Name:_____

CSE351 Winter 2016, Final Examination
March 16, 2016

Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 4:20.**
- There are **125 (not 100) points**, distributed **unevenly** among **11** questions (most with multiple parts):
- **The exam is printed double-sided.**
- The last two pages of the exam have useful reference information.

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty and some in the middle can be done fairly quickly. **Skip around and make sure you get to all the questions.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. *C data and locality* (**10** points) We consider two versions of C type definitions for a collection of student names and exam scores:

```
// Version A
struct Student {
  char * name;
  int exam_score;
};
struct ExamScores {
  int numStudents;
  struct Student * students; // array of length numStudents
};

// Version B: uses 2 arrays instead of 1, where name and score for
// each student is (implicitly) at the same array index
struct ExamScores {
  int numStudents;
  char ** names; // array (of strings) of length numStudents
  int * exam_scores; // array of length numStudents
};
```

Here is correct "version A" code for a function `get_score` that looks up a student's score by name. Note `strcmp` is defined in the C standard library for comparing strings (it returns 0 for equal strings).

```
int get_score(char * name, struct ExamScores * scores) {
  for(int i = 0; i < scores->numStudents; i++) {
     if(strcmp(name,scores->students[i].name)==0)
       return scores->students[i].exam_score;
  }
  return -1; // special value for name-not-found
}
```

   (a) Rewrite `get_score` for "Version B."

   (b) For a large number of students, would you expect "version A" or "version B" to get better performance for calls (with various names) to `get_score`? *Explain your answer.*

Name:_____

2. *Caching* (**18** points) For this problem, assume:

   - 64-bit addresses
   - A single-level 128KB data cache
   - A 32-byte cache-line size
   - The cache "starts empty."
   - The local variables below are all in registers.

   Consider this C code:

```
void f(int * x, int * y, int n) {
  for(int i=0; i < n; i++) {
     x[i] = y[i] / 2;
  }
}
```

   (a) Suppose the cache is direct-mapped.

      i. Which bits in the addresses of `&x[0]` and `&y[0]` would have to be the same for these addresses to map to the same (1-element) set in the cache? Draw a picture if you need to make clear which bits you mean.

      ii. If `&x[0]` and `&y[0]` do map to the same (1-element) cache set (and, for simplicity, have an offset of 0 in the cache line), what would you expect the cache miss rate to be for an execution of `f`?

      iii. If `&x[0]` and `&y[0]` do not map to the same (1-element) cache set (and, for simplicity, have an offset of 0 in the cache line), what would you expect the cache miss rate to be for an execution of `f`?

   (b) Now suppose the cache is 4-way set-associative.

      i. Which bits in the addresses of `&x[0]` and `&y[0]` would have to be the same for these addresses to map to the same (4-element) set in the cache? Draw a picture if you need to make clear which bits you mean.

      ii. If `&x[0]` and `&y[0]` do map to the same (4-element) cache set (and, for simplicity, have an offset of 0 in the cache line), what would you expect the cache miss rate to be for an execution of `f`?

      iii. If `&x[0]` and `&y[0]` do not map to the same (4-element) cache set (and, for simplicity, have an offset of 0 in the cache line), what would you expect the cache miss rate to be for an execution of `f`?

Name:_____

3. *Caching and Locality* (**10** points) Consider this C code:

```
void f(int * a, int * b, int n) {
  for(int i=0; i < n; i++) {
     a[i] = b[i] / 2;
  }
  for(int i=0; i < n; i++) {
     a[i] += 7;
  }
}
```

(a) Rewrite the function so that it (1) performs the same computations but in a different order, (2) has better temporal locality than before, and (3) has no worse spatial locality than before.

(b) Suppose you did the same rewriting for a Java method (where in Java `a` and `b` would have type `int[]`). In Java would this provide:

- a much larger increase in locality,
- a much smaller increase in locality, or
- about the same increase in locality?

(c) In the C version, your rewriting is probably *wrong* for certain unusual argument values. Give an example where the code above and your answer to part (a) update memory differently. Notes:

- This is a tricky question not worth many points.
- Hint: In Java, there are no such examples.

Name:_____

4. *Processes* (**12** points) In this problem, assume Linux.

   (a) Can the same program be executing in more than one process simultaneously?

   (b) Can a single process change what program it is executing?

   (c) When the operating system performs a context switch, what information does *NOT* need to be saved/maintained in order to resume the process being stopped later (circle all that apply):

   - The page-table base register
   - The value of the stack pointer
   - The time of day (i.e., value of the clock)
   - The contents of the TLB
   - The process-id
   - The values of the process' global variables

   (d) Give an example of an exception (asynchronous control flow) in which it makes sense to later re-execute the instruction that caused the exception.

   (e) Give an example of an exception (asynchronous control flow) in which it makes sense to abort the process.

Name:_____

5. *Fork* (**5** points) Consider this code using Linux's `fork`:

```
int x = 7;
if(fork()) {
   x++;
   printf(" %d ", x);
   fork();
   x++;
   printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

What are *all* the different possible outputs (order of things printed) for this code? (Hint: There are four of them.)

Name:_____

6. *Protection* (**10** points) We saw several different ways to prevent a running program from doing something, including:

   A. A software check inserted by the compiler that is performed when the program executes (by having extra instructions/work in the code)

   B. A check performed by the compiler itself (failing compilation if the check fails)

   C. A hardware check performed directly by the hardware when the program executes (where the earlier decision of what is allowed may be set by the program and/or the operating system, but then the hardware does the checking)

   D. Simply providing no assembly instruction / operands that can perform the operation

   For each of the following, answer (A), (B), (C), or (D) for the most common way to prevent it:

   (a) Executing code that is on the call-stack

   (b) A process accessing the (private) memory of another process

   (c) A process accessing a portion of virtual memory it has not been given by the operating system

   (d) Branching (i.e., taking a conditional jump) based on whether a memory access hits or misses in the cache

   (e) A buffer overflow in Java

Name:_____

7. *Virtual Memory* (**19** points)

   (a) Can a memory access ever be a TLB hit and cause a page fault?

   (b) Can a memory access ever be a TLB miss and cause a page fault?

   (c) Assume a page size of 64KB, a TLB with 128 entries, 4-way set associativity in the TLB, and a direct-mapped single-level cache of size 128KB. What are the maximum number of distinct addresses (in bytes) that a program could access in-between TLB misses? That is, during a sequence on TLB hits with no misses, how many different bytes of memory could possibly be accessed. (Hint: You do not need all the information given. Hint: We are definitely not assuming the TLB starts empty.)

   (d) Consider a memory location ℓ when accessing ℓ results in a TLB hit. For each of the following, answer "all", "some", or "none."

      i. How many of the bits for ℓ's virtual page number are in the TLB?

      ii. How many of the bits for ℓ's physical page number are in the TLB?

      iii. How many of the bits for ℓ's virtual page offset are in the TLB?

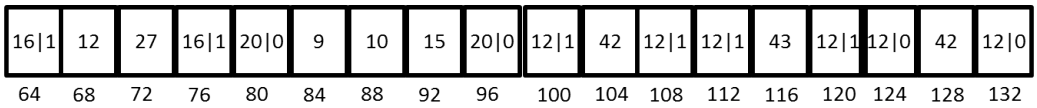
      iv. How many of the bits for ℓ's physical page offset are in the TLB?

   (e) Why does the operating system switch to running a different program after a page fault? (Answer in at most two sentences.)

   (f) Why is writing to the page-table base register something that only the operating system (not "regular" code in a process) should be allowed to do? (Answer in at most two sentences.)

Name:_____

8. *Memory Allocation* (**11** points) Consider the (tiny) heap below using an implicit free-list allocator with coalescing, where each rectangle represents 4 bytes and a header or boundary tag of the form `x|y` indicates a block of size `x` (in base-10) where `y` being `1` means allocated. Addresses (in base-10) are written below the rectangles. Unlike your Lab 5 allocator, the allocator for this heap uses boundary tags for allocated blocks.



(a) Suppose the next call to the allocator is `free(104)`. Show the state of the heap after this call completes by indicating in the picture of the heap above what addresses have different contents and what the new contents are.

(b) Suppose the next allocator call (after part (a)) is `malloc(4)`. Under a *first-fit policy*, what address would the call to `malloc` return?

(c) Suppose the next allocator call (after part (a)) is `malloc(4)`. Under a *best-fit policy*, what address would the call to `malloc` return?

(d) Given your answer to part (a) (i.e., after doing this free), what is the smallest `z` such that `malloc(z)` would not succeed unless the allocator expanded the size of the heap?

Name:_____

9. *Memory Bugs* (**13** points) Consider this buggy C code involving a singly-linked list of ints:

```
struct IntList {
  int i;
  struct IntList * next;
};

// insert x at beginning of list and return pointer to new head of list
struct IntList * add_at_front(int x, struct IntList * lst) {
  struct IntList * ans = (struct IntList*)malloc(sizeof(struct IntList*));
  ans->next = lst;
  ans->i = x;
  return lst;
}

// deallocate all list nodes (using recursion)
void free_entire_list(struct IntList * lst) {
  free(lst->i);
  free(lst);
  free_entire_list(lst->next);
}
```

(a) `add_at_front` has two bugs. Rewrite the function to fix both.

(b) `free_entire_list` has two bugs. Rewrite the function to fix both.

(c) Consider the memory allocator from Lab 5. Which of the four bugs you fixed, while still a bug in C, would happen not to cause problems when using this memory allocator? Explain your answer.

Name:_____

10. *C vs. Java* (**11** points) Consider this Java code (left) and somewhat similar C code (right) running on x86-64:

```
public class Foo {
  private int[] x;
  private int y;
  private int z;
  private Bar b;
  public Foo() {
     x = null;
     b = null;
  }
}
```

```
struct Foo {
  int x[6];
  int y;
  int z;
  struct Bar * b;
};

struct Foo * make_foo() {
  struct Foo * f = (struct Foo *)malloc(sizeof(struct Foo));
  f->x = NULL;
  f->b = NULL;
  return f;
}
```

(a) In Java, `new Foo()` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `Foo`'s fields? (Do *not* include space for any header information, vtable pointers, or allocator data.)

(b) In C, `malloc(sizeof(struct Foo))` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `struct Foo`'s fields? (Do *not* include space for any header information or allocator data.)

(c) The function `make_foo` attempts to be a C variant of the `Foo` constructor in Java. One line fails to compile. Which one and why?

(d) What, if anything, do we know about the values of the `y` and `z` fields after Java creates an instance of `Foo`?

(e) What, if anything, do we know about the values of the `y` and `z` fields in the object returned by `make_foo`?

Name:_____

11. *vtables* (**6** points)

    (a) In an implementation of Java, can a vtable for a subclass ever have more entries than a vtable for a superclass? Explain in 1–2 English sentences.

    (b) In an implementation of Java, can a vtable for a subclass ever have the same number of entries as a vtable for a superclass? Explain in 1–2 English sentences.

    (c) In an implementation of Java, can a vtable for a subclass ever have fewer entries than a vtable for a superclass? Explain in 1–2 English sentences.

REFERENCES

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = .5$
$2^2 = 4$	$2^{-2} = .25$
$2^3 = 8$	$2^{-3} = .125$
$2^4 = 16$	$2^{-4} = .0625$
$2^5 = 32$	$2^{-5} = .03125$
$2^6 = 64$	$2^{-6} = .015625$
$2^7 = 128$	$2^{-7} = .0078125$
$2^8 = 256$	$2^{-8} = .00390625$
$2^9 = 512$	$2^{-9} = .001953125$
$2^{10} = 1024 = 1KB$	$2^{-10} = .0009765625$
$2^{20} = \ldots = 1MB$	
$2^{30} = \ldots = 1GB$	

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

Some Lab 5 definitions: These are identical to those in Lab 5. You don't need them all.

```
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;
};

/* Size of a word on this architecture. */
# define WORD_SIZE sizeof(void*)

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

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

/* 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 ((x) & ~(ALIGNMENT - 1))

/* 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;
```