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 2022 Final Exam

Monday, May 2, 2022

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 (except for 2 double-sided note sheets).
- You may not use any electronic devices or anything other than what we provide, your notes sheets, 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)

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

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

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

	Machine 1: 4-bit w/2s complement signed	**Machine 2: 6-bit w/2s complement signed**
Binary representation of -6 decimal	*Soln: 1010*	*Soln: 111010*
Binary representation of 10 decimal	*Soln: UNABLE*	
Binary representation of -Tmin	*Soln: 1000*	
Integer (Decimal) value of (-4 - 6)	*Soln: 6*	

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 7 bits
  - There is 1 sign bit *s*.
  - There are *k* = 3 exponent bits.
  - You need to determine the number of fraction bits.
- Format B:
  - There are 8 bits
  - There is 1 sign bit s.
  - There are n = 3 fraction bits.

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

	Format A	**Format B**
Total Number of Bits (Decimal)	**7**	**8**
Number of Sign Bits (Decimal)	**1**	**1**
Number of Fraction Bits (Decimal)	*Soln: 3*	**3**
Number of Exponent Bits (Decimal)	**3**	*Soln: 4*
Bias (Decimal)	*Soln: 3*	*Soln: 7*
Decimal value or interpretation of bit pattern: 1111111	*Soln: -NaN*	
Decimal value or interpretation of bit pattern: 01111000		*Soln: Inf*
1000010 **(Decimal value, unrounded, use fractions)**	*Soln: -1/16*	
00101101 **(Decimal value, unrounded, use fractions)**		*Soln: 1-5/8*
((-1.0*NaN) == -NaN)		*Soln:* *False*
`// x and y are negative floats` `((x + y) < 0)`	Circle one: **Soln: Always true** Always false It depends	

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

Part A: Array Sizes (1 points)

Consider the following definitions in an x86-64 system with 8-byte pointers and 4-byte `int`s:

Definition A	Definition B
`int numbersA[3][2][2];`	`int *numbersB = numbersA;`

2(A)(1) (0.5 point): How many bytes are allocated to `numbersA`? (Write "UNKNOWN" if not knowable).
Hint: Think sizeof()

Soln: 48 bytes

2(A)(2) (0.5 point): How many bytes are allocated to `numbersB`? (Write "UNKNOWN" if not knowable).
Hint: Think sizeof()

Soln: 8-bytes

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

Part B Array Allocation and Initialization (5 points):

`numbersC` is intended to be used as an array of `int`s. Please complete the following code. Please read all of "Part B" before answering any part of "Part B".

2(B)(1) (0.5 point)

```
// Complete EXACTLY ONE of the definitions below. Your choice
#define TYPE1 _________________
typedef _________________ TYPE1;
```

Soln:

```
#define TYPE1 int**
typedef int** TYPE1;
```

2(B)(2) (0.5 point)

```
// Complete EXACTLY ONE of the definitions below. Your choice
#define TYPE2 _________________
typedef _________________ TYPE2;
```

Soln:

```
#define TYPE2 int*
typedef int* TYPE2;
```

2(B)(3) (2 points)

```
// This function should allocate space for intArray
// and initialize the elements of intArray to 0, 1, 2, 3, ...
// in a way that allows the array to be used outside of this
// function after this function is called.
// It should return 0 upon success and non-zero upon failure
int initArray(TYPE1 intArray, size_t length) {


  // Soln:
  *intArray = (int *) malloc (length * sizeof(int));
  if (!intArray) return -1;


  for (int index=0; index < length; index++)
    (*intArray)[index] = index;

  return 0;


}
```

2(B)(4) (2 points)

```
// This function should free the allocation(s) made in initArray(...) above.
// It should return 0 upon success and non-zero upon failure

int freeArray(TYPE2 intArray) {

  // Soln:
  if (!intArray) return -1;
  free (intArray);
  return 0;

}
```

Part C: 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 {
  short s1;     // 2-byte type
  double  d;   //  8-byte  type
  short s2;
} partB;
```

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

```
sizeof(struct partB)
```

Soln:
ssXXXXXXddddddddssXXXXXX
24 bytes

2(C)(1) (2 points): Rewrite the struct above to minimize its size after alignment-mandated padding:

Soln: Answers may vary but should all be the same size as this:

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

ssssXXXXdddddddd
16 bytes

Question 3: Assembly, Stack Discipline, Calling Convention, and x86-64 ISA

Part A: Loops and Calling Convention (7 points)

Consider the following code:

```
.LC0:
        .string "count: %d\n"
        .text
function:
        pushq   %r14
        movl    %edx, %r14d
        pushq   %r13
        movl    %esi, %r13d
        pushq   %r12
        xorl    %r12d, %r12d
        pushq   %rbp
        movl    %edi, %ebp
        pushq   %rbx
        cmpl    %edx, %edi
        jge     .L3
.L2:
        movl    %r13d, %ebx
        testl   %r13d, %r13d
        jns     .L6
        jmp     .L7
.L16:
        subl    $1, %ebx
        addl    $1, %r12d
        cmpl    $-1, %ebx
        je      .L7
.L6:
        testb   $1, %bl
        je      .L16
        movl    $88, %edi
        subl    $1, %ebx
        call    putchar@PLT
        cmpl    $-1, %ebx
        jne     .L6
.L7:
        addl    $1, %ebp
        cmpl    %ebp, %r14d
        jne     .L2
.L3:
        movl    %r12d, %edx
        leaq    .LC0(%rip), %rsi
        movl    $1, %edi
        xorl    %eax, %eax
        call    __printf_chk@PLT
        popq    %rbx
        movl    %r12d, %eax
        popq    %rbp
        popq    %r12
        popq    %r13
        popq    %r14
        ret
```

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

Soln: 2. There are two backward jumps.

3(A)(2) (2 points): Which of the following exist among the loops in the code. Circle all that apply:

Sequential loops (one loop after another loop)
Soln: Nested loops (one loop within another loop)
Staggard loops (one loop overlapping another loop without being nested)
None of the above

3(A)(3) (1 points): How many arguments does this function receive (and use)?

Soln: 3

3(A)(4) (2 points): For each argument you listed, please indicate either (a) which **specific** register was used to pass it in, or (b) that it was sourced from the stack (**you don't need to give the address**). Please leave any extra blanks empty (*Hint:* You won't need all of them).

Argument	**Specific register or "Stack"**
1st	*Soln: %edi*
2nd	*Soln: %esi*
3rd	*Soln: %edx*
4th	*Soln: {blank}*
5th	*Soln: {blank}*

Consider the following function activation. Consistent with your answer to the question above, it includes more arguments that the function actually requires. Please ignore any extra arguments.

```
function(0, 10, 10, 10, 10, 1);
```

3(A)(2) (2 points): How many times does the inner-most loop run?
Hint: If the inner-most loop is nested, you may need to consider the loops in which it is nested.

Solution: 110 times (10x outer, 11x inner)

Part B: Conditionals (8 points)

Consider the following code:

```
Dump of assembler code for function foo:
   0x000000000040052d <+0>:     cmp    $0x5,%esi
   0x0000000000400530 <+3>:     ja     0x400558 <foo+43>
   0x0000000000400532 <+5>:     mov    %esi,%eax
   0x0000000000400534 <+7>:     jmpq   *0x400620(,%rax,8)
   0x000000000040053b <+14>:    lea    0xa(%rdi),%eax
   0x000000000040053e <+17>:    retq
   0x000000000040053f <+18>:    mov    $0x2,%edi
   0x0000000000400544 <+23>:    mov    $0x55555556,%edx
   0x0000000000400549 <+28>:    mov    %edi,%eax
   0x000000000040054b <+30>:    imul   %edx
   0x000000000040054d <+32>:    sar    $0x1f,%edi
   0x0000000000400550 <+35>:    mov    %edx,%eax
   0x0000000000400552 <+37>:    sub    %edi,%eax
   0x0000000000400554 <+39>:    retq
   0x0000000000400555 <+40>:    and    $0x1,%edi
   0x0000000000400558 <+43>:    lea    (%rdi,%rsi,1),%eax
   0x000000000040055b <+46>:    retq
End of assembler dump.
```

Consider also the following memory dump:

```
(gdb) x/14gx 0x400610
0x400610:       0x0000000000020001      0x0000000000000000
0x400620:       0x0000000000400555      0x000000000040053b
0x400630:       0x000000000040053f      0x000000000040053f
0x400640:       0x0000000000400558      0x0000000000400544
0x400650:       0x0000003c3b031b01      0xfffffdb000000006
0x400660:       0xfffffdf000000088      0xfffffedd00000058
0x400670:       0xffffff0c000000b0      0xffffff40000000c8
```

(3)(B)(1) (1 points): How many “if statements” are likely present in the C Language code from which this assembly was compiled? At what address of the assembly code shown above does each occur?

This code was compiled from C Language code containing a switch statement. **Please do not include any “if statement” present in the assembly that is likely part of the switch statement** in the original C code, i.e. do not count any “if statement” that is used to manage one or more “cases” of a “switch statement”.

Soln:
0
There is one forward jump, which is a candidates 0x400530. But, it is considering the switch control variable, comparing it to a bound, and jumps to the default case. So it is likely handing a “case” of the switch, specifically the default case.

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

Soln: 0,1,3,5
Negative values and values above 5 are managed by the default case. Note that negatives look like large integers when compared using unsigned "ja".

(3)(B)(3) (1 point): Is there a default case? If so, at what address does it begin? How do you know?

Soln: Yes. `0x400558`. *It is used cases larger than 4 and larger than (but not) 5.*

(3)(B)(4) (2 points): Which case(s), if any, share exactly the same code? How do you know?

Soln: Cases 2 and 3. They have the same pointer in the jump table.

(3)(B)(5) (2 points): Which case(s), if any, fall through to the next case after executing some of their own code? How do you know?

Soln: Cases 2/3 and 5, and cases 0 and default.

If we look at the code block beginning with where the 3rd entry in the jump table points, it overlaps the code block pointed to by the 5th entry in the jump table without a jump or return to prevent it from falling through.

The same is true if we look at the code beginning with the 0th entry in the jump table and the the default case, that follows.

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

Part A: Caching (8 points)

Given a model described as follows:

- Associativity: 4-way set associative
- Total size: 128 bytes (not counting meta data)
- Block size: 8 bytes/block
- Replacement policy: Set-wise LRU
- 8-bit addresses

4(A)(1) (1 point) How many bits for the block offset?
Soln: 8 bytes = 3 bits to index

4(A)(2) (1 point) How many bits for the set index?
Soln: (128 bytes) / (8 bytes/block) / (4 blocks/set) = 4 sets; 2 bit indexes 4 sets.

4(A)(3) (1 point) How many bits for the tag?
Soln: (8 bit address) - (3 bits for block offset) - (2 bit for set index) = 3 bits left over for tag

4(A)(4) (5 points, ½ point 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):**			
0XC1	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X65	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0XE5	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X45	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X61	***Hit***	*Miss*	*Capacity*	*Cold*	*Conflict*	***N/A***
0XC1	***Hit***	*Miss*	*Capacity*	*Cold*	*Conflict*	***N/A***
0XD7	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X27	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0XE1	Hit	***Miss***	Capacity	*Cold*	**Conflict**	N/A
0XC1	**Hit**	*Miss*	Capacity	Cold	*Conflict*	**N/A**

Part B: Locality (4 points)

4(B)(1): Consider the following code running on a host with 4-byte integers and 32-byte cache blocks. Further assume that SIZE1 and SIZE2 are extremely large.

```
int array[SIZE1][SIZE2];
int sum=0;
for (int outer=0; outer<SIZE1; outer++)
  for (int inner=0; inner<(SIZE2-OFFSET); inner+=STEP)
   sum += array[outer][inner] + array[outer][inner+OFFSET];
```

4(B)(1)(a) (2 points)

Consider the impact upon locality of increasing STEP from 1 to 2 and of increasing OFFSET from 1 to 2. If either would likely have less impact upon locality, please explain which it is and why. If the impact upon locality is likely the same, please explain why.

Soln: I'd expect that changing STEP would be worse. It affects both array accesses, vs just the 2nd one.

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

```
short array[ROWS][COLS];
int sum=0;
for (int col=0; col<COLS; col++)
  for (int row=0; row<(ROWS-1); row++)
    sum += array[row][col]+ array[row][col+1];
```

Imagine an array extremely large in all dimensions, a short size of 2 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?

Soln: 50%. There is no locality across accesses to the inner loop as the outer loop is skipping around. The 1st access misses, but the 2nd access hits since they are next to each other and cache aligned.

Part C: Memory Hierarchy and Effective Access Time (3 points)

Imagine a computer system as follows:

- 3-level memory hierarchy (L1 cache, L2 cache, Main memory)
- 3.5nS effective memory access time
- L1: 1nS access time, 10% miss rate
- L2: 20nS access time
- Main memory: 100nS 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 is the miss rate for the L2 cache?

3.5nS = 1nS + (0.1)20nS + (0.1)(MRL2)(100ns)
*0.5nS = 0.1*MRL2*100nS*
*0.5nS = MRL2 * 10nS*
MRL2 = 0.5nS / 10nS = 5/100 =5%
MRL2=5%

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

Consider the following code series of malloc's and free's:

```
ptr1 = malloc(2);
free (ptr1);
ptr2 = malloc(24);
ptr3 = malloc(8);
free(ptr2);
free(ptr3)
ptr4 =
malloc(40);
ptr5 =
malloc(8);
free (ptr4);
ptr6 = malloc(16);
```

And a malloc implementation as below:

- Explicit list
- Best-fit
- Headers of size 8 bytes
- Footer size of 8-bytes
- Every block is always constrained to have a size a multiple of 8 (In order to keep payloads aligned to 8 bytes).
- No minimum block size (beyond what is structurally needed)
- If no unallocated block of a large enough size to service the request is found, sbrk is called to grow the heap enough to get a new block of the smallest size that can viably service the request
- The heap is unallocated until it grows in response to the first malloc.
- Constant-time coalescing is employed.
- The heap never shrinks

NOTE: You do NOT need to simplify any mathematical expressions. Your final answer may include multiplications, additions, and divisions.

4(A) (2 points) After the given code sample is run, how many total bytes have been requested via sbrk? In other words, how many bytes are allocated to the heap? Draw a figure showing the heap and where each ptr is located.

Soln:

```
ptr1 = malloc(2);   // 8+8+8,    HS=24
free (ptr1);        //           HS=24,   FL=24
ptr2 = malloc(24);  // 8+24+8,   HS=64,   FL=24
ptr3 = malloc(8);   // 8+8+8,    HS=64,   FL=x
free(ptr2);         //           HS=64,   FL=40
free(ptr3);         //           HS=64,   FL=64
ptr4 = malloc(40);  // 8+40+8,  HS=64,  FL=x
ptr5 = malloc(8);   // 8+8+8,    HS=88,  FL=x
free (ptr4);        //           HS=88,  FL=64
ptr6 = malloc(16);  // 8+16+8     HS=88,  FL=32
```

88 bytes requested via sbrk
Heap: {ptr6:8+16+8}{free:32}{ptr5:8+8+8)

5(B) (2 points) How many of those bytes are used for currently allocated blocks (vs currently free blocks), including internal fragmentation and header information?

Soln: 56 bytes for allocated blocks

5(C)(2 points) How much internal fragmentation is there due to padding (Answer in bytes)? (*Hint: Free blocks have no internal fragmentation).*

Soln: 0B

5(D)(2 points) How much internal fragmentation is there due to headers and footers (Answer in bytes)? (*Hint: Free blocks have no internal fragmentation).*

Soln: 32B

5(E)(2 points) Imagine that the user wrote a 20-character string to the buffer allocated ptr6. What would be the most likely result? And why? Circle the most likely result and then explain below.

A. It would be correct
B. It would be incorrect code, but would likely work correctly in this environment
C. It would likely work until the next huge allocation.
D. It would likely work until the next coalesce or very small allocation.

Soln:

(D) It would likely over-write the footer. This wouldn't necessarily be noticed until a coalesce or until an allocation of the next block. The next block is a small 8-byte block.

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 16 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 256 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)(1 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	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PPN/ PPO					*N*	*N*	*N*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*

6(A)(2)(1 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	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
VPO/ VPN	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*
TLBI/ TLBT	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*I*								

6(A)(3) (1 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.

Soln: One entry per page. 8 bits per page number means 256 pages.

6(A)(4) (1 points): How many sets are in the TLB?
Soln: 2. 8 total entries, 4 entries/set = 2 sets.

Part B: Hits and Misses (12 points)

Shown below are the **initial** states of the TLB and **partial** page table.
TLB (V=VALID, R=READ, W=WRITE, NR=Not Resident, e.g. swapped):

Index	Tag	PPN	BITS	Scratch space for you
0	66	2	V-R	
0	28	1	V-RW	
0	7D	3	V-R	
0	2D	C	NR	
1	79	4	NR	

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

Index/VPN	PPN	BITS	Scratch space for you
50	1	V-RW	
5A	C	NR	
AA	A	V-RW	
CC	2	V-R	
F0	B	V-RW	
F3	4	NR	
FC	3	V-READ	

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.

Operation	**Virtual Address**	**TLB Hit or Miss?**	**Page Table Hit or Miss?**	**Page Fault? Yes or No?**	**PPN If Knowable**
Read	CC01	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* *No* *Not knowable*	***2***
Read	F301	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	***Yes*** *No* *Not knowable*	
Read	5010	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* ***No*** *Not knowable*	***1***
Read	5011	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* ***No*** *Not knowable*	***1***
Write	F0AC	*Hit* ***Miss*** *Not knowable*	***Yes*** *No* *Not applicable*	*Yes* ***No*** *Not knowable*	***B***
Write	FCBC	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	***Yes*** *No* *Not knowable*	***3***
Read	5A56	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	***Yes*** *No* *Not knowable*	
Write	CC23	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	***Yes*** *No* *Not knowable*	***2***
Write	5045	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* ***No*** *Not knowable*	***1***
Read	FC12	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* ***No*** *Not knowable*	***3***
Write	AACC	*Hit* ***Miss*** *Not knowable*	***Yes*** *No* *Not applicable*	*Yes* ***No*** *Not knowable*	***A***
Read	F001	***Hit*** *Miss* *Not knowable*	*Yes* *No* ***Not applicable***	*Yes* ***No*** *Not knowable*	***B***

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

Part A (3 points):

Please consider the following code:

```
void main(){
  int pid1,


  printf ("A"); fflush(stdout);
  if (pid1=fork()) {
     printf ("B"); fflush(stdout);
     wait(NULL);
     printf ("C"); fflush(stdout);
   } else {
     printf ("D"); fflush(stdout);
     if (fork()) {
       printf ("E"); fflush(stdout);
     } else {
       printf ("F"); fflush(stdout);
     }
   }
 }
```

7(A)(1) (1 points): Give one possible output string

Soln: Many are possible, e.g ABDEFC

7(A)(2) (1 points): Give one output string that has the correct output characters (and number of each character), but in an impossible order.

Soln: Many possible, e.g. ABFEDC

7(A)(3) (1 points): Why can't the output you provided in 7(A)(2) be produced? Specifically, what constraint(s) from the code does it violate?

Soln: They need to describe a violated ordering constraint.

Continued on next page.

Part B (3 points):

Please consider the following code:

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

int main(int argc, char* argv[]){
  char buffer[4] = "abc";

  // Assume "file.txt" exists but is initially empty

  int fd0 = open("file.txt", O_RDWR);
  int fd1 = open("file.txt", O_RDWR);
  int fd2 = 0;

  write(fd0, buffer+1, 2);
  read(fd1, buffer, 1);
  dup2 (fd1,fd2);  // int dup2(int oldfd, int newfd);

  read(fd2, buffer, 1);
  write(fd0, buffer, 3);

  read(fd2, buffer, 1);

  return 0;
}
```

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

Soln: bccbc

7(B)(2) (1 points): How many entries are there in the system-wide open file table related to this code?

Soln: 2, one from each open

Continued on next page.

7(B)(3) (1 points): For each listed file descriptor variable, identify the file descriptor table entry pointed to by each file descriptor variable. Name the file descriptor entries FT1, FT2, FT3, FT4, etc.

File descriptor variable	**File table entry, e.g. FT0, FT1, FT2, FT3**
fd0	***Soln: FT0***
fd1	***Soln: FT2***
fd2	***Soln: FT2***

Continued on next page

Part C (4 points):

Please consider the following code:

```
#include <stdio.h>
#include <wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

int count = 0;

void inthandler(int sig){
    count++;
    printf("SIGINT received. Sig count = %d\n", count);
    return;
}

void childhandler(int sig) {
    int status;

    count++;
    printf("SIGCHLD received. Sig count = %d\n", count);

    while (waitpid(0, &status, WNOHANG)>0))
    ;

    return;
}

void main() {
    pid_t pid; // pid of child process

    signal(SIGINT, inthandler);
    signal(SIGCHLD, childhandler);

    pid = fork();
    if(!pid){
      kill(getppid(), SIGINT);
      exit(5); // Exit status is 5
    }

    sleep(5);

    printf("count = %d\n", count);
    exit(0);  // Exit status is 0
}
```

7(C)(1) (2 points): Is it possible for this code to deadlock? If so, how?

Soln: Yes. printf() is not async-signal safe.

7(C)(2) (1 points): There a critical (problematic shared) resource (variable)? What is it?

Soln: count

7(C)(3) (1 points): Assuming that deadlock is either impossible or doesn't happen, is it possible for the critical resource you identified above to suffer from concurrency, i.e. hold a wrong or inconsistent value, etc? If so, how could that happen?

Soln: Yes. The handlers could read the same value, update it, and lose the update upon writing. The value could also change between the time main puts it into the buffer and the time that buffer is printed.

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

In the crazy world in which we live, CMU decides to adopt two different salad and toppings bars, one for people wearing masks and one for unmasked people as follows:

- CMU expects that fewer people will be wearing masks at lunch time than otherwise, so they arrange:
  - Space for three (3) people at the “masked bar”
  - Space for five (5) people at the “unmasked bar”
- Because of the usual grocery store confusion, customers can’t really tell how long the lines are for either area, so people won’t put on or take off their masks to eat faster.
- Each customer may choose to wait for whichever area they wish
- Managers want to be able to check to see how long each line is at any time, so that they can potentially adjust the sizes of each bar later.

Please model this situation as C-like pseudo-code with proper concurrency control via semaphores. Legal semaphore operations are as follows:

- sem_init (sem_t, count)
- sem_p (sem_t)
- sem_v (sem_t)
- where sem_t is a semaphore variable type.

Specifically, please write the pseudocode for the following methods:

```
// Constants to let us name/identify each area
// These could just as easily be an enum or #defined.
// These aren’t counts or precedence/priority. They are just identifiers
const int MASKED = 0;
const int UNMASKED = 1;

// Declare and initialize any needed semaphores
// and/or shared variables here.
// You can assume they are global and shared.
void initialize() {
  // Hint: Think about what the type(s) of resources are and how
  // many instances of each type there are. Find a way to account
  // for each of those pool(s) of resources

  // these declarations are given
  // See the print_line_lengths() function for one example of their use
  // they track the number of people waiting for each area.
  int masked_line = unmasked_line = 0;

  sem_p maskedSem, unmakedSem, countMutex;
  sem_init(maskedSem, 3);
  sem_init (unmaksedSem, 5);
  sem_init (countMutex,1);


}
```

```
// Customers call this to wait for the salad bar area of their choice
void waitForSaladBar(int maskedOrUnmasked) { // Remember MASKED and UNMASKED?

 P(count_mutex);
 if (MASKED == maskedOrUnmasked)
   masked_line++;
 else
   unmasked_line++
 V(count_mutex)


 if (MASKED == maskedOrUnmasked)
    P(maskedSem);
  else
    P(unmaskedSem);

 P(count_mutex);
 if (MASKED == maskedOrUnmasked)
   masked_line--;
 else
   unmasked_line--;
 V(count_mutex)

}


// Customers call this when done with the salad bar
void doneWithSaladBar(int maskedOrUnmasked) { // Remember MASKED and UNMASKED?
  // Hint: Which pool(s) of resources are being given up here?
  // What needs to happen to make them available?
  if (MASKED == maskedOrUnmasked)
    V(maskedSem);
  else
    V(unmaskedSem);
}


// Managers call this to print line lengths.
void print_line_lengths() {
  // Your code here
  P(count_mutex)

  printf ("Masked line: %d\n", masked_line);
  printf ("Unmasked line: %d\n", unmasked_line);
  fflush(sdout);

  // Your code here
  V(count_mutex)
}
```