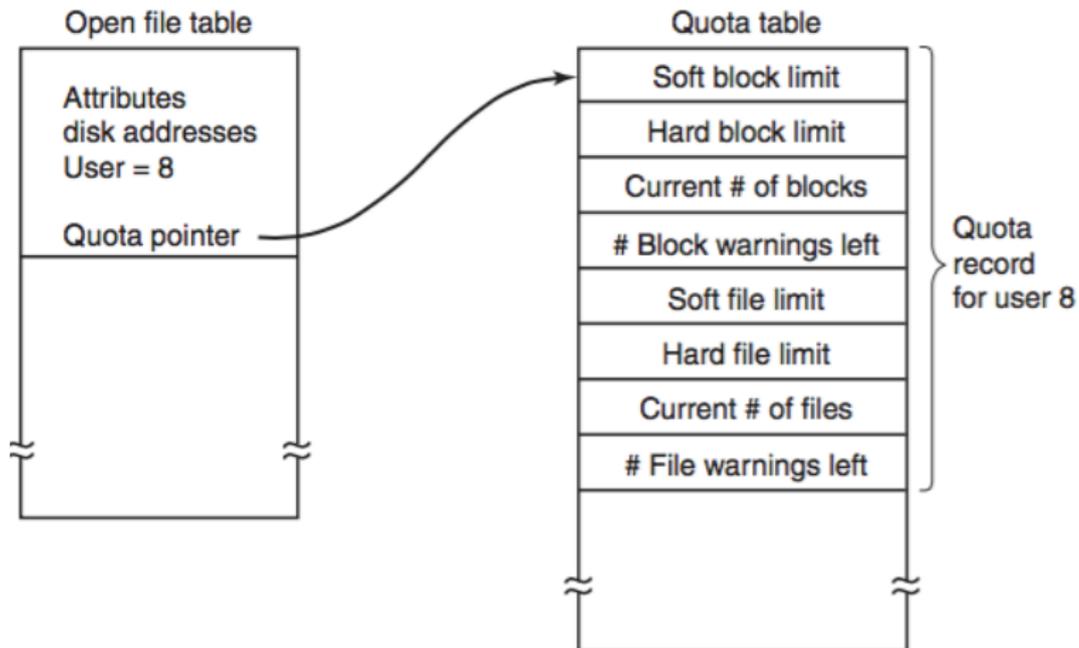


Figure: Quotas are kept track of on a per-user basis in a quota table. (Source: (Tanenbaum and Bos, 2015))

File-system performance

Access to disk is much slower than access to memory:

- Memory access is approximately a million times as fast as disk access.

As a result many file systems have been designed with:

- Various optimizations to improve performance.
- Lets have a look at some:
 - Caching;
 - Block Read Ahead;
 - Reducing disk-arm motion;

Caching

Technique used to reduce disk accesses: **block cache**;

- Collection of disk blocks kept in memory for performance reasons;
- Check all read requests to see if block is in cache:
 - If block \in cache:
 - Request can be satisfied without a disk access.
 - If block \notin cache:
 - Block is read into cache;
 - Then copied to wherever it is needed;

Typically there are many blocks in the cache:

- Need to quickly determine if a given block is present;
- Use a hash table: hash device and disk address;
- All the blocks with the same hash value are chained together

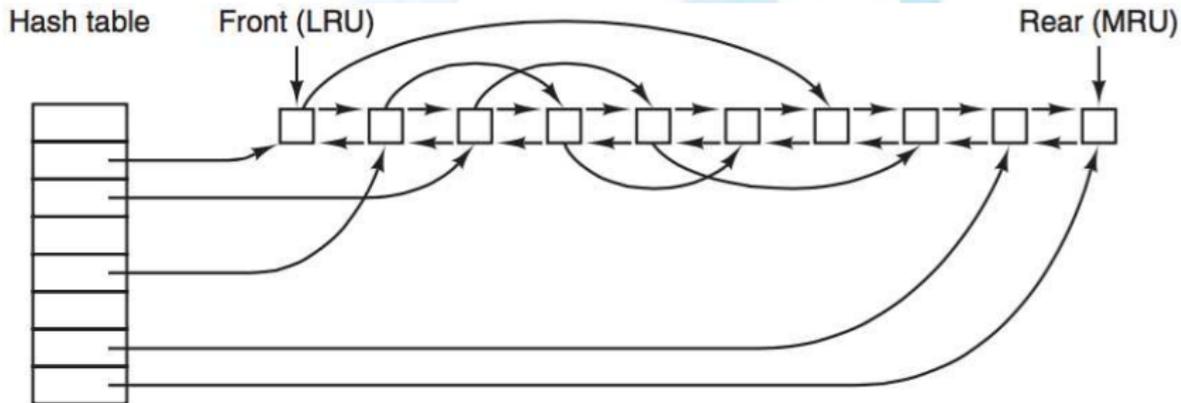


Figure: The buffer cache data structures. (Source: (Tanenbaum and Bos, 2015))

When a block has to be loaded into a full cache:

- Some block has to be removed:
 - And rewritten to the disk if it has been modified;
- Page-replacement algorithms can be used:
 - FIFO, LRU, LFU...

Block read-ahead

Second technique for improving performance:

- Get blocks into the cache before they are needed to increase the hit rate;
- Many files are read sequentially;
- When the file system (FS) is asked to produce block k in a file:
 - FS produces the k block;
 - FS also checks if block $k + 1 \in$ cache:
 - If block \notin cache FS schedules a read for block $k + 1$;
 - Hoping that when it is needed it will already be in cache;

Read-ahead strategy works only for files that are read sequentially:

- If a file is being randomly accessed:
 - **Read ahead does not help!**
 - In fact: hurts performance:
 - Blocks will be read unnecessarily;
 - Potentially useful blocks will be removed from cache;
 - Modified blocks evicted from cache will have to be written to disk;
 - Disk could be reading useful blocks;

Reducing Disk-Arm motion

Idea: Reduce amount of disk-arm motion:

- By putting blocks that are likely to be accessed in sequence;
 - Ideally contiguous, but in close proximity already helps;
- When an output file is written:
 - FS has to allocate the blocks one at a time;
 - Easy to do using a bitmap;
 - Harder to do with a list of free blocks;
 - List would need to be sorted;

Movements are relevant only for magnetic disks:

Do you know any other type of mass storage devices? Any ideas?

Movements are relevant only for magnetic disks:

Do you know any other type of mass storage devices? Any ideas?

Solid-state disks (SSD):

- No moving parts =);
- These are based on flash technology:
 - Random accesses are just as fast as sequential ones;
 - Many of the problems of traditional disks go away;
 - Unfortunately, new problems emerge =(
 - Each disk block can be written only a limited number of times;
 - Great care is taken to spread the wear on the disk evenly.

Defragmenting Disks

When the operating system is initially installed:

- Data are installed consecutively at the beginning of the disk;
- All free disk space is in a single contiguous unit following the installed files;

Can you see any problem with this structure as time goes on? Any ideas?

Defragmenting Disks

Can you see any problem with this structure as time goes on? Any ideas?

- Files are created and removed:
 - Disk becomes full of **holes**;
 - Non-contiguous empty space spread throughout the disk;
 - When a new file is created:
 - Blocks used for it may be spread all over the disk;
 - Giving poor performance.

Performance can be restored by:

- 1 Moving files around to make them contiguous;
- 2 Putting all free space contiguously;

Linux file systems like ext2 and ext3:

- Generally suffer less from defragmentation than Windows:
 - Due to the way disk blocks are selected;
- Manual defragmentation is rarely required;

SSDs do not really suffer from fragmentation at all:

- Defragmenting an SSD is counterproductive;
- Not only is there no gain in performance:
 - Writing to SSDs wears them out;
 - Defragmenting them merely shortens their life.

References I



Tanenbaum, A. and Bos, H. (2015).

Modern Operating Systems.

Pearson Education Limited.