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

Practice

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 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: 6-bit w/2s complement signed	**Machine 2: 4-bit w/2s complement signed**
Binary representation of -7 decimal	*Soln: 111001*	*Soln: 1001*
Binary representation of -17 decimal	▨	*Soln: Unable*
Decimal value of +Tmax	*Soln: 011111 = 31*	▨
Binary representation of -1 decimal	▨	*Soln: 1111*

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 5 bits
  - There is 1 sign bit *s*.
  - There are *k* = 2 exponent bits.
  - You need to determine the number of fraction bits.
- Format B:
  - There are 6 bits
  - There is 1 sign bit s.
  - The bias is 1.

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

	Format A	**Format B**
Total Number of Bits (Decimal)	**5**	**6**
Number of Sign Bits (Decimal)	**1**	**1**
Number of Fraction Bits (Decimal)	*Soln: 2*	Soln: 3
Number of Exponent Bits (Decimal)	**2**	*Soln: 2*
Bias (Decimal)	*Soln: 1*	**1**
Smallest magnitude negative number (Decimal value)		*Soln: ⅛* *100001 = ⅛ x 2^0*
+Infinity (Binary bit pattern)	*Soln: 01100*	
110110 **(Decimal value, unrounded)**		Soln: *-7/4 = -1-1/2* E=(2-1)=1 (1+½+¼) x 2^0
00011 **(Decimal value, unrounded)**	*Soln: 3/4* *E = (1 - 1) = 0* *(½+¼) x 2^0*	
011100 **Interpretation of bit pattern**		Circle one: *Soln:NaN* Normalized/Denormalized/Infinity/Nan

`// x and y are floats`
`// x and y are positive`
`(x+y) > (x)`
	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: No. Although longs and ints use the same form of representation, the scanf() reding the int won't reset the high order bits of the long. Since they are initialized, they can be anything.

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:
        movl    $88, %edi           # 88 is ASCII for 'X'
        call    putchar
        addl    $1, -8(%rbp)
.L3:
        movl    -8(%rbp), %eax
        cmpl    -24(%rbp), %eax
        jl      .L4
        movl    $10, %edi           # 10 is ASCII for '\n'
        call    putchar
        subl    $1, -4(%rbp)
.L2:
        cmpl    $0, -4(%rbp)
        jg      .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: 90. The outer loop runs 10 times. The inner loop runs 9 times per iteration of the outer loop. Thus the inner loop runs 90 times in total.

Continued on next page.

Part B: Conditionals (8 points)

Consider the following code:

```
Dump of assembler code for function function:
   0x0000000000400533 <+0>:     cmp    %esi,%edi
   0x0000000000400535 <+2>:     jg     0x400563 <function+48>
   0x0000000000400537 <+4>:     cmp    $0x5,%edi
   0x000000000040053a <+7>:     ja     0x400553 <function+32>
   0x000000000040053c <+9>:     mov    %edi,%eax
   0x000000000040053e <+11>:    jmpq   *0x400620(,%rax,8)
   0x0000000000400545 <+18>:    mov    $0x6,%edi
   0x000000000040054a <+23>:    lea    0x2(%rdi),%eax
   0x000000000040054d <+26>:    retq
   0x000000000040054e <+27>:    mov    $0xffffffec,%edi
   0x0000000000400553 <+32>:    mov    %edi,%eax
   0x0000000000400555 <+34>:    shr    $0x1f,%eax
   0x0000000000400558 <+37>:    add    %edi,%eax
   0x000000000040055a <+39>:    sar    %eax
   0x000000000040055c <+41>:    retq
   0x000000000040055d <+42>:    mov    $0x2,%eax
   0x0000000000400562 <+47>:    retq
   0x0000000000400563 <+48>:    repz retq
```

Consider also the following memory dump:

```
0x400610:      0x0000000000020001      0x0000000000000000
0x400620:      0x0000000000400553      0x000000000040055d
0x400630:      0x0000000000400545      0x000000000040054a
0x400640:      0x000000000040054a      0x000000000040054e
0x400650:      0x000000443b031b01      0xfffffdb000000007
0x400660:      0xfffffdf000000090      0xfffffedd00000060
0x400670:      0xfffffee3000000b8      0xffffff15000000d0
```

(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 addresses 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".

Continued on next page.

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 range of integer input values are managed by non-default cases of the switch statement? How do you know?

Soln: 1-4
Negative values, 0, 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. `0x400553`. It is used for both the 0 case and any case larger than 5.

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

Soln: Cases 3 and 4. 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 and 5.

If we look at the code block beginning with where the 2nd 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:

- Associativity: 2-way set associative
- Total size: 512 bytes (not counting meta data)
- Block size: 64 bytes/block
- Replacement policy: Set-wise LRU
- 16-bit addresses

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

Soln: 64 bytes = 6 bits to index

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

Soln: (512 bytes) / (64 bytes/block) / (2 blocks/set) = 4 sets; 2 bit indexes 4 sets.

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

Soln: (16 bit address) - (6 bits for block offset) - (2 bit for set index) = 8 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):**
0xFF30	Hit ***Miss***	Capacity ***Cold*** Conflict N/A
0XAA00	Hit ***Miss***	Capacity ***Cold*** Conflict N/A
0XFF07	***Hit*** Miss	Capacity Cold Conflict ***N/A***
0X5580	Hit ***Miss***	Capacity ***Cold*** Conflict N/A
0XAA80	Hit ***Miss***	Capacity ***Cold*** Conflict N/A
0X0000	*Hit* **Miss**	Capacity **Cold** Conflict N/A
0XAA30	Hit ***Miss***	Capacity Cold ***Conflict*** N/A
0XAA88	***Hit*** Miss	Capacity Cold Conflict ***N/A***
0XAAE8	Hit ***Miss***	Capacity ***Cold*** Conflict N/A
0X0038	***Hit*** Miss	Capacity Cold Conflict ***N/A***

Part B: Locality (4 points)

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

```
int array[ARRAY_SIZE];
int sum=0;
for (int index=0; index<(ARRAY_SIZE-1); index+= step)
   sum += array[index]+ array[index+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 decreases because the stride skips past parts of the cache. Temporal locality remains, because it is still accessing adjacent elements, so an access is repeated.

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

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

Imagine an extremely large array, 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 miss rate for accesses to "array" to be? Why?

Soln: 50%. 4 ints fit per block. The 1st access misses and brings in 4. It skips past the next one and hits. Then steps to the next block and repeats.

Continued on next page

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

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

- The DRAM has a 100nS access time.
- The SRAM has a 10nS access time.
- 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 maximum acceptable miss rate to achieve a system performance of 20nS?

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

MISS_RATE =

Soln:
*EFFECTIVE_ACCESS_TIME = HIT_TIME + MISS_RATE*MISS_PENALTY*
Miss penalty = 100nS
*20nS = 10nS + MISS_RATE * 100nS*
*10nS = MISS_RATE*100nS*
MISS_RATE = 10nS/100nS
MISS_RATE=1/10th

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
- Footer size of 8 bytes
- Every block is constrained to have a size that is a multiple of 8 (In order to keep payloads aligned to 8 bytes).
- The structure of the header and footer are the same. Each contain the size of the block and, encoded within it, a single bit to indicate 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.
- A complete (left+right) constant-time coalesce is employed.
- Block splitting is permitted so “leftover” space after an allocation can be returned to the free list so long as it is large enough to satisfy the constraints above

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

Continued on next page.

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: 256B = 4*(8B + 48B + 8B)

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: 256B. Everything is currently allocated.

4(C)(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: 88B = 4*(8B + 8B + 6B)

4(D)(2 points) How much more or less internal fragmentation would there be if constant-time coalesce were not used (assume a full left-right coalesce is still done, just not a constant-time coalesce) and the headers and footers were optimized accordingly?

Soln: 32B. Only headers would be needed, saving 8B per allocated block.

4(E)(2 points) Which of the following best describes the complexity of full (left+right) coalescing of this data structure without the footers? And, why? Circle one, and write your explanation below.

Constant time/O(1)   Linear time/O(n)   Quadratic time/O(N^2) Exponential time/O(2^n)

Soln: Linear time/O(n). Without the footers, there is no way to "look left", so, to get access to neighbors in both directions, one would have to traverse the list.

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 128 bytes.
- The TLB is 2-way set associative with 4 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	**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*	*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. 5 bits per page number means 32 pages.

Continued on next page.

Part B: Hits and Misses (12 points)

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

TLB:

Index	**Tag**	**PPN**	**Valid**	**Scratch space for you**
0	0x4	12	1	***vpn=0 0000 = 0x10 = 16***
0	0x1	6	0	***vpn=0 0100 = 0x04 = 4***
1	0x4	2	1	***vpn=1 0001 = 0x11 = 17***
1	0x6	1	0	***vpn=1 1001 = 0x15 = 21***

Page Table:

Index/VPN	**PPN**	**Valid**	**Scratch space for you**
4	5	1	***TLB Miss, No fault***
16	12	1	***TLB Hit***
17	2	1	***TLB Hit***
21	8	0	***TLB Miss, Page Fault***

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**
0x22A	Hit **Miss** Not knowable	Yes ***No*** Not knowable	***5***
0x82A	***Hit*** Miss Not knowable	Yes ***No*** Not knowable	***12***
0x8B7	***Hit*** Miss Not knowable	Yes ***No*** Not knowable	***2***
0xA87	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);
    if (!fork()) {
     printf ("C"); fflush(stdout);
     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 ACDDB
A->C->D
A->B

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. ADCDB

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: Many are possible. They should violate one or more of the below:
A->C->D
A->B

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);
    fd2=5;
    dup2(fd1, fd2);
    read (fd1, &c, 1); printf ("%c", c); fflush(stdout);
    read (fd2, &c, 1); printf ("%c", c); fflush(stdout);
  } else {
    read (fd1, &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 file descriptors point to the same system-wide open file table entry so they fully cooperate.

Continued on next page

Part C (4 points):

Please consider the following code:

```
#include <stdio.h>

void handler (int signo) {

  // Some code here

  printf ("The signal received is: %d\n", signo);

  // Some code here
}

// Imagine any essential but missing code to be here

void main() {

  printf ("Before handlers installed.\n");
  signal(SIGNO1, handler);
  signal(SIGNO2, handler);
  signal(SIGNO3, handler);
  printf ("After handlers installed.\n");

  // Imagine a bunch of truly important code here

  printf ("All done.\n");
}
```

7(C)(1) (2 points): The code above is not correct. Specifically, it has a problem related to one or more shared resources. Please describe what is shared.

Soln: The libc buffers used by printf() are shared.

7(C)(2) (1 points): The code above is not correct. Please explain how a deadlock might result, even if the code would be correct in light of thread-based concurrency without signal handlers.

Soln: Should these buffers use locking, it is possible that a printf() outside of the handler will acquire the lock, and then signal handler will run and attempt to acquire the same lock. Once this happens, deadlock results as the main program can't run until the signal handler finishes.

Continued on next page.

7(C)(3) (1 points): The code above is not correct. Despite the error(s), please explain how correct output might result.

Soln: The execution may interleave the code in such a way that the unsafe concurrency does not occur.

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

Consider the goal of writing a concurrent program to achieve the following:

- The main thread creates two peer threads
- Each peer thread is passed a unique integer thread ID (either 0 or 1)
- The main thread then waits for each thread to terminate.
- Each peer thread prints its thread ID and then terminates
- The program is hosted on a Linux Shark Machine
- Each function call within the program returns successfully.

Each of the following 5 programs represent an attempt at a correct solution, but some suffer from concurrency-related problem(s).

- Please write CORRECT for each correct solution.
- Please write INCORRECT and describe the CONCURRENCY problem(s) for each incorrect attempt at a solution.

In case it is helpful, please recall that pthread create takes in four arguments:

- 1st argument is where the ID of the new thread is stored upon successful creation.
- 2nd argument represents the attributes, which may be ignored for this problem.
- 3rd contains the routine to be called, (or the function to be executed)
- 4th argument contains the argument passed to the thread routine

Continued on next page.

8(a)(1) (3 points)

```
void  *foo (void *vargp ) {
  int myid;

  myid =  *((int *) vargp);
  free (vargp) ;
  printf ("Thread %d\n", myid ) ;
}


int main () {
  pthread_t tid [2];

  int i, *ptr;

  for (i=0; i<2; i++) {
    ptr = malloc (sizeof(int));
    *ptr = i ;
    pthread_create(&tid[i], 0 , foo, ptr);
  }

  pthread_join (tid[0], 0);
  pthread_join (tid[1] ,0);
}
```

Your response:

Soln: Correct. Each thread is getting its own copy of the variable via malloc(), so nothing is shared. This variable is being correctly freed after it is copied into a per-thread local variable managed by the stack.

Continued on next page.

8(a)(2) (3 points)

```
void *foo (void  *vargp) {
  int myid;
  myid =  *((int *) vargp) ;
  printf ("Thread %d\n" , myid);
}

int main() {
  pthread_t tid[2] ;
  int i;

  for (i=0; i<2; i++) {
    Pthread_create (&tid[i], NULL, foo, &i);

  pthread_join(tid[0], NULL);
  pthread_join(tid[0], NULL);
```

Your response:

Soln: There exists only one copy of "i", which is being passed by reference and shared.

8(a)(3) (3 points)

```
void  foo(void *vargp) {
  int myid;
  myid = (int) vargp;
  printf ("Thread %d\n", myid );
}

int main () {
  pthread_t tid[2];
  int i;

  for (i=0; i<2; i++)
    pthread_create (&tid[i], 0, foo, i);

  pthread_join (tid[0], 0);
  pthread_join (tid[1 ], 0);
}
```

Soln: In this case, "i" is getting passed by value. So, each thread is getting its own copy of "i".

Continued on next page.

8(a)(4) (3 points)

```
sem_t s ; /* semaphore s */

void  foo(void  *vargp) {
  int myid;

  P(&s);
  myid =  *((int *) vargp);
  V(&s);

  printf ("Thread %d\n", myid);
}

int main () {
  pthread_t tid [2];
  int i;

  sem_init (&s, 0, 1); /* S=1, initially */

  for (i=0; i<2; i++) {
    pthread_create(&tid[i], 0, foo, &i);
  pthread_join(tid[0], 0);
  pthread_join (tid[1], 0);
 }
```

Soln: Wrapping semaphores around the copt in foo() doesn't help because it can still be corrupted by the increment in the for-loop in main().

Continued on next page.

8(a)(5) (3 points)

```
sem_t s; /* semaphore s */

void  *foo (void  *vargp) {
  int myid;

  myid =  *((int *) vargp);

  V(&s);
  printf("Thread %d\n", myid);
}

int main() {
  pthread_t tid [2];
  int i;

  sem_init (&s , 0 , 0 ); /* S=0, initially */

  for (i=0; i<2; i++) {
    Pthread_create(&tid[i],0 ,foo , &i ) ;
    P(&s);
  }

  pthread_join(tid[0], 0);
  pthread_join(tid[1], 0);
}
```

Soln: This works. The P(&s) after the create forces the id to be copied into a local variable before the next one can interfere with it.