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 2022 Final Exam

Friday, December 16, 2022

Instructions:

- Make sure that your exam is not missing any sheets (check page numbers at bottom: there should be 21 pages)
- 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 a machine using two's complement arithmetic for signed integers:

- 5-bit integers
- 2s complement signed representation

Fill in the five empty boxes in the table below when possible and indicate "UNABLE" when impossible. An "Everyday" number or expression has the value it would be understood to have in middle school arithmetic. A "C expression" has the value it would have if evaluated in a C Language program running on the machine.

| Goal | Machine 1: 5-bit w/2s complement signed | True or False |
|---|---|---|
| "Everyday number"<br>-6 | | X |
| "Everyday number"<br>21 | | X |
| "Everyday expression"<br>(-1*(-14 - 2)) | | X |
| C Expression:<br>((-1*(-14 - 2)) < -15) | X | |
| Tmin (Most negative number) | | X |

**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 7 bits
    - There is 1 sign bit s.
    - There are n = 3 exponent bits.

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

| | **Answer** |
|---|---|
| **Total Number of Bits (Decimal)** | **7** |
| **Number of Sign Bits (Decimal)** | **1** |
| **Number of Exponent Bits (Decimal)** | **3** |
| **Number of Fraction Bits (Decimal)** | |
| **Bias (Decimal)** | |
| **Bit pattern for -Inf (“Negative Infinity”)** | |
| **The absolute difference, in decimal or as a power of 2, between any two adjacent denormalized numbers** | |
| **1 100 010 (Decimal value, unrounded)** | |

**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.

**Part A (2 points):** Consider the following struct. How much memory is required? Answer in bytes.

```
struct {
  char c;
  long l;
  short s;
} examStruct1;
```

**Part B (2 points):** Rewrite the struct to require as little memory as possible:

**Part C (2 point):** How much memory, in bytes, is saved by the reorganization of the struct?

**Part D (2 points):** Consider the following array. How far apart are array[2][3] and array[3][2]? Answer in bytes.

```
short array[4][4];
```

**Part E (2 points):** Consider the following, what is the address of `es1p->l`? Please answer in hexidecimal. This question refers to the structure in Part A.

```
examStruct1 array_es1[5]; // Assume this starts at address 0x1000
examStruct1 *es1p = &(array_es1[3]);
```

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

Assume that all subparts pertain to the “Shark Machine” environment.

**Part A: Calling Convention (3 points)**

Consider the following code:

```
fn:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    %edx, -12(%rbp)
        movl    -8(%rbp), %eax
        imull   -12(%rbp), %eax
        addl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

**3(A)(1) (1 points):** How many arguments are used by the function? How do you know?

**3(A)(2) (1 points):** Does the function return a value? How do you know?

**3(A)(3) (1 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 | |
| 2nd | |
| 3rd | |
| 4th | |
| 5th | |

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

**Part B: Conditionals and Loops (6 points)**

*Hint:* It might help to look at your argument table in 3(A)(3) and to recall that $.LC0 is read-only data, such as a printf() format string.

Consider the following code:

```
main:
.LFB3:
        pushq   %r12
        pushq   %rbp
        xorl    %ebp, %ebp
        pushq   %rbx
        subq    $64, %rsp
.L2:
        movslq  %ebp, %r12
        movl    %ebp, %ebx
        salq    $2, %r12
.L5:
        movslq  %ebx, %rax
        movl    $.LC0, %edi
        addl    $1, %ebx
        addq    %r12, %rax
        movl    (%rsp,%rax,4), %esi
        xorl    %eax, %eax
        call    printf
        cmpl    $4, %ebx
        jne     .L5
        addl    $1, %ebp
        cmpl    $4, %ebp
        jne     .L2
        addq    $64, %rsp
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret
```

**3(B)(1) (2 points):** How many loops are there? How do you know? Count each nested loop separately.

**3(B)(2) (2 points):** How many “if statements” are there (do not count any ifs used to control loops)? How do you know?

**3(B)(3) (2 points):** How many times is printf() called? Explain your answer.

**Part C: Switch statement (6 points)**

Consider the following compiled from C Language code containing a switch statement and no if statements.

```
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x0000000000400550 <+0>:     cmp    $0x5,%esi
   0x0000000000400553 <+3>:     ja     0x400590 <foo+64>
   0x0000000000400555 <+5>:     mov    %esi,%esi
   0x0000000000400557 <+7>:     jmpq   *0x400640(,%rsi,8)
   0x000000000040055e <+14>:    mov    %eax,%eax
   0x0000000000400560 <+16>:    shl    $0x2,%edi
   0x0000000000400563 <+19>:    add    $0x2,%edi
   0x0000000000400566 <+22>:    lea    0x7(%rdi),%edx
   0x0000000000400569 <+25>:    mov    %edx,%eax
   0x000000000040056b <+27>:    retq
   0x000000000040056c <+28>:    nopl   0x0(%rax)
   0x0000000000400570 <+32>:    lea    0x3(%rdi),%edx
   0x0000000000400573 <+35>:    test   %edi,%edi
   0x0000000000400575 <+37>:    cmovns %edi,%edx
   0x0000000000400578 <+40>:    sar    $0x2,%edx
   0x000000000040057b <+43>:    mov    %edx,%eax
   0x000000000040057d <+45>:    retq
   0x000000000040057e <+46>:    mov    %eax,%eax
   0x0000000000400580 <+48>:    lea    0x1(%rdi),%edx
   0x0000000000400583 <+51>:    mov    %edx,%eax
   0x0000000000400585 <+53>:    retq
   0x0000000000400586 <+54>:    nopw   %cs:0x0(%rax,%rax,1)
   0x0000000000400590 <+64>:    mov    %edi,%eax
   0x0000000000400592 <+66>:    mov    $0x55555556,%edx
   0x0000000000400597 <+71>:    sar    $0x1f,%edi
   0x000000000040059a <+74>:    imul   %edx
   0x000000000040059c <+76>:    sub    %edi,%edx
   0x000000000040059e <+78>:    mov    %edx,%eax
   0x00000000004005a0 <+80>:    retq
End of assembler dump.
```

Consider also the following memory dump, with the address obscured. Assume that it begins with the 0th entry of the switch statement's jump table.

```
(gdb) x/16gx 0xXXXXXX

        :       0x0000000000400580      0x0000000000400590
        :       0x0000000000400560      0x0000000000400563
        :       0x0000000000400566      0x0000000000400570
        :       0x0000003c3b031b01      0xffffffd9000000006
        :       0xffffffdd000000088     0xffffffdf5000000c8
        :       0xffffffee000000058     0xffffff40000000b0
        :       0xffffffb0000000e8      Cannot access memory at address 0xYYYYYY
```

*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?

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

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

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

**Part A: Caching (9 points)**

Given a model described as follows:

- Associativity: 2-way set associative
- Total size: 32 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?

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

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

**4(A)(4) (6 points, ½ point per row)**: 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):** | | | |
|---|---|---|---|---|---|---|
| 0x22 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xB9 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xA0 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xA3 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0x9B | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0x42 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0x23 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xA3 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0x9B | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0x42 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xBA | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |
| 0xA8 | *Hit* | *Miss* | *Capacity* | *Cold* | *Conflict* | *N/A* |

**Part B: Locality (4 points)**

Analyze the cache performance of the following piece of code. Assume that the matrix arr1 is aligned so that arr1[0][0] is the first element of a cache block, and that arr2 is laid out in memory immediately after arr1. Assume memory accesses for right operands are done before memory accesses for left operands. Assume a write-back/write-allocate cache.

```
int arr1[4][4];
short arr2[4][4];

for (int rindex=0; rindex<4; rindex++) {
  for (int cindex=0; cindex<4; cindex++) {
    arr1[rindex][cindex] = arr2[rindex][3-cindex];

  }
}
```

**B(1)(4 points):** Please determine the miss rate for each array under each configuration. Leave your answer as a reduced fraction

| Question | Data Item | Cache Configuration | Miss rate |
|---|---|---|---|
| **B(1)(1 point)** | arr1 | Direct-mapped, 4 sets, 16-byte blocks | |
| **B(2)(1 point)** | arr1 | 2-way set associative, 4 sets, 8 byte blocks | |
| **B(3) (1 point)** | arr2 | Direct-mapped, 4 sets, 16-byte blocks | |
| **B(4) (1 point)** | arr2 | 2-way set associative, 4 sets, 8 byte blocks | |

**Part C: Memory Hierarchy and Effective Access Time (2 points)**

Imagine a computer system as follows:

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

What is the access time for the L1 cache?

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

Considering a malloc implementation as described below:

- Explicit list, ordered by address smallest-to-largest, allocation via first-fit (*not* next-fit)
- Headers of size 8 bytes, Footer size of 8-bytes (both free and allocated blocks have footers)
- Blocks must be a multiple of 8-bytes (In order to keep payloads aligned to 8 bytes).
- 8 byte pointers. Minimum block size: 32 bytes
- If there is no unallocated block of a large enough size to service the request, sbrk is called to grow the heap enough to get a new block of the smallest size that can service the request (after coalescing).
- The heap is unallocated until it grows in response to the first malloc call. There are no dummy headers or footers.
- Constant-time coalescing, as discussed in lecture, is used.
- The heap never shrinks.

**5(A) (10 points, 1 point per line)** Please complete the following table with the values *after* the requested operation completes. The following definitions may be helpful reminders:

- *Total Heap Size* is the number of bytes between the base of the heap and the brk point, i.e. top of the heap.
- *Aggregate Request Size* is the total number of bytes requested via malloc() and not yet free()d.
- *Total Internal Fragmentation* is the difference, in bytes, between the total size of all allocated blocks and the sum of all of the corresponding requests. *Hint:* Unallocated blocks do not contribute to internal fragmentation.

| | **Operation** | **Total Heap Size** | **Aggregate Request Size** | **Total Internal Fragmentation** |
|---|---|---|---|---|
| **5(A)(1)(1 point)** | `ptr1 = malloc (6);` | | | |
| **5(A)(2)(1 point)** | `ptr2 = malloc (2);` | | | |
| **5(A)(3)(1 point)** | `free(ptr1);` | | | |
| **5(A)(4)(1 point)** | `free (ptr2);` | | | |
| **5(A)(5)(1 point)** | `ptr1 = malloc(32);` | | | |
| **5(A)(6)(1 point)** | `ptr2 = malloc(1);` | | | |
| **5(A)(7)(1 point)** | `ptr3 = malloc(24);` | | | |
| **5(A)(8)(1 point)** | `free (ptr1);` | | | |
| **5(A)(9)(1 point)** | `free (ptr3);` | | | |
| **5(A)(10)(1 point)** | `malloc(2);` | | | |

**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 10 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.
- 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 point):** 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** | | | | | | | | | | | | |

**6(A)(2)(1 point):** 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** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|
| **VPO/ VPN** | | | | | | *VPO* | *VPO* | *VPO* | *VPO* | *VPO* |
| **TLBI/ TLBT** | | | | | | | | | | |

**6(A)(3)(1 point):** How many entries exist within each page table? *Hint:* This is the same as the total number of pages within each virtual address space.

**Part B: Hits and Misses (12 points)**

Shown below are the **initial** states of the TLB and **partial** page table.

**TLB** (I=INVALID, V=VALID, R=READ, W=WRITE, N=Not Resident, e.g. swapped):

| Set | Tag | PPN | BITS | Scratch space for you |
|---|---|---|---|---|
| 0 | 00 | 5 | V-RW | |
| 0 | 10 | 12 | I-R | |
| 1 | 00 | 7 | V-RN | |
| 1 | 11 | 2 | V-RW | |

**Page Table** (I=INVALID, V=VALID, R=READ, W=WRITE, N=Not Resident, e.g. swapped):

| Index/VPN | PPN | BITS | Scratch space for you |
|---|---|---|---|
| 0 | 5 | V-RW | |
| 1 | 7 | V-RN | |
| 2 | 9 | I-RW | |
| 3 | 11 | V-R | |
| 4 | 13 | V-R | |
| 5 | 15 | I-R | |
| 6 | 27 | V-RW | |
| 7 | 2 | V-RW | |

*Continued on next page.*

**Part B: Hits and Misses, *cont.* (12 points, 2 points per line)**

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

| **Oper-ation** | **Virtual Address** | **TLB Hit or Miss?** | **Page Table Hit or Miss?** | **Page Fault? Yes or No?** | **PPN If Knowable** |
|---|---|---|---|---|---|
| Read | 0x326 | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |
| Write | 0x3EF | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |
| Read | 0x3ED | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |
| Read | 0x17A | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |
| Read | 0x02F | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |
| Write | 0x326 | Hit Miss Not knowable | Hit Miss N/A | Yes No Not knowable | |

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

**Part A (3 points):**

Please consider the following code:

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

  if (fork()) {
    printf ("B"); fflush(stdout);
  } else {
    printf ("C"); fflush(stdout);
    if (fork()) {
      wait (NULL);
    }
    printf ("E"); fflush (stdout);
  }
```

**7(A)(1) (1 point):** Give one possible output string.

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

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

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

**Part B (3 points):**

Please consider the following code:

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

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

 // Assume "file.txt" is initially non-existant or empty.
 int fd0 = open("file.txt", O_RDWR | O_CREAT, 0666);
 int fd1 = -1;

 write(fd0, buffer, 3);

 if (!fork()) {
   write(fd0, buffer+3, 3);

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

   write(fd1, "XYZ", 3);

   dup2 (fd1,fd0);  // int dup2(int oldfd, int newfd);

   write(fd0, "UVW", 3);
  }

 return 0;
}
```

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

**7(B)(2) (1 point):** If each process (parent and child) were just about to "return 0", how many entries are there in the system-wide open file table related to this code?

**Question 7: Process Representation and Lifecycle + Signals and Files, *cont.* (10 points)**
**Part C (4 points):**

Consider the C code below. Assume that no errors prevent any processes from running to completion.

```
int count = 0;
sigset_t newset, oldset1, oldset2;

void inthandler(int sig){
    sigprocmask(SIG_BLOCK, &newset, &oldset1);

    Sio_printf("SIGINT received\n");

    sigprocmask(SIG_BLOCK, &oldset1, NULL);

   return;
}

void childhandler(int sig){
  int status;

  sigprocmask(SIG_BLOCK, &newset, &oldset2);

  wait(&status);

  //Assume a process terminated by an uncaught signal has an exit status of 0.
  count += WEXITSTATUS(status);

  sigprocmask(SIG_BLOCK, &oldset2, NULL);

  return;
}

void main(){
  int i;          // for loop iterator
  pid_t pid[3];   // pids of child processes

  sigemptyset(&oldset1);
  sigemptyset(&newset);
  sigaddset(&newset, SIGCHLD);
  sigaddset(&newset, SIGINT);

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

  for(i=0; i<3; i++){ // Fork 3 child processes
    pid[i] = fork();

    if(!pid[i]){ // If child process
      do_work(); // Don't be concerned with the detail.
      exit(5);
    }
  }

  // Parent process only
  for(i=0; i<3; i++){
    kill(pid[i], SIGINT);
  }
  sleep(5);
  printf("count = %d\n", count);
  exit(0);
}
```

*Continued on next page.*

**Question 7: Process Representation and Lifecycle + Signals and Files, *cont.* (10 points)**
**Part C, *cont.* (4 points):**

**7(C)(1)(2 points)** What is the ***minimum*** number of times “SIGINT received” could be printed? Why?

**7(C)(2)(2 points)** List all possible values of count that could be printed:

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

Consider the following C code:

| 8(A)(2) | 8(A)(3) | Code |
|---|---|---|
| | | 1. /* Initialize semaphores */ |
| | | 2. mutex1 = 1; |
| | | 3. mutex2 = 1; |
| | | 4. mutex3 = 1; |
| | | 5. mutex4 = 1; |
| | | 6 |
| | | 7. void thread1() { |
| | | 8.  P(mutex1); |
| | | 9.  P(mutex2); |
| | | 10.  P(mutex4); |
| | | 11 |
| | | 12.  /* Access Data */ |
| | | 13.  V(mutex4); |
| | | 14.  V(mutex2); |
| | | 15.  V(mutex1); |
| | | 16. } |
| | | 17 |
| | | 18. void thread2() { |
| | | 19.  P(mutex4); |
| | | 20.  P(mutex3); |
| | | 21.  P(mutex2); |
| | | 22 |
| | | 23.  /* Access Data */ |
| | | 24 |
| | | 25.  V(mutex2); |
| | | 26.  V(mutex3); |
| | | 27.  V(mutex4); |
| | | 28. } |

**8(A)(1) (2 points)** Is it possible for the code above to deadlock? Yes No

**8(A)(2) (3 points)** Consider your answer to (A) above. If you answered “No”, explain why not. If you answered “Yes”, then please provide a schedule that results in deadlock. Do this by numbering, i.e. 1, 2, 3, etc, the semaphore operations (Ps and Vs, only) in the code above with an execution order that results in deadlock. Use the 8(A)(2) column to record your answer.

**8(A)(3) (3 points)** Is it possible for the code above to execute without deadlocking? If not, explain why not. If it is possible, please provide a schedule that does *not* results in deadlock. Do this by numbering, i.e. 1, 2, 3, etc, the semaphore operations (Ps and Vs, only) in the code above with an order in which they execute without deadlock. Use the 8(A)(3) column to record your answer.

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

**Part B (7 points): Concurrency Control**

Imagine a situation such that

- Widgets are made in a mold, four (4) at a time.
- After manufacture, widgets are placed on a shelf that can hold up to twelve (12) at a time.
- Widgets are shipped in boxes 6 at a time.
- The shelving and boxing of widgets occur in multiple independent threads which call the functions below.

Please complete the following code for the shelving and boxing of widgets **by adding semaphores and semaphore operations as needed**.

```
#define SHELF_SIZE 12
#define MOLD_SIZE 4
#define BOX_SIZE 6

widget shelf[SHELF_SIZE];
sem_t mutex;

// Hint: Declare any additional needed shared, global variables here.
// Hint: Semaphores may be declared as sem_t, e.g. "sem_t someSemaphore;"
```

```
// This function is called everytime a new batch of widgets is produced.
// The function finds a place on the shelf for each widget in the batch.

void ShelveNewWidgetBatch () {
  int shelfSpaceNeeded = MOLD_SIZE; // Widgets are always produced in groups of MOLD_SIZE.

  while (shelfSpaceNeeded > 0) {
    // Hint: Add code here




    for (unsigned int index=0; index < SHELF_SIZE; index++) {
      if (shelf[index] == NULL) {
        shelf[index] = getWidgetFromBatch(); // Don't worry about internals of this function
        shelfSpaceNeeded--;
        // Hint: Add code here




        break;
      }
    }
    // Hint: Add code here



  }
}
```

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

**Part B (7 points), *cont.*: Concurrency Control**

```
// As widgets are placed on the shelf, this function takes widgets from the shelf
// and packs them into a box for shipping.

void PackBoxOfWidgets(box_t *box) {
  int widgetsNeededForBox = BOX_SIZE;


  while (widgetsNeededForBox > 0) {

    // Hint: Add code here









    for (unsigned int index=0; index < SHELF_SIZE; index++) {
      if (shelf[index] != NULL) {
        addWidgetToBox (box, shelf[index]);
        shelf[index] = NULL;
        widgetsNeededForBox--;

        // Hint: Add code here









        break;
      }
    }
    // Hint: Add code here









  }
}
```

**The End (of the whole exam!)! You made it!**