Computer Science 61

Midterm Exam Solutions

October 22, 2015

82 minutes/82 points

Instructions

The points allocated to each problem correspond to how much time we think the problem should take.

You will submit the exam via your code.seas repository and the grading server. If you do not have a current pset repository, you will need to pull the cs61-psets repository so you have the midterm directory. As with the problem sets, you should do:

```
% git remote show handout
```

If this reports an error, run:

```
git remote add handout git://code.seas.harvard.edu/cs61/cs61-psets.git
```

Then run

```
git pull handout master
```

This will merge the midterm directory with your previous work; you should not run into any conflicts. After you've done this, run git push to save this merge back to code.seas.

Please place all answers for the exam in the file named exam.txt in the midterm directory. Be sure to label each question clearly, so there is no possible way we can be confused about which question you are answering. Please do **not** place your name anywhere in the exam.txt file. When you have completed the exam, edit the file policy.txt, and attest that you have followed the rules for the exam by writing, "I have neither given nor received help on this exam and have followed the rules as stated." Then type your name beneath it, indicating compliance. The exam is open book, open note, open computer. You may access the book and your own notes in paper form. You may also use a computer or equivalent to access class materials. However, you may not access non-class materials except as explicitly allowed below, and you may not write programs to answer questions. Specifically:

- You may access a browser and a PDF reader.
- You may access your own notes electronically.
- You may access an Internet site on which your own notes are stored.
- You may access system manual pages (man).
- You may access the cs61.seas.harvard.edu/wiki/2015 site (but not any prior years, even though they are linked off the site).

But, you may not access Google or Wikipedia or anything else except as directly linked from the cs61.seas.harvard.edu/wiki/2015 site. In particular:

- You may not access Piazza.
- You may not access course videos.
- You may not run a C compiler, assembler, on-line disassembler, calculator, or anything similar. Simple reading applications only.
- You may not access other students' notes.
- You absolutely may not contact other humans via IM or anything like it.
- You may not access solutions from any previous exam, by paper or computer.

Any violations of this policy, or the spirit of this policy, are breaches of academic honesty and will be treated accordingly. Please appreciate our flexibility and behave honestly and honorably.

Exam Policy Statement

At the end of your exam, assuming you have followed the rules of the exam, please write the following statement in the file policy.txt in the midterm directory of the cs61-psets repository. Place your name after it, asserting that you have followed the rules of the exam.

Unless otherwise stated, please make the following assumptions:

- A 32-bit little endian architecture
- The terms kilobyte (KB), megabyte (MB), and gigabyte (GB) refer to the binary values (i.e., 2 to some power) not decimal (i.e., 10 to some power).

1. Data Representation (10 points)

Sort the following expressions in ascending order by value, using the operators <, =, >. For example, if we gave you:

A. int i = 6;
B. int j = 0x6;
C. int k = 3;

you would write k < i = j or k < j = i.

a. unsigned char u = 0x191; truncated to 0x91 = 9*16+1 = 145
b. char c = 0x293; truncated to 0x92, but signed, so -0x6D
c. unsigned long l = 0xFFFFFFFF; That's 2^32 - 1
d. int i = 0xFFFFFFFF; That's -1
e. int j = i + 3; That's 2
f. 4 GB That's 2^32
g. short *s; sizeof(*s); That's 2
h. long l = 256; That's 256
i. (binary) 10000000000000000000000000000000 2^36
j. unsigned long Q = 0xACE - 0x101; 0x9CD=2509

1 points per operator/relationship (2 for the equals) b < d < e = g < a < h < j < c < f < i

or, if you typed variable names (as the example did, which was dumb on my part), it might be:

c < i < j = s < u < l(h) < Q < l(c) < 4 GB < (binary)10000000000000000000000000000000000000000

2. Memory (10 points)

For the following questions, select the part(s) of memory from the list below that best describes where you will find the object.

1. heap
2. stack
3. between the heap and the stack
4. in a read-only data segment
5. in a text segment starting at address 0x08048000
6. in a read/write data segment
7. in a register

Assume the following code, compiled without optimization.

```
#include
#include

const long maxitems = 1000;

struct info {
        char name[20];
        unsigned int age;
        short height;
} s = { "sushi", 1, 9 };

int
main(int argc, char *argv[])
{
        static long L = 0xbadf00d;
        unsigned long u = 0x8badf00d;
        int i, num = maxitems + 1;
        struct info *sp;

        printf("What did you do? %lx?\n", u);
        while (num > maxitems || num < 10) {
                printf("How much of it did you eat? ");
                scanf (" %d", &num);
        }
        sp = (struct info *)malloc(num * sizeof(*sp));
        for (i = 0; i < num; i++)
                sp[i] = s;

        return (0xdeadbeef);
}
```

A. The value 0xdeadbeef, when we are returning from main.
B. The variable maxitems
C. The structure s
D. The structure at sp[9]
E. The variable u
F. main
G. printf
H. argc

I. The number the user enters
J. The variable L

```
1 point each
A = 7 (in a register)
B = 4 (in a read-only data segment)`
C = 6 (in a read-write data segment)
D = 1 (the heap)
E = 2 (the stack)
F = 5 (in a text segment starting at 0x08048000)
G = 3 between the heap and the stack
H = 2 (the stack)
I = 2 (the stack)
J = 6 (in a read-write data segment)
```

3. Pot Pourri (10 points)

A. What does the following instruction place in %eax? `sarl $31, %eax`
B. True/False: A direct-mapped cache with N slots can handle any reference string with < N distinct addresses with no misses except for compulsory misses.
C. What is 1 (binary) TB in hexadecimal?
D. Write the answer to the following in hexadecimal: 0xabcd + 12
E. True/False: The garbage collector we discussed is conservative, because it only runs when we tell it to.
F. True/False: Given the definition `int array[10]` the following two expressions mean the same thing: `&array[4]` and `array + 4`.
G. Using the matrix multiply from lecture 12, in what order should you iterate over the indices i, j, and k to achieve the best performance.
H. True/False: `fopen, fread, fwrite, and fclose` are system calls.
I. Which do you expect to be faster (on the CS50 appliance): insertion sorting into a linked list of 1000 elements or into an array of 1000 elements?
J. What does the hardware do differently when adding signed versus unsigned numbers?

A. It fills eax with the sign bit of eax (i.e., all 0's or all 1's)
B. False
C. 1 TB = 2^40 = 1 followed by 40 zeros: so those 0's turn into the 10 hex 0's preceded by a 1: 0x10000000000
D. 12 = 0xC; 0xD + 0xC = (25 = 0x19), so the answer is 0xABD9
E. False (conservative because it never relaims something it shouldn't, but might not reclaim things it could).
F. True
G. ikj
H. False (they are calls to standard IO)
I. The array (see inclass and section work)
J. Nothing

4. Assembly and Data Structures (23 points)

For each code sample below, indicate the **most likely** type of the data being accessed. (If multiple types are equally likely, just pick one.)

A. `movzbl %al, %eax`
B. `movl -28(%ebp), %edx`

C. `movsbl -32(%ebp), %eax`
D. `movzwl -30(%ebp), %eax`

(2 points each)

A. movzbl %al, %eax (unsigned char)
B. movl -28(%ebp), %edx (either int, long, uint, or ulong)
C. movsbl -32(%ebp), %eax (char)
D. movzwl -30(%ebp), %eax (unsigned short)

For each code sample below, indicate the **most likely** data structure being accessed (assume that g_var is a global variable). Be as specific as possible.

E. `movzwl 6(%edx, %eax, 12), %eax`
F. `movl (%edx, %eax, 4), %eax`

```
movzbl  4(%eax), %eax
movsbl  %al, %eax
```

(3 points each -- you can get partial for the field of array of structures; 2 points for recognizing that it's a field in an array of structures; 3 points for the correct type of the field)

E. (unsigned short field in an array of structures) `movzwl 6(%edx, %eax, 8), %eax`
F. (array of ints, uints, longs or ulongs) `movl (%edx, %eax, 4), %eax`
G. There are at least three possible answers here
    1. char field from a structure (the one I intended)
    2. the 4th element of a char string
    3. an unsigned char and a signed char

```
movzbl  4(%eax), %eax
movsbl  %al, %eax
```

For the remaining questions, indicate for what values of the register contents will the jump be taken.

H.
```
xorl    %eax, %eax
jge     LABEL
```

I.
```
testb   $1, %eax
jne
```

J.
```
cmpl    %edx, %eax
jlt     LABEL
```

(2 points each)

H. Always
I. Any odd value (the fact that we're only looking at the lowest byte is pretty irrelevant)
J. Jumpe if EAX is less than EDX

5. Caching - part I (5 points)

Assume that we have a cache that holds four pages. Assume that each letter below indicates an access to a page. Answer the following questions as they pertain to the following sequence of accesses.

`E D C B A E D A A A B C D E`

A. What is the hit rate assuming an LRU replacement policy?
B. What pages will you have in the cache at the end of the run?
C. What is the best possible hit rate attainable if you could see into the future?

(2 points for each replacement policy and 1 point for the right pages in the cache)

A. Let's see what the cache looks like at each stage (the 4 letters represent the state of the cache and 1's after a line indicate hits). They do not have to have the resulting cache sorted.

```
E D C B
D C B A
C B A E
B A E D 1 (hit on D changes the order of the LRU to the next line)
B E D A 1 1 1 (hits on A, A, B -- changes to next order)
E D A B
D A B C 1 (hit on D, reorders to next line)
A B C D
B C D E
```

(2 points) So, the answer is 5/14 and

B. (1 point) what's left in the cache is: B C D E
C. With Belady's, we get:

```
E D C B
A E D B 1 1 1 1 1 1
C E D B 1 1
```

(2 points) So, our hit rate is 8/14 (or 4/7).

6. Caching - part II (12 points)

Intel and CrossPoint have announced a new persistent memory technology with performance approaching that of DRAM. Your job is to calculate some performance metrics to help system architectects decide how to best incorporate this new technology into their platform.

Let's say that it takes 64ns to access one (32-bit) word of main memory (DRAM) and 256ns to access one (32-bit) word of this new persistent memory, which we'll call NVM (non-volatile memory). The block size of the NVM is 256 bytes. The NVM designers are quite smart and although it takes a long time to access the first byte, when you are accssing NVM sequentially, the devices perform read ahead and stream data efficiently -- at 32 GB/second, which is identical to the bandwidth of DRAM.

A. Let's say that we are performing random accesses of 32 bits (on a 32-bit processor). What fraction of the accesses must be to main memory (as opposed to NVM) to achieve performance within 10% of DRAM?
B. Let's say that they write every byte of a 256 block in units of 32 bits. How much faster will write-back cache perform relative to a write-through cache? (An approximate order of magnitude will be sufficient; showing work can earn partial credit.)
C. Why might you not want to use a write-back cache?

A. (5 points) Let X be the fraction of accesses to DRAM: access time = 64X + 256(1-X). We want that to be <= 1.1*64 (within 10% of DRAM). So, 1.1*64 = 70.4. So, let's solve for: 64X + 256(1-X) = 70.4.

```
64X + 256 - 256X = 70.4.
(256X - 64X) = 256 - 70.4
```

```
192X = 186
X = 186/192
about .97
```

So, we need a hit rate in main memory of 97%

(4 for setting up the problem correctly -- partial credit for something that is almost right, but not quite; 1 for doing the algebra correctly).

B. (5 points) Write-through is going to cost 256ns/4 byte write = 256 * 64 = 2^8*2^6 = 2^14 = 16 K ns which is roughly 16 microseconds. If we assume a write-back, then it will take us 64 * 64ns to write into the DRAM, but then we get to stream the data from DRAM into the NVM at a rate of 32 GB/sec. So, 64*64 ns = 2^12 ns = 4 microseconds to write into DRAM. Let's convert 32 GB/second ito KB -- that's about 32 KB/microsecond. We need 1/4 of 1 KB which is 1/128 of a microsecond, which is about 8 ns. So, it's really really really fast to stream the data -- once you know that, then you also realize that the real difference is just the relative cost of writing to DRAM versus the cost of writing to NVM. So, the writeback cache is almost 4 times faster than the write through cache.

   You can get full credit by saying something like: the time to stream the data out of the DRAM into the NVM at the sequential speed is tiny relative to the time to write even a single word to DRAM, so the ultimate difference is the difference in writing to DRAM relative to NVM which is a ratio of 4:1. So, the writeback cache is about 4 times faster (because it is running at almost the full DRAM speed).

C. (2 points) A write-through cache will have very different persistence guarantees. If you need every 4-byte write to be persistent, then you have no choice but to implement a write-through cache.

7. Memory and Pointers (12 points)

If multiple processes are sharing data via mmap, they may have the file mapped at different virtual addresses. In this case, pointers to the same object will have different values in the different processes. One way to store pointers in mmapped memory so that multiple processes can access them consistently is using *relative* pointers. Rather than storing a regular pointer, you store the offset from the beginning of the mmapped region and add that to the address of the mapping to obtain a real pointer. An alternative representation is called *self-relative pointers*. In this case, you store the difference in address between the current location (i.e., the location containing the pointer) and the location to which you want to point. Neither representation addresses pointers between the mmapped region and the rest of the address space; you may assume such pointers do not exist.

A. State one advantage that relative pointers have over self-relative pointers.
   The key thing to understand is that both of these approaches use relative pointers and both can be used to solve the problem of sharing a mapped region among processes that might have the region mapped at different addresses.

   (2 points) Possible advantages:

   - Within a region, you can safely use memcpy as moving pointers around inside the region does not change their value. If you copy a self relative pointer to a new location, its value has to change. That is, imagine that you have a self-relative pointer at offset 4 from the region and it points to the object at offset 64 from the region. The value of the self relative pointer is 60. If I copy that pointer to the offset 100 from the region, I have to change it to be -36.
   - If you save the region as a uintptr_t or a char *, then you can simply add the offset to the region; self-relative-pointers will always be adding/subtracting from the address of the location storing

the pointer, which may have a type other than char *, so you'd need to cast it before performing the addition/subtraction.

- You can use a larger region: if we assume that we have only N bits to store the pointer, then in the base+offset model, offset could be an unsigned value, which will be larger than the maximum offset possible with a signed pointer, which you need for the self-relative case. That is, although the number of values that can be represented by signed and unsigned numbers differs by one, the implementation must allow for a pointer from the beginning of the region to reference an item at the very last location of the region -- thus, your region size is limited by the largest positive number you can represent.

B. State one advantage that self-relative pointers have over relative pointers.
(2 points) Possible advantages:

- You don't have to know the address at which the region is mapped to use them. That is, given a location containing a self-relative pointer, you can find the target of that pointer.

For the following questions, assume the following setup:

```
char *region;   /* Address of the beginning of the region. */

/*
 * The following are sample structures you might find in
 * a linked list that you are storing in an mmaped region.
 */
struct ll1 {
        unsigned  value;
        TYPE1     r_next;       /* Relative Pointer. */
};

struct ll2 {
        unsigned  value;
        TYPE2     sr_next;      /* Self-Relative Pointer. */
};

struct ll1 node1;
struct ll2 node2;
```

C. Propose a type for TYPE1 and give 1 sentence why you chose that type.
(2 points)
I would either say off_t, because that's how we refer to offsets in a file. Alternately, you could use an reasonably large unsigned or even signed type; we accepted uintptr_t and ptrdiff_t as well as int, unsigned, long.

D. Write a C expression to generate a (properly typed) pointer to the element referenced by the r_next field of ll1. (2 points) (struct ll1 *)(region + node1.r_next)

E. Propose a type for TYPE2 and give 1 sentence why you chose that type.
(2 points -- no partial)
This must be a signed type since you can both positive and negative values. A ptrdiff_t seems ideal; an off_t is fine as well. Again, we accepted any reasonably sized signed type.

F. Write a C expression to generate a (properly typed) pointer to the element referenced by the sr_next field of ll2. (2 points) (struct ll2 *)((char *)&node2.sr_next + node2.sr_next)