Procedures

Getting Started

To obtain a copy of today's activity, log into a shark machine and do the following:

```
$ wget http://www.cs.cmu.edu/~213/activities/machine-procedures.tar
$ tar xf machine-procedures.tar
$ cd machine-procedures
```

Record your answers to the discussion questions below. You may wish to refer back to the activity from last week (https://www.cs.cmu.edu/~213/activities/gdb-and-assembly.pdf) which contains a list of relevant GDB commands.

1 Activity 1: Calls

In the `machine-procedures` directory that you created, run the `calls` binary from within GDB, like this:

```
$ gdb --args ./calls
(gdb) r
```

The program will instruct you as you progress through the activity. These questions accompany the program; when it prompts you to answer a problem, discuss with your partner and write your answer here.

Problem 1. Fill in the contents of the stack:

0x 0x0000000000015213	← $rsp = 0x 7fffffffdf10
0x 0x000000000040117a	
...	

Problem 2. What was the meaning of the second number on the stack?

the return address

Problem 3. What does the `ret` instruction do?

pop the stack and store the pop address to rip

Problem 4. Given your answer to Problem 3, what must it be that `call` does?

push rip(the address of
the instruction immediatly
after the call) onto the
stack, and jumps to the
address of its operand

Problem 5. What special optimization of calls has been applied to `returnOneOpt`? Why does this optimization work for `returnOneOpt`? Can it be used for any call?
the call is changed into jmp, because returnOneOpt do nothing after the call to abs.NO

The transformation you just witnessed is a simple example of tail-call optimization. Because a call was the last instruction before a ret, within a function that doesn't adjust the stack pointer, the compiler could skip allocating a stack frame: both the call and the ret could be replaced with a simple jmp to the called function. That function must end in its own ret instruction, which will use the return address pushed by the call to returnOneOpt. This optimization is especially valuable when applied to recursive functions.

2 Activity 2: Arguments and Local Variables

In the `machine-procedures` directory that you created, run the `locals` binary from within GDB, like this:

```
$ gdb --args ./locals
(gdb) r
```

The program will instruct you as you progress through the activity. These questions accompany the program; when it prompts you to answer a problem, discuss with your partner and write your answer here.

Problem 6. What is the type of the data `seeArgs` passes as the first argument to `printf`? (You should be able to answer this question based solely on what you already know about `printf`.) Given this, and what you saw when you followed the instructions up to this point, what does the GDB command `x/s` do?

const char* ; print the value as
string

Problem 7. When `seeMoreArgs` calls `printf`, where did the compiler place arguments 7 and 8? Why do you think this happened?

it place them on the stack in reverse order;because there is no
enough place in registers to hold these arguments

Problem 8. Where does the function `getV` allocate its array? How does it pass this location to `getValue`?

on the stack;   by copying the pointer to rdi

Problem 9. Which registers are treated as call-preserved by `mult4`? Which register does `mult4` expect to contain a return value? (It may help to disassemble `mult2` as well.)

rbx, r12, r13; rax

Problem 10. What does the function `mrec` do?

```
if (x == 1)
    return x
return x *mrec(x - 1)
```

calculating factorial

3 Activity 3 (Optional, Time Permitting): Endianness Preview

Rerun `gdb -args ./calls` and continue to the point where you printed the stack before.

Problem 11. The first eight bytes of the stack contain the number `0x15213`. What do you expect the first *two* bytes of the stack to contain?

0x13 0x52

Problem 12. Check your hypothesis by running `x/2xb $rsp`. What did the first two bytes of the stack contain? What can you deduce about the order in which each integer's bytes are stored?

0x13 0x52   it store the least significant byte first, namely, the smallest address

Appendix: x86-64 ELF Calling Convention Summary

The following table lists all of the x86-64 integer registers, indicates whether each is call-preserved or call-clobbered, and gives the conventional function of each.

Register	Call Treatment	Function
`%rax`	Clobbered	Return value
`%rbx`	Preserved	
`%rcx`	Clobbered	Argument #4
`%rdx`	Clobbered	Argument #3
`%rbp`	Preserved	
`%rsp`	Preserved	Stack pointer
`%rsi`	Clobbered	Argument #2
`%rdi`	Clobbered	Argument #1
`%r8`	Clobbered	Argument #5
`%r9`	Clobbered	Argument #6
`%r10`	Clobbered	
`%r11`	Clobbered	
`%r12`	Preserved	
`%r13`	Preserved	
`%r14`	Preserved	
`%r15`	Preserved	