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, Fall 2021 Final Exam

Makeup-Backup

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 8-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: 8-bit w/2s complement signed**
Binary representation of 11 decimal	*Soln: UNABLE*	*Soln: 0000 1011*
Binary representation of -6 decimal		*Soln: 1111 1010*
Binary representation of -Tmin	*Soln: 1000*	
Binary representation of -1 decimal		*Soln: 1111 1111*

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* = 3 exponent bits.
  - You need to determine the number of fraction bits.
- Format B:
  - There are 6 bits
  - There is 1 sign bit s.
  - There are n = 3 fraction bits.

Fill in the empty (non grayed-out) boxes on the net page as instructed

	Format A	Format B
Total Number of Bits (Decimal)	**6**	**6**
Number of Sign Bits (Decimal)	**1**	**1**
Number of Fraction Bits (Decimal)	*Soln: 2*	Soln: 3
Number of Exponent Bits (Decimal)	**3**	*Soln: 2*
Bias (Decimal)	*Soln: 3*	*Soln: 1*
-Infinity (Binary bit pattern)	*Soln: 111100*	
101010 (Decimal value, unrounded)		Soln: E=(1-1)=0 -1-1/4 x 2^0 *-5/4 = -1-1/4*
000010 (Decimal value, unrounded)	*Soln:* *E = (1 - 3) = -2* *⅛ = 1/2 x 2^{-2}*	
0 (Binary bit pattern)		*Soln:* *000000*
`// x,y, and z are floats` `(x*y/z) == (x/z*y)`	Circle one: *Soln: Depends* Always equal Always unequal It depends	

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

Part A: Arrays (5 points)

Consider the following code running in an x86-64 system with 8-byte pointers and 4-byte `int`s. Assume it successfully prints each and every element of the numbers array.

```
void fn(int **numbers) {
  for (int row=0; row < 3; row++)
    for (int col=0; col < 2; col++)
      printf ("numbers[%d][%d]=%d", row, col, numbers[row][col]);
}
```

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

Soln: 8-Bytes (The size of a pointer)

2(A)(2) (1 point): What is the minimum size of the memory allocation directly referenced by `numbers`?

Soln: 24-bytes (The size of 3 pointers, one for each element of the array directly referenced by `numbers`*).*

2(A)(3) (3 points) Write C Language code to free all dynamic memory associated with numbers. It is not necessary to set the `numbers` pointer to NULL once done.

```
void fn(int **numbers) {

  // Soln
  for (int row=0; row < 3; row++)
    free (numbers[row])
  free (numbers)
}
```

Continued on next page.

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

Part B: Structs, Unions, and Alignment (5 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 c1;      // 1-byte type
  int i;        // 4-byte type
  char c2;
} partB;
```

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

```
sizeof(struct partB)
```

Soln:
cXXXiiiicXXX
12 bytes

2(B)(2) (1 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 {
  char c1;       // 1-byte type
  char c2;
  int i;         // 4-byte type
} partB;
```

2(B)(3) (1 points): How many bytes are required for the struct you designed for 2(B)(2) above?

Soln: 8-bytes

Continued on next page.

2(B)(3) (1 points): How many bytes are required for the following union?
Hint: Think sizeof()

```
union {
  int i;      // 4-byte type
  short s;    // 2-byte type
  long l;    // 8-byte type
} u;
```

Soln: 8-bytes, the size of the largest type

2(B)(4) (1 points): Given the definition above and the code below, and assuming an x86-64 host, is the code below guaranteed to print the same value twice? Why or why not?

```
union u;

scanf("%d", &u.i);

printf ("%d\n", u.i);
printf ("%ld\n", u.l);
```

Soln: Yes. Both int and long use the same binary/2s-complement number representation which means that values can be truncated to the left as long as all truncated values are 1s or 0s.

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

Part A: Loops and Calling Convention (7 points)

Consider the following code:

```
function:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    -20(%rbp), %eax
        movl    %eax, -4(%rbp)
        jmp     .L2
.L5:
        movl    $0, -8(%rbp)
        jmp     .L3
.L4:
        addl    $1, -8(%rbp)
.L3:
        movl    -8(%rbp), %eax
        cmpl    -4(%rbp), %eax
        jl      .L4
        movl    $88, %edi            # 88 is ASCII for 'X"
        call    putchar
        movl    $10, %edi            # 10 is ASCII for '\n'
        call    putchar
        addl    $1, -4(%rbp)
.L2:
        movl    -4(%rbp), %eax
        cmpl    -24(%rbp), %eax
        jl      .L5
        nop
        leave
        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) (1 points): How many arguments does this function receive (and use)?

Soln: 2

Continued on next page.

3(A)(3) (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: Unused*
4th	*Soln Unused*
5th	*Soln: Unused*

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(10, 9, 8, 7, 6);
```

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: 0. None. Since 10 is greater than 9, the outer-most loop never runs.

Continued on next page.

Part B: Conditionals (8 points)

Consider the following code:

```
(gdb) disassemble function
Dump of assembler code for function function:
   0x0000000000400533 <+0>:     cmp    %esi,%edi
   0x0000000000400535 <+2>:     jg     0x400561 <function+46>
   0x0000000000400537 <+4>:     cmp    $0x5,%edi
   0x000000000040053a <+7>:     ja     0x400557 <function+36>
   0x000000000040053c <+9>:     mov    %edi,%eax
   0x000000000040053e <+11>:    jmpq   *0x400630(,%rax,8)
   0x0000000000400545 <+18>:    mov    $0x1,%edi
   0x000000000040054a <+23>:    mov    %edi,%eax
   0x000000000040054c <+25>:    imul   %edi,%eax
   0x000000000040054f <+28>:    add    %edi,%eax
   0x0000000000400551 <+30>:    retq
   0x0000000000400552 <+31>:    mov    $0xffffffec,%edi
   0x0000000000400557 <+36>:    mov    %edi,%eax
   0x0000000000400559 <+38>:    shr    $0x1f,%eax
   0x000000000040055c <+41>:    add    %edi,%eax
   0x000000000040055e <+43>:    sar    %eax
   0x0000000000400560 <+45>:    retq
   0x0000000000400561 <+46>:    mov    $0xffffffff,%eax
   0x0000000000400566 <+51>:    retq
   0x0000000000400567 <+52>:    mov    $0x8,%eax
   0x000000000040056c <+57>:    retq
End of assembler dump.
```

Consider also the following memory dump:

```
(gdb) x/10gx 0x400620
0x400620:       0x0000000000020001      0x0000000000000000
0x400630:       0x0000000000400545      0x000000000040054a
0x400640:       0x0000000000400557      0x000000000040054a
0x400650:       0x0000000000400567      0x0000000000400552
0x400660:       0x000000443b031b01      0xfffffda000000007
```

Continued on next page.

(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:
1
There are two forward jumps, which are candidates `0x400535` *and* `0x40053A`*. But, the second one is considering the switch control variable, comparing it to a bound, and jumps into code listed in the jump table. So, the one at* `0x400535` *is likely an “if statement” in the C code, whereas the other 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: 0,1,3,4,5
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. `0x400557`*. It is used for both the 2 case and any case larger than 5.*

Note that 2s entry in the jump table is the same as the default case’s entry, as shown by the initial if statement.

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

Soln: Cases 1 and 3. They have the same pointer in the jump table.

Continued on next page.

(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 0 and 5.

If we look at the code block beginning with where the 0th entry in the jump table points, it overlaps the code block pointed to by the next entry (and the entry after that) 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 5th entry in the jump table and the 6th entry, 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:

- Number of sets: 2
- Total size: 48 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 blocks per set?

Soln: 6 blocks (48 bytes)/(8 bytes/block). (6 blocks) / (2 sets) = 3 blocks/set

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

Soln: (8 bit address) - (3 bits for block offset) - (1 bit for set index) = 4 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):**			
0xF1	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0xA2	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0xAB	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0XF7	***Hit***	Miss	Capacity	Cold	Conflict	***N/A***
0X5C	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0XF9	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X00	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0X87	Hit	***Miss***	Capacity	***Cold***	Conflict	N/A
0XA1	Hit	***Miss***	Capacity	Cold	***Conflict***	N/A
0XA2	***Hit***	Miss	Capacity	Cold	Conflict	***N/A***

Part B: Locality (4 points)

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

```
int array[SIZE1][SIZE2];
int sum=0;
for (int outer=0; outer<SIZE1; outer+=STEP)
   for (int inner=0; inner<(SIZE2-1); inner++)
   sum += array[inner][outer] + 2*array[inner][outer+1];
```

Considering only access to “array”, as “step” increases (significantly), please mark how each type of locality would be impacted. Please also explain why in the space provided.

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

Soln: Spatial locality is likely unaffected because the hits from outer vs outer+1 are unaffected and the column-oriented stride is unhelpful for arrays which have a row-major projection. The hits from outer vs outer+1 are unaffected. Temporal locality is likely unaffected, because the element is still getting re-used from one pass through the loop to the next.

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

```
int array[ROWS][COLS];
int array2[ROWS][COLS];
int sum=0;
 for (int row=0; index<ROWS; row ++)
   for (int col=0; col<(COLS-1); col ++)
     sum += array[col][row] + array2[row][col+1];
```

Imagine arrays extremely large in all dimensions, an int size of 4 bytes, and a cache block size of 16 bytes. To the nearest whole percent or simple fraction, what would you expect the combined miss rate for accesses to “array” and “array2” within the inner loop to be? Why?

Continued on next page.

Soln:

The access to array[col][row] is likely a miss each time. Walking across the column is going to hurt. Access to array2 is likely to have a MHHH pattern, except for the beginning of each row which will be _MHH, because the 0th element is not accessed, and the end of each row, which may very with length. But, assuming the row is large, the MHHH will dominate, so together, and neglecting the edge cases, we have MMMM MHHHH

Miss rate: 5/8

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

Imagine a system with a main memory layered beneath a cache:

- The cache has a 10nS access time.
- The overall effective access time is 11nS.
- The cache miss rate is 1%.
- In the event of a miss, memory access time and cache access time do not overlap: They occur 100% sequentially, one after the other.

What is the main memory access time? **FOR SIMPLICITY, AVOID COMPLEX CALCULATION AND LEAVE YOUR ANSWER AS A SIMPLE FRACTION**

MEMORY_ACCESS_TIME=

Soln:
*EFFECTIVE_ACCESS_TIME = CACHE_ACCESS_TIME + MISS_RATE*MISS_PENALTY*
EFFECTIVE_ACCESS_TIME = CACHE_ACCESS_TIME +

MISS_RATE(MEMORY_ACCESS_TIME*
*11nS = 10nS + 0.01*MEMORY_ACCESS_TIME*
*11nS = 10nS + 0.01*MEMORY_ACCESS_TIME*
*1nS = 0.01*MEMORY_ACCESS_TIME*
100nS = MEMORY_ACCESS_TIME
MEMORY_ACCESS_TIME=100ns

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

Consider the following code:

```
#define N 4
void *pointers[N];
int i;

for (i = 0; i < N; i++) {
  pointers[i] = malloc(6);
}

for (i = 0; i < N; i++) {
  free(pointers[i]);
}

for (i = 0; i < N; i++) {
  pointers[i] = malloc(42);
}
```

And a malloc implementation as below:

- Implicit list
- Headers of size 8 bytes
- No footers.
- Every block is always constrained to have a size a multiple of 8 (In order to keep payloads aligned to 8 bytes).
- The header of each block stores the size of the block, and since the 3 lowest order bits are guaranteed to be 0, the lowest order bit is used to store whether the block is allocated or free.
- A first-fit allocation policy is used.
- If no unallocated block of a large enough size to service the request is found, sbrk is called for the smallest multiple of 8 that can service the request.
- The heap is unallocated until it grows in response to the first malloc.
- No coalescing or block splitting is done.

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?

Soln: 288B = 4*(8B +8B) + 4*(8B + 48B)

4(B) (2 points) After the given code sample is run, how many of those bytes are used for currently allocated blocks (vs currently free blocks), including internal fragmentation and header information?

Soln: 224B = 4*(8B + 48B)

4(C) (2 points) After the given code sample is run, how many of those bytes are used to store free blocks (versus currently allocated blocks), including header information?

Soln: 64B=4*(8B+8B)

4(D)(2 points) After the given code sample is run, how much internal fragmentation is there (Answer in bytes)? (*Hint: Free blocks have no internal fragmentation).*

Soln: 56B = 4*(8B + 6B)

4(E)(2 points) After the given code sample is run, how many bytes smaller would the heap be if constant-time (immediate) coalescing were employed?

Soln: 56B. See 4(C) above

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 128 bytes.
- The TLB is 2-way set associative with 16 total 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	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PPN/ PPO	*N*	*N*	*N*	*N*	*N*	*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*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*
TLBI/ TLBT	*T*	*T*	*T*	*T*	*T*	*T*	*I*	*I*	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. 9 bits per page number means 512 pages.

Continued on next page.

Part B: Hits and Misses (12 points)

Shown below are a **partial** TLB and **partial** page table.

TLB:

Index	Tag	PPN	Valid	Scratch space for you
0	0x0A	6	1	***VPN = 00 1010 000 = 0 1010 0000 = 0x180 = 384***
0	0x1B	2	0	***VPN = 01 1011 000 = 0 1101 1000 = 0x0D0 = 208***
1	0x24	11	0	***VPN = 10 0100 001 = 1 0010 0001 = 0x121 = 289***
1	0x31	5	0	
2	0x35	7	0	
2	0x2A	10	0	
3	0x19	12	0	
3	0x02	16	0	
4	0x18	13	0	
4	0x20	15	0	

Page Table:

Index/VPN	PPN	Valid	Scratch space for you
96	3	1	***Miss, No Fault, VPN=0x60, 0 011 0000 0000 1000***
208	12	1	***Miss, No fault , VPN=0xD0***
289	4	0	***Miss, Fault***
384	6	1	***Hit,VPN=180***

For each address shown below, please indicate if it is a TLB Hit or Miss, whether or not it is a page fault, or if either can't be determined from the information provided. Additionally, if knowable from the information provided, please provide the valid PPN

Virtual Address	TLB Hit or Miss?	Page Fault? Yes or No	PPN If Knowable
0x6008	Hit ***Miss*** Not knowable	Yes ***No*** Not knowable	***3***
0xD010	Hit ***Miss*** Not knowable	Yes ***No*** Not knowable	***12***
0x1800	Hit ***Miss*** Not knowable	Yes ***No*** Not knowable	***6***
0x9080	Hit ***Miss*** Not knowable	***Yes*** No Not knowable	***Not knowable***

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

Part A (3 points):

Please consider the following code:

```
void main(){
   printf ("A"); fflush(stdout);
    fork();
    printf ("C"); fflush(stdout);
    if (!fork()) {
      printf ("D"); fflush(stdout);
    } else {
     printf ("B"); fflush(stdout);
    }
}
```

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

Soln: Many are possible, e.g ACCDBDB

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 are possible, e.g. ABBCDCD

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: Within any process C need come before D and B. It is therefore impossible for both Bs to come before both Cs. Similarly, within any process, A needs to come before C, so outputs that violate this constraint are also not possible.

Continued on next page.

Part B (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); printf ("%c", c); fflush(stdout);

  if (!fork()) {
    read (fd1, &c, 1); printf ("%c", c); fflush(stdout);
    sleep(1);
    read (fd1, &c, 1); printf ("%c", c); fflush(stdout);
    read (fd1, &c, 1); printf ("%c", c); fflush(stdout);
  } else {
    fd2=5;
    dup2(fd1, fd2);
    close (fd1);
    c='X';
    read (fd2, &c, 1); printf ("%c", c); fflush(stdout);
  }
}
```

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

Soln: ABCDE

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

Soln: 1

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

Soln: All of the fd entries here are created from the same original one and aliased. They all point to the same system-wide open file table entry.

Continued on next page

Part C (4 points):

Please consider the following code:

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

int i = 4;
void handler(int s) {
  i--; // Decrement max call counter
  write (1, "X", 1);
  if (i) {
    kill(getpid(), SIGUSR1); // Note that this is a system call
    // kill(getpid(), SIGUSR1); *** UNCOMMENT FOR PART C(3) ***
  }
}

int main() {
  signal(SIGUSR1, handler);
  kill(getpid(), SIGUSR1); // Note that this is a system call

  printf("%d\n", i);

  return 0;
}
```

7(C)(1) (1 points): What is the fewest number of times “X” might get printed? Why?

Soln: 4. The critical section, i–, can’t be interrupted, so the count back works fine.

7(C)(2) (1 points): What is the greatest number of times “X” might get printed? Why?

Soln: 4. The critical sectioni– can’t be interrupted, so the count back works fine.

7(C)(3) (1 points): If the commented line is uncommented, what is the maximum number of times “C” might get printed? Why?

Soln: This is a disaster. The critical section, i–, could get interrupted, not update, and generate “infinite signal handling”.

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

Consider a grocery store as follows:

- Four (4) cash registers in the self-checkout area
- An additional three (3) cash registers in the traditional cashier-staffed checkout area.
- Each area (not each register) has its own line.
- Because of the usual grocery store confusion, no one can really tell how long the lines are.
- Each customer may choose to wait in the cashier-supported line, or the self-checkout line, but may not switch lines.

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 a check-out lane's
// configuration as self-service or cashier-service and
// compare a lane's configuration to see which it is
// These could just as easily be an enum or #defined.
const int SELF = 0;
const int CASHIER = 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
  sem_p cashierSem, selfSem;
  sem_init(cashierSem, 3);
  sem_init (selfSem, 4);
}
```

Continued on next page.

```
// Customers call this to wait for a cash register
void waitForCashRegister(int selfOrCashier) {
  // Hint: Which pool(s) of resources are used here?
  // What needs to happen for this to occur safely?
  if (SELF == selfOrCashier)
    P(selfSem);
  else
    P(cashierSem);
}

// Customers call this when done with the cash register
void doneWithCashRegister(int selfOrCashier) {
  // Hint: Which pool(s) of resources are being given up here?
  // What needs to happen to make them available?
  if (SELF == selfOrCashier)
    V(selfSem);
  else
    V(cashierSem);
}
```