Andrew ID: Full Name:

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

Fall 2024

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)

Assume we are running code on two machines using two's complement arithmetic for signed integers.

- Machine 1 has 6-bit integers
- Machine 2 has 4-bit integers.

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

	Machine 1: 8-bit w/2s complement signed	**Machine 2: 6-bit w/2s complement signed**
Binary representation of -38 decimal	1101 1010	UNABLE
Decimal value of +Tmax		01 1111
Decimal value of (30 + 24)		-10
Binary representation of -Tmin	1000 0000	

Commented [1]: Shouldn't this be 31 and not the binary form of 31 if we're asking for the decimal value?

Commented [2R1]: agreed

Continued on next page.

Part B: Floats (5 points, 1/2 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 is 1 sign bit *s*.
  - There are *k* = 2 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)	**1**
Number of Exponent Bits (Decimal)	**2**
Number of Fraction Bits (Decimal)	**3**
Bias (Decimal)	**1**
Largest magnitude negative number (Decimal value)	**-3 3/4**
+Infinity (Binary bit pattern)	**0 111 00**
001101 (Decimal value, unrounded)	**1 5/8**
100010 (Decimal value, unrounded)	**-1/4**
3 ¼ (Binary bit pattern)	**010101**
What is the rounding error as a decimal fraction when the decimal number below is represented?	**0 00 0111** **1/16**

Commented [3]: Was 11, should be 00

Commented [4]: shouldn't there be 2 exponent bits and 3 fraction bits? so the answer should be 0 11 000

Commented [5]: this should be -5/8 if interpreted with 2 exponent bits and 3 fraction bits, currently this is correct only if there's 3 exponent bits and 2 fraction bits

Commented [6]: There is no decimal number below represented here??

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 8-byte `long`s:

Definition
`unsigned long numbers[3][2];`

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

8 bytes * 3 * 2 = 48 bytes

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

0x10000 + (((1*2)+3)*8) = 0x10000 + 0x28 = 0x10028

Part B : Structs and Alignment (4 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 {
  char c;    // 1-byte type
  short s1; // 2-byte type
  double d; // 8-byte type
  short s2;
} partB;
```

Continued on nest 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)
```

24 Bytes

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

```
struct {
  char c;    // 1-byte type
  short s1; // 2-byte type
  short s2;
  double d; // 8-byte type
} partB;
```

OR

```
struct {
  double d;
  short s1;
  short s2;
  char c;
} partB;
```

2(B)(3) (2 points): Consider your revised struct from 2(B)(2) above and the definition of `arrayB3` given below, what is the distance, measured in bytes, between the address of `arrayB3[3].s2` and the address of `arrayB3[5].d`?

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

```
struct {
  char c;    // 1-byte type
  //            1-byte padding
  short s1; // 2-byte type
  short s2;  // 2-byte type
  //            2-byte padding
  double d; // 8-byte type
```

```
} revisedPartB;
```

sizeof(revisedPartB) = 16 bytes
(&(arrayB3[3]) – &(arrayB3[5])) = 2 * sizeof(revisedPartB) = 2 * 16 bytes = 32 bytes

OR IF THE STRUCT IS REVERSED:
1 * sizeof(revisedPartB) = 16 bytes
+ 4 for the difference between .s2 and the next .d
= 20 bytes

Offset of field `.s2` is 4 bytes
Offset of field `.d` is 8 bytes

32 bytes – 4 bytes + 8 bytes = 36 bytes

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:
   0x0000000000001129 <+0>:     endbr64
   0x000000000000112d <+4>:     push   %rbp
   0x000000000000112e <+5>:     mov    %rsp,%rbp
   0x0000000000001131 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000001134 <+11>:    mov    %esi,-0x18(%rbp)
   0x0000000000001137 <+14>:    movl   $0x0,-0x8(%rbp)
   0x000000000000113e <+21>:    movl   $0x0,-0x4(%rbp)
   0x0000000000001145 <+28>:    cmpl   $0x0,-0x14(%rbp)
   0x0000000000001149 <+32>:    je     0x117f <loop+86>
   0x000000000000114b <+34>:    movl   $0x0,-0x8(%rbp)
   0x0000000000001152 <+41>:    jmp    0x1172 <loop+73>
   0x0000000000001154 <+43>:    mov    -0x8(%rbp),%eax
   0x0000000000001157 <+46>:    cltq
   0x0000000000001159 <+48>:    lea    0x0(,%rax,4),%rdx
   0x0000000000001161 <+56>:    lea    0x2eb8(%rip),%rax        # 0x4020
<numbers>
   0x0000000000001168 <+63>:    mov    (%rdx,%rax,1),%eax
   0x000000000000116b <+66>:    add    %eax,-0x4(%rbp)
   0x000000000000116e <+69>:    addl   $0x1,-0x8(%rbp)
   0x0000000000001172 <+73>:    mov    -0x8(%rbp),%eax
   0x0000000000001175 <+76>:    cmp    -0x18(%rbp),%eax
   0x0000000000001178 <+79>:    jl     0x1154 <loop+43>
   0x000000000000117a <+81>:    mov    -0x4(%rbp),%eax
   0x000000000000117d <+84>:    jmp    0x11ae <loop+133>
   0x000000000000117f <+86>:    mov    -0x18(%rbp),%eax
   0x0000000000001182 <+89>:    mov    %eax,-0x8(%rbp)
   0x0000000000001185 <+92>:    jmp    0x11a5 <loop+124>
   0x0000000000001187 <+94>:    mov    -0x8(%rbp),%eax
   0x000000000000118a <+97>:    cltq
   0x000000000000118c <+99>:    lea    0x0(,%rax,4),%rdx
   0x0000000000001194 <+107>:   lea    0x2e85(%rip),%rax        # 0x4020
<numbers>
   0x000000000000119b <+114>:   mov    (%rdx,%rax,1),%eax
   0x000000000000119e <+117>:   sub    %eax,-0x4(%rbp)
   0x00000000000011a1 <+120>:   subl   $0x1,-0x8(%rbp)
   0x00000000000011a5 <+124>:   cmpl   $0x0,-0x8(%rbp)
   0x00000000000011a9 <+128>:   jns    0x1187 <loop+94>
   0x00000000000011ab <+130>:   mov    -0x4(%rbp),%eax
   0x00000000000011ae <+133>:   pop    %rbp
   0x00000000000011af <+134>:   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?

Two. There are two backward jumps.

3(A)(2) (2 points): How many arguments does this function receive (and use)?

Two %rdi and %rsi (%edi and %esi). They are the registers used for the first two arguments, are immediately preserved on the stack, and used on the code.

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

One. It chooses which of the two loops should run. It is a conditional forward jump that doesn't escape a loop.

3(A)(4) (1 points): If one or more loops are controlled (start at, end at, or otherwise configured) by an argument, please indicate which argument, if not, please write "NO". In either case, how do you know?

2nd argument. %rsi is used as the end of the 1st loop and the beginning of the 2nd loop.

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. It uses a very common form of the switch statement on the shark machines, but a slightly different one than some prior exams. Rather than keeping absolute addresses**, this 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:
=> 0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     cmp    $0xa,%esi
   0x0000555555555170 <+7>:     ja     0x5555555551aa <foo+65>
   0x0000555555555172 <+9>:     mov    %esi,%eax
   0x0000555555555174 <+11>:    lea    0xe89(%rip),%rdx        # Hint: %rdx =0x555555556004
   0x000055555555517b <+18>:    movslq (%rdx,%rax,4),%rax
   0x000055555555517f <+22>:    add    %rdx,%rax
   0x0000555555555182 <+25>:    notrack jmp *%rax
   0x0000555555555185 <+28>:    lea    0x3(%rdi),%eax          # Hint: 0x55555555600c
   0x0000555555555188 <+31>:    ret
   0x0000555555555189 <+32>:    lea    -0x3(%rdi),%eax#        Hint: 0x555555556014
   0x000055555555518c <+35>:    ret
   0x000055555555518d <+36>:    lea    -0x1(%rdi),%eax
   0x0000555555555190 <+39>:    ret
   0x0000555555555191 <+40>:    add    $0x2,%edi
   0x0000555555555194 <+43>:    movslq %edi,%rsi
   0x0000555555555197 <+46>:    imul   $0x55555556,%rsi,%rsi
   0x000055555555519e <+53>:    shr    $0x20,%rsi
   0x00005555555551a2 <+57>:    sar    $0x1f,%edi
   0x00005555555551a5 <+60>:    mov    %esi,%eax
   0x00005555555551a7 <+62>:    sub    %edi,%eax
   0x00005555555551a9 <+64>:    ret
   0x00005555555551aa <+65>:    lea    (%rsi,%rdi,1),%eax
   0x00005555555551ad <+68>:    ret
End of assembler dump.
```

Consider also the following memory dump.

```
(gdb) x/20wx  0x555555556000
0x555555556000: 0x00020001      0xfffff1a6      0xfffff1a6      0xfffff181
0x555555556010: 0xfffff1a6      0xfffff185      0xfffff1a6      0xfffff189
0x555555556020: 0xfffff1a6      0xfffff189      0xfffff18d      0xfffff190
0x555555556030: 0x000a6425      0x3b031b01      0x00000038      0x00000006
0x555555556040: 0xffffefec      0x0000006c      0xfffff01c      0x00000094
0x555555556000: 131073  -3674   -3674   -3711
0x555555556010: -3674   -3707   -3674   -3703
0x555555556020: -3674   -3703   -3699   -3696
0x555555556030: 680997  990059265       56      6
0x555555556040: -4116   108     -4068   148
```

Commented [7]: i know many students were concerned with not being able to calculate the hex addresses subtracted by the offsets. maybe providing what 1 specific offset corresponds to and letting students determine the cases based on the difference between offsets may be more helpful?

Commented [8R7]: I think that's what this hint is doing, its telling you that this line is reached by the offset at that address in the jump table, and counting from the start of the jump table its executing case 6 and then you use that information to find out where all the other offsets are in relation to it

Commented [9R7]: ah i can see that now -- thanks for clarifying! personally, i'd be less confused if the hint stated the offset that line corresponds to rather than the hex address, but i don't think it's a super necessary change

Commented [10R7]: facts I think adding the decimal offset here would make it more helpful that the student should look at the jump table, or mentioning a little context for the hints in the instruction

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?

0x555555556004. It is the base of the indirect jump @ <+18>

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

Yes. 0x5555555551aa <foo+65>. It is the address that is jumped to by the guard.

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

Hint: Give the case number not the address.

Case 9. There is no return before it continue down into assembly code that is, according to the start point in the jump table, part of case 10.

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

2, 4, 6, 8, 9, and 10. Their entries in the table are not the start point for the default case as identified by the initial guard jump.

Continued on next page.

4. Caching, Locality, Memory Hierarchy, Effective Access Time
Part A: Caching (8 points)

Given a model described as follows:

- Associativity: 2-way set associative
- Total size: 128 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?

4

4(A)(2) (0 point) How many bits for the set index?

2

4(A)(3) (0 point) How many bits for the tag?

2

4(A)(4) (8 points, 1 points each): For each of the following addresses, please indicate if it hits, or misses, and if it misses, if it suffers from a capacity miss, a conflict miss, or a cold miss:

Address	**Circle one (per row):**		**Circle one (per row):**			
0x61	Hit	**Miss**	Capacity	**Cold**	Conflict	N/A
0XAC	Hit	**Miss**	**Allocate**	Replace		N/A
0X6A	**Hit**	Miss	Capacity	Cold	Conflict	**N/A**
0X7E	Hit	**Miss**	**Allocate**	Replace		N/A
0X7E	**Hit**	Miss	Capacity	Cold	Conflict	**N/A**
0XEE	Hit	**Miss**	Allocate	**Replace**		N/A
0XAD	Hit	**Miss**	Capacity	Cold	**Conflict**	N/A
0X61	Hit	**Miss**	Allocate	**Replace**		N/A

Commented [11]: what does allocate and replace correspond to here? the instructions only mention capacity, conflict, or cold miss as things to note in the table, so it could be helpful to clarify what these terms are referring to in the instructions

Commented [12R11]: I second this, just adding anything about allocate vs replace meaning add it into the set versus kick something out of the set would be helpful, since students don't see this terminology anywhere else

Continued on next page.

4. Caching, Locality, Memory Hierarchy, Effective Access Time, *cont.*

4(B)(1) (2 points): Consider the following code:

```
short array[ARRAY_SIZE];
int sum=0;
for (int index=0; index<(ARRAY_SIZE-1); index+= step)
   sum += array[index]+ array[index+1];
```

Imagine that the data type increases from a short to an int to a long. As the data size increases, holding the cache configuration constant, how does each of spatial and temporal locality change? Please mark the table below.

Spatial	Decrease	Increase	**Unaffected**
Temporal	Decrease	Increase	**Unaffected**

4(B)(2) (2 points): Consider the following code:

```
long
array[ROWS][COLS];
long sum=0;
for (int count=0; count < REPEAT; count++)
  for (int row=0; index<ROWS; row +=2)
    for (int col=0; col<COLS; col +=2)
      sum += array[row][col] + array[row][col+1];
```

Imagine an extremely large array (relative to the cache size), a long size of 8 bytes, and a cache block size of 16 bytes. To the nearest whole percent or simple fraction, what would you expect the miss rate for accesses to “array” to be? Why?

50%.
array[row][col+1] misses each time, but array[row][col] hits each time, because it was loaded with the prior iteration of the loop. Since the array is huge there is no benefit from the repetition.

Continued on next page

4. Caching, Locality, Memory Hierarchy, Effective Access Time

Part C: Effective Access Time (3 points)
Imagine a system with a DRAM-based main memory layered beneath an SRAM cache.

- The SRAM cache has a 5nS access time.
- The penalty for an SRAM cache miss is 45nS.

Your goal is to design a system with a system with a memory access time of X, 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 20nS?

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

5ns + *X45ns = 20nS**
***X**45ns = 15ns**
***X*= 15ns/45ns**
***X* = 1/3**

Continued on next page.

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

5(A)(1) (2 points): When implementing malloc, it was suggested that certain lists be accessed in a circular fashion vs a head-first fashion. Was this intended to improve throughput, utilization, both, or neither? Why?

Throughput. It prevented favoring the head-most (or tail-most) parts of the list and, thereby, draining those areas of the most useful blocks, but walking over them more frequently, anyway.

5(A)(2) (2 points): In a general purpose allocator utilizing segregated lists, it is often less useful to perform best-fit within an explicit list than it is in a general purpose allocator with only one segregated list. Why?

Since the list are already organized by size, the benefit to utilization is more limited, making it less likely to be worth the throughput cost.

5(A)(3) (2 points): When implementing an explicit list allocator, it was necessary (within the bounds of reason) to keep the underlying implicit list. Why?

Without it, there'd be no reasonable way to coalesce, since coalescing can only be performed by physically adjacent blocks.

5(A)(4) (2 points): Why is free() unable to accept a pointer within an allocated block, i.e. why can it only accept a pointer to the beginning of the payload?

It needs to be able to locate the header from the pointer it is given.

5(A)(5) (2 points): Provide two (2) reasons that malloc might pad the requested payload.

(i) **Alignment. It needs to ensure that each payload pointer meets the largest alignment rule in place on the system.**
(ii) **Throughput. It is too costly to generate tiny blocks that are unlikely to be used.**

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 128 bytes.
- The TLB is 4-way set associative with 8 total entries.
- The TLB may cache invalid entries
- A single level page table is used

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.

Commented [13]: I think there's an extra offset bit. Page size = 128 = 2^7, but we have 8 bits

Bit	9	8	7	6	5	4	3	2	1	0
PPN/ PPO	*N*	*N*	*O*	*O*	*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 (top line) and each of the TLBI and TLBT (bottom line). 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*	*O*	*O*	*O*	*O*	*O*	*O*	*O*
TLBI/ TLBT	*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.

32

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

2 sets

7. 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	0	5	X-RW	***TLB Miss***
0	1	9	X-RW	***TLB Miss, PT Miss***
0	2	13	X-R	***TLB Miss, PT Hit***
0	3	27	V-RW	***TLB Hit***
1	0	7	V-NR	***TLB Hit***
1	1	11	X-RW	***TLB Miss, PT-Hit***
1	2	15	V-R	***TLB Hit, PT Hit***
1	5	19	X-RW	***Not needed***

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

Index/VPN	PPN	BITS	Scratch space for you
0	5	V-RW	***PT-Hit***
1	7	V-NR	***PT-Hit***
2	9	X-RW	***PT-Miss***
3	11	V-RW	***PT-Hit***
4	13	V-R	***PT-Hit, Invalid***
5	15	V-R	***PT-Hit, Invalid***
6	27	V-RW	***PT-Hit***

Continued on next page.

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

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

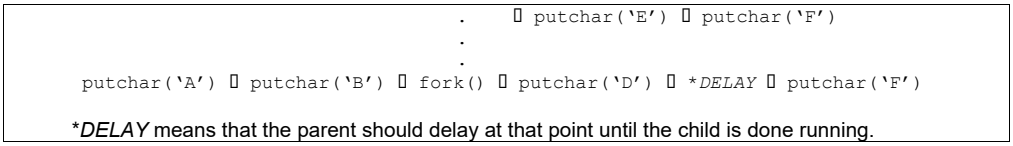
Consider the following memory access trace e.g. sequence of memory operations listed in order of execution, as shown in the first two columns (operation, virtual address). It begins with the TLB and page table in the state shown above.

Please complete the remaining columns 0011 0000

Operation	Virtual Address	TLB Hit or Miss?	Page Table Hit or Miss?	Page Fault? Yes or No?	PPN If Knowable
Write	0x208	Hit **Miss** Not knowable	**Hit** Miss N/A	**Yes** No Not knowable	***13***
Write	0x280	**Hit** Miss Not knowable	Hit Miss **N/A**	**Yes** No Not knowable	***15***
Write	0x308	**Hit** Miss Not knowable	Hit Miss **N/A**	Yes **No** Not knowable	***27***
Read	0x100	Hit **Miss** Not knowable	Hit **Miss** N/A	**Yes** No Not knowable	
Write	0x180	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	***11***
Read	0x004	Hit **Miss** Not knowable	**Hit** Miss N/A	Yes **No** Not knowable	***5***
Read	0x084	Hit **Miss** Not knowable	Hit Miss **N/A**	Yes No **Not knowable**	

Question 7: Process Representation and Lifecycle + Signals and Files (10 points)
Part A (4 points):

Please consider the following following process tree:



**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(){
   putchar('A');
   putchar('B');

   if (!fork()) {
     putchar('E');
   } else {
     putchar('D');
     wait(NULL);
   }

   putchar('F');
}
```

7(A)(2) (1 points): How many possible output strings are there?

Two (2)
AB (ED/DE) FF

Commented [14]: I think it should be 3: ABDEFF, ABEDFF, ABEFDF

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 code and an input file that consists of "ABCDEFGHIJKLMNOP":

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void main() {
  int fd1, fd2;
  char c;

  fd1=open("files.txt", O_RDONLY);
  read (fd1, &c, 1); write(STDOUT_FILENO, &c, 1);
  dup2(fd1, fd2);
  read (fd1, &c, 1); write(STDOUT_FILENO, &c, 1);
  read (fd2, &c, 1); write(STDOUT_FILENO, &c, 1);
  if (!fork()) {
    read (fd1, &c, 1); write(STDOUT_FILENO, &c, 1);
    read (fd1, &c, 1); write(STDOUT_FILENO, &c, 1);
    read (fd2, &c, 1); write(STDOUT_FILENO, &c, 1);
   } else {
    read (fd1, &c, 1); write(STDOUT_FILENO, &c, 1);
  }
```

7(B)(1) (1 points): What are the first three (3) characters of output?

ABC

7(B)(2) (1 points): How many possibilities are there for the next four (4) characters of output?

28 possibilties

Commented [15]: This is too difficult of a math question. I think we should just ask how many possibilities are there for the character that the parent writes.

I don't even think the answer is 28

7(B)(3) (1 points): Please explain your answer to 7(B)(2) above.

All of the parent (due to the dup2) and child (nature of fork) file descriptors share the same file table entry, so they are all sharing the same offset state.

The parent~~child~~'s read() can occur before any of the three child ~~read parent~~ read()/write()s or after the last one. So, it has 74 possible interleavings. The child write can occur any time after that.

Commented [16]: I think this explanation has child and parent swapped. The child has 3 read/write pairs, while the parent only has 1.

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

Part B: Files (3 points), *cont :*

Let's label some points:

```
if (!fork()) {
  (A)read (fd1, &c, 1);
  (B)write(STOUT_FILENO, &c, 1);
  (C)read (fd1, &c, 1);
  (D)write(STOUT_FILENO, &c, 1);
  (E)read (fd2, &c, 1);
  (F) write(STOUT_FILENO, &c, 1);
  (G)
} else {
  read (fd1, &c, 1); write(STOUT_FILENO, &c, 1);
}
```

A: If the child read occurs here, the child write can, in effect, occur at:
A, B, C, D, E, F, G
B: If the child read occurs here, the child write can, in effect, occur at:
B, C, D, E, F, G
C: If the child read occurs here, the child write can, in effect, occur at:
C, D, E, F, G
D: If the child read occurs here, the child write can, in effect, occur at:
D, E, F, G
E: If the child read occurs here, the child write can, in effect, occur at:
E, F, G
F: If the child read occurs here, the child write can, in effect, occur at:
F, G
G: If the child read occurs here, the child write can, in effect, occur at:
G

28 possibilities

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): Within the SIGCHLD handler of your shell, your waitpid() was within a loop. Why?

The pending bit of signal state is exactly that, one bit. It is not a queue. It indicates only whether or not at least one instance of that signal was received. There is no queue or counter. As a result, it is unclear how many processes are ready for waitpid(). We just know that it is at least one (1).

7(C)(2) (1 points): Within the SIGCHLD handler of your shell it was necessary to block other SIGCHLD signals. Why?

The SIGCHLD handler maintained the job list and other state that was not safe for concurrency. If the signal handler could interleave with itself, the state could be corrupted.

Continued on next page.

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

Please consider the following code which defines a singly linked list. Assume that a ”dummy” head node always exists.

```
// https://docs.oracle.com/cd/E19683-01/806-6867/6jfpgdcng/index.html#sync-50939
node1_t *delete(node_t startingPoint, int value)
{
    node1_t *prev, *current;

    prev = startingPoint;
    pthread_mutex_lock(&prev->lock);
    while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);
            pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return(current);
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
}
```

And recall that mutexes can be initialized, locked, and unlocked as below:

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_mutex_init(&mutex, NULL); // Intializes as an unlocked mutex
```

8(A)(1) (2 points) Imagine this code is used for a simple singly linked list with a distinguished, “dummy” head node such that there is only one `startingPoint` and it always exists. Is the code above free of concurrency control problems? If yes, write “Yes”. If not, please write “No” and explain. Please focus only on concurrency control problems.

Yes.

Continued on next page.

8(A)(2) (3 points) Now imagine the provided code above is used for a singly linked, but circular, list without a distinguished head node or a dummy. In other words imagine that each thread using this list might be starting at a different node. Is the code free of concurrency control problems? If yes, write "Yes". If not, please write "No" and explain.

Please ignore problems, such as an empty list or the item not being found, that are unrelated to concurrency control.

To avoid the potential for an endless loop, please assume the following single change:

```
while ((current = prev->link) != NULL) {
```

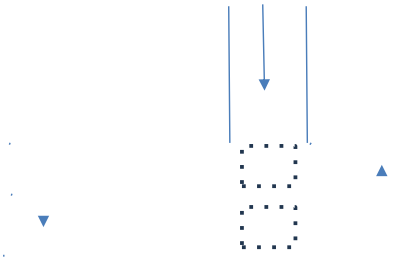
---------- becomes ----------

```
while ((current = prev->link) != startingPoint) {
```

No. Since the list is circular there is a circular dependency for the lock. Imagine a 2-node list and one thread attepting to delete the 1st node and another thread attempting to delete the 2nd node.

8(B) Designing Concurrency Control (10 points)

Consider the following intersection:



- The north-south road has a single southbound lane, as shown.
- The east-west 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.

Continued on next page.

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

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);`

8(B)(1) (7 points): Please implement the following:

```
// Decare and initialize global variables here (3 points)
pthread_mutex_t north_mutex;
pthread_mutex_t south_mutes;
pthread_init(&north_mutex, NULL);
pthread_init(&south_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) && (EBOUND == turn_to) ) {
    pthread_mutex_lock(north_mutex);
    pthread_mutex_lock(south_mutex);
    vehicleMovesThroughIntersection(initial_dir, turn_to);
    pthread_mutex_unlock(south_mutex);
    pthread_mutex_unlock(north_mutex);
    return;
  }

  if ((SBOUND == initial_dir) && (WBOUND == turn_to) ) {
    pthread_mutex_lock(north_mutex);
    vehicleMovesThroughIntersection(initial_dir, turn_to);
    pthread_mutex_unlock(north_mutex);
    return;

  }
```

Continued on next page.

8(B) Designing Concurrency Control (10 points), *cont.*
8(B)(1) (7 points): Please implement the following, *cont.:*

```
    if (EBOUND == initial_dir) { // Must stay EBOUND
      pthread_mutex_lock(south_mutex);
      vehicleMovesThroughIntersection(initial_dir, turn_to);
      pthread_mutex_unlock(south_mutex);
      return;
     }

    if (WBOUND == initial_dir) { // Must stay WBOUND
      pthread_mutex_lock(north_mutex);
      vehicleMovesThroughIntersection(initial_dir, turn_to);
      pthread_mutex_unlock(south_mutex);
      return;
    }

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

The end. All done. You made it! Happy break!