ECE 7650 Scalable and Secure Internet Services and Architecture
---- A Systems Perspective

Part I: Operating system overview:

Disk and File System
# What Disks Look Like

## Specifications

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Parallel-ATA</th>
<th>Serial-ATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>PATA-133</td>
<td>SATA 3.0Gb/s</td>
</tr>
<tr>
<td>Capacity (GB)</td>
<td>500 / 400 / 320 / 250</td>
<td>←</td>
</tr>
<tr>
<td>Data heads (physical)</td>
<td>6 / 6 / 4 / 4</td>
<td>←</td>
</tr>
<tr>
<td>Data disks</td>
<td>3 / 3 / 2 / 2</td>
<td>←</td>
</tr>
</tbody>
</table>

## Performance

<table>
<thead>
<tr>
<th></th>
<th>Parallel-ATA</th>
<th>Serial-ATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data buffer</td>
<td>8 MB</td>
<td>16 MB / 8 MB</td>
</tr>
<tr>
<td>Rotational speed (rpm)</td>
<td>7,200</td>
<td>←</td>
</tr>
<tr>
<td>Media transfer rate (max. Mbits/sec)</td>
<td>998</td>
<td>←</td>
</tr>
<tr>
<td>Interface transfer rate (max. MB/sec)</td>
<td>133</td>
<td>300</td>
</tr>
<tr>
<td>Average seek time (ms) (read, typical)</td>
<td>8.5</td>
<td>←</td>
</tr>
</tbody>
</table>

## Reliability

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Error rate (non-recoverable)</td>
<td>1 in 10E14</td>
<td>←</td>
</tr>
<tr>
<td>Start/stops (at 40°C)</td>
<td>50,000</td>
<td>←</td>
</tr>
<tr>
<td>Availability</td>
<td>24/7</td>
<td>←</td>
</tr>
</tbody>
</table>

--

Hitachi Deskstar T7K500 SATA
How Disks Work

Flash Animation

See http://cis.poly.edu/cs2214rvs/disk.swf
More about disks

• Typical disk parameters:
  - 2-30 heads (2 per platter)
  - Diameter: 1.8-5.25in
  - Capacity: 20MB-500GB
  - Sector size: 64 bytes to 8K bytes (most PC disks: 512 byte sectors)
  - 700-20480 tracks per surface
  - 16-1600 sectors per track
  - 8-32MB built-in cache for disk caching/prfething

• Drive attached to computer via I/O bus
  - Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI
  - Host controller in computer uses bus to talk to disk controller built into drive or storage array
What’s important about disks from OS perspective

- Disks are big & slow - compared to RAM

- Access to disk requires
  - Seek (move arm to track) – to cross all tracks anywhere from 20-50ms, on average takes 1/3 of that.
  - Rotational delay (wait for sector to appear under track) 7,200rpm is 8.3ms per rotation, on average takes ½ of that (4.15ms).
  - Transfer time (fast: 512 bytes at 998 Mbit/s is about 3.91us)

- Seek and rotation delay dominates

- Random Access is expensive, which is hard to be improved.

- Possible solutions:
  - avoid seeks
  - seek to short distances
  - amortize seeks by doing bulk transfers
  - leverage buffer cache to reduce misses on random blocks (see DULO paper)

- Access time is time to get first byte of a request. Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

- OS must improve both access time and disk bandwidth.
Technology Trends: Unbalanced system improvements

Latencies of Cache, DRAM and Disk in CPU Cycles

The disks in 2000 are more than 57 times “SLOWER” than their ancestors in 1980.
Technology Trends: Unbalanced system improvements
Disk Scheduling

- Disk scheduler is to merge pending requests and determine their order and timing to be sent to disk for service.
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a queue of requests for blocks on the following disk cylinders (the cylinders are in the range 0-199)

\[98, 183, 37, 122, 14, 124, 65, 67\]

Disk head initially is at cylinder 53.
Illustration shows total head movement of 640 cylinders.
SSTF

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.
**SCAN**

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Can be implemented using a dispatch queue sorted sector by sector.
- Illustration shows total head movement of 236 cylinders.
C-SCAN

• Provides a more uniform wait time than SCAN.

• The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

• Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

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

• Version of C-SCAN

• Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

queue 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
Anticipatory Scheduling

- Key idea: sometimes wait for process whose request was last serviced,
  - overcomes deceptive idleness
- Keeping disk idle for short intervals with informed decisions could
  - Improves throughput
  - Achieves desired proportions
Solid State Drive (SSD)

- Solid state drives are based on NAND flash memory
  - No moving parts;
  - Performance characteristics driven by electronics and physics – more like RAM than spinning disk
  - Relative technological newcomer, so costs are still quite high in comparison to hard drives, but dropping fast
SSD Performance: Reads

- Unit of read is a page, typically 4KB large
- Today’s SSD can typically handle 10,000 – 100,000 reads/s
  - 0.01 – 0.1 ms read latency (50-1000x better than disk seeks)
  - 200-400 MB/s read throughput (1-3x better than disk seq. throughput)
SSD Performance: Writes

• Writes
  – flash media must be erased before it can be written to
  – unit of erase is a block, typically 64-256 pages long
    • usually takes 1-2ms to erase a block
    • blocks can only be erased a certain number of times before they become unusable – typically 10,000 – 1,000,000 times
  – unit of write is a page
    • writing a page can be 2-10x slower than reading a page

• Writing to an SSD is complicated
  – random write to existing block: read block, erase block, write back modified block
    • leads to hard-drive like performance (300 random writes / s)
  – sequential writes to erased blocks: fast!
    • SSD-read like performance (100-200 MB/s)
I/O System in Linux
Various Data Unit Sizes

- Sector is the transfer unit used by hardware block devices;
  - Writing a single byte requires read-modify-write

- Virtual Filesystem and the mapping layer use the block as minimal disk storage unit inside a filesystem

- Each segment is a memory page—or a portion of a memory page—including chunks of data that are physically adjacent on disk.

- The disk caches work on "pages" of disk data, each of which fits in a page frame
Disk Caching – Buffer Cache

• How much memory should be dedicated for it?
  ➢ In some systems, set aside a portion of physical memory for buffer cache, and use the rest for virtual memory pages (such as page cache);
    – search for victim buffer/page from the same pool
    — Windows allows user to limit buffer cache size
  
  — Disadvantage: cannot use idle memory of other pool, bad prediction of buffer caches accesses can result in poor VM performance

  ➢ In newer systems, integrated into virtual memory system (e.g., page cache) in Linux

• How should concurrent access be mediated (multiple processes may be attempting to write/read to the same sector)?
  ➢ Consistency is guaranteed because all I/O operations must go through buffer cache!)
Buffer Cache Replacement

- Similar to VM Page Replacement
  - But theoretically it can do exact LRU because IO sys-calls are used (e.g. in Linux `mark_page_accessed()` is ultimately called!)
  - In Linux, LRU approximation is used due to its unified disk cache where the same cache contains both memory-mapped pages and ordinary file system I/O

- Linux two-queue replacement
  - Active LRU list tends to include the pages that have been accessed recently,
  - Inactive LRU list tends to include the pages that have not been accessed for some time.
    - In the flag of a page frame descriptor (`page->flags`), there are `PG_lru` and `PG_active`.
  - Clearly, pages should be stolen from the inactive list.
  - Functions that operate on the lists: `add_page_to_active_list()` `add_page_to_inactive_list()`
    `del_page_from_active_list()` `del_page_from_inactive_list()` `del_page_from_lru()` `activate_page()`
    `lru_cache_add()` `lru_cache_add_active()`
Linux Replacement

- Linux uses *PG_referenced* bit in the *page->flags* to derive ‘recency’ for its replacement decision;
- Linux scans the two lists so that:
  - If not accessed for a relatively long period of time, a page in the active list can be scanned and demoted into the inactive list, and a page in the inactive list can be scanned and replaced out of memory.

How aggressively Linux scans the active list depends on some heuristic values, such as how difficult to find a victim page.

Linux has preference in replacement of pages, e.g., it is reluctant to replace pages that are mapped to User Mode address spaces.
Buffer Cache Writeback Strategies

- Write-Through:
  - Good for NVRAM disk or USB stick
  - Poor performance for hard disk – every write causes disk access

- (Delayed) Write-Back:
  - Makes individual writes faster – just copy & set bit
  - Absorbs multiple writes
  - Allows write-back in batches

- Problem: what if system crashes before you’ve written data back?
  - Trade-off: performance in no-fault case vs. damage control in fault case
  - If crash occurs, order of write-back can matter

- Current practices:
  - Must write-back on eviction (naturally)
  - Periodically (every 30 seconds or so)
  - When user demands:
    - fsync(2) writes back all modified data belonging to one file, sync(1) writes back entire cache
  - Some systems guarantee write-back on file close
## Files vs Disks

<table>
<thead>
<tr>
<th>Disk Abstraction</th>
<th>File Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block oriented</td>
<td>Byte oriented</td>
</tr>
<tr>
<td>Block numbers</td>
<td>Names</td>
</tr>
<tr>
<td>No protection</td>
<td>Access protection</td>
</tr>
<tr>
<td>No guarantees beyond block write</td>
<td>Consistency guarantees</td>
</tr>
</tbody>
</table>
Overview

File Operations:
create(), unlink(), open(),
read(), write(), close()

File System

- Uses names to specify files
- Views a file as sequence of bytes

Must implement translation
(file name, file offset) →
(disk id, block, offset in a block)

Must manage free space on disk

Buffer Cache

Uses disk id + block LBN

Device Driver
Filesystem Requirements

• Naming
  ➢ Should be flexible, e.g., allow multiple names for the same file
  ➢ Support hierarchy for easy of use

• Persistence
  ➢ Want to be sure that data has been written to disk in case crash occurs

• Sharing/Protection
  ➢ Want to share files with other users
  ➢ Want to restrict who has access to files

• Speed and efficiency for different access patterns
  ➢ Sequential access
  ➢ Random access

• Minimum Space Overhead
  ➢ Disk space for user data is reduced for storing metadata

• Twist: all metadata that is required to do translation must be stored on disk
  ➢ Translation scheme should minimize number of additional accesses for a given access pattern
  ➢ This is different from page tables, where are not subject to paging!
Common Filesystem Model

```c
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
    O_WRONLY|O_CREAT|O_TRUNC, 0600);

do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);

close(outf);
close(inf);
```
The Big Picture

Data structures to keep track of *open* files
- struct file
- struct dentry
- struct inode

Open file table

Cached data and metadata in buffer cache

File Data
- Directory Data
- File Descriptors (inodes)
- Filesystem Information

On-Disk Data Structures

Per-process file descriptor table

PCB

... 5 4 3 2 1 0
File Open/Access

(a) Open (file name)
- User space
- Kernel memory
- Secondary storage
- Directory structure
- File-control block

(b) Read (index)
- User space
- Kernel memory
- Secondary storage
- Index
- Per-process open-file table
- System-wide open-file table
- Data blocks
- File-control block
**In-memory Data Structure for File Access**

- **file** – represents a file opened by a process
  - With separate offsets for byte-stream
- **dentry** – an in-memory object representing directory entry
- **inode** – represents a distinct file
  - at most 1 in-memory instance per distinct file
  - number of openers & other properties
  - Used to find the disk addresses of blocks in the file

Generally:
- Reflects how processes are currently using files
- Lifetime of objects determined by open/close
  - Reference counting is used
Filesystem Information

• “superblock” stores information such as size of entire filesystem, etc.
  ➢ Location of file descriptor table & free map

• Free Block Map
  ➢ Bitmap used to find free blocks
  ➢ Typically cached in memory

• Superblock & free map often replicated in different positions on disk

Free Block Map
010001111010101010101
Super Block
File Allocation Methods

• An allocation method refers to how disk blocks are allocated for files:

  ➢ Contiguous allocation

  ➢ Linked allocation

  ➢ Indexed allocation
**Contiguous Allocation**

- Each file occupies a set of contiguous blocks on the disk
  - Simple – only starting location (block #) and length (number of blocks) are required.
  - Wasteful of space (dynamic storage-allocation problem)
  - Files cannot grow

![Diagram showing contiguous allocation with a directory listing the files and their locations.](image)
Linked Allocation

• Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
  ➢ Simple – need only starting address
  ➢ Free-space management system – no waste of space
  ➢ Mapping
File-Allocation Table

FAT stored at beginning of disk & replicated for redundancy

FAT cached in memory

Size: n-bit entries, m-bit blocks → $2^{(m+n)}$ Bytes disk space limit
  - n=12, 16, 28
  - m=9 … 15 (0.5KB-32KB)

As disk size grows, m & n must grow
  - Growth of n means larger in-memory table

<table>
<thead>
<tr>
<th>Filename</th>
<th>Length</th>
<th>First Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>“a”</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>“b”</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>“c”</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>“d”</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

DOS FAT

```
<table>
<thead>
<tr>
<th>FILE_NO</th>
<th>ENTRY_INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>
```
Indexed Allocation

• Brings all pointers together into the *index block.*
  - Need index table
  - Dynamic access without external fragmentation, but have overhead of index block.
Indexed Allocation – Mapping

outer-index

index table

file
Combined Scheme: UNIX (4K bytes per block)
Assume all files are 2KB in size (observed median files is about 2KB)

- Larger blocks: faster reads (because seeks are amortized & more bytes per transfer)
- More wastage (2KB file in 32KB block means 15/16th are unused)

Source: Tanenbaum, Modern Operating Systems
Storing Inodes in FFS

- Used in Unix v7, BSD 4.3, Ext2/3
- Cylindergroups have superblock+bitmap+inode list+file space
- Try to allocate file & inode in same cylinder group to improve access locality

![Diagram of Storing Inodes in FFS](image-url)
Positioning Inodes

• Putting inodes in fixed place makes finding inodes easier
  ➢ Can refer to them simply by inode number
  ➢ After crash, there is no ambiguity as to what are inodes vs. what are regular files
  ➢ Use “df –ih” on Linux to see how many inodes are used/free

• Disadvantage: limits the number of files per filesystem at creation time
Name Resolution

- Need to find file descriptor (inode) through a given file pathname
- Hierarchical approaches:
  - File system forms a tree (or DAG)
- How to tell regular file from directory?
  - Set a bit in the inode
- Data Structures
  - Linear list of (inode, name) pairs
  - B-Trees that map name -> inode
    - Scalable to large number of files: in growth, in lookup time
    - Overhead for small directories

```
   23 | multi-oom   | 15 | sample.txt
```

• Offset 0
Absolute Paths

• How to resolve a path name such as “/dict/w/w7”?
  ➢ Split into tokens using “/” separator
  ➢ Find inode corresponding to root directory
    ➢ Use fixed inode # for root
  ➢(*) Look up “dict” in root directory, find inode
  ➢ If not last component in path, check that inode is a directory. Go to (*), looking for next comp
  ➢ If last component in path, return its inode
Some Issues in Name Resolution

• Must have a way to scan an entire directory without other processes interfering -> need a “lock” function

• Most OS cache translations in “namei” cache – maps absolute pathnames to inode
Example: Linux VFS

- Reality: system must support more than one filesystem at a time
  - Users should not notice a difference unless unavoidable
- Most systems, Linux included, use an object-oriented approach:
  - VFS-Virtual Filesystem
Example: Linux VFS Interface

```c
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
};
```
RAID – Redundant Arrays of Inexpensive Disks

• Idea born around 1988
• Original observation: it’s cheaper to buy multiple, small disks than single large expensive (and high-speed) disk (SLED)
  ➢ SLEDs don’t exist anymore, but multiple disks arranged as a single disk still useful
• Can reduce latency by writing/reading in parallel
• Can increase reliability by exploiting redundancy
• Several arrangements are known, seven of them have “standard numbers”
• Can be implemented in hardware/software
RAID 0

- RAID: **Striping** data across disk
- Advantage: If disk accesses go to different disk, can read/write in parallel, decrease in latency
- Disadvantage: Decreased reliability ($\text{MTTF(Array)} = \frac{\text{MTTF(Disk)}}{\#\text{disks}}$)
RAID 1

- RAID 1: **Mirroring** (all reads/writes go to both disks)
- Advantages:
  - Redundancy, Reliability – have backup of data
  - Better read performance than single disk – why?
  - About same write performance as single disk
- Disadvantage:
  - Inefficient storage use
RAID 4

- RAID 4: Striping + Block-level parity
- Advantage: need only N+1 disks for N-disk capacity & 1 disk redundancy
- Disadvantage: small writes (less than one stripe) may require 2 reads & 2 writes
  - Read old data, read old parity, write new data, compute & write new parity
  - Parity disk can become bottleneck
**RAID 5**

- RAID 5: Striping + Block-level Distributed Parity
- Like RAID 4, but avoids parity disk bottleneck
- Best large read & large write performance
- Only remaining disadvantage is small writes