Input/Output

Fred Long

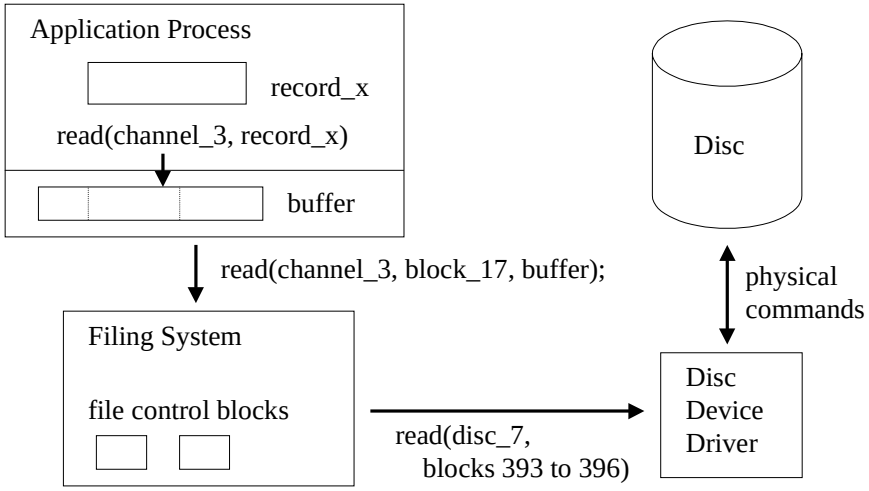
based on slides originally written by Mike Tedd
and modified by Edel Sherratt

Introduction

- Devices vary enormously:
  - simple character I/O, e.g., terminals
  - fixed size blocks, e.g., discs
  - variable size blocks, e.g., magnetic tapes
- I/O speed varies enormously:
  - keyboard: at most a few characters per second
  - VDU screen: perhaps 1000 chars per second
  - discs: millions of cps at peak but typically around 30 complete operations per second

- Processes want:
  - convenient operations, e.g., record I/O
  - device independence
  - sensible support for logical collections of data
- The operating system bridges this gap between physical demands of the devices and logical demands of the processes
- At the lowest level, the *device driver* provides "raw" I/O and hides all the physical problems of the device

- Device drivers try to make all devices look similar
- The filing system understands the layout of file-structured devices, and translates file-oriented requests into device-oriented ones
- Library code, linked into a program, will handle the blocking and unblocking of records (such library code is often called the *access method*)



Synchronous/Asynchronous I/O

- *Synchronous*, or *blocking*, I/O is when the process making a request is blocked until the request can be completed
- Problems for two types of process:
  - those with other work that could be done while this I/O request is handled, e.g., back-ups to tape from disc, a program with several threads but a single OS process

  - processes that want to wait for more than one device or other request
- Hence the need for *asynchronous*, or *non-blocking*, I/O
- Here, the I/O request is dealt with if possible but, if not, the process is quickly returned to, with an indication of status
- The process then continues, and it is notified when the I/O request completes

Unix I/O

- In Unix, all devices, files, etc., are treated as simple streams of bytes
- Devices have names in the file system, e.g., /dev/tty03 /dev/rmt2
- Each process has a “standard input” stream and a “standard output” stream, inherited from its parent
- These streams are typically connected to the keyboard and screen of the current terminal

- The standard I/O streams can be set to be files or pipes
- This is supported by the command line interface shells:
  - prog < file_x > file_y
  - prog_a | prog_b
- In the second example, the shell will first create a pipe, then it will run prog_a passing the pipe as stdout, and prog_b with the pipe as stdin
- The two child processes will be unaware of the nature of the I/O

- 'Standard' Unix has only synchronous I/O
- This leads to costly and clumsy use of extra processes
- BSD and Sun Unix have a simple form of non-blocking I/O

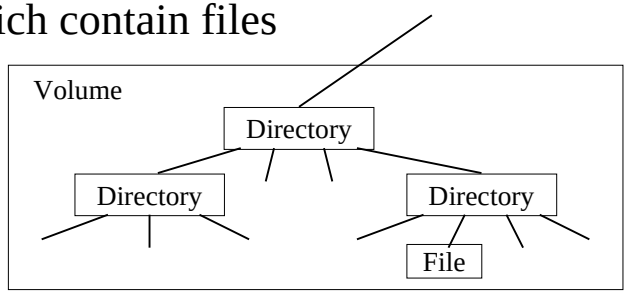
Filing Systems

- Discs are usually organised by a *filing system* (FS)
- Collections of data are given names and allocated space on the disc
- Processes (and users) refer to the data by file names, and by the position of the data within files

- The FS hides considerations of physical layout
- Some other devices are organised by the filing system too, e.g., tapes, bulk memory, CD drives
- We refer to *file-structured* devices

Logical View of Files

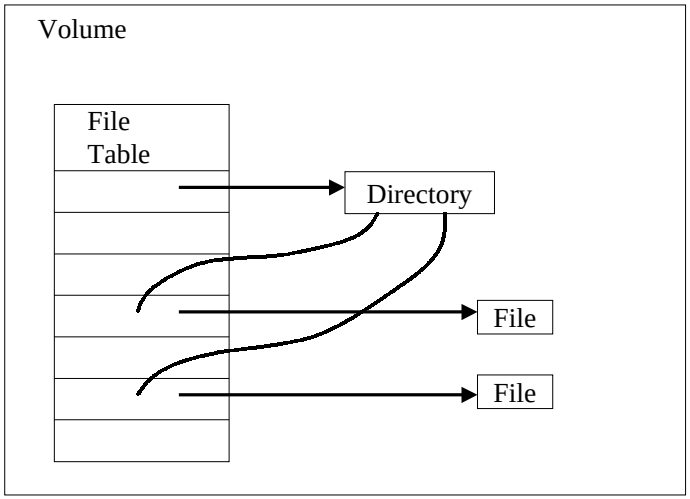
- One disc is split into one or more *volumes* which contain files



A full filename will be the path from the system root
e.g., /usr/spool/News/comp/risks/371

File Tables

- It is usual to have a table in each volume, with an entry for every file in the volume (Unix calls these “inode”s)
- Files are identified (by the FS internally) as position numbers in the file table
- So, pointers from a directory “go through” the file table
- This is done so that we get: improved resilience; easier file moving; a place to put other file information



Organisation of a Single File

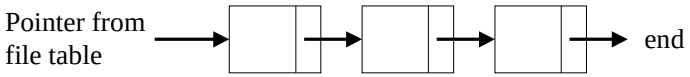
- Objectives:
  - avoid wasting disc space, e.g., through fragmentation
  - keep blocks of the same file on the same or nearby cylinder to minimise head movement
- Three approaches:
  - contiguous files
  - chained files
  - file extents

Contiguous Files

- Each file is held as a single contiguous area on disc
  - simple in concept
  - efficient in use
  - problems of fragmentation
  - difficult to grow files

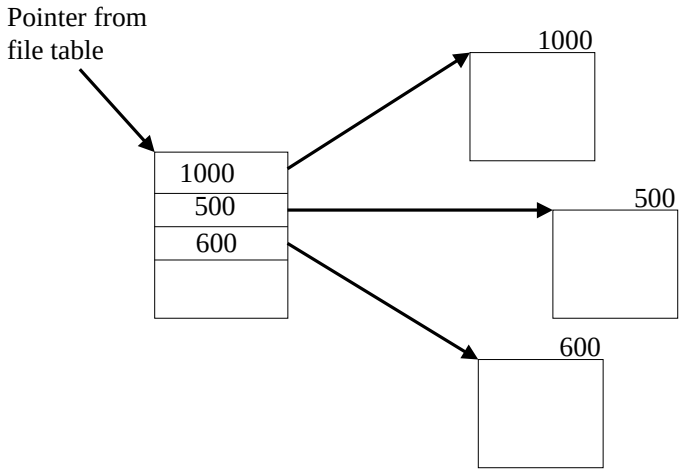
Chained Files

- A file is held as a linked list of blocks
  - avoids most fragmentation
  - lots of head movement unless very careful allocation, can deteriorate badly on nearly full discs
  - random access difficult



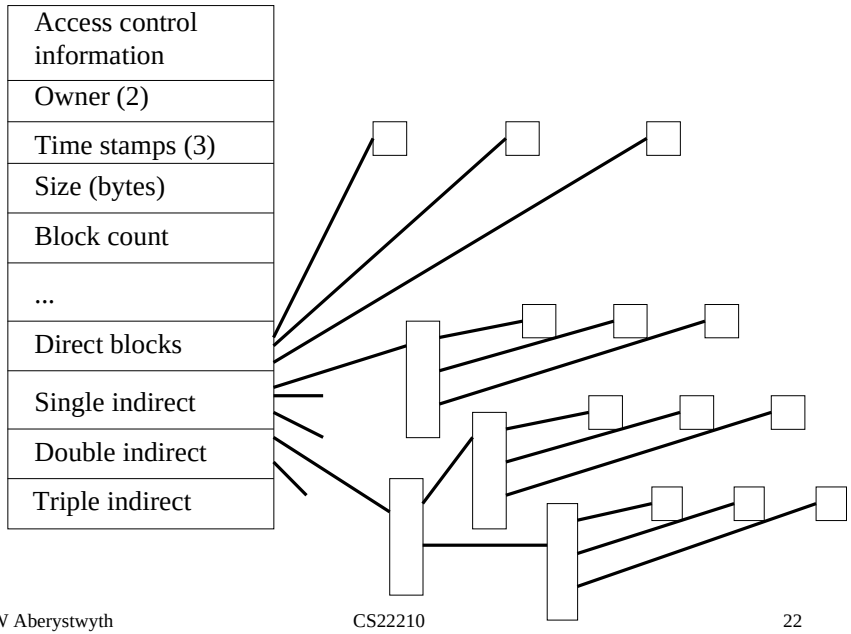
File Extents

- A file is stored as a set of contiguous pieces, with a table showing their sizes and where they are
  - the FS can do arithmetic to find a file location, e.g., in example on next slide, location 1700 is offset 200 in the third extent
  - works very well so long as the disc is not almost full
  - large files may need extra levels of table



Unix Inodes

- Unix uses the idea of file extents, but with fixed size extents
- An *inode* is a single file table entry
- Uses indirection for the later blocks
- Increases the level of indirection as the file gets larger



Free Space Management

- The FS has to keep track of free space so that it can allocate suitable space for new or expanding files
- Free space chaining is not very suitable for allocating space in the most efficient places, or for amalgamating adjacent free blocks
- It is usual to have a *bit map* or *bit vector* — an array of bits, one for each disc block to show if it is free

- The bit map is held on disc (it is large)
- Recent processors have bit-manipulation instructions, e.g., to return the offset in a word of the first set bit
- The bit map enables a natural amalgamation of adjacent free space
- Can easily look in the part of the bit map that relates to the current cylinder
- It is usual to reconstruct the bit map after a system crash

Caching Disc Accesses

- The FS uses a set of main store buffers to hold frequently accessed data
- This gives major savings in accessing:
  - file table
  - extent data for open files
  - bit map
  - frequently accessed directories
- Any changed information is written back to disc at regular intervals

File Compression

- It is possible to represent data in more efficient ways, using different coding schemes, and taking advantage of patterns in the data
- Even without losing data, compression ratios of 2, 10, sometimes even 100, can be achieved
- It depends crucially on the nature of the data (compressed data won't compress much further)

- Compression is an old idea whose time has come, we now have:
  - faster processors
  - but not that much faster discs
  - greater use of networks
  - more 'idle' time
  - more use of pictures and video
- So, now we commonly see tools to compress files for distribution

- De facto standards exist for distributing compressed data, such as:
  - Zip for object code distribution
  - GIF and JPG for images
  - PDF for documents
  - MPEG for video
  - MP3 for audio
- There are products (e.g., Stacker) and OS facilities (e.g., in MS Windows) to hold data on disc in compressed form
- Files are compressed as they are written, and uncompressed as they are read, in 'real time'

- At a cost in processor time, space is being saved on disc — the disc might ‘feel’ twice the size
- The smaller compressed data transfers quicker, which can mean that the system is (overall) faster on the fastest processors
- We are even starting to see products that hold main store data in compressed form

Windows NT File System

- A file in the NTFS is not a simple byte stream, as in Unix
- Instead, it is a structured object consisting of *attributes*
- Each attribute is a byte stream
- Some attributes are standard for all files, e.g., the file name
- Other attributes depend on the file type

- A directory has attributes that implement an index for the file names in the directory
- Ordinary data files have an attribute that contains the data
- Every file is referenced by the *master file table* (MTF)
- Small attributes are stored in the MTF itself
- Large attributes are stored in extents, with pointers to each extent stored in the MTF
- Very large files, or highly fragmented ones, may need to use indirection

- The NTFS name space is organised as a hierarchy of directories
- Every directory uses a *B+ tree* to store an index of the file names in the directory
- A B+ tree is used because:
  - reorganising the tree is very efficient
  - every path from the root to a leaf has the same length
- The *index root* of a directory contains the top level of the B+ tree
- For large directories, all but the root will be contained in extents normally held on disc