15-213/18-213, Spring 2012

Final Exam

Friday, May 11, 2012

Instructions:

- Make sure that your exam is not missing any sheets, then write your Andrew ID and full name on the front.

- This exam is closed book, closed notes (except for provided note sheets). You may not use any electronic devices.

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.

- The problems are of varying difficulty. The point value of each problem is indicated. Good luck!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(08):</td>
</tr>
<tr>
<td>2</td>
<td>(08):</td>
</tr>
<tr>
<td>3</td>
<td>(10):</td>
</tr>
<tr>
<td>4</td>
<td>(09):</td>
</tr>
<tr>
<td>5</td>
<td>(08):</td>
</tr>
<tr>
<td>6</td>
<td>(10):</td>
</tr>
<tr>
<td>7</td>
<td>(10):</td>
</tr>
<tr>
<td>8</td>
<td>(15):</td>
</tr>
<tr>
<td>9</td>
<td>(12):</td>
</tr>
<tr>
<td>10</td>
<td>(10):</td>
</tr>
</tbody>
</table>

TOTAL (100):
Problem 1. (8 points):

Ints and Floats

A. Consider a signed integer representation similar to two’s complement. The only difference is that the most significant bit has a positive value, and all other bits contribute a negative value. The absolute value each bit contributes is the same as it would be in the original format. Is this new format the same as the original two’s complement format in terms of which numbers it can represent?

(a) Yes
(b) No

B. Let $n \geq 1$ be an arbitrary integer. Consider a two’s complement representation using $n$ bits. Assume at least 1 of the $n$ bits is set to 1. Given only this constraint, the largest integer we can represent for any given $n$ is greater than or equal to 0.

(a) True
(b) False

You’ve been asked to design the floating-point unit for Harry Q. Bovik’s new microprocessor. Harry is sure that he wants to use the IEEE standard for floating-point numbers, but he isn’t sure of some of the other design parameters.

Harry is concerned about precision in his system. He would like to be able to represent positive integers up to 64 without having to round.

A. What is the least number of fraction bits and the least number of exponent bits to make this possible? (Note that the total number of bits may change)

Number of fraction bits: _____

Number of exponent bits: _____

Consider extending our floating point format by one bit.

A. Thinking about the effect on the range (defined as the difference between the smallest and largest representable finite number). Which of the following is true?

- The range will be increased by:
  (a) increasing the width of the fraction by one more bit
  (b) increasing the width of the exponent by one more bit
  (c) both (a) and (b)
  (d) neither (a) nor (b)

B. Which of the following is true about rounding error?

- The rounding error for all numbers remains unchanged or is reduced by
  (a) increasing the width of the fraction by one more bit
  (b) increasing the width of the exponent by one more bit
  (c) both (a) and (b)
  (d) neither (a) nor (b)
Problem 2. (8 points):

Struct Layout

Consider the data structure declarations below. (Note that this is a single declaration which includes several data structures; they are shown horizontally to fit on the page.)

```c
struct s1 {
    struct s2 a;
    struct s2 *b;
    struct s1 *c;
    double d;
    int e[4];
};
struct s2 {
    char i[3];
    struct s3 *j;
    int k;
};
struct s3 {
    int f;
    struct s1 g;
    struct s2 *h;
};
```

For each of these four C procedures, fill in the missing offsets in the corresponding Linux IA-32 assembly code immediately below it. If there is no offset, you must write a 0 in the blank. (If you give the wrong answer below but write the correct sizes next to the structures above, you might get some partial credit.)

A. int proc1(struct s1 *x) {
    return x->e[1];
}

```assembly
proc1:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl _____(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

B. int proc2(struct s2 *x) {
    return x->j->g.e[3];
}

```assembly
proc2:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 8(%ebp),%eax
    movl _____(%eax),%eax
    movl _____(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

C. int proc3(struct s3 *x) {
    return x->h->k;
}

```assembly
proc3:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl _____(%eax),%eax
    movl _____(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

D. int proc4(struct s3 *x) {
    return x->g.c->b->k;
}

```assembly
proc4:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl _____(%eax),%eax
    movl _____(%eax),%eax
    movl _____(%eax),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
Problem 3. (10 points):

Stack discipline.

A. Multiple choice.

1. Consider a hypothetical modification to x86 stack discipline: instead of storing the return address to the calling function on the stack, it is placed into a special return-address register. Which of the following would be true of x86 stack discipline after the modification?

(a) It supports nested function calls.
(b) It supports recursion.
(c) Both (a) and (b).
(d) Neither (a) nor (b).

2. Which of the following could not be considered inverse operations in terms of stack manipulation?

(a) call and leave.
(b) call and ret.
(c) push and pop.
(d) All of the above.

3. Which of the following is true about the differences between x86 and x86-64?

(a) Because the pointer size is larger in x86-64, the size of a function’s stack frame in a 64-bit program is always larger than that in a 32-bit program.
(b) Because there is typically no argument build area in x86-64 stack frames, the size of a function’s stack frame in a 64-bit program is always smaller than that in a 32-bit program.
(c) Because x86-64 must be backward compatible with x86, the size of a function’s stack frame in a 64-bit program is always the same as that in a 32-bit program.
(d) None of the above.

4. While debugging a program with GDB, you notice that the value of %eip is very close to the value of %esp. Which of the following is the most likely explanation?

(a) Execution was just interrupted by the delivery of a signal.
(b) The program is the victim of a buffer overflow exploit.
(c) The program just completed a call to fork, which returned -1.
(d) There is no “explanation.” It is perfectly normal for %eip and %esp to be similar sometimes.
B. Assume that a program under execution ends up in the following state with the given stack diagram and associated register values:

```
+------------+
0xffff101c | 0x00 |
+------------+
0xffff1018 | 0x00 |
+------------+
0xffff1014 | 0x08048301 |
+------------+
0xffff1010 | 0xffff1040 |
+------------+
0xffff100c | 0xdead |
+------------+
0xffff1008 | 0xbeef |
+------------+
0xffff1004 | 0x00 |
+------------+
0xffff1000 | 0x00 |
+------------+

%esp: 0xffff1008
%ebp: 0xffff1010
%eip: 0x080483c2
```

The following are three more sets of stack diagrams and associated register values. Each state represents the result of executing exactly one additional instruction from the above state. For each of the new states, identify the mystery instruction. If the instruction requires operands, supply them. For your convenience, fields that differ between the original state and the new state are labeled with an asterisk: (*)

<table>
<thead>
<tr>
<th>Original State</th>
<th>New State 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff101c</td>
<td>0x00</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1018</td>
<td>0x00</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1014</td>
<td>0x08048301</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1010</td>
<td>0xffff1040</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff100c</td>
<td>0xdead</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1008</td>
<td>0xbeef</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1004</td>
<td>0x00</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
<tr>
<td>0xffff1000</td>
<td>0x00</td>
</tr>
<tr>
<td>+------------+</td>
<td>+------------+</td>
</tr>
</tbody>
</table>

%esp: 0xffff1008 %esp: 0xffff1014 (*)
%ebp: 0xffff1010 %ebp: 0xffff1040 (*)
%eip: 0x080483c2 %eip: 0x080483c3 (*)

MYSTERY INSTRUCTION: ________________
## Original State vs. New State 2

<table>
<thead>
<tr>
<th>Address (Original)</th>
<th>Value (Original)</th>
<th>Address (New)</th>
<th>Value (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff101c</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1018</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1014</td>
<td>0x8048301</td>
<td></td>
<td>0x8048301</td>
</tr>
<tr>
<td>0xffff1010</td>
<td>0xffff1040</td>
<td></td>
<td>0xffff1040</td>
</tr>
<tr>
<td>0xffff100c</td>
<td>0xdead</td>
<td></td>
<td>0xdead</td>
</tr>
<tr>
<td>0xffff1008</td>
<td>0xbeef</td>
<td></td>
<td>0xbeef</td>
</tr>
<tr>
<td>0xffff1004</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1000</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
</tbody>
</table>

%esp: 0xffff1008 %esp: 0xffff1004
%ebp: 0xffff1010 %ebp: 0xffff1010
%eip: 0x80483c2 %eip: 0x8048c94

**MYSTERY INSTRUCTION:** ________________

## Original State vs. New State 3

<table>
<thead>
<tr>
<th>Address (Original)</th>
<th>Value (Original)</th>
<th>Address (New)</th>
<th>Value (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff101c</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1018</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1014</td>
<td>0x8048301</td>
<td></td>
<td>0x8048301</td>
</tr>
<tr>
<td>0xffff1010</td>
<td>0xffff1040</td>
<td></td>
<td>0xffff1040</td>
</tr>
<tr>
<td>0xffff100c</td>
<td>0xdead</td>
<td></td>
<td>0xdead</td>
</tr>
<tr>
<td>0xffff1008</td>
<td>0xbeef</td>
<td></td>
<td>0xbeef</td>
</tr>
<tr>
<td>0xffff1004</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>0xffff1000</td>
<td>0x00</td>
<td></td>
<td>0x00</td>
</tr>
</tbody>
</table>

%esp: 0xffff1008 %esp: 0xffff1004
%ebp: 0xffff1010 %ebp: 0xffff1010
%eip: 0x80483c2 %eip: 0x8048c94

**MYSTERY INSTRUCTION:** ________________
Problem 4. (9 points):
Unix I/O.

Suppose that the disk file hello.txt consists of the five ASCII characters “hello”, the disk file foobar.txt consists of the six ASCII characters “foobar”, and the disk file world.txt consists of the five ASCII characters “world”. Answer the following questions.

A. What is the output of the following program?

```c
/* any necessary includes */
char buf[20] = {0}; /* init to all zeroes */

int main(int argc, char* argv[]) {
    int fd1 = open("hello.txt", O_RDWR);
    int fd2 = open("foobar.txt", O_RDWR);
    int fd3 = open("world.txt", O_RDWR);

    dup2(fd3, fd2);
    dup2(fd1, fd3);

    read(fd1, buf, 2);
    read(fd2, &buf[2], 2);
    read(fd3, &buf[4], 2);

    close(fd1);
    close(fd2);
    close(fd3);

    printf("buf = %s\n", buf);
    return 0;
}
```

Output: buf = _______________
B. What is the output of the following program?

```c
/* any necessary includes */
char buf[20] = {0}; /* init to all zeroes */

int main(int argc, char* argv[]) {
    int fd1 = open("hello.txt", O_RDWR);
    int fd2 = open("foobar.txt", O_RDWR);
    int fd3 = open("hello.txt", O_RDWR);

    write(fd1, "213", 3);
    read(fd2, buf, 2);
    read(fd3, &buf[2], 2);

    close(fd1);
    close(fd2);
    close(fd3);

    printf("buf = %s\n", buf);
    return 0;
}
```

Output: buf = _______________
C. Before taking 213, a student wrote the following code to generate two files, `even.txt` and `odd.txt`, from `foobar.txt`. The program reads `foobar.txt` and writes the even indexed characters to `even.txt` and the odd number indexed character into `odd.txt`. Note that the index of the first character is 0, which is even. In other words, the contents of `even.txt` should be “foa”, and the contents of `odd.txt` should be “oob”. However, the following code doesn’t produce the expected outputs. Identify the problem and suggest a fix. (No more than 3 sentences.)

```c
/* any necessary includes */
int main(int argc, char *argv[])
{
    char c;
    int idx = 0;
    int fd0 = open("foobar.txt", O_RDONLY);
    int fd1 = open("even.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    int fd2 = open("odd.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    if(!fork()) {
        while(read(fd0, &c, 1) > 0) {
            if (idx % 2 == 0) {
                write(fd1, &c, 1);
            }
            idx++;
        }
    } else {
        while(read(fd0, &c, 1) > 0) {
            if (idx % 2 == 1) {
                write(fd2, &c, 1);
            }
            idx++;
        }
    }

    close(fd0);
    close(fd1);
    close(fd2);
    return 0;
}
```
Problem 5. (8 points):
Caching and Threads
Suppose we have an array of $N$ ints. Assume $N$ is even. Suppose we have a single-core machine with one simple cache that only has one cache line, which holds 8 bytes of data.

Part A
We want to compute the sum of the elements in the array by iterating through the array in order of increasing index. Assume the partial sum is stored in a register. How many cache misses does this procedure incur? Give your answer in terms of $N$.

Part B
We want to sum up the elements in the array using 2 threads. The first thread sums up the elements at even indices and the second thread sums up the elements at the odd indices. Assume that the partial sums are stored in registers. In the worst case, what is the number of cache misses incurred?

Part C
Again, we want to sum up the elements in the array using 2 threads. Now, the first thread sums up the elements between indices 0 and $(N/2) - 1$ and the second thread sums up the elements between indices $(N/2)$ and $N - 1$. Assume that the partial sums are stored in registers. In the worst case, what is the number of cache misses incurred?

Part D
Suppose we change the architecture by increasing the number of cores to 2, where each core has a private cache with one 8-byte cache line and there is a global shared fully-associative cache with two 8-byte cache lines. This will likely improve the performance of the procedures from:

1. Part (B) but not Part (C)
2. Part (C) but not Part (B)
3. both Parts (B) and (C)
4. neither Parts (B) nor (C)
Problem 6. (10 points):

Malloc

Consider a malloc implementation that uses a doubly linked explicit free list with 8 byte headers and 8 byte footer, similar to the one described in lecture.

- Malloc fulfills memory requests via a first-fit approach, starting from the head of the free list.
- If a fitting block is found in the free list, and the fit is larger than the request, malloc does not split off the extra space.
- If the request cannot be satisfied by any blocks in the free list, or the free list is empty, the heap will be extended to accommodate the request.
- The heap is never extended by more than what is needed to fulfill the request to malloc. Each allocated block must be 8-byte aligned in memory.
- When free-ing, the new block is inserted at the head of the free list.
- Calls to free will only coalesce the lower adjacent free block.
- Any coalesced blocks are removed from the free list before they are coalesced, and the entire new free block is inserted at the beginning of the free list.

A) Assume the following trace:

```c
a = malloc(1);
b = malloc(2);
c = malloc(3);
d = malloc(4);
e = malloc(5);
```

1. How large is the heap (excluding any prologue and epilogue)?

2. How much internal and external fragmentation exists once the trace has completed?
B) Now assume the following new trace:

```c
a = malloc(1);
b = malloc(2);
c = malloc(3);
d = malloc(4);
e = malloc(5);
free(e);
free(d);
free(c);
free(b);
free(a);
f = calloc(1,2);
g = calloc(2,2);
h = calloc(3,2);
```

1. How large is the heap (excluding any prologue and epilogue)?

2. How much internal and external fragmentation exists once the trace has completed assuming no future requests?
Problem 7. (10 points):
Synchronization.

A. Multiple choice.

1. Consider the following pseudocode:

```c
int foo() {
    static sem_t mutex;
    static int count;
    P(&mutex); // wait
    int bar = count++;
    V(&mutex); // signal
    return bar;
}
```

This function is:
(a) Thread-safe and reentrant.
(b) Thread-safe but not reentrant.
(c) Reentrant but not thread-safe.
(d) Neither thread-safe nor reentrant.

2. Harry Q. Bovik decides to remove all synchronization primitives from a multithreaded program. Which of the following outcomes is not a possible consequence of this modification?
(a) Nondeterministic behavior
(b) Incorrect behavior
(c) Faster running time
(d) Deadlock
B. This problem is about using semaphores to synchronize deletions from a singly-linked list shared between an arbitrary number of threads. List nodes are represented by the following struct:

```c
struct list_node_t {
    int value;
    // Pointer to the next node in the list.
    struct list_node_t * next;
    // Semaphore for synchronizing access to this node.
    // Assume it is initialized to 1.
    sem_t mutex;
};
typedef struct list_node_t list_node;
```

Further assume:

- The linked list is initialized with an arbitrary number of elements.
- For simplicity, use $P(node)$ and $V(node)$ to synchronize on node->mutex. Recall that $P$ is equivalent to wait and $V$ is equivalent to signal.
- head->next is a pointer to the first element of the linked list, where head is a constant, globally-defined list element. (This allows you to lock the pointer to the beginning of the list using $P(head)$)
- Elements are removed by calling the list_remove(int value) function, which finds and deletes the first element containing value.

Your task is to use semaphore operations to correctly synchronize list_remove. Add the appropriate $P$ and $V$ operations to the function psuedo-code on the following page.

Hints:

- Only add $P$ and $V$ operations in the blank spaces. Each blank may require zero, one, or two operations.
- Use only the semaphores in the list nodes. Do not introduce any additional semaphores.
- More points will be awarded to solutions supporting more concurrent operations.
void list_remove(int value) {
    node* prev_node = head;

    node* cur_node = prev_node->next;
    while (cur_node != NULL) {

        if (cur_node->value == value) {
            // Remove cur_node from list.
            prev_node->next = cur_node->next;

            // Assume cur_node is correctly freed here, etc.
            return;
        }
    }
    // Advance indexes.

    prev_node = cur_node;
    cur_node = cur_node->next;

}  // value was not found in the list.
Problem 8. (15 points):
The following problems concerns the components of the memory hierarchy and the way virtual addresses are translated into physical addresses.

- The memory is byte addressable, and memory accesses are to **1-byte** (not 4-byte) words.
- Virtual addresses are 16 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 256 bytes.
- The TLB has 8-way with 2 sets.
- The cache is 3-way set-associative, with a 4-byte line size and 24 total lines.

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

<table>
<thead>
<tr>
<th>TLB</th>
<th>Index</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27</td>
<td>D</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0B</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4F</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1F</td>
<td>C</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>49</td>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>52</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td>B</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1F</td>
<td>B</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0C</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>08</td>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>42</td>
<td>B</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>48</td>
<td>A</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>66</td>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6C</td>
<td>C</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td>10</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td></td>
<td>11</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>9</td>
<td>0</td>
<td></td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>03</td>
<td>F</td>
<td>1</td>
<td></td>
<td>13</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>7</td>
<td>0</td>
<td></td>
<td>14</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>8</td>
<td>0</td>
<td></td>
<td>15</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>0</td>
<td>1</td>
<td></td>
<td>16</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>4</td>
<td>1</td>
<td></td>
<td>17</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>08</td>
<td>8</td>
<td>1</td>
<td></td>
<td>18</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>0</td>
<td>0</td>
<td></td>
<td>19</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>4</td>
<td>1</td>
<td></td>
<td>1A</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>A</td>
<td>0</td>
<td></td>
<td>1B</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>0C</td>
<td>B</td>
<td>0</td>
<td></td>
<td>1C</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0D</td>
<td>3</td>
<td>1</td>
<td></td>
<td>1D</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>D</td>
<td>0</td>
<td></td>
<td>1E</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0F</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1F</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Index</td>
<td>Tag</td>
<td>Valid</td>
<td>Line</td>
<td>Tag</td>
<td>Valid</td>
<td>Line</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>1</td>
<td>7E813E16</td>
<td>5D</td>
<td>0</td>
<td>6781211C</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
<td>1</td>
<td>F2E25610</td>
<td>4C</td>
<td>0</td>
<td>62C2E1A0</td>
</tr>
<tr>
<td>2</td>
<td>1C</td>
<td>1</td>
<td>21DB6EE7</td>
<td>74</td>
<td>1</td>
<td>D08CEAFF</td>
</tr>
<tr>
<td>3</td>
<td>7F</td>
<td>1</td>
<td>53C9FA45</td>
<td>44</td>
<td>0</td>
<td>D36D3A19</td>
</tr>
<tr>
<td>4</td>
<td>3C</td>
<td>1</td>
<td>8F6B663A</td>
<td>4B</td>
<td>0</td>
<td>95EB6C01</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>0</td>
<td>9770C6EB</td>
<td>05</td>
<td>0</td>
<td>41FD3F5D</td>
</tr>
<tr>
<td>6</td>
<td>3F</td>
<td>1</td>
<td>C4C13D4F</td>
<td>5F</td>
<td>1</td>
<td>FC79DCE1</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>1</td>
<td>DE8C2265</td>
<td>78</td>
<td>1</td>
<td>D99937C5</td>
</tr>
</tbody>
</table>
Part 1

A The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn’t exist, don’t draw it on the diagram.)

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

![Virtual Address Diagram](image)

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

![Physical Address Diagram](image)
**Part 2**

For the given virtual address, indicate the TLB entry accessed, the physical address, and the cache byte value returned in **hexadecimal**. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs.

If there is a cache miss, enter “-” for “Cache Byte returned”. If there is a page fault, enter “-” for “PPN” and leave parts C and D blank.

**Virtual address:** 0x17FC

<table>
<thead>
<tr>
<th>A</th>
<th>Virtual address format (fill in each box with either a 0 or a 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>Address translation (numeric answers should be given in hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>VPN</td>
<td>0x</td>
</tr>
<tr>
<td>TLB Tag</td>
<td>0x</td>
</tr>
<tr>
<td>TLB Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Page Fault? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>PPN</td>
<td>0x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>Physical address format (fill in each box with either a 0 or a 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>Physical memory reference (numeric answers should be given in hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>Block offset</td>
<td>0x</td>
</tr>
<tr>
<td>Cache Index</td>
<td>0x</td>
</tr>
<tr>
<td>Cache Tag</td>
<td>0x</td>
</tr>
<tr>
<td>Cache Hit? (Y/N)</td>
<td></td>
</tr>
<tr>
<td>Value of Cache Byte Returned</td>
<td>0x</td>
</tr>
</tbody>
</table>
Problem 9. (12 points):

Ill communication.

A couple of vandals named Ranthon Owe and Modd Towry have sabotaged the arithmetic units on all of the shark machines! Now they are unable to perform unsigned addition. One workaround is to write a pair of simple programs: an addition server and client. The addition server would run on a machine that does support unsigned addition, and the client would run on a shark machine and contact the server with requests for solutions to addition problems.

For simplicity’s sake, the system will support only the addition of exactly two uint32_t.s. The communication protocol is as follows:

- To add two unsigned integers, the client first connects to the server.
- The client then sends a packet of exactly eight bytes. The first four bytes is one of the uint32_t in network byte order, and the last four bytes is the other uint32_t in network byte order.
- The server reads the packet from the client and tokenizes it into the two original uint32_t.s. The unsigned integers are added together, and the server sends a single four-byte packet, which consists of the uint32_t which is the result of the addition in network byte order.
- Since each connection is good for exactly one addition, the connection is closed after a single addition operation.

The following is an overview illustration:

```
(4 bytes) (4 bytes)
+-----------------------------------+
| uint32_t a | uint32_t b |
+-----------------------------------+
(4 bytes)
+----------------+
| uint32_t a + b |
+----------------+
```

Hints:

- Recall that uint32_t is a four-byte data type.
- Be careful about byte order.
- Assume that all system calls always succeed, but be wary of short reads and writes.
- The following POSIX sockets functions may be necessary:

  ```
  int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
  int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  ssize_t read(int fd, void *buf, size_t count);
  ssize_t write(int fd, const void *buf, size_t count);
  ```
A. Complete the implementation of the addition client by writing the `add` function. Assume that the socket address `addr` has already been set up with the full address of the addition server and that the socket file descriptor `fd` has already been set up for a connection. The `add` function should connect to the server, make an addition request, read the response, and return the result of the addition.

Examine the empty `add` function and the following set of code snippets, each labeled with a letter. Instead of writing actual code in the body of `add`, list a series of labels corresponding to the code snippets in the exact order in which they should be executed. Give the simplest possible correct implementation.

```c
struct sockaddr addr;
int fd;

uint32_t add(uint32_t a, uint32_t b)
{
    uint32_t sum;

    // Write code snippet labels here:

}  
```

A fd = accept(INADDR_ANY, &addr, &len);
B char buf[8];
   memcpy(buf, &a, 4);
   memcpy(buf + 4, &b, 4);
C connect(fd, (struct sockaddr *)"shark.ics.cs.cmu.edu", 20);
D connect(fd, (struct sockaddr *)&addr, sizeof(addr));
E read(fd, (char *)&sum, 4);
F return ntohl(sum);
G return sum;
H ssize_t nr = 0;
   while(nr < 4)
       nr += read(fd, (char *)&sum + nr, 4 - nr);
I ssize_t nw = 0;
   while(nw < 8)
       nw += write(fd, (char *)buf + nw, 8 - nw);
J uint32_t buf[2];
   buf[0] = htonl(a);
   buf[1] = htonl(b);
K write(fd, (char *)buf, 8);
```
B. Complete the implementation of the addition server by writing the `loop` function. Assume that the socket file descriptor `listen_fd` has already been set up to listen for incoming connections. The `loop` function should continuously listen for incoming connections; once a connection with a client is established, it should read the request, perform the addition, send the response, then close the connections.

Examine the empty `loop` function and the following set of code snippets, each labeled with a letter. Instead of writing actual code in the body of `loop`, list a series of labels corresponding to the code snippets in the exact order in which they should be executed. Give the simplest possible correct implementation.

```c
int listen_fd;

void loop()
{
    struct sockaddr addr;
    socklen_t len;

    while(1)
    {
        int fd;
        uint32_t a, b, sum;

        // Write code snippet labels here:


    }
}
```

A. `a = ntohl(buf[0]);`  
B. `b = ntohl(buf[1]);`  
C. `char buf[8];`  
D. `read(fd, (char *)buf, 8);`  
E. `close(fd);`  
F. `connect(fd, (struct sockaddr *)"shark.ics.cs.cmu.edu", 20);`  
G. `fd = accept(listen_fd, &addr, &len);`  
H. `memcpy(&a, buf, 4);`  
I. `memcpy(&b, buf + 4, 4);`  
J. `uint32_t buf[2];`  
K. `ssize_t nw = 0;`  
L. `while(nw < 4)
        nw += write(fd, (char *)&sum + nw, 4 - nw);`  
M. `sum = htonl(a + b);`  
N. `sum = htonl(ntohl(a) + ntohl(b));`  
O. `ssize_t nr = 0;`  
P. `while(n < 8)
        nr += read(fd, (char *)buf + nr, 8 - nr);`  
Q. `write(fd, (char *)&sum, 4);`
Problem 10. (10 points):

Process Control
Consider the following C programs:

// fork.c

main()
{
    int i = 0;
    int pid = 0;
    int status = 0;
    int spoon = 0;

    for(i = 0; i < 5; i++)
    {
        spoon = i;
        if(!fork()) while(1);
        printf("%d,", spoon);
    }

    while(spoon > 2)
    {
        pid = fork();
        if(pid) spoon--;
        else break;
    }

    if(spoon == 4) {
        execl("./knife", NULL);
        printf("%d,", spoon);
    }

    if(pid) {
        waitpid(-1, &status, 0);
        printf("%d", 10*WEXITSTATUS(status));
    } else printf("%d", spoon);

    exit(spoon);
}

//knife.c

main()
{
    printf("knife, ");
    exit(0);
}
Assume the following:

- All processes run to completion, and no system calls fail.
- `printf` is atomic and calls `fflush(stdout)` after printing its argument(s) but before returning.

The file `knife.c` is compiled to `knife`, and `fork.c` is compiled to `fork`, and the two programs reside in the same directory. List 10 possible outputs when `fork` is run. If 10 outputs do not exist, leave the rest of the spaces blank. List the possible outputs here

________  ________
________  ________
________  ________
________  ________
________  ________