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 September 8 (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` 15213	← `$rsp = 0x` 7fffffffdb30
`0x` 00000040117a	
. . .	

Note: You might not get exactly the same values for $rsp and the return address.

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

The second number on the stack is the function's return address.

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

`ret` pops from the top of the stack to `%rip` (incrementing `%rsp` by 8 bytes).

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

`call` pushes the value of `%rip` to the stack (decrementing `%rsp` by 8 bytes), then unconditionally branches (jumps) to the call address described by 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?

In `returnOneOpt`, the compiler saw that `call abs` would be followed immediately by `ret`, so it replaced both instructions with `jmp abs`. This is called "tail call" optimization. It works because the `ret` in the called function (`abs`) will have exactly the same effect that the `ret` removed from `returnOneOpt` would have had—restoring the stack to what it was before `returnOneOpt` was called and returning to `returnOneOpt`'s caller. This optimization cannot be used on every call—it only works when a call is followed immediately by a return. (Sometimes there can be a couple of stack adjustment instructions in between.) That's why it's called *tail* call optimization.

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?

The first argument of `printf` should always be a format string, which has type `const char *`. `x/s` prints out the C string found at the specified location in memory.

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

Arguments 7 and 8 were pushed onto the stack in reverse order. This happened because the compiler ran out of integer argument registers.

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

getV allocates its array on the stack. It passes this location to getValue by using a normal pointer stored in a standard argument register.

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, and %r13 are call-preserved. %rax contains return values.

Problem 10. What does the function mrec do?

mrec computes the factorial of its integer argument.

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?

Logically, among the eight bytes there should be three with the values 0x01, 0x52, and 0x13, and the other five should all be zero. They *could* be in any order, but it would make sense for their order to relate somehow to the place value of the bits. . . and that's as far as we can guess.

(We *did* mention, briefly at the end of a previous class, the additional piece of information you need to answer this question, but you might have missed it.)

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?

We see the bytes 0x13 and 0x52 — each integer's bytes are stored *least* significant to *most* significant.

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	