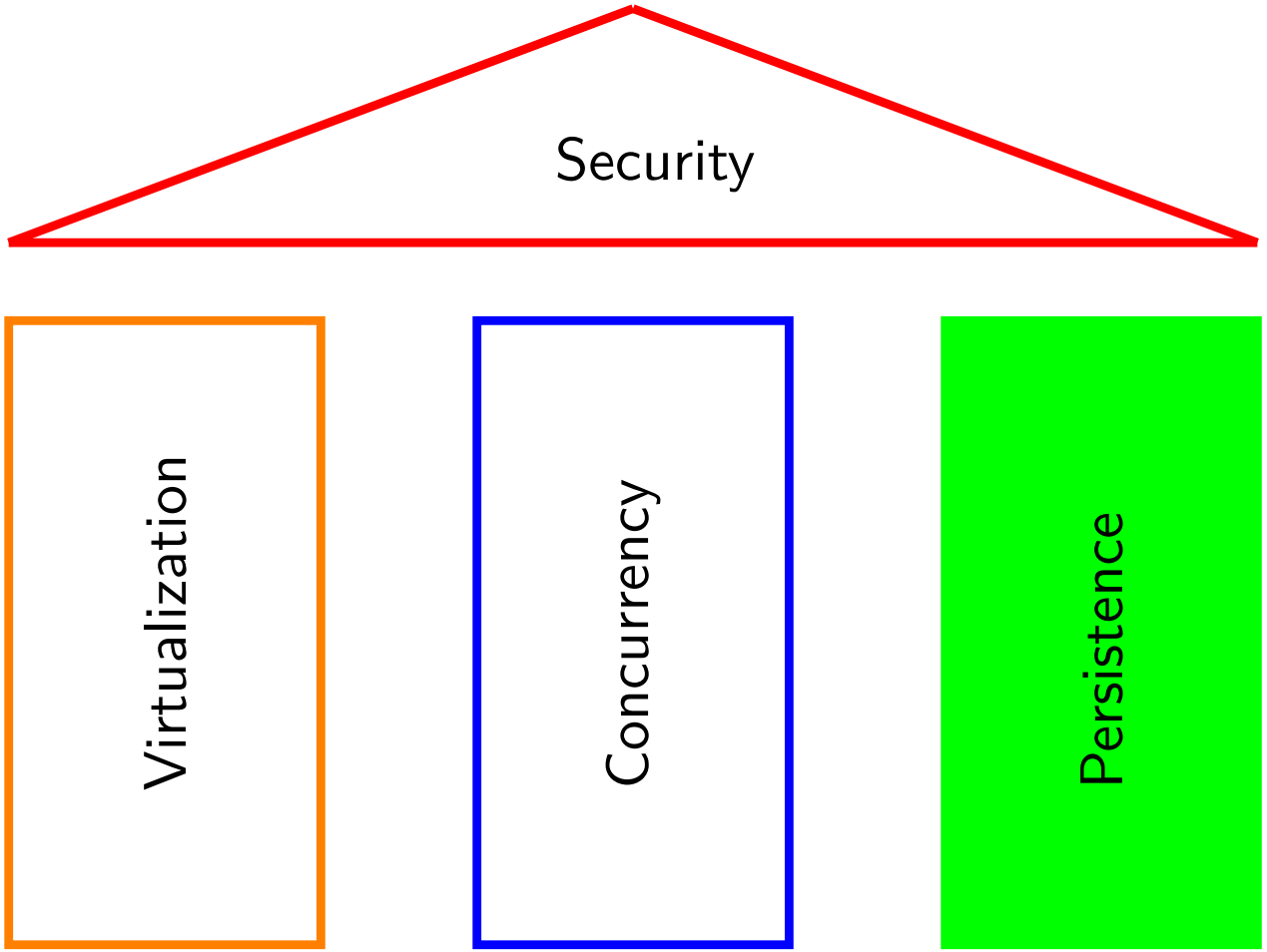
CS323 Operating Systems

Persistence Summary

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Persistence



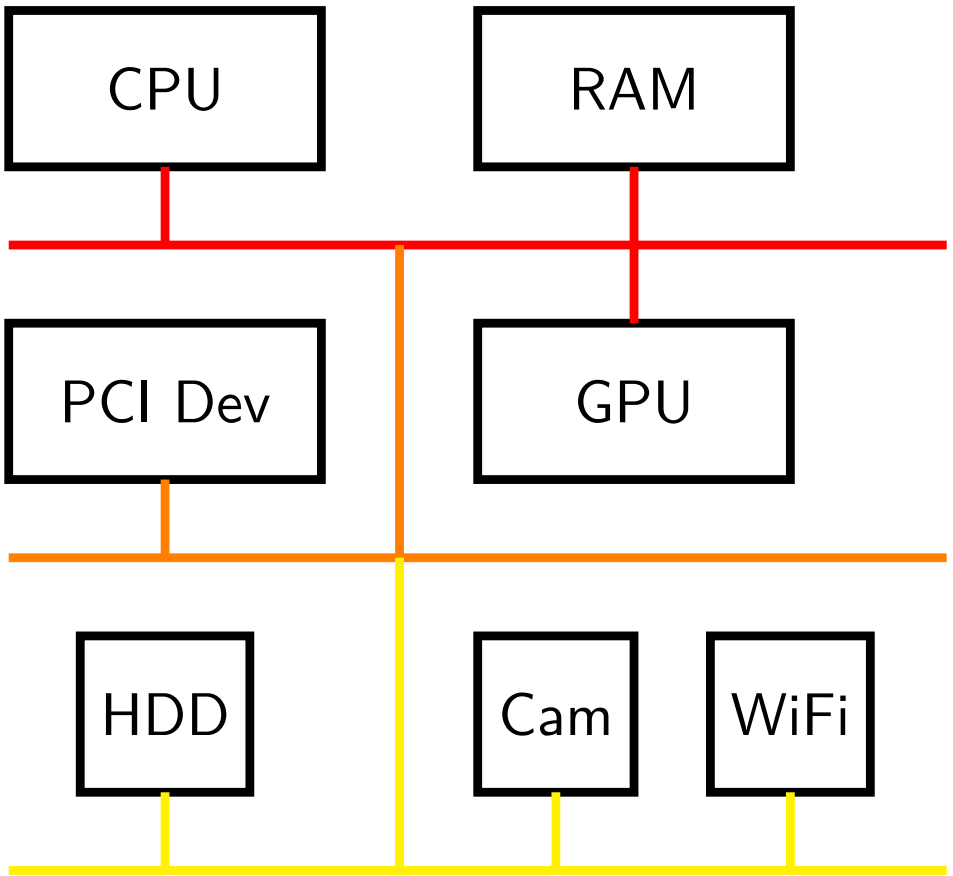
Persistence Topics

- Device interaction and device drivers
- IO Scheduling and harddrive throughput
- Filesystem API
- Filesystem implementation
  - Inodes and devices
  - File descriptors
  - File names
- Crash resistance
- Journaling

Motivation

- So far we have talked about the CPU and about RAM
- How do we get data into RAM?
  - Load programs and data from storage
  - Read and write packets from the network (maybe even streams?)
  - Write data to a terminal or the screen
  - Read data from input devices such as keyboard/mouse/camera
- Devices provide input/output (IO) to a system
- IO allows information to *persist* (RAM is volatile)!
- Enables interesting computation!

IO buses

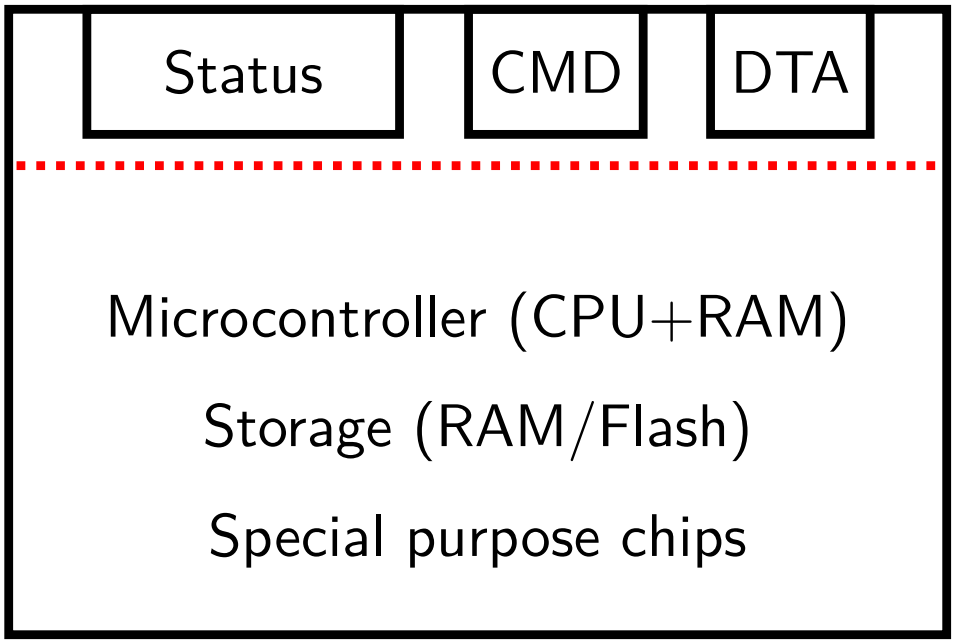


Northbridge, ~19,600 MB/s
memory bus / PCI-E

Southbridge ~4,000 MB/s
I/O bus (e.g., PCI)

Peripheral I/O bus, ~2,000 MB/s
(e.g., SATA, USB)

Canonical IO device

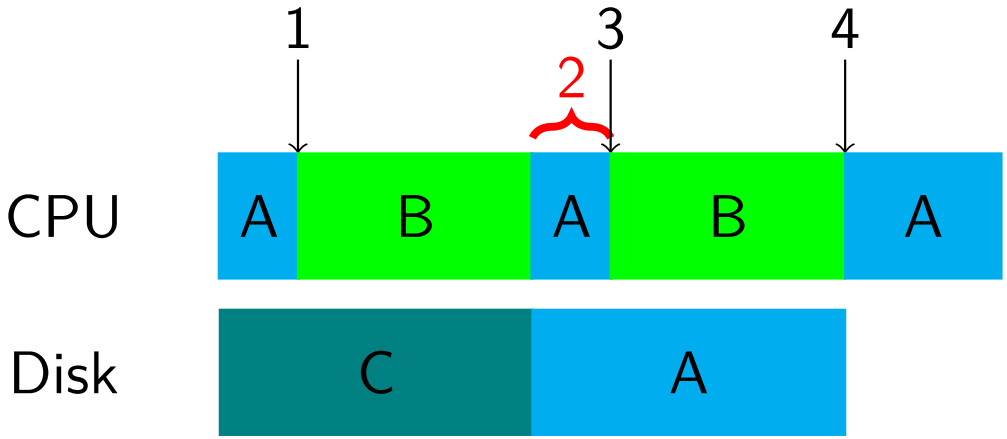


(Device registers)

(Device internals)

- OS communicates based on agreed protocol (through "driver")
- Device signals OS through memory or interrupt

IO data transfer

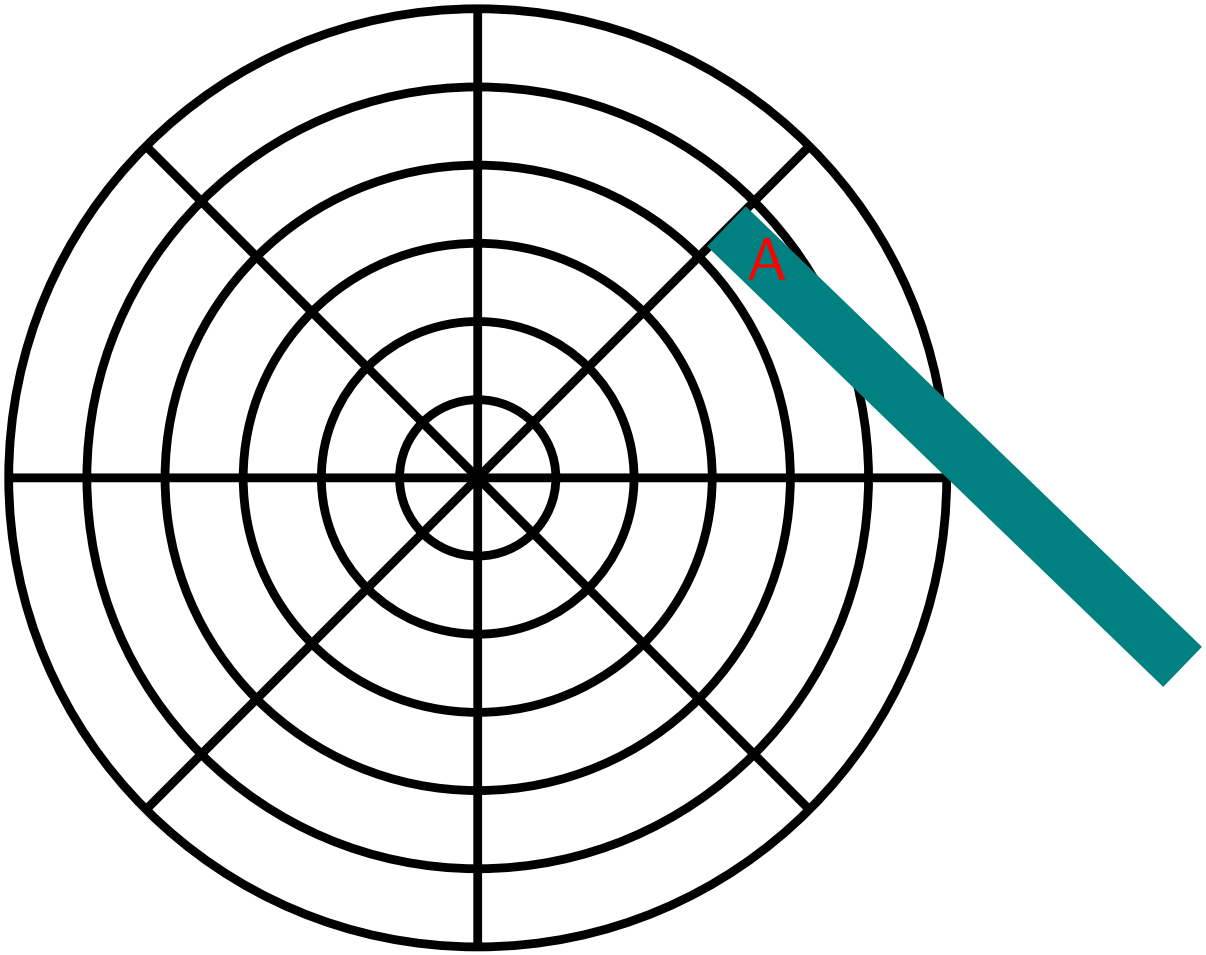


- **PIO (Programmed IO)**: CPU tells the device *what* data is
  - One instruction for each byte/word
  - Efficient for a few bytes, scales terribly
- **DMA (Direct Memory Access)**: tell device *where* data is
  - One instruction to send a pointer
  - Efficient for large data transfers

Support for different devices

- Challenge: different devices have *different protocols*
- Drivers are specialized pieces of code for a particular device
  - Low end communicates with the device
  - High end exposes generic interface to OS
- Drivers are an example of *encapsulation*
  - Different drivers adhere to the same API
  - OS only implements support for APIs based on device class
- Requirement: well-designed interface/API
  - Trade-off between versatility and over-specialization
  - Due to device class complexity, OS ends with layers of APIs

Hard disk



- IO: seek time, rotation time, transfer time

RAID: Redundant Array of Inexpensive Disks

- Idea: build logical disk from (many) physical disks
- RAID0: Striping (no mirroring or parity)
  - n performance, n capacity, 0/n can fail
- RAID1: Data mirroring (no parity or striping)
  - n performance, (n-1)/n can fail
- RAID2: bit level striping (historic, sync'd, one parity drive)
- RAID3: byte level striping (historic, sync'd, one parity drive)
- RAID4: block level striping (historic, one drive holds parity)
- RAID5: block level striping, distributed parity
  - n performance, n-1 capacity, 1/n can fail
- RAID6: block level striping, distributed parity
  - n performance, n-2 capacity, 2/n can fail
- RAID 01: two stripes (RAID0) that are mirrored (RAID1)
- RAID 10: stripe (RAID0) a set of mirrored devices (RAID1)

IO/Driver Summary

- Overlap IO and computation as much as possible!
  - Use interrupts
  - Use DMA
- Driver classes provide common interface
- Storage: read/write/seek of blocks
- Minimize random IO (i.e., quick sort is really bad on HDDs)
- Carefully schedule IO on slow devices
- RAID virtualizes disks

Purpose of a file system

- Given: set of persistent blocks
- Goal: manage these blocks efficiently. How?
- Provide mechanisms to organize files and their metadata (e.g., owner, permissions, or type)
- Share files (concurrently?) among users and processes
  - Decide on locking granularity and binding operations
  - Semantics of operations like truncating in a shared world

The `file` abstraction

- A file is a linear persistent array of bytes
  - Operations: read or write
  - Metaoperations: create, delete, modify permissions/user/...
- Different perspectives
  - File name (human readable)
  - Inode and device number (persistent ID)
  - File descriptor (process view)
- Directory contains subdirectories
  - List of directories, files, inode mappings

Different names for different use cases

- inode/device id (53135/2)
  - Unique internal name
  - Records metadata about the file (size, permissions, owner)
- path (`/foo/bar/baz`)
  - Human readable name
  - Organizes files in a hierarchical layout
- file descriptor (5)
  - Process internal view
  - Avoids frequent path to inode traversal
  - Remembers offset for next read/write

File API

- `int open(char *path, int flag, mode_t mode)`
- `size_t read(int fd, char *buf, size_t nbyte)`
- `size_t write(int fd, char *buf, size_t nbyte)`
- `int close(int fd)`

Open translates a string name to an inode. OS allocates a file descriptor that points to that inode and returns the file descriptor table index. The path is only traversed once, the OS can cache inodes and each process keeps track of its open files.

Virtual File System

- File systems (EXT4, NTFS, FAT) use different data structures
- A Virtual File System (VFS) abstracts from the real filesystem
- VFS abstracts the FS as objects with specific operations
  - Superblock (mount): a life filesystem
  - File (open): a file opened by a process ("open file description")
  - Directory entry cache: speeds up path to inode translation
  - Inode (lookup): a filesystem object (e.g., file or directory)
- System call logic (open, seek, ...) maps to VFS operations
- When implementing a new FS, implement the VFS API
- System calls are now independent of FS implementation

Allocating file data

- Contiguous
- Linked blocks (blocks end with a next pointer)
- File-allocation tables (table that contains block references)
- Indexed (inode contains data pointers)
- Multi-level indexed (tree of pointers)

For each approach, think about fragmentation, ability to grow/shrink files, sequential access performance, random access performance, overhead of meta data.

Inodes

- The inode stores all file metadata
- File type
- File uid, gid
- File permissions (for user, group, others)
- Size
- Access time
- Create time
- Number of links

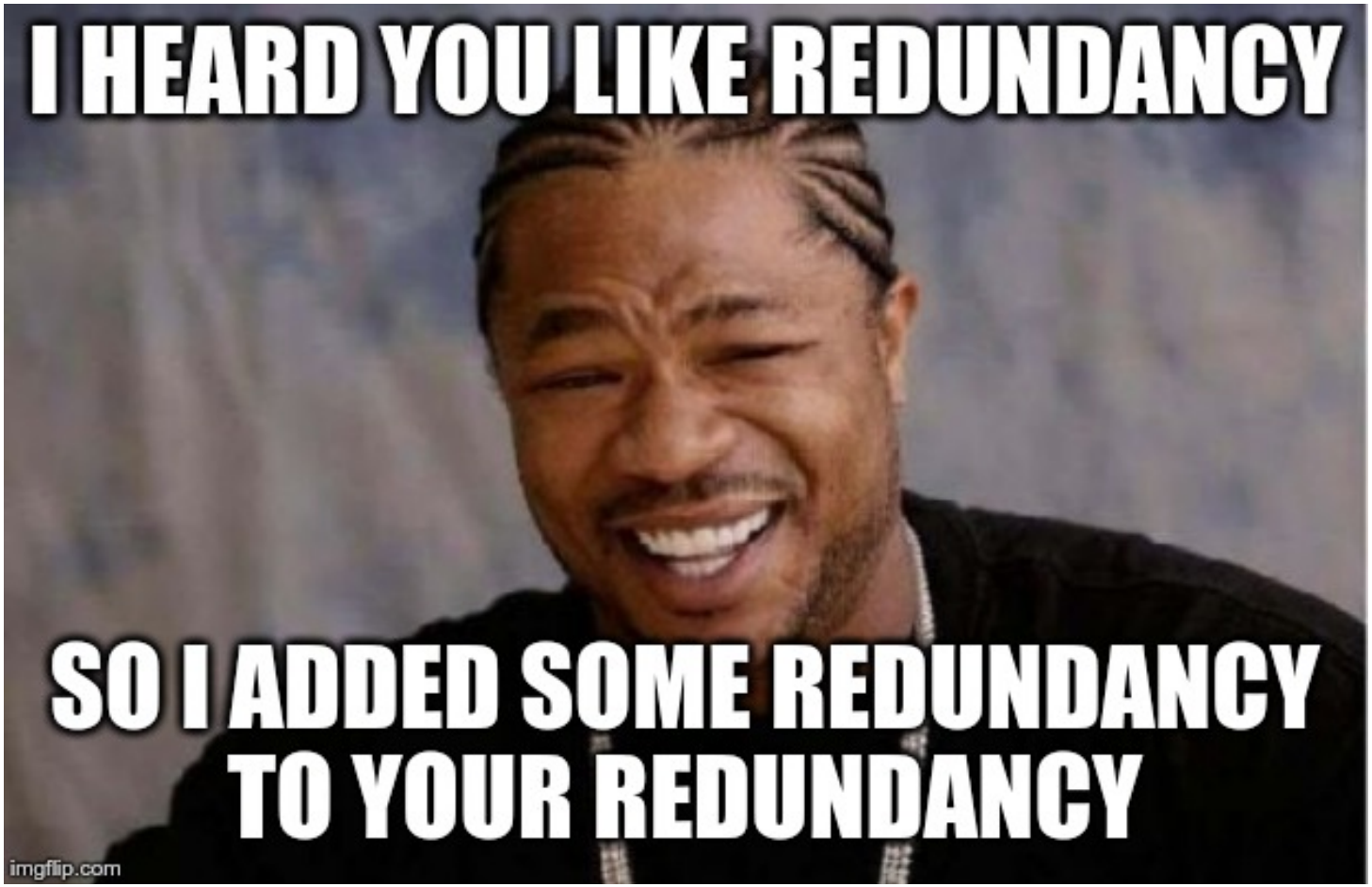
Consistency

Redundant data must be consistent to ensure correctness. Otherwise functionality may break.

- Keeping redundant data consistent is challenging
- Filesystem may perform several writes to redundant blocks
- The sequence of writes is not atomic
- Interrupts due to power loss, kernel bugs, hardware failure

Journaling strategy

- Never delete (or overwrite) ANY old data until you have received confirmation that ALL new data is committed
  - Add redundancy to fix the problem with redundancy



Persistence summary

- Drivers and IO allow abstraction and persistence
- Filesystem API: handle interaction with the file system
- Three ways to identify a file
  - File names (for humans)
  - Inodes and devices (on the disk)
  - File descriptors (for a process)
- Filesystem implementation
  - Inodes for metadata
  - Bitmaps for inodes/data blocks
  - Superblock for global metadata
- Crash resistance: filesystem check (FSCK)
- Journaling: keep track of metadata, enforce atomicity
  - All modern filesystems use journaling
  - FSCK still useful due to bitflips/bugs