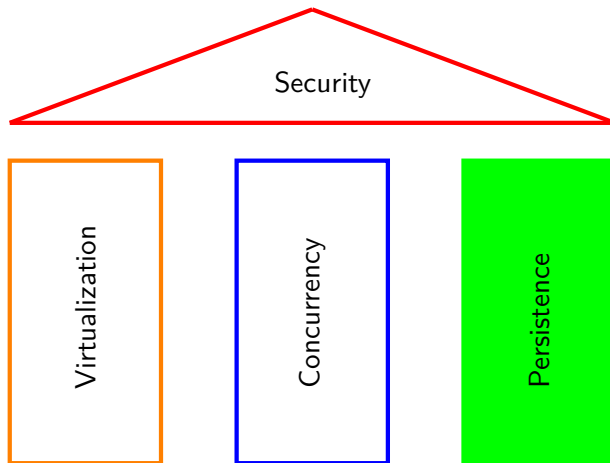


CS323 Operating Systems

Drivers and IO

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021



Topics covered in this lecture

- How does the OS interact with devices
- Design of device drivers
- Types of IO devices
- Components of hard drives
- Calculating hard drive throughput
- Scheduling algorithms for IO

This slide deck covers chapters 36, 37 in OSTEP.

- 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!

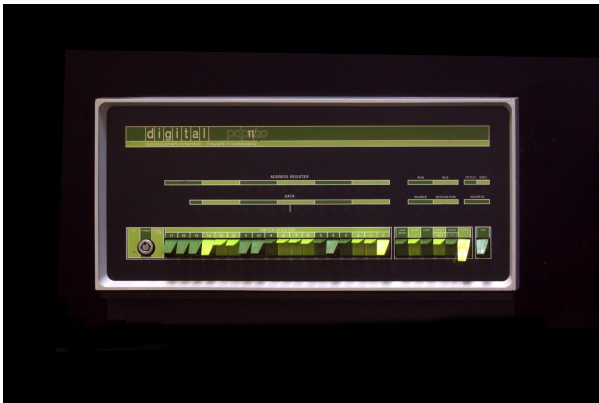


Figure 1: PDP-11/20 at Musée Bolo

- Peripherals connected to UNIBUS
- Each peripheral sets memory area it “listens” to
- No memory protection

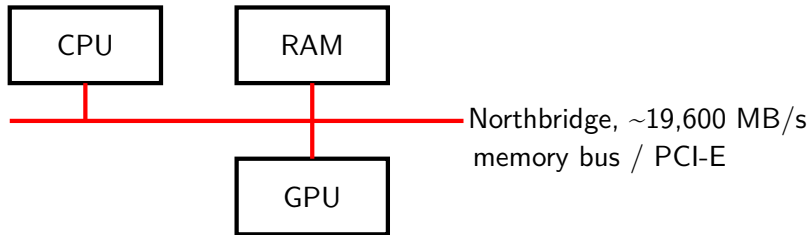
Ancient device history

- Each device (not type!) had a unique hardware interface
- Applications contained code to communicate with devices
- Application polled device to set/get information

Modern device interface

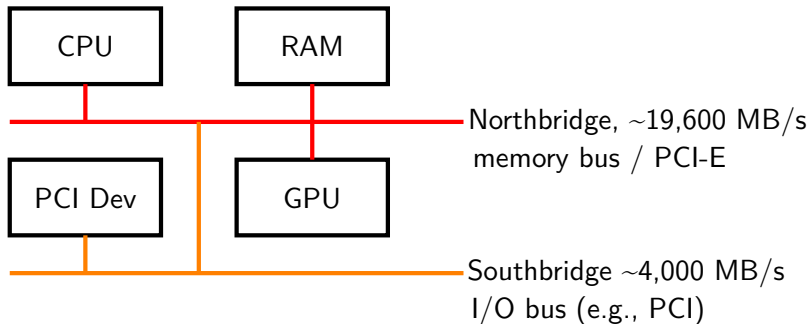
- The OS handles device management (and access)
- OS exposes a uniform interface to applications
- IO is interrupt driven

Hardware support for devices: Northbridge



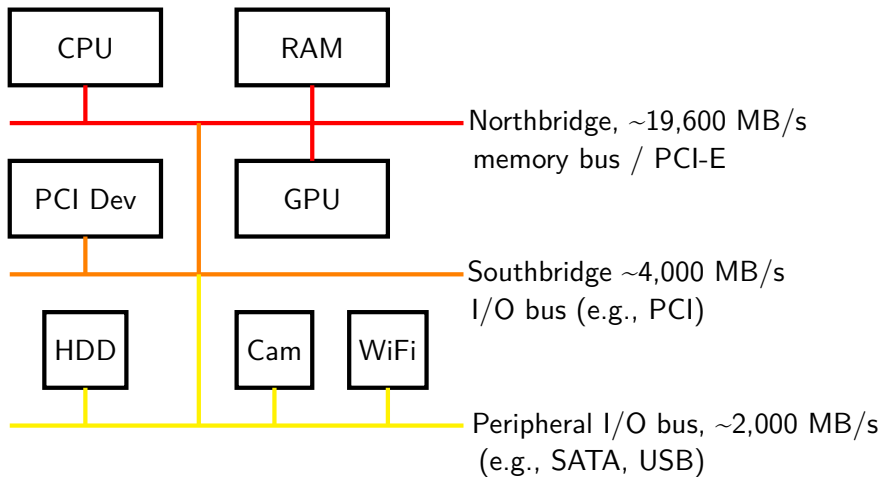
What about other devices?

Hardware support for devices: Southbridge



What about “slow” IO?

Hardware support for devices: Other IO

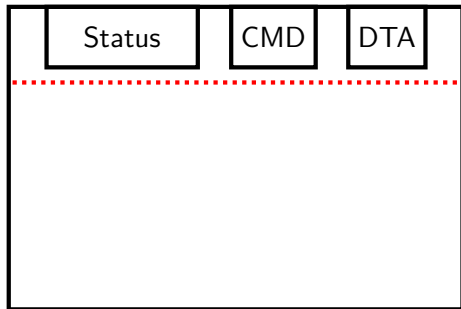


- Northbridge is also called memory controller hub
- Southbridge is also called I/O Controller Hub (ICH / Intel) or Fusion Controller Hub (FCH / AMD)
- The southbridge is connected to the CPU through the northbridge which has a direct connection

- Northbridge is also called memory controller hub
- Southbridge is also called I/O Controller Hub (ICH / Intel) or Fusion Controller Hub (FCH / AMD)
- The southbridge is connected to the CPU through the northbridge which has a direct connection

But how do devices work?

Canonical device (1/2)

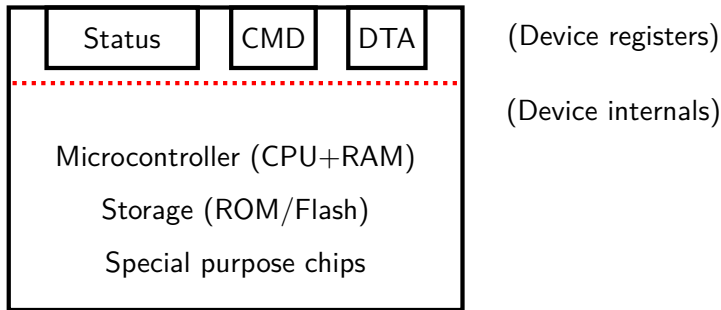


(Device registers)

(Device internals)

- OS writes device registers (by executing CPU instructions)
- Device internals are hidden (abstraction)

Canonical device (2/2)



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

Device protocol (1/3)

```
while (STATUS == BUSY) ; // 1. spin  
// 2. Write data to DTA register  
*dtaRegister = DATA;  
// 3. Write command to CMD register  
*cmdRegister = COMMAND;  
while (STATUS == BUSY) ; // 4. spin
```

- Wait until device is ready
- Set data and command (why send data first?)
- Wait until command has completed

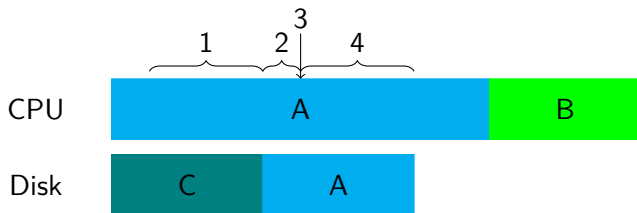
Device protocol (1/3)

```
while (STATUS == BUSY) ; // 1. spin  
// 2. Write data to DTA register  
*dtaRegister = DATA;  
// 3. Write command to CMD register  
*cmdRegister = COMMAND;  
while (STATUS == BUSY) ; // 4. spin
```

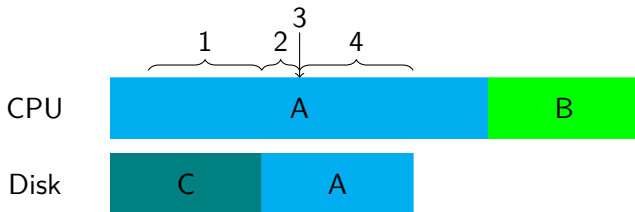
- Wait until device is ready
- Set data and command (why send data first?)
- Wait until command has completed

Where do you see problems?

Device protocol (2/3)

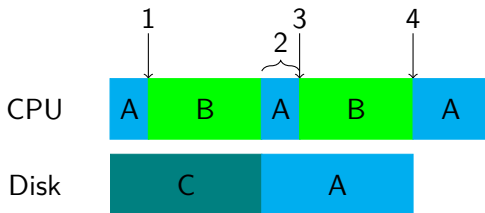


Device protocol (2/3)



- Busy waiting (1. and 4.) wastes cycles twice
- CPU should not need to wait for completion of command
- Solution: embrace asynchronous communication
 - Inform device of request (to get rid of 1.)
 - Wait for signal of completion (to get rid of 4.)

Device protocol (3/3): interrupts



- Instead of spinning, the OS (driver) waits for an interrupt
 - Interrupts are handled centrally through a dispatcher
 - On interrupt arrival, wake up kernel thread that waits on that interrupt

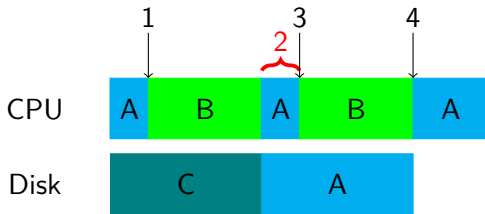
- Can interrupts lead to worse performance than polling?

- Can interrupts lead to worse performance than polling?
- Yes: livelock (e.g., a flood of arriving network packets)
 - A livelock is *similar to a deadlock* (no process makes progress, resulting in starvation) with the difference that the states of the processes constantly change
 - For example: network packets arrive; interrupt handling and context switch is costly, prohibiting the application from reacting to the packets and them simply queuing up.

Interrupt performance

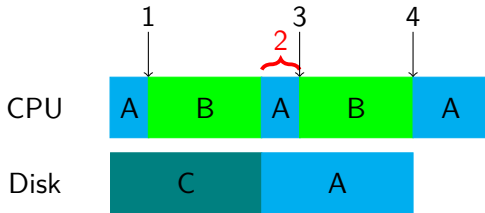
- Can interrupts lead to worse performance than polling?
- Yes: livelock (e.g., a flood of arriving network packets)
 - A livelock is *similar to a deadlock* (no process makes progress, resulting in starvation) with the difference that the states of the processes constantly change
 - For example: network packets arrive; interrupt handling and context switch is costly, prohibiting the application from reacting to the packets and them simply queuing up.
- Real systems therefore use a mix between polling and interrupts
 - Interrupts allow overlap between computation and IO, most useful for slow devices;
 - Use polling for short bursts or small amounts of data
- Another optimization is *coalescing*, i.e., the device waits for a bit until more requests complete, then batch sends everything.

Optimize data transfer



- **PIO (Programmed IO):** CPU tells the device *what* data
 - One instruction for each byte/word
 - Efficient for a few bytes, scales terribly

Optimize data transfer



- **PIO (Programmed IO):** CPU tells the device *what* data
 - 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

How to transfer data?

- IO ports
 - Each device has an assigned IO port
 - Special instructions (`in/out` on x86) communicate with device
- Memory mapped IO
 - Device maps its registers to memory
 - Loads/stores interact with device

How to transfer data?

- IO ports
 - Each device has an assigned IO port
 - Special instructions (`in/out` on x86) communicate with device
- Memory mapped IO
 - Device maps its registers to memory
 - Loads/stores interact with device
- Both are used in practice. Architectures now support both. Differences are a matter of choice and preference.

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

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

Complexity of API layers

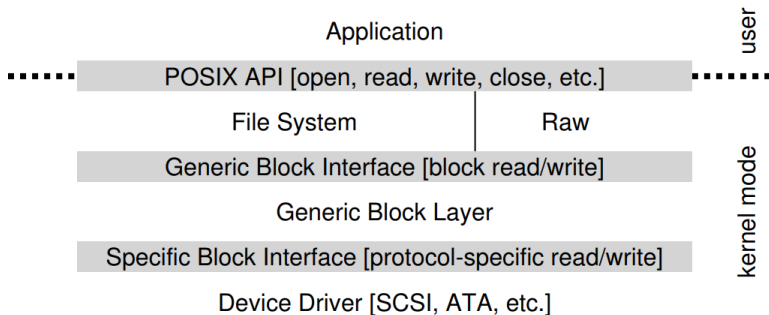
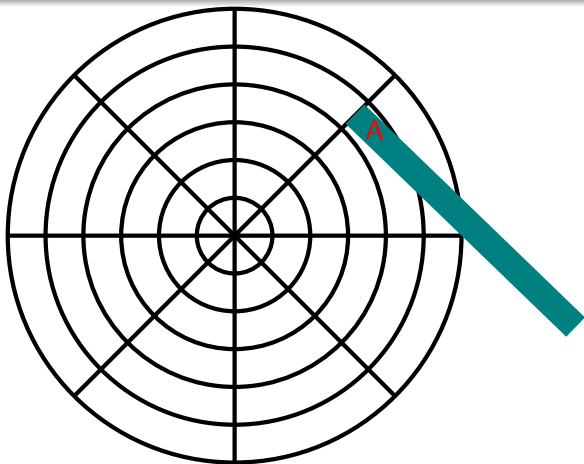


Figure 2: File system stack

IO subsystem: hard disks

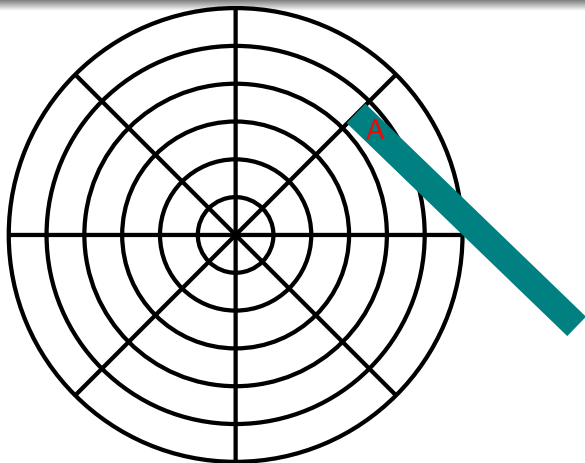
- Disk has a sector-addressable address space
- Sectors are 512 or 4096 bytes
- Main operations: read/write

Hard disk



- IO cost is sum of:
 - seek time (adjust angle of reader)
 - rotation time (rotate the start of the sector to the reader)
 - transfer time (rotate sector under reader)

Disc drives



- Hard drive spinning up
- Open hard drive spinning up
- How floppy drives work
- Alternate use for floppy drives

HDD: seek, rotate, transfer

- Seek is costly (several ms, “an eternity”)
 - function of cylinder distance
 - operations: accelerate, coast, decelerate, settle
- Rotate: 7200 RPM, 8.3 ms/rotation
- Transfer: (100MB/s): 5 us for 512 B

HDD: seek, rotate, transfer

- Seek is costly (several ms, “an eternity”)
 - function of cylinder distance
 - operations: accelerate, coast, decelerate, settle
- Rotate: 7200 RPM, 8.3 ms/rotation
- Transfer: (100MB/s): 5 us for 512 B
- Seeks and rotation is slow, transfer is fast
 - Sequential access is much faster (blocks are ordered)
 - Random access is slow

- Given concurrent IO requests, how should they be scheduled?
 - Different constraints than CPU scheduling
 - ... but the same algorithms!

- Given concurrent IO requests, how should they be scheduled?
 - Different constraints than CPU scheduling
 - ... but the same algorithms!
- Position of disk head relative to requested position matters more than length of the transfer

Optimization: buffering

- A buffer cache between the disk and the higher level of the OS keeps most recently used disk blocks around
- Proactively fetches blocks that are likely accessed
- Keep track of dirty blocks that need to be written back

First come first serve

- Assume 10ms average seek and rotation time
- Requests: 30001, 70001, 30002, 70002, 30003, 70003: 60ms
 - How can we improve?

First come first serve

- Assume 10ms average seek and rotation time
- Requests: 30001, 70001, 30002, 70002, 30003, 70003: 60ms
 - How can we improve?
- Requests: 30001, 30002, 30003, 70001, 70002, 70003: 20ms
 - Shortest seek time: seek to closest block (minding starvation)
 - Elevator/SCAN: go outwards/inwards, serve requests along
 - Clever optimizations of shortest seek time/elevator

Completely fair queuing (Linux)

- Queue for each process
- Weighted round-robin between queues with slice-time proportional to priority
- Yield slice only if idle for given time

Completely fair queuing (Linux)

- Queue for each process
- Weighted round-robin between queues with slice-time proportional to priority
- Yield slice only if idle for given time

Scheduling algorithms for CPU time are also useful for I/O

Going beyond one disk

- A single disk has many limitations
 - Single point of failure
 - Limited performance
 - Limited size
- File systems work on a single disk (or partition)
- How can we increase flexibility?

Going beyond one disk

- A single disk has many limitations
 - Single point of failure
 - Limited performance
 - Limited size
- File systems work on a single disk (or partition)
- How can we increase flexibility?
- One more layer of indirection: a virtual disk!

RAID: Redundant Array of Inexpensive Disks

- Idea: build logical disk from (many) physical disks

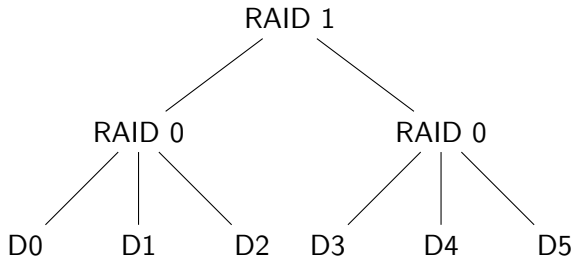
RAID: Redundant Array of Inexpensive Disks

- 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

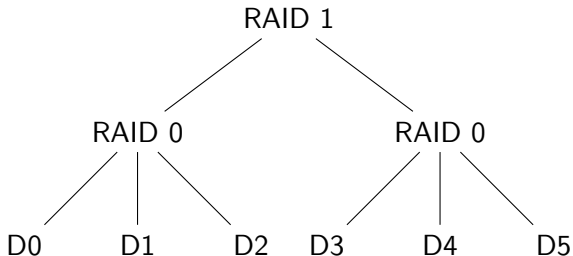
RAID: combinations

- RAID 01: two stripes (RAID0) that are mirrored (RAID1)
- RAID 10: stripe (RAID0) a set of mirrored devices (RAID1)
- Which one is more reliable?

RAID 01

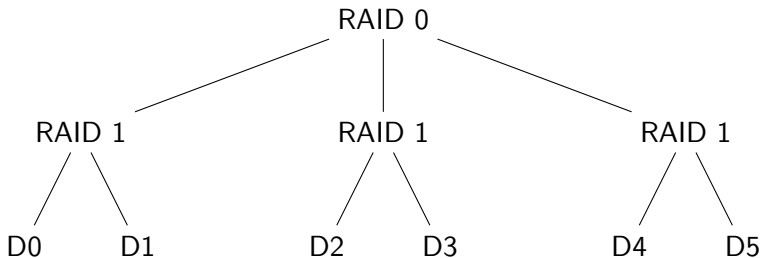


RAID 01



- During rebuild (a disk has failed), no other drive from the *alternate* group may fail
- Given D0 fails, if any of D3, D4, or D5 fails the system halts (3/5)
- Given D0 fails, if D3 fails the system is dead (1/5)

RAID 10



- Of each mirror, at least one disk must remain healthy
- Given D0 fails, if D1 fails the system is dead (1/5)
- Given D0 fails, any other disk (except D1) may fail without impact (1/5)

- 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

Don't forget to get your learning feedback through the Moodle quiz!