Final Exam
CSC 252
6 May 2025
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Jack Cashman, Ethan Chen, Weikai Lin, Jiaqi Nie, Chenrui Wang, Boyi Zhang

Name: _____

Problem 0 (4 points):	_____
Problem 1 (12 points):	_____
Problem 2 (24 points):	_____
Problem 3 (42 points):	_____
Problem 4 (48 points):	_____
Problem 5 (36 points)	_____
Total (166 points):	_____

Remember "**I don't know**" is given 15% partial credit, but you must erase everything else. This does not apply to extra credit questions (if any).

Your answers to all questions must be contained in the given boxes. Use spare space to show all supporting work to earn partial credit.

You have 180 minutes to work.

Please sign the following. I have not given nor received any unauthorized help on this exam.

Signature:_____

Have a good summer break and GOOD LUCK!!!

Problem 0: Warm-up (4 Points)

What's the most surprising thing you've learned after the midterm?

Problem 1: Miscellaneous (12 points + 4 points extra credit)

(8 points) Represent hexadecimal value 66 in decimal and binary.

(4 points) Do we need branch prediction in a single-cycle microarchitecture? Explain.

(4 points extra credit) We very briefly mentioned out-of-order execution in the class, where the idea is for the hardware to automatically figure out how to reorder instructions to speed-up the execution while maintaining program correctness. Can this be done by a compiler? What are the trade-offs?

Problem 2: Floating-Point Arithmetics (24 points + 8 points extra credit)

Part a) (8 points)

(4 points) Write -12$\frac{3}{16}$ in the binary-normalized scientific notation.

-1.1000011×2^3

(4 points) Can $\frac{3}{16} + \frac{1}{2^{30}}$ be represented precisely in the IEEE single precision format? Briefly explain your reasoning.

No, at 3/16, there are not enough fraction bits to represent the addition of 1/(2^30).

Part b) (16 points + 8 points extra credit) Now we are going to design a new 13-bit floating-point standard, with main characteristics consistent with the IEEE floating point standards that we have discussed in the class. The new standard must satisfy the following two requirements:

- It must precisely represent the value -15.625;
- The smallest positive value it can represent is 2^{-36}

(4 points) How many bits are used for the fraction in this new standard?

6

(4 points) How many bits are used for the exponent field in this new standard?

6

(8 points) represent 7.96875 (1.1111111×2^2) in your new 13-bit standard with the following rounding methods and give your answer in binary.

1) round-to-nearest-even and

2) Round-toward-zero.

1) 0 100010 000000
2) 0 100001 11111111

(4 points extra credit) Calculate the product of 0011111110000 and 0100000000000, both written in this standard. Write your answer in the same standard.

0100000110000

(4 points extra credit) What is the next larger representable value after 0011111111111 in your new standard? Express your answer using scientific notation.

2

Problem 3: Assembly Programming (42 points)

Conventions (unless otherwise noted):

1. For this section, the assembly shown uses the AT&T/GAS syntax **opcode src, dst** for instructions with two arguments where **src** is the source argument and **dst** is the destination argument. For example, this means that **mov a, b** moves the value **a** into **b.**
2. All C code is compiled on a **64-bit machine**, where arrays grow toward higher addresses.
3. We use the x86 calling convention. That is, for functions that take two arguments, the first argument is stored in **%edi (%rdi)** and the second is stored in **%esi (%rsi)** at the time the function is called; the return value of a function is stored in **%eax (%rax)** at the time the function returns.
4. We use the **Little Endian** byte order when storing multi-byte variables in memory.

In the buffer bomb lab, you learned how to exploit buffer overflow vulnerabilities in programs. Buffer overflow opportunities are not always present. Luckily (or unluckily), there are other vulnerabilities that you can learn to exploit.

A **use-after-free (UAF)** bug occurs when a program frees a heap object but later continues to use the (now-dangling) pointer. Unlike a classic stack buffer overflow, the memory layout is no longer under the program's control; an attacker may reclaim the same chunk and overwrite the previously valid data.

Refer to the code below:

```
struct Obj {
      char  data[8];
      void (*func)();
};

void secret() { puts("You called the secret function!"); }
void hello()  { puts("Hello world"); }

int main() {
      // Section 1
      Obj *o = (Obj*)malloc(sizeof(Obj));
      o->func = hello;

      // Section 2
      free(o);

      // Section 3
      char *buf = (char*)malloc(16);
      puts("Enter payload:");
      fgets(buf, 16, stdin);
```

```
    // Section 4
    o->func();
    return 0;
}
```

In this code, an object, `o`, is initialized with malloc in section 1. The object is later freed in section 2. In section 3, a C string with the same size as the original object `o` is allocated. **Assume `malloc()` will choose to place the new C string `buf` at the same memory address as the object `o`.**

Then, the string `buf` is written to by user input using `fgets()`. `fgets()` is similar to `gets()`, but takes two additional arguments in addition to the buffer address. The second argument to `fgets()` is the maximum number of characters to read (including the `null` terminator). The third argument is the input stream to read from (in this case, `stdin`). The `fgets()` function here is safe from buffer overflow. However, in section 4, the freed object `o`'s function pointer `func()` is called. Recall that `o` has already been freed, and the content of `o` has been overwritten by the user's input into `buf`.

The C code is then compiled to the following assembly code on a 64 bit x86 machine. Address space layout randomization is turned off.

```
00000000004011c4 <main>:
  4011c4:  55                    push   %rbp
  4011c5:  48 89 e5              mov  %rsp,%rbp
  4011c8:  48 83 ec 10           sub  $0x10,%rsp
  4011cc:  bf 10 00 00 00        mov  $0x10,%edi
  4011d1:  e8 ca fe ff ff        call   4010a0
  4011d6:  48 89 45 f0           mov  %rax,-0x10(%rbp)
  4011da:  48 8b 45 f0           mov  -0x10(%rbp),%rax
  4011de:  48 c7 40 08 ab 11 40  movq   $0x4011ab,0x8(%rax)
  4011e6:  48 8b 45 f0           mov  -0x10(%rbp),%rax
  4011ea:  48 89 c7              mov  %rax,%rdi
  4011ed:  e8 7e fe ff ff        call   401070
  4011f2:  bf 10 00 00 00        mov  $0x10,%edi
  4011f7:  e8 a4 fe ff ff        call   4010a0
  4011fc:  48 89 45 f8           mov  %rax,-0x8(%rbp)
  401200:  bf 22 20 40 00        mov  $0x402022,%edi
  401205:  e8 76 fe ff ff        call   401080
  40120a:  48 8b 15 1f 2e 00 00  mov  0x2e1f(%rip),%rdx
  401211:  48 8b 45 f8           mov  -0x8(%rbp),%rax
  401215:  be 10 00 00 00        mov  $0x10,%esi
  40121a:  48 89 c7              mov  %rax,%rdi
  40121d:  e8 6e fe ff ff        call   401090
```

```
401222:  48 8b 45 f0          mov   -0x10(%rbp),%rax
401226:  48 8b 40 08          mov   0x8(%rax),%rax
40122a:  ff d0                 call   *%rax
40122c:  b8 00 00 00 00       mov   $0x0,%eax
401231:  c9                          leave
401232:  c3                          ret
```

(20 points) What are the C functions that each of the `call` instructions in addresses `4011d1`, `4011ed`, `4011f7`, `401205`, and `40121d` actually call?

```
4011d1: malloc
4011ed: free
4011f7: malloc
401205: puts
40121d: fgets
```

(4 points) What is the meaning of the value stored in `%rax` after the function call at address `4011d1`?

The value stored in rax at that point represents the location in memory of the object o.

(4 points) Judging from the assembly code above, what is the address of the `hello()` function?

4011ab

(4 points) What is stored at the memory location `402022`?

The string “Enter payload:”

(4 points) The address of the `secret()` function is `401196`. How would this be encoded in a 64-bit little endian machine?

9611400000000000

(6 points) Write an `stdin` input to `fgets()` that will make the program call the secret function (similar to the "exploit string" you saw in the buffer overflow lab). Express your answer in hexadecimal and explain your answer.

0000000000000000961140000000000
The first 8 bytes can be anything

Problem 4: Cache (48 points)

- There is a machine where the physical addresses are 8-bit long and the physical memory is byte addressable.
- The cache is direct-mapped and contains 8 cache lines.

(4 points) How many bits in the physical addresses are needed for the set index?

3

(4 points) Should each cache line have a replacement bit? Explain your answer.

We have a program that accesses a list of the following provided memory addresses. First few Hit/Miss results are given, while others are intentionally hidden for you to figure out. Assume that the cache is initially empty.

#	Address	Hit/Miss
1	0000 0000	Miss
2	0000 0001	Hit
3	0000 0010	Hit
4	0000 0100	Miss
5	0000 0000	Hit
6	0000 0100	Hit
7	1000 0000	Miss
8	0000 0000	Miss

(4 points) Using the hit/miss pattern above, determine the number of offset bits.

2

(4 points) How many bits are there for the tag field?

3

(4 points) What is the total data capacity of the cache in bytes? Show your math.

8*2^2

(20 points) Fill in the blank part (Miss and Hit) and give the cache hit rate here.

50%

(4 points) If we replay the **same trace a second time** immediately after the first run (i.e., cache state from the first run is kept), what is the hit rate for the second run?

62.5%

(4 points) Redesign the cache so that you get a 100% hit rate for the second run with minimal tag overhead. You may change only the organization of the cache while keeping the total data capacity the same. Assume the LRU replacement policy.

2-way set-associative.

Problem 5: Virtual Memory (32 points)

Assume a system with the following characteristics:

1. Virtual address space is 512 KB and is byte addressable.
2. Physical memory is 64 KB and is byte addressable.
3. Page size is 512 Bytes ($512 = 2^9$).
4. One level page table, where each page table entry contains a valid bit, and the physical page number.
5. Integer is 32 bits.

Part a) (20 points)

(4 points) What is the size of the PPN and the VPN?

PPN = **7**
VPN = **10**

(4 points) What is the size of one PTE, assuming that the only overhead in the PTE is one valid bit and that the disk address is 3 bits longer than the PPN?

PTE = 1 + 3 + PPN(7) = **11**

(4 points) Assume that you've decided to experiment with a page size of 8 KB; what is the new size of PTE?

PPN = 3
PTE = 1 + 3 + PPN(3) = **7**

(4 points) How would the page size need to be changed in order for the size of PTE to be 4 bits?

PTE = 4 = 1 + 3+ PPN(n)
n must be equal to 0 so there must be no page table which is only possible given a page size equivalent to the physical memory size of **64 KB**

(4 points) What is the relationship between page size and PTE size?

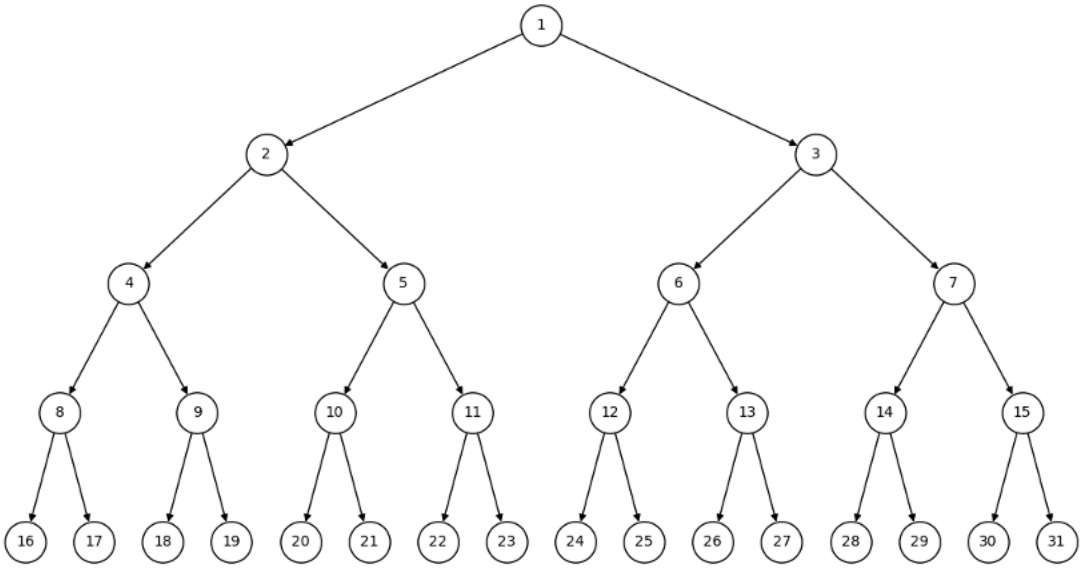
There's an **inverse relationship** between page and page table entry size. With fewer pages less bits need to be used to refer to those pages but when a fixed memory size divides into less pages then the amount of data in each page is larger. When the size of the pages are reduced the number of pages increases and in turn so does the page table.

(4 points) How does the addition of a TLB impact the latency of TLB misses?

When a TLB is added to a virtual memory system the latency of TLB misses **increases** with the latency of the TLB search. When a TLB is used in a virtual memory system the page table searches must first look to see if the given page table is cached in the TLB, then, in the case of a miss the page is retrieved from memory just as it would in a non TLB system. While the latency significantly deceased in the case of a TLB hit, on TLB misses the latency can be expressed as (TLB SEARCH TIME) + (MEMORY SEARCH TIME) which is greater than in a machine without TLB which would only have the latency of memory search time.

Part b) (12 points)

For this part, we are working with a balanced binary tree (see visualization below), where there are exactly 2^n nodes in layer n (n starts from 0). This tree is stored in level-order (breadth-first) in memory. This means that the root node (denoted as node 1) is stored at Tree[0], and nodes 2 and 3 (the children of the root, from left to right) are stored at Tree[1], Tree[2], respectively, and so on. In this particular tree, each node stores 512 bits. Assume that the tree is page-aligned, i.e., the root note is stored at the start of a page. For all the problems below, you may assume that the first page is already loaded to the physical memory.



(4 points) What is the **maximum** number of page faults possible, where you must visit exactly one node in each layer, starting from the root note? Give a traversal path (e.g., root → node 2 → ...) that leads to that number. Note that you can visit only nodes that are the children of your current node.

Two. Example path: 1 -> 2 -> 5 -> 10 -> 21

(4 points) Given the same constraints above, What is the **minimum** number of page faults possible? Give a traversal path.

One. 1 -> 2 -> 4 -> 8 -> 16

(4 points) What is the **minimum** page size, in bytes, for there to exist a path that will result in **zero** page fault? Your number should be a power of 2. Show your work.

2^10 or 1024 bytes. Need to store at least nodes 1-16 (inclusively).