Section 9: Cache, Clock Algorithm, Banker’s Algorithm and Demand Paging

CS162
Oct 23-24, 2018

Contents

1 Vocabulary 2

2 Problems 3
  2.1 Caching ................................................................. 3
  2.2 Clock Algorithm ..................................................... 4
  2.3 Banker’s Algorithm .................................................. 5
  2.4 Demand Paging ....................................................... 6
  2.5 Virtual Memory ...................................................... 7
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 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 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?

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?

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?
2.2 Clock Algorithm

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

<table>
<thead>
<tr>
<th>10 Bits</th>
<th>10 Bits</th>
<th>12 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table ID</td>
<td>Page ID</td>
<td>Offset</td>
</tr>
</tbody>
</table>

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

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
### 2.3 Banker’s Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

<table>
<thead>
<tr>
<th>Total</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T/R</th>
<th>Current</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0 2 2</td>
<td>4 3 3</td>
</tr>
<tr>
<td>T2</td>
<td>2 2 1</td>
<td>3 6 9</td>
</tr>
<tr>
<td>T3</td>
<td>3 0 4</td>
<td>3 1 5</td>
</tr>
<tr>
<td>T4</td>
<td>1 3 1</td>
<td>3 3 4</td>
</tr>
</tbody>
</table>

Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Repeat the previous question if the total number of C instances is 8 instead of 9.
2.4 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?

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.

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?
2.5 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  174184 1007372 96316 49 642 1395 678 123 128 0 1 99 0 0
0  0  174240 1007372 96316 0 0 0 0 48 88 0 0 100 0 0
0  0  174240 1007372 96316 0 0 0 0 33 75 0 0 100 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).

```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.

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  18584  120  0  0  0  0  1440  2647  2339  5  51  43  0  0
 4  1 1285828  20560  128  580  88  592440  14248  592440  18289  2171  3  58  36  4  0
 4  3 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  4740  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 697036  1921696  564  10096  28  0  288  0  77  148  0  0  100  0  0
 1  0 696980  878128  1337720  11272  412  0  1038840  0  11128  14854  1  25  54  21  0
 1  0 696980  21732  1895476  9348  0  0  1286640  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?)