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, Spring 2024 Final Exam

Thursday, May 2, 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 any electronic devices or anything other than what we provide and writing implements, such as pens and pencils.
- 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)

1(A)(2 points) How many bits would be needed to represent the decimal number 2037 as a 2s complement signed number? Please assume that there are no constraints beyond those given, e.g. don't be concerned about alignment.

12 bits
negative bit plus….11 bits
1 3 7 15 31 63 127 255 511 1023 2047

1(B)(2 points) Please show the bit pattern present when the decimal number -27 is represented as a 8-bit negative number.

-27 = -128 + 64 + 32 + 4 + 1
1 1 1 0 0 1 0 1

1(C)(2 points) Consider a 4-bit 2s complement signed "int". What would be the output of the following code:

```
int x = -6;
int y = -5;
int z = -1;

z = x + y;

printf ("%d", z); fflush(stdout)
```

```
-6 = -8 + 0 + 1 + 0          1010
-5 = -8 + 0 + 1 + 1         +1011
                            ---------
                             0101
```

The output is "5".

1(D)(2 points) Consider an 8-bit unsigned number. What is the greatest magnitude positive number that can be represented?

1 1 1 1 1 1 1 1 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

1(E)(2 points) What is the result if Tmin (the greatest magnitude negative number) is added to Tmax (the greatest magnitude positive number)? *Hint: Consider the usual signed ints.*

-1
1000 + 0111 = -1
1000 0000 + 0111 1111 = -1

etc

Question 1: Representation: "Simple" Scalars (10 points)
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:
  - There are 8 bits
  - There are n = 4 exponent bits.
  - ***Hint:*** bias = $(2^{k-1} - 1)$

Fill in the empty (non grayed-out) boxes as instructed.

	Answer
Total Number of Bits (Decimal)	**8**
Number of Exponent Bits (Decimal)	**4**
Number of Fraction Bits (Decimal)	***3***
Bias (Decimal)	***$2^{4-1} - 1 = 7$***
The maximum possible difference between a Real number and the corresponding Float, to the nearest power of 2. Consider only numbers within the representative range without the use of an infinity or NaN.	**Consider** **0 1110 000 vs 0 1110 001** **$1.001\text{x } 2^7$ - 1.000×2^7** **$= 2^{-3}\text{x}2^7 /2 = 2^3 = 8$**
1 0000 010 **(As a reduced decimal fraction)**	***negative, denorm*** ***exp = 1 - 7 = -6*** ***0.010×2^{-6}*** ***$= -1.0\text{x}2^{-8}$*** ***= - 1/256***
How would the number below be represented as closely as possible? (Hint: Consider round-even, answer in decimal or as a decimal fraction) **-1 11/16 (-27/16)**	***-1 3/4 or -7/4*** ***1.1011×2^0 is either*** ***$1.101 \text{ x}2^7$ or 1.110×2^7*** ***and 1.110×2^7 is even***

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

Please consider a “Shark” machine for all parts of this question: 1-byte chars, 2-byte shorts, 4-byte ints, 8-byte longs, 8 byte doubles, 4 byte floats, and 8 byte doubles

Part A (3 points): Consider the following struct. How much memory is required? Answer in bytes.

```
 struct {
   char c1;
   int i;
   char c2;
   short s;
   long l;
} examStruct1;
```

24B: *c1=1B + padding=3B + i=4B + c2=1B + padding=1B + s=2B + padding=4B + l=8B*

Part B (2 points): How many bytes could be saved by reordering the fields of examStruct1?

```
 struct {
   char c1;
   char c2;
   short s;
   int i;
   long l;
} examStruct1;
```

8 bytes of padding can be saved

Part C (3 points): Consider the following array. How far apart are the addresses of array[0][1] and array[1][0]? Answer in bytes. Base your answer upon the struct in Part A.

```
struct examStruct1 array[2][3];

00 01 02
10 11 12          ==>      00 01 01 10 11 12
                              ^     ^
```

2 structs apart @ 24 bytes/struct

48 bytes apart

Part D (2 points): Consider "`struct examStruct1 es1;`". What is the offset in bytes of `es1.s` within the referenced struct.

c1=1B + padding=3B + i=4B + c2=1B + padding=1B, then s starts.
10 bytes

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

Part A: Calling Convention (4 points)

3(A)(1) (2 points): Consider the following code. How many bytes of the stack are used for parameter passing upon the calling of the function, fun()?

```
#include <stdio.h>

void fun (char c, int x[], short s) {
  x[0] = 5;

  printf ("%c %d %d\n", c, x[0], s);
}

void main() {
  char c = 'a';
  int x[4] = {0, 1, 2, 3};
  short s = -1;

  fun (c, x, s);

  printf ("%c %d %d\n", c, x[0], s);
}
```

0 bytes. Arrays are passed by reference, so all arguments are passed via registers.

3(A)(2) (2 points): Consider the following code. How many bytes of the stack are used for parameter passing upon the calling of the function?

```
#include <stdio.h>

struct sstruct {
  int x;
  int y;
};

void fun (char c, struct sstruct s, int i) {
  s.x = 5;

  printf ("%c %d %d\n", c, s.x, i);
}

void main() {
  char c = 'a';
  struct sstruct s = {1,2};
  int i = -1;
  fun (c, s, i);

  printf ("%c %d %d\n", c, s.x, i);
}
```

8 Bytes The struct is passed via the stack, everything else is passed via registers.

Question 3: Assembly, Stack Discipline, Calling Convention, and x86-64, *cont.* (15 points)
Part B: Conditionals and Loops (5 points)

Consider the following code:

```
Dump of assembler code for function loop:
   0x00000000000011ad <+0>:     endbr64
   0x00000000000011b1 <+4>:     push   %r14
   0x00000000000011b3 <+6>:     push   %r13
   0x00000000000011b5 <+8>:     push   %r12
   0x00000000000011b7 <+10>:    push   %rbp
   0x00000000000011b8 <+11>:    push   %rbx
   0x00000000000011b9 <+12>:    mov    %edi,%r13d               #%edi is 1st arg
   0x00000000000011bc <+15>:    mov    %esi,%ebp                #%esi is 2nd arg
   0x00000000000011be <+17>:    mov    %edx,%r12d               #%edx is 3rd arg
   0x00000000000011c1 <+20>:    cmp    %esi,%edi
   0x00000000000011c3 <+22>:    jge    0x1204 <loop+87>
   0x00000000000011c5 <+24>:    mov    %edi,%ebx
   0x00000000000011c7 <+26>:    lea    0xe36(%rip),%r14         # 0x2004
   0x00000000000011ce <+33>:    jmp    0x11d7 <loop+42>
   0x00000000000011d0 <+35>:    add    %r12d,%ebx
   0x00000000000011d3 <+38>:    cmp    %ebx,%ebp
   0x00000000000011d5 <+40>:    jle    0x1204 <loop+87>
   0x00000000000011d7 <+42>:    call   0x1090 <rand@plt>
   0x00000000000011dc <+47>:    test   %eax,%eax
   0x00000000000011de <+49>:    jg     0x1204 <loop+87>
   0x00000000000011e0 <+51>:    call   0x1090 <rand@plt>
   0x00000000000011e5 <+56>:    test   %eax,%eax
   0x00000000000011e7 <+58>:    jg     0x11d0 <loop+35>
   0x00000000000011e9 <+60>:    mov    %r12d,%r8d
   0x00000000000011ec <+63>:    mov    %ebp,%ecx
   0x00000000000011ee <+65>:    mov    %ebx,%edx
   0x00000000000011f0 <+67>:    mov    %r14,%rsi
   0x00000000000011f3 <+70>:    mov    $0x1,%edi
   0x00000000000011f8 <+75>:    mov    $0x0,%eax
   0x00000000000011fd <+80>:    call   0x1080 <__printf_chk@plt>
   0x0000000000001202 <+85>:    jmp    0x11d0 <loop+35>
   0x0000000000001204 <+87>:    mov    %r12d,%r8d
   0x0000000000001207 <+90>:    mov    %ebp,%ecx
   0x0000000000001209 <+92>:    mov    %r13d,%edx
   0x000000000000120c <+95>:    lea    0xdf1(%rip),%rsi         # 0x2004
   0x0000000000001213 <+102>:   mov    $0x1,%edi
   0x0000000000001218 <+107>:   mov    $0x0,%eax
   0x000000000000121d <+112>:   call   0x1080 <__printf_chk@plt>
   0x0000000000001222 <+117>:   pop    %rbx
   0x0000000000001223 <+118>:   pop    %rbp
   0x0000000000001224 <+119>:   pop    %r12
   0x0000000000001226 <+121>:   pop    %r13
   0x0000000000001228 <+123>:   pop    %r14
   0x000000000000122a <+125>:   ret
End of assembler dump.
```

***Hint*: Please be careful to understand the code. Answering these questions isn’t as simple as counting forward or backward jumps.**

Continued on next page.

3(B)(1) (2 points): Consider well-written C Language code. Is the loop shown above most representative of a `while () {….,` a `do { … } while()`, or a `for {} loop`? How do you know?

For loop the arguments given provide the start, stop, and increment values, which are then copied to other registers and used for that purpose. See for example: +35, +38, and +40.

3(B)(2) (1 points): Are there any 'break' statements in the loop? If so, at what line is/are the associated jump(s)? Give the line number(s) in the form <+23> or <+27> or, more generally, <+line_no>
.

Yes. 1. See +47, +49, and +51.

3(B)(3) (1 points): Are there any 'continue' statements in the loop? If so, at what line is/are the associated jump(s)? Give the line number(s) in the form <+23> or <+27> or, more generally, <+line_no>
.

Yes. 1. See +51, +56, and +58

3(B)(4) (1 points): How many ?-operators (ternary operators) are there? Explain your answer.

None. There are no conditional moves.

Continued on next page.

Part C: Switch statement (6 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 add 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.

```
Dump of assembler code for function foo:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     cmp    $0xa,%esi
   0x0000555555555170 <+7>:     ja     0x5555555551ad <foo+68>
   0x0000555555555172 <+9>:     mov    %esi,%eax
   0x0000555555555174 <+11>:    lea    0xe89(%rip),%rdx    # %rdx = 0x555555556004
   0x000055555555517b <+18>:    movslq (%rdx,%rax,4),%rax # Note that if ($rax == -3671) here
   0x000055555555517f <+22>:    add    %rdx,%rax,          # %rax = 0x00005555555551ad here
=> 0x0000555555555182 <+25>:    notrack jmp *%rax          # It may help to think about the
   0x0000555555555185 <+28>:    lea    0xa(%rdi),%eax      # delta from the above address and
   0x0000555555555188 <+31>:    ret                        # offset
   0x0000555555555189 <+32>:    lea    0x0(,%rdi,4),%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>:    sub    $0x1,%edi
   0x00005555555551ad <+68>:    lea    (%rdi,%rsi,1),%eax
   0x00005555555551b0 <+71>:    ret
End of assembler dump.
```

Commented [1]: Case 1

Commented [2]: Case 2

Commented [3]: Case 3

Commented [4]: Case 5

Commented [5]: Case 9, 10

Commented [6]: Default case, cases 0, 4, 6, 7, 8

Consider also the following memory dump.

```
(gdb) x/16lx 0x555555556004
0x555555556004: 0xffffff1a9      0xfffff181      0xfffff185      0xfffff18d
0x555555556014: 0xfffff1a9      0xfffff190      0xfffff1a9      0xfffff1a9
0x555555556024: 0xfffff1a9      0xfffff1a6      0xfffff1a6      0x000a6425
0x555555556034: 0x3b031b01      0x00000038      0x00000006      0xffffefec

(gdb) x/16d 0x555555556004
0x555555556004: -3671   -3711   -3707   -3699
0x555555556014: -3671   -3696   -3671   -3671
0x555555556024: -3671   -3674   -3674   680997
0x555555556034: 990059265       56      6       -4116
```

Continued on next page.

Part C: Switch statement, *cont.* (6 points)

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

`0x555555556004` **This is the address that is used as the base in +18**

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

`0x5555555551ad` **See +4 and +7. it goes there for negative inputs and inputs above 10 (i.e. 0xa)**

(3)(C)(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.

Yes.
Case 3 (-3699) falls through to case 5. It does the add at +40 before falling down to +43, which is the address of case 5 (-3696).
Cases 9 and 10 fall through to the default case. (+65 to +68)

Continued on next page.

Question 4: Caching, Locality, Memory Hierarchy, Effective Access Time (15 points)

Part A: Caching (12 points)

Given a model described as follows:

- Associativity: 2-way set associative
- Total size: 128 bytes (not counting metadata)
- Block size: 32 bytes/block
- Replacement policy: Set-wise LRU
- 8-bit addresses

4(A)(1) (1 point) How many bits for the block offset?

5. 2^5 = 32 bits 00000 - 11111 = 0 - (16+8+4+2+1 = 31)

4(A)(2) (1 point) How many bits for the set index?

128 bytes / (32 bytes/block) = 4 blocks; 4 blocks / (2 blocks/set) = 2 sets
2 sets need 1 bits for the set index

4(A)(3) (1 point) How many bits for the tag?

8 bits - 5 offset bits - 1 set bits = 2 tag bits

4(A)(4) (12 points): For each of the following addresses, please indicate if it hits, or misses, and if it misses, the type of miss:

Address	**Circle one (per row):**	**Circle one (per row):**
0xA1	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0xE1	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0xA2	[✓ Hit] Miss	Capacity Compulsory/Cold Conflict [✓ N/A]
0x21	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0xE2	Hit [✓ Miss]	Capacity Compulsory/Cold [✓ Conflict] N/A
0x81	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0xC1	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0x22	[✓ Hit] Miss	Capacity Compulsory/Cold Conflict [✓ N/A]
0x61	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0x41	Hit [✓ Miss]	Capacity [✓ Compulsory/Cold] Conflict N/A
0x82	Hit [✓ Miss]	[✓ Capacity] Compulsory/Cold Conflict N/A
0xC2	Hit [✓ Miss]	[✓ Capacity] Compulsory/Cold Conflict N/A

Continued on next page

Question 4: Caching, Locality, Memory Hierarchy, Effective Access Time (15 points)
Part B: Memory Hierarchy and Effective Access Time (3 points)

Imagine a computer system as follows:

- 2-level memory hierarchy (L1 cache, Main memory)
- L1: 10% miss rate
- Main memory: 50nS access time, 0% miss rate
- Memory accesses at different levels of the hierarchy **do not** overlap

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

What L1 cache access time is required for the overall effective memory access time to be 15nS? Show your work.

15ns = x + 0.1 * 50ns

15ns = x + 5ns

10ns = x

x = 10ns

Continued on next page.

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

Part A (2 points): When implementing the implicit list, you added a footer to each block. What operation required this addition and why?

Constant-time coalesce, so it could coalesce both left and right. The footer served as an implicit prev pointer for the successor block

Part B (2 points): When implementing implicit lists you were able to include an allocated/free bit, as well as possibly other status bits, without extending the size of the header, footer, or block overall. How were you able to do this and why didn't that cause other problem(s)?

The status bits were encoded by Xor into the low order bit(s) of the size. This worked because a minimum size was required for alignment, and possibly also by policy, so these bits for size purposes would always be zero and could be recovered by masking them.

Part C (2 points): If you got far enough in malloc lab (we talked about it even if you didn't), you were able to, in some cases, remove the footer from blocks in the implicit list. Under what circumstances could this be done, and why?

The footer could be removed from allocated blocks because allocated blocks can't be coalesced and coalescing is the only operation that required the implicit prev pointer provided by the footer.

Part D (2 points): After implementing the implicit list allocator, you augmented it to also provide explicit prev and next pointers. What biggest advantage was enabled by this, even before getting to segregated lists?

These pointers allow one to traverse only the unallocated blocks within the list. This can be much faster than needing to traverse all blocks, including allocated blocks. It is also true that these pointers enabled the list to be organized in different ways, but this benefit was smaller, at least until we got to segregated lists.

Part E (2 points): After implementing explicit lists, you organized them into a segregated list allocator. What was the biggest advantage was provided by this approach?

By organizing the free blocks into size classes, it made it possible to consider only blocks that were more likely to be an appropriate size, rendering moot the need to do things such as search for the best fit.

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 12 bits wide.
- The page size is 16 bytes.
- The TLB is 2-way set associative with 8 total entries.
- The TLB may cache invalid entries
- TLB REPLACES THE ENTRY WITH THE LOWEST TAG (NOT LRU)
- A single level page table is used

Part A: Interpreting addresses (3 points)

6(A)(1)(1 points): Please label the diagram below showing which bit positions are interpreted as each of the VPO (label as O) and VPN (Label as N). Leave any unused entries blank.

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

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

Bit	11	10	9	8	7	6	5	4	3	2	1	0
TLBI/ TLBT	T	T	T	T	T	T	I	I	X	X	X	X

6(A)(3)(1 points): How many entries exist within each page table?

8-bit virtual page numbers range from 00000000 to 11111111 for 2^8= 256 pages.

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

8 total entries/(2 entries/set) = *4 sets*

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

Part B: Hits and Misses (12 points)

- Shown below are the **initial** states of the TLB and page table.
- You may assume any entry **not shown** for the page table is **not valid.**
- We will grading only the trace, but you probably want to annotate the TLB and Page Table tables as you go to stay organized.

TLB

X=Invalid (for read or write, regardless of those bits), V=VALID, R=READ, W=WRITE:

Set	Tag	PPN	BITS	Additional scratch space for you
			X	**Set 0: 0-1,4 –>1-7,8 1-2 –> 3-6**
			X	**Set 1: 2-3,5**
			X	***Notation:***
			X	***Tag-Time,Time,Time,...***
			X	***x → y x replace by y***
			X	
			X	
			X	

Page Table

X=Invalid (for read or write, regardless of those bits), V=VALID, R=READ, W=WRITE:

Index/VPN	PPN	BITS	Scratch space for you
0x00	7	V-RW	
0x04	11	V-RW	
0x0A	100	V-R	
0x0C	20	X-RW	

Continued on next page.

Part B: Hits and Misses, *cont.* (12 points)

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.

Note: N/A or Not knowable means the choices do not apply or there is not enough information given. If you can not deduce a PPN from the information given, please write N/A for "PPN If Knowable"

Please complete the remaining columns

Subpart	**Operation**	**Virtual Address**	**TLB Hit or Miss?**	**Page Fault? Yes or No?**	**PPN, if Knowable Or, "Not knowable"**
1	R	0x00A	Hit [✓ **Miss**]	Yes [✓ **No**]	**7**
2	W	0x041	Hit [✓ **Miss**]	Yes [✓ **No**]	**11**
3	W	0x0AB	Hit [✓ **Miss**]	[✓ **Yes**] No	**100**
4	W	0x000	[✓ **Hit**] Miss	Yes [✓ **No**]	**7**
5	R	0x0A2	[✓ **Hit**] Miss	Yes [✓ **No**]	**100**
6	R	0x0CC	Hit [✓ **Miss**]	[✓ **Yes**] No	**Not knowable**
7	W	0x042	[✓ **Hit**] Miss	Yes [✓ **No**]	**11**
8	R	0x041	[✓ **Hit**] Miss	Yes [✓ **No**]	**11**

Continued on next page.

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

Part A (4 points):

Please consider the following code:

```
void main(){
   printf ("A"); fflush(stdout);

   if (!fork()) {
      printf ("B"); fflush(stdout);

      if (!fork()) {
         printf ("C"); fflush(stdout);
      }

      printf ("D"); fflush(stdout);
   }
   printf ("E"); fflush(stdout);
}
```

7(A)(1) (2 points): Draw the process graph, using the same notation we did in class, for the code above.

$$\begin{tikzcd}
 & & & \text{fork} \arrow[r] & C \arrow[r] & D \arrow[r] & E \\
 & \text{fork} \arrow[r] & B \arrow[r] & \text{fork()} \arrow[u, dotted, no head] \arrow[r] & D \arrow[r] & E & \\
A \arrow[r] & \text{fork()} \arrow[u, dotted, no head] \arrow[rr] & & E & & & z
\end{tikzcd}$$

7(A)(2) (1 points): Give one valid output for the program above.

AEBDECDE

7(A)(3) (1 points): Give one invalid output for the program above that has an ordering problem involving B, C, and/or D.

AEBDECED

Continued on next page.

Question 7: Process Representation and Lifecycle + Signals and Files, *cont.* (10 points)
Part B (6 points):

Please consider the following code:

```
int main(int argc, char* argv[]) {
 char buffer[7] = "abcdef";
 char buffer2[7];

  // Assume "file.txt" is initially non-existent or empty.
  int fd0 = open("file.txt", O_RDWR | O_CREAT, 0666);
  int fd1 = 10;

  write(fd0, buffer, 2);
  dup2 (fd0,fd1);  // int dup2(int oldfd, int newfd); copies oldfd over newfd
  write(fd1, buffer, 2);

  if (!fork()) {
    write(fd1, "P", 1);
    write(fd0, buffer+3, 3);

    fd1 = open("file.txt", O_RDWR | O_CREAT, 0666);

    write(fd1, "X", 1);

    write(fd0, "A", 1);
 } else {
    wait(NULL);
    write(fd1, "C", 1);
    write(fd0, buffer, 3);
 }

 return 0;
}
```

7(B)(1) (2 points): What is the content of the output file after this code completes?
XbabPdefACabc

7(B)(2) (2 points): If the child process was just about to "return 0", how many entries are there in the system-wide open file table related to this code (ignore stdin, stdout, stderr), assuming open file table garbage collection is done only when program terminates?

Two. One open() in the parent and one open() in the child. There has been one open at this point in time. Forks don't create new file table entries.

7(B)(3) (2 points): Ignoring stdin, stdout, and stderr, what is the greatest number of file descriptor table entries in use at one time across all parent and child process(es)?
Five. The parent opens one, the dup2s one, the child gets copies of these (2+2 = 4), then the child opens another (4 + 1 = 5). It doesn't matter that the local variable name used to hold the returned fd in the child is also fd1. It just matters that the open created a new one.

Continued on next page.

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

Part A (6 points): Correct

Consider the following C code. Assume that both threads have been spawned and are running concurrently.

8(A) Code

```
 1.  // Based upon:
 2.  // Harris Hyman, "Comments on a problem in concurrent programming control",
 3.  // Communications of the ACM, v.9 n.1, p.45, Jan. 1966.
 4.
 5.  int flag[2] = {0,0};
 6.  int turn = 0;
 7.  int counter = 0;
 8.
 9.  void *threadMain (void *arg) {
 10.    int i = (int) arg;
 11.
 12.    while (1) { // Main work loop
 13.
 14.       flag[i] = 1; // Indicate this thread wants access to critical section
 15.
 16.       while (turn != i) { // While it isn't this thread's turn
 17.          while (flag [1-i] == 1) // While the other thread is using the critical section
 18.          ; // Just spin
 19.
 20.          turn = i; // Claim this thread's turn, excluding other thread
 21.       }
 22.
 23.       // Critical section
 24.       counter++;
 25.
 26.       // Release critical section, allowing the other thread to claim critical section
 27.       flag[i] = 0;
 28.
 29.    } // Back to the top of the main work loop
 30. }
 31.
 32. void main() {
 33.    pthread_t thread0, thread1;
 34.    int ret0, ret1;
 35.
 36.    ret0 = pthread_create( &thread0, NULL, threadMain, (void*) 0);
 37.    ret1 = pthread_create( &thread1, NULL, threadMain, (void*) 1);
 38.
 39.    return (ret0 + ret1) // 0 on pthread_create() success, non-zero on pthread_create() fail
 40. }
```

8(A)(1) ~~(3 points)~~ Consider the code above. In thinking about it you might consider the following initial sequence of events and imagine what might come next:

- Imagine that the 0th thread is dispatched first and executes through the end of line 17 before the scheduler preempts it
- Imagine then that the 1st thread runs through line 24.

Continued on next page.

8(A)(1) (~~1 point;~~ 2 points) Does the above code provide correct concurrency control for the identified critical section? Circle either “Yes” or the type of violation:

Yes

Deadlock

Race

Starvation

Other type of Progress violation

8(A)(2) (~~2 points~~ → 4 points) If this code provides correct concurrency control, please explain how the critical section is protected. If not, please illustrate, using line numbers, a violation and explain the nature and cause of the violation.

IRace: Thread 1 first can get to line 19 because at this point turn is 0 (not 1) and flag[0] is not on yet. Then before thread 1 can change turn, thread 0 executes and it can skip the entire outer while (turn != i) loop because turn is still 0. Only after this does thread 1 change the turn variable to 1 and now both threads are executing the critical section of code concurrently.

Starvation: A thread could quickly change flag[i] to 0 and 1 (lines 27 and 14) and because it isn’t marked as volatile, the other thread could get stuck on line 17 and think the flag is always set.

Question #8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW, *cont.* (15 pts)

Part B (9 points): Concurrency Control

Consider a situation in which a threaded network server accept()s connections from clients and stages them for handling by a fixed-sized pool of worker threads, which in turn interact with the client.

To stage a connection for handling by a thread, the server places the file descriptor of the accept()ed client session into the element of an array associated with an available worker thread and signals/wakes-up the worker thread using a concurrency control primitive.

If all worker threads are busy, the server needs to wait before staging the work and looping back to accept() more client connections.

Your task is to complete the C-like pseudocode code that follows, maximizing concurrency while enduring correctness and approximate fairness. What is provided is just just pseudocode. Don’t let details unrelated to the concurrency control problem distract you. **Read the provided code and comments: They are very important.**

The only concurrency control primitive available is the semaphore type as shown below:

- `sem_t // The data type for a semaphore`
- `sem_init (sem_t, unsigned int initial_value)`
- `sem_p(sem_t)`
- `sem_v(sem_t)`

Continued on next page.

Question #8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW, *cont.* (15 pts)
Part B (9 points): Concurrency Control, *cont.*

```
#define NUM_WORKER_THREADS 5
#define AVAILABLE -1

int connections[NUM_WORKER_THREADS] =
                        { AVAILABLE, AVAILABLE, AVAILABLE, AVAILABLE, AVAILABLE };


// Declare any variables you want, including semaphores, and perform any
// initialization you want here
// Hint: You can use arrays of semaphores.

sem_t bufferLock;
sem_t availableSem;

sem_t ready[NUM_WORKER_THREADS];

sem_init(buffer_lock, 1);
sem_init(availableSem,NUM_WORKER_THREAD);

for (int index=0; index < NUM_WORKER_THREADS; index++)
  sem_init(ready[index],1);


// There is just one of these in the whole process. It accepts new connections and
// hands them off to threads
int serverMain (int sockfd) {
  int clientfd;

  while (clientfd = accept(sockfd, NULL, NULL) 0 {

      // Write code here that causes the server to place the connection with an
      // available worker and to wait until one is available, if none are available

      sem_P(availableSem); // This should really be before the accept, but…
      sem_P(buffer_lock);

      for (int index=0; index<NUM_WORKER_THREADS; index++)
        if (ready[index] == AVAILABLE]) break;
      ready[availableSem] = clientfd;

      sem_V(buffer_lock);
  }
}
```

Continued on next page.

Question #8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW, *cont.* (15 pts)
Part B (9 points): Concurrency Control, *cont.*

```
// This is the body of each individual client worker thread. It interacts with
// a client it is given via a file descriptor in the connections[] array
// There are NUM_WORKER_THREAD of these threads within the process
// Each has a unique threadID from 0 – (NUM_WORKER_THREAD -1)
int clientThreadMain(int myThreadID) {

    while (1) {
      // Do anything you needed for concurrency control before interacting
      // with the client here.



      // This function does all of the interaction with the assigned client
      interactWithClient(connections[myThreadID];)

      // Do anything needed for concurrency control after interacting with the
      // client and before marking self as available for another client here



      // Mark self as available for another client session

      sem_P(buffer_lock);
      ready[myThreadID] = AVAILABLE;
      sem_V(buffer_lock);



      // Do anything needed for concurrency control after
      // marking self as available for another client and before looping back to
      // start over waiting for and handline a new client here.

      sem_V(availableSem);



   }
}
```

The end! Exam done! Good work!