CSE 120: Homework #3 Solutions

Fall 2015

1. Nachos VM Worksheet

2. When using physical addresses directly, there is no virtual to physical translation overhead. Assume it takes 100 nanoseconds to make a memory reference. If we used physical addresses directly, then all memory references will take 100 nanoseconds each.

   a. If we use virtual addresses with page tables to do the translation, then without a TLB we must first access the page table to get the approprate page table entry (PTE) for translating an address, do the translation, and then make a memory reference. Assume it also takes 100 nanoseconds to access the page table and do the translation. In this scheme, what is the effective memory reference time (time to access the page table + time to make the memory reference)?

      100 + 100 = 200 ns

   b. If we use a TLB, PTEs will be cached so that translation can happen as part of referencing memory. But, TLBs are very limited in size and cannot hold all PTEs, so not all memory references will hit in the TLB. Assume translation using the TLB adds no extra time and the TBL hit rate is 75%. What is the effective average memory reference time with this TLB?

      0.75*100 + 0.25*200 = 75 + 50 = 125 ns

   c. If we use a TLB that has a 99.5% hit rate, what is the effective average memory reference time now? (This hit rate is close to what TLBs typically achieve in practice.)

      0.995*100 + 0.005*200 = 99.5 + 1 = 100.5 ns

3. Consider a 32-bit system with 1K pages and simple single-level paging.

   a. With 1K pages, the offset is 10 bits. How many bits are in the virtual page number (VPN)?

      32 - 10 = 22 bits

   b. For a virtual address of 0xFFFF, what is the virtual page number?

```
0x0000FFFF = 0000 0000 0000 0000 1111 1111 1111 1111
&
0xFFFFFC00 = 1111 1111 1111 1111 1111 1100 0000 0000 (first 22 bits are page
=
0x0000FC00 = 0000 0000 0000 0000 1111 1100 0000 0000 = 0xFC00
```

```
0xFC00 is the page address.  The page number shifts off the offset from the

0x0000FC00 = 0000 0000 0000 0000 1111 1100 0000 0000
shift right >> 10 bits:
0x0000003F = 0000 0000 0000 0000 0000 0000 0011 1111

0x3F is the virtual page number.  (We'll accept 0xFC00 here, but
generally speaking it is important to distinguish between the virtual
page number and the virtual address of the page.)
```

c. For a virtual address of 0xFFFF, what is the value of the offset?

```
0x0000FFFF = 0000 0000 0000 0000 1111 1111 1111 1111
&
0x000003FF = 0000 0000 0000 0000 0000 0011 1111 1111 (last 10 bits are the c
=
0x000003FF = 0000 0000 0000 0000 0000 0011 1111 1111 = 0x3FF

0x3FF is the offset.
```

d. What is the physical address of the base of physical page number 0x4?

```
The offset is 10 bits.  To convert a page number to a page address,
we shift the page number left by the offset (10 bits):

  0x4 << 10 bits
= 0100 << 10 bits
= 0001 0000 0000 0000
= 0x1000
```

e. If the virtual page for 0xFFFF is mapped to physical page number 0x4, what is the physical address corresponding to the virtual address 0xFFFF?

The physical page for 0xFFFF is physical page 0x4.
The physical address of physical page 0x4 is 0x1000.
The offset of virtual address 0xFFFF is 0x3FF.
So the physical address of virtual address 0xFFFF is (0x1000 & 0x3FF) = 0x13FF.

4. [Crowley] Suppose we have a computer system with a 44-bit virtual address, page size of 64K, and 4 bytes per page table entry.
   a. How many pages are in the virtual address space? (Express using exponentiation.)

      The virtual address space is 2^{44} bytes in size. Thus, there are $2^{44}/2^{16} = 2^{28}$ pages.

   b. Suppose we use two-level paging and arrange for all page table pages (both master and secondary) to fit into a single page frame. How will the bits of the address be divided up?

      Each page table can hold $2^{16}/4$, or 2^{14}, page table entries. Thus, for each page table

level, we will use 14 bits of the virtual address to index into that page table. We will use 16 bits to address a specific byte within the data page. Conveniently, this all adds up to 44 bits, so the breakdown is: 14 bits for top-level page table, 14 bits for next page table, and 16 bits for offset.

c. Suppose we have a 4 GB program such that the entire program and all necessary page tables (using two-level pages from above) are in memory. (Note: It will be a *lot* of memory.) How much memory, in **page frames**, is used by the program, including its page tables?

4 GB translates to 2^{16} pages of memory. So, we will need 2^{16} page table entries, which can fit into 4 page tables. We also need to add in one additional page table for the top-level of the page table tree. Thus, we need 2^{16} + 5 page frames to store all data and page table pages.

5. [Crowley] Suppose we have an average of one page fault every 20,000,000 instructions, a normal instruction takes 2 nanoseconds, and a page fault causes the instruction to take an additional 10 milliseconds. What is the average instruction time, taking page faults into account? Redo the calculation assuming that a normal instruction takes 1 nanosecond instead of 2 nanoseconds.

Page faults incur an additional cost of 10 ms for every 20,000,000 instructions, which is an average additional cost of .5 ns per instruction. So, if instructions take 2 nanoseconds, the average instruction time is 2.5 ns, and if instructions take 1 nanosecond, the average instruction time is 1.5 ns.

6. [Tanenbaum] If FIFO page replacement is used with four page frames and eight pages (numbered 0–7), how many page faults will occur with the reference pattern 427253323126 if the four frames are initially empty? Which pages are in memory at the end of the references? Repeat this problem for LRU.

```
[FIFO]

+ 4 : <- 4
+ 42 : <- 2
+ 427 : <- 7
  427 : (2)
+ 4275 : <- 5
+ 2753 : <- 3, -> 4
  2753 : (3)
  2753 : (2)
  2753 : (3)
+ 7531 : <- 1, -> 2
+ 5312 : <- 2, -> 7
+ 3126 : <- 6, -> 5

= 8 page faults with pages 3126 in memory.
```

```
[LRU]

+ 4 : <- 4
+ 42 : <- 2
+ 427 : <- 7
  472 : (2 moved to end on access)
+ 4725 : <- 5
+ 7253 : <- 3, -> 4
  2753 : (3)
  7532 : (2 moved to end on access)
  7523 : (3 moved to end on access)
+ 5231 : <- 1, -> 7
  5312 : (2 moved to end on access)
+ 3126 : <- 6, -> 5

= 7 page faults with pages 3126 in memory.
```

7. If many programs are kept in main memory, then there is almost always another program ready to run on the CPU when a page fault occurs. Thus, CPU utilization is kept high. If, however, we allocate a large amount of physical memory to just a few of the programs, then each program produces a smaller number of page faults. Thus, CPU utilization is kept high among the programs in memory.

   What would the working set algorithm try to accomplish, and why? (Hint: These two cases represent extremes that could lead to problematic behavior.)

   The working set algorithm will try to keep in memory the working sets for the processes, i.e., the pages that each process needs to keep page faults to a reasonable level.

   (More explanation, not needed in an answer)

   Keeping every process in memory does mean that there will be a process to context switch to on a page fault. However, if the system is heavily faulting, then the OS will spend a lot of overhead serving page faults (thrashing). Much CPU time will be spent serving page faults, reducing CPU utilization for applications.

   Allocating large amounts of memory to a small number of processes will reduce their page fault rates considerably, but it is also likely that those few processes will be given more physical memory than they need. We could instead use that physical memory for other processes, increasing the number of active processes in the system and reducing average execution time.

8. [Silberschatz] Consider a demand-paging system with the following time-measured utilizations:

   CPU utilization: 20%
   Paging disk: 97.7% (demand, not storage)
   Other I/O devices: 5%

   For each of the following, say whether it will (or is likely to) improve CPU utilization. Briefly explain your answers.

a. Install a faster CPU

   Not likely to help, since the CPU is not the bottleneck.

b. Install a bigger paging disk

   Will not help. We are told storage capacity of the paging disk is not an issue. Using additional capacity—paging more data out to disk—is likely to hurt, not help performance.

c. Increase the degree of multiprogramming

   Will probably hurt. Memory in the system is overcommitted, so adding more programs will probably make the situation worse. (If, however, the additional jobs added are CPU-intensive but do not require much memory, then it is possible utilization may increase, as these extra jobs use the CPU cycles previously spent waiting on paging.)

d. Decrease the degree of multiprogramming

   Opposite of the case above—likely to help utilization by freeing up memory.

e. Install more main memory

   Most likely to improve utilization, by reducing the need to page data to disk.

f. Install a faster hard disk, or multiple controllers with multiple hard disks

   May help marginally, by speeding reads and writes to the paging disk, though since disks are still slow, unlikely to help a great deal.

g. Add prepaging to the page-fetch algorithms

   Could help, by bringing pages in to memory shortly before they are needed and reducing the need to wait on the paging disk. However, could also hurt if the pages brought in are not actually needed, thus increasing paging activity and memory pressure.

h. Increase the page size

   No large impact, though may hurt by increasing internal fragmentation and wasting memory (which is already in short supply).

---

voelker@cs.ucsd.edu