CSE 410 Final Exam 6/05/12

Name _________________________________

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

There are ***13*** problems worth a total of 100 points. The point value of each problem is indicated in the table on the next page. Write your answer 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 page of the test contains a table of powers-of-two and reminders about some common x86 instructions and conventions. Feel free to separate that page from the exam if it is convenient. Other pages containing code for questions can also be detached if convenient – 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.

CSE 410 Final Exam 6/05/12

Score __________________ / 100

1. ______ / 10

2. ______ / 10

3. ______ / 10

4. ______ / 10

5. ______ / 8

6. ______ / 8

7. ______ / 3

8. ______ / 4

9. ______ / 3

10. ______ / 6

11. ______ / 6

12. ______ / 12

13. ______ / 10

Question 1. (10 points) (bits) An array of boolean (0/1) values can be stored very compactly if we store 8 bits of the array in each byte. For instance here is an array of 24 bits stored in 3 bytes.

0	0	0	0	0	0	1	0	0	1	1	1	1	0	0	0	1	0	0	0	0	0	0	1
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				2							1									0			

In this array, bits 0, 7, 11-14, and 17 are 1 and all the rest are 0. The bits of the array are numbered from right to left. The bytes holding the bits are also numbered right to left, as shown by the numbers in the last row.

For this problem, complete the following C function so it returns the value of the specified bit in the array. You may assume that parameter `index` has an appropriate value (i.e., you don't need to worry about what to do if it is negative, or larger than the array size, or too large to be stored in a signed `int` variable). The array pointer `b` has type `char *`, which means that it points to an array of bytes.

Hint: shifts, `&`, etc. Suggestion: break your code into a few small assignment statements to make it easier to follow.

```
/* return the value of the bit whose location in */
/* bit array b is given by the variable index.   */
int getBit(char * b, int index) {










}
```

Question 2. (10 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 and ends like this:

```
int g(int x, int y) {
  int ans;
   …
  return ans;
}
```

But beyond that, all we've been able to find is some output produced using gdb. When we disassemble the code, here's what we get:

```
(gdb) disas g
Dump of assembler code for function g:
   0x0000000000400474 <+0>:      push   %rbp
   0x0000000000400475 <+1>:      mov    %rsp,%rbp
   0x0000000000400478 <+4>:      mov    %edi,-0x14(%rbp)
   0x000000000040047b <+7>:      mov    %esi,-0x18(%rbp)
   0x000000000040047e <+10>:     cmpl   $0x6,-0x14(%rbp)
   0x0000000000400482 <+14>:     ja     0x4004b4 <g+64>
   0x0000000000400484 <+16>:     mov    -0x14(%rbp),%eax
   0x0000000000400487 <+19>:     mov    0x4005d8(,%rax,8),%rax
   0x000000000040048f <+27>:     jmpq   *%rax
   0x0000000000400491 <+29>:     movl   $0x1,-0x4(%rbp)
   0x0000000000400498 <+36>:     jmp    0x4004bb <g+71>
   0x000000000040049a <+38>:     mov    -0x18(%rbp),%eax
   0x000000000040049d <+41>:     mov    %eax,-0x4(%rbp)
   0x00000000004004a0 <+44>:     jmp    0x4004bb <g+71>
   0x00000000004004a2 <+46>:     movl   $0x7,-0x4(%rbp)
   0x00000000004004a9 <+53>:     jmp    0x4004bb <g+71>
   0x00000000004004ab <+55>:     movl   $0x6,-0x4(%rbp)
   0x00000000004004b2 <+62>:     jmp    0x4004bb <g+71>
   0x00000000004004b4 <+64>:     movl   $0x0,-0x4(%rbp)
   0x00000000004004bb <+71>:     mov    -0x4(%rbp),%eax
   0x00000000004004be <+74>:     pop    %rbp
   0x00000000004004bf <+75>:     retq
```

And a look at the memory locations apparently referenced in the code shows this:

```
(gdb) x /10gx 0x4005d8
0x4005d8:  0x00000000004004b4    0x000000000040049a
0x4005e8:  0x000000000040049a    0x0000000000400491
0x4005f8:  0x00000000004004a2    0x00000000004004ab
0x400608:  0x00000000004004a2    0x0000002c3b031b01
0x400618:  0xfffffe6400000004    0xfffffeb000000048
```

(continued next page – feel free to remove this page for reference)

Question 2 (cont.) In the space below, translate the assembly language function given on the previous page into C. The function heading and return statement at the end are written for you.

```
int g(int x, int y) {
  int ans;




  return ans;
}
```

Question 3. (10 points) (buffers) 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 x, y;
  char s[2];
  x = 1;
  y = x+1;
  Gets(s);
  return y;
}
```

When this function was assembled on an x86-64 machine, it produced the following assembly code:

```
f: pushq   %rbp
   movq    %rsp, %rbp
   subq    $16, %rsp
   movl    $1, -4(%rbp)
   movl    -4(%rbp), %eax
   addl    $1, %eax
   movl    %eax, -8(%rbp)
   leaq    -16(%rbp), %rax
   movq    %rax, %rdi
   call    Gets
   movl    -8(%rbp), %eax
   leave
   ret
```

Your job is to create a string of bytes to be read by `Gets` that will cause this function to return the value 5 instead of the value it ordinarily returns, which is 2.

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`.

Hint: you will find it useful to sketch the layout of the stack frame and the variables in it to decide how many bytes your exploit string needs and what the contents should be.

Question 4. (10 points) (caches) You are not required to show your work on these questions, but showing some work might help if it is necessary to award partial credit.

(a) Suppose we have a 512 byte, 2-way set associative cache with a block size of 32 bytes. How many sets (rows) are there in this cache?

(b) Suppose we have a memory system with a single level cache. The cache has an access time of 2ns and main memory has an access time of 200ns. The average hit rate is 98%. What is the effective access time of this memory system?

(c) True or false: Increasing the associativity of a cache is likely to reduce conflict misses.

(d) True or false: Even a small decrease in the miss rate of a cache can result in a dramatic performance increase.

Question 5. (8 points) (Hit or miss?)

Suppose we have a direct-mapped cache containing 128 bytes total with 16-byte cache blocks. What is the miss rate of the following code?

```
int x[2][32];
int i;
int sum = 0;

for (i = 0; i < 32; i++) {
   sum += x[0][i] * x[1][i];
}
```

Assumptions:

- The cache is initially empty.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- All variables and code other than the array `x` itself are stored in registers (i.e., they do not affect the cache).
- Integers occupy 4 bytes each.

Question 6. (8 points) (Virtual memory) Suppose we have a system with 32-bit virtual addresses, 20-bit physical (real memory) addresses, and 8K pages.

(a) How many bits are included in the virtual page number and page offset parts of a virtual address on this system?

Virtual page number bits ________________

Page offset bits __________________

(b) How many bits are included in the physical page number part of a physical (real memory) address?

Physical page number bits ________________

(c) How many page table entries (PTEs) are needed for this virtual memory system?

Number of PTEs _______________________

Question 7. (3 points) (VM again) One of the reasons a virtual memory system may perform poorly is if it starts *thrashing*. **Very** briefly, what does this mean?

Question 8. (4 points) (something exceptional) After the operating system handles an exception, the system can either resume execution of the interrupted process at the instruction following the one that caused the exception, resume execution by re-executing the instruction that caused the exception, or terminate the process. Below is a list of possible exceptions. For each one, indicate by writing a, b, or c what the OS would normally do after handling the exception.

(a) Continue execution at the instruction following the one that generated the exception.

(b) Continue execution by re-executing the instruction that caused the exception.

(c) Terminate the process

______ Division by 0

______ Process causes a page fault

______ Memory reference out of bounds (e.g., segfault)

______ System call (`int`) instruction executed by process to read data

Question 9. (3 points) In the classic Unix file system, a directory is a special file containing one entry for each file in that directory. What information is stored in the directory entry itself along with the file name? Place an X next to all of the following that are found in the directory entry, and only check items that are included in the directory file itself as opposed to elsewhere on the disk. Leave all the answers blank if none of them apply.

_____ disk address of first file data block

_____ inode number of the file

_____ full path name of the file (e.g., `/a/b/c`)

Question 10. (6 points) (OS hardware support) Some of the features included in modern processors were introduced because they were needed to implement multi-processing operating systems. For each of the following, put an X in the blank space if the feature is included to provide support for operating systems. Leave the entry blank if it is an ordinary feature of the instruction set not specific to operating system support.

_____ condition code registers

_____ system call (`int`) instruction

_____ privileged execution mode

_____ call instruction

_____ stack pointer (sp) register

_____ interval timer (can be set to cause an interrupt/exception after a short time)

Question 11. (6 points) (OS data structures) Which of the following are items that are likely to be found in the Process Control Block (PCB) used by the operating system to describe a process? Place an X in the blank space if it is something that would appear in a PCB, leave the entry blank if it is not.

_____ Current process state (running, waiting, etc.)

_____ Current time of day

_____ CPU time used by process

_____ Link to next PCB on the same linked list (ready or waiting queue, for example)

_____ Contents of the process' virtual memory page tables

_____ Process program counter if the process is not running.

Question 12. (12 points) (Compare and contrast☺) For each of the following pairs of terms, give a ***very* brief** explanation of the key difference between the two terms or concepts. "Very brief" = a sentence or two that shows you understand the key ideas.

(a) program, process

(b) write-through, write-back

(c) temporal locality, spatial locality

(d) process, thread

Question 13. (10 points) (finis) Consider the following program:

```
int main() {
  if (fork() != 0) {
    printf("good ");
  } else {
    fork();
    printf("bye ");
    printf("y'all ");
  }
  return 0;
}
```

We'd like to know what output can be produced when this program is run. If it is possible for the program to produce different output if it is run more than once, give two of the possible results. If it always produces the same output, give that output and indicate that it is unique.

Have a great summer!!!

REFERENCE:

Powers of 2:

$2^0 = 1$	$2^8 = 256$
$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{20} = 1048576$
$2^6 = 64$	$2^{30} = 1073741824$
$2^7 = 128$	$2^{32} = 4294967296$

Assembly Code Instructions:

pushl	push a value onto the stack
leave	restore ebp or rbp from the stack
ret	pop return address from stack and jump there
movl	move 4 bytes between immediate values, registers and memory
movzbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes with 0s.
movsbl	move 1 byte into the low-order byte of a long word, filling the other 3 bytes by sign-extending the low-order byte that was moved
addl	add first operand to second with result stored in second
subl	subtract first operand from second with result stored in second
imull	multiply first operand and second with result stored in second
sall	left shift second operand by count given in first operand
sarl	right shift second operand by count given in first operand
andl	logical bitwise AND of first and second operands, result stored in second
xorl	logical bitwise XOR of first and second operands, result stored in second
jmp	jump to address
je	conditional jump to address if zero flag set
jne	conditional jump to address if zero flag is not set
cmpl	subtract first operand from second and set flags
testl	logical and of first and second operands to set flags
nop	“no operation” – does nothing

x86-64 Parameter Registers

First 6 arguments passed in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9 in that order.