# OPS File Systems

## Handout Introduction

These notes were originally written using (Tanenbaum, 1992) but later editions (Tanenbaum, 2001; 2008) contain the same information.

## Introduction

Before we look at how a file system can be implemented we are going to take a brief look at filing systems from a users point of view. Most of this will probably be familiar to you but we cover it so that we have a sound basis for the discussions that follow.

We use files almost without thinking of why we need them. We can summarise why we need to use files as follows.

- It allows us to store data between processes. If we simply held data in main memory that data would be lost when the process terminated and certainly when the computer is switched off.
- It allows us to store large volumes of data. Certainly, larger volumes of data than can be stored in memory.
- Using files allows more than one process to access the data at the same time. Although, this is possible using a shared address space it is often more convenient to store data in a file and allow processes to access it as and when they require.

### *File Naming*

Different operating systems have different file naming conventions.
For example, MS-DOS only allows an eight character filename (and a three character extension). This limitation also applies to Windows 3.1 (which, as we said in an earlier handout is not really an operating system – it is just a graphical user interface loaded on top of an operating system).
Windows 95 and Windows NT allow file names up to 255 characters (although the full path name is only allowed to be a maximum of 260 characters).

In addition to the limits of the length of a filename, there are also limits as the characters that can be used. MS-DOS, for example, does not allow the space character as part of a filename. Windows 95/NT does allow this character.
Other characters which are (typically) illegal in filenames include '?', '/', '\' and '*'. The reasons are that these characters have special significance to the operating system. For example, the asterix and the question mark can be used a wildcard characters when searching for files and the slash characters are used as directory separators.

Some operating systems also distinguish between upper and lower case characters. For example, to MS-DOS, the filename ABC, abc, and AbC all represent the same file. To UNIX these would all represent different files.

### *File Extensions*

In many operating systems (typically operating systems which run on PC's) the filename is made up of two parts; separated by a full stop. The part of the filename up to the full stop is the actual filename. The part following the full stop is often called a ***file extension***.
In MS-DOS the extension is limited to three characters. Under UNIX and Windows 95/NT they can be longer.

File extensions are used to tell the operating system what type of data the file contains. Or, to put it another way, the file extension ***associates*** the file with a certain application so that if the user opens the file using the default application then the operating system knows which application to load.

As an example, under the Windows environment a file with an extension of 'DOC' is likely to be a file that was created by Microsoft Word. If the user double clicks on the file then the operating system (by virtue of the registry in the case of Windows 95/98/NT) knows that a file extension of 'DOC' is associated with the Microsoft Word Processor.
But, there are no hard and fast rules. Using tools provided with the operating system (Explorer in the case of Windows 95/98/NT) the user is able to change the file associations. Many users will change the file association for the 'TXT' file extension. By default these files (ASCII text files) will be opened by an operating system supplied program called Notepad, which is a simple text editor. Many users would prefer to edit text files using their word processor because it provides many more facilities. Therefore, they change the file association of 'TXT' so that the default application is the word processor.
There are no restrictions as to what can be done (at least under Windows) so users can change whatever they like although some changes would not make sense, for example, opening a file that contains a bit mapped image using Notepad.

UNIX allows a file to have more than one extension associated with it. That is a filename can have more than one period in it. This allows filenames of the form *life.c.z***.** This is likely to be a C program that has been compressed using a suitable program (e.g. gzip).

Some common file extensions are shown below; although there are hundreds (if not thousands) more.

| Extension | File Contents |
|---|---|
| BIN | Binary File |
| C | C Program File |
| CPP | C++ Program File |
| DLL | Dynamic Link Library |
| DOC | Microsoft Word file |
| EXE | Executable File |
| HLP | Help File |
| TXT | Text File |
| XLS | Microsoft Excel File |

## *File Attributes*

In addition to the file name and the data that the file contains the operating system also maintains a set of ***attributes*** associated with each file.
The attributes differ from operating system to operating system but typical ones are shown below

| Attribute | Description |
|---|---|
| Archive Flag | Bit Field : has the file been archived? |
| Creation Date/Time | Date and Time file was created |
| Creator | User ID of the person creating the file |
| Hidden Flag | Bit Field : Is the file a hidden file? |
| Last Accessed Date/Time | Date and Time file was last accessed |
| Owner | The ID of the current owner |
| Password | Password required to access the file |
| Protection | Access rights to the file |
| Read-Only | Bit Field : Ids the file read only? |
| Size in Bytes | How large is the file |
| System Flag | Bit Field : Is the file a system file? |
| Temporary Flag | Bit Field : Should the file be deleted at end of the process? |

## *File Structure*

Files can be structured in various ways. One approach adopted by UNIX, MS-DOS and Windows is simply to have the file stored as a sequence of bytes. It is up to the program that accesses the file to interpret the byte sequence.
This structure is useful in that the programs can do whatever they like as the operating system does not impose any restrictions. In addition, the program can decide which extension to give the file so that the correct application is used if the file is opened using its associations.

One way of adding structure to a file is to split the file into fixed length records. This method is common on mainframes and can be traced back to the use of punched cards. When it became possible to store card (images) on disc it was sensible to read 80 bytes at a time as the programmer knew that each 80 bytes represented a card.
But, a record size of 80 was just the first step. By defining the file using a different value a different record size cold be set up, to meet the requirements of the programmer (e.g. 160 for prints produced for a line printer).

A development of the fixed length record was a file that contained variable length records. It was quickly recognised that a fixed length record could be wasteful of space. For example, if you defined a file to contain names and addresses, every record had to be the same size, even though some people have short names and some have long names. In fact, the field size was dictated by the longest possible record that could possibly stored.
Some operating systems recognised this limitation and did not waste so much space but the programmer still had to allow space in his/her program.

Variable length records allowed each field to have its size specified so that a surname field (for example) could have a different size to a line of an address.
However, this method also has its problems. Consider what they are. One possible answer is given in a separate handout (see question 1).

A further type of file structure is to have an index (possibly more than one) within the file. Each index entry points to a record within the file. If the file is sorted on the index (though the data itself does not have to be sorted) then random access is possible.
These types of files are often used on mainframe computers (and thus large data processing systems) and, although the structure of a database is a lot more complicated, they use the concept of an index to access the data as fast as possible.
Due to the way the file is structured they may need to be reorganised after a time else they will be become inefficient. The reasons for this were covered in the lecture.

## *File Access*

Above, whilst looking at file structures, we mentioned ***random access***. We ought to say what we mean by this.
There are essentially two ways to access a file. We can start at the beginning and read every record until we find the one we require. When mainframe operating systems kept the majority of their data on magnetic tape only ***sequential*** access was required. This led to a model of ***batch updating*** which was explained in the lectures.
But, as disc space became cheaper, more and more files were stored on disc and this led to the ability to access data *randomly*. Or more exactly, it allowed the program to directly access the record it required without having to read all the data that preceded it.

## *Directories*

Many operating systems provide some form of ***directory*** mechanism. Certainly MS-DOS, Windows and UNIX do, although, under Windows 95 Microsoft tended to call them folders but they seem to have reverted to the term directory in Windows 98 and Windows NT.
The purpose of a directory is to

- Allow like files to be grouped together
- Allow operations to be performed on a group of files which have something in common, for example, copy the files or set one of their attributes.
- Allow files to have the same filename (as long as they are in different directories). This allows more flexibility in naming files.

A typical directory entry contains a number of entries; one per file. All the data (filename, attributes and disc addresses) can be stored within the directory. Alternatively, just the filename can be stored in the directory together with a pointer to a data structure which contains the other details.

## *Hierarchical Directory Systems*

The common method of organising directories is to allow a tree like structure. This is, the filing system has a root directory which can contain other directories. This is by far the most common method.
An alternative is to have a single directory (the root) with all files contained within that directory. To the user, this scheme appears as if there are no directories at all.
A solution that comes in between these two is to allow a root directory and then one directory per user. This allows each user to name their files how they please but does not allow them to logically group their files further as they cannot create further sub-directories.

As an exercise, consider an operating systems that only allows one directory (the root) but does allow arbitrarily long filenames. Can you think of a way of simulating a hierarchical filing system? If so, how? A suggested answer is in a separate handout (see question 2).

## *Path Names*

When specifying a filename we can do so in one of two ways. We can specify its ***absolute*** path name. That is, we specify the entire filename. For example (these examples are given in MS-DOS, but the same logic applies to UNIX)

```
C:\COURSES\OPS\FILE SYSTEMS
```

This says the file, 'FILE SYSTEMS' is on the C: drive, and within the OPS directory which is a sub-directory of COURSES.
In fact, we could shorten this to

```
\COURSES\OPS\FILE SYSTEMS
```

The difference is that we are assuming that the default drive is the C: drive. The important point is that if a filename begins with a '\' (or a '/' under UNIX) then the pathname begins from the root directory and we are supplying an absolute path name.

Alternatively, we can specify a ***relative*** path name. The relative path name depends on what the ***current working directory*** (sometimes called just the ***current directory*** or the ***working directory***) is. If the process that is running currently has a working directory of 'C:\COURSES' then the relative path name for the above file would be

```
OPS\FILE SYSTEMS
```

(Note the absence of the '\' at the start).

As a note of interest, in an interactive session, under UNIX you can find out the current working directory by typing 'PWD' (print working directory). Under MS-DOS it is usual to change the command prompt so that the current working directory is displayed on the command prompt. This is achieved by issuing a

```
PROMPT $p$g
```

(normally in a startup procedure such as AUTOEXEC.BAT).

The $p tells MS-DOS to display the current drive and working directory. The $g tells MS-DOS to display a '>' character, which is the usual MS-DOS prompt character.

As an exercise, every directory contains two entries. These are '.' and '..' (often called dot and dotdot). What do you think they represent and give some example of commands they use them?
Some suggested answers are given in a separate handout (see question 3).

## File System Implementation

In this and subsequent sections we look at various ways in which filing systems can be implemented. In particular, we will look at how files and directories are stored, how disc space is managed and how we can make the filing system efficient and reliable.

Implementing a file system can be reduced to keeping track of which block is associated with which file. Below we look at some methods.

### *Contiguous Allocation*

One implementation is to allocate *n* contiguous blocks to a file. If a file was 100K in size and the block was 1K then 100 contiguous blocks would be required. This method has two advantages.

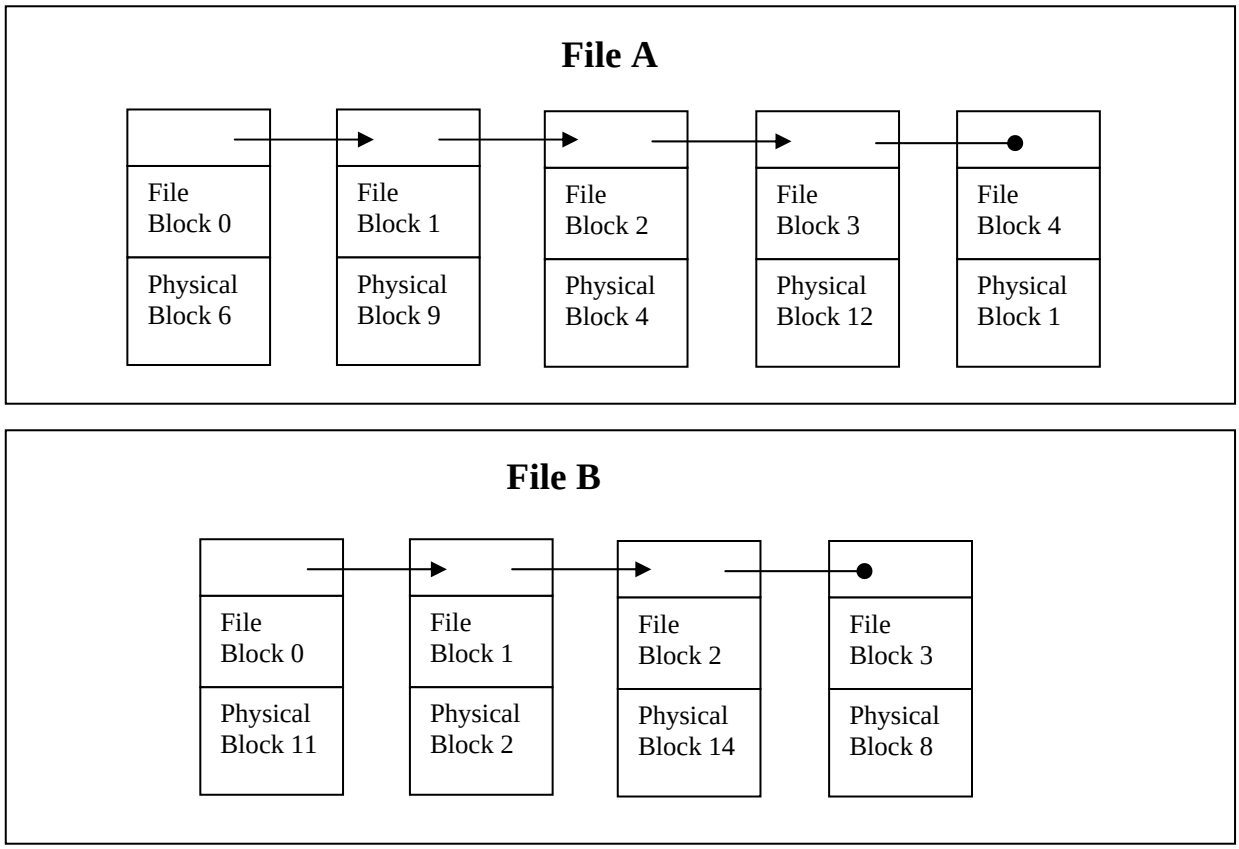
1. It is simple to implement as keeping track of the blocks allocated to a file is reduced to storing the first block that the file occupies and its length, which is likely to be stored as one of its attributes anyway.
2. The performance of such an implementation is good as the file can be read as a contiguous file. The read write heads have to move very little, if at all. You will never find a filing system that performs as well.

But, this method also has two significant drawbacks.

- The operating system does not know, in advance, how much space the file can occupy. If the file is sized at 100K and later grows (or tries to) to 150K there may not be enough contiguous space to meet the new file size. Of course, more space than is initially requested could be reserved but this is not only wasteful, but still does not guarantee that the space allocated will not be exceeded.
- This method leads to fragmentation. As files are created and deleted *holes* will be left which may not be big enough to place a file. A defragmentation process could be run periodically but this is wasteful of CPU resources and takes a long time.

*Linked List Allocation*

Blocks of a file could be represented using linked lists. All that needs to be held is the address of the first block that the file occupies. Each block of the file contains not only data but also a pointer to the next block. The diagram below shows such an implementation for two files.




The advantages of this method include

- Every block can be used, unlike a scheme that insists that every file is contiguous.
- No space is lost due to ***external*** fragmentation (although there is ***internal*** fragmentation within the file, which can lead to performance issues).
- The directory entry only has to store the first block. The rest of the file can be found from there.
- The size of the file does not have to be known beforehand (unlike a contiguous file allocation scheme).
- When more space is required for a file any block can be allocated (e.g. the first block on the free block list).

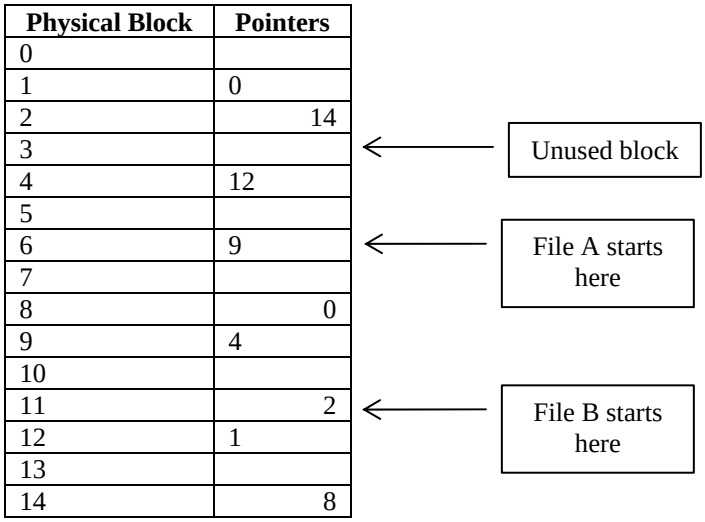
The disadvantages of this method include

- Random access is very slow (as it needs many disc reads to access a random point in the file). In fact, the implementation described above is really only useful for sequential access.
- Space is lost within each block due to the pointer. This does not allow the number of bytes to be a power of two. This is not fatal, but does have an impact on performance.
- Reliability could be a problem. It only needs one corrupt block pointer and the whole system might become corrupted (e.g. writing over a block that belongs to another file).

*Linked List Allocation Using an Index*

We can eliminate the problem of wasting space by having to store a pointer and the problem of not having random access by taking the pointer from each block and storing them in an index or table and placing this

in memory. Below is a diagram showing how the files (A and B) would be represented using a linked list allocation with an index.



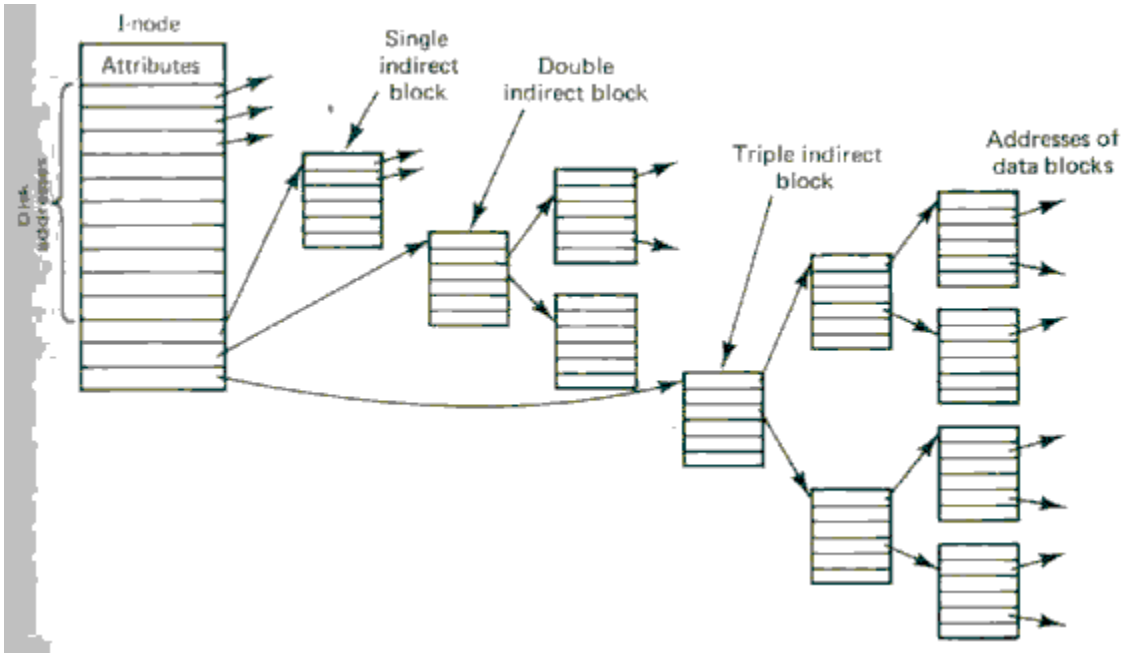

If we consider file B, it occupies blocks 11, 2, 14 and 8, but as the pointers are in memory random access is much faster as a given offset can be located by using only memory accesses until the correct block has been reached.

The main disadvantage is that the entire table must be in memory all the time. For a large disc with, say, 500,000 1K blocks (500MB) the table will have 500,00 entries.

## *I-Nodes*

The last filing system implementation we are going to consider is called an i-***node*** (for ***index-node***). This is the scheme that UNIX uses.
All the attributes for the file are stored in an i-node entry, which is loaded into memory when the file is opened. The i-node also contains a number of direct pointers to disc blocks. Typically there are twelve direct pointers.
In addition there are three additional, indirect pointers. These pointers point to further data structures which eventually lead to a disc block address.
The first of these pointers is a single level of indirection, the next pointer is a double indirect pointer and the third pointer is a triple indirect pointer.
Depending on the size of the disc (and the file), the triple indirect pointer may not be needed as all the blocks associated with the file can be accessed using the other pointers.

The structure of an i-node, and its associated entries, is shown below.



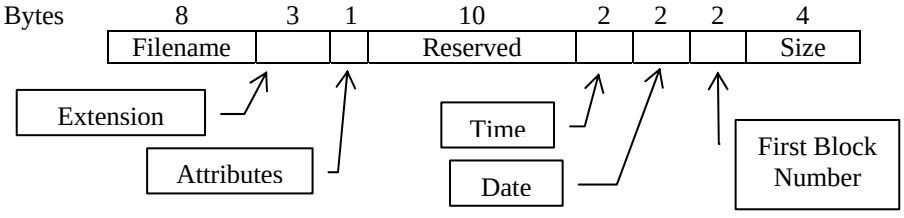

*Implementing Directories*

If a request to open a file is done the ASCII path name is used to locate the correct directory entry. The entry in the directory contains all the information needed to access the file. For example, using a contiguous allocation scheme the directory entry will contain the first disc block (all other disc blocks can be found from there). The same is true for linked list allocations.
For an i-node implementation the directory entry contains the i-node number which, once loaded into memory, can be used to access the file.
Therefore, the only job of the directory entry is to provide a mapping from an ASCII filename to the disc blocks that contain the data.
In addition, the directory entry may contain the attributes of the file. This is the case for an i-node implementation but other schemes may hold this information in the directory entry itself, or may provide a pointer to this information.

To give an example, we will look at how two directory systems are implemented; MS-DOS and UNIX.

## MS-DOS

Under MS-DOS a directory entry is 32 bytes long. It is split as follows



The most important part (for our present discussions) is the first block number. This is a pointer to the data structure shown above when we discussed linked list allocation using an index.

We should also state, for completeness, that MS-DOS allows directories within directories, leading to a tree like structure.

For the interested student (Tanenbaum, 1992) goes into the implementation of the MS-DOS file system in a lot more detail than we have done here.

### UNIX

A typical UNIX system directory entry just contains an i-node number and a filename. Unlike MS-DOS, all its attributes are stored in the i-node so there is no need to hold this information in the directory entry.

| Bytes | 2 | 14 |
|---|---|---|
| | i-node number | Filename |

The only other detail we should cover is how an i-node is located from its number. In fact, it is very simple. All the i-nodes have a fixed location on the disc so locating an i-node is a very simple (and fast) function.

To give us a better understanding let us consider how UNIX locates a file when given an absolute path name.

Assume the path name is */user/gk/ops/notes*. The procedure operates as follows.

- The system locates the root directory i-node. As we said above, this is easy as the entry is on a fixed place on the disc.
- Next it looks up the first path entry (user) in the root directory, to find the i-node number of the file /user.
- Now it has the i-node number for /user it can access the i-node data to locate the next i-node number (i.e. for /gk).
- This process is repeated until the actual file has been located.

Accessing a relative path name is identical except that the search is started from the current working directory.

## Disk Space Management

In the discussions above we can probably agree that a contiguous allocation scheme is probably unrealistic except in exceptional circumstances (in which case it will prove to be a very good scheme).
Therefore, allocating files as a list of blocks seems like a good idea. However, up till now, we have not mentioned the block size of the file. In this section we will consider this question.
We will also look at how we keep track of what blocks are free on the disc. We obviously need to know this information in order to allocate them as necessary.

### *Block Size*

Whatever block size we choose then every file must occupy this amount of space as a minimum (as each file must consist of at least one block). If we choose a large allocation unit, such as a cylinder (which is say 32K) then even a 1K file will occupy 32K of the disc.
However, choosing a small allocation size (of say 1K) means that files will occupy many blocks which results in more time accessing the file as more blocks have to be located and accessed.

To take this argument further, accessing a block (which is in a *random* part of the disc) means the read/write heads have to move (the seek time to the correct cylinder/track). Next the correct sector has to come round so that it is under the read/write heads (rotational delay or latency).

Therefore, there is a compromise between a block size, fast access and wasted space. The usual compromise is to use a block size of 512 bytes, 1K bytes or 2K bytes.
If you are interested a study is reported in (Mullender, 1984).

### *Tracking Free Blocks*

As well as tracking which blocks are associated with each file we also need to have some way of tracking the free blocks so that we can allocate them as necessary.
Two methods are in common use.

## Linked List

Using this method some of the free blocks (which will no longer be free!) are given over to holding disc block numbers that are free. The blocks that contain the free block numbers are linked together so we end up with a linked list of free blocks.

We can calculate the maximum number of blocks we need to hold a complete free list (i.e. an empty disc) using the following reasoning.

- Assume that we need a 16-bit number to store a block number (that is block numbers can be in the range 0 to 65535).
- Assume that we are using a 1K block size.
- A block can hold 512 block addresses. That is, 1024*8 [number of bits in a byte] / 16 [bits needed for a block address]
- Assume that one of the addresses is used as a pointer to the next block.
- For a 20Mb disc we need, at most, 41 blocks to hold all the free block numbers. That is, 20*1024 [maximum number of blocks] / 511 [number of disc addresses in a block].

We can generalise this and say

Maximum Number of Blocks =

ROUNDUP((SizeOfDisc * K) / ( ( (K * BitsPerByte) / BitsForDiscAddress) -1) )

Where

| | | |
|---|---|---|
| SizeOfDisc | : | Size of the disc in MB (20 in the above example) |
| K | : | 1024 (i.e. 1K) |
| BitsPerByte | : | 8 (usual number of bits in a byte) |
| BitsForDiscAddress | : | Bits required to hold disc block number (16 in the above example) |

(Note : this formula could be simplified , but it is presented as it is so it can be read in conjunction with the notes).

And ROUNDUP is a function that rounds up to the nearest integer

## Bit Map

Using this scheme a bit map is used to keep track of the free blocks. That is, there is a bit for each block on the disc. If the bit is 1 then the block is free. If the bit is zero, the block is in use (of course, these conventions can be reversed).
To put it another way, a disc with *n* blocks requires a bit map with *n* entries.

If we consider a 20Mb disc with 1K blocks then we can calculate the number of blocks needed to hold the disc map.

- A 20Mb disc has 20480 (20 * 1024) blocks.
- We need 20480 bits for the map, or 2560 (20480 / 8) bytes.

- A block can store 1024 bytes so we need 2.5 blocks (2560 / 1024) blocks to hold a complete bit map of the disc. This would obviously be rounded up to 3.

We can generalise this to

Maximum Number of Blocks =

ROUNDUP ((SizeOfDisc * K) / BitsPerByte / K)

Where the variables have the same meaning as above.
(Note : this formula could be simplified , but it is presented as it is so it can be read in conjunction with the notes).

## Comparison

Generally, the bit map scheme requires a lesser number of blocks than the linked list scheme. This is hardly surprising as the bit map scheme only uses one bit per block, whereas the linked list scheme uses sixteen bits per block.
Only when the disc is nearly full does the linked list version need fewer blocks. This is due to the fact that the bit map scheme requires a constant number of blocks but the number of blocks required by the linked list scheme decreases as the disc gets fuller.

If you have the inclination, the above discussions have been implemented on a spreadsheet (which is available via the web site for this course). This allows you to compare the two methods we have discussed and also allows you to adjust the parameters to see the effect it has.

The only advantage that a linked list scheme has over a bit map scheme is when only a small amount of memory can be given over to keeping track of free blocks. Assume, the operating system can only allow one (free list) block to be held in memory and that the disc is nearly full.
Using a bit map scheme, there is a good chance that the block held in memory will indicate that every block is being used. This means a disc access must be done in order to get the next part of the bit map. With a linked list scheme, once a block containing pointers of free blocks has been brought into memory then we will be able to allocate 511 blocks before doing another disc access.

# References


- Mullender, S.J. and Tanenbaum, A.S. 1984. Immediate Files. Software – Practice and Experience, Vol. 14, pp 365-368
- Tanenbaum, A.S. 1992. Modern Operating Systems. Prentice Hall.
- Tanenbaum, A., S. 2001. Modern Operating Systems (2nd ed.). Prentice Hall.
- Tanenbaum, A., S. 2008. Modern Operating Systems (3rd ed.). Prentice Hall.