CSE351 Spring 2018, Final Exam
June 6, 2018

Please do not turn the page until 2:30.

Last Name:	
First Name:	
Student ID Number:	
Name of person to your left:	
Name of person to your right:	
Signature indicating: All work is my own. I had no prior knowledge of the exam contents nor will I share contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade.	

Rules:

- The exam is closed-book, closed-note, etc. except for one 8.5 x 11 inch sheet of paper.
- The last page contains a reference page that was also posted in advance. *Please remove this last piece of paper and do not turn it in.*
- **Please stop promptly at 4:20.**
- There are **125 (not 100!) points**, distributed **unevenly** among **8** questions (all with multiple parts):
- **The exam is printed double-sided.**

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. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**21** points) (Arrays) Consider these C function definitions and assume x86-64 and Linux:

```
void g(int x[4][3],int *y[4]) {
  for(int i=0; i < 4; i++)
    for(int j=0; j < 3; j++)
        x[i][j] = y[i][j];
}

void f() {
   int a[4][3] = {{0,0,0},{0,0,0},{0,0,0},{0,0,0}};
   int r1[3] = {1,2,3};
   int r2[3] = {4,5,6};
   int r3[3] = {7,8,9};
   int * b[4] = {r1,r2,r3,r1};

   g(a,b);
}
```

(a) In `f`, how many bytes of stack space does each local-variable occupy? (Ignore any padding between variables.)

i. `a`

ii. `r1`

iii. `r2`

iv. `r3`

v. `b`

(b) Assume in `g` that `x`, `y`, `i`, and `j` are kept in registers and no other registers are used (except the stack pointer of course). When `f` calls `g`, how many times does the execution of `g` access a memory location in `f`'s stack frame?

(c) Answer *always*, *sometimes*, or *never* for each of the following:

i. When `f` calls `g`, `x[0] < x[3]`.

ii. When `f` calls `g`, `y[0] < y[3]`.

iii. For some other arbitrary unshown call to `g`, `x[0] < x[3]`.

iv. For some other arbitrary unshown call to `g`, `y[0] < y[3]`.

(d) For each of the following, answer *legal* if this would be a well-defined assignment inside `g` and *illegal* otherwise.

i. `x[1][-1] = 0;`

ii. `y[1][-1] = 0;`

iii. `x[1] = x[2];`

iv. `y[1] = y[2];`

Solution:

(a)
 i. 48 bytes
 ii. 12 bytes
 iii. 12 bytes
 iv. 12 bytes
 v. 32 bytes

(b) 36 (3 for each loop iteration: 1 for the multidimensional array and 2 for the multi-level array)

(c)
 i. always
 ii. never
 iii. always
 iv. sometimes

(d)
 i. legal
 ii. illegal (this is perhaps a poorly-worded question – it's illegal under the assumption that `y[1]` points to the beginning of an array)
 iii. illegal
 iv. legal (this is perhaps a poorly-worded question – it's legal under the assumption that `y` points to an array of at least three pointers)

Name:____

2. (**12** points) (Struct Layout) Assume x86-64 and Linux and that all fields should be properly aligned.

   (a) Consider this `struct` definition:

   ```
   struct S {
     int x;
     int * p;
   };
   ```

   Do all of the following:

   - Indicate what `sizeof(struct S)` would evaluate to.
   - Draw the layout of the struct, indicating the size and offset of each field.
   - For any padding, indicate whether it is in *internal* or *external* fragmentation.

   (b) Repeat the previous problem for this `struct` definition:

   ```
   struct S {
     int x;
     int * p;
     int y;
   };
   ```

   (c) Repeat the previous problem for this `struct` definition:

   ```
   struct S {
     int * p;
     int x;
     int y;
   };
   ```

Solution:

(a) size is 16 bytes;
x at offset 0 for 4 bytes,
then padding for 4 bytes (internal fragmentation),
then p at offset 8 for 8 bytes

(b) size is 24 bytes;
x at offset 0 for 4 bytes,
then padding for 4 bytes (internal fragmentation),
then p at offset 8 for 8 bytes,
then y at offset 16 for 4 bytes,
then padding for 4 bytes (external fragmentation)

(c) size is 16 bytes,
p at offset 0 for 8 bytes,
then x at offset 8 for 4 bytes,
then y at offset 12 for 4 bytes.
(No padding.)

Name:____

3. (**18** points) (Memory Hierarchy) For this problem, assume these memory-hierarchy parameters, some of which are not relevant:

- A single-level, direct-mapped cache with 32KiB of data and 256B blocks. Assume this cache is used for data and not for instructions.
- Virtual memory with 64KiB pages and an 8-entry fully-associative TLB
- 32-bit virtual addresses and 24-bit physical addresses

(a) Assume the cache and TLB start "cold," which means all entries are invalid. Let `a` point to (i.e., hold the virtual address of) a 4-byte aligned array of 32-bit ints. Suppose we access `a[0]`, which will be a cache miss and a TLB miss, then we access `a[1]`, and then we access `a[2]`.

For each row of this table, give, in hex or binary, one possible pointer value in `a` that would lead to the described behavior — or answer "impossible" if there is no such address for that row.

Pointer value in `a`	Access of `a[1]`	Access of `a[2]`
	Cache hit	Cache hit
	Cache hit	Cache miss
	Cache miss	Cache hit
	Cache miss	Cache miss
	Cache hit and TLB miss	Cache hit and TLB hit
	Cache miss and TLB miss	Cache hit and TLB hit

(b) Continuing the previous problem, of all 4-byte-aligned virtual addresses, what *fraction* of them lead to both `a[1]` and `a[2]` being cache hits as in the first row of the table.

(c) Now assume `a` holds 16Ki ints (i.e., 64KiB of data) and starts at address 0xFFFF0000 and that we execute assembly corresponding to this C code assuming `sum` and `i` are in registers. Notice the C code has two loops (which is silly in practice).

```
int i, sum=0;
for(i=0; i < (1 << 14); i++)
   sum += a[i];
for(i=0; i < (1 << 14); i++)
   sum += a[i];
```

i. What is the cache miss rate for this code? (You can answer with a fraction.)

ii. Does doubling the size of the cache to 64KiB with no other changes make the miss rate for this code *better*, the *same*, or *worse*?

iii. Does making the cache 4-way set-associative with LRU replacement and no other changes make the miss rate for this code *better*, the *same*, or *worse*?

iv. Does doubling the size of the block size to 512B with no other changes make the miss rate for this code *better*, the *same*, or *worse*?

Solution:

(a)

Virtual address of `a`	Access of `a[1]`	Access of `a[2]`
Any 4-byte-aligned 32-bit address where the low 8 bits are *not* 11111000 or 11111100, i.e., not F8 or FC.	Cache hit	Cache hit
Any 32-bit address where the low 8 bits are 11111100 (i.e., F8)	Cache hit	Cache miss
Any 32-bit address where the low 8 bits are 11111100 (i.e., FC)	Cache miss	Cache hit
Impossible	Cache miss	Cache miss
Impossible	Cache hit and TLB miss	Cache hit and TLB hit
Any 32-bit address where the low 16 bits are 11111111 11111100 (i.e., FFFC)	Cache miss and TLB miss	Cache hit and TLB hit

(b) 62/64 (i.e., 31/32). (Explanation: 64 4-byte-aligned positions in a 256B cache block and 2 of them don't work)

(c) i. 1/64

ii. better

iii. same

iv. better

Name:____

4. (**13** points) (Processes and Fork)

Recall a *context switch* is when the operating system switches which process is executing. For each of the following system changes, answer *more* if the change would cause the operating system and/or hardware to do more work to complete a context switch. Otherwise, answer *same* meaning the same amount of work would be needed. No explanations required.

(a) Adding more physical address space

(b) Adding more entries to the TLB

(c) Adding more registers to the ISA

(d) Adding more instructions to the ISA

(e) Adding more entries to the interrupt vector table

(f) Switching from a single-level to a multi-level page table

Describe what the following code would print when executed. Hint: Do not write out what it would print.

(g)
```
printf("A");
for(int i=0; i < 10; i++) {
  fork();
}
printf("B");
```

Solution:

(a) Same
(b) More
(c) More
(d) Same
(e) Same
(f) Same
(g) It would print `A` once and then `B` 1024 times.

Name:____

5. (**14** points) (Virtual Memory)

   (a) Assume these virtual-memory parameters:
   - 1 GiB physical address space
   - 4 GiB virtual address space
   - 16 KiB page size
   - A TLB with 2-way set associativity and 16 total entries

   i. How many bits will be used for the page offset?

   ii. How many bits will be used for the virtual page number (VPN)?

   iii. How many bits will be used for the physical page number (PPN)?

   iv. How many bits will be used for the TLB index?

   v. How many bits will be used for the TLB tag?

   (b) Virtual memory holds instructions and stack data and heap data.

   i. For pages holding instructions, do we expect the number of page faults for most programs to be low due to *temporal locality*, *spatial locality*, *both*, or *neither*?

   ii. For pages holding stack data, do we expect the number of page faults for most programs to be low due to *temporal locality*, *spatial locality*, *both*, or *neither*?

   iii. For pages holding heap data, do we expect the number of page faults for most programs to be low due to *temporal locality*, *spatial locality*, *both*, or *neither*?

   (c) A summer intern is working on hardware design for the ChippyMcChipFace processor. On this processor, after the OS handles a page fault, the re-executed instruction suffers from a TLB miss. The intern proposes a somewhat complex mechanism to avoid this TLB miss and argues it will improve performance. Explain in roughly one sentence why implementing this change is not good engineering.

Solution:

(a) i. 14
ii. 18
iii. 16
iv. 3
v. 15

(b) i. both
ii. both
iii. both

(c) A page fault takes tens of thousands of times longer than a TLB miss — avoiding the cost of one TLB miss per page fault is insignificant so not worth any extra complexity.

Name:____

6. (**13** points) (Using Dynamic Memory) Consider these C `struct` and function definitions:

```
struct A {                       struct B {                       struct C {
  double arr[4];                   int x;                           int x;
  int * p;                         char y;                          char y;
};                                 struct A a;                      struct A * a;
                                 };                               };
```

```
struct B* make_a_new_b() {
  struct B * ans = (struct B*)malloc(sizeof(struct B));
  ans->x = 0;
  ans->y = 'A';
  for(int i=0; i<4; i++)
    ans->a.arr[i] = 0.0;
  ans->a.p = NULL;
  return ans;
}
struct C* make_a_new_c() {  /* see part (a) */ }
```

(a) Complete the body of `make_a_new_c` so that the `struct C` object pointed to by the returned value has all the same initialized data as the object created by `make_a_new_b`. Your solution should have two calls to `malloc` instead of one and a few other small changes from `make_a_new_b`.

Of course, normally we use allocated objects before deallocating them, but the code fragments below attempt to deallocate immediately after allocation. For each fragment, indicate one of the following, no explanations necessary:

- *correct* if the allocated memory is deallocated correctly with no space leaks
- *leak* if the code would compile and execute correctly but would cause a space leak
- *no compile* if the code would not compile
- *illegal execution* if the code would compile but running it could cause a memory bug

(b)
```
struct B * foo = make_a_new_b();
free(foo);
```

(c)
```
struct C * foo = make_a_new_c();
free(foo);
```

(d)
```
struct B * foo = make_a_new_b();
free(foo->a);
free(foo);
```

(e)
```
struct C * foo = make_a_new_c();
free(foo->a);
free(foo);
```

Solution:

(a)
```
struct C* make_a_new_c() {
  struct C * ans = (struct C*)malloc(sizeof(struct C));
  ans->x = 0;
  ans->y = 'A';
  ans->a = (struct A*)malloc(sizeof(struct A));
  for(int i=0; i<4; i++)
    ans->a->arr[i] = 0.0;
  ans->a->p = NULL;
  return ans;
}
```

(b) correct

(c) leak

(d) no compile

(e) correct

Name:____

7. (**23** points) (Dynamic Memory Allocator) This problem uses a strange and limited implementation of `malloc` and `free` that works for x86-64. The code is below. You do *not* need to understand all the code before you start answering the questions on the next page. The allocator works as follows:

   - The total heap size is set when the heap is initialized and never grows or shrinks.
   - All blocks are 128B. They are never split or coalesced.
   - Unless the heap is full, `malloc` requests for $\leq$ 128B succeed. Larger requests never succeed.
   - There is no allocator data (e.g., boundary tags, free-list pointers, etc.) in the heap itself. Instead, a separate area of contiguous memory (pointed to by `all_alloc_bits`) is used to track which blocks are currently allocated.
   - Call the block that `heap_start` points to block 0, then the next (i.e., adjacent) block 1, and so on. Then the i^{th} ***bit*** of the array pointed to by `all_alloc_bits` should be be 1 if and only if block i is currently allocated. (This is not exactly precise for the code below due to endianness, but you don't really need to think about endianness — just treat the code below as correct and, when asked, complete the blanks in a consistent fashion so the allocator works correctly.)

```
#define HEAP_SIZE (1<<25)
#define BLOCK_SIZE (1<<7)
#define BITS_IN_BYTE 8
int num_blocks = HEAP_SIZE / BLOCK_SIZE;
long * all_alloc_bits;
char * heap_start;

void heap_init() {
  heap_start = ... // ptr to HEAP_SIZE bytes acquired from the OS
  all_alloc_bits = ... // ptr to num_blocks / BITS_IN_BYTE bytes acquired from the OS
  for(int i=0; i < ((num_blocks / BITS_IN_BYTE) / sizeof(long)); i++)
    all_alloc_bits[i] = 0;
}
void* malloc(size_t n) { // we assume heap_init was already called exactly once
  if(n > BLOCK_SIZE)
    return NULL; // block too big
  for(int i=0; i < (num_blocks / sizeof(long)); i++)  // for each long of allocated bits
    if(all_alloc_bits[i] !=  ~ 0L) { // the L just means the 0 has type long. Also note the ~
      for(int j=63; j >= 0; j--) { // for each bit in this long
        unsigned long next_bit = 1L << j;
        if(!(all_alloc_bits[i] & next_bit)) {
          all_alloc_bits[i] = all_alloc_bits[i] | next_bit;
          return heap_start + (64 * i + (63 - j)) * BLOCK_SIZE;
        }
      }
    }
  return NULL; // heap full
}
void free(void* p) { // zero out the "is allocated bit" for the block p points to
  int block_num = ((char*)p - heap_start) / ________________;
  int i = block_num / 64;
  int j = _____________ % _____________;
  unsigned long bit = (1UL << 63) >> j; // the UL just means the 1 has type unsigned long
  all_alloc_bits[i] = ____________ & ______________;
}
```

Name:____

(a) How many *bytes* are in the heap? (Ignore the space for is-allocated bits.)

(b) How many *blocks* are in the heap?

(c) Is this allocator *first-fit* or *next-fit*? Explain in roughly one sentence.

(d) Explain in roughly one sentence why it doesn't make sense to ask if this allocator is *best-fit*.

(e) Consider the line `if(all_alloc_bits[i] != ~ 0L) {` in `malloc`. If we replace this with `if(1) {` (where recall 1 is "true"), does the allocator still work correctly? If so, explain why this line is there anyway. If not, explain why not.

(f) Fill in the five blanks in `free` so that `free` works correctly:

```
int block_num = ((char*)p - heap_start) / _________________;
int i = block_num / 64;

int j = ________________ % _______________;
unsigned long bit = (1UL << 63) >> j;

all_alloc_bits[i] = _____________________ & ______________;
```

In the remaining questions, assume `p` has type `int*` and holds a pointer previously returned by `malloc` that has not yet been passed to `free`. In other words, it points to the beginning of an allocated block.

(g) In general, it is a bug for a program to call `free(p+1)`. What would actually happen for this allocator?

(h) In general, it is a bug for a program to call `free(p); free(p)` (i.e., free it twice in a row). What would actually happen for this allocator?

(i) In general, it is a bug for a program to call `free` with a pointer to a stack-allocated variable. What would actually happen for this allocator?

Solution:

(a) 2^{25}, i.e., 32MiB

(b) 2^{18}, i.e., 256Ki

(c) It is *first-fit*, the loops in `malloc` always look for an unallocated block starting at the beginning of the heap.

(d) All blocks are the same size, never split nor coalesced, so all unallocated blocks are equally good fits.

(e) Yes, it still works correctly. This is an optimization that checks if the 64 allocated bits in a long are all set, in which case we can move on to the next long without the inner loop.

(f)
```
int block_num = ((char*)p - heap_start) / BLOCK_SIZE;
int i = block_num / 64;
int j = block_num % 64;
unsigned long bit = (1UL << 63) >> j;
all_alloc_bits[i] = all_alloc_bits[i] & ~ bit;
```

(g) It would behave exactly like `free(p)`, i.e., it would free the block pointed into. (Reason: integer division in `int block_num = ((char*)p - heap_start) / BLOCK_SIZE;`).

(h) It would behave exactly like one call to `free(p)`, i.e., nothing bad would happen. (Reason: all free does is unset the allocated bit. Provided the block hasn't been reallocated yet, unsetting a bit twice is no problem.)

(i) We will try to unset some wrong bit in memory, hopefully leading to a segmentation fault but potentially corrupting some other data. (Reason: `int block_num = ((char*)p - heap_start) / BLOCK_SIZE;`) will produce a block number that is way too big, causing `all_alloc_bits[i]` to be way out of bounds.)

Name:____

8. (**11** points) (Java)

   (a) For each course topic below that we studied using C, answer *yes* if the topic is *directly relevant* in Java as well (else answer *no*).

   i. Floating-point operations often produce small rounding errors that can compound over many operations.

   ii. Pointer arithmetic is scaled by the size of the pointed-to object.

   iii. Using uninitialized data can lead to garbage results that depend on whatever happened to be in that memory previously.

   iv. Keeping your working set small helps improve the performance of memory operations in general, without concern for the exact parameters of a machine's memory hierarchy.

   (b) Some of the safety checks that are performed in Java but not in C require extra data in memory, i.e., fields that are part of Java data but not part of the analogous C data. For each of the following, answer *yes* if Java needs such extra data to perform the operation (else answer *no*).

   i. Throwing an `ArrayIndexOutOfBoundsException` if an array index is too large.

   ii. Throwing a `NullPointerException` if the `e` in `e.m()` evaluates to `null`.

   iii. Throwing a `ClassCastException` if the `e` in `(Foo)e` does not evaluate to an instance of `Foo`.

   (c) Consider this Java code, which is part of a larger program.

```
class Foo {
  int x;
  boolean sameAsX(int y) { return y == this.x; }
  boolean sameAs7(int y) { return y == 7; }
}
class Bar {
  boolean whyNotBoth(Foo f, int z) {
     return f.sameAsX(z) && f.sameAs7(z);
  }
}
```

   Your friend suggests that when compiling the method call `f.sameAs7(z)` above, the compiler can optimize out the instruction that passes `this` as a procedure argument since the `sameAs7` method in `Foo` does not use it. Explain in roughly 1–2 sentences why this "optimization" is wrong.

Solution:

(a) i. yes
   ii. no
   iii. no
   iv. yes

(b) i. yes
   ii. no
   iii. yes

(c) A subclass of `Foo` could override `sameAs7` with a method that uses the `this` pointer and the first argument to `whyNotBoth` could be an instance of this subclass. With the optimization, the callee will not have the proper `this` pointer it needs.

This page intentionally blank. You can use it if you need more room than you have on another page, but please indicate on the other page to look here! Write, "see extra pages!"

This page intentionally blank. You can use it if you need more room than you have on another page, but please indicate on the other page to look here! Write, "see extra pages!"