Operating Systems Principles

Device I/O, Techniques & Frameworks

Mark Kampe
(markk@cs.ucla.edu)

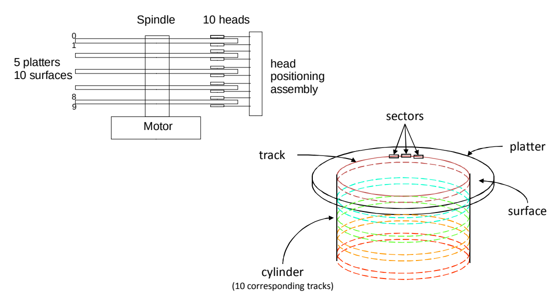
Final Project

- Value … 10% of course grade (same as P1, P3)
- You have two options:
  – OS research paper
    - get topic approved by TA this or next week
  – InternetOfThings embedded security project
    - tell TA this week, check out Edison next week
- (draft) project descriptions on course calendar
  web.cs.ucla.edu/classes/spring16/cs111/projects/Paper.html
  web.cs.ucla.edu/classes/spring16/cs111/projects/Edison.html

Device I/O, Techniques & Frameworks

12A. Disks
12B. Low Level I/O Techniques
12C. Higher Level I/O Techniques
12D. Plug-in Driver Architectures

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 >

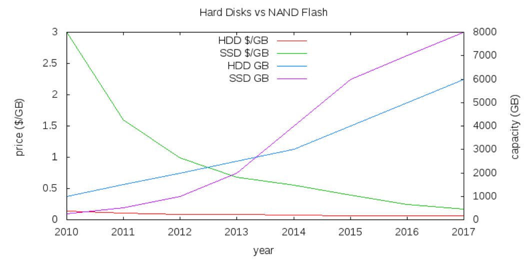
Disks have Dominated File Systems

- 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

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



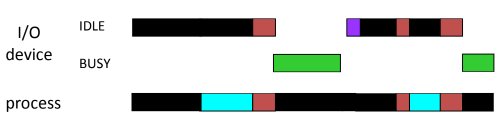
The Changing I/O Landscape

- Storage paradigms
  - old: swapping, paging, file systems, data bases
  - 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

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

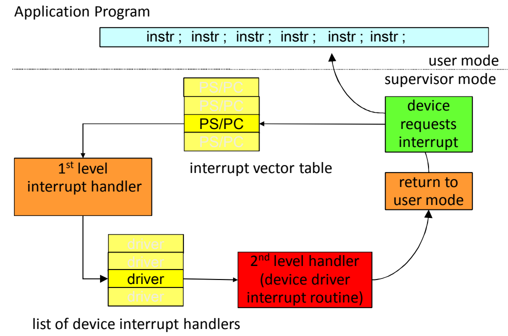
Direct Memory Access

- 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



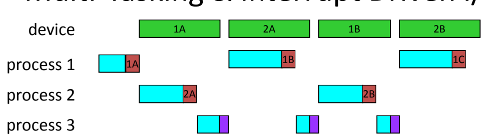
Keeping Key Devices Busy

- 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

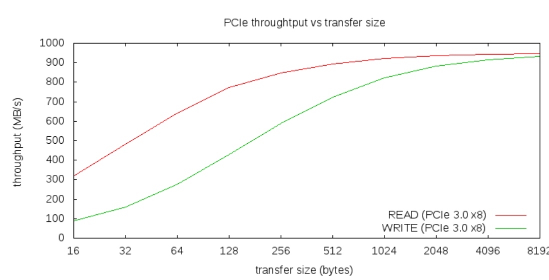
```
xx_read/write() {
    allocate a new request descriptor
    fill in type, address, count, location
    insert request into service queue
    if (device is idle) {
            disable_device_interrupt();
                xx_start();
            enable_device_interrupt();
    }
    await completion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

Bigger Transfers are Better



(Bigger Transfers are Better)

- disks have high seek/rotation overheads
  - larger transfers amortize down the cost/byte
- all transfers have per-operation overhead
  - instructions to set up operation
  - device time to start new operation
  - time and cycles to service completion interrupt
- larger transfers have lower overhead/byte
  - this is not limited to s/w implementations

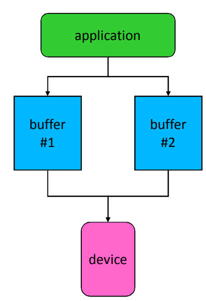
Input/Output Buffering

- Fewer/larger transfers are more efficient
  - they may not be convenient for applications
  - natural record sizes tend to be relatively small
- Operating system can buffer process I/O
  - maintain a cache of recently used disk blocks
  - accumulate small writes, flush out as blocks fill
  - read whole blocks, deliver data as requested
- Enables read-ahead
  - OS reads/caches blocks not yet requested

Deep Request Queues

- Having many I/O operations queued is good
  - maintains high device utilization (little idle time)
  - reduces mean seek distance/rotational delay
  - may be possible to combine adjacent requests
- Ways to achieve deep queues:
  - many processes making requests
  - individual processes making parallel requests
  - read-ahead for expected data requests
  - write-back cache flushing

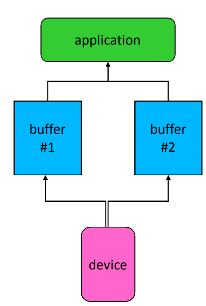
Double-Buffered Output



(double-buffered output)

- multiple buffers queued up, ready to write
  - each write completion interrupt starts next write
- application and device I/O proceed in parallel
  - application queues successive writes
    - don't bother waiting for previous operation to finish
  - device picks up next buffer as soon as it is ready
- if we're CPU-bound (more CPU than output)
  - application speeds up because it doesn't wait for I/O
- if we're I/O-bound (more output than CPU)
  - device is kept busy, which improves throughput
  - but eventually we may have to block the process

Double-Buffered Input



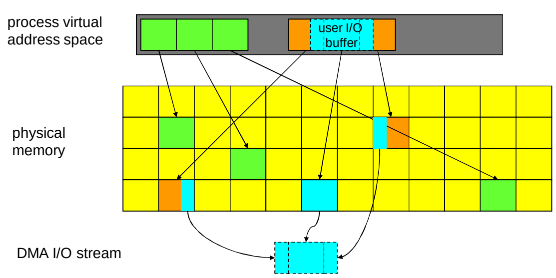
(double buffered input)

- have multiple reads queued up, ready to go
  - read completion interrupt starts read into next buffer
- filled buffers wait until application asks for them
  - application doesn't have to wait for data to be read
- when can we do chain-scheduled reads?
  - each app will probably block until its read completes
    - so we won't get multiple reads from one application
  - we can queue reads from multiple processes
  - we can do predictive read-ahead

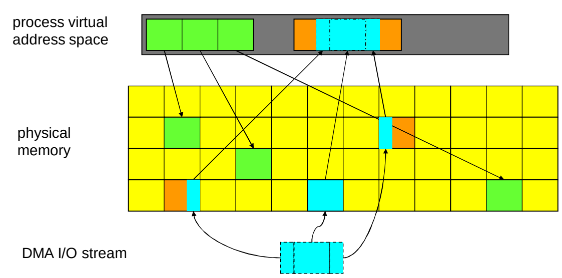
Scatter/Gather I/O

- many controllers support DMA transfers
  - entire transfer must be contiguous in physical memory
- user buffers are in paged virtual memory
  - user buffer may be spread all over physical memory
  - *scatter*: read from device to multiple pages
  - *gather*: writing from multiple pages to device
- three basic approaches apply
  - copy all user data into contiguous physical buffer
  - split logical req into chain-scheduled page requests
  - I/O MMU may automatically handle scatter/gather

"gather" writes from paged memory



"scatter" reads into paged memory



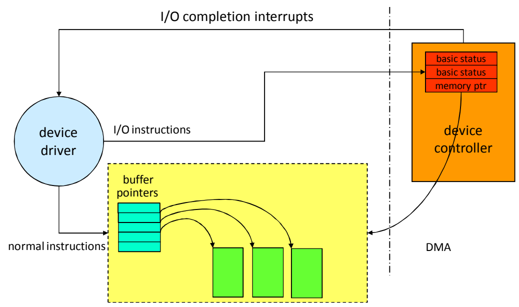
mechanisms: memory mapped I/O

- 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



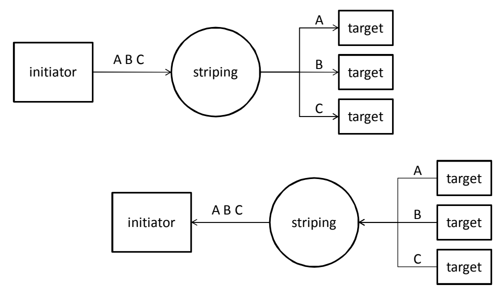
(I/O Mechanisms: smart controllers)

- Smarter controlers 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 perormance
  - they can do automatic error handling & retries
- abstract away details of underlying devices

User-Mode Device Drivers

- why are drivers integrated into the OS
  - they need to used (privileged) I/O instructions
  - they need to service I/O interrupts
  - they are trusted with multi-user data
- these reasons become less compelling
  - memory mapped devices don't need I/O instrs
  - polled smart devices may not need interrupts
  - privileged processes are trusted
  - performance/robustness may be better

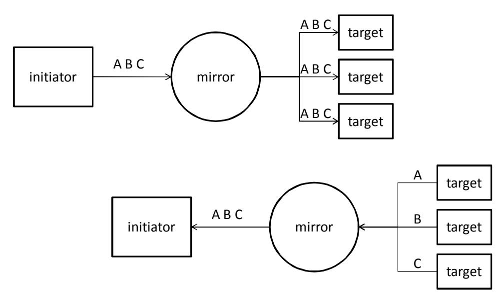
Data Striping for Bandwidth



(Data Striping for Bandwidth)

- spread requests across multiple targets
  - increased aggregate throughput
  - fewer operations per second per target
- used for many types of devices
  - disk or server striping
  - NIC bonding
- potential issues
  - more/shorter requests may be less efficient
  - source can generate many parallel requests
  - target throughput is the bottleneck

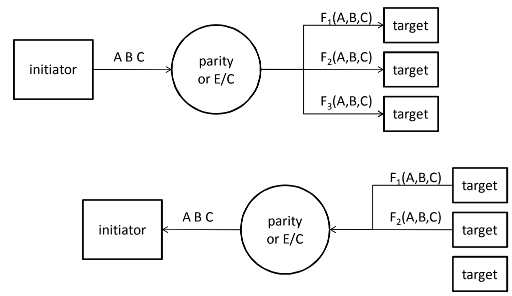
Data Mirroring for Reliability



(Data Mirroring for Reliability)

- mirror writes to multiple targets
  - redundancy in case a target fails
  - spread reads across multiple targets
    - increased aggregate throughput, reduced ops/target
- used for all types of persistent storage
  - disks, NAS, distributed key/value stores
- potential issues
  - added write traffic on the source
  - 2x-3x storage requirements on targets

Parity/Erasure Coding for Efficiency



(Parity/Erasure Coding for Efficiency)

- N out of M encoding (with M/N overhead)
  - accumulate N writes from source
  - compute M versions of that collection
  - send a version to each of M targets
- Commonly used for archival storage
- Potential issues
  - greatly increased source computational load
  - deferred writes for parity block accumulation
  - expensive updates, recovery (and EC reads)

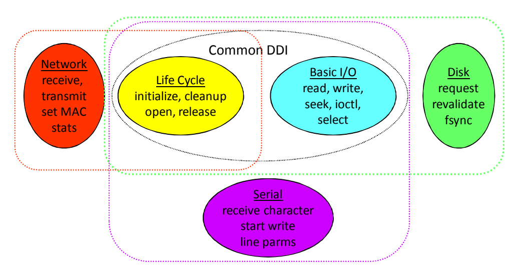
Drivers – generalizing abstractions

- OS defines idealized device classes
  - disk, display, printer, tape, network, serial ports
- classes define expected interfaces/behavior
  - all drivers in class support standard methods
- device drivers implement standard behavior
  - make diverse devices fit into a common mold
  - protect applications from device eccentricities
- software analog to h/w device controllers
  - device drivers connect a device controller to an OS

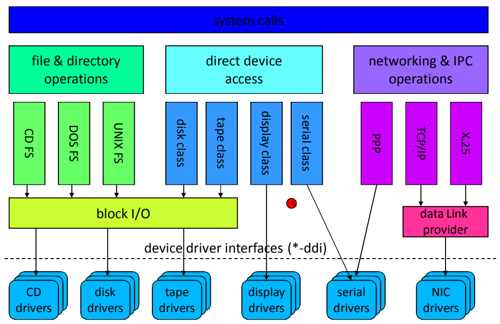
Device Driver Interface (DDI)

- standard (top-end) device driver entry-points
  - basis for device independent applications
  - enables system to exploit new devices
  - a critical interface contract for 3rd party developers
- some correspond directly to system calls
  - e.g. open, close, read, write
- some are associated w/OS frameworks
  - disk drivers are meant to be called by block I/O
  - network drivers are meant to be called by protocols

DDIs and sub-DDIs



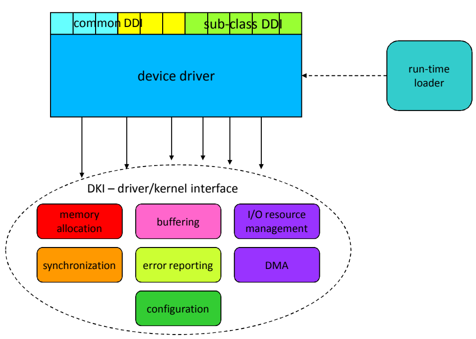
Standard Driver Classes & Clients



Drivers – simplifying abstractions

- encapsulate knowledge of how to use device
  - map standard operations into operations on device
  - map device states into standard object behavior
  - hide irrelevant behavior from users
  - correctly coordinate device and application behavior
- encapsulate knowledge of optimization
  - efficiently perform standard operations on a device
- encapsulation of fault handling
  - knowledge of how to handle recoverable faults
  - prevent device faults from becoming OS faults

Kernel Services for device drivers



(Driver/Kernel Interface)

- (bottom-end) services OS provides to drivers
  - analogous to an ABI for device driver writers
- must be very well-defined and stable
  - to enable 3rd party driver writers to build drivers
  - so old drivers continue to work on new OS versions
- each OS has its own DKI, but they are all similar
  - memory allocation, data transfer and buffering
  - I/O resource (e.g. ports, interrupts) mgt, DMA
  - synchronization, error reporting
  - dynamic module support, configuration, plumbing

Criticality of Stable Interfaces

- Drivers are independent from the OS
  - they are built by different organizations
  - they are not co-packaged with the OS
- OS and drivers have interface dependencies
  - OS depends on driver implementations of DDI
  - drivers depends on kernel DKI implementations
- These interfaces must be carefully managed
  - well defined and well tested
  - upwards-compatible evolution

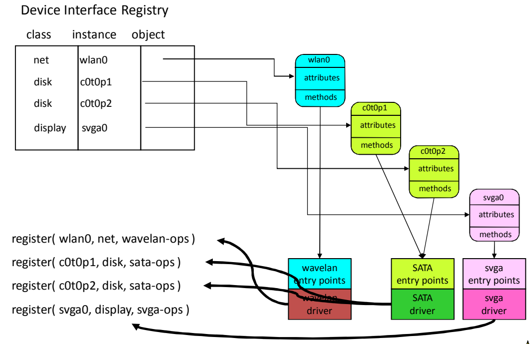
UNIX: special files

- how does one open an instance of a device
  - like everything else, by opening some named file
- special files
  - files that are associated with a device instance
  - UNIX/LINUX uses <block/character, major, minor>
    - major number corresponds to a particular device driver
    - minor number identifies an instance under that driver
- opening special file opens the associated device
  - open/close/read/write/etc calls map into calls to the appropriate DDI entry-points of the selected driver

UNIX: device instances

- minor device # is an instance under a driver
  - meaning of minor number is entirely driver-specific
- instances may be physically distinct
  - e.g. different serial ports, different disk drives
- instances may refer to multiplexed sub-devices
  - e.g. one of four FDISK partitions on a hard disk
  - e.g. a sub-channel on a communications interface
- instances may merely select different options
  - e.g. enable rewind-on-close for a tape drive
  - e.g. different densities for diskettes

Registering Dynamic Driver Instances



(driver instance/interface registration)

- driver must register each device instance
  - register name, class, and instance # of device
  - so programs will know that instance is available
- register driver methods for accessing that device
  - driver advertises its entrypoints for all methods
    - which methods depend on the class and driver
  - enables other s/w to use device instance/call driver
- OS includes services to register and un-register
  - e.g. register_chrdev(major ID, minor ID, operations)
  - create special file for accessing device instance

Assignments

- for the next lecture:
  - File Formats (Wikipedia)
  - Arpaci ch 39 … Files and Directories
  - Arpaci ch 40 … File System Implementation
  - FAT (DOS) file system format
  - Object Stores (history, architecture)
  - Key-Value Stores (introduction, types)
  - FUSE (file systems in user mode)