Operating System Principles: Devices, Device Drivers, and I/O
CS 111
Operating Systems
Peter Reiher
Outline

• Devices and device drivers
• I/O performance issues
• Device driver abstractions
So You’ve Got Your Computer . . .

It’s got memory, a bus, a CPU or two

But there’s usually a lot more to it than that

And who knows what else?
Welcome to the Wonderful World of Peripheral Devices!

• Our computers typically have lots of devices attached to them

• Each device needs to have some code associated with it
  – To perform whatever operations it does
  – To integrate it with the rest of the system

• In modern commodity OSes, the code that handles these devices dwarfs the rest
Peripheral Device Code and the OS

- Why are peripheral devices the OS’ problem, anyway?
- Why can’t they be handled in user-level code?
- Maybe they sometimes can, but . . .
- Some of them are critical for system correctness
  - E.g., the disk drive holding swap space
- Some of them must be shared among multiple processes
  - Which is often rather complex
- Some of them are security-sensitive
- Perhaps more appropriate to put the code in the OS
Where the Device Driver Fits in

• At one end you have an application
  – Like a web browser

• At the other end you have a very specific piece of hardware
  – Like an Intel Gigabit CT PCI-E Network Adapter

• In between is the OS

• When the application sends a packet, the OS needs to invoke the proper device driver

• Which feeds detailed instructions to the hardware
Device Drivers

• Generally, the code for these devices is pretty specific to them
• It’s basically code that *drives* the device
  – Makes the device perform the operations it’s designed for
• So typically each system device is represented by its own piece of code
• The *device driver*
• A Linux 2.6 kernel came with over 3200 of them . . .
Typical Properties of Device Drivers

- Highly specific to the particular device
  - System only needs drivers for devices it hosts
- Inherently modular
- Usually interacts with the rest of the system in limited, well defined ways
- Their correctness is critical
  - Device behavior correctness and overall correctness
- Generally written by programmers who understand the device well
  - But are not necessarily experts on systems issues
Abstractions and Device Drivers

• OS defines idealized device classes
  – Disk, display, printer, tape, network, serial ports

• Classes define expected interfaces/behavior
  – All drivers in class support standard methods

• Device drivers implement standard behavior
  – Make diverse devices fit into a common mold
  – Protect applications from device eccentricities

• Abstractions regularize and simplify the chaos of the world of devices
What Can Driver Abstractions Help With?

• Encapsulate knowledge of how to use the device
  – Map standard operations into operations on device
  – Map device states into standard object behavior
  – Hide irrelevant behavior from users
  – Correctly coordinate device and application behavior

• Encapsulate knowledge of optimization
  – Efficiently perform standard operations on a device

• Encapsulate fault handling
  – Understanding how to handle recoverable faults
  – Prevent device faults from becoming OS faults
How Do Device Drivers Fit Into a Modern OS?

• There may be a lot of them
• They are each pretty independent
• You may need to add new ones later
• So a pluggable model is typical
• OS provides capabilities to plug in particular drivers in well defined ways
• Then plug in the ones a given machine needs
• Making it easy to change or augment later
Layering Device Drivers

- The interactions with the bus, down at the bottom, are pretty standard
  - How you address devices on the bus, coordination of signaling and data transfers, etc.
  - Not too dependent on the device itself
- The interactions with the applications, up at the top, are also pretty standard
  - Typically using some file-oriented approach
- In between are some very device specific things
A Pictorial View

User space

System Call

App 1

App 2

App 3

Kernel space

Device Call

Device Drivers

USB bus controller

PCI bus controller

Hardware

USB bus

PCI bus

NIC
Device Drivers Vs. Core OS Code

• Device driver code is in the OS, but . . .
• What belongs in core OS vs. a device driver?
• Common functionality belongs in the OS
  – Caching
  – File systems code not tied to a specific device
  – Network protocols above physical/link layers
• Specialized functionality belongs 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
  – Drivers aren’t schedulable processes
• They work at different speed than the CPU
  – Typically slower
• They can do their own work while CPU does something else
• They use interrupts to get the CPU’s attention
Devices and Busses

- Devices are not connected directly to the CPU
- Both CPU and devices are connected to a bus
- Sometimes the same bus, sometimes a different bus
- Devices communicate with CPU across the bus
- Bus used both to send/receive interrupts and to transfer data and commands
  - Devices signal controller when they are done/ready
  - When device finishes, controller puts interrupt on bus
  - Bus then transfers interrupt to the CPU
  - Perhaps leading to movement of data
CPUs and Interrupts

• Interrupts look very much like traps
  – Traps come from CPU
  – Interrupts are caused externally to CPU
• Unlike traps, interrupts can be enabled/disabled by special CPU instructions
  – Device can be told when they may generate interrupts
  – Interrupt may be held *pending* until software is ready for it
The Changing I/O Landscape

• To quote a recent Nobel Prize winner, “the times they are a’changing”

• Storage paradigms
  – Old: swapping, paging, file systems, data bases
  – New: NAS, distributed object/key-value stores

• I/O traffic
  – Old: most I/O was disk I/O
  – New: network and video dominate many systems

• Performance goals:
  – Old: maximize throughput, IOPS
  – New: low latency, scalability, reliability, availability
Device Performance

- The importance of good device utilization
- How to achieve good utilization
Good Device Utilization

• Key system devices limit system performance
  – File system I/O, swapping, network communication

• If device sits idle, its throughput drops
  – This may result in lower system throughput
  – Longer service queues, slower response times

• Delays can disrupt real-time data flows
  – Resulting in unacceptable performance
  – Possible loss of irreplaceable data

• It is very important to keep key devices busy
  – Start request $n+1$ immediately when $n$ finishes
Poor I/O Device Utilization

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
How To Do Better

• The usual way:
  – Exploit parallelism
• Devices operate independently of the CPU
• So a device and the CPU can operate in parallel
• But often devices need to access RAM
  – As does the CPU
• How to handle that?
What’s Really Happening on the CPU?

• Modern CPUs try to avoid going to RAM
  – Working with registers
  – Caching on the CPU chip itself

• If things go well, the CPU doesn’t use the memory bus that much
  – If not, life will be slow, anyway

• So one way to parallelize activities is to let a device use the bus instead of the CPU
Direct Memory Access (DMA)

- Allows any two devices attached to the memory bus to move data directly
  - Without passing it through the CPU first
- Bus can only be used for one thing at a time
- So if it’s doing DMA, it’s not servicing CPU requests
- But often the CPU doesn’t need it, anyway
- With DMA, data moves from device to memory at bus/device/memory speed
Keeping Key Devices Busy

• Allow multiple requests to be pending at a time
  – Queue them, just like processes in the ready queue
  – Requesters block to await eventual completions

• Use DMA to perform the actual data transfers
  – Data transferred, with no delay, at device speed
  – Minimal overhead imposed on CPU

• When the currently active request completes
  – Device controller generates a completion interrupt
  – OS accepts interrupt and calls appropriate handler
  – Interrupt handler posts completion to requester
  – **Interrupt handler selects and initiates next transfer**
xx_read/write() {
    allocate a new request descriptor
    fill in type, address, count, location
    insert request into service queue
    if (device is idle) {
        disable_device_interrupt();
        xx_start();
        enable_device_interrupt();
    } 
    await completion of request
    extract completion info for caller
}

xx_start() {
    get next request from queue
    get address, count, disk address
    load request parameters into controller
    start the DMA operation
    mark device busy
}

xx_intr() {
    extract completion info from controller
    update completion info in current req
    wakeup current request
    if (more requests in queue)
        xx_start()
    else 
        mark device idle
}
Multi-Tasking & Interrupt Driven I/O

device

process 1

process 2

process 3

1. $P_1$ runs, requests a read, and blocks
2. $P_2$ runs, requests a read, and blocks
3. $P_3$ runs until interrupted
4. Awaken $P_1$ and start next read operation
5. $P_1$ runs, requests a read, and blocks
6. $P_3$ runs until interrupted
7. Awaken $P_2$ and start next read operation
8. $P_2$ runs, requests a read, and blocks
9. $P_3$ runs until interrupted
10. Awaken $P_1$ and start next read operation
11. $P_1$ runs, requests a read, and blocks
Bigger Transfers are Better

![Graph showing PCIe throughput vs transfer size. The graph compares READ (PCIe 3.0 x8) and WRITE (PCIe 3.0 x8) operations. The throughput increases with transfer size, with READ operations generally reaching higher throughput than WRITE operations.](image-url)
(Bigger Transfers are Better)

• Disks have high seek/rotation overheads
  – Larger transfers amortize down the cost/byte
• All transfers have per-operation overhead
  – Instructions to set up operation
  – Device time to start new operation
  – Time and cycles to service completion interrupt
• Larger transfers have lower overhead/byte
  – This is not limited to software implementations
I/O and Buffering

• Most I/O requests cause data to come into the memory or to be copied to a device
• That data requires a place in memory
  – Commonly called a buffer
• Data in buffers is ready to send to a device
• An existing empty buffer is ready to receive data from a device
• OS needs to make sure buffers are available when devices are ready to use them
OS Buffering Issues

• Fewer/larger transfers are more efficient
  – They may not be convenient for applications
  – Natural record sizes tend to be relatively small

• Operating system can consolidate I/O requests
  – Maintain a cache of recently used disk blocks
  – Accumulate small writes, flush out as blocks fill
  – Read whole blocks, deliver data as requested

• Enables read-ahead
  – OS reads/caches blocks not yet requested
Deep Request Queues

• Having many I/O operations queued is good
  – Maintains high device utilization (little idle time)
  – Reduces mean seek distance/rotational delay
  – May be possible to combine adjacent requests
  – Can sometimes avoid performing a write at all

• Ways to achieve deep queues:
  – Many processes making requests
  – Individual processes making parallel requests
  – Read-ahead for expected data requests
  – Write-back cache flushing
Double-Buffered Output

application

buffer #1

buffer #2

device
Performing Double-Buffered Output

• Have multiple buffers queued up, ready to write
  – Each write completion interrupt starts the next write
• Application and device I/O proceed in parallel
  – Application queues successive writes
    • Don’t bother waiting for previous operation to finish
  – Device picks up next buffer as soon as it is ready
• If we're CPU-bound (more CPU than output)
  – Application speeds up because it doesn’t wait for I/O
• If we're I/O-bound (more output than CPU)
  – Device is kept busy, which improves throughput
  – But eventually we may have to block the process
Double-Buffered Input

- Application
- Buffer #1
- Buffer #2
- Device
Performing Double Buffered Input

• Have multiple reads queued up, ready to go
  – Read completion interrupt starts read into next buffer

• Filled buffers wait until application asks for them
  – Application doesn't have to wait for data to be read

• When can we do chain-scheduled reads?
  – Each app will probably block until its read completes
    • So we won’t get multiple reads from one application
  – We can queue reads from multiple processes
  – We can do predictive read-ahead
Scatter/Gather I/O

• Many device controllers support DMA transfers
  – Entire transfer must be contiguous in physical memory
• User buffers are in paged virtual memory
  – User buffers may be spread all over physical memory
  – *Scatter*: read from device to multiple pages
  – *Gather*: writing from multiple pages to device
• Three basic approaches apply
  1. Copy all user data into contiguous physical buffer
  2. Split logical request into chain-scheduled page requests
  3. I/O MMU may automatically handle scatter/gather
“Gather” Writes From Paged Memory

- process virtual address space
- physical memory
- DMA I/O stream
- user I/O buffer
"Scatter" Reads Into Paged Memory

process virtual address space

physical memory

DMA I/O stream
Memory Mapped I/O

• DMA may not be the best way to do I/O
  – Designed for large contiguous transfers
  – Some devices have many small sparse transfers
    • E.g., consider a video game display adaptor

• Instead, treat registers/memory in device as part of the regular memory space
  – Accessed by reading/writing those locations

• For example, a bit-mapped display adaptor
  – 1Mpixel display controller, on the CPU memory bus
  – Each word of memory corresponds to one pixel
  – Application uses ordinary stores to update display

• Low overhead per update, no interrupts to service

• Relatively easy to program
Trade-off: Memory Mapping vs. DMA

- **DMA performs large transfers efficiently**
  - Better utilization of both the devices and the CPU
    - Device doesn't have to wait for CPU to do transfers
  - But there is considerable per transfer overhead
    - Setting up the operation, processing completion interrupt

- **Memory-mapped I/O has no per-op overhead**
  - But every byte is transferred by a CPU instruction
    - No waiting because device accepts data at memory speed

- **DMA better for occasional large transfers**
- **Memory-mapped better frequent small transfers**
- **Memory-mapped devices more difficult to share**
Generalizing Abstractions for Device Drivers

- Every device type is unique
  - To some extent, at least in hardware details
- Implying each requires its own unique device driver
- But there are many commonalities
- Particularly among classes of devices
  - All disk drives, all network cards, all graphics cards, etc.
- Can we simplify the OS by leveraging these commonalities?
- By defining simplifying abstractions?
Providing the Abstractions

- The OS defines idealized device classes
  - Disk, display, printer, tape, network, serial ports
- Classes define expected interfaces/behavior
  - All drivers in class support standard methods
- Device drivers implement standard behavior
  - Make diverse devices fit into a common mold
  - Protect applications from device eccentricities
- Interfaces (as usual) are key to providing abstractions
Device Driver Interface (DDI)

- Standard (top-end) device driver entry-points
  - “Top-end” – from the OS to the driver
  - Basis for device-independent applications
  - Enables system to exploit new devices
  - A critical interface contract for 3rd party developers

- Some entry points correspond directly to system calls
  - E.g., open, close, read, write

- Some are associated with OS frameworks
  - Disk drivers are meant to be called by block I/O
  - Network drivers are meant to be called by protocols
DDIs and sub-DDIs

Network
receive, transmit
set MAC stats

Life Cycle
initialize, cleanup
open, release

Basic I/O
read, write,
seek, ioctl,
select

Disk
request
revalidate
fsync

Serial
receive character
start write
line parms

Common DDI
Standard Driver Classes & Clients

- file & directory operations
- direct device access
- networking & IPC operations

- block I/O

- device driver interfaces (*-ddi)

- CD drivers
- disk drivers
- tape drivers
- display drivers
- serial drivers
- NIC drivers

- data Link provider

- CD FS
- DOS FS
- UNIX FS

- disk class
- tape class
- display class
- serial class

- system calls

- TCP/IP
- X.25
- PPP
- X.25
- data Link provider

- block I/O

- device driver interfaces (*-ddi)
Drivers – Simplifying Abstractions

• Encapsulate knowledge of how to use a device
  – Map standard operations into operations onto device
  – Map device states into standard object behavior
  – Hide irrelevant behavior from users
  – Correctly coordinate device and application behavior

• Encapsulate knowledge of optimization
  – Efficiently perform standard operations on a device

• Encapsulation of fault handling
  – Knowledge of how to handle recoverable faults
  – Prevent device faults from becoming OS faults
Kernel Services for device drivers

- Common DDI
- Sub-class DDI

Device driver

- Memory allocation
- Buffering
- Error reporting
- I/O resource management
- DMA
- Configuration

DKI - driver/kernel interface

Run-time loader
Driver/Kernel Interface

• Specifies bottom-end services OS provides to drivers
  – Things drivers can ask the kernel to do
  – Analogous to an ABI for device driver writers
• Must be very well-defined and stable
  – To enable 3rd party driver writers to build drivers
  – So old drivers continue to work on new OS versions
• Each OS has its own DKI, but they are all similar
  – Memory allocation, data transfer and buffering
  – I/O resource (e.g. ports, interrupts) mgt, DMA
  – Synchronization, error reporting
  – Dynamic module support, configuration, plumbing
Criticality of Stable Interfaces

• Drivers are largely independent from the OS
  – They are built by different organizations
  – They might not be co-packaged with the OS

• OS and drivers have interface dependencies
  – OS depends on driver implementations of DDI
  – Drivers depends on kernel DKI implementations

• These interfaces must be carefully managed
  – Well defined and well tested
  – Upwards-compatible evolution
Linux Device Driver Abstractions

• An example of how an OS handles device drivers
• Basically inherited from earlier Unix systems
• A class-based system
• Several super-classes
  – Block devices
  – Character devices
  – Some regard network devices as a third major class
• Other divisions within each super-class
Why Classes of Drivers?

• Classes provide a good organization for abstraction
• They provide a common framework to reduce amount of code required for each new device
• The framework ensure all devices in class provide certain minimal functionality
• But a lot of driver functionality is very specific to the device
  – Implying that class abstractions don’t cover everything
Character Device Superclass

• Devices that read/write one byte at a time
  – “Character” means byte, not ASCII
• May be either stream or record structured
• May be sequential or random access
• Support direct, synchronous reads and writes
• Common examples:
  – Keyboards
  – Monitors
  – Most other devices
Block Device Superclass

• Devices that deal with a block of data at a time
• Usually a fixed size block
• Most common example is a disk drive
• Reads or writes a single sized block (e.g., 4K bytes) of data at a time
• Random access devices, accessible one block at a time
• Support queued, asynchronous reads and writes
Why a Separate Superclass for Block Devices?

• Block devices span all forms of block-addressable random access storage
  – Hard disks, CDs, flash, and even some tapes

• Such devices require some very elaborate services
  – Buffer allocation, LRU management of a buffer cache, data copying services for those buffers, scheduled I/O, asynchronous completion, etc.

• Important system functionality (file systems and swapping/paging) implemented on top of block I/O

• Block I/O services are designed to provide very high performance for critical functions
Network Device Superclass

• Devices that send/receive data in packets
• Originally treated as character devices
• But sufficiently different from other character devices that some regard as distinct
• Only used in the context of network protocols
  – Unlike other devices
  – Which leads to special characteristics
• Typical examples are Ethernet cards, 802.11 cards, Bluetooth devices
Identifying Device Drivers

• The major device number specifies which device driver to use for it

• Might have several distinct devices using the same drivers
  – E.g., multiple disk drives of the same type
  – Or one disk drive divided into logically distinct pieces

• Minor device number distinguishes between those
Accessing Linux Device Drivers

- Done through the file system
- Special files
  - Files that are associated with a device instance
  - UNIX/LINUX uses <block/character, major, minor>
    - Major number corresponds to a particular device driver
    - Minor number identifies an instance under that driver
      - Opening a special file opens the associated device
        - Open/close/read/write/etc. calls map to calls to appropriate entry-points of the selected driver