CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 19

Two-Level Scheduling (Con't)
Device Drivers

April 9th, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Tessellation (finished)
- Devices and Device Drivers

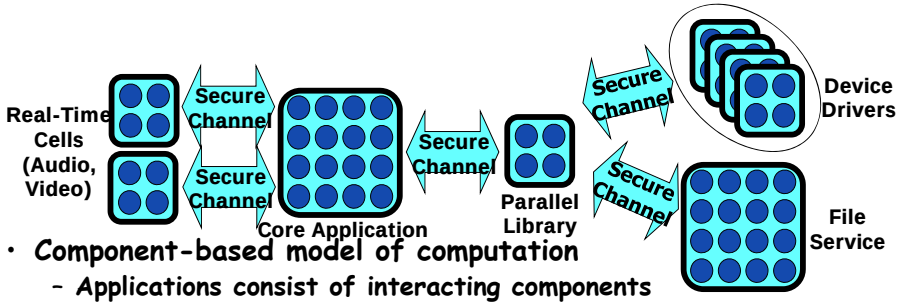
Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: the Cell

- Properties of a Cell
  - A user-level software component with guaranteed resources
  - Has full control over resources it owns ("Bare Metal")
  - Contains at least one memory protection domain (possibly more)
  - Contains a set of secured channel endpoints to other Cells
  - Hardware-enforced security context to protect the privacy of information and decrypt information (a Hardware TCB)
- Each Cell schedules its resources exclusively with application-specific user-level schedulers
  - Gang-scheduled hardware thread resources ("Harts")
  - Virtual Memory mapping and paging
  - Storage and Communication resources
    - » Cache partitions, memory bandwidth, power
  - Use of Guaranteed fractions of system services
- Predictability of Behavior ⇒
  - Ability to model performance vs resources
  - Ability for user-level schedulers to better provide QoS

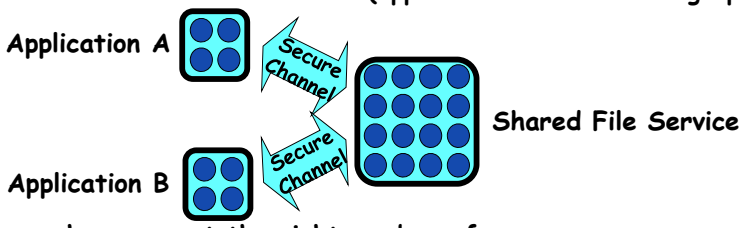
Applications are Interconnected Graphs of Services



- Component-based model of computation
  - Applications consist of interacting components
  - Explicitly asynchronous/non-blocking
  - Components may be local or remote
- Communication defines Security Model
  - Channels are points at which data may be compromised
  - Channels define points for QoS constraints
- Naming (Brokering) process for initiating endpoints
  - Need to find compatible remote services
  - Continuous adaptation: links changing over time!

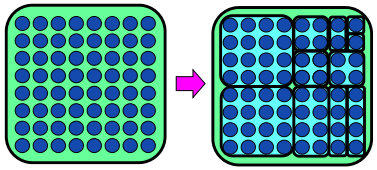
Impact on the Programmer

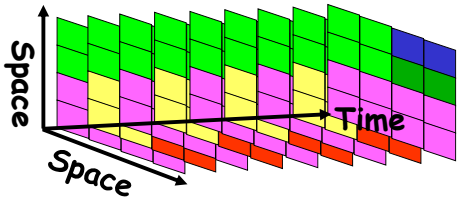
- Connected graph of Cells ⇔ Object-Oriented Programming
  - Lowest-Impact: Wrap a functional interface around channel
    - » Cells hold "Objects", Secure channels carry RPCs for "method calls"
    - » Example: POSIX shim library calling shared service Cells
  - Greater Parallelism: Event triggered programming
- Shared services complicate resource isolation:
  - How to guarantee that each client gets guaranteed fraction of service?
  - Distributed resource attribution (application as distributed graph)



- Must somehow request the right number of resources
  - Analytically? AdHoc Profiling? Over commitment of resources?
  - Clearly doesn't make it easy to adapt to changes in environment

Space-Time Partitioning ⇒ Cell

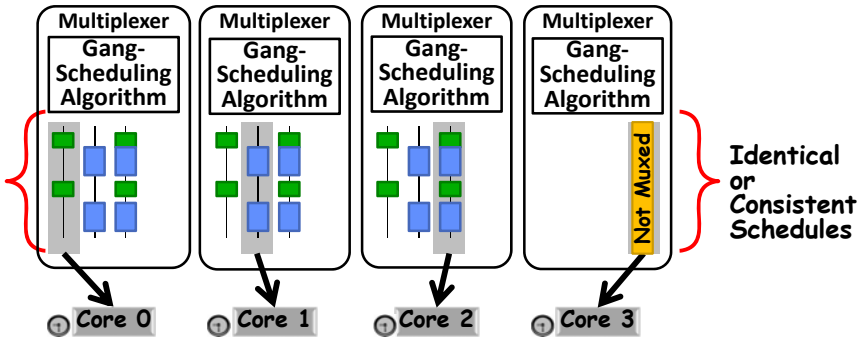




- Spatial Partition: Performance isolation
  - Each partition receives a vector of basic resources
    - » A number HW threads
    - » Chunk of physical memory
    - » A portion of shared cache
    - » A fraction of memory BW
    - » Shared fractions of services
- Partitioning varies over time
  - Fine-grained multiplexing and guarantee of resources
    - » Resources are gang-scheduled
- Controlled multiplexing, not uncontrolled virtualization
- Partitioning adapted to the system's needs

Efficient Space-Time Partitioning

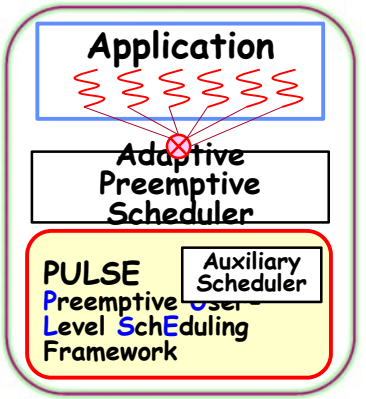
- Communication-Avoiding Gang Scheduling



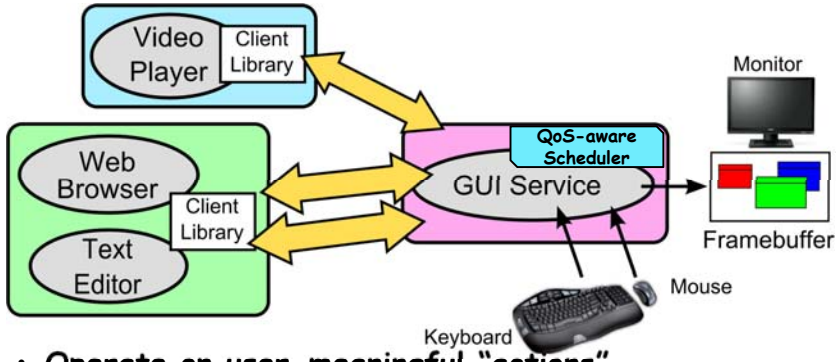
- Supports a variety of Cell types with low overhead
  - Cross between EDF (Earliest Deadline First) and CBS (Constant Bandwidth Server)
  - Multiplexers do not communicate because they use synchronized clocks with sufficiently high precision

Adaptive, Second-Level Preemptive Scheduling Framework

- PULSE: Preemptive User-Level SchEduling Framework for adaptive, premptive schedulers:
  - Dedicated access to processor resources
  - Timer Callback and Event Delivery
  - User-level virtual memory mapping
  - User-level device control
- Auxiliary Scheduler:
  - Interface with policy service
  - Runs outstanding scheduler contexts past synchronization points when resizing happens
  - 2nd-level Schedulers not aware of existence of the Auxiliary Scheduler, but receive resize events
- A number of adaptive schedulers have already been built:
  - Round-Robin, EDF, CBS, Speed Balancing

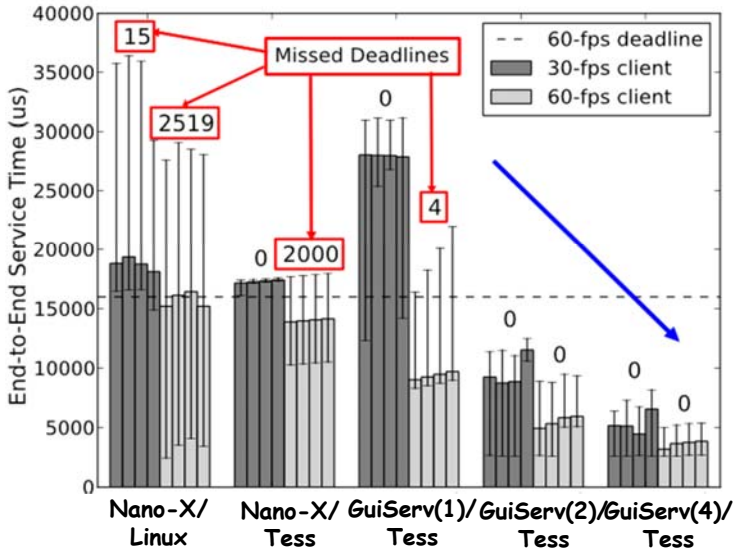


Example: Tesselation GUI Service

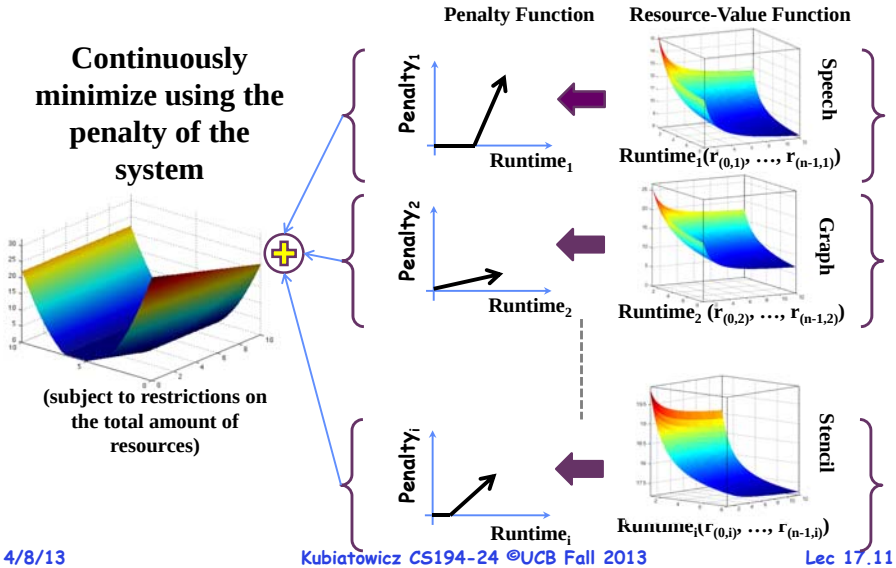


- Operate on user-meaningful "actions"
  - E.g. "draw frame", "move window"
- Service time guarantees (soft real-time)
  - Differentiated service per application
  - E.g. text editor vs video
- Performance isolation from other applications

Composite Resource with Greatly Improved QoS

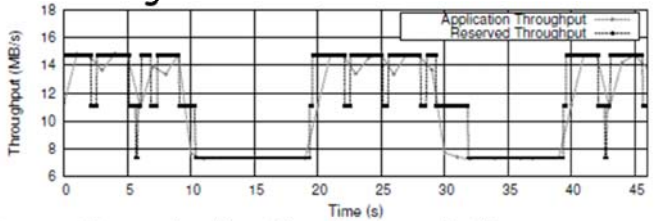


Tackling Multiple Requirements: Express as Convex Optimization Problem



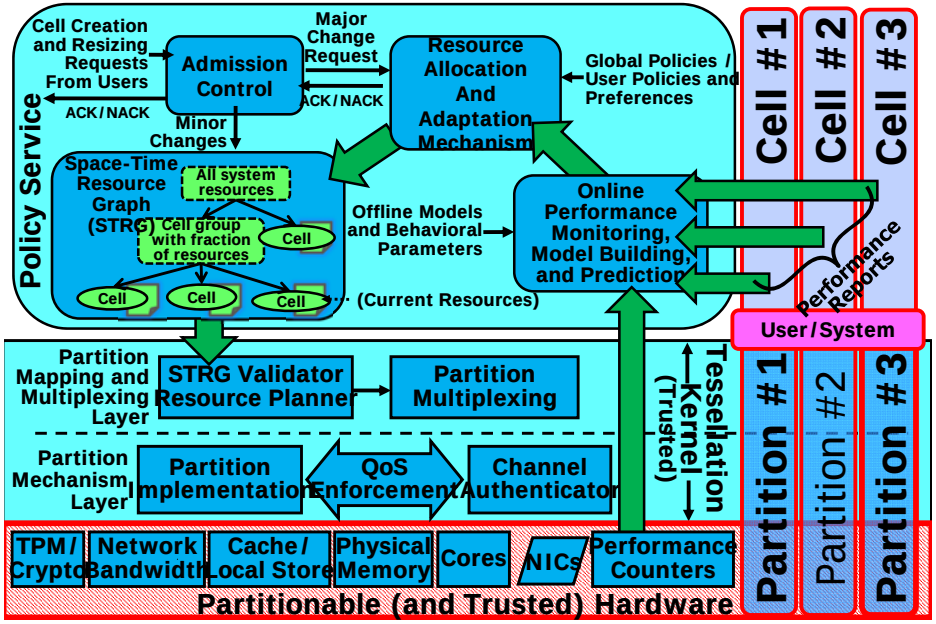
Feedback Driven Policies

- Simple Policies do well but online exploration can cause oscillations in performance
- Example: Video Player interaction with Network
  - Server or GUI changes between high and low bit rate
  - Goal: set guaranteed network rate:



- Alternative: Application Driven Policy
  - Static models
  - Let network choose when to decrease allocation
  - Application-informed metrics such as needed BW

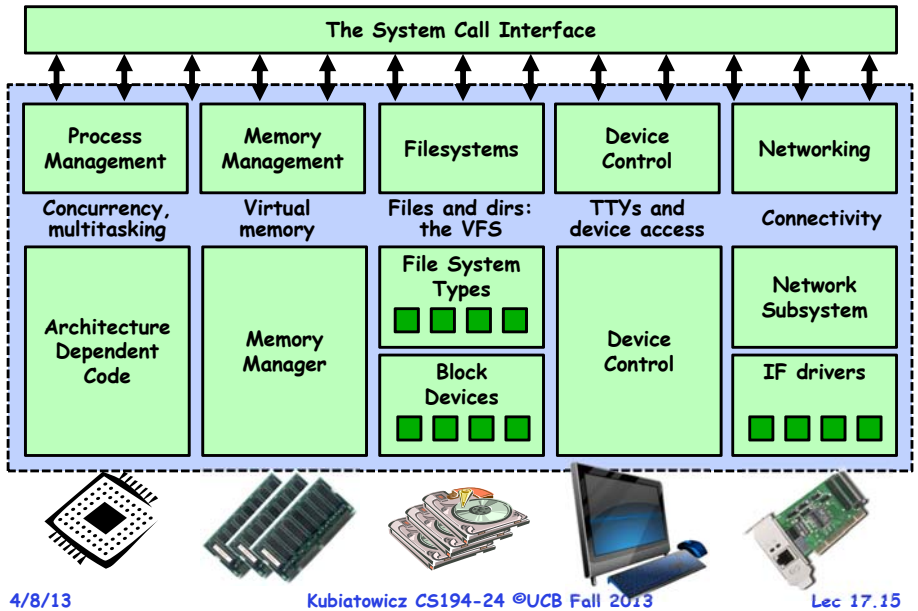
Architecture of Tessellation OS



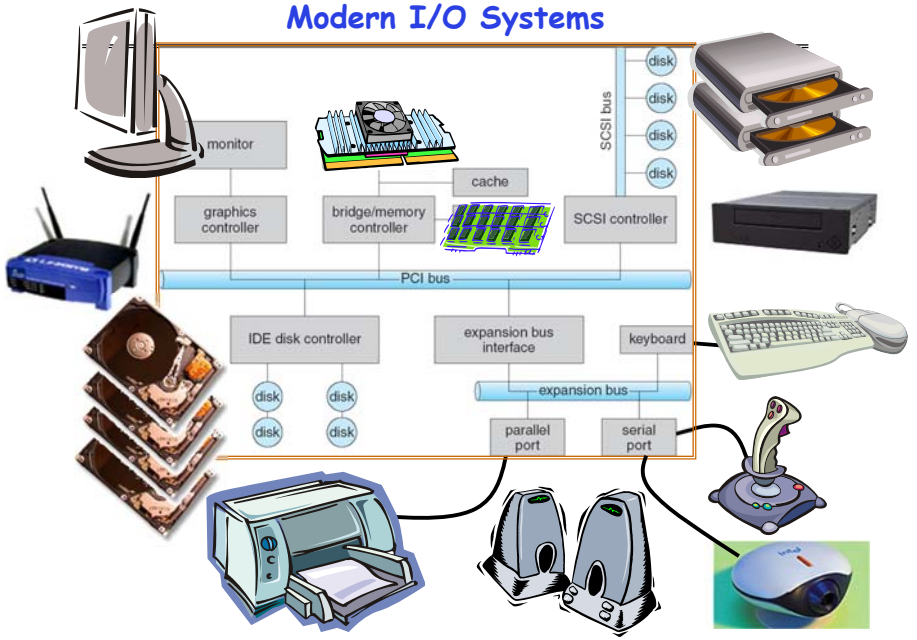
Administrivia

- Midterm I Results:
  - AVG: 65.6
  - STDdev: 16.1
- Usually B+- centered
  - Will see how the numbers end up working out…
- May be no class next Wednesday
  - Stay tuned

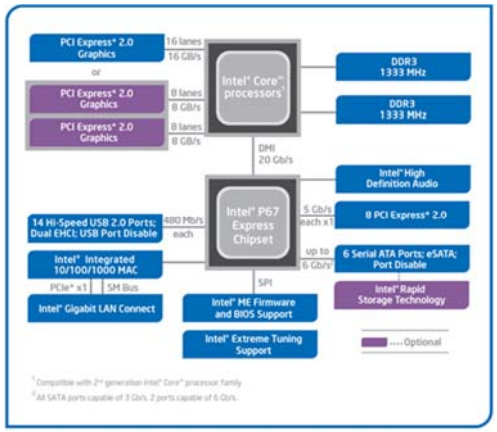
Kernel Device Structure



Modern I/O Systems



SandyBridge I/O: PCH



SandyBridge
System Configuration

- Platform Controller Hub
  - Used to be "SouthBridge," but no "NorthBridge" now
  - Connected to processor with proprietary bus
    - » Direct Media Interface
  - Code name "Cougar Point" for SandyBridge processors
- Types of I/O on PCH:
  - USB
  - Ethernet
  - Audio
  - BIOS support
  - More PCI Express (lower speed than on Processor)
  - Sata (for Disks)

Why Device Drivers?

- Many different devices, many different properties
  - Devices different even within class (i.e. network card)
    - » DMA *vs* Programmed I/O
    - » Processing every packet through device driver *vs* setup of packet filters in hardware to sort packets automatically
    - » Interrupts vs Polling
    - » On device buffer management, framing options (e.g. jumbo frames), error management, …
    - » Authentication mechanism, etc
  - Provide standardized interfaces to computer users
    - » Socked interface with TCP/IP
- Factor portion of codebase specific to a given device
  - Device manufacturer can hide complexities of their device behind standard kernel interfaces
  - Also: Device manufacturer can fix quirks/bugs in their device by providing new driver

The Goal of the I/O Subsystem

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

```
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!

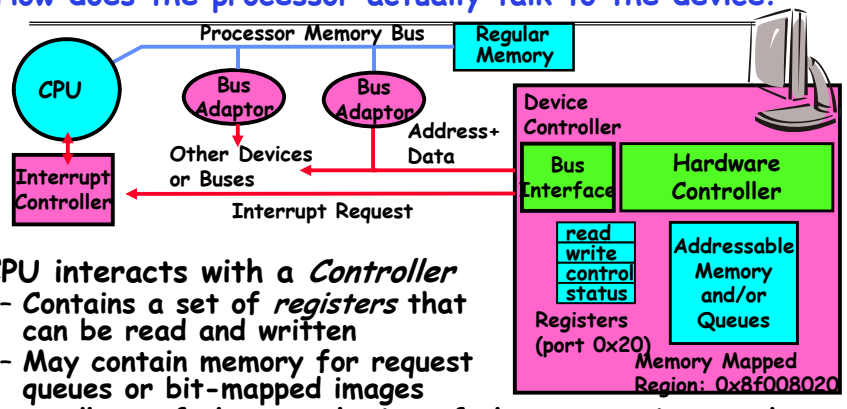
Want Standard Interfaces to Devices

- 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
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- Network Devices: *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own 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 Timing?

- Blocking Interface: "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), 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

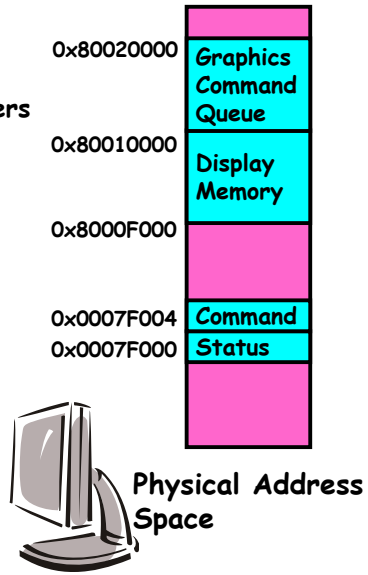
How does the processor actually talk to the device?



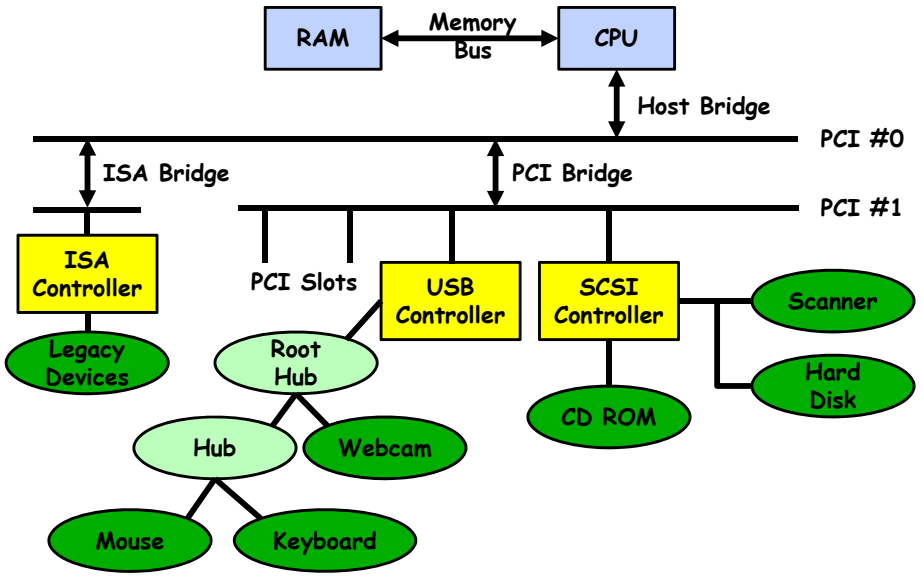
- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » 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

Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe 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 page tables

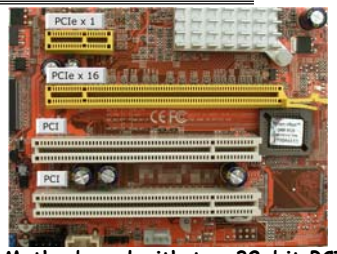


Example: PCI Architecture



PCI Bus Variants

- PCI History:
  - PCI 1.0, 1992: Original
  - PCI 2.0, 1993: Connector and add-in card spec
  - PCI 2.1, 1995: Clarifications and added 66 MHz spec
  - PCI 2.2, 1998: Added ECNs and improved readability
  - PCI 3.0, 2002: removed support for 5.0 volt keyed connector



A Motherboard with two 32-bit PCI slots and two sizes of PCI Express

- PCI Express (PCIe): Serial communication over 1, 2, 4, 8, 16, and 32 bit "lanes"
  - PCIe 1.0a, 2003: Per-lane data rate of 250MB/s, transfer rate of 2.5 gigatransfer (GT/s) which includes overhead bits
  - PCIe 2.0, 2007: Doubles transfer rate to 5 GT/s total. Fully backward-compatible with 1.0a
  - PCIe 3.0, 2010: Upgrade to 8 GT/s with lower overhead
  - PCIe 4.0, 2011: Upgrade to 16 GT/s

PCI Details

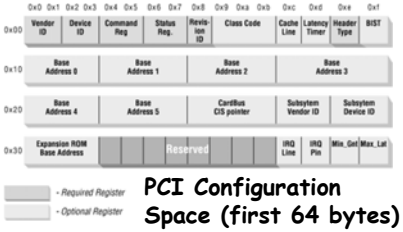
- PCI Addressing:
  - PCI peripherals identified by a *bus* number, a *device* number, and a *function* number
  - Host up to 256 buses, each bus up to 32 devices, each device can be a multifunctionboard board with a maximum of 8 functions
    - » Consequently each function identified by 16-bit address or "key"
    - » Linux device drivers don't need to deal with addresses, since they use a pci_dev structure to act on devices
  - Most workstations feature at least 2 PCI buses – organized as a tree
- Communicating with PCI cards:
  - Normal operation: Memory locations, I/O ports
    - » Memory and I/O regions accessed via ports or memory-mapped addresses
    - » Memory space: 32-bit or 64b-bit addresses (although latter available only to certain platforms)
    - » I/O space: 32-bit, leading to 4GB of ports
  - Interrupts:
    - » Every PCI slot has four interrupt pins; each device function can use one of them without worrying about how pins routed to CPU
    - » PCI Spec requires interrupt lines to be shareable
    - » 32-bit addressing for I/O ports
  - Configuration space:
    - » 256 bytes of configuration space
    - » Configuration transactions performed by calling specific kernel functions

PCI Details (con't)

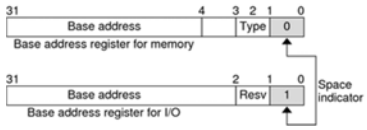
- Device identification:
  - vendorID (16 bits): global registry of vendors
  - deviceID (16-bits): vendor-assigned device
  - class (16-bits): top 8 bits identify "base class" (i.e. network)
  - subsystem vendorID/subsystem deviceID
    - » Used to help identify bridges/interfaces
- Example initialization:

```
#ifndef CONFIG_PCI
# error "This driver needs PCI support to be available"
#endif

int mydev_find_all_devices(void) {
   struct pci_dev *dev = NULL;
   int found;
   if (!pci_present()) return -ENODEV;

   for (found=0; found < MYDEV_MAX_DEV;) {
      dev = pci_find_device(MYDEV_VENDOR, MYDEV_ID, dev);
      if (!dev) /* no more devices are there */
         break; /* do device-specific actions and count the device */
      found += mydev_init_one(dev);
   }
   return (index == 0) ? -ENODEV : 0;
}
```


PCI Details (con't)



PCI Configuration Space (first 64 bytes)



PCI Configuration Space (Address Registers):
Type: 0x0: 32-bits
0x2: 64-bits (2 regs)

- Access configuration space with special functions:

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *ptr);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *ptr);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *ptr);
int pci_write_config_byte (struct pci_dev *dev, int where, u8 val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 val);
```

- Example: Figure out which interrupt line

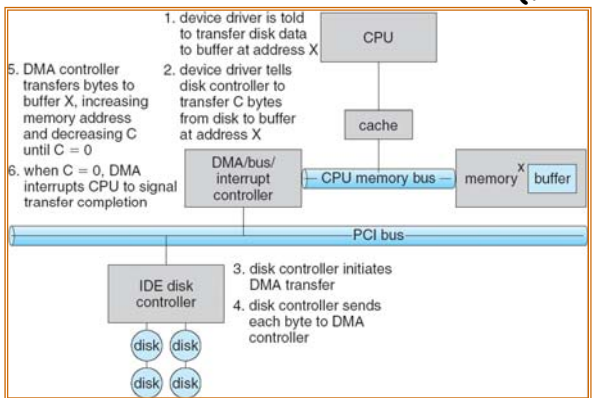
```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) { /* deal with error */ }

int request_irq(myirq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

Device Drivers

- **Device Driver:** 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
- Linux Device drivers often installed via a Module
  - Interface for dynamically loading code into kernel space
  - Modules loaded with the "insmod" command and can contain parameters
- Driver-specific structure
  - One per driver
  - Contains a set of standard kernel interface routines
    - » Open: perform device-specific initialization
    - » Read: perform read
    - » Write: perform write
    - » Release: perform device-specific shutdown
    - » Etc.
  - These routines registered at time device registered

Transfering 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 to/from memory directly
- Sample interaction with DMA controller (from book):



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 an interrupt whenever it needs service
  - Handled in top half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- **Polling:**
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

Interrupt handling

- Interrupt routines typically divided into two pieces:
  - Top half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » Handles any direct access to hardware
    - » Handles any time-sensitive aspects of handling interrupts
    - » Runs in the ATOMIC Context (cannot sleep)
  - Bottom half: accessed later to finish processing
    - » Perform any interrupt-related work not performed by the interrupt handler itself
    - » Scheduled "later" with interrupts re-enabled
    - » Some options for bottom halves can sleep
- Since you typically have two halves of code, must remember to *synchronize* shared data
  - Since interrupt handler is running in interrupt (ATOMIC) context, cannot sleep!
  - Good choice: spin lock to synchronize data structures
  - Must be careful never to hold spinlock for too long
    - » When non-interrupt code holds a spinlock, must make sure to disable interrupts!
    - » Consider "spin_lock_irqsave()" or "spin_lock_bh()" variants
  - Consider lock free queue variants as well

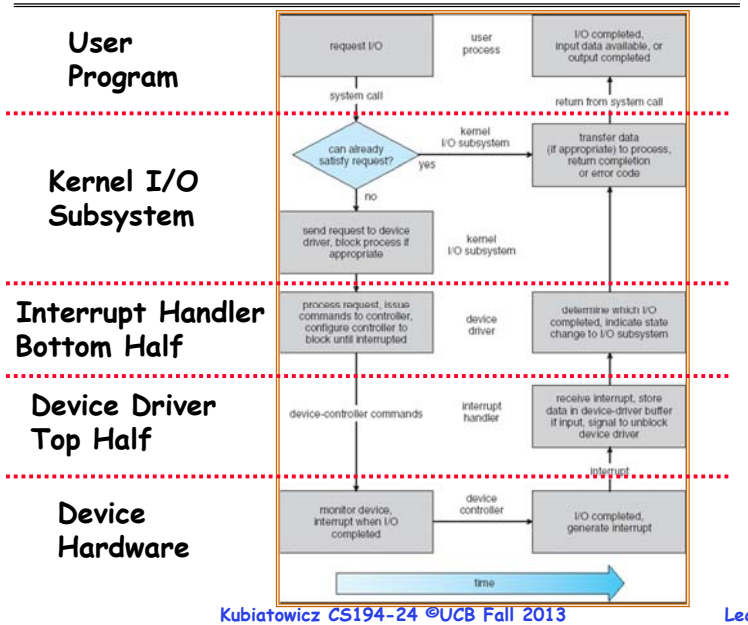
Options for Bottom Half

- Bottom Half used for handling work after interrupt is re-enabled (i.e. deferred work):
  - Perform any interrupt-related work not performed by the interrupt handler
  - Ideally most of the work
  - What to minimize amount of work done in an interrupt handler because they run with interrupts disabled
- Many different mechanisms for handling bottom halves
  - Original "Bottom Half" (deprecated)
  - Task Queues
    - » Put work on a task queue for later execution
  - Softirqs are statically defined bottom halves that can run simultaneously on any processor
  - Tasklets: dynamically created bottom halves built on top of softirq mechanism
    - » Only one of each type of tasklet can run at given time
    - » Simplifies synchronization

More on Synchronization

- Must always be aware that interrupt handlers can interrupt running code!
  - Must come up with a synchronization methodology to deal with this issue
  - May need to deal with multiple processors
- Some possible ways of dealing with synchronization:
  - Build some sort of lock-free queue implemented as a circular buffer
  - Spinlocks
  - Lock variables that are atomically incremented and decremented
- Note about spinlocks
  - Many variants, make sure to use variants to disable interrupts as well as spin
  - Bovet Chapter 9 has lots of discussion of synchronization

Life Cycle of An I/O Request



Summary

- 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
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)
- Notification mechanisms
  - Interrupts
  - Polling: Report results through status register that processor looks at periodically
- Device Driver: Code specific to device which handles unique aspects of device