Swapping Policies

Chapter 22

Previously in CS212…

- We looked at mechanisms to avoid having to physically hold all the information from our processes in memory at once by swapping content to and from the disk

- But how do we determine which pages are removed and added to memory when it’s time to swap?

Cache Management

- Since main memory can only hold a subset of all the pages in the system it is essentially a form of cache

- When selecting replacement pages, we want to avoid **cache misses** (inversely, optimizing for **cache hits**)
  - Miss – data needed is not in memory and needs to be fetched from disk
  - Hit – data needed is in memory

- We can use these conditions to determine performance of the replacement policy

Average Memory Access Time (AMAT)

- A metric to evaluate how effective a page replacement strategy is based on:
  - The cost of accessing memory (T_M)
  - The cost of accessing disk (T_D)
  - The probability of a cache miss (P_{Miss})

- The formula is: $AMAT = T_M + (P_{Miss} * T_D)$
  - Note that we always incur the memory access, while the disk access is an added penalty for a miss

Example

- Assume we have 8 pages
- The current state of physical memory is:
  - [page 0, page 2, page 3, FREE, FREE]
- The current state of swap space is:
  - [page5, page1, page 6, page 7, page 4]
- What happens when we access memory in the following order
  - 0, 3, 2, 5, 2, 3, 6, 0
  - Hit, Hit, Hit, Miss, Hit, Hit, Miss, Hit = P_{Miss} = 25% (2/8 = .25)
- If RAM access takes 100 ns and Disk takes 10 ms
  - 100ns + (.25 * 10ms) = 2.5001ms

Take Aways

- Disk access is **EXPENSIVE**
  - It dominates the AMAT even with a small miss rate
- Determining how to handle data in memory and swap space will be critical to mitigating cache miss penalties
- This is where page replacement policies come in

Optimal Replacement Policy

- Memory requests are known in advance
- Based on this information, we replace pages that will requested again the furthest in the future
  - Results in the fewest possible cache misses
- Does this sound suspiciously like it requires clairvoyant superpowers?
  - That’s because it kind of does, or at least very predictable behavior
  - A poor fit for general purpose operating systems
    - Lacks a practical implementation
- So why do we care about it at all?
  - A good benchmark for comparing policies against the ideal

Optimal Replacement Policy Example

- Assume a cache that can hold three pages
- Note the first few misses
- Since the cache will not be pre-populated with pages, we suffer from a **cold-start** or **compulsory misses** the first time we access a page
- When the cache is full and we access 3, we suffer a **capacity miss**

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Hit rate = (6 / (5 + 6)) = 54.5%

Hit rate w/o compulsory misses = (6 / (5 + 6 – 4)) = 85.7%

FIFO Replacement Policy

- Keep a queue of all pages accessed
- When we need to replace a page, remove the first page in the queue (the “front” of the queue)
- Simple, but poor performance

FIFO Replacement Policy

- The hit rate is (4 / (7 + 4)) = 36.4%
- Quite a bit worse than optimal (54.5%)
- Doesn't consider the importance of the pages in the cache

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

Random Replacement Policy

- Pick a page in the cache at random and remove it
- Doesn’t require any overhead
- Simple to implement
- Effectiveness depends on “luck”

Historical Replacement Policies

- While FIFO and Random are simple they don’t take into consideration how recently things have been accessed

- The principle of locality says we will most likely need to access certain parts of code or data structures frequently

- What if we considered this information when evicting pages from cache?
  - **Least Recently Used (LRU)** – Replace the page that hasn’t been accessed within a given period
  - **Least Frequently Used (LFU)** – Replace the page that has been used the least within a given period (keeps track of how many accesses)

LRU vs LFU Example

- Assume the following page accesses:
  - 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 4

LRU

- 1
- 1, 2
- 1, 2, 3
- 2, 3, 1 <--- repeats for each 1
- 3, 1, 2
- 1, 2, 3
- 2, 3, 4 <--- evict 1

LFU

- 1(1)
- 1(1), 2(1)
- 1(1), 2(1), 3(1)
- 1(12), 2(1), 3(1) <--- repeats for each 1
- 1(12), 2(2), 3(2)
- 1(12), 2(2), 4(1) <--- evict 3

Takeaways

- If there is no locality to the pages reference, all the replacement policies perform the same (except optimal)

- Random is *generally* better than FIFO, and can be effective for "sequential looping" workloads (where it bests all but optimal)

- Assuming a more typical workload, LRU and LFU should be closer to optimal
  - Does that matter vs. FIFO or Random?
  - The more costly the miss the more the difference matters for performance

Implementing Historical Replacement Policies

- ”Perfect” implementation requires quite a bit of overhead
- Implementation options might require:
  - Updating a data structure to ensure the optimal page is ready for eviction
  - Updating a cache record with data to indicate access counts or time accessed
    - Loop/check **all** cache records to find the best eviction choice
- Needs to be done for EVERY MEMORY ACCESS
  - Could have serious performance impact

Approximating LRU

- With a little hardware, we keep a **reference bit** to indicate that a page in cache has been used and set it to one

- The OS then used a pointer to a cache entry and check the reference bit.
  - If it's 1, reset it to 0 and check the next page (or a random page)
  - Repeat until we find a 0 entry

- The **clock algorithm** approach

- Performance is close to a strict LRU

Additional Considerations

- Dirty/Modified Bit
  - A dirty bit indicates that a cache entry has been changed from its original state
- This means if we remove it from cache, we must write the updated data back to disk
  - Extra expensive operation
- We may opt to remove a cache entry that has not been used recently/frequently that also doesn’t have its modified bit set to avoid the extra operation

Other VM Policies

- Page Selection
  - Demand paging – pull pages into memory as they are needed
  - Prefetching – try to load pages into memory that will likely be needed in the future
    - Need to be confident in this decision
- Writing pages to disk
  - Individually write pages individually as needed
  - Group pages to be written to optimize disk access

You are overburdened…

- What happens if you have enough processes running to overtax your memory resources?
- The OS may resort to swapping pages to and from disk repeatedly
  - This is known as **thrashing** and causes excessive performance degradation
- Old OSes tried to avoid this by simply not running a subset of processes until resource became available
  - **Admission control**
- New OSes may simply terminate large processes

Next Time

- We wrap up the discussion of Virtual Memory
- Examine how the concepts we have covered relate to the Linux Operating System