CSE 451: Operating Systems
Winter 2012

I/O System

Gary Kimura

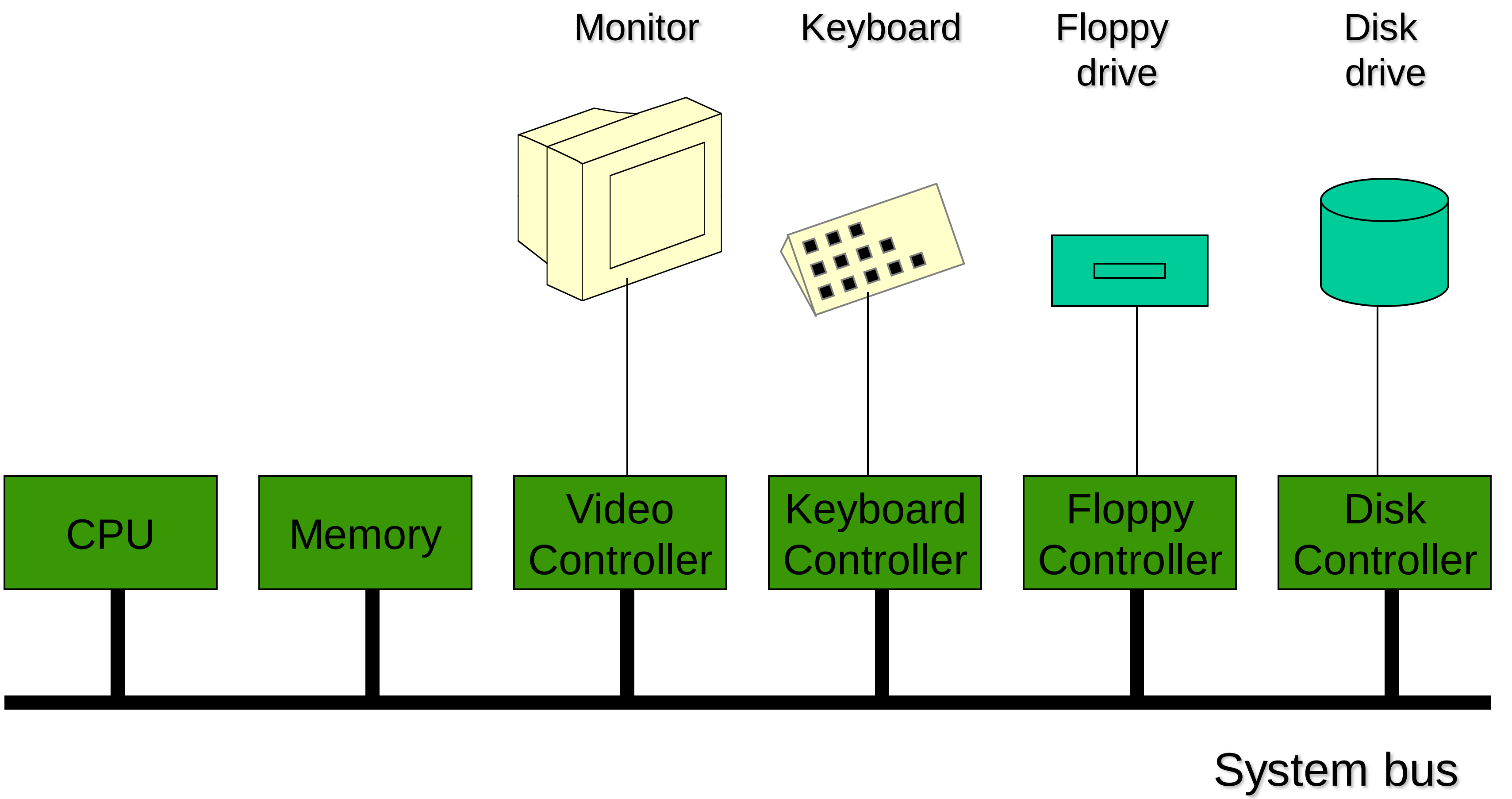
What’s Ahead

- Principles of I/O Hardware
- Structuring of I/O Software
- Layers of an I/O System
- Operation of an I/O System

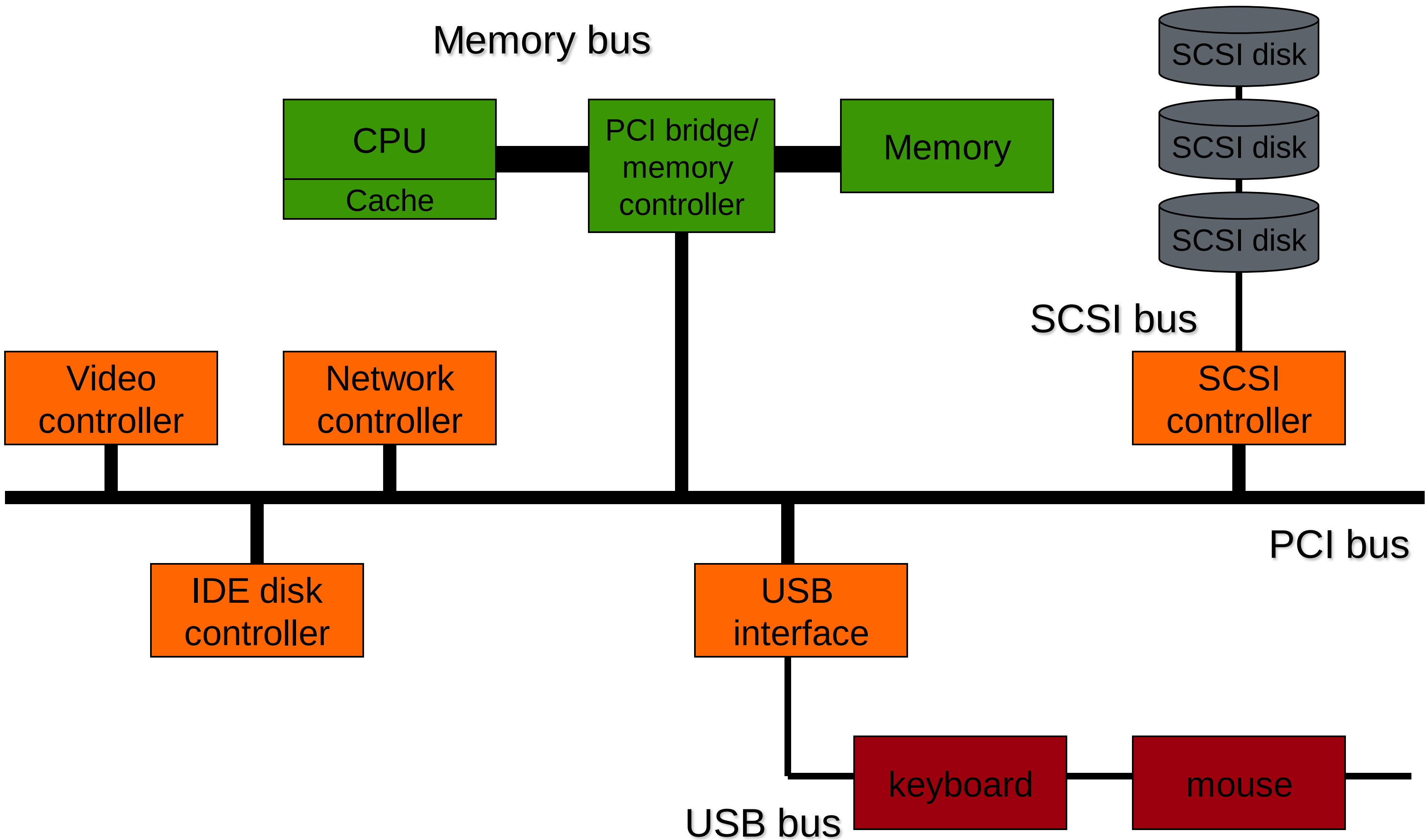
Hardware Environment

- Major components of a computer system:
  CPU, memories (primary/secondary), I/O system
- I/O devices:
  - Block devices – store information in fixed-sized blocks;
    typical sizes: 128-4096 bytes
  - Character devices – delivers/accepts stream of characters (bytes)
- Device controllers:
  - Connects physical device to system bus (Minicomputers, PCs)
  - Mainframes use a more complex model:
    Multiple buses and specialized I/O computers (I/O channels)
- Communication:
  - Memory-mapped I/O, controller registers
  - Direct Memory Access - DMA

I/O Hardware - Single Bus



I/O Hardware - Multiple Buses



Diversity among I/O Devices

The I/O subsystem has to consider device characteristics:

- Data rate:
  - may vary by several orders of magnitude
- Complexity of control:
  - exclusive vs. shared devices
- Unit of transfer:
  - stream of bytes vs. block-I/O
- Data representations:
  - character encoding, error codes, parity conventions
- Error conditions:
  - consequences, range of responses
- Applications:
  - impact on resource scheduling, buffering schemes

Organization of the I/O Function

- Programmed I/O with polling:
  - The processor issues an I/O command on behalf of a process
  - The process busy waits for completion of the operation before proceeding
- Interrupt-driven I/O:
  - The processor issues an I/O command and continues to execute
  - The I/O module interrupts the processor when it has finished I/O
  - The initiator process may be suspended pending the interrupt
- Direct memory access (DMA):
  - A DMA module controls exchange of data between I/O module and main memory
  - The processor requests transfer of a block of data from DMA and is interrupted only after the entire block has been transferred

Flow of a blocking I/O request

1. Thread issues blocking read() system call
2. Kernel checks parameters; may return buffered data and finish
3. Idle device: Driver allocates kernel buffer; sends command to controller
4. Busy device: Driver puts I/O request on device queue
5. Thread is removed from run queue; added to wait queue for device
6. Interrupt occurs; handler stores data; signals device driver to release first thread on device wait queue
7. Handler takes next request from queue, allocates kernel buffer; sends command to controller
8. Awoken thread is in device driver, cleans up
9. Thread resumes execution at completion of read() call

Flow of an asynchronous I/O request

1. Thread issues readasync() system call with synchronization object
2. Kernel checks parameters; may return buffered data immediately, signal synchronization object and finish
3. I/O request is scheduled (initiated on hardware or queued in device driver if busy)
4. Thread returns from readasync()
5. Thread continues, and eventually issues wait(synchronization object)
6. Interrupt occurs, driver retrieves data from hardware if necessary (PIO)
7. Interrupt code starts next request, if any
8. Interrupt code calls wakeup(synchronization object)
9. Interrupt code returns

Only a slight difference from blocking call: use process's synchronization object

But what code really can run during interrupts?

Interrupt-time code

- Kernel/user interruptions occur at arbitrary points
  - Inconsistent data (linked lists not set up correctly, data structures in transition)
  - What’s the *least* that can be counted on?
  - MM? No.
- Kernel needs to deliver an environment where *efficient/effective* processing can be performed
  - Unix: scheduler is the only thing available. The interrupt code will wakeup() the thread that is awaiting service. Some drivers will be able to start next request at this time.
  - Windows: scheduler is available but also means for enqueueing DPC/APC (Deferred Procedure Call/Asynchronous Procedure Call)

DPC/APC – What?

- An architecture for executing a body of code in a clean environment *without a context switch*.
- Kernel has notion of IRQL (I/O Request Level).
  - Interrupts from hardware have certain priorities: timer, disk, keyboard/mouse
  - IRQL is used to mask lower levels so that timely/correct responses can be made; interrupts with lower priority are held off until IRQL is lowered
  - Control is arbitrated through PIC
  - IRQL is union of hardware and software interrupt events: DPC and APC are lower priority than HW interrupts
  - KeRaiseIrql() and KeLowerIrql()

IRQLs

- Example:
  - Power Fail
  - Inter-processor interrupt
  - Clock
  - Device N
  - Device N-1
  - ..
  - Device 0
  - DPC/Dispatch
  - APC
  - Passive (aka running user code)

DPC – deferred procedure call

- A DPC procedure is called in an environment that allows calling scheduler primitives (wake()), access timers, reschedule when quantum expires
- Executes in the current thread when IRQL is lowered sufficiently.
- Used by device drivers to minimize the amount of work performed during H/W interrupt. Why?
- Cannot block! (not touch paged-out memory, take spinlocks, etc)

APC – Asyncrhonous Procedure Call

- “Lower priority interrupt” than DPC. Only executes when no other pending DPCs exist
- Can execute at “current” thread or “that” thread.
- Has full range of kernel services (I/O, MM, synchronization, etc).

Unix I/O Device Interrupt Processing

1. Interrupt occurs, interrupt handler saves state
2. Wakes up thread that was waiting on I/O
3. Selects next request to process
4. Wakes up corresponding thread A
5. Returns from interrupt
6. …
7. Context switch to thread A
8. Issue commands to device
9. Waits on completion
10. Context switch to …

Windows I/O Device Interrupt Processing

1. Interrupt occurs, interrupt handler saves state
2. Enables DPC
3. Returns from interrupt
4. DPC executes
5. Wakes up thread waiting on I/O
6. Enables APC in current thread
7. Exits DPC
8. APC executes
9. Selects next request to process
10. Issue commands to device
11. Exits APC
12. NO CONTEXT SWITCHES!

Principles of I/O Software

- Layered organization
- Device independence
- Error handling
  - Error should be handled as close to the hardware as possible
  - Transparent error recovery at low level
- Synchronous vs. Asynchronous transfers
  - Most physical I/O is asynchronous
  - Kernel may provide synchronous I/O system calls
- Sharable vs. dedicated devices
  - Disk vs. printer

Structuring of I/O software

1. User-level software
2. Device-independent OS software
3. Device drivers
4. Interrupt handlers

Interrupt Handlers

- Should be hidden by the operating system
- Every thread starting an I/O operation should block until I/O has completed and interrupt occurs (OS with no async system calls)
- Interrupt handler transfers data from device (controller) and un-blocks process

Device Driver

- Contains all device-dependent code
- Handles one device
- Translates abstract requests into device commands
  - Writes controller registers
  - Accesses mapped memory
  - Queues requests
- Driver may block after issuing a request:
  - Interrupt will un-block driver (returning status information)

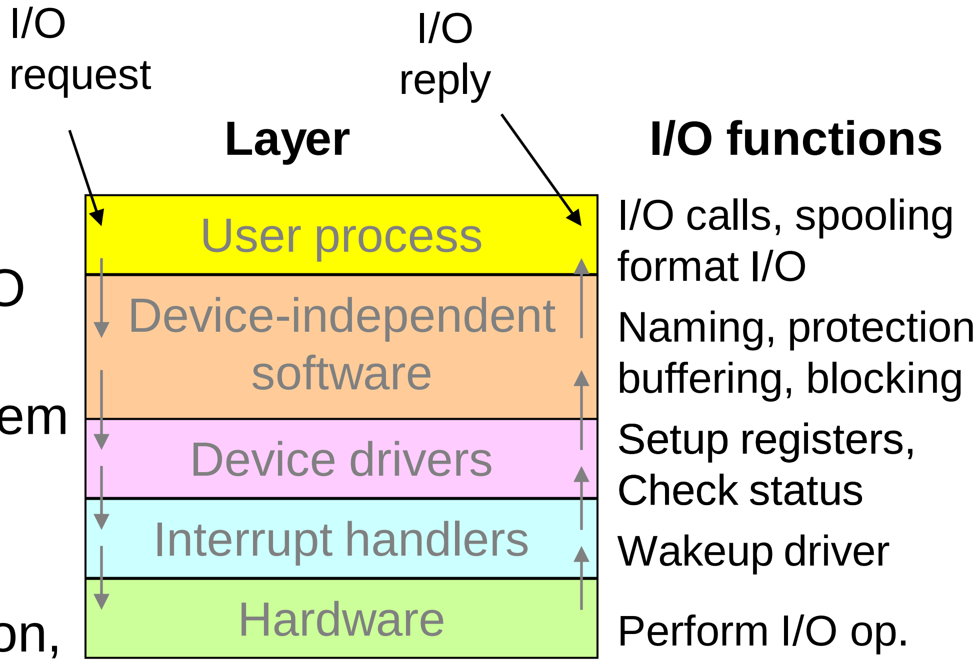
Device-independent I/O Software

Functions of device-independent I/O software:

- Uniform interfacing for the device drivers
- Device naming
- Device protection
- Providing a device-independent block size
- Buffering
- Storage allocation on block devices
- Allocating and releasing dedicated devices
- Error reporting

Layers of the I/O System

- User-Space I/O Software
- System call libraries (read, write,...)
- Spooling
  - Managing dedicated I/O devices in a multiprogramming system
  - Daemon process, spooling directory
  - lpd – line printer daemon, sendmail – simple mail transfer protocol



Application I/O Interfaces

The OS system call interface distinguished device classes:

- Character-stream or block
- Sequential or random-access
- Synchronous or asynchronous
- Sharable or dedicated
- Speed of operation
- Read/write, read only, write only