NAME: ______________________________ UID: ______________

CS 4400: Computer Systems

Final Exam
Fall 2010

Please give your solutions in the space provided on the exam. If you choose to show your work on the exam, be sure to clearly indicate your final solution to each problem.

The exam is open-book, but closed-notes. *In addition, no laptops, calculators, cell phones, or other electronic devices are allowed.*

The point value of each question is clearly marked, so allocate your time wisely. The exam is worth a total of 100 points.

You must complete all work by 3p, there are no exceptions.

Make sure that you have 16 numbered pages.

Problem 1	/ 16 points
Problem 2	/ 14 points
Problem 3	/ 12 points
Problem 4	/ 15 points
Problem 5	/ 9 points
Problem 6	/ 6 points
Problem 7	/ 6 points
Problem 8	/ 12 points
Problem 9	/ 10 points
Total	/ 100 points

/ 16 points

1. The following problem concerns the way virtual addresses are translated into physical addresses.

   - The memory is byte addressable.
   - Memory accesses are to **1-byte words** (not 4-byte words).
   - Virtual addresses are 16 bits wide.
   - Physical addresses are 13 bits wide.
   - The page size is 512 bytes.
   - The TLB is 8-way set associative with 16 total entries.
   - The cache is 2-way set associative, with a block size of 4 bytes and a capacity of 64 bytes.

   In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB, the page table (only the first 32 pages), and the cache are as follows.

TLB			
Index	Tag	PPN	Valid
0	09	4	1
	12	2	1
	10	0	1
	08	5	1
	05	7	1
	13	1	0
	10	3	0
	18	3	0
1	04	1	0
	0C	1	0
	12	0	0
	08	1	0
	06	7	0
	03	1	0
	07	5	0
	02	2	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	6	1	10	0	1
01	5	0	11	5	0
02	3	1	12	2	1
03	4	1	13	4	0
04	2	0	14	6	0
05	7	1	15	2	0
06	1	0	16	4	0
07	3	0	17	6	0
08	5	1	18	1	1
09	4	0	19	2	0
0A	3	0	1A	5	0
0B	2	0	1B	7	0
0C	5	0	1C	6	0
0D	6	0	1D	2	0
0E	1	1	1E	3	0
0F	0	0	1F	1	0

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	19	1	99	11	23	11	00	0	99	11	23	11
1	15	0	4F	22	EC	11	2F	1	55	59	0B	41
2	1B	1	00	02	04	08	0B	1	01	03	05	07
3	06	0	84	06	B2	9C	12	0	84	06	B2	9C
4	07	0	43	6D	8F	09	05	0	43	6D	8F	09
5	0D	1	36	32	00	78	1E	1	A1	B2	C4	DE
6	11	0	A2	37	68	31	00	1	BB	77	33	00
7	16	1	11	C2	11	33	1E	1	00	C0	0F	00

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

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag

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

(b) 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.

PPO The physical page offset
PPN The physical page number
CO The block offset within the cache line
CI The cache index
CT The cache tag

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

(c) Give the format of the virtual address.

Virtual address: `31DE`

Virtual address format (one bit per box)

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

(d) For the given virtual address, fill in the following table. If there is a page fault, enter ”–” for ”PPN”.

Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

(e) If there is a page fault, leave this part blank. Otherwise, indicate the format of the physical address.

Physical address format (one bit per box)

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

(f) If there is a page fault, leave this part blank. Otherwise, fill in the following table. If there is a cache miss, enter ”–” for ”Cache Byte returned”.

Physical memory reference

Parameter	Value
Byte offset	0x
Cache Index	0x
Cache Tag	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

/ 14 points

2. Consider an allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows.

```
                31                            2 1 0
                 ----------------------------------
      Header    |    Block Size (bytes)      |     |
                |____________________________|_____|
                |                                  |
                |                                  |
                |                                  |
                |                                  |
                |                                  |
                |__________________________________|
      Footer    |    Block Size (bytes)      |     |
                |____________________________|_____|
```

Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows.

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` indicates the use of the next adjacent block: 1 for allocated, 0 for free.

Given the contents of the heap shown on the left side *of the next page*, show the new contents of the heap (in the table provided on the right side *of the next page*) after a call to `free(0x400b010)` is executed. Your answers should be given as hex values (feel free to omit leading zeros). Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed. Also assume that any blocks not shown are allocated.

Address		Address	
0x400b030	0x00000017	0x400b030	
0x400b02c	0x00000002	0x400b02c	
0x400b028	0x00000001	0x400b028	
0x400b024	0x00000017	0x400b024	
0x400b020	0x0000001d	0x400b020	
0x400b01c	0xfffffffc	0x400b01c	
0x400b018	0xfffffffd	0x400b018	
0x400b014	0xfffffffe	0x400b014	
0x400b010	0xffffffff	0x400b010	
0x400b00c	0x0000001d	0x400b00c	
0x400b008	0x00000016	0x400b008	
0x400b004	0x200b601c	0x400b004	
0x400b000	0x800b511c	0x400b000	
0x400affc	0x00000016	0x400affc	

/ 12 points

3. Consider an allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes. The usage of the remaining 3 lower order bits is as follows:

   - `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
   - `bit 1` indicates the use of the previous block: 1 for allocated, 0 for free.
   - `bit 2` indicates the use of the next block: 1 for allocated, 0 for free.

   Three helper routines are defined to facilitate the implementation of `free(void* p)`. The functionality of each routine is explained in the comment above the function definition. Circle *Yes* or *No* to indicate whether the body of the function correctly implements the functionality described. If you circle *No*, rewrite the code such that it is correct.

   Note that an `int` *requires four bytes.*

```
/* Given a pointer to a valid block header or footer, returns the size of the
   block (number of bytes). */
int size(void* hp) {
  return (*hp) & (~0x7);    /* Correct? */
}
```

   Circle one: Yes or No

   Rewritten line of code: ______________________________

```
/* Given a pointer p to an allocated block, i.e,. p is a pointer returned by
   a previous malloc/realloc call; returns a pointer to the block's footer. */
void* footer(void* p) {
  return (char*)p + size((char*)p - 4) - 8;    /* Correct? */
}
```

   Circle one: Yes or No

   Rewritten line of code: ______________________________

```
/* Given a pointer to a valid block header or footer, returns the usage of
   the previous block, 1 for allocated, 0 for free. */
int prev_allocated(void* hp) {
  return (*(int*)hp) & 0x4;    /* Correct? */
}
```

   Circle one: Yes or No

   Rewritten line of code: ______________________________

/ 15 points

4. Using the space provided, answer each of the following questions about threads. *Try to be concise*—a short, correct answer will receive more points than a long, rambling answer that contains the correct information hidden within it.

   (a) A context switch between two threads in the same process is faster than a context switch between two processes.

      i. Explain why.

      ii. Assume that you are running a badly programmed version of UNIX in which context switches between threads are actually slower than context switches between processes. Describe a situation in which you, as a programmer, would still prefer to use multiple threads as opposed to multiple processes.

   (b) A global variable is "shared" if it is accessed by more than one thread during the execution of a program. Recall that most of the time, a thread should only access a shared global variable while holding the lock that protects that variable.

      i. Describe a situation in which a correct program would not need to hold any lock while accessing a shared global variable.

ii. Explain why it is critical that each of the P and V operations on a semaphore occur indivisibly.

(c) Consider two multi-threaded web servers. Server A creates a new thread for each incoming connection. Server B, on the other hand, pre-creates a fixed pool of threads (eight, for example) at startup time and uses these to handle all connections. First, describe a workload that will cause A to perform better than B. Second, describe a workload that will cause B to perform better than A. In each case, explain why there is a difference in performance. We say that a web server performs better than another if it handles more requests per second.

/ 9 points

5. Consider the following three programs, which attempt to use three semaphores, `a`, `b`, and `c`, for mutual exclusion.

Program 1 Initially, `a` is 1, `b` is 1, and `c` is 1.

Thread 1	*Thread 2*
P(a)	P(c)
P(b)	P(a)
P(c)	V(c)
V(a)	P(b)
V(b)	V(a)
V(c)	V(b)

Circle *one* of the following outcomes. When **Program 1** executes, deadlock:

always occurs   might occur   never occurs

Program 2 Initially, `a` is 1, `b` is 1, and `c` is 1.

Thread 1	*Thread 2*
P(a)	P(a)
P(b)	V(a)
P(c)	P(c)
V(a)	P(b)
V(b)	V(c)
V(c)	V(b)

Circle *one* of the following outcomes. When **Program 2** executes, deadlock:

always occurs   might occur   never occurs

Program 3 Initially, `a` is 1, `b` is 1, and `c` is 1.

Thread 1	*Thread 2*
P(a)	P(b)
P(c)	P(a)
V(c)	P(c)
V(a)	V(a)
P(b)	V(c)
V(b)	V(b)

Circle *one* of the following outcomes. When **Program 3** executes, deadlock:

always occurs   might occur   never occurs

/ 6 points

6. Match each IA32 assembly routine on the left with the equivalent C function on the right.

```
foo1:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  sall $4,%eax
  subl 8(%ebp),%eax
  movl %ebp,%esp
  popl %ebp
  ret

foo2:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  testl %eax,%eax
  jge .L4
  addl $15,%eax
.L4:
  sarl $4,%eax
  movl %ebp,%esp
  popl %ebp
  ret

foo3:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%ecx
  xorl %eax,%eax
  testl %ecx,%ecx
  jge .L6
  incl %eax
.L6
  movl %ebp,%esp
  popl %ebp
  ret
```

```
int choice1(int x) {
  return (x < 0);
}

int choice2(int x) {
  return (x << 31) & 1;
}

int choice3(int x) {
  return 15 * x;
}

int choice4(int x) {
  return (x + 15) / 4;
}

int choice5(int x) {
  return x / 16;
}

int choice6(int x) {
  return (x >> 31);
}
```

Assembler routine `foo1` corresponds to C function ____________.

Assembler routine `foo2` corresponds to C function ____________.

Assembler routine `foo3` corresponds to C function ____________.

/ 6 points

7. Consider the following code fragment containing the incomplete definition of a data type `struct matrix_entry` with four fields.

```
struct matrix_entry{

  _________ a;

  _________ b;

  short  c;

  _________ d;

};
```

Also consider the following C code and assembly code.

```
struct matrix_entry matrix[2][5];

short return_entry(int i, int j) {
  return matrix[i][j].c;
}
```

```
return_entry:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  leal (%eax,%eax,4),%eax
  addl 12(%ebp),%eax
  sall $4,%eax
  movl matrix+6(%eax),%eax
  movl %ebp,%esp
  popl %ebp
  ret
```

Complete the definition of `struct matrix_entry` so that the assembly code could be generated for function `return_entry` on a Linux/x86 machine.

- Note that there are multiple correct answers.
- Choose your answers from the following types, assudming these sizes and alignments.

Type	Size (bytes)	Alignment (bytes)
char	1	1
short	2	2
int	4	4
double	8	4

/ 12 points

8. This problem concerns the run-time stack for the following C functions.

```
/* copy string x to buf */
void foo(char *x) {
  int buf[1];
  strcpy((char *)buf, x);
}

void callfoo() {
  foo("abcdefghi");
}
```

Functions `foo` and `callfoo` have the following disassembled form on an IA32 machine.

```
080484f4 <foo>:
080484f4: 55              pushl  %ebp
080484f5: 89 e5           movl   %esp,%ebp
080484f7: 83 ec 18        subl   $0x18,%esp
080484fa: 8b 45 08        movl   0x8(%ebp),%eax
080484fd: 83 c4 f8        addl   $0xfffffff8,%esp
08048500: 50              pushl  %eax    # push x
08048501: 8d 45 fc        leal   0xfffffffc(%ebp),%eax
08048504: 50              pushl  %eax    # push buf
08048505: e8 ba fe ff ff  call   80483c4 <strcpy>
0804850a: 89 ec           movl   %ebp,%esp
0804850c: 5d              popl   %ebp
0804850d: c3              ret

08048510 <callfoo>:
08048510: 55              pushl  %ebp
08048511: 89 e5           movl   %esp,%ebp
08048513: 83 ec 08        subl   $0x8,%esp
08048516: 83 c4 f4        addl   $0xfffffff4,%esp
08048519: 68 9c 85 04 08  pushl  $0x804859c    # push string address
0804851e: e8 d1 ff ff ff  call   80484f4 <foo>
08048523: 89 ec           movl   %ebp,%esp
08048525: 5d              popl   %ebp
08048526: c3              ret
```

Note the following:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '\0' character) to address `dst`. It does *not* check the size of the destination buffer.
- IA32 machines are little endian.
- C strings are null-terminated (i.e., terminated by a character with value `0x00`).
- Characters 'a' through 'i' have ASCII codes `0x61` through `0x69`.

Consider what happens on an IA32 machine when `callfoo` calls `foo` with the input string "`abcdefghi`".

(a) List the contents of the following memory locations immediately after `strcpy` returns to `foo`. *Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.* (Note that `buf[x]` is simply the contents in memory at address `buf + 4x`.)

`buf[0]` = `0x` ____________________

`buf[1]` = `0x` ____________________

`buf[2]` = `0x` ____________________

(b) Immediately *before* the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

`%ebp` = `0x` ____________________

(c) Immediately *after* the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

`%eip` = `0x` ____________________

/ 10 points

9. Consider the following C program. Assume that all functions return normally and that the proper header files have been included.

```
int main() {
   int status;
   printf("start\n");
   printf("%d\n", !fork());
   if(wait(&status) != -1)
      printf("%d\n", WEXITSTATUS(status));
   printf("end\n");
   exit(2);
}
```

Recall the following:

- Function `wait` returns `-1` when there is an error, e.g., when there is no child.
- Macro `WEXITSTATUS` extracts the exit status of the terminating process.

(a) Draw a diagram that illustrates the processes at run-time.

(b) Give *three* possible outputs of this program.