CPSC 213, Winter 2013, Term 2 — Final Exam **Solution**

Date: April 14, 2014; Instructor: Mike Feeley (Solutions written by Maher Kader)

1 (5 marks) **Variables and Memory.** Consider the following C code with three global variables, `a`, `b`, and `c`, that are stored at addresses `0x1000`, `0x2000`, `0x3000`, respectively.

```
void foo() {
    a[0] = 1;
    b[0] = 2;
    c = a;
    c[0] = 3;
    c = b;
    *c = 4;
}
```

```
int a[1];
int b[1];
int* c;
```

Describe what you know about the content of memory following the execution of `foo()` on a 32-bit **Big Endian** processor. List only memory locations whose address and value you know. List each byte of memory separately using the form "`byte_address: byte_value`". List all numbers in hex.

0x1000: 0x00
0x1001: 0x00
0x1002: 0x00
0x1003: 0x03

0x2000: 0x00
0x2001: 0x00
0x2002: 0x00
0x2003: 0x04

0x3000: 0x00
0x3001: 0x00
0x3002: 0x20
0x3003: 0x00

You can approach this question by tracking how memory changes line by line. You can do this in your head on the exam instead of writing everything out.

Upon entering `foo()`:
0x1000: 0x00000000 `(int a[1])`
0x2000: 0x00000000 `(int b[1])`
0x3000: 0x00000000 `(int* c)`
`a[0] = 1;`
0x1000: 0x00000001
0x2000: 0x00000000
0x3000: 0x00000000
`b[0] = 2;`
0x1000: 0x00000001
0x2000: 0x00000002
0x3000: 0x00000000
`c = a;`
0x1000: 0x00000001
0x2000: 0x00000002
0x3000: 0x00001000
`c[0] = 3;`
0x1000: 0x00000003
0x2000: 0x00000002
0x3000: 0x00001000
`c = b;`
0x1000: 0x00000003
0x2000: 0x00000002
0x3000: 0x00002000
`*c = 4;`
0x1000: 0x00000003
0x2000: 0x00000004
0x3000: 0x00002000

Then translate everything to the byte-by-byte format as requested in the question. Remember it's Big Endian in this case! (what would the answer be if it was Little Endian instead?)

2 **(8 marks)** **Global Variables.** Consider the following C declaration of global variables.

```
int a;
int *b;
int c[10];
```

Recalling that `&` is the C *get address* operator, which of the following can be computed statically? Justify your answers.

2a `&b`

Statically known. The compiler decides the address of global variables.

2b `&b[4]`

Not statically known. The compiler doesn't know the address of an element in a dynamic array.

`&b[4] = &*(b + 4) = b + 4`. The compiler doesn't know the value of `b`

2c `&c[4]`

Statically known. The compiler chooses where a globally declared array is located and thus knows the address of any of its elements.

Now answer this question.

2f Give the assembly code the compiler would generate for "`c[a] = *b;`". Use labels for static values.

```
ld $c, r0        # r0 = &c[0]
ld $a, r1        # r1 = &a
ld 0(r1), r1     # r1 = a
ld $b, r2        # r2 = &b
ld 0(r2), r2     # r2 = b
ld 0(r2), r2     # r2 = *b
st r2, (r0, r1, 4) # c[a] = *b

# static data area (not required in answer)
.pos 0x1000
a: .long 0
b: .long 0
c: .long 0 # c[0]
   .long 0 # c[1]
   .long 0 # c[2]
   .long 0 # c[3]
   .long 0 # c[4]
   .long 0 # c[5]
   .long 0 # c[6]
   .long 0 # c[7]
   .long 0 # c[8]
   .long 0 # c[9]
```

3 **(8 marks)** **Instance Variables and Local Variables.** Consider the following C declaration of global variables.

```
struct X {
    int a;
    int b;
};
struct X c;
struct X* d;
```

Which of the following can be computed statically? Justify your answers.

3a `&c.a`

Address of member of a statically allocated is statically known.

3b `&d->a`

Not statically known. Need to get the value of the pointer which is only known at runtime to compute that address.

3c `(&d->a) - (&d->b)`

The difference in offset between two struct members is always statically known regardless of where the struct is stored.

Now answer this question.

3d Give the assembly code the compiler would generate for "`d->b = c.b;`".

```
ld $d, r0        # r0 = &d
ld 0(r0), r0     # r0 = d
ld $c, r1        # r1 = &c
ld 4(r1), r1     # r1 = c.b
st r1, 4(r0)     # d->b = c.b

# data areas (not required in answer)
# static data area
.pos 0x1000
c: .long 0       # c.a
   .long 0       # c.b
d: .long 0x2000

# heap
.pos 0x2000
d_data: .long 0 # d->a
        .long 0 # d->b
```

4 (10 marks) **Write Assembly Code.** Give the assembly code the compiler would generate to implement the following C procedure, assuming that arguments are passed on the stack. Just this procedure. Use labels for static values. Comment every line of your code.

```
int computeSum (int* a) {
    int sum=0;
    while (*a>0) {
        sum = add (sum, *a);
        a++;
    }
    return sum;
}
```

```
.pos 0x100
computeSum: deca r5          # allocate space on stack for return address
            st r6, 0(r5)     # store return address on stack
            ld $0, r0        # r0 = 0 (sum)
            ld 4(r5), r1     # r1 = a (argument passed on stack)
loop_guard: ld 0(r1), r2     # r2 = *a
            bgt r2 loop_body # goto loop_body if *a > 0
            br loop_end      # executed if the above branch condition fails
loop_body:  deca r5          # allocate arg 2
            deca r5          # allocate arg 1
            st r0, 0(r5)     # arg 1 = sum
            st r2, 4(r5)     # arg 2 = *a
            gpc $6, r6       # r6 = return address
            j add            # call add(sum, *a)
                             # return value is in r0 so sum already = add(...)
            inca r5          # deallocate arg 2
            inca r5          # deallocate arg 1
            inca r2          # a++
            br loop_guard
loop_end:   ld 0(r5), r6     # r6 = return address
            inca r5          # deallocate return address
            j 0(r6)          # return sum (sum is already in r0)
```

You don't have to copy the way I've commented the code verbatim, I'm slightly more descriptive than necessary for teaching purposes. But don't forget to comment your code, and it helps to be explicit about which registers represent which variables for your own sake, and the grader's.

We allocate the return address on the stack at the beginning because `computeSum` is not a leaf procedure (it calls another function `add(...)`).

The way I've written the loop is just one of the ways you could have done it. I prefer translating loops/if statements as closely as possible to the code's structure (unless explicitly asked to do otherwise) because I find it easier to think about the code and convince myself about its correctness. So I structure my conditional branch to represent the *true* case of the loop guard. Again, this is personal preference and you wouldn't lose marks for structuring your loop differently.

There are some subtleties here with how I chose r0 to be the variable `sum`. I could have stored it on the stack, but it's easier to store it in registers when writing assembly under exam conditions. This should be acceptable as long as the question doesn't explicitly ask you to store local variables on the stack. Since `add(...)` stores the return value in r0, when the function returns, it's automatically assigned to be `sum`'s value! Note that if the statement was instead something like `sum = sum + add(...)` then I couldn't do this, because I would have to save `sum` to another register before it gets overwritten so I could add it to the result. Having `sum` in r0 also means that I don't have to do any extra work of preparing r0 before returning at the end of `computeSum`.

Finally, don't forget that `a` is a pointer, so `a++` does pointer arithmetic instead of just +1.

5 **(12 marks)** **Read Assembly Code.** Consider the following SM213 code.

```
X:  deca  r5              #  allocate space for return address on stack
    deca  r5              #  allocate local variable on stack (x)
    st    r6, 4(r5)       #  store return address on stack
    ld    $0, r1          #  r1 = 0 (i)
    st    r1, 0(r5)       #  local var x = 0
    ld    12(r5), r2      #  r2 = arg 2 (b)
    ld    16(r5), r3      #  r3 = arg 3 (c)
    not   r3              #  r3 = ~c (bitwise NOT)
    inc   r3              #  r3 = -c
L0: mov   r1, r4          #  r4 = i
    add   r3, r4          #  r4 = i - c
    beq   r4, L2          #  goto L2 if i - c == 0, ie i == c
    bgt   r4, L2          #  goto L2 if i - c > 0, ie i > c
    ld    (r2,r1,4), r4   #  r4 = b[i]
    deca  r5              #  allocate space for arg (to pass) on stack
    st    r4, 0(r5)       #  arg (to pass) = b[i]
    gpc   $2, r6          #  r6 = return address
    j     *12(r5)         #  call a(b[i]) where a is arg 1 that we received
    inca  r5              #  deallocate arg off stack
    ld    $1, r4          #  r4 = 1
    and   r0, r4          #  r4 = a(b[i]) & 1
    beq   r4, L1          #  goto L1 if a(b[i]) & 1 == 0
    ld    0(r5), r4       #  r4 = x
    add   r0, r4          #  r4 = x + a(b[i])
    st    r4, 0(r5)       #  x = r4, ie x += a(b[i])
L1: inc   r1              #  i++
    br    L0              #  goto L0
L2: ld    0(r5), r0       #  r0 = x
    ld    4(r5), r6       #  r6 = return address
    inca  r5              #  deallocate local variable
    inca  r5              #  deallocate return address
    j     (r6)            #  return x
```

5a Add a comment to every line of code. Where possible use variables names and C pseudo code in your comments to clarify the connection between the assembly code and corresponding C statements.

If you do this part of the question well, the next section is trivial. The important thing to do here is to identify every single variable that is in the code and assign it a name so you can figure out what the code is trying to do. Every variable here can be classified as one of the following (the names can be whatever you like, as long as they're consistent):

1. Local variables on the stack (`int x`) and in registers (`int i`)
2. Arguments passed to X (`int (*a)(int)`, `int* b`, `int c`)
3. Arguments passed by X to whatever it calls (`b[i]`)

The most difficult one to identify is the variable `i` stored in r1. It's initialized to 0, but then also saved on the stack *to initialize another variable* which we call `x` here. You know that `x` is a local variable because it's saved to the stack and not near a function call. We know that `i` is its own variable because r1 goes on to have a value other than what's stored in `x`.

It is extremely important to be aware of the layout of the stack throughout the execution of the program. I'd highly recommend drawing out the stack so that when you see load/store offsets, you can figure out exactly which variable it's referring to. If you're not sure how I determined that `b` and `c` are arguments 2 and 3 (instead of 1 and 2), draw the stack. This will also help you figure out that the instruction `j *12(r5)` is reading a function pointer off the stack.

Notice how in the comments, I always try to refer to values by their variable name rather than register name (as the question asks me to do). This will help immensely in the next step. Also notice how I've commented and rearranged the conditional branches - this will also help you when translating the code to C.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

```
int X(int (*a)(int), int* b, int c) {
  int x = 0;
  for (int i = 0; i < c; i++) {
    if (a(b[i]) & 1)
        x += a(b[i]);
  }
  return x;
}
```

Pay attention to how the loop guard `i < c` was derived from the assembly code - the assembly code said `goto L2 if i >= c` meaning "end the loop if `i >= c`". The opposite of this is then "only enter the loop if `i < c`". This is one possible way the loop could have been written. If you wanted to translate the assembly code exactly as you saw it, this is what you would get:

```
int i = 0;
while (1) {
  if (i - c == 0) break;
  if (i - c > 0) break;
  if (a(b[i]) & 1)
    x += a(b[i]);
  i++;
}
```

You probably wouldn't lose marks for this, but I think the way it's written in the answer is more likely to be the original C code. Learn to identify common looping patterns like iterating from 0 to some number. At this point, you may want to even rename your variables to show that you really understand what the code is doing. Something like this:

```
int X(int (*fn)(int), int* array, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (fn(array[i]) & 1)
      sum += fn(array[i]);
  }
  return sum;
}
```

You don't have to name every variable like this in an exam. I do this here just to show you exactly what the code is doing. However if you do choose to use good variable names, it will help you convince yourself that you've translated the code correctly. It should be clear now that this code is applying the given function to all odd elements in the array and summing the results. (`n & 1 == 1` implies `n` is odd).

6 (3 marks) **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

Is there a bug in this code? If so, carefully describe what it is.

Yes there is a bug in this code. The function does not performing bounds checking on the array `b` and we are not given its size, so this could modify anything in memory.

7 **(6 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}

void three () {
    int* ret = one();
    two();
}
```

7a Is there a bug in this code? If so, carefully describe what it is.

Yes. `one()` returns a dangling pointer to a local variable. Local variables have undefined values after the function returns, because other functions will use the same spot on the stack when they're called.

7b What is the value of "`*ret`" at the end of `three`? Explain carefully.

`*ret = 2` because when `two()` is called, it allocates `zot` at the same place where `loc` used to be allocated and overwrites its value.

This question requires you to make an assumption that all local variables are stored on the stack (as opposed to registers, or optimized away by the compiler). Now you must be comfortable with visualizing the stack and how it changes over time.

Below I have drawn what the stack might look like after just entering the function `three()`. Unknown values like return addresses I denote as ... but of course they would actually have some numeric value: the caller's memory address to return to.

address	value	meaning
0x1000	0	unallocated
0x1004	0	unallocated
0x1008	0	unallocated
0x100C	0	local var `ret`
0x1010	...	return address to `three()`'s caller

Now `three()` calls `one()`. Before the line `return &loc;` executes, this is what the stack would look like. Pay attention to the address of `loc` because that is what `one()` will return.

address	value	meaning
0x1000	0	unallocated
0x1004	1	local var `loc`
0x1008	...	return address to `three()`
0x100C	0	local var `ret`
0x1010	...	return address to `three()`'s caller

Now when `one()` returns and before we call `two()`, the stack will be as follows. **Notice how elements on the stack are unallocated but retain their values**. What would the value of `*ret` be at this point?

address	value	meaning
0x1000	0	unallocated
0x1004	1	unallocated
0x1008	...	unallocated
0x100C	0x1004	local var `ret`
0x1010	...	return address to `three()`'s caller

Below is the stack when we enter `two()`. It shows you why `*ret = 2` when we return to `three()`.

address	value	meaning
0x1000	0	unallocated
0x1004	2	local var `zot`
0x1008	...	return address to `three()`
0x100C	0x1004	local var `ret`
0x1010	...	return address to `three()`'s caller

8 (12 marks) Static and Dynamic Procedure Calls.

8a Procedure calls in C are normally static. Method invocations in Java are normally dynamic. Carefully explain the reason why Java uses dynamic method invocation and what benefit this provides to Java programs.

Java's method invocations are dynamic because they read a jump table at runtime to determine which method to call. Which method is called depends on the actual type of the object, since each class has its own jump table.

Dynamic method invocation allows for polymorphic dispatch to occur. Polymorphism is a powerful tool makes it easy to add functionality to existing code by simply extending a class and overriding methods.

8b Carefully explain an important disadvantage of dynamic invocation in Java or other languages.

Every method call has the additional overhead of performing a memory read to determine which method to call. This can affect performance, especially for very short methods where a memory read might consist of a significant amount of the method's execution time.

8c Demonstrate the use of function pointers in C by writing a procedure called `compute` that:

1. has three arguments: a non-empty array of integers, the size of the array, and a function pointer;
2. computes either the array min or max depending only on the value of the function pointer argument;
3. contains a `for` loop, no `if` statements, and one procedure call (per loop).

Give the C code for `compute`, the two procedures that it uses (i.e., that are passed to it as the value of the function-pointer argument), and two calls to `compute`, one that computes min and the other that computes max (be sure to indicate which is which).

```
int compute(int* array, int size, int (*fn)(int, int)) {
    int acc = array[0];
    for (int i = 1; i < size; i++) {
        acc = fn(acc, array[i]);
    }
    return acc;
}

int max(int a, int b) {
  return a > b ? a : b;
}

int min(int a , int b) {
  return a < b ? a : b;
}

int arr[5] = {7, -4, 1, 9, 3};

compute(arr, 5, max); // returns max of array
compute(arr, 5, min); // returns min of array
```

The first thing you should do when asked this type of question is to figure out what your function signature should look like - how many arguments does it have, what types are they, and what type is the return value? Also give your parameters meaningful names.

Make sure you think about **edge cases**. If we weren't told that the array is non-empty, we wouldn't be able to do `int acc = array[0];` without potentially segfaulting or causing undefined behavior.

Search for 'ternary operator' if you're not sure about the syntax of the expressions in `max` and `min`. Knowing shorthand coding tricks like this will save you time when writing code in an exam.

9 **(6 marks)** **Switch Statements.** There are two ways to implement `switch` statements in machine code. For purposes of this question, lets call them A and B.

9a Describe A, very briefly.

A sequence of `if` statements.

9b Describe B, very briefly.

A jump table of labels corresponding to switch-cases.

9c State precisely one situation where A would be preferred over B and why.

If there are very few cases to consider, then the overhead of using a jumptable is higher than a few statements. Reading memory is much slower than executing a conditional branch.

OR: If the case values are spread apart (sparsely populated), there will be a lot of wasted memory in the jumptable because we have to represent a contiguous range of values in a jump table. e.g.

```
switch (i) {
  case 1:
    j = 5;
    break;
  case 1000000:
    j = 10;
    break;
 }
```

would require a jump table with one million elements!

9d State precisely one situation where B would be preferred over A and why.

When you have lots of cases to check and their values are close together (densely populated), the jump table is the best choice. When there are N cases, it takes `O(N)` to test all cases, whereas it will always take `O(1)` with a jump table. If the values are closer together, then we waste less memory creating the jump table.

10 **(9 marks)** **IO Devices.** Three key hardware features used to incorporate IO Devices with the CPU and memory are Programmed IO (PIO), Direct Memory Access (DMA) and interrupts.

10a Carefully explain the difference between PIO and DMA; give one advantage of DMA.

The CPU uses PIO to read or write to an IO device one word at a time. IO Devices use DMA to read or write memory directly, without involving the CPU. An advantage of PIO is that the CPU can use it to transfer data to an IO device, or control it; e.g., to signal the IO device that the CPU wants something, Another advantage is that PIO has lower overhead and lower latency for very small transfers because it avoids the overhead of setting up a DMA. The advantage of DMA is that the transfer occurs asynchronously to the CPU and so the CPU is free to do other things during the transfer. For any transfer larger than 64-128 bytes, DMA typically transfers with lower overhead and latency than PIO.

10b Demonstrate why interrupts are needed by carefully explaining what programs would have to do differently to perform IO if interrupts didn't exist and what disadvantages this approach would have.

If interrupts didn't exist, a program would have to repeatedly *poll* the IO device to determine whether the device had information (e.g., keyboard presses) for it. The disadvantages of polling are that it wastes CPU cycles unnecessarily when the IO device doesn't have information for the CPU. If polling is very frequent, then this overhead is very high.

Infrequent polling may not be a suitable solution either because it increases the latency (i.e., delay) between when an IO device notifies the CPU that it wants its attention, and when the CPU actually notices it. This would increase, for example, the latency of disk and network reads. There is an undesirable tradeoff between latency and CPU 'wastage'.

10c Explain how interrupts would be added to the Simple Machine simulator by indicating where the interrupt-handling logic would be added and saying roughly what it would do.

Insert a new stage just before the fetch stage (or after the execute stage). Check there to see if there is an interrupt pending and if so, jump to the address specified in the interrupt vector table. This will invoke the specific interrupt service routine for the given type of interrupt.

11 (10 marks) Threads.

11a Threads can be used to manage the asynchrony inherent in I/O-device access (e.g., reading from disk). Carefully explain how threads help.

Because threads execute asynchronously themselves, you can write code that looks synchronous but executes asynchronously. This is easier to read, write, and think about for programmers.

IO can take quite long (eg millions of CPU cycles), and in this time the CPU can be doing other things. Threads provide this flexibility because they can easily be blocked when they are waiting for something, and the CPU can go ahead and execute other threads.

11b Carefully describe in plain English the sequence of steps a user-level thread system such as *uthreads* follows to switch from one thread to another. Ensure that your answer explains the role of the *ready queue* and explains how the hardware switches from running one thread to the other.

A thread switch occurs in uthreads when a thread calls `uthread_block()` or `uthread_yield()`. This places the current thread on the ready queue and dequeues something else off the ready queue to begin execution.

The actual transfer of execution is achieved by pushing the register of the first thread onto its stack, saving its current stack pointer in its TCB and then switching the value of the stack pointer register to point to the target thread's stack. Then the CPU pops the registers of this new thread from its stack, and resumes execution.

11c What is the role of the *thread scheduler*?

The thread scheduler decides which thread should execute next.

11d Explain priority-based, round-robin scheduling.

Each thread has an assigned priority. When a thread finishes execution (or blocks/yields), the next thread chosen to execute is the thread with the highest priority.

11e Explain what else is needed to ensure that threads of equal priority get an equal share of the CPU?

In the above explanation of priority-based, round-robin scheduling *starvation* is possible because a single high priority thread could dominate all the CPU time if it never finishes execution. *Quantum preemption* would allow for all threads of equal priority to get an equal share of the CPU by periodically interrupting and switching threads.

12 (16 marks) Synchronization

12a Explain the difference between busy-waiting and blocking. Give one advantage of blocking.

Busy waiting occurs when the a thread waits for a lock to be free by actively polling the lock's value, spinning in a loop repeatedly reading it until it sees that it is available.

Blocking waiting occurs when a thread waits for a lock by sleeping so that other threads can use the CPU. It is then the responsibility of the thread that releases the lock to wakeup the waiting thread.

12b Consider the following program in which `inc` and `dec` can run concurrently.

```
spinlock_t s;
int        c;
void dec() {
    int success = 0;
    while (success==0) {
        while (c==0) {}
        spinlock_lock (s);
        if (c>0) {
            c = c - 1;
            success = 1;
        }
        spinlock_unlock (s);
    }
}
void inc() {
    spinlock_lock (s);
    c = c + 1;
    spinlock_unlock (s);
}
```

Re-implement the program to eliminate all busy waiting using *monitors* and *condition variables*. You may make the changes in place above or re-write some or all of the code below.

```
monitor m = new_monitor();
cond_variable gtzero = new cond_variable(m);
int c;

void dec() {
    monitor_enter(m);
    while (c == 0)
        cond_wait(gtzero);
    c = c - 1;
    monitor_exit(m);
}

void inc() {
    monitor_enter(m);
    c = c + 1;
    cond_signal(gtzero);
    monitor_exit(m);
}
```

If you are familiar with how the producer/consumer problem is implemented with condition variables, this question should be straightforward. If you're not, **go study that right now**. You should be able to easily identify the pattern of spinning on a value until it meets a condition, then acquiring a lock to verify that condition actually holds. Also look into how spinlocks are implemented, they use quite similar logic to acquire the shared lock (by spinning on the lock's value before attempting the atomic exchange).

I use pseudo code here to represent monitor/condition variable operations (recall that a monitor is basically a blocking lock). You could have instead used uthread function calls like `uthread_mutex_lock` and `uthread_cond_wait` if you wanted to. Pay attention to how the condition variable is initialized - don't forget you must associate it with a mutex!

12c Assume that monitors are implemented in such a way that a thread inside of a monitor is permitted to re-enter that monitor repeatedly without blocking (e.g., when `bar` calls `zot`, which calls `foo`, `foo` is permitted to enter monitor `x`). Indicate whether the following procedures could cause deadlock in multi-threaded program that contained them (and other procedures as well). Explain why or why not. If they could, say whether you could eliminate this deadlock by only adding additional monitors or additional monitor enter or exits (you may not remove monitors). If so, show how.

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (x);
    zot ();
    monitor_exit (x);
}
void zot () {
    monitor_enter (y);
    foo ();
    monitor_exit (y);
}
```

Consider the case where Thread 1 is executing `bar()` and Thread 2 is executing `zot()`. Let's say Thread 1 acquires `x` at the same time that Thread 2 acquires `y`. Now when Thread 1 calls `zot()` and tries to acquire `y`, it will block (because Thread 2 holds it). Similarly, when Thread 2 proceeds to call `foo()` and attempt to acquire `x`, it blocks because Thread 1 holds it. So a deadlock is possible.

Since the concurrent execution of `bar()` and `zot()` causes a deadlock, we can avoid the deadlock by preventing them from executing concurrently:

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (z);
    monitor_enter (x);
    zot ();
    monitor_exit (x);
    monitor_exit (z);
}
void zot () {
    monitor_enter (z);
    monitor_enter (y);
    foo ();
    monitor_exit (y);
    monitor_exit (z);
}
```

Doing well on this question requires you to be very comfortable with imagining the execution of threads. You know that a deadlock occurs when two threads are blocked in such a way that neither can release the resource that the other is waiting for. Here we only have two locks/resources - `x` and `y`.

Knowing that, you should start working backwards from a deadlock. One thread will be blocked waiting for `x` and the other `y`. There is only one place where a thread can be waiting for `y` - at `zot()`. So one of the threads must be blocked on that call for a deadlock to occur *and that thread must be holding* `x` already. This implies that the thread must have started by calling something that acquired `x` before calling `zot()`; ie `bar()`. In the answer we call this 'Thread 1'.

Now all that we have to do is come up with a second thread that needs to be blocked on `x` while holding `y`, and we will have a deadlock. Using similar logic as above, we can figure out that this happens when `foo()` is called by `zot()`.