CSM 61C Spring 2020

Virtual Memory

Exam Question Compilation Solutions

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.

Nope, that can happen with or without VM

☐ Provide the illusion that the computer has access to storage the size of DRAM but at the speed of disk.

vice-versa

F-1: You may need to context switch for this question

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?
Physical page size is going to be the size of physical memory divided by the number of pages in physical memory. 1MiB = 2^20 and 256 pages = 2^8, thus 2^20/2^8 = 2^12 bytes

b. What is the size of the virtual pages in bytes?
Virtual and physical pages are always the same size, thus 2^12 bytes.

c. What is the maximum number of virtual pages a process can use?
Similar to part a, the number of pages in virtual memory will be the size of virtual memory divided by the size of a page. Virtual memory in this case is 2^32 bytes, thus 2^32/2^12 = 2^20 pages.

d. What is the minimum number of bits required for the page table base address register?
The page table base register holds a physical address which is a pointer to the start of the page table for the current running process. Thus, since this register holds a physical address, and every physical address is 20 bits (because physical memory is 2^20 bytes), 20 bits are needed.

Everybody Got Choices

i. The page table is stored in main memory. True, page table must be stored in memory to be able to be used.

ii. Every virtual page is mapped to a physical page False, for a couple reasons that I can think of. Virtual memory is usually much bigger than physical memory, so every virtual page cannot be mapped to a physical page. Also, the operating system may prevent some virtual pages to be mapped to physical pages for access reasons (restricted memory regions)

iii. The TLB is checked before the page table Definitely True

iv. The penalty for a page fault is about the same as the penalty for a cache miss False. Pages are much bigger than cache blocks, and disk is much slower than memory, so reading a page from disk into memory is much slower than reading a block from memory into a cache

v. A linear page table takes up more memory as the process uses more memory False. The basic page table we cover is a linear page table, which is stored all in memory with enough indexes for all of the virtual page numbers. If the process uses more memory, more entries in the page table will be valid (valid bit set to 1), but the page table will always be the same size.

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.

Solve in order of blue circles

Virtual Addresses are $\log_2(16\text{MiB}) = \log_2(2^{24}) = 24$ bits

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

(4) VPN = Addr size − offset = 24 − 12

Virtual Page Number: 12 (3) Virtual Address Offset: 12

pages are same size

PPN bits = $\log_2(2) = 1$, as we have 2 pages

(1) Physical Page Number: 1 Physical Address Offset: 12

(2) 13 bit addr − 1 bit PPN

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)

Each page is $2^{12} = 4096$B. Thus, the code fits in one page, so to have no faults, the data must be in the other. Each array is the same size, so 2*ARRAY_SIZE ≤ PAGE_SIZE ⇒ max

`ARRAY_SIZE` = 512 arr size is $\frac{4096\text{B}}{2} = 2048\text{B} \Rightarrow 512$ ints

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:

```
for (int i=0; i< ARRAY_SIZE; i++){
   a[i]=i;
}
for (int i=0; i< ARRAY_SIZE; i++){
   b[i]=ARRAY_SIZE-i;
}
```

We can make ARRAY_SIZE = page size because we no longer worry about a & b knocking each other, or the code, out of memory

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?

27 bits

SHOW YOUR WORK

Number of reachable virtual addresses: log2(32 TiB) = 45
Bits needed to reach all addresses in a page: log2(256 KiB) = 18
So the virtual page number bits are: 45 - 18 = 27

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?

14×2^{27} bits

SHOW YOUR WORK

We need to figure out the number of bits in the physical page number. It is the same method except we use the physical address space:
Number of reachable physical addresses: log2(4 GiB) = 32
So PPN size is 32 - 18 = 14. We do not have any metadata bits so the total number of bits in a PTE is 14. To figure out how many entries we need, we need to look at the total number of virtual page numbers we have = 27. This means we need 2^{27} entries in the page table. This means we need a total of 14×2^{27} bits in our page table.

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.

760 cycles

SHOW YOUR WORK

Translation AMAT = 5 + ⅛(500 + 2/1000(1M))
= 5 + ⅛(500 + 2000)
= 5 + ⅛(2500)
= 5 + 500
= 505
plus
Data access AMAT = 5 + 50% (500)
= 5 + 250
= 255
AMAT (overall) = 505 + 255 = 760

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: **20** Offset: **12**

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

PPN: **16** Offset: **12**

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 (All changes are in red. Everything else remained the same)

TLB

VPN	PPN
0x6	**0x1**

PAGE TABLE

VPN	Valid Bit	PPN

9. 0x2F4

TLB

VPN	PPN
0x2	**0x2**

PAGE TABLE

VPN	Valid Bit	PPN
0x1	**0**	**X (doesn't matter)**
0x2	**1**	**0x2**

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:

Solution: 16KB

(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".

Solution: Tag bits:[63:14] Index bits:N/A Offset bits:[13:0]

(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

Solution: $3 = 4 < 1 = 2$

(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	○ Cache Block Size
○ Page Size	● TLB Capacity
○ Word Size	○ TLB Associativity
○ Main Memory Size	○ Page Table Depth
○ Cache Capacity	○ 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.

Solution:

Virtual Address	Virtual Page Number	Physical Page Number	Physical Address	TLB Hit, Page Table Hit, Page Fault?
0x10	0x1 = 0b00 01	0x12	0x120	Page Table Hit
0x5C	0x5 = 0b01 01			Page Fault
0x39	0x3 = 0b00 11	0x5C	0x5C9	Page Table Hit
0x1F	0x1 = 0b00 01	0x12	0x12F	TLB Hit

TLB:

VPN	PPN
0x1 → 0x2	0x12 → 0x9
0x3 → 0x1	0x5C → 0x12

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?

VPN: 8, Offset: 12.

Each virtual address is log_2(1 MiB) = 20 bits in total. We know that each page is 4 KiB, so the offset is log_2(4 KiB) = 12 bits. This leaves 8 bits for the virtual page number.

2. 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. As physical addresses are log_2(1 GiB) = 30 bits wide, we have 18 bits for the physical page number.

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

```
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;
}
```

Process 1

```
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!
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%.

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.

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.

5. 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.