CSE 351

Final Exam Review
Final Exam Review

• The final exam will be comprehensive, but more heavily weighted towards material after the midterm

• We will do a few problems from previous years’ finals together as a class
  • PLEASE ask questions if you get lost!
Quiz

- We have another quiz we want to spend a few minutes on
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.

False
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
   False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.

False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.

True
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.

   False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.

   True

3. Shifting an int by 3 bits to the left (<< 3) is the same as multiplying it by 8.
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
   False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.
   True

3. Shifting an int by 3 bits to the left (<< 3) is the same as multiplying it by 8.
   True
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
   False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.
   True

3. Shifting an int by 3 bits to the left (<< 3) is the same as multiplying it by 8.
   True

4. In C, endianness makes a difference in how character strings (char*) are stored.
Quiz

1. A 4-byte integer can be moved into a 32-bit register using a movw instruction.
   False

2. On a 64-bit architecture, casting a C integer to a double does not lose precision.
   True

3. Shifting an int by 3 bits to the left (<< 3) is the same as multiplying it by 8.
   True

4. In C, endianness makes a difference in how character strings (char*) are stored.
   False
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.
Quiz (cont)

5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True
Quiz (cont)

5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.

True
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.

True

7. An instruction cache takes advantage of only spatial locality.
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.

True

7. An instruction cache takes advantage of only spatial locality.

False
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.

True

7. An instruction cache takes advantage of only spatial locality.

False

8. Caches are part of the instruction set architecture (ISA) of a computer.
5. In C, storing multi-dimensional arrays in row major order makes it possible for pointer arithmetic to determine the address of an array element.

True

6. A struct can’t have internal fragmentation if the elements of the struct are ordered from largest to smallest.

True

7. An instruction cache takes advantage of only spatial locality.

False

8. Caches are part of the instruction set architecture (ISA) of a computer.

False
9. Caches make computers slower by getting between the CPU and memory.
Quiz (cont)

9. Caches make computers slower by getting between the CPU and memory.

False
Quiz (cont)

9. Caches make computers slower by getting between the CPU and memory.

False

10. On a 64-bit architecture, if a cache block is 32 bytes, and there are 256 sets in the cache, the tag will be 53 bits.
9. Caches make computers slower by getting between the CPU and memory.

False

10. On a 64-bit architecture, if a cache block is 32 bytes, and there are 256 sets in the cache, the tag will be 53 bits.

False
Quiz (cont)

9. Caches make computers slower by getting between the CPU and memory.

False

10. On a 64-bit architecture, if a cache block is 32 bytes, and there are 256 sets in the cache, the tag will be 53 bits.

False

11. A process’s instructions are typically in a read-only segment of memory.
Quiz (cont)

9. Caches make computers slower by getting between the CPU and memory.

False

10. On a 64-bit architecture, if a cache block is 32 bytes, and there are 256 sets in the cache, the tag will be 53 bits.

False

11. A process’s instructions are typically in a read-only segment of memory.

True
12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.
Quiz (cont)

12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True
Quiz (cont)

12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.
12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.

True
12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.

True

14. Two running instances of the same process share the same memory address space.
12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.

True

14. Two running instances of the same process share the same memory address space.

False
12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.

True

14. Two running instances of the same process share the same memory address space.

False

15. Java generally has better performance than C.
Quiz (cont)

12. A shared library can be accessed from multiple virtual address spaces, but with only one copy in physical memory.

True

13. Virtual memory allows programs to act as if there is more physical memory than there actually exists on the computer.

True

14. Two running instances of the same process share the same memory address space.

False

15. Java generally has better performance than C.

False
Processes

• List the two important illusions that the process abstraction provides to programs.
• For each illusion, list a mechanism involved in its implementation.
Processes

• List the two important illusions that the process abstraction provides to programs.
• For each illusion, list a mechanism involved in its implementation.

• 1. Logical control flow: the process executes as if it has complete control over the CPU. The OS implements this by interleaving execution of different processes via context-switching (exceptional control flow...).
• 2. Private linear address space: the process executes as if it has access to a private contiguous memory the size of the virtual address space.
Virtual Memory

- One purpose of virtual memory is to allow programs to use more memory than is available in the physical memory by storing some parts on disk transparently. Name some other useful thing that can be done with the virtual memory system.
Virtual Memory

- One purpose of virtual memory is to allow programs to use more memory than is available in the physical memory, by storing some parts on disk transparently. Name some *other* useful things that can be done with the virtual memory system.

- **1.** Sharing of a single physical page in multiple virtual address spaces (e.g., shared library code).
- **2.** Memory protection mechanisms (e.g., page-granular read/write/execute permissions or protecting one process’s memory from another).
TLBs

• Does a TLB (Translation Lookaside Buffer) miss always lead to a page fault? Why or why not?
TLBs

• Does a TLB (Translation Lookaside Buffer) miss always lead to a page fault? Why or why not?

• No. The TLB caches page table entries. After a TLB miss, we do an in-memory page table lookup. A page fault occurs if the page table entry is invalid.
Java vs C

• Name some differences between Java references and C pointers.
Java vs C

- Name some differences between Java references and C pointers.

  1. *C allows pointer arithmetic; Java does not.*
  2. *C pointers may point anywhere (including the middles of memory objects); Java references point only to the start of objects.*
  3. *C pointers may be cast arbitrarily (even to non-pointer types); casts of Java references are checked to make sure they are type-safe.*
Let's look at a program which includes the definition for a data structure type:

typedef struct data_struct {
    int a;
    int *b;
    int c;
} data_struct;

This is a small snippet of code for a function foo, which has just been called and in turn calls print_struct:

```c
int foo() {
    data_struct x;
    int n = 13;
    x.a = ???;
    x.b = &n;
    x.c = 3;
    print_struct(&x);
}
```

Definition of the print_struct function:

```c
void print_struct(data_struct *y) {
    printf("%p\n", y);
    printf("%d\n", *(y->b + y->c));
    <<execution is suspended here>>
}
```
• Execution is suspended after the printf statements in `print_struct` but before it returns to `foo`.
• The stack at this point of the execution of the program is shown below in 4-byte blocks.
• Note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address:

```
0x7fffffff0000001f: 0x00000000
0x7fffffff00000028: 0x00000003
0x7fffffff00000030: 0x00000004
0x7fffffff00000034: 0x00000005
0x7fffffff00000038: 0x00000006
0x7fffffff0000003c: 0x00000007
0x7fffffff00000040: 0x00000008
0x7fffffff00000041: 0x00000009
0x7fffffff00000042: 0x0000000a
0x7fffffff00000043: 0x0000000b
0x7fffffff00000044: 0x0000000c
0x7fffffff00000045: 0x0000000d
```

Note:
- 0x00203748
- 0x00000000
- 0x00000001
- 0x00000002
- 0x00000003
- 0x00000004
- 0x00000005
- 0x00000006
- 0x00000007
- 0x00000008
- 0x00000009
- 0x0000000a
- 0x0000000b
- 0x0000000c
- 0x0000000d
Stacks and Structs

• Execution is suspended after the printf statements in print_struct but before it returns to foo.
• The stack at this point of the execution of the program is shown below in 4-byte blocks
• Note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address:

```
0x7fffffffffffffffa040: 0x00203748
0x7fffffffffffffff03c: 0x00000001
0x7fffffffffffffff038: 0x0000015f
0x7fffffffffffffff034: 0x00000000
0x7fffffffffffffff030: 0x00402741
0x7fffffffffffffff02c: 0x00000000
0x7fffffffffffffff028: 0x00000003
0x7fffffffffffffff024: 0x7fffffff
0x7fffffffffffffff020: 0x00000000
0x7fffffffffffffff01c: 0x00000000
0x7fffffffffffffff018: 0x00000000
0x7fffffffffffffff014: 0x00000000
0x7fffffffffffffff010: 0x00000000
0x7fffffffffffffff00c: 0x00402053
```

• What is the value stored in the stack at the 8-bytes starting at location 0x7fffffffffffffff00c to 0x7fffffffffffffff013 and what does it represent?
Stacks and Structs

- Execution is suspended after the printf statements in print_struct but before it returns to foo.
- The stack at this point of the execution of the program is shown below in 4-byte blocks.
- Note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address:

  0x7fffffffffffffff040: 0x00203748
  0x7fffffffffffffff03c: 0x00000001
  0x7fffffffffffffff038: 0x0000015f
  0x7fffffffffffffff034: 0x00000000
  0x7fffffffffffffff030: 0x00402741
  0x7fffffffffffffff02c: 0x00000000
  0x7fffffffffffffff028: 0x00000003
  0x7fffffffffffffff024: 0x7fffffff
  0x7fffffffffffffff020: 0xffffa014
  0x7fffffffffffffff01c: 0x00000000
  0x7fffffffffffffff018: 0x00000007
  0x7fffffffffffffff014: 0x0000000d
  0x7fffffffffffffff010: 0x00000000
  0x7fffffffffffffff00c: 0x00402053

- What is the value stored in the stack at the 8-bytes starting at location 0x7fffffffffffffff00c to 0x7fffffffffffffff013 and what does it represent?

- 0x00000000000402053 which represents the return address to be used when print_struct returns to foo.

- Remember endian-ness!
- Erratum: In actuality we would expect the return address to be 8-byte aligned, but here it is not (0x7ff...a00c)

  << high order bytes of return address from print_struct
  << low order bytes of return address from print_struct
Stacks and Structs

- Execution is suspended after the printf statements in print_struct but before it returns to foo.
- The stack at this point of the execution of the program is shown below in 4-byte blocks
- Note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address:

```
0x7fffffffffffffff040:  0x00203748
0x7fffffffffffffff03c:  0x00000001
0x7fffffffffffffff038:  0x0000000f
0x7fffffffffffffff034:  0x00000000
0x7fffffffffffffff030:  0x00402741
0x7fffffffffffffff02c:  0x00000000
0x7fffffffffffffff028:  0x00000003
0x7fffffffffffffff024:  0x7fffffff
0x7fffffffffffffff020:  0xfffa014
0x7fffffffffffffff01c:  0x00000000
0x7fffffffffffffff018:  0x00000007
0x7fffffffffffffff014:  0x0000000d
0x7fffffffffffffff010:  0x00000000
0x7fffffffffffffff00c:  0x00402053
```

- What value was assigned to x.a in the function foo and at what address is it stored on the stack?
## Stacks and Structs

- Execution is suspended after the printf statements in print_struct but before it returns to foo.
- The stack at this point of the execution of the program is shown below in 4-byte blocks.
- Note that the stack is shown as is tradition, from bottom to top, with the top-most of the stack at the bottom or lowest address:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffff000000fa040</td>
<td>0x00203748</td>
<td>&lt;&lt; padding (external fragmentation), offset +20</td>
</tr>
<tr>
<td>0x7fffffff000000fa03c</td>
<td>0x00000001</td>
<td>&lt;&lt; x.c, offset +16</td>
</tr>
<tr>
<td>0x7fffffff000000fa038</td>
<td>0x0000015f</td>
<td>&lt;&lt; high order bytes of x.b</td>
</tr>
<tr>
<td>0x7fffffff000000fa034</td>
<td>0x00000000</td>
<td>&lt;&lt; low order bytes of x.b, offset +8</td>
</tr>
<tr>
<td>0x7fffffff000000fa030</td>
<td>0x00402741</td>
<td></td>
</tr>
<tr>
<td>0x7fffffff000000fa02c</td>
<td>0x00000000</td>
<td>&lt;&lt; padding (internal fragmentation)</td>
</tr>
<tr>
<td>0x7fffffff000000fa028</td>
<td>0x00000003</td>
<td></td>
</tr>
<tr>
<td>0x7fffffff000000fa024</td>
<td>0x7fffffff</td>
<td></td>
</tr>
<tr>
<td>0x7fffffff000000fa020</td>
<td>0xfffffa014</td>
<td>&lt;&lt; low order bytes of x.b, offset +8</td>
</tr>
<tr>
<td>0x7fffffff000000fa020</td>
<td>0xfffffa014</td>
<td>&lt;&lt; low order bytes of x.b, offset +8</td>
</tr>
<tr>
<td>0x7fffffff000000fa01c</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7fffffff000000fa018</td>
<td>0x00000007</td>
<td>&lt;&lt; x.a, offset +0</td>
</tr>
<tr>
<td>0x7fffffff000000fa014</td>
<td>0x0000000d</td>
<td>&lt;&lt; int n = 13</td>
</tr>
<tr>
<td>0x7fffffff000000fa010</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7fffffff000000fa00c</td>
<td>0x00402053</td>
<td></td>
</tr>
</tbody>
</table>
**Structs**

typedef struct data_struct {
    int a;
    int *b;
    int c;
} data_struct;

Take a look at `struct_test.c`