Section 8: Cache, Clock Algorithm, and Demand Paging

CS162

March 15, 2019

Contents

1 Vocabulary

- **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.
- **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.

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?

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.

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?

Since it's two way set associative, the cache is split into two 4KB banks. The offset will take 6 bits, since $2^6 = 64$. Each bank can store 64 blocks, since $2^{12}/2^6 = 2^6$, so there will be 6 index bits. Then the rest of the bits will be for the tag. It will look like this:

20 Bits Tag	6 Bits Index	6 Bits Offset

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?

Since the page table has two levels, there are three reads for each access. For each access, the average access time is: $0.9 * 5 + 0.1 * (5 + 50) = 10$ns. Since there are 3 accesses, we multiply this by 3 to get an average read time of 30ns.

2.2 Clock Algorithm

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

10 Bits	10 Bits	12 Bits
Table ID	Page ID	Offset

Show the format of a PTE complete with bits required to support the clock algorithm.

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

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: 1, F: 1, C: 0, D: 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?

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.

2.4 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
```

I ran C (rows 2-6), B (rows 8-13), then A (rows 15-20). Program C has a lot of context switches (cs), but no swap or IO activity. Both program A and B have a lot of disk I/O, but program A is read-only IO (bi) and program B is the only one that should be swapping to disk (so).

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 the system's memory needs grow, the disk cache must shrink. If we avoid paging-in those cached parts of the swapfile, we can immediately evict the swapfile from memory without needing to modify any page tables.

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?

The memory returned by malloc is sparsely allocated. It won't actually be allocated until I begin using it. The memset will force the kernel to materialize those pages, because I write to them.

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?)

The kernel could compress pages in memory. It's usually faster to uncompress memory than it is to page-in from disk. If the entire array is the same byte repeated over and over, then the kernel could achieve very good compression ratios on the array. The kernel must mark infrequently-used pages as invalid, so that it has a chance to uncompress those pages inside the page fault exception handler.

2.5 Wait and Exit

This problem is designed to help you with implementing wait and exit in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

2.5.1 Thinking about what you need to do

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children:

- Child's exit status, so that "wait" can return it.
- Child's thread id, for "wait" to compare against its argument.
- A way for the parent to block until the child dies (usually a semaphore).
- A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

2.5.2 Code

Data structures to add to thread.h for waiting logic:

```
Pseudocode:

process_wait (tid_t child_tid) {
    iterate through list of child processes
    if child process tid matches tid parameter, call sema_down
    on the semaphore associated with that child process (when
    child process exits, it will sema_up on that same semaphore,
    waking up the parent process)
    --- after waking ---
    set exit code to terminated child process's exit code
    decrease ref_cnt of child //why? need to free memory, "zombie processes"
    return exit_code
}

process_exit (void) {
    sema_up on semaphore that parent might be sleeping on
    remove all child processes from child process list
    decrement ref_cnt
}


Code:

struct wait_status
    {
      struct list_elem elem;      /* 'children' list element. */
      struct lock lock;           /* Protects ref_cnt. */
      int ref_cnt;                /* 2=child and parent both alive,
             1=either child or parent alive,
             0=child and parent both dead. */
      tid_t tid;                  /* Child thread id. */
      int exit_code;              /* Child exit code, if dead. */
      struct semaphore dead;      /* 1=child alive, 0=child dead. */
    };


  struct wait_status *wait_status; /* This process's completion state. */
  struct list children;            /* Completion status of children. */
```

Implement wait:

```
    - Find the child in the list of shared data structures.
      (If none is found, return -1.)

    - Wait for the child to die, by downing a semaphore in the
      shared data.

    - Obtain the child's exit code from the shared data.

    - Destroy the shared data structure and remove it from the
      list.

    - Return the exit code.
```

Implement exit:

```
    - Save the exit code in the shared data.

    - Up the semaphore in the data shared with our parent
      process (if any).  In some kind of race-free way (such
      as using a lock and a reference count or pair of boolean
      variables in the shared data area), mark the shared data
      as unused by us and free it if the parent is also dead.

    - Iterate the list of children and, as in the previous
      step, mark them as no longer used by us and free them if
      the child is also dead.

      - Terminate the thread.
```

3 Calling Conventions and Argument Passing

3.1 Calling Conventions

Sketch the stack frame of `helper` before it returns.

```
void helper(char* str, int len) {
 char word[len];
 strncpy(word, str, len);
 printf("%s", word);
 return;
}

int main(int argc, char *argv[]) {
 char* str = "Hello World!";
 helper(str, 13);
}
```

```
13
str
return address
saved EBP
!
d
l
...
l
e
H
```