CSE 410 Final Exam 12/10/13

Name __________________________________

Do **not** write your id number or any other confidential information on this page.

There are ***10*** problems worth a total of 120 points. The point value of each problem is indicated in the table on the next page. Write your answers neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are mostly independent of each other.

The last pages of the test contains a table of powers-of-two and reminders about some common x86 instructions and conventions. Feel free to separate these pages from the rest of the exam. Other pages containing code for questions can also be detached if you like – the bottom of the page will indicate if this is okay.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers. Please wait to turn the page until everyone is told to begin.

Score ___________________ / 120

1. _______ / 15

2. _______ / 15

3. _______ / 7

4. _______ / 7

5. _______ / 14

6. _______ / 10

7. _______ / 25

8. _______ / 6

9. _______ / 6

10. _______ / 15

Question 1. (15 points) (some mystery code, or the ghosts of midterms past)

Once again one of the interns has lost the source code to an important function. We have been able to discover that the function starts like this:

```
int f(int a, int b, int c) { ... }
```

But beyond that, all we've been able to find is an assembly file produced by gcc when it compiled the function on an x86-64 machine:

```
f:      cmpl   %esi, %edi
        jle    .L2
        leal   (%rsi,%rdx), %eax
        ret
.L2:
        addl   %esi, %edi
        cmpl   %edx, %esi
        movl   $0, %eax
        cmovg  %edi, %eax    # cmovg = conditional move greater
        ret
```

In the space below, translate the assembly language function given above into C. The function heading is written for you. (Reminder: there is useful reference information on the last two pages of the exam.)

```
int f(int a, int b, int c) {




}
```

Question 2. (15 points) (buffers and stack frames) Consider the following function, which calls the same `Gets` function used in the buffer overflow lab to read a sequence of bytes.

```
int f(int a, int b) {
  char s[2];
  int x=a;
  int y=x+b;
  Gets(s);
  return y;
}
```

When this function was compiled on an x86-64 machine, `gcc` produced the following assembly code:

```
f:  pushq  %rbp
    movq   %rsp, %rbp
    subq   $32, %rsp     #### location for (a), next page ####
    movl   %edi, -20(%rbp)
    movl   %esi, -24(%rbp)
    movl   -20(%rbp), %eax
    movl   %eax, -4(%rbp)
    movl   -24(%rbp), %eax
    movl   -4(%rbp), %edx
    addl   %edx, %eax
    movl   %eax, -8(%rbp)
    leaq   -16(%rbp), %rax
    movq   %rax, %rdi
    call   Gets
    movl   -8(%rbp), %eax
    leave
    ret
```

Answer questions about this function on the next page. You may remove this page for reference if you wish.

Question 2. (cont.) (a) (10 points) Below is a chart showing the layout of the stack right after execution of the `pushq/movq/subq` instructions at the beginning of the function, marked by `####` in the code. The picture is drawn using 32-bit words since almost all of the values in the stack frame are 32-bit integers.

Label each 32-bit word below with the name of the variable or temporary value it contains. If some word or parts of a word are unused you should leave them blank. Be sure to show where the `char` array `s` is located, even though it does not occupy a full 32-bit word. Also show where the return address and old `%rbp` values that have been pushed onto the stack are located. (And remember that those addresses are 64-bit values so they will occupy two of these 32-bit slots.)

Offset from `%rbp`:		
+16		
+12		
+8		
+4		
0		← `%rbp`
-4		
-8		
-12		
-16		
-20		
-24		
-28		
-32		← `%rsp`
-36		

(b) (5 points) Give the values of a string of bytes to be read by `Gets` that will cause this function to return the value 7 instead of the value it would normally return. You should give your answer as a string of hex digits giving the byte values for the input in the same format used as input to `sendstring` in lab 3, i.e., a pair of hex digits for each byte, like `31 32 33`.

Some short questions about the memory hierarchy. You are not required to show your work, but it's not a bad idea to show some details in case we need to figure out what happened if we need to award partial credit.

Question 3. (7 points) (cache geometry) The Intel i7 processor has a L3 cache with the following characteristics:

Total data size	8MB
Block size	64 bytes
16-way associative	

How many sets (rows) are there in this cache?

Question 4. (7 points) (access times) Suppose we have a memory system with a single-level cache and the following characteristics:

Cache access time	2 nsec
Main memory access time	300 nsec
Hit ratio	98%

What is the average access time of this memory system?

Question 5. (14 points) (hit or miss?)

(a) (7 points) Suppose we have a direct-mapped cache containing 128 (`0x80`) total bytes with 32-byte (`0x20`) cache blocks. What is the miss rate of the following code?

```
double x[32], y[32];
int i;

for (i = 0; i < 32; i++) {
  y[i] = 2*x[i];
}
```

Assumptions:

- The cache is initially empty.
- Array `x` begins at memory address `0x100` and array `y` begins at memory address `0x200`.
- All variables and code other than the arrays `x` and `y` are stored in registers (i.e., they do not affect the data cache).
- Doubles occupy 8 bytes each.

(b) (7 points) Now suppose we replace the cache from part (a) with another cache that has the same total size of 128 bytes, same block size of 32 bytes, but is 2-way associative (i.e., each set has two blocks and there are half as many sets as in part (a)). What is the miss rate now if we execute the same code from part (a) under the same assumptions except for these changes?

Question 6. (10 points) (which is best?) Here are two functions that store zeros in the upper-right triangular half of a square array.

```
#define SIZE 10000

void zero1(double matrix[SIZE][SIZE]) {
  int r,c;
  for (c=0; c<SIZE; c++) {
    for (r=0; r<=c; r++) {
      matrix[r][c] = 0.0;
    }
  }
}

void zero2(double matrix[SIZE][SIZE]) {
  int r,c;
  for (r=0; r<SIZE; r++) {
    for (c=r; c<SIZE; c++) {
      matrix[r][c] = 0.0;
    }
  }
}
```

Given that they both do the same thing, is there any reason to prefer one over the other? Give a brief technical justification for your answer.

Question 7. (25 points) (caches and virtual memory)
We have a memory system with the following characteristics:

- 16 bit virtual addresses (4 hex digits), page size of 64 bytes
- 12 bit physical addresses (3 hex digits), same page size (of course)
- Memory cache with 16 entries, direct mapped, 4-byte blocks
- Page table with 1024 entries; only the first 16 shown below
- TLB with 16 entries, 4-way set associative

The current state of the memory system is shown in the following tables. You can remove this page for reference while working on the parts of this question.

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	33	1	08	–	0	06	–	0	03	11	1
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page Table (First 16 entries)

VPN	PPN	Valid	VPN	PPN	Valid
000	28	1	008	13	1
001	–	0	009	17	1
002	09	1	00A	33	1
003	02	1	00B	–	0
004	–	0	00C	–	0
005	16	1	00D	2D	1
006	–	0	00E	11	1
007	–	0	00F	0D	1

Cache

Index	Tag	Valid	B0	B1	B2	B3	Index	Tag	Valid	B0	B1	B2	B3
0	28	1	99	1F	23	11	8	11	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Question 7. (cont.) (a) (5 points) Label the bits corresponding to each of the components of the virtual address, namely, the virtual page number (VPN), the virtual page offset (VPO), the TLB set index (TLBI), and the TLB tag value (TLBT).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(b) (5 points) Label the bits corresponding to each of the components of the physical address, namely, the physical page number (PPN), the physical page offset (PPO), the cache set index (CI), the cache tag value (CT), and the cache byte offset (CO).

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

(c) (15 points) Indicate the result when each virtual address in the table below is used to access memory. You should specify whether there is a TLB miss, page fault, and/or cache miss, the physical address referenced, and the contents of memory at that location. In some cases there is not enough information to determine what value is accessed or whether there is a cache miss or not. In those cases, write ND (for Not Determinable) in the appropriate entry. Fill in each row of the table using the initial conditions shown in the tables on the previous page; accesses in previous rows do not affect the result of later rows. (Hint: There is a binary-hex conversion table at the end of the test.)

Virtual Address	Physical Address	Value	TLB Miss?	Page Fault?	Cache Miss?
0x03A0					
0x006C					
0x0002					

A couple of short questions on disks and files.

Question 8. (6 points) Suppose we have a hard disk that rotates at 6000 rpm (100 revolutions per second) and has an average seek time of 5 msec. What is the average expected time to access a block at some arbitrary location on the disk?

Question 9. (6 points) What's the difference between the directory entry for a file and the file's inode on a classic Unix file system? A brief answer should be sufficient.

Question 10. (15 points) Almost done! Consider the following program:

```
int main() {
  int p, q;
  int val = 1;
  p = fork();
  printf("fork returned %d\n", p);
  if (p > 0) {
    q = fork();
    val++;
    printf("fork returned %d\n", q);
    printf("val = %d\n", val);
  } else {
    printf("adios\n");
  }
  return 0;
}
```

For this problem, assume that there are no other processes on the system, and that when we run this program, the process id of the initial process is 1000. Each time a new process is created by `fork()` the new process is assigned the next available number: 1001, 1002, and so forth.

Below show two possible output sequences written by this program when it is executed. If the program can only produce one possible output sequence, give that sequence and explain why it is the only one possible.

Best wishes for the holidays!

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024$	$2^{-10} = .0009765625$

Assembly Code Instructions:

`push`	push a value onto the stack and decrement the stack pointer
`pop`	pop a value from the stack and increment the stack pointer
`call`	jump to a procedure after first pushing a return address onto the stack
`ret`	pop return address from stack and jump there
`mov`	move a value between registers or registers and memory
`cmov`*cc*	conditionally move a value between registers or registers and memory depending on condition codes *cc*.
`lea`	compute effective address and store in a register
`add`	add 1st operand to 2nd with result stored in 2nd
`sub`	subtract 1st operand from 2nd with result stored in 2nd
`and`	bit-wise AND of two operands with result stored in 2nd
`or`	bit-wise OR of two operands with result stored in 2nd
`sar`	shift data in the 2nd operand to the right (arithmetic shift) by the number of bits specified in the 1st operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`j`*cc*	conditional jump to address depending on condition codes *cc* (many possible versions such as `jle`, `jg`, `je`, `ja`, etc.)
`cmp`	subtract 1st operand from 2nd operand and set flags
`test`	bit-wise AND 1st operand from 2nd operand and set flags

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15`. `rsp` is a special register.

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Binary-Hex conversion

```
0x0 = 0b0000          0x8 = 0b1000
0x1 = 0b0001          0x9 = 0b1001
0x2 = 0b0010          0xA = 0b1010
0x3 = 0b0011          0xB = 0b1011
0x4 = 0b0100          0xC = 0b1100
0x5 = 0b0101          0xD = 0b1101
0x6 = 0b0110          0xE = 0b1110
0x7 = 0b0111          0xF = 0b1111
```