Andrew ID:
Full Name:

Hint: This is an old school handwritten exam. There is no authenticated login. If we can't read your AndrewID, we won't easily know who should get credit for this exam. If we can't read either your AndrewID or Full Name, we're in real bind. Please write neatly :-)

18-213/18-613, Spring 2022 Final Exam (SOLUTIONS)

Tuesday, May 10, 2022

Instructions:

- Make sure that your exam is not missing any sheets (check page numbers at bottom)
- Write your Andrew ID and full name on this page (and we suggest on each and every page)
- This exam is closed book and closed notes (except for 2 double-sided note sheets).
- You may not use any electronic devices or anything other than what we provide, your notes sheets, and writing implements, such as pens and pencils.
- Write your answers in the space provided for the problem.
- If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The point value of each problem is indicated.
- **Good luck!**

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

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

Part A: Integers (5 points, 1 point per blank)

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

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

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

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

Problem 1. (5 points):

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Numeric values are encoded as a value of the form $V = M \times 2^E$, where E is exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). Any rounding of the significand is based on *round-to-even*.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. Give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., 3/4).

Value	Floating Point Bits	Rounded value
9/32	*001 00*	1/4
1/32	*000 00*	*0*
1/16	*000 01*	*1/16*
3/32	*000 10*	*1/8*
1	*011 000*	*1*
12	*110 10*	*12*

Problem 2. (10 points):

Structs and arrays. The next two problems require understanding how C code accessing structures and arrays is compiled. Assume the x86-64 conventions for data sizes and alignments.

You are given the following C code:

```
#include "decls.h"

typedef struct {
    int x[CNT2];          /* Unknown constant */
    int y;
    int z[CNT3];          /* Unknown constant */
} struct_a;

typedef struct{
    struct_a data[CNT1];  /* Unknown constant */
    int idx;
} struct_b;

void set_y(struct_b *bp, int val)
{
    int idx = bp->idx;
    bp->data[idx].y = val;
}
```

You do not have a copy of the file `decls.h`, in which constants `CNT1`, `CNT2`, and `CNT3` are defined, but you have the following x86-64 code for the function `set_y`:

```
set_y:
    bp in %rdi, val in %esi
  movslq   168(%rdi),%rax
  leaq     (%rax,%rax,2), %rax
  movl     %esi, 12(%rdi,%rax,8)
  ret
```

Based on this code, determine the values of the three constants

A. `CNT1` = *7*

B. `CNT2` = *3*

C. `CNT3` = 2

Problem 3. (15 points):

Loops. Consider the following x86-64 assembly function, called `looped`:

```
looped:
        # a in %rdi, n in %esi
        Movl    $0, %edx
        testl   %esi, %esi
        Jle     .L4
        Movl    $0, %ecx
.L5:
        movslq  %ecx,%rax
        Movl    (%rdi,%rax,4), %eax
        Cmpl    %eax, %edx
        cmovl   %eax, %edx
        Incl    %ecx
        cmpl    %edx, %esi
        jg      .L5
.L4:
        movl    %edx, %eax
        ret
```

Fill in the blanks of the corresponding C code.

- You may only use the C variable names `n`, `a`, `i` and `x`, not register names.
- Use array notation in showing accesses or updates to elements of `a`.

```
int looped(int a[], int n)
{
    int i;
    int x = 0_____;

    for(i = 0_____; x < n_____; i++) {

        if (x < a[i]_____)

            x = a[i]_____;
    }
    return x;
}
```

Problem 4. (15 points):

Cache memories. This problem requires you to analyze both high-level and low-level aspects of caches. You will be required to perform part of a a cache translation, determine individual hits and misses, and analyze overall cache performance.

For this problem, you should assume the following:

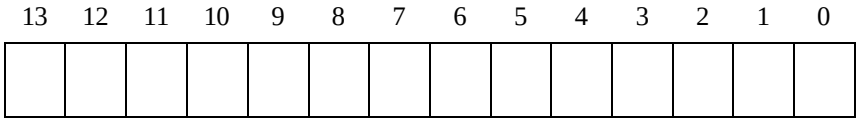
- Memory is byte addressable.
- Physical addesses are 14 bits wide.
- The cache is 2-way set associative with an 8 byte block-size and 2 sets.
- Least-Recently-Used (LRU) replacement policy is used.
- `sizeof(int)` = 4 bytes.

Continued on next page.

A. The following question deals with a matrix declared as `int arr[4][3]`. Assume that the array has already been initialized.

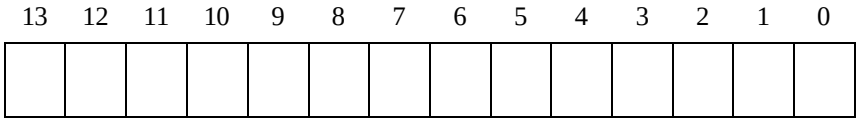
(a) (1 point) The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

`CO` The block offset within the cache line
`CI` The set index
`CT` The cache tag



(b) (1 point) Given that the address of `arr[0][0]` has value **0x2CCC**, perform a cache address translation to determine the block offset and set index for the first item in the array.

`CI = 0x`*1*_____
`CO = 0x`*4*_____



(c) (3 points) For each element in the matrix `int arr[4][3]`, label the diagram below with the set index that it will map to.

`arr[4][3]`	Col 0	Col 1	Col 2
Row 0	*1*	*0*	*0*
Row 1	*1*	*1*	*0*
Row 2	*0*	*1*	*1*
Row 3	*0*	*0*	*1*

B. (6 points) The following questions also deals with `int arr[4][3]` and the cache defined at the beginning of the problem. Assume the cache stores only the matrix elements; variables `i`, `j`, and `sum` are stored in registers.

```
int i, j;
int sum = 0;

for(i=0; i<4; i++){
  for(j=0; j<3; j++){
    sum += arr[i][j];
  }
}

/* second access begins */
for(i=2; i>=0; i=i-2){
  for(j=0; j<3; j++){
    sum += arr[i][j];
    sum += arr[i+1][j];
  }
}
/* second access ends */
```

Assume the above piece of code is executed. Fill out the table to indicate if the corresponding memory access will be a hit (**h**) or a miss (**m**) when accessing the array `arr[4][3]` for the **second** time (between the comments 'second access begins' and 'second access ends').

`arr[4][3]`	Col 0	Col 1	Col 2
Row 0	*M*	*M*	*H*
Row 1	*M*	*H*	*M*
Row 2	**h**	*H*	*H*
Row 3	*H*	*H*	*H*

The following grids can be used as scrap space:

C. The following question deals with a different matrix, declared as `int arr[5][5]`. Again assume that `i`, `j`, and `sum` are all stored in registers.

Consider the following piece of code:

```
#define ITERATIONS 1
int i, j, k;
int sum = 0;

for(k=0; k<ITERATIONS; k++){
  for(i=0; i<5; i++){
    for(j=0; j<5; j++){
      sum += arr[i][j];
    }
  }
}
```

For each of the following caches, specify the total number of **cache misses** for the above code. **Important:** Assume that the matrix is aligned so that `arr[0][0]` is the first element in a cache block.

(a) (2 points) If ITERATIONS is 1 (Total accesses: 25).

i. Direct-mapped, 16 byte block-size, 4 sets

Number of cache misses *7*

ii. 2-way set associative, 8 byte block-size, 2 sets

Number of cache misses *13*

(b) (2 points) If ITERATIONS is 2 (Total accesses: 50).

i. Direct-mapped, 64 byte block-size, 2 sets

Number of cache misses *2*

ii. 2-way set associative, 32 byte block-size, 1 set

Number of cache misses *8*

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

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

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

And a malloc implementation as below:

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

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

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

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

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


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

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

Soln: 56 bytes for allocated blocks

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

Soln: 0B

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

Soln: 32B

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

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

Soln:

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

6. Virtual Memory, Paging, and the TLB (15 points)

This problem concerns the way virtual addresses are translated into physical addresses. Imagine a system has the following parameters:

- Virtual addresses are 16 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 256 bytes.
- The TLB is 4-way set associative with 8 total entries.
- The TLB may cache invalid entries
- A single level page table is used

Part A: Interpreting addresses

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

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PPN/ PPO					*N*	*N*	*N*	*N*	*O*	*O*	*O*	*O*	*O*	*O*	*O*	*O*

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

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

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

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

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

Part B: Hits and Misses (12 points)

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

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

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

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

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

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

Please complete the remaining columns.

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

Problem 7. (5 points):

Process control.

A. What are the possible output sequences from the following program:

```
int main() {
    if (fork() == 0) {
        printf("a");
        exit(0);
    }
    else {
        printf("b");
        waitpid(-1, NULL, 0);
    }
    printf("c");
    exit(0);
}
```

Circle the possible output sequences: ***abc*** acb ***bac*** bca cab cba

B. What is the output of the following program?

```
pid_t pid;
int counter = 2;

void handler1(int sig) {
    counter = counter - 1;
    printf("%d", counter);
    fflush(stdout);
    exit(0);
}

int main() {
    signal(SIGUSR1, handler1);

    printf("%d", counter);
    fflush(stdout);

    if ((pid = fork()) == 0) {
        while(1) {};
    }
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```

OUTPUT: *213*_____

Problem 7. (5 points):

File I/O. This problem tests your understanding of how Linux represents and shares files. You are asked to show what each of the following programs prints as output:

- Assume that file `infile.txt` contains the ASCII text characters "`15213`";
- You may assume that system calls do not fail;
- When a process with no children invokes `waitpid(-1,NULL,0)`, this call returns immediately;
- *Hint*: each of the following questions has a unique answer.

A.

```
1  int main() {
2    int fd;
3    char c;
4
5    fd = open("infile.txt", O_RDONLY, 0);
6
7    fork();
8    waitpid(-1, NULL, 0);
9
10   read(fd, &c, sizeof(c));
11   printf("%c", c);
12
13   return 0;
14 }
```

OUTPUT: *15*_____

B.

```
1  int main() {
2    int fd;
3    char c;
4
5    fork();
6    waitpid(-1, NULL, 0);
7
8    fd = open("infile.txt", O_RDONLY, 0);
9
10   read(fd, &c, sizeof(c));
11   printf("%c", c);
12
13   return 0;
14 }
```

OUTPUT: *11*_____

Problem 8. (8 points):

Concurrency and sharing. Consider a concurrent C program with two threads and a shared global variable `cnt`. The threads execute the following lines of code:

Thread 1	Thread 2
`/* Increment cnt */` `cnt++;`	`/* Decrement cnt */` `cnt--;`

Suppose that these lines of C code compile to the following assembly language instructions:

Thread 1	Thread 2

`movl cnt,%eax  # L1: Load cnt`
`inc %eax  # U1: Update cnt`
`movl %eax,cnt  # S1: Store cnt`
	`movl cnt,%eax  # L2: Load cnt`
	`dec %eax  # U2: Update cnt`
	`movl %eax,cnt  # S2: Store cnt`

At runtime, the operating system kernel will choose some ordering of these instructions. Since we are not explicitly synchonizing the threads, some of these orderings will produce the correct value for `cnt` and others will not.

Each of the sequences shown below gives a possible ordering of the instructions when the two threads execute. Assuming that `cnt` is initially zero, what is the value of `cnt` in memory after each of the sequences completes?

A. cnt=0; L_1, U_1, S_1, L_2, U_2, S_2 `cnt ==` *0*

B. cnt=0; L_1, U_1, L_2, S_1, U_2, S_2 `cnt ==` *-1*

C. cnt=0; L_2, U_2, S_2, L_1, U_1, S_1 `cnt ==` *0*

D. cnt=0; L_1, L_2, U_2, S_2, U_1, S_1 `cnt ==` *1*

Problem 8. (7 points):

Synchronization. This question will test your understanding of synchronizations, deadlocks and use of semaphores. For these questions, assume each function is executed by a unique thread on a uniprocessor system.

A. Consider the following C code:

```
/* Initialize semaphores */
mutex1 = 1;
mutex2 = 1;
mutex3 = 1;
mutex4 = 1;
```

```
 void thread1() {

     P(mutex4); __
     P(mutex2); __
     P(mutex3); ____

     /* Access Data */

     V(mutex4); __
     V(mutex2); __
     V(mutex3); ____

 }
```

```
void thread2() {

     P(mutex1); __
     P(mutex2); __
     P(mutex4); ____

     /* Access Data */

     V(mutex1); __
     V(mutex2); __
     V(mutex4); ____

}
```

A. Can this code deadlock? *Yes* No

B. If yes, then indicate a feasible sequence of calls to the P or V operations that will result in a deadlock. Place an ascending sequence number (1, 2, 3, and so on) next to each operation in the order that it is **called**, even if it never returns. For example, if a P operation is called but blocks and never returns, you should assign it a sequence number.

Note that there are several correct solutions to this problem.

The deadlock occurs because mutex4 and mutex2 are allocated in different orders. Here is one example:

t1p4 -> t2p1 -> t2p2 -> t2p4 -> t1p2

At this point, thread 1 is blocked forever on mutex2 and thread 2 is blocked forever on mutex 4.

B. Consider the following three threads and three semaphores:

```
/* Initialize semaphores */
s1 = 1;
s2 = 0;
s3 = 0;

/* Initialize x */
x = 0;
```

```
void thread1()
{
    P(s1)
    x = x + 1;
    V(s2)
}
```

```
void thread2()
{
    P(s2)
    x = x + 2;
    V(s3)
}
```

```
void thread3()
{
    P(s3)
    x = x * 2;
    V(s1)
}
```

Add P(), V() semaphore operations (using semaphores s1, s2, s3) in the code for thread 1, 2 and 3 such that the concurrent execution of the three threads can only result in the value of x = 6.

Workspace/Scratch/Ungraded

Workspace/Scratch/Ungraded