CSE351 Autumn 2015 – Final Exam (16 Dec 2015)

Please read through the entire examination first! We designed this exam so that it can be completed in 110 minutes and, hopefully, this estimate will prove to be reasonable.

There are 10 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.

POINTS WILL BE DEDUCTED if you are writing/erasing after the final bell has rung!

Good Luck!

Name (**as it appears on your ID**)**:**__Sample Solution____

Student Number:__________________________

UWNet ID:__________________________

Problem	**Max Score**	**Score**
1 (Potpourri)	10	
2 (Caches)	6	
3 (Caches & Structs)	12	
4 (Virtual Memory)	10	
5 (Processes)	10	
6 (Memory Allocation)	10	
7 (Java)	10	
8 (Variety Pack)	14	
9 (Assembly)	10	
10 (C Pointers & Structs)	8	
TOTAL	**100**	

1. Potpourri! True/False (10 total, 1 pt each)

	True	False
A. On a write hit, a cache that is write-back will immediately write a value from the cache back to memory.		X
B. Casting a C int into a float will lose precision. **Everyone got the point for this question. We meant to say "may" instead of "will". "may" would be true, "will" would be false since it will not always happen.**		X
C. The number of entries in a jump table for a switch statement will be equal to the number of cases listed plus one for the default case.		X
D. To maximize temporal locality, it is best to access array elements with a stride 1 access pattern.		X
E. An x86 program which uses lea instructions can be translated to a functionally equivalent version (without accounting for performance) which does not use any lea instructions.	X	
F. Increasing the associativity of a cache is the best way to improve the hit rate when accessing values from an array in order.		X
G. If we were to reverse the direction that the program stack grows, stack based buffer overflows would no longer work.		X
H. Reading memory from the heap is slower than reading from a local variable allocated on the stack.		X
I. The Java Virtual Machine reads in instructions written in Java and translates them into Java bytecodes.		X
J. In C, casting a variable from a char* to a float* will not change the bit pattern stored there.	X	

2. Caches – 6 pts

Given the following 2-way set-associative cache and its contents in a system with a 12-bit address:

Index	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7	Tag	V	B0	B1	B2	B3	B4	B5	B6	B7
0	07	1	99	1F	34	56	99	1F	34	56	11	1	DE	AD	BE	EF	DE	AD	BE	EF
1	03	1	27	A4	C5	23	00	00	00	01	1C	1	1F	2E	11	09	1F	2E	11	09
2	01	1	54	21	65	78	54	21	65	78	0F	0	CA	FE	12	34	CA	FE	12	34
3	0F	1	01	02	03	04	05	06	07	08	1C	0	12	34	56	78	13	24	57	68
4	21	1	17	C4	35	43	01	30	05	21	26	1	00	35	2A	2E	F8	E9	A1	95
5	03	1	A7	B4	D5	E3	F0	A0	B0	00	1C	1	2F	3E	44	68	2F	6E	71	55
6	02	0	27	A4	C5	23	00	40	02	01	2E	1	10	25	26	27	28	29	31	99
7	11	0	18	E4	37	73	71	08	95	22	06	1	07	34	AA	EE	FF	E5	BB	77

A. How many bits are used for the tag? **6**

B. How many bits are used for the index? **3**

C. What are the results of the following read operations (specify whether it is a hit or miss and the value if determinable from the information given, otherwise just write ND for non-determinable)? Assume the cache uses a LRU replacement policy and that reads are executed in the order given below (addresses are given in hex).

Address to be read	**Tag (give bits)**	**Set (give bits)**	**Block Offset (give bits)**	**Hit or Miss (H or M)**	**Value read (or ND)**
0x30C	***001100***	***001***	***100***	***M***	***ND***
0x1BD	***000110***	***111***	***101***	***H***	***E5***

3. Structs & Caches – 12 pts

Given an x86-64 system with a direct mapped, 8-byte block, 256 set cache:

A. How many bytes total are in this cache?

2048

B. How many bits will be required for the cache block offset?

3

C. If physical addresses are 32 bits, how many bits are in the cache tag?

32 – 3 – 8 = 21

D. Write C code that will fill every byte in this cache with characters from an array of **`sentence`** structs, each containing an **`adjective, noun,`** and **`verb:`**

```
struct sentence {
   char adjective[4];
   char noun[4];
   char verb[4];
};
typedef struct sentence sentence;
```

See the code on the next page where you should add your instructions. You can assume:

1. The array **`paragraph`** (initialized below with **`fillParagraph`**()) is a valid array of 256 **`sentence`** structs, each field of each struct is full of chars.
2. The address of the **`paragraph`** array maps to the first block of the cache.
3. All variables other than the array **`paragraph`** are stored in registers.
4. The cache is flushed (emptied) after the call to **`fillParagraph`**().

You can use the function:

```
        void access(char* c);
```

to access the cache, passing in fields from the struct.

Here are your requirements:

- **Most importantly, fill the cache without EVER accessing the characters in the noun field of ANY `sentence` struct.** Do not use pointer arithmetic or other techniques to access the chars in the `noun` field, you should bring those characters into the cache without directly accessing them.

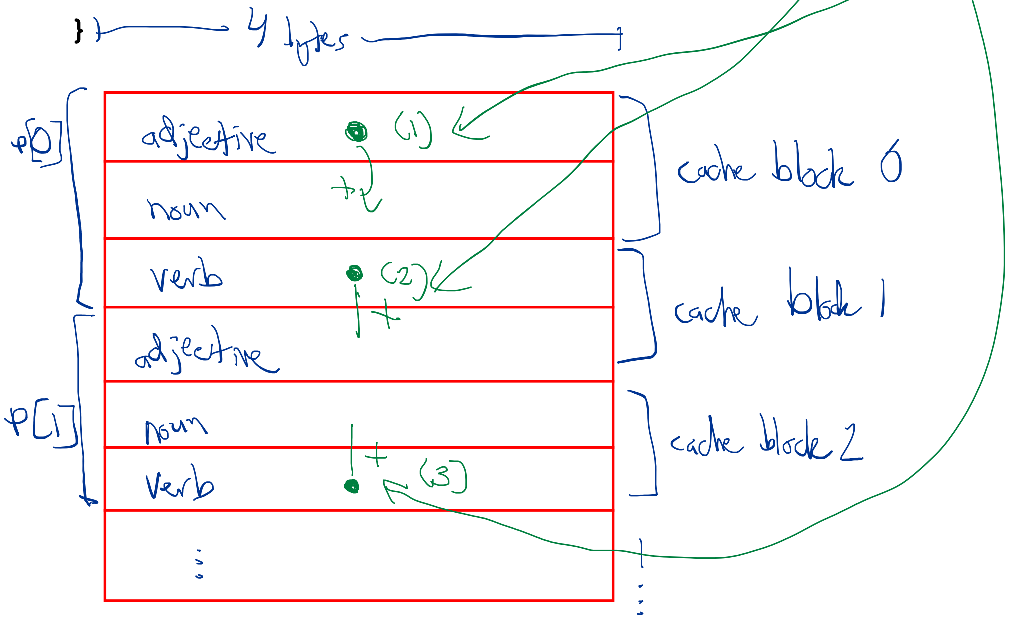
- **For full credit, your code should make as few calls to `access` as possible.**

- **For full credit, your code should stop executing once the cache is full, or the entire array has been read in, whichever comes first.**

```
void fillCache() {
   int i;
   sentence paragraph[256];

   // Fill adjective, noun, and verb fields of
   // each sentence with chars.
   fillParagraph(&paragraph);

   // Empty the cache
   flushCache();

   // YOUR CODE HERE
```

Correction: should be
if (i % 2 == 0)
(Access adjective & verb for even-numbered elements)

```
   for (i = 0; i < 170; i++) {
      ~~if (i % 2)~~ {
         access(paragraph[i].adjective); (1)
         access(paragraph[i].verb); (2)
      } else {
         access(paragraph[i].verb); (3)
      }
   }
   access(paragraph[i].adjective);

}
```



4. Virtual Memory – 10 pts
We have a system with the following properties:

- a virtual address of 18 bits,
- a physical address of 15 bits,
- pages that are 32 bytes,
- a corresponding page table, and
- a TLB with 32 entries total that is 4-way set associative.

A. How many bits will be used for the TLB tag (TT)? **10**

B. How many bits will be used for the TLB index (TI)? **3**

C. How many bits will be used for the Physical page number (PPN)? **10**

D. A page table will contain how many entries? **2^{13} = 8192**

E. Given the Virtual Address: 0x37624

Give the bits for the following:

Virtual page number **(VPN)**	TLB tag **(TT)**	TLB index **(TI)**	Physical page offset **(PPO)**
1101110110001	**1101110110**	**001**	**00100**

F. Say this is a TLB Miss, what happens next? (If more than one of these may happen next, give the one that would happen first)

a. Go to physical memory to find the page table

b. Go to the cache to find the page table

c. This is a page fault, the OS will need to do a context switch while it brings the page in from disk.

G. **True**/False – It is possible for your process and my process to both access the same physical page.

5. Processes – 10 pts

A) What is `exec()` used for? Give an example of when it is used.

`exec` replaces the current process' code with the <u>code</u> for a different <u>program</u>. It is used in the fork-exec model to get a child process to execute a program different than its parent. The example we used in class was when the user types a command like `ls` at the command line, the bash shell first forks a child process running bash and then calls `exec(ls)` in the child to get it to run the `ls` program. Note: process and program are two very different things! `exec` does not create processes, fork does that.

B) On a context switch, circle all of the following that would be saved:

TLB contents	**<u>Stack Pointer</u>**	Instruction Cache Contents	Heap Contents	**<u>Register Contents</u>**	Stack Contents	**<u>Condition Codes</u>**

The wording on this question was a little vague. Heap and Stack contents would need to be <u>preserved</u> on a context switch – a process should not lose any of these! But they do not necessarily need to be explicitly saved at the time of context switch. We counted it as o.k. if you circled heap or stack.

C) Given the following C program:

```
void sunny() {
      int n = 1;

      if (fork() == 0) {
        n = n << 1
        printf("%d, ", n);
        n = n << 1
      }
      if(fork() == 0) {
        n = n + 700;
      }
      printf("%d, ", n);
};
```

Which of the following outputs are possible for this function (circle all that apply):

a. **<u>`2, 4, 1, 701, 704,`</u>**

b. **<u>`1, 2, 4, 704, 701,`</u>**

c. **<u>`2, 704, 4, 701, 1,`</u>**

d. **<u>`701, 2, 704, 4, 1,`</u>**

e. `1, 704, 2, 4, 701,`

f. **<u>`2, 1, 704, 4, 701,`</u>**

6. Memory Allocation – 10 pts

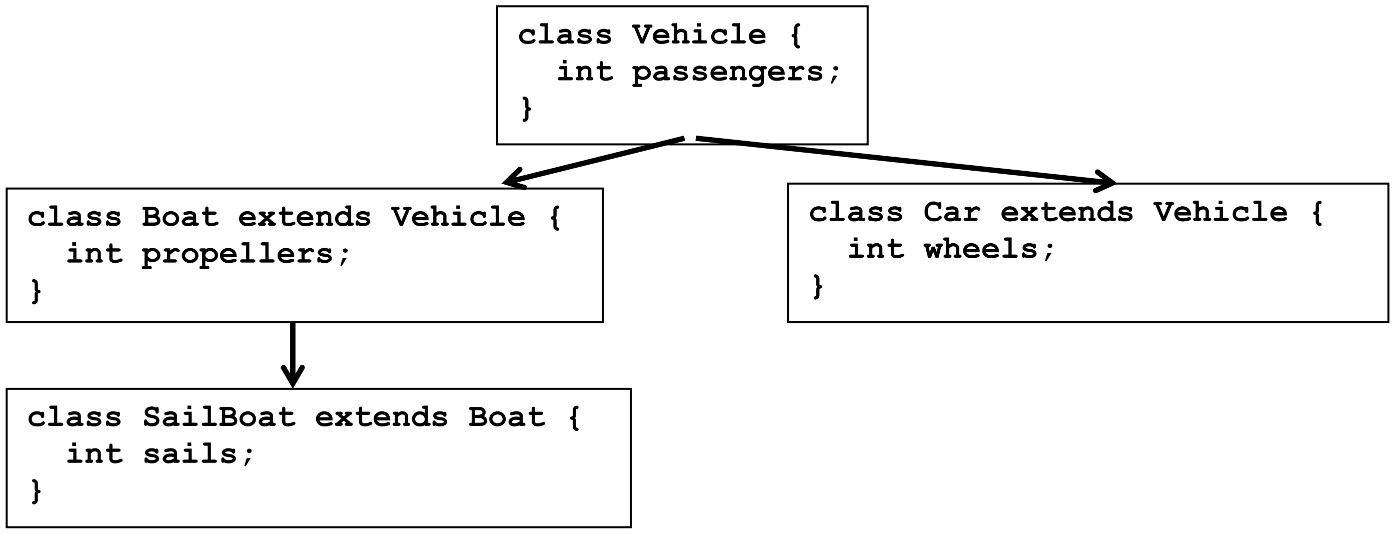
Help! Your lab5 files have been corrupted and it looks like the code may not be coalescing free blocks properly. To track down the problem, you need to implement the `findFive()` method which will find the first five blocks in your free list that might need to be coalesced with their previous block. `findFive()` scans your free list, and checks each block to see if the tag bits in its header indicate that the preceding block is free. If it finds a candidate block whose header bits indicate that its preceding block is free it will:

- Add the size of this block to a running total of the sizes.
- Add a pointer to this block to an array of pointers.
- No blocks in the free list should be modified.

`findFive()` is passed a pointer to a `FirstFive struct` that has just been allocated on the heap using `malloc` and should update it to contain the info described above. If no such blocks are found, then the `firstFiveTotalBytes` field should equal 0, otherwise it should contain the sum of the sizes of all candidate blocks found, up to a maximum of 5. Unused pointer fields do not need to be set. `findFive()` should return as soon as 5 blocks have been found. You do not need to check that a preceding block has already been added to the list, it is fine to have two neighboring blocks on your list. See lab 5 code at the end of the exam.

```
struct FirstFive {
  size_t firstFiveTotalBytes;
  struct BlockInfo* firstFivePtrs[5];
};
typedef struct FirstFive FirstFive;

static void findFive(FirstFive* result) {

   BlockInfo * curFreeBlock;
   curFreeBlock = FREE_LIST_HEAD;
   // INSERT YOUR CODE HERE. SHOULD BE 10-15 LINES.

   result->firstFiveTotalBytes  = 0;
   int index = 0;
   while (curFreeBlock != NULL) {
     // The block preceeding this free block is not used!
     if ((curFreeBlock->sizeAndTags & TAG_PRECEDING_USED) == 0) {
       result ->firstFiveTotalBytes += SIZE(curFreeBlock->sizeAndTags);
       result ->firstFivePtrs[index] = curFreeBlock;
       index++
     }
     if (index == 5)
       return;
     else
       curFreeBlock = curFreeBlock->next;
  }




   return;
}
```

7. Java – 10 pts
Given the following Java class hierarchy:



```
class Vehicle {
  int passengers;
}
```

```
class Boat extends Vehicle {
  int propellers;
}
```

```
class Car extends Vehicle {
  int wheels;
}
```

```
class SailBoat extends Boat {
  int sails;
}
```

And the following additional code:

```
class FinalExam {

     public static void main(String[] args) {
          Car      c   = new Car();      // line 1
          Boat     b1  = new Boat();     // line 2
          Boat     b2  = new Vehicle(); // line 3
          Vehicle  v   = new SailBoat();// line 4
          SailBoat sb1 = (SailBoat) b1; // line 5
          SailBoat sb2 = (SailBoat) v;  // line 6
     }
}
```

Circle all of the items below that will be true:

i. Line 3 will cause a compiler error.

ii. Line 4 will cause a compiler error.

iii. Line 5 will cause a compiler error.

iv. Line 6 will cause a compiler error.

v. Line 3 will cause a run-time error.

vi. Line 4 will cause a run-time error.

vii. Line 5 will cause a run-time error.

viii. Line 6 will cause a run-time error.

ix. Each object will have a copy of the vtable for that class. **(objects share the same vtable)**

x. Given that no constructor exists for the Car class, the initial value of `c.wheels` is unknown. **(fields are initialized to null/zero)**

xi. Variable `b1` will be on the heap. **(What b1 refers to will be on the heap, b1 will be in a register or on the stack.)**

8. Variety Pack – 7 pts this page, 14 pts total

A. (3 pts) Given

```
int a = 0x0A;
int b = 0x10;
```

What do the following expressions evaluate to (write your answer in hex):

```
  i.   a ^ b        ___0x1A_________________
 ii.   a + b        ___0x1A _________________
iii.   a | b        ___0x1A _________________
```

B. (2 pts) What will this function print?

```
void func() {
   int p[6] = {0,1,0,3,0,3};
   if ( *(&p[3]) == *(p + 5) ) {
         printf("true");
   } else {
         printf("false");
   }
}
```

Circle one:

[✓ `true`]   `false`   Compiler Error   Run-time Error

C. (2 pts) Given the two vegetarian structs shown below:

```
struct whopper {
   int buns[2];
   double lettuce;              O
   short tomato;
   int sauce;
};
struct big_mac {
   short buns[2];
   int lettuce;            X
   char* pickles;
   short cheese;
};
```

Draw a **circle** next to the burger with the most **internal fragmentation**.
Draw an **x** next to the burger with the most **external fragmentation**.
If there is a tie, put the mark next to both of them. If one burger has the most of both types of fragmentation, put both marks there.

8. (cont.) 7 pts this page

D. (1 pt) With garbage collection, who identifies objects that you are no longer using and frees them?

Programmer   Compiler   **Language (Java) Runtime**   Operating System

E. (1 pt) In C, who determines whether an array is allocated on the stack or the heap?

Programmer   Compiler   Language (C) Runtime   Operating System

F. (1 pt) Who determines what physical page a virtual page maps to?

Programmer   Compiler   Language (C, Java) Runtime   **Operating System**

G. (2 pts) In C, given a multidimensional array of char* a[4][5], if a starts at address 0, at what byte address is the element a[3][1]? (Give your address in decimal)

128

H. (2 pts) If **`%rax`** contains 2 and **`%rdx`** contains 5, what will be in **`%rax`** after the following instruction is executed (If this contains an error or the value cannot be determined say so):

```
leaq 2(%rax,%rdx,4), %rax
```

24

9. C to Assembly (10 points)

Given the following C function:

```
long snowy(long *a, long i, long max){
  long result = *a;
  while (i < max) {
    result = result * 4;
    i++;
  }
  return result;
}
```

Write **x86-64** bit assembly code for this function here. Comments are not required but could help for partial credit. We are not judging you on the efficiency of your code, just the correctness. It is fine to leave off the size suffixes if you prefer to (e.g. b, w, l, q). We have already filled in part of the code for you below – which you should use. Feel free to add other labels as needed.

```
  snowy:

            movq (%rdi), %rax
            jmp  .check


while_loop:

            salq $2, %rax
            addq $1, %rsi
   check:
            cmpq %rdx, %rsi


            jl <while_loop>
            ret
```

10. Pointers, arrays and structs (8 points)

Given the following declarations and code in C, assuming an x86-64 system:

```
typedef struct foo {
     int *x;
     char y[4];
     double z;
} foo;

foo* a;
foo b;
a = &b;
```

Fill in the following table. If you cannot tell what the expression evaluates to, write "UNKNOWN".

C Expression	Evaluates to?	Resulting data type
`a`	`0x10000000`	`foo*`
`b.y`	`0x10000008`	`char*`
`a->z`	`UNKNOWN`	`double`
`&(b.z)`	`0x10000010`	`double*`
`a->x`	`UNKNOWN`	`int*`
`&(a->y[1])`	`0x10000009`	`char*`
`b.y[2]`	`UNKNOWN`	`char`
`*a`	`UNKNOWN` (+ see below)	`foo`
`&(b.y[6])`	`0x1000000e`	`char*`

+ ("b" was also accepted for this one although this was not really what we were looking for)

REFERENCES

Powers of 2:

2^0 = 1	
2^1 = 2	2^{-1} = .5
2^2 = 4	2^{-2} = .25
2^3 = 8	2^{-3} = .125
2^4 = 16	2^{-4} = .0625
2^5 = 32	2^{-5} = .03125
2^6 = 64	2^{-6} = .015625
2^7 = 128	2^{-7} = .0078125
2^8 = 256	2^{-8} = .00390625
2^9 = 512	2^{-9} = .001953125
2^{10} = 1024	2^{-10} = .0009765625

Hex Help:

0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7
0x8	1000	8
0x9	1001	9
0xa	1010	10
0xb	1011	11
0xc	1100	12
0xd	1101	13
0xe	1110	14
0xf	1111	15

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`	compute effective address and store in a register
`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
`sal`	shift data in the dst to the left (arithmetic shift) by the number of bits specified in 1st operand
`shl`	shift data in the dst to the left (logical shift) by the number of bits specified in the 1st operand
`jmp`	jump to address
`jg`	conditional jump to address if greater than
`jle`	conditional jump to address if less than or equal
`jne`	conditional jump to address if zero flag is not set
`cmp`	subtract src (1st operand) from dst (2nd) and set flags
`test`	bit-wise AND src and dst and set flags

Suffixes for `mov` instructions:

`s` or `z` for sign-extended or zero-ed, respectively

Suffixes for all instructions:

`b`, `w`, `l`, or `q` for byte, word, long, and quad, respectively

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.

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

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:

```
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:

```
/* 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 …
```