Name: _____ Student ID:_____

CS 354 Practice Final Exam

Test topics:

- Virtual memory
  - What is virtual and physical memory
  - Why do we need virtual memory
  - Page-based virtual memory
    - Why pages?
    - Single-level page table
    - Multi-level page table
  - Swapping/present bits/virtual memory as a cache
  - TLBs—fully associative and set associative
- Dynamic memory allocators
  - Implicit vs explicit allocators
  - OS vs library separation
  - Internal and external fragmentation
  - Building a memory allocator
    - Track free blocks—implicit vs explicit free list
    - Choosing free blocks—first fit, best fit, next fit
    - Extra space in blocks we are allocating—splitting blocks
    - What to do when deallocating—coalescing blocks
  - Assignment
- Linking and Loading
  - Assembler
    - Absolute and relative/indirect addressing
  - Linker
  - Loader
- Networking
  - Client-server programming model
  - Hub, bridge, switch, router
  - 7 layer OSI model
  - Packets
  - DNS
  - Socket programming

Name: _____ Student ID:_____

Question 1: Virtual memory

Virtual Memory and Paging

Virtual memory is a key aspect of modern computing systems. In this question, we'll explore how a simple VMworks, and discuss some of its downsides.

Assume you have the following **linear page table** (i.e., the simplest page table, that is, just an array of page-table entries) which is used to implement a virtual memory for a given process; each entry is 4-bytes in size and described further (as needed) below.

```
0x00000000
0x80000018
0x00000000
0x00000000
0x00000000
0x80000019
0x00000000
0x00000000
0x8000000d
0x80000000
0x00000000
0x80000007
0x8000001c
0x00000000
0x00000000
0x8000001e
```

The size of each page is **4KB**, and the entire virtual address space is **64KB** in size.

1a How many pages are in a virtual address space?

1b How many page table entries are there in a single page table? What is PTBR in the context of page tables and what does it contain?

Name: _____ Student ID:_____

1c Assume the following format of a page table entry: 1 bit which determines whether the page is valid or not, and the remaining bits are the PFN (physical frame number) of the translation. How many bytes in the address space defined by the page table above can the process access legally?

1d Assume the program accesses virtual address 0x8e73. Given the page table above, is this access legal? If so, what physical address does this translate to? (Show your work)

1e Assume the program accesses virtual address 0x4a2f. Given the page table above, is this access legal? If so, what physical address does this translate to? (Show your work)

1f Assuming the page table above, what bad thing happens when the following C code is executed? Which line of C causes this bad thing to happen? Explain.

```
int *p;
int x;
p = NULL;
x = *p;
```

1g Assume you can change what the pointer `p` is set to from NULL to some other value. What values could you set `p` to in order to avoid any problems while running the above code snippet?

Name: _____ Student ID:_____

1h Assume the page table above and the following assembly code sequence. What bad thing happens when this code executes? Which line of assembly causes this bad thing to happen? Explain.

```
mov $100, %eax
mov (%eax), %ebx
```

1i Assume you could change one value in the PAGE TABLE above to ensure the assembly code runs without that bad thing happening. What would you change in the page table? Would other bad things happen as a result?

1j (the downside) Overall, virtual memory seems to be useful for giving a process (a running program) the illusion that it has its own private memory. But virtual memory also has negatives. What are they? Given that the negatives exist, should we still use virtual memory?

Name: _____ Student ID:_____

Question 2: TLBs

The Translation Lookaside Buffer (TLB)

The TLB is a special cache used to help implement virtual memory efficiently. In this question, we'll explore how the TLB works and then discuss some of the downsides.

Assume we have a system with 4KB pages and a 32-entry TLB that is fully associative. Let's also assume we have the following array that is accessed frequently by the running program, say in a loop:

```
int m[SIZE];
```

We say that a data structure is "covered" by the TLB if, when accessing it frequently, the total number of pages that the data structure resides upon is less than the number of entries in the TLB; that is, if there is little other memory traffic on-going, each access to the data structure in question will likely yield a TLB hit.

2a How big can `SIZE` be before the array `m` is not "covered" by the TLB?

Imagine we now have the following loop which accesses the array:

```
int m[SIZE];
int i, tmp = 0;
for (i = 0; i < SIZE: i++) {
    tmp += m[i];
}
```

2b How many references to the TLB will such a loop yield? (Don't forget about instructions!) Make any assumptions you need to, such as loop variable `i` and the counter `tmp` are likely held in registers and so forth.

2c Now, assume that `SIZE` is so big that the array will not be "covered" by the TLB. Thus, as we repeatedly run the code above, it will likely generate a number of TLB misses; how many TLB misses will one run through this entire code sequence generate? How many hits?

2d Assume a 32-bit virtual address and 4-KB page size. Now assume that the array `m` was located at virtual address `0x40000000`. Assume that the page tablemaps the part of the address space holding `m` to a contiguous set of physical pages, starting at `0x10000000`. Finally, assume that the page table is a linear (array) page table, and is located at physical memory location 0x20000000. If SIZE is set to 4096, and all accesses to `m` are TLB misses (i.e., it is the first time we ever run this piece of code), which physical memory locations will be accessed during one run through this loop? (For simplicity, you can ignore references caused by instruction fetches)

2e (the downside) TLBS are not perfect. One place they achieve imperfection is in their replacement policy; when making room for a new translation, the TLB must kick out an old one. Assume the TLB uses a policy known as "least recently used" (LRU) to kick out an old translation, i.e., the TLB keeps track of when each entry is accessed and kicks out the oldest one when it needs to put a new translation in. When does an approach such as LRU work really poorly?

Name: _____ Student ID:_____

Question 3: Webservers and Networking

Web servers are a critical part of the infrastructure of the modern world. In this question, we'll explore some of the aspects of how a web server works. Later questions try to test your understanding of Networking in general.

3a A typical web server first calls `socket()`, `bind()`, and `listen()` before entering into a loop in which it calls `accept()`. What is the approximate role of these calls in the web server's operation?

3b Web servers are built on top of a simple protocol known as HTTP. Describe the basics of an HTTP get request; What does the client send to the server? What does the server send back? Assume HTTP 1.0 (but you can also elaborate about 1.1 if you'd rather).

3c Name 3 of the layers in the OSI model.

3d Explain what static and dynamic content mean in the context of web servers.

3e Explain one difference between all of these: **Hub, Bridge, Switch, Router**

Name: _____ Student ID:_____

Question 5: Linking and Loading

5a Explain why generating position independent code is useful for external procedure calls in shared libraries? Name one table like data structure that is used to facilitate position independent code generation.

5b Explain briefly in one or two lines what does loading an executable object file mean?

5c What is the major difference between an ELF relocatable object file and an ELF executable object file.

5d What are the different kinds of symbols in the context of a linker?

5e What the two phases of Relocation?

5f Name two real world applications where loading shared libraries dynamically from applications is useful.

5g Name any two unix tools used to manipulate object files.

Name: _____ Student ID:_____

Solve the practice problem 9.4 and homework problems 9.11 , 9.12, 9.13 below.

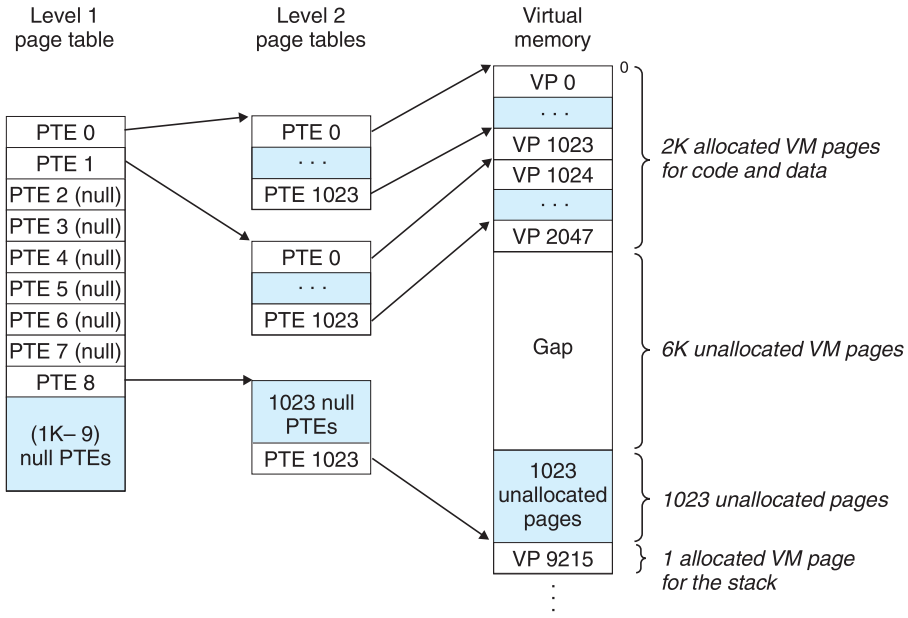


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

paged in and out by the VM system as they are needed, which reduces pressure on main memory. Only the most heavily used level 2 page tables need to be cached in main memory.

Figure 9.18 summarizes address translation with a k-level page table hierarchy. The virtual address is partitioned into k VPNs and a VPO. Each VPN i, $1 \leq i \leq k$, is an index into a page table at level i. Each PTE in a level-j table, $1 \leq j \leq k-1$, points to the base of some page table at level $j+1$. Each PTE in a level-k table contains either the PPN of some physical page or the address of a disk block. To construct the physical address, the MMU must access k PTEs before it can determine the PPN. As with a single-level hierarchy, the PPO is identical to the VPO.

Accessing k PTEs may seem expensive and impractical at first glance. However, the TLB comes to the rescue here by caching PTEs from the page tables at the different levels. In practice, address translation with multi-level page tables is not significantly slower than with single-level page tables.

9.6.4 Putting It Together: End-to-end Address Translation

In this section, we put it all together with a concrete example of end-to-end address translation on a small system with a TLB and L1 d-cache. To keep things manageable, we make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).

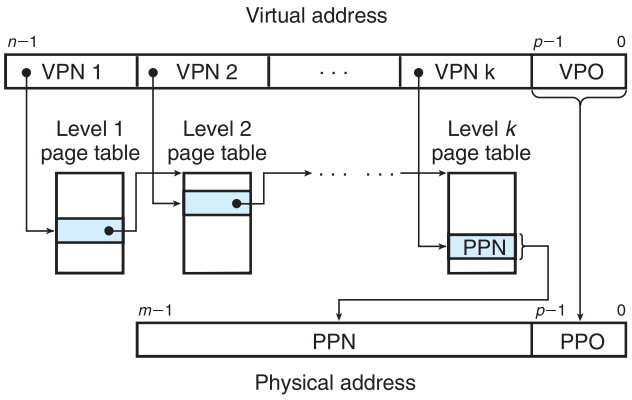


Figure 9.18 **Address translation with a k-level page table.**

- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes ($P = 64$).
- The TLB is four-way set associative with 16 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total sets.

Figure 9.19 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order 6 bits of the virtual and physical addresses serve as the VPO and PPO respectively. The high-order 8 bits of the virtual address serve as the VPN. The high-order 6 bits of the physical address serve as the PPN.

Figure 9.20 shows a snapshot of our little memory system, including the TLB (Figure 9.20(a)), a portion of the page table (Figure 9.20(b)), and the L1 cache (Figure 9.20(c)). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware as it accesses these devices.

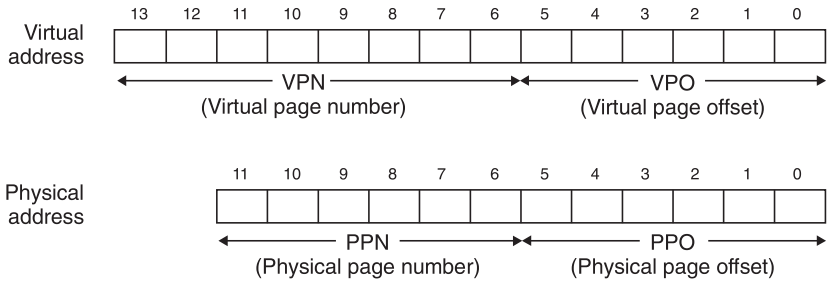
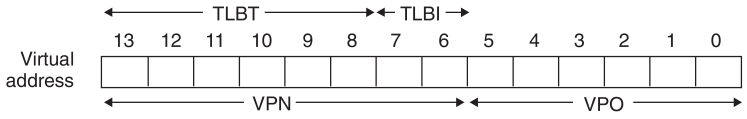


Figure 9.19 **Addressing for small memory system.** Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).



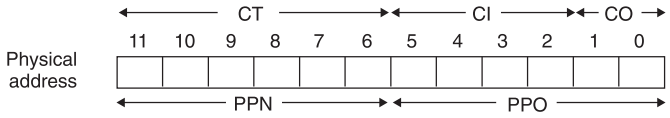
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

(a) TLB: Four sets, 16 entries, four-way set associative

VPN	PPN	Valid
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

(b) Page table: Only the first 16 PTEs are shown



Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

(c) Cache: Sixteen sets, 4-byte blocks, direct mapped

Figure 9.20 **TLB, page table, and cache for small memory system.** All values in the TLB, page table, and cache are in hexadecimal notation.

- *TLB*: The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the 2 low-order bits of the VPN serve as the set index (TLBI). The remaining 6 high-order bits serve as the tag (TLBT) that distinguishes the different VPNs that might map to the same TLB set.
- *Page table.* The page table is a single-level design with a total of $2^8 = 256$ page table entries (PTEs). However, we are only interested in the first sixteen of these. For convenience, we have labeled each PTE with the VPN that indexes it; but keep in mind that these VPNs are not part of the page table and not stored in memory. Also, notice that the PPN of each invalid PTE is denoted with a dash to reinforce the idea that whatever bit values might happen to be stored there are not meaningful.
- *Cache.* The direct-mapped cache is addressed by the fields in the physical address. Since each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset (CO). Since there are 16 sets, the next 4 bits serve as the set index (CI). The remaining 6 bits serve as the tag (CT).

Given this initial setup, let's see what happens when the CPU executes a load instruction that reads the byte at address `0x03d4`. (Recall that our hypothetical CPU reads one-byte words rather than four-byte words.) To begin this kind of manual simulation, we find it helpful to write down the bits in the virtual address, identify the various fields we will need, and determine their hex values. The hardware performs a similar task when it decodes the address.

	TLBT						TLBI							
	0x03						0x03							
bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

To begin, the MMU extracts the VPN (`0x0F`) from the virtual address and checks with the TLB to see if it has cached a copy of PTE `0x0F` from some previous memory reference. The TLB extracts the TLB index (`0x03`) and the TLB tag (`0x3`) from the VPN, hits on a valid match in the second entry of Set `0x3`, and returns the cached PPN (`0x0D`) to the MMU.

If the TLB had missed, then the MMU would need to fetch the PTE from main memory. However, in this case we got lucky and had a TLB hit. The MMU now has everything it needs to form the physical address. It does this by concatenating the PPN (`0x0D`) from the PTE with the VPO (`0x14`) from the virtual address, which forms the physical address (`0x354`).

Next, the MMU sends the physical address to the cache, which extracts the cache offset CO (`0x0`), the cache set index CI (`0x5`), and the cache tag CT (`0x0D`) from the physical address.

	CT						CI				CO	
	0x0d						0x05				0x0	
bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

Since the tag in Set 0x5 matches CT, the cache detects a hit, reads out the data byte (0x36) at offset CO, and returns it to the MMU, which then passes it back to the CPU.

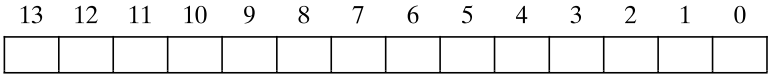
Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another possibility is that the PTE is valid, but the necessary memory block misses in the cache.

Practice Problem 9.4

Show how the example memory system in Section 9.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned." If there is a page fault, enter "–" for "PPN" and leave parts C and D blank.

Virtual address: 0x03d7

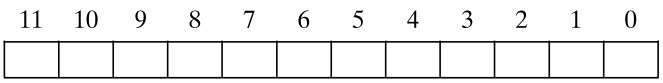
A. Virtual address format



B. Address translation

Parameter	Value
VPN	_____
TLB index	_____
TLB tag	_____
TLB hit? (Y/N)	_____
Page fault? (Y/N)	_____
PPN	_____

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	_____
Cache index	_____
Cache tag	_____
Cache hit? (Y/N)	_____
Cache byte returned	_____

9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. The Core i7 is based on the Nehalem microarchitecture. Although the Nehalem design allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatability mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* includes four cores, a large L3 cache shared by all of the cores, and a

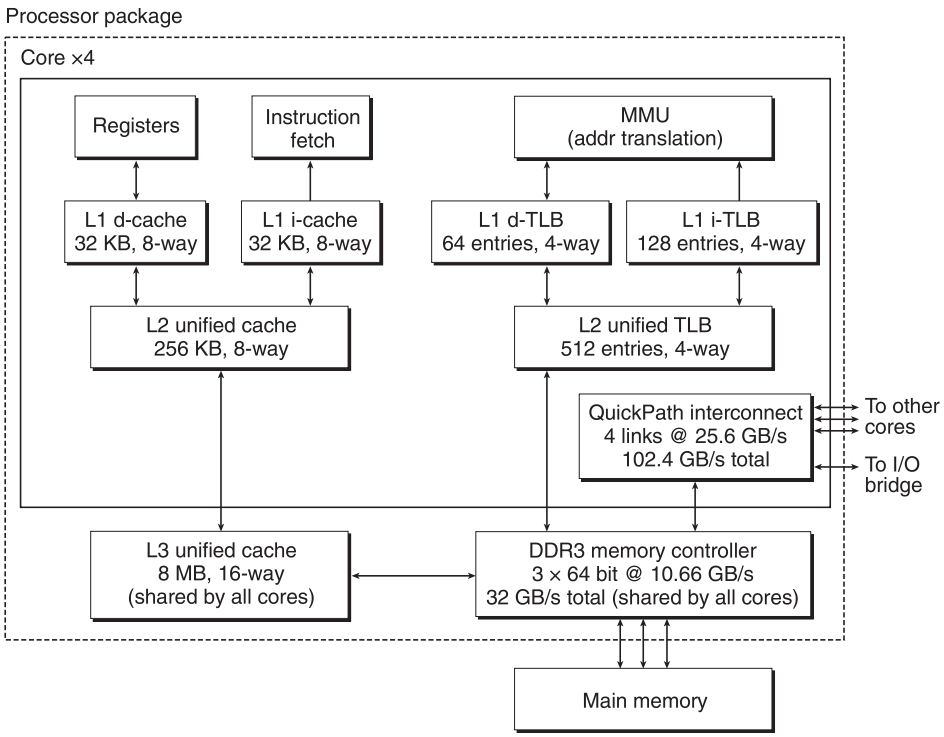


Figure 9.21 **The Core i7 memory system.**

scription of the Linux virtual memory system. Intel Corporation provides detailed documentation on 32-bit and 64-bit address translation on IA processors [30].

Knuth wrote the classic work on storage allocation in 1968 [61]. Since that time there has been a tremendous amount of work in the area. Wilson, Johnstone, Neely, and Boles have written a beautiful survey and performance evaluation of explicit allocators [117]. The general comments in this book about the throughput and utilization of different allocator strategies are paraphrased from their survey. Jones and Lins provide a comprehensive survey of garbage collection [54]. Kernighan and Ritchie [58] show the complete code for a simple allocator based on an explicit free list with a block size and successor pointer in each free block. The code is interesting in that it uses unions to eliminate a lot of the complicated pointer arithmetic, but at the expense of a linear-time (rather than constant-time) free operation.

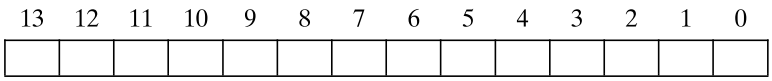
Homework Problems

9.11 ◆

In the following series of problems, you are to show how the example memory system in Section 9.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache Byte returned." If there is a page fault, enter "–" for "PPN" and leave parts C and D blank.

Virtual address: `0x027c`

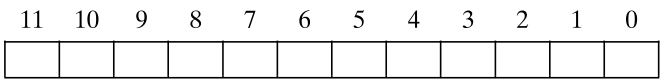
A. Virtual address format



B. Address translation

Parameter	Value
VPN	_____
TLB index	_____
TLB tag	_____
TLB hit? (Y/N)	_____
Page fault? (Y/N)	_____
PPN	_____

C. Physical address format



D. Physical memory reference

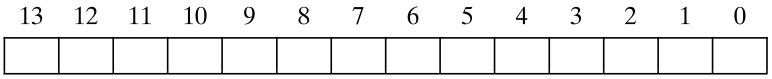
Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

9.12 ◆

Repeat Problem 9.11 for the following address:

Virtual address: `0x03a9`

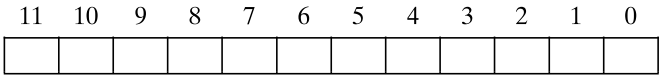
A. Virtual address format



B. Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit? (Y/N)	
Page fault? (Y/N)	
PPN	

C. Physical address format



D. Physical memory reference

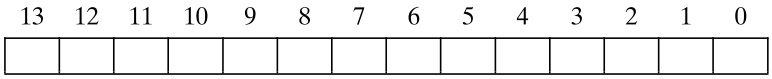
Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

9.13 ◆

Repeat Problem 9.11 for the following address:

Virtual address: `0x0040`

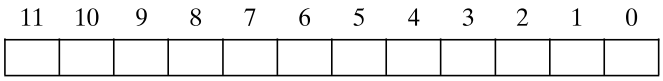
A. Virtual address format



B. Address translation

Parameter	Value
VPN	_____
TLB index	_____
TLB tag	_____
TLB hit? (Y/N)	_____
Page fault? (Y/N)	_____
PPN	_____

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	_____
Cache index	_____
Cache tag	_____
Cache hit? (Y/N)	_____
Cache byte returned	_____

9.14 ◆◆

Given an input file `hello.txt` that consists of the string "`Hello, world!\n`", write a C program that uses `mmap` to change the contents of `hello.txt` to "`Jello, world!\n`".

9.15 ◆

Determine the block sizes and header values that would result from the following sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment, and uses an implicit free list with the block format from Figure 9.35. (2) Block sizes are rounded up to the nearest multiple of 8 bytes.