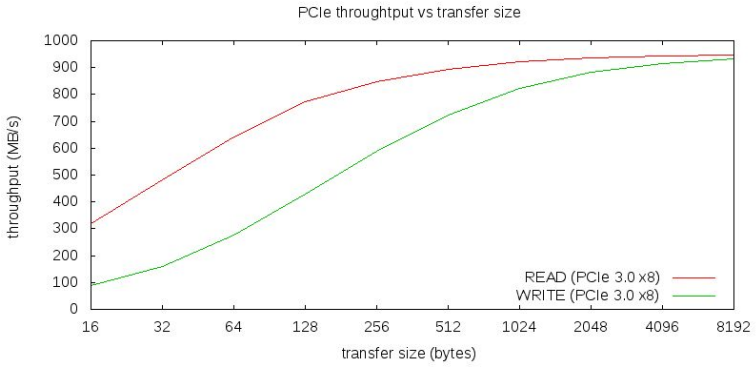
I/O Techniques and Performance

- 10D Efficient I/O Operations
- 10E RAID Performance and Reliability
- 10I Polled/non-blocking I/O
- 10J User-mode Asynchronous I/O
- 10U User-mode Device Drivers

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
  - amortize startup costs over more bytes
  - this is not limited to s/w implementations

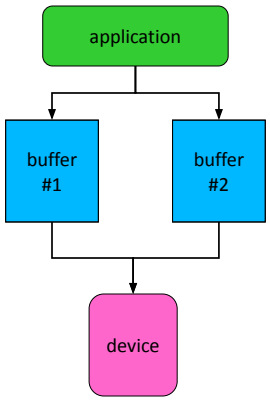
Input/Output Buffering

- 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

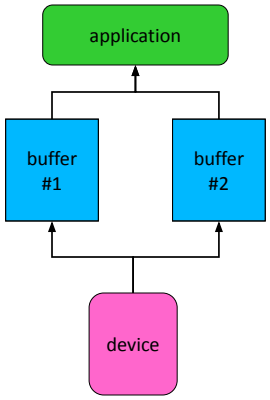
Double-Buffered Output



(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, 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



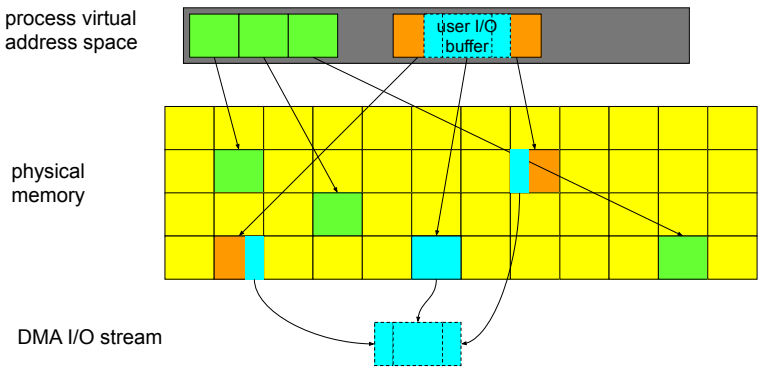
(double buffered input)

- 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?
  - most apps will probably block until 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
  - some apps can queue parallel reads

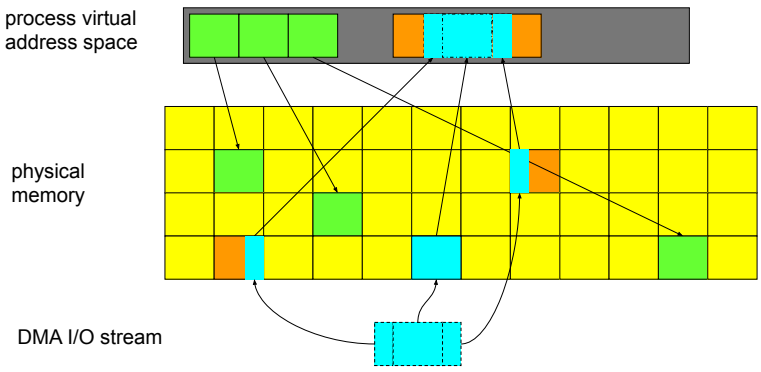
Scatter/Gather I/O

- 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

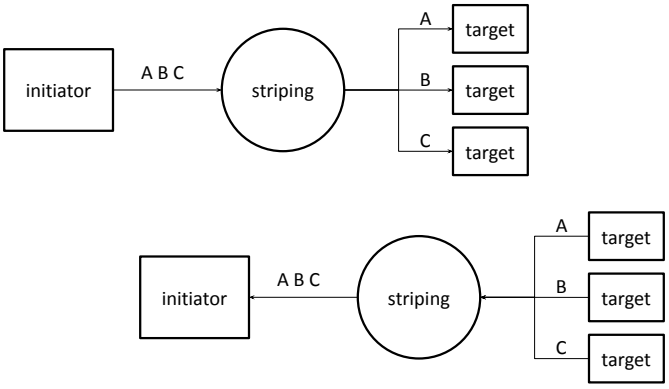
"gather" writes from paged memory



"scatter" reads into paged memory



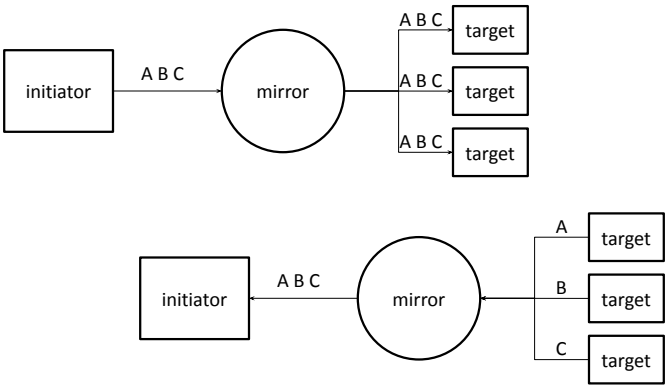
Data Striping for Bandwidth



(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/server striping/sharding
  - NIC bonding
- potential issues
  - more/shorter requests may be less efficient
  - source can generate many parallel requests
  - striping agent throughput is the bottleneck

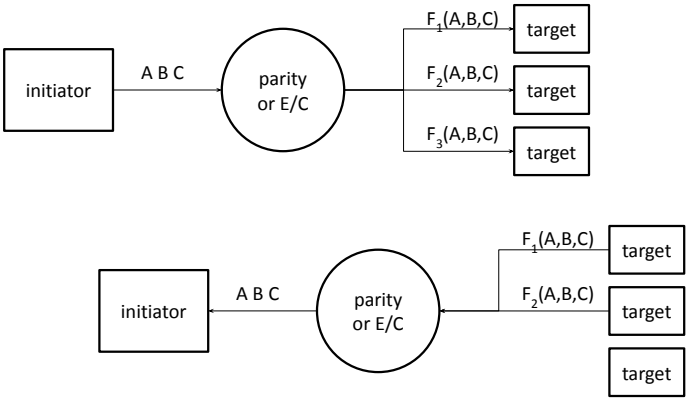
Data Mirroring for Reliability



(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
  - deciding which (conflicting) copy is correct

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)
  - choosing the right ratio

Error Detection/Correction Terms

- Parity (typically XOR)
  - typically one bit per byte, detect single-bit errors
  - used as redundancy, it can recover one lost bit/block
- Cyclical Redundancy Check (CRC)
  - multiple bits per record, detect multi-bit burst errors
- Error Correcting Coding (ECC)
  - fixed ratio, capable of detection and correction
  - e.g. Reed-Soloman (204,188) can correct 8 bad bits
- Erasure Coding (distributed Reed-Soloman)
  - transforms K blocks into N (>K)
  - all can be recovered from any K (of those N) blocks

Parallel I/O Paradigms

- Busy, but periodic checking just in case
  - new input might cause a change of plans
- Multiple semi-independent streams
  - each requiring relatively simple processing
- Multiple semi-independent operations
  - each requiring multiple, potentially blocking steps
- Many balls in the air at all times
  - numerous parallel requests from many clients
  - keeping I/O queues full to improve throughput

Enabling Parallel Operations

- Threads are an obvious solution
  - one thread per-stream or per-request
  - streams or requests are handled in parallel
  - when one thread blocks, others can continue
- There are other parallel I/O mechanisms
  - non-blocking I/O
  - multi-channel poll/select operations
  - asynchronous I/O

Non-Blocking I/O

- check to see if data/room is available
  - but do not block to wait for it
  - this enables parallelism, prevents deadlocks
- a file can be opened in a non-blocking mode
  - open(name, flags | O_NONBLOCK)
  - fcntl(fd, F_SETFL, flags | O_NONBLOCK)
- if data is available, *read(2)* will return it
  - otherwise it fails with EWOULDBLOCK
- can also be used with *write(2)* and *open(2)*

Multi-Channel Poll/Select

- there are multiple possible input sources
  - parallel streams (e.g. ssh input/output)
  - multiple request generating clients
- *poll(2)*/*select(2)* wait for first interesting event
  - a list of file descriptors to be checked
  - a list of interesting events (input, output, error)
  - a maximum time to wait
  - a signal mask (to use while waiting)
- do read/write on indicated file descriptor(s)

Worker Threads

- Consider a web or remote file system server
  - it receives thousands of requests/second
  - each requires multiple (blocking) operations
  - create a thread to serve each new request
- Thread creation is relatively expensive
  - continuous creation/destruction seems wasteful
  - solution: recycle the worker threads
    - thread blocks when its operation finishes
    - it is awakened when a new operation needs servicing
- we still have switching and synchronization

NBIO vs. Poll/Select vs. Threads

- NBIO … very simple
  - occasional checks for unlikely input
  - cost of wasted spins is not a concern
- Poll/Select … efficient multi-stream processing
  - multiple sources of interesting input/event
  - wait for the first available, serve one at a time
- Parallel Threads … for complex operations
  - all operations proceed in parallel, not just I/O
  - blocking operation does not block other threads
- None are practical for massive parallelism

Asynchronous I/O

- Huge numbers of parallel I/O operations
  - many parallel clients w/many parallel requests
  - deep I/O queues to improve throughput
  - make sure completions processed correctly
- thread per operation is too expensive
- we want to queue many parallel operations
  - receive asynchronous completion notifications
  - OS has always handled high traffic I/O this way
  - increasingly many applications now do as well

Scheduling Asynchronous I/O

int aio_read(struct aiocb *)

```
struct aiocb {
   int    aio_filedes;     // file descriptor
   off_t  aio_offset;      // file offset
   void  *aio_buf;         // local buffer
   int    aio_nbytes;      // byte count
   int    aio_reqprio;     // request priority
   sigevent aio_sigevent;  // notification method
}
```

- if successful, operation has been queued
  - it will complete at some time in the future
- a very large number of ops can be outstanding

Completion Checks/Waits

- we can poll the status of any operation
  - int aio_error(struct aiocb *)
  - int aio_return(struct aiocb *)
  - returns 0, EINPROGRESS, or completion error
- we can await completion of some operations
  - int aio_suspend(struct aiocb *, #items, timeout)
  - returns when one or more complete (or timeout)
- we can cancel or force any operations
  - int aio_cancel(struct aiocb *)
  - int aio_fsync(fd)

Completion Notifications

struct sigevent {

```
   int sigev_notify;  // by signal or thread
   int sigev_signo;   // notification signal #
   int sigev_value;   // param to signal handler
   void (*handler)(int);// handler to invoke
   …
```

- user-mode analog of completion interrupts
  - completion generates a specified signal
  - completion creates a specified thread

Signals for Event Notifications

- Signals were originally designed for exceptions
  - infrequent events, most often fatal
  - multiple race conditions in handling/disabling
  - bad semantics were “good enough”
- Now they are used for event notifications
  - continuous events in normal operation
  - loss of even a single event is unacceptable
  - they need to be safe and reliable
- We have long known how to do this properly
  - make them more like h/w interrupts

sigaction(2)

```
int sigaction (int signum, sigaction *new, sigaction *old)
    struct sigaction {
      void (*handler)(int);          // handler
      void (*action)(int, siginfo); // handler
      sigset_t     mask;             // signals to block
      int          flags;            // handling options
```

- mask eliminates reentrancy races
- siginfo passes much info about cause of signal
- *sigreturn(2)* controls return from handler

Asynchronous I/O: Back to the Future

- OS I/O always asynchronous, interrupt driven
  - necessary to achieve throughput and efficiency
  - apps were given comforting synchronous illusion
    - until they needed major throughput and efficiency
- simpler, more s/w-like mechanisms were tried
  - they were much less efficient
  - they proved race-prone under heavy use
- h/w interrupt model is refined, well proven
  - if there was a simpler way, we would be using it
  - the same model works well for s/w events too

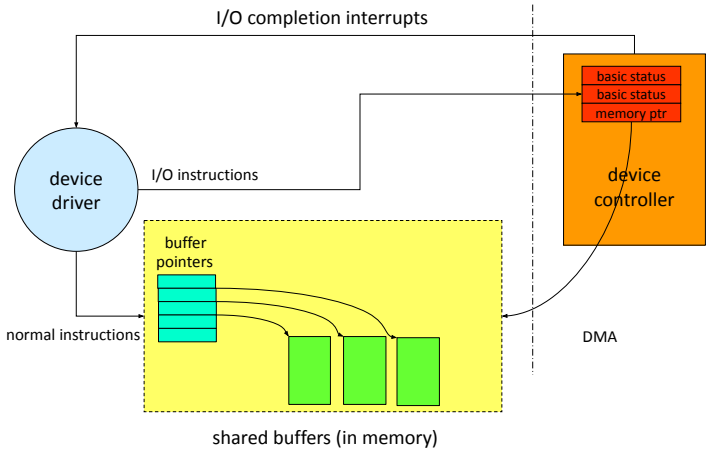
User-Mode Drivers: Why?

- Kernel-mode code is brittle
  - if it crashes, it takes the OS with it
- Kernel-mode code is hard to build and test
  - correctness rules are extremely complex
  - debugging tools are relatively crude
- Kernel-mode code is hard to upgrade
  - often necessary to reboot the system
- Kernel-mode code is not necessarily fast
  - system calls and interrupts are very expensive
  - processes can be pinned to memory and cores

User-Mode Drivers: How?

- Doesn't I/O require privileged instructions?
  - many ISAs allow user-mode I/O to limited ports
  - I/O space may be mapped into user address space
- Doesn't I/O require access to DMA controller?
  - some devices (e.g. graphics) don't need DMA
  - smart devices have on-board DMA controllers
    - all DMA is done to/from device-owned memory
- Doesn't I/O require interrupt handling?
  - smart devices have request queues, polled status

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 Drivers: Security?

- There is lots of trusted user-mode code
  - init, login, mail delivery, network protocols, …
  - there are even user-mode file systems
- Accessing I/O space is a privileged operation
  - it can be restricted to specific (privileged) UIDs
  - only a few programs can run w/those UIDs
  - file system security protects those programs
- Privileged User-mode code can be trusted
  - and safer than loadable kernel modules

User-Mode Drivers: Limitations

- They cannot use kernel services or data
  - they are ordinary user-mode programs
  - they do not execute in kernel mode
  - they do not run in kernel address space
- They open a driver to access the device
  - driver maps I/O device into process address space
  - driver handles configuration, interrupts, errors
- They cannot service interrupts
  - they must poll for asynchronous completions
  - but they may get signals for asynchronous errors

Reading and Assignments

Reading:

- Arpaci C39 … files
- Arpaci C40.1-3,6-8 … file systems
- Arpaci C40.5 … file system free space
- File Types, Key Value Stores, Object Storage
- (DOS) FAT file systems

Projects:

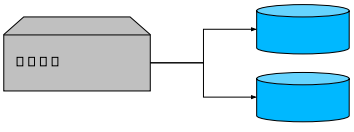
- we will help with P4B problems in the lab

Supplementary Slides

RAID

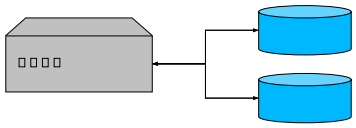
- disks are the weak point of any computer system
  - reliability: disk drives are subject to mechanical wear
    - mis-seeks: resulting in corrupted or unreadable data
    - head crashes: resulting in catastrophic data loss
  - performance: limited seek and transfer speeds
- these limitations are inherent in the technology
  - moving heads and rotating media
- don't try to build super-fast or reliable disks
  - build Redundant Array of Independent Disks
  - combine multiple cheap disks for better performance

Striping (RAID-0)



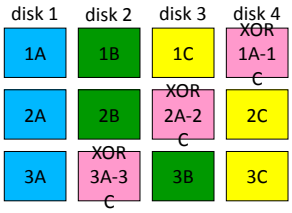
- combine them to get a larger virtual drive
  - striping: alternate tracks are on alternate physical drives
  - concatenation: 1st 500 cylinders on drive 1, 2nd 500 on drive 2
- benefits
  - increased capacity (file systems larger than a physical disk)
  - read/write throughput (spread traffic out over multiple drives)
- cost
  - increased susceptibility to failure

Mirroring (RAID-1)



- two copies of everything
  - all writes are sent to both disks
  - reads can be satisfied from either disk
- benefits
  - redundancy (data survives failure of one disk)
  - read throughput (can be doubled)
- cost
  - requires twice as much disk

Block-wise Striping w/Parity (RAID-5)



- dedicate 1/Nth of the space to parity
  - write data on N-1 corresponding blocks
  - Nth block contains XOR of the N-1 data blocks
- benefits
  - data can survive loss of any one drive
  - much more space efficient than mirroring
- cost
  - slower and more complex write performance

RAID implementation

- RAID is implemented in many different ways
  - as part of the disk driver
    - these were the original implementations
  - between block I/O and the disk drivers (e.g. Veritas)
    - making it independent of disks and controllers
  - absorbed into the file system (e.g. zfs)
    - permitting smarter implementation
  - built into disk controllers
    - potentially more reliable
    - significantly off-loads the host OS
    - exploit powerful Storage Area Networking (SAN) fabric

select(2)

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout, sigset_t *sigmask)

- fd_set is a bit-map of interesting file descriptors
- returns when event, timeout, or signal
- parameters updated to reflect what happened

- Created in 4.2BSD (1983)
  - older, more widely adopted

poll(2)

```
int ppoll( stuct pollfd *fds, int nfds, struct
    timespec *, sigset_t *)
struct pollfd {
    int fd;
    short events;     // requested events
    short revents;    // returned events
}
```

- returns when event, timeout, or signal
- revents reflect what happened

- Created in UNIX SVR3 (1986)
  - newer, perhaps a little better thought out