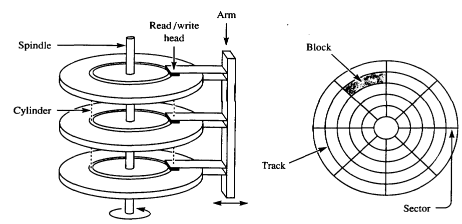
Lecture 15: File system interface

- File system interface
  - reasons for delegating storage management to OS
  - file definition and operations on a file
  - file access patterns
  - directories
- File system structures
  - I-nodes
  - on-disk file system structures
  - in-memory file system structures
- Disk space allocation methods
  - continuous
  - chained (linked)
  - indexed
  - multilevel indexing

Why OS manages secondary storage?

- Levels of abstraction
- Hardware underneath

	applications	daemons	servers
User Interface	create( ) delete( )	open( ) rename( )	close( ) link( ) read( ) write( )
Device-Independent Interface	tracks	sectors	blocks
	seek( )	readblock( )	writeblock( )
Device Interface	disk	other hardware	



Why OS manages secondary storage? (cont.)

- Important to the user:
  - Persistence — data stays around between power cycles and crashes
  - Ease of use — can easily find, examine, modify, etc. data
  - Efficiency — uses disk space well
  - Speed — can get to data quickly
  - Protection — others can't corrupt (or sometimes even see) my data
- OS provides:
  - File system with directories and naming — allows user to specify directories and names instead of location on disk
  - Disk management — keeps track of where files are located on the disk, accesses those files quickly
  - Protection — no unauthorized access

What is a file

- A *file* is a logical unit of storage:
  - A series of records (IBM mainframes)
  - A series of bytes (UNIX, most PCs)
  - A resource fork and data fork (Macintosh)
    - Resource fork — labels, messages, etc.
    - Data fork — code and data
- What is stored in a file?
  - C++ source code, object files, executable files, shell scripts, PostScript…
  - Macintosh OS explicitly supports file types — TEXT, PICT, etc.
  - Windows uses file naming conventions — ".exe" and ".com" for executables, etc.
  - UNIX looks at contents to determine type:
    - Shell scripts — start with "#!"
    - PostScript — starts with "%!PS-Adobe…"
    - Executables — starts with *magic number*

File Operations

- **Create**(*name*)
  - Constructs an *i-node* on disk to represent the newly created file
    - Adds an entry to the *directory* to associate *name* with that *i-node*
  - Allocates disk space for the file
    - Adds disk location to file descriptor
- *fileid* = **Open**(*name*, *mode*)
  - Allocates a unique identifier called the *file ID* (identifier) (returned to the user)
  - Sets the mode (r, w, rw) to control concurrent access to the file
- **Close**(*fileId*)
- **Delete**(*fileId*)
  - Deletes the file's inode from the disk, and removes it from the directory

Common file access patterns

- Sequential access
  - Data is processed in order, one byte at a time, always going forward
  - Most accesses are of this form
  - Example: compiler reading a source file
- Direct / random access
  - Can access any byte in the file directly, without accessing any of its predecessors
  - Example: accessing database record 12
- Keyed access
  - Can access a byte based on a *key* value
  - Example: database search, dictionary
  - OS usually does not support keyed access
    - User program must determine the address from the key, then use random access (provided by the OS) into the file

File Operations (cont.)

- **Read**(*fileId*, *from*, *size*, *bufAddress*)
  - Random access read
  - Reads *size* bytes from file *fileId*, starting at position *from*, into the buffer specified by *bufAddress*

```
for (pos=from, i=0 ; i < size ; i++)
    bufAddress[i] = file[pos++];
```

- **Read**(*fileId*, *size*, *bufAddress*)
  - Sequential access read
  - Reads *size* bytes from file *fileId*, starting at the current file position *fp*, into the buffer specified by *bufAddress*, and then increments *fp* by *size*

```
for (pos=fp, i=0 ; i < size ; i++)
    bufAddress[i] = file[pos++];
fp += size;
```

- **Write** — similar to **Read**

Directories and naming

- Directories of named files
  - User and OS must have some way to refer to files stored on the disk
  - OS needs to use numbers (index into an array of inodes) (efficient, etc.)
  - User wants to use textual names (readable, mnemonic, etc.)
  - OS uses a *directory* to keep track of names and corresponding file indices
- Simple naming
  - One name space for the entire disk
    - Every name must be unique
  - Implementation:
    - Store directory on disk
    - Directory contains <name, index> pairs
  - Used by early mainframes, early Macintosh OS, and MS DOS

Directories and naming (cont.)

- User-based naming
  - One name space for each user
    - Every name in that user's directory must be unique, but two different users can use the same name for a file in their directory
  - Used by TOPS-10 (DEC mainframe from the early 1980s)
- Multilevel naming
  - Tree-structured name space
  - Implementation:
    - Store directories on disk, just like files
    - Each directory contains <name, index> pairs in no particular order
      - The file pointed to by a directory can be another directory
        - Names have "/" separating levels
      - Resulting structure is a tree of directories
  - Used by UNIX

Unix directories

- multilevel naming - unambiguous but cumbersome:
- this file's name is: /home/mikhail/public_html/classes/os.s00/L17files.PDF
- OS maintains *current working directory* as part of process state thus a program may refer to files relative to current working directory
- each directory has two special entries:
  . - refers to the directory itself
  .. - refers to parent directory (root directory has .. pointing to itself)
- each file can be listed in several different directories. Each reference is called *hard-link;* file continues to exist as long as there is at least one hard-link

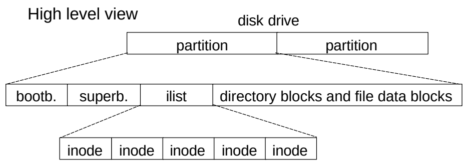
I-nodes

- Every file is described by an i-node (UNIX term - stands for index node), which may contain (varies with OS):
  - Type
  - Access permissions — read, write, etc.
  - Link count — number of directories that contain this file
  - Owner, group
  - Size
  - Access times — when created, last accessed, last modified
  - Blocks where file is located on disk
- Not included:
  - Name of file

Unix file system

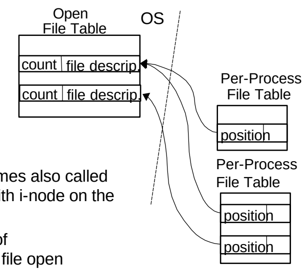
- An *i-node* represents a file
  - All *i-nodes* are stored on the disk in a fixed-size array called the *ilist*
    - The size of the ilist array is determined when the disk is initialized
    - The index of a file descriptor in the array is called its *inode number*, or *inumber*
  - I-nodes for active files are also cached in memory in the *active inode table*
- A UNIX disk may be divided into *partitions*, each of which contains:
  - Blocks for storing directories and files
  - Blocks for storing the ilist
    - i-nodes corresponding to files
  - Some special i-nodes
    - Boot block — code for booting the system
    - Super block - size of disk, number of free blocks, list of free blocks, size of ilist, number of free inodes in ilist, etc.

Unix file system (cont.)



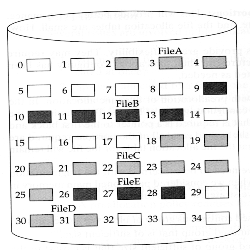
- Note that in practice the picture is more complicated because the directory structure is added:
  - a directory is a file that contains the list of file names and inodes

File descriptors, file tables



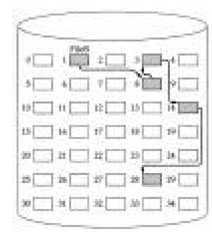
- *Open file table* (one, belongs to OS)
  - Lists all open files
  - Each entry contains:
    - A *file descriptor (*sometimes also called i-node - don't confuse with i-node on the disk!)
    - Open count — number of processes that have the file open
- *Per-process file table* (many)
  - List all open files for that process
  - Each entry contains:
    - Pointer to entry in open file table
    - Current position (offset) in file

Continuous allocation



- OS keeps an ordered list of free blocks
- allocates contiguous groups of blocks when it creates a file
- I-node must store start block and length of file
- Used in IBM 370, some write-only disks
- Have to use usual dynamic allocation techniques (first-fit, best-fit, etc.)
- advantages:
  - simple
  - efficient sequential access, reasonably efficient random access
  - small number of seeks (head movement), most may be on same track
  - good for storing data on CDROM
- disadvantages:
  - May have difficulty changing file size (size is specified when file is created)
  - Problems with external fragmentation
- *compaction* - relocation of files so that there is no free space between them

Chained (linked) allocation

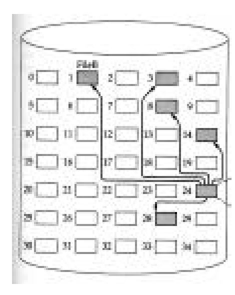


- OS keeps an ordered list of free blocks
- File descriptor stores pointer to first block
- Each block stores pointer to next block
- Used in DEC TOPS-10, Xerox Alto
- Advantages:
  - No external fragmentation
  - Any free space is as good as any other
  - Can easily change file size (no need to declare in advance)
- Disadvantages:
  - Reasonable, but not great, for sequential access (one seek each time)
  - Inefficient random access — have to follow many links
  - Lost space due to lots of pointers (small, but adds up)
  - Lots of seeks to find a block
  - Not very robust — lose a block, lose rest of file
- compaction can be used so that blocks of one file are located continuously on the disk - optimizes disk access.

Linked allocation variant: FAT

- File allocation table (FAT) a variant of linked list allocation
- used in MS-DOS, OS/2, Windows
- the beginning of each partition contains the indexing table FAT
- directory entry has the number of the first block
- FAT
  - has an entry for each block
  - the FAT entry indexed by the first block has the number of the second block and so on
  - last block's FAT entry has reserved EOF symbol
  - unused blocks have \0 entry

Indexed allocation

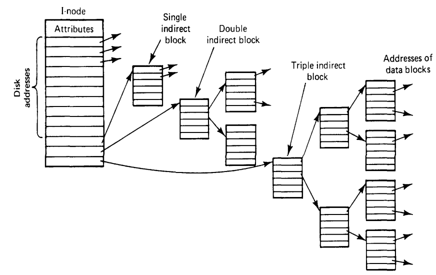


- OS keeps a list of free blocks
  - OS allocates an array (called the index block) to hold pointers to all the blocks used by the file
  - Allocates blocks only on demand
- Used in DEC VMS, Nachos
- Advantages:
  - Can easily grow up to maximum size
  - Both sequential and random accesses are easy
- Disadvantages:
  - Limit on maximum file size
  - Lots of seeks since data isn't contiguous
  - Wasted space for pointers (need a whole block even for one pointer)

Multilevel indexed allocation

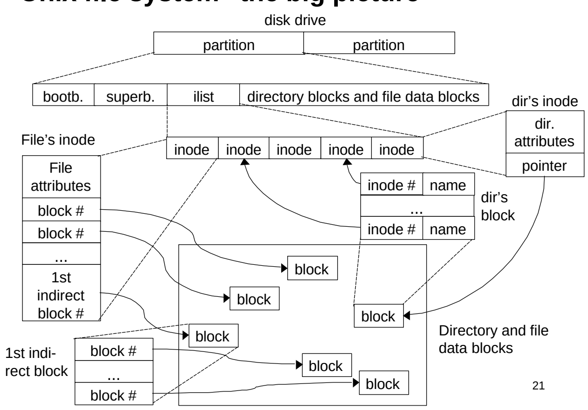
- Used in UNIX (numbers below are for traditional UNIX, BSD UNIX 4.1)
- Each inode contains 13 *block pointers*
  - First 10 pointers point to data blocks (each 512 bytes long) of a file
    - If the file is bigger than 10 blocks (5,120 bytes), the 11th pointer points to a *single indirect block*, which contains 128 pointers to 128 more data blocks (can support files up to 70,656 bytes)
      - If the file is bigger than that, the 12th pointer points to a *double indirect block*, which contains 128 pointers to 128 more single indirect blocks (can support files up to 8,459,264 bytes)
        - If the file is bigger than that, the 13th pointer points to a *triple indirect block*, which contains 128 pointers to 128 more double indirect blocks
  - Max file size is 1,082,201,087 bytes

Multilevel indexed allocation (cont.)



- Advantages:
  - Simple to implement
  - Supports incremental file growth
  - Works well for small files
- Disadvantages:
  - Inefficient access to very large files
  - Data gets spread around - more seeks

Unix file system “the big picture”



Working with directories (Lookup)

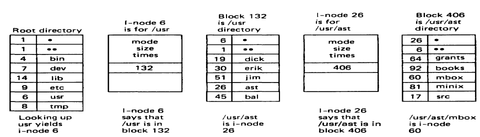


Fig. 4-16. The steps in looking up */usr/ast/mbox*.

- A directory is a table of entries:
  - 2 bytes — inumber
  - 14 bytes — file name (improved in BSD 4.2 and later)
- Search to find the file begins with either root, or the current working directory
  - Inode 2 points to the root directory (“ / ”)
  - Example above shows lookup of /usr/ast/mbox