Final practice

Final Exam: Wed March 21th 3:30 - 6:30pm
CEMEX

This is our registrar-scheduled exam time. There is no alternate exam. If you need arrangements for OAE accommodations, please follow up by email to cs107@cs.

The final is a closed-book exam. We will provide the reference sheet of C library functions and x86 instructions. You may bring one double-sided sheet of paper (size 8.5" x 11") prepared with your own printed and/or hand-written notes. No additional paper materials are to be used during the exam. We will use the BlueBook exam software, so be sure to bring your laptop, fully-charged and ready to go.

Material
Check your rear-view mirror for the very impressive list of things you've learned in 107!

- C—strings, arrays, pointers, &, *, void*, pointer arithmetic, typecasts, function pointers
- Data representation—bits, bytes, ASCII, two's complement integers, floating point, pointers, aggregate types (arrays and structs)
- x86-64 assembly—data access and addressing modes, arithmetic and logical ops, implementation of C control structures, call/return
- Address space—layout and purpose of text/data/stack/heap segments, handling of globals/locals/parameters
- Runtime stack—caller/callee protocol, parameter passing, use of registers
- Memory and heap—stack versus heap allocation, heap allocator strategies and tradeoffs
- Performance—compiler optimizations, profiling

The topic coverage for the final exam will be comprehensive and span the entire quarter but expect more emphasis on material covered after the midterm. The problems in this handout focus on post-midterm content (i.e. >= lab5 and assign5). For review on the earlier topics, rework the midterm exam, midterm prep materials, or previous labs/assignments.

Some of these are adapted from problems that appeared on previous CS107 final exams; others were written anew. The solutions are available in a separate handout. Work the problems in exam-like conditions for best effect. Reviewing the solutions afterward can confirm your understanding, but be wary of reading the problems side-by-side with the solution as that may mislead you into a false sense of readiness.

Our manifesto of advice and encouragement on how to prepare for and crush a CS107 exam is on our web site at http://cs107.stanford.edu/advice/exams.html. Please check it out!
Floating point representation

• Consider the following function:

```c
float sum(float x, float y, float z)
{
    return x + y + z;
}
```

Below are four calls to `sum`. For each call, identify whether it produces a mathematically correct result. If the result is incorrect, explain why/how it is incorrect.

a) `sum(1 << 30, 1 << 30, 1 << 30)`
b) `sum(1 << 30, -(1 << 30), 2)`
c) `sum(1 << 30, 2, -(1 << 30))`
d) `sum(FLT_MAX, FLT_MAX, -FLT_MAX)`

• A normalized IEEE 32-bit float is stored as the following bit pattern:

```
N EEEEEEEE SSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
```

where N is the sign bit (1 if negative), \( E \) is the 8-bit exponent (with a bias of 127), and \( S \) is the 23-bit significand, with an implicit leading "1."

a) What is true about the bit pattern for all exact powers of 2 that fall within the range of a normalized IEEE 32-bit float?

b) What is the bit representation for decimal 0.25?

• In ten words or less, explain the biggest advantage of a floating point number representation versus a fixed-point format.
x86 assembly
Below are two versions of the assembly generated for the binky function. The version on the left was compiled -Og, the right was compiled -O2.

```
binky:
push %rbx
sub $0x10,%rsp
movl $0x10,(%rsp)
movl $0x0,0x4(%rsp)
mov %esi,0x8(%rsp)
movsbl (%rdi),%eax
mov %eax,0xc(%rsp)
lea 0x8(%rsp),%rbx
jmp .L2
.L1:
mov (%rsp),%edx
lea (%rdx,%rdx,4),%edi
mov %edi,(%rsp)
mov (%rax),%rax
lea (%rsp,%rax,4),%rsi
callq <winky>
test %rax,%rax
jne .L1
xor %eax,%eax
add $0x18,%rsp
retq
```  
```
binky:
sub $0x18,%rsp
movl $0x10,(%rsp)
movl $0x0,0x4(%rsp)
mov %esi,0x8(%rsp)
movsbl (%rdi),%eax
mov %eax,0xc(%rsp)
lea 0x8(%rsp),%rax
.L1:
mov (%rsp),%edx
lea (%rdx,%rdx,4),%edi
mov %edi,(%rsp)
mov (%rax),%rax
lea (%rsp,%rax,4),%rsi
callq <winky>
test %rax,%rax
jne .L1
xor %eax,%eax
add $0x18,%rsp
retq
```  

Fill in the blanks in the C code below to match the unoptimized assembly from above left. There will be exactly one typecast. Note this is nonsense code, not intended to do anything meaningful.

```c
int binky(char *param1, long param2)
{
    int arr[4] = { _____________, _____________, _____________, ____________};
    void *ptr = _____________________________________________________________;
    while ( ________________________________________________________ ) {
        ____________________________________________________________;
        ____________________________________________________________;
    }
    return 0;
}
```

- The unoptimized binky has two jump/branch instructions; the optimized has only one. What changed in the control flow and why is this transformation legal?
- The unoptimized binky has a call to strlen; the optimized does not. What happened to the call?
- The optimized binky contains an xor instruction, the unoptimized does not. What instruction(s) is that xor replacing?
Runtime stack
That famously sloppy bank programmer is at it again. The (buggy!) atm functions below are intended to allow a withdrawal of any amount up the user's entire balance. Its C source is below on the left and its generated assembly is on the right.

```c
int how_much(int max)
{
    char buf[10];
    int amount;
    do {
        printf("How much to withdraw? ");
        gets(buf);
        amount = atoi(buf);
    } while (amount >= max);
}

int main(int argc, char *argv[])
{
    int balance = lookup_balance_for_user();
    unsigned int request = how_much(balance);
    balance -= request;
    set_balance_for_user(balance);
    printf("Balance now $%d.\n", balance);
    return 0;
}
```

```assembly
how_much:
    push %rbx
    sub $0x10,%rsp
    mov %edi,%ebx
    .L1:
        mov $0x4006a4,%edi
        mov $0x0,%eax
        callq <printf>
        mov %rdi,%rdi
        callq <get>
        mov %rdi,%rdi
        callq <atoi>
        cmp %ebx,%eax
        jge .L1
        add $0x10,%rsp
        pop %rbx
    retq

main:
    push %rbx
    callq <lookup_balance_for_user>
    mov %eax,%ebx
    mov %eax,%edi
    callq <how_much>
    sub %eax,%ebx
    mov %ebx,%edi
    callq <set_balance_for_user>
    mov %ebx,%esi
    mov $0x40072b,%edi
    mov $0x0,%eax
    callq <printf>
    mov $0x0,%eax
    pop %rbx
    retq
```

The functions `lookup_balance_for_user` and `set_balance_for_user` (not shown) are called correctly and you can assume they work properly. The `main` and `how_much` functions are buggy.

- The prototype for the `how_much` function states that it returns an integer, but the function has no return statement. What will be the return value of the function? *(Hint: trace the assembly)*
- You have a measly $107 dollars in your account, but aspire to someday have your balance reach six figures. You note that the programmer's careless mixing of signed and unsigned types makes it vulnerable to a rather simple attack. Explain what input you can provide to the atm program to crank up your balance to your desired goal today.
• Now don you your white hat and tell the bank how they can eliminate this loophole by adding exactly one typecast into the C code. Indicate what typecast is needed where and explain how it works.

• With that fixed and your balance restored to $107, you are back to pondering what further mischief you can muster. There is another possible exploit that allows you to control your balance, this one much sneakier than the first, that preys on the inherent weakness in `gets`. Explain how to construct an input to the atm program that will achieve your balance goal. *(Hint: diagram the stack frames and trace the assembly)*
**Heap allocator**

You are writing code for a simple allocator that uses a block header, no footer, and maintains an explicit free list. Implementation details of this allocator include:

- The total block size (size of payload + size of header) for all blocks is required to be an exact power of 2. The payload must be at least 8 bytes and block header is 1 byte, so the minimum total block size rounds up to 16. Payload pointers are not required to be aligned.
- The block header is an 8-bit unsigned char that bit-mashes together the block information:
  - most significant bit is 1 if block is freed, 0 if in-use
  - lower 7 bits store \( \log_2 \) of the total block size
- The allocator maintains an explicit free list as a singly-linked list stored in the block payload. A global variable points to the header of the first free block (or NULL if the free list is empty). The first 8 bytes of the payload of a free block store a pointer to the header of another free block. The last free block on the list stores NULL in its payload.

Here is an example heap after a few requests have been serviced:

```
0x7000  7010  7030  7050  7060
F:0     F:0     F:1     0x0     F:0
Sz:4    Sz:5    Sz:5    0x0     Sz:4
                 0x7030
```

This heap starts at address 0x7000 and has size 112 bytes. Three blocks are in-use blocks, two are free. The in-use payloads are shown shaded in gray. The header of the first block has the most significant bit off (block is in-use) and the lower bits store 4 (total block size is 16 or \( 2^4 \)). The free list points to the header at 0x7060, the payload of that block points to the header at 0x7030, the payload of that block stores NULL.

Below are the allocator’s global variables, constants, and type definitions.

```c
// type: header stores per-block housekeeping
typedef unsigned char Header;

static Header *heap_start;    // first header in heap segment
static size_t heap_size;      // number of bytes in heap segment
static Header *free_list;     // header of first block on free list (NULL if none)

// masks to divide header into most significant bit and all other bits
#define MSB_MASK   (1 << ((sizeof(Header)<<3) -1))
#define SIZE_MASK  (MSB_MASK - 1)
```
• The `MSB_MASK` and `SIZE_MASK` are used to extract the most significant bit from the other bits in the header. You run the preprocessor and see that it replicates these complex expressions at each use and are concerned this will degrade throughput. Never fear, gcc has your back! What specific compiler optimization will be applied to avoid repeatedly re-computing the mask?

• The `is_free` function is given a pointer to a block’s header and returns true if the block is free, false otherwise.

```c
bool is_free(Header *hdr)
{
    return /* expr */ ;
}
```

`is_free` is missing the test expression. From the choices below, circle all options that evaluate to the correct result.

- `(*hdr & MSB_MASK) == MSB_MASK`
- `(*hdr & MSB_MASK)`
- `~(*hdr ^ MSB_MASK)`
- `*hdr < 0`
- `*(signed char *)hdr < 0`
- `*(int *)hdr < 0`
- `(int)*hdr < 0`

• Implement `get_blocksize`. Given a pointer to a block’s header, the function returns the total number of bytes in the block.

```c
size_t get_blocksize(Header *hdr)
{
}
```

• Implement `get_neighbor`. Given a pointer to a block’s header, the function returns a pointer to the header of the neighboring block to the right, i.e., at next higher address in heap. If this block is lastmost in the heap, the function returns `NULL`.

```c
Header *get_neighbor(Header *hdr)
{
}
```

The first 8 bytes of the payload of a free block are used to store a pointer to the header of the next block on the free list. Helper functions `get_next` and `set_next` read and write those links.

```c
Header *get_next(Header *hdr);
void set_next(Header *first, Header *second);
```

• Implement the `set_next` function to link from first to second using a single call to `memcpy`.

```c
void set_next(Header *first, Header *second)
{
    memcpy(___________________ , _______________________ , ___________________);
}
```
• On second thought, you recall the lab8 performance timings and realize there has to be a better way. Re-implement `set_next` to link from first to second using a single pointer assignment.

```c
void set_next(Header *first, Header *second)
{
}
```

You will use `set_next` and `get_next` in implementing `remove_from_freelist` and can assume both functions work correctly.

The function `remove_from_freelist` is given a pointer to a header currently on the free list. An implementation is started below that searches for the entry on the list. Add the necessary code to then remove the entry from the list.

```c
void remove_from_freelist(Header *to_remove)
{
    Header *prev = NULL, *cur = free_list;
    while (cur != to_remove) {
        prev = cur;
        cur = get_next(cur);
    }
}
```

• In this allocator, two adjacent blocks can only be coalesced if they are both free and have the same total block size. Why must they be the same size?

• Implement the `myfree` function. It should mark the block as free and add it to front of the free list. If the neighboring block to the right is free and same size, it should be coalesced into this one and the neighbor’s header is removed from the free list. Attempt to coalesce only a single neighbor, not a sequence of neighbors. You may make use of any of the previous helper functions and can assume they work correctly.

```c
void myfree(void *payload)
{
}
```

• This allocator maintains the free list in a singly-linked list and profiling shows traversing the list to be a performance bottleneck for both `mymalloc` (which traverses to find an appropriately-sized block) and `myfree` (which traverses within `remove_from_freelist` when coalescing). You are considering two possible strategies to combat this: double-linking the list or segregating the list by size. Consider each proposed change and provide your cost/benefit evaluation of pursuing it. Your analysis should address the specific impacts of:

  What is the added code complexity?
  What is the expected effect on throughput of `mymalloc`? on throughput of `myfree`?
  What is the change in utilization?

Strategy A: double-link the free list.

Strategy B: segregate the free list by size into multiple lists.