I/O

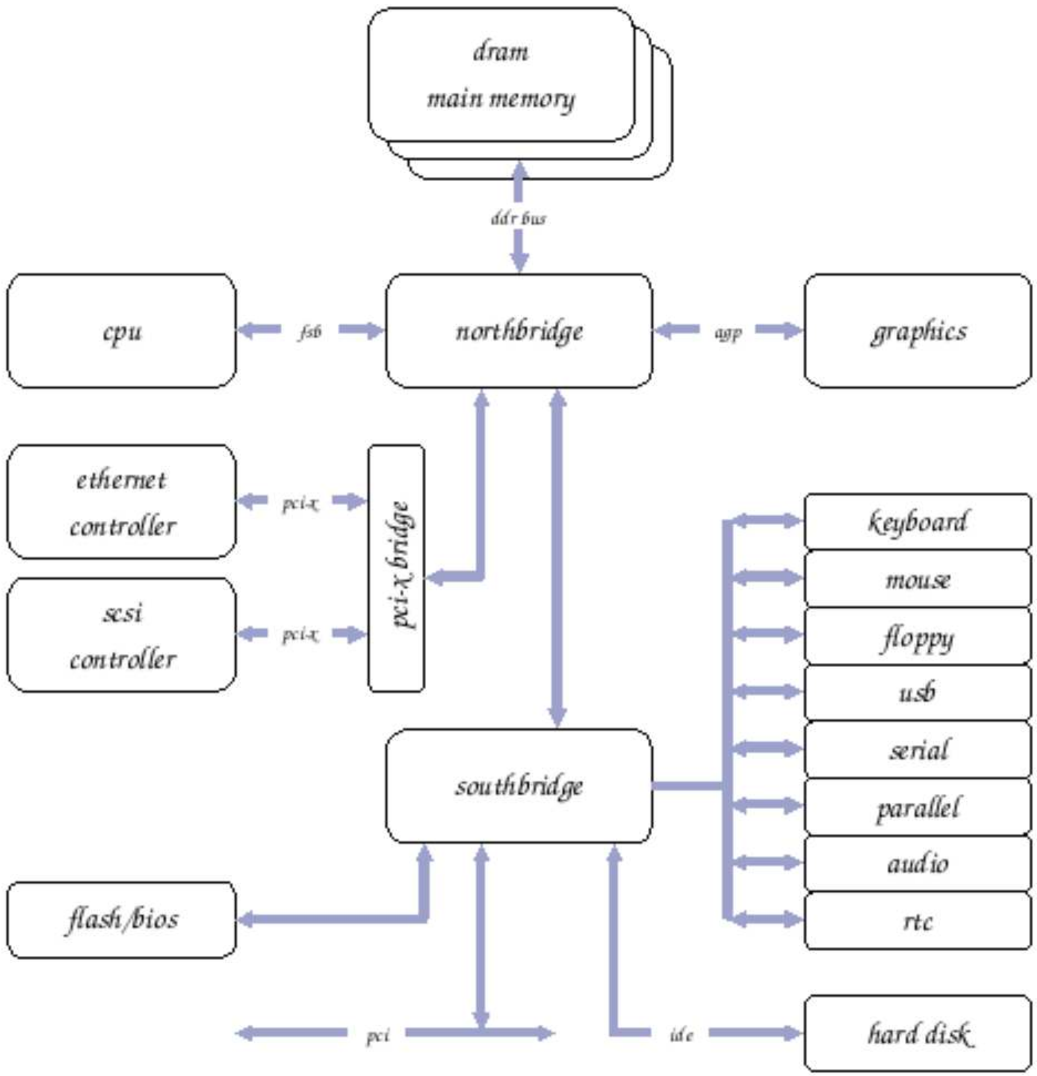
Operating Systems

Oleg Goldshmidt

ogoldshmidt@computer.org

Lecture 8

What Is An (Intel) Computer?



Basic I/O Concepts I

- controlling the variety of I/O devices is a major function of an OS
  - a wide variety of hardware devices
  - huge differences in speed (slow compared to CPU)
  - different protocols, register sets, etc
  - user interaction
- most of the OS code is related to I/O
  - device drivers for all the supported devices

Basic I/O Concepts II

- basic hardware-related concepts
  - port — connection through which a device communicates with the computer
  - bus — common set of wires and a common protocol used by a number of devices
  - controller — a collection of electronics that can operate a device, a port, or a bus
    - a single chip (e.g., for a serial port)
    - a circuit board (e.g., a SCSI controller)
    - host adapters and device built-in controllers
- device drivers
  - encapsulate the oddities of specific devices
  - present a convenient I/O interface to OS

I/O Device Operation I

- controllers communicate with the CPU through a set of registers
  - data registers
  - control registers
- I/O instructions: specify the transfer of a byte or a word to an I/O port address
- memory-mapped I/O: controller registers are mapped into the address space of the CPU; CPU uses normal reads and writes for I/O
- combination (e.g., graphics adapters): I/O ports for control, memory-mapped region to hold the screen contents

I/O Device Operation II

- programmed I/O and DMA
  - controller registers are usually accessed through programmed I/O
  - data can be accessed via either programmed I/O (typical for character devices) or DMA (typical for block devices)
- host may poll the controller's status register to get access to the device
  - controller indicates state through the busy bit in the status register (notation: `status:busy`)
  - host sets the `command-ready` bit in the control register when a command is available
  - reasonable for fast controllers and devices

Polling Example

- busy-wait cycle for write
  - host reads the `status:busy` bit until clear
  - host sets the `control:write` bit
  - host writes a byte into the `data-out` register
  - host sets the `control:command-ready` bit
  - controller sets the `status:busy` bit
  - controller reads the `control` register, sees `write`
  - controller reads the `data-out` register, transfers the data to the device
  - controller clears the `control:command-ready` bit
  - controller clears the `status:error` bit
  - controller clears the `status:busy` bit

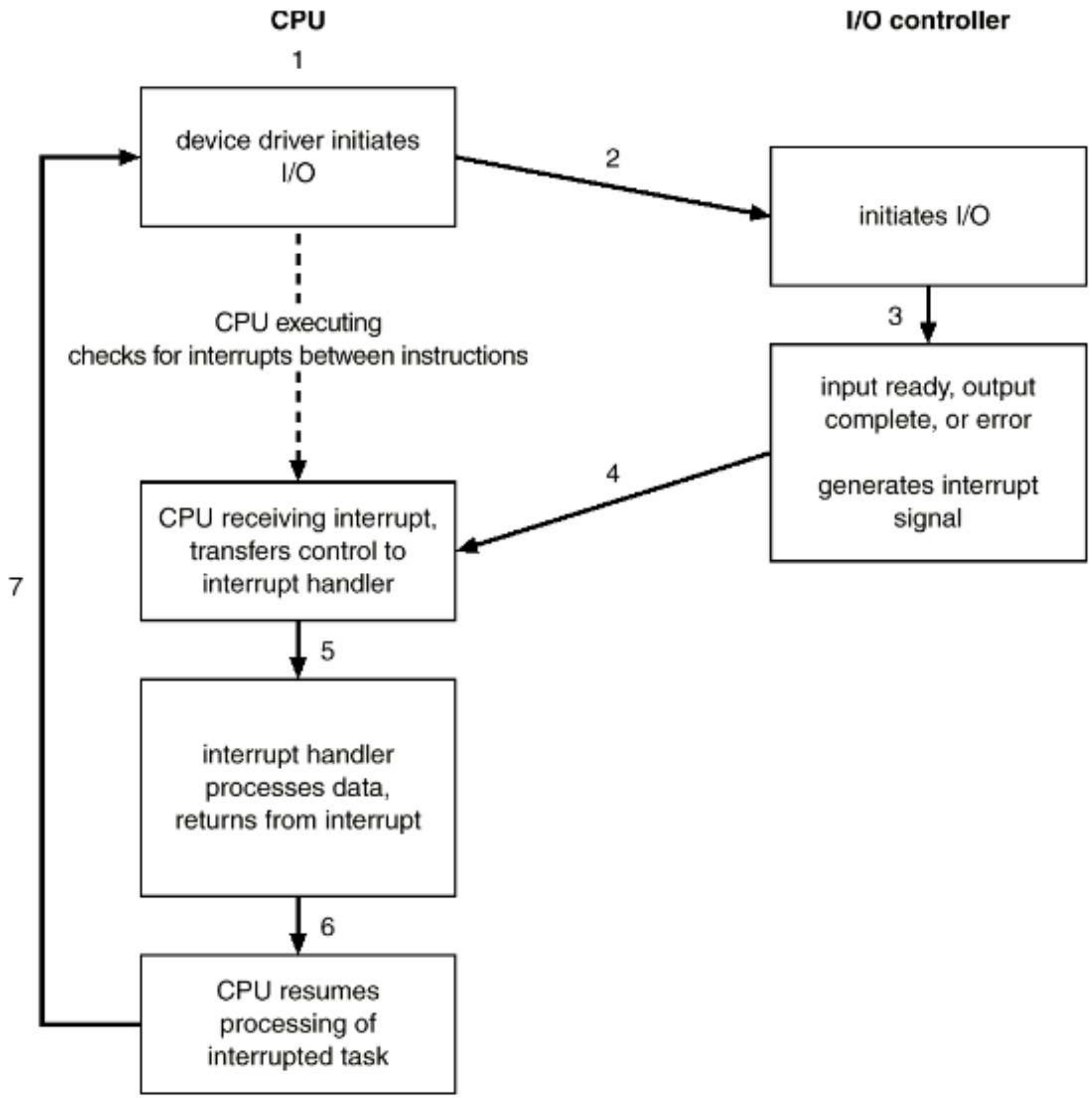
Interrupts I

- CPU checks a special wire ("interrupt request line") after every instruction
- if there is a signal (raised by a controller)
  - saves state
  - jumps to interrupt handler, executes
  - restores state, continues
- response to an asynchronous event
- requirements
  - deferred during critical processing
  - efficient dispatch to proper handler
  - multiple interrupt priority levels

Interrupts II

- maskable and nonmaskable interrupts
  - nonmaskable — unrecoverable errors
  - maskable — used by controllers to request service
- interrupt vector and chaining
- boot time probing and configuration
- generic mechanism
  - exceptions, VM paging, system calls

Interrupt Cycle



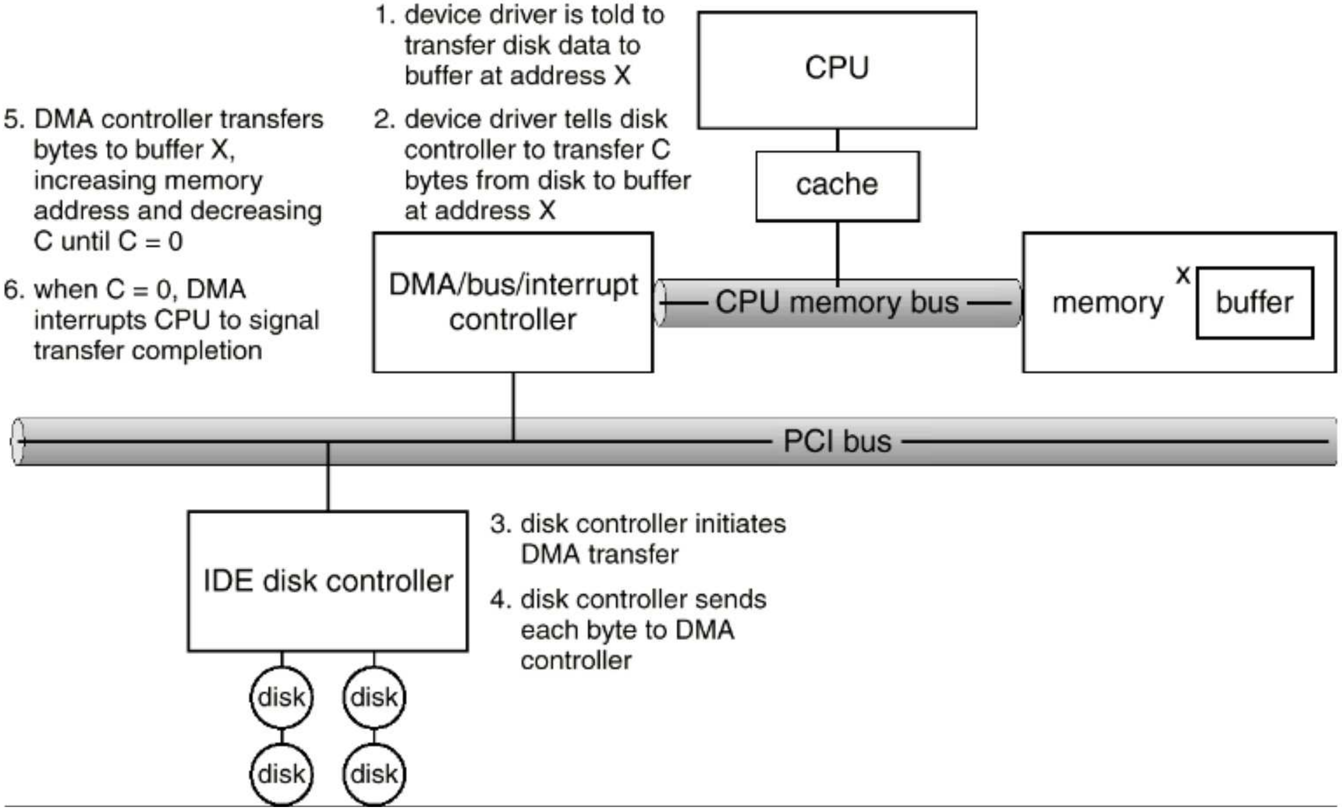
DMA: Direct Memory Access

- for large data transfers it is wasteful to use the main CPU to watch status bits and feed individual bytes to the controller (PIO)
- solution: offload some work to a special purpose processor — DMA controller
  - host initiates data transfer, provides the DMA controller with an address to transfer the data from/to, transfer size
  - DMA controller reads or writes the right amount of data accessing the main memory directly
  - raises an interrupt when done

DMA Example

- reading from disk
  - device driver initiates transfer of C bytes of data from disk to a buffer at address X
  - DMA controller is programmed accordingly, does handshaking with the disk controller
    - disk controller raises a signal on DMA-request wire when a word is available for transfer
    - DMA controller seizes the memory bus and raises a signal on the DMA-acknowledge wire
    - disk controller transfers the data and clears DMA-request
  - The hardware knows to increment the address X and decrement count C until done
  - when done the DMA controller interrupts the CPU

DMA Details



- when the DMA controller seizes the memory bus the CPU is prevented from accessing the main memory (but it can still access the caches) — “cycle stealing”

Principles of I/O Software I

- device independence
  - we would like to have as much software as possible — at least the user applications — to be device-independent
  - device classes
    - character
    - block
    - network
    - IDE
    - SCSI
    - USB
    - etc.
- error handling as close as possible to the source of errors (retries, resends, etc.)

Principles of I/O Software II

- character and block devices
  - character: get/put + buffers + libraries
  - block: read/write (and seek for random access), memory mapping.
- sequential and random access
- synchronous and asynchronous
  - synchronous — easy to understand and use
  - asynchronous — more efficient
- blocking and non-blocking I/O
  - non-blocking — does what it can and returns
  - asynchronous — will do everything, but later
- sharable and dedicated devices: is simultaneous access by more than one process or thread possible?

Structure of I/O Software

- 4 basic layers
  - interrupt handlers
  - device drivers
  - device-independent OS software (file systems, print spool handlers, communication protocol stacks etc.)
  - user applications

Interrupt Handlers

- almost all systems have them
- mark the end of an I/O operation on a controller
- sometimes signal an “independent” event (key pressed, a mouse movement)
- they should be short and effective
- sometimes divided into two stages
  - quick essential treatment to clear the interrupt
  - heavier, longer tasks deferred for later handling
  - bottom-halves and top-halves (Linux)
  - deferred procedure calls (Windows)

Device Drivers

- all device dependent code should go there
- a device driver handles a specific device, a device type, or a class of related devices
- device drivers issue the commands to the controllers and check that the results are as expected
- device drivers accept device independent requests from the software layers above and translate them in actual "command" to the device
- after an I/O operation is started the results may be available immediately or after a time in which case the driver will have to wait
- an interrupt — indicating the end-of-operation will wake up the driver

Device-Independent and User Software

- device-independent I/O software
  - service libraries — e.g., for buffer manipulation; might have device dependent parts but are common to all
    - buffer allocation, buffering
    - storage allocation on block devices
    - allocation and release of non-shared devices
    - error reporting
  - interface software
    - uniform driver interfacing
    - device naming and location
    - device protection
- user level I/O software: standard libraries, spoolers, utilities

I/O Subsystem I

- scheduling
  - queue I/O requests per device
  - queues reordered for efficiency, fairness
- use of buffering
  - cope with speed mismatch
    - double buffering
  - deal with different data transfer sizes
    - fragmentation and reassembly of network packets
  - support “copy semantics” for application I/O
    - copy to a kernel buffer before writing to device

I/O Subsystem II

- caching - fast memory holding copy of data
  - always just a copy with fast access
  - key to performance
- spooling - hold output for a device
  - if the device cannot interleave data streams
  - a daemon process or a kernel thread
- device reservation - provides exclusive access to a device
  - system calls for allocation and deallocation
  - watch out for deadlock

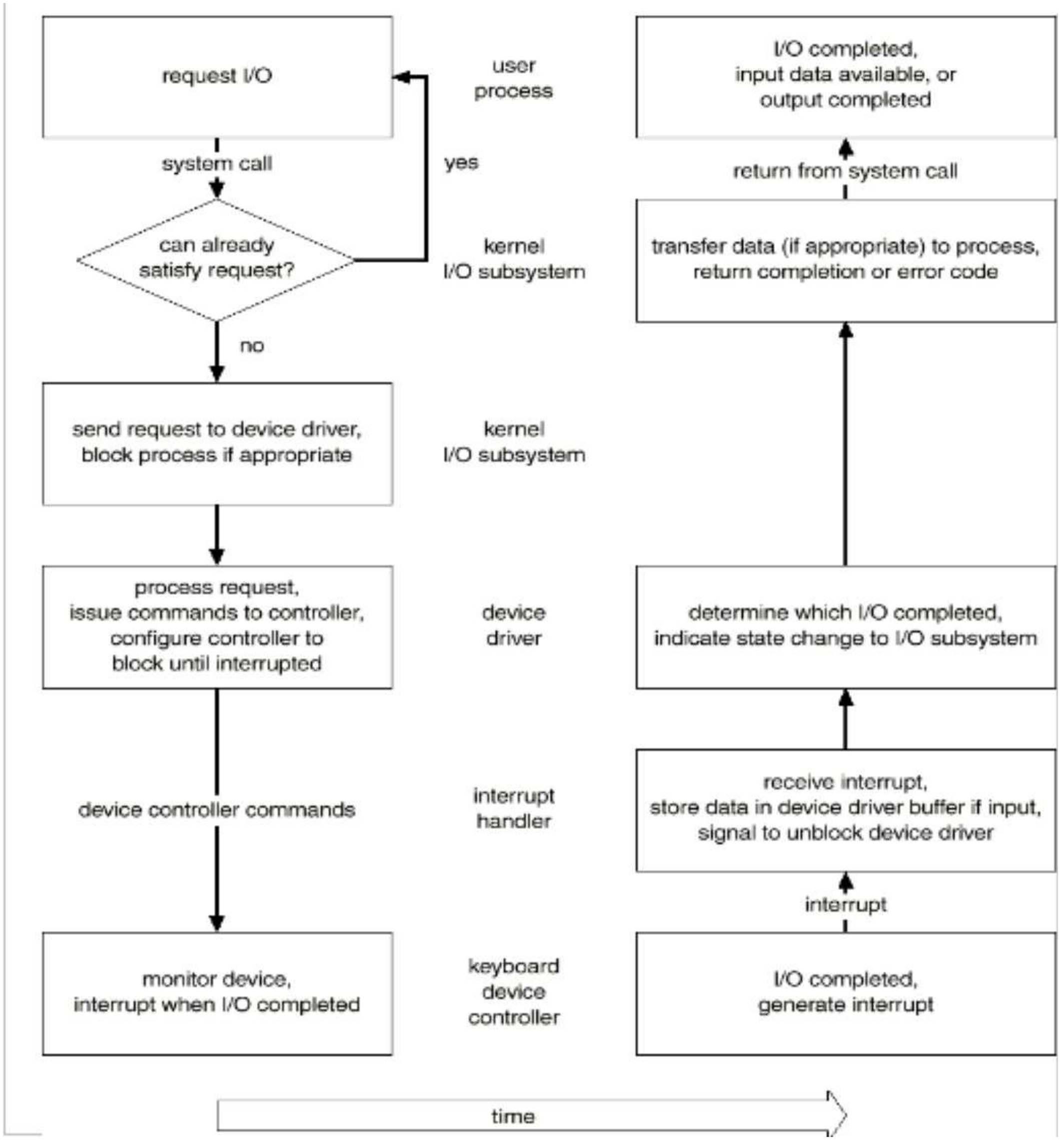
Kernel Data Structures

- kernel keeps state info for I/O components, including open file tables, network connections, character device state
  - `lsof(8)` (UNIX, Linux)
  - `netstat(8)` (UNIX, Linux)
  - ...
- many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- some use object-oriented methods and message passing to implement I/O
  - may add overhead compared to procedural techniques using shared data structures
  - may simplify design, add robustness, flexibility

I/O Example: Reading A File

- filesystem maps the file name to disk blocks
  - DOS: file access table
  - UNIX: inode structure
- mapping a file name to a disk controller
  - DOS: part of the file name identifies the specific hardware device ("C:\dir\file")
    - "C:" $\rightarrow$ port address through device table
    - device and filesystem namespaces are separate
  - UNIX: "everything is a file" ("/dev/hda")
    - single namespace — mount table identifies device
    - device major number identifies driver
    - device minor number $\rightarrow$ port address or memory-mapped address of the controller (index in the device table)

Life Of An I/O Transaction



I/O Performance I

- I/O performance is critical as CPU speed increases faster than I/O and I/O becomes critical bottleneck
  - programmed I/O can be more efficient than interrupt-driven I/O in some cases
    - interrupt handling may be inefficient
    - blocking/unblocking involves context switch
- Networking generates context switches
- 2 performance metrics
  - bandwidth (how much data can we move through the system in a unit of time)
  - I/O operations/sec
  - other metrics (e.g., latency) may be relevant in some cases (e.g., network I/O)

I/O Performance II

- software is critical for I/O performance — it should make best use of resources
  - reduce the number of context switches
  - reduce copying of data
  - reduce frequency of interrupts
  - offload to DMA, hardware
  - balance CPU, memory, bus, I/O performance

How Many Interrupts Can We Handle?

- what if a device generates a lot of interrupts?
- can happen, e.g., for network devices: a NIC flooded by packets
  - high network load due to legitimate traffic
  - denial of service attack
- if the interrupt handler eats all the cycles the device consumer (ultimately, the application) will not run (“livelock”)
- the OS will lose interrupts (i.e., packets, data)
- possible mechanism: set a timer, when it expires and the device claims it is ready, it indicates an interrupt was lost — invoke the handler manually

Interrupt Coalescing (Linux NAPI)

- receive one packet, disable further receive interrupts, raise an interrupt for the first packet
- the interrupt signals the kernel that the NIC is to be polled
- the backlog will be processed, the maximal backlog size that can be processed in one invocation is tunable
- interupts are enabled again
- interrupt-driven for low load, polled for high load
- may reduce parallelism on SMP systems (only one CPU polls)
- interrupts (and packets) may be dropped — the network infrastructure is relied upon to retransmit

Data Integrity

- CPU, memory, etc. are (or should be) disposable
- data are unique, irreplaceable, mission-critical
- detach the disks from the computers: Fiber Channel, iSCSI
- what if your disk fails?
- backup
  - what to back up?
  - how much to back up?
  - how often to back up?
  - when to back up?
  - where to back up?
- what if your disk fails between backups?

RAID

- Patterson, Gibson, Katz (1987, UC Berkeley): “A Case for Redundant Arrays of Inexpensive Disks (RAID)”
  - combine multiple small, inexpensive disk drives into an array of disk drives which yields performance exceeding that of a Single Large Expensive Drive (SLED)
  - the array of drives appears to the computer as a single logical drive
  - improve fault tolerance by storing data redundantly
  - improve performance by parallelizing I/O

RAID 0 — Striping

- write a $500\,\mathrm{M}$ to a $40\,\mathrm{G}$ disk – the disk is the bottleneck
- what if you have 2 $20\,\mathrm{G}$ disks, write $250\,\mathrm{M}$ to each?
- or 5 $8\,\mathrm{G}$ disks, write $100\,\mathrm{M}$ to each?
- organize data in “stripes” in a round-robin fashion
- stripes can be as small as a 512-byte sector, or can be many M each
- no redundancy, just performance improvement
  - one of the disks fails — you lose your data
- use identical disks — the slowest is the bottleneck, the smallest determines the size

RAID 1 — Mirroring

- what about reliability?
- make one disk automatic backup for the other in a RAID of 2
- the RAID controller writes to several disks automatically
- can be more than 2 disks for multiple redundancy
- obvious tradeoff — loss in capacity
- again — use identical disks

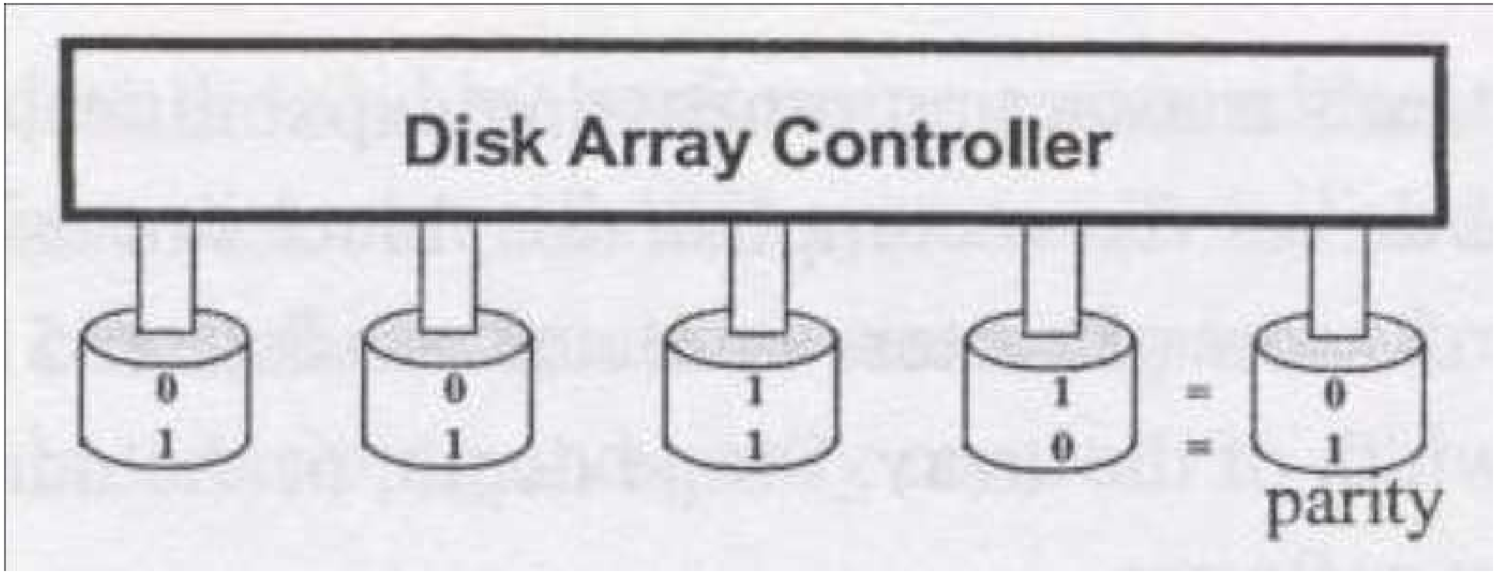
RAID 0/1, 1/0 And Spanning

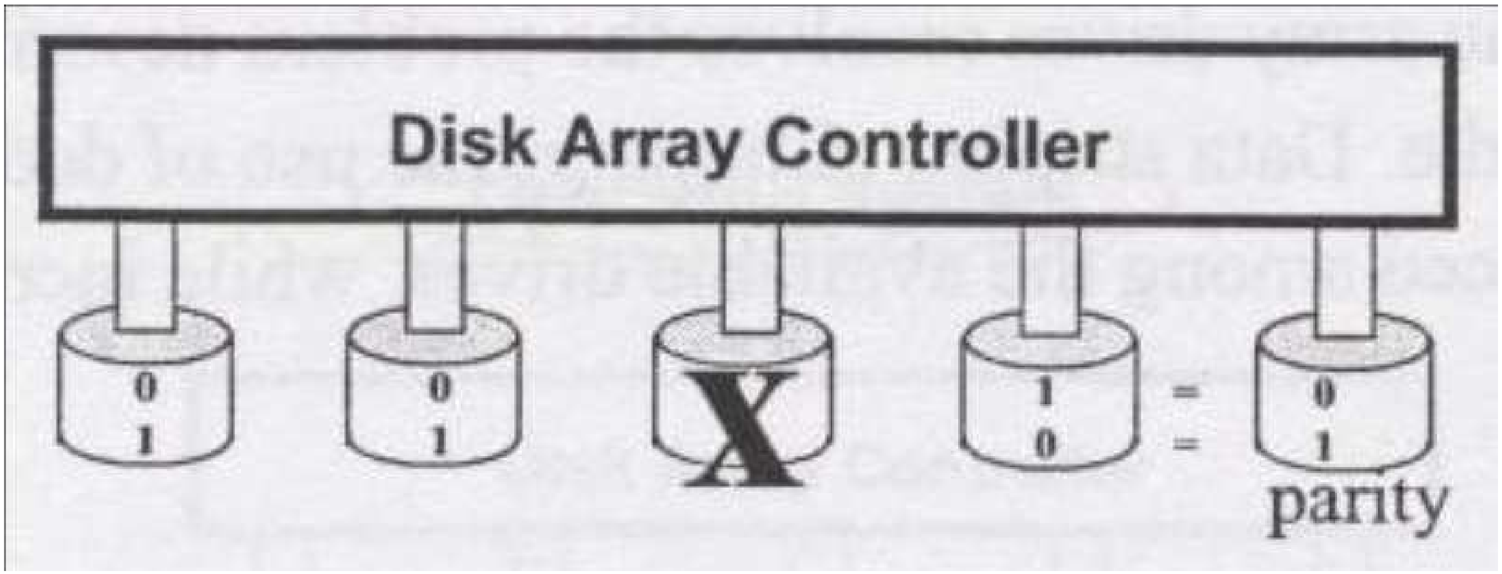
- how to achieve both performance and reliability?
- combine RAID 0 and RAID 1 — mirror 2 sets of striped disks
  - set up a RAID 0 array
  - duplicate it, make one a mirror of the other
- RAID 1/0 — a stripe of mirrors — is also possible
- spanning
  - a convenience feature (not performance or reliability)
  - 2 or more drives, possibly different, can be combined to form a larger logical drive
  - some OS (e.g., WinNT) can do this on their own

RAID Levels 2 to 4

- RAID 2: error correction for disks who cannot do it on their own
  - rarely used, as most modern disks have error correction
- RAID 3: byte-level striping, with parity stored on a separate disks
  - similar to RAID 4, byte-level striping requires HW support for efficiency
- RAID 4:
  - stores parity information on one drive
  - allows recovery from a single disk failure
  - efficient for reads, large or sequential writes
  - worse performance for small random writes

RAID 4: Parity

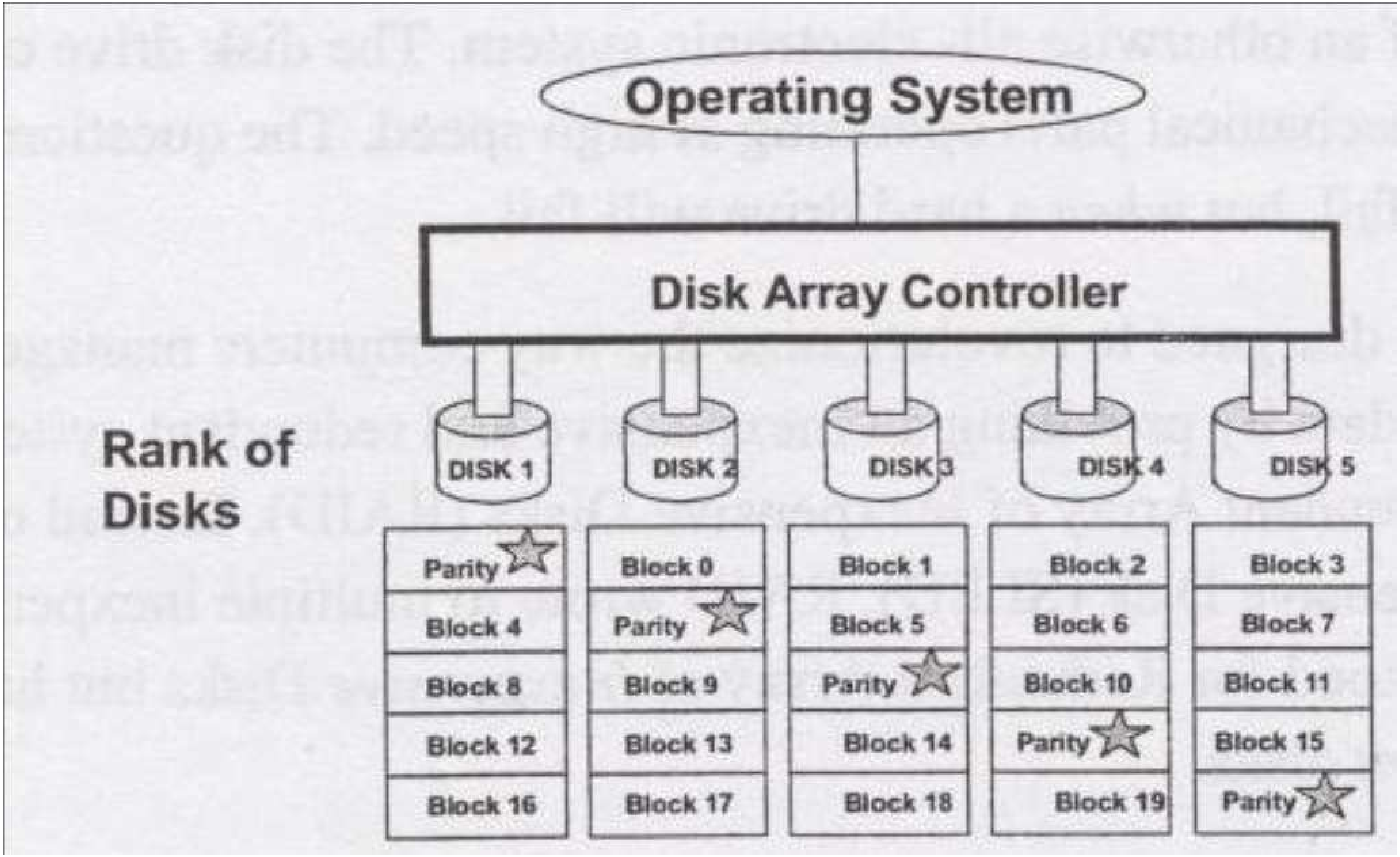




RAID 5

- similar to RAID 4, but parity data is distributed among the drives in the array
- may speed up small random writes since the single parity disk of RAID 4 is no longer a bottleneck
- must skip the parity data for reads — lower performance
- requires at least 3 disks, typically 5 disks.

RAID 5: Parity



Hardware vs. Software RAID

- hardware RAID: the host sees a single disk instead of the array
  - controller-based or external SCSI RAID
  - a RAID controller may span multiple SCSI channels
- software RAID — dependent on the OS, not on hardware
  - occupies host memory, consumes CPU
  - performance depends on the host CPU, load
  - what if the software fails to boot because of a failure in one of the array drives?
  - may require a separate (not included in the array) boot drive