Section 8: Demand Paging

CS 162

October 23, 2020

Contents

1 Vocabulary

- **Demand Paging** - The process where the operating system only stores pages that are "in demand" in the main memory and stores the rest in persistent storage (disk). Accesses to pages not currently in memory page fault and the page fault handler will retrieve the request page from disk (paged in). When main memory is full, then as new pages are paged in old pages must be paged out through a process called eviction. Many cache eviction algorithms like least recently used can be applied to demand paging, the main memory is acting as the cache for pages which all start on disk.
- **Working Set** - The subset of the address space that a process uses as it executes. Generally we can say that as the cache hit rate increases, more of the working set is being added to the cache.
- **Resident Set Size** - The portion of memory occupied by a process that is held in main memory (RAM). The rest has been paged out onto disk through demand paging.
- **Thrashing** - Phenomenon that occurs when a computer's virtual memory subsystem is constantly paging (exchanging data in memory for data on disk). This can lead to significant application slowdown.
- **Inverted Page Table** - The inverted page table scheme uses a page table that contains an entry for each physical frame, not for each logical page. This ensures that the table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space. The inverted page table is a global structure – there is only one in the entire system. It stores reverse mappings for all processes. Each entry in the inverted table contains has a tag containing the task id and the virtual address for each page. These mappings are usually stored in associative memory (remember fully associative caches from 61C?). Associatively addressed memory compares input search data (tag) against a table of stored data, and returns the address of matching data. They can also use actual hash maps.
- **Policy Misses** - The miss that occurs when pages were previously in memory but were selected to be paged out because of the replacement policy.
- **Random** - Random: Pick a random page for every replacement. Unpredictable and hard to make any guarantees. TLBs are typically implemented with this policy.
- **FIFO** - First In, First Out: Selects the oldest page to be replaced. It is fair, but suboptimal because it throws out heavily used pages instead of infrequently used pages.
- **MIN** - Minimum: Replace the page that won't be used for the longest time. Provably optimal. To approximate MIN, take advantage of the fact that the past is a good predictor of the future (see **LRU**).
- **LRU** - Least Recently Used: Replace the page which hasn't been used for the longest time. An approximation of MIN. Not actually implemented in reality because it's expensive; see **Clock**
- **Belady's Anomaly** - The phenomenon in which increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern. This is common for FIFO, Second Chance, and the random page replacement algorithm. For more information, check out http://nob.cs.ucdavis.edu/classes/ecs150-2008-02/handouts/memory/mm-belady.pdf
- **Clock** - Clock Algorithm: An approximation of LRU. Main idea: replace *an* old page, not the *oldest* page. On a page fault, check the page currently pointed to by the 'clock hand. Checks a use bit which indicates whether a page has been used recently; clears it if it is set and advances the clock hand. Otherwise, if the use bit is 0, selects this candidate for replacement.

Other bits used for Clock: "modified"/"dirty" indicates whether page must be written back to disk upon pageout; "valid" indicates whether the program is allowed to reference this page; "read-only"/"writable" indicates whether the program is allowed to modify this page.

- **Nth Chance** - Nth Chance Algorithm: An approximation of LRU. A version of Clock Algorithm where each page gets N chances before being selected for replacement. The clock hand must sweep by N times without the page being used before the page is replaced. For a large N, this is a very good approximation of LRU.
- **Second Chance List** - Second-Chance List Algorithm: An approximation of LRU. Divides pages into two - an active list and a second-chance list. The active list uses a replacement policy of FIFO, while the second-chance list uses a replacement policy of LRU. Not required reading, but if you're interested in the details, this algorithm is covered in detail in this paper: https://users.soe.ucsc.edu/~sbrandt/221/Papers/Memory/levy-computer82.pdf. The version presented in lecture and for the purposes of this course includes some significant simplifications.

Question2.2 Detailed explanation

模拟过程

1. **访问 0x294 (VPN 0)**
   - TLB未命中（空），查页表：无效 → 页错误。
   - 分配物理页：PPN = VPN + 1 = 1。更新页表：VPN 0 有效，PPN=1。
   - 添加TLB条目：VPN 0 → PPN 1。
   - 结果：TLB未命中，页错误。
2. **访问 0xA76 (VPN 2)**
   - TLB未命中（仅有VPN 0），查页表：无效 → 页错误。
   - 分配物理页：PPN = 3。更新页表：VPN 2 有效，PPN=3。
   - 添加TLB条目：VPN 2 → PPN 3。
   - 结果：TLB未命中，页错误。
3. **访问 0x5A4 (VPN 1)**
   - TLB未命中（有VPN 0,2），查页表：有效，PPN=2。
   - 添加TLB条目：VPN 1 → PPN 2。
   - 结果：TLB未命中，无页错误。
4. **访问 0x923 (VPN 2)**
   - TLB命中（VPN 2 → PPN 3）。
   - 结果：TLB命中。
5. **访问 0xCFF (VPN 3)**
   - TLB未命中（有VPN 0,2,1），查页表：无效 → 页错误。
   - 分配物理页：PPN = 4。更新页表：VPN 3 有效，PPN=4。
   - 添加TLB条目：VPN 3 → PPN 4。TLB现在有4个条目。
   - 结果：TLB未命中，页错误。
6. **访问 0xA12 (VPN 2)**
   - TLB命中（VPN 2 → PPN 3）。
   - 结果：TLB命中。
7. **访问 0xF9F (VPN 3)**
   - TLB命中（VPN 3 → PPN 4）。
   - 结果：TLB命中。
8. **访问 0x392 (VPN 0)**
   - TLB命中（VPN 0 → PPN 1）。
   - 结果：TLB命中。
9. **访问 0x341 (VPN 0)**
   - TLB命中（VPN 0 → PPN 1）。
   - 结果：TLB命中。

2 Paging

2.1 Demand Paging

An up-and-coming big data startup has just hired you do help design their new memory system for a byte-addressable system. Suppose the virtual and physical memory address space is 32 bits with a 4KB page size.

Suppose you know that there will only be 4 processes running at the same time, each with a Resident Set Size (RSS) of 512MB and a working set size of 256KB. What is the minimum amount of TLB entries that your system would need to support to be able to map/cache the working set size for one process? What happens if you have more entries? What about less?

A process has a working set size of 256KB which means that the working set fits in 64 pages. This means our TLB should have 64 entries. If you have more entries, then performance will increase since the process often has changing working sets, and it should be able to store more in the TLB. If it has less, then it can't easily translate the addresses in the working set and performance will suffer.

Suppose you run some benchmarks on the system and you see that the system is utilizing over 99% of its paging disk IO capacity, but only 10% of its CPU. What is a combination of the of disk space and memory size that can cause this to occur? Assume you have TLB entries equal to the answer from the previous part.

The CPU can't run very often without having to wait for the disk, so it's very likely that the system is thrashing. There isn't enough memory for the benchmark to run without the system page faulting and having to page in new pages. Since there will be 4 processes that have a RSS of 512MB each, swapping will occur as long as the physical memory size is under 2GB. This happens regardless of the number of TLB entries and disk size. If the physical memory size is lower than the aggregate working set sizes, thrashing is likely to occur.

Out of increasing the size of the TLB, adding more disk space, and adding more memory, which one would lead to the largest performance increase and why?

We should add more memory so that we won't need to page in new pages as often.

2.2 Cached Paging

Consider a machine with a page size of 1024 bytes. There are 8KB of physical memory and 8KB of virtual memory. The TLB is a fully associative cache with space for 4 entries that is currently empty. Assume that the physical page number is always one more than the virtual page number. This is a sequence of memory address accesses for a program we are writing: 0x294, 0xA76, 0x5A4, 0x923, 0xCFF, 0xA12, 0xF9F, 0x392, 0x341.

VPN：0, 2, 1, 2, 3, 2, 3, 0, 0

Here is the current state of the page table.

Valid Bit	Physical Page Number
0	NULL
1	2
0	NULL
0	4
0	5
1	6
1	7
0	NULL

Explain what happens on a memory access.

First, we check the TLB. If the cached translation exists, we directly access the physical memory. If we get a TLB miss, then we must do a page walk in the page table to find an entry if it exists. If the entry is invalid or missing, we bring in the page, update our page table, and add the translation to our cache for future accesses.

How many TLB hits and page faults are there? What are the contents of the cache at the end of the sequence?

TLB hits: 5, Page Faults: 3

1. TLB miss (cold cache), PF 2. TLB miss (cold cache), PF 3. TLB miss (cold cache), hit 4. TLB hit, hit 5. TLB miss (cold cache), PF 6. TLB hit, hit 7. TLB hit, hit 8. TLB hit, hit 9. TLB hit hit

Valid Bit	Physical Page Number
1	1
1	2
1	3
1	4
0	5
1	6
1	7
0	NULL

Tag	Physical Page Number
0	1
2	3
1	2
3	4

3 Page Replacement Algorithms

We will use this access pattern for the following section.

Page	A	B	C	D	A	B	D	C	B	A

3.1 FIFO

How many misses will you get with FIFO? 7 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A			D				C		
2		B			A					
3			C			B				

3.2 LRU

How many misses will you get with LRU? 8 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A			D						A
2		B			A			C		
3			C			B				

3.3 MIN

How many misses will you get with MIN? 5 misses

Page	A	B	C	D	A	B	D	C	B	A
1	A									
2		B								
3			C	D				C		

3.4 FIFO vs. LRU

LRU is an approximation of MIN, which is provably optimal. Why does FIFO still do better in this case?

The LRU algorithm is based on a heuristic, trying to exploit temporal locality. It approximates MIN by assuming that the least recently used cache entry is the cache entry that will be needed at the furthest point in the future (i.e we can evict it now because 'the past is a good predictor of the future'). However, as seen in this access pattern, this is not always true.

3.5 Improving Cache Performance

If we increase the cache size, are we always guaranteed to get better cache performance?

3.5.1 FIFO

No; this phenomemon is known as Belady's anomaly. Increasing the cache size may actually worsen performance. The FIFO algorithm is a good example of this. See the **Belady's Algorithm** definition in the **Vocabulary** section for more details.

3.5.2 LRU

Yes. Given the same access pattern, the contents of a cache of size S is always a subset of the contents of a cache with size S + 1. This holds true for any stack algorithm; LRU is one.

3.5.3 MIN

Yes. Given the same access pattern, the contents of a cache of size S is always a subset of the contents of a cache with size S + 1. This holds true for any stack algorithm; MIN is one.

4 Clock Algorithm

4.1 Clock Page Table Entry

Suppose that we have a 32-bit virtual address split as follows:

10 Bits Table ID	10 Bits Page ID	12 Bits Offset

Assume that the physical address is 32-bit as well. Show the format of a page table entry (PTE) complete with bits required to support the clock algorithm.

20 Bits PPN	8 Bits Other	1 Bit Dirty	1 Bit Use	1 Bit Writable	1 Bit Valid

4.2 Clock Algorithm Step-through

For this problem, assume that physical memory can hold at most four pages. What pages remain in memory at the end of the following sequence of page table operations and what are the use bits set to for each of these pages?

Page	A	B	C	A	C	D	B	D	A	E	F

E: 1, F: 1, C: 0, D: 0

Recall that the clock hand only advances on page faults. No page replacement occurs until $t = 10$, when all pages are full. At $t = 10$, all pages have the use bit set. The clock hand does a full sweep, setting all use bits to 0, and selects page 1 (currently holding A) to be paged out. The clock hand advances and now points to page 2 (currently holding B). At $t = 11$, we check page 2's use bit, and since it is not set, select page 2 to be paged out. F is brought in to page 2. The clock hand advances and now points to page 3. We reach the end of the input and end.

Note: The table shows the clock hand position before page faults occur.

Page	A	B	C	A	C	D	B	D	A	E	F
1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	E: 1	E: 1
2		B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 0	F: 1
3			C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 0	C: 0
4						D: 1	D: 1	D: 1	D: 1	D: 0	D: 0
Clock	1	2	3	4	4	4	1	1	1	1	2

模拟步骤详解

1. **访问 A**：缺页，加载到帧0，使用位置1，指针移至帧1。
   内存：[A(1), -, -, -]
2. **访问 B**：缺页，加载到帧1，使用位置1，指针移至帧2。
   内存：[A(1), B(1), -, -]
3. **访问 C**：缺页，加载到帧2，使用位置1，指针移至帧3。
   内存：[A(1), B(1), C(1), -]
4. **访问 A**：命中，使用位置1（不变），指针仍在帧3。
   内存：[A(1), B(1), C(1), -]
5. **访问 C**：命中，使用位置1，指针仍在帧3。
   内存：[A(1), B(1), C(1), -]
6. **访问 D**：缺页，加载到帧3，使用位置1，指针移至帧0。
   内存：[A(1), B(1), C(1), D(1)]
7. **访问 B**：命中，使用位置1，指针仍在帧0。
   内存：[A(1), B(1), C(1), D(1)]
8. **访问 D**：命中，使用位置1，指针仍在帧0。
   内存：[A(1), B(1), C(1), D(1)]
9. **访问 A**：命中，使用位置1，指针仍在帧0。
   内存：[A(1), B(1), C(1), D(1)]
10. **访问 E**：缺页，指针扫描：
    - 帧0 (A)：使用位1→置0，指针移至帧1。
    - 帧1 (B)：使用位1→置0，指针移至帧2。
    - 帧2 (C)：使用位1→置0，指针移至帧3。
    - 帧3 (D)：使用位1→置0，指针移至帧0。
    - 帧0 (A)：使用位0，替换为E，使用位置1，指针移至帧1。
      内存：[E(1), B(0), C(0), D(0)]
11. **访问 F**：缺页，指针扫描：
    - 帧1 (B)：使用位0，替换为F，使用位置1，指针移至帧2。
      内存：[E(1), F(1), C(0), D(0)]