Part III – Persistence

Chapter 36: I/O Devices

- Before persistence, we need to understand how the OS interacts with I/O devices. Consider the general problem:
  - How should I/O be integrated into systems?
  - It's general mechanisms? And, how do we make them efficient?
- See Figure 36.1 for a general depiction of a typical system hierarchical structure. Upshot:
  - Single CPU connected to the main memory via a **memory bus**.
  - High performance I/O devices (graphics card eg.) would then be connected to the system via a **I/O bus***. *[*in modern systems: PCI or Peripheral Component Interconnect]*
  - Lastly, for low performance devices (disks, mice eg.) are connected via the **peripheral bus** (USB, SATA eg)
- Yes, the hierarchical structure is important due to physics and cost. The way systems are designed, it makes sense to put components that do high performance closer to the CPU and low performance devices further away.
- See Figure 36.2, for a better understanding of how make device interaction more efficient. Upshot:
  - Two components:
    - Hardware interface – just as the software needs an interface, so does the hardware, to provide the software the means of controlling its operation
    - Internals structure – specific device abstraction for the system to be able to use it
  - Cannonical (not real/simplified) protocol:
    - 3-registers within the Interface:
      - STATUS – current status of device
      - COMMAND – tells the devices to perform a task
      - DATA - send/receive data to/from device
    - **Reading/writing into these registers allows the OS to control device behavior**
  - Typical interaction of the OS with a device:

```
While ( STATUS == BUSY)
  ; // wait until device is ready to receive command
    // repeatedly checks status of register (polling)
Write data to DATA
Write command to COMMAND
  ( data is present + Device starts + executes command)
While ( STATUS == BUSY)
  ; // OS waits for device to finish by polling again
```

  - Advantage: protocol works + simple.
  - Disadvantage: polling is inefficient – wastes a lof of CPU time waiting for device to finish. Might as well have switched to another ready process; better use of CPU resources
- How can the OS check the device status without frequently polling, thus lowering CPU overload? Ans: Using **Interrupts**
- Instead of polling; issue request; put calling process to sleep; context switch to another task; when usage of a device finishes, it raises a hardware interrupt. This causes CPU to jump into OS at a predetermined interrupt service routine/interrupt handler.

- Handler finishes request, wakes waiting process for I/O.
  [NB: Interrupts allow for overlap of computation & I/O]
- See diagram in section 36.4, the OS is running process 2 on CPU, while Disk handles process 1, after Disk is done, interrupts occurs, process 1 wakes up (by OS) and runs. Both CPU and Disk utilized properly.

  [NB-1: If I/O device completes tasks very quickly, then interrupts would slow the system down – due to constant switching from ready-process to issuing-process. This is expensive in terms of CPU. NB-2: If I/O device speed is unknown then a "hybrid" approach is used – start with polling, if it takes too long then switch to interrupt.]
- **More efficient** data movement through **DMA** (Direct Memory Access)
  - When using programmed I/O (PIO) to transfer large chunks of data to a device – too much burden on CPU
  - Solution: OS tells DMA engine where the data is in memory, how much data to copy, where to send it. See diagram in section 36.5.
- Efficiency, done. But, how does the OS *actually* talk with a device? Explicitly or otherwise?
  - Method 1: **I/O instructions** – send data to specific registers
  - Method 2: **Memory mapped I/O** – makes device registers available as memory locations
- Final problem: How to build a device-neutral OS? Every device can potentially come with its own interface. Eg: How to create a file system that can work for any type of disk that is connected with an OS?
- Using **Abstraction**. Eg: Consider the Linux File System Stack, see Figure 36.3.
  - Identify the file system/application and what class it belongs to
  - file system/application issues block read/write requests to generic block layer which routes to the appropriate device driver.
- *"OS has millions of lines of code…"*, some say. What they are really saying is that the OS has millions of lines of device-driver code.
  [NB-1: During installation, most of this code isn't active; only a few devices are connected at a time NB-2: Device specific code are written by non-kernel devs, which is why they tend to be buggy and primary contributor to kernel crashes.]

Chapter 37: Hard Disk Drives

- HDDs have been the main form of persistent data storage in systems for decades, and its behavior has shaped the development of file systems.
- Crux: How do modern HDDs store data? What is the interface? How is data laid out and accessed? How does disk scheduling improve performance?
- Modern HDD interface:
    - Large number of 512-blocks; each called a sector
    - Read/write functionality in every sector
    - N-Sectors numbered from 0 – (n-1); i.e. an array of sectors
- Things to cover:
    - Basic geometry
    - How requests are processed?
        - Single-track latency – rotational delay
        - Multiple Tracks – seek time
    - I/O time math stuff
    - Disk scheduling
        - SSTF – shortest seek time first
        - SCAN/C-SCAN - SSTF without starvation
        - SPTF – shortest positioning time first

Chapter 38: Redundant Arrays of inexpensive Disks (RAIDs)

- Why can't we have the biggest, bad-dest, most awesome disk of all time? I/O operations are slow and as a result, it becomes a bottleneck for the entire system. How can we make a large, fast, and reliable storage system? Key techniques in doing so? Trade-offs between different approaches?

- RAIDs is about:
    - Using multiple independent disks, in parallel, into 1-large + reliable entity
    - Done **transparently**, i.e. OS can't tell the difference and doesn't need new code to be compatible
    - Advantages over a single disk?
        - **Performance**: using multiple disks in parallel can greatly speed up I/O times
        - **Capacity**: large data sets demand large disks
        - **Reliability**: without RAID techniques, spreading data across multiple disks would make data vulnerable should a single disk fail – subject to some **redundancy**. Should a failure occur with a single disk, RAIDs can keep operating as normal.

Interface & internals

- Just as a single disk, the OS would interpret RAID to be a linear array of blocks, each of which can be read/written by a file system.
- When a logical I/O request is made, RAID calculates which disk needs to be accessed in order to complete the request
- Then it issues a physical I/O (the nature of which is dependent on the RAID level).
- From a system point of view, RAID is pretty much a specialized computer. It contains:
    - Microcontroller (firmware) – handles RAID operations
    - Volatile memory (DRAM) – data buffer blocks when reading/writing

Fault Model

- From a design perspective, RAIDs need to detect and recover when a disk fault occurs, and so recognizing the nature of a fault is critical.
- Two models:
    - **Fail-stop**: the disk will be in two-states – *working* (read/write) or *failed* (assume data is permanently lost).
        - Major assumption: faults are easily detected
    - More complex faults later…

RAID Evaluation

- Several approaches to building RAID, each with their own characteristics (strengths/weaknesses)
- Fundamentally, RAIDs are designed around three axes:
    - **Capcity**: given N disks, each with B blocks => N x B available (assume no redundancy). [NB: this is typically the case for RAID-0 (striping). However, in the case for RAID-1 (mirroring), which keeps two copies of each block, the useful capacity in this case = (N x B)/2 RAID-4/5 (parity-based) would fall in the middle of the two]
    - **Reliability**: RAID can withstand only a single disk failure (when we get to data integrity, we'll revisit this) and still function as normal
    - **Performance**: difficult to categorize as it is based on the workload that the disk will handle

RAID-0 : Striping

- No redundancy
- Serves as an upper-bound on performance and capacity
- See Figure 38.1 – simple 4-disk striping model; blocks are distributed in a "round-robin" fashion; each block has a size of 4KB, also called **chunk**; all blocks within the same row is a **stripe**, which can hold 16 KB of data.

- RAID-0 is designed to extract maximum parallelism, when requests to data are in the form of contiguous chunks. [NB: It is also possible to increase chunk size, see Figure 38.2, where two-blocks within the same disk form a chunk. Thus, in this case, two rows would make a stripe.]

RAID Mapping Problem

- Crux: given a logical block to read/write, how does RAID know which physical disk and offset to access?
- Looking at Figure 38.1, chunk size = 1 block = 4KB, given local block address A:
    - Disk = A % (#disks)
    - Offset = A / (#disks)
- However, chunks can vary in size and so..
    - Disk = (A / chunk size) % (#disks)
    - Offset = (A / (chunk size * (#disks)) * (chunk size + A % chunk size)

Chunks

- Its size mostly affects performance of the array
    - Small size = many files will get stripped across multiple disks
        - Advantage: increasing parallelism of reads/writes to a single file
        - Disadvantage: positioning time increases
    - Large size
        - Advantage: positioning time decreases
        - Disadvantage: parallelism decreases and RAID relies more on multiple concurrent requests to obtain higher throughput

RAID-0 : Stripping analysis (evaluating the 3-axes)

- **Capacity**: perfect. Given N disks, each of size B blocks -> stripping delivers N x B useful capacity
- **Reliability**: perfect, but in a *bad way*. If any one of the disks fails, all data is lost. Doomed!
- **Performance**: perfect. All disks are utilized, often in parallel, for user I/O requests.

RAID-0 Performance Evaluation (2-metrics)

- Metric 1: **single-request latency** – helps understand level of parallelism of a single logical I/O request.
- Metric 2: **steady-state throughput** – total bandwidth of many concurrent requests (focus on this)
- To understand throughput, go back to workloads. There are two kinds:
    - Sequential = large contiguous chunks (eg: searching for key word in file)
        - Most efficient mode for disk – less time seeking and waiting for rotation, focused on transferring data
    - Random = small, random localization (eg: DBMS query)
        - Complete opposite of sequential
        - Focused on searching and waiting for rotation and transferring data

    [NB: in the real-world, workloads will be a mix of sequential and random]