I/O Devices, Controllers, and Drivers

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 18
https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D Ch 11
Recall: What might the OS do on a page fault?

• If the access is right below the stack...
  • OS might allocate a new stack page and retry the instruction

• If the access is a write to a page after fork()...
  • OS might copy the page, mark as writable, and retry the instruction

• If the access is one that the process has no good reason to make...
  • OS typically terminates the process (segmentation fault)
  • (e.g., for page marked kernel only)

• If access is to a page whose contents are in secondary storage...
  • OS brings in page from secondary storage to memory (demand paging)
Recall: Demand Paging

• Modern programs require a lot of physical memory
  • Memory per system growing faster than 25%-30%/year
• But they don’t use all their memory all of the time
  • 90-10 rule: programs spend 90% of their time in 10% of their code
  • Wasteful to require all of user’s code to be in memory
• Solution: use main memory as “cache” for disk
Recall: Page Replacement Policies

• Why do we care about Replacement Policy?
  • Replacement is an issue with any cache, but particularly important with pages
    • The cost of being wrong is high: must go to disk
    • Must keep important pages in memory, not toss them out

• FIFO (First In, First Out)
  • Throw out oldest page. Be fair – let every page live in memory for about the same amount of time.
  • Bad – throws out heavily used pages instead of infrequently used

• RANDOM
  • Pick random page for every replacement
  • Typical solution for TLB’s. Simple hardware
  • Pretty unpredictable – makes it hard to make real-time guarantees
Recall: MIN

• **MIN (Minimum):**
  • Replace page that won’t be used for the longest time
  • Great (provably optimal), but can’t really know future...
    • Clairvoyant algorithm
  • Also called Belady’s Algorithm of Belady’s Theoretically Optimal Paging

• *But past is a good predictor of the future ...*
Recall: LRU

• LRU (Least Recently Used):
  • Replace page that hasn’t been used for the longest time
  • Relies on temporal locality

• How to implement LRU? Use a list!

• Approximates MIN based on temporal locality
Recall: Working Set Model

- As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space.
Recall: Another Model of Locality, Zipf

Zipf: likelihood of accessing item of rank $r$ is $\propto \frac{1}{r^a}$

Although rare to access items below the top few, there are so many that it yields a “heavy tailed” distribution
  - Substantial value from even a tiny cache
  - Substantial misses from even a very large cache
Recall: Cache Behavior under Working Set Model

- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Applicable to memory caches and pages. Others?
Recall: Bélády’s Anomaly

• FIFO example:

• After adding memory:
  • Resident pages could be totally different
  • With LRU/MIN, always a superset

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recall: Problems with LRU

• Not optimal (to be expected)

• How to implement LRU?
  • Requires mutating linked list on every memory access
  • Trap to OS on every memory access?
    • Way too slow
  • Have hardware manipulate a linked list?
    • Too complex

• We will use hardware support to approximate LRU
Recall: What’s in a Page Table Entry (PTE)?

• What is in a Page Table Entry (or PTE)?
  • “Pointer to” (address of) next-level page table or to actual page
  • Permission bits: valid, read-only, read-write, write-only
• Example: Intel x86 architecture PTE:

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>P</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PC</th>
<th>PW</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

  P: Present (same as “valid” bit in other architectures)
  W: Writeable
  U: User accessible
  PWT: Page write transparent: external cache write-through
  PCD: Page cache disabled (page cannot be cached)
  A: Accessed: page has been accessed recently
  D: Dirty (PTE only): page has been modified recently
  L: L=1 ⇒ 4MB page (directory only). Bottom 22 bits of virtual address serve as offset
Approximating LRU: Clock Algorithm

• **Clock Algorithm (NRU):** Arrange physical pages in circle with single clock hand
  • Approximate LRU (*approximation to approximation to MIN*)
  • Replace *an old* page, not *the oldest* page

• **Details:**
  • Hardware sets “use” bit (“accessed” bit) in PTE on each reference
    • Some hardware sets use bit in the TLB, with writeback to PTE
  • **On page fault:**
    • Advance clock hand (not real time)
    • Check use bit: 1→used recently; clear use bit and continue advancing clock hand
    • 0→not used recently; choose this page for replacement
Discussion: Hardware Support

• Do we really need hardware support? No!
  • Can emulate “use” and “modified” bits by marking all pages invalid and trapping to OS
  • On use, set use bit and then mark page as “read-only”
  • On write, set use/modified bits and then mark page as “read-write”

• Given that, without hardware support, we have to take some extra page faults, is there a better approximation of LRU we can use?
  • Second-Chance List
Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page (FIFO) from end of Active list to front of SC list and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Second-Chance List Algorithm

• How many pages for second chance list?
  • If 0 ⇒ FIFO
  • If all ⇒ LRU, but page fault on every page reference

• Pick intermediate value. Result is:
  • Pro: Few disk accesses (page only goes to disk if unused for a long time)
  • Con: Increased overhead trapping to OS (software / hardware tradeoff)
Second-Chance List Algorithm

• Second-Chance List is very flexible, and useful even if you do have hardware support
  • Linux uses a variant of this
  • Principle: keep likely victims on a separate list
Free List

• “Pageout daemon” runs clock algorithm in background to keep a certain number of pages free
• Advantage: faster for page fault
  • Moves eviction algorithm outside of the critical path
 Allocation of Page Frames

• How do we allocate memory among different processes?
  • Does every process get the same fraction of memory? Different fractions?
  • Should we completely swap some processes out of memory?

• Each process needs *minimum* number of pages
  • Make sure that all processes that are loaded into memory can make forward progress
  • Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    • instruction is 6 bytes, might span 2 pages; 2 pages to handle *from*; 2 pages to handle *to*

• Possible Replacement Scopes:
  • Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  • Local replacement – each process selects from only its own set of allocated frames
Memory Sharing (Scheduling)

• **Equal allocation** (Fixed Scheme):
  • Every process gets same amount of memory
  • Example: 100 frames, 5 processes → process gets 20 frames

• **Proportional allocation** (Fixed Scheme)
  • Allocate according to the size of process
  • Computation proceeds as follows:
    \[ s_i = \text{size of process } p_i \text{ and } S = \sum s_i \]
    \[ m = \text{total number of frames} \]
    \[ a_i = \text{(allocation for } p_i) = \frac{s_i}{S} \times m \]

• **Priority Allocation**:
  • Proportional scheme using priorities rather than size
  • Same type of computation as previous scheme
  • Possible behavior: If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number

• Perhaps we should use an adaptive scheme instead???
  • What if some application just needs more memory?
Page-Fault Frequency Allocation

• Can we reduce capacity misses by dynamically changing the number of pages per application?

• Establish “acceptable” page-fault rate
  • If actual rate too low, process loses frame
  • If actual rate too high, process gains frame

• Question: What if we just don’t have enough memory?
Thrashing

• If a process has too few pages in memory, the page-fault rate is very high, leading to:
  • Low CPU utilization
  • OS spends most of its time swapping to disk
• Thrashing ≡ a process is busy swapping pages in and out
• Questions:
  • How do we detect Thrashing?
  • What is best response to Thrashing?
Locality in a Memory Reference Pattern

• Program Memory Access Patterns have temporal and spatial locality
  • Group of Pages accessed along a given time slice called the “Working Set”
  • Working Set defines minimum number of pages needed for process to behave well
• Not enough memory for Working Set $\Rightarrow$ Thrashing
  • Better to swap out process?
Using the Working Set to Inform Scheduling

- \( \Delta \equiv \) working-set window \( \equiv \) fixed number of page references
- \( W_{S_i} \) (working set of Process \( P_i \)) = total set of pages referenced in the most recent \( \Delta \) (varies in time)
  - if \( \Delta \) too small will not encompass entire locality
  - if \( \Delta \) too large will encompass several localities
  - if \( \Delta = \infty \Rightarrow \) will encompass entire program
- \( D = \sum |W_{S_i}| \equiv \) total demand frames
- if \( D > m \Rightarrow \) Thrashing
  - Policy: if \( D > m \), then suspend/swap out the entire process
  - This can improve overall system behavior by a lot!
What about Compulsory Misses?

• Recall that compulsory misses are misses that occur the first time that a page is seen
  • Pages that are touched for the first time
  • Pages that are touched after process is swapped out/swapped back in

• Clustering:
  • On a page-fault, bring in multiple pages “around” the faulting page
  • Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages

• Working Set Tracking:
  • Use algorithm to try to track working set of application
  • When swapping process back in, swap in working set
Recall: How does the OS know what to do?

• A page fault could mean a variety of things...

• OS keeps track of a memory map for each process

• OS needs to store additional info about each page to know what to do
  • Can use extra bits in the PTE
  • Typically, OS keeps additional information about pages in a data structure called the supplemental page table, which it consults on page faults
Reverse Page Mapping ("Coremap")

• When evicting a page frame, how to know which PTEs to invalidate?
  • Hard in the presence of shared pages (forked processes, shared memory, ...)

• Reverse mapping mechanism must be very fast
  • Must hunt down all PTEs pointing at given page frame when freeing a page
  • Must hunt down all PTEs when seeing if page is “active”

• Implementation options:
  • For every page descriptor, keep linked list of page table entries that point to it
    • Management nightmare – expensive
  • Linux 2.6: Object-based reverse mapping
    • Link together memory region descriptors instead (much coarser granularity)
Announcements

• Project 2 code is due on Tuesday
• Project 2 final report and Scheduling Lab are due on Wednesday

• Homework 4 is due on Monday
  • No coding portion
  • It’s the shortest homework, but not necessarily the easiest
  • Look at it TODAY!

• Reminder: Anonymous Feedback Form
  • http://bit.ly/cs162su20anon
Recall: Five Components of a Computer

Diagram from “Computer Organization and Design” by Patterson and Hennessy
Requirements of I/O

- So far in CS 162, we have studied:
  - Abstractions: the APIs provided by the OS to applications running in a process
  - Synchronization/Scheduling: How to manage the CPU

- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But... thousands of devices, each slightly different
    - How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    - How can we make them reliable???
  - Devices unpredictable and/or slow
    - How can we manage them if we don’t know what they will do or how they will perform?
Recall: OS Basics, I/O

- OS provides common services in the form of I/O
Recall: Range of Timescales

Jeff Dean: “Numbers Everyone Should Know”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
Example: Device Transfer Rates in Mb/s (Sun Enterprise 6000)

• Device rates vary over 12 orders of magnitude!!!
• System must be able to handle this wide range
  • Better not have high overhead/byte for fast devices
  • Better not waste time waiting for slow devices
In a Nutshell

- I/O devices you recognize are supported by I/O controllers
- CPU accesses them by reading and writing I/O registers like memory
  - Write commands and arguments, read status and results
Modern I/O Systems
What’s a Bus?

• Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  • Operations: e.g., Read, Write
  • Control lines, Address lines, Data lines

• Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data

• Very high BW close to processor (wide, fast, and inflexible), low BW with high flexibility out in I/O subsystem
Why a Bus?

• Buses let us connect $n$ devices over a single set of wires, connections, and protocols
  • $O(n^2)$ relationships with 1 set of wires (!)

• Downside: Only one transaction at a time
  • The rest must wait
  • “Arbitration” aspect of bus protocol ensures the rest wait
PCI Bus Evolution

- PCI started life out as a bus
- But a parallel bus has many limitations
  - Multiplexing address/data for many requests
  - Slowest devices must be able to tell what’s happening (e.g., for arbitration)
  - **Bus speed is set to that of the slowest device**
PCI Express “Bus”

• No longer a parallel bus
• Really a **collection of fast serial channels** or “lanes”
• Devices can use as many as they need to achieve a desired bandwidth
• Slow devices don’t have to share with fast ones

• One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI Express
  • The physical interconnect changed completely, but the old API still worked
Example: Typical PCI Architecture

CPU

RAM

Memory Bus

Host Bridge

PCI Bridge

PCI Slots

USB Controller

SATA Controller

Root Hub

Hub

Webcam

Mouse

Keyboard

Legacy Devices

Scanner

Hard Disk

DVD ROM

ISA Bridge

ISA Controller

PCI #0

PCI #1
How does the Processor Talk to the Device?

- CPU interacts with a **Controller**
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues, etc.
- Processor accesses registers in two ways:
  - **Port-Mapped I/O**: in/out instructions
    - Example from the Intel architecture: `out 0x21, AL`
  - **Memory-mapped I/O**: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions
Port-Mapped I/O in Pintos Speaker Driver

/* Reads and returns a byte from PORT. */
static inline uint8_t
inb (uint16_t port)
{
  /* See [IA32-v2a] "IN". */
  uint8_t data;
  asm volatile("inb %w1, %b0" : "a" (data) : "Nd" (port));
  return data;
}

/* Writes byte DATA to PORT. */
static inline void
outb (uint16_t port, uint8_t data)
{
  /* See [IA32-v2b] "OUT". */
  asm volatile("outb %b0, %w1" : "a" (data), "Nd" (port));
}

/* Sets the PC speaker to emit a tone at the given FREQUENCY, in Hz. */
void
speaker_on (int frequency)
{
  if (frequency >= 20 && frequency <= 20000)
  {
    /* Set the timer channel that's connected to the speaker to
       output a square wave at the given FREQUENCY, then
       connect the timer channel output to the speaker. */
    enum intr_level old_level = intr_disable ();
    pit_configure_channel (2, 3, frequency);
    outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) | SPEAKER_GATE_ENABLE);
    intr_set_level (old_level);
  }
  else
  {
    /* FREQUENCY is outside the range of normal human hearing.
       Just turn off the speaker. */
    speaker_off ();
  }
}

/* Turn off the PC speaker, by disconnecting the timer channel's
output from the speaker. */
void
speaker_off (void)
{
  enum intr_level old_level = intr_disable ();
  outb (SPEAKER_PORT_GATE, inb (SPEAKER_PORT_GATE) & ~SPEAKER_GATE_ENABLE);
  intr_set_level (old_level);
}
Demo: Pintos is Real!

• You can run Pintos and play songs using the speaker driver!

• We’ll provide the code and instructions to (optionally) try out
  • Booting Pintos natively has inherent risks
  • But it also works fine in VMWare
Example: Memory-Mapped Display Controller

• Memory-Mapped:
  • Hardware maps control registers and display memory into physical address space
    • Addresses set by HW jumpers or at boot time
  • Simply writing to display memory (also called the "frame buffer") changes image on screen
    • Addr: 0x8000F000 — 0x8000FFFF
  • Writing graphics description to cmd queue
    • Say enter a set of triangles describing some scene
    • Addr: 0x80010000 — 0x8001FFFF
  • Writing to the command register may cause on-board graphics hardware to do something
    • Say render the above scene
    • Addr: 0x0007F004
• Can protect with address translation
Demo: Memory-Mapped I/O
There’s more than just a CPU in there!
Chip-Scale Features of Skylake (x86 in 2015)

• Significant pieces:
  • Four OOO cores with deeper buffers
  • Integrated GPU, System Agent (Mem, Fast I/O)
  • Large shared L3 cache with on-chip ring bus
    • 2 MB/core instead of 1.5 MB/core
    • High-BW access to L3 Cache

• Integrated I/O
  • Integrated memory controller (IMC)
    • Two independent channels of DRAM
  • High-speed PCI-Express (for Graphics cards)
  • Direct Media Interface (DMI) Connection to PCH (Platform Control Hub)
Skylake I/O: Platform Controller Hub (PCH)

Platform Controller Hub
- Connected to processor with proprietary bus
  - Direct Media Interface
- Types of I/O on PCH:
  - USB, Ethernet
  - Thunderbolt 3
  - Audio, BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)

Sky Lake System Configuration
Operational Parameters for I/O

• Data granularity: Byte vs. Block
  • Some devices provide single byte at a time (e.g., keyboard)
  • Others provide whole blocks (e.g., disks, networks, etc.)

• Access pattern: Sequential vs. Random
  • Some devices must be accessed sequentially (e.g., tape)
  • Others can be accessed “randomly” (e.g., disk, CD, etc.)
    • Fixed overhead to start transfers
  • Some devices require continual monitoring (polling)
  • Others generate interrupts when they need service

• Transfer Mechanism: Programmed I/O and DMA
Transferring Data To/From Controller

- **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size

- **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

- **Sample interaction with DMA controller (from OSTEP book):**
Transferring Data To/From Controller

• Programmed I/O:
  • Each byte transferred via processor in/out or load/store
  • Pro: Simple hardware, easy to program
  • Con: Consumes processor cycles proportional to data size

• Direct Memory Access:
  • Give controller access to memory bus
  • Ask it to transfer data blocks to/from memory directly

• Sample interaction with DMA controller (from OSTEP book):

1. Device driver is told to transfer disk data to buffer at address X
2. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. Disk controller initiates DMA transfer
4. Disk controller sends each byte to DMA controller
5.DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. When C = 0, DMA interrupts CPU to signal transfer completion
Aside: Linux Memory Details

- Memory management in Linux considerably more complex than the examples we have been discussing

- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)

- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)

- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped

- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)

- Allocation priorities
  - Is blocking allowed/etc
Aside: Linux Virtual Memory Map

32-Bit Virtual Address Space

- Kernel Addresses: 0x00000000 - 0xFFFFFFFF
- User Addresses: 0x00000000 - 0xC0000000

64-Bit Virtual Address Space

- Kernel Addresses: 0x0000000000000000 - 0xFFFFFFFFFFFFFFFF
- User Addresses: 0x0000000000000000 - 0x00007FFFFFFF
- "Canonical Hole": 0xFFFF800000000000 - 0xFFFFFFFF

3GB Total

- 1GB Physical: 0xC0000000 - 0xFFFFFFFF
- 128TiB: 0xFFFF800000000000 - 0xFFFFFFFF

64TiB Physical

- 896MB Physical: 0xFFFF800000000000 - 0xFFFFFFFF

Kernel Addresses

Empty Space

User Addresses

32-Bit Virtual Address Space

64-Bit Virtual Address Space
I/O Device Notifying the OS

• The OS needs to know when:
  • The I/O device has completed an operation
  • The I/O operation has encountered an error

• I/O Interrupt: Device generates interrupt when it needs service
  • Handles unpredictable events well, but high overhead

• Polling: OS periodically checks device-specific status register
  • Low overhead, but may waste cycles for infrequent or unpredictable I/O

• Actual devices combine both polling and interrupts
  • E.g., high-bandwidth network adapter
Kernel Device Structure

The System Call Interface

- Process Management
- Memory Management
- Filesystems
- Device Control
- Networking

- Concurrency, multitasking
- Virtual memory
- Files and dirs: the VFS
- Device Control
- Connectivity

- Architecture Dependent Code
- Memory Manager
- File System Types
- TTYs and device access
- Network Subsystem

- Block Devices
- Connectivity
- IF drivers

Concurrency, multitasking, Virtual memory, Files and dirs: the VFS, Device Control, Connectivity

Architecture Dependent Code, Memory Manager, File System Types, TTYs and device access, Network Subsystem
Device Drivers

• Device-specific code in the kernel that interacts directly with the device hardware
  • Supports a standard, internal interface
  • Same kernel I/O system can interact easily with different device drivers
  • Special device-specific configuration supported with the ioctl() system call

• Device Drivers typically divided into two pieces:
  • Top half: accessed in call path from system calls
    • implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    • This is the kernel’s interface to the device driver
    • Top half will start I/O to device, may put thread to sleep until finished
  • Bottom half: run as interrupt routine
    • Gets input or transfers next block of output
    • May wake sleeping threads if I/O now complete

• In Linux, this convention is reversed!
Recall: Life Cycle of an I/O Request

User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware
The Goal of the I/O Subsystem

• Provide Uniform Interfaces, Despite Wide Range of Different Devices
  • This code works on many different devices:

    ```c
    FILE fd = fopen("/dev/something", "rw");
    for (int i = 0; i < 10; i++) {
        fprintf(fd, "Count %d\n", i);
    }
    close(fd);
    • Why? Because code that controls devices ("device driver") implements standard interface

• We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  • Can only scratch surface!
Want Standard Interface to Devices

- **Block Devices**: e.g. disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include open(), read(), write(), seek()
  - Raw I/O or file-system access
  - Memory-mapped file access possible

- **Character Devices**: e.g. keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - May not be buffered like block devices
  - Libraries layered on top allow line editing

- **Network Devices**: e.g. Ethernet, Wireless, Bluetooth
  - Different enough from block/character devices to have an extended interface
  - Unix and Windows include **socket** interface
    - Separates network protocol from network operation
    - Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes
How Does User Deal with I/O Timing?

• **Blocking Interface: “Wait”**
  • When request data (e.g. `read` syscall), put process to sleep until data is ready
  • When write data (e.g. `write` syscall), put process to sleep until device is ready for data

• **Non-blocking Interface: “Don’t Wait”**
  • Returns quickly from read or write request with count of bytes successfully transferred
  • Read may return nothing, write may write nothing

• **Asynchronous Interface: “Tell Me Later”**
  • When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  • When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Conclusion

• I/O Devices Types:
  • Many different speeds (0.1 bytes/sec to GBytes/sec)
  • Different Access Patterns:
    • Block Devices, Character Devices, Network Devices
  • Different Access Timing:
    • Blocking, Non-blocking, Asynchronous

• I/O Controllers: Hardware that controls actual device
  • Processor Accesses through I/O instructions, load/store to special physical memory

• Notification mechanisms
  • Interrupts
  • Polling: Report results through status register that processor looks at periodically

• Device drivers interface to I/O devices
  • Provide clean Read/Write interface to OS above
  • Manipulate devices through PIO, DMA & interrupt handling
  • Three types: block, character, and network