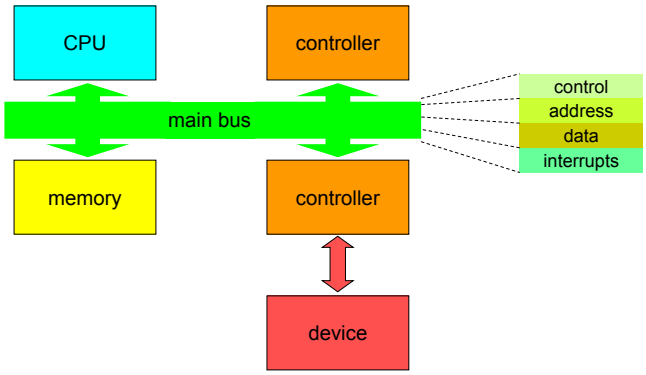
Introduction to Device I/O

- 10A I/O Architectures
- 10B I/O Mechanisms
- 10C Disk I/O

I/O architectures: busses



Memory type busses

- Initially back-plane memory-to-CPU interconnects
  - a few “bus masters”, and many “slave devices”
  - arbitrated multi-cycle bus transactions
    request, grant, address, respond, transfer, ack
    operations: read, write, read/modify/write, interrupt
- originally most busses were of this sort
  - ISA, EISA, PCMCIA, PCI, cPCI, video busses, ...
  - distinguished by
    form-factor, speed, data width, hot-plug, maximum length, ...
    bridging, self identifying, dynamic resource allocation, …

TERMS: Bus Arbitration & Mastery

- bus master
  - any device (or CPU) that can request the bus
  - one can also speak of the “current bus master”
- bus slave
  - a device that can only respond to bus requests
- bus arbitration
  - process of deciding to whom to grant the bus
    may be based on time, geography or priority
    may also clock/choreograph steps of bus cycles
    bus arbitrator may be part of CPU or separate

Network type busses

- evolved as peripheral device interconnects
  - SCSI, USB, 1394 (firewire), Infiniband, PCIe
  - cables and connectors rather than back-planes
  - designed for easy and dynamic extensibility
  - originally slower than back-plane, but no longer
- much more similar to a general purpose network
  - packet switched, topology, routing, node identity
  - may be master/slave (USB) or peer-to-peer (1394)
  - may be implemented by controller or by host

I/O architectures: devices & controllers

- I/O devices
  - peripheral devices that interface between the computer and other media (disks, tapes, networks, serial ports, keyboards, displays, pointing devices, etc.)
- device controllers connect a device to a bus
  - communicate control operations to device
  - relay status information back to the bus
  - manage DMA transfers for the device
  - generate interrupts for the device
- controller usually specific to a device and a bus

Device Controller Registers

- device controllers export registers to the bus
  - registers in controller can be addressed from bus
  - writing into registers controls device or sends data
  - reading from registers obtains data/status
- register access method varies with CPU type
  - may require special instructions (e.g. x86 IN/OUT)
    privileged instructions restricted to supervisor mode
  - may be mapped onto bus like memory
    accessed with normal (load/store) instructions
    I/O address space not accessible to most processes

A simple device: 16550 UART

offset	*contents*								*Register*
0	x	x	x	x	x	x	x	x	Data Register
1					MDM	STS	XMT	RCV	Interrupt Enable Register
2					MDM	STS	XMT	RCV	Interrupt Register
3	speed	BRK		PARITY		STOP	WORDLEN		Line Control Register
4							DTR	RTS	Modem Control Register
5	RCV	EMT	XMT	BRK	FER	PER	OVR	RER	Line Status Register
6					DCD	RI	DSR	CTS	Modem Status Register

A 16550 presents seven 8-bit registers to the bus.

All communication between the bus and the device (send data, receive data, status and control) is performed by reading from, and writing to these registers.

(16550 UART registers)

- 0: data – read received byte, write to transmit a byte
  - (or LSB of speed divisor when speed set is enabled)
- 1: interrupt enables – for transmit done, data received, cd/ring
  - (or MSB of speed divisor when speed set is enabled)
- 2: interrupt registers – currently pending interrupt conditions
- 3: line control register – character length, parity and speed
- 4: modem control register – control signals sent by computer
- 5: line status register – xmt/rcv completion and error conditions
- 6: modem status registers – received modem control signals

Scenario: direct I/O with polling

```
uart_write_char( char c ) {
        while( (inb(UART_LSR) & TR_DONE) == 0);
        outb( UART_DATA, c );
}
char uart_read_char() {
        while( (inb(UART_LSR) & RX_READY) == 0);
        return( inb(UART_DATA) );
}
```


(mechanisms: direct polled I/O)

- all transfers happen under direct control of CPU
  - CPU transfers data to/from device controller registers
  - transfers are typically one byte or word at a time
  - may be accomplished with normal or I/O instructions
- CPU polls device until it is ready for data transfer
  - received data is available to be read
  - previously initiated write operations are completed
- advantages
  - very easy to implement (both hardware and software)

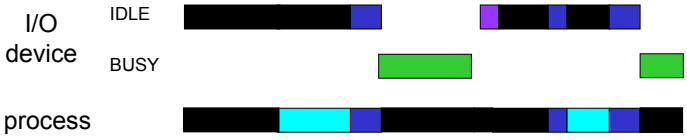
performance of direct I/O

- CPU intensive data transfers
  - each byte/word requires multiple instructions
- CPU wasted while awaiting completion
  - busy-wait polling ties up CPU until I/O is completed
- devices are idle while we are running other tasks
  - I/O can only happen when an I/O task is running
- how can these problems be dealt with
  - let controller transfer data without attention from CPU
  - let application block pending I/O completion
  - let controller interrupt CPU when I/O is finally done

importance of good device utilization

- key system devices limit system performance
  - file system I/O, swapping, network communication
- if device sits idle, its throughput drops
  - this may result in lower system throughput
  - longer service queues, slower response times
- delays can disrupt real-time data flows
  - resulting in unacceptable performance
  - possible loss of irreplaceable data
- it is very important to keep key devices busy
  - start request *n+1* immediately when *n* finishes

Poor I/O device Utilization



1. process waits to run
2. process does computation in preparation for I/O operation
3. process issues read system call, blocks awaiting completion
4. device performs requested operation
5. completion interrupt awakens blocked process
6. process runs again, finishes read system call
7. process does more computation
8. process issues read system call, blocks awaiting completion

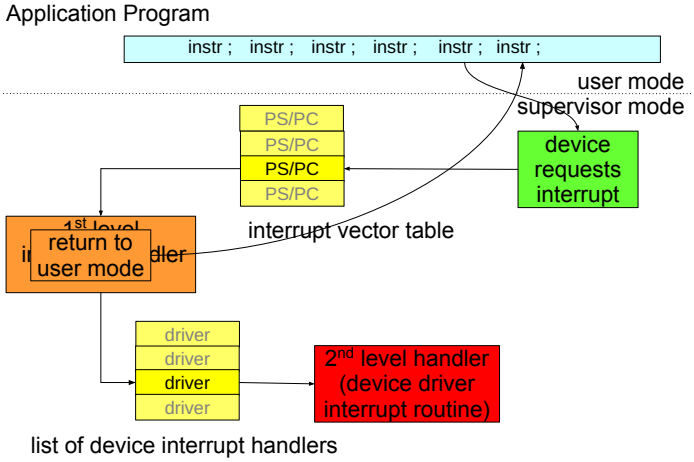
Direct Memory Access

- bus facilitates data flow in all directions between
  - CPU, memory, and device controllers
- CPU can be the bus-master
  - initiating data transfers w/memory, device controllers
- device controllers can also master the bus
  - CPU instructs controller what transfer is desired
    what data to move to/from what part of memory
  - controller does transfer w/o CPU assistance
  - controller generates interrupt at end of transfer

I/O Interrupts

- device controllers, busses, and interrupts
  - busses have ability to send interrupts to the CPU
  - devices signal controller when they are done/ready
  - when device finishes, controller puts interrupt on bus
- CPUs and interrupts
  - interrupts look very much like traps
    traps come from CPU, interrupts are caused externally
  - unlike traps, interrupts can be enabled/disabled
    a device can be told it can or cannot generate interrupts
    special instructions can enable/disable interrupts to CPU
    interrupt may be held *pending* until s/w is ready for it

Interrupt Handling



Keeping Key Devices Busy

- allow multiple requests pending at a time
  - queue them, just like processes in the ready queue
  - requesters block to await eventual completions
- use DMA to perform the actual data transfers
  - data transferred, with no delay, at device speed
  - minimal overhead imposed on CPU
- when the currently active request completes
  - device controller generates a completion interrupt
  - interrupt handler posts completion to requester
  - interrupt handler selects and initiates next transfer

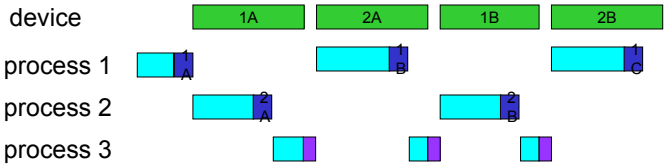
Interrupt Driven Chain Scheduled I/O

```
xx_read/write() {
    allocate a new request descriptor
    fill in type, address, count, location
    insert request into service queue
    if (device is idle) {
        disable_device_interrupt();
        xx_start();
        enable_device_interrupt();
    }
    await completion of request
    extract completion info for caller
}

xx_start() {
    get next request from queue
    get address, count, disk address
    load request parameters into controller
    start the DMA operation
    mark device busy
}
```

```
xx_intr() {
    extract completion info from controller
    update completion info in current req
    wakeup current request
    if (more requests in queue)
        xx_start()
    else
        mark device idle
}
```


Multi-Tasking & Interrupt Driven I/O



1. P_1 runs, requests a read, and blocks
2. P_2 runs, requests a read, and blocks
3. P_3 runs until interrupted
4. Awaken P_1 and start next read operation
5. P_1 runs, requests a read, and blocks
6. P_3 runs until interrupted
7. Awaken P_2 and start next read operation
8. P_2 runs, requests a read, and blocks
9. P_3 runs until interrupted
10. Awaken P_1 and start next read operation
11. P_1 runs, requests a read, and blocks

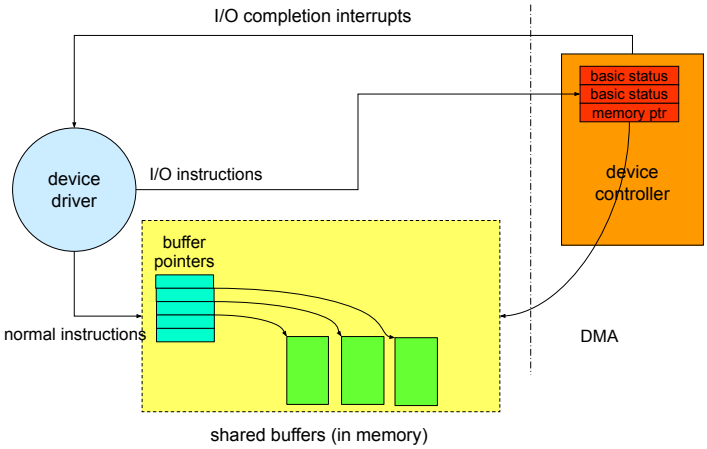
mechanisms: memory mapped I/O

- DMA may not be the best way to do I/O
  - designed for large contiguous transfers
  - some devices have many small sparse transfers
    e.g. consider a video game display adaptor
- implement as a bit-mapped display adaptor
  - 1Mpixel display controller, on the CPU memory bus
  - each word of memory corresponds to one pixel
  - application uses ordinary stores to update display
- low overhead per update, no interrupts to service
- relatively easy to program

Trade-off: memory mapped vs. DMA

- DMA performs large transfers efficiently
  - better utilization of both the devices and the CPU
    device doesn't have to wait for CPU to do transfers
  - but there is considerable per transfer overhead
    setting up the operation, processing completion interrupt
- memory-mapped I/O has no per-op overhead
  - but every byte is transferred by a CPU instruction
    no waiting because device accepts data at memory speed
- DMA better for occasional large transfers
- memory-mapped better frequent small transfers
- memory-mapped devices more difficult to share

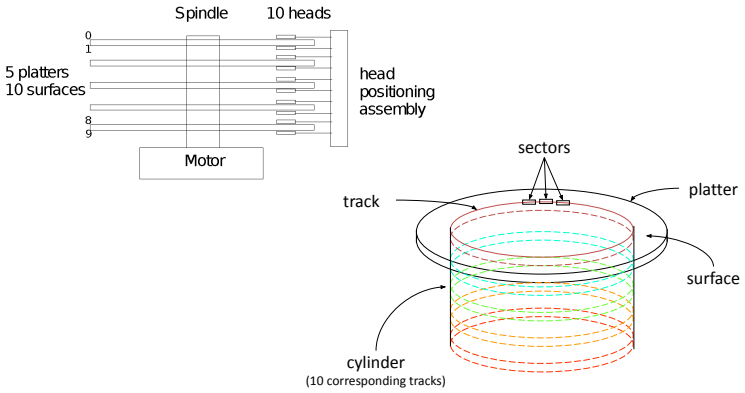
Smart Device Controller



(I/O Mechanisms: smart controllers)

- Smarter controllers can improve on basic DMA
- they can queue multiple input/output requests
  - when one finishes, automatically start next one
  - reduce completion/start-up delays
  - eliminate need for CPU to service interrupts
- they can relieve CPU of other I/O responsibilities
  - request scheduling to improve performance
  - they can do automatic error handling & retries
- abstract away details of underlying devices

Disk Drives and Geometry



(Disk drive geometry)

- spindle
  - a mounted assembly of circular platters
- head assembly
  - read/write head per surface, all moving in unison
- track
  - ring of data readable by one head in one position
- cylinder
  - corresponding tracks on all platters
- sector
  - logical records written within tracks
- disk address = <cylinder / head / sector >

Disks have dominated File Systems

- fast swap, file system, database access
- minimize seek overhead
  - organize file systems into cylinder clusters
  - write-back caches and deep request queues
- minimize rotational latency delays
  - maximum transfer sizes
  - buffer data for full-track reads and writes
- we accepted poor latency in return for IOPS

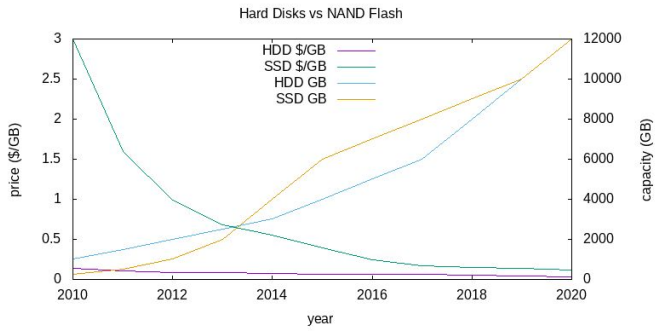
(Optimizing disk performance)

- don't start I/O until disk is on-cyl/near sector
  - I/O ties up the controller, locking out other operations
  - other drives seek while one drive is doing I/O
- minimize head motion
  - do all possible reads in current cylinder before moving
  - make minimum number of trips in small increments
- encourage efficient data requests
  - have lots of requests to choose from
  - encourage cylinder locality
  - encourage largest possible block sizes

Disk vs SSD Performance

	Cheeta (archival)	Barracuda (high perf)	Extreme/Pro (SSD)
RPM	7,000	15,000	n/a
average latency	4.3ms	2ms	n/a
average seek	9ms	4ms	n/a
transfer speed	105MB/s	125MB/s	540MB/s
sequential 4KB read	39us	33us	10us
sequential 4KB write	39us	33us	11us
random 4KB read	13.2ms	6ms	10us
random 4KB write	13.2ms	6ms	11us

Random Access: Game Over



The Changing I/O Landscape

- Storage paradigms
  - old: swapping, paging, file systems, databases
  - new: NAS, distributed object/key-value stores
- I/O traffic
  - old: most I/O was disk I/O
  - new: network and video dominate many systems
- Performance goals:
  - old: maximize throughput, IOPS
  - new: low latency, scalability, reliability, availability

Reading and Assignments

Reading

- Arpaci C33-33.6 … events
- Arpaci C38 … RAID
- poll(2), select(2), sigaction(2) system calls