Andrew ID: **SOLUTION**

Full Name: **SOLUTION**

Hint: This is an old school handwritten exam. There is no authenticated login. If we can't read your AndrewID, we won't easily know who should get credit for this exam. If we can't read either your AndrewID or Full Name, we're in real bind. Please write neatly :-)

18-213/18-613 Final Exam

Spring 2025

Instructions:
- Make sure that your exam is not missing any sheets (check page numbers at bottom)
- Write your Andrew ID and full name on this page (and we suggest on each and every page)
- This exam is closed book and closed notes.
- You may not use anything other than what we provide, except writing implements, such as pens and pencils, and a simple arithmetic calculator.
- Write your answers in the space provided for the problem.
- If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The point value of each problem is indicated.
- **Good luck!**

Problem #	Scope	Max Points	Score
1	Data Representation: "Simple" Scalars: Ints and Floats	10	
2	Data Representation: Arrays, Structs, Unions, and Alignment	10	
3	Assembly, Stack Discipline, Calling Convention, and x86-64 ISA	15	
4	Caching, Locality, Memory Hierarchy, Effective Access Time	15	
5	Malloc(), Free(), and User-Level Memory Allocation	10	
6	Virtual Memory, Paging, and the TLB	15	
7	Process Representation and Lifecycle + Signals and Files	10	
8	Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW	15	
TOTAL	***Total points across all problems***	***100***	

Question 1: Representation: “Simple” Scalars (10 points)

Part A: Integers (5 points, 1 point per blank)

Fill in the five empty boxes in the table below when possible and indicate “UNABLE” when impossible.

	Machine 1: 6-bit w/2s complement signed	**Machine 2: 6-bit unsigned**
Binary representation of -28 decimal	**10 0100**	**UNABLE**
Decimal value of 101010		**42**
Decimal value of (-24 - 10)	**30**	
Binary representation of Tmin (*Hint: It’s negative)*	**100000**	

Continued on next page.

Part B: Floats (5 points, 1 point per blank)

For this problem, please consider a floating point number representation based upon an IEEE-like floating point format as described below.

- Format A:
  - There are 6 bits
  - There are 0 (zero, none) sign bits.
  - There are k = 3 exponent bits.
  - You need to determine the number of fraction bits.

✔ Fill in the empty (and not grayed-out) boxes as instructed.

✔ When decimal values are requested, reduced fractions are okay.

✔ Should rounding be required, "round even"

	Format
Total Number of Bits (Decimal)	**6**
Number of Sign Bits (Decimal)	****** 0 ******
Number of Exponent Bits (Decimal)	**3**
Number of Fraction Bits (Decimal)	**3** (No points, just for you)
Bias (Decimal)	**3** (No points, just for you)
Largest number (Decimal value)	**15**
binary 000111 (Decimal)	**7/32**
Smallest distance between any two adjacent points on the number line (Decimal)	**1/32**
Largest distance between any two adjacent points on the number line, excluding special values (Decimal)	**1**
Infinity (Binary)	**111000**

Continued on next page.

Question 2: Representation: Arrays, Structs, Unions, Alignment, etc. (10 points)

Part A: Array Size and Layout (4 points)

Consider the following definition in an x86-64 system with 8-byte pointers and 2-byte `short`s:

Definition
`unsigned short numbers[4][3];`

2(A)(1) (2 point): How many bytes are allocated to `numbers`? (Write "UNKNOWN" if not knowable).
Hint: Think sizeof()

24 bytes

2(A)(2) (2 point): If the address of `numbers[2][1]` is `0x10000`, what is the address of numbers[3][2]?

0x10008

Part B : Structs and Alignment (6 points)

For this question please assume "Natural alignment", in other words, please assume that each type must be aligned to a multiple of its data type size.

Please consider the following struct:

```
struct {
  double d;      // 8-byte type
  char ca[2];
  int i;         // 4-byte type
  char c;        // 1-byte type
} partB;
```

Continued on next page.

Question 2: Representation: Arrays, Structs, Unions, Alignment, etc.(10 points), *cont.*

Part B : Structs and Alignment (6 points), *cont.*

2(B)(1) (2 point): What would you expect to be the value of the expression below?

`sizeof(struct partB)`

```
struct {
  double d;      // 8-byte type
  char ca[2];    // 2 bytes (2 * 1-byte type)
                 // 2-bytes padding
  int i;         // 4-byte type
  char c;        // 1-byte type
                 // 7-bytes padding
} partB;         // 24-bytes, total
```

24-bytes

2(B)(2) (2 points): Rewrite the struct above to minimize its size after alignment-mandated padding:

```
struct {
  char c;        // 1-byte
  char ca[2];    // 2-bytes
                 // 1-byte (padding)
  int i;         // 4-bytes
  double d;      // 8-byte type
} rewrite;       // 16-bytes, total
```

2(B)(3) (2 points): Consider the original definition we provided above and the definition of `arrayB3` given below, what is the distance, measured in bytes, between the address of `arrayB3[3].c` and the address of `arrayB3[5].d`?

```
struct partB arrayB3[10];
```

&arrayB3[4] - &arrayB3[3].c = 8 and sizeof(arrayB3[4]) = 24, so 8+24=32

Continued on next page

3. Assembly, Stack Discipline, Calling Convention, and x86-64 ISA (15 points)
Part A: Loops and Calling Convention (7 points)

Consider the following code:

```
(gdb) disassemble loop
Dump of assembler code for function loop:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     sub    $0x20,%rsp
   0x0000000000001155 <+12>:    mov    %edi,-0x14(%rbp)
   0x0000000000001158 <+15>:    mov    %esi,-0x18(%rbp)
   0x000000000000115b <+18>:    mov    %edx,-0x1c(%rbp)
   0x000000000000115e <+21>:    movl   $0x0,-0xc(%rbp)
   0x0000000000001165 <+28>:    jmp    0x11e3 <loop+154>
   0x0000000000001167 <+30>:    movl   $0x0,-0x8(%rbp)
   0x000000000000116e <+37>:    jmp    0x11ab <loop+98>
   0x0000000000001170 <+39>:    mov    -0x8(%rbp),%ecx
   0x0000000000001173 <+42>:    movslq %ecx,%rax
   0x0000000000001176 <+45>:    imul   $0x2aaaaaab,%rax,%rax
   0x000000000000117d <+52>:    shr    $0x20,%rax
   0x0000000000001181 <+56>:    mov    %ecx,%esi
   0x0000000000001183 <+58>:    sar    $0x1f,%esi
   0x0000000000001186 <+61>:    mov    %eax,%edx
   0x0000000000001188 <+63>:    sub    %esi,%edx
   0x000000000000118a <+65>:    mov    %edx,%eax
   0x000000000000118c <+67>:    add    %eax,%eax
   0x000000000000118e <+69>:    add    %edx,%eax
   0x0000000000001190 <+71>:    add    %eax,%eax
   0x0000000000001192 <+73>:    sub    %eax,%ecx
   0x0000000000001194 <+75>:    mov    %ecx,%edx
   0x0000000000001196 <+77>:    test   %edx,%edx
   0x0000000000001198 <+79>:    je     0x11a6 <loop+93>
   0x000000000000119a <+81>:    mov    $0x59,%edi
   0x000000000000119f <+86>:    call   0x1050 <putchar@plt>
   0x00000000000011a4 <+91>:    jmp    0x11a7 <loop+94>
   0x00000000000011a6 <+93>:    nop
   0x00000000000011a7 <+94>:    addl   $0x2,-0x8(%rbp)
   0x00000000000011ab <+98>:    mov    -0x8(%rbp),%eax
   0x00000000000011ae <+101>:   cmp    -0x18(%rbp),%eax
   0x00000000000011b1 <+104>:   jl     0x1170 <loop+39>
   0x00000000000011b3 <+106>:   movl   $0x0,-0x4(%rbp)
   0x00000000000011ba <+113>:   jmp    0x11d4 <loop+139>
   0x00000000000011bc <+115>:   mov    -0x4(%rbp),%eax
   0x00000000000011bf <+118>:   and    $0x7,%eax
   0x00000000000011c2 <+121>:   test   %eax,%eax
   0x00000000000011c4 <+123>:   je     0x11de <loop+149>
   0x00000000000011c6 <+125>:   mov    $0x4e,%edi
   0x00000000000011cb <+130>:   call   0x1050 <putchar@plt>
   0x00000000000011d0 <+135>:   addl   $0x1,-0x4(%rbp)
   0x00000000000011d4 <+139>:   mov    -0x4(%rbp),%eax
   0x00000000000011d7 <+142>:   cmp    -0x1c(%rbp),%eax
   0x00000000000011da <+145>:   jl     0x11bc <loop+115>
   0x00000000000011dc <+147>:   jmp    0x11df <loop+150>
   0x00000000000011de <+149>:   nop
   0x00000000000011df <+150>:   addl   $0x1,-0xc(%rbp)
   0x00000000000011e3 <+154>:   mov    -0xc(%rbp),%eax
   0x00000000000011e6 <+157>:   cmp    -0x14(%rbp),%eax
   0x00000000000011e9 <+160>:   jl     0x1167 <loop+30>
   0x00000000000011ef <+166>:   nop
   0x00000000000011f0 <+167>:   nop
   0x00000000000011f1 <+168>:   leave
   0x00000000000011f2 <+169>:   ret
End of assembler dump.
```

Continued on next page.

3. Assembly, Stack Discipline, Calling Convention, and x86-64 ISA, *cont.*
Part A: Loops, Conditionals, and Calling Convention (7 points)*, cont.*

For your reference: Arguments are passed in the order %rdi, %rsi, %rdx, %rcx, %r8, and %r9. %rax is used for return values.

3(A)(1) (2 points): How many loops does this function have? How do you know?

3 backward jumps forming cycles

3(A)(2) (2 points): How many arguments does this function receive (and use)? How do you know?
Hint: The first 6 arguments are passed via registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order.

Three. %rdi, %rsi, and %rdx are saved to the stack. These values from the stack are then used as part of loop test for each of the three different loops

3(A)(3) (2 points): How many C Language **if** statements are likely contained within this code? *Hint:* Do not count conditionals that likely control the repetition of **for** loops.

One. Guarding what looks like a break.

3(A)(4) (1 points): How many break statements are there? At what address is each located?

One. 0x11dc <+147>

Continued on next page.

3. Assembly, Stack Discipline, Calling Convention, and x86-64 ISA, *cont.*
Part B: Loops, Conditionals, and Calling Convention (8 points)

Consider the following compiled from C Language code containing a switch statement and no if statements. Remember that the **jump table keeps offsets from its own start address. The address of each code block is the address of the beginning of the jump table plus the value of the code block's jump table entry.** You'll see this address before the relevant jump in the assembly. It might make things easier for you to note the address indicated by the lowest jump table entry and think of the other entries relative to that one.

```
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x0000000000001169 <+0>:     endbr64
   0x000000000000116d <+4>:     push   %rbp
   0x000000000000116e <+5>:     mov    %rsp,%rbp
   0x0000000000001171 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000001174 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000000000001177 <+14>:    cmpl   $0x8,-0x8(%rbp)
   0x000000000000117b <+18>:    ja     0x11cc <foo+99>
   0x000000000000117d <+20>:    mov    -0x8(%rbp),%eax
   0x0000000000001180 <+23>:    lea    0x0(,%rax,4),%rdx
   0x0000000000001188 <+31>:    lea    0xe75(%rip),%rax        # 0x2004
   0x000000000000118f <+38>:    mov    (%rdx,%rax,1),%eax
   0x0000000000001192 <+41>:    cltq
   0x0000000000001194 <+43>:    lea    0xe69(%rip),%rdx        # 0x2004
   0x000000000000119b <+50>:    add    %rdx,%rax
   0x000000000000119e <+53>:    notrack jmp *%rax
   0x00000000000011a1 <+56>:    addl   $0x2,-0x4(%rbp)
   0x00000000000011a5 <+60>:    jmp    0x11d2 <foo+105>
   0x00000000000011a7 <+62>:    addl   $0x3,-0x4(%rbp)
   0x00000000000011ab <+66>:    mov    -0x4(%rbp),%eax
   0x00000000000011ae <+69>:    lea    0x3(%rax),%edx
   0x00000000000011b1 <+72>:    test   %eax,%eax
   0x00000000000011b3 <+74>:    cmovs  %edx,%eax
   0x00000000000011b6 <+77>:    sar    $0x2,%eax
   0x00000000000011b9 <+80>:    mov    %eax,-0x4(%rbp)
   0x00000000000011bc <+83>:    jmp    0x11d2 <foo+105>
   0x00000000000011be <+85>:    mov    -0x4(%rbp),%eax
   0x00000000000011c1 <+88>:    imul   -0x8(%rbp),%eax
   0x00000000000011c5 <+92>:    mov    %eax,-0x4(%rbp)
   0x00000000000011c8 <+95>:    addl   $0x5,-0x4(%rbp)
   0x00000000000011cc <+99>:    mov    -0x8(%rbp),%eax        # 0x2004 - 3640 is 0x11cc
   0x00000000000011cf <+102>:   add    %eax,-0x4(%rbp)
   0x00000000000011d2 <+105>:   mov    -0x4(%rbp),%eax
   0x00000000000011d5 <+108>:   pop    %rbp
   0x00000000000011d6 <+109>:   ret
End of assembler dump.
```

Consider also the following memory dump.

```
(gdb) x/20wx 0x2000
0x2000: 0x00020001      0xfffff1c8      0xfffff19d      0xfffff1a3
0x2010: 0xfffff1a7      0xfffff1c8      0xfffff1c8      0xfffff1ba
0x2020: 0xfffff1ba      0xfffff1c4      0x000a6425      0x3b031b01
0x2030: 0x00000038      0x00000006      0xffffeff4      0x0000006c
0x2040: 0xfffff024      0x00000094      0xfffff034      0x000000ac

(gdb) x/20wd 0x2000
0x2000: 131073  -3640   -3683   -3677
0x2010: -3673   -3640   -3640   -3654
0x2020: -3654   -3644   680997  990059265
0x2030: 56      6       -4108   108
0x2040: -4060   148     -4044   172
```

Continued on next page.

Question 3: Assembly, Stack Discipline, Calling Convention, and x86-64, *cont.* (15 points)

Part B: Conditionals, *cont.* (8 points)

(3)(B)(1) (2 point): At what address does the jump table shown above begin? How do you know?

0x2004. Note the indirect jump at 0x119e <+53>. It builds up an index into the jump table from the 2nd argument, which it moves onto the stack, "mov %esi,-0x8(%rbp)", and then uses from there to index into the table. The comments show 0x2004 used as the base of the table.

(3)(B)(2) (2 points): Is there a default case? If so, at what address does it begin? How do you know?

Yes. Note the guard at <+14> and <+18>, and that goes to 0x11cc <+99>

(3)(B)(3)(2 points): Which case(s), if any, fall through to the next case <u>after executing some of their own code</u>? How do you know?

Hint: Give the case number not the address.

Cases 2, 6, 7, and 8. These don't get out of the code block via a return or jump, etc. They keep executing, running past the entry point for another case.

(3)(B)(2) (2 points): What integer input values are managed by default cases of the switch statement? How do you know?

Anything greater than 8, 0, 4, or 5. These go to 0x11cc via the guarding jump or the jump table.

Continued on next page.

4. Caching, Locality, Memory Hierarchy, Effective Access Time
Part A: Cache Configuration (3 points)

Given a model described as follows:

- Associativity: 2-way set associative
- Total size: 256 bytes (not counting meta data)
- Block size:16 bytes/block
- Replacement policy: Set-wise LRU
- 8-bit addresses

4(A)(1) (0 point) How many bits for the block offset? (No points, just for you).

4 bits 0000 – 1111 ==> 0 – 15, 16 bytes/block

4(A)(2) (0 point) How many bits for the set index? (No points, just for you).

(**256B/cache) / (16B/block) = 16 blocks**
16 blocks / (2 blocks/set) = 8 sets
8 sets requires 3 bits to index, 000 – 111 ==> 0 - 7

4(A)(3) (0 point) How many bits for the tag? (No points, just for you).

Just 1. 8 bits - 3 bits (index) - 4 bits (offset) = 1 left over for the tag

4(A)(4) (3 points) Given this cache configuration, which of the following types of misses are possible: Conflict, Compulsory (Cold), Capacity. Explain.

Compulsory misses are possible, because the cache can't cache what it hasn't seen.

But neither capacity nor conflict misses are possible. Since 8-bit addresses can address 256B, and this is exactly the cache size, the cache can hold the entire memory. Since it is 2-way set associative, and the one (1) tag bit can code for two states, each line maps to two addresses, each of which can fit into the line.

Continues on next page.

4(B) Cache Trace (8 points, 1 point each line, ½ point each entry):

Below is an address trace giving a sequence of memory accesses. The trace begins at the beginning of time, e.g. the cache is empty.

For each of the following addresses, please indicate if it hits or misses. If it misses, further categorize it a capacity miss, a conflict miss, or a compulsory (cold) miss, or note whether the miss results in allocating an unused entry or replacing/evicting a valid entry, as indicated within the table. Mark your answers in the table.

Address	**Circle one (per row):**	**Circle one (per row):**
0x1C	Hit **Miss**	Capacity **Compulsory** Conflict N/A
0X1A	**Hit** Miss	Allocate Replace **N/A**
0X34	Hit **Miss**	Capacity **Compulsory** Conflict N/A
0X92	Hit **Miss**	**Allocate** Replace N/A
0X35	**Hit** Miss	Capacity Compulsory Conflict **N/A**
0XC1	Hit **Miss**	**Allocate** Replace N/A
0X9C	**Hit** Miss	Capacity Compulsory Conflict **N/A**
0XBA	Hit **Miss**	**Allocate** Replace N/A

4(C) (2 points) Blocking: In cache lab you used blocking to improve the performance of a matrix transpose. In the context of matrix transposition, what is blocking, and how does it improve performance?

As memory accesses move down a column of a 2D array, each access brings a cache block worth of memory into the cache, eventually displacing an aging block, 1-for-1. The result is that a rectangular area of memory is cached and moves downward with the sweep. Memory access within the rectangular area, if not row-wise, still benefits from spatial and temporal locality and can be found within the cache.

Continued on next page.

4(C)Effective Access Time (2 points)

Imagine a system with a DRAM-based main memory layered beneath an SRAM cache.

- The SRAM cache has a 15nS access time.
- The penalty for an SRAM cache miss is an additional 40nS.

Your goal is to design a system with a system with a memory access time of 25nS, or better. What is the maximum SRAM cache miss rate that can be tolerated? Round up to a whole number.

FOR SIMPLICITY, AVOID COMPLEX CALCULATION AND LEAVE YOUR ANSWER AS A SIMPLE FRACTION

What is the maximum acceptable miss rate to achieve a system performance of 25nS?

4(C)(1) (3points) MISS_RATE =

25ns = 15nS + MISS_RATE * 40nS
10nS = MISS_RATE * 40nS
10nS/40nS = MISS_RATE
MISS_RATE = 1/4 = 0.25

Continued on next page.

Question 5: Malloc(), Free(), and User- Memory Allocation (10 points)

5(A)(1) (2 points): As compared to an explicit list allocator, how does a segregated list allocator help to balance utilization and throughput? Explain.

Since only lists with good candidates are searched, it isn't necessary to sort lists or brute-force search them to avoid bad matches, nor is it necessary to accept an overly large match just to avoid the penalty of further searching.

5(A)(2) (2 points): When implementing implicit lists, we chose to include ways to access both *prev* and *next* blocks, and we kept these even after moving forward to implementing explicit and segregated lists. What optimization did this enable? And how?

These pointers enabled constant time coalesce by enabling a look left. This is needed with explicitly lists and segregated lists just as much without. Those techniques provide a logical organization to improve the search for a match, but don't impact coalesce, which necessarily works with address-neighbors, at all.

5(A)(3) (3 points): Once we sufficiently optimized our malloc implementation, we were able to reduce internal fragmentation due to footers. What was the key observation that enabled us to achieve this? And given this observation, where did our footers live? Explain.

The key observation is that metadata is free as long as it is in unallocated blocks. It only comes with an opportunity cost when it is in an allocated block. Our footers lived only in allocated blocks.

5(A)(4) (3 points): What advantage does a segregated list allocator have over an allocator that simply calls brk()/sbrk() to grow the heap for each malloc() call and does the same to lower the brk point, shrinking the heap, when possible?

It provides a mechanism to re-use recently freed blocks that aren't contiguous with the top of the heap and therefore can't be deallocated. This dramatically reduces fragmentation and dramatically increases utilization. In fact, most modern malloc's don't shrink the heap, ever.

Continued on next page.

6. Virtual Memory, Paging, and the TLB (15 points)

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 12 bits wide.
- Physical addresses are 10 bits wide.
- The page size is 64 bytes.
- The TLB is 2-way set associative with 4 total entries.
- The TLB may cache invalid entries
- A single level page table is used
- The replacement policy for the TLB is to replace invalid entries before valid entries and, in the event of two of a kind, to replace the lowest tag within the set (Regardless of whether or not this is the smartest thing to do).

Part A: Interpreting addresses

6(A)(1)(2 points): Please label the diagram below showing which bit positions are interpreted as each of the PPO and PPN. Leave any unused entries blank.

Bit	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PPN/ PPO	***N***	***N***	***N***	***N***	***O***	***O***	***O***	***O***	***O***	***O***

6(A)(2)(2 points): Please label the diagram below showing which bit positions are interpreted as each of the VPO and VPN and each of the TLBI (TLB Index) and TLBT (TLB Tag). Leave any unused entries blank.

Bit	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
VPO/ VPN	***N***	***N***	***N***	***N***	***N***	***N***	***O***	***O***	***O***	***O***	***O***	***O***
TLBI/ TLBT	***T***	***T***	***T***	***T***	**T**	**I**						

6(A)(3) (2 points): How many entries exist within each page table? *Hint:* This is the same as the total number of pages within each virtual address space.

2^12 address bits = 4096 bytes, 4096 bytes / 64bytes/page = 64 pages.

6(A)(4) (2 points): How many sets are in the TLB?

4 entries/tlb / 2 entries/set = 2 sets/TLB

Continued on next page.

Virtual Memory, Paging, and the TLB (15 points)*, cont.*

Part B: Hits and Misses (7 points)

Shown below are the **initial** states of the TLB and **partial** page table.

TLB (X=INvalid, V=VALID, R=READ, W=WRITE, NR=Not Resident, e.g. swapped):

Set	Tag	PPN	BITS	Scratch space for you
0	4	4	X-R	
0	8	2	V-RW	
1	8	5	V-R	
1	10	1	V-RW	

Page Table (X=INvalid V=VALID, R=READ, W=WRITE, NR=Not Resident, e.g. swapped):

Index/VPN	PPN	BITS	Scratch space for you
0x8 (0b1000)	3	V-R	
0xD (0b1101)	11	X-RW	
0x10 (0b1_0000)	2	V-RW	
0x11 (0b1_0001)	5	V-R	
0x15 (0b1_0101)	1	V-RW	
0x1D (0b1_1101)	7	V-R	

Continued on next page.

Virtual Memory, Paging, and the TLB (15 points)*, cont.*

Part B: Hits and Misses (7 points), *cont.*

Consider the following memory access trace i.e. sequence of memory operations listed in order of execution, as shown in the first two columns (operation, virtual address). It begins with the TLB and page table in the state shown above.

Please complete the remaining columns

Operation	Virtual Address	TLB Hit or Miss?	Page Table Hit or Miss?	Page Fault? Yes or No?	PPN If Knowable
Read	0x540	**Hit** Miss Not knowable	Hit Miss **N/A**	Yes **No** Not knowable	***1***
Write	0x440	**Hit** Miss Not knowable	Hit Miss **N/A**	**Yes** No Not knowable	***5***
Read	0x740	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	***7***
Write	0x200	Hit **Miss** Not knowable	**Hit** Miss N/A	**Yes** No Not knowable	***3***
Read	0x340	Hit **Miss** Not knowable	Hit **Miss** N/A	**Yes** No Not knowable	***UNKNOWN***
Read	0x440	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	***5***
Read	0x200	**Hit** Miss Not knowable	Hit Miss **N/A**	Yes **No** Not knowable	***3***

Continued on next page.

Question 7: Process Representation and Lifecycle + Signals and Files (10 points)

Part A (4 points):

Please consider the following following process tree:

```
                       .    □ putchar('C') □ putchar('D') )□ putchar('F')
                       .
                       .
putchar('A') □ fork()□ putchar('B') □ *DELAY □ putchar('E' )□ putchar('F')
```

**DELAY* means that the parent should delay at that point until the child is done running.

7(A)(1) (3 points): Please fill in the main() method below such that it implements the specification provided by the tree above with as few calls to putchar() as possible.

```
void main(){
  // Your code here
  putchar('A');
  if (fork()) {
    // Parent
    putchar('B');
    wait(NULL);
    putchar('E');
  } else {
    // Child
    putchar('C');
    putchar('D');
  }

  putchar('F');
}
```

7(A)(2) (1 points): Consider only the last three characters of the output. Please list all possibilities.

FEF

Continued on next page.

Question 7: Process Representation and Lifecycle + Signals and Files (10 points), *cont.*

Part B: Files (3 points):

Please consider the following:

```
int foo() {
  int pid;
  int fd1, fd2;

  fd1 = open("/file1", O_RDWR);

  dup2(fd1, 1); // Note that fd=1 is stdout
  printf("A");  // printf()s print to fd=1

  if ((pid = fork()) == 0) {
    printf("B");
    fd2 = open("/file1", O_RDWR);
    dup2(fd2, 1);
    printf("C");

    /* POINT X */

  } else {
    waitpid(pid, NULL, 0);
    printf("D");
    close(fd1);
    printf("E");
  }

  exit(2);
}
```

7(B)(1) (1 point) How many processes share the open file structure referred to by fd1 at "POINT X" in the code?

2

7(B)(2) (1 point) How many file descriptors (total among all processes) share the open structure referred to by fd1 at "POINT X" in the code?

3

7(B)(3) (1 point) Assuming that /file1 was empty before running this code, what are its contents after the execution is complete?

CBDE

Continued on next page

Question 7: Process Representation and Lifecycle + Signals and Files (10 points)
Part C: Signals (3 points), ***cont :***

7(C)(1) (2 points): In order for your shell to properly handle background jobs it had to implement asynchronous signal handling. What signal did it need to handle and what would have been the cost/consequence of not handling it (properly)?

It had to handle SIGCHLD with an asynchronous signal handler. Otherwise, if it continued to prompt for, and run, another job, it would not have been able to reap the background job, and resources would be leaked in the form of "zombie" defunct processes.

7(C)(2) (1 points): Imagine that a signal is received while the receiving process blocks it. What happens when should the receiving process subsequently unblock the signal? And, why does this make sense?

The signal will be received. This makes sense because otherwise when signals are blocked for concurrency control, they would be lost forever and the action wouldn't be delayed, but untaken.

Continued on next page.

Question 8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW (15 points)

Please consider the following code:

```
// Based upon:
// https://courses.cs.vt.edu/cs3214/spring2012/butta/documents/SampleFinalCS3214F10.pdf

// here, 'pop' returns the first element
// of the pool's queue, or NULL if queue
// is empty

while ( !getShuttingDown(queue) ) {
  struct work_t *work = NULL;

  pthread_mutex_lock(&queue->mutex);
  work = dequeue(queue); // Return the first element, or NULL if empty
  pthread_mutex_unlock(&queue->mutex);

  while ( work == NULL ) {
    pthread_mutex_lock(&queue->mutex);
    work = dequeue(queue); // Not an async safe function
    pthread_mutex_unlock(&queue->mutex);

    if ( getShuttingDown(queue) ) {
      return NULL;
    }
  }

  // do work
}
```

8(A)(1) (2 points) Consider the above code which accesses a work queue that is shared among threads; dequeues work, if available, and repeats unless and until the work queue is shut down. Is this code *correct*? *Hint:* Consider whether it is safe, whether it makes progress, and whether there is starvation.

Yes. It is safe. It does not get stuck, starve any thread, or leave the critical resource (queue) unprotected.

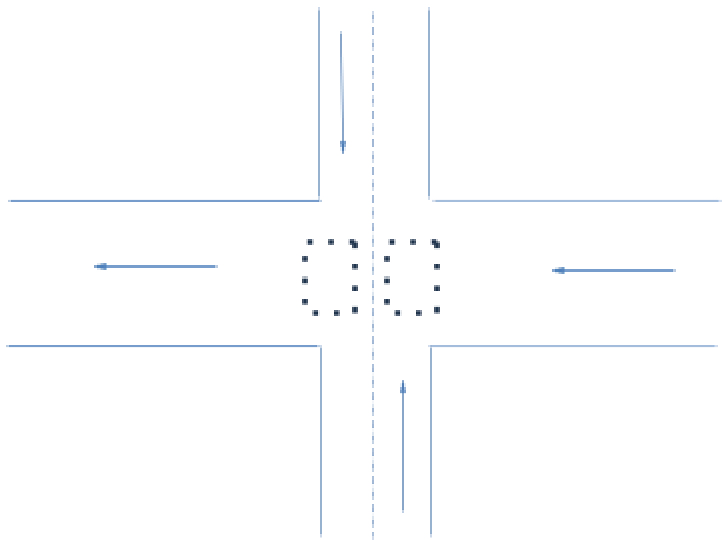
8(A)(1) (3 points) Consider the code again. With respect to efficiency, it does have some room for improvement. Please describe the inefficient behavior and a strategy, based upon the techniques we learned in class, that you might implement to address it.

It has a busy wait loop, which means it is spinning accomplishing nothing while something could be getting work done. This could be treated like the producer-consumer problem, which would allow the worker thread to block in a semaphore vs spinning, while waiting for work.

Continued on next page.

8(B) Designing Concurrency Control (10 points)

Consider the following intersection:



- The east-west road has a single westbound lane, as shown.
- The north-south road has two lanes, one in each direction, as shown.
- Each car knows the road it is on and the turn it wants to take.
- Potential collisions exist in the regions noted with dotted boxes.

Recall the following:

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
  - `attr is NULL in our use.`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

And, assume that consts or typedefs exist for

- NBOUND
- SBOUND
- EBOUND
- WBOUND

As does the following function which actually moves the vehicle,but does no concurrency control:

- `void driveThrough(direction_t initial_dir, direction_t turn_to);`

Continued on next page.

8(B) Designing Concurrency Control (10 points), *cont.*

```
// Declare and initialize global variables here (3 points)
pthread_mutex_t east_mutex;
pthread_mutex_t west_mutex;
pthread_mutex_init(&east_mutex, NULL);
pthread_mutex_init(&west_mutex, NULL);


// Moves car into intersection, used by worker threads (4 points)
// CallsDriveThrough(…)
void advance (direction_t initial_dir, direction_t turn_to) {

  if (SBOUND == initial_dir) {
    pthread_mutex_lock(west_mutex);
    driveThrough(initial_dir, turn_to);
    pthread_mutex_unlock(west_mutex);
    return;
  }

  if ((NBOUND == initial_dir) && (NBOUND == turn_to)) {
    pthread_mutex_lock(east_mutex);
    driveThrough(initial_dir, turn_to);
    pthread_mutex_unlock(east_mutex);
    return;
  }

  if ((NBOUND == initial_dir) && (WBOUND == turn_to)) {
    pthread_mutex_lock(west_mutex);
    pthread_mutex_lock(east_mutex);
    driveThrough(initial_dir, turn_to);
    pthread_mutex_unlock(east_mutex);
    pthread_mutex_unlock(west_mutex);
    return;
  }

  if ((WBOUND == initial_dir) && (NBOUND == turn_to)) {
    pthread_mutex_lock(east_mutex);
    driveThrough(initial_dir, turn_to);
    pthread_mutex_unlock(east_mutex);
     return;
   }

  if ((WBOUND == initial_dir)&& (WBOUND == turn_to) ) {
    pthread_mutex_lock(west_mutex);
    pthread_mutex_lock(east_mutex);
    driveThrough(initial_dir, turn_to);
    pthread_mutex_unlock(east_mutex);
    pthread_mutex_unlock(west_mutex);
    return;
   }

 // Should never occur
 handleError ("Invalid direction", initial_dir, turn_to);
```

```
            return;
        }
    }
```

All done! Great work! Enjoy the summer!