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.

```
low addr 0  0x00000000  <—— PTBR
         1  0x80000018
         2  0x00000000
         3  0x00000000
         4  0x00000000
         5  0x80000019
         6  0x00000000
         7  0x00000000
         8  0x8000000d
            0x80000000
            0x00000000
            0x80000007
            0x8000001c
            0x00000000
            0x00000000
high addr   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?

64kB / 4kB = 16 pages

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?

16 entries

page table base register (CR3) → address of start (base) of page table

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?

7 pages · 4kB = 28kB

0x1000

0x0E73 0x06

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)

12-bit page offset (4kB pages) 0xE73

VPN -> 0x8

PPN -> D -> 0xDE73

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)

~~Some~~ no VPN -> 4

invalid

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.

```
1 int *p;
2 int x;
3 p = NULL;
4 x = *p;
```

dereferences null -> looks @ VPN 0

Seg fault! or page fault

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?

any valid pages (0x8E73)

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
```

VPN -> 0

page fault (seg fault)

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?

Change 0th entry

Null would no longer be an invalid pointer

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?

Slow, high energy, high power

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];
```

→ TLB reach

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];
}
```

TLB reach
→ max TLB cover

$32 \cdot 4kB = 128kB$

$\frac{128kB}{sizeof(int)} = \frac{128kB}{4} = 32K$ entries $= 32 \cdot 1024 = 32768$ (2^{15})

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.

Data accesses: 1 iteration: 1 access to TLB
all iters: 32768 accesses

Inst. accesses: 1 iteration: $3 \cdot 32768$ for all
assume ~~Possible~~: 3 because 3 insts in loop
5-7

Name: ______________________ Student ID:______________

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?

1 miss per 2^{10} accesses

↳ 4kB → $4B \cdot 2^{10} = 4kB$

$SIZE / 2^{10}$ = misses per loop execution

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)

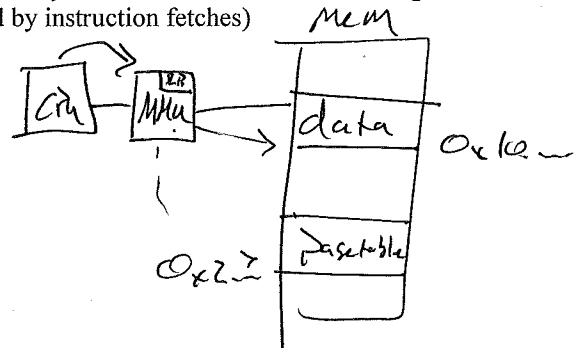
1) 0x20...

2) 0x10---

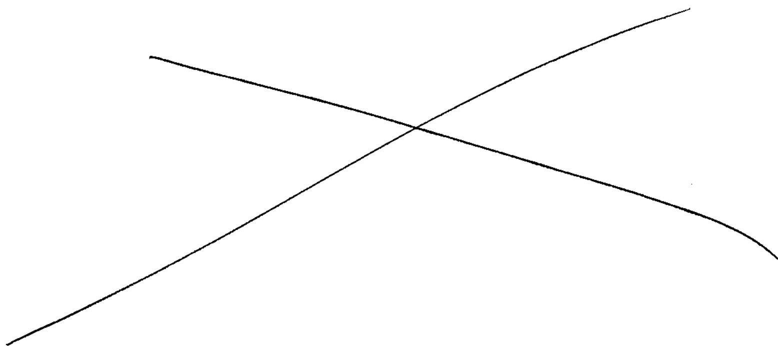
3) 0x10---4

4) 0x10-8

⋮



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: ______________

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?

7 pages · 4kB = 28kB

0x1000

0x0E73 0x06

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)

12-bit page offset (4kB pages) 0xE73

VPN → 0x8

PPN → D → 0xDE73

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)

no VPN → 4

invalid

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.

```
1 int *p;
2 int x;
3 p = NULL;
4 x = *p;
```

dereferences null → looks @ VPN 0

Seg fault! or page fault

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?

any valid pages (0x8E73)

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
```

VPN -> 0

page fault (seg fault)

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?

Change 0th entry

Null would no longer be an invalid pointer

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?

Slow, high energy, high power

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];
```

→ TLB reach

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];
}
```

TLB reach
→ max TLB cover

$32 \cdot 4KB = 128KB$

$\frac{128kB}{sizeof(int)} = \frac{128kB}{4} = 32K$ entries $= 32 \cdot 1024 = 32768$ (2^{15})

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.

Data accesses: 1 iteration: 1 access to TLB

all iters: 32768 accesses

Inst. accesses: 1 iteration: $3 \cdot 32768$ for all

~~Possible~~ assume: 3 because 3 insts in loop

5-7

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?

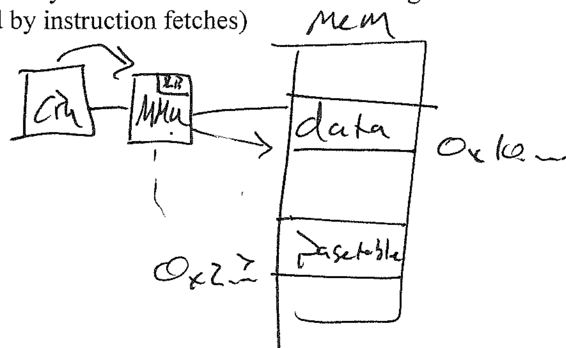
1 miss per 2^{10} accesses

↳ 4kB → $4B \cdot 2^{10} = 4kB$

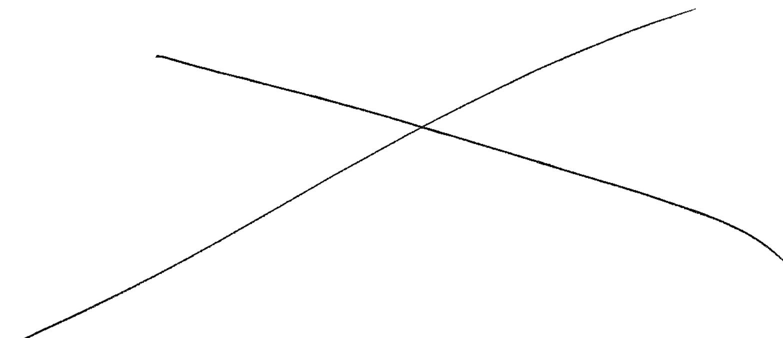
$SIZE / 2^{10}$ = misses per loop execution

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)

1) 0x20...
2) 0x10----
3) 0x10---4
4) 0x10.--8
⋮



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?

Socket -> cretes socket descriptor (fd for socket)
bind -> binds socket to port/IP address
listen -> listens for client connection
accept -> accepts connection + creates new fd for connection

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).

Client -> GET <path> <version> GET / http/1.1

Server -> <version> <code> } headers http/1.0 200 ok
⋮ data ⋮ <html>

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

Application
presentation
session
transport
network
data link <-
physical

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

Static returns file itself -> same for every one
dynamic -> dynamically creates file -> can be unique for each visitor

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

hub -> broadcasts to all ports @ phys layer
Bridge/Switch -> sends messages only to needed ports @ data-link layer
router -> routes between networks @ Networ layer

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.

PIC code → code that can be loaded into any memory location

Uses global offset table to look up addresses. Absolute addresses left out

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

Copy object (code + data) into memory
set up stack + heap
jumps to start of code

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

relocatable doesn't have abs. addresses
compiled w/ PIC

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

global / local

strong / weak

5e What the two phases of Relocation?

1) relocating sections + symbol defines
2) relocating symbol refs within sections

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

dynamically choosing libs
Saving DRAM
Saving disk space

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

Objdump readelf
ldd

→ binutils

info

	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.

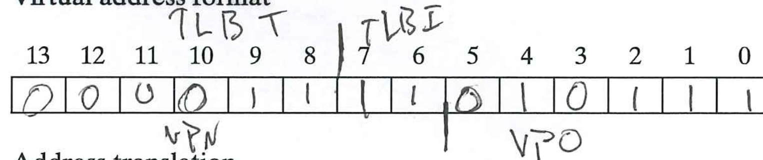
Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another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

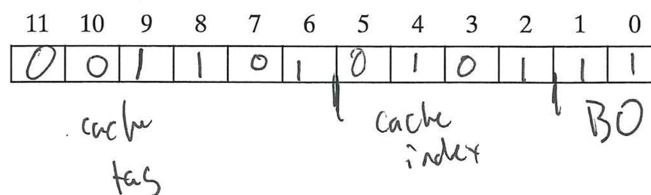
A. Virtual address format



B. Address translation

Parameter	Value
VPN	0xOF
TLB index	0x3
TLB tag	0x03
TLB hit? (Y/N)	Y
Page fault? (Y/N)	N
PPN	0xOD

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	0x3
Cache index	0x5
Cache tag	0x0D
Cache hit? (Y/N)	Y
Cache byte returned	0x1D

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

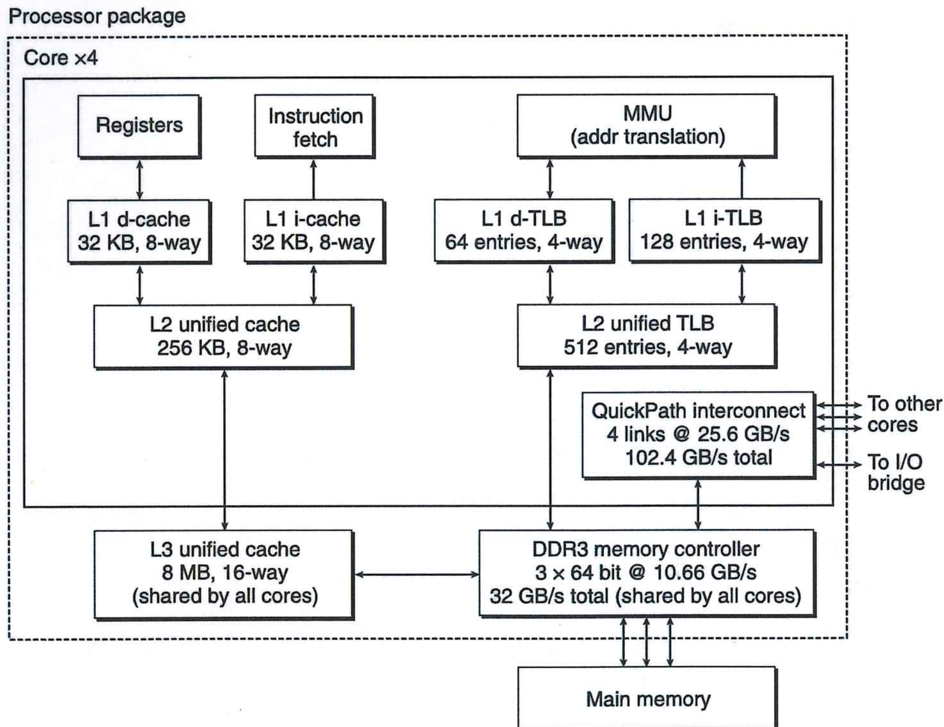


Figure 9.21 **The Core i7 memory system.**

D. Physical memory reference

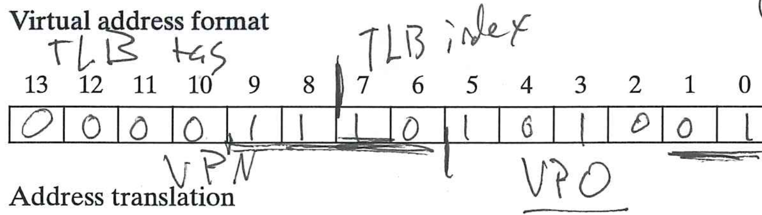
Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

9.12 ◆

Repeat Problem 9.11 for the following address:

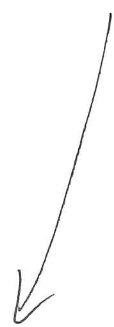
Virtual address: 0x03a9

A. Virtual address format

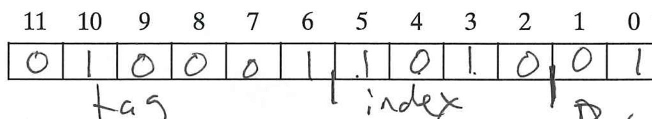


B. Address translation

Parameter	Value
VPN	0x0E
TLB index	0x2
TLB tag	0x03
TLB hit? (Y/N)	N
Page fault? (Y/N)	N
PPN	0x11



C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	0x1
Cache index	0xA
Cache tag	0x11
Cache hit? (Y/N)	N
Cache byte returned	—

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.

```
0x0000000c
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?

64KB / 4KB = 16 pages

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?

16 entries

↳ CR3

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;
```

seg fault / page fault / nullptr error

→ VPN = 0

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?

12-bits

0x1000

0x12AC

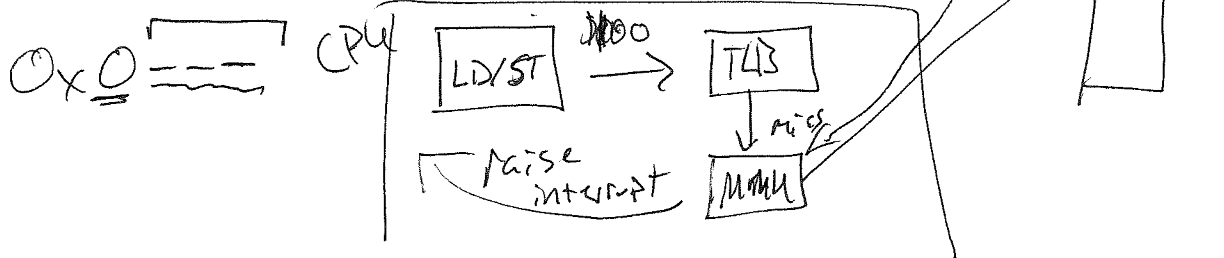
Name: ______________________ Student ID: ______________

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.

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

page table

VPN -> 0 -> page fault / seg fault



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?

Overwrite data from another process

null is null -> null maps to valid page

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];
```

→ reach

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];
}
```

reach? → $32 \cdot 4KB = 128KB$

$$\frac{128kB}{sizeof(int)} = \frac{128kB}{4B} = 32,768 = SIZE$$

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.

Single iteration: 1 access (data)

instructions?: 7 accesses

7

8 accesses

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.

PIC -> works no matter what address it is loaded to

dynamically linked objects or shared objects global addresses must be dynamically looked up in.... offset table GOT

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

- copy object (code + data) into memory
- set up memory (allocating stack + heap)
- jumps to beginning of code

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

doesn't have abs/rel addresses ↳ PIC

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

Global / local -> local - used only in local object
global - used from any object
Strong / weak -> Strong is initialized or function, ~~local~~ weak otherwise

5e What the two phases of Relocation?

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

Save memory if apps share
Not re do work
Updates easy since you can dynamically choose

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

Objdump
ldd
readelf
into binutils