# Computer Science 605.611

Problem Set 10

1. A virtual memory system contains 1048576 pages. Each page is 8192 bytes in size.

Each logical or virtual page is mapped to some frame within the 2 GB (231 byte) physical

memory. A single level forward page map table is used to translate virtual addresses

into physical addresses. Assume that each PTE contains only the following three items:

* + Frame number
  + Valid bit
  + Dirty bit

a) (3) How many PTEs (page table entries) are required for the forward page map table

(PMT)?

You would need 1048576 PTEs for this page map table.

b) (3) What is the minimum number of bits required for each frame number?

Number of frames = 2^31 / 8192 = 2^18. Log(2^18) = **18 bits**

c) (5) What is the number of the PTE within the forward PMT that must be used to translate the

virtual address 0x42B1412C into a physical address? PTEs are numbered starting from 0.

Since each page is 8192 bytes (2^13) we know that the lower 13 bits of the virtual address.

represent an offset. 0x42B1412C in binary is 01000010101100010100000100101100, removing the lower 13 bits gives us 0100001010110001010, converted to decimal is **136586**, or our PTE

d) (5) Recall that memory addresses are unsigned binary numbers. What is the

virtual address of the final byte in page number 17719? Express your answer in hex.

First we would need to convert 17719 into hex, 0x4536. The offset of the last byte is 8191 or 1FFF in hex. So the Virtual address is **0x45371FFF**.

e) (3) How many bits are contained in each virtual address on this system?

Since the number of virtual pages is 1048576 or 2^20 and each page is 2^13 bytes. The number of bits contained on the page would be 20 + 13 = **33 bits**.

2. For a different virtual memory system, the instruction sw $4, 80($6) causes a page fault when it attempts to write to the final word within some virtual memory page. The 32-bit word is written at offset 16380 within the page. In response to the page fault, page 10456 is loaded into frame 580. This system employs 32-bit virtual addresses and 24-bit physical addresses.

a) (5) Use hex to show the contents of register $6 when the sw $4, 80($6) instruction causes

the page fault.

Contents of $6 = **0x2908400**\_\_\_

Since we know that the page number loaded is 10456 and the offset is 16380. The page that

Causes the fault would look like $6 = page num + offset = 29084000

b) (3) Each PTE contains a valid bit, a modify bit, and a frame number. If the sum of the

number of bits required for these three items is not a multiple of 8, the PTE is padded on the left

with enough 0 pad bits to make the size of the PTE a multiple of 8.

What is the minimum number of zero pad bits required to make the PTE width a multiple of 8?

The minimum number of pad bits = \_**4**\_.

Since there are the number of frames can be calculated as 2^24 / 16384 = 2^10 (10 bit width)

And the total width of PTE = frame number width + valid bit + dirty bit = 10 + 1 + 1 = 12 bits.

Since we need to make this a multiple of 8 we can do (8 – (12mod8)) or 4 pad bits.

3. A virtual memory system uses 31-bit virtual addresses and employs a two-level page map table system like the system described in 10L: mod10\_3.2.1\_Multi-level\_Page\_Tables. Each page is 8192 bytes in size. The system uses one first-level table containing eight entries. Each entry can hold the 20-bit physical address of one of the possible 8 second-level page map tables. Memory space for the first-level table is allocated initially. But memory space for each second-level table is not allocated until the program references some page that corresponds to an entry within that second-level table. This approach saves memory space because second-level tables that are not referenced by a program are not allocated. Each entry in a second-level table indicates whether the corresponding page is in memory. If the page is in memory, the second-level table entry contains the number of the memory frame that contains the page.

With this approach, each virtual address is partitioned into 3 fields as shown below. The leftmost field indicates which second-level table to use. The middle field indicates which entry in the selected second-level table to use. The rightmost field within the virtual address contains the offset within the page to the data item referenced by the virtual address.

|  |  |  |
| --- | --- | --- |
| Second-level Page Table# | Page Table Entry# | Page Offset |

ProgramA references virtual addresses in the range 0x1FFF8008 to 0x200030C4 on this two-level system.

a) (3) How many bits are required for the Page Offset field? The number of bits required = \_\_**13 bits**\_\_

page size = 8192 bytes, num of bits required = log\_2(page size) = log\_2(8192) = 13

b) (5) How many entries in the first level table are used by ProgramA? The number of first-level table entries used by ProgramA = \_\_**2**\_\_\_

We need to find the range that the two addresses hold. 1 = 001 and 2 = 010. Since the number of entries in table 1 is 8 (2^3) so 3 bits in first table entry. Only entry 1 and 2 are used leaving us with 2 first level table entries used.

c) (5) How many PTEs must be contained in each second-level allocated for ProgramA?

The number of entries in each second-level table = \_\_**32768**\_\_\_

Since we can determine the page table entry# = 15 bits (31 – 13 (offset) = 18 – 3 (table entry) = 15). 2^15 = 32768 entries.

d) (5) For each second-level table allocated for ProgramA, how many entries in the second-level table are actually used by ProgramA?

1FFF8008 = 001 **1111 1111 1111 100** 0 0000 0000 1000

200030C4 = 010 **0000 0000 0000 001** 1 0000 1100 0100

There are **6 table entries needed** for the middle numbers to “meet up”. **4 from table 1 and 2 from table 2**

4. This problem is similar to problem 1, but it deals with a virtual memory system that contains

524288 pages, each of which is 8192 bytes in size. Instead of a forward page map table, this

system employs an inverted page map table. Each logical or virtual page can map to some

frame within the system’s 2 GB (2^31 bytes) physical memory. The system executes only a

single task at a time, so no task ID is required.

a) (3) How many PTE’s (page table entries) are required if a single level inverted page map

table is used to translate virtual addresses to physical addresses?

The number of PTE’s required is the same as the amount of virtual pages, **524288**

b) (3) Each PTE in the inverted page map table must contain at least three fields. What does

each field indicate and how many bits are contained in each field?

The three fields needed would be the page number, valid bit and modification bit. TO find the

Number of bits contained we can simply do log\_2(524288) = 19 bits, and the other two bits are

Only 1 bit each.

Page number = 19 bits, modification bit = 1, valid bit = 1

c) (3) With the inverted PMT, which PTE(s) must be examined or read to translate virtual

address 0x42B1412C to a physical address? PTE’s are numbered starting from 0.

To find which PTE(s) are examined is relatively simple. Virtual Page number = virtual address/

Page size = 0x42B1412C/8192 = ~**136586**.

d) (3) Recall that memory addresses are unsigned binary numbers and bytes are numbered

starting from 0. What is the physical address of byte number 1 in the final frame? Express your

answer in hex.

To find the byte number 1 in the final frame we need to do a few things. Last frame starting =

Total physical memory – page size = 2^31 – 8192 = 0x7FFFFFFF – 0x2000 = 0x7FFFFDFF.

We then need to add 1 for physical address so **0x7FFFFE00**.

5. (3) A fetch of an instruction from address 0x2FE0C848 on a MIPS system causes a hit in the I-cache. The direct-mapped I-cache uses 64-byte cache lines and contains a total of 2097152 bytes of data. Show, in decimal, the tag, the line number and the offset within the line for the address 0x2FE0C848.

Tag = **\_\_383\_\_\_**

Line# = **\_\_801\_\_\_**

Offset =**\_\_8\_\_\_**

Todo this we need to determine the width of each of these pieces. Since we know that there are 64 byte cache lines, we can determine that the offset field would need to be 6 bits (2^6 = 64). To find the line# we need to convert 2097152 into a power of 2, 2^21 and divide it by one cache line or 2^6, 2^21/2^6 = 2^15, thus 15 bits for a line#. Luckily since we know that they are 32 bits in width the tag can be found to be the first 11 bits. (x + 15 + 6 = 32, x=11). Now we need to convert address 0x2FE0C848 into binary, so 00101111111 000001100100001 001000, in decimal each of these numbers becomes 383, 801, 8.

6. A system has a main memory with a read access time of 1800 ns and uses a unified L1 cache that operates in look-aside mode. The cache has a read hit ratio of 96%. As you know, the purpose of the cache is to make the main memory appear to have a shorter access time.

a) (3) What cache read access time is required to make the system appear to have an average memory read access time of 340 ns?

Lets start by analyzing the average memory access read time. 340ns = (cache hit ratio \* Cache access time) + (cache miss ratio \* main memory access time) = (0.96 \* x) + (0.04 \* 1800ns). 340ns = 0.96x + (0.04 \* 1800ns) = 354.16 = x + 75 = **279.16ns or ~280ns**.

b) (3) Suppose that instead the cache operates in look-through mode and has a read hit ratio of 85%. What cache read access time is required to make the system appear to have an average memory read access time of 280 nano-seconds?

To calculate this we can look at it as 280ns = 0.85x + (0.15(x + 1800ns) = 280 = 0.85x + .15x + 270. **10ns** = x

7. A matrix, in which each element is a single precision 32-bit floating point number, contains

1024 rows and 128 columns. Row numbers range from 0 to 1023. Column numbers range from

0 to 127. The starting virtual address assigned for the matrix is 0x400B6000.

For the purposes of this problem, assume that a contiguous memory area containing only 32

frames is reserved for pages that must be loaded in response to the matrix references. Frames

and pages are 8192 bytes in size. The starting physical address for the memory area allocated

for the matrix is 0x37F6000. Prior to executing the code, there are no matrix elements in

memory. The first reference to each new page triggers a page fault and causes the entire page to

be loaded from disk into the next available frame. An LRU page replacement policy is used

when needed.

The code shown below doubles each of the first 2048 matrix elements beginning at the starting

address of the matrix. Note that the bgtz is a delayed branch instruction.

lui $2, 0x400B ; Matrix starting address

ori $2, 0x6000

ori $3, $0, 2048 ; 2048 elements

loop2:

lwc1 $f4, 0($2) ; read the next element

add.s $f4, $f4, $f4 ; double the element

swc1 $f4, 0($2) ; store the element

addiu $3, $3, -1 ; row elements remaining

bgtz $3, loop2 ; repeat for next element

addiu $2, $2, 4 ; point to next row element

a) (3) If the matrix elements are arranged in row major order (i.e., one row after another),

indicate the range of row numbers and the range of column numbers for the elements that are

updated by this code.

The rows updated are \_0\_\_ to \_\_15\_\_\_

The columns updated are \_\_\_0\_\_ to \_\_127\_\_

b) (3) If the matrix elements are arranged in column major order (i.e., one column after

another), indicate the range of row numbers and the range of column numbers for the elements

that are updated by this code.

The rows updated are \_0\_\_ to \_\_1023\_\_\_

The columns updated are \_\_\_0\_\_ to \_1\_\_\_

c) (5) Assume that the matrix elements are arranged in row major order. How many page faults are caused by some different code that steps sequentially through all rows and updates every matrix element in every row?

Number of page faults = \_\_\_\_64\_\_\_\_\_

d) (3) Assuming row major storage order, show the contents of register $2 required by the instruction swc1 $f2,0($2) to write to matrix element M[34][56] (i.e., row number 34 and column number 56). The rows and columns are numbered starting from 0.

Contents of $2 = 0x400BA4E0\_\_\_\_

Offset = (34\*128+56)\*4 = 17632 + 0x400B6000 = 400BA4E0

e) (5) Assume that the matrix elements are arranged instead in column major order. How many

page faults are caused by some different code that steps sequentially through all rows and

updates every matrix element in every row?

1024 rows \* 128 columns \* 4 bytes = 54288 bytes / 8192 = 64

Each column is 4096 bytes so 64/2 = **32 page faults**

f) (3) Assuming column major storage order, show the contents of register $2 required by the instruction swc1 $f2,0($2) to write to matrix element M[34][56] (row 34 and column 56). The rows and columns are numbered starting from 0.

Contents of $2 = 0x400F2568\_\_\_\_

Offset = (56 \* 1024 + 34) \* 4 = 0x400B6000 + 229512 = 0x400F2568

8. A cache contains a total of 4194304 lines. The cache is organized as a 1-way set associative cache with a line size of 512 bytes.

a) (3) What is the proper 32-bit memory address format to use with this cache? Indicate the number of bits in each of the fields required for the memory address and show the proper order for the fields within the address.

To determine the fields width we need to do a few things. First we can find how large the offset is by doing log\_2(512) = 9 bits. To find the index we simply need to do a log\_2(4194304) = 22 bits. Finally the tag can be found by doing 32 – 9 -22 = 1 bit.

**Offset = 9 bits**

**Index = 22 bits**

**Tag = 1 bit**

b) (3) How many different tags must be examined to detect a write miss in this cache?

To determine how many tags must be examined we just need to analyze how the system works.

Since it is a 1 way set associative only **1 tag** must be examined to detect a write miss in the

cache

9. A cache contains a total of 8388608 lines. The cache is organized as a 4-way set associative cache with a line size of 128 bytes.

a) (3) What is the proper 32-bit memory address format to use with this cache? Indicate the number of bits in each of the fields and the order of the fields required for the memory address.

Block size is 128bytes, offset bits = log(128) = 7 bits. Log(Num of lines /4) = index bits = log2(8388608/4) = 21 bits. 32 – (21+7) = 4 tag bits.

Tag bits **= 4 bits**

index bits **= 21 bits**

offset bits = **7 bits**

b) (3) How many different tags must be examined to detect a read miss in this cache?

To detect a miss read requires **4 different tags**.