

# File Structure

Steve Martin  
Dave Latham

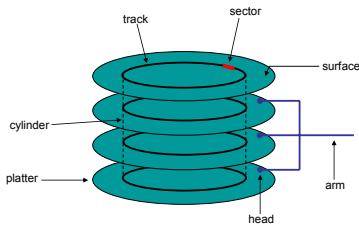
Copyright note: figures and some elements from slides by Hank Levy, CSE451, University of Washington. Thank you, Hank!

# Reading

- Peterson and Silberschatz: 8.3-8.7
- Peterson, Silberschatz and Galvin: 9.3-9.10
- Silberschatz and Galvin: 11.1-11.3, 11.5, 12.1-12.6, 13.1-13.3
- Dietel, pp. 305-316, 322-336
- Tannenbaum: 6.1-6.3
- Silberschatz, Galvin, and Gagne: 11, 12

# Review: physical disk structure

- Disk components
  - platters
  - surfaces
  - tracks
  - sectors
  - cylinders
  - arm
  - heads



# What is a File?

- A named collection of bits, usually stored on a disk, with some properties
  - contents, size, owner, last read/write time, protection, type...
- From the OS's standpoint, a file is a bunch of disk blocks on a secondary storage device.
  - Even if the file is used at a higher level of abstraction (records, bytestreams), this doesn't matter to the file management system.
- A file's *type* can be encoded in the file's name or contents
  - Windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, ...
  - Unix has a smattering of both
    - in content via magic numbers or initial characters (e.g., #!)
- Files are managed by the *file system*

# Basic file operations

## Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

## Windows NT

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, ...)
- WriteFile(handle, ...)
- FlushFileBuffers(handle, ...)
- SetFilePointer(handle, ...)
- CloseHandle(handle, ...)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

# File access methods

- Some file systems provide different access methods that specify ways the application will access data
- Sequential access
  - Read bytes one at a time, in order
  - This is the most common mode.
- Random access
  - Random access given a block/byte #
  - Examples: data set for demand paging, libraries, databases...
- Keyed/Indexing access
  - FS contains an index to a particular field of each record in a file
  - Apps can find a file based on value in that record (similar to DB)
  - Can be considered a form of random access

## Directories

- Directories provide:
  - A way for users to organize their files
  - A convenient file name space for both users and FS's
- Most file systems support multi-level directories
  - Naming hierarchies (`/`, `/usr`, `/usr/local`, `/usr/local/bin`, ...)
- Most file systems support the notion of current directory
- A directory is typically just a file that contains special metadata
  - Directory = list of (name of file, file attributes)
  - Attributes include such things as:
    - Size, protection, location on disk, creation time, access time, ...

## Directories (2)

- Let's say you want to open `"/one/two/three"` in Unix

```
fd = open("/one/two/three", O_RDWR);
```
- What goes on inside the Unix file system?
  - Open directory `"/"` (well known, can always find)
  - Search the directory for "one", get location of "one"
  - Open directory "one", search for "two", get location of "two"
  - Open directory "two", search for "three", get loc. of "three"
  - Open file "three"
  - Permissions are checked at each step
- Unix FS spends lots of time walking down directory paths
  - This is why open is separate from read/write (session state)
  - OS will cache prefix lookups to enhance performance
    - `/a/b`, `/a/bb`, `/a/bbb` all share the `"/a"` prefix

## File System Issues

- Disk Management
  - Make efficient use of disk space
  - Fast access to files
  - Provide hardware independent view of disk to user (and somewhat, to OS also)
- Naming
  - How do users refer to files? (directories, links, etc)
- Protection
  - Want to protect users from each other
  - Want to have files from various users on same disk
  - Want to permit *controlled* sharing
- Reliability
  - Information must be persistent—must last safely for long periods of time.

## Disk Management Issues

- How should the blocks of a file be placed on disk?
  - How should files be organized for best performance?
- How should the map to find and access the blocks in a file look?
- Must take into account user and file characteristics
  - Most files are small, especially in Unix
  - A large proportion of the disk is allocated to a few large files.
  - Many I/O operations are made to large files
  - Most file I/O operations (60-80%) are reads
  - Most file I/O operations are sequential
- From the above characteristics, per-file cost must be low but large files must have good performance.

## File Block Layout and Access

- File Descriptors
  - Data structure that stores:
    - file attributes
    - map which tells you where the blocks of your file are.
  - Stored on disk along with the files
  - In Unix, called *inodes*
  - Don't get confused from Nachos!
- File block layout and access
  - Contiguous
  - Linked
  - Index or tree structured
- This is just standard data structures stuff, only on disk
  - Pointers now point to blocks instead of addresses in memory

## Contiguous Allocation

- Contiguous allocation
  - Allocate file in contiguous set of blocks or tracks
    - When creating a file, make user specify length and allocate all at once, record in file descriptor.
  - Keep a 'free list' of unused areas of the disk
- What are some advantages of this scheme?
- Disadvantages?

## Contiguous Allocation (2)

- Contiguous allocation advantages
  - Sequential and random access are easy
  - Low overhead
  - Simple
  - Few seeks
  - Very good sequential access performance
- Disadvantages:
  - Horrible fragmentation!
  - Hard to predict needs at file creation time
  - May over-allocate
  - Hard to enlarge files
- Can improve this scheme by allocating in fixed contiguous blocks called extents—i.e. if one block isn't enough, ask for another
  - Does this sort of fix sound familiar?

## Linked Allocation

- Linked Allocation
  - Link the blocks of the file together as a linked list.
  - In file descriptor, just keep pointer to first block.
  - In each block of file keep pointer to next block.
- Advantages?
- Disadvantages?

## Linked Allocation (2)

- Advantages of linked file allocation:
  - Files can be extended easily
  - No external fragmentation problem
  - Sequential access is easy—just chase links!
- Disadvantages:
  - How can we do random access?
  - Lots of seeking, blocks are likely not laid on the disk right next to one another
    - Why is this bad?
  - Some overhead in each block for link.

## Indexed Allocation

- (Simple) Indexed allocation
  - Keep array of block pointers for each file
  - File maximum length must be declared at creation time
  - Allocate an array to hold pointers to all blocks
    - But don't allocate blocks
  - Fill in the pointers dynamically using a free list of open blocks.
- Advantages?
- Disadvantages?

## Indexed Allocation (2)

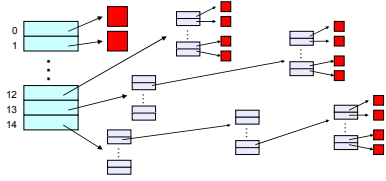
- Advantages of indexed file allocation:
  - Not as much space wasted by over predicting
    - Why?
  - Sequential and random access are easy
  - Only waste space in index
- Disadvantages
  - Still have to set maximum file size
    - Can have overflow scheme if file is larger than predicted
  - Blocks are probably allocated all over disk, so will have lots of seeks.
  - Index array may be large
    - Requires a large file descriptor, more overhead.

## Multi-level Indexed Allocation

- This is the VAX Unix solution.
  - In general, any sort of multi-level tree-like structure. Specifically, what we will describe is what Berkeley 4.3 BSD Unix does.
- Each file descriptor contains file properties and 15 pointers.
  - First 12 point to data blocks
  - Next three to indirect, doubly-indirect, and triply-indirect blocks
    - 256 pointers in each block
  - Descriptor space isn't allocated until needed
- File is a tree of pointers to blocks.

## Indexed allocation example

- Create a 'tree' of block pointers to find all the blocks of a given file:



## Multi-level Indexed Allocation (2)

- Advantages of Multi-level Indexed Allocation
  - Simple
  - Easy to implement
  - Incremental expansion
  - Easy access to small files
    - Why?
  - Good random access to blocks
  - Easy to insert blocks in the middle of file
  - Easy to append to file
  - Small file map
- Disadvantages
  - Maximum file size is still fixed, although it is large
  - Indirect mechanism isn't efficient for large files
    - Three descriptor ops for each operation
  - File usually isn't allocated continuously, so need to seek between blocks.

## How do we find a free block?

- If all blocks are the same size, can use *bit map*
  - One bit per disk block, cache parts of map in memory
- If blocks are variable size, use free list
  - Requires free storage area management
    - Fragmentation and compaction
- In Unix, blocks are grouped for efficiency
  - Each block in free list contains pointers to many free blocks, plus pointer to next block on list.
  - Aren't many references involved in allocation or de-allocation
- Organization of free list means files get spread to different blocks all over disk.
  - Why is this bad?

## How do we find a free block? (2)

- A more efficient solution
  - Allocate groups of sequential blocks
  - Use multi-level index scheme, except each pointer is to a sequence of free blocks on disk.
- When we need a free block, try to allocate next physical block on track (or in cylinder)
- If we have detected a sequence of sequential writes, grab a group of blocks at a time, and release them later if unused
  - Size of group depends on number of sequential writes we've seen so far.
- Keep part of disk unallocated always to maximize probability of finding a sequential block to allocate.