Operating Systems
CSE 231
Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 11: Dec 3 – Dec 7

Overview

- I/O management is a major component of operating system design and operation
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent

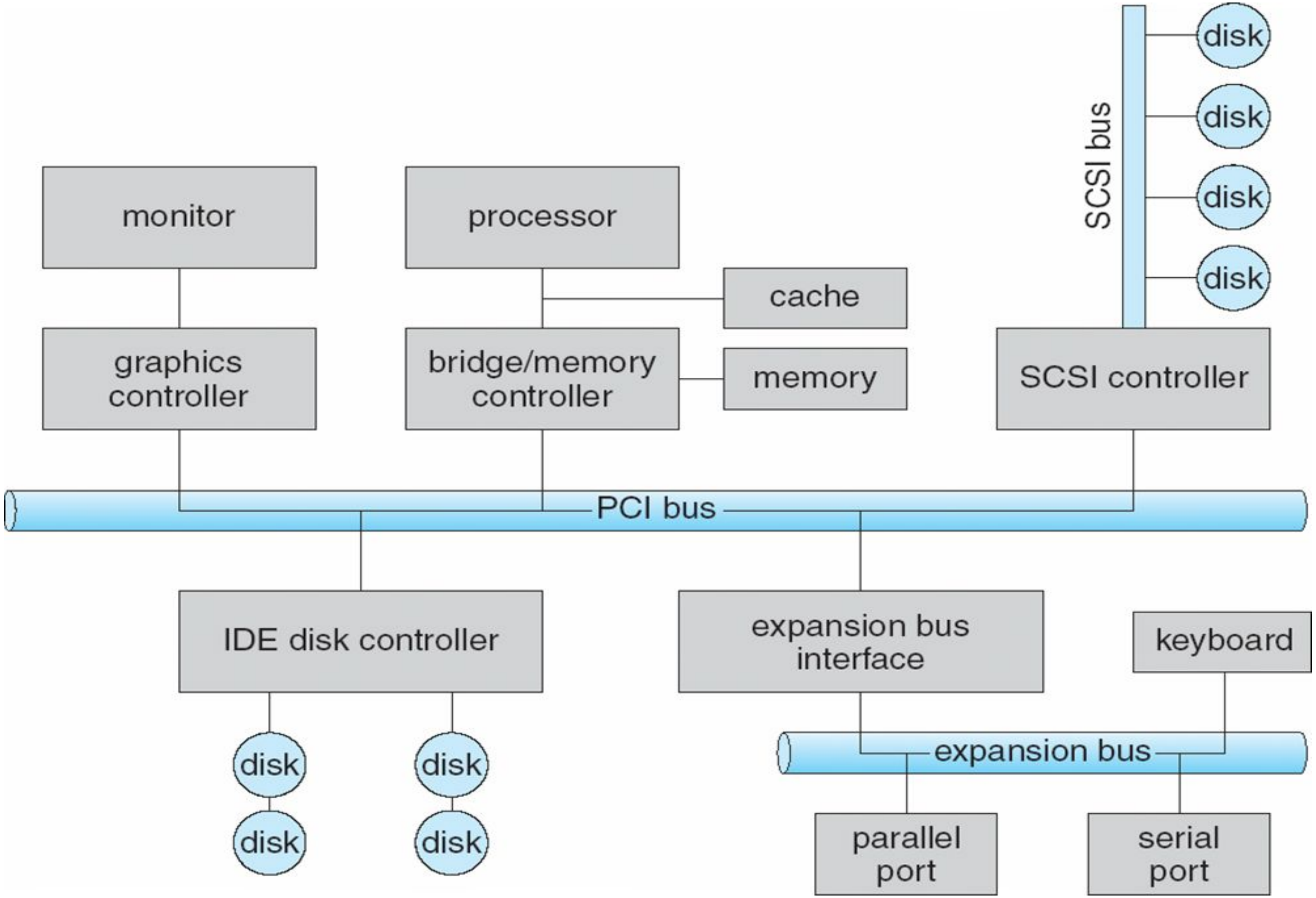
- Ports, busses, device controllers connect to various devices

- **Device drivers** encapsulate device details
  - Present uniform device-access interface to I/O subsystem

I/O Hardware

- Incredible variety of I/O devices
  - Storage
  - Transmission
  - Human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device
  - **Bus** - **daisy chain** or shared direct access
    - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
    - **expansion bus** connects relatively slow devices
  - **Controller** (**host adapter**) – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc
      - Some talk to per-device controller with bus controller, microcode, memory, etc

A Typical PC Bus Structure



I/O Hardware (Cont.)

- I/O instructions control devices
- Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- Devices have addresses, used by
  - Direct I/O instructions
  - **Memory-mapped I/O**
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

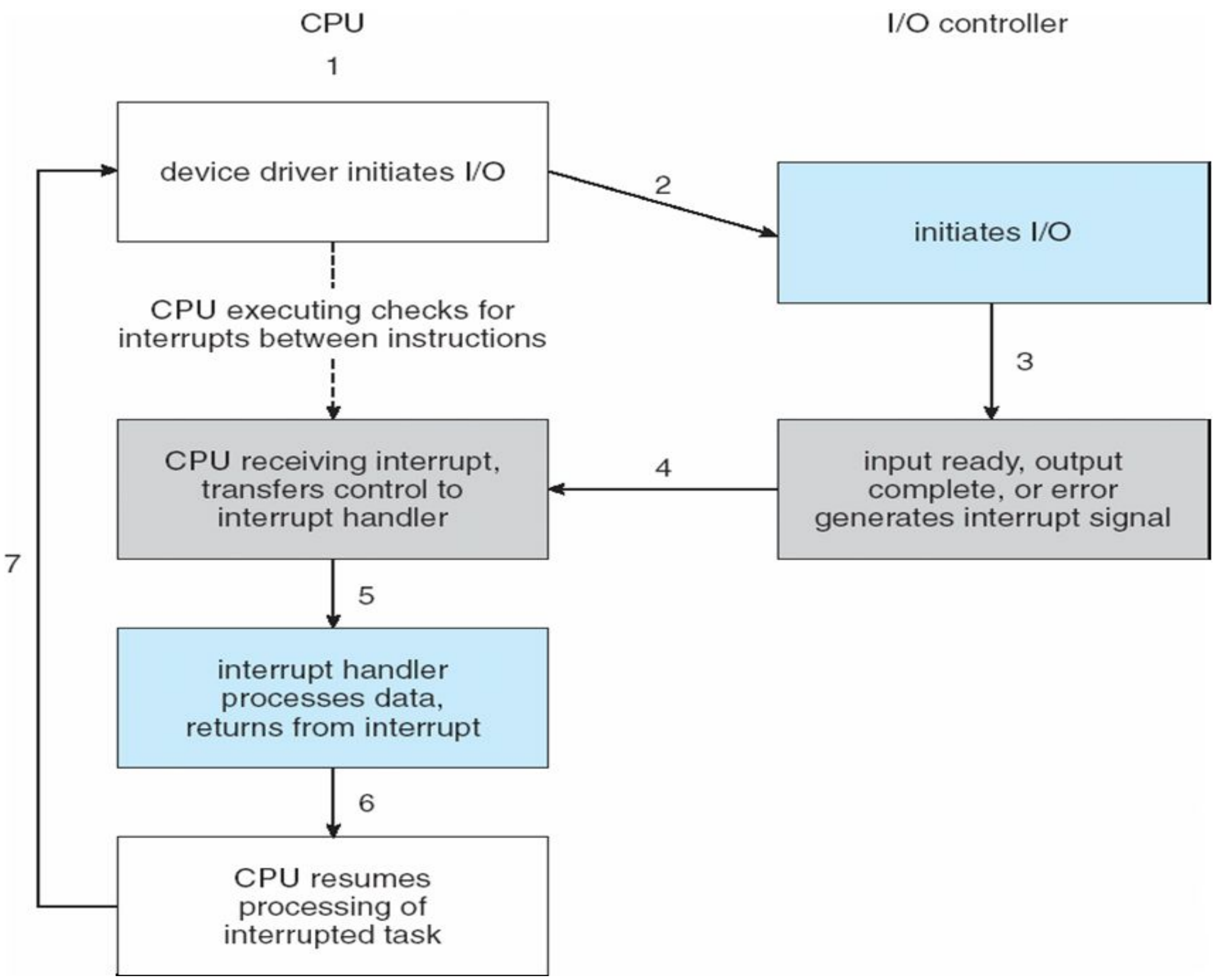
Polling

- For each byte of I/O
  1. Read busy bit from status register until 0
  2. Host sets read or write bit and if write copies data into data-out register
  3. Host sets command-ready bit
  4. Controller sets busy bit, executes transfer
  5. Controller clears busy bit, error bit, command-ready bit when transfer done
- Step 1 is **busy-wait** cycle to wait for I/O from device
  - Reasonable if device is fast
  - But inefficient if device slow
  - CPU switches to other tasks?
    - But if miss a cycle data overwritten / lost

Interrupts

- Polling can happen in 3 instruction cycles
    - Read status, logical-and to extract status bit, branch if not zero
    - How to be more efficient if non-zero infrequently?
- CPU **Interrupt-request line** triggered by I/O device
    - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
    - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
    - Context switch at start and end
    - Based on priority
    - Some **nonmaskable**
    - Interrupt chaining if more than one device at same interrupt number

Interrupt-Driven I/O Cycle



Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

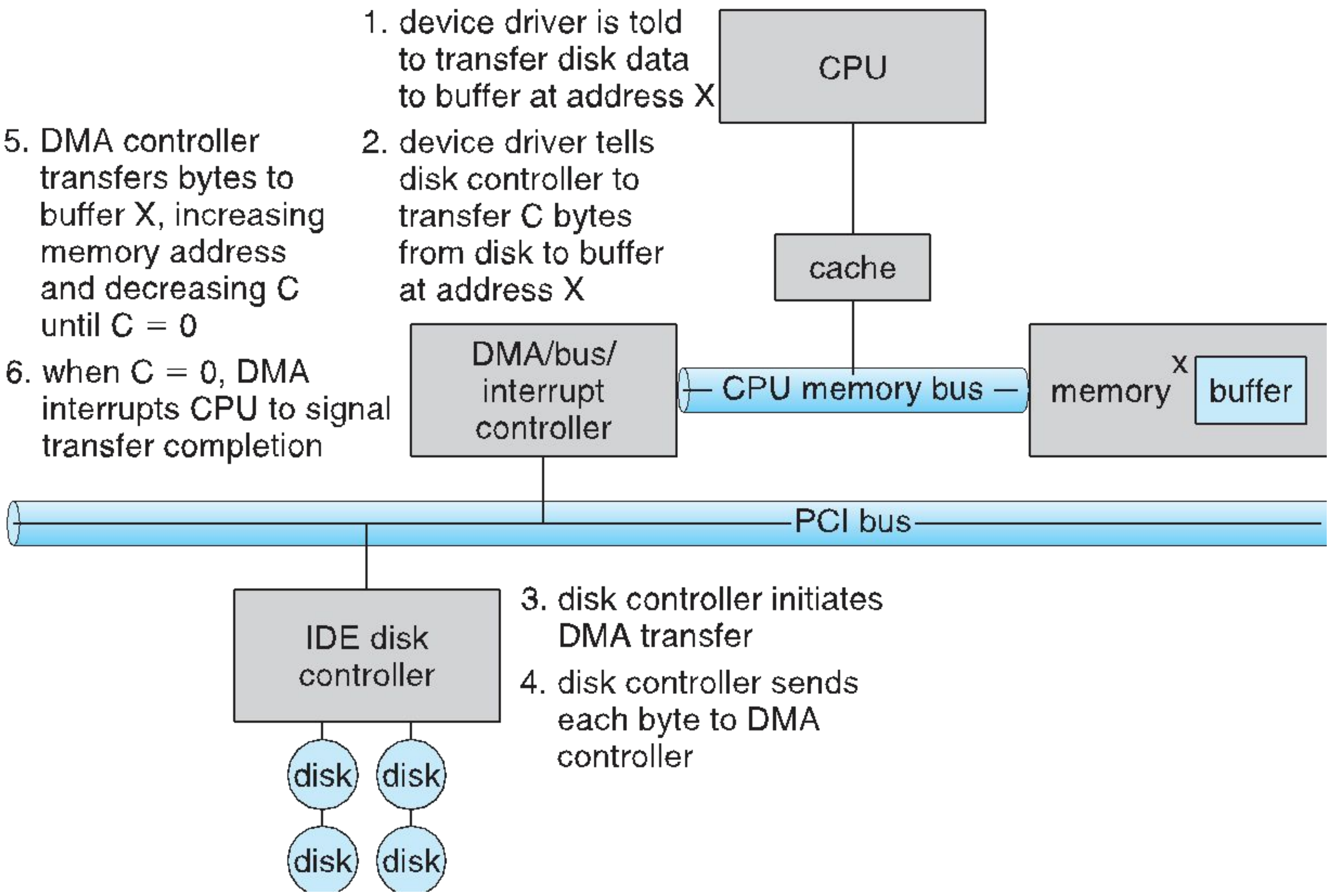
Interrupts (Cont.)

- Interrupt mechanism also used for **exceptions**
  - Terminate process, crash system due to hardware error
- Page fault executes when memory access error
- System call executes via **trap** to trigger kernel to execute request
- Multi-CPU systems can process interrupts concurrently
  - If operating system designed to handle it
- Used for time-sensitive processing, frequent, must be fast

Direct Memory Access

- Used to avoid **programmed I/O** (one byte at a time) for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
    - Source and destination addresses
    - Read or write mode
    - Count of bytes
    - Writes location of command block to DMA controller
    - Bus mastering of DMA controller – grabs bus from CPU
        - **Cycle stealing** from CPU but still much more efficient
    - When done, interrupts to signal completion
- Version that is aware of virtual addresses can be even more efficient - **DVMA**

Six Step Process to Perform DMA Transfer



Interrupts and interrupt handlers

- Interrupt handlers
  - The function the kernel runs in response to a specific interrupt is called an *interrupt handler* or *interrupt service routine* (ISR)
  - The interrupt handler for a device is part of the devices's *driver*
  - The interrupt handler execute in as short a period as possible

Registering an interrupt handler

```
int request_irq (unsigned int irq,
        irqreturn_t (*handler)(int, void *, struct pt_regs *),
        unsigned long irqflags, const char * devname, void *dev_id)
```

- Parameters
  - irq
    - Specifies the interrupt number to allocate
  - handler
    - A pointer to the actual interrupt handler that services this interrupt
  - irqflags
    - IRQF_DISABLED, IRQF_SAMPLE_RANDOM, IRQF_TIMER, IRQF_SHARED
    - devname
    - An ASCII text representation of the device associated with the interrupt
  - dev_id
    - ``Cookie'' used primarily for shared interrupt lines
- Note request_irq() might sleep and, therefore, cannot be called from interrupt context or other situations where code cannot block

Registering an interrupt handler

```
if (request_irq(irqn, my_interrupt, IRQF_SHARED, "my_device", my_dev)) {
        printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
        return -EIO;
}
```

Freeing an interrupt handler

```
void free_irq (unsigned int irq, void *dev)
```

- If the interrupt line is shared, the handler identified via dev is removed but the corresponding interrupt is NOT disabled
- free_irq().

Writing an interrupt handler

```
static irqreturn_t intr_handler (int irq, void *dev_id, struct pt_regs *regs)
```

- Parameter
  - regs
    - Holds a pointer to a structure containing the processor registers and state prior to servicing the interrupt. They are rarely used, except for debugging
- Return value of an interrupt handler
  - IRQ_NONE
    - Detects an interrupt for which its device was not the originator
  - IRQ_HANDLED
    - Correctly invoked, and its device did cause the interrupt

Shared handlers

- A shared handler is registered and executed much like a non-shared handler.
- Three main differences are
  - The IRQF_SHARED flag must be set in the flags argument to request_irq()
  - The dev argument must be unique to each registered handler.
  - The interrupt handler must be capable of distinguishing whether its device actually generated an interrupt

Interrupt context

- When executing an interrupt handler, the kernel is in *interrupt context*
  - Interrupt context is not associated with a process
  - Interrupt context cannot sleep
  - Interrupt context is time critical
  - Interrupt handler does not receive its own stack
    - Instead, it shares the kernel stack of the process or idle task's stack

Implementation of interrupt handling

```
/* register rtc_interrupt on rtc_irq */
if (request_irq(rtc_irq, rtc_interrupt, IRQF_SHARED, "rtc", (void *)&rtc_port)) {
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
        return -EIO;
}
```

Implementation of interrupt handling

```
static irqreturn_t rtc_interrupt(int irq, void *dev)
{
        /*
         * Can be an alarm interrupt, update complete interrupt,
         * or a periodic interrupt. We store the status in the
         * low byte and the number of interrupts received since
         * the last read in the remainder of rtc_irq_data.
         */

        spin_lock(&rtc_lock);

        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

        if (rtc_status & RTC_TIMER_ON)
            mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

        spin_unlock(&rtc_lock);

        /*
         * Now do the rest of the actions
         */
        spin_lock(&rtc_task_lock);
        if (rtc_callback)
                rtc_callback->func(rtc_callback->private_data);
        spin_unlock(&rtc_task_lock);
        wake_up_interruptible(&rtc_wait);

        kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);

        return IRQ_HANDLED;
}
```

Interrupt control

- Reasons to control the interrupt system generally boil down to needing to provide *synchronization*
  - Kernel code more generally needs to obtain *some sort of lock* to prevent access to shared data simultaneously from another processor
  - These locks are often obtained in conjunction with *disabling local interrupts*

Disabling and Enabling interrupts

- To disable interrupts locally for the current processor (and only the current processor) and later enable them:

```
local_irq_disable();
/* interrupts are disabled */
local_irq_enabled();
```

- Problem is you don't know if the interrupt was already enabled or disabled to begin with.
- Save and restore interrupt states.

```
local_irq_save(flags); /*Save state and disable */
local_irq_restore(flags); /* Revert to previously saved state */
```

Disabling and Enabling Interrupts Locally

- To disable interrupts locally for the current processor (and only the current processor) and later enable them:

```
local_irq_disable();
/* interrupts are disabled */
local_irq_enabled();
```

- Problem is you don't know if the interrupt was already enabled or disabled to begin with.
- Save and restore interrupt states.

```
local_irq_save(flags); /*Save state and disable */
local_irq_restore(flags); /* Revert to previously saved state */
```

Disabling and Enabling Interrupts Globally

- Enabling and disabling interrupts for the entire system – across all CPUs.

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

- disable_irq()function does not return until any currently executing handler completes.
- The function disable_irq_nosync() does not wait for current handlers to complete.
- The function synchronize_irq() waits for a specific interrupt handler to exit, if it is
- executing, before returning.

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- New devices talking already-implemented protocols need no extra work
- Each OS has its own I/O subsystem structures and device driver frameworks
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Synchronous** or **asynchronous** (or both)
  - **Sharable** or **dedicated**
  - **Speed of operation**
  - **read-write, read only,** or **write only**

Linux Block I/O Layer

- Two main kinds of I/O devices – character vs block.
- Character devices – read/write data in sequence ; could be multiple bytes (not necessarily a single character at a time). E.g. USB data, printer, keyboard etc.
- Block devices – Read in chunks (always multibyte), chunks aka blocks, data indexed on blocks. Possibility of going ``back and forth'' unlike character devices – e.g. hard disk/CD-ROMS/DVDs etc.

BIO structure

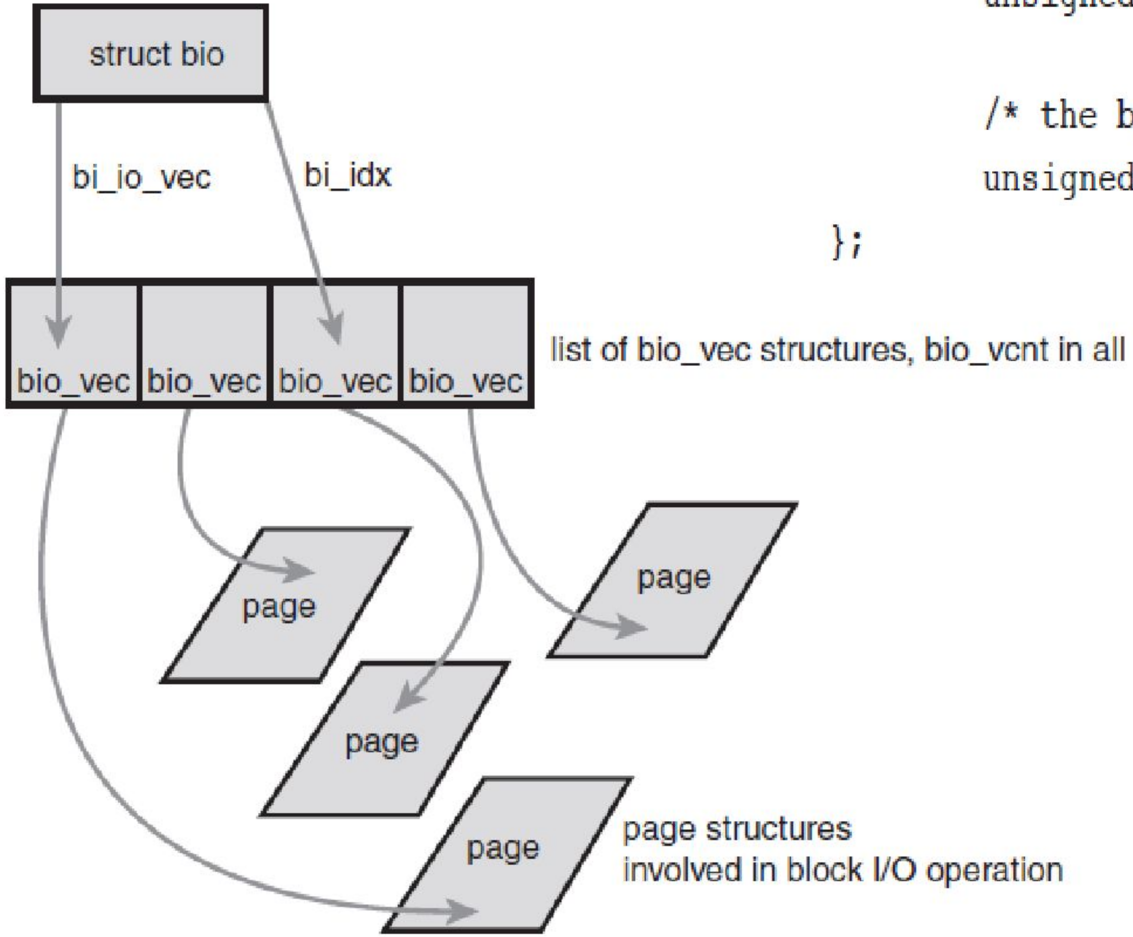
- This structure represents block I/O operations that are in flight (active) as a list of *segments* (a contiguous chunk of memory).
- The bio structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory.

BIO structure

```
struct bio_vec {
        /* pointer to the physical page on which this buffer resides */
        struct page     *bv_page;

        /* the length in bytes of this buffer */
        unsigned int    bv_len;

        /* the byte offset within the page where the buffer resides */
        unsigned int    bv_offset;
};
```



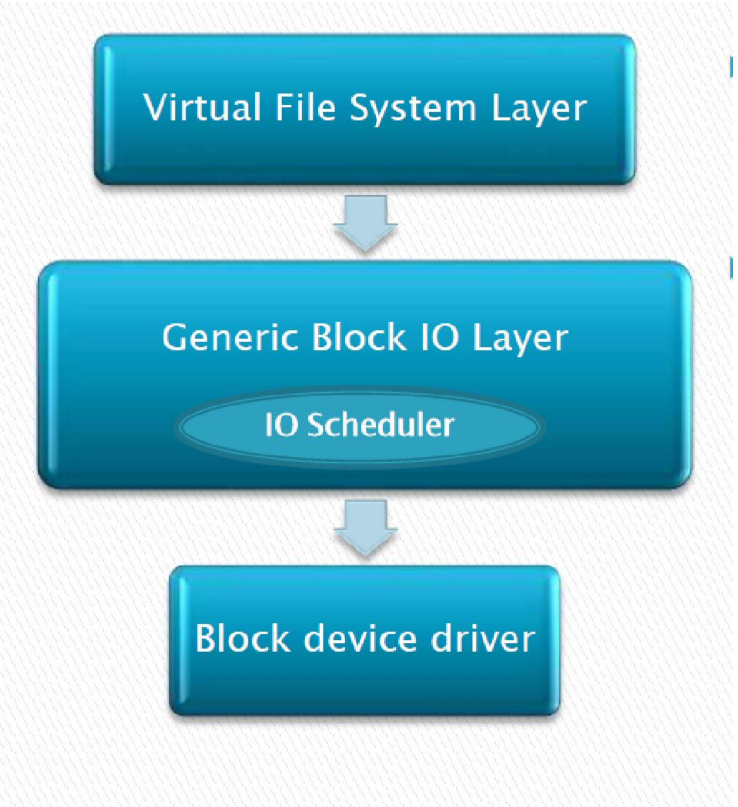
Block I/O Schedulers

- It is the subsystem of the kernel which performs “merging and sorting” on block I/O requests to improve the performance of thesystem.
- Process Scheduler, which share the processor among the processes on system.

◦ I/O Scheduler and Process Scheduler are two different subsystems in Kernel.

- I/O Scheduler Goal:
  - Minimize disk seeks.
  - ensure optimum disk performance and
  - provide fairness among IO requests

Block I/O Subsystem



- Block IO layer
  - receives IO requests from FileSystem layer
  - Maintains IO requests queue
- I/O Scheduler
  - schedules these IO requests and decides the order of the requests.
  - It selects one request at a time and dispatches it to block device.
  - It perform two actions on request queue
- Merging
- Sorting
- Manages the request queue with goal of reducing seeks and improving the throughput

I/O Scheduler

- Primary actions: Merging and Sorting
- Merging
  - Coalescing of two or more requests into one request operating on one or more adjacent on-disk sectors.
  - Reduces the overhead of multiple requests into a single request
  - Minimizes the seek operations
- Sorting
  - Keeps request queue sorted, sector-wise
  - Minimizes the individual seek, keeping the disk head moving in straight line.

Adding Request to the Request Queue (Traditional Linus Elevator – kernel 2.4)

- Four operations are possible if any new request to be added into the request queue.
  - If a request to an adjacent on-disk sector is in the queue, the existing request and the new requests are merged into single request.
  - If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue to prevent starvation of the other, older, requests.
  - If there is a suitable location sector-wise in the queue, the new request is inserted there. Keeping the queue sorted by physical location on the disk.
  - Insert new request at the tail of the queue if above scenarios not met.

Deadline I/O Scheduler (DIS)

- Each request is associated with an expiration time.
  - 500 milli seconds for read requests
  - 5000 milli seconds for write requests
- Maintains 3 queues: Normal sorted queue, read requests queue, write requests queue.
- Performs merging/sorting on sorted queue when new request comes.
  - New request is also inserted into either read queue or write queue depends on the type of requests read or write.
- DIS pulls the request from sorted queue and dispatched it to device driver.
  - If any request from read/write queue expires then DIS pulls the request from these queues.
- Ensures that no requests are processes on or before expiration time.

Anticipatory I/O Scheduler

- Deadline I/O scheduler minimizes the read latency (since more preference has given to read requests) but it compromise on throughput - considering a system undergoing heavy write activity.
- AIS = DIS + Anticipation Heuristic.
- After the request is submitted to device driver, AIS sits idle for few milliseconds (default 6 milliseconds), thinking that there is a good chance of receiving new read request which is adjacent to the submitted request. If so, this newly request is immediately served.
- After waiting period elapses, it continue to pull request from request queues similar to DIS.
- Ideal for servers
  - Perform very poorly on certain uncommon workloads invloving seek-happy databases.

CFQ Scheuler

- Designed for specialized workloads.
- Different from DIS and AIS schedulers
- CFQ maintains one sorted request queue for each process submitting IO requests.
- CFQ services these queues in Round Robin fashion and selects configurable number of requests (default 4) from each queue.
- Provides fairness at a per-process level
- Intended workload is Multimedia and recommended for desktop workloads.

Noop I/O Scheduler

- It performs only merging and does not perform sorting.
- It is intended for block devices that are truly random-access, such as flash memory cards.
- If a block device has a little or no overhead associated with “seeking”, then noop I/O Scheduler is ideal choice.

Conclusion

- Block devices uses the Anticipatory I/O Scheduler by default.
- Select CFQ for desktop/multimedia workloads
- Select Noop for block devices which are truly random access (flash memory cards) and for block devices which doesn’t have seeking overhead.

Kernel Modules

- Are you tired of waiting for your kernel to compile?
- Kernel modules provide a way to quickly modify a running kernel.
- They can be separately compiled and be dynamically added and be removed.
- When added they become part of the kernel with access to the rest of the kernel.

Kernel Module Structure

- Kernel modules consist of
  - An initialization routine that is called when the module is loaded
  - An exit routine that is called when the module is removed.
  - Functions and variables that can be exported for use by other parts of the kernel, including other modules.
  - Meta data that can be accessed by tools and the kernel.

Applications of Modules

- Proc files
- Device drivers
- Interrupt Handlers
- File systems
- System calls
- Monitoring and replacing core functionality such as scheduling

Sample Module

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>

int init_my_module(void);
void exit_my_module(void);

int init_my_module()
{
  printk(KERN_ALERT "my_module\n");
  return 0;
}

void exit_my_module()
{
  printk("exiting my module");
}

/* Example exported function */
int my_function(int arg1)
{
  printk("my_function\n");
  return 0;
}

EXPORT_SYMBOL(my_function1);

module_init(init_my_module);
module_exit(exit_my_module);

MODULE_LICENSE("GPL");
```

Development Modes

- Standalone
  - Work in separate directory

- Integrated
  - Put module code in source code tree
    - Pick appropriate directory and place module code there
    - Add line (obj-m += …) to Makefile in that directory (see info make for how to set variables, and the kbuild documentation on configuration.)

Standalone Makefile

```
# Makefile for kernel module development

obj-m := my_module_one.o
obj-m += my_module_two.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Loading and Unloading Modules

- Commands
  - insmod - inserts module
  - rmmod - removes module
  - lsmod - lists modules
  - depmod - control dependencies
  - modinfo - display module meta data