Discussion 6: Paging, Caches

October 23, 2024

Contents

1 Paging

1.1 Concept Check

1. True or False: The page table base pointer contains the virtual address of the page table.

   False. It must be a physical address. If it were a virtual address, it would have to go through address translation, but then you run into the paradox of having to know where the page table is located in physical memory!

2. True or False: If a user process accesses an address that generates a page fault, the OS will terminate this process for access violation.

   False. A page fault can mean many things: this memory can be just unmapped or previously mapped but not resident in main memory at the time of access. The OS can handle such cases appropriately and map a new page in the user address space.

3. What are some advantages of having a larger page size? What about its disadvantages?

   Having a larger page size would decrease the number of page table entries and thus the size of the page table. It could also potentially improve TLB performance as a result. Its main disadvantage is internal fragmentation.

1.2 Little Page Translation

The following table shows the 4 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 Number
0	7
1	9
0	3
1	2

If there are 1024 bytes per page, what is the physical address corresponding to the 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 (i.e. valid bit = 1) and lives in physical page 2, so the corresponding physical address is `(2<<10)+0x374 = 0xB74`

1.3 Demand Pages

An up-and-coming 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.

1. 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 if you have fewer entries?

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.

2. 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 working set of 256 KB each, swapping will occur as long as the physical memory size is under 1 MB. 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.

3. 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 Caches

2.1 Translation Trivia

1. 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.

   Level 1 = 2^{24} bytes
   Level 2 = 2^{35} bytes
   Level 3 = 2^{46} bytes

   So in total, 3 levels of page tables are required.

2. 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 (e.g. 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.

3. 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.

4. 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.

5. 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

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

There are 5 TLB hits and 3 page faults.

Address	TLB	Memory
`0x294`	Miss	Page Fault
`0xA76`	Miss	Page Fault
`0x5A4`	Miss	Valid
`0x923`	Hit	Valid
`0xCFF`	Miss	Page Fault
`0xA12`	Hit	Valid
`0xF9F`	Hit	Valid
`0x392`	Hit	Valid
`0x341`	Hit	Valid

The page table looks like

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

The TLB looks like

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

2.2 AMAT Calculations

Assume you are building a memory scheme with single level page tables. Each main memory access takes 50 ns and each TLB access takes 10 ns.

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

$$(10 + 50) \cdot x + (1 - x) \cdot (10 + 50 + 50) = 61$$

Solving for x gives a necessary TLB hit rate of 98%.

2. Assuming a TLB hit rate of 50%, how does the AMAT of this scenario compare to no TLB?

With a TLB with a hit rate of 50%, the AMAT is

$$(10 + 50) \cdot 0.5 + (1 - 0.5) \cdot (10 + 50 + 50) = 85$$

Without a TLB, the AMAT is simply the cost of a page table look up and the memory access.

$$50 + 50 = 100$$

3. To improve your system, you add a two level paging scheme and a cache. The cache has a 90% hit rate with a lookup time of 20 ns. Additionally, the TLB hit rate is now improved to 95%. What is the average time to read a location from memory?

AMAT is $0.9 \cdot 20 + 0.1 \cdot (20 + 50) = 25$ ns. Page tables are held in memory as well, so each page table lookup will incur a memory access. There are three total memory accesses (two for page table lookup + 1 for actual data lookup), so the average time to read a location from memory is $25 \cdot 3 = 75$ ns.

However, remember that we have a TLB as well. If the TLB hits, we only have one memory access (i.e. the actual data lookup) since TLB accesses are not considered as memory accesses. Otherwise, we have to traverse through the page tables. As a result, the average time to read a location from memory is $0.95 \cdot (10 + 25) + 0.05 \cdot (10 + 75) = 37.5$ ns.

2.3 Associativity Analysis

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.

1. First, you create a direct mapped cache and 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 associative cache to miss every time.

2. 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 each cache block, giving $\log_2 64 = 6$.

Recall that for a set associative cache, each set holds N candidate 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), giving a virtual address breakdown of

Tag	Index	Offset
20	6	6

2.4 Replacement Roulette

Assume your program has the following memory access pattern.

A	B	C	D	A	B	D	C	B	A

1. How many misses will you get with FIFO?

   7.

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

2. How many misses will you get with LRU?

   8.

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

3. How many misses will you get with MIN?

   5.

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

4. If we increase the cache size, are we always guaranteed to get better cache performance? Explain for FIFO, LRU, and MIN.

   FIFO suffers from Belady's anomaly, so there isn't a guarantee. On the other hand, LRU and MIN are stack algorithms, meaning the contents of a cache with size S is always a subset of the contents of a cache with size $S + 1$.

2.5 On the Clock

1. Suppose that we have a 10-10-12 virtual address split using a two-level paging scheme. Assume that the physical address is 32-bit as well and PTE is 4 bytes. Show the format of a PTE complete with bits required to support the clock algorithm.

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

2. Assume that physical memory can hold at most four pages. The program you run has the following memory access pattern.

t	1	2	3	4	5	6	7	8	9	10	11	12
Page	A	B	C	A	C	D	B	D	A	E	B	F

   What pages remain in memory at the end of the following sequence of page table operations? What are the use bits set to for each of these pages?

E: 1, B: 0, F: 1, 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. In this scenario, the clock algorithm does not do a great job of evicting the least recently used page since A was just used in $t = 9$.

At $t = 12$, we check page 3's use bit, and since it is not set, select page 3 to be paged out. F is brought in to page 3. The clock hand advances and now points to page 4. In this scenario, the clock algorithm performed well and did not evict B as it was recently accessed.

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

t	1	2	3	4	5	6	7	8	9	10	11	12
Page	A	B	C	A	C	D	B	D	A	E	B	F
1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	E: 1	E: 1	E: 1
2		B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 0	B: 1	B: 0
3			C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 0	C: 0	F: 1
4						D: 1	D: 1	D: 1	D: 1	D: 0	D: 0	D: 0
Clock	1	2	3	4	4	4	1	1	1	1	2	2