

CS152  
Computer Architecture and Engineering  
Virtual Memory and Address Translation  
**SOLUTIONS**

*Assigned March 3*

Problem Set #3 Solutions

*Due March 12*

---

<http://inst.eecs.berkeley.edu/~cs152/sp09>

---

### **Problem 3.1: Virtual Memory Bits**

#### **Problem 3.1.A**

---

The answer depends on certain assumptions in the OS. Here we assume that the OS does everything that is reasonable to keep the TLB and page table coherent. Thus, any change that OS software makes is made to both the TLB and the page table.

However, the hardware can change the U bit (whenever a hit occurs this bit will be set) and the M bit (whenever a page is modified this bit will be set). Thus, these are the only bits that need to be written back. Note that the system will function correctly even if the U bit is not written back. In this case the performance would just decrease.

It is also important to note, that if the entry is laid out properly in memory, all the hardware-modified bits in the TLB can be written back to memory with a single memory write instruction. Thus it makes no difference whether one or two bits have been modified in the TLB, because writing back one bit or two bits still requires writing back a whole word.

#### **Problem 3.1.B**

---

An advantage of this scheme is that we do not need the TLB Entry Valid bit in the TLB anymore. One bit savings is not very much.

A disadvantage of this scheme is that the kernel needs to ensure that all TLB entries always are valid. During a context switch, all TLB entries would need to be restored (this is time-consuming). And, in general, whenever a TLB entry is invalidated, it will have to be replaced with another entry.

#### **Problem 3.1.C**

---

Changes to exceptions: “Page Table Entry Invalid” and “TLB Miss” exceptions are replaced with exceptions “TLB Entry Invalid” and “TLB No Match”

The TLB Entry Invalid exception will be raised if the VPN matches the TLB tag but the (combined) valid bit is false. When this exception is raised the kernel will need to consult the page table entry to see if this is a TLB miss (valid bit in page table entry is true), or an access of an invalid page table entry (valid bit in page table entry is false). Depending on what the cause of the exception was, it will then have to perform the necessary operations to recover.

The TLB No Match exception will be raised if the VPN does not match any of the TLB tags. If this exception is raised the kernel will do the same thing it did when a TLB Miss occurred in the previous design.

### **Problem 3.1.D**

---

When loading a page table entry into the TLB, the kernel will first check to see if the page table entry is valid or not. If it is valid, then the entry can safely be loaded into the TLB. If the page table entry is not valid, then the Page Table Entry Invalid exception handler needs to be called to create a valid entry before loading it into the TLB. Thus we only keep valid page table entries in the TLB. If a page table entry is to be invalidated, the TLB entry needs to be invalidated.

Changes to exceptions: Page Table Entry Invalid exception is not raised by the TLB anymore.

### **Problem 3.1.E**

---

The solution for Problem 3.1.C ends up taking two exceptions, if the PTE has the combined valid bit set to invalid. The first exception will be the TLB Miss exception, which will call a handler. The handler will load the corresponding PTE into the TLB and restart the instruction. The instruction will cause **another** exception right away, because the valid bit will be set to invalid. The exception will be the TLB Entry Invalid exception.

The solution for Problem 3.1.D will only take one exception, because the handler for Page Table Entry Invalid exception will get called by the TLB Miss handler. When the instruction that caused the exception is restarted, it will execute correctly, because the handler will have created a valid PTE and put it in the TLB.

Thus Bud Jet's solution in 3.1.D will be faster.

### **Problem 3.1.F**

---

Yes, the R bit can be removed in the same way we removed the V bit in 3.D. When loading a page table entry into the TLB we check if the data page is resident or not. If it is resident, we can write the entry into the TLB. If it is not resident, we go to the nonresident page handler, loading the page into memory before loading the entry into the

TLB. Thus, we only keep page table entries of resident pages in the TLB. In order to preserve this invariant, the kernel will have to invalidate the TLB entry corresponding to any page that gets swapped out. There's no performance penalty since the page was going to be loaded in from disk anyway to service the access that triggered the fault.

### **Problem 3.1.G**

---

The OS needs to check the permissions before loading the entry into the TLB. If permissions were violated, then the Protection Fault handler is called. Thus, we only keep page table entries of pages that the process has permissions to access.

### **Problem 3.1.H**

---

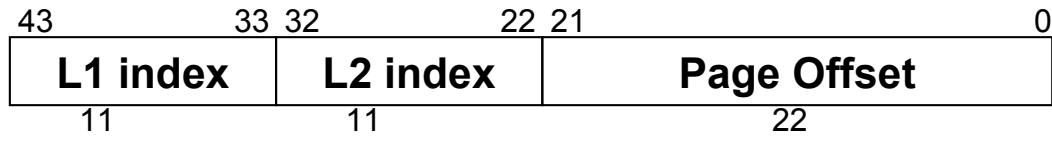
Whenever a page table entry is loaded into the TLB the U bit in the page table PTE can be set. Thus, we do not need the U bit in the TLB entry anymore.

Whenever a Write Fault happens (store and W bit is 0) the kernel will check the page table PTE to see if the W bit is set there. If it is not set the old Write Fault handler will be called. If the W bit is set, then the kernel will set the M bit in the page table PTE, set the W bit in the TLB entry to 1, and restart the store instruction. Thus, the M bit is not needed in the TLB either, and hence, TLB entries do not need to be written back to the page table anymore.

## Problem 3.2: Page Size and TLBs

### Problem 3.2.A

---



The L1 index and L2 index fields are the same, but the Page Offset field subsumes the L3 index and increases to 22 bits.

### Problem 3.2.B

### Page Table Overhead

---

$$PTO_{4KB} = \frac{16 \text{ KB} + 16 \text{ KB} + 8 \text{ KB}}{3 \text{ MB}} = \frac{40 \text{ KB}}{3 \text{ MB}} = 1.3\%$$

$$PTO_{4MB} = \frac{16 \text{ KB} + 16 \text{ KB}}{4 \text{ MB}} = \frac{32 \text{ KB}}{4 \text{ MB}} = 0.8\%$$

For the 4KB page mapping, one L3 table is sufficient to map the 768 pages since each contains 1024 PTEs. Thus, the page table consists of one L1 table (16KB), one L2 table (16KB), and one L3 table (8KB), for a total of 40 KB. The 768 4KB data pages consume exactly 3MB. The total overhead is 1.3%..

The page table for the 4MB page mapping, requires only one L1 table (16KB) and one L2 table (16KB), for a total of 32 KB. A single 4MB data pages is used, and the total overhead is 0.8%.

### Problem 3.2.C

### Page Fragmentation Overhead

---

$$PFO_{4KB} = \frac{0}{3 \text{ MB}} = 0\%$$

$$PFO_{4MB} = \frac{1 \text{ MB}}{3 \text{ MB}} = 33\%$$

With the 4KB page mapping, all 3MB of the allocated data is accessed. With the 4MB page mapping, only 3MB is accessed and 1MB is unused. The overhead is 33%.

### Problem 3.2.D

---

	Data TLB misses	Page table memory references (per miss)
4KB:	768	3
4MB:	1	2

The program sequentially accesses all the bytes in each page. With the 4KB page mapping, a TLB miss occurs each time a new page of the input or output data is accessed for the first time. Since the TLB has more than 3 entries (it has 64), there are no misses during the subsequent accesses within each page. The total number of misses is 768. With the 4MB page mapping, all of the input and output data is mapped using a single page, so only one TLB miss occurs.

For either page size, a TLB miss requires loading an L1 page table entry and then loading an L2 page table entry. The 4KB page mapping additionally requires loading an L3 page table entry.

### Problem 3.2.E

---

**1.01×**      10×      1,000×      1,000,000×

Although the 4KB page mapping incurs many more TLB misses, with either mapping the program executes 2M loads, 1M adds, and 1M stores (where  $M = 2^{20}$ ). With the 4MB mapping, the single TLB miss is essentially zero overhead. With the 4KB mapping, there is one TLB miss for every 4K loads or stores. Each TLB miss requires 3 page table memory references, so the overhead is less than 1 page table memory reference for every 1000 data memory references. Since the TLB misses likely cause additional overhead by disrupting the processor pipeline, a 1% slowdown is a reasonable but probably conservative estimate.

### Problem M3.3: 64-bit Virtual Memory

This problem examines page tables in the context of processors with a 64-bit addressing.

#### Problem 3.3.A

#### Single level page tables

---

12 bits are needed to represent the 4KB page. There are  $64-12=52$  bits in a VPN. Thus, there are  $2^{52}$  PTEs. Each is 8 bytes.  $2^{52} * 2^3 = 2^{55}$ , or 32 petabytes!

#### Problem 3.3.B

#### Let's be practical

---

$2^2$  segments \*  $2^{(44-12)}$  virtual pages =  $2^{34}$  PTEs.  $2^3$  (bytes/PTE) \*  $2^{34}$  PTEs =  $2^{37}$  bytes.

It is possible to interpret the question as there being 3 segments of  $2^{44}$  bytes. Thus we'd need:

$3$  segments \*  $2^{(44-12)}$  virtual pages =  $2^{33} + 2^{32}$  PTEs.  $2^3 * (2^{33} + 2^{32}) = 2^{36} + 2^{35}$  bytes.

#### Problem 3.3.C

#### Page table overhead

---

The smallest possible page table overhead occurs when all pages are resident in memory. In this case, the overhead is

$$8(2^{11} + 2^{11} * 2^{11} + 2^{11} * 2^{11} * 2^{10}) / 2^{44} \approx 2^{35} / 2^{44} \approx 1 / 2^9$$

The largest possible page table overhead occurs when only one data page is resident in memory. In this case, we need 1 L0 page table, 1 L1 page table, 1 L2 page table in order to get data page. Thus the overhead is:

$$8(2^{11} + 2^{11} + 2^{10}) / 2^{12} = 10$$

#### Problem 3.3.D

#### PTE Overhead

---

PPN is  $40-12=28$  bits.  $28+1+1+3=33$  bits.

There are 31 wasted bits in a 64 bit page table entry. It turns out that some of the "wasted" space is recovered by the OS to do bookkeeping, but not much.