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 2023 Final Exam

Monday, December 11, 2023

Instructions:

- Make sure that your exam is not missing any sheets (check page numbers at bottom)
- Write your Andrew ID and full name on this page (and we suggest on each and every page)
- This exam is closed book and closed notes.
- You may not use any electronic devices or anything other than what we provide and writing implements, such as pens and pencils.
- Write your answers in the space provided for the problem.
- If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The point value of each problem is indicated.
- **Good luck!**

Problem #	**Scope**	**Max Points**	**Score**
1	Data Representation: "Simple" Scalars: Ints and Floats	10	
2	Data Representation: Arrays, Structs, Unions, and Alignment	10	
3	Assembly, Stack Discipline, Calling Convention, and x86-64 ISA	15	
4	Caching, Locality, Memory Hierarchy, Effective Access Time	15	
5	Malloc(), Free(), and User-Level Memory Allocation	10	
6	Virtual Memory, Paging, and the TLB	15	
7	Process Representation and Lifecycle + Signals and Files	10	
8	Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW	15	
TOTAL	***Total points across all problems***	***100***	

Question 1: Representation: "Simple" Scalars (10 points)
Part A: Integers (5 points, 1 point per blank)

Assume we are running code on a machine representing "int" numbers as follows:

- 6-bits
- 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.

Goal	**Machine 1: 6-bit w/2s complement signed**	**True or False**
"Everyday number" 5	(Answer should show bits)	
"Everyday number" -13	(Answer should show bits)	
"C Expression" (-31 - 3)	(Answer should be a decimal number)	
C Expression: (-31 > 3U)		(Answer should be True or False)
Tmin (Most negative number) + Tmax (Most positive number)	(Answer should be a decimal number)	

Question 1: Representation: "Simple" Scalars (10 points)

Part B: Floats (5 points, 1 point per blank)

For this problem, please consider a floating point number representation based upon an IEEE- like floating point format as described below.

- Format:
  - There are 6 bits
  - All values are greater than or equal to 0 (A departure from IEEE)
  - There are n = 3 exponent bits.

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

	Answer
Total Number of Bits (Decimal)	**6**
Number of Exponent Bits (Decimal)	**3**
Number of Fraction Bits (Decimal)	
Bias (Decimal)	
The absolute difference, represented as a reduced fraction or as a power of two, between any two adjacent denormalized numbers	
100 101 (Decimal value, unrounded)	
Bit representation of the value shown below, or the closest possible representable value to it. *Hint:* Round even. **15.5, a.k.a. 15 1/2, a.k.a. 31/2**	

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

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

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

```
struct {
  double d;
   char c;
   float f;
} examStruct1;
```

Part B (2 points): How many bytes of padding are contained within examStruct1 ?

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

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

Part D (2 points): What is the offset in bytes of `array[0][1].c` within the referenced struct.

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

Continued on next page.

Question 3: Assembly, Stack Discipline, Calling Convention, and x86-64 (15 points)
Part A: Calling Convention (4 points)

3(A)(1) (1 points): Consider an instance of `struct s` being passed into function `fun()`, as shown below. Please label each field of the struct with REGISTER if its value is passed into the function via a register and STACK if its value is passed into the function via the stack.

```
struct s {
  int i;         // LABEL HERE:
  char c;        // LABEL HERE:
  short s;       // LABEL HERE:
  char str[10];  // LABEL HERE:
}

void fun (struct s argument);
```

3(A)(2) (1 points): Consider an instance of an array being passed into function `fun()`, as shown below. Assuming the array, itself, is allocated as a local variable by the caller. Are the array elements found by the callee in one or more REGISTER(S) or on the STACK? (Write REGISTER(S) or write STACK.)

```
typedef int numbers[4];
void fun (numbers nums); // Same as: void fun (int nums[4])
```

3(A)(3) (2 points): Consider the function below as compiled and linked on one of our shark machines. How many bytes of the stack are used for parameter passing upon the calling of the function?

```
void fun (int a, int b, char c, short d, long e, long f, int g, long h);
```

Continued on next page.

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

Consider the following code:

```
Dump of assembler code for function loop:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000001154 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000000000001157 <+14>:    jmp    0x117c <loop+51>
   0x0000000000001159 <+16>:    mov    -0x8(%rbp),%eax
   0x000000000000115c <+19>:    cmp    -0x4(%rbp),%eax
   0x000000000000115f <+22>:    jge    0x1172 <loop+41>
   0x0000000000001161 <+24>:    mov    -0x8(%rbp),%eax
   0x0000000000001164 <+27>:    mov    %eax,%edx
   0x0000000000001166 <+29>:    shr    $0x1f,%edx
   0x0000000000001169 <+32>:    add    %edx,%eax
   0x000000000000116b <+34>:    sar    %eax
   0x000000000000116d <+36>:    mov    %eax,-0x4(%rbp)
   0x0000000000001170 <+39>:    jmp    0x117c <loop+51>
   0x0000000000001172 <+41>:    cmpl   $0x64,-0x4(%rbp)
   0x0000000000001176 <+45>:    jg     0x1186 <loop+61>
   0x0000000000001178 <+47>:    addl   $0x1,-0x4(%rbp)
   0x000000000000117c <+51>:    mov    -0x4(%rbp),%eax
   0x000000000000117f <+54>:    cmp    -0x8(%rbp),%eax
   0x0000000000001182 <+57>:    jle    0x1159 <loop+16>
   0x0000000000001184 <+59>:    jmp    0x1187 <loop+62>
   0x0000000000001186 <+61>:    nop
   0x0000000000001187 <+62>:    nop
   0x0000000000001188 <+63>:    pop    %rbp
   0x0000000000001189 <+64>:    ret
End of assembler dump.
```

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

3(B)(1) (2 points): Is the loop shown above most representative of a `while () {….}` or a `do { … } while()`? How do you know?

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

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

3(B)(4) (1 points): How many ?-operators (ternary operators) are there? 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. It uses a very common form of the switch statement on the shark machines, but a slightly different one than some prior exams. Rather than keeping absolute addresses**, this jump table keeps offsets from its own start address. The address of each code block is the address of the beginning of the jump table plus the value of the code block's jump table entry.** You'll see this add before the relevant jump in the assembly. It might make things easier for you to note the address indicated by the lowest jump table entry and think of the other entries relative to that one.

```
Dump of assembler code for function foo:
   0x00000000004017c0 <+0>:     endbr64
   0x00000000004017c4 <+4>:     push   %rbp
   0x00000000004017c5 <+5>:     mov    %edi,%ebp
   0x00000000004017c7 <+7>:     lea    0x96836(%rip),%rdi
   0x00000000004017ce <+14>:    push   %rbx
   0x00000000004017cf <+15>:    mov    %esi,%ebx
   0x00000000004017d1 <+17>:    sub    $0x8,%rsp
   0x00000000004017d5 <+21>:    call   0x40c7c0 <puts>
   0x00000000004017da <+26>:    mov    0xc3f0f(%rip),%rdi
   0x00000000004017e1 <+33>:    call   0x40c470 <fflush>
   0x00000000004017e6 <+38>:    lea    0x2(%rbx),%esi
   0x00000000004017e9 <+41>:    cmp    $0xc,%esi
   0x00000000004017ec <+44>:    ja     0x401818 <foo+88>
   0x00000000004017ee <+46>:    lea    0x96823(%rip),%rdx        # %rdx = 0x498018
   0x00000000004017f5 <+53>:    movslq (%rdx,%rsi,4),%rax
   0x00000000004017f9 <+57>:    add    %rdx,%rax
   0x00000000004017fc <+60>:    jmp    *%rax
   0x00000000004017ff <+63>:    nop
   0x0000000000401800 <+64>:    lea    0x0(,%rbp,8),%eax
   0x0000000000401807 <+71>:    sub    %ebp,%eax
   0x0000000000401809 <+73>:    mov    %eax,%ebp
   0x000000000040180b <+75>:    add    $0x8,%rsp
   0x000000000040180f <+79>:    lea    0x2(%rbp),%eax
   0x0000000000401812 <+82>:    pop    %rbx
   0x0000000000401813 <+83>:    pop    %rbp
   0x0000000000401814 <+84>:    ret
   0x0000000000401815 <+85>:    nopl   (%rax)
   0x0000000000401818 <+88>:    movslq %ebp,%rax
   0x000000000040181b <+91>:    add    $0x8,%rsp
   0x000000000040181f <+95>:    sar    $0x1f,%ebp
   0x0000000000401822 <+98>:    imul   $0x55555556,%rax,%rax
   0x0000000000401829 <+105>:   pop    %rbx
   0x000000000040182a <+106>:   shr    $0x20,%rax
   0x000000000040182e <+110>:   sub    %ebp,%eax
   0x0000000000401830 <+112>:   pop    %rbp
   0x0000000000401831 <+113>:   ret
   0x0000000000401832 <+114>:   nopw   0x0(%rax,%rax,1)
   0x0000000000401838 <+120>:   mov    %ebp,%eax
   0x000000000040183a <+122>:   add    $0x8,%rsp
   0x000000000040183e <+126>:   shr    $0x1f,%eax
   0x0000000000401841 <+129>:   pop    %rbx
   0x0000000000401842 <+130>:   add    %ebp,%eax
   0x0000000000401844 <+132>:   pop    %rbp
   0x0000000000401845 <+133>:   sar    %eax
   0x0000000000401847 <+135>:   ret
   0x0000000000401848 <+136>:   nopl   0x0(%rax,%rax,1)
   0x0000000000401850 <+144>:   add    $0x8,%rsp
   0x0000000000401854 <+148>:   lea    0x9(%rbp),%eax
   0x0000000000401857 <+151>:   pop    %rbx
   0x0000000000401858 <+152>:   pop    %rbp
   0x0000000000401859 <+153>:   ret
   0x000000000040185a <+154>:   nopw   0x0(%rax,%rax,1)
   0x0000000000401860 <+160>:   add    $0x8,%rsp
   0x0000000000401864 <+164>:   lea    -0x2(%rbp),%eax
   0x0000000000401867 <+167>:   pop    %rbx
   0x0000000000401868 <+168>:   pop    %rbp
   0x0000000000401869 <+169>:   ret
End of assembler dump.
```

Question 3: Assembly, Stack Discipline, Calling Convention, and x86-64, *cont.* (15 points)
Part C: Switch statement, *cont.* (6 points)

Consider also the following memory dump.

```
(gdb) x/20dw 0x498008
0x498008:       1952673397      544108393       560951142       0
0x498018:       -616376 -616448 -616448 -616448
0x498028:       -616472 -616448 -616461 -616448
0x498038:       -616392 -616448 -616416 -616448
0x498048:       -616416 0       0       0
(gdb) x/20xw 0x498008
0x498008:       0x74636e75      0x206e6f69      0x216f6f66      0x00000000
0x498018:       0xfff69848      0xfff69800      0xfff69800      0xfff69800
0x498028:       0xfff697e8      0xfff69800      0xfff697f3      0xfff69800
0x498038:       0xfff69838      0xfff69800      0xfff69820      0xfff69800
0x498048:       0xfff69820      0x00000000      0x00000000      0x00000000
```

(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 code? How do you know?

Hint: Give the case number not the address.

Continued on next page.

Question 4: Caching, Locality, Memory Hierarchy, Effective Access Time (15 points)
Part A: Caching (12 points)

Given a model described as follows:

- Associativity: 4-way set associative
- Total size: 128 bytes (not counting metadata)
- Block size: 16 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) (9 points): For each of the following addresses, please indicate if it hits, or misses, and if it misses, the type of miss:

Address	**Circle one (per row):**		**Circle one (per row):**			
0x2A	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0x80	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0x28	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xF7	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0x0A	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xEA	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xA8	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xF0	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xD5	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0xBA	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0x9A	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A
0x8F	Hit	Miss	Capacity	Compulsory/Cold	Conflict	N/A

Continued on next page

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

Imagine a computer system as follows:

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

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

What L1 cache access time is required for the overall effective memory access time to be 10nS?

Continued on next page.

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

Part A (2 points): Please identify one check, i.e. invariant that can be verified, that can be performed by the heap checker within an implicit list free block. This check should only consider a single block at a time. Assume that all blocks have headers and footers and that constant-time coalesce is possible.

Part B (2 points): Please identify one additional check, i.e. invariant that can be verified, that can be performed by the heap checker within an implicit list allocator. You may compare across adjacent blocks. And, you should assume that allocated blocks are footerless.

Part C (2 points): Please identify one additional check, i.e. invariant that can be verified, that can be performed by the heap checker within an explicit list allocated. You may compare across adjacent blocks.

Part D (2 points): Please identify one additional check, i.e. invariant that can be verified, that can be performed by the heap checker within a segregated list allocator. This check should only consider a single block at a time.

Part E (2 points): Is a best-fit policy more likely to be worth the cost in a segregated list allocator or a simple explicit list allocator? Why?

Continued on next page.

Question 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 8 bits wide.
- Physical addresses are 8 bits wide.
- The page size is 32 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 points): Please label the diagram below showing which bit positions are interpreted as each of the VPO and VPN. Leave any unused entries blank.

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
VPN/ VPO								

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

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TLBI/ TLBT								

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

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

Continued on next page.

Question 6: Virtual Memory, Paging, and the TLB (15 points)
Part B: Hits and Misses (12 points)

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

TLB

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

Set	Tag	PPN	BITS	Scratch space for you
0	00	1	X-RW	
0	10	5	X-R	
1	01	3	X-RW	
1	11	2	X-R	

Page Table

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

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

Continued on next page.

Question 6: Virtual Memory, Paging, and the TLB (15 points)
Part B: Hits and Misses, *cont.* (12 points)

Consider the following memory access trace e.g. sequence of memory operations listed in order of execution, as shown in the first two columns (operation, virtual address). It begins with the TLB and page table in the state shown above.

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

Please complete the remaining columns

Subpart	Operation	Virtual Address	TLB Hit or Miss?	Page Fault? Yes or No?	PPN If Knowable
1	Write	0x40	Hit Miss Not knowable	Yes No Not knowable	
2	Write	0x82	Hit Miss Not knowable	Yes No Not knowable	
3	Read	0x24	Hit Miss Not knowable	Yes No Not knowable	
4	Read	0xA1	Hit Miss Not knowable	Yes No Not knowable	
5	Read	0x22	Hit Miss Not knowable	Yes No Not knowable	
6	Write	0xA8	Hit Miss Not knowable	Yes No Not knowable	
7	Read	0xA5	Hit Miss Not knowable	Yes No Not knowable	
8	Write	0x43	Hit Miss Not knowable	Yes No Not knowable	

Continued on next page.

Question 7: Process Representation and Lifecycle + Signals and Files (10 points)
Part A (4 points):

Please consider the following code:

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

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

    if (!fork()) {
       printf ("C"); fflush(stdout);
    } else {
       printf ("D"); fflush(stdout);
    }
  }
  printf ("E"); fflush(stdout);
}
```

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

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

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

Continued on next page.

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

Please consider the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

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

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

  write(fd0, buffer, 2);

  if (fork()) {
    wait(NULL);
    write(fd0, "P", 1);
    write(fd0, buffer+3, 3);

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

    write(fd1, "X", 1);

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

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

 }

 return 0;
}
```

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

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

7(B)(3) (2 points): If the child process was just about to "return 0", how many inode entries associated with regular files in the file system are in use by these two processes?

Continued on next page.

Question #8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW (15 points)
Part A (6 points): Deadlock

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

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

8(A)(1) (3 points) Is it possible for the code above to deadlock? Yes No

8(A)(2) (3 points) Consider your answer to 8(A)(1) above. If you answered “No”, explain why deadlock is impossible. 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.

Continued on next page.

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

Consider a situation where you and your partner are working on an embedded systems project.

- Your partner assembles two (2) SMALL_PARTS into an ASSEMBLY.
- Your job is to test each assembly.
- Because of the shape of the parts, the desk can hold:
  - Three (3) SMALL_PARTS, OR
  - One (1) completed ASSEMBLY and one (1) SMALL_PART
- There is a bin for parts that pass the testing and another for parts that do not pass the testing. These bins are, for practical purposes, infinite in size.

Below and on the next two pages is C-like pseudocode for threads implementing your role and your partner's role, as well as for global declaration and initialization. This is just pseudocode. Don't let details unrelated to the concurrency control problem distract you. Read the provided comments: They are important.

Your task is to add proper concurrency control to the provide code. The only concurrency control primitives you can use are via the semaphore type and functions shown below:

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

```
// The space below should be used to declare and initialize any shared
variables.
```

Continued on next page.

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

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

```
void *partnerThread (void *args) {
   part_t part1, part2;
   assembly_t assembly;



   while (1) {



     part1 = getPartOne(); // No one gets parts, except your partner
     part2 = getPartTwo(); // No one gets parts, except your partner



     placePartOnDesk(part1); // You and your partner share the desk
     placePartOnDesk(part2); // as described above



     assembly = assembleParts(part1, part2);



     placeAssemblyOnDesk (assembly); // This is the same desk as above









  }
}
```

Continued on next page.

Question #8: Concurrency Control: Maladies, Semaphores, Mutexes, BB, RW, *cont.* (15 pts)
Part B (7 points): Concurrency Control, cont.

```
void *yourThread (void *args) {
  assembly_t assembly1;



  while (1) {



    assembly = getAssemblyFromDesk(); // You and your partner share the desk



    // No one conducts inspections or places parts in bins, except you
    if (PASSES_INSPECTION == inspectAssembly(assembly)) {



      storeAssembly(SELLABLE_BIN, assembly);



    } else {



      storeAssembly(REJECT_BIN, assembly)



  }



}
```

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