File Allocation Transcript

File Allocation

Hello, and welcome to this lesson on **File Allocation**.

The file allocation problem is about how to actually store a collection of files on the fixed-sized blocks that are proposed by the disk.

As we will see, the allocation of blocks to files has a significant effect on the file system performance. So we will first explore the wide variety of requirements that need to be addressed when designing a file allocation system.

We will then look at a number of these approaches, starting with the simplest one, **contiguous allocation,** where files are stored each on a contiguous sequence of disk blocks. We will see that this approach, while simple, is actually quite poor except for particular applications such as multimedia playback in CDs or DVDs.

Another approach is **linked allocation**, which appears in quite a number of variations.

One such variation, the **File Allocation Table** or **FAT**, was originally developed for early PCs and has survived until today, where it is still popular for portable storage media.

We will conclude with **indexed allocation**, a variation of which is used in UNIX and its derivatives since the mid-seventies.

At the end of this lesson you will know the most common file allocation techniques and you will be able to explain their performance trade-offs. You will also be able to implement your own file allocation scheme for a simple operating system.

Let’s begin.

The File Allocation Problem

Every operating system that provides support for pervasive storage has the problem that it has multiple files, while each are sequences of potentially variable-sized records, which need to be stored on one or more storage devices, which in turn can be viewed as large arrays of fixed-sized blocks.

The question now is, how do you actually store those file records in the provided blocks? This problem is one of “**file allocation**”.

File allocation must be done in a way that satisfies a variety of requirements.

A rather trivial one is that we need to use the space on the storage device effectively. We should not unduly waste space on the device. One particularly poor approach to file allocation in this respect could be to make do without multiplexing the storing of multiple files on a single device, and store each file on a different device. This approach would clearly not be effectively using the space.

In addition to space considerations, the performance of file operations must be supported as well.

For example, it should be easy and quick to perform random access of locations in a file. Except for contiguous allocation, where the next block allocated to a file is implicitly known; it's the next block after the current one; this next block must be identified, which may require the system to first find this information by reading other blocks on the device. Since access to each block on a disk is very expensive, simple read operations can lead to multiple disk accessed simply to find the correct block on the disk.

Sequential access to the file, where the file is traversed from the beginning to the end, needs to be efficiently supported as well.

Modifications to files must be supported efficiently, as well as the creation and/or deletion of files.

Blocking

In general, file allocation is simplified because files at this level of the system are not really treated as sequences of records anymore, but rather as sequences of fixed-sized blocks, which typically have the same size as disk blocks.

This is called "**blocking**" and some applications, data base systems for example, take advantage of blocking by aligning their data structures with block boundaries.

Files at the blocking level are seen as sequences of blocks, which can be either cached in memory or moved to and from disk.

Some designers call the blocks on the disk "**physical blocks**", while the blocks in files after blocking are called "**logical blocks**".

Contiguous Allocation

As we pointed out earlier, the simplest file allocation scheme is contiguous allocation, where files are mapped onto sequences of adjacent physical blocks on the disk.

If we do that, we need a very simple table to store, at which physical block a file begins, and what its length is in blocks.

This table could be stored at the beginning of the disk, in the first few blocks. These blocks would be read into memory whenever the operating system boots or mounts the disk.

The table would be written back to disk when the system shuts down or unmounts the disk, or more frequently to avoid problems when the system unexpectedly crashes.

Here we have a file that starts at block 4 and is 5 blocks long.

File 2 starts at block 10 and is 2 blocks long. (5)

While File 3 starts at block 16 and is 10 blocks long.

This allocation method has several advantages.

For example, it minimized the head movement on the disk. This is because a file is stored on a set of neighboring blocks, and so a traversal of the file, for example, has the head step slowly through the tracks containing track data.

In general, it is very easy to access the file sequentially: one starts at the first block and keeps accessing subsequent blocks until one reaches the end. Random accesses are easy as well. If a thread wants to access the Nth byte in a file, it simply divides N by the block size and adds the start block to know which block to read. By computing N modulo, the block size the system knows which byte inside the block to access.

The disadvantages of the scheme are significant for general workloads, however. (10)

First, it is difficult to make modifications to a file that change its size, such as deleting or adding records.

Let’s assume that we want to add to File 1 some data after its block B.

In order to do this, we need to grow the size of the file by one block, and mark the new size, now 6 in the file table. (15)

We then copy the blocks after the insertion point over one by one and write the new block into its place.

If we now need to add a new block, G in this case, we have a problem, as we cannot grow the file any further without overwriting blocks of the next file.

The problem is that the maximum size of a file needs to be known a priori. (20)

This all leads to external fragmentation. There is space for to grow existing files or create new ones. The problem is that the space is not contiguous.

In the case of disk blocks, this is easier to fix than for memory, because we can just copy the blocks to a new location and update the file table. For example, the system can periodically “**de-fragment**” the available blocks by moving the file blocks to generate long sequences of free blocks.

One common way to deal with files that grow is to pre-allocate blocks.

If one has blocks pre-allocated, one does not need to worry about running into other files when appending to a file.

The problem with pre-allocation, of course, is that it introduces internal fragmentation.

Linked Allocation

An approach that does not suffer from external fragmentation is the so-called "**linked allocation**".

The idea is to store the file's logical blocks wherever one finds an available disk block, possibly scattered all over the disk, and then to link each block to the next one by a next pointer.

The file table now contains the number of the first block of the file, and possibly the last block as well.

The advantages of linked allocation, in addition to the elimination of external fragmentation, are that blocks can be very easily deleted or inserted. (5)

If we want to insert logical block C after B, we first find a free block, in this case Block 13 and write the new logical block to it.

After we update the pointers, the new block is inserted.

Also, there is no a-priori upper limit on the file size. The file size is limited by the number of available blocks.

Linked allocation has a number of disadvantages, however. (10)

First, random access to an arbitrary location in the file is very difficult and expensive. In order to know where the logical block with the requested location in the file is stored, one has to read all the previous logical blocks of the file from the beginning in order to work one's way to the correct block. Since each block read is expensive, this makes for very inefficient random access.

Sequential access is not cheap either, because the subsequent blocks of a file can be scattered all over the disk. This means that the disk head has to travel a lot as we traverse the file.

Also, maintaining the next pointer in the logical block is very cumbersome. First, it uses up space, but it also reduces the available size of the logical block from a power-of-two to some other value, which greatly complicates keeping track where one is in the file.

Finally, reliability is a major problem.

For example, if the disk head plows into the platter and makes Block 16 unreadable, then we (15) loose the pointer to the next logical block.

As a result we lose all the remaining blocks of the file.

Variations Linked Allocation

Here are many variations of the basic linked allocation approach.

A popular one is to maintain the pointers in a separate list, which one loads into memory at system startup or when the disk is mounted.

For example, the system keeps an array of next pointers.

The file starts at Block number 9 and continues at block number 4.

The next block is block number 13 (5) and so on...

[..] until we reach block 23, which is the last block.

We mark this setting the next pointer to negative 1.

If we have another file stored in blocks 6 and 7, this is reflected in the array by having entry 6 containing the value 7, thus pointing to block seven, and the entry in block 7 be negative 1.

Example: FAT (File Allocation Table)

An example of such a linked allocation with a separate list is the very popular File Allocation Table of FAT approach, which has been used for many years in Windows products and is still used in many portable media, such as USB drives or SD cards for cameras and smart phones.

As an example, we will look at **FAT-16**, which means that the file allocation table has 16-bit entries. There are also 12-bit and 32-bit versions, called FAT-12 and FAT-32, respectively.

To keep the number of blocks manageable, blocks are combined into fixed-size clusters, which can vary from a single block to a few dozen blocks in size. Instead of linking up individual blocks, we link up clusters. The allocation table therefore has one entry per cluster.

In the following we assume that clusters have size 1 blocks.

The file allocation table is an array of 16-bit entries, of which we show 30 here.

The first entry contains an identifier for the file allocation table. (5)

The second entry contains the identifier used to mark the last block in a sequence. In this case the value is hex FFFF, which is the same as -1.

The following entries are marked as reserved in some way to indicate that these physical blocks should not be used to store files. This is because these blocks (marked in light blue) are used to store the file allocation table on disk.

For our file, we store the next pointer in Entry 9, which points to Block 4.

Entry 4 points to Block 13, and so on.

The last entry in the list, Entry 23, is set to -1 to indicate the end for this file. (10)

If a second file is stored on the disk, this is stored on the allocation table as well.

To create a new file, we find a free block, and mark it in the allocation table.

The logical block of the new file is stored on the disk.

We grow the file by finding a new block and linking it to the current in the allocation table. (15)

The same happens when we grow the file further.

One interesting aspect is that the file allocation table also acts as free-block-table that allows us to identify free blocks. Bad blocks can be directly marked in the allocation table as well.

When we shut down the system or unmount the file system, the file allocation table is stored on the reserved blocks on disk.

Please note that this is a simplified picture of the FAT file system, as we ignore a number of complications.

For example, in addition to the file allocation table, we have to store other information at the beginning of the disk, such as a boot sector and information about the root directory and other.

Also, in many versions of the FAT file system the file allocation table is stored twice on disk. This is in order to protect it from accidental damages to physical blocks that contain the allocation table. If one block goes bad, the allocation table can be reconstructed from the other copy.

Indexed Allocation

The so-called “**indexed allocation**” scheme stores the location of physical blocks of a file in a separate so-called index block.

Let’s say that we use Block 7 as index block for our file.

The content of the index block is an array of block numbers, where the number at index I denotes the physical block where Logical Block I is stored.

In this case, the 6th block of the file is stored in physical block 23.

The file table now contains the number of the index block. For file 1, this is Block 7. (5)

Indexed allocation has two advantages.

First, it is great at supporting random access.

We divide the location in the file by the block size and look up the block number in the index. We then use the remainder to get to the actual location in the physical block.

Second, there is no external fragmentation.

We can therefore say that indexed allocation combines the best from continuous and linked allocation.

One problem with indexed allocation is the internal fragmentation in the index blocks.

Because the index blocks need to be dimensioned to accommodate the largest file size, they have to be quite large; meaning that we need to allocate multiple blocks on disk per index block. Large index blocks leads to increased internal fragmentation. (10)

Supporting Large Files: Linked Index Blocks

One way to support very large files without growing the index blocks is to have small index blocks and link them into chains as the file grows in size.

- **We start with a single index block.**
- **The first entry points to the first physical block. We call this a data block.**
- **The next entry points to the second data block, and so on. (5)**
- **The last entry is used point to the next index block.**

As the file grows, entries in the second index block are used to point to data blocks.

The last entry is again used to point to the next index block, and so on.

The problem with linking index blocks is that the chain of index blocks can become large for very large files, thus making random accesses expensive again.

Supporting Large Files: Multilevel Indexing

An alternative is to use multilevel indexing.

A single index block, in this simplified case with 5 entries, can manage files that are up to 5 data blocks long.

If we instead have the entries in the index block...

[…] point to index blocks instead of data blocks, then those index blocks point to five data blocks each, which means that we can manage files that are up to 25 blocks long.

We call the index blocks on the right the **first-level index blocks**, and the single index bock on the left the **second-level index blocks**.

If we now have the first-level index blocks

[…] point to index blocks as well instead of data blocks, then we can support files that are 125 data blocks in size.

We now have first-level, second-level, and third-level index blocks.

One disadvantage of this multilevel approach is that even for small file sizes, we have to look up three index blocks to get to the right data block, which can be quite expensive if we don't cache the index blocks.

Index Block Scheme in UNIX

In the mid-seventies, the designers of the first UNIX file system used a multi-level index scheme that is very flexible in accommodating small and large file sizes.

In the data structure that is used to manage a file (we will later call this an i-node) the system maintains an index array with 13 entries.

The first 10 entries each point to a data block.

This allows the system to manage files up to 10 blocks in size.

We call these entries "**direct indices**".

If the file grows beyond 10 blocks,

The system uses an additional entry, which is called a "**single indirect index**".

This entry points to an index block, whose entries point to data blocks. If we use blocks that are 512B in size, and an entry is 4Byte wide, then we can store 128 entries in an index block. Using the single indirect index, the system can manage files up to 138 blocks in size. At 512 a block, this would make for files that are 69kB long.

If the file grows beyond that size, (5) an additional entry in the index array of the i-node is used, which we call the "**double indirect**" index.

This index points to an index block, whose entries each point to an index block, whose entries then turn point to data blocks. Now we can manage files that are 128 squared times 512 Byte plus the 69kB from above in size, which sums up to 8MB plus 69kB.

If the file grows beyond that, a triple-indirect index is used (10), which points to an index block, whose entries point to index blocks, whose entries again point to index blocks, whose entries finally point to data blocks.

Now we can manage files that are 128 cubed times 512B plus the 8 odd MB from before, for a total of up to a bit over 1GB! Note that 1GB in the mid-seventies was orders of magnitudes more memory than anybody could imagine in their wildest dreams.

The UNIX index block scheme supports random access quite effectively.

Say we have a file system with 1kB blocks and we want to access byte offset 9'000 in the file.

9000 divided by 1K is 8, so we look up entry 8.

Say that this entry points to block 367.

We look up data block 367 and use the remainder from the earlier division, which is 808, as offset into the data block to get to the data.

For larger offsets this is a bit more complicated.

Let's repeat the exercise with offset 350'000.

350'000 divided by 1024 is 341. We cannot use direct indexing, which can used for blocks up to 9. We cannot use single indirect indexing because this supports only up to 266 blocks.

So we use double indirect indexing.

The second-level index block is located in Block 9156.

We use the first index, which gets us to the first-level index block, which is stored in Block 331.

Here we used index 341-266, which makes 75, to get the number of the data block.

The data block is stored in Block 3333, at offset 816.

File Allocation Conclusion

With this we have come to the conclusion of this lesson on File Allocation.

All file systems have to deal with how to store the logical blocks of the files onto the physical blocks provided by the storage device.

We have looked at a number of allocation methods, starting with "**contiguous allocation"** to linked allocation, including the venerable file-allocation-table or FAT approach, to indexed allocation.

We have finished the lesson by looking at how UNIX has historically been using a form of multilevel index blocks to efficiently manage both small files as well as very large files.

We truly hope that you enjoyed this lesson.

You are now in a position to describe and implement most traditional file allocation techniques as part of a simple operating system.

Thank you for watching.