I/O Buffering and Streaming
I/O Buffering and Caching

I/O accesses are reads or writes (e.g., to files)
Application access is arbitrary (offset, len)
Convert accesses to read/write of fixed-size blocks or pages
Blocks have an (object, logical block) identity
Blocks/pages are cached in memory
  • Spatial and temporal locality
  • Fetch/replacement issues just as VM paging
  • Tradeoff of block size
I/O

- Application processing
- I/O initiation (e.g., syscall, driver, etc.)
- I/O access request latency (e.g., disk seek, network)
- Block transfer (disk surface, bus, network)
- I/O completion overhead (e.g., block copy)
Effective Bandwidth

Call this the *gap* $g$

Define $G$ to be transfer time per byte (bandwidth = $1/G$)

Block size is $B$ bytes; transfer time is $BG$

What’s the *effective bandwidth* (throughput)?
Impact of Transfer Size

\[ \frac{B}{(g_i + BG_i)} \]

- \( B \) = transfer size
- \( g_i \) = overhead (\( \mu s \))
- \( G \) = inverse bandwidth

For these curves, \( G \) matches 32-bit 33MHz PCI and Myrinet LANai-4 link speed (132 MB/s).
Bubbles in the I/O Pipeline

The CPU and I/O units are both underutilized in this example. In this case, latency is critical for throughput.

There are “bubbles” in the pipeline: how to overlap activity on the CPU and I/O units?

- Multiprogramming is one way
  But what if there is only one task?

Goals: keep all units fully utilized to improve throughput.
  Hide the latency
Prefetching or Streaming
Prediction

Compiler-driven
  • Compile-time information about loop nests, etc.

Markov prediction
  • “Learn” repeated patterns as program executes.

Pre-execution
  • Execute the program speculatively, and watch its accesses

Query optimization or I/O-efficient algorithm
  • “Choreograph” I/O accesses for complex operation

How to get application-level hints to the kernel?
  Hinting or asynchronous I/O
Readahead: the system predictively issues I/Os in advance of need. This may use low-level asynchrony or create threads to issue the I/Os and wait for them to complete (e.g., RPC-based file systems such as NFS).
Prefetching and Streaming I/O: Examples

Parallel disks

Latency for arm movement

Network data fetch
E.g., network memory
Fetch from server cache

Latency for request propagation
Prefetching and I/O Scheduling

Asynchronous I/O or prefetching can expose more information to the I/O system, which may allow it to schedule accesses more efficiently.

E.g., read one large block with a single seek/rotation.
The I/O Pipeline and I/O Overhead

Network data fetch
Bandwidth-limited

Faster network
CPU-limited

In this example, overhead rather than latency is the bottleneck for I/O throughput. How important is it to reduce I/O overhead as I/O devices get faster?
Can Prefetching Hurt Performance?

Prefetching “trades bandwidth for latency”.
- Need some bandwidth to trade…

Mispredictions impose a cost.

How deeply should we prefetch?
- Prefetching requires memory for the prefetch buffer.
- Must prefetch deeply enough to absorb bursts.
- How much do I need to avoid stalls

Fixed-depth vs. variable depth
- *Forestall*
Buffers with valid data are retained in memory in a buffer cache or file cache.

Each item in the cache is a buffer header pointing at a buffer.

Blocks from different files may be intermingled in the hash chains.

System data structures hold pointers to buffers only when I/O is pending or imminent.

- busy bit instead of refcount
- most buffers are “free”

Most systems use a pool of buffers in kernel memory as a staging area for memory<->disk transfers.
Why Are File Caches Effective?

1. **Locality of reference**: storage accesses come in clumps.
   - **spatial locality**: If a process accesses data in block B, it is likely to reference other nearby data soon.
     (e.g., the remainder of block B)
     
     example: reading or writing a file one byte at a time
   - **temporal locality**: Recently accessed data is likely to be used again.

2. **Read-ahead**: if we can predict what blocks will be needed soon, we can *prefetch* them into the cache.
   - most files are accessed sequentially
I/O Caching vs. Memory Caches

Associativity

software to track references

variable-cost backing storage (e.g., rotational)

what's different from paging?

- but don't need to sample to track references

Also: access properties are different
I/O Block Caching: When, What, Where?

Question: should I/O caching be the responsibility of the kernel?

…or…

Can/should we push it up to the application level?

(Be sure you understand the tradeoffs.)
Replacement

What’s the right cache replacement policy for sequentially accessed files?

How is replacement different from virtual memory page cache management?

How to control the impact of deep prefetching on the cache?

- Integrated caching and prefetching
Handling Updates in the File Cache

1. Blocks may be modified in memory once they have been brought into the cache.

   Modified blocks are *dirty* and must (eventually) be written back.

2. Once a block is modified in memory, the write back to disk may not be immediate (*synchronous*).

   - *Delayed writes* absorb many small updates with one disk write.
     How long should the system hold dirty data in memory?

   - *Asynchronous writes* allow overlapping of computation and disk update activity (*write-behind*).
     Do the *write* call for block \( n+1 \) while transfer of block \( n \) is in progress.

   - Thus file caches also can improve performance for writes.
Write-Behind

This is *write-behind*.
Delayed Writes

This is a *delayed write* strategy.
Write Batching/Gathering

This combines delayed write and write-behind.
Exploiting Asynchrony in Writes

Advantages:

- Absorb multiple writes to the same block.
- Batch consecutive writes to a single contiguous transfer.
- Blocks often “die” in memory if file is removed after write.
- Give more latitude to the disk scheduler to reorder writes for best disk performance.

Disadvantages:

- Data may be lost in a failure.
- Writes may complete out of order.
- What is the state of the disk after a failure?
- When to execute writes?
  - sync daemon with flush-on-close