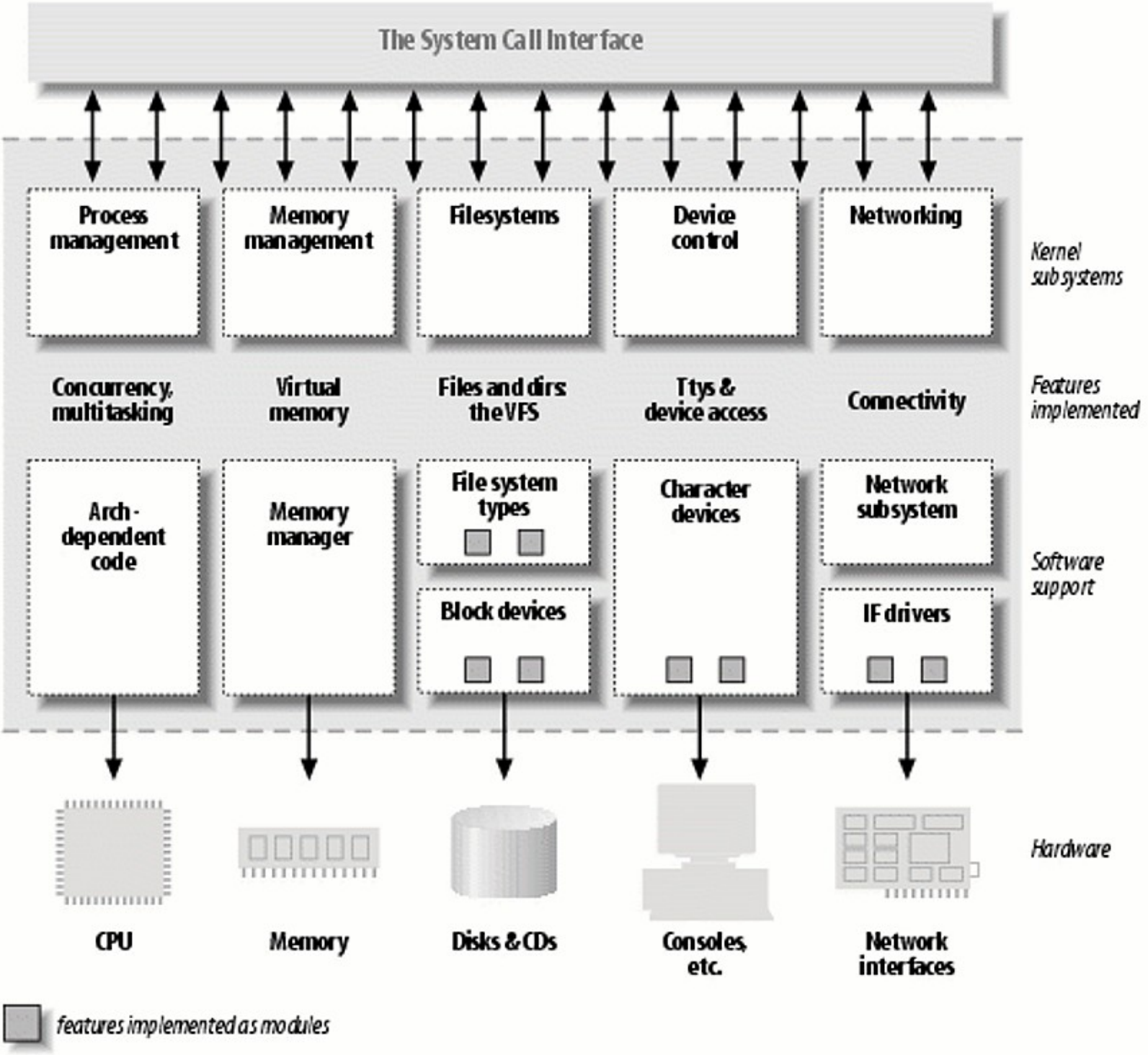
CS5460/6460: Operating Systems

Lecture 24: Device drivers

Anton Burtsev
April, 2014


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Device drivers

- Conceptually
  - Implement interface to hardware
  - Expose some high-level interface to the kernel or applications
    - What this interface should look like?
  - In UNIX everything is a ...

Device drivers

- Conceptually
  - Implement interface to hardware
  - Expose some high-level interface to the kernel or applications
    - What this interface should look like?
  - In UNIX everything is a **file**

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

Classes of devices

- Character
  - Accessed as a stream of bytes
    - Text console, serial ports
    - /dev/console, /dev/tty1
- Block
  - I/O performed in units of blocks
    - Hard disks, CD drives, USB sticks
    - /dev/sda

Classes of devices

- But what about network devices? Graphic cards?
  - No easy file mapping
    - Although it doesn't mean you can't come up with one if it fits your needs
  - Device as a file paradigm doesn't work
    - Well they expose different interfaces
    - Network cards are accessible through sockets

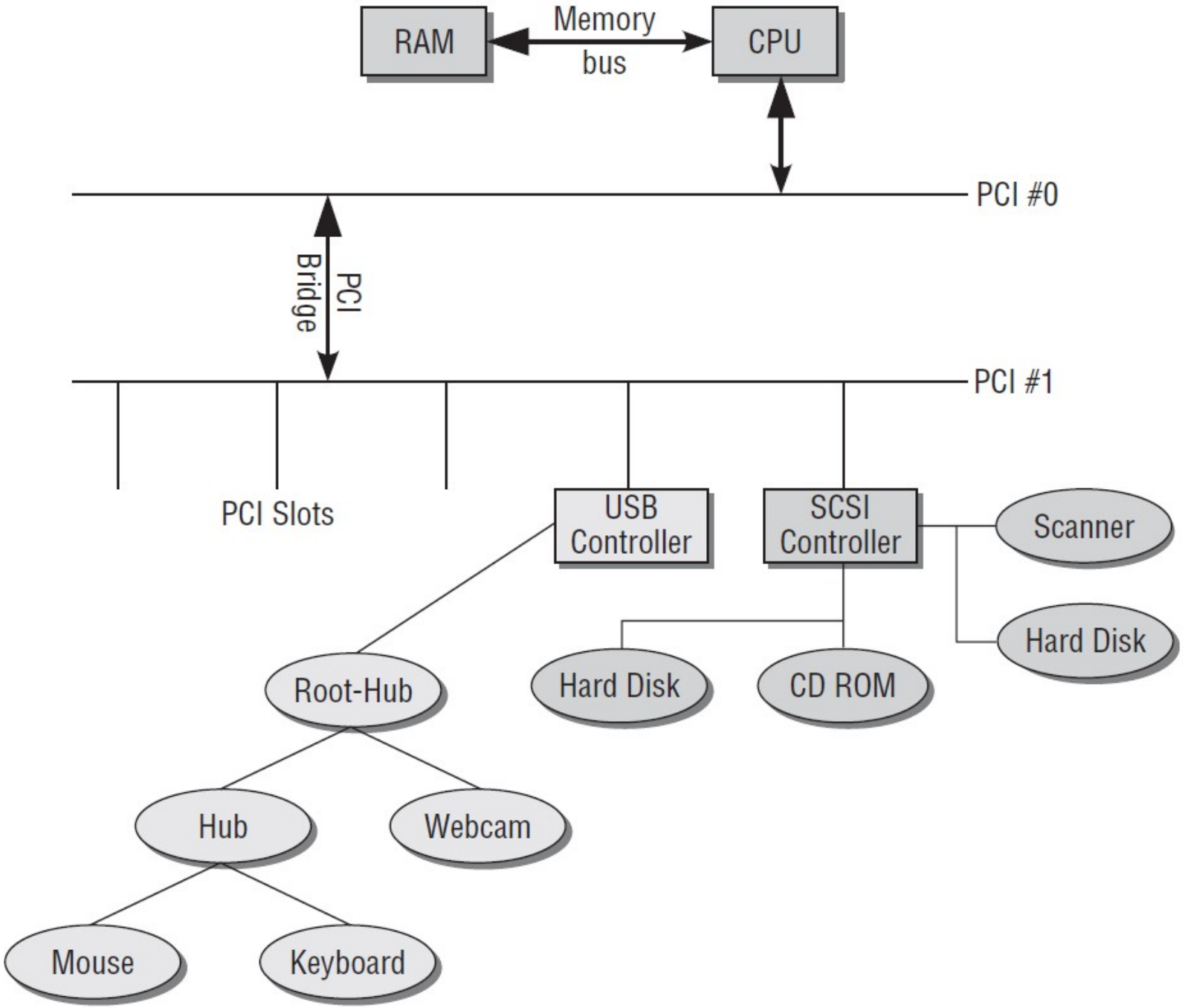
Detour into hardware

Device drivers and hardware

- Device driver doesn't strictly need to talk to hardware
  - /dev/random – stream of random numbers
  - /dev/mem – reads physical memory
  - /dev/null – input goes nowhere

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

- 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 peripherals

- I/O memory mapping
  - Modern CPUs allow mapping port addresses to memory locations
  - Then it's just possible read/write memory
    - GPU devices
    - System buses like PCI

Interacting with peripherals

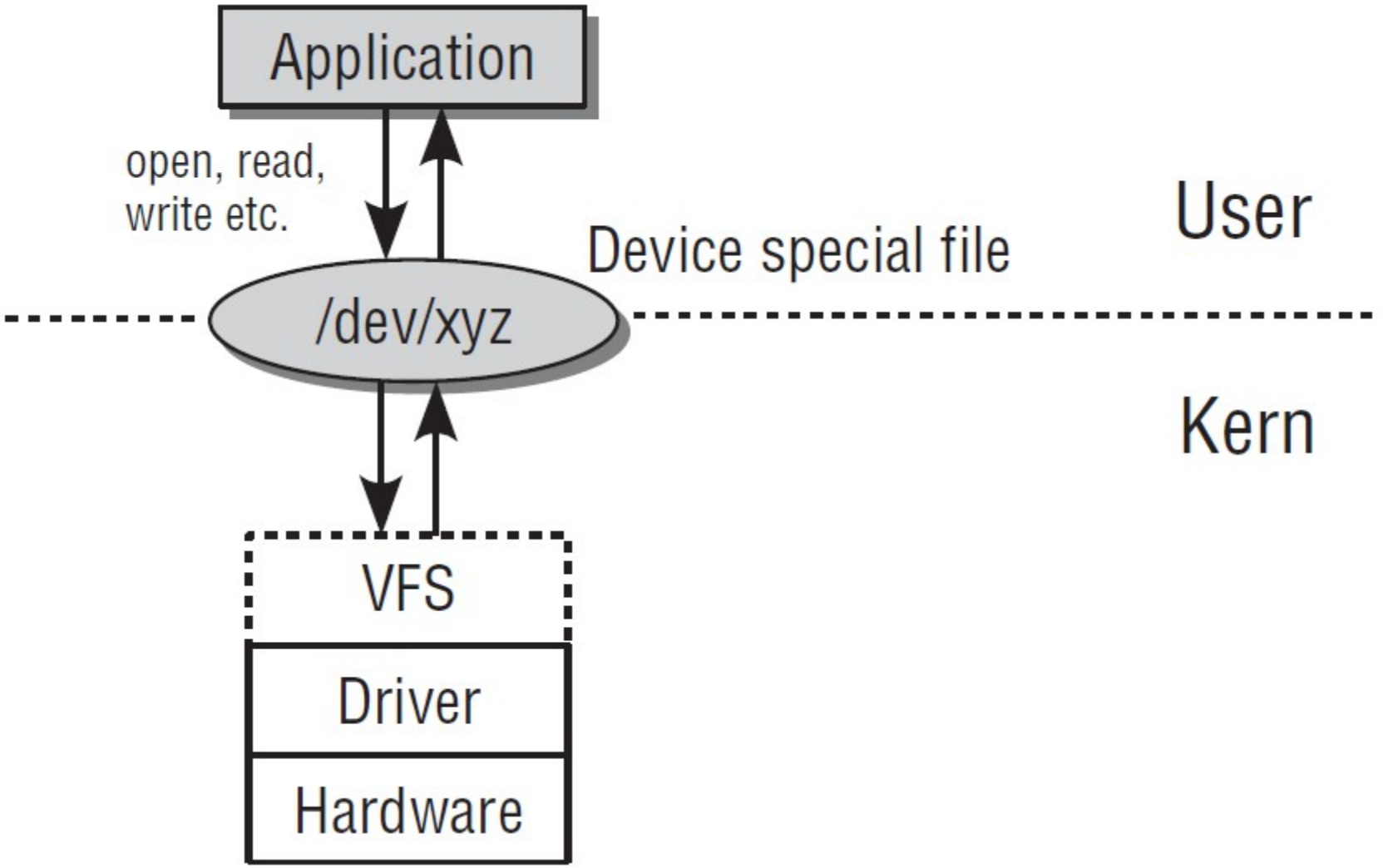
- Interrupts
  - CPU provides several interrupt lines
  - One line can be shared across several devices
- Polling
  - Periodic check of the device state for whether more data is available

Back to the Linux kernel

Linux exports devices as files

- Lets assume you have a modem attached to the serial port
  - echo "ATZ" > /dev/ttyS0
    - Sends initialization string to the modem
- To read your hard drive
  - cat /dev/sda

Device files (/dev/xyz)



Major and minor numbers

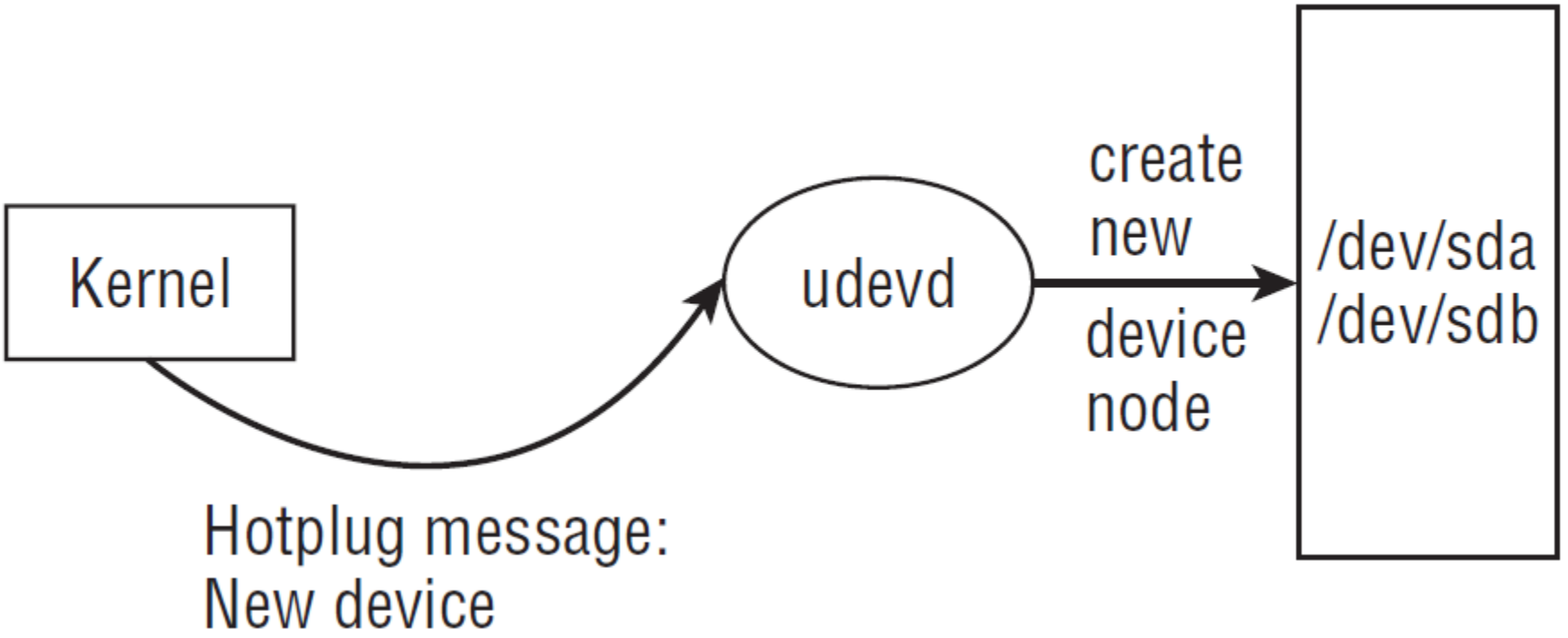
- Each device file has two numbers
  - Identify device driver for this device
  - Major number: device driver
  - Minor number: device number

```
wolfgang@meitner> ls -l /dev/sd{a,b} /dev/ttyS{0,1}
brw-r----- 1 root disk 8,  0 2008-02-21 21:06 /dev/sda
brw-r----- 1 root disk 8, 16 2008-02-21 21:06 /dev/sdb
crw-rw---- 1 root uucp 4, 64 2007-09-21 21:12 ttyS0
crw-rw---- 1 root uucp 4, 65 2007-09-21 21:12 ttyS1
```

/dev

- Back in the days /dev/ was static
  - Now there are 20K device numbers are allocated
  - Most are not used on your system
- Today, /dev/ is a temporary file system
  - All device names are generated on the fly
  - By udevd daemon

udevd



- Udevd listens for hotplug messages from the kernel
  - Creates new device nodes

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

```
static const char    hw_string[] = "Hello world from kernel mode!\n\0";
static const ssize_t hw_size = sizeof(hw_string);

static ssize_t device_file_read(struct file *file_ptr, char __user *user_buffer,
size_t count, loff_t *position) {

  /* If position is behind the end of a file we have nothing to read */
  if( *position >= hw_size )
      return 0;

  /* If a user tries to read more than we have, read only as many bytes as we
have */
  if( *position + count > hw_size )
      count = hw_size - *position;

  if( copy_to_user(user_buffer, hw_string + *position, count) != 0 )
       return -EFAULT;

   /* Move reading position */
   *position += count;
   return count;
}
```

Build, compile...

- New device appears in /proc/devices

```
Character devices:
1 mem
4 tty
4 ttyS
  ...
250 Simple-driver
  ...
```

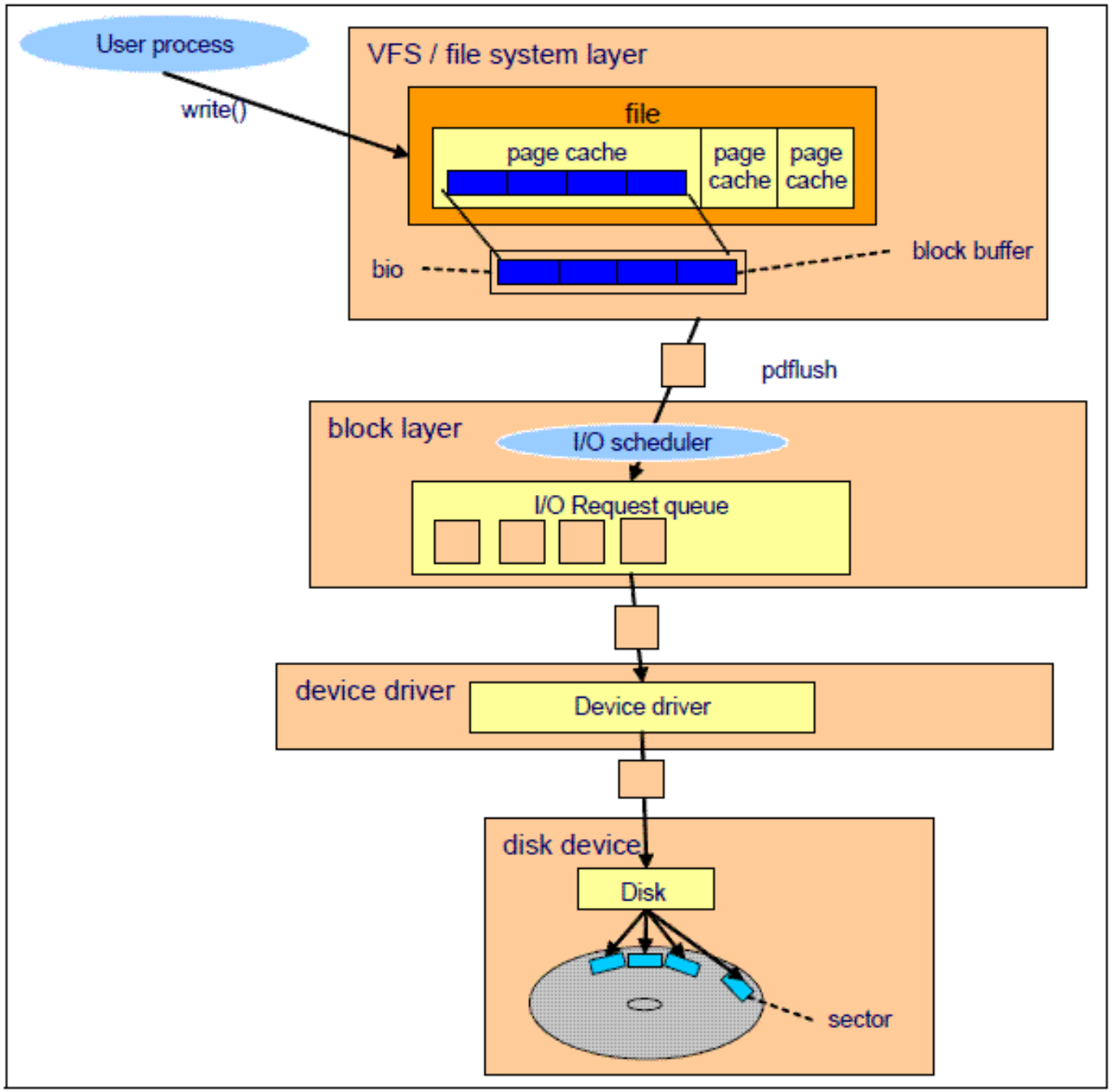
- Make a device file for our device

```
$> mknod /dev/simple-driver c  250 0
```

- Access device

```
$> cat /dev/simple-driver
Hello world from kernel mode!
```

Block devices


User process
write()
VFS / file system layer
file
page cache
page cache
page cache
bio
block buffer
pdflush
block layer
I/O scheduler
I/O Request queue
device driver
Device driver
disk device
Disk
sector

Elevator I/O schedulers

- Original name is after the way of how elevator moves
  - Up or down, picking up passengers on the way
- Same with disk
  - Disk arm moves inside or outside
  - Requests are serviced only in the direction of the arm movement until it reaches the edge

Linux elevators

- Noop
  - First come, first served
- Deadline
  - Assigns a deadline to each request
  - Tries to reorder requests to minimize seek times for requests before deadline
- Anticipatory scheduler
  - Tries to anticipate behavior of a process
  - Assumes that reads are not independent, more reads will follow the initial read
  - Delay seeks for some time anticipating reads to a nearby location
- CFQ (Completely Fair Queuing)
  - Assign each thread a time slice in which it is allowed to submit requests to disk
  - Each thread gets a fair share of I/O throughput

Conclusion

Thank you!