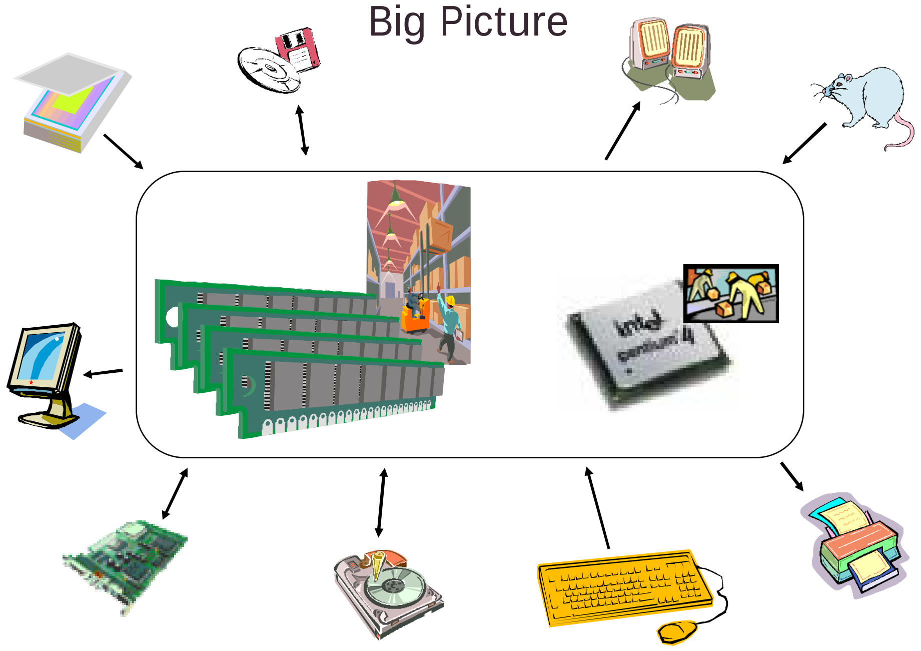
I/O Devices

22/10-2003

Pål Halvorsen

(including slides from Tore Larsen, Kai Li, and Andrew S. Tanenbaum)

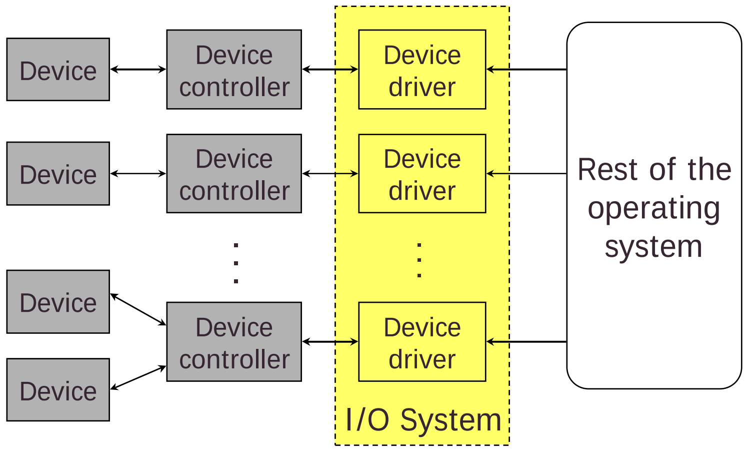
Big Picture



Today we talk about I/O

- characteristics
- interconnection
- devices & controllers (disks will be lectured in detail later)
- data transfers
- I/O software
- buffering
- ...

I/O: “Bird’s Eye View”



I/O Devices

- Keyboard, mouse, microphone, joystick, magnetic-card reader, graphic-tablet, scanner, video/photo camera, loudspeaker, microphone, scanner, printer, display, display-wall, network card, DVD, disk, floppy, wind-sensor, etc. etc.
- Large diversity:
  - many, widely differing device types
  - devices within each type also differs
- Speed:
  - varying, often slow access & transfer compared to CPU
  - some device-types require very fast access & transfer (e.g., graphic display, high-speed networks)
- Access:
  - sequential vs. random
  - read, write, read & write
- ...
- *Expect to see new types of I/O devices, and new application of old types*

I/O Devices

- Block devices: store information in fixed-size blocks, each one with its own address
  - common block sizes: 512 B – 64 KB
  - addressable
  - it is possible to read or write each block independently of all others
  - e.g., disks, floppy, tape, CD, DVD, ...
- Character devices: delivers or accepts a stream of characters, without regard to any block structure
  - it is not addressable and does not have any seek operation
  - e.g., keyboards, mice, terminals, line printers, network interfaces, and most other devices that are not disk-like...
- Does all devices fit in?
  - clocks and timers
  - memory-mapped screens

Device Controllers

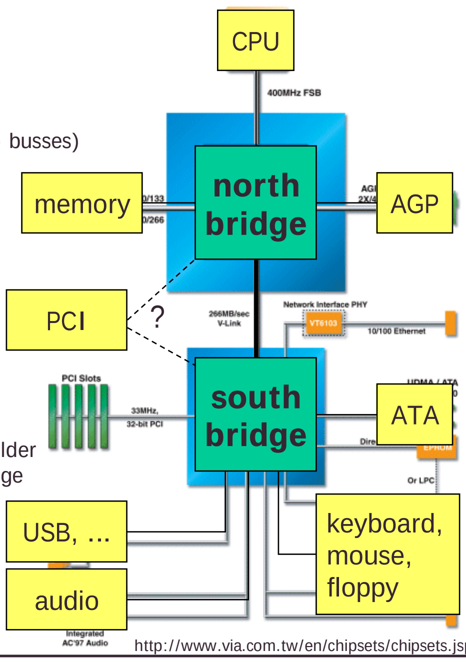
- Piece of HW that controls one or more devices
- Location
  - integrated on the host motherboard
  - PC-card (e.g., PCI)
  - embedded in the device itself
    (e.g., disks often have additional embedded controllers)

Four Basic Questions

- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

North/South Bridge Architecture: Via P4X266 Chipset

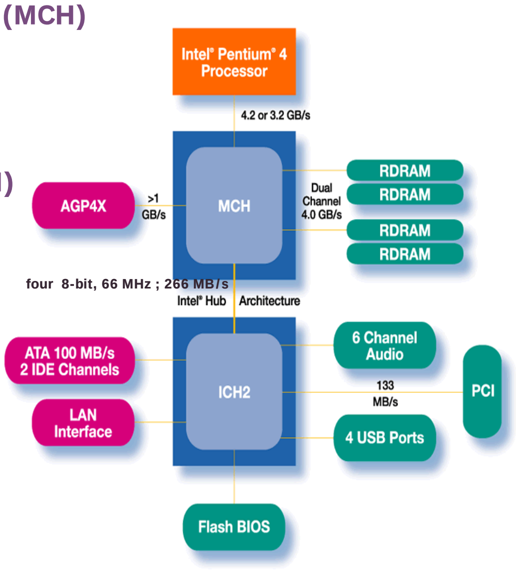
- The **north bridge** manages traffic from
  - CPU & caches
  - memory
  - advanced graphics ports (AGPs)
  - (peripheral component interconnect (PCI) busses)
- The **south bridge** manages traffic from
  - universal serial bus (USB)
  - IEEE 1394
  - ATA
  - (PCI busses)
  - keyboard & mouse
  - ...
- Via P4X266
  - PCI on south bridge
  - Increased south-north link compared to older
  - Integrated 10/100 Ethernet on south bridge
- Other chipsets include
  - Intel 440MX (BX) (both integrated) (http://www.intel.com/design/chipsets/440MX/index.htm)
  - Via P4 PB Ultra (PB 400)
  - Via EPIA



http://www.via.com.tw/en/chipsets/chipsets.jsp

Hub Architecture: Intel 850 Chipset

- The **memory controller hub (MCH)** manages traffic from
  - CPU & caches
  - memory
  - AGP
- The **I/O controller hub (ICH)** manages traffic from
  - all other devices....

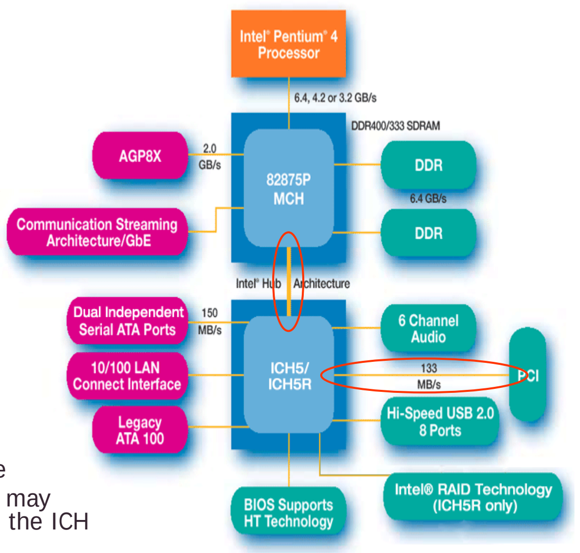


- Most of the Intel 8XX chipsets have the hub architecture

http://www.intel.com/design/chipsets/850/index.htm

Hub Architecture: Intel 875P Chipset

- MCH improvements
  - AGP:
    4x → 8x
  - memory interface:
    200 → 400 MHz
  - system (front side) bus:
    400/533 → 800 MHz
  - Gbps network interface
- But, still only
  - four 8-bit, 66 MHz (266 MBps) hub-to-hub interface
  - 32 bit, 33 MHz PCI bus
- However, some chipsets (e.g., 840) have a 64-bit, 33/66 MHz PCI Controller Hub (P64H) connected directly to the MCH by a 2x (16 bit) wide hub interface
- Server chipsets (e.g., E7500) may have several P64Hs replacing the ICH


Intel® Pentium® 4 Processor
6.4, 4.2 or 3.2 GB/s
DDR400/333 SDRAM
AGP8X
2.0 GB/s
82875P MCH
DDR
6.4 GB/s
DDR
Communication Streaming Architecture/GbE
Intel® Hub Architecture
Dual Independent Serial ATA Ports
150 MB/s
6 Channel Audio
10/100 LAN Connect Interface
ICH5/ ICH5R
133 MB/s
PCI
Legacy ATA 100
Hi-Speed USB 2.0 8 Ports
BIOS Supports HT Technology
Intel® RAID Technology (ICH5R only)

http://www.intel.com/design/chipsets/875P/index.htm

Four Basic Questions

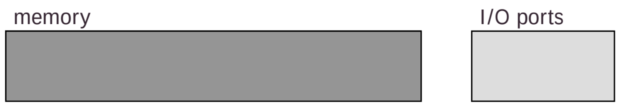
- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

Accessing Device Controller Registers

- To communicate with the CPU, each controller have a few registers where operations are specified
- Additionally, some devices need a memory buffer
- Two alternatives: port I/O and memory mapped I/O

Port I /O

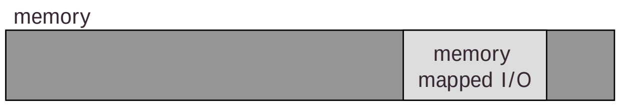
- Devices registers mapped onto “ports”; a separate address space



- Use special I/O instructions to read/write ports
- Protected by making I/O instructions available only in kernel/supervisor mode
- Used for example by IBM 360 and successors

Memory Mapped I/ O

- Device registers mapped into regular address space



- Use regular move (assignment) instructions to read/ write registers
- Use memory protection mechanism to protect device registers
- Used for example by PDP-11

Memory Mapped I/O vs. Port I/O

- Ports:
  – special I/O instructions are CPU dependent
- Memory mapped:
  + memory protection mechanism allows greater flexibility than protected instructions
  + may use all memory reference instructions for I/O
  – cannot cache device registers
    (must be able to selectively disable caching)
  – I/O devices do not see the memory address - how to route only the right memory address onto slower peripheral buses (may initiate bridge at setup time to transfer required address areas)
- Intel Pentium use a hybrid
  – address 640K to 1M is used for memory mapped I/O data buffers
  – I/O ports 0 to 64K is used for device control registers

Four Basic Questions

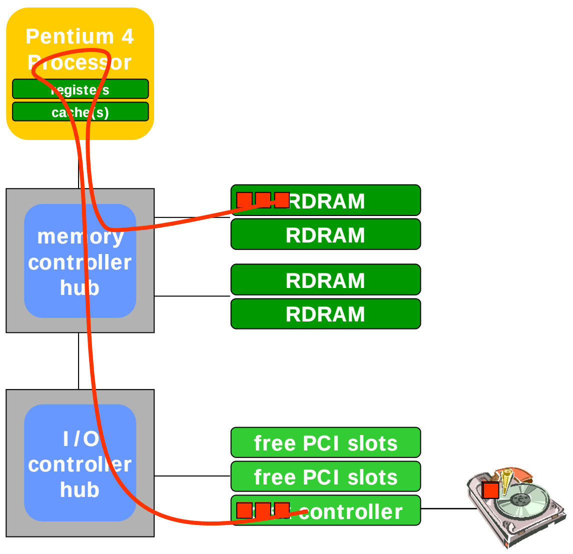
- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

Performing I/O Data Transmissions

- **Programmed I/O (PIO)**
  - the CPU handles the transfers
  - transfers data between registers and device
- Interrupt driven I/O
  - use CPU to transfer data, but let an I/O module run concurrently
- **Direct Memory Access (DMA)**
  - an adaptor accesses main memory
  - transfers blocks of data between memory and device
- Channel
  - simple specialized peripheral processor dedicated to I/O
  - handles most transmission, but less control
  - shared memory. No private memory.
- Peripheral Processor (PPU)
  - general processor dedicated to I/O control and transmission
  - shared and private memory. (CDC 6600, 1964)

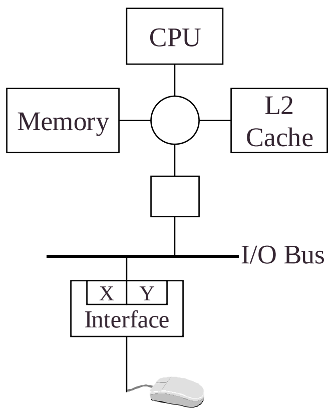
PIO

- Device delivers data to controller
- PIO:
  - CPU reads data from controller buffer to register
  - CPU writes register to memory location
- CPU is busy moving data



PIO: “Slow” Input Device

- Device
  - data registers
  - status register (ready, busy, interrupt, ...)
- A simple mouse design
  - put (X, Y) in data registers on a move
  - interrupt
- Perform an input on an interrupt
  - reads values in X, Y registers
  - sets ready bit
  - wakes up a process/thread or execute a piece of code



PIO Output Device

- Device
  - Data registers
  - Status registers (ready, busy, ...)
- Perform an output
  - Polls the busy bit
  - Writes the data to data register(s)
  - Controller sets busy bit and transfers data
  - Clear busy bit
- Writing string to printer:

```
copy_from_user(buffer, p, count);                  /* p is the kernel bufer */
for (i = 0; i < count; i++) {                      /* loop on every character */
     while (*printer_status_reg != READY) ;        /* loop until ready */
     *printer_data_register = p[i];                /* output one character */
}
return_to_user( );
```

Interrupt-Driven I/O

- Writing a string to the printer using interrupt-driven I/O

  a) code executed when print system call is made

  b) interrupt service procedure

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

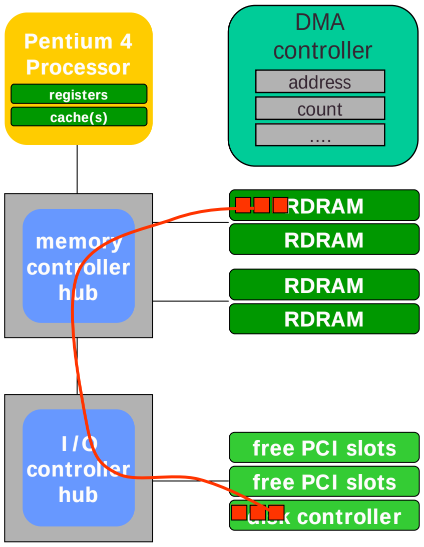
(a)

```
if (count == 0) {
     unblock_user( );
} else {
     *printer_data_register = p[i];
     count = count – 1;
      i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

DMA

- Device delivers data to controller
- DMA:
  1. set up DMA controller
  2. DMA controller initiates transfer
  3. data is moved (increasing address, reducing count)
  4. disk controller notifies DMA controller when finished (count = 0)
  5. DMA controller interrupts
- CPU is free
- Cycle stealing



DMA

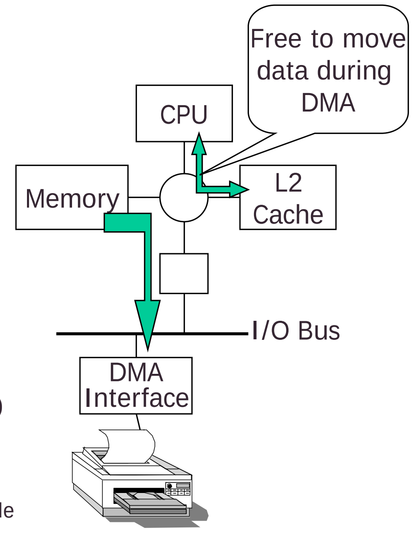
- Perform DMA from host CPU
  - device driver call (kernel mode)
  - wait until DMA device is free
  - initiate a DMA transaction (command, memory address, size)
  - block
- Interrupt handler (on completion)
  - wakeup the blocked process
- DMA interface
  - DMA data to device (size--; address++)
  - interrupt on completion (size == 0)
- Printing a string using DMA
  a) code executed when print system call is made
  b) interrupt service procedure

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)



PIO vs. DMA

- DMA:
  - + supports large transfers, latency of requiring bus is amortized over hundreds/thousands of bytes
  - – may be expensive for small transfers
  - – overhead to handle virtual memory and cache consistence
  - o is common practice
- PIO:
  - – uses the CPU
  - – loads data into registers and cache
  - + potentially faster for small transfers with carefully designed software

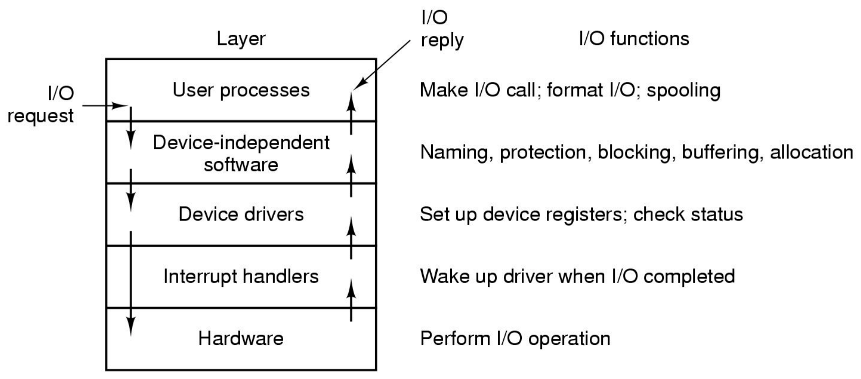
Four Basic Questions

- How are devices connected to CPU/memory?
- How are device controller registers accessed & protected?
- How are data transmissions controlled?
- Synchronization: interrupts versus polling?

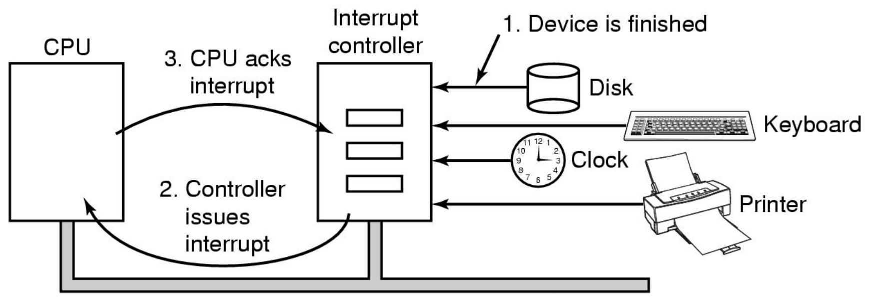
Synchronization: interrupts vs. polling

- Polling:
  - processor polls the device while waiting for I/O to complete
  - wastes cycles – inefficient
- Interrupt:
  - device asserts interrupt when I/O completed
  - frees processor to move on to other tasks
  - interrupt processing is costly and introduces latency penalty
- Possible strategy:
  - apply interrupts, but reduce interrupts frequency through careful driver/controller interaction

Layered I/O Software


Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Interrupts Revisited



Interrupts Revisited

- Steps performed
  1. Check that interrupts are enabled, and check that no other interrupt is being processed, no interrupt pending, and no higher priority simultaneous interrupt
  2. Interrupt controller puts a index number identifying the device on the address lines and asserts CPUs interrupt signal
  3. Save registers not already saved by interrupt hardware
  4. Set up context for interrupt service procedure
  5. Set up stack for interrupt service procedure
  6. Acknowledge interrupt controller, re-enable interrupts
  7. Copy registers from where saved (stack)
  8. Run service procedure
  9. Set up MMU context for process to run next
  10. Load new process' registers
  11. Start running the new process
- Details of interrupt handling varies among different processors/computers

Device Driver Design Issues

- Operating system and driver communication
  - Commands and data between OS and device drivers
- Driver and hardware communication
  - Commands and data between driver and hardware
- Driver operations
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Interface

- Open(deviceNumber)
  - Initialization and allocate resources (buffers)
- Close(deviceNumber)
  - Cleanup, deallocate, and possibly turnoff
- Device driver types
  - Block: fixed sized block data transfer
  - Character: variable sized data transfer
  - Terminal: character driver with terminal control
  - Network: streams for networking

Device Driver Interface

- Block devices:
  - read(deviceNumber, deviceAddr, bufferAddr)
    - transfer a block of data from “deviceAddr” to “bufferAddr”
  - write(deviceNumber, deviceAddr, bufferAddr)
    - transfer a block of data from “bufferAddr” to “deviceAddr”
  - seek(deviceNumber, deviceAddress)
    - move the head to the correct position
    - usually not necessary
- Character devices:
  - read(deviceNumber, bufferAddr, size)
    - reads “size” bytes from a byte stream device to “bufferAddr”
  - write(deviceNumber, bufferAddr, size)
    - write “size” bytes from “bufferSize” to a byte stream device

Some Unix Device Driver Interface Entry Points

- init(): Initialize hardware
- start(): Boot time initialization (require system services)
- open(dev, flag, id): initialization for read or write
- close/release(dev, flag, id): release resources after read and write
- halt(): call before the system is shutdown
- intr(vector): called by the kernel on a hardware interrupt
- read()/write(): data transfer
- poll(pri): called by the kernel 25 to 100 times a second
- ioctl(dev, cmd, arg, mode): special request processing

Device-Independent I/O Software

- Functions of the device-independent I/O software:

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicate devices
Providing a device-independent block size
...

Why Buffering

- Speed mismatch between the producer and consumer
  - Character device and block device, for example
- Adapt different data transfer sizes
  - Packets vs. streams
- Support copy semantics
- Deal with address translation
  - I/O devices see physical memory, but programs use virtual memory
- Spooling
  - Avoid deadlock problems
- Caching
  - Avoid I/O operations

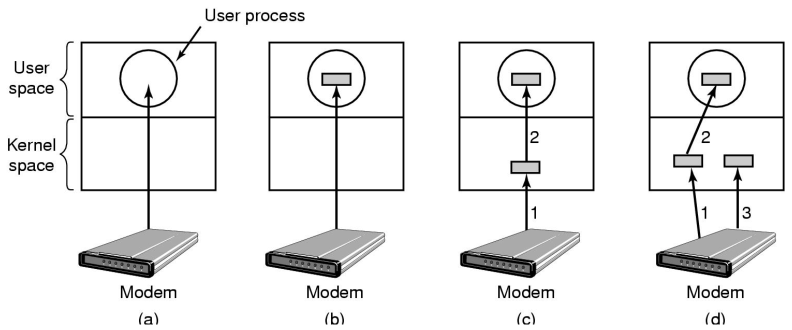
Buffering

a) No buffer
   - interrupt per character/block

b) User buffering
   - user blocks until buffer full or I/O complete
   - paging problems!?

c) Kernel buffer, copying to user
   - what if buffer is full/busy when new data arrives?

d) Double kernel buffering
   - alternate buffers, read from one, write to the other



Detailed Steps of Blocked Read

1. A process issues a read call which executes a system call
2. System call code checks for correctness and cache
3. If it needs to perform I/O, it will issues a device driver call
4. Device driver allocates a buffer for read and schedules I/O
5. Controller performs DMA data transfer, blocks the process
6. Device generates an interrupt on completion
7. Interrupt handler stores any data and notifies completion
8. Move data from kernel buffer to user buffer and wakeup blocked process
9. User process continues

Asynchronous I/O

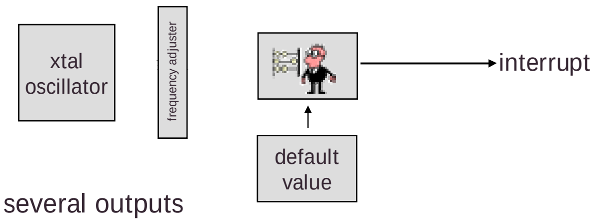
- Why do we want asynchronous I/O?
  - Life is simple if all I/O is synchronous
- How to implement asynchronous I/O?
  - On a read
    - copy data from a system buffer if the data is there
    - otherwise, block the current process
  - On a write
    - copy to a system buffer, initiate the write and return

Summary

- A large fraction of the OS is concerned with I/O
- Several ways to do I/O
- Several layers of software

Example: Clocks

- Old, simple clocks used power lines and caused an interrupt at every voltage pulse (50 - 60 Hz)
- New clocks use
  - quartz crystal oscillators generating periodic signals at a very high frequency
  - counter which is decremented each pulse – if zero, it causes an interrupt
  - register to load the counter



- May have several outputs
- Different modes
  - One–shot: counter is restored only by software
  - Square–wave: counter is reset immediately (e.g., for clock ticks)

Examples: Clocks

- HW only generates clock interrupts
- It is up to the clock software to make use of this
  - Maintaining time-of-day
  - Preventing processes from running longer than allowed
  - Accounting for CPU usage
  - Handling ALARM system call
  - Providing watchdog timers
  - Doing profiling, monitoring, and statistics gathering

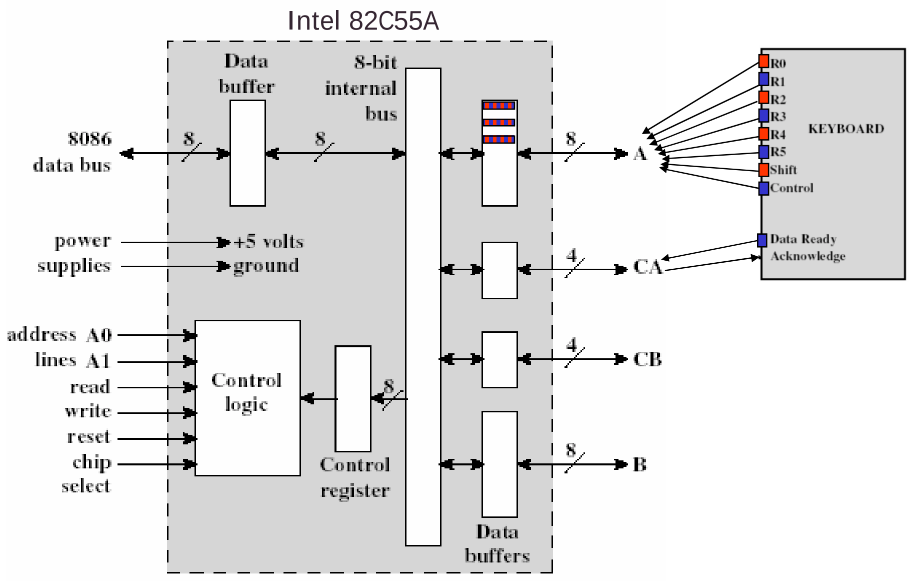
Example: Keyboard

- Keyboards provide input as a sequence of bits
- Example - coded with IRA - (international reference alph.): “K” = $b_7b_6b_5b_4b_3b_2b_1$ = 1001011
- Raw mode vs. Cooked mode
- Buffering

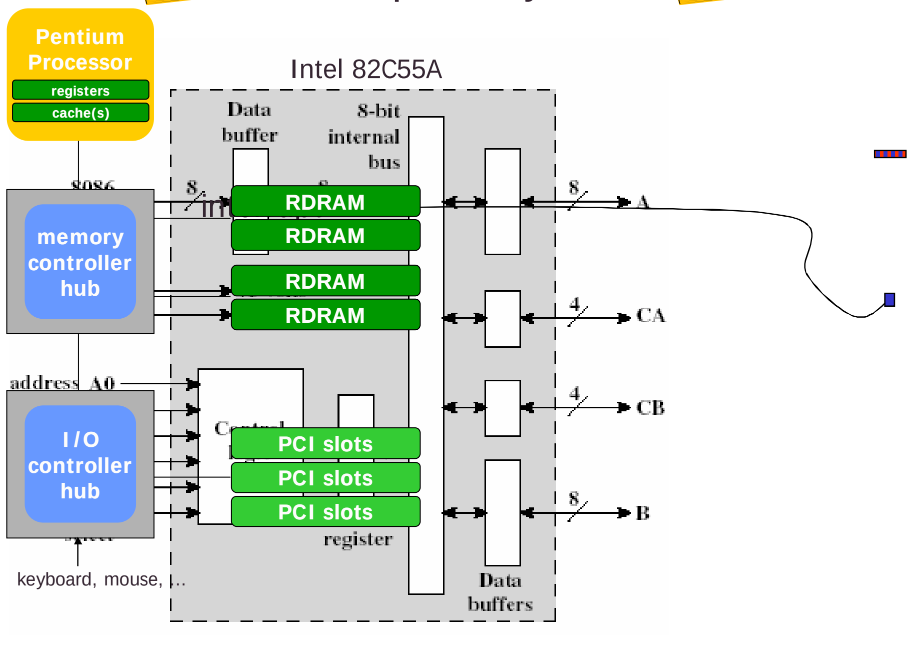
bit position

			b_7	0	0	0	0	🟥	1	1	1
			b_6	0	0	1	1	🟥	0	1	1
			b_5	0	1	0	1	🟥	1	0	1
b_4	b_3	b_2	b_1								
0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	STX	DC2	"	2	B	R	b	r
0	0	1	1	ETX	DC3	#	3	C	S	c	s
0	1	0	0	EOT	DC4	$	4	D	T	d	t
0	1	0	1	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	ACK	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	'	7	G	W	g	w
1	0	0	0	BS	CAN	(	8	H	X	h	x
1	0	0	1	HT	EM	)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
🟥	🟥	🟥	🟥	VT	ESC	+	;	🟥	[	k	{
1	1	0	0	FF	FS	,	<	L	\	l	|
1	1	0	1	CR	GS	-	=	M	]	m	}
1	1	1	0	SO	RS	.	>	N	^	n	~
1	1	1	1	SI	US	/	?	O	_	o	DEL

Example: Keyboard



Example: Keyboard



Pentium Processor
registers
cache(s)
memory controller hub
I/O controller hub
keyboard, mouse, ...
Intel 82C55A
Data buffer
8-bit internal bus
RDRAM
RDRAM
RDRAM
RDRAM
PCI slots
PCI slots
PCI slots
address A0
register
Data buffers
8
A
4
CA
4
CB
8
B