CSM 61C Fall 2019

Virtual Memory

Mentoring 9: November 11, 2019

1 VM Warm Up

1.1 The system in question has 1MiB of physical memory, 32-bit virtual addresses, and 256 physical pages. The memory management system uses a fully associative TLB with 128 entries and an LRU replacement scheme.

(a) What is the size of the physical pages in bytes?

1 MiB = 2^{20} B of physical memory and 256 = 2^8 physical pages means that 2^{20} B / 2^8 Pages = 2^{12} B / Page. Resulting in 2^{12} B or 4 KiB physical pages.

(b) What is the size of the virtual pages in bytes?

The size of virtual pages is the same size as physical pages, since all pages are the same size in the same system. Thus 2^{12} B or 4 KiB physical pages.

(c) What is the maximum number of virtual pages a process can use?

We have 4 KiB pages, resulting in $\log_2(2^{12}) = 12$ bits for page offset.
#VA bits = #VPN bits + #Page offset bits → #VPN bits = #VA bits - #Page offset bits
#VPN bits = 32 - 12 = 20, resulting in 2^{20} total virtual pages.

(d) What is the minimum number of bits required for the page table base address register?

Recall that the page table is also stored in memory, so the page table base address register must also span a physical address, aka $\log_2(2^{20})$ bits or 20 bits.

(e) Answer True or False to the following questions:
The page table is stored in main memory.
Every virtual page is mapped to a physical page.
The TLB is checked before the page table.
The penalty for a page fault is about the same as the penalty for a cache miss.
A linear page table takes up more memory as the process uses more memory.

T, F, T, F, F

2 Virtual Potpourri

2.1 For the following questions, assume the following:

- 16-bit virtual addresses
- 4 KiB page size
- 16 KiB of physical memory with LRU page replacement policy
- Fully Associative TLB with 4 entries and an LRU replacement policy

(a) What is the maximum number of virtual pages per process?

4 KiB page size = 12 offset bigs → 4 bits for VPN → 2^4 virtual pages = 16 virtual pages

(b) How many bits wide is the page table base register?

The page table base register is in memory so it's the same length as the physical address. 16 KiB physical memory = 2^{14} B so we have 14 bits.

2.2 For the following parts, assume that:

- Only the code and two arrays take up memory
- The arrays are both page-aligned (starts on page boundary)
- The arrays are the same size and do not overlap
- ALL of the code fits in a single page and this is the only process running

```
void scale_n_copy (int32_t *base, int32_t *copy, uint32_t num_entries, int32_t scalar) {
    for (uint32_t i = 0; i < num_entries; i++)
        copy[i] = scalar * base[i];
}
```

(a) If scale_n_copy were called on an array with N entries, where N is a multiple of the page size, how many page faults can occur in the worst-case scenario?

We want to calculate how many pages the N entries will span, which means that we have $\frac{N}{(2^{12}/2^2)}$ since we have N integers and $\frac{2^{12}}{2^2}$ integers per page. This gives us $\frac{N}{2^{10}}$ pages, but we have two arrays, meaning we have $2 * \frac{N}{2^{10}}$ pages used, resulting in $\frac{N}{2^9}$ page faults. However, do not forget that code requires a page as well, resulting in $\frac{N}{2^9} + 1$ page faults.

(b) In the best-case scenario, how many iterations of the loop can occur before a TLB miss?

We have 4 TLB entries, however one is reserved for the code. This leaves us with three pages left to use, but because the arrays are page aligned and we iterate through both arrays simultaneously, we can have a maximum one page for each array (figure out why this makes sense, if we have 2 pages for one array, we are limited by the array that only has one page before we have to miss). A page is 2^{12} B which means 2^{10} integers, meaning we can iterate through 2^{10} times since we have a page for each array.

3 Context Switch!

3.1 In this question, you will be analyzing the virtual memory system of a single-processor, single-core computer with 4 KiB pages, 1 MiB virtual address space and 1 GiB physical address space. The computer has a single TLB that can store 4 entries. You may assume that the TLB is fully associative with an LRU replacement policy, and each TLB entry is as depicted below.

TLB Entry

Valid Bit	Permission Bits	LRU Bits	Virtual Page Number	Physical Page Number

(a) Given a virtual address, how many bits are used for the Virtual Page Number and Offset?

VPN: 8, Offset: 12. There is 1 MiB of virtual memory so each virtual address is $\log_2(2^{20}) = 20$ bits in total. We know that each page is 4 KiB, so the offset is $\log_2(2^{12}) = 12$ bits. This leaves 8 bits for the virtual page number.

(b) Given a physical address, how many bits are used for the Physical Page Number and Offset?

PPN: 18, Offset: 12 The offset will remain 12 bits, as the page size is constant. Physical memory is 1 GiB, so, as physical addresses are $\log_2(2^{30}) = 30$ bits wide, we have 18 bits for the physical page number

3.2 For the next 2 parts, consider that we are running the following code, in parallel, from two distinct processes whose virtual memory specifications are the same as that of above. Both arrays are located at page aligned addresses. As a note, $65536 = 2^{16}$.

Process 0	Process 1

int a[65536];
for (int i = 0; i < 65536; i += 256) {
  a[i] = i;
  a[i + 64] = i + 64;
  a[i + 128] = i + 128;
  a[i + 192] = i + 192;
}
	int b[65536]
	for (int j = 0; j < 65536; j += 256) {
	  int x = j + 256;
	  b[x - 1] = j;
	  b[x - 2] = j+1;
	  b[x - 3] = j+2;
	  b[x - 4] = j+3;
	}

As our computer has only a single processor, the processes must share time on the CPU. Thus, for each iteration of the processes' respective for loop, the execution on this single processor follows the diagram at the top of the next page. A blank slot for a process means that it is not currently executing on the CPU.

Time	Process 0	Process 1
0	a[i] = i;	
1	a[i+64] = i + 64;	
2		int x = j + 256;
3		b[x-1] = j;
4		b[x-2] = j + 1;
5	a[i+128] = i + 128;	
6	a[i+192] = i + 192;	
7		b[x-3] = j + 2;
8		b[x-4] = j + 3;

(a) What is the TLB hit rate for executing the above code assuming that the TLB starts out cold (i.e. all entries are invalid)? Only consider accesses to data and ignore any effects of fetching instructions. You may assume that the variables i, j and x are stored in registers and therefore do not require memory accesses. Remember: you must flush the TLB on a context switch from one process to another!

50%. The first access is a[0], which brings in the VPN translation for a[64], the next access. When we switch processes, the TLB will be flushed, so the first two accesses to the array b will follow the pattern miss hit. When we switch back to process 0, we access a[128] (miss because TLB empty) and a[192] (hit because brought in by a[128]). The execution of the second two accesses to b are also miss hit because the TLB is flushed in between. This pattern continues to give a hit rate of 50

3.3 As opposed to the TLB architecture described above, let us consider a tagged TLB. In a tagged TLB, each entry additionally contains the Address Space ID (ASID), which uniquely identifies the virtual address space of each process. A tagged TLB entry is shown below.

Tagged TLB Entry

Valid Bit	Permission Bits	LRU Bits	ASID	VPN	PPN

On a lookup, we consider a hit to be if the (VPN, ASID) pair is present in the tagged TLB. This redesign allows us to keep entries in the TLB even if they are not a part of the process running on the CPU, so we do not have to flush the TLB when switching between processes.

Consider that we are using a tagged TLB and running the code in the manner described above.

(a) What is the hit rate for the tagged TLB for time 0-8 assuming it again starts out cold? You may make the same assumptions about the variables i, j, x and ignore the effects of fetching instructions.

75%. This time, when we we switch processes at time step 5, we don't flush the TLB so the last 4 accesses are hits. Then, we get 2/4 on the first four accesses, then 4/4 on the last half.

(b) What is the hit rate for the tagged TLB for all iterations assuming it again starts out cold? You may make the same assumptions about the variables i, j, x and ignore the effects of fetching instructions.

15/16. We first access a[0] , which brings in page 0 of the array for process 0. The first access is a miss, but the following access of a[64] is a hit because the mapping is in the TLB. When we switch to process 1, we access array b twice. These accesses, however, will be in the same page because they lie within a 1 KiB range and the start address of b is page aligned. Thus, we will miss the first time and hit on the second. When we switch back to process 0, the entry is already there (we don't flush anymore) so we get 2 hits. Going back to process 1 will give us the same result. The next miss will occur when we get to the next page, which occurs after 4 iterations because each iteration moves 1 KiB. As there are 4 accesses per iteration, we have 15 hits for every 16 accesses (per process). Thus, in total, we have a hit rate of 15/16.

(c) What is the smallest number of entries the TLB can have to still have the hit rate found in part 4?

2 Entries. We need to maintain the mapping for each processes.

4 VM Copy

4.1 For the following questions, assume the following (IEC prefixes are on the green sheet):

- You can ignore any accesses to instruction memory (code)
- 16 EiB virtual address space per process
- 256 MiB page size
- 4 GiB of physical address space
- Fully Associative TLB with 5 entries and an LRU replacement policy
- All arrays of doubles are page-aligned (start on a page boundary) and do not overlap
- All arrays are of a size equivalent to some nonzero integer multiple of 256 MiB
- All structs are tightly packed (field are stored contiguously)
- All accesses to structs and arrays go out to caches/memory (there is no optimization by reusing values loaded into registers)

```
void dblCpy(doubleFun* measurer, double* dblsToCpy) {
    measurer->fun = 0;
    for (uint32_t i = 0; i < ARRAY_SIZE; i += 4) {
        measurer->dbl[i] += dblsToCpy[i];
        measurer->fun += dblsToCpy[i];
    }
    measurer->fun /= ARRAY_SIZE;
}
```

(a) How many virtual page number bits are there and virtual address offset bits are there?

VPN: 36 Offset: 28

(b) How many physical page number bits are there and physical address offset bits are there?

PPN: 4 Offset: 28

4.2 (a) Assume the TLB has just been flushed. What TLB hit to miss ratio would be encountered if sizeof(double) * ARRAY_SIZE = 256 MiB and we run the above code?

$6(2^{23})$ TLB hit: 3 TLB misses

- Our loop has 2^{28} / $(2^3 * 2^2) = 2^{23}$ iterations. For each iteration, there are 6 memory accesses. For the loop entirely, there are $6(2^{23})$ total memory accesses.
- Before the loop, there is one memory access. After the loop there are two memory accesses (read & write measurer→fun).
- Thus, there are $6(2^{23})+3$ total memory accesses. We require three pages, one for dblsToCpy and two for doubleFun all that miss initially when loading up. Thus, our hit-miss ratio is $6(2^{23}) : 3$.

(b) In the best-case scenario, how many iterations can be executed with no TLB misses? Use IEC prefixes when reporting your answer.

2^{24} iterations = 16 MiB. In the best case scenario, all 5 slots in the TLB are full of valid info. We have 2 pages worth of measurer→dbl, 1 page for dbl→fun, and 2 pages for dblsToCpy. This means that our arrays are 2^{29} B (2 pages) in length. We index by every 4 elements of 8-byte doubles each time to obtain 2^{29} / 2^2 / 2^3 = 2^{24} iterations.