I/O and Device Drivers

Peripheral devices

- Attached to a bus
- Peripherals are built to perform certain specific commands
- Signals on the bus from the peripheral indicate the result

Device Performance

- Most devices slow compared to the CPU, bus, and RAM
- Leads to challenges with event completion
- System must operate at CPU speeds, not blocked by device speeds

Peripheral Device Code on OS

- Only one process can handle a peripheral device at a time
- The order in which events occur is critical for correctness
- Some of them are shared between multiple processes or are security sensitive
- Code on the OS better to manage this

Device Driver

- Application and the hardware with the OS between
- When the app sends a message, the OS invokes the proper device driver
- Detailed instructions fed to the hardware

Device Driver Code

- Generally the code is specific to each device (code that drives the device)
- Each system device represented by its own piece of code

Properties of Device Drivers

- Highly specific to the particular device
- Inherently modular
- Usually interacts with the rest of the system in limited but well defined ways
- Correctness is important
- Written by programmers who understand the device

Abstractions and Device Drivers

- OS defines idealized device classes
    - Flash drive, display, printer, network
- Classes define expected interfaces/behavior
    - All drivers support standard methods
- Device drivers implement standard behavior
    - Makes different devices fit into a standard model
    - Abstracts the details of each device

Abstractions

- Abstracts the knowledge of how to use the device
    - Maps standard device operations in operations on device

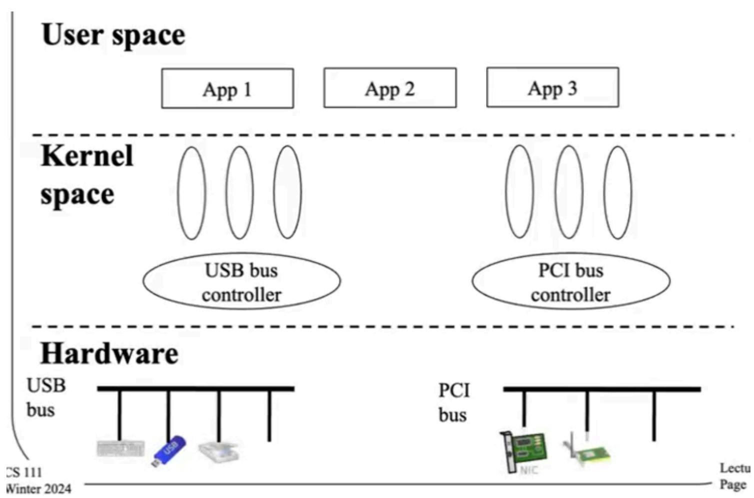
  - Hides irrelevant behaviors
- Abstracts knowledge of optimization
- Abstracts fault handling
  - Understands how to handle faults and prevents device faults form becoming OS faults

Device Drivers in a Modern OS

- Independent
- Pluggable model is typical
- OS provides capabilities to plug in particular drivers in well defined ways

Layering Device Drivers

- Interactions with the bus are standardized
  - Very specific ways to send signals through the bus
- Interactions with the app are standardized
  - File oriented approach



- App wants to rad from flash drive
- Does through system calls and the bus controller software
- Signal moves into hardware bus

Device Driver vs OS Code

- Common functionality belongs in the OS
  - Caching
  - File system code not tied to a specific device

- Specialized functionality goes in the drivers
  - Things that differ in different pieces of hardware
  - Things that only pertain to the particular piece of hardware

Devices and Interrupts

- Devices are primarily interrupt driven
    - Drives aren't processes and aren't scheduled
- Devices slower than the CPU
- They can do their own work while the CPU works on something else
- Devices use interrupts to get the CPU's attention

Devices and Busses

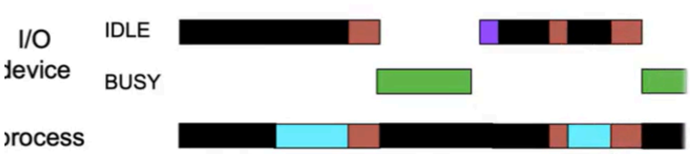
- Devices aren't connected directly to the CPU
- Both CPU and devices are connected to a bus (same or different)
- Bus used both to send and receive interrupts to CPU

CPUS and Interrupts

- Interrupts similar to traps but traps come from the CPU and interrupts are from external devices
- Interrupts can be enabled or disabled by special CPU instructions
    - Devices can be told when they can generate interrupts
    - Interrupt may be held pending until the software is ready

Device Performance

- Importance for good device utilization
- If device is idle, its throughput drops
- Delays can disrupt real time data flows
    - Results in loss of data and unacceptable performance
- Very important to keep key devices busy
    - CPU must not be held up waiting for devices



1. process waits to run
2. process does computation in preparation for I/O operation
3. process issues read system call, blocks awaiting completion
4. device performs requested operation
5. completion interrupt awakens blocked process
6. process runs again, finishes read system call
7. process does more computation
8. Process issues read system call, blocks awaiting completion

Improving Performance

- Exploit parallelism since devices operate independently of the CPU
- Device and CPU can operate in parallel

CPU operations

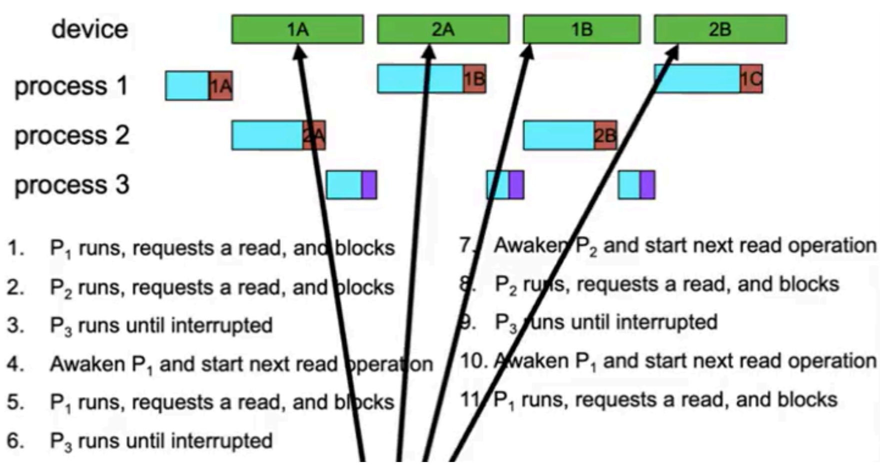
- Modern CPUs avoid going into RAM by using registers or caching
- Therefore, CPU doesn't use memory bus that much
- One way to parallelize activities is to let a device use the bus instead of the CPU

Direct Memory Access (DMA)

- Allows 2 devices attached to the bus to move data directly
- Bus can only be used for one thing at a time
- If bus is doing DMA, it's not servicing CPU requests
- With DMA, data moves from device to memory at bus/device.memory speed
- DMA allows the device to access RAM without the CPU

Keeping Devices Busy

- Multiple requests to be pending at a time
    - Queue them similar to a ready queue
- Use DMA to perform the actual data transfers
- When the current active request completes
    - Device controller generates a complete interrupt



- Driver runs continuously, and awakens processes when ready
- Only works with DMA

Large Transfers

- Each transfer has operation related overhead
    - DMA related, device related, OS related
    - Instructions to set up operation
    - Device time to start new operation
    - Time and cycles to service completion interrupt

- Large transfers have lower overhead/byte
    - Not limited to software implementations

I/O and Buffering

- Most I/O requests cause data to come into the memory or be copied to a device
- Data requires a place in memory called buffer
- Data in a buffer is ready to send to a device
- An existing buffer is ready to retrieve data from a device
- OS needs to make sure buffer available when devices need them

OS Buffering Issues

- Fewer/larger transfers are more efficient
  - However, record styles tend to be small
- Operating system can consolidate I/O requests
  - Maintain a cache of recently used disk blocks
  - Accumulate small writes, then read whole blocks and deliver data as requested
- Read ahead
  - OS reads blocks that aren't requested yet
  - OS moves data for the next blocks into buffer so the data is ready

Deep Request Queues

- <u>Having many I/O operation queued is good</u>
  - Maintains high device utilization
  - Reduces mean seek distance/rotational delay for disks
  - May be possible to combine adjacent requests
  - Can sometimes avoid performing a write at all

- Ways to achieve deep queues
  - Many processes or threads make requests
  - Individual processes make parallel requests
  - Read ahead do expected data requests
  - Write back cache is flushing

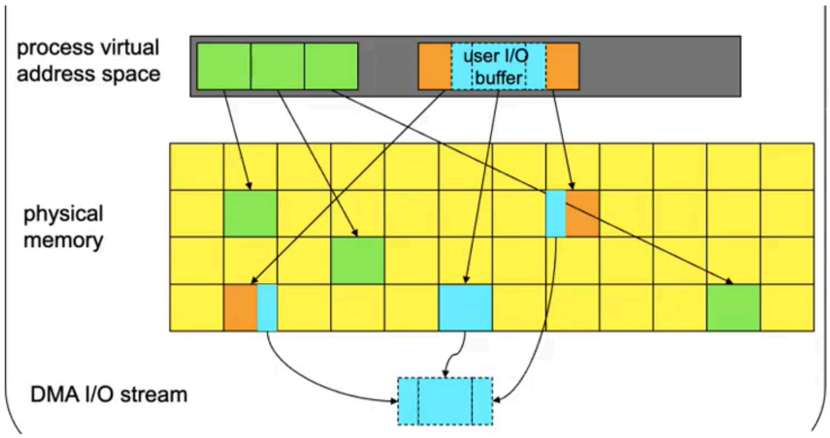
Scatter/Gather I/O

- Many device controllers support DMA transfers

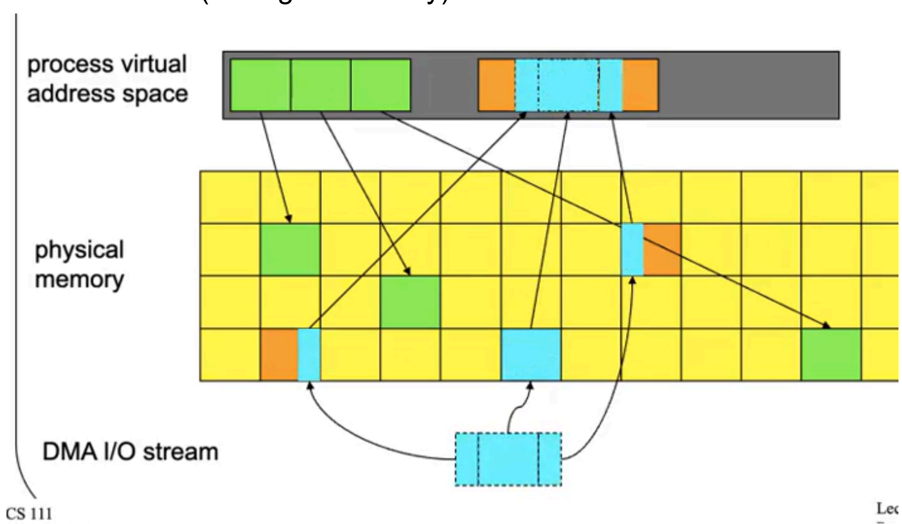
- User buffers are in paged VM, so buffers are spread over physical memory
  - Scatter: read from device to multiple page frames
  - Gather: writing from multiple page frames to device

- Basic approaches
  - Copy all user data into physically contiguous bugger
  - Split logical request into chain scheduled page requests
  - I/O MMU could automatically scatter/gather

Gather (Reading from memory)



Scatter (writing to memory)



Memory Mapped I/O

- DMA may not always be the best way to do I/O
  - Designed for large contiguous transfers
- Treat registers and memory as part of the same address space
  - Accessed through reads and writes to those locations
- Low overhead per update with no interrupts to service and easy to program

DMA vs Memory Mapping

- DMA performs large transfers efficiently
  - Better usage of both the device and CPU
  - Considerable transfer overhead
  - DMA better for occasional large transfers

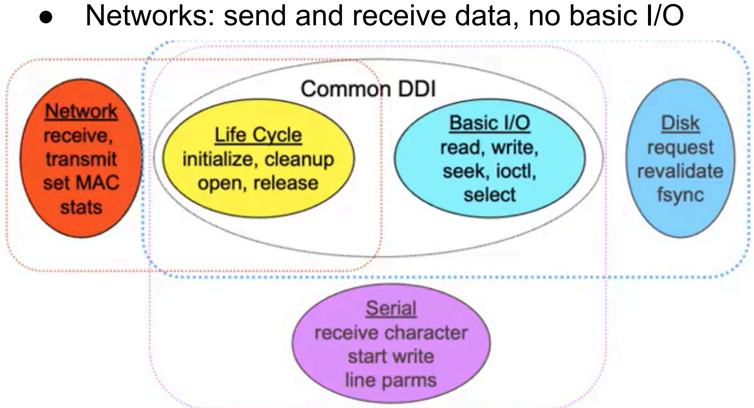
- Memory mapped I/O has no per operation overhead
    - However, every byte is transferred by CPU instruction
    - No waiting because device accepts data at memory speed
    - Memory mapped better for frequent small transfers
    - Memory-mapped devices are more difficult to share

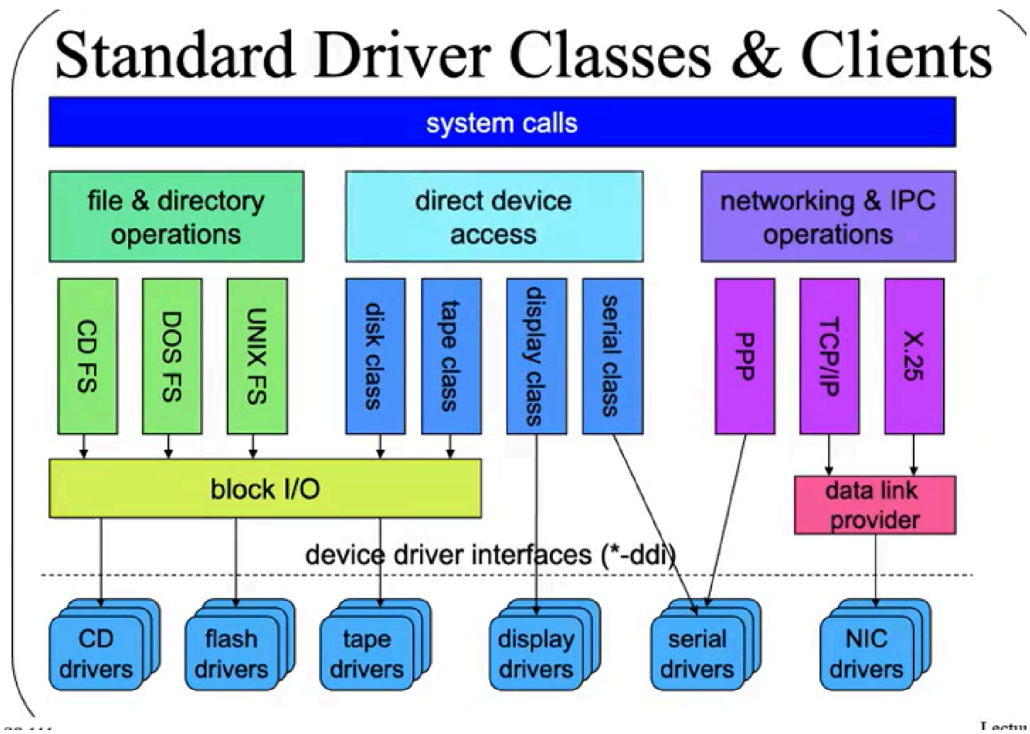
Generalizing Abstractions for Device Drivers
- Every device type is unique in hardware details
- In classes of devices, there are similarities
    - Flash drives, network cards

Providing abstractions
- OS defines device classes
- Classes defined the expected interfaces and behaviors
- Device drivers implement standard behavior
    - Make diverse drivers fit into common model
- Interfaces are key to providing abstractions

Device Driver Interface (DDI)
- Standard top-end device driver entry points
    - Top-end from the OS to the driver
    - Enables system to exploit new devices
    - Critical interface for 3rd party device makers
- Entry points correspond directly to sys calls
- Some are associated with OS frameworks
    - Ex. All flash drives meant to be called by block I/O

DDI and sub-DDI
- Life cycle: life cycle of managing the device
- Basic I/O: basic operations done using the device
- Disk: if the device needs disk access
- Networks: send and receive data, no basic I/O



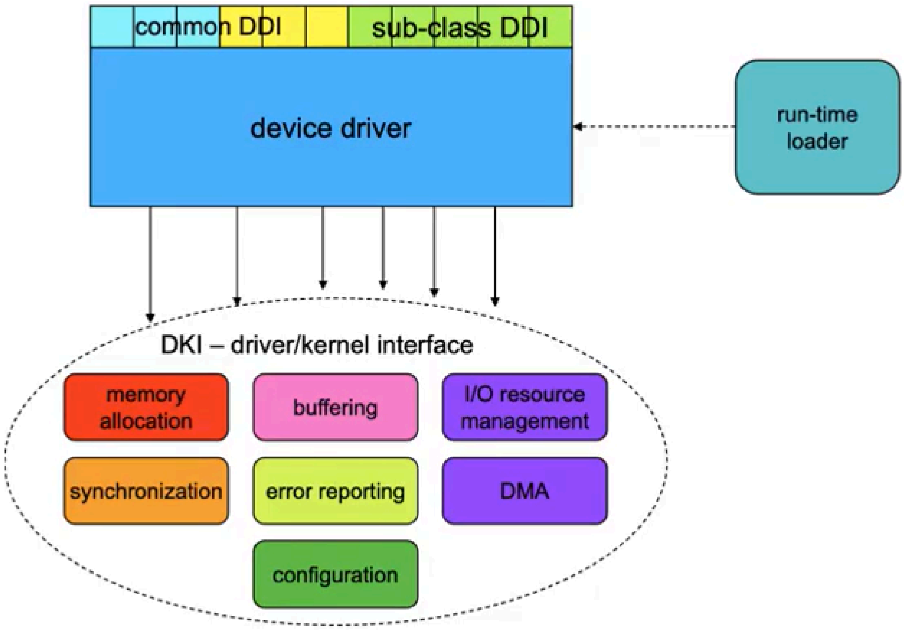


Simplifying Abstrations

- Encapsulate knowledge of how to use device
    - Maps standard operations to device specific operations
    - Map device states into standard object behavior
- Encapsulates optimizations and prevent fault handling from going to OS

Driver/Kernel Interface

- Bottom end services OS provides to drivers
    - Operations driver can ask the kernel to do
- Well defined and stable
    - Allows 3rd party driver writers to build drivers
    - Old drivers continue to work on new OS versions
- Each OS has it own DKI
    - Memory allocation, data transfer, I/O, interrupts, DMA, synchronization, error reporting



DKI: Kernel Services for Device Drivers

- Device drivers need OS-assistance
- Ex. Allocating memory, locking on the device
- Runtime loader: device that is inert but it usable once plugged in, activating the runtime loader to get the device driver

    Stable Interfaces

    - Drivers are largely independent from the OS
        - Built but different companies and might not have been packaged with the OS
    - OS and drivers have interface dependencies
        - OS depends on driver implementations of DDI
        - Drivers depend on kernel DKI implementations
    - Well defined and well tested

Linux Device Driver Abstractions

- Class based system
- Several super classes
    - Character devices (keyboards)
    - Block devices (moving blocks of data, requiring caching)

    Character Device Superclass

    - Devices that read/write one byte at a time
    - Either stream or record structured
    - May be sequential or random access
    - Support direct, sync reads/writes

- Ex. Keyboards, monitors

Block Device Superclass
- Devices that deal with a fixed block of data at a time
- Ex. disk drive, reads a single 4K block
- Random access devices, accessible one block at a time
- Support queued async reads and writes

Separate Superclass for Block
- Block devices span all forms of block addressable storage
- Such devices require buffer allocation, LRU buffer, data copying services
- Important system functionality (file systems and swapping and paging)
- Block I/O designed to provide high performance for critical functions
- Many important functions such as demand paging are handled by the OS

Network Device Superclass
- Devices the send and receive data packets
- Used in network protocols

Identifying Device Drivers
- Major device number specifies which device driver to use for it
- Might have several distinct devices using the same drivers
    - Multiple disk drives of the same type
    - Each disk drive can use the same device driver code
- Opening a special file opens the associated device
    - Open/close/read/write calls map to calls to appropriate entry points of the selected driver