Full Name: _____

"On my honor as a University of Colorado at Boulder student I have neither given nor received unauthorized assistance on this work."

CSCI 2400, Fall 2017

Practice Final Exam Solutions

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name on the front.
- Write your answers in the space provided for each problem. Feel free to use the back of each page to help you determine the answer, but make sure your answer is entered in the space provided on the front of the page. Then in the last 15 minutes of the class, you will use your laptop to upload your answers to the Moodle.
- This exam is CLOSED BOOK and no electronics are allowed, except a laptop that is only to be used in the last 15 minutes of the exam to upload your answers to the Moodle. You can use one page of personal notes and the printed midterm packet of tables.

Problem	Possible	Score
1		
2		
3		
4		
5-7		
8		
Total		

1. **[Points]** The number of printfs is **12** (full 5 points)
   Partial credit:
   - 1 point for Answers: 16
   - 2 points for Answers: 8
   - 3 points for Answers: 6, 24
2. **[Points]**
   (a) How many SIGCHLD signals get generated as this program executes? **Answer: 1 [+4]**
   (b) For the parent process that started the main() function running in while loop, If the user input ctrl+c , what are the printed values for the count1 and count2 variables? **Answer: Count1 = 2, Count2 = 0 [+4]**
   (c) For the parent process that started the main() function running in while loop, If the user input ctrl+z , what are the printed values for the count1 and count2 variables? **Answer: Count1 = 1, Count2 = 1 [+4]**

3. **[Points]**

This question is designed to test your understanding of memory addressing.

- Virtual addresses are 16 bits wide.
- The page size is 256 bytes.
- The TLB is 2-way set associative with 8 total entries.
- Cache is Direct-Mapped, with 16 sets and 16 bytes-per-block.
- Physical addresses are 12 bits wide.
- The memory is byte addressable.
- Memory accesses are to 1-byte words.

In the following tables, **all numbers are given in hexadecimal**. 'V?' is used to indicate the 'Valid?' bit for entries in the page table, TLB, and cache. The contents of the TLB and cache, as well as the first 32 elements in the page table are included here for your reference:

TLB

	Tag	PPN	V?		Tag	PPN	V?
0	05	6	1		24	1	1
1	12	0	1		30	5	1
2	33	d	0		15	d	1
3	07	b	1		23	1	0

Cache

	V?	Tag	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	1	1	1c	9f	be	c2	a9	7f	3d	8b	6e	3a	f9	a6	a2	c9	f4	de
1	1	e	e8	ba	c2	ba	5b	3a	1c	64	a1	59	0b	6a	28	64	c5	23
2	0	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	1	d	11	60	d2	d3	0c	41	af	d8	a0	da	33	1d	30	f0	65	5b
4	0	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	1	5	33	31	2a	20	c0	16	12	ec	73	a8	b1	5c	89	3f	bc	35
7	1	b	00	6c	36	ea	ea	a1	e3	bb	6d	fc	e4	77	75	57	e8	74
8	1	b	6b	11	6c	e2	af	57	8f	d0	c5	24	ce	ec	9c	b7	8b	b9
9	1	e	84	58	11	6a	fa	e1	50	b8	bb	e4	c2	65	15	5e	f6	1a
a	1	7	0c	b8	61	db	32	90	7a	08	01	42	eb	45	2b	6a	47	49
b	1	c	fc	d5	5b	b8	72	61	6f	83	a7	24	27	b5	0c	87	2e	51
c	1	9	56	5c	57	0d	35	14	38	b1	d2	59	0d	69	e2	34	9c	4b
d	1	9	db	69	a9	c1	81	0c	de	c6	a5	d3	57	d5	d0	97	1d	9d
e	1	b	3e	c7	00	ed	cf	6f	52	6a	0d	1c	f1	0c	89	bf	f8	c2
f	1	6	6d	20	23	db	8c	52	77	c9	73	35	8e	2f	cf	19	13	8b

Page Table

VPN	PPN	V?	VPN	PPN	V?
00	-	0	10	a	1
01	-	0	11	-	0
02	6	1	12	1	1
03	b	1	13	-	0
04	5	1	14	6	1
05	-	0	15	b	1
06	-	0	16	f	1
07	-	0	17	e	1
08	2	1	18	9	1
09	4	1	19	0	1
0a	c	1	1a	b	1
0b	0	1	1b	5	1
0c	-	0	1c	c	1
0d	f	1	1d	6	1
0e	1	1	1e	-	0
0f	d	1	1f	b	1

The first questions you'll be answering for this problem are to do with what parts of the virtual and physical addresses map to what. Here, each square cell represents one bit of the address, and you'll be labelling each bit with 1 or 2 of the acronyms which follow. Not all of these components are present in, for example, a virtual address. But every bit should be associated with at least one component.

PO Page Offset

VPN Virtual Page Number

PPN Physical Page Number

TLBI TLB Index

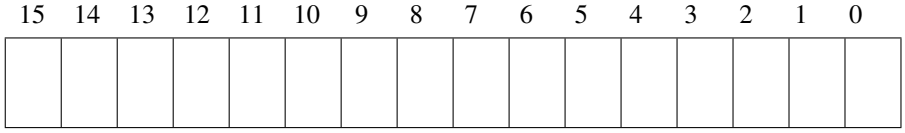
TLBT TLB Tag

CI Cache Index

CT Cache Tag

CO Cache Offset

(a) **[Points]** Label the components of a Virtual Address:



Answer:

The VPN is (15-8), the VPO is (7-0).

The TLBT is (15-10), the TLBI is bit (8-9).

(b) **[Points]** Label the components of a Physical Address:

11 10 9 8 7 6 5 4 3 2 1 0

Answer:
The PPN is (11-8) and the PPO is (7-0).

For each of the given virtual addresses below, fill out all values in the associated table. If a value is invalid or can't be known, write a '-'.

(c) **[Points] Virtual address**: `0x1f45`

Parameter	Value
VPN	0x1F
TLB Index	0x3
TLB Tag	0x07
TLB Hit? (Y/N)	Y
Page Fault? (Y/N)	N
PPN	0x0xb
Physical Address	0xb45

(d) **[Points] Virtual address**: `0x0cf5`

Parameter	Value
VPN	0x0c
TLB Index	0x0
TLB Tag	0x03
TLB Hit? (Y/N)	-
Page Fault? (Y/N)	Y
PPN	0x-
Physical Address	0x-

(e) **[Points]** This next question will ask about the cache. In order to minimize the effects of error propogation, start with a fresh **Physical Address**: `0xbed`, and fill out the values for the following parameters (using the same memory system described above).

Parameter	Value
CO	0xd
CI	0xe
CT	0xb
Cache Hit? (Y/N)	Y
Byte Returned?	0x00

4. **[Points]**

Suppose our memory allocator uses an implicit free list with both header and footer. Assume a word size of eight bytes, and that all blocks are aligned to addresses divisible by eight. You should assume that the addresses you see span the entire heap, and thus if there is not room for a block size requested by `malloc`, it will fail. You should also assume that a block is marked as allocated by setting the least significant bit of the header and footer to 1. Similarly, a block is marked as free by setting the least significant bit of the header and footer to 0. Note that each row in the heap pictured below represents one eight-byte word.

Address	Value
FF00	00 . . . 00 39
FF08	?? . . . ??
FF10	?? . . . ??
FF18	?? . . . ??
FF20	?? . . . ??
FF28	?? . . . ??
FF30	00 . . . 00 39
FF38	00 . . . 00 21
FF40	?? . . . ??
FF48	?? . . . ??
FF50	00 . . . 00 21
FF58	00 . . . 00 29
FF60	?? . . . ??
FF68	?? . . . ??
FF70	?? . . . ??
FF78	00 . . . 00 29
FF80	00 . . . 00 20
FF88	?? . . . ??
FF90	?? . . . ??
FF98	00 . . . 00 20
FFA0	00 . . . 00 21
FFA8	?? . . . ??
FFB0	?? . . . ??
FFB8	00 . . . 00 21
FFC0	00 . . . 00 41
FFC8	?? . . . ??
FFD0	?? . . . ??
FFD8	?? . . . ??
FFE0	?? . . . ??
FFE8	?? . . . ??
FFF0	?? . . . ??
FFF8	00 . . . 00 41

Unless clearly marked otherwise, assume all numbers are in hexadecimal!

Suppose that, after some sequence of `malloc`'s and `free`'s, the state of the heap is as you see it on the left. Then, assume that the following calls to `malloc` and `free` are made:

```
l0: free(0xff40);
l1: free(0xff60);
l2: void* p1 = malloc(0x50);
l3: void* p2 = malloc(0x30);
l4: free(0xffa8);
```

And answer the following:

(a) **[Points]** What is `p1`?

Answer:
0xff40

(b) **[Points]** What is `p2`?

Answer:
0

(c) **[Points]** What is the largest requested size that malloc could successfully complete after `l4`? (For example, in `l2`, `malloc` is called with a requested size of $0x50$.)

Answer:
0x10 (or 16)

(d) **[Points]** What is the largest requested size that `malloc` could have successfully completed after `l1`?

Answer:
0x58 (or 88)

Unless clearly marked otherwise, assume all numbers are in hexadecimal!

5. **[Points]** For the following code, identify the symbols listed in the symbol table of the ELF relocatable object files (.o), whether that symbol is defined or undefined, and if defined, then in which section of the corresponding ELF file that the symbol would be defined. **Note: if the symbol is not in the table or not defined in the data section, use '–' to indicate**.

main.c

```
extern int fibonacci();
char lizard = 's';
int n;

int main(){
  int answer = fibonacci();
  return 0;
}
```

fib.c

```
extern int star;
int n = 7;

int fibonacci(){
  int first, second, next;
  for (int i=0; i<n; i++){
    if (i<=1){
      next = i;
    } else {
      next = first + second;
      first = second;
      second = next;
    }
  }
  return next;
}
```

main.o

Symbol Name	Defined/undefined	Section
fibonacci	undefined	–
lizard	defined	.data
main	defined	.text
n	defined	.bss

fib

Symbol Name	Defined/undefined	Section
fibonacci	defined	.text
n	defined	.data
star	undefined	–

6. **[Points]** For the code in Question 1, the sizes of the .text and .data sections of the .o relocatable object files are listed below. The two object files above are then linked together with the command line `ld -o p main.o fib.o`. Assume the object files are combined similar to the order shown in the lecture slides and the starting address of the .text section of the unified executable object file starts at 0x62c9f75d. What is the relocated address of lizard?

File+Section	Size (Byte)
main.o's .text	22
main.o's .data	1
fib.o's .text	58
fib.o's .data	4

Answer:

x is initialized so it is in the main.o's .data section. Then the relocation address is: starting address + main.o's .text + fib.o's .text = 0x62c9f75d + 22 + 58 = 0x62c9f75d + 0x50 = 0x62c9f7ad. +4 for 0x62c9f7ad, +2 for 0x62c9f75d + 22 + 58 + 1 = 0x62c9f75d + 0x51 = 0x62c9f7ae, +1 for 0x62c9f75d + 22 + 58 + 1 + 4 = 0x62c9f7ad + 0x55 = 0x62c9f7b2, otherwise 0.

7. **[Points]** Also for the code in Question 1, when the two .o files above are linked together with the command line `ld -o p main.o fib.o`, there are the following claimed statements about the runtime addresses of the merged and relocated various subsections' ordering, from lowest to highest addresses:

   (i) .data(fib.o) $<$ .data(main.o) $<$ .bss(fib.o)

   (ii) .data(main.o) $<$ .text(main.o) $<$ .bss(main.o)

   (iii) .data(main.o) $<$ .bss(main.o) $<$ .bss(fib.o)

   (iv) .text(fib.o) $<$ .data(main.o) $<$ .data(fib.o)

   (v) .bss(main.o) $<$ .data(fib.o) $<$ .text(main.o)

   Which one of the following statements below is correct?

Answer:

iii and iv are the only two correct answers. +3 for (b), +2 for (d) or (e), otherwise 0.