COS 318: Operating Systems

I/O Device and Drivers

Kai Li and Andy Bavier
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Topics

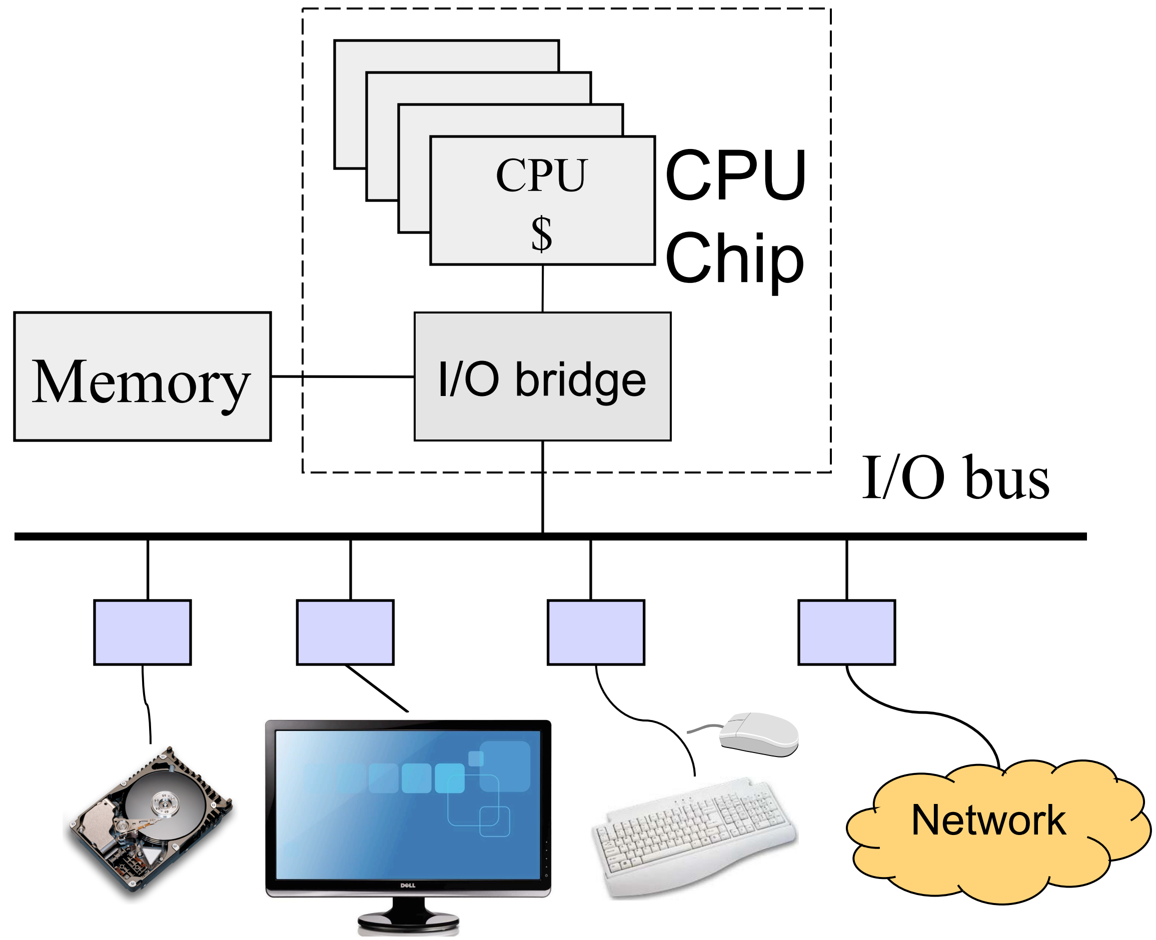
- I/O devices
- Device drivers
- Synchronous and asynchronous I/O

Input and Output

- A computer's job is to process data
  - Computation (CPU, cache, and memory)
  - **Move data into and out of a system** (between I/O devices and memory)
- Challenges with I/O devices
  - Different categories: storage, networking, displays, etc.
  - Large number of device drivers to support
  - Device drivers run in kernel mode and can crash systems
- Goals of the OS
  - Provide a generic, consistent, convenient and reliable way to access I/O devices
  - Achieve potential I/O performance in a system

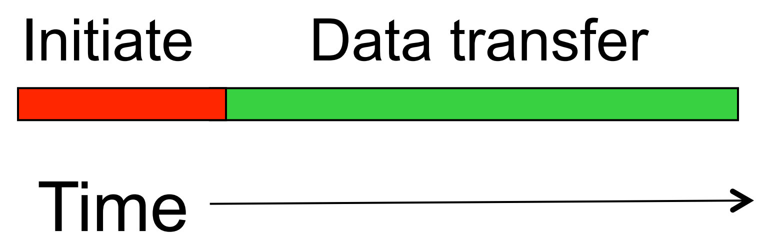
Revisit Hardware

- Compute hardware
  - CPU cores and caches
  - Memory controller
  - I/O bus logic
  - Memory
- I/O Hardware
  - I/O bus or interconnect
  - I/O controller or adaptor
  - I/O device
- Interact with devices
  - Programmed I/O (PIO)
  - Interrupts
  - Direct Memory Access (DMA)



Latency, Bandwidth, and Abstraction

- Overhead
  - CPU time to initiate an operation
- Latency
  - Time to transfer one byte
  - Overhead + 1 byte reaches destination
- Bandwidth
  - Rate of I/O transfer, once initiated
  - Bytes/sec
- General method
  - Different transfer rates
  - Abstraction of byte transfers
  - Block of bytes as a transfer unit to prorate overhead



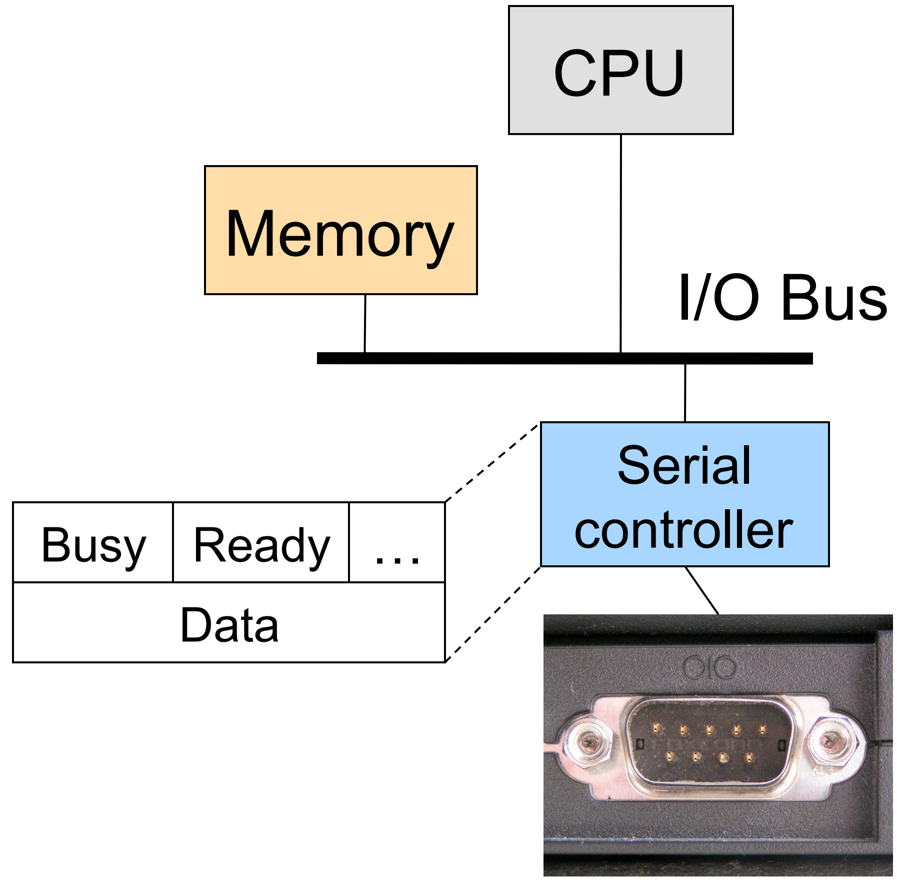
Device	**Transfer rate**
Keyboard	10Bytes/sec
Mouse	100Bytes/sec
…	…
10GE NIC	1.2GBytes/sec

Programmed I/O

- Example
  - RS-232 serial port
- Simple serial controller
  - Status registers (ready, busy, …)
  - Data register
- Output

  CPU:
  - Wait until device is not “busy”
  - Write data to “data” register
  - Tell device “ready”

  Device
  - Wait until “ready”
  - Clear “ready” and set “busy”
  - Take data from “data” register
  - Clear “busy”



Polling in Program I/O

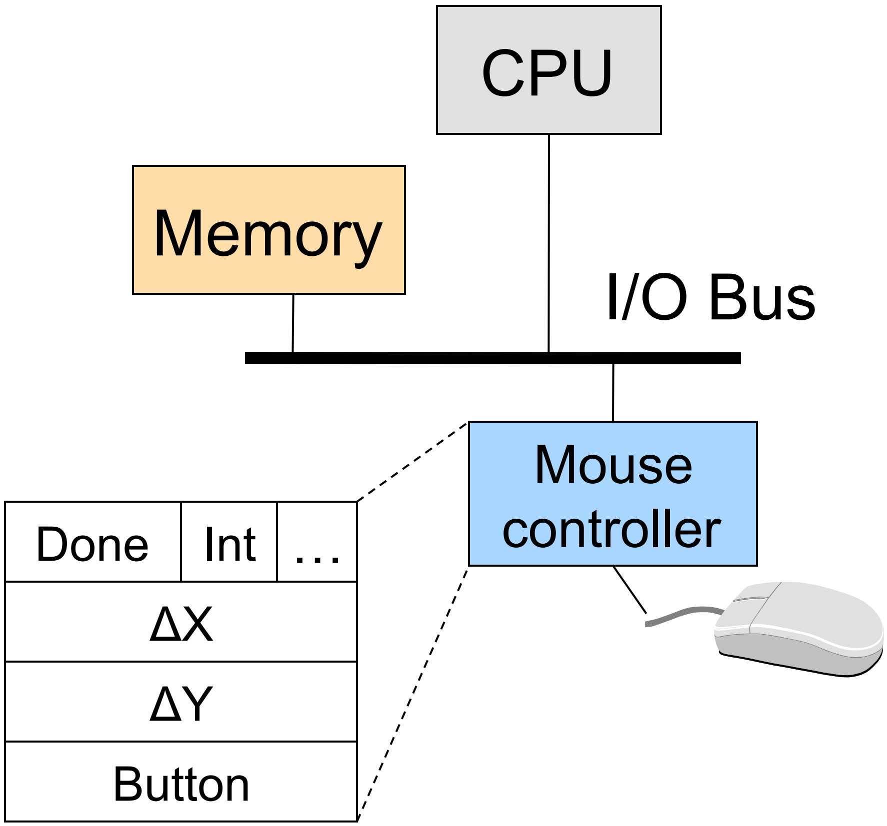
- Wait until device is not “busy”
  - A polling loop!
- Advantages
  - Simple
- Disadvantage
  - Slow
  - Waste CPU cycles
- Example
  - If a device runs 100 operations / second, CPU may need to wait for 10 msec or 10,000,000 CPU cycles (1Ghz CPU)
- Interrupt mechanism will allow CPU to avoid polling

Interrupt-Driven Device

- Example
  - Mouse
- Simple mouse controller
  - Status registers (done, int, …)
  - Data registers (ΔX, ΔY, button)
- Input

  Mouse:
  - Wait until "done"
  - Store ΔX, ΔY, and button into data registers
  - Raise interrupt

  CPU (interrupt handler)
  - Clear "done"
  - Move ΔX, ΔY, and button into kernel buffer
  - Set "done"
  - Call scheduler



Direct Memory Access (DMA)

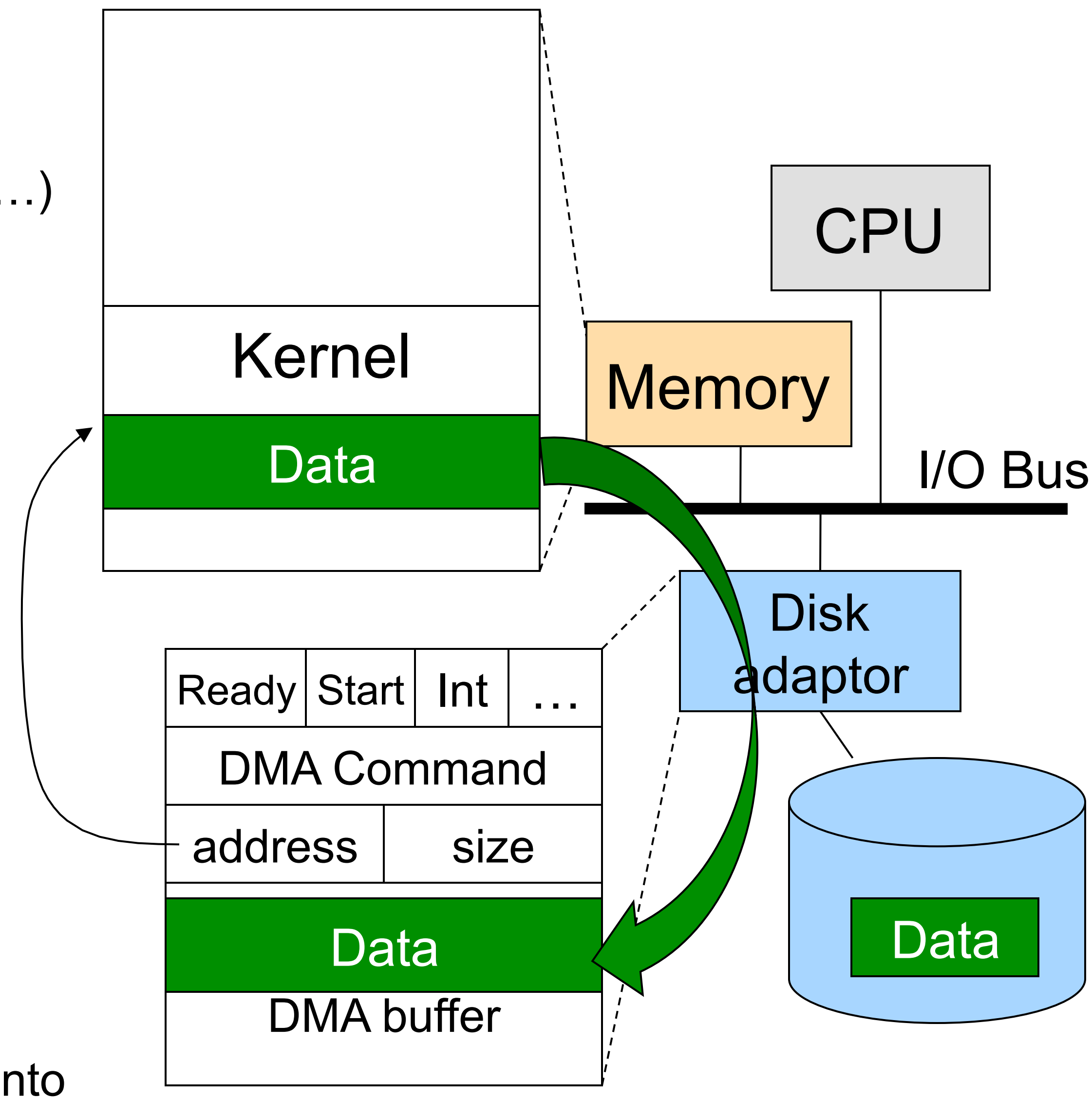
- Example
  - Disk
- A simple disk adaptor
  - Status register (done, interrupt, …)
  - DMA command
  - DMA memory address and size
  - DMA data buffer
- DMA Write

  CPU:
  - Wait until DMA device is “ready”
  - Clear “ready”
  - Set DMAWrite, address, size
  - Set “start”
  - Block current thread/process

  Disk adaptor:
  - DMA data to device (size--; address++)
  - Interrupt when “size == 0”

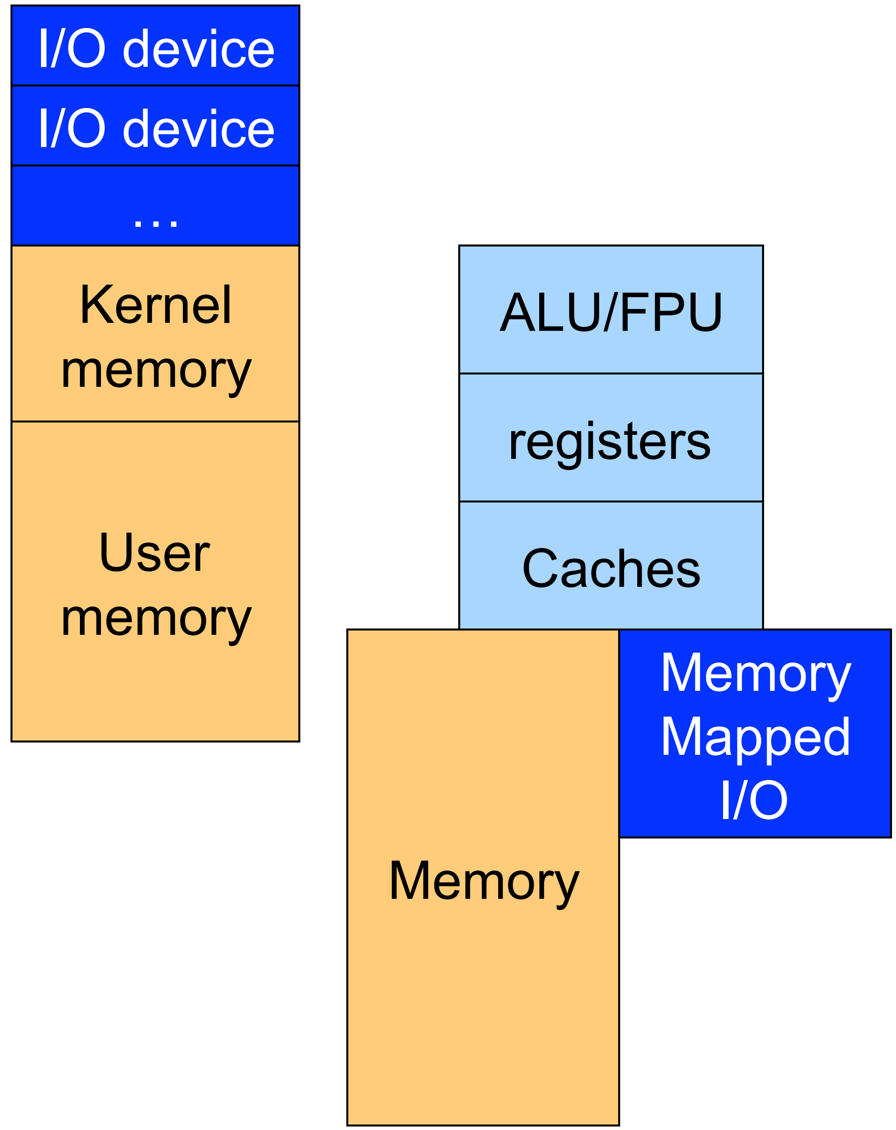
  CPU (interrupt handler):
  - Put the blocked thread/process into ready queue

  Disk: Move data to disk

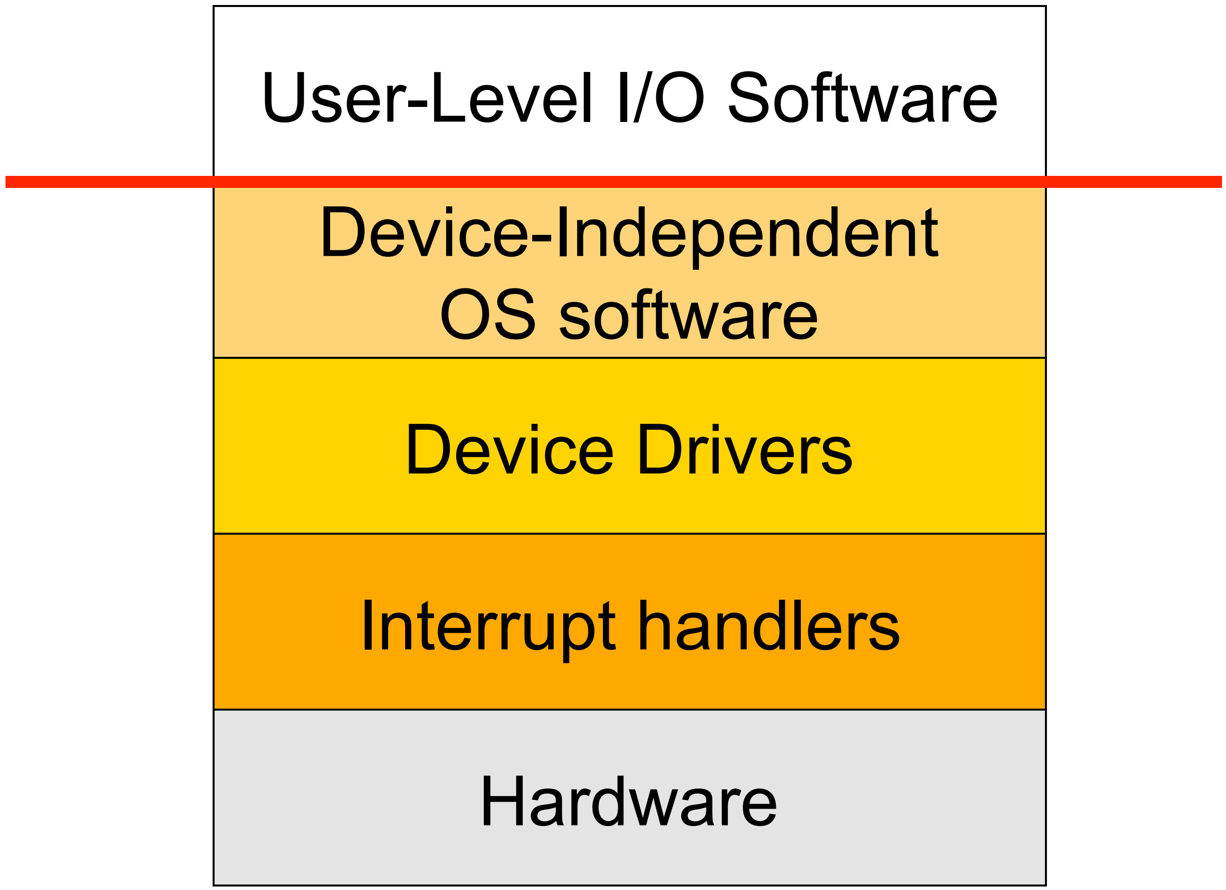


Where Are I/O Registers?

- Memory mapped I/O
  - A portion of physical memory for each device
- Advantages
  - Simple and uniform
  - CPU instructions can access these registers as memory
- Issues
  - These "memory locations" should not be cached
  - Mark them not cacheable



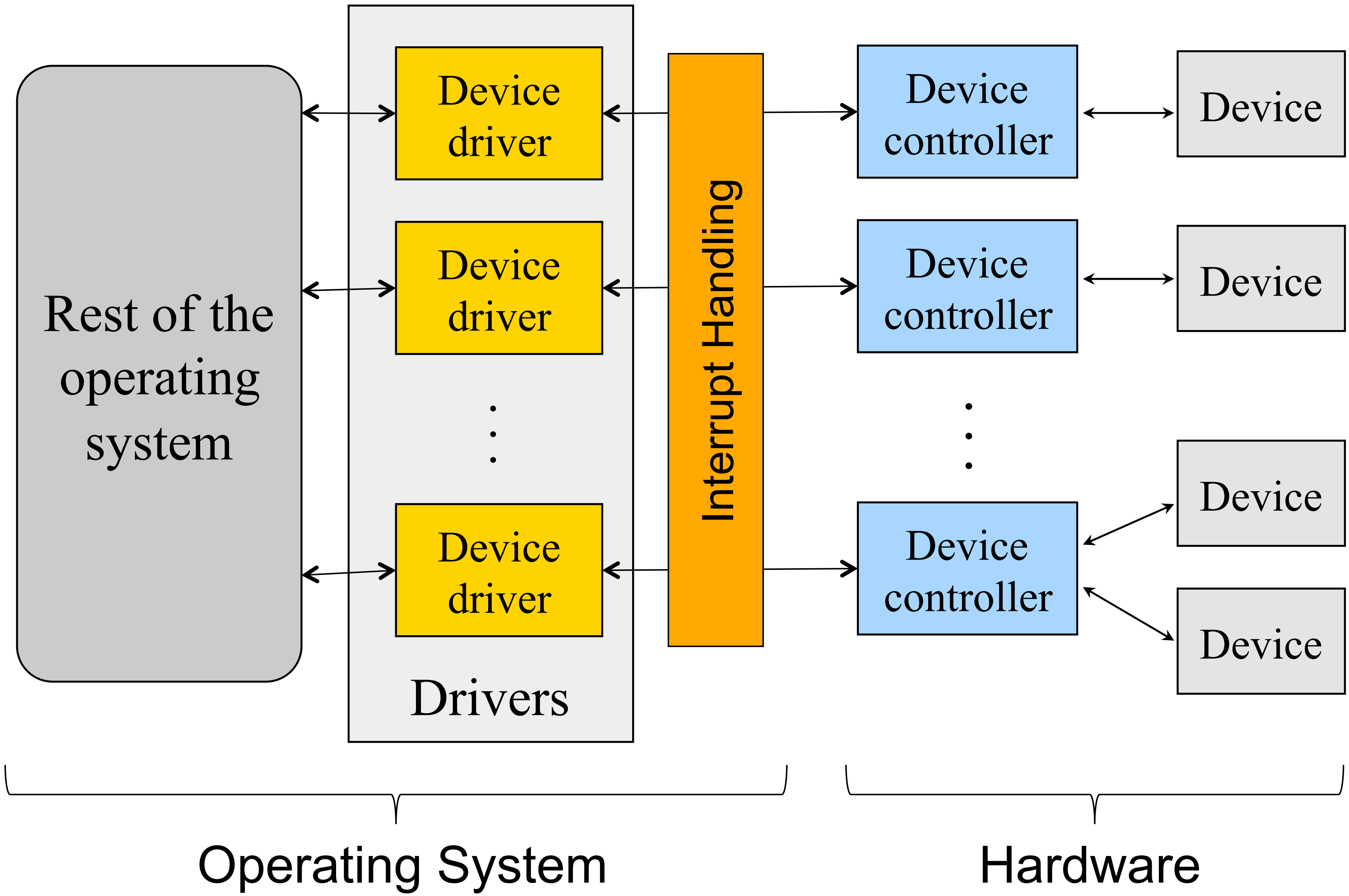
I/O Software Stack



Recall Interrupt Handling

- Save context
- Mask interrupts
- Set up a context for interrupt service
- Set up a stack for interrupt service
- Acknowledge the interrupt controller, enable it if needed
- Save entire context to PCB
- **Run the interrupt service**
- Unmask interrupts if needed
- Possibly change the priority of the process
- Run the scheduler

Device Drivers



What Does A Device Driver Do?

- Provide “the rest of the OS” with APIs
  - Init, Open, Close, Read, Write, …
- Interface with controllers
  - Commands and data transfers with hardware controllers
- Driver operations
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Operations

- Init (deviceNumber)
  - Initialize hardware
- Open(deviceNumber)
  - Initialize driver and allocate resources
- Close(deviceNumber)
  - Cleanup, deallocate, and possibly turnoff
- Device driver types
  - Character: variable sized data transfer
  - Block: fixed sized block data transfer
  - Terminal: character driver with terminal control
  - Network: streams for networking

Character and Block Interfaces

- Character device interface
  - read(deviceNumber, bufferAddr, size)
    - Reads “size” bytes from a byte stream device to “bufferAddr”
  - write(deviceNumber, bufferAddr, size)
    - Write “size” bytes from “bufferAddr” to a byte stream device
- Block device interface
  - read(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “deviceAddr” to “bufferAddr”
  - write(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from “bufferAddr” to “deviceAddr”
  - seek(deviceNumber, deviceAddress)
    - Move the head to the correct position
    - Usually not necessary

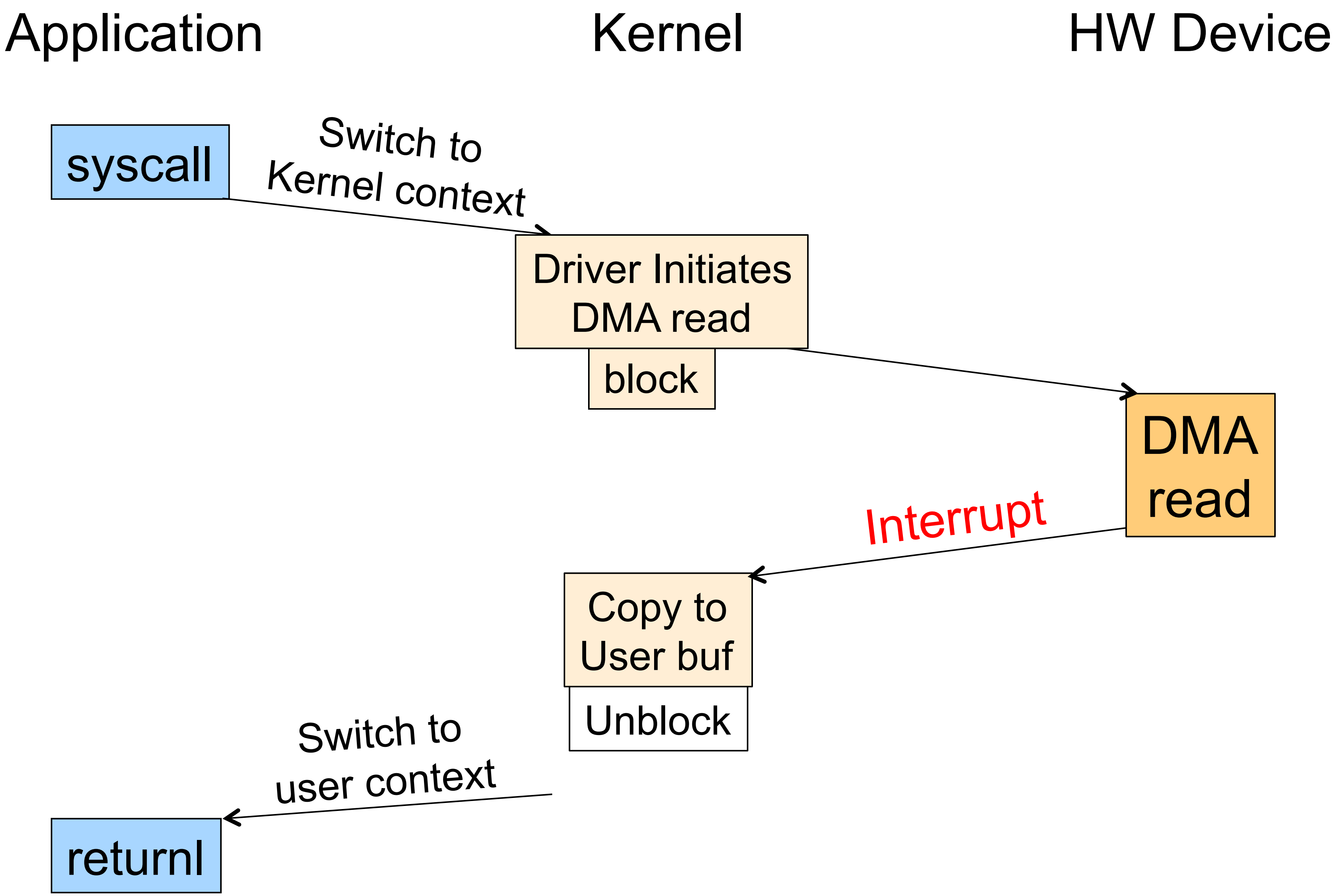
Unix Device Driver Entry Points

- `init()`
  - Initialize hardware
- `start()`
  - Boot time initialization (require system services)
- `open(dev, flag, id)` and `close(dev, flag, id)`
  - Initialization resources for read or write and release resources
- `halt()`
  - Call before the system is shutdown
- `intr(vector)`
  - Called by the kernel on a hardware interrupt
- `read(…)` and `write()` calls
  - Data transfer
- `poll(pri)`
  - Called by the kernel 25 to 100 times a second
- `ioctl(dev, cmd, arg, mode)`
  - special request processing

Synchronous vs. Asynchronous I/O

- Synchronous I/O
  - read() or write() will block a user process until its completion
  - OS overlaps synchronous I/O with another process
- Asynchronous I/O
  - read() or write() will not block a user process
  - Let user process do other things before I/O completion
  - I/O completion will notify the user process

Synchronous Read



Synchronous Read

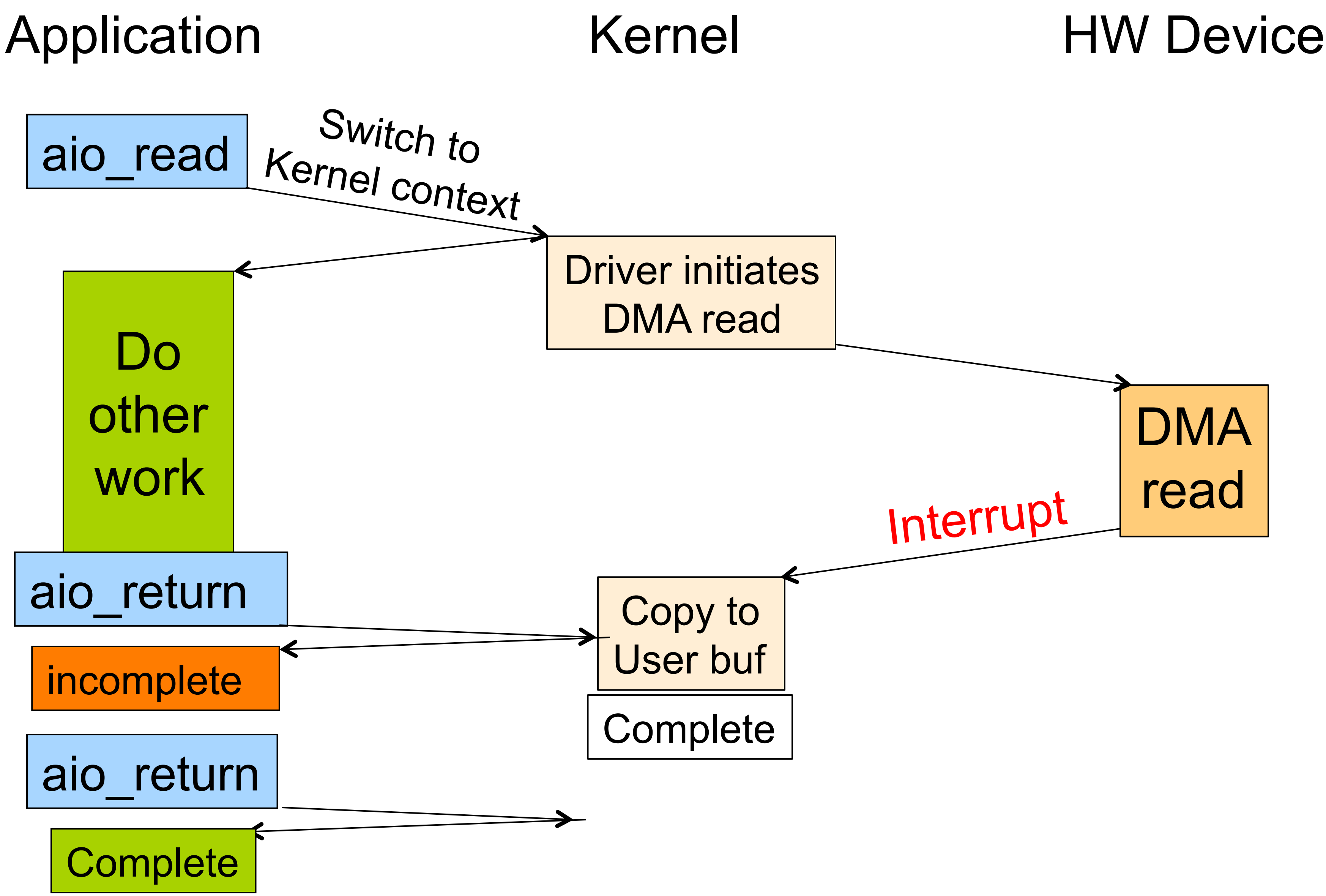
- A process issues a read call which executes a system call
- System call code checks for correctness and buffer cache
- If it needs to perform I/O, it will issues a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Initiate DMA read transfer
- Block the current process and schedule a ready process
- Device controller performs DMA read transfer
- Device sends an interrupt on completion
- Interrupt handler wakes up blocked process (make it ready)
- Move data from kernel buffer to user buffer
- System call returns to user code
- User process continues

Asynchronous I/O

POSIX P1003.4 Asynchronous I/O interface functions: (available in Solaris, AIX, Tru64 Unix, Linux 2.6,…)

- aio_cancel: cancel asynchronous read/write requests
- aio_error: retrieve Asynchronous I/O error status
- aio_fsync: asynchronously force I/O completion, and sets errno to ENOSYS
- aio_read: begin asynchronous read
- aio_return: retrieve status of Asynchronous I/O operation
- aio_suspend: suspend until Asynchronous I/O completes
- aio_write: begin asynchronous write
- lio_listio: issue list of I/O requests

Asynchronous Read



Why Buffering in Kernel?

- Speed mismatch between the producer and consumer
  - Character device and block device, for example
  - Adapt different data transfer sizes (packets vs. streams)
- DMA requires contiguous physical memory
  - I/O devices see physical memory
  - User programs use virtual memory
- Spooling
  - Avoid deadlock problems
- Caching
  - Serve for same requests of the same data
  - Reduce I/O operations

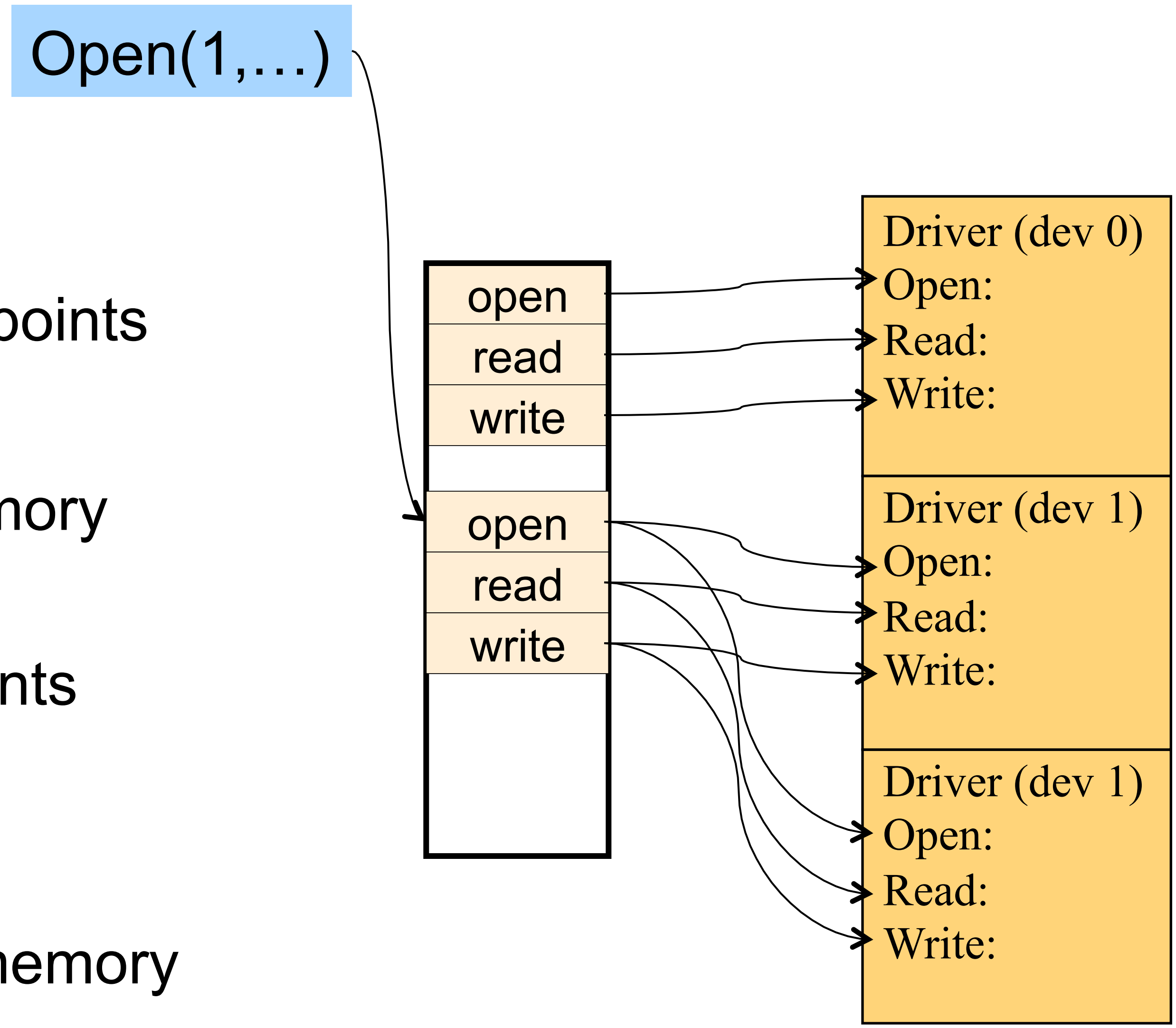
Design Issues

- Statically install device drivers
  - Reboot OS to install a new device driver

- Dynamically download device drivers
  - No reboot, but use an indirection
  - Load drivers into kernel memory
  - Install entry points and maintain related data structures
  - Initialize the device drivers

Dynamic Binding of Device Drivers

- Indirection
  - Indirect table for all device driver entry points
- Download a driver
  - Allocate kernel memory
  - Store driver code
  - Link up all entry points
- Delete a driver
  - Unlink entry points
  - Deallocate kernel memory



Issues with Device Drivers

- Flexible for users, ISVs and IHVs
  - Users can download and install device drivers
  - Vendors can work with open hardware platforms
- Dangerous
  - Device drivers run in kernel mode
  - Bad device drivers can cause kernel crashes and introduce security holes
- Progress on making device driver more secure
  - Checking device driver codes
  - Build state machines for device drivers

Summary

- IO Devices
  - Programmed I/O is simple but inefficient
  - Interrupt mechanism supports overlap CPU with I/O
  - DMA is efficient, but requires sophisticated software
- Device drivers
  - Dominate the code size of OS
  - Dynamic binding is desirable for many devices
  - Device drivers can introduce security holes
  - Progress on secure code for device drivers but completely removing device driver security is still an open problem
- Asynchronous I/O
  - Asynchronous I/O allows user code to perform overlapping