

CISP310 Exam 2

Prof. Tak Auyeung

April 26, 2005

Instructions: You may bring any material that is hand-written or printed *prior* to the examination to help you. You can also bring a calculator if you think it may help you. However, you can only use the calculator for numerical computations only.

You, as an individual, are expected to do your own work. This means you cannot seek, receive or otherwise acquire any assistance except clarifications from the professor during an examination. Any communication involving the contents of the subject matter or the examination is considered cheating. Do not initiate or accept such communication, or the result of your examination is automatically voided.

Each correct answer is worth one point, each wrong answer is worth -0.25 point, and each unanswered question is worth zero point.

Make sure you write down your name on the upper right corner *first*, otherwise I cannot give points to anonymous students!

Notation: The name of a local variable assumes different roles in different parts of this test. In pseudocode or C code, the name of a local variable refers to the variable itself. In assembly code, unless otherwise noted, the name of a local variable is the displacement from `ebp` in number of bytes to the local variable. The same applies to parameters.

Types: `int16` is a 16-bit signed integer, `int32` is a 32-bit signed integer, `char` is an 8-bit character. We assume all addresses and pointers are 32-bit.

Conventions: Return values are returned by registers. 16-bit and 8-bit values are returned by `ax`, 32-bit values are returned by `eax`.

The baseline is 10, there are 12 questions.

1 The following is a partial subroutine. Most of the prologue and epilogue is included. The exit sequence is missing. Fill in the blank.

```
sub1:
    pushl %ebp
    movl  %esp,%ebp
    addl  $lastLocalVar,%esp
    ...
    ---
    ret
```

- A `movl %esp,%ebp`
`popl %esp`
- B `movl %esp,%ebp`
`popl %ebp`

- C `subl $lastLocalVar,%esp`
`popl %ebp`
- D `movl %ebp,%esp`
`popl %ebp`
- E 1c and 1d

2 The displacements to local variables are defined as follows:

```
oldBP = 0
v3 = oldBP - v3_size
v2 = v3 - v2_size
v1 = v2 - v1_size
lastLocalVar = v1
```

In order to write to variable `v2`, it is common to use instructions such as `movl $0,v2(%ebp)`, assuming `v2` is a 32-bit integer.

What happens when `v1_size` is mistakenly defined to be less than what it should? What would be the effect of `movl $0,v2(%ebp)`? Assume everything else is properly defined.

- A only `v3` is overwritten
- B only `v2` is overwritten
- C only `v1` is overwritten
- D both `v1` and `v2` are overwritten
- E both `v2` and `v3` are overwritten

3 The following subroutine adds `int32`, and returns the sum in `eax`. Its C prototype is as follows:

```
int32 add2(int32 x, int32 y);
```

The following code implements the caller's code as follows:

```
i = add2(i, i);
```

is implemented by

```
pushl i(%ebp)
pushl i(%ebp)
call add2
movl %eax,i(%ebp)
addl $8,%esp
```

Is there a problem with this invocation code? If so, what is it?

- A There is no problem.
- B The last two instructions should swap places.
- C `addl $8,%esp` should be `addl $8,%ebp`
- D `addl $8,%esp` should be `addl 8,%ebp`
- E There should be one `pushl i(%ebp)`

4 The following code is used by a caller to invoke the subroutine `sub2`. Assume the stack cleanup code is done properly. Regardless of the implementation of `sub2`, how many bytes does the caller use on the stack?

```
pushl $52
pushw myVar(%ebp)
call sub2
---
```

- A 4
- B 6
- C 8
- D 10
- E 52

5 Assume the following C prototypes:

```
int32 f1(int32 x, int32 y);
int32 f2(int32 x);
int32 f3(int32 x);
```

Assuming that `v1` is a local variable of the caller, how do we implement the following code?

```
v1 = f1(f2(5678), (f3(1234)));
```

- A

```
pushl $1234
call f3
movl %eax,(%esp)
pushl $5678
call f2
movl %eax,(%esp)
call f1
addl $8,%esp
movl %eax,v1(%ebp)
```
- B

```
pushl $1234
call f3
addl $4,%esp
pushl %eax
pushl $5678
call f2
addl $4,%esp
pushl %eax
call f1
addl $8,%esp
movl %eax,v1(%ebp)
```
- C

```
pushl $1234
pushl $5678
call f3
popl %eax
call f2
popl %eax
call f1
movl %eax,v1(%ebp)
```

- D both 5a and 5b
- E both 5b and 5c

6 I need to preserve the value of `ebx` and `ecx`. However, the subroutine `sub3` can potentially overwrite these two registers. The calling subroutine declares 8 bytes from the call frame as follows:

```
oldBP = 0
localVarN = oldBP - localVarN_size
...
localVar1 = localVar2 - localVar1_size
savedEBX = localVar1 - 4
savedECX = savedEBX - 4
lastLocalVar = savedECX
```

Space is reserved on the stack using `addl $lastLocalVar,%esp`

How do I preserve these registers? Assume the original code to call `sub3` is as follows:

```
pushl $123 # 2nd parameter
pushl $94 # 1st parameter
call sub3
# parameter clean up code
```

A `pushl $123 # 2nd parameter`
`pushl $94 # 1st parameter`
`movl %ebx,savedEBX(%ebp)`
`movl %ecx,savedECX(%ebp)`
`call sub3`
`movl savedEBX(%ebp),%ebx`
`movl savedECX(%ebp),%ecx`
`# parameter clean up code`

B `pushl $123 # 2nd parameter`
`pushl $94 # 1st parameter`
`movl savedEBX(%ebp),%ebx`
`movl savedECX(%ebp),%ecx`
`call sub3`
`movl %ebx,savedEBX(%ebp)`
`movl %ecx,savedECX(%ebp)`
`# parameter clean up code`

C `movl %ebx,savedECX(%ebp)`
`movl %ecx,savedEBX(%ebp)`
`pushl $123 # 2nd parameter`
`pushl $94 # 1st parameter`
`call sub3`
`# parameter clean up code`
`movl savedEBX(%ebp),%ebx`
`movl savedECX(%ebp),%ecx`

D 6a and 6b

E 6a and 6c

7 Assume a subroutine has a C prototype as follows:

```
void *f4(void);
```

This means it returns a pointer (address), and there is no parameter. The following is the implementation in assembly language:

```
f4:
    movl (%esp),%eax
    ret
```

The following is how this subroutine is used, assume `mystery` is a local variable of the caller:

```
call f4
movl %eax,mystery(%ebp)
```

Choose the correct explanation.

A The subroutine may crash (segmentation fault) when it attempts to execute the `ret` instruction because the stack pointer is no longer pointing to the return address.

B The subroutine may crash (segmentation fault) when it attempts to execute the `ret` instruction because the return address stored on the stack is altered.

C The subroutine does not crash, but the stack is not balanced.

D `mystery` (as a variable) stores the address of the `movl` instruction in the caller's code.

E `mystery` (as a variable) stores some unspecified value.

8 In C notation, "`int32 *p;`" means `p` is a pointer to a 32-bit integer. "`int32 i;`" means `i` is an 32-bit integer. "`*p`" means the location pointed to by `p`. How do we store the value of `i` into wherever `p` points to? In other words, how do we implement "`*p = i;`"? Assume `p` and `i` are either local variables or parameters.

A `pushl i(%ebp)`
`movl p(%ebp),%eax`
`popl (%eax)`

B `movl $i,%eax`
`addl %ebp,%eax`
`movl p(%ebp),%ebx`
`movl %eax,(%ebx)`

C `movl $i,%eax`
`addl %ebp,%eax`
`movl p(%ebp),%ebx`
`movl %ebx,(%eax)`

D `movl i(%ebp),%eax`
`movl %eax,p(%ebp)`

E `pushl i(%ebp)`
`popl p(%ebp)`

9 A subroutine call should have used the following code:

```
pushw $1234
pushl myVar(%ebp)
call sub5
# parameter clean up
```

However, the programmer makes a mistake, and it becomes the following:

```
pushl $1234
pushl myVar(%ebp)
call sub5
# parameter clean up
```

Assuming that the subroutine does not attempt to change the parameters, what is the result of the erroneous code? Assume the “parameter clean up code” is correct with respect to the parameters actually passed.

- A The program always crashes in the subroutine `sub5`.
- B The program may not crash, but the second parameter becomes 0 instead of 1234.
- C The program always crashes as the caller returns to its own caller.
- D The program may not crash, but `sub5` may not perform its calculations correctly.
- E The program works fine as is, since 1234 can be represented in 16 bits.

10 Subroutine `sub6` expects two 32-bit parameters. What happens when the following code is used to call `sub6`?

```
pushl $1234
pushl $2345
pushl $3456
pushl myVar(%ebp)
call sub6
# parameter clean up
```

You can think of the prototype of `sub6` as follows:

```
void sub6(int32 x, int32 y);
```

- A 2345 becomes x, 1234 becomes y
- B 2345 becomes x, 3456 becomes y
- C 3456 becomes x, 2345 becomes y
- D 1234 becomes x, 2345 becomes y
- E the value of parameters x and y cannot be determined

11 In our discussion of stack and invocation frames, which register restores its value without being saved on and restored from the stack?

- A `%eax`
- B `%ebp`
- C `%ecx`
- D `%esp`
- E all registers are saved on and restored from the stack

12 Using our “conventional” use of `%esp` and `%ebp` in an invocation frame, which of the following condition indicates trouble?

- A `%esp < %ebp`
- B `%ebp < %esp`
- C `%esp <= %ebp`
- D `%ebp <= %esp`
- E `%ebp <> %esp` (<> means not equal)