CSCE 313-200
Introduction to Computer Systems
Spring 2022

File System III
Dmitri Loguinov
Texas A&M University

March 31, 2022
Chapter 11: Roadmap

11.1 I/O devices
11.2 I/O function
11.3 OS design issues
11.4 I/O buffering
11.5 Disk scheduling
11.6 RAID
11.7 Disk cache
11.8-11.10 Unix, Linux, Windows

hw2 top speed:
116
116
123
140
191
RAID

- Redundant Array of Inexpensive Disks (RAID)
  - Nowadays “I” is Independent
- RAID-0 (striping)
  - Non-redundant sequential writing to all disks
  - Each stripe has some fixed block size (e.g., 64 KB)
  - R/W speed N*S for N disks
  - Any failure renders array unusable, all data lost
- RAID-1 (mirroring)
  - One spare for each disk
- RAID-1 (cont’d)
  - R/W speed N*S/2
  - Tolerates single disk failure, may survive up to N/2 failures, but may also crash with just 2
RAID

- RAID-2 and 3
  - Require synchronized disks
  - Not popular in practice
- All RAID levels 4+ compute block/stripe parity
  - Usually an XOR of all blocks
  - Failure of a disk allows recovery of block by XORing parity with remaining blocks
- RAID-4
  - Bottlenecks on parity disk (e.g., modification of blocks 2 and 6 cannot proceed in parallel)
- RAID-5
  - Parity split over all disks
  - Read speed $S^*(N-1)$
  - Tolerates failure of any single disk, crashes if 2 or more fail concurrently
RAID

• RAID-6
  - Dual parity, read speed $S \times (N-2)$
  - Tolerates failure of any 2 disks, crashes if 3 or more fail
  - On some cards, write speed 30% slower than RAID-5

• RAID-XY or X+Y
  - Several RAID-X arrays organized into a RAID-Y

• Windows also offers a spanned volume in software
  - Writes to one disk until full, then switches to the next →
Chapter 11: Roadmap

11.1 I/O devices
11.2 I/O function
11.3 OS design issues
11.4 I/O buffering
11.5 Disk scheduling
11.6 RAID
11.7 Disk cache
11.8-11.10 Unix, Linux, Windows
In caching, the main issue is achieving high hit rates.

Classical LRU (Least Recently Used)
- Evict the item that hasn’t been used the longest

In practice, doubly-linked queue/list is enough
- Most-recent items inserted at the tail, old evicted at the head

How to quickly find accessed item in the queue?
- Linear scanning is slow
**Disk Cache**

- **Idea**: maintain a hash table that stores a pointer to the item’s location in the queue/list
- **How to update the hash table during eviction?**
  - Either look up item in hash table or store a reverse pointer

no need to store items in both hash table and LRU queue
Disk Cache

- Age and frequency of usage may not be related
  - More accurate method may be LFU (Least Frequently Used)
  - Assign counter C to items, how often it has been accessed
  - Sort items by C, evict the one with the smallest counter
- Requires a min-heap ordered by access counters
• LFU complexity
  - $O(1)$ for cache hit, $\log N$ for reinsertion (existing item)
  - $O(1)$ for cache miss, $\log N$ for eviction (new item)
• Could also use a balanced binary search tree
  - Left-most child is always evicted
• Another approach: organize counters into doubly-linked list
  - Each counter has a list of nodes that tie for their value of C
  - Nodes contain pointers to actual items which are part of the hash table as before
• Constant-time access/insertion/eviction
Problem #1: LFU is biased against new items, which it may evict immediately after insertion
   - As an improvement, evict every K cache requests and use LRU within each linked list of nodes that have the same C

Problem #2: items with large counters stay virtually forever in the cache
   - Suppose an item gets 1M initial hits due to locality, but is then never needed again
   - It will not get evicted until C = 1M is the *smallest* counter in the heap/list

Goal: prevent fresh items from being immediately evicted and discount the importance of back-to-back access
Disk Cache

- Hybrid LRU-LFU methods
  - Attempt to register only long-term usage
- **New section** is similar to LRU
  - Items move to the tail on access, **counters unchanged**
  - Eviction moves from the head to the **old section**
- **Old section** is similar to LFU, sorted by counter
  - Hits increment C and move item to tail of new section

![Disk Cache Diagram]
Research suggests that the LFU (old) section is still biased against new blocks, evicts them right away.

Solution: create a middle section to build up counters
- On hits, middle-aged items increment counters and move to the tail of new section.
- When item is old, its C should reflect its long-term usage.
Chapter 12: Roadmap

12.1 Overview
12.2 File organization
12.3 Directories
12.4 Sharing
12.5 Record blocking
12.6 Secondary storage
12.7 File security
12.8-12.10 Unix, Linux, Windows
As before, a file is just a bunch of bytes

Our next task is to figure out how to organize these bytes within the file to enable ease of operation
- Mostly concerned here with data lookup and retrieval

Assume data is split into items/records
- Each record has multiple fields (e.g., name, age, SSN)

1) Pile is the most general
- Records dumped into file as they become available to the program, in no particular order, \n separator
- Different records may have different length or # of fields, typically read by humans
- e.g., Unix syslog file into which all kernel modules write
2) Sequential file (sorted or unsorted)
- One field in each record is the key, everything else is value
- Search for a given key or range

Fixed-size fields
- E.g., payroll database with all fields padded to same size

Variable-size fields
- E.g., graph (key = nodeID, value = degree + adjacency list)

If sorted by key
- If fixed-size values, binary search to find records
- If variable-size, need unambiguous record separators
- Painful to add elements as resorting the file is expensive
3) Indexed Sequential
- File structure that has the main file with data (usually huge) and a separate file containing the index for keys

Suppose the main file is Google’s word → URL mapping
- Query maps hashes of words to pages with them

Binary search on the index, find offset in main file
If index is too big to fit in RAM and binary search is inefficient, a k-level index is possible.

Assume level-1 index size $F$, read I/O block size $B$
- Binary search needs $\log_2(F/B)$ seeks
- On the other hand, k-level index needs k-1 seeks

$F = 10$ TB file, $B = 1$ MB block size $\rightarrow 23$ seeks, while multi-index above does it in $k-1 = 2$ seeks
File Organization

• 4) Indexed
  - Separate index for every possible field, allows database-like operations on fields

• Main challenge for indexed files is keeping the index updated when it doesn’t fit in RAM

• 5) Hashed file
  - Treat file contents as RAM, hash items directly to some offset

```c
uint64 N; // hash table size
// preallocate file of size N * sizeof(item)
void Hash (Item x) {
    off = HashFunction (x.key) % N;
    file.Seek (off * sizeof(Item));
    file.Write (&x, sizeof(Item));
}
```

• What to do with collisions?