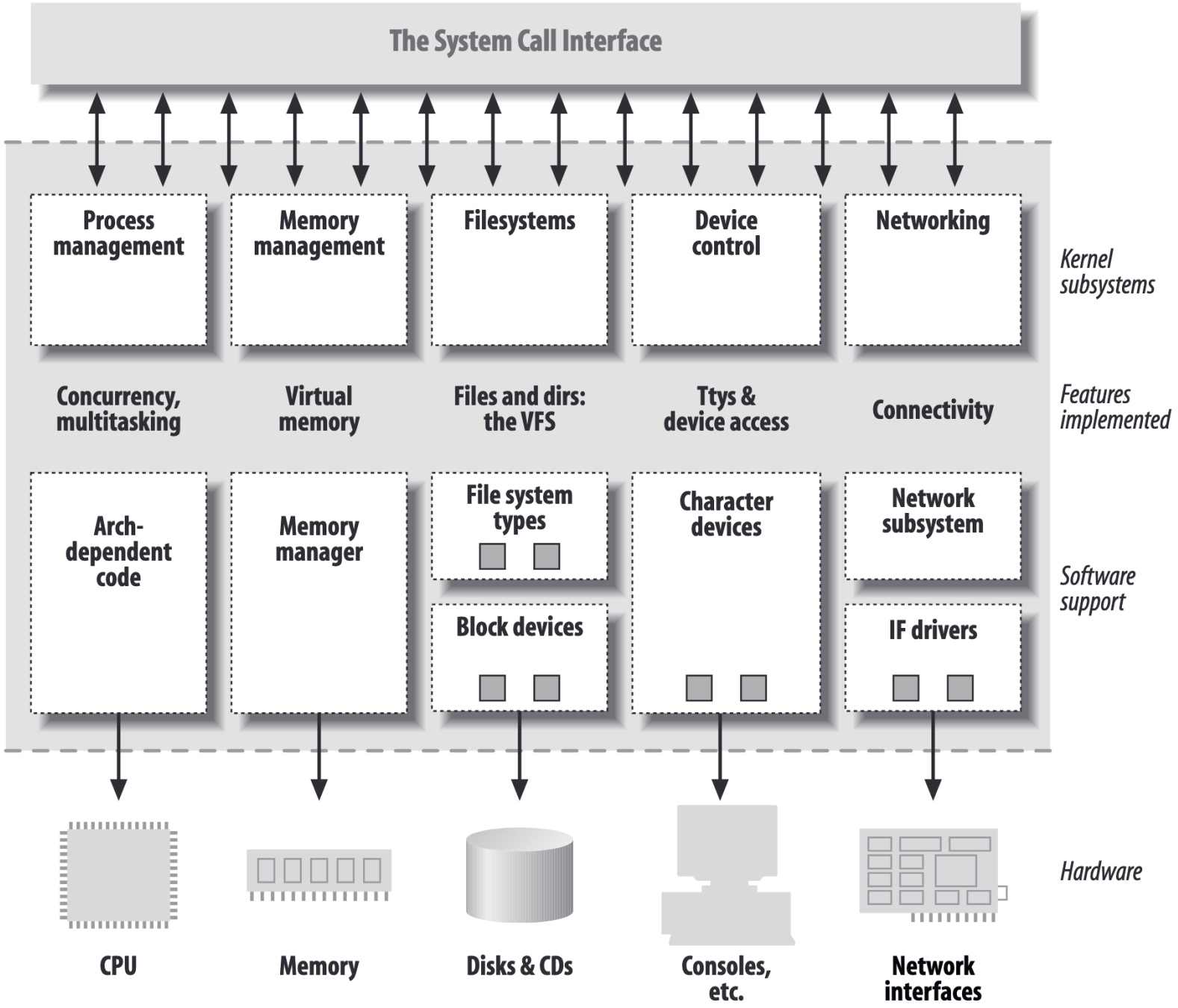
cs6465: Advanced Operating System Implementation

Lecture 05: Device drivers

Anton Burtsev

September, 2024



https://lwn.net/Kernel/LDD3/

Device drivers

- Conceptually
- Implement interface to hardware
- Expose some high-level interface to the kernel or applications
  - In the past: everything was a file

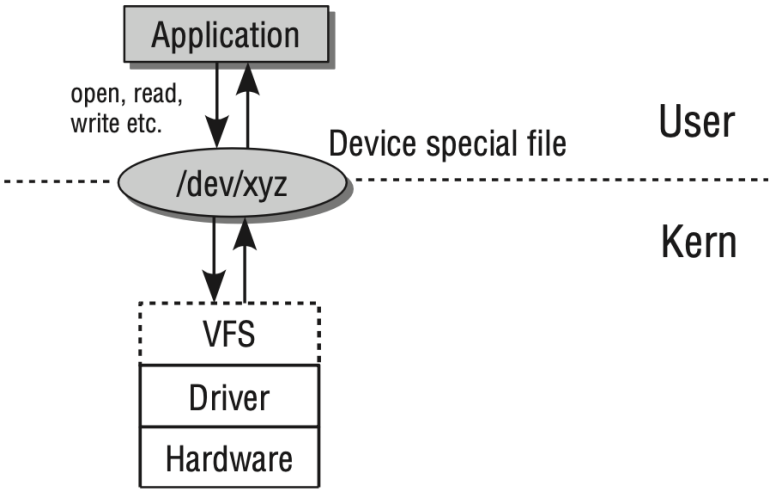


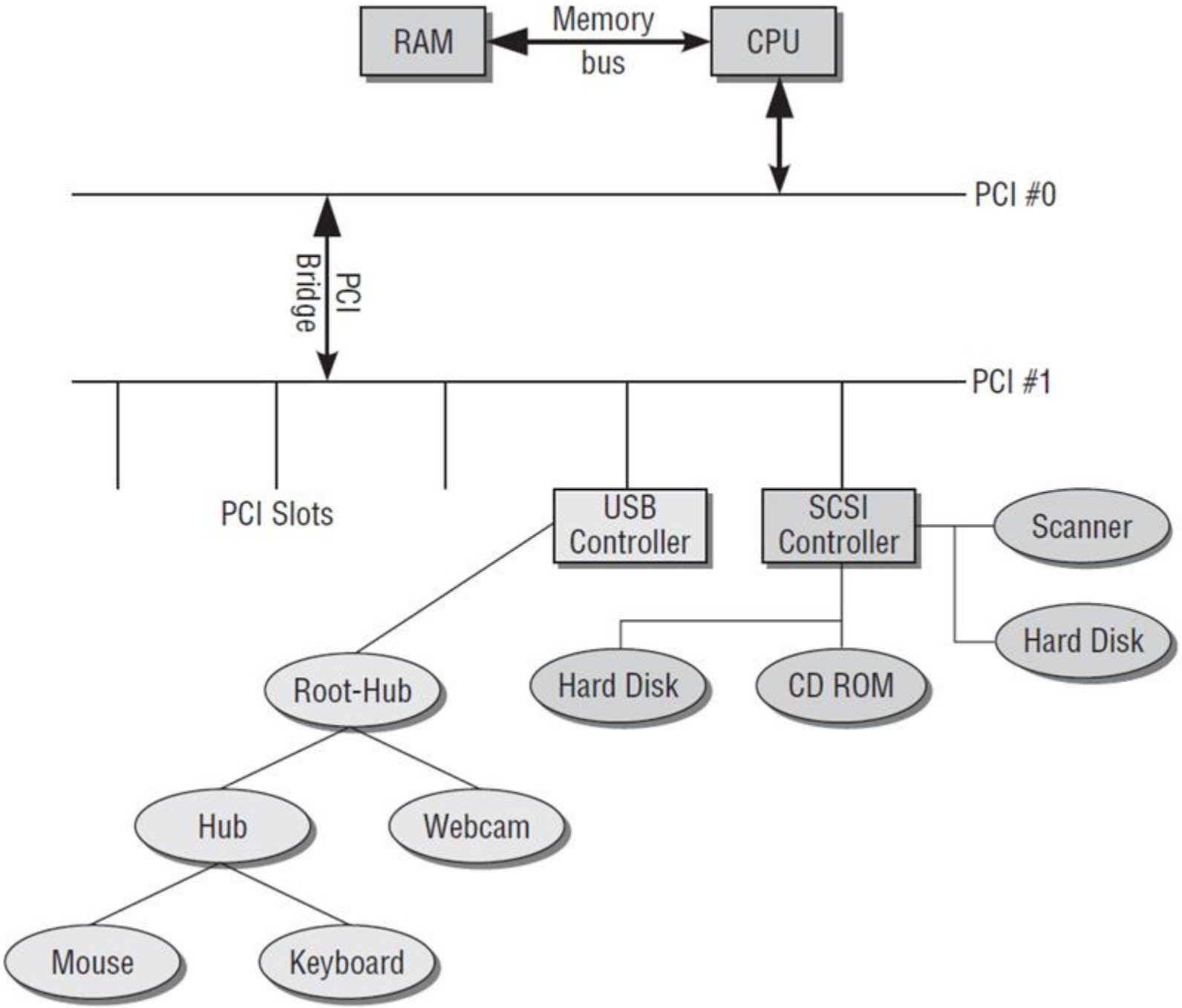
Figure 6-1: Layer model for addressing peripherals.

Devices in UNIX

- In UNIX devices expose file-like interface
- They are files in the file system
  - /dev/sda, /dev/dsp
- Applications can read and write into them
  - dd if=/dev/sda of=/my-disk-image bs=1M
  - cat thesis.txt > /dev/lp

Bus subsystem

- Buses are the mechanism that enable the flow of data across CPU, memory, and devices


RAM
Memory bus
CPU
PCI #0
PCI Bridge
PCI #1
PCI Slots
USB Controller
SCSI Controller
Scanner
Hard Disk
Root-Hub
Hard Disk
CD ROM
Hub
Webcam
Mouse
Keyboard

Buses

- PCI (Peripheral Component Interconnect) – main system bus on most architectures
- USB (Universal Serial Bus) – external bus, hotplug capability, devices are connected in a tree
- SCSI (Small Computer System Interface) – high-throughput bus used mainly for disks

Interacting with peripherals

- Old way: separate I/O and memory space
  - Memory
- I/O ports
  - Device is identified by the port number
  - 2^16 ports (64K ports)
  - in, out instructions to read and write data from a port
  - Connect straight to a peripheral

Interacting with devices

- Initial x86 model: separate memory and I/O space
  - Memory uses virtual addresses
  - Devices accessed via ports
- A port is just an address (like memory)
  - Port 0x1000 is not the same as address 0x1000
  - Different instructions – inb, inw, outl, etc

Memory mapped I/O

- Map devices onto regions of physical memory
  - Hardware basically redirects these accesses away from RAM at same location (if any), to devices
  - Of course, you “lose” some RAM

Interacting with peripherals

- Interrupts
- CPU provides several interrupt lines
  - One line can be shared across several devices
- Polling
  - Periodic check of the device state for whether more data is available

Configuration

- How do we know which ports and memory addresses map to devices?
  - In the past it was static
  - Remember the ISA memory hole in xv6
  - 640KB to 1MB
  - It’s an old ISA bus standard

Configuration: PCI

- Now everything (both memory and I/O ports) is dynamic
  - Typically, BIOS, but kernel can remap too
  - On startup BIOS performs an enumeration protocol

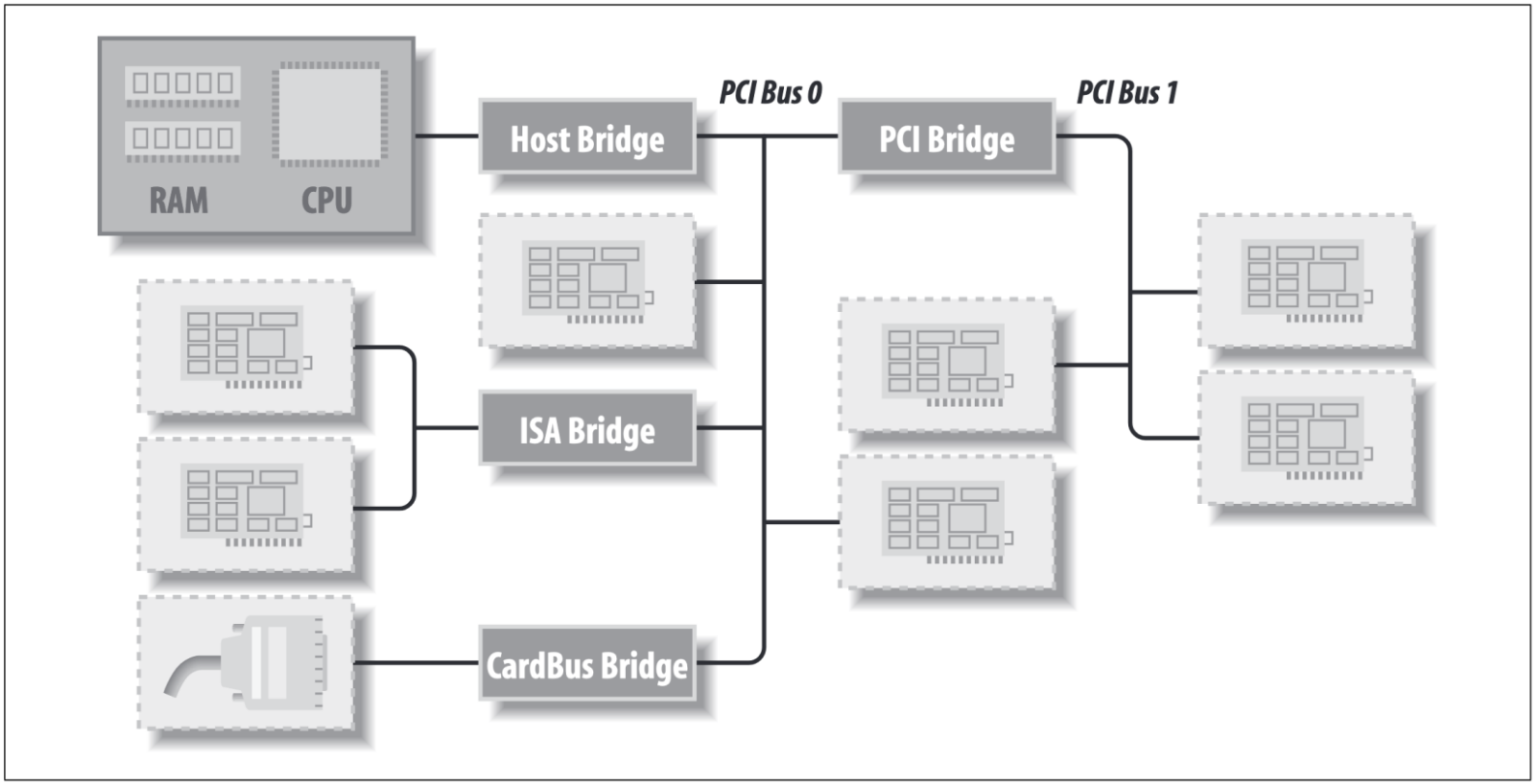


Figure 12-1. Layout of a typical PCI system

- Most desktop systems have 2+ PCI buses
  - Joined by a bridge device
  - Forms a tree structure (bridges have children)

PCI (geographical) addressing

- Each peripheral listed by:
  - Bus Number (up to 256 per domain or host)
    - A large system can have multiple domains
  - Device Number (32 per bus)
  - Function Number (8 per device)
    - Function, as in type of device, not a subroutine
    - E.g., video capture card may have one audio function and one video function
- Devices addressed by a 16 bit number

To assign a memory address

- Bios tries slot #1, slot #2, etc.
  - For non-present devices the system returns a non-present for function #0
  - Each device is programmed to have a configuration space at a specific memory address

https://en.wikipedia.org/wiki/PCI_configuration_space

Configuration registers

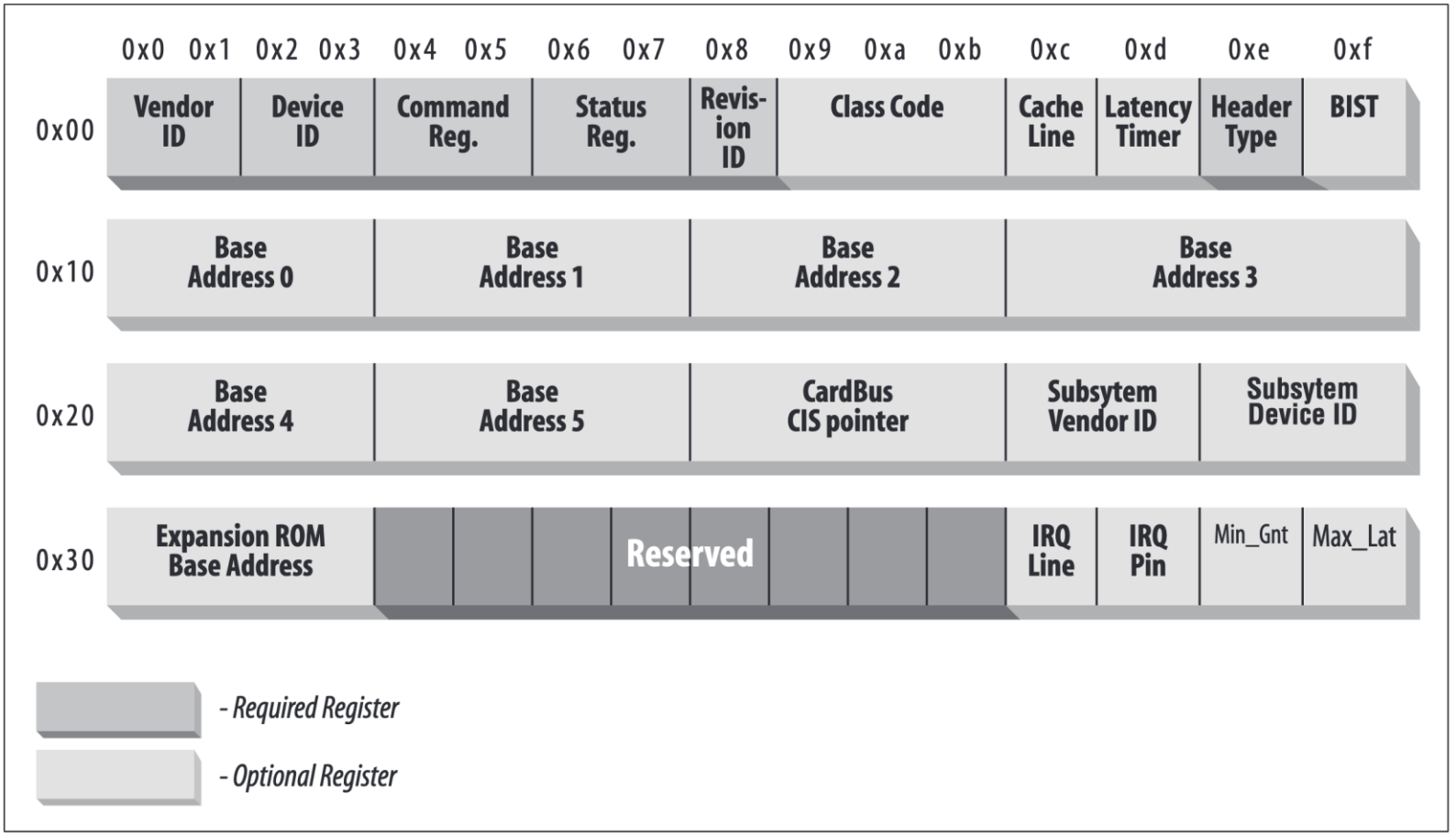


Figure 12-2. The standardized PCI configuration registers

- First 64 bytes are standard

Implementing device drivers

Kernel modules

- Linux allows extending itself with kernel modules
- Most device drivers are implemented as kernel modules
- Loadable at run-time on demand, when device is detected

Hello world module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

File operations

- Remember devices are exported as special files
- Each device needs to implement a file interface
- Each inode and file has a pointer to an interface
- Set of functions which are used for opening, reading, writing, etc.
- Same with device files
  - Each device file has a pointer to a set of functions

File operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, struct iovec *, unsigned, loff_t *);
    ssize_t (*writev) (struct file *, struct iovec *, unsigned, loff_t *);
};
```

File operations

- You don't need to implement all file operations
- Some can remain NULL
- Kernel will come up with some default behavior

```
static struct file_operations simple_driver_fops =
{
   .owner   = THIS_MODULE,
   .read    = device_file_read,
};
```

Register with the kernel

- Register character device with the kernel

```
static int device_file_major_number = 0;
static const char device_name[] = "Simple-driver";
static int register_device(void)
{
   result = register_chrdev( 0, device_name, &simple_driver_fops );
   if( result < 0 )
   {
      printk( KERN_WARNING "Simple-driver:  can\'t register
            character device with errorcode = %i", result );
      return result;
   }

   device_file_major_number = result;
};
```

Read function

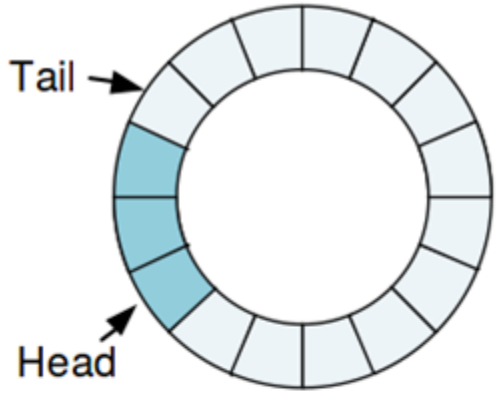
```
ssize_t (*read) (struct file *, char *,  size_t, loff_t *);
```

- First arg – pointer to the file struct
  - Private information for us, e.g. state of this file
- Second arg – buffer in user space to read data into
- Third arg – number of bytes to read
- Fourth arg – position in a file from where to read

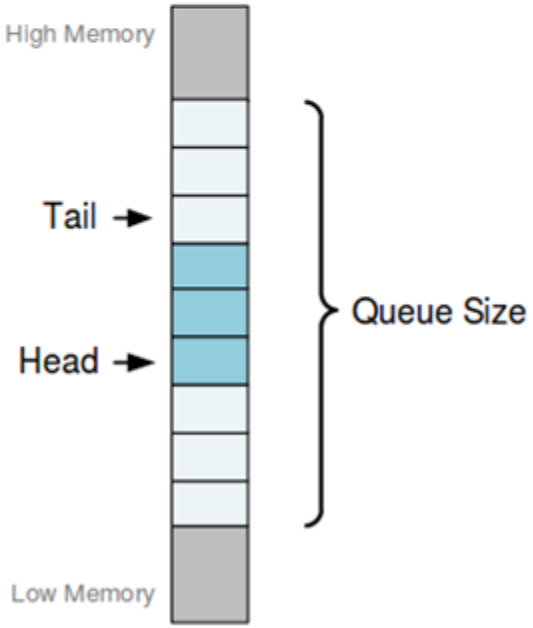
Example: Non-Volatile Memory Express (NVMe)

Fast communication queues

- Circular queues to pass messages (e.g., commands and command completion notifications.)
  - Typically queues are located in host memory
  - Queues may consist of a contiguous block of physical memory or optionally a non-contiguous set of physical memory pages.
  - A Queue consists of set of fixed sized elements
- Tail
  - Points to next free element
- Head
  - Points to next entry to be pulled off, if queue is not empty



Logical View



Physical View in Memory

Queue Size (Usable)

Number of entries in the queue - 1

- Minimum size is 2, Maximum is ~ 64K for I/O Queues and 4K for Admin Queue

Queue Empty

- Head == Tail

Queue Full

- Head == Tail + 1 mod # Of Queue Entries.

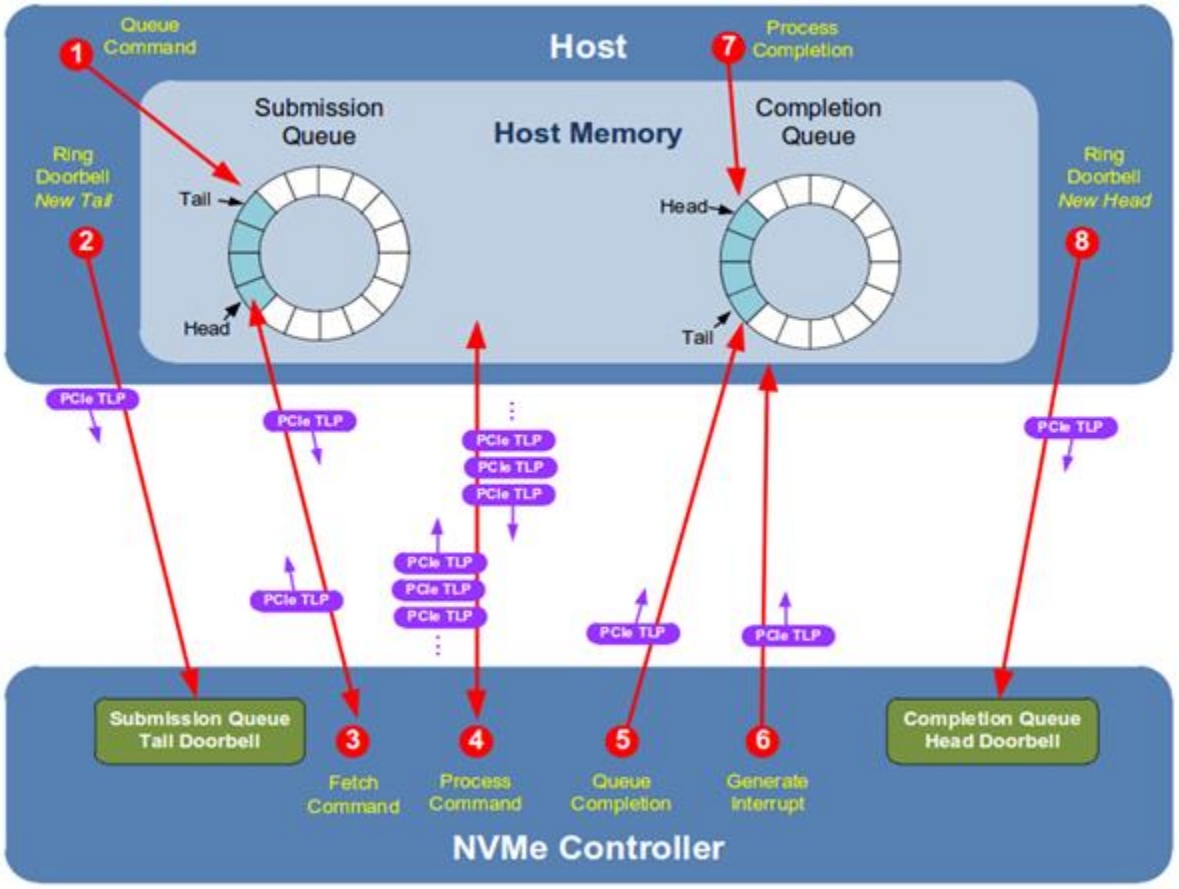
Types of queues

- Admin Queue for Admin Command Set
  - One per NVMe controller with up to 4K elements per queue
  - Used to configure IO Queues and controller/feature management
- I/O Queues for IO Command Sets (e.g., NVM command set)
  - Up to 64K queues per NVMe controller with up to 64K elements per queue
  - Used to submit/complete IO commands

Each type has:

- Submission Queues (SQ)
  - Queues messages from host to controller
  - Used to submit commands
  - Identified by SQID
- Completion Queues (CQ)
  - Queues messages from controller to host
  - Used to post command completions
  - Identified by CQID
  - May have an independent MSI-X(Message Signalled Interrupts) interrupt per completion queue

NVMe Commands Submission



Command Submission:

1. Host writes command to submission queue.
2. Host writes updated submission queue tail pointer to doorbell.

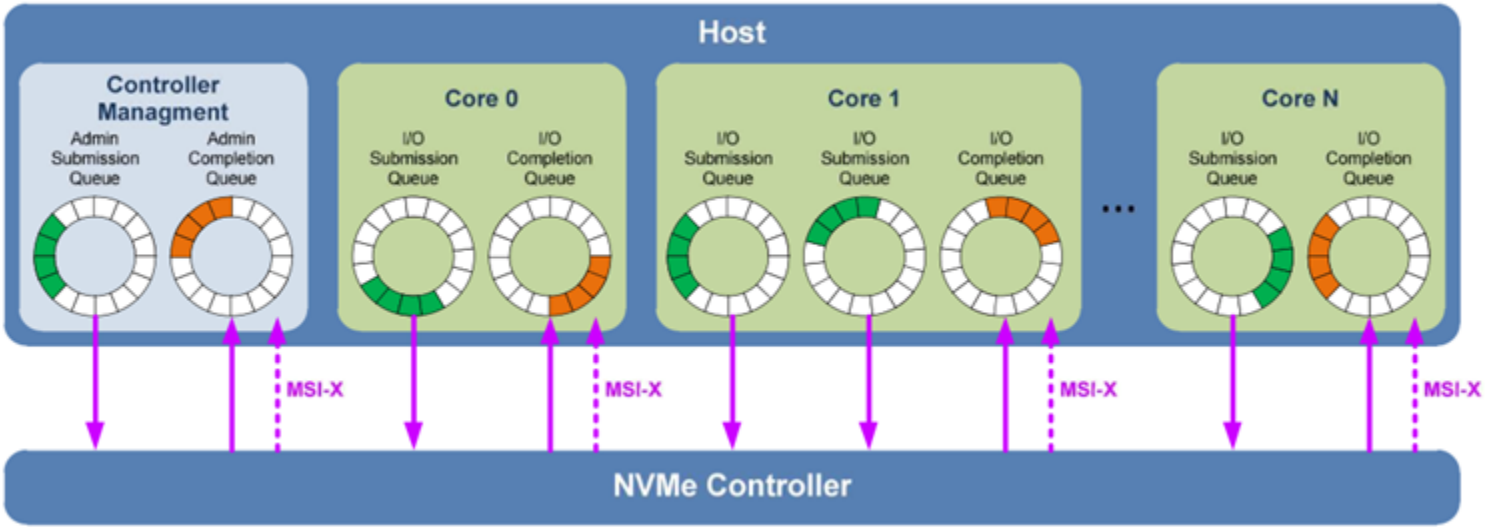
Command Processing:

1. Controller fetches command.
2. Controller processes command.

Command Completion:

1. Controller writes completion to Completion queue.
2. Controller generates interrupt.
3. Host processes completion.
4. Host writes updated Completion Queue head pointer to doorbell.

Scalability



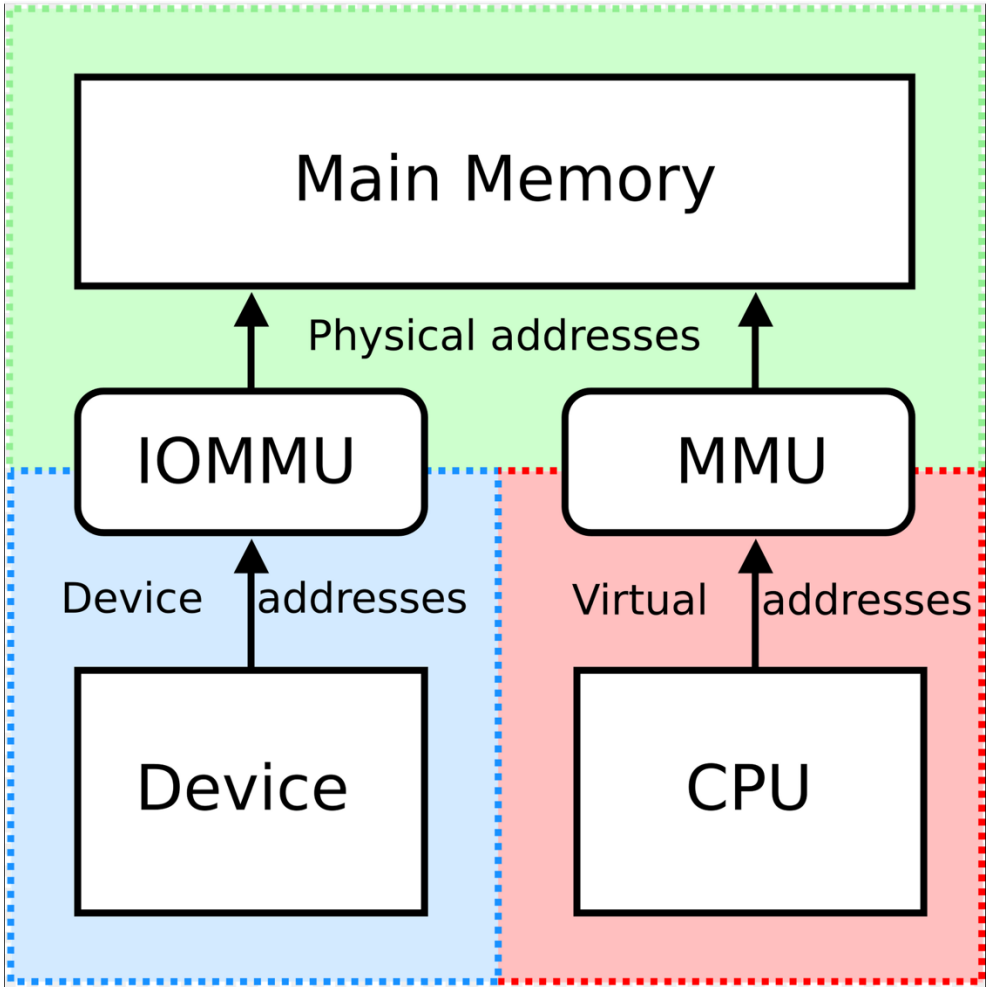
- Per core: One or more submission queues, one completion queue, and one MSI-X interrupt.
  - High performance and low latency command issue
  - No locking between cores
- Up to ~2^32 outstanding commands
  - Support for up to ~ 64K I/O submission and completion queues
  - Each queue supports up to ~ 64K outstanding commands

A couple of problems

- Devices can be malicious
- If I can write to a network card's control register and tell it where to write the next packet
  - What if I give it an address used for something else?
  - Like another process's address space
    - Nothing stops this
  - DMA privilege effectively equals privilege to write to any address in physical memory!
- It's also painful to allocate continuous regions of memory
  - For ring buffers, etc.

IOMMU

- Input–output memory management unit
- A page table for all device memory accesses



Thank you!