# Contents of Lecture 6

- I/O Systems
- Hard disks
- File System Interface and Framework

# Buses

- Buses are cheap (compared with e.g. switches) but can limit performance.
- Bus speed is limited by physical length and number of devices (and therefore bus load).
- Traditional bus classification:
  - **CPU-memory buses**: designed to match a particular combination of CPU and memory to optimise memory bandwidth. Designer knows characteristics of all involved devices.
  - **I/O buses**: follow bus standards.
- A hybrid of these two, sometimes called a mezzanine bus [H&P 3ed, p692], can improve I/O performance. An example of such a bus is the typical PC system bus, the **PCI bus** (**Peripheral Component Interconnect**).

# I/O device interface

**Typical registers on a device**

- **Status**: read by the host.
- **Control**: written by the host.
- **Data-out**: written by the host.
- **Data-in**: read by the host.

# Example of Polling

1. The host loops and reads the **Busy** bit in the status register until it becomes false.
2. The host sets the **Write-command** bit in the control register and writes a byte to the data-out register.
3. The host sets the **Command-ready** bit in the control register.
4. The controller notices Command-ready and sets the Busy bit.
5. The controller performs the write operation to the device.
6. The controller clears Command-ready in the control register, and the **Error** and busy bits in the status register

# More about Polling

- Polling is simple to use.
- For slow devices which don't need fast response, it can be used e.g. periodically every 1 ms.
- For most devices, however, interrupts are preferable.

# Interrupts

- An interrupt is a signal from the outside world to the processor (via a bus).
- Before the processor starts executing a new instruction, it checks whether an interrupt has happened, and if so, it empties its pipeline and jumps to the kernel.
- With the interrupt signal, also come an address which is an offset into a table called the **interrupt vector**. The table contains instructions that will be executed when the corresponding interrupt happens.
- Typically a processor has several levels of interrupts and a register which tells it which interrupts the processor currently should ignore.
- Recall: in Solaris an interrupt is serviced by a thread, and this approach nicely achieves priorities among all threads.

# Direct Memory Access (DMA)

- In **Programmed I/O** it is the processor which accesses the memory with data to/from a device. This is usually wasteful use of the processor.
- In **Direct Memory Access** (DMA) it is instead a special controller which talks to the device controller and accesses memory.
- This **DMA controller** is programmed with source address, destination address and amount of data to transfer.
- Assume we read from a disk to memory. The device controller interrupts the DMA controller when it has data and the DMA controller reads e.g. one word of data from the device controller and writes it to memory repeatedly until all data from the current disk access is fetched. Then the DMA controller interrupts the processor.

# More about DMA

- Often device controllers have a DMA controller.
- When the DMA controller uses the bus, it is called **cycle stealing**.
- If the processor also uses the bus a lot then the benefit from DMA is somewhat reduced, but assuming most accesses can be serviced by the cache hierarchy, DMA is very useful.
- More sophisticated DMAs can be programmed with more than a copy instruction and are called **I/O processors** or **channel controllers**. The kernel can download programs into them.

# Layers in file-system implementations

- Consider a Java program which opens a file.
- A number of software layers will be at work:
  - The Java application — portable
  - The JVM — will make the system call to open a file, called `open` on e.g. UNIX
  - The kernel virtual file system interface — will translate file name to inode
  - The file system open routine — specific for e.g. EXT2
  - The disk device driver — specific for a device controller
- The purpose of the device driver is to hide details from the rest of the kernel.

# Magnetic Disks

- Magnetic disks have dominated non-volatile storage since 1965.
- A disk consists of a number, often 1-12, of metal or glass **platters** covered with a magnetic material on both surfaces.
- The platters rotate at a speed between 3600 to 15000 rounds per minute (RPM).
- Each platter is divided into **tracks** and each track into **sectors** (often 512 bytes).
- On each track is recorded a sequence of (1) the sector number, (2) a gap, (3) user data, and (4) error correction code.
- The tracks at the same radius constitute a **cylinder**.

# More about magnetic disks

- There is a **read/write head** over each surface and all heads move in conjunction.
- The disk access latency consists of a **seek time** (time to move the read/write heads), **rotational latency** and a **transfer time**.
- Manufacturers specify minimum, maximum and average seek times with average defined as the sum of all $N$ possible seek times divided $N$. Due to locality, the average seek time seen in practise is much lower (maybe 25% of the specified depending on the OS and the application).
- An example value of minimum, maximum, and average are 2.5 ms, 23.0 ms, and 12.0 ms for an IBM Travelstar disk.
- Transfer times, or actually **transfer rates** can vary e.g. from 3MB/s to 88 MB/s.

# Up-to-date info of an internal disks

- SATA2: Price SEK 1390 including VAT (April 8, 2010).
- SATA3: Price SEK 1290 including VAT (March 29, 2011).
- SATA3: Price SEK 1089 including VAT (April 23, 2012).
- SATA3: Price SEK 769 including VAT (April 22, 2013).
- Capacity 2 TB.
- The SATA2 disk: average seek time 8.9 ms.
- The SATA3 disk: average seek time 4.2 ms
- Rotational speed: 7200 RPM (meaning a rotational latency of 4.2 ms if we must wait half of a rotation).
- Approximate cost 2004: 500 GB at SEK 6000 (external Firewire)

# Disk scheduling

- Typically there are a number of disk access requests and we must find a clever algorithm to reduce the amount of head movement.
- Earlier disks were addressed with cylinder, track, and sector numbers and e.g. BSD UNIX optimised disk accesses using this info.
- Modern disks are instead addressed with a logical block number, and they may renumber blocks to replace a bad block (which does not work) which may make a scheduling algorithm perform worse.
- To reduce seek times the OS anyway must calculate the cylinder number from the logical disk block.

# Disk scheduling algorithms

- **FCFS Scheduling**: First-come, first-served (i.e. FIFO): bad.
- **SSTF Scheduling**: Shortest-seek-time-first: may cause starvation and is not recommended since some request can take a lot of time in an unpredictable way.
- **SCAN Scheduling**: Works like an elevator going up and down and performs disk accesses while moving.
- **C-SCAN Scheduling**: Circular scan: only performs accesses while moving in one direction and then jumps all the way back. This reduces waiting time for requests which "just missed" the read/write heads.
- **LOOK and C-LOOK Scheduling**: The obvious improvement over SCAN to only move the heads as far as needed (i.e. as far as there are requests).

# More about disk scheduling on modern disks

- Since modern disks are addressed as a huge array of logical blocks, it is somewhat difficult for the OS to optimise for reducing rotational latency.
- Modern disk controllers accept multiple requests and can perform its own disk scheduling.
- While this sounds like good news it might not be so: The kernel may prioritise:
  - fetching a VM page over doing application I/O,
  - writing modified VM pages may be more important than reading if we are running out of free pages, and
  - writing a modified inode may be more important than writing file data.
- The kernel may decide to give the disk controller one request at a time for some types of I/O.

# Disk formatting

- A newly produced disk platter just has a magnetic surface with no information on it. **Low-level formatting** or **physical formatting** consists of writing information used by the disk controller.
- The disk controller needs to detect the start of a new sector and therefore a special bit pattern is written before each sector, then comes the sector number, e.g. 512 bytes of sector data, and error correcting code for the sector.
- This low-level formatting is done by the disk manufacturer so that it can discover bad blocks (see next page).
- The next step is to partition the disk using e.g. `pdisk` on UNIX.
- **Logical formatting** means creating a file system e.g. with `mkfs` on a partition.

# Bad blocks

- Most newly produced disks have **bad blocks**, e.g. SCSI disks can have a list of bad blocks which should not be used. These initial bad blocks are not a big problem.
- Typically each cylinder has some extra sectors to replace a bad block with one on the same cylinder. Two schemes:
  - **Sector sparing**: instead of a bad block a spare block is used.
  - **Sector slipping**: instead of a bad block the *next* block is used, i.e. all blocks on the track starting with the bad block move one step to the right (copying starts from the right...).
- A bad block detected during disk use requires manual intervention since a file is damaged and, if possible, must be fetched from backup.

# RAID

***Redundant Array of Independent Disks***

- Pioneered during the 1980s by David Patterson at Berkeley who still [see H&P 3ed] calls it Redundant Array of *Inexpensive* Disks.
- Patterson was also one of the pioneers in **RISC** machines (he coined the term) and with faster processors, they realised industry soon would need faster disks.
- Their argument was as follows: 50 small drives can be faster than one big since the 50 smaller have independent arms.
- Problem to address was **mean time to failure** (MTTF) — see below.
- In 2000, RAID was a $ 27 billion industry with 80% of all non-PC disks in RAIDs.

# Two disks from the late 1980s

- Example big disk: IBM 3380
  - Capacity: 7,5 GB
  - Approximate price: SEK 700000 (from USD 10-18 per MB)
  - Transfer rate: 3 MB/s
  - Rated MTTF: 30000 hours (100000 hours, or 11 years, in practise)
- Example small disk: Conners CP3100
  - Capacity: 0,1 GB
  - Approximate price: SEK 6000 (from USD 7-10 per MB)
  - Transfer rate: 3 MB/s
  - Rated MTTF: 30000 hours (unknown (to me) hours in practise)
- Same transfer rates for both disks!

# Problem with taking 100 inexpensive disks

- Buy 100 small disks at SEK 600000
- Capacity: 10 GB
- Assumption: constant failure rate and independent failures (which is used to calculate MTTF by disk vendors).
- MTTF: $30000/100 = 300$ hours is **bad**!
- The RAID levels proposed by Patterson have MTTFs which exceed the disk's expected lifetime (several years).

# RAID notation

- **D** is the number of disks with data (excluding check disks)
- **G** is the number of disks in a protection group (excluding check disks)
- **C** is the number of check disks in a protection group
- Example parameters: Thinking Machines' Data Vault: $G = 32$, $C = 8$ (this is a Level 2 RAID system)

# Disk access types

- **Supercomputers** (such as the Thinking Machines CM-2) tend to make few large disk accesses
- **Database systems** tend to make many small disk accesses
- As we will see, supporting database systems efficiently is the challenge.

# RAID Level 0

- Level 0 was not proposed by Patterson but usually listed as an option in disk formatting programs.
- Level 0 is **disk striping** which had been in use before RAID.
- The idea is to improve performance by storing a file's data on different disks so that they can fetch data in parallel.
- Logical block $i$ is stored in disk number $i \mod N$, for $N$ disks.
- There is no redundancy at this level.
- Used e.g. in video-editing systems.

# RAID Level 1

- Level 1 is **disk mirroring** and is the traditional way of tolerating disk failure.
- Data is always written to two disks.
- Mirroring is the most expensive RAID variant.
- Mirroring has been in use long before RAID was invented.
- Used e.g. by Tandem (which produced fault-tolerant machine; now part of HP)

***HP is Hewlett-Packard while H&P is* the *Hennessy-Pattersson book***

# RAID Level 2

- Memory-style error-correcting code: e.g. three extra bits per stored byte. With these extra bits, a one-bit error in a byte in memory can be corrected.
- The difference between memory and disks is that memory does not know *which bit* was wrong but a disk controller checks the sector with its checks (regardless of whether RAID is used) and hence knows *which disk* failed (if any).
- Therefore storing the extra bits beyond a single **parity bit** serves no purpose.
- Thinking Machines used to sell arrays with 32 disks data and 7 check disks.
- Although Level 2 is better than Level 1 (in terms of disk usage), there are currently no products using this level according to H&P 3rd ed.

# RAID Level 3

- Level 3 is **bit-interleaved parity** (interleaved within a group) $C = 1$.
- The parity bit is the sum modulo two of all bits in e.g. a byte.
- By storing the parity bit in a separate disk, the data in a faulty disk can be computed from the other disks and the parity disk.
- Performance problems (mostly) for database systems:
  - All disks in a group must participate in an I/O request.
  - Can do at most $D/G$ I/O requests at a time.
  - It takes time to compute the parity bit.

# RAID Level 4

- Level 4 is **block-interleaved parity**.
- A sector of user data is written to one data disk since there is no bit-interleaving with the other disks in the protection group.
- The problem for database systems with Level 3 is that every access went to all disks is solved in Level 4.
- Sectors from $G$ disks still share a sector on the check disk.
- `new parity = old data XOR new data XOR old parity.`
- Writing one sector of data now requires two reads and two writes which can be done two at a time (i.e. parallel reads and parallel writes).

# RAID Level 5 and 6

- The parity disk in Level 4 may become a bottleneck.
- In RAID Level 5, the parity sectors are spread out among the $G + C$ disks to avoid this problem.
- In RAID Level 6, called the **P+Q redundancy scheme** again error-correcting codes are used, now for dealing with multiple-disk failures.

# Overview of the UNIX File System History

- The classic UNIX file system with the design of i-nodes was actually the beginning of UNIX.
- At Berkeley a faster file system implementation which optimized disk seek and rotation times was created.
- A faster file system called the log structured fs (also from Berkeley) is still much faster.
- Log structured, or journaling, file systems make reboots much faster (eg: Linux Ext3)
- Current state-of-the-art file systems are optimized for much larger files.

# UNIX File System Abstractions

- Files — persistent storage of data as a sequence of bytes.
- Directories — a set of mappings from strings to files.
  - a directory is also a file with the mappings as data blocks.
  - a particular file may be listed in different directories.
  - a file may even be listed in one directory using different file names.
- File systems — how the data is organized on disk blocks. May introduce some constraints such as
  - a maximum file name length or
  - file size.

# Example: directory listing

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char** argv)
{
        char*           name;
        DIR*            dir;
        struct dirent*  dirent; // must have d_ino and d_name.

        name = argv[1];
        dir = opendir(name);
        if (dir == NULL)
                error("could not open directory \"%s\"", name);
        while (dirent = readdir(dir))
                printf("%8d %s\n", dirent->d_ino, dirent->d_name);
        closedir(dir);
        return 0;
}
```

# File attributes

- File type
- Hard link count
- File size in bytes
- File owner
- Time stamps
- Permissions
- Info on where on the disk the data blocks are
- Except for last item above the system calls `stat` and `fstat` find this info plus inode number and disk.
- `stat` takes a file name and `fstat` a file descriptor as parameter.

# Open file objects

- When a file is opened, a file descriptor is returned to the process.
- The kernel creates an `open file object` with access-mode (read/write/append) and the current position in the file.
- After reading or writing, the current position is updated.
- The system calls `dup` and `dup2` create another file descriptor for an already opened file which refers to the same open file object
- After `fork`, the parent and child share open file objects.
- A file descriptor can be sent to another process using the `ioctl` system call with the command `I_SENDFD`, and then receieved with `I_RECVFD`.

# File I/O

- In addition to `read` and `write` which take a file descriptor, a buffer, a size UNIX also supports scatter/gather I/O.
- Writing to a file opened with mode `O_APPEND` always writes to the end, useful e.g. when multiple processes write to the same file.
- What happens when multiple processes write to the file without using `O_APPEND`?
- In a multithreaded program, I/O and seek calls may need to be protected.
- **Advisory file locks** are not enforced by the kernel.
- **Mandatory file locks** are enforced by the kernel.

# Vnode/Vfs Architecture

- Early UNIX systems could only mount one file system type.
- An object oriented approach is now used instead.
- A **vnode** (virtual node) represents a file in the kernel.
- A **vfs** (virtual file system) represents a file system in the kernel.
- The vnode struct contains file-system indepependent attributes and two pointers:
  - one pointer to file-system specific inode information
  - one pointer to file-system specific function pointers (shared by each inode of that file system type).
- For example, to close a file the macro `VOP_CLOSE` is used:

```
#define VOP_CLOSE(vp, ...) (*((vp)->v_op->vop_close))(vp, ...)
```