Section 7: Virtual Memory, Caches

CS 162

October 16, 2020

Contents

1 Vocabulary **2**

2 Paging and Address Translation **4**
2.1 Conceptual Questions 4
2.2 Page Allocation 5
2.3 Address Translation 7

3 Caching **9**
3.1 Direct Mapped vs. Fully Associative Cache 9
3.2 Two-way Set Associative Cache 9
3.3 Average Read Time 10
3.4 Average Read Time with TLB 10

1 Vocabulary

- **Virtual Memory** - Virtual Memory is a memory management technique in which every process operates in its own address space, under the assumption that it has the entire address space to itself. A virtual address requires translation into a physical address to actually access the system's memory.
- **Memory Management Unit** - The memory management unit (MMU) is responsible for translating a process' virtual addresses into the corresponding physical address for accessing physical memory. It does all the calculation associating with mapping virtual address to physical addresses, and then populates the address translation structures.
- **Address Translation Structures** - There are two kinds you learned about in lecture: segmentation and page tables. Segments are linearly addressed chunks of memory that typically contain logically-related information, such as program code, data, stack of a single process. They are of the form (s,i) where memory addresses must be within an offset of i from base segment s. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware or more specifically to the RAM.
- **Translation Lookaside Buffer (TLB)** - A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them mutliple times through page table accesses.
- **Cache** - A repository for copies that can be accessed more quickly than the original. Caches are good when the frequent case is frequent *enough* and the infrequent case is not too expensive. Caching ensures locality in two ways: temporal (time), keeping recently accessed data items 'saved', and spatial (space), since we often bring in contiguous blocks of data to the cache upon a cache miss.
- **AMAT** - Average Memory Access Time: a key measure for cache performance. The formula is (Hit Rate x Hit Time) + (Miss Rate x Miss Time) where Hit Rate + Miss Rate = 1.
- **Compulsory Miss** - The miss that occurs on the first reference to a block. This also happens with a 'cold' cache or a process migration. There's essentially nothing that you can do about this type of miss, but over the course of time, compulsory misses become insignificant compared to all the other memory accesses that occur.
- **Capacity Miss** - The miss that occurs when the cache can't contain all the blocks that the program accesses. One solution for capacity misses is increasing the cache size.
- **Conflict Miss** - The miss that occurs when multiple memory locations are mapped to the same cache location (i.e a collision occurs). In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache. These technically do not exist in virtual memory, since we use fully-associative caches.
- **Coherence Miss** - Coherence misses are caused by external processors or I/O devices that update what's in memory (i.e invalidates the previously cached data).
- **Tag** - Bits used to identify the block - should match the block's address. If no candidates match, cache reports a cache miss.

- **Index** - Bits used to find where to look up candidates in the cache. We must make sure the tag matches before reporting a cache hit.
- **Offset** - Bits used for data selection (the byte offset in the block). These are generally the lowest-order bits.
- **Direct Mapped Cache** - For a 2^N byte cache, the uppermost (32 - N) bits are the cache tag; the lowest M bits are the byte select (offset) bits where the block size is 2^M. In a direct mapped cache, there is only one entry in the cache that could possibly have a matching block.
- **N-way Set Associative Cache** - N directly mapped caches operate in parallel. The index is used to select a set from the cache, then N tags are checked against the input tag in parallel. Essentially, within each set there are N candidate cache blocks to be checked. The number of sets is X / N where X is the number of blocks held in the cache.
- **Fully Associative Cache** - N-way set associative cache, where N is the number of blocks held in the cache. Any entry can hold any block. An index is no longer needed. We compare cache tags from all cache entries against the input tag in parallel.

2 Paging and Address Translation

2.1 Conceptual Questions

If the physical memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

Increases by 1 bit. Assuming the page size remains the same, there are now twice as many physical pages, so the physical page number needs to expand by 1 bit.

If the physical memory size (in bytes) is doubled, how does the number of entries in the page table change?

No change. The number of entries in the page table is determined by the size of the virtual address and the size of a page – it's not affected by the size of physical memory.

If the virtual memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

No change. The number of bits in a page table entry is determined by the number of control bits (usually 2: dirty and resident) and the number of physical pages – the size of each entry is not affected by the size of virtual memory.

If the virtual memory size (in bytes) is doubled, how does the number of entries in the page map change?

The number of entries doubles. Assuming the page size remains the same, there are now twice as many virtual pages and so there needs to be twice as many entries in the page map.

If the page size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

Each entry is one bit smaller. Doubling the page size while maintaining the size of physical memory means there are half as many physical pages as before. So the size of the physical page number field decreases by one bit.

If the page size (in bytes) is doubled, how does the number of entries in the page table change?

There are half as many entries. Doubling the page size while maintaining the size of virtual memory means there are half as many virtual pages as before. So the number of page table entries is also cut in half.

The following table shows the first 8 entries in the page table. Recall that the valid bit is 1 if the page is resident in physical memory and 0 if the page is on disk or hasn't been allocated.

Valid Bit	Physical Page
0	7
1	9
0	3
1	2
1	5
0	5
0	4
1	1

If there are 1024 bytes per page, what is the physical address corresponding to the hexadecimal virtual address 0xF74?

The virtual page number is 3 with a page offset of 0x374. Looking up page table entry for virtual page 3, we see that the page is resident in memory (valid bit = 1) and lives in physical page 2. So the corresponding physical address is `(2<<10)+0x374 = 0xB74`

2.2 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.

2.3 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: 1 memory lookup. Hit in TLB, once for actual memory operation.
Worst-case scenario: 4 memory lookups. Miss 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 Caching

3.1 Direct Mapped vs. Fully Associative Cache

A big data startup has just hired you to 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.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that uses an LRU replacement policy. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache does. What's a possible access pattern that could cause this to happen?

Let's say each cache held X amount of blocks. An access pattern would be to repeatedly iterate over $X + 1$ consecutive blocks, which would cause everything in the fully associated cache to miss every time.

3.2 Two-way Set Associative Cache

Instead, your boss tells you to build a 8KB 2-way set associative cache with 64 byte cache blocks.

How would you split a given virtual address into its tag, index, and offset numbers?

The number of offset bits is determined by the size of the cache blocks. Thus, the offset will take 6 bits, since $2^6 = 64$.

Recall that for a set associative cache, each 'set' holds N 'candidiate' blocks. Thus, to find the index we must find how many sets there are. We divide by N first to get total bytes per bank, then find how many blocks fit in each bank to get the number of blocks. Since it's two way set associative, the cache is split into two 4KB banks. Each bank can store 64 blocks, since total bytes per bank / block size = $2^{12}/2^6 = 2^6$, so there will be 6 index bits. This matches what we expect, which is that the whole cache can hold 128 blocks.

The remaining bits will be used as the tag (32-6-6 = 20).

It will look like this:

20 Bits	6 Bits	6 Bits
Tag	Index	Offset

3.3 Average Read Time

You finish building the cache, and you want to show your boss that there was a significant improvement in average read time.

Suppose your system uses a two level page table to translate virtual addresses, and your system uses the cache for the translation tables and data. Each memory access takes 50ns, the cache lookup time is 5ns, and your cache hit rate is 90%. What is the average time to read a location from memory?

Recall that page tables are held in memory as well. Since the page table has two levels, there are three reads for each access: read from first-level page table, read from second-level page table, and finally read from the physical page. Because the system also uses the cache for the translation tables, accessing a page table costs the same as going to memory. Thus, to get our final answer we should calculate the average access time, then multiply that by 3 to get the average read time.

The average access time is: $0.9 * 5 + 0.1 * (5 + 50) = 10$ns. The miss time includes the cache lookup time, as well as the time for a memory access. Since there are three accesses, we multiply this by 3 to get an average read time of 30ns.

3.4 Average Read Time with TLB

In addition to the cache, you add a TLB to aid you in memory accesses, with an access time of 10ns. Assuming the TLB hit rate is 95%, what is the average read time for a memory operation? You should use the answer from the previous question for your calculations.

If the TLB hits, we only need to read one page - the physical page mapped to in the TLB (we don't consider TLB accesses as physical memory accesses), with an access time of 10ns for a single read. Otherwise, we need to read the page table again; as in the previous part, the average read time for three accesses is 30ns. Thus, the average read time is $0.95 * (10 + 10) + 0.05 * (10 + 30) = 21$ns