CSE 351 Final Exam - Winter 2017

March 15, 2017

Please read through the entire examination first, and make sure you write your name and NetID on all pages! We designed this exam so that it can be completed in 100 minutes.

There are 8 problems for a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space, you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones, no laptops). Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good Luck!

Name: __________________________

Student ID: ______________________

Section: __________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max Score</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. C and Assembly</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2. Buffer Overflow</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3. Caches</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4. Processes</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5. Virtual Memory</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6. Memory Allocation</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7. Memory Bugs/Java</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8. Pointers and Values</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>

| Extra Credit             | 10        |       |
1. C and Assembly (15 points)

Consider the following (partially blank) x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit, and the machine is little-endian. All the values in memory are in hex, and the address of each cell is the sum of the row and column headers: for example, address 0x1019 contains the value 0x18.

Assembly code:

```assembly
foo:
    movl $0, _____

L1:
    cmpl $0x0, %rdi
    je L2
    cmpl _____, 0x1(%rdi)
    je _____
    movl 0x8(%rdi), %rdi
    jmp _____

L2:
    ret

L3:
    movl (%rdi), %eax
    jmp L2
```

C code:

```c
typedef struct person {
    char height;
    char age;
    struct person* next_person;
} person;

int foo(person* p) {
    int answer = _____;
    while (_______) {
        if (p->age == 24) {
            answer = p->_______;
            break;
        }
        p = ______________;
    }
    return answer;
}
```

Memory Listing

Bits not shown are 0.

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x01</th>
<th>...</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>80</td>
<td>1B</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1008</td>
<td>80</td>
<td>1B</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1010</td>
<td>3F</td>
<td>18</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1018</td>
<td>3F</td>
<td>18</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1020</td>
<td>00</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1028</td>
<td>18</td>
<td>10</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1030</td>
<td>18</td>
<td>10</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1038</td>
<td>40</td>
<td>40</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1040</td>
<td>40</td>
<td>40</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1048</td>
<td>00</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

(a) Given the code provided, fill in the blanks in the C and assembly code.
(b) Trace the execution of the call to `foo((person*) 0x1028)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place the assembly instruction and the values of the appropriate registers after that instruction executes. You may leave those spots blank when the value does not change. You might not need all steps listed on the table.

(c) Briefly describe the value that `foo` returns and how it is computed. Use only variable names from the C version in your answer.
2. Buffer Overflow (15 points)

The following code runs on a 64-bit x86 Linux machine. The figure below depicts the stack at point A before the function `gatekeeper()` returns. The stack grows downwards towards lower addresses.

```c
void get_secret(char*);
void unlock(void);
void backdoor(void);

void gatekeeper() {
    char secret[8];
    char buf[8];
    fill_secret(secret);
    gets(buf);
    if (strcmp(buf, secret) == 0)
        unlock();
    A:
        return 0;
}
```

You are joining a legion of elite hackers, and your final test before induction into the group is gaining access to the CIA mainframe. `gatekeeper()` is a function on the mainframe that compares a password you provide with the system’s password. If you try to brute force the password, you will be locked out, and your hacker reputation will be tarnished forever.

Assume that `fill_secret()` is a function that places the mainframe’s password into the `secret` buffer so that it can be compared with the user-provided password stored in `buf`.

Recall that `gets()` is a `libc` function that reads characters from standard input until the newline (\n) character is encountered. The resulting characters (not including the newline) are stored in the buffer that’s given to `gets()` as a parameter. If any characters are read, `gets()` appends a null-terminating character (\0) to the end of the string.

`strcmp()` is a function that returns 0 if two (null-terminated) strings are the same.

(a) Explain why the use of the `gets()` function introduces a security vulnerability in the program.

(b) You think it may be possible to unlock the mainframe, even without the correct password. Provide a hexadecimal representation of an attack string that causes the `strcmp()` call to return 0, such that `unlock()` is then called. `gatekeeper()` should return normally, as to avoid raising any suspicion.
(c) The function `backdoor()` is located at address 0x0000000000000351. Construct a string that can be given to this program that will cause the function `gatekeeper()` to unconditionally transfer control to the function `backdoor()`. Provide a hexadecimal representation of your attack string.

(d) How should the program be modified in order to eliminate the vulnerabilities the function `gets()` introduces?

(e) Describe two types of protection operating systems and compilers can provide against buffer overflow attacks. Briefly explain how each protection mechanism works.
3. Caches (15 points)

For all sections, assume 512 byte, 16-bit physical address space, byte-addressable memory.

(a) Assume a 2-way set-associative, write-back cache with LRU replacement that allocates on a write-miss. Assume the cache is initially empty. Suppose we execute the following code segment which copies an array \( A[256][4] \), starting at address 0x1000 into an array \( B[256][4] \), starting at address 0x2000.

```c
char[256][4] A;
char[256][4] B;
for (i = 0; i < 256; i++) {
    for(j = 0; j < 4; j++) {
        A[i][j] = B[i][j];
    }
}
```

We know the cache miss rate for executing the code above is 1/8.

Determine block size, # of sets, # of tag bits: 

(b) Assume an initially empty, direct mapped cache. Calculate the cache miss rate for the given code and different block sizes.

```c
char[256][4] A;
char[256][4] B;
for (j = 0; j < 4; j++) {
    for(i = 0; i < 256; i++) {
        A[i][j] = B[i][j];
    }
}
```

4-byte block: ____________________________
8-byte block: ____________________________
16-byte block: ____________________________

(c) Given the code below, draw the line plot (time vs. SIZE) runtime diagram. Remember to label the axes of your plot.

```c
int array[SIZE];
int sum = 0;
for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        sum += array[j];
    }
}
```
4. Processes (10 points)

(a) After a context switch, the VPN to PPN mappings in the TLB from the previous running process no longer apply. A simple solution to this problem is to "shoot down" the TLB, by invalidating all the entries in the TLB, but this can often cause inefficiency if there is frequent context switching.

What additional information can be added to the TLB that can be utilized to reduce this inefficiency in the TLB on a context switch? *Hint: consider how processes can be uniquely identified by the MMU.*

(b) Suppose you are in control of the CPU and operating system, and you realize that you have a process A that requires a large uninterrupted chunk of CPU time to perform its important work. What would you adjust to ensure that this can happen?

(c) Consider an OS running a process A which incurs a timer interrupt at time $t_1$. The OS context switches to some other processes which do some work. Later, the OS context switches back to process A at time $t_2$. Note that process A was not run between $t_1$ and $t_2$.

Circle the items which are guaranteed to be the same at time $t_1$ and $t_2$.

- Register %rbx
- Process A’s Page Table
- Instruction Pointer
- L1 Cache
- Page fault handler code
- Page Table Base Register
(d) Consider the following C program, running on an x86-64 Linux machine. The program starts running at function main. Assume that printf flushes immediately.

```c
int main() {
    int* x = (int*) malloc(sizeof(int));
    *x = 1;
    if (fork() == 0) {
        spoon(x);
    } else {
        *x = 8 * *x;
        printf("%d\n", *x);
    }
}

void spoon(int* x) {
    printf("%d\n", *x);
    if (fork() == 0) {
        *x = 2 * *x;
    } else {
        *x = 4 * *x;
    }
    printf("%d\n", *x);
}
```

Provide **two** possible outputs of running this code.

Output 1: 

Output 2:
5. Virtual Memory (15 points)

Below is the entire physical memory of a very tiny machine that implements virtual memory. All the values are in hex, and the address of each cell is the sum of the row and column headers: for example, physical address 0x1a contains the value 0xdf. The page table is located in address 0x0, please described what format you chose for the page table entry, i.e., especially where the physical page number is.

<table>
<thead>
<tr>
<th>Address</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1a</td>
<td>aa</td>
<td>10</td>
<td>21</td>
<td>70</td>
<td>98</td>
<td>84</td>
<td>82</td>
</tr>
<tr>
<td>0x08</td>
<td>2c</td>
<td>4a</td>
<td>31</td>
<td>48</td>
<td>9e</td>
<td>02</td>
<td>11</td>
<td>0a</td>
</tr>
<tr>
<td>0x10</td>
<td>8b</td>
<td>7c</td>
<td>5a</td>
<td>92</td>
<td>8d</td>
<td>06</td>
<td>b4</td>
<td>2b</td>
</tr>
<tr>
<td>0x18</td>
<td>35</td>
<td>21</td>
<td>df</td>
<td>82</td>
<td>06</td>
<td>32</td>
<td>67</td>
<td>91</td>
</tr>
<tr>
<td>0x20</td>
<td>47</td>
<td>06</td>
<td>2a</td>
<td>02</td>
<td>89</td>
<td>18</td>
<td>ad</td>
<td>f7</td>
</tr>
<tr>
<td>0x28</td>
<td>18</td>
<td>89</td>
<td>a1</td>
<td>ff</td>
<td>47</td>
<td>3f</td>
<td>8e</td>
<td>1f</td>
</tr>
<tr>
<td>0x30</td>
<td>62</td>
<td>ef</td>
<td>00</td>
<td>11</td>
<td>1b</td>
<td>94</td>
<td>70</td>
<td>a3</td>
</tr>
<tr>
<td>0x38</td>
<td>22</td>
<td>00</td>
<td>5b</td>
<td>9a</td>
<td>0a</td>
<td>78</td>
<td>28</td>
<td>05</td>
</tr>
</tbody>
</table>

The virtual memory system uses 8-bit virtual addresses and 6-bit physical addresses. A single page is 16 bytes. Data is 1 byte-addressable. There is a fully associative TLB with 2 entries that is empty at the start of execution. It evicts according to LRU. Assume all page faults are handled properly by the OS, and pages are brought in. The page table base register is 0x00. The page table entry layout is below, where bit 0 is the least significant.

<table>
<thead>
<tr>
<th>PPN</th>
<th>unused</th>
<th>vld</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6  5</td>
<td>4  3  2  1  0</td>
</tr>
</tbody>
</table>

(a) Fill in the blanks below, explaining what happens when the following reads are executed in order. The first column is the virtual address to be read. In the second, put “Hit” or “Miss”. In the third, put the value read or “Fault” if there’s a page fault.

<table>
<thead>
<tr>
<th>Address</th>
<th>TLB</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xee</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xe4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(b) As concisely as possible, provide three benefits of a virtual memory system (in general, unrelated to the system on the previous page).

(i)

(ii)

(iii)
6. Memory Allocation (10 points)

(a) static void * searchFreeList(size_t reqSize) {
    BlockInfo* freeBlock;

    freeBlock = FREE_LIST_HEAD;
    while (freeBlock != NULL) {
        if (SIZE(freeBlock->sizeAndTags) >= reqSize) {
            return freeBlock;
        } else {
            freeBlock = freeBlock->next;
        }
    }
    return NULL;
}

Consider an allocator that uses the above function to search for a free block, allocates it, and returns exactly that block. Assume it increases the heap size as necessary.

This strategy is correct but rather naive. Describe two ways in which it could be improved to cause less fragmentation.

*Note: we want improvements that cause less fragmentation; we don’t care about runtime performance.*

(b) The strcpy(char* dest, char* src) function copies the string pointed to by src into the array pointed to by dest, including the terminating null character (and stopping at that point).

Describe two ways this code may produce a runtime error:

```c
char* copyFoo() {
    char* foo = "foo";
    char* copy = (char*) malloc(3 * sizeof(char));
    strcpy(copy, foo);
    return copy;
}
```
7. Memory Bugs (10 points)

For each code section below, briefly describe the memory bug and any security implications it may have. Also, state whether this kind of bug can occur in Java, briefly explaining why or why not.

(a) int val;
...
scanf("%d", val);

(b) void read_packet() {
      char s[8];
      int i;
      gets(s); /* reads \123456789" from stdin */
      ...
}

(c) int* search(int* p, int val) {
      while (p && *p != val)
         p += sizeof(int);
      return p;
}

(d) int* foo() {
      int val;
      return &val;
}

(e) x = (int*) malloc( N * sizeof(int) );
      // <manipulate x>
      free(x);
      ...
      y = (int*) malloc( M * sizeof(int) );
      for (i=0; i<M; i++)
         y[i] = x[i]++;

8. Pointers and Values (10 points)

Consider the C code:

```c
struct foo {
    short ss[4];
    char q;
};

struct bar {
    struct bar* next;
    char x;
    struct foo baz;
};

int main() {
    int question = 1;
    struct bar b = { // this .field notation initializes structs in
        .next = &b, // the way you would expect
        .x = 'e',
        .baz = {
            .ss = {1,2,3,4},
            .q = 'H'
        }
    };
    int answer = 42;

    // <<< BREAK >>>
    return 0;
}
```

The above C code runs until the shown BREAK. Fill in the table below with the address, value, and type of the given C expressions at that point. Answer N/A if it is not possible to determine the address or value of the expression. The first row has been filled in for you, revealing the address of `b`. Note the code was not optimized by the compiler so the variables are stored in the order they appear in the code above.

<table>
<thead>
<tr>
<th>C Expression</th>
<th>Address</th>
<th>Value</th>
<th>Type (char/short*/etc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0x1000</td>
<td>the whole struct</td>
<td>struct bar</td>
</tr>
<tr>
<td>answer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.next</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.baz.ss[2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>question</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
E. (Extra Credit) Mystery function – again :) (10 points)

What does the function below compute?

```c
int mystery(unsigned x) {
    int k;
    for (k = 0; x != 0; k++) x = x & 2*x;
    return k;
}
```

*Hint:* Try and execute the code by hand with $x = 0011\ 1111\ 1111\ 0011\ 1111\ 0011\ 1111\ 1000$ (x is in binary here).
Reference
You may tear this sheet off if you wish.

Powers of 2:

\[
\begin{align*}
2^0 &= 1 \\
2^1 &= 2 \\
2^2 &= 4 \\
2^3 &= 8 \\
2^4 &= 16 \\
2^5 &= 32 \\
2^6 &= 64 \\
2^7 &= 128 \\
2^8 &= 256 \\
2^9 &= 512 \\
2^{10} &= 1024 \\
2^{11} &= 2048 \\
2^{12} &= 4096 \\
2^{13} &= 8192 \\
2^{14} &= 16384 \\
2^{15} &= 32768 \\
2^{16} &= 65536 \\
2^{17} &= 131072 \\
2^{18} &= 262144 \\
2^{19} &= 524288 \\
2^{20} &= 1048576 \\
2^{21} &= 2097152 \\
2^{22} &= 4194304 \\
2^{23} &= 8388608 \\
2^{24} &= 16777216 \\
2^{25} &= 33554432 \\
2^{26} &= 67108864 \\
2^{27} &= 134217728 \\
2^{28} &= 268435456 \\
2^{29} &= 536870912 \\
2^{30} &= 1073741824 \\
2^{31} &= 2147483648 \\
2^{32} &= 4294967296 \\
\end{align*}
\]

Hex/decimal/binary help:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>
Assembly Code Instructions:

- **push**: push a value onto the stack and decrement the stack pointer
- **pop**: pop a value from the stack and increment the stack pointer
- **call**: jump to a procedure after first pushing a return address onto the stack
- **ret**: pop return address from stack and jump there
- **mov**: move a value between registers and memory
- **lea D(base, index, scale), dest**: compute effective address (does not load) and place in register dest. 
  \[ \text{dest} = D + (\text{base} + (\text{index} \times \text{scale})) \text{ when scale is 1,2,4,8} \]
- **add**: add src (1st operand) to dst (2nd) with result stored in dst (2nd)
- **sub**: subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
- **and**: bit-wise AND of src and dst with result stored in dst
- **or**: bit-wise OR of src and dst with result stored in dst
- **sar**: shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1st operand
- **jmp**: jump to address
- **je/jne**: conditional jump to address if zero flag is / is not set
- **js/jns**: conditional jump to address if sign flag is / is not set
- **cmp**: subtract src (1st operand) from dst (2nd) and set flags, discard result
- **test**: bit-wise AND src and dst and set flags, discard result

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, rbx, rbp, r12, r13, r14, and r15. rsp is a special register.

<table>
<thead>
<tr>
<th>%rax</th>
<th>Return Value</th>
<th>%r8</th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>Callee Saved</td>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>%r10</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>%r12</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>%r13</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
<td>%r14</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
<td>%r15</td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>
Reference from Lab 5

The functions, macros, and structs from lab5. These are all identical to those in the lab. Note that some of them will not be needed in answering the exam questions.

Structs:

```c
struct BlockInfo {
    // Size of the block (in the high bits) and tags for whether the
    // block and its predecessor in memory are in use. See the SIZE()
    // and TAG macros, below, for more details.
    size_t sizeAndTags;
    // Pointer to the next block in the free list.
    struct BlockInfo* next;
    // Pointer to the previous block in the free list.
    struct BlockInfo* prev;
};
```

Macros:

```c
/* Macros for pointer arithmetic to keep other code cleaner. Casting
to a char* has the effect that pointer arithmetic happens at the
byte granularity. */
#define UNSCALED_POINTER_ADD ...
#define UNSCALED_POINTER_SUB ...

/* TAG_USED is the bit mask used in sizeAndTags to mark a block as
used. */
#define TAG_USED 1

/* TAG_PRECEDING_USED is the bit mask used in sizeAndTags to indicate
that the block preceding it in memory is used. (used in turn for
coalescing). If the previous block is not used, we can learn the
size of the previous block from its boundary tag */
#define TAG_PRECEDING_USED 2;
```
/* SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags' field. Also, calling SIZE(size) selects just the higher bits of 'size' to ensure that 'size' is properly aligned. We align 'size' so we can use the low bits of the sizeAndTags field to tag a block as free/used, etc, like this:

sizeAndTags:
+-----------------------------------------------+
| 63 | 62 | 61 | 60 | . . . | 2 | 1 | 0 |
+-----------------------------------------------+

high bit      low bit

Since ALIGNMENT == 8, we reserve the low 3 bits of sizeAndTags for tag bits, and we use bits 3-63 to store the size.
Bit 0 \(2^0 == 1\): TAG_USED
Bit 1 \(2^1 == 2\): TAG_PRECEDING_USED

*/
#define SIZE ...

/* Alignment of blocks returned by mm_malloc. */
#define ALIGNMENT 8

/* Size of a word on this architecture. */
#define WORD_SIZE 8

/* Minimum block size (to account for size header, next ptr, prev ptr, and boundary tag) */
#define MIN_BLOCK_SIZE ...

/* Pointer to the first BlockInfo in the free list, the list's head. A pointer to the head of the free list in this implementation is always stored in the first word in the heap. mem_heap_lo() returns a pointer to the first word in the heap, so we cast the result of mem_heap_lo() to a BlockInfo** (a pointer to a pointer to BlockInfo) and dereference this to get a pointer to the first BlockInfo in the free list. */
#define FREE_LIST_HEAD ...