Midterm

Rules

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 your own class materials and public class materials. However, you *may not* access other materials except as explicitly allowed below. Specifically:

- You may access a browser and a PDF reader.
- You may access your own notes and problem set code electronically.
- You may access an internet site on which your own notes and problem set code are stored.
- You may access this year's course site.
- You may access pages directly linked from the course site, including our lectures, exercises, and section notes, and our preparation materials for the midterm.
- You **may** run a C/C++ compiler, including an assembler and linker, or a calculator.
- You may use a Python interpreter.
- You may access manual pages.

But:

- You **absolutely may not** contact other humans via IM or anything like it.
- You **may not** access Piazza.
- You **may not** access an on-line disassembler, compiler explorer, or similar application.
- You **may not** access Google or Wikipedia or anything else except as directly linked from the course site.
- You **may not** access solutions from any previous exam, by paper or computer, **except** for those on the course site.

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.

Completing your exam

First, merge your local cs61-exams repository with our handout. You should do this before beginning the exam.

```
$ git pull git://github.com/cs61/cs61-f18-exams.git master
```

You will enter your answers in the `midterm/midterm.md` file. Do not place your name in this file. (This enables us to grade all exams blindly.)

When you have completed the exam, edit the file `midterm/policy.txt` to sign your name. This is your promise that you have obeyed the exam rules in letter and spirit.

Then commit your changes and push them to your repository on GitHub:

```
$ git commit -a -m "Midterm Exam Answers"
$ git push
```

If you get an error message that you do not have access to push to github.com/cs61/cs61-f18-exams.git, this means that you are trying to push to our repository. Instead, push explicitly to your own repository:

```
$ git push https://github.com/cs61/cs61-f18-exams-YOURGITHUBREPONAMEHERE master
```

Make sure that you have entered your exam repository URL on the grading server for the midterm exam.

Notes

Assume a Linux operating system running on the x86-64 architecture unless otherwise stated. If you get stuck, move on to the next question. If you're confused, explain your thinking briefly for potential partial credit.

1. Architecture

QUESTION 1A. Write a C++ expression that will evaluate to false on all typical 32-bit architectures and true on all typical 64-bit architectures.

QUESTION 1B. Give an integer value that has the same representation in big-endian and little-endian, and give its C++ type. Assume the same type sizes as x86-64.

QUESTION 1C. Repeat question B, but with a different integer value.

QUESTION 1D. Complete this C++ function, which should return true iff it is called on a machine with little-endian integer representation. Again, you may assume the same type sizes as x86-64.

```
bool is_little_endian() {



}
```

2. Undefined behavior

The following code is taken from the well-known textbook *Mastering C Pointers* (with some stylistic changes). Assume it is run on x86-64.

```c
#include <stdlib.h>
#include <stdio.h>
void main() {
    int* x = malloc(400);
    if (x == nullptr) {
        printf("Out of memory\n");
        exit(0);
    } else {
        for (int y = 0; y <= 198; ++x) {
            *(x + y) = 88;
        }
    }
}
```

QUESTION 2A. List all situations in which this program will **not** execute undefined behavior. (There is at least one.)

QUESTION 2B. True or false? The expression `++x` could be replaced with `++y` without changing the meaning of the program.

Here's an updated version of the program.

```
#include <stdlib.h>
#include <stdio.h>
void main() {
    int* x = malloc(sizeof(int) * 200);
    if (x == nullptr) {
        printf("Out of memory\n");
        exit(0);
    } else {
        for (int y = 0; y <= 198; ++y) {
            *(x + y) = 88;
        }

        int i = random() % 200;
        printf("x[%d] == %d\n", i, x[i]);
    }
}
```

QUESTION 2C. True or false? The expression `*(x + y)` could be replaced with `x[y]` without changing the meaning of the program.

QUESTION 2D. True or false? The expression `*(x + y)` could be replaced by `* (int*) ((uintptr_t) x + y)` without changing the meaning of the program.

3. Odd alignments

QUESTION 3A. Write a `struct` definition that contains exactly seven bytes of padding on x86-64.

QUESTION 3B. Can an x86-64 `struct` comprise more than half padding? Give an example if so, or explain briefly why not.

The remaining questions consider a new architecture called **x86-64-rainbow**, which is like x86-64 plus special support for a fundamental data type called **color**. A color is a three-byte data type with red, green, and blue components, kind of like this:

```
struct color {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};
```

But unlike `struct color` on x86-64, which has size 3 and alignment 1, `color` on x86-64-rainbow has size 3 and **alignment 3**. All the usual rules for C abstract machine sizes and alignments still hold.

QUESTION 3C. What is the alignment of pointers returned by `malloc` on x86-64-rainbow?

QUESTION 3D. Give an example of an x86-64-rainbow `struct` that *ends* with *more than 16 bytes* of padding.

4. Debugging allocators

In problem set 1, you built a debugging allocator that could detect many kinds of error. Some students used exclusively **internal metadata**, where the debugging allocator's data was stored within the same block of memory as the payload. Some students used **external metadata**, such as a separate hash table mapping payload pointers to metadata information.

QUESTION 4A. Which kind of error mentioned by the problem set could **not** be detected by external metadata alone?

The problem set requires the m61 library to detect invalid frees, which includes double frees. However, double frees are so common that a special double-free error message can help users. Jia Tolentino wants to print such a message. Her initial idea is to keep a set of recently freed pointers:

```
static void* recently_freed[10];
static size_t recently_freed_index;

static bool is_recently_freed(void* ptr) {
    for (int i = 0; i != 10; ++i) {
        if (recently_freed[i] == ptr) {
            return true;
        }
    }
    return false;
}

static void add_recently_freed(void* ptr) {
    recently_freed[recently_freed_index] = ptr;
    recently_freed_index = (recently_freed_index + 1) % 10;
}
```

QUESTION 4B. What eviction policy is used for the `recently_freed` array?

Jia uses these helpers in `m61_free` as follows. (You may assume that `is_invalid_free(ptr)` returns true for every invalid or double free.)

```
void m61_free(void* ptr, const char* file, int line) {
    if (ptr != nullptr) {
        if (is_recently_freed(ptr)) {
            fprintf(stderr, "... double-free message ...");
            abort();
        } else if (is_invalid_free(ptr)) {
            fprintf(stderr, "... invalid-free message ...");
            abort();
        } else {
            add_recently_freed(ptr);
            ... actually free `ptr` ...
        }
    }
}
```

She has not yet changed `m61_malloc`, though she knows she needs to. Help her evaluate whether her design achieves the following mandatory and desirable (that is, optional) requirements.

> **Note**: As you saw in tests 32 and 33 in pset 1, aggressive attacks from users can corrupt metadata arbitrarily. You should assume for the following questions that such aggressive attacks do not occur. The user might free something that was never allocated, and might double-free something, but will never overwrite metadata.

QUESTION 4C. *Mandatory*: The design must not keep unbounded, un-reusable state for pointers that have been freed. Write "Achieved" or "Not achieved" and explain briefly.

QUESTION 4D. *Desirable*: The design should report a double-free as a double-free, not an invalid free. Write "Achieved" or "Not achieved" and explain briefly.

QUESTION 4E. *Mandatory*: The design must never report **valid** frees as double-frees or invalid frees. Write "Achieved" or "Not achieved" and explain briefly.

QUESTION 4F. Describe code to be added to `m61_malloc` that will achieve all mandatory requirements, if any are required.

5. Calling convention and optimizations

QUESTION 5A. Some aspects of the x86-64 calling convention are conventional, meaning that different operating systems could easily make different choices. Others are more fundamental in that they are built in to the x86-64 instruction set. Which of the following aspects of the calling convention are truly conventional (different operating systems could easily make different choices)? List all that apply.

1. The stack grows down
2. `%rsp` is a callee-saved register
3. `%rbx` is a callee-saved register
4. `%rbp` is a callee-saved register
5. The first argument register is `%rdi`
6. The second argument register is `%rsi`
7. The return register is `%rax`

QUESTION 5B. We saw the compiler sometimes place loop variables in callee-saved registers. For example, here:

```
extern unsigned g(unsigned i);

unsigned f(unsigned n) {
    unsigned sum = 0;
    for (unsigned i = 0; i != n; ++i) {
        sum += g(i);
    }
    return sum;
}
```

the `i`, `n`, and `sum` objects were stored in callee-saved registers `%rbx`, `%r12`, and `%rbp`, respectively. Describe a situation when this choice has a meaningful chance of boosting the overall performance of `f`.

QUESTION 5C. Write two substantively different C++ functions that could compile to the same assembly on x86-64. ("Substantively different" means you can't just change the function's name and types.)

QUESTION 5D. Which of the following properties of a function could you **reliably** infer from compiler-generated assembly? List all that apply; assume the function's name is not provided, and that the function does not call any other functions. Explain briefly if unsure.

1. The types of its arguments
2. Whether it can dereference a pointer argument
3. Whether it can use at least one of its arguments
4. Whether it has unused arguments

6. Bignums

x86-64 processors have built-in support for integers of up to 64 bits. For larger integers, we must turn to software. This simple "bignum" type represents a 128-bit unsigned integer.

```
struct bignum {
    unsigned long x, y;
};

void hex_print_bignum(bignum n) {
    if (n.y == 0) {
        printf("0x%lx", n.x);
    } else {
        printf("0x%lx%016lx", n.y, n.x);
    }
}
```

If `bignum bn` represents 2^{65}, then `hex_print_bignum(bn)` will print `0x20000000000000000` (that's a 2 followed by 16 zeros).

QUESTION 6A. Does `bignum` use a big-endian representation, a little-endian representation, or a mix?

QUESTION 6B. Complete the following function, which returns true iff `a > b`.

```
bool bignum_greater(bignum a, bignum b) {



}
```

QUESTION 6C. Complete the following function, which returns the sum `a + b`. (You may use the fact that if `x` and `y` are unsigned integers using computer arithmetic, then the addition `z = x + y` overflows iff `z < x || z < y`.)

```
bignum bignum_add(bignum a, bignum b) {



}
```

QUESTION 6D. Complete the same function **in x86-64 assembly**. Thanks to the calling convention, this function behaves as if the signature were as follows:

```
bignum* bignum_add(bignum* ret, unsigned long a_x, unsigned long a_y,
                   unsigned long b_x, unsigned long b_y);
// Stores result in `*ret` and returns `ret`.
```

You may translate your C++ code directly, or you may use the x86-64 `CF` carry flag and associated instructions to simplify the translation.

```
bignum_add:
```

7. Disassembly

Consider the following function in assembly:

```
_Z3gcdjj:
        movl    %esi, %eax
        cmpl    %esi, %edi
        jne     .L3
        jmp     .L2
.L8:
        subl    %eax, %edi
        cmpl    %edi, %eax
        je      .L2
.L3:
        cmpl    %edi, %eax
        jb      .L8
        subl    %edi, %eax
        cmpl    %edi, %eax
        jne     .L3
.L2:
        ret
```

QUESTION 7A. True or False:

1. The function likely takes 2 arguments.
2. The function accesses 4-byte integer values in primary memory.
3. The function likely works with signed integers.
4. There is likely a loop in the function.

5. The register `%eax` likely contains a loop index (a value updated by a fixed amount on each loop iteration).

QUESTION 7B. Give a input to this function such that it never returns.

QUESTION 7C. How would you modify the function, in assembly, such that it eventually returns for all inputs?

QUESTION 7D. Give one set of arguments for which the function terminates, and the corresponding return value.

8. Buffer overflows

```
_Z4buf1PKc:
        subq    $40, %rsp
        movq    %rdi, %rsi
        movq    %rsp, %rdi
        call    strcpy@PLT
        movq    %rsp, %rax
        addq    $40, %rsp
        ret
```

QUESTION 8A. Give argument(s) to function `_Z4buf1PKc`, including types and values, that would cause a buffer overflow that overwrote at least part of the function's return address; or explain briefly why this is impossible.

```
_Z4buf2m:
        subq    $24, %rsp
        movq    %rdi, %rdx
        leaq    16(%rsp), %rdi
        leaq    .LC0(%rip), %rsi
        movl    $0, %eax
        call    sprintf@PLT
        movl    $0, %eax
.L2:
        cmpb    $0, 16(%rsp,%rax)
        je      .L3
        incl    %eax
        jmp     .L2
.L3:
        addq    $24, %rsp
        ret
.LC0:
        .string "%zu"
```

QUESTION 8B. Which object in function `_Z4buf2m` is referenced using `%rip`-relative addressing?

QUESTION 8C. Give argument(s) to this function, including types and values, that would cause a buffer overflow that overwrote at least part of the function's return address; or explain briefly why this is impossible.

QUESTION 8D. It is possible to change two lines of assembly in this function so that the revised function performs the same task, but never causes buffer overflow. Describe how.

9. Get–set interfaces

The following questions concern **get–set interfaces**. These are map-like data structures which support two operations:

void set(*container*, *key*, *value*)
Change *container*'s value for *key* to *value*.

value_type get(container, key)
Returns *container*'s current value for *key*. This is the most recent *value* set by **set(*container*, *key*, *value*)** or, if there is no previous **set**, the natural default for the value type (0, `nullptr`, empty string—whatever's appropriate).

Here's the assembly definition of `xxx1`, which is either **get** or **set** for a data structure.

```
_Z4xxx1Pmm:
    movq    (%rdi,%rsi,8), %rax
    ret
```

QUESTION 9A. Is this **get** or **set**?

QUESTION 9B. What kind of data structure is this?

QUESTION 9C. What can you tell about the type of *key* arguments for this data structure?

QUESTION 9D. What can you tell about the type of *value* arguments?

QUESTION 9E. Write a C++ version of the *other* operation for the same data structure, including a function signature and any type definitions required.

Here's the assembly definition of `xxx2`, which is either **get** or **set** for a different data structure.

```
_Z4xxx2P4nodem:
.L28:
    testq   %rdi, %rdi
    je  .L30
    movq    (%rdi), %rax
    cmpq    %rsi, %rax
    je  .L35
    cmpq    %rax, %rsi
    jnb .L27
    movq    16(%rdi), %rdi
    jmp .L28
.L27:
    movq    24(%rdi), %rdi
    jmp .L28
.L30:
    xorl    %eax, %eax
    ret
.L35:
    movq    8(%rdi), %rax
    ret
```

QUESTION 9F. Is this **get** or **set**?

QUESTION 9G. Write a C++ version of the *other* operation for the same data structure, including a function signature and any type definitions required. You will need to call `malloc` or `new`.

10. Reference strings and hit rates

QUESTION 10A. Write a purely-sequential reference string containing at least five accesses.

QUESTION 10B. What is the hit rate for this reference string? Tell us the eviction algorithm and number of slots you've chosen.

The next two questions concern this ten-element reference string:

1 2 1 2 3 4 1 5 1 1

We consider executing this reference string starting with different cache contents.

QUESTION 10C. A three-slot LRU cache processes this reference string and observes a 70% hit rate. What are the initial contents of the cache?

QUESTION 10D. A three-slot FIFO cache processes this reference string with initial contents 4 1 2 and observes a 60% hit rate. Which slot was next up for eviction when the reference string began?

The eviction algorithms we saw in class are entirely **reactive**: they only insert a block when that block is referenced. This limits how well the cache can perform. A read cache can also be **proactive** by inserting blocks *before* they're needed, possibly speeding up later accesses. This is the essence of prefetching.

In a proactive caching model, the cache can evict and load **two or more** blocks per access in the reference string. A **prefetching policy** decides which additional, non-accessed blocks to load.

QUESTION 10E. Describe an access pattern for which the following prefetching policy would be effective.

> When accessing block *A*, also load block *A*+1.

QUESTION 10F. Write a reference string and name an eviction policy for which this prefetching policy would be **less** effective (have a lower hit rate) than no prefetching at all.

11. Coherence

QUESTION 11A. Which of the kinds of cache we discussed in class are typically coherent?

QUESTION 11B. Which of the kinds of cache we discussed in class are typically single-slot?

Stdio-like caches are **not** coherent. The remaining questions concern potential mechanisms to make them coherent with respect to disk files.

> **Pedantic note.** Sometimes a read-from-cache operation will occur concurrently with (at the same time as) a write to stable storage. The read operation counts as coherent whether or not it reflects the concurrent write, because logically the read and write occurred "at the same time" (neither is older).

QUESTION 11C. First, the new `bool changed()` system call returns true if and only if a `write` was performed on some file in the last second.

Describe briefly how `changed` could be used to make a stdio cache coherent, or explain why it could not.

QUESTION 11D. Second, the new `int open_with_timestamp(const char* filename, unsigned long* timestamp, ...)` system call is like `open`, except that every time a change is made to the underlying `filename`, the value in `*timestamp` is updated to the time, measured in milliseconds since last boot, of the last `write` operation on the file represented by file descriptor `fd`.

Describe briefly how `open_with_timestamp` could be used to make a stdio cache coherent, or explain why it could not.

QUESTION 11E. Describe briefly how `mmap` could be used to make a stdio cache coherent, or explain why it could not.

12. System calls

QUESTION 12A. The following system calls have just been made:

```
int fd = open("f.txt", O_WRONLY | O_CREAT | O_TRUNC);
ssize_t nw = write(fd, "CS121 is awesome!", 17); // returned 17
```

What series of system calls would ensure that, after all system calls Your task is to write a series of system calls so that, after all system calls complete, the file `f.txt` contains the text "`CS 61 is terrible`" (without the quotation marks)? Minimize the number of bytes written.

QUESTION 12B. Which of the following file access patterns might have similar output from the `strace` utility? List all that apply or say "none."

1. Sequential byte writes using stdio
2. Sequential byte writes using `mmap`
3. Sequential byte writes using system calls

QUESTION 12C. Which of the following file access patterns might have similar output from the `strace` utility? List all that apply or say "none."

1. Sequential byte writes using stdio
2. Sequential block writes using stdio
3. Sequential byte writes using system calls
4. Sequential block writes using system calls

QUESTION 12D. Which of the following file access patterns might have similar output from the `strace` utility? List all that apply or say "none."

1. Reverse-sequential byte writes using stdio
2. Reverse-sequential block writes using stdio
3. Reverse-sequential byte writes using system calls
4. Reverse-sequential block writes using system calls

13. Potpourri

QUESTION 13A. True or false? Pointer arithmetic is only valid on pointers that point into arrays. Explain briefly.

QUESTION 13B. True or false? All x86-64 fundamental data types have alignments that are powers of 2. Explain briefly.

QUESTION 13C. True or false? Pointer arithmetic on `char*` pointers behaves the same way as address arithmetic (i.e., arithmetic on `uintptr_t` unsigned integers), including with respect to undefined behavior. Explain briefly.

QUESTION 13D. True or false? System calls are just as fast as function calls. Explain briefly.

14. The end

QUESTION 14A. How are you using lecture notes? What improvements would you suggest? Any answer but no answer will receive full credit.

QUESTION 14B. How are you using section? What improvements would you suggest? Any answer but no answer will receive full credit.

QUESTION 14C. (College only) Write any secret words you know.