Section 9: Paging

CS 162

July 22, 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 frames 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

2 Address Translation (7/20)

2.1 Page Allocation

Suppose that you have a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

How large is each page? Assume memory is byte addressed.

32 bytes

Suppose that a program has the following memory allocation and page table.

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

What will the page table look like if the program runs the following function? Page out the least recently used page of memory if a page needs to be allocated when physical memory is full. Assume that the stack will never exceed one page of memory.

```
#define PAGE_SIZE 1024;

void helper(void) {
    char *args[5];
    int i;
    for (i = 0; i < 5; i++) {
        // Assume malloc allocates an entire page every time
        args[i] = (char*) malloc(PAGE_SIZE);
    }
    printf("%s", args[0]);
}
```

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
Heap	100	00
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	00
Heap	101	11
N/A	110	NULL
Stack	111	01

What happens when the system runs out of physical memory? What if the program tries to access an address that isn't in physical memory? Describe what happens in the user program, the operating system, and the hardware in these situations.

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in physical memory. The computer hardware traps to the kernel and current state information is saved. The system will then find out which virtual page was needed. If the virtual address is valid, the system checks for a free page. If there are no free pages in memory, a page replacement policy is applied to remove a page. The page is brought in from disk, the faulting instruction is backed up to the state it had when it began state information is restored, and execution is resumed.

2.2 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

Since each PTE is 4 bytes and each page contains 8KB, then a one-page page table would point to 2048 or 2^{11} pages, addressing a total of $2^{11} * 2^{13} = 2^{24}$ bytes.

Depth 1 = 2^{24} bytes Depth 2 = 2^{35} bytes Depth 3 = 2^{46} bytes So in total, 3 levels of page tables are required.

List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, plus several bits – read, write, execute, and valid. This information can all fit into 4 bytes, since if physical memory is 2^{33} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation.

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

Best-case scenario: 2 memory lookups. once in TLB, once for actual memory operation. Worst-case scenario: 5 memory lookups. once in TLB + 3 page table lookups in addition to the actual memory operation.

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:

- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61ns?

```
(10+50)*x+(1-x)*(50+10+50) = 61

solve for x gives x = .98 = 98% hit rate
```

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

```
With a TLB with a hit rate of 0.5:
x = 0.5
avg_time = (10+50)*x+(1-x)*(50+10+50)
avg_time = 85

Without a TLB:
time = 50 + 50
time = 100
```

3 Paging

3.1 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.

3.2 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.

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

3.3 Inverted Page Tables

3.3.1 Inverted Page Table

Why IPTs? Consider the following case:
- 64-bit virtual address space
- 4 KB page size
- 512 MB physical memory

How much space (memory) needed for a single level page table? Hint: how many entries are there? 1 per virtual page. What is the size of a page table entry? access control bits + physical page #.

```
One entry per virtual page

 - 2^64 addressable bytes / 2^12 bytes per page = 2^52 page table entries

 Page table entry size

 - 512 MB physical memory = 2^29 bytes
- 2^29 bytes of memory/2^12 bytes per page = 2^17 physical pages
- 17 bits needed for physical page number
{ Page table entry = ~4 bytes
- 17 bit physical page number = ~3 bytes
- Access control bits = ~1 byte

Page table size = page table entry size * # total entries

{ 2^52 page table entries * 2^2 bytes = 2^54 bytes (16 petabytes)

i.e. A WHOLE LOT OF MEMORY
```

How about multi level page tables? Do they serve us any better here?

What is the number of levels needed to ensure that any page table requires only a single page (4 KB)?

```
{ Assume page table entry is 4 bytes
{ 4 KB page / 4 bytes per page table entry =
1024 entries
{ 10 bits of address space needed
{ ceiling(52/10) = 6 levels needed

7 memory accesses to do something? SLOW!!!
```

3.3.2 Linear Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
{ add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB

Iterate through all entries.
For each entry in the inverted page table,
compare process ID and virtual page
number in entry to the requested process
ID and virtual page number
Extremely slow. must iterate through 2^17 entries of the hash table
worst-case scenario.
```

3.3.3 Hashed Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
{ add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB

Linear inverted page tables require too many
memory accesses.
- Keep another level before actual inverted page
table (hash anchor table)
{ Contains a mapping of process ID and virtual page
number to page table entries
- Use separate chaining for collisions
- Lookup in hash anchor table for page table entry
{ Compare process ID and virtual page number
- if match, then found
- if not match, check the next pointer for another page table
entry and check again

So, with a good hashing scheme and a hashmap proportional to
the size of physical memory, O(1) time. Very efficient!
```

4 Page Replacement Algorithms

We will use this access pattern for the following section.

Page	A	B	C	D	A	B	D	C	B	A

4.1 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				

4.2 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				

4.3 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		

4.4 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.

4.5 Improving Cache Performance

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

4.5.1 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.

4.5.2 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.

4.5.3 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.