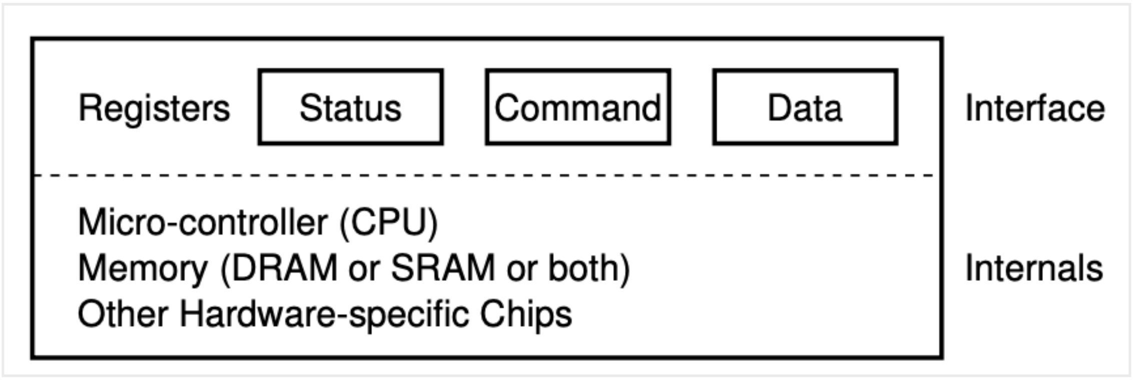
Deadlock: Deadlocks can only happen when these four conditions hold:

1. mutual exclusion
   Problem: Threads claim exclusive control of resources that they require
   Strategy: Eliminate locks!
2. hold-and-wait
   Problem: Threads hold resources while waiting for additional resources
   Strategy: Acquire all locks atomically, Can release locks over time, but cannot acquire again until all have been released
   How: Use a meta lock
   Drawbacks:
   - Must know ahead of time which locks will be needed
   - Must be conservative (acquire any lock possibly needed)
   - Degenerates to just having one big lock (reduces concurrency)
3. no preemption
   Problem: Resources (e.g., locks) cannot be forcibly removed from other threads
   Strategy: if thread can not get what it wants, release what it holds
   Drawbacks:
   - **Potential Livelock:** No processes make progress, but state of involved processes constantly changes
   - **Classic solution:** Exponential random back-off
4. circular wait
   Circular chain that each thread holds a resource (e.g., lock) requested by next thread in chain
   Example: Lock Ordering in Xv6
   - a lock on the directory
   - a lock on the new file's inode
   - a lock on a disk block buffer
   - idelock
   - ptable.lock
5. **Practical Solution:**
   - Decide which locks must be acquired before others
   - If A before B, never acquire A if B is already held!
   - Document and write code accordingly

Concurrency:
???
Persistence:
I/O



Interface: present to the rest of the system, and system software can control the operations.
Internal: implement the abstraction presented.
Status: read to see the current status of the device;
Command: tell the device to perform a certain task;
Data: pass data to the device, or get data from the device.
Steps: polling for status ready→OS send data→OS send command→polling for status finished
Approach 1: Special I/O instructions

- Addition to the CPU's instruction set that allows accessing hardware
- e.g, IN and OUT in x86

Approach 2: Memory-Mapped I/O

- Device registers are mapped into memory
- OS can write to device registers like to any other memory location
- *Not* the same as mmap!

Problems

1. Polling waste CPU time.
2. Data movement is very CPU intensive

Solution 1: SPIN-free Device Access

- Instead of Polling, go to sleep
- Process state changes to BLOCKED
- Device notifies when it is done using an interrupt
- Problems:
    - If device is very fast, we get very frequent interrupts
    - Leads to context switch overhead

Solution 2: DMA (Direct Memory Access)
OS telling DMA engine where the data in memory, how much data to copy, and which device to send. At that point, the OS is done with the transfer and can proceed with other work.
DMA is complete raise an interrupt.

- Faster than copying to CPU and then to disk
- CPU can do other things while data is being moved
- Requires specialized hardware

DISK
Platter:

- 2-sided, A drive can have multiple platters, read or written

- Rotates track at fixed speeds

Track:
- Split into fixed-size sectors (often 512 bytes),
- Contain redundant encoding to recover from corrupted bits
- exposed to the OS as a linear array

Spindle: spins the platter around at a constant fixed rate with unit **Rotations Per Minute.**
Disk Arm + Read/write Head: Moves between tracks of the platter
Controller:
- executes operations stored in command buffer
- Writes output to status or data registers
- translates track and sector location accesses to disk into internal actions
- track of multiple actions at once

Access speed
seek: move disk arm to correct track
wait (rotation): wait for sector to rotate under arm
transfer: read/write data
e.g. Read one sector (512B) avg. seek: 7ms avg. rotate: 3ms avg. transfer: ~0ms (200Mb/s) throughput is 512b/10ms ≈ 50kb/s
Slow!!!
- Random I/O are dominated by seek and rotation
- Sequential accesses can be much faster

Disk Scheduling
Goal: Maximize throughput by minimizing **seek** and **rotation** times
- Duration of each access is known
- **Shortest-Seek First (SSF**): To maximize throughput pick next operation with **smallest seek** time, Can be implemented by the OS as **nearest block first**
  - Does not account for **rotation**
  - Disk arm might stay on same track for a long time
  - Some operations could starve
- **SCAN or "Elevator", F-SCAN for "Freeze", C-SCAN for "Circular"**
  - Does not take rotation time into account, only **seek**
  - Better starvation free "Shortest-Job First" algorithms exist
- **Shortest Positioning Time First (SPTF)**: both **seek and rotate** accounted.