CS416 – I/O

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)

I/O Devices

So far we have talked about how to abstract and manage CPU and memory

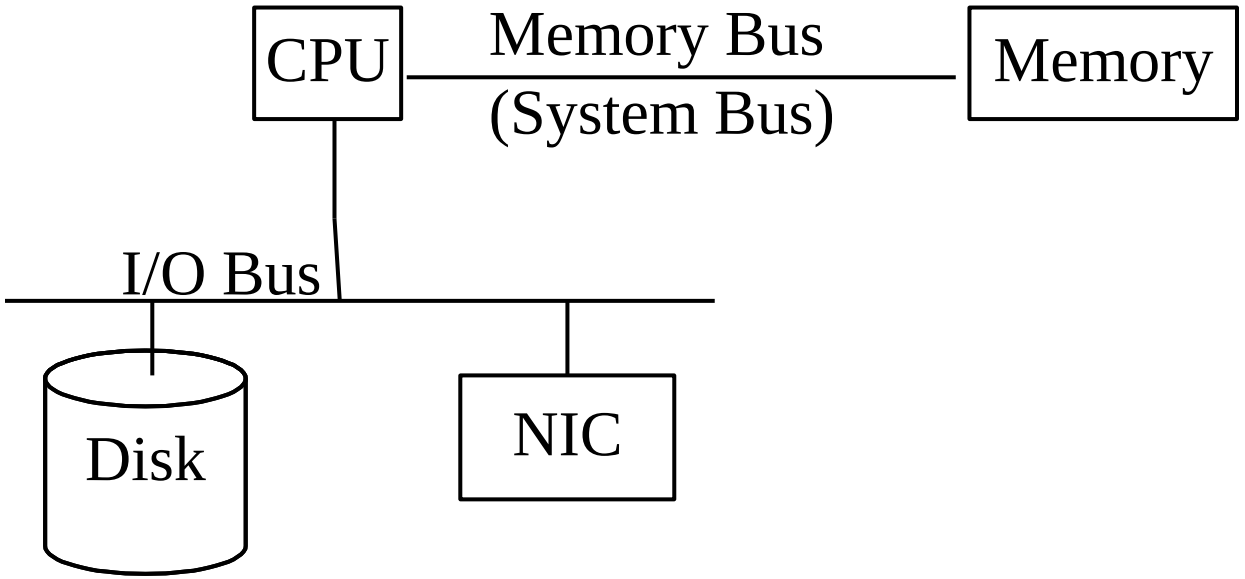
Computation “inside” computer is useful only if some results are communicated “outside” of the computer

I/O devices are the computer’s interface to the outside world (I/O ≡ Input/Output)

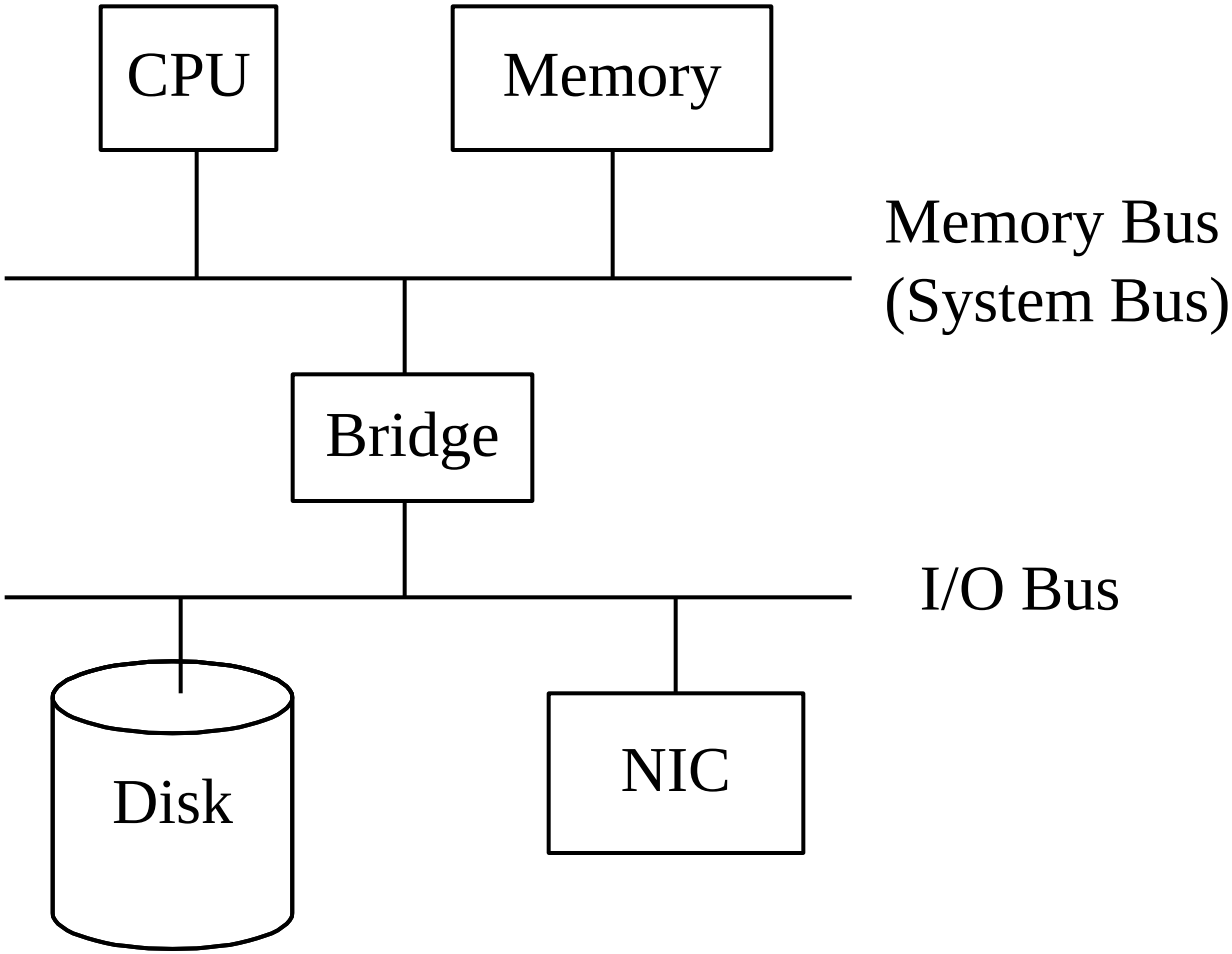
Example devices: display, keyboard, mouse, speakers, network interface, and disk

Hypothetical Architecture !

- Bus: Physical Support for Communication
  - I have pretended that the I/O bus directly communicates with CPU
- I/O Addressing:
  - I/O devices now have a separate addressing (not same as memory)
  - Requires special instruction to load and store in I/O address space
  - Adds complexity to the design



Memory Mapped I/O Addressing



Memory Mapped I/O Addressing

Most I/O devices can be treated by

- write: send them commands and data
- read: receive acknowledgements and data from them

A common access model is ‘just like memory’: Memory-mapped I/O

- I/O device ‘occupies’ a block of main memory addresses
- a simple I/O device (e.g. printer) needs just two ports:
  - one port address for a command buffer
  - one port address for a data buffer
- writing to command buffer port addr sends command
- writing to data buffer port addr sends data
- reading from data buffer port addr accepts data
- makes some main memory addr space inaccessible
- ‘shadowed’ by the I/O address space
- nowadays, impact is minimal (we have lots of memory)

Device I/O Port Locations

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)

How does the Communication between CPU – I/O happen ?

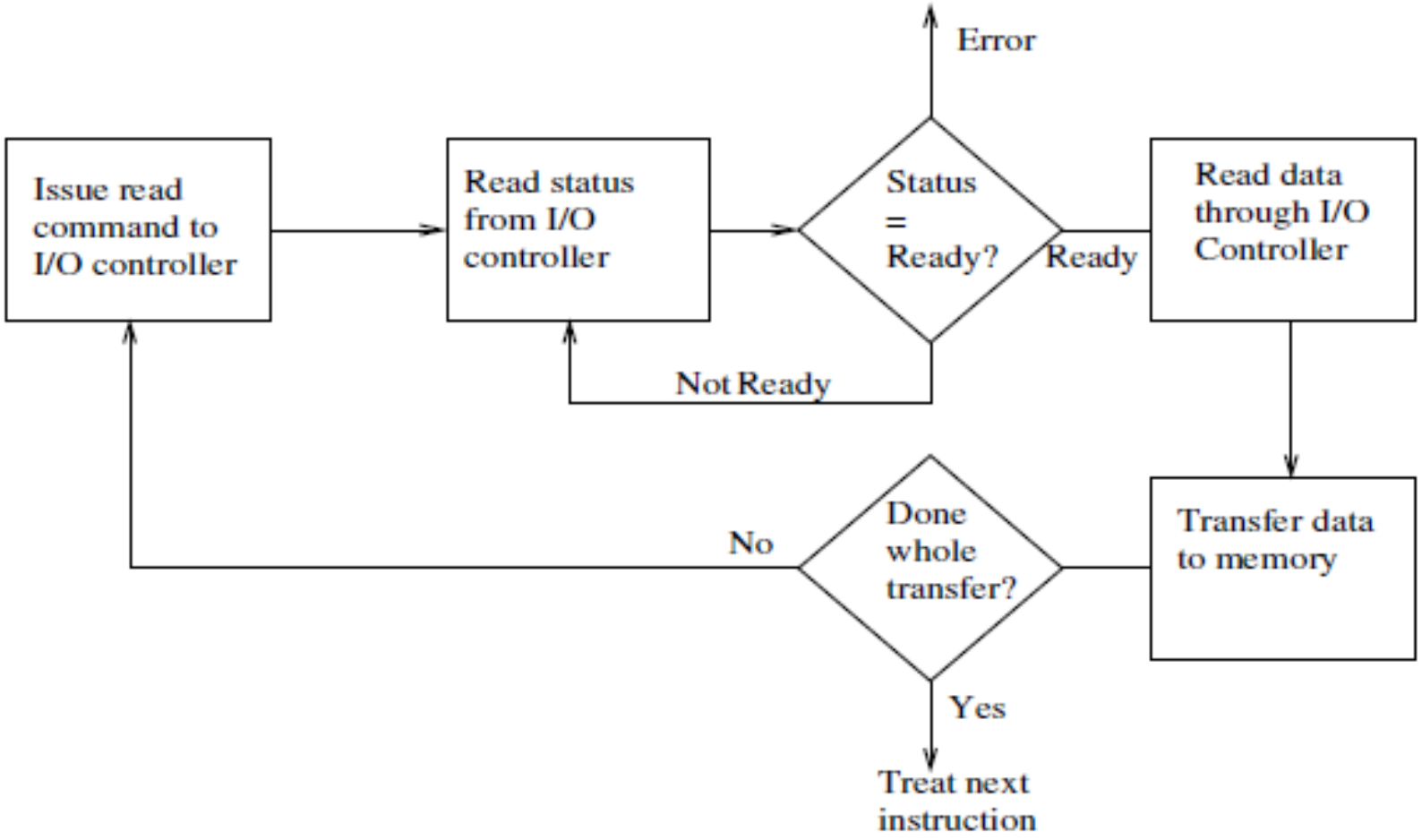
I/O Device Communication Protocols

There are 3 main protocols in use for communicating with I/O devices:

- Programmed I/O (PIO)
  - periodically check state of I/O device
  - usually wastes a lot of CPU time
  - mouse likely will not have moved at all between polls
- Interrupt-driven I/O
  - the CPU issues an I/O instr and carries on other work
  - when completed, device issues interrupt request to CPU
- Direct Memory Access (DMA)
  - separate controller handles I/O without CPU involvement
  - CPU shifts all I/O requests to the DMA handler

Programmed I/O (PIO)

Busy-wait cycle to wait for I/O from device



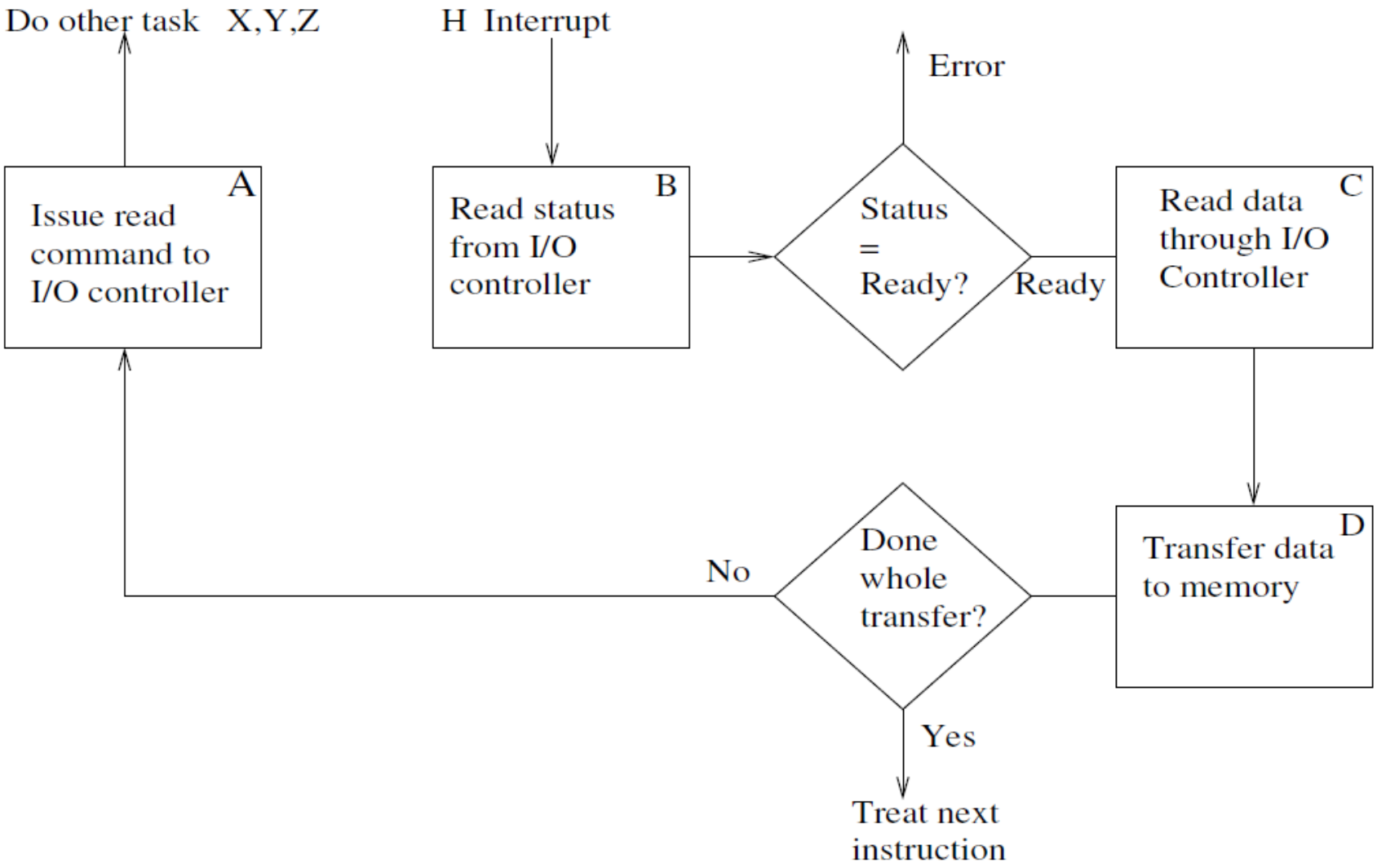
Interrupt Driven I/O

Each peripheral has interrupt request line (IRQ)

- instead of a program polling peripheral for completion , peripheral tells the CPU when it is finished
- peripheral raises interrupt on its IRQ line
- CPU launches interrupt handler code in response
- response initiated by CPU!
  - running program is ‘interrupted’
  - CPU state is saved
  - handler runs
  - CPU state is restored
  - previously running program continues
- IRQs checked at end of each Fetch-Decode-Execute cycle
  - when an instruction execution has just been completed
  - no instruction is interrupted half-way through

No interrupt can be missed - peripheral 1-1 ! IRQ mapping

Interrupt Driven I/O Cycle



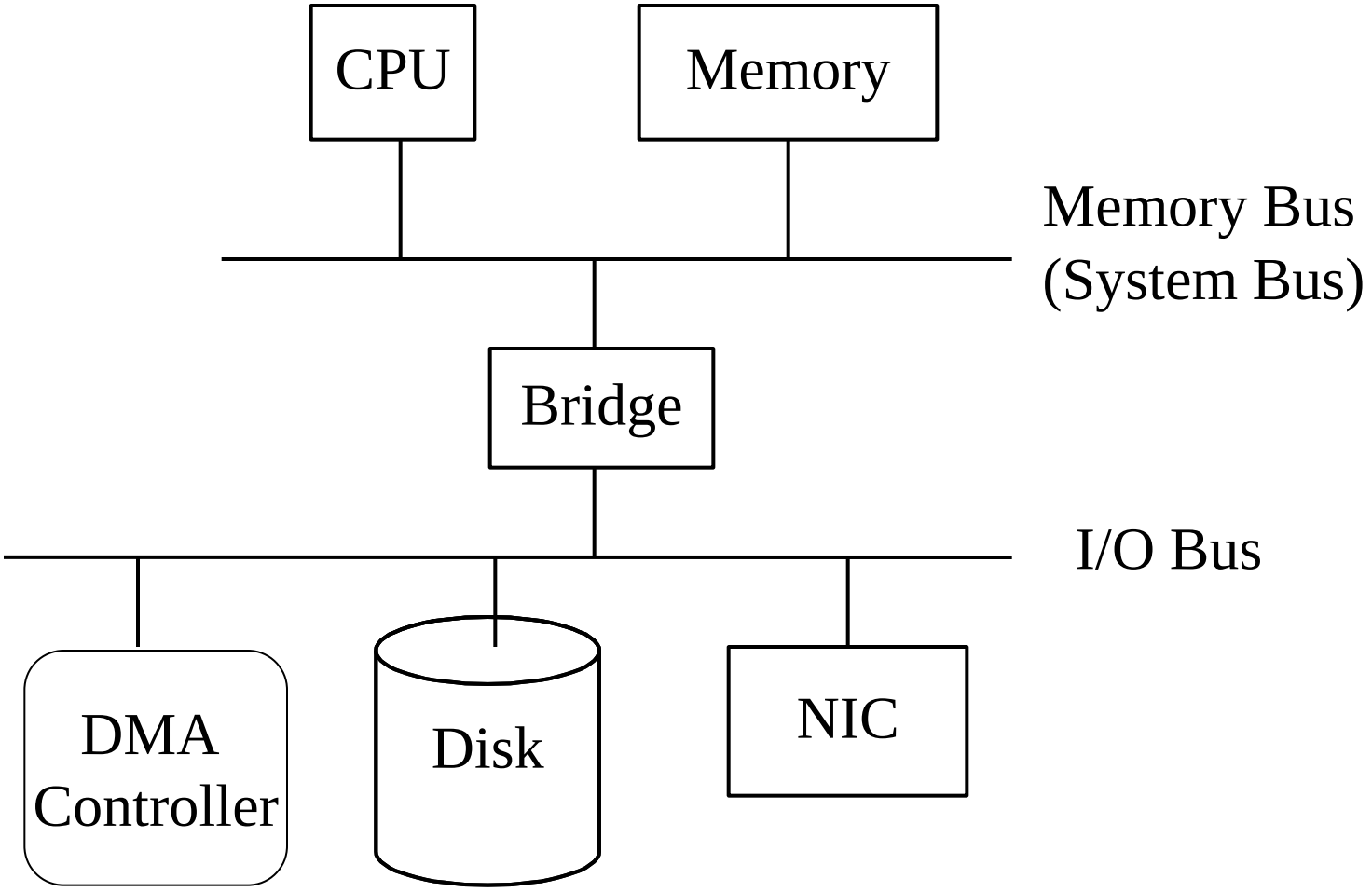
Problems with Interrupts

Interrupt-driven I/O is better than PIO, but . . .

- CPU is still involved in spending a lot of time in I/O
  - E.g. reading from disk to memory, CPU repeatedly . . .
    - sends read instruction to disk controller
    - runs handler which saves regs when signalled via IRQ
    - the handler reads the disk controller buffer to mem
    - registers are restored and CPU moves on to next instr
    - that's a lot of work for a few bytes transfer!
- One IRQ can block another!
  - solution: priorities; higher priority IRQ/device can
  - interrupt lower priority IRQ handler while it is running
- Better solution is to tell another device to deal with it all
- That's 'direct memory access' (DMA)

DMA

Requires an Extra Hardware Unit (DMA Controller) attached to the I/O bus



DMA Controller

- DMA controller is a mini-processor dedicated to I/O
  - specializes in transferring data between I/O and Memory
- When the CPU wants to transfer a block of data
  - Issues a command to the DMA controller containing
    - address of I/O device
    - starting offset of data block
    - size of block
    - the data transfer direction (read or write)
    - The location in Main Memory to read from or write to
- the DMA unit then does the I/O and raises interrupt when finished
  - meanwhile, the CPU goes on processing
- DMA controller uses system bus for transfers
  - exerts control over memory and other I/O devices
  - most useful and efficient in large (‘block’) transfers

DMA Functions

- The DMA controller uses the bus independently of CPU
  - either it sneaks use of the bus when CPU doesn't use it
  - or it takes the bus from the CPU by force (cycle stealing)
- DMA transfers do not involve the CPU
  - CPU orders DMA controller to do it and leaves it to do it
  - when transfer is finished the DMA controller signals an IRQ
  - then appropriate handler is run
- DMA is most useful and efficient in large block transfers
  - e.g. loading large blocks of data to memory from disk

Pros and Cons of DMA

- DMA doesn't tie up CPU; efficient for large transfers

- But it is difficult to coordinate with cache memory
  - a copy of memory recently accessed by the CPU is cached and buffered in special high-speed memory close to the CPU
- DMA bypasses this cache:
  - it is possible that the CPU will have written to memory but the data only have reached cache/buffers and not yet reached main memory
  - if DMA reads main memory it will read the old data, not the data the CPU has written
  - conversely, DMA may have written to memory and the CPU may see the old data still in the cache

OS: Abstractions and Access Methods

OS must virtualizes a wide range of devices into a few simple abstractions:

- Storage (read, write, seek)
  - Hard drives, Tapes, CDROM
- Networking
  - Ethernet, radio, serial line
- Multimedia
  - DVD, Camera, microphones

Operating system should provide consistent methods to access the abstractions

- Otherwise, programming is too hard

User/OS method interface

The same interface is used to access devices (like disks and network lines) and more abstract resources like files

4 main methods:

- open()
- close()
- read()
- write()

Semantics depend on the type of the device (block, char, net)

These methods are **system calls** because they are the methods the OS provides to all processes.

Unix I/O Methods

`fileHandle = open(pathName, flags, mode)`

a file handle is a small integer, valid only within a single process, to operate on the device or file

pathname: a name in the file system. In unix, devices are put under /dev. E.g. /dev/ttya is the first serial port, /dev/sda the first SCSI drive

flags: blocking or non-blocking …

mode: read only, read/write, append …

`errorCode = close(fileHandle)`

Kernel will free the data structures associated with the device

Unix I/O Methods

`byteCount = read(fileHandle, byte [] buf, count)`

read at most count bytes from the device and put them in the byte buffer buf. Bytes placed from 0^{th} byte.

Kernel can give the process less bytes, user process must check the byteCount to see how many were actually returned.

A negative byteCount signals an error (value is the error type)

`byteCount = write(fileHandle, byte [] buf, count)`

write at most count bytes from the buffer buf

actual number written returned in byteCount

a Negative byteCount signals an error

I/O semantics

From this basic interface, three different dimension to how I/O is processed:

- blocking vs. non-blocking
- buffered vs. unbuffered
- synchronous vs. asynchronous

The OS tries to support as many of these dimensions as possible for each device

The semantics are specified during the open() system call

Blocking vs. Non-Blocking I/O

Blocking –process is suspended until all bytes in the **`count`** field are read or written

E.g., for a network device, if the user wrote 1000 bytes, then the operating system would write the bytes to the device one at a time until the write() completed.

+ Easy to use and understand

- if the device just can't perform the operation (e.g. you unplug the cable), what to do? Give up and return the successful number of bytes.

Non-blocking – the OS only reads or writes as many bytes as is possible without suspending the process

+ Returns quickly

-more work for the programmer! (Event Driven Programming)

-Example : select() command checks if a given interface is capable of accepting 1000 bytes at any point. If yes, it returns.

-The programmer now has to issue a write() command if the return value is successful.

Buffered vs. Unbuffered I/O

- Sometime we want the ease of programming of blocked I/O without the long waits if the buffers on the device are small.
- Buffered I/O allows the kernel to make a copy of the data
  - NOTE: Buffering is done in the Kernel Address Space
- write() side: allows the process to write() many bytes and continue processing
- read() side: As device signals data is ready, kernel places data in the buffer. When the process calls read(), the kernel just copies the buffer.
- Why not use buffered I/O?
  - Extra copy overhead (From Kernel to User Space)
  - Delays sending data

Synchronous vs. Asynchronous I/O

- Synchronous I/O: the user process does not run at the same time the I/O does --- it is suspended during I/O
- So far, all the methods presented have been synchronous.

- Asynchronous I/O: The read() or write() call returns a small object instead of a count.
  - separate set of methods in unix: aio_read(), aio_write()
  - The user can call methods on the returned object to check "how much" of the I/O has completed
  - The user can also allow a signal handler to run when the the I/O has completed.

Three Device Types

Most operating system have three device types:

Character devices

- Used for serial-line types of devices (e.g. USB port)

Network devices

- Used for network interfaces (E.g. Ethernet card)

Block devices:

- Used for mass-storage (E.g. disks and CDROM)

What you can expect from the read/write methods changes with each device type

Character Devices

Device is represented by the OS as an ordered stream of bytes

- bytes sent out to the device by the write() system call
- bytes read from the device by the read() system call

Byte stream has no “start”, just open and start/reading writing

The user has no control of the read/write ratio, the sender process might write() 1000 bytes, and the receiver may have to call 1000 read calls each receiving 1 byte.

Network Devices

Like I/O devices, but each write() call either sends the entire block (packet), up to some maximum fixed size, or none.

On the receiver, the read() call returns all the bytes in the block, or none.

Block Devices

OS presents device as a large array of blocks

Each block has a fixed size (1KB - 8KB is typical)

User can read/write only in fixed sized blocks

Unlike other devices, block devices support random access

We can read or write anywhere in the device without having to ‘read all the bytes first’

But how to specify the nth block with current interface?

The file pointer

OS adds a concept call the file pointer

A file pointer is associated with each open file, if the device is a block device

the next read or write operates at the position in the device pointed to by the file pointer

The file pointer is measured in bytes, not blocks

Seeking in a device

To set the file pointer:

```
absoluteOffset = lseek(fileHandle, offset, whence);
```

whence specifies if the offset is absolute, from byte 0, or relative to the current file pointer position

the absolute offset is returned; negative numbers signal error codes

For devices, the offset should be a integral number of bytes relative to the size of a block.

How could you tell what the current position of the file pointer is without changing it?

Block Device Example

You want to read the 10^{th} block of a disk

each disk block is 2048 bytes long

```
fh = open(/dev/sda, , , );
pos = lseek(fh, size*count, );
if (pos < 0 ) error;
bytesRead = read(fh, buf, blockSize);
if (bytesRead < 0) error;
…
```

Optimizations in Kernel I/O Subsystem

Buffering – Storage in Main Memory to speed up data transfers between devices

Caching - fast memory holding copy of data

- Always just a copy

Spooling - hold output for a device

- If device can serve only one request at a time (Example : Printer)