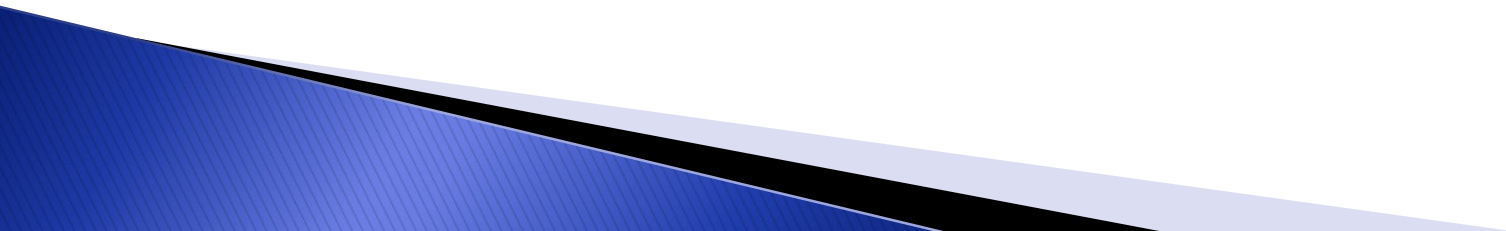


Section 3: Functions, Registers & the ABI



Call instruction

- ▶ **call Dest**
- ▶ *Dest* will be a label which has been placed in the assembly language source code at the address of the procedure to be called.
- ▶ When the **call** instruction is executed, the address of the instruction after the call instruction is pushed onto the stack (that is, the return address is pushed), and the address of *Dest* is assigned to the PC (%rip).
- ▶ This means that, when the called procedure begins execution, the return address is the last thing that has been pushed onto the stack.

How do we determine return address?

- ▶ PC typically contains address of the current instruction (intel %rip holds the address of the next instruction)
- ▶ A call instruction has a one-byte opcode followed by either
 - An 8-byte address or
 - A 4-byte offset
- ▶ So... either 9 or 5 bytes after the call opcode is the address of the next instruction.
- ▶ That is the return address to push 😊

Calling Functions

- ▶ We've discussed how to separate space on the stack for each function by using **stack frames**
- ▶ If main(), or some other function, fills many (all?) of the 16 integer registers with valid data, then calls another function, what happens to that data? What registers can the called function use to perform it's work?

What to do? What to do? 😊

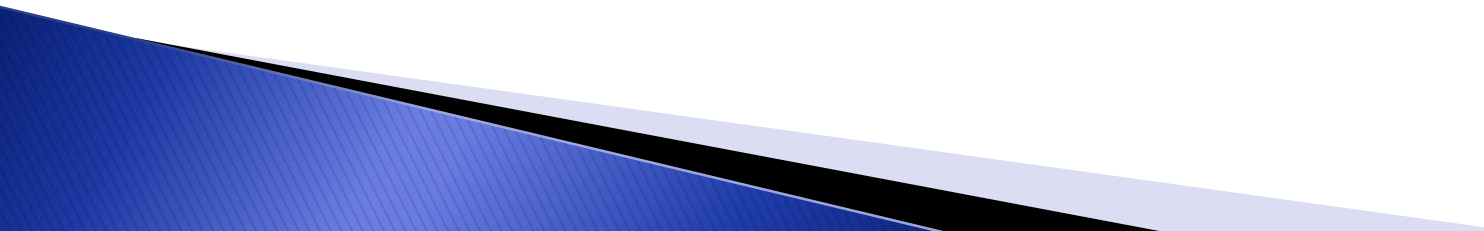


Options:

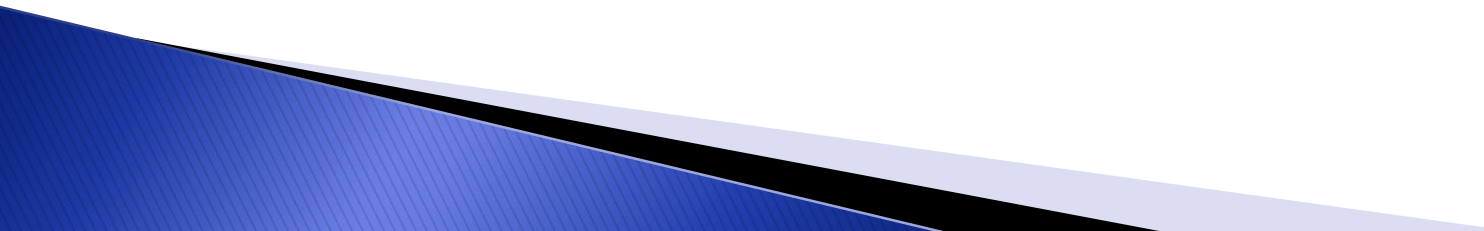
1. The function that is performing the call has to save *every, single* register it's using to the stack prior to making the call, then pop them back into the appropriate registers upon return.
2. The function that is called has to save *every, single* register it plans to use to the stack prior to doing any “real” work, then pop the values back into the correct registers before returning to the calling function.

Both seem a little harsh! Can't we both just get along???

How about a little cooperation?



X86 Register Usage Conventions

- ▶ Although only one procedure can be active at a given time, the 16 registers are “shared” by all procedures.
 - ▶ Therefore, we need a way to ensure that when one procedure (the caller) calls another (the callee), values that the caller needs after return will not be overwritten.
 - ▶ To ensure this, conventions have been adopted as to which procedure, the caller or callee, is responsible for preserving a given register (other than %rsp).
 - ▶ We will be adopting register save conventions used by X86-64 in a C programming environment.
- 

X86 Register Usage Conventions

- ▶ **Caller Saved Registers:** registers that the “Caller” function pushes to the stack prior to calling the “Callee” function, IF the register contains data needed by the “Caller” after the “Callee” function returns. Caller must pop them from the stack after Callee returns.
- ▶ **Callee Saved Registers:** registers that the “Callee” function must push to the stack if the “Callee” function wishes to use the register. “Callee” **must** assume there is data in each of these registers that is important to the “Caller” function. Callee function must pop these register back prior to returning to the “Caller”.
- ▶ The first 6 parameters are passed from the caller to the callee in registers %rdi, %rsi, %rdx, %rcx, %r8 and %r9, respectively.
- ▶ If the callee returns a value to the caller, it is returned in register %rax.
- ▶ Confused? Check out Figure 3.2, p 180 of Bryant/O’Halloran

Register Conventions

%rax	Return value	%r8	5 th parameter
%rbx	Callee Saved	%r9	6 th parameter
%rcx	4 th parameter	%r10	Caller Saved
%rdx	3 rd parameter	%r11	Caller Saved
%rsi	2 nd parameter	%r12	Callee Saved
%rdi	1 st parameter	%r13	Callee Saved
%rsp	Stack Pointer	%r14	Callee Saved
%rbp	Callee Saved	%r15	Callee Saved

What did all that register stuff really mean?

Register Allocation

- ▶ What can be used?
- ▶ When do you save?
- ▶ Linux uses what is call System V ABI to define this.

System V ABI – 64 bit processors

- ▶ What is System V ABI? – a “bible” of sorts with respect to how to interface to C standard libraries when not using C code....X86-64, for example.
- ▶ Here is the link to a reasonable “draft” copy from 2013:

<https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

A bug seems to be that if you just try to click on this link, it doesn't work, but if you copy/paste the link in a browser window it comes up just fine.

There are many interface standards within the ABI, register usage and caller/callee parameters a just a couple...

System V ABI says

- ▶ This first six integer or pointer arguments are passed in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9 – *in this order*.
- ▶ %rax is used for return values
- ▶ %rsp must be restored when control is returned to the caller function
- ▶ If the callee wishes to use registers %rbx, %rbp or %r12–%r15, it must save/then restore their original values before returning control to the caller.
- ▶ All other registers must be saved by the caller if it wishes to preserve their values

So which registers do we use?

- ▶ It depends!
 - On what our function is passed
 - On what registers our function wants to use
 - On what other functions our function might call
 - How we can minimize save/restore activity
 - Efficiency is why we got here in the first place, remember?

All functions

- ▶ Have to save and restore any of these registers it plans to use:
 - %rbp: probably using as part of the stack frame and planned to restore it anyway
 - %rsp: if we don't restore it, the program will probably crash. Assume all functions deal with %rsp correctly.
 - %rbx and %r12-%r15: Have to save these before we use them

Leaf Functions

- ▶ Leaf functions make no calls to any other function
- ▶ Can freely use %rdi, %rsi, %rdx, %rcx, %r8 and %r9 even when passed fewer than 6 parameters.
- ▶ Can freely use %r10 and %r11 since these are caller-saved registers
- ▶ Can freely use %rax as long as function fills it with the return value prior to returning to the caller.

Functions that call other functions

- ▶ Trade-offs to be made:
 - If we make many calls, we have to save and restore any of the parameter registers as well as %r10/%r11 before and after **each** call if we still want to use the values they had prior to the call.
 - If we use %rbx, %rbp, %r12-%r15, we only have to save them one time (at the beginning) and restore them one time (at the end).

Recursive procedure calls

- ▶ Because the stack frame for the procedure is set up at the beginning of its code, a procedure which calls itself recursively will get a **new stack frame** each time it is called.
- ▶ The stack frame for the second call of the procedure will be above the stack frame for the first (i.e., closer to the top of the stack, but at a lower-numbered address), and so on.
- ▶ When each call returns, the frame pointer of the previous call will be restored, and at that point, what is at the top of the stack will be the return address from the previous call.
- ▶ Therefore, when the ret instruction is executed at the end of the recursive function's assembly code, execution will return to the point in the code of the function from which the call was made.
- ▶ So, just how deep of a recursive procedure do you want to have in your code given all of the resources each call is going to use? Hmmmm?

Linux/Unix C-Library function calls

- ▶ We can call any C-Library function that we used in our C-language programs from any x86-64 program that we write
- ▶ Parameters must be passed using the caller/callee/return value paradigm described above
- ▶ Assume that all caller saved registers will be trashed after return and plan accordingly 😊

System V ABI – 64 bit processors

- ▶ From section 3.5.7 Variable Argument Lists:

Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that all arguments are passed on the stack, and arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the AMD64 architecture because some arguments are passed in registers. Portable C programs must use the header file in order to handle variable argument lists. **When a function taking variable-arguments is called, %rax must be set to the total number of floating point parameters passed to the function in vector registers.**

- ▶ Since we won't be passing any floating point parameters (we're only using integers), we will **always** have to set %rax to zero before calling a function that allows a variable argument list.

*the section above references AMD64 architecture, but x86-64 is equivalent

Variable Argument Lists

- ▶ What functions did we use in C that had variable argument lists?
 - printf() family
 - scanf() family
- ▶ You have make a point to set %rax to zero prior to calling printf() or scanf(), because if you do not, expect your program to seg fault.
- ▶ No only that, but **fully expect** the information in all “caller saved registers” to be **totally trashed** upon return