Exercise VM: Virtual Memory

Parts of Addresses

x86 virtual addresses have three component parts.

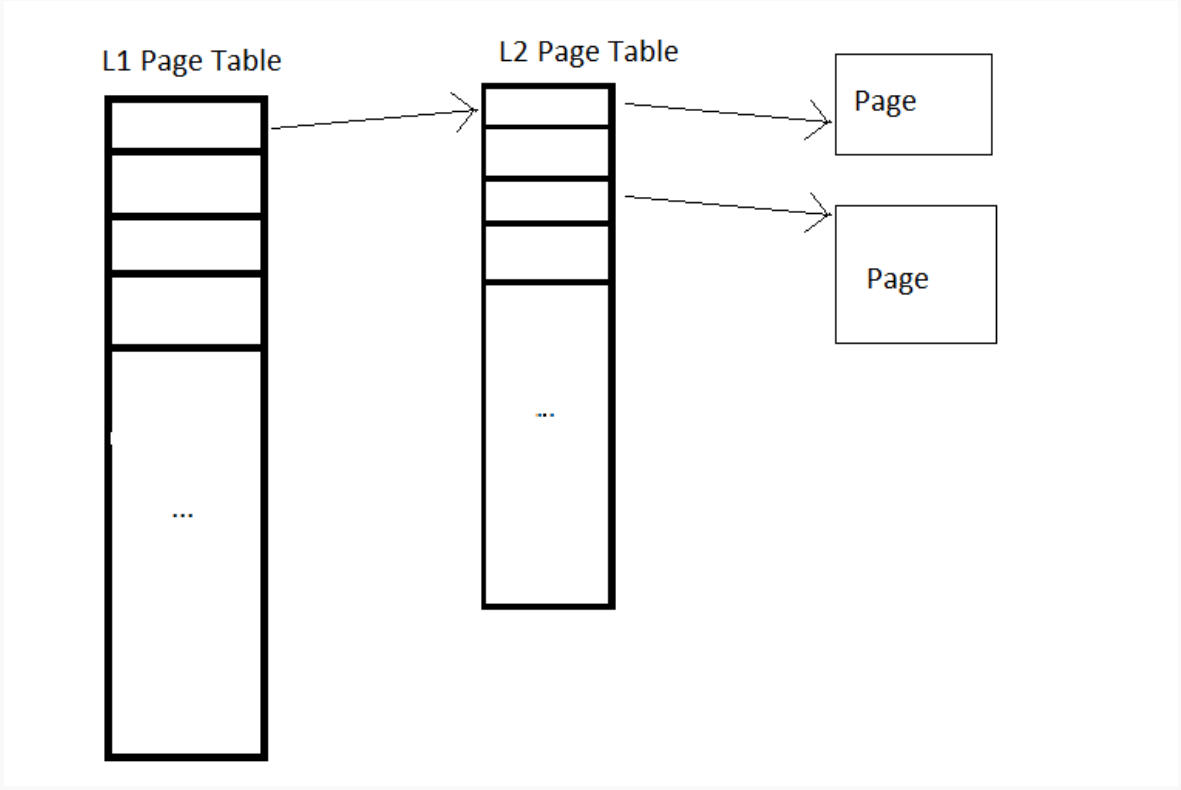
- Bits 0 through 11 (the least significant 12 bits) are the *page offset*. We'll say PAGEOFFSET for short.
- Bits 12 through 21 (the next most significant 10 bits) are the *level-2 page table index*. We'll say L2PAGEINDEX for short.
- Bits 22 through 31 (the most significant 10 bits) are the *level-1 page table index*. We'll say L1PAGEINDEX for short.

These parts are calculated in C as follows:

```
unsigned PAGEOFFSET(uint32_t va) {
   return va & 0xFFF;
}

unsigned L2PAGEINDEX(uint32_t va) {
   return (va >> 12) & 0x3FF;
}

unsigned L1PAGEINDEX(uint32_t va) {
   return va >> 22;
}
```



How to do this in your head? If you have an address in hexadecimal 0xMNOPQRST (where each capital letter represents a hexadecimal digit), then:

- PAGEOFFSET(0xMNOPQRST) == 0xRST. That's just the lower 3 hex digits.
- L2PAGEINDEX(0xMNOPQRST) == (0xOPQ & 0x3FF). That's the next 3 hex digits, except that you need to mask off the top two bits, to get a number between 0x000 and 0x3FF.
- L1PAGEINDEX(0xMNOPQRST) == (0xMNO >> 2). That's the top 3 hex digits shifted right by two. This is not as easy as the other ones. If you're confused write it in binary.

Part A. What is:

1. PAGEOFFSET(0)? 0
2. L2PAGEINDEX(0)? 0
3. L1PAGEINDEX(0x9000)? 0

4. PAGEOFFSET(1023)? 1023
5. L1PAGEINDEX(0x10000000)? 0x40
6. L2PAGEINDEX(0x10000000)? 0
7. L2PAGEINDEX(0x10000FFF)? 0
8. PAGEOFFSET(0x00801000)? 0
9. L2PAGEINDEX(0x00801000)? 1
10. L1PAGEINDEX(0x00801000)? 2
11. PAGEOFFSET(0x00F0089A)? 0x89A
12. L2PAGEINDEX(0x00F0089A)? 0x300
13. L1PAGEINDEX(0x00F0089A)? 3

Part B. For each question, give a virtual address that satisfies the constraints.

1. PAGEOFFSET = 0 0x97865000
2. PAGEOFFSET = 0 and L1PAGEINDEX = 12 0x033FF000
3. L1PAGEINDEX = 8, L2PAGEINDEX = 128, PAGEOFFSET = 256
   0x02080100
4. L1PAGEINDEX = 0, L2PAGEINDEX = 0xC, PAGEOFFSET = 0x7FF
   0x0000C7FF

Pages

The x86 *page size* is 4KB = 2^{12} bytes = 4096 bytes = 0x1000 bytes.

A *page* is 4096 bytes of contiguous memory whose first address is a multiple of 4096.

For instance, we might refer to “physical page 0x1000.” This means the physical page of memory comprising addresses 0x1000 through 0x1FFF.

Not every 4096 bytes of contiguous memory is a page. For example, consider 4096 bytes of memory starting at physical address 0x1800. The byte range is 0x1800 through 0x27FF. This overlaps with *two* physical pages: the portion from 0x1800–0x1FFF is part

of physical page 0x1000, and the portion from 0x2000-0x27FF is part of physical page 0x2000.

Any single byte belongs to exactly one page. Two contiguous bytes may belong to 1 or 2 physical pages, it depends how they're aligned.

Virtual address translation

x86 virtual address translation uses a *two-level page table.* This is a two-level tree.

The tree has a single root node, called the *level-1 page table*, and between 0 and 1024 level-2 nodes, which are called *level-2 page table pages* (PTPs). Each of these nodes occupy a single physical page.

Both L1 and L2 page tables are arrays of 1024 four-byte *entries*.

Each CPU has a *current level-1 page table* register, which holds the physical address of the currently active level-1 page table page.

To look up the physical address corresponding to virtual address *va*, the processor's memory management unit does the following steps. This version is incomplete because we are ignoring flags.

1. Split *va* into its components (PAGEOFFSET, L2PAGEINDEX, L1PAGEINDEX).
2. Let *l1pt* be the current level-1 page table register.
3. Let *l1pte* = *l1pt*->entry[L1PAGEINDEX(*va*)]. That is, look up the entry at index L1PAGEINDEX(*va*) in *l1pt* and call it *l1pte*.
4. Treat *l1pte* as the physical address of a page table page called *pagetable* (a level-2 node).
5. Let *l2pte* = *pagetable*->entry[L2PAGEINDEX(*va*)]. That is, look up the entry at index L2PAGEINDEX(*va*) in *pagetable* and call it *l2pte*.
6. Treat *l2pte* as the physical address of a data page called *page*.

7. The answer is physical address *page* + PAGEOFFSET(*va*).

In C pseudocode, we might write it this way. Again, this version is incomplete because we are ignoring flags.

```
typedef struct data_page {
    uint8_t data[4096];
} data_page;

typedef struct pseudo_l2ptp {
    data_page *entry[1024];
} pseudo_ptp;

typedef struct pseudo_l1ptp {
    pseudo_l2ptp *entry[1024];
} pseudo_l1ptp;

uint8_t *pseudo_virtual_memory_translate(uint32_t va, pseudo_l1ptp *l1ptp) {
    unsigned L1PAGEINDEX = va >> 22;
    pseudo_ptp *pagetable = l1ptp->entry[L1PAGEINDEX];

    unsigned L2PAGEINDEX = (va >> 12) & 0x3FF;
    data_page *page = pagetable->entry[L2PAGEINDEX];

    unsigned PAGEOFFSET = va & 0xFFF;
    return &page->data[PAGEOFFSET];
}
```

Flags

The lower 12 bits of every entry are used for flags. These flags determine whether pages exist and set access permissions.

The flags are:

`PTE_P == 1`

Present. If set, this virtual page is present in physical memory. If not set, the page

doesn't exist (and the rest of the entry isn't used).

PTE_W == 2

Writable. If set, this virtual page may be written. If not set, any attempt, by the kernel or by a process, to write to this virtual page will cause a page fault.

PTE_U == 4

User accessible. If set, this virtual page may be accessed by unprivileged user processes. If not set, any attempt by a process to read or write this virtual page will cause a page fault. The kernel can still use the page.

The combination of these flags has value `0x7`. Remember this value.

With flags, the lookup procedure is as follows.

1. Split *va* into its components (PAGEOFFSET, L2PAGEINDEX, L1PAGEINDEX).
2. Let *l1_pt* be the current level-1 page table.
3. Let *l1_pte*= *l1_pt*->entry[L1PAGEINDEX(*va*)]. That is, look up the entry at index L1PAGEINDEX(*va*) in *l1_pt* and call it *l1_pte.*
4. If (*l1_pte* & `PTE_P`) == 0, the address is not present. Induce a page fault exception.*
5. Otherwise, clear the lowest 12 bits of *l1_pte* and treat the result as a physical address of a page table page called *pagetable* (a level-1 node).
6. Let *pte* = *pagetable*->entry[L2PAGEINDEX(*va*)]. That is, look up the entry at index L2PAGEINDEX(*va*) in *pagetable* and call it *pte*.
7. If (*pte* & `PTE_P`) == 0, the address is not present. Induce a page fault exception.
8. If (*pte* & `PTE_W`) == 0 and the lookup is for a write, the access is not allowed. Induce a page fault exception.
9. If (*pte* & `PTE_U`) == 0 and the lookup is for a process (rather than the

kernel), the access is not allowed. Induce a page fault exception.

10. Otherwise, clear the lowest 12 bits of *pte* and treat the result as the physical address of a data page called *page*.
11. The answer is physical address *page* + PAGEOFFSET(*va*).

*(Actually, `PTE_W` and `PTE_U` checks also occur on the *l1_pte* too.)

Again in C pseudocode:

```
typedef struct data_page {
    uint8_t data[4096];
} data_page;

typedef struct pseudo_l2ptp {
    uint32_t entry[1024];
} pseudo_ptp;

typedef struct pseudo_l1ptp {
    uint32_t entry[1024];
} pseudo_l1ptp;

uint8_t *pseudo_virtual_memory_translate(uint32_t va,
pseudo_l1_page_table_page *l1_pagetable) {
    unsigned L1PAGEINDEX = va >> 22;
    uint32_t l1_pte = l1_pagetable->entry[L1PAGEINDEX];
    if (!(l1_pte & PTE_P))
        return NULL;
    pseudo_ptp *pagetable = (pseudo_ptp *)(l1_pte & 0xFFFFF000); // ==
PTE_ADDR(l1_pte)

    unsigned L2PAGEINDEX = (va >> 12) & 0x3F;
    uint32_t pte = pagetable->entry[L2PAGEINDEX];
    if (!(pte & PTE_P))
        return NULL;
    data_page *page = (data_page *)(pte & 0xFFFFF000);

    unsigned PAGEOFFSET = va & 0xFFF;
```

```
    return &page->data[PAGEOFFSET];
}
```

Example

Assume that the contents of physical memory are as follows. The “Contents” are 4-byte integers. The “Index”es treat the page as an array of four-byte integers, so “Physical Address” always equals “Physical Page + 4 × Index.” Contents not given are 0.

Physical Page	Index	(Physical Address)	Contents
0x1000	0	(0x1000)	0x2007
0x2000	0	(0x2000)	0x3007
0x3000	0	(0x3000)	1
	1	(0x3004)	2
	2	(0x3008)	3

Assume that the current L1 page table has physical address 0x1000.

Q1. Which physical address corresponds to virtual address 0x0?

A1. The physical address 0x3000. Here PAGEOFFSET = L2PAGEINDEX = L1PAGEINDEX = 0. Index 0 (L1PAGEINDEX) in the L1 PTP holds 0x2007. Mask away the flags and the PTP is at 0x2000. Index 0 (L2PAGEINDEX) in the PTP holds 0x3007. Mask away the flags and the data page is at 0x3000.

Q2. How many different physical pages are accessible through this L1 page table?

A2. Exactly one, the physical page at address 0x3000.

Q3. How many virtual addresses are accessible (a byte load from that address will not cause a page fault)?

A3. 4096: one page’s worth.

Q4. What will happen if the kernel attempts to access *virtual address* 0x3000?

A4. A page fault. Even the kernel is subject to virtual memory translation.

Example Questions

Now assume the same memory contents, but that the current L1 page table has physical address **0x2000**.

Q5. Which physical address corresponds to virtual address 0x0?

Q6. Which physical address corresponds to virtual address 0x100?

Q7. Which physical address corresponds to virtual address 0x1050?

Q8. Which physical address corresponds to virtual address 0x2040?

Q9. How many different physical data pages are accessible through this L1 page table?

Q10. How many virtual addresses are accessible (a byte load from that address will not cause a page fault)?

Scroll down when you're ready for answers.

Answers

A5. 0x0. Index L1PAGEINDEX = 0 in the level-1 page table page holds 0x3007. Mask away the flags and look up index L2PAGEINDEX = 0 in the level-2 PTP: you'll get the value 1. The page is present (`PTE_P == 1`), so no fault. Its address is 0.

A6. 0x100.

A7. None (page fault). The entry at index L2PAGEINDEX = 1 in the level-2 PTP has value 2; `(2 & PTE_P) == 0` so the page is not present.

A8. 0x40. The entry at index L2PAGEINDEX = 2 in the level-2 PTP has value 3. This is present. It's not a problem that it maps to the same physical page as L2PAGEINDEX = 0!

A9. Exactly one, the physical page at address 0.

A10. 8192. Virtual addresses 0–0xFFF and 0x2000–0x2FFF are all accessible. It's true that only 4096 *physical* addresses are accessible, but each pa is accessible from two different vas.

Exercises

Consider this different memory map.

Physical Page	Index	(Physical Address)	Contents
0x1000	0	(0x1000)	0x7007
	1	(0x1004)	0x8007
0x2000	0	(0x2000)	0x1007
	3	(0x200C)	0x3007
0x3000	128	(0x3200)	0xA007
0x6000	0	(0x6000)	0x2007
	1	(0x6004)	0x8007
0x7000	0	(0x7000)	0x9007
0x8000	0	(0x8000)	0x2007
	1	(0x8004)	0x2007
	2	(0x8008)	0x2007
	3	(0x800C)	0x2007

Part C. Assume the current level-1 page table has physical address 0x2000. For each virtual address, give the corresponding physical address (or FAULT if accessing the

address would cause a fault).

1. 0x00000000 0x7000
2. 0x00000001 0x7001
3. 0x00000FFF 0x7FFF
4. 0x00001000 0x8000
5. 0x00C08003 FAULT
6. 0x00C80003 0xA003
7. 0x00C00F00 FAULT
8. 0x00001F00 0x8F00

Part D. Assume the current level-1 page table has physical address 0x2000. For each *physical* address, give *all virtual* addresses that map to that physical address (or NONE if no virtual address maps to that physical address).

1. 0x00007000 0x00000000
2. 0x00000000 NONE
3. 0x0000AFFF 0x00C80FFF
4. 0x00008888 0x00001888
5. 0xA0000000 NONE
6. 0x00008001 0x00001001
7. 0x0000A002 0x00C80002

Part E. Assume the current level-1 page table has physical address 0x1000.

1. How many different physical data pages may be addressed using this level-1 page table? 2 (at 0x9000 and 0x2000)
2. How many accessible virtual addresses exist for this level-1 page table?

   5 x 2^12 == 0x5000

Part F. Assume the current level-1 page table has physical address 0x6000.

1. Give a physical address that is inaccessible from this level-1 page table

(has no corresponding virtual address). One example: 0x100000FF

2. Write C statements that, if executed under this L1 page table, would make the physical address you named accessible at some virtual address you choose. (Remember that all addresses used in C programs are virtual.)

   *(uint32_t *)0x00400004 = 0x10000007

3. Give the virtual address you chose for the last part.

   0x000010FF