Section 12: Cache, Clock Algorithm, and Demand Paging

CS162

August 5, 2019

Contents

1 Vocabulary

never seen before
→ never knew to bring in

3Cs

- **Compulsory Miss** The miss that occurs on the first reference to a block. 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** This miss occurs when the cache can't contain all the blocks that the program accesses. Usually the solution to capacity misses is to increase the cache size.

- **Conflict Miss** Conflict misses occur when multiple memory locations are mapped to the same cache location. In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache.

extra C

- **Coherence Miss** Coherence misses are caused by external processors or I/O devices that updates what's in memory.

- **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.

- **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 phiscial 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.

- **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.

2 Problems

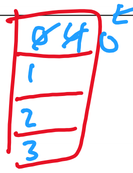
2.1 Caching

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.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that replaces the least recently used pages when the cache is full. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache. What's a possible access pattern that could cause this to happen?

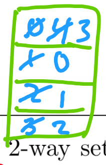
← 0 & 4 swap each other out → misses only on 0, 4

DM Cache



access 5 blocks repeatedly : 0 1 2 3 4

FA cache



2 ← LRU gets evicted before use

→ all missos

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?

$2^{13} \Rightarrow$ index + offset = 13 bits

$2^{6} \Rightarrow$ offset bits = 6

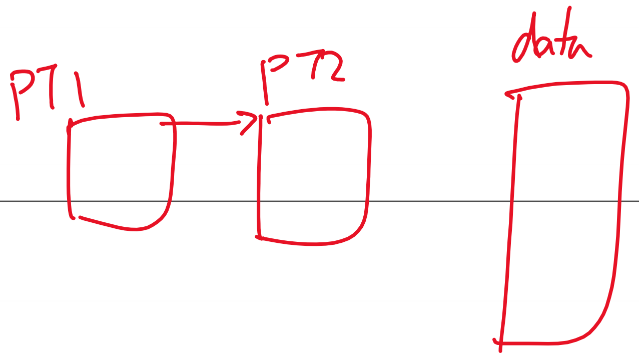
(13 - 6 = 7 index bits) / 2 = 6 index bits

2 way SA

T	I	O
⇒ 20	6	6

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?

AMAT = HT + MR · MP = 5 + 0.1 · 50 = 10ns



3 lookups

⇒ 30ns

2.2 Clock Algorithm

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

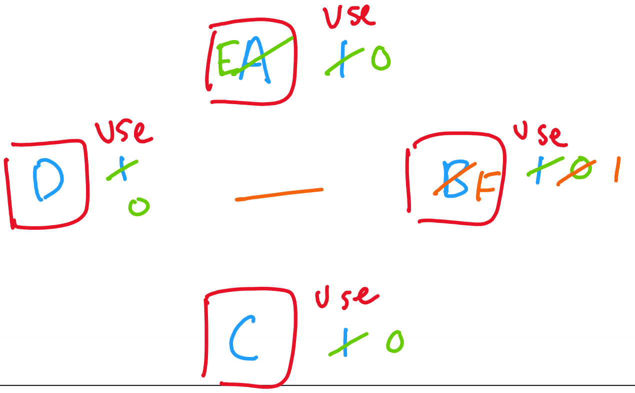
10 Bits Table ID	10 Bits Page ID	12 Bits Offset

Show the format of a PTE complete with bits required to support the clock algorithm. ← 32 bit phys addr

	PPN	Dirty	Use	Write	read	other
bits	20	1	1	1	1	8

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 is accessed
- Page B is accessed
- Page C is accessed
- Page A is accessed
- Page C is accessed
- Page D is accessed
- Page B is accessed
- Page D is accessed
- Page A is accessed
- Page E is accessed ●
- Page F is accessed ●



	E	F	C	D
use	1	1	0	0

2.3 Demand Paging

Your boss has been so impressed with your work designing the caching that he has asked for your advice on designing a TLB to use for this system. 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. Assuming the same system as the previous problem (32 bit virtual and physical address space, 4KB page size), 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?

$\frac{256KB}{4KB/page} = 64$ pages ⇒ 64 translations ⇒ 64 entries

more → good (maybe slows down lookup slightly)

very good for multiple processes

less → bad, can't even store working set for 1 process

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.

Thrashing - CPU idling while paging

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?

More memory > TLB > disk

going to disk is more expensive than going to mem

storage space doesn't change much

RAM caches disk
TLB caches RAM

2.4 Virtual Memory

`vmstat` is a Linux performance debugging tool that provides information about **virtual memory** on your system. When you run it, the output looks like this:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 174184 1007372  96316   49  642  3095   678  123  128  0  1 99  0  0
 0  0      0 174240 1007372  96316    0    0     0     0   48   88  0  0 100  0  0
 0  0      0 174240 1007372  96316    0    0     0     0   33   75  0  0 100  0  0
 0  0      0 174240 1007372  96316    0    0     0     0   32   73  0  0 100  0  0
```

The `1` means "recompute the stats every 1 second and print them out". The first line contains the average values since boot time, while the second line contains the averages of the last second (current averages). Here's a reference for what each one of the columns means.

```
Procs
    r: The number of runnable processes (running or waiting for run time).
    b: The number of processes in uninterruptible sleep.

Memory
    swpd: the amount of virtual memory used.
    free: the amount of idle memory.
    buff: the amount of memory used as buffers.
    cache: the amount of memory used as cache.
    inact: the amount of inactive memory.  (-a option)
    active: the amount of active memory.  (-a option)

Swap
    si: Amount of memory swapped in from disk (/s).
    so: Amount of memory swapped to disk (/s).

IO
    bi: Blocks received from a block device (blocks/s).
    bo: Blocks sent to a block device (blocks/s).

System
    in: The number of interrupts per second, including the clock.
    cs: The number of context switches per second.

CPU
    These are percentages of total CPU time.
    us: Time spent running non-kernel code.  (user time, including nice time)
    sy: Time spent running kernel code.  (system time)
    id: Time spent idle.  Prior to Linux 2.5.41, this includes IO-wait time.
    wa: Time spent waiting for IO.  Prior to Linux 2.5.41, included in idle.
    st: Time stolen from a virtual machine.  Prior to Linux 2.6.11, unknown.
```

Take a look at these 3 programs (A, B, C).

```
char *buffer[4 * (1L << 20)];
int A(int in) {
  // "in" is a file descriptor for a file on disk
  while (read(in, buffer, sizeof(buffer)) > 0);
}

int B() {
    size_t size = 5 * (1L << 30);
    int *x = malloc(size);
    memset(x, 0xCC, size);
}

sem_t sema;
pthread_t thread;
void *foo() { while (1) sem_wait(&sema); }
int C() {
    pthread_create(&thread, NULL, foo, NULL);
    while (1) sem_post(&sema);
}
```

I ran these 3 programs one at a time, but in a random order. What order did I run them in? Can you tell where (in the vmstat output) one program stopped and another started? Explain.

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0 684688  25216 1822136  60860   75  748  3645   779  146  296  1  1 98  0  0
 1  0 684688  25268 1822136  60868    0    0     0     0 18150 735898  6 44 51  0  0
 1  0 684688  25268 1822136  60868    0    0     0     0 61864 1270088  6 77 17  0  0
 1  0 684688  25268 1822136  60868    0    0     0     0 59497 1102825  8 71 21  0  0
 1  0 684688  25268 1822136  60868    0    0     0     0 94619 766431 11 79 10  0  0
 0  0 684688  25612 1822136  60868    0    0     0     0 13605 237430  2 13 85  0  0
 0  0 684688  25612 1822136  60868    0    0     0     0   61  115  0  1 100  0  0
 3  0 694520  18544   3212  45040   64 11036   264 11144 2647 2339  5 51 43  0  0
 4  1 1285828  20560    128    580   88 592440 14248 592440 18289 2171  3 58 36  4  0
 3  0 1866176  21492    128   2132    0 578404  8972 578404 47646 1691  2 70 28  1  0
 3  0 2350636  17820    136   2640    0 487732 11708 487788 17404 1881  1 58 39  1  0
 2  0 2771016  22168    544   4360 2072 417272 15372 417272 17460 2192  2 57 39  3  0
 0  0 697036 1922160    560   9712 1516 418224 16508 418228 47747 2616  0 64 30  6  0
 0  0 697032 1921696    564  10096   28    0   288     0   77  148  0  0 100  0  0
 1  0 696980 878128 1037720  11272  412    0 1038840     0 11128 14854  1 25 54 21  0
 1  0 696980  21732 1895476   9348    0    0 1286460     0 13610 18224  0 31 46 22  0
 0  2 696980  20992 1896496   9072    0    0 1297536    20 13745 19164  0 36 43 21  1
 1  1 696980  20228 1897784   8648    0    0 1283324    32 13659 18931  0 36 41 23  0
 1  1 696960  21048 1897404   8716   48    0 1215152     0 12601 17672  0 34 45 21  0
 0  0 696952  23048 1899112   9004    8    0 470112     0 5100 7073  0 13 81  6  0
 0  0 696952  23048 1899112   9004    0    0     0     0   45   89  0  0 100  0  0
```

If you have extra available physical memory, Linux will use it to cache files on disk for performance benefits. This disk cache may also include parts of the swapfile. Why would caching the swapfile be better than paging-in those pages immediately?

If I remove the line "`memset(x, 0xCC, size);`" from program B, I notice that the **vmstat** output does not have a spike in swap (si and so) nor in io (bi and bo). My system doesn't have enough physical memory for a 5GB array. Yet, the array is not swapped out to disk. Where does the array go? Why did the `memset` make a difference?

Program B has a 5GB array, but the whole thing just contains `0xCCCCCCCC`. Based on this observation, can you think of a way to reduce program B's memory footprint without changing any of program B's code? (What can the kernel do to save memory?)

2.5 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.

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.

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.

```
#define PAGE_SIZE 1024; // replace with actual page size

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]);
}
```

2.6 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?

last ws: 3levels

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

lastws: PPN, permission, dirty ...

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

3 lookups, 1 actual mem op ⇒ 4

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

1 TLB hit, 1 actual mem ⇒ 2

1 TLB miss, 3 lookups, 1 mem op ⇒ 5

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.

$$61ns = (\underline{10ns} + \underline{50ns}) + MR(\underline{50ns})$$

$$1ns = MR(50ns) \quad MR = 1/50$$

$$\boxed{HR = 98\%}$$

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

$$\text{AMAT w/ TLB} = (10ns + 50ns) + 0.5(50ns)$$

$$= 85ns$$

$$\text{AMAT w/o TLB} = \overset{\text{lookup}}{50ns} + \overset{\text{data access}}{50ns}$$

$$= 100ns$$

not AMAT formula since no Hit rates!

2.7 Inverted Page Tables

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 #.

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)?

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

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