

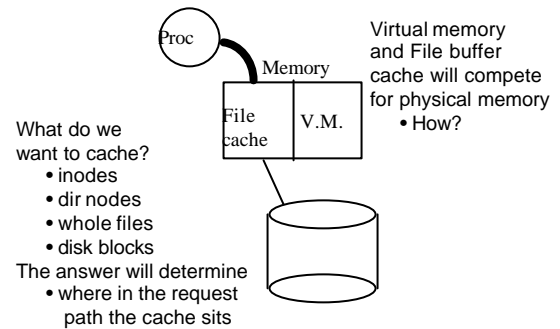
Outline for Today's Lecture

Administrative:

Objective:

- Focus in on File Buffer Cache

File Buffer Cache



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

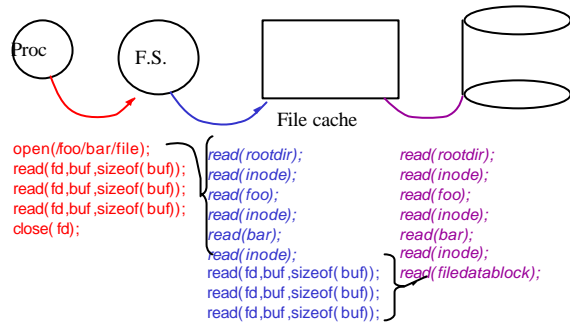
What to Cache? Locality in File Access Patterns (UNIX Workloads)

- Most files are **small** (often fitting into one disk block) although most bytes are transferred from longer files.
- Accesses tend to be **sequential and 100%**
 - **Spatial locality**
 - What happens when we cache a huge file?
- Most opens are for **read** mode, most bytes transferred are by read operations

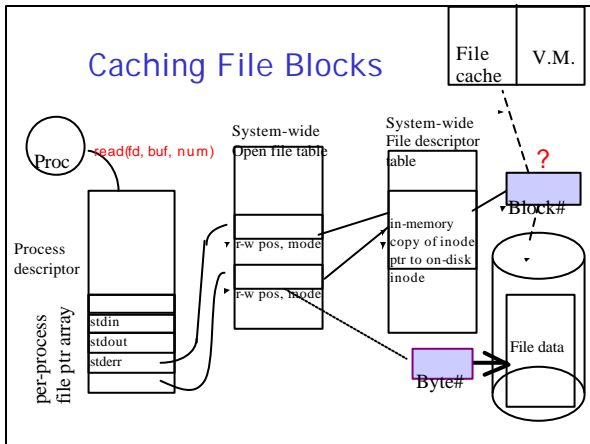
What to Cache? Locality in File Access Patterns (continued)

- There is significant **reuse** (re-opens) – most opens go to files repeatedly opened & quickly. Directory nodes and executables also exhibit good **temporal locality**.
 - Looks good for caching!
- Use of temp files is significant part of file system activity in UNIX – very limited reuse, short lifetimes (less than a minute).
- Long absolute pathnames are common in file opens
 - Name resolution can dominate performance – why?

Access Patterns Along the Way



Caching File Blocks



Issues for I/O Cache Structure

Goal: maintain K slots in memory as a cache over a collection of m items on secondary storage ($K \ll m$).

1. What happens on the first access to each item?

Fetch it into some slot of the cache, use it, and leave it there to speed up access if it is needed again later.

2. How to determine if an item is resident in the cache?

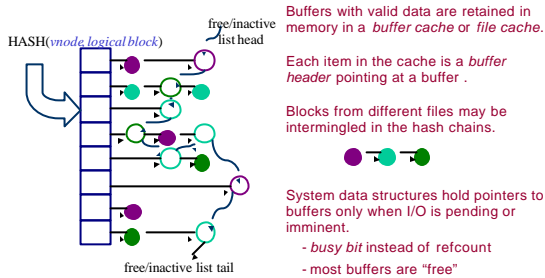
Maintain a *directory* of items in the cache: a hash table.

Hash on a unique identifier (*tag*) for the item (fully associative).

3. How to find a slot for an item fetched into the cache?

Choose an unused slot, or select an item to replace according to some policy, and *evict* it from the cache, freeing its slot.

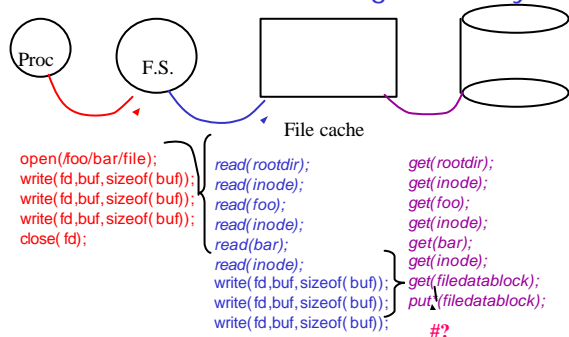
File Block Buffer Cache



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.
 - Write-back, write-through (104?)
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.
3. Knowing data gets to disk
 - Force it but you can't trust it - *fsync*

Access Patterns Along the Way



Mechanism for Cache Eviction/Replacement

- **Typical approach:** maintain an ordered *free/inactive list* of slots that are candidates for reuse.
 - *Busy* items in *active use* are not on the list.
 - E.g., some in-memory data structure holds a pointer to the item.
 - E.g., an I/O operation is in progress on the item.
 - The best candidates are slots that do not contain valid items.
 - Initially all slots are free, and they may become free again as items are destroyed (e.g., as files are removed).
 - Other slots are listed in order of *value* of the items they contain.
 - These slots contain items that are valid but *inactive*: they are held in memory only in the hope that they will be accessed again later.

Replacement Policy

- The effectiveness of a cache is determined largely by the policy for ordering slots/items on the free/inactive list. Defines the *replacement policy*
- A typical cache replacement policy is *LRU*
 - Assume *hot* items used recently are likely to be used again.
 - Move the item to the tail of the free list on every *release*.
 - The item at the front of the list is the *coldest* inactive item.
- Other alternatives:
 - FIFO: replace the *oldest* item.
 - MRU/LIFO: replace the most recently used item.

Viewing Memory as a Unified I/O Cache

A key role of the I/O system is to manage the page/block cache for performance and reliability.

tracking cache contents and managing page/block sharing
choreographing movement to/from external storage
balancing competing uses of memory

Modern systems attempt to balance memory usage between the VM system and the file cache.

Grow the file cache for file-intensive workloads.
Grow the VM page cache for memory-intensive workloads.
Support a consistent view of files across different style of access.
unified buffer cache

Prefetching

- To avoid the access *latency* of moving the data in for that first cache miss.
- **Prediction!** “Guessing” what data will be needed in the future. How?
 - It’s not for free:
Consequences of guessing wrong
Overhead – removal of useful stuff, disk bandwidth consumed

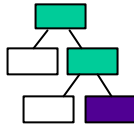
Intrafile prediction

- Sequential access suggests prefetching block $n+1$ when n is requested
- Upon seek (sequentiality is broken)
 - Stop prefetching
 - Detect a “stride” or pattern automatically
 - Depend on hints from program
 - Compiler generated “prefetch” statements
 - User supplied
 - How often is this issue relevant? Big files, nonsequential files, predictable accesses.

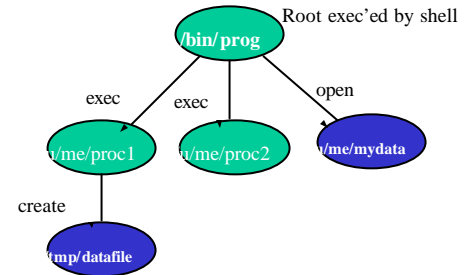
Interfile prediction

Related files – what does that mean?

- Directory nodes that are ancestors of a cached object must be cached in order to resolve pathname.
- Detection of “file working sets”
 - Trees representing program executions are constructed
- Capture semantic relationships among files in “semantic distance” measure – SEER system



Program Execution Tree



Generalize at Hoard Time

- Continuous monitoring and logging file access patterns
- To determine all of a program's dependencies
- Essentially union all saved trees (a limited number based on LRU) from previous executions of this program
- Per-application not per-user basis (global)
- Set of heuristics to fix over-generalization (if requested)
 - differentiate between data (private?) and application files (e.g. extensions, directory, time)

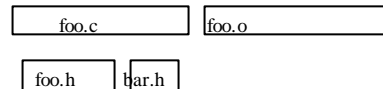
SEER's Hoarding Scheme: Semantic Distance

- **Observer** monitors user access patterns, classifying each access by type.
- **Correlator** calculates semantic distance among files
- **Clustering algorithm** assign each file to one or more **projects**
- Only entire projects are hoarded.

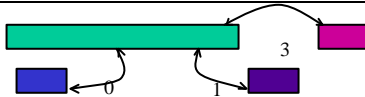
Defining Semantic Distance

- Temporal semantic distance - elapsed time between two file references
Time scale effects :-(
• Sequence-based semantic distance - number of intervening file references between 2, of interest. At what point? Open? Close?
• Lifetime semantic distance - accounts for concurrently open files - overlapping lifetimes

Example of Lifetime Distance



Distance is 0 if A not closed before B opened (Overlap)
intervening opens including itself otherwise
foo.c -> foo.h 0
foo.c -> bar.h 0
foo.c -> foo.o 3



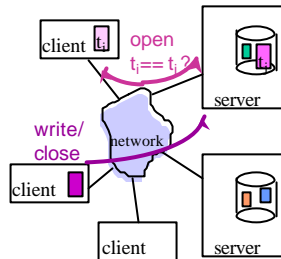
- How to turn semantic distance between two references into semantic distance between files? Summarize - geometric mean.
- Using months of data. Only store n nearest neighbors for each file and files within distance M
- *External investigators* can incorporate some extra info (e.g. heuristics used by Tait, makefile)

Real World Complications

- Meaningless clutter in the reference stream (e.g. find command)
- Shared libraries - an apparent link between unrelated files - want to hoard but not use in distance calculations and clustering
- Rare but critical files, temp files, directories
- Multi-tasking clutter
- Delete and recreate by same filename.

Sun NFS Cache Consistency

- Server is **stateless**
- Requests are self-contained.
- **Blocks** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with "true" timestamp at server on Open. (Good for an interval)
- Updates delayed but flushed before Close ends.



28

Reliability Issues

- Server crashes
 - State (if any kept) lost, reconstruct upon recovery (dialog with clients?)
 - Stateless server - all requests from clients are self-contained
- Network partitions
 - Client response - optimistic (continue to use what's in cache) or pessimistic (conservative)

NFS as a "Stateless" Service

The NFS server maintains no transient information about its clients; there is no state other than the file data on disk.

Makes failure recovery simple and efficient.

- **no record of open files**
- **no server-maintained file offsets:** read and write requests must explicitly transmit the byte offset for the operation.
- **no record of recently processed requests:** retransmitted requests may be executed more than once.
 - Requests are designed to be **idempotent** (repeatable) whenever possible (e.g., no append mode for writes, no exclusive create)

Drawbacks of a Stateless File Service

The stateless nature of NFS has compelling design advantages (simplicity), but also some key drawbacks:

- Update operations are disk-limited because they *must be committed synchronously* at the server. Otherwise, live clients can "see" data loss if server crashes.
- NFS cannot (quite) preserve local *single-copy semantics*.
 - Files may be removed while they are open on the client.
 - Idempotent operations cannot capture full semantics of Unix FS.
- Retransmissions can lead to correctness problems and can quickly saturate an overloaded server.
- Server keeps no record of blocks held by clients, so cache consistency is problematic.

The Synchronous Write Problem

Stateless NFS servers must commit each operation to stable storage before responding to the client.

- Interferes with FS optimizations, e.g., clustering, LFS, and disk write ordering (seek scheduling).
 - Damages bandwidth and scalability.
- Imposes disk access latency for each request.
 - Not so bad for a logged write; much worse for a complex operation like an FFS file write.

The synchronous update problem occurs for any storage service with reliable update (*commit*).

"Committing" a Transaction

Begin Transaction

lots of reads and writes

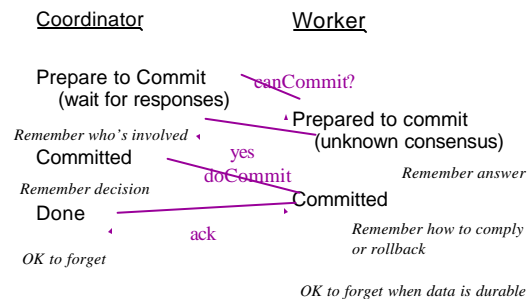
Commit or Abort Transaction

Atomic Transactions

ACID property - data is recoverable.

- Atomicity - a transaction must be all-or-nothing.
- Consistency - a transaction takes system from one consistent state to another
- Isolation - No intermediate effects are visible to others
- Durability - the effects of a committed transaction are permanent

2-Phase Commit



Explicit First-class Replication

- File name maps to set of replicas, one of which will be used to satisfy request
 - Goal: availability
- Update strategy
 - Atomic updates - all or none
 - Primary copy approach
 - Voting schemes
 - Optimistic, then detection of conflicts

Multiple Copy Schemes

- Primary Copy
 - Of all copies, there is one which is “primary” to which updates are sent. Secondary copies eventually get updates. Reads can go to any copy. How to take over when primary gone?
- Voting
 - An operation must acquire locks on some subset of copies (overlapping read or write quorums)