CSM 61C Spring 2020

Virtual Memory

Exam Question Compilation

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2019 Final Q10d
- Fall 2015 Final F1
- Spring 2015 Final F1
- Fall 2019 Final Q9
- Summer 2018 Final Q11
- Spring 2018 Final Q12
- Fall 2017 Final Q12

d) Virtual memory allows us to: (select all that apply)

- ☐ Pretend that programs do not have to share the address space with other programs.
- ☐ Have more stable and secure computer systems.
- ☐ Divide the entire address space into 4 sections specifically for static, code, heap, and stack.
- ☐ Provide the illusion that the computer has access to storage the size of DRAM but at the speed of disk.

SID: _____

F-1: You may need to *context switch* for this question (9 points)

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?

b. What is the size of the virtual pages in bytes?

c. What is the maximum number of pages a process can use?

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

Everybody Got Choices

e. Answer "Yup!" (True) or "Nope!" (False) to the following questions

i. The page table is stored in main memory **Yup!** **Nope!**
ii. Every virtual page is mapped to a physical page **Yup!** **Nope!**
iii. The TLB is checked before the page table **Yup!** **Nope!**
iv. The penalty for a page fault is about the same as the penalty for a cache miss **Yup!** **Nope!**
v. A linear page table takes up more memory as the process uses more memory **Yup!** **Nope!**

SID: _____

F1: *Paging all CS61C students* (9 points)

Consider a byte-addressed machine with a 13-bit physical address space that can hold two pages in memory. Every process is given 16MiB of virtual memory and pages are evicted with an LRU replacement scheme.

a) What are the sizes of the following fields in bits?

Virtual Page Number: _____ Virtual Address Offset: _____

Physical Page Number: _____ Physical Address Offset: _____

b) Consider the following code snippet:

```
// a and b are both valid pointers to
// different arrays of length ARRAY_SIZE
void enumerate(int* a, int* b) {
     for (int i = 0; i < ARRAY_SIZE; i++) {
          a[i] = i;
          b[i] = ARRAY_SIZE - i;
     }
}
```

The compiled binary for the program containing this code snippet weighs in at 4096B. If this code was executed on the machine, what is the maximum value of `ARRAY_SIZE` that would allow this code to execute with 0 page faults in the best-case scenario? (Answer in IEC prefix: 8Gi, 32Ti, etc)

`ARRAY_SIZE` = _____

c) How could we modify the above code snippet to allow a larger `ARRAY_SIZE` and execute with the fewest page faults in the best-case scenario? Write the new code below:

Q9) We've got VM! Where? **(15 pts = 2 + 3 + 5 + 5*1)**

Your system has a 32 TiB virtual address space with a single level page table. Each page is 256 KiB. On average, the probability of a TLB miss is 0.2 and the probability of a page fault is 0.002. The time to access the TLB is 5 cycles and the time to transfer a page to/from disk is 1,000,000 cycles. The physical address space is 4 GiB and it takes 500 cycles to access it. The system has an L1 physically indexed and tagged cache which takes 5 cycles to access and a hit rate of 50%. On a TLB miss, the MMU checks physical memory next.

a) How many bits is the Virtual Page Number?

_____ bits

SHOW YOUR WORK

b) What is the total size of the page table (in **bits**), assuming we have **no** permission bits or any other metadata in a page table entry, just the translation?

_____ bits

SHOW YOUR WORK

c) What is the average memory access time (in cycles) for a single memory access for the current process? Assume the page table is resident in DRAM.

_____ cycles

SHOW YOUR WORK

d) Which of the following, if any, **must be done** when we switch to a different process? Do **not** select any option that is unnecessary.

		Yes	No
1)	Update page table address register	◯	◯
2)	Evict pages for the previous process from RAM	◯	◯
3)	Clear TLB dirty bits	◯	◯
4)	Clear cache valid bits	◯	◯
5)	Clear TLB valid bits	◯	◯

SID: _____

Question 11: TL;Br (Too Long; But read) (10 pts)

Consider a machine with 4 KiB pages, a 32-bit virtual address space with 256 MiB of DRAM for main memory. It has a single level of page table and a TLB containing 4 entries which is fully associative.

1. How many bits are there for the VPN? How many bits are there for the offset of the virtual address?

VPN: _____ Offset: _____

2. How many bits are there for the PPN? How many bits are there for the offset of the physical address?

PPN: _____ Offset: _____

Next let's identify how translations of various **virtual addresses** will be resolved. For each translation **identify if the result is a TLB hit, a Page Table Hit, or a Page Fault**. **Assume each access restarts from the original layout of the TLB and Page Table**. Assume **any page table entries not shown have a valid bit of 0.**

TLB

VPN	PPN
0x6	0x15
0x4	0x31

PAGE TABLE

VPN	Valid Bit	PPN
0x0	1	0x3
0x1	1	0x7
……...	……...	……...
0x4	1	0x31
0x5	0	0x3
0x6	1	0x15
0x7	1	0x11
……...	……...	……...

3. 0x7ABC
   Ⓐ TLB Hit Ⓑ Page Table Hit Ⓒ Page Fault

4. 0x3000
   Ⓐ TLB Hit Ⓑ Page Table Hit Ⓒ Page Fault

5. 0x6423
   Ⓐ TLB Hit Ⓑ Page Table Hit Ⓒ Page Fault

6. 0x5221
   Ⓐ TLB Hit Ⓑ Page Table Hit Ⓒ Page Fault

7. 0x20282
   Ⓐ TLB Hit Ⓑ Page Table Hit Ⓒ Page Fault

Finally let's consider how the TLB and Page Table change (or don't) as a result of memory accesses. Assume we have **256 B Pages, 2 TLB entries, 4 physical pages and 8 virtual pages in our machine**. These initially appear as shown below. After each access fill in the new contents of the TLB and PT. Assume we evict from main memory and the TLB by evicting the smallest VPN. **Once again assume each access restarts from the original layout of the TLB and Page Table**. If a row in either the TLB or the Page Table does not change from the original, you can either fill it in again or **leave it blank** in the same location.

TLB

VPN	PPN
0x1	0x2
0x5	0x3

PAGE TABLE

VPN	Valid Bit	PPN
0x0	0	0x3
0x1	1	0x2
0x2	0	0x1
0x3	1	0x0
0x4	0	0x0
0x5	1	0x3
0x6	1	0x1
0x7	0	0x2

SID: _____

8. 0x608

TLB

VPN	PPN

PAGE TABLE

VPN	Valid Bit	PPN

9. 0x2F4

TLB

VPN	PPN

PAGE TABLE

VPN	Valid Bit	PPN

Problem 12 ***[F-4] Virtual Memory*** **(20 points)**

Demand paging (storing part of a process' memory on disk) is yet another example of caching in computer systems. If we think of main memory as a cache for disk, what are the properties of this cache? Assume a machine with 64 bit addresses, 16KB pages, a 4-way fully associative TLB, and 8B words.

(a) Associativity?

- ○ Direct Mapped
- ○ Fully Associative
- ○ N-Way Set Associative

(b) Block size: _____

(c) Address layout. Your answer should be of the form [N:M] where N is the bit number of the most significant bit of the field and N is the bit number of the least significant bit of the field. For example, if the tag consists of the first 4 least-significant bits, you should write [3:0]. If the field is not applicable to paging, you may write "N/A".

Tag bits:_____ Index bits:_____ Offset bits:_____

(d) Write policy?

- ○ Write Through
- ○ Write Back

(e) Allocation policy?

- ○ Write Allocate
- ○ Write No Allocate

TLB Reach. We have written a strange and mysterious summation function. It uses a mystery constant called `T`. You may assume that `T` is defined (but you don't know what to) and that `arr` will always have enough elements (the function will never access outside of `arr`). The function is run on a machine with the following properties:

- 64 bit addresses
- 4 byte words
- 4KiB pages
- 4GiB of main memory
- 1MiB fully-associative cache with 64 byte blocks
- 2 entry fully associative TLB
- 4 level page table with 8 byte entries
- The OS uses LRU when paging to disk

```
#define NITER 10*1024*1024
#define T ???       // see below

int MysterySum(int *arr) {
     int i = 0;
     int sum = 0;
     for(; i < NITER / 2; i++)
          int p = (i % T)*4096;
          int b = i % 4096;
          sum += arr[p + b];
     }

     /* Timer starts here*/
     for(; i < NITIR; i++) {
          int p = (i % T)*4096;
          int b = i % 4096;
          sum += arr[p + b];
     }
     /* Timer ends here */

     return sum;
}
```

(f) **Performance of T**

Rank the the following values of T based on how fast the second loop only executes (assuming the first loop has already ran). You should state whether pairs of values are $<$ or $=$. For example, you should write $1 < 2$ if T=1 causes the second loop to run strictly slower than T=2. Likewise, you could write 8=2 if 8 is about as fast as 2.

T = 1, 2, 3, 4

(g) **System Design**

What system parameter would you change in order to maximize system performance for T=27. You must mark only one of the following (pick the one with the largest performance gain):

- ○ Address Size
- ○ Page Size
- ○ Word Size
- ○ Main Memory Size
- ○ Cache Capacity
- ○ Cache Block Size
- ○ TLB Capacity
- ○ TLB Associativity
- ○ Page Table Depth
- ○ Page Table Entry Size

(h) **Page Table Walk**

Given the list of virtual addresses, find the corresponding physical addresses. For each address, you must also note whether the access was a TLB hit, Page Table hit, or Page Fault (by writing yes/no for each). If the access is a page fault, you should leave the PPN and PA fields blank. Do not add this entry to the TLB.

Our virtual memory space has 16-byte pages and maintains a fully-associative, two-entry TLB with LRU replacement. The page table system is hierarchical and has two levels. The two most-significant bits of the VPN index the L1 table, and the two least-significant bits of the VPN index the L2 table.

Virtual Address	Virtual Page Number	Physical Page Number	Physical Address	TLB Hit, Page Table Hit, Page Fault?
0x10				
0x5C				
0x39				
0x1F				

Page Table Base Register	0x00

Memory:

Address	Contents
0x00	0x20
0x04	
0x08	0x10
0x0C	
0x10	
0x14	0x1C
0x18	0x28
0x1C	
0x20	
0x24	0x12
0x28	0x09
0x2C	0x5C

TLB:

VPN	PPN

Q12: Virtual Memory

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

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

2. Given a physical address, how many bits are used for the Physical Page Number and Offset?

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;

3. 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!

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.

4. What is the **hit rate** for the tagged TLB 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.

5. What is the smallest number of entries the TLB can have to still have the hit rate found in part 4?