I/O Systems

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- **Device characteristics**
  - Block device vs. Character device
  - Direct I/O vs. Memory-mapped I/O
  - Polling vs. Interrupts
  - Programmed I/O vs. DMA
  - Blocking vs. Non-blocking I/O

- **I/O software layers**
A Typical PC Bus Structure
I/O Devices (1)

- **Block device**
  - Stores information in fixed-size blocks, each one with its own address.
  - 512B – 32KB per block
  - It is possible to read or write each block independently of all the other ones.
  - Disks, tapes, etc.

- **Character device**
  - Delivers or accepts a stream of characters.
  - Not addressable and no seek operation.
  - Printers, networks, mice, keyboards, etc.
I/O Devices (2)

- Device controller (or host adapter)
  - I/O devices have components:
    - Mechanical component
    - Electronic component
  - The electronic component is the device controller.
    - May be able to handle multiple devices.
  - Controller’s tasks
    - Convert serial bit stream to block of bytes.
    - Perform error correction as necessary.
    - Make available to main memory.
Accessing I/O Devices (1)

- **Direct I/O**

  - Use special I/O instructions to an I/O port address.

<table>
<thead>
<tr>
<th>I/O address range (hexadecimal)</th>
<th>device</th>
</tr>
</thead>
<tbody>
<tr>
<td>000–00F</td>
<td>DMA controller</td>
</tr>
<tr>
<td>020–021</td>
<td>interrupt controller</td>
</tr>
<tr>
<td>040–043</td>
<td>timer</td>
</tr>
<tr>
<td>200–20F</td>
<td>game controller</td>
</tr>
<tr>
<td>2F8–2FF</td>
<td>serial port (secondary)</td>
</tr>
<tr>
<td>320–32F</td>
<td>hard-disk controller</td>
</tr>
<tr>
<td>378–37F</td>
<td>parallel port</td>
</tr>
<tr>
<td>3D0–3DF</td>
<td>graphics controller</td>
</tr>
<tr>
<td>3F0–3F7</td>
<td>diskette-drive controller</td>
</tr>
<tr>
<td>3F8–3FF</td>
<td>serial port (primary)</td>
</tr>
</tbody>
</table>
### Accessing I/O Devices (2)

- **Memory-mapped I/O**
  - The device control registers are mapped into the address space of the processor.
    - The CPU executes I/O requests using the standard data transfer instructions.
  - I/O device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O
    - Can give a user control over specific devices but not others by simply including the desired pages in its page table.
  - Reading a device register and testing its value is done with a single instruction.
Polling vs. Interrupts (1)

### Polled I/O

- CPU asks ("polls") devices if need attention.
  - ready to receive a command
  - command status, etc.

- **Advantages**
  - Simple
  - Software is in control.
  - Efficient if CPU finds a device to be ready soon.

- **Disadvantages**
  - Inefficient in non-trivial system (high CPU utilization).
  - Low priority devices may never be serviced.
Polling vs. Interrupts (2)

- **Interrupt-driven I/O**
  - I/O devices request interrupt when need attention.
  - Interrupt service routines specific to each device are invoked.
  - Interrupts can be shared between multiple devices.
  - **Advantages**
    - CPU only attends to device when necessary.
    - More efficient than polling in general.
  - **Disadvantages**
    - Excess interrupts slow (or prevent) program execution.
    - Overheads (may need 1 interrupt per byte transferred)
Polling vs. Interrupts (3)

1. CPU
   - device driver initiates I/O
   - CPU executing checks for interrupts between instructions

2. I/O controller
   - initiates I/O

3. input ready, output complete, or error generates interrupt signal

4. CPU receiving interrupt, transfers control to interrupt handler
   - interrupt handler processes data, returns from interrupt

5. CPU resumes processing of interrupted task

6. CPU resumes processing of interrupted task
Programmed I/O vs. DMA

- Programmed I/O
  - CPU transfers data between I/O device and memory.
    - Read I/O data from memory-mapped I/O ports
    - Store I/O data to memory buffer
  - Pros
    - Effective for small I/O data
  - Cons
    - Precious CPU cycles are used for transferring I/O data
    - Slow I/O device
Programmed I/O vs. DMA

- DMA (Direct Memory Access)
  - Bypasses CPU to transfer data directly between I/O device and memory.
  - Used to avoid programmed I/O for large data movement.
Blocking vs. Non-Blocking I/O

- **Blocking I/O**
  - Process is suspended until I/O completed.
  - Easy to use and understand.

- **Nonblocking I/O**
  - I/O call returns quickly, with a return value that indicates how many bytes were transferred.
  - A nonblocking read() returns immediately with whatever data available – the full number of bytes requested, fewer, or none at all.
Goals of I/O Software

- **Goals**
  - Device independence
  - Uniform naming
  - Error handling
  - Buffering
  - Sharable vs. dedicated devices
I/O Software Layers

- User-level I/O Software
  - Device-independent I/O Software
    - Device Drivers
    - Interrupt Handlers
  - Hardware
    - Network
      - Storage devices
      - Input devices
      - Display device
## Interrupt Handlers

### Handling interrupts

- **Critical actions**
  - Acknowledge an interrupt to the PIC.
  - Reprogram the PIC or the device controller.
  - Update data structures accessed by both the device and the processor.

- **Noncritical actions**
  - Update data structures that are accessed only by the processor. (e.g., reading the scan code from the keyboard)

- **Noncritical deferred actions**
  - Actions may be delayed.
  - Copy buffer contents into the address space of some process (e.g., sending the keyboard line buffer to the terminal handler process).
  - **Bottom half (Linux)**

**Reenable interrupts**

**Return from interrupts**
Device Drivers (1)

- **Device drivers**
  - Device-specific code to control each I/O device interacting with device-independent I/O software and interrupt handlers.
  - Requires to define a well-defined model and a standard interface of how they interact with the rest of the OS.
  - Implementing device drivers:
    - Statically linked with the kernel.
    - Selectively loaded into the system during boot time.
    - Dynamically loaded into the system during execution. (especially for hot pluggable devices).
Device Drivers (2)
Device Drivers (3)

The problem

- Reliability remains a crucial, but unresolved problem
  - 5% of Windows systems crash every day
  - Huge cost of failures: stock exchange, e-commerce, ...
  - Growing “unmanaged systems”: digital appliances, consumer electronics devices

- OS extensions are increasingly prevalent
  - 70% of Linux kernel code
  - Over 35,000 drivers with over 120,000 versions on Windows XP
  - Written by less experienced programmer

- Extensions are a leading cause of OS failure
  - Drivers cause 85% of Windows XP crashes
  - Drivers are 7 times buggier than the kernel in Linux
Device-Independent I/O SW (1)

- **Uniform interfacing for device drivers**
  - In Unix, devices are modeled as special files.
    - They are accessed through the use of system calls such as open(), read(), write(), close(), ioctl(), etc.
    - A file name is associated with each device.
  - Major device number locates the appropriate driver.
    - Minor device number (stored in i-node) is passed as a parameter to the driver in order to specify the unit to be read or written.
  - The usual protection rules for files also apply to I/O devices.
Device-Independent I/O SW (3)

- **Buffering**
  - (a) Unbuffered
  - (b) Buffered in user space
  - (c) Buffered in the kernel space
  - (d) Double buffering in the kernel

![Buffering Diagram]

User space

Kernel space

User process

Modem (a)

Modem (b)

Modem (c)

Modem (d)
Device-Independent I/O SW (4)

- Error reporting
  - Many errors are device-specific and must be handled by the appropriate driver, but the framework for error handling is device independent.
  - Programming errors vs. actual I/O errors
  - Handling errors
    - Returning the system call with an error code.
    - Retrying a certain number of times.
    - Ignoring the error.
    - Killing the calling process.
    - Terminating the system.
Device-Independent I/O SW (4)

- Allocating and releasing dedicated devices
  - Some devices cannot be shared.
    1. Require processes to perform open()’s on the special files for devices directly.
       - The process retries if open() fails.
    2. Have special mechanisms for requesting and releasing dedicated devices.
       - An attempt to acquire a device that is not available blocks the caller.

- Device-independent block size
  - Treat several sectors as a single logical block.
  - The higher layers only deal with abstract devices that all use the same block size.
Storage Systems

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- HDDs (Hard Disk Drives)
- Disk scheduling policies
Secondary Storage

Secondary storage usually
- is anything that is outside of “primary memory”.
- does not permit direct execution of instructions or data retrieval via machine load/store instructions.

Characteristics
- It’s large: 100GB and more
- It’s cheap: 1TB SATA2 disk costs ₩66,000.
- It’s persistent: data survives power loss.
- It’s slow: milliseconds to access.
HDDs (1)

Electromechanical
- Rotating disks
- Arm assembly

Electronics
- Disk controller
- Buffer
- Host interface
HDDs (2)

- **Seagate Barracuda ST31000528AS (1TB)**
  - 4 Heads, 2 Discs
  - Max. recording density: 1413K BPI (bits/inch)
  - Avg. track density: 236K TPI (tracks/inch)
  - Avg. areal density: 329 Gbits/sq.inch
  - Spindle speed: 7200rpm (8.3ms/rotation)
  - Average seek time: < 8.5ms (read), < 9.5ms (write)
  - Max. internal data transfer rate: 1695 Mbits/sec
  - Max. I/O data transfer rate: 300MB/sec (SATA-2)
  - Max. sustained data transfer rate: 125MB/sec
  - Internal cache buffer: 32MB
  - Max power-on to ready: < 10.0 sec
HDDs (3)

- **Hard disk internals**

  - Our Boeing 747 will fly at the altitude of only a few mm at the speed of approximately 65mph periodically landing and taking off.
  - And still the surface of the runway, which consists of a few mm-thick layers, will stay intact for years.
Managing Disks (1)

- Interacting with disks
  - Specifying disk requests requires a lot of info:
    - Cylinder #, surface #, track #, sector #, transfer size, etc.
  - Older disks required the OS to specify all of this
    - The OS needs to know all disk parameters.
  - Modern disks are more complicated.
    - Not all sectors are the same size, sectors are remapped, etc.
  - Current disks provide a higher-level interface (e.g., SCSI)
    - The disks exports its data as a logical array of blocks [0..N-1]
    - Disk maps logical blocks to cylinder/surface/track/sector.
    - Only need to specify the logical block # to read/write.
    - As a result, physical parameters are hidden from OS.
Managing Disks (2)

- Disk performance
  - Performance depends on a number of steps
    - **Seek**: moving the disk arm to the correct cylinder
      → depends on how fast disk arm can move (increasing very slowly)
    - **Rotation**: waiting for the sector to rotate under head
      → depends on rotation rate of disk (increasing, but slowly)
    - **Transfer**: transferring data from surface into disk controller, sending it back to the host.
      → depends on density of bytes on disk (increasing, and very quickly)
  - Disk scheduling:
    - Because seeks are so expensive, the OS tries to schedule disk requests that are queued waiting for the disk.
FCFS

- FCFS (= do nothing)
  - Reasonable when load is low.
  - Long waiting times for long request queues.

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
**SSTF**

- **Shortest seek time first**
  - Minimizes arm movement (seek time)
  - Maximizes request rate
  - Unfairly favors middle blocks
  - May cause starvation of some requests

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
SCAN

- **Elevator algorithm**
  - Service requests in one direction until done, then reverse
  - Skews wait times non-uniformly

```
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
```

![Diagram of elevator algorithm with queue positions]
C-SCAN

- **Circular SCAN**
  - Like SCAN, but only go in one direction (e.g. typewriters)
  - Uniform wait times

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
LOOK / C-LOOK

• Similar to SCAN/C-SCAN, but the arm goes only as far as the final request in each direction.

queue: 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
Modern Disks

- Intelligent controllers
  - A small CPU + many kilobytes of memory.
  - They run a program written by the controller manufacturer to process I/O requests from the CPU and satisfy them.
  - Intelligent features:
    - Read-ahead: the current track
    - Caching: frequently-used blocks
    - Command queueing
    - Request reordering: for seek and/or rotational optimality
    - Request retry on hardware failure
    - Bad block/track identification
    - Bad block/track remapping: onto spare blocks and/or tracks
I/O Schedulers

- I/O scheduler’s job
  - Improve overall disk throughput
    - Merging requests to reduce the number of requests
    - Reordering and sorting requests to reduce disk seek time
  - Prevent starvation
    - Submit requests before deadline
    - Avoid read starvation by write
  - Provide fairness among different processes
  - Guarantee quality-of-service (QoS) requirement
NAND Flash-based Storage

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- NAND flash memory
- Flash Translation Layer (FTL)
- OS implications
Flash Memory Characteristics

- **Flash memory**
  - Non-volatile, Updateable, High-density
  - Low cost, Low power consumption, High reliability

- **Erase-before-write**
  - Read
  - Write or Program: $1 \rightarrow 0$
  - Erase: $0 \rightarrow 1$

- **Read faster than write/erase**

- **Bulk erase**
  - Erase unit: block
  - Program unit: byte or word (NOR), page (NAND)
NAND Flash Types (1)

- **SLC NAND Flash**
  - Small block (≤ 1Gb)
  - Large block (≥ 1Gb)

- **MLC NAND Flash**

- **TLC NAND Flash**

Source: Micron Technology, Inc.
NAND Applications

- Universal Flash Drives (UFDs)
- Flash cards
  - CompactFlash, MMC, SD, Memory stick, ...
- Embedded devices
  - Cell phones, MP3 players, PMPs, PDAs, Digital TVs, Set-top boxes, Car navigators, ...
- Hybrid HDDs
- Intel Turbo Memory
- SSDs (Solid-State Disks)
SSDs (1)

- HDDs vs. SSDs

2.5” HDD       Flash SSD  
(101x70x9.3mm)  

1.8” HDD       Flash SSD  
(78.5x54x4.15mm)
## SSDs

<table>
<thead>
<tr>
<th>Feature</th>
<th>SSD (Samsung)</th>
<th>HDD (Seagate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Samsung SSD 840 EVO</td>
<td>ST3000DM001 (Barracuda 7200.14)</td>
</tr>
<tr>
<td>Capacity</td>
<td>1TB (19nm 128GB TLC x 64, 3 cores)</td>
<td>3TB (3 Disc, 6 Heads, 7200 RPM)</td>
</tr>
<tr>
<td>Form factor</td>
<td>2.5” Weight: 53g</td>
<td>3.5” Weight: 626g</td>
</tr>
<tr>
<td>Host interface</td>
<td>Serial ATA-3 (6.0 Gbps) Host transfer rate: 600MB</td>
<td>Serial ATA-3 (6.0 Gbps) Host transfer rate: 600MB</td>
</tr>
<tr>
<td>Power consumption</td>
<td>Active: 0.1W Idle/Standby/Sleep: 0.045W</td>
<td>Active: 8.0W Idle: 5.8W, Standby/Sleep: 0.75W</td>
</tr>
<tr>
<td>Performance</td>
<td>Sequential read: Up to 540 MB/s</td>
<td>Power-on to ready: 15 sec</td>
</tr>
<tr>
<td></td>
<td>Sequential write: Up to 525 MB/s</td>
<td>Average latency: 4.16 msec</td>
</tr>
<tr>
<td></td>
<td>Average latency: 4.16 msec</td>
<td></td>
</tr>
<tr>
<td>Measured performance(^1)</td>
<td>Sequential read: 428.7 MB/s</td>
<td>Sequential read: 145.7 MB/s</td>
</tr>
<tr>
<td>(On MacBook Pro, 256KB for sequential, 4KB for random)</td>
<td>Sequential write: 374.0 MB/s</td>
<td>Sequential write: 138.8 MB/s</td>
</tr>
<tr>
<td></td>
<td>Random read: 96.0 MB/s</td>
<td>Random read: ? MB/s</td>
</tr>
<tr>
<td></td>
<td>Random write: 366.3 MB/s</td>
<td>Random write: 1.09 MB/s</td>
</tr>
<tr>
<td>Price(^2)</td>
<td>585,990 won (586 won/GB)</td>
<td>118,280 won (39 won/GB)</td>
</tr>
</tbody>
</table>

\(^1\) Source: [http://www.anandtech.com](http://www.anandtech.com)  
\(^2\) Source: [http://www.enuri.com](http://www.enuri.com) (As of Nov. 23, 2014)
NAND Constraints (1)

- **No in-place update**
  - Require sector remapping (or address translation)

- **Bit errors**
  - Require the use of error correction codes (ECC)

- **Bad blocks**
  - Factory-marked & run-time bad blocks
  - Require bad block remapping

- **Limited program/erase cycles**
  - < 100K for SLCs
  - < 3K for MLCs
  - Require wear-leveling
What is FTL?

- A software layer to make NAND flash fully emulate traditional block devices (e.g., disks).

Source: Zeen Info. Tech.
FTL (2)

- SSDs internals

Source: Indilinx
OS Implications (1)

- NAND flash has different characteristics compared to disks
  - No seek time
  - Asymmetric read/write access times
  - No in-place-update
  - Good sequential read/sequential write/random read performance, but bad random write performance
  - Wear-leveling
  - ...
  - Traditional operating systems have been optimized for disks. What should be changed?
OS Implications (2)

- **SSD support in OS**
  - Turn off “defragmentation” for SSDs
  - New “TRIM” command
    - Remove-on-delete
  - Align file system partition with SSD layout
  - Flash-aware file systems (e.g., F2FS in Linux)
  - Larger block size (4KB)
File Systems Overview

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- File system basics
- Directory structure
- File system mounting
- File sharing
- Protection
Basic Concepts

- **Requirements for long-term information storage**
  - Store a very large amount of information
  - Survive the termination of the process using it
  - Access the information concurrently by multiple processes

- **File system**
  - Implement an abstraction for secondary storage (files)
  - Organizes files logically (directories)
  - Permit sharing of data between processes, people, and machines.
  - Protect data from unwanted access (security)
Files

- **File**
  - A named collection of related information that is recorded on secondary storage.
    - persistent through power failures and system reboots
  - OS provides a uniform logical view of information storage via files.

- **File structures**
  - Flat: byte sequence
  - Structured:
    - Lines
    - Fixed length records
    - Variable length records
Storage: A Logical View

- Abstraction given by block device drivers:

<table>
<thead>
<tr>
<th>512B</th>
<th>512B</th>
<th>512B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

- Operations
  - Identify(): returns N
  - Read(start sector #, # of sectors)
  - Write(start sector #, # of sectors)

Source: Sang Lyul Min (Seoul National Univ.)
For each file, we have

- File contents (data)
  - File systems normally do not care what they are
- File attributes (metadata)
  - File size
  - Owner, access control lists
  - Creation time, last access time, last modification time, ...
- File name

File access begins with...

- File name
  - open ("/etc/passwd", O_RDONLY);
File System Basics (2)

- File system: A mapping problem
  - \(<\text{filename, data, metadata}> \rightarrow \langle\text{a set of blocks}\rangle\)
File System Basics (3)

- **Goals**
  - Performance + Reliability

- **Design issues**
  - What information should be kept in metadata?
  - How to locate metadata?
    - Mapping from pathname to metadata
  - How to locate data blocks?
  - How to manage metadata and data blocks?
    - Allocation, reclamation, free space management, etc.
  - How to recover the file system after a crash?
  - ...
### File Attributes

#### Attributes or metadata

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Who can access the file and in what way</td>
</tr>
<tr>
<td>Password</td>
<td>Password needed to access the file</td>
</tr>
<tr>
<td>Creator</td>
<td>ID of the person who created the file</td>
</tr>
<tr>
<td>Owner</td>
<td>Current owner</td>
</tr>
<tr>
<td>Read-only flag</td>
<td>0 for read/write; 1 for read only</td>
</tr>
<tr>
<td>Hidden flag</td>
<td>0 for normal; 1 for do not display in listings</td>
</tr>
<tr>
<td>System flag</td>
<td>0 for normal files; 1 for system file</td>
</tr>
<tr>
<td>Archive flag</td>
<td>0 for has been backed up; 1 for needs to be backed up</td>
</tr>
<tr>
<td>ASCII/binary flag</td>
<td>0 for ASCII file; 1 for binary file</td>
</tr>
<tr>
<td>Random access flag</td>
<td>0 for sequential access only; 1 for random access</td>
</tr>
<tr>
<td>Temporary flag</td>
<td>0 for normal; 1 for delete file on process exit</td>
</tr>
<tr>
<td>Lock flags</td>
<td>0 for unlocked; nonzero for locked</td>
</tr>
<tr>
<td>Record length</td>
<td>Number of bytes in a record</td>
</tr>
<tr>
<td>Key position</td>
<td>Offset of the key within each record</td>
</tr>
<tr>
<td>Key length</td>
<td>Number of bytes in the key field</td>
</tr>
<tr>
<td>Creation time</td>
<td>Date and time the file was created</td>
</tr>
<tr>
<td>Time of last access</td>
<td>Date and time the file was last accessed</td>
</tr>
<tr>
<td>Time of last change</td>
<td>Date and time the file has last changed</td>
</tr>
<tr>
<td>Current size</td>
<td>Number of bytes in the file</td>
</tr>
<tr>
<td>Maximum size</td>
<td>Number of bytes the file may grow to</td>
</tr>
</tbody>
</table>
File Operations

- Unix operations

```c
int creat(const char *pathname, mode_t mode);
int open(const char *pathname, int flags, mode_t mode);
nint close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int stat(const char *pathname, struct stat *buf);
int chmod(const char *pathname, mode_t mode);
int chown(const char *pathname, uid_t owner, gid_t grp);
int flock(int fd, int operation);
int fcntl(int fd, int cmd, long arg);
```
Directories

- **Directories**
  - For users, provide a structured way to organize files
  - For the file system, provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk

- **A hierarchical directory system**
  - Most file systems support multi-level directories
  - Most file systems support the notion of a current directory (or working directory)
    - Relative names specified with respect to current directory
    - Absolute names start from the root of directory tree
A directory is ...

- Typically just a file that happens to contain special metadata
  - Only need to manage one kind of secondary storage unit.
- Directory = list of (file name, file attributes)
- Attributes include such things as:
  - size, protection, creation time, access time,
  - location on disk, etc.
- Usually unordered (effectively random)
  - Entries usually sorted by program that reads directory.
Pathname Translation

- open("/a/b/c", ...)
  - Open directory "/" (well known, can always find)
  - Search the directory for "a", get location of "a"
  - Open directory "a", search for "b", get location of "b"
  - Open directory "b", search for "c", get location of "c"
  - Open file "c"
  - (Of course, permissions are checked at each step)

- System spends a lot of time walking down directory paths
  - This is why open is separate from read/write.
  - OS will cache prefix lookups to enhance performance.
    - /a/b, /a/bb, /a/bbb, etc. all share the "/a" prefix
Directory Operations

- Unix operations
  - Directories implemented in files.
    - Use file operations to manipulate directories
  - C runtime libraries provides a higher-level abstraction for reading directories.
    - DIR *opendir(const char *name);
    - struct dirent *readdir(DIR *dir);
    - void seekdir(DIR *dir, off_t offset);
    - int closedir(DIR *dir);
  - Other directory-related system calls.
    - int rename(const char *oldpath, const char *newpath);
    - int link(const char *oldpath, const char *newpath);
    - int unlink(const char *pathname);
Mounting

- A file system must be mounted before it can be available to processes on the system
  - Windows: to drive letters (e.g., C:𝑊, D:𝑊, …)
  - Unix: to an existing empty directory
    (= mount point)
File Sharing

- File sharing
  - File sharing has been around since timesharing.
  - File sharing is incredibly important for getting work done.
    - Basis for communication and synchronization.
    - On distributed systems, files may be shared across a network (e.g., NFS).
  - Three key issues when sharing files
    - Semantics of concurrent access:
      - What happens when one process reads while another writes?
      - What happens when two processes open a file for writing?
    - Concurrency control using locks
    - Protection
Consistency Semantics

- **UNIX semantics**
  - Writes to an open file are visible immediately to other users that have this file open at the same time.
  - One mode of sharing allows users to share the pointer of current location into the file.
    - via fork() or dup().

- **AFS session semantics**
  - Writes to an open file are not visible immediately.
  - Once a file is closed, the changes made to it are visible only in sessions starting later.

- **Immutable-shared-files semantics**
  - Once a file is declared as shared by its creator, it cannot be modified.
**Protection**

- **Representing protection**
  - Access control lists (ACLs)
    - For each object, keep list of subjects and their allowed actions.
  - Capabilities
    - For each subject, keep list of objects and their allowed actions.

<table>
<thead>
<tr>
<th>subjects</th>
<th>objects</th>
<th>ACL</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>/etc/passwd</td>
<td>rw</td>
</tr>
<tr>
<td></td>
<td>/home/jinkyu</td>
<td>rw</td>
</tr>
<tr>
<td></td>
<td>/home/guest</td>
<td>rw</td>
</tr>
<tr>
<td>jinkyu</td>
<td></td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>/home/jinkyu</td>
<td>rw</td>
</tr>
<tr>
<td></td>
<td>/home/guest</td>
<td>r</td>
</tr>
<tr>
<td>guest</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>/home/guest</td>
<td>r</td>
</tr>
</tbody>
</table>
File System Internals

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topics

- **File system implementation**
  - File descriptor table, File table
  - Virtual file system

- **File system design issues**
  - Directory implementation: filename $\rightarrow$ metadata
  - Allocation: metadata $\rightarrow$ a set of data blocks
  - Reliability issues
  - Performance issues
Overview

- **User’s view on file systems:**
  - How files are named?
  - What operations are allowed on them?
  - What the directory tree looks like?

- **Implementor’s view on file systems:**
  - How files and directories are stored?
  - How disk space is managed?
  - How to make everything work efficiently and reliably?
Disk Layout

- Master Boot Record
  - Boot code
  - Partition table

- Partition 1 (active)

- Partition 2

- Partition 3

- Boot block:
  - fs metadata (type, # blocks, etc.)

- Super block:
  - Data structures for free space mgmt.

- Bitmaps:

- i-nodes:
  - File metadata

- Root dir:
  - Files & directories
File System Internals

System call interface

Virtual File System (VFS)

minix  nfs  ext2  dosfs  mmfs  procfs

buffer cache

device driver

File System
**In-memory Structures**

- **file table (system-wide open-file table)**
  - count
  - offset
  - file attributes

- **in-memory partition table**

- **directory cache**

- **buffer cache**

**process A**

**per-process file descriptor table (per-process open-file table)**

**process B**
• Virtual File System

- Manages kernel-level file abstractions in one format for all file systems.
- Receives system call requests from user-level (e.g., open, write, stat, etc.)
- Interacts with a specific file system based on mount point traversal.
- Receives requests from other parts of the kernel, mostly from memory management.
- Translates file descriptors to VFS data structures (such as vnode).
Linux: VFS common file model

- The superblock object
  - stores information concerning a mounted file system.
- The inode object
  - stores general information about a specific file.
- The file object
  - stores information about the interaction between an open file and a process.
- The dentry object
  - stores information about the linking of a directory entry with the corresponding file.

In order to stick to the VFS common file model, in-kernel structures may be constructed on the fly.
Directory Implementation (1)

- **Directory structure**
  - Table (fixed length entries)
  - Linear list
    - Simple to program, but time-consuming.
    - Requires a linear search to find an entry.
    - Entries may be sorted to decrease the average search time and to produce a sorted directory listing easily (e.g., using B-tree).
  - Hash table
    - Decreases the directory search time.
    - A hash table is generally fixed size and the hash function depends on that size. (need mechanisms for collisions)
    - The number of files can be large:
      1. enlarge the hash table and remap.
      2. use a chained-overflow hash table.
The location of metadata

- In the directory entry
- In the separate data structure (e.g., i-node)
- A hybrid approach
Directory Implementation (3)

- Supporting long file names

![Diagram showing directory entries and file attributes.](image-url)
Allocation (1)

- Contiguous allocation
  - A file occupies a set of contiguous blocks on the disk.
  - Used by IBM VM/CMS
Contiguous allocation (cont’d)

- Advantages
  - The number of disk seeks is minimal.
  - Directory entries can be simple:
    <file name, starting disk block, length, etc.>

- Disadvantages
  - Requires a dynamic storage allocation: First / best fit.
  - External fragmentation: may require a compaction.
  - The file size is hard to predict and varying over time.

- Feasible and widely used for CD-ROMS
  - All the file sizes are known in advance.
  - Files will never change during subsequent use.
**Allocation (3)**

- **Modified contiguous allocation**
  - A contiguous chunk of space is allocated initially.
    - When the amount is not large enough, another chunk of a contiguous space (an *extent*) is added.
  - **Advantages**
    - Still the directory entry can be simple.
      - `<name, starting disk block, length, link to the extent>`
  - **Disadvantages**
    - Internal fragmentation: if the extents are too large.
    - External fragmentation: if we allow varying-sized extents.
  - **Used by Veritas File System (VxFS).**
Allocation (4)

- **Linked allocation**
  - Each file is a linked list of disk blocks.
Linked allocation (cont’d)

• Advantages
  – Directory entries are simple:
    <file name, starting block, ending block, etc.>
  – No external fragmentation: the disk blocks may be scattered anywhere on the disk.
  – A file can continue to grow as long as free blocks are available.

• Disadvantages
  – It can be used only for sequentially accessed files.
  – Space overhead for maintaining pointers to the next disk block.
  – The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes.
  – Fragile: a pointer can be lost or damaged.
Linked allocation using a FAT

- A section of disk at the beginning of each partition is set aside to contain a file allocation table (FAT).
- FAT should be cached to minimize disk seeks.
  - Space overhead can be substantial.
- Random access time is improved.
- Used by MS-DOS, OS/2
  - cf. FAT-16: 2GB limitation with 32KB block size
Indexed allocation

- Bring all the pointers together into one location (index block or i-node)
- Each file has its own index block.
Indexed allocation (cont’d)

• Advantages
  – Supports direct access, without suffering from external fragmentation.
  – I-node need only be in memory when the corresponding file is open.

• Disadvantages
  – Space overhead for indexes:
    (1) Linked scheme: link several index blocks
    (2) Multilevel index blocks
    (3) Combined scheme: UNIX
      - 12 direct blocks, single indirect block,
        double indirect block, triple indirect block
Free Space Management (1)

- **Bitmap or bit vector**
  - Each block is represented by 1 bit.
    - 1 = free, 0 = allocated
  - Simple and efficient in finding the first free block.
    - May be accelerated by CPU’s bit-manipulation instructions.
  - Inefficient unless the entire vector is kept in main memory.
    - Clustering reduces the size of bitmaps.
Free Space Management (2)

- **Linked list**
  - Link together all the free disk blocks, keeping a pointer to the first free blocks.
  - To traverse the list, we must read each block, but it’s not a frequent action.
  - The FAT method incorporates free-block accounting into the allocation data structure.
Reliability (1)

- **File system consistency**
  - File system can be left in an inconsistent state if cached blocks are not written out due to the system crash.
  - It is especially critical if some of those blocks are i-node blocks, directory blocks, or blocks containing the free list.
  - Most systems have a utility program that checks file system consistency
    - Windows: scandisk
    - UNIX: fsck
Reliability (2)

- **fsck: checking blocks**
  - Reads all the i-nodes and mark used blocks.
  - Examines the free list and mark free blocks.

<table>
<thead>
<tr>
<th>Consistent</th>
<th>Missing block -- add it to the free list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks in use</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>Free blocks</td>
<td>0 0 1 0 1 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duplicated free block -- rebuild the free list</th>
<th>Duplicated data block -- allocate a new block and copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks in use</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>Free blocks</td>
<td>0 0 1 0 2 0 0 0</td>
</tr>
</tbody>
</table>
Reliability (3)

- **fsck: checking directories**
  - Recursively descends the tree from the root directory, counting the number of links for each file.
  - Compare these numbers with the link counts stored in the i-nodes.
  - Force the link count in the i-node to the actual number of directory entries.

<table>
<thead>
<tr>
<th>i-node</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- i-node #5
  - count=3

- i-node #12
  - count=2
### Journaling file systems

- Fsck’ing takes a long time, which makes the file system restart slow in the event of system crash.
- Record a log, or journal, of changes made to files and directories to a separate location. (preferably a separate disk).
- If a crash occurs, the journal can be used to undo any partially completed tasks that would leave the file system in an inconsistent state.
- IBM JFS for AIX, Linux
  - Veritas VxFS for Solaris, HP-UX, Unixware, etc.
  - SGI XFS for IRIX, Linux
  - Reiserfs, ext3 for Linux
**Performance (1)**

- **Buffer cache**
  - Applications exhibit significant locality for reading and writing files.
  - Idea: cache file blocks in memory to capture locality in buffer cache (or buffer cache).
    - Cache is system wide, used and shared by all processes.
    - Reading from the cache makes a disk perform like memory.
    - Even a 4MB cache can be very effective.
  - Issues
    - The buffer cache competes with VM.
    - Like VM, it has limited size.
    - Need replacement algorithms again.
      (References are relatively infrequent, so it is feasible to keep all the blocks in exact LRU order)
Read ahead

- File system predicts that the process will request next block.
  - File system goes ahead and requests it from the disk.
  - This can happen while the process is computing on previous block, overlapping I/O with execution.
  - When the process requests block, it will be in cache.
- Compliments the disk cache, which also is doing read ahead.
- Very effective for sequentially accessed files.
- File systems try to prevent blocks from being scattered across the disk during allocation or by restructuring periodically.
### UNIX FS (1)

#### On-disk layout

<table>
<thead>
<tr>
<th>Boot Block</th>
<th>Super Block</th>
<th>i-node List</th>
<th>Data Blocks</th>
</tr>
</thead>
</table>

- **Boot block**: stores boot code
- **Super block**:
  - Basic info. of the file system
  - Head of freelists of i-nodes and data blocks
- **i-node list**
  - Referenced by index into the i-node list
  - All i-nodes are the same size
- **Data blocks**
  - A data block belongs to only one file
UNIX FS (2)

- **i-node**
  - File metadata
  - Pointers for data blocks belonging to the file
UNIX FS (3)

- Directory
  - Directory is a special file
  - Array of <file name, i-node number>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>100</td>
</tr>
<tr>
<td>..</td>
<td>24</td>
</tr>
<tr>
<td>foo</td>
<td>212</td>
</tr>
<tr>
<td>bar</td>
<td>37</td>
</tr>
</tbody>
</table>

- The i-node number of the root directory is fixed or specified in the superblock
- Different names can point to the same i-node number (“hard link”)
  - Each i-node keeps track of the link count