Lecture 34 — File Allocation Methods

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 17, 2018

Allocation of Disk Space

There is a need to choose a strategy for how to allocate disk space.

There are three major ways that we could allocate the disk space to files:
Contiguous, linked, and indexed; each has its advantages and disadvantages.

Contiguous Allocation

Contiguous: a file occupies a set of contiguous blocks on disk.

So a file is allocated, starting at block b and is n blocks in size, the file takes up blocks $b, b+1, b+2, \ldots, b+(n-1)$.

This is advantageous, because if we want to access block b on disk, accessing $b+1$ requires no head movement, so seek time is nonexistent to minimal.

Contiguous Allocation

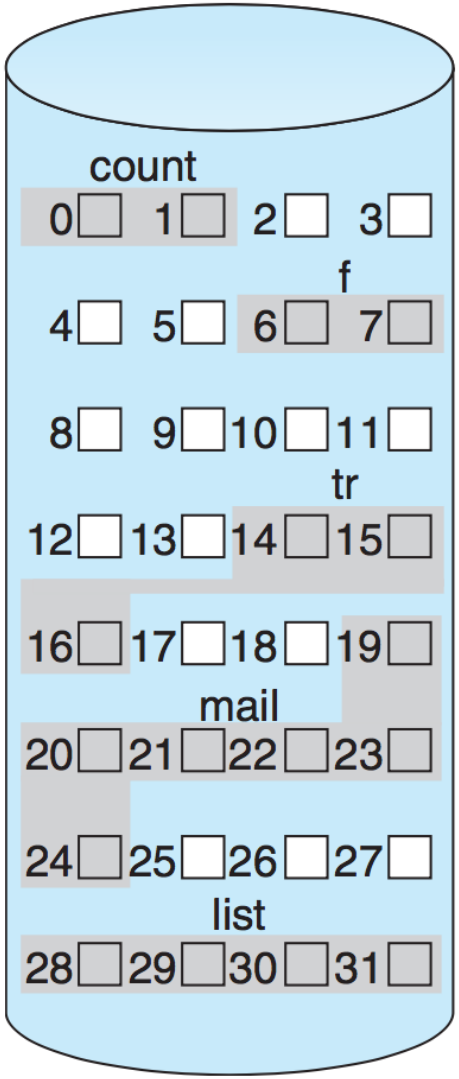
All that we need to maintain are b and n: the start location and length of the file.

Both sequential and direct access are very easy: the first block of a file is at b.

To access a block i at some offset into the file, it's at the base address b plus i.

Checking if the access is valid is also an easy operation: if $i < n$ then it is valid.

Contiguous Allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

This takes us back to a problem we have seen once before: the memory allocation problem.

If we need a memory block of size *N*:

(1) can we find a contiguous block of *N* or greater to meet that allocation?

(2) if there is more than one block, which one do we choose?

As before, we suffer the problem of external fragmentation, plus a bit of internal fragmentation in the last block of the file.

Contiguous Allocation and Compaction

There was also talk on the subject of memory allocation of compaction.

This was moving memory allocations around in memory to create some larger free spaces that can be allocated.

But for languages like C, it was not realistic, because of the impossibility of updating all pointers (unlike Java where we can update all references).

Compaction

We can do compaction on disk, but it's slow and computationally expensive.

Doing compaction can be done when the system has nothing else to do (idle priority operation) or on some schedule when it would be minimally disruptive.

The middle of the night...?

Contiguous Allocation

Another problem: how much space is a file going to take?

If it is just a copy-paste operation, the copy is the same size as the original.

When a user opens a new document, how big will it be?

If we allocate too little space, we may be able to tack on space at the end, or that block may be allocated, forcing us to move the file and reallocate it.

If the value we choose is too large, then significant space will be wasted for small files (and many files tend to be relatively small).

Linked Allocation

Linked allocation is a solution to the problems of contiguous allocation.

Instead of a file being all in consecutive blocks, we maintain a linked list of the blocks, and the blocks themselves may be located anywhere on the disk.

The directory listing just has a pointer to the first and last blocks (head and tail of the linked list).

Linked Allocation

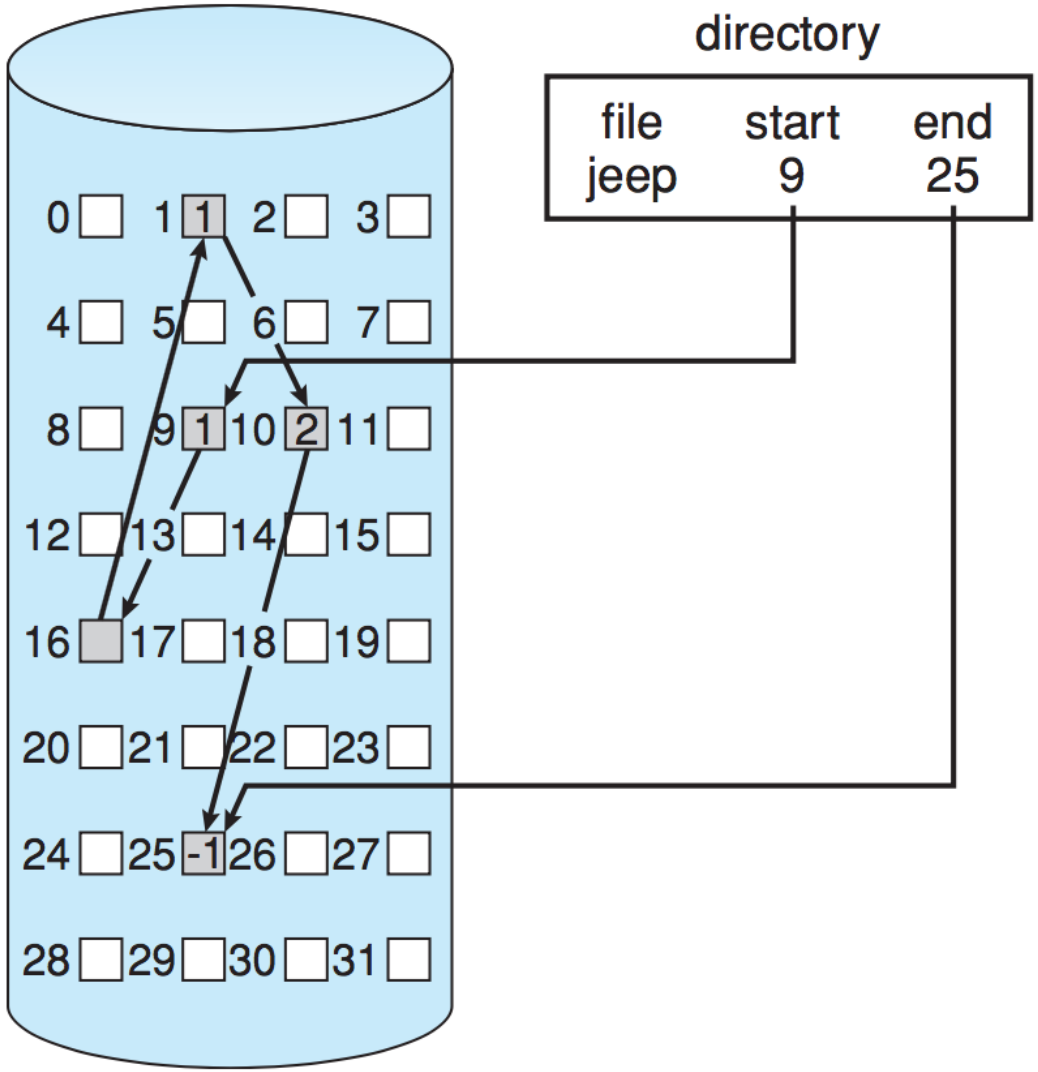
If a new file is created, it will be created with size zero and the head and tail pointers are null.

When a new block is needed, it can come from anywhere and will just be added to the linked list.

Thus, compaction and relocation are not really an issue.

Unfortunately, however, accessing block *i* of a file is no longer as simple as computing an offset from the first block; it requires following *i* pointers (a pain).

Linked Allocation



Linked Allocation

A possible solution to the problem of following so many pointers (and the overhead of maintaining so many) is to group up the blocks into clusters.

A cluster is comprised of, say, four blocks.

Then we waste less memory maintaining pointers and it improves disk accesses because there is less seeking back and forth to various disk locations.

File Allocation Table (FAT)

One variation of linked allocation is the File Allocation Table (FAT).

Used in Windows before NTFS came in with Windows NT/2000/XP/7/8...

FAT32 hangs on these days for USB Flash Drives.

File Allocation Table

At the beginning of the disk, there is a table to maintain file allocation data.

The table has one entry for each block and is indexed by block number.

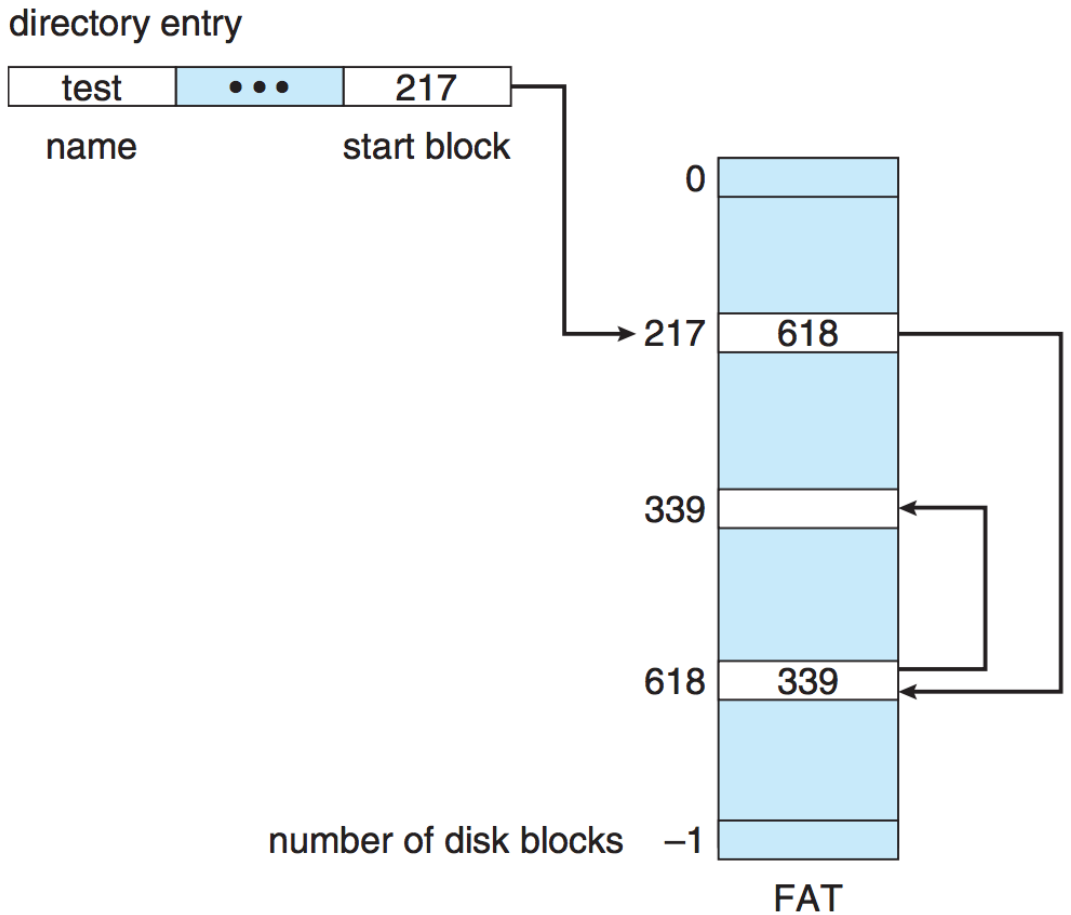
It works like a linked list; the directory entry has the first block of the file and the table entry under that block number has the index of the next block.

The chain continues until the last block; there is a special end-of-file value.

An unused block has a table value of 0.

Thus, to allocate a new block, find the first 0-valued entry, and replace the previous end-of-file value with the address of the new block.

File Allocation Table



The FAT itself should be cached in memory, otherwise the disk is going to have to seek back to it unbearably often.

Indexed Allocation

If we stuck to pure linked allocation, we still have the problem that accessing some part in the middle of the file is a pain.

We have to follow and retrieve a lot of pointers to the different blocks.

The idea of indexed allocation is to take all the pointers and put them into one location: an index block.

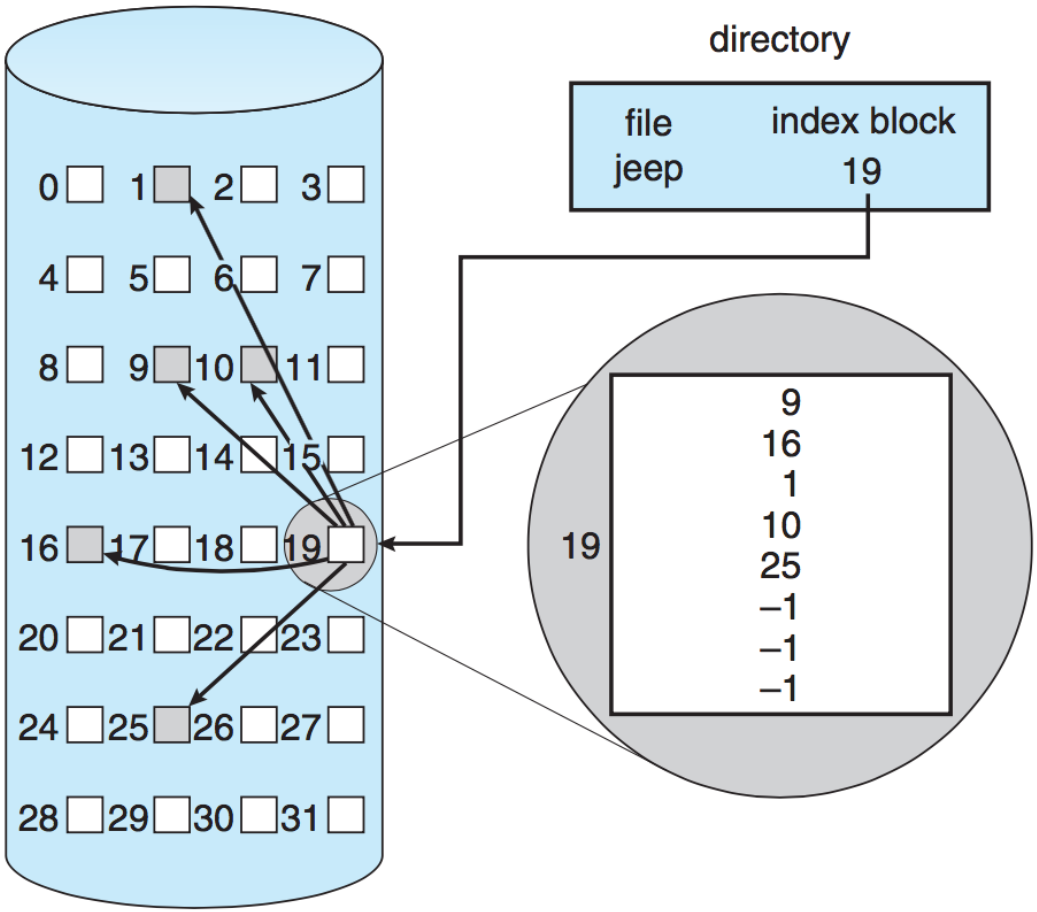
Indexed Allocation

So, the first block of the file contains a whole bunch of pointers.

To get to block *i*, just go to index *i* of the index block and we can get the location of block *i* much more efficiently than we could in linked allocation.

All pointers to blocks start as null, and when we add a new block, add its corresponding entry into the index block.

Indexed Allocation



Block Size

Like many of the other systems we have examined, there is a need to make a decision about the size of a block.

If a file needs only 1-2 blocks, one whole block is allocated for the pointers which contains only 1-2 entries.

That suggests we want the index to be small, but what if we need more pointers than fit into one block? There are a few mechanisms for this.

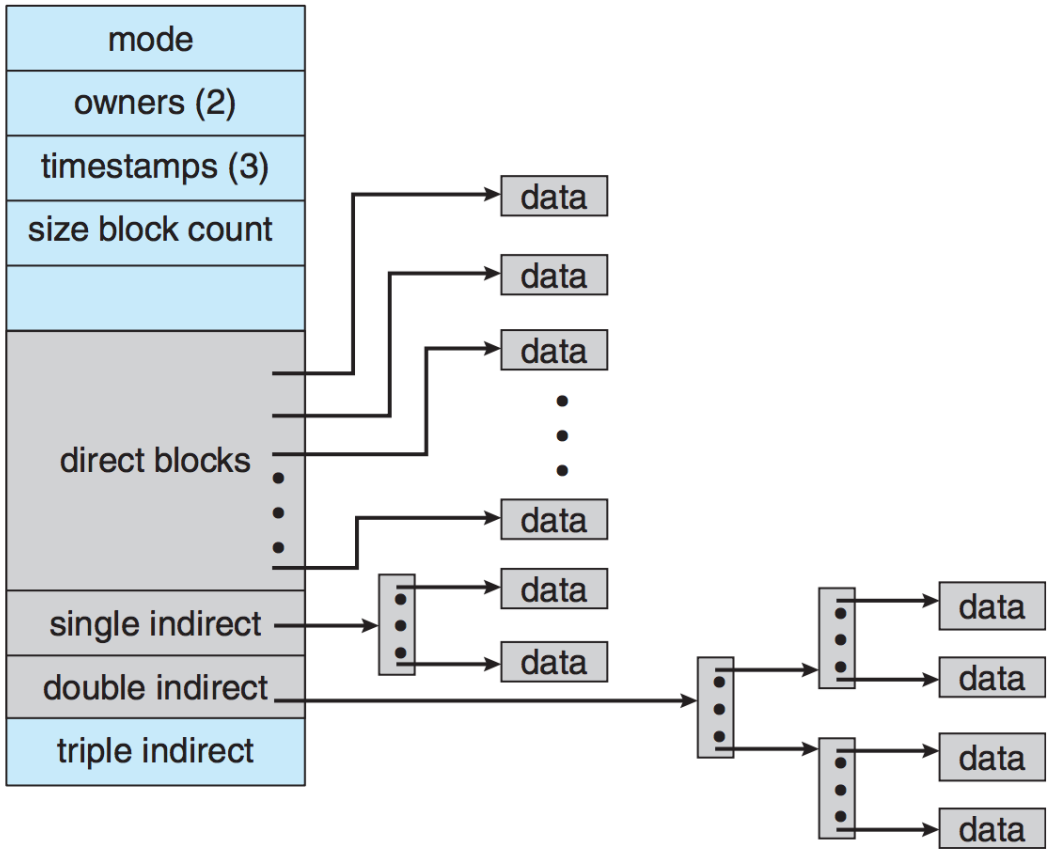
Block Size

What if we need more pointers than fit into one block?

1. **Linked Scheme**
2. **Multilevel Index**
3. **Combined Scheme**

UNIX inodes

We can finally show a visual representation of an inode.



Free Space Management

As with memory, the system will keep track of the free space available.

Bit vectors work just the way we would expect: create a structure in memory where a bit represents a block.

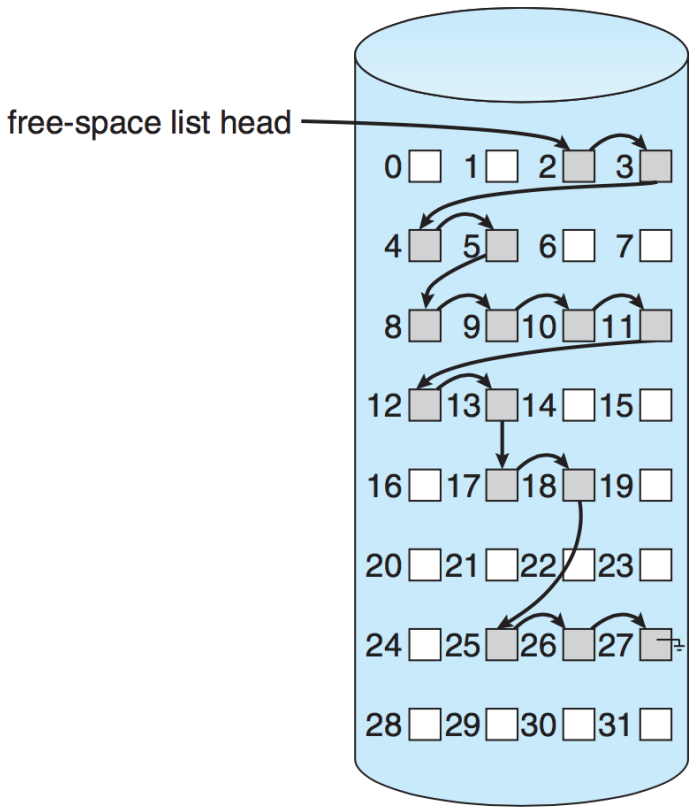
If it is free, the bit is 1; allocated is 0.

So a bit vector looks like `01001101110111111000000....`

The problem with this strategy is, of course, that the bigger the disk, the more overhead is needed to store this bit vector.

Free Space Management

The next rather obvious approach is a linked list: the head points to the first free space block, and we can traverse the list to get to the next free block.



Grouping

Grouping: the first free block will be used to store the addresses of n free blocks, the first $n - 1$ of which are actually free.

The last block contains a block with another set of n free blocks.

So if we need a large number of free blocks, we can find them quickly, instead of walking through the list one block at a time, which takes a lot of disk accesses.

Counting

Counting is a slight improvement on this where we also store a number k of free contiguous blocks after each address.

So, if block 27 is followed by three consecutive free blocks, instead of having 27, 28, 29, 30 as entries, it will show (27, 4).

The entries may be stored in a balanced tree for efficient operations.

Preallocation

The efficient use of a disk depends on the disk-allocation/directory algorithms.

UNIX inodes, for example, are preallocated, so even a disk containing no files has some of its space taken up by the inodes.

Preallocation & distribution of inodes improves the file system performance.

UNIX allocation and free-space algorithms keep the file data near the inode, where possible, to reduce seek time.

Consistency Checking and Journalling

Unfortunately, an error, crash, or power failure or something similar may result in a loss of data or inconsistent data in the file system.

The directory structures, pointers, inodes, FCBs, et cetera are all data structures and if they become corrupted it may lead to serious problems.

Consistency Checking

We could check for inconsistent data periodically (e.g., on system boot up) and many operating systems do so.

This is, of course, an operation that will consume a very large amount of time while the whole disk is scanned.

UNIX: `fsck`. Windows: `chkdsk`/`scandisk`.

These tools will look for inconsistent states (e.g., a file that claims to be 12 blocks but the linked list contains only 5) and will attempt to repair it.

Its level of success depends on the nature of the problem and the implementation of the file system.

Preventing the Problem

Obviously we would like to prevent the problem, if we can.

Atomic operations: an operation should either succeed completely, or not take place at all.

This approach is used in the Windows NTFS system as well as Mac OS HFS+.

Transactions

We might be familiar, at this point, with the concept of the transaction.

Before making any changes, make a list of all the things we plan to do.

Then do the things written down.

Then we consider the transaction complete.

Transactions

All metadata changes are written sequentially to a log file; once the changes are written to the log, control returns to the program that requested the operation.

Meanwhile, the log entries are actually carried out.

As changes are made, a pointer is updated to indicate which of the log entries have really happened and which have not.

When an entire transaction is completed, it is removed from the log file. If the system crashes, the log file will contain zero or more transactions.

If zero, there is no problem: nothing was in progress at the time of the crash.

If there are some, then the transactions were not completed and the operations should still be carried out.

If a transaction was aborted (not committed), we walk backwards through the log entries to undo any completed operations.

Thus, we go back to the state before the start of the transaction.

Data Loss

Even though a particular write may not have taken place because of a crash, resulting in some data loss, the system will always remain in a consistent state.

As a side benefit, we can sometimes re-order the writes to get better performance (e.g., schedule them to get better disk utilization).

ZFS Approach

The approach in the Solaris ZFS approach is similar, but not identical.

Blocks are never overwritten with new data.
    Instead, a transaction writes all data and metadata to new blocks.

Only when the transaction is complete, any references to the old blocks are replaced with the location of the new blocks.

Then the old pointers and blocks can be cleaned up (reused or disposed of).

Example: NTFS (Windows File System)

NTFS uses several different storage levels:

1. **Sector**
2. **Cluster**
3. **Volume**

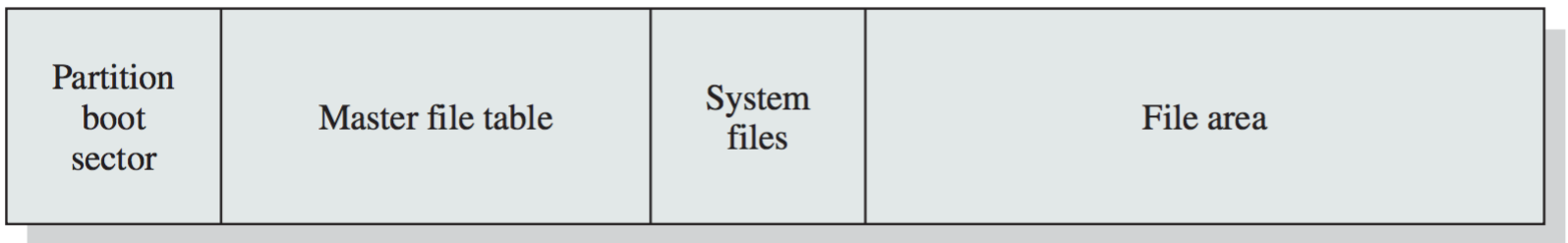
The cluster is the fundamental unit of allocation of NTFS.

This allows the file system to be independent of the size of physical sectors on the disk.

A volume contains file system information, a collection of files, and free space.

The logical volume may be some of a physical disk, all of one, or spread across multiple physical disks.

NTFS Volume Layout



NTFS System Files

The Master File Table (MFT) contains information about all the files and folders.

A block is allocated to system files that contain important system information:

1. **MFT2**
2. **Log File**
3. **Cluster Bitmap**
4. **Attribute Definition Table**

NTFS File and Directory Attributes

Attribute Type	Description
Standard information	Includes access attributes (read-only, read/write, etc.); time stamps, including when the file was created or last modified; and how many directories point to the file (link count)
Attribute list	A list of attributes that make up the file and the file reference of the MFT file record in which each attribute is located. Used when all attributes do not fit into a single MFT file record
File name	A file or directory must have one or more names.
Security descriptor	Specifies who owns the file and who can access it
Data	The contents of the file. A file has one default unnamed data attribute and may have one or more named data attributes.
Index root	Used to implement folders
Index allocation	Used to implement folders
Volume information	Includes volume-related information, such as the version and name of the volume
Bitmap	Provides a map representing records in use on the MFT or folder

NTFS Journalling

NTFS uses journalling to ensure that the file system will be in a consistent state at all times, even after a crash or restart.

There is a service responsible for maintaining a log file that will be used to recover in the event that things go wrong.

Note that the goal of recovery is to make sure the system-maintained metadata is in a consistent state; user data can still get lost.

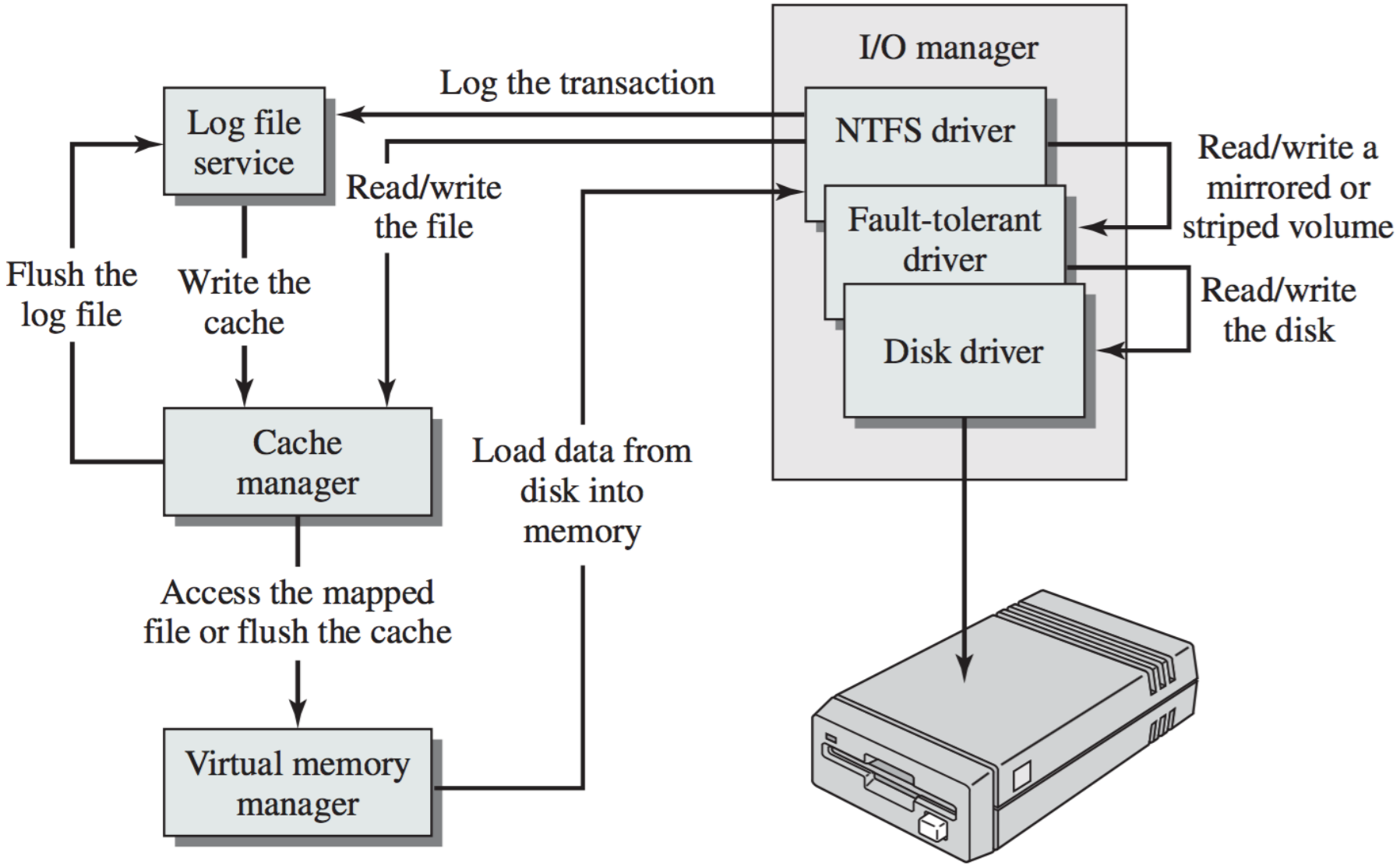
This was a Microsoft design decision.

NTFS Journalling Implementation

The actual implementation of journalling:

1. Record the change(s) in the log file in the cache.
2. Modify the volume in the cache.
3. The cache manager flushes the log file to disk.
4. Only after the log file is flushed to disk, the cache manager flushes the volume changes.

Visual Overview of NTFS



... Why there is a floppy disk drive... I have no idea...

Backups

One final note about hard disk drives.

We often think of them as places for permanent storage of data, but hard drives can and do fail.

So please, take backups of important data.