CS240 Operating Systems, Communications and Concurrency

File Systems

A file is a collection of related information, defined by its creator and stored on non volatile storage.

A file system is the software which maps files onto storage devices in the system and provides an interface to the user for accessing and manipulating files.

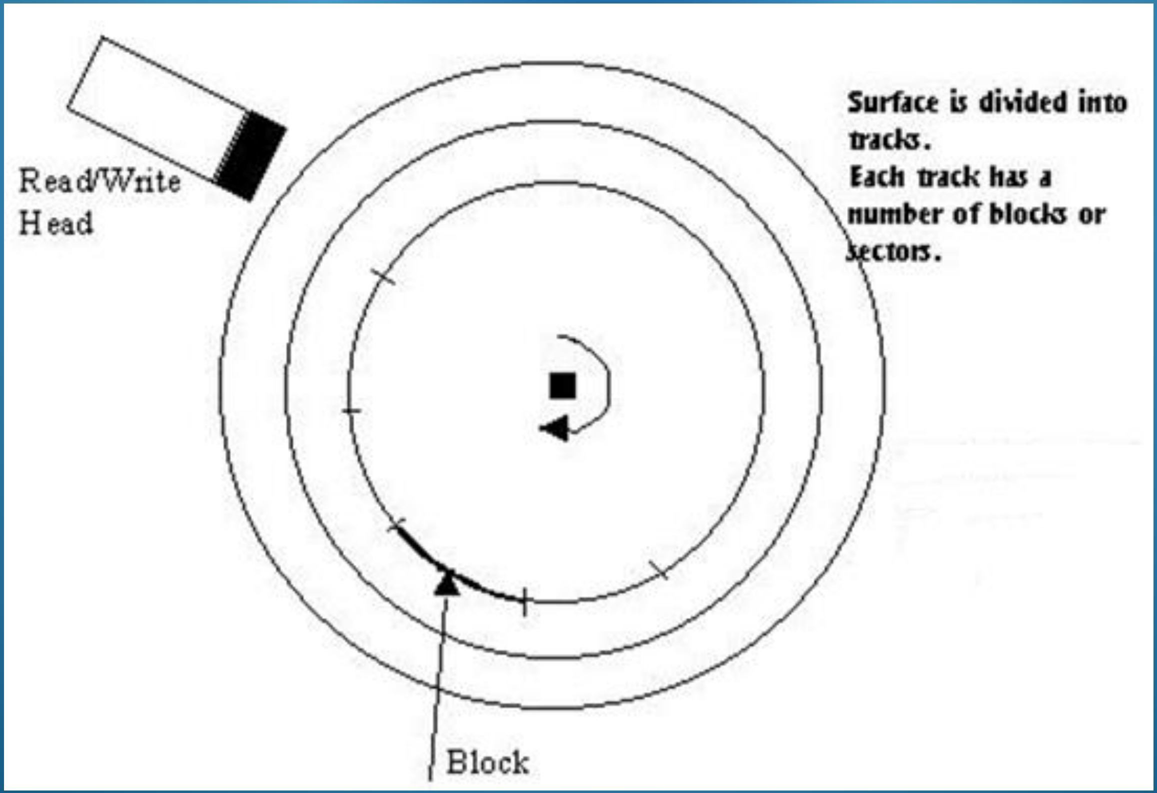
This section examines some basic concepts of files and file systems, how space is managed and allocated, how to improve efficiency of file access operations and how files are organised into hierarchical directories.

CS240 Operating Systems, Communications and Concurrency

Solid State Drives offer greater performance, consume less power, are more environmentally rugged and make less noise. For these reasons, many mobile devices, use solid state non volatile memory. On desktop systems, magnetic drives are still dominant and remain significantly cheaper for large capacities. Our file system designs and optimization strategies are generally focused on magnetic drives and improving the latency of read/write operations.

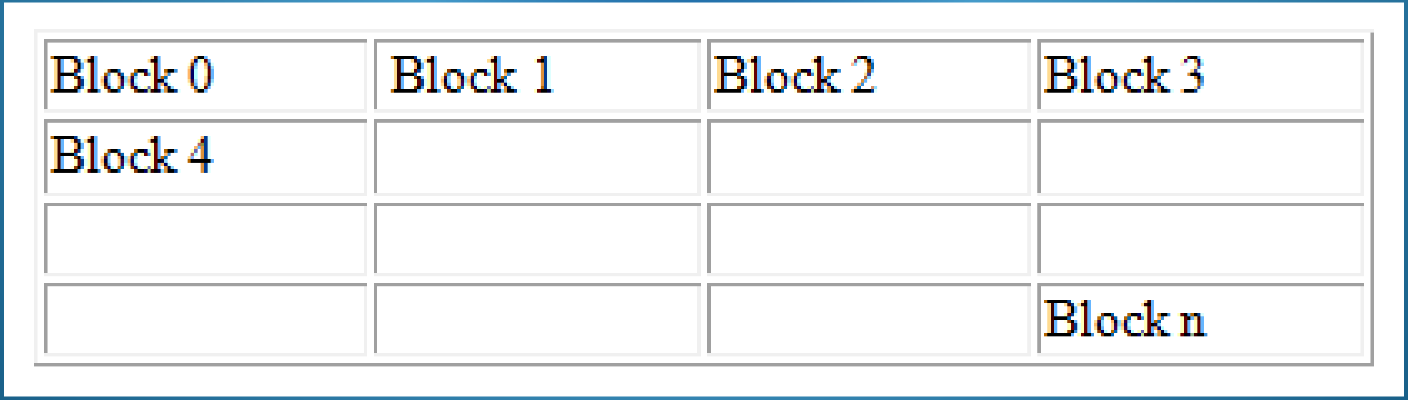
CS240 Operating Systems, Communications and Concurrency

The magnetic surface is organized as a collection of fixed sized data blocks. The address of each block consists of a surface (platter) number, track number and block number on a particular device.


Read/Write Head
Surface is divided into tracks.
Each track has a number of blocks or sectors.
Block

CS240 Operating Systems, Communications and Concurrency

We can view the surface of the disk as consisting of a fixed number of uniquely identifiable data blocks from 0..n. The number of blocks multiplied by the block size gives the data capacity of the magnetic disk.



Block 0	Block 1	Block 2	Block 3
Block 4			
			Block n

CS240 Operating Systems, Communications and Concurrency

<u>**Design Requirements**</u>

The <u>file system designer</u> must decide how these blocks are allocated and deallocated in order to achieve the maximum performance from the file system as well as ensuring file persistence and data integrity. At the device driver level, the designer will have to be aware of the physical device characteristics.

The <u>file system user</u> is concerned with how to access files, how to organise the internal structure of a file, organise collections of files into logical groupings and how to share them with other users.

CS240 Operating Systems, Communications and Concurrency

Design Requirements

A file system has a number of requirements:

Mapping files onto devices
Organising files into directories
File Protection, File Sharing
Data Integrity & Persistence
Support for a variety of storage devices
User Interface

CS240 Operating Systems, Communications and Concurrency

Design Requirements

The File System must provide an **interface to the user** which would typically offer the following functionality:-

Open	Prepare a file to be referenced within the operating system
Close	Prevent further reference until file is reopened
Create	Build a new file within the collection
Delete	Remove a file
Copy	Create another version of the file with a new name
Rename	Change the name of a file

GetAttributes File Access is controlled by permissions attached to the file, also size, time of modification and creation

Set Attributes Alter file attributes that can be set, command line utilities like touch and chmod

CS240 Operating Systems, Communications and Concurrency

File Organisation

A file may be an unstructured collection of bytes, (e.g a text file or executable binary file) or may be structured as a collection of fixed or variable sized records, where each record contains a set of related fields (e.g. name, address, phone number). The first field of a record is usually a unique key field.

The file system may provide interface operations which allow the file to be indexed, searched, processed and updated at the record level rather than just at the byte level.

CS240 Operating Systems, Communications and Concurrency

<u>File Access Operations</u>

Additional API operations enable a process, which understands the format of a file, to access, interpret and modify its contents.

Read	Copy some bytes from a file to a process
Write	Output some bytes from a process to a file
Seek	Search to a specified point in the file
Append	Seek to the end of a file and add data there

<u>Structured File Operations using a record index key</u>

Retrieve	Read a record or group of records from a file
Update	Modify an existing file record
Insert	Add a new record into the file at a designated offset
Remove	Remove a specific record from a file

CS240 Operating Systems, Communications and Concurrency

Let us explore this interface by looking at what is required to implement some of the operations, what data structures are needed and how efficiently they perform.

Designing a suitable User Interface

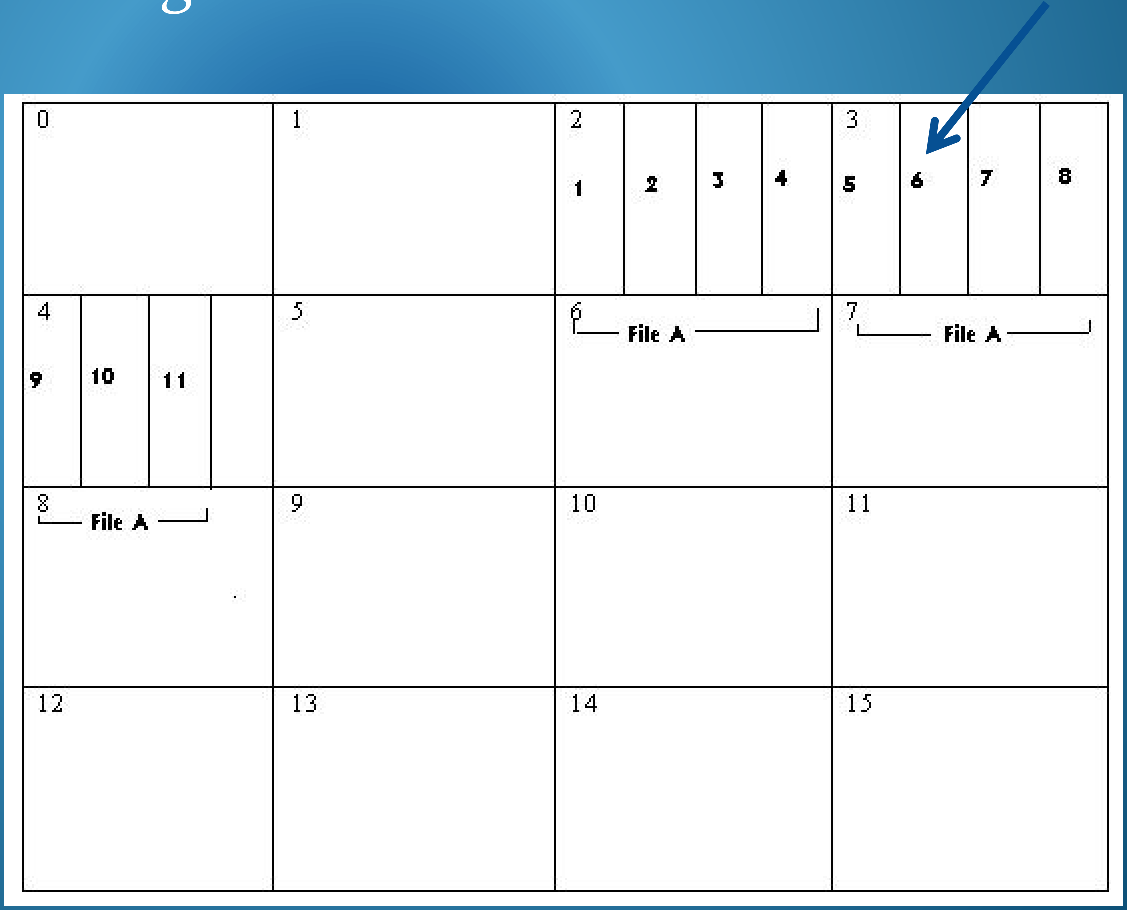
Let's say that we have a file consisting of `11x250 byte` records stored on a disk which has `16` physical blocks of size of `1000 bytes` and we want to access the 6th record to read into program memory.

Read(FileA, Record=6, Destination=Buffer, NumBytes=250)

CS240 Operating Systems, Communications and Concurrency

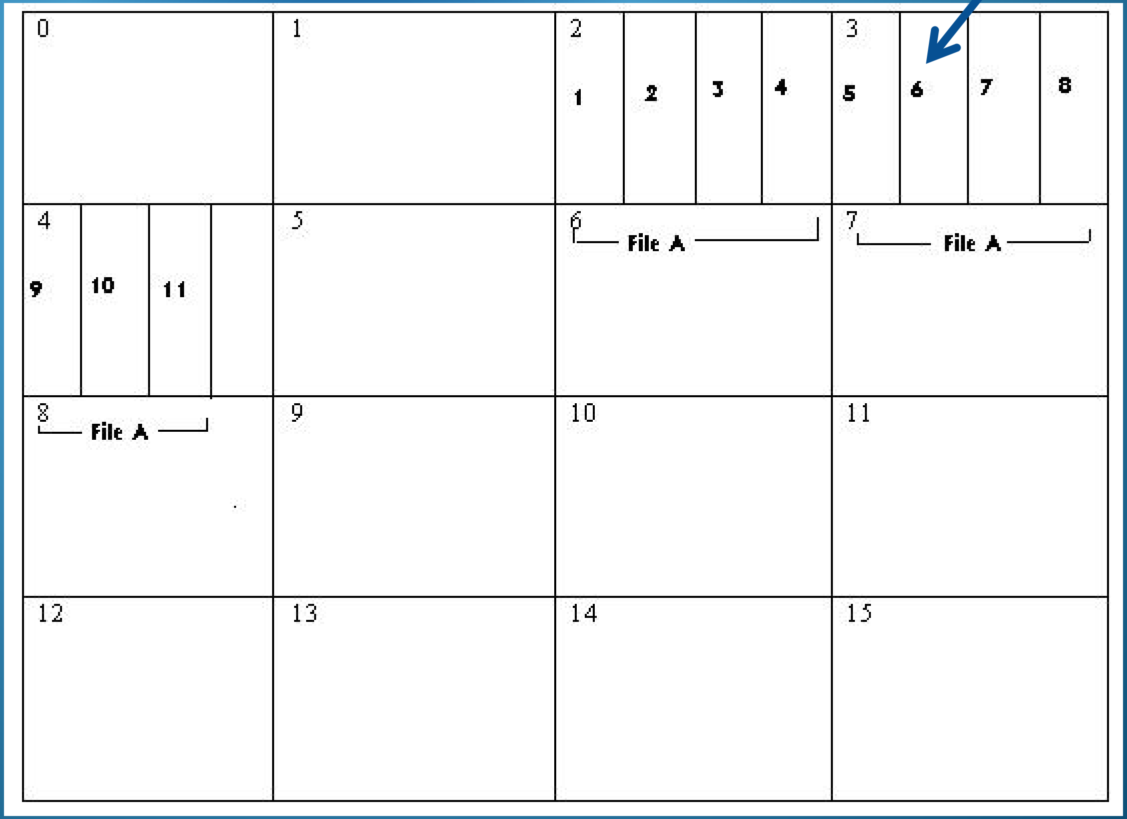
Read(FileA, Record=6, Destination=Buffer, NumBytes=250)

FileA stored on three contiguous blocks.



CS240 Operating Systems, Communications and Concurrency

To carry out the request, the file system must determine the physical location of file A on the disk, read the disk block containing record 6 and extract record 6 and move the data to location Buffer.



CS240 Operating Systems, Communications and Concurrency

Directory File

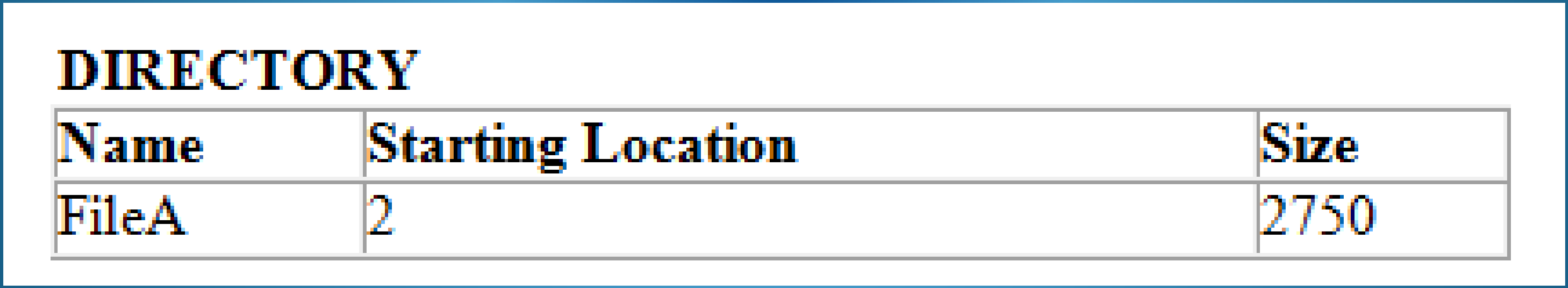
In order to determine the location of FileA, the file system requires a data structure to store this and other information relating to each file that is stored on a particular storage device. This data structure is called a file system **directory**.

Each file on the storage device has its own entry in the directory which gives details of its name, type, size, location information, who can access it, usage accounting history and other system specific details.

CS240 Operating Systems, Communications and Concurrency

Directory File

All we need to include in the directory entry to facilitate access to all the data in our FileA would be an entry as follows:-



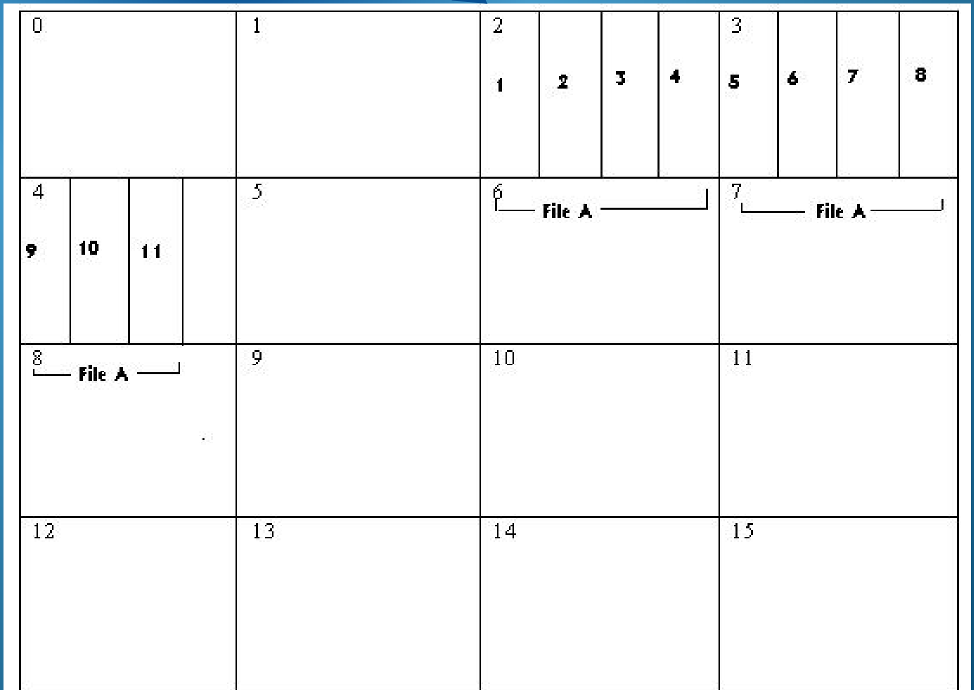
DIRECTORY

Name	Starting Location	Size
FileA	2	2750

(Assuming contiguous allocation of file space)

CS240 Operating Systems, Communications and Concurrency

With contiguous allocation, if the position of the first block of the file is known, the size of the file is known and the size of the physical disk blocks are known, then it is possible to determine where the remaining blocks of the file are located without having to explicitly store this location information.



CS240 Operating Systems, Communications and Concurrency

So, to satisfy our initial request:-

Read(FileA, Record=6, Destination=Buffer, NumBytes=250)

the file system would:-

Consult the **directory** to determine the starting location of FileA on the disk and then:-

******Read The Disk Block With Record 6 ******

CS240 Operating Systems, Communications and Concurrency

Which block contains Record 6?

We know that physical disk blocks are `1000bytes` in size. We are also told that records within this file are `250bytes` in length.

The application has a **logical view** of FileA as a collection of `11 records of 250bytes that occupies 2750 bytes.`

The file system has a **physical view** of FileA as being an object which occupies three particular physical blocks (blocks 2, 3 and 4) on the storage device.

CS240 Operating Systems, Communications and Concurrency

Which block contains Record 6?

1. In order to access record 6, its starting byte position(record offset) in FileA is calculated as follows:-

(6-1) * 250 = 1250 Byte offset from start of file

2. To determine which logical block it is in, we divide by the block size and see how many times it goes in evenly (block offset):-

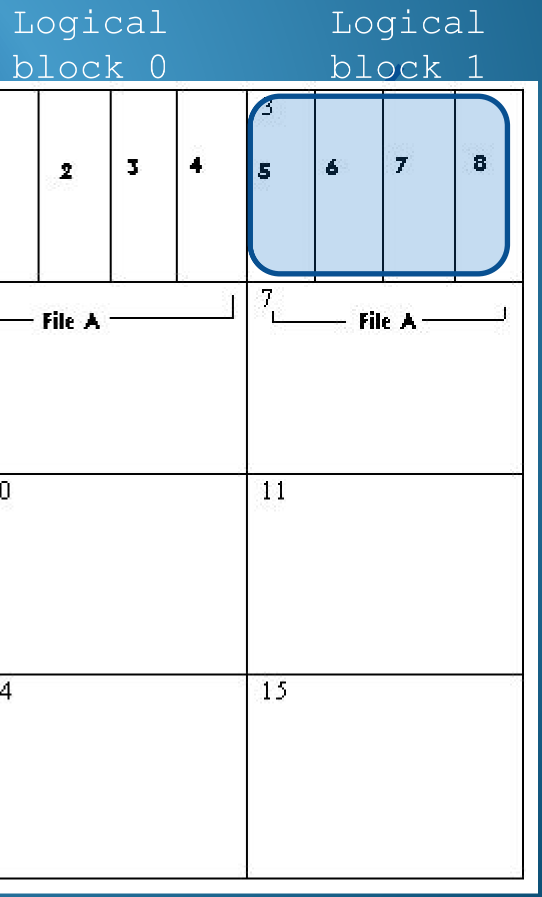
1250 / 1000 = 1. (2nd Block, Block numbers begin at 0…)

3. We then map the physical location of the block by adding on the starting block number "2" for the file as found in the directory and the file system then reads physical block 3 from the storage device.

CS240 Operating Systems, Communications and Concurrency

Read(FileA, Record=6, Destination=Buffer, NumBytes=250)

FileA stored on three contiguous blocks.



CS240 Operating Systems, Communications and Concurrency

Extract Record 6 and Move to Buffer

We now have logical block `1` in memory and we need to extract record 6. Record 6 begins at byte number `250` within the block and records are `250bytes` long.

So we copy the bytes from offset `250 to 499` out of block `1` into the Buffer variable supplied with our original request to the file system and we are finished.

Read(FileA, Record=6, Destination=Buffer, NumBytes=250)

NB: In practice, it might not be all as simple as this as arbitrary data sizes might not fit conveniently into fixed sized disk blocks and may span block boundaries.

CS240 Operating Systems, Communications and Concurrency

Improving Efficiency

Now that we understand the basics of how file I/O operations are carried out by the file system, it is time to analyse the potential performance of our method described so far and to consider some practical alterations to our design.

CS240 Operating Systems, Communications and Concurrency

Consider what the file system would have to do if our program wished to read each of the `11` records of FileA in sequence.

For each of the `11` records in FileA, our algorithm would

1. Look up the directory to determine the location of FileA
2. Calculate the Physical Block Location of the block containing our record and read that block from the storage device..
3. Calculate the offset of the desired record and copy its content to a program buffer area.

CS240 Operating Systems, Communications and Concurrency

First of all, where is the directory stored?

The directory must be stored somewhere on the storage device that it describes so that the file system can initially find the directory itself after a power down or if the storage device is moved to another computer.

Otherwise the contents of the storage device would be indecipherable.

CS240 Operating Systems, Communications and Concurrency

Our algorithm above therefore would cause the file system to access the disk 11 times to read the block containing the directory information and search that information for our file entry.

We would also carry out a further 11 disk read operations to get the blocks containing each of the records of FileA.

This gives a total of 22 Disk Read Operations to read the 11 blocks from FileA.

CS240 Operating Systems, Communications and Concurrency

Large directories may describe thousands of files.

This means that they may occupy quite a number of blocks on a storage device themselves.

If we were to search all of these directory blocks each time we wanted to know the starting location of our FileA we would have many more disk read operations to read our `11` records sequentially.

CS240 Operating Systems, Communications and Concurrency

In practice, to avoid having to do this, file systems provide a function for opening a file for access before any read or write operations can be carried out.

When a file is opened, the directory is searched once for the file location information or part of it.

This information is then kept in memory by the file system and is consulted for all future read and write operations. If the file is modified, then its directory entry may need to be altered (perhaps if its size changed for example). The directory can be altered in memory until we are finished with the file and can then be written back to the storage device once when we close the file.

CS240 Operating Systems, Communications and Concurrency

The approach of opening and closing the file before read operations would reduce the number of directory operations from 11 down to 1 in our example.

We could also achieve better access performance to the file, if we kept a **cache** of disk blocks (a small subset of recently used blocks perhaps) belonging to the file in memory.

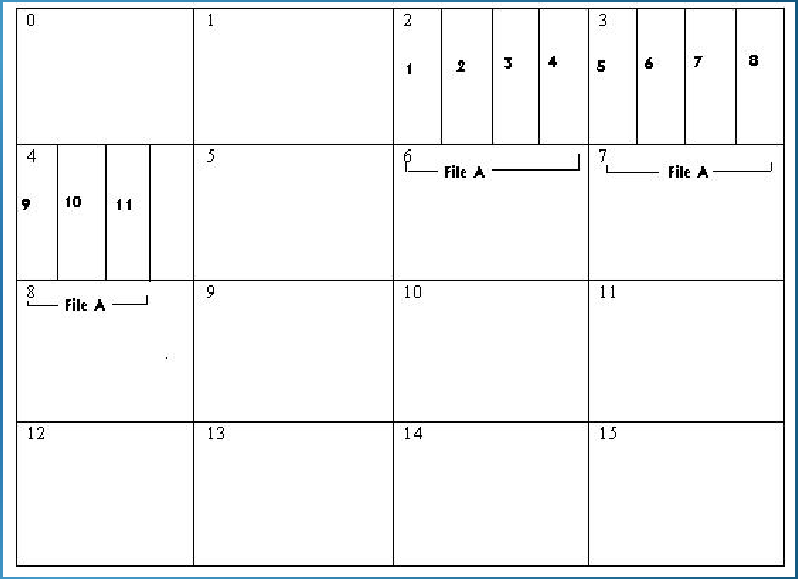
To read record 2, we would note that block 0 of the file was already in the cached area since we read record 1 and we could access record 2 directly without generating a disk read operation. We could therefore reduce our data block read operations from 11 down to 3. Overall reduction from 22 to 4.

CS240 Operating Systems, Communications and Concurrency

Keeping Track of Free Space

Automatic File Space Allocation

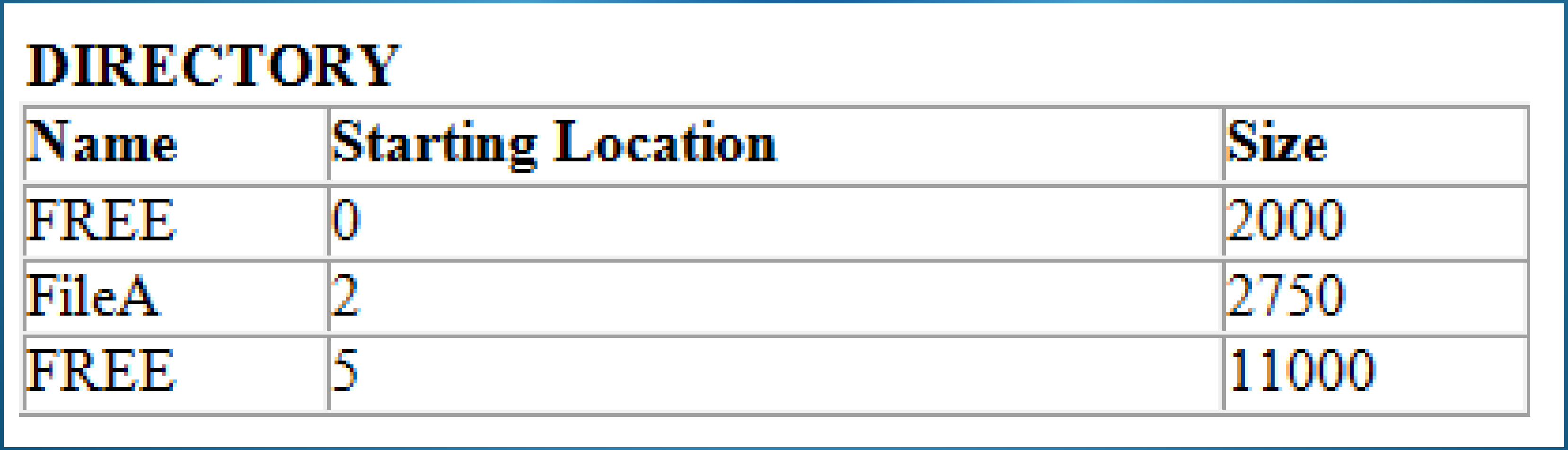
In order to make it easier for users to store files, the file system must keep track of where the free areas of the storage device are located.



When new files are created, or existing files are extended, free blocks (or groups of free blocks called clusters) can be allocated from the free areas automatically by the file system without the user having to specify where to put the file.

CS240 Operating Systems, Communications and Concurrency

Using our existing directory data structure, we can completely describe the state of the disk with the following entries (again assuming contiguous allocation.)

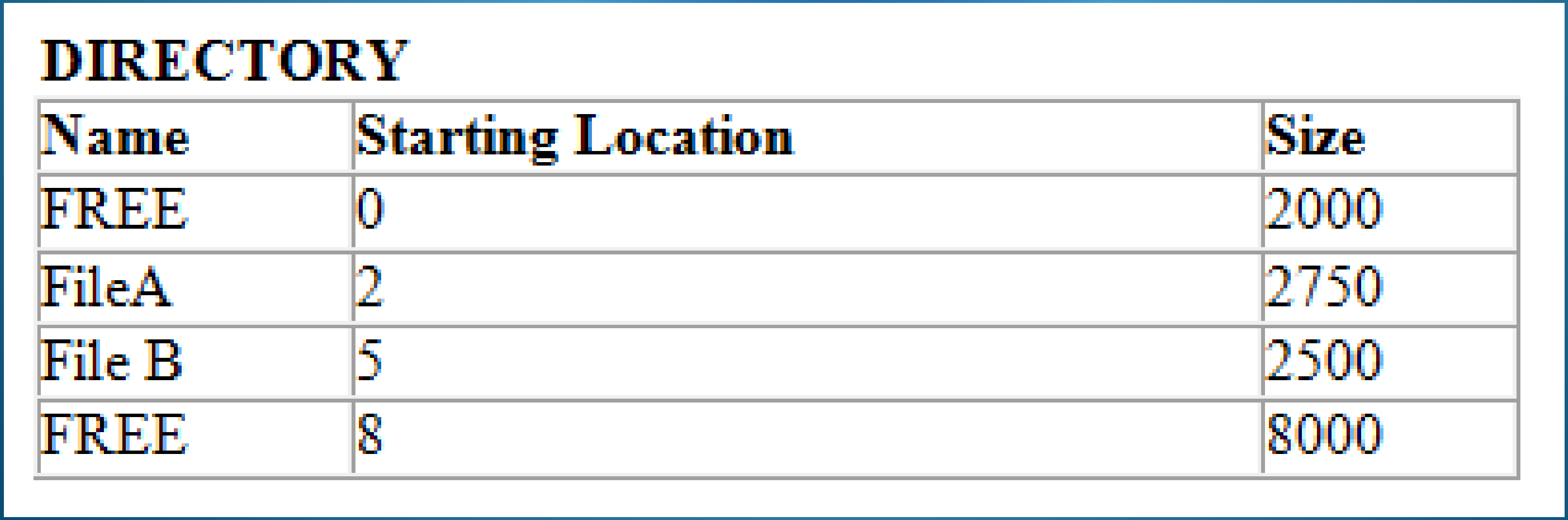


DIRECTORY

Name	Starting Location	Size
FREE	0	2000
FileA	2	2750
FREE	5	11000

CS240 Operating Systems, Communications and Concurrency

If we want to create space for FileB which will occupy `2500bytes` then the file system could scan the directory for a contiguous free section which was large enough to accommodate the new file. As the space is larger than needed, a smaller Free entry could be made in the directory as follows:-



DIRECTORY

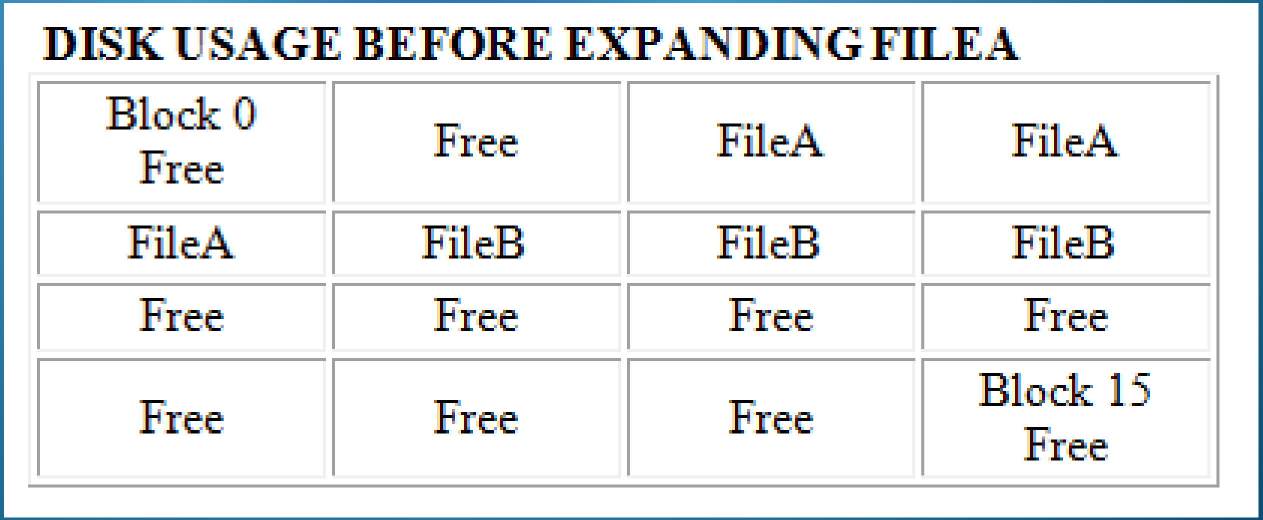
Name	Starting Location	Size
FREE	0	2000
FileA	2	2750
File B	5	2500
FREE	8	8000

CS240 Operating Systems, Communications and Concurrency

Dynamic Space Allocation

When files are created, it is often not known how much space will be required for their data as their sizes may change over time.

If we were to add two more records to FileA, its size would grow to 3250 bytes, requiring a fourth physical block to be allocated to it.



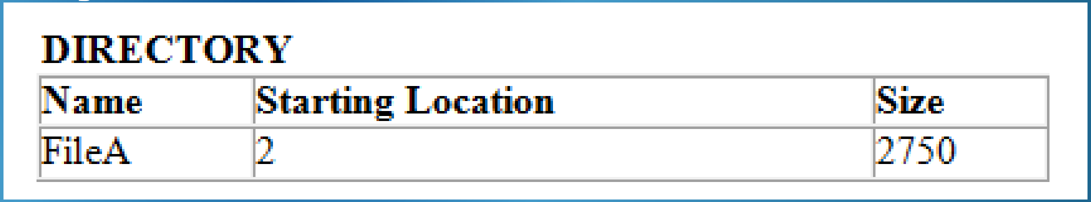
DISK USAGE BEFORE EXPANDING FILEA

Block 0 Free	Free	FileA	FileA
FileA	FileB	FileB	FileB
Free	Free	Free	Free
Free	Free	Free	Block 15 Free

CS240 Operating Systems, Communications and Concurrency

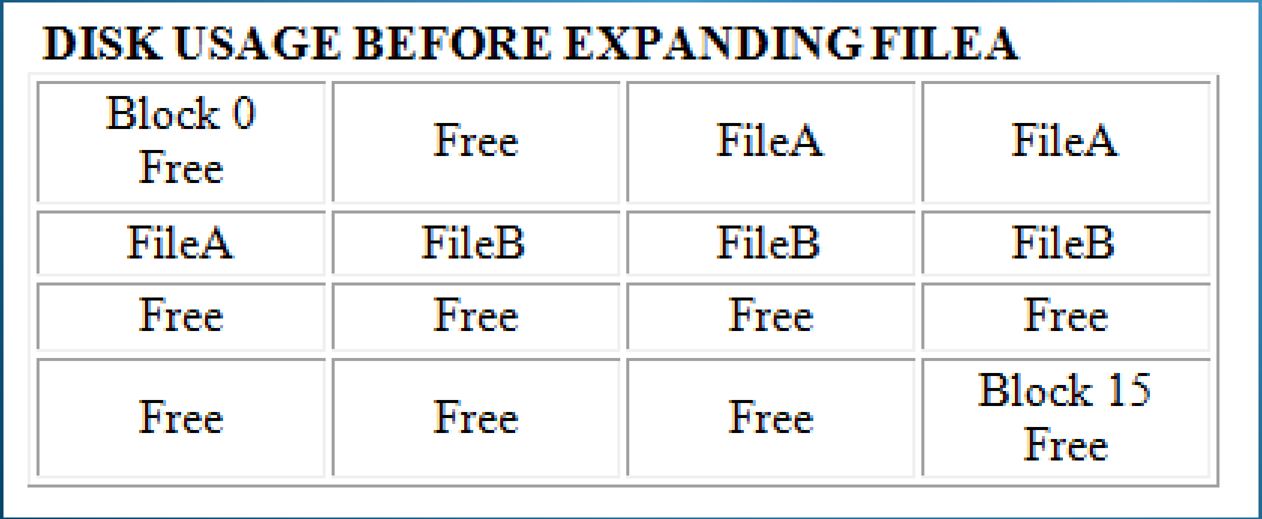
Dynamic Space Allocation

If we use a block from another free position on the device, then using our existing directory information, how would we find where we put it?



DIRECTORY

Name	Starting Location	Size
FileA	2	2750



DISK USAGE BEFORE EXPANDING FILEA

Block 0 Free	Free	FileA	FileA
FileA	FileB	FileB	FileB
Free	Free	Free	Free
Free	Free	Free	Block 15 Free

CS240 Operating Systems, Communications and Concurrency

Dynamic Space Allocation

If we had originally allocated four blocks to FileA, FileB would have started at an alternative location.

However, it may have been wasteful to overallocate space to FileA if it never needed this space as the space couldn't be used by other files once already allocated and this would represent wastage of our storage resources.

We need to be able to add new blocks dynamically to some files over time.

CS240 Operating Systems, Communications and Concurrency

Dynamic Space Allocation

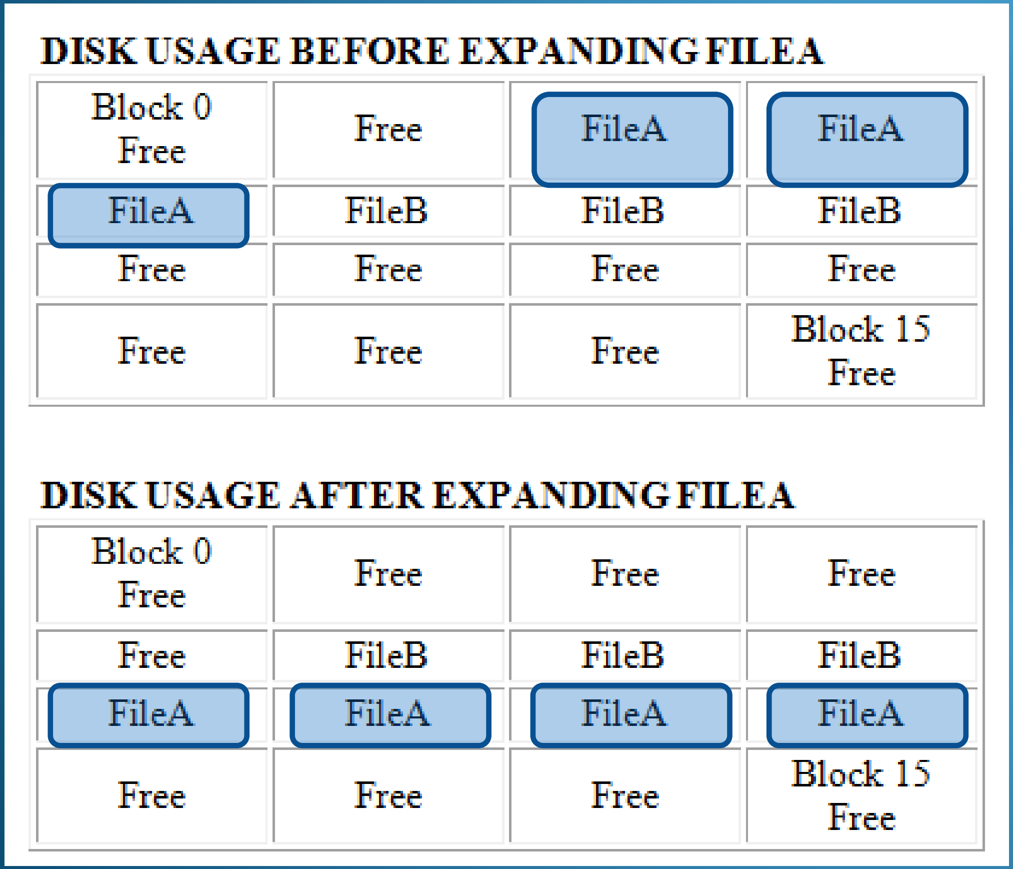
To resolve the situation and because we are using contiguous allocation of space, one of the files needs to be moved.

Let's move all of FileA to a larger contiguous area.

The file system could search for a free space large enough for FileA and its additional required space and copy its existing blocks to that area.

The original area occupied by FileA would be marked as free.

CS240 Operating Systems, Communications and Concurrency

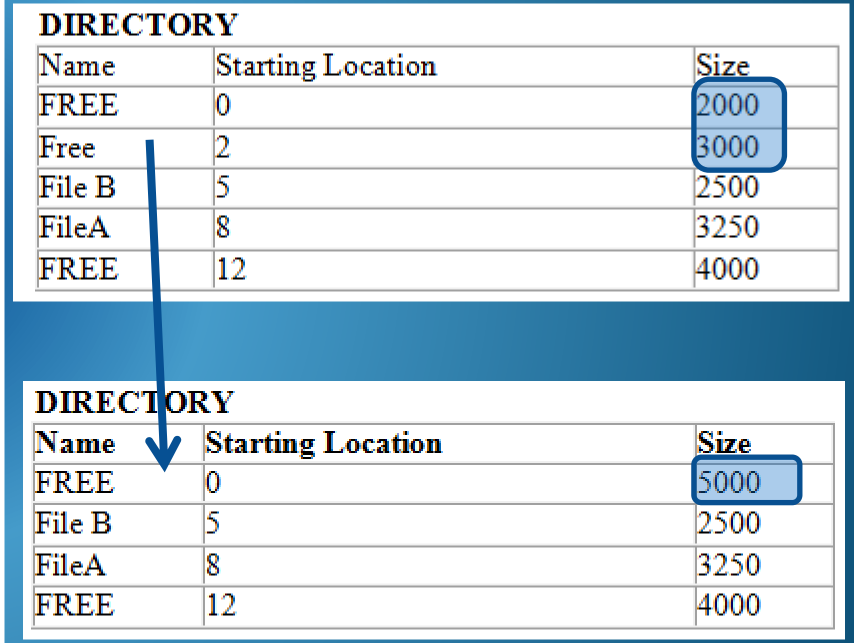


DISK USAGE BEFORE EXPANDING FILEA

Block 0 Free	Free	FileA	FileA
FileA	FileB	FileB	FileB
Free	Free	Free	Free
Free	Free	Free	Block 15 Free

DISK USAGE AFTER EXPANDING FILEA

Block 0 Free	Free	Free	Free
Free	FileB	FileB	FileB
FileA	FileA	FileA	FileA
Free	Free	Free	Block 15 Free



DIRECTORY

Name	Starting Location	Size
FREE	0	2000
Free	2	3000
File B	5	2500
FileA	8	3250
FREE	12	4000

DIRECTORY

Name	**Starting Location**	**Size**
FREE	0	5000
File B	5	2500
FileA	8	3250
FREE	12	4000

Other Space Allocation Techniques

Contiguous allocation of space is very efficient from the point of view of simple directory entries and easy to calculate file location information.

However, if files are dynamically changing in size, it becomes quite inefficient to implement dynamic space allocation in this way due to the need for file relocation.

CS240 Operating Systems, Communications and Concurrency

Linked Allocation

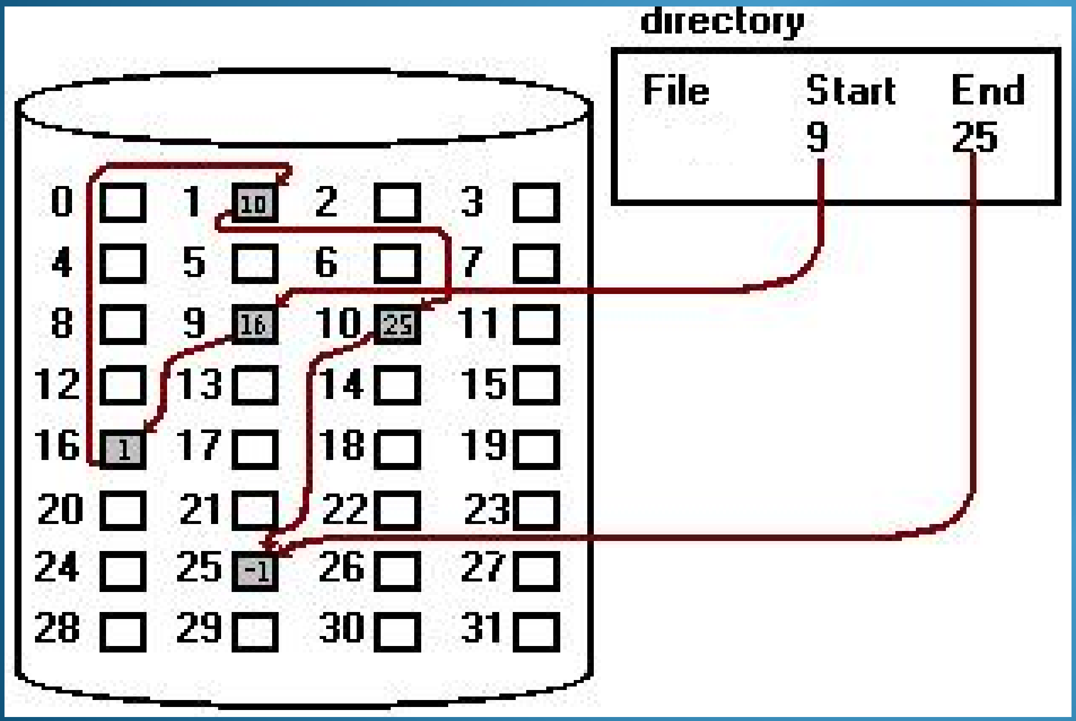
With linked allocation, the directory entry contains a pointer to the location of the first block in the file. However, each file block then contains a pointer to its successor, with the last block containing a special pointer termination code.

A small space is reserved in each file block by the file system to contain the pointer and this space is not available for a user's file data.

The free space can also be managed as a linked list of blocks in the same way.

CS240 Operating Systems, Communications and Concurrency

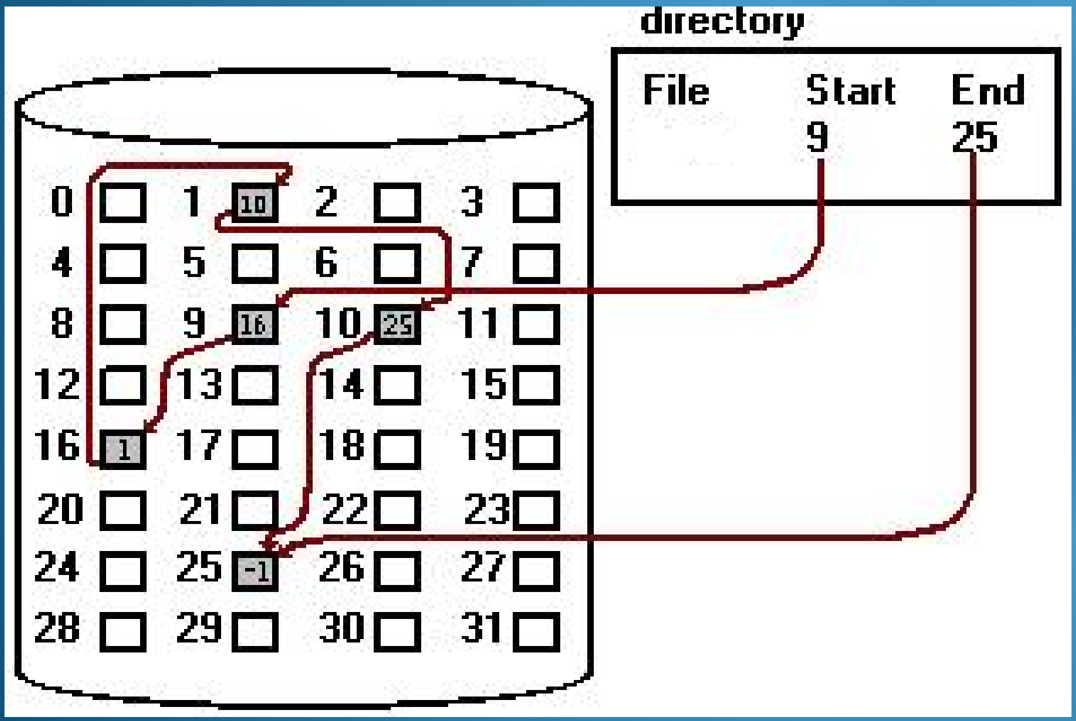
Linked Allocation



Linked allocation gives quite **good performance for sequential access** processing of a file, as each block is read in turn it provided information to locate the next block and this information doesn't have to be sought elsewhere.

CS240 Operating Systems, Communications and Concurrency

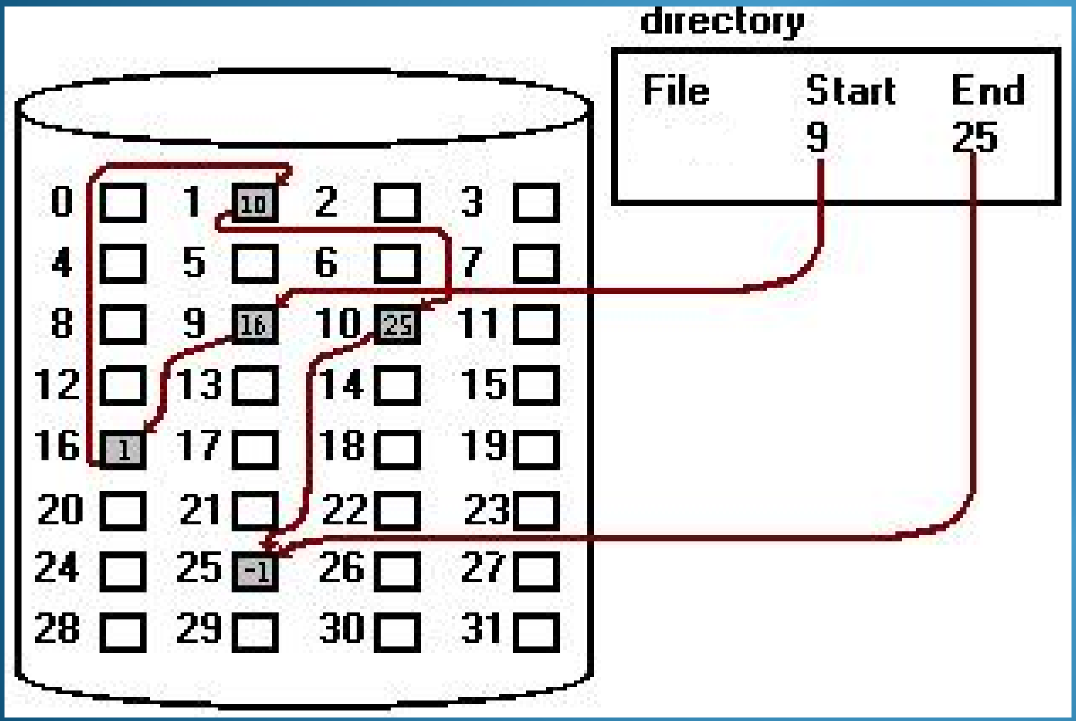
Linked Allocation



Random access to data is not as efficient. To access a random record location, you need to know where the block containing it is located. To find this out, you must begin at the first logical block and read all other blocks until you come to the location of the block which contains the required record.

CS240 Operating Systems, Communications and Concurrency

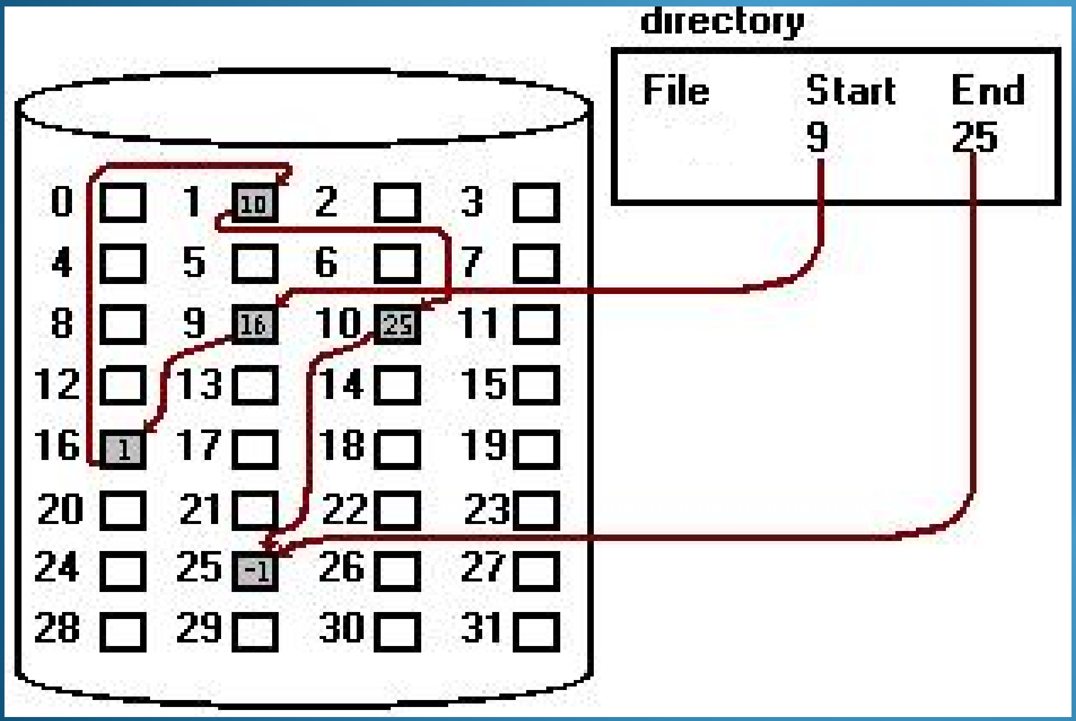
Linked Allocation



There is also a resource overhead associated with the method compared to contiguous allocation. We need to use some of the available storage on the device for storing pointer information.

CS240 Operating Systems, Communications and Concurrency

Linked Allocation

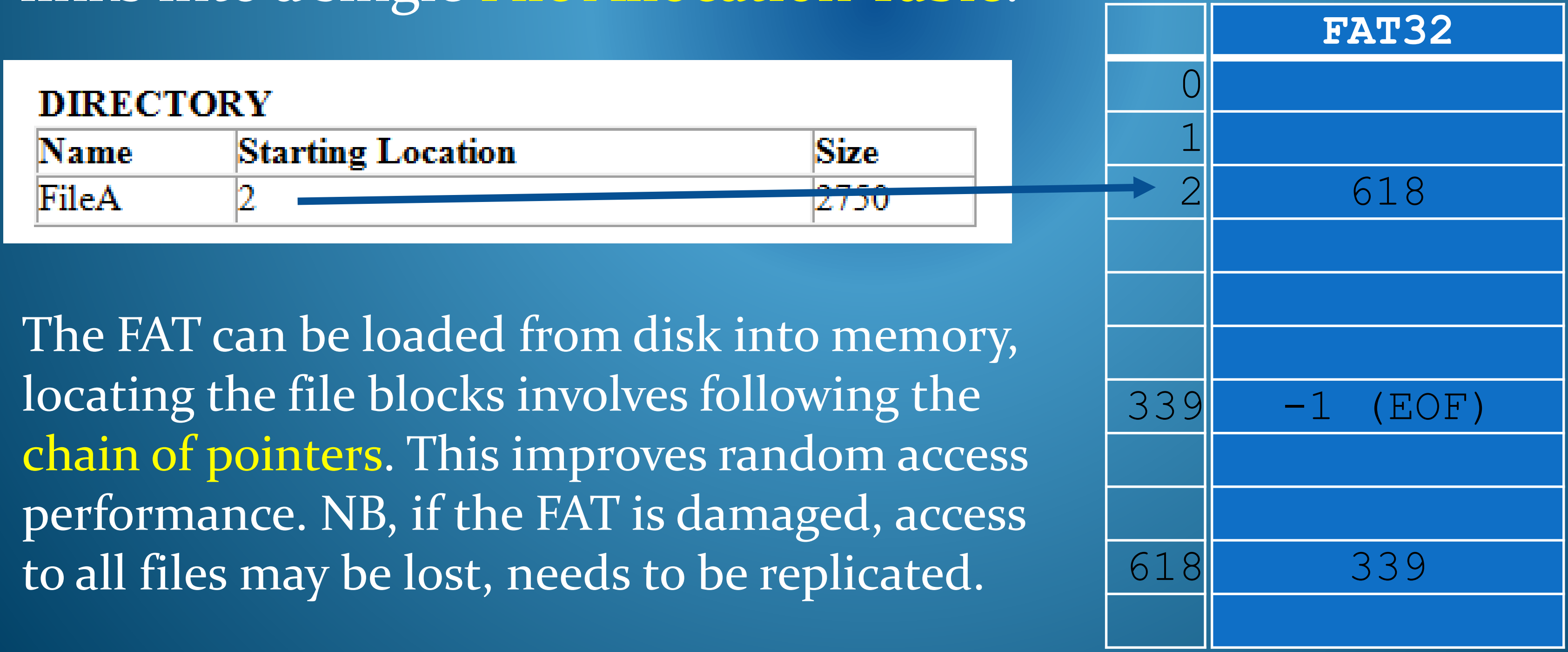


The linked allocation approach is **not as robust** as the contiguous approach. Losing a file block for example, breaks the chain of pointers and all blocks after the damaged block could be lost as there is no way of knowing that they belong to any particular file.

CS240 Operating Systems, Communications and Concurrency

Linked Allocation

An implementation of the linked allocation idea, to gain higher access performance, can be achieved by pulling all the links into a single File Allocation Table.


DIRECTORY

Name	Starting Location	Size
FileA	2	2750

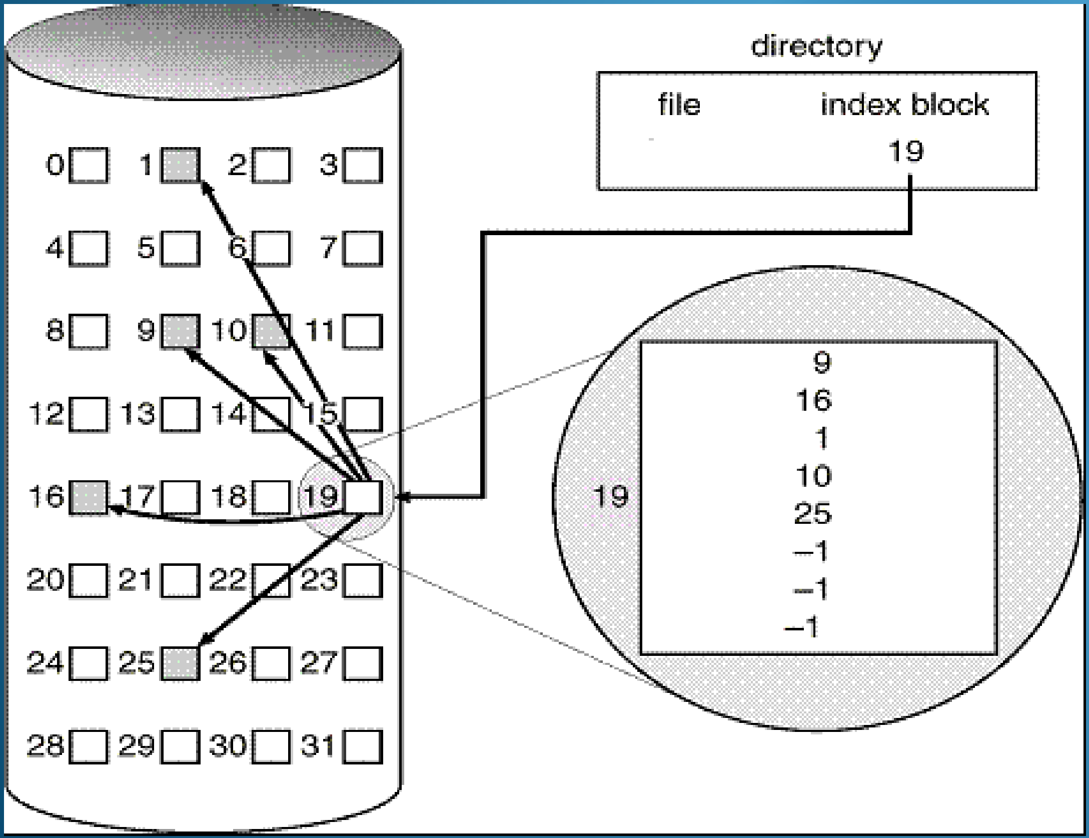
	FAT32
0	
1	
2	618
339	-1 (EOF)
618	339

Overhead of 4 bytes per block

The FAT can be loaded from disk into memory, locating the file blocks involves following the chain of pointers. This improves random access performance. NB, if the FAT is damaged, access to all files may be lost, needs to be replicated.

CS240 Operating Systems, Communications and Concurrency

Indexed Allocation

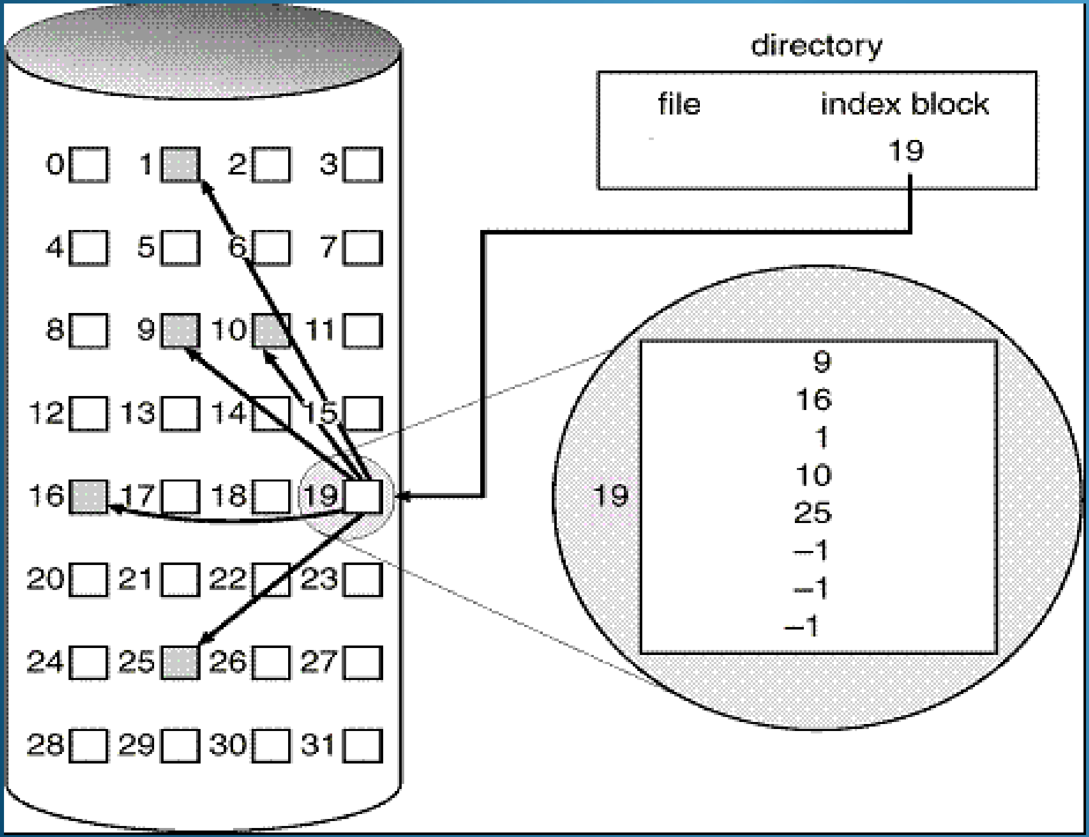


Indexed Allocation

An index block is assigned to each file. The index block is a table which maps logical file blocks to their physical block locations. The directory entry for the file then contains a pointer to the index block for that file from which the location of all of the file blocks can be determined immediately.

CS240 Operating Systems, Communications and Concurrency

Indexed Allocation

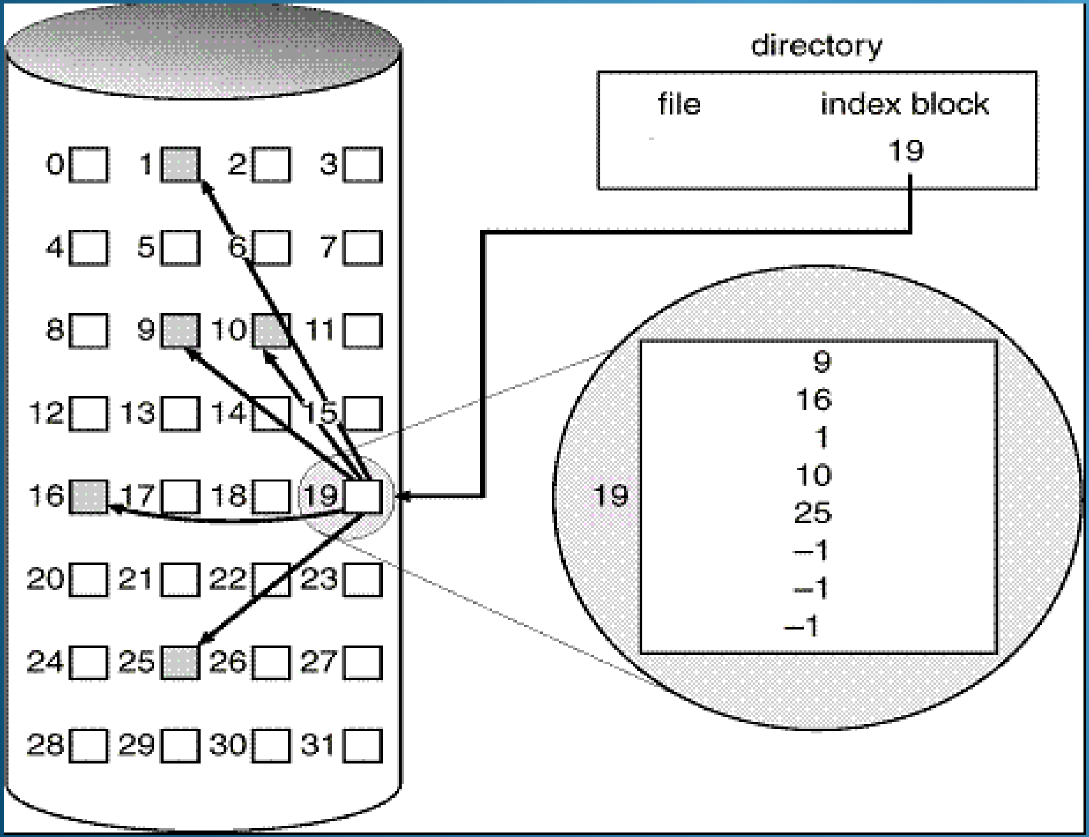


Indexed Allocation

Indexed allocation allows the logical blocks of a file to be scattered across the storage device wherever free blocks might be available for it. As data blocks can be located arbitrarily by consulting the index block, this allocation method gives good performance for both sequential and random record access.

CS240 Operating Systems, Communications and Concurrency

Indexed Allocation

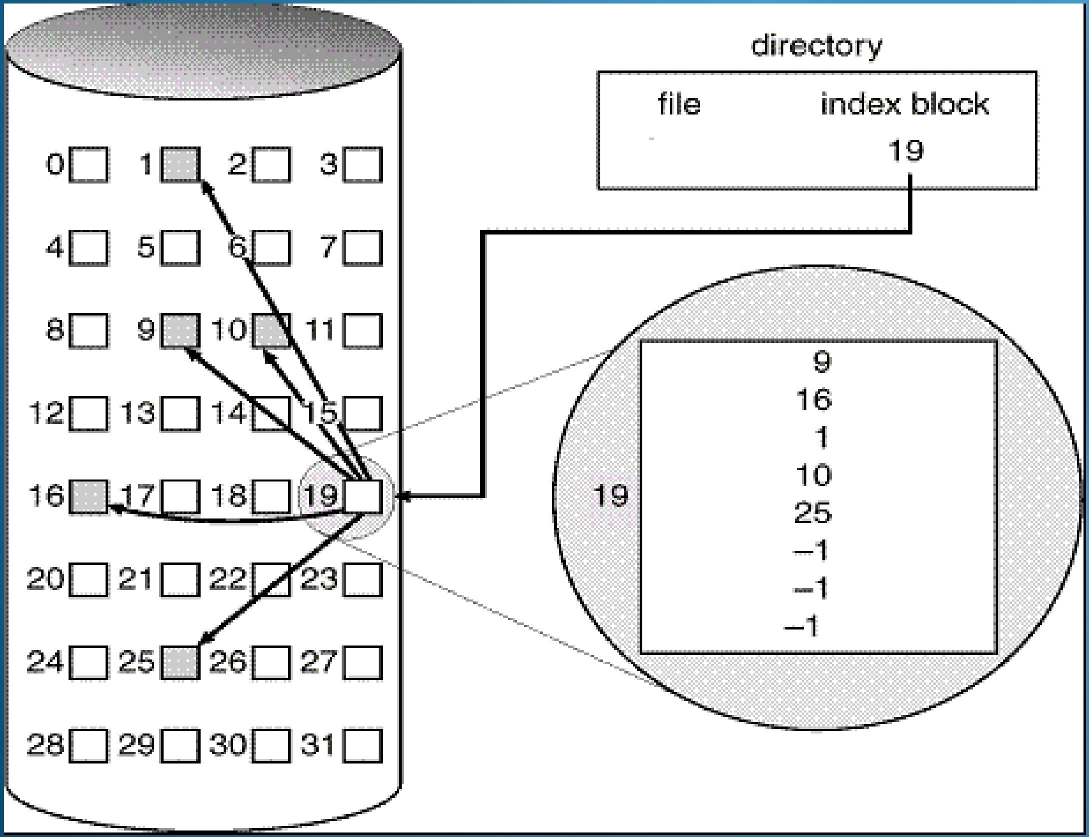


Indexed Allocation

There is also a resource overhead associated with the method compared to contiguous allocation as we need to use some of the available storage for storing the index block(s) for each file. For small files, much of the index block would be wasted and the usage of our storage might not be as efficient as we might like.

CS240 Operating Systems, Communications and Concurrency

Indexed Allocation



Indexed Allocation

The free space can also be managed by having its own index block.

The index block is a critical structure for locating the file blocks. If the index block is damaged or unreadable, it may be impossible to find the data blocks of the file.

CS240 Operating Systems, Communications and Concurrency

Which Allocation Method is Best

File Systems may use a combination of allocation techniques for different types of files.

Files which are static in size may be allocated space contiguously (saving on pointers or index blocks) and giving high access performance as the disk head doesn't have to travel far in order to read all file blocks. Executable files are examples of such files which hardly ever change and are read in their entirety when accessed. Could allocate contiguous clusters with cluster linkage through a FAT structure for dynamic files.

Large database files which require frequent random record access would use indexed allocation to achieve good access performance and efficient dynamic space allocation.

CS240 Operating Systems, Communications and Concurrency

Managing a Hierarchical File System Name Space

As the number of files increases, the number of unique names we use to identify each file becomes unwieldy.

It becomes more difficult to choose new names. If all of our files, perhaps numbering in the thousands, are all contained in the same directory then the number of names involved makes it difficult to remember the name of a particular file we are looking for.

CS240 Operating Systems, Communications and Concurrency

Managing a Hierarchical File System Name Space

If a single directory contains all the files, we have a **flat name space**.

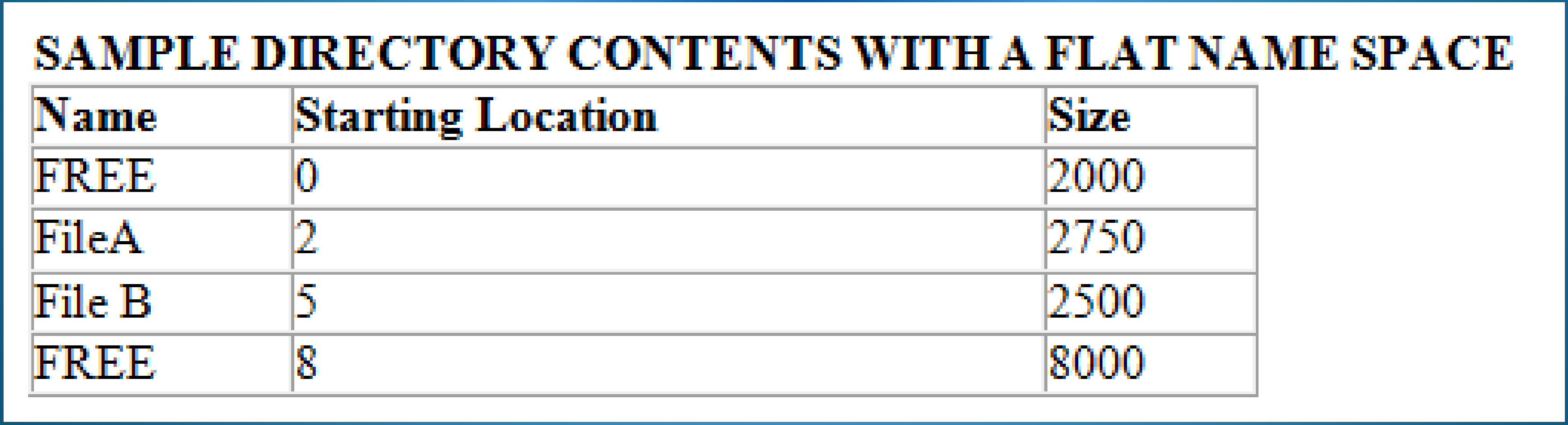
We need to organise files into groups which might relate to a particular application or contain related documents or data and organise those files into separate directories of their own.

As users of the system, when we are focused on a particular activity, all of the files relating to that activity can easily be identified within the subdirectory of interest. It also presents a suitable model for sharing portions of our file space with other users.

CS240 Operating Systems, Communications and Concurrency

Managing a Hierarchical File System Name Space

Our directory structure developed so far contains two types of entries, namely, entries for files and entries for free space regions.



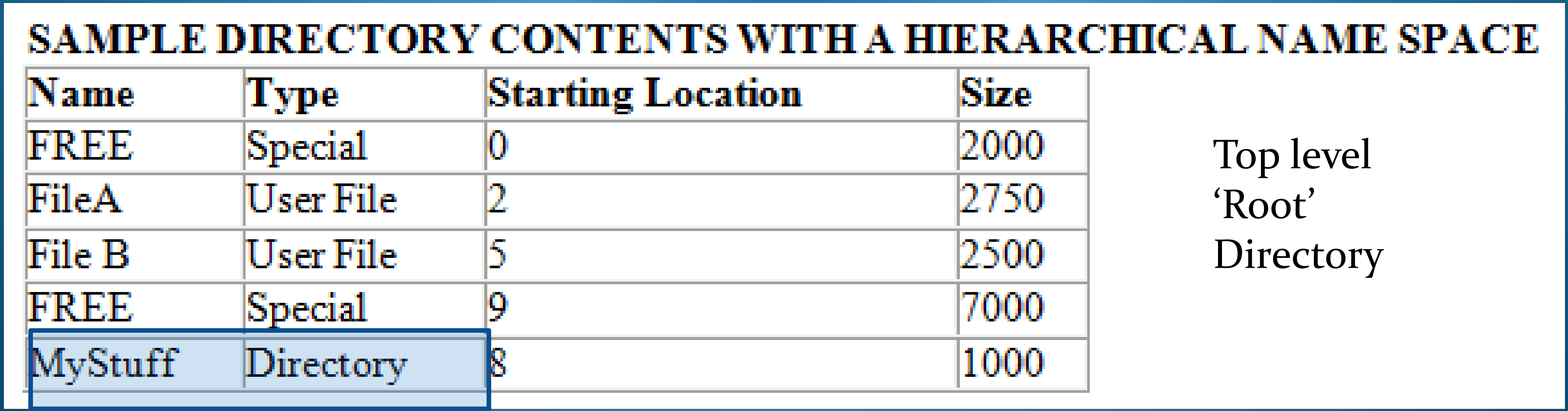
SAMPLE DIRECTORY CONTENTS WITH A FLAT NAME SPACE

Name	Starting Location	Size
FREE	0	2000
FileA	2	2750
File B	5	2500
FREE	8	8000

CS240 Operating Systems, Communications and Concurrency

Managing a Hierarchical File System Name Space

We need to add a third type of entry to support the idea of subdirectories. A directory is a special file containing information about a subset of the file name space and is interpreted and manipulated by file system software. Each directory file will occupy one or more storage blocks.



SAMPLE DIRECTORY CONTENTS WITH A HIERARCHICAL NAME SPACE

Name	Type	Starting Location	Size
FREE	Special	0	2000
FileA	User File	2	2750
File B	User File	5	2500
FREE	Special	9	7000
MyStuff	Directory	8	1000

Top level 'Root' Directory

CS240 Operating Systems, Communications and Concurrency

Managing a Hierarchical File Name Space

Every file in the system must be uniquely identifiable.

Using subdirectories, we, or other users, may create files which have the same name as other files already existing in the system but which are in other subdirectories.

The full name of the file is created by prepending the location or path to the file within the hierarchical name space. This means that only files within the same directory have to have unique names.

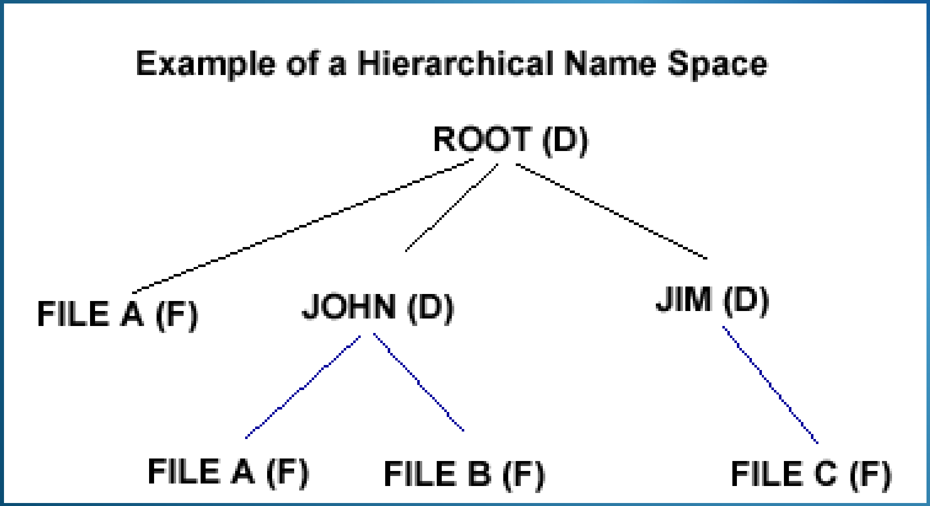
Root Directory – Traversing the file system namespace

The name space ‘tree’ begins with a top level directory file known as the root directory.

This directory may contain subdirectories which in turn may contain other subdirectories until we come to the location of the file being sought.

CS240 Operating Systems, Communications and Concurrency

On Unix systems, the symbol '/' is used to represent the root directory. On windows systems, the symbol '\' is used to represent the root directory. The full path name is unique for every file.



The full path name of the four files in unix would be:
/FILE A
/JOHN/FILE A
/JOHN/FILE B
/JIM/FILE C

The full path name of the four files in windows would be:
\FILE A
\JOHN\FILE A
\JOHN\FILE B
\JIM\FILE C

Current Directories & Search Paths

Sometimes, when we are working with files from a particular subdirectory within the name space, it can be awkward to have to use the full path name to the file all the time, especially if the name space is large and there are many levels in the hierarchy. The path name becomes very long.

Operating systems support the idea of a 'current directory'. If a user supplies the name of a file without beginning at the root directory, then the name may be assumed to be relative to the 'current directory' – an environmental variable that can be changed to the user's locality of interest in the namespace.

A designated set of locations may be searched for the abridged file name in sequence as specified in the PATH environment variable and the current directory may be specified optionally in the PATH.

Navigating the File Namespace

Frequently, users need to alter their current directory as their current activities change. A commonly required function is to move to the directory which contains your current directory (i.e. move upwards in the file tree to the parent directory). Unfortunately, in our directory scheme so far we don't have a convenient way of identifying the parent directory without giving its name from the root of the name space.

Unix and Windows include a pointer to the parent directory within every directory file to provide an easy means of navigating up the file system for the user. If you wish to change your directory to the level above you use the command 'cd ..‘

The symbol '..' is used to represent the parent directory.

Current Directories

An additional entry is also included in the directory to point to itself, in case we ever need to obtain the location of the current directory file from the storage device.

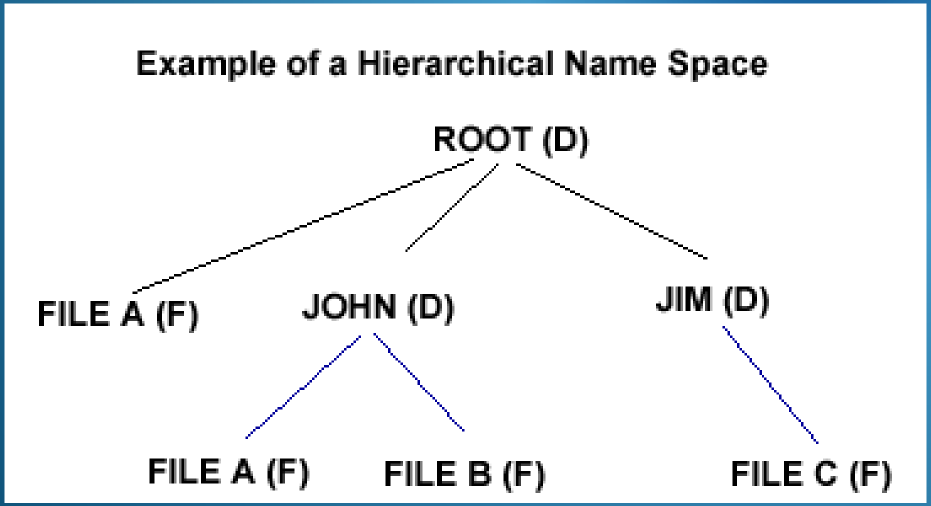
The current directory is identified with the symbol '.' on its own. So executing cd '.' has the effect of moving you to your current directory (i.e. no change). However, in programs or using the shell, you can refer to the directory files '.' and '..' which represent the current and parent directories respectively and this can be a useful short syntax.

If your current directory is not in your PATH then you can access files in your current directory more easily using the syntax ./filename instead of the full path name.

CS240 Operating Systems, Communications and Concurrency

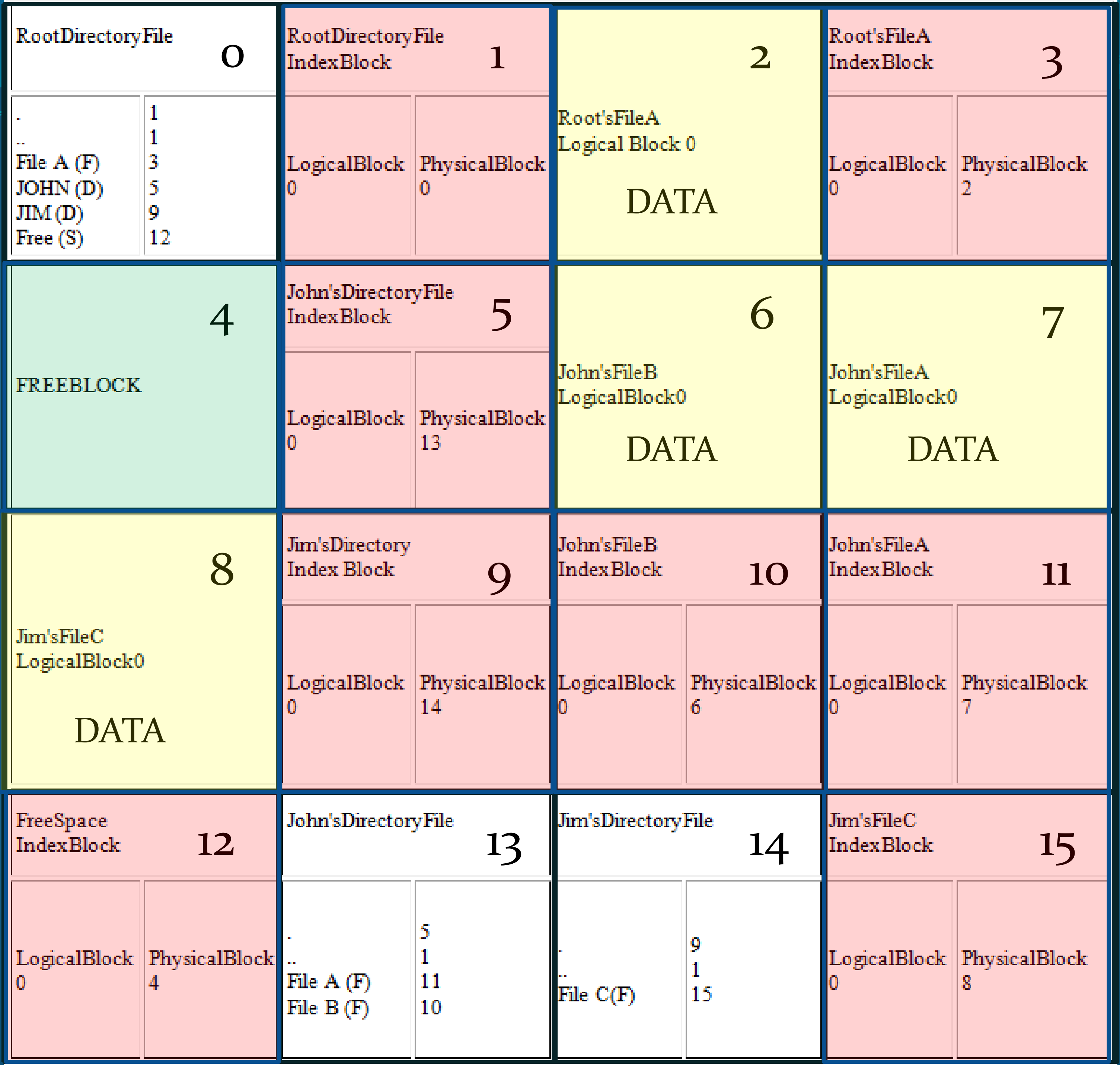
A Sample Implementation

To complete this section, we will demonstrate how the hierarchical name space below could be implemented on a system that uses indexed allocation of space and supports the identification of current and parent directories.



Note: There are 3 User Directories, 4 User Files. We will need 7 index blocks plus 1 for tracking the free block list.

Assume the four files use only one block of disk data each.



RootDirectoryFile 0 . 1 .. 1 File A (F) 3 JOHN (D) 5 JIM (D) 9 Free (S) 12	RootDirectoryFile IndexBlock 1 LogicalBlock 0 | PhysicalBlock 0	2 Root'sFileA Logical Block 0 DATA	Root'sFileA IndexBlock 3 LogicalBlock 0 | PhysicalBlock 2
4 FREEBLOCK	John'sDirectoryFile IndexBlock 5 LogicalBlock 0 | PhysicalBlock 13	6 John'sFileB LogicalBlock0 DATA	7 John'sFileA LogicalBlock0 DATA
8 Jim'sFileC LogicalBlock0 DATA	Jim'sDirectory Index Block 9 LogicalBlock 0 | PhysicalBlock 14	John'sFileB IndexBlock 10 LogicalBlock 0 | PhysicalBlock 6	John'sFileA IndexBlock 11 LogicalBlock 0 | PhysicalBlock 7
FreeSpace IndexBlock 12 LogicalBlock 0 | PhysicalBlock 4	John'sDirectoryFile 13 . 5 .. 1 File A (F) 11 File B (F) 10	Jim'sDirectoryFile 14 . 9 .. 1 File C(F) 15	Jim'sFileC IndexBlock 15 LogicalBlock 0 | PhysicalBlock 8

CS240 Operating Systems, Communications and Concurrency

Unix File System

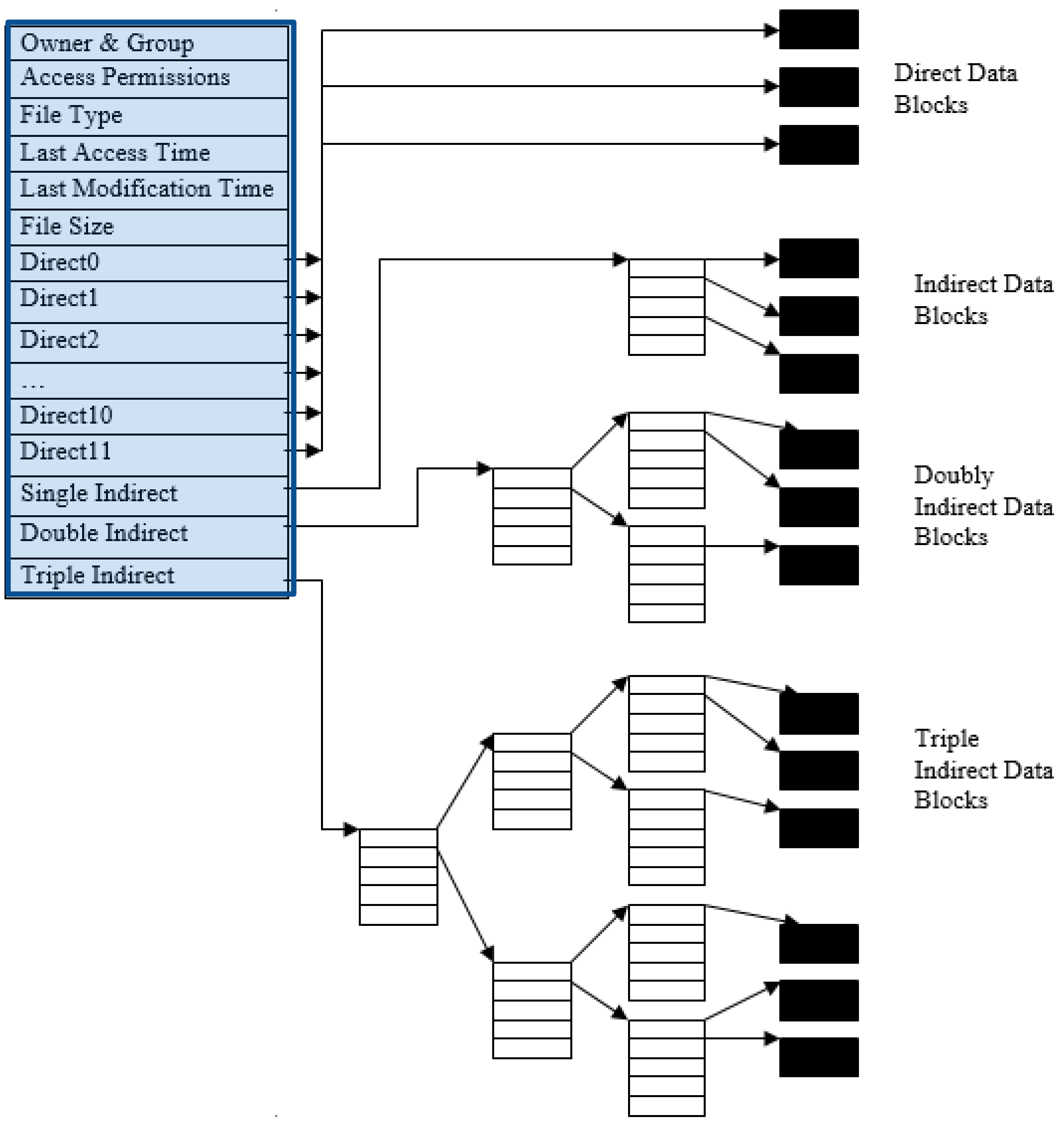
Index Nodes (inodes)

A file on a Unix file system is described by a data structure known as an inode. An inode (index node or information node) describes a file's meta information (its owner, access permissions etc) and contains information about the data blocks associated with the file on the storage device.

Inode structure

The first 12 entries address data block locations directly on the storage device. If each block is 8Kbytes in size, then a total of 8192*12 = 98,304 bytes are directly accessible from the contents of the inode. If the file requires more data blocks, then indirect blocks are created. If disk block addresses are 4bytes, then each indirect block can store up to 2048 disk block pointers. Using up to a single level of indirection, a total of (12+2048)*8192 = 16,875,520 bytes may be accessed. It can be seen that if double and triple indirect blocks are used that a file has a theoretical maximum size of 8 billion blocks or 70 trillion bytes. As can be seen, the system supports efficient random seek across very large files.

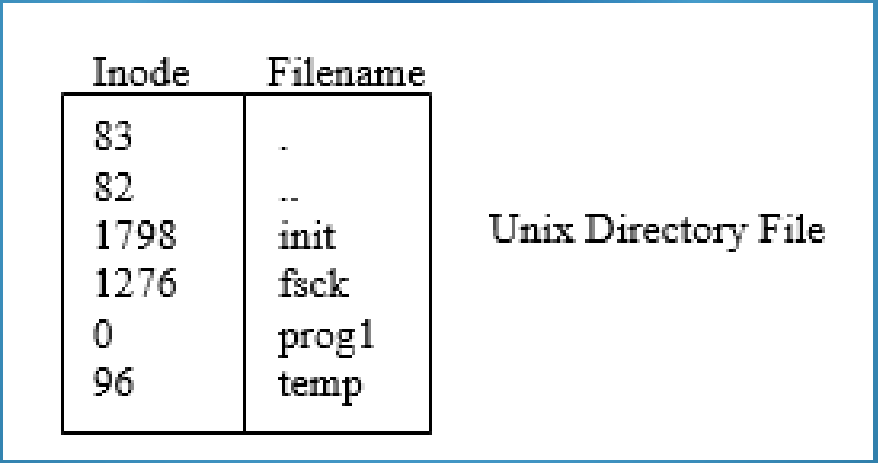
The structure of a typical inode is given below:-



CS240 Operating Systems, Communications and Concurrency

Directory Files

A Unix Directory File maintains mapping information between the text name assigned to a file and the inode structure which describes it.



Inode	Filename
83	.
82	..
1798	init
1276	fsck
0	prog1
96	temp

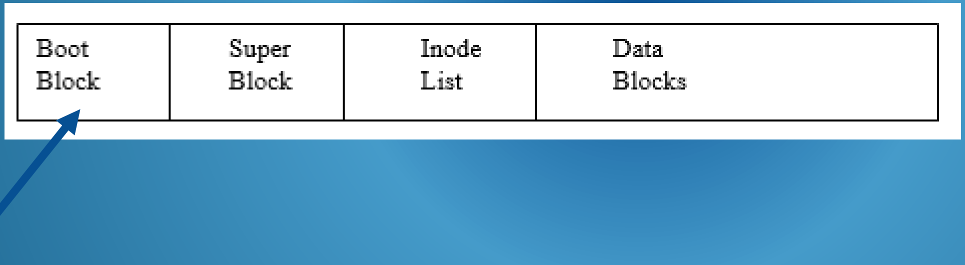
Unix Directory File

File names may be up to 255 bytes in size. The directory contains a pointer to the inode which describes itself '.' and the inode which describes its parent directory '..' to allow simple traversal of the namespace.

CS240 Operating Systems, Communications and Concurrency

File System Structure

The Unix File System has the following general structure as a contiguous area of blocks on a storage device.

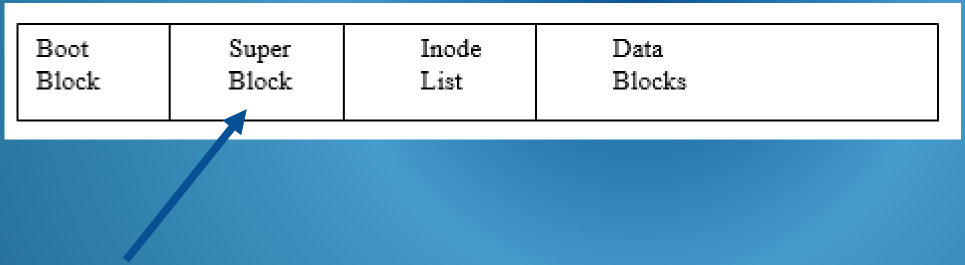


A **Boot Block** contains the source code for bootstrapping the Unix kernel. The bootstrap procedure stored in the ROM BIOS of the system is responsible for locating the device containing the boot block partition and bringing the boot program into memory.

CS240 Operating Systems, Communications and Concurrency

File System Structure

The Unix File System has the following general structure as a contiguous area of blocks on a storage device.

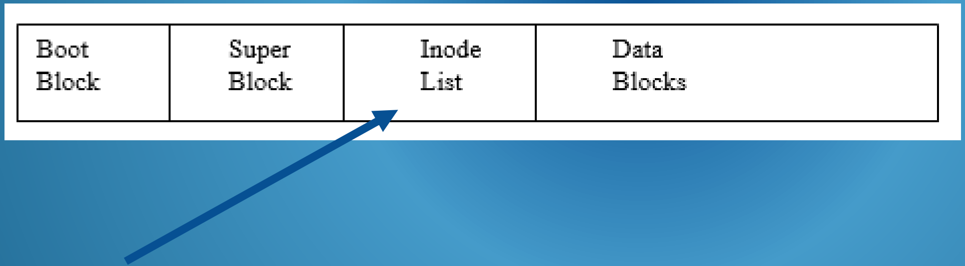


The **Super Block** contains information about the inodes in the file system and the list of free data blocks.

CS240 Operating Systems, Communications and Concurrency

File System Structure

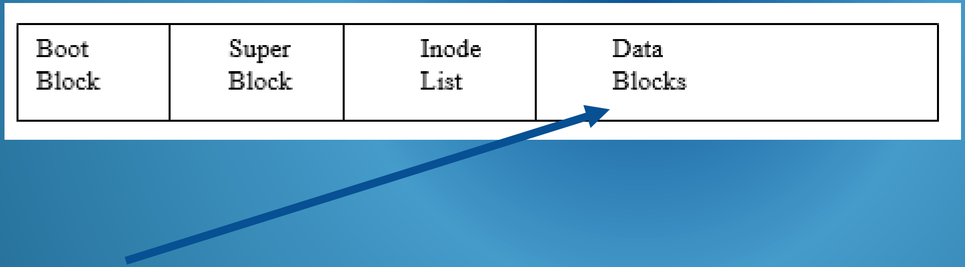
The Unix File System has the following general structure as a contiguous area of blocks on a storage device.



The **inode List** is a collection of inode structures, some of which are in use and others which are free. The size of the inode list determines the maximum number of files that can exist in the file system.

File System Structure

The Unix File System has the following general structure as a contiguous area of blocks on a storage device.



The **Data Blocks** are the largest part of the file system. Data blocks store the file data information as well as directory information and index block information when large files are in use.

Unified File Namespace

Unlike other operating systems where we must specify drive letters to access file systems on different storage devices, Unix unifies the different file systems as one logical hierarchical file system.

Hard disks are often partitioned into several individual logical devices each of which can store a file system structure. These separate file systems can then be mounted into the file tree.

When the system is booted the kernel automatically mounts the root file system. Other file systems are then mounted at directory mount points in the name space.

The system traverses these mount points transparently when parsing the pathname to a file by looking up a mount table whenever it crosses a mount point. The mount table tells the kernel information about the file system that must be searched for the file.

CS240 Operating Systems, Communications and Concurrency

Kernel Data Structures

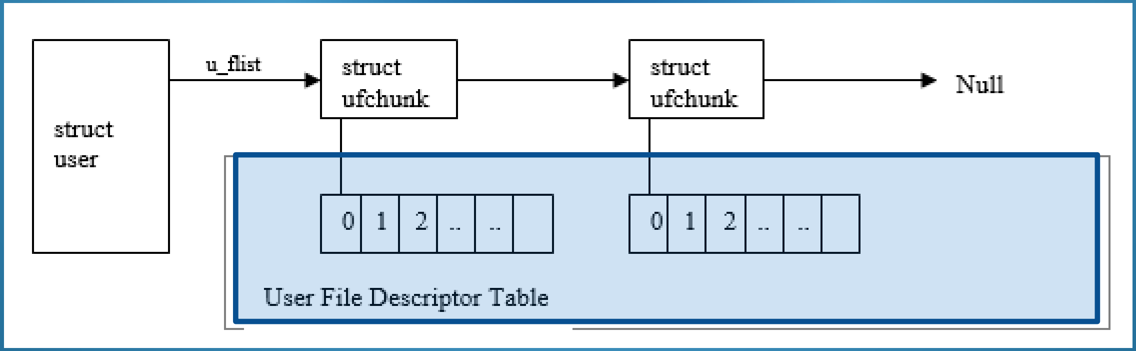
There are three key data structures used internally by the kernel for managing access to files in Unix:-

The User Descriptor Table
The System File Table
The vnode Structure

Kernel Data Structures

The User Descriptor Table

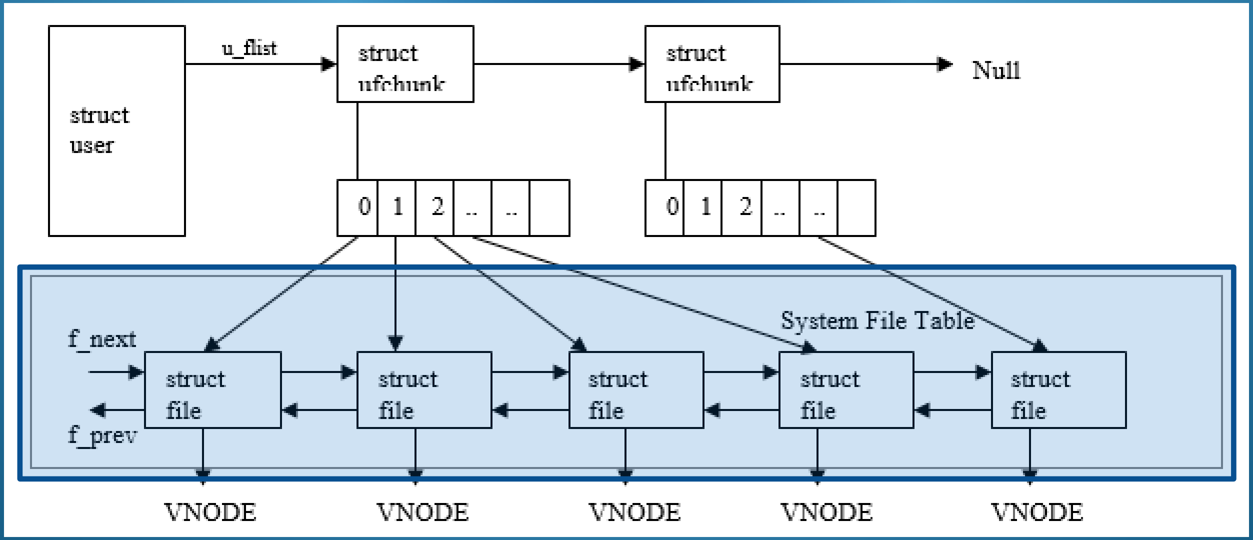
Every active process has its own User Descriptor Table.
This table is essentially an array of pointers to open file structures and other I/O object structures (such as pipes) currently in use by the process. The array elements are numbered from 0 and files are referenced by using one of the array index numbers as a parameter to system calls such as read and write.



Kernel Data Structures

The System File Table

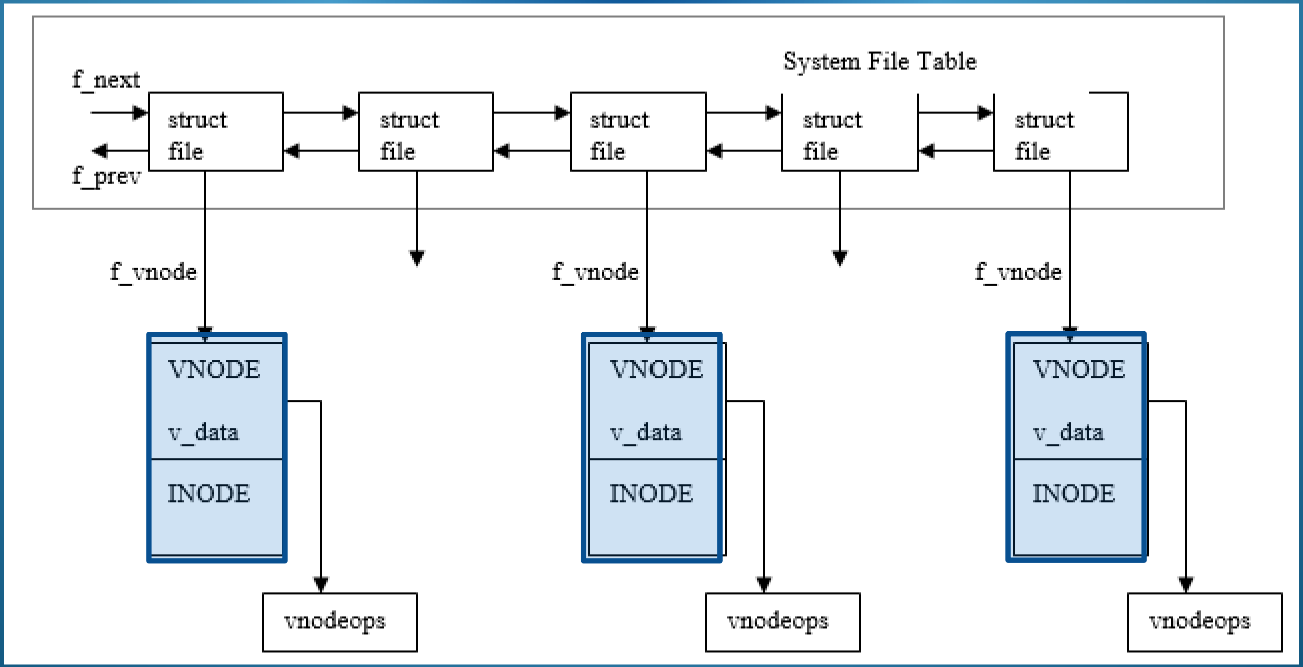
Each element in the User Descriptor Table is a pointer to a file structure. When a file descriptor (from the User Descriptor Table) is allocated to a process, a pointer is put in the allocated position which links to the associated structure in the System File Table. The System File Table is a centralised data structure containing a file structure element for every file or I/O object that is currently in use in the system. Note that for pipe objects, the System File Table contains two file structure elements: One for the reading end and one for the writing end.



Kernel Data Structures

The vnode structure

A vnode (virtual node) is a structure allocated dynamically on a per file basis. The vnode is used to reference the file and contains pointer information to the data blocks of the file obtained from the file's index node (or inode) which is obtained from the information stored on the storage device. Note that every time a file is opened, a new file structure entry is created in the File System Table, but, only one instance of a vnode is required no matter how many times the file has been opened.


System File Table
f_next
f_prev
struct file
struct file
struct file
struct file
struct file
f_vnode
f_vnode
f_vnode
VNODE
v_data
INODE
VNODE
v_data
INODE
VNODE
v_data
INODE
vnodeops
vnodeops
vnodeops