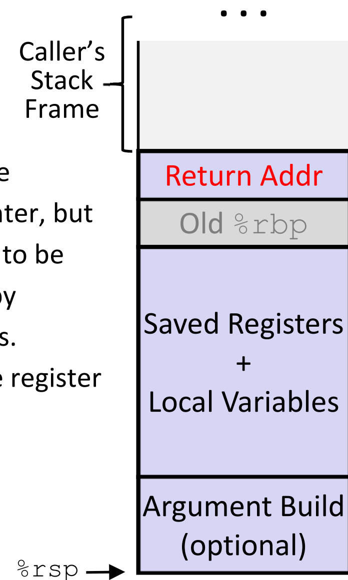
CSE 351 Lecture 12 – Procedures & Recursion

Stack Frame Details



A diagram of a typical stack frame layout is shown on the right. The return address, which is pushed onto the stack by call, marks the beginning of the stack frame. Older assembly code used `%rbp` as the *frame pointer*, an indicator of the beginning of the current stack frame. The old value of `%rbp` would need to be saved if it is being used as a frame pointer, but this is considered optional in x86-64. Next, old register values that needed to be saved would be pushed (see register saving conventions below), followed by allocated space for local variables that aren't being stored solely in registers. Finally, if this procedure calls another procedure, it may need to push more register values and then arguments 7+ onto the stack.

> **Note:** CSPP considers the return address to be the end of the caller's stack frame (*e.g.* p.240). The distinction is arbitrary, but we are choosing to follow Intel's reference documentation.

Every stack frame is organized the same way, though each frame can be a different size and will omit parts that it doesn't need, since accessing the stack (in memory) is much slower than registers. So, if the callee procedure in the diagram had more than six arguments, arguments 7+ would be part of the caller's argument build area just above the stored return address.

---

x86-64 Register Saving Conventions

The ***register saving conventions*** are the part of the calling conventions that describe how we should deal with register reuse to avoid accidentally destroying another procedure's data. Since procedures have access to all of the general-purpose registers during their execution, we will designate registers as either *callee-saved* or *caller-saved*. Register values can be saved by pushing them onto the stack or copying them into a register of the opposite type (caller-saved vs. callee-saved).

In a ***callee-saved*** register, it is the callee's responsibility to save the old value before using/manipulating the register. The callee then restores the old value before returning to the caller. The saving is one of the first things done by the callee and the restoring is one of the last (just before `ret`). From the perspective of the caller, it assumes that the values it has in the callee-saved registers will remain the same when control is passed back to it.

In a ***caller-saved*** register, it is the caller's responsibility to save the old value (if the caller needs it later) before passing control to the callee. The caller then restores the old value after the callee returns. The saving is done right before calling the callee (but before the argument build, if needed) and the restoring is done right after the `call`. From the perspective of the callee, it assumes that is has free reign to change the values in the caller-saved registers without worrying about their old values.

Register	Use	Convention	Register	Use	Convention
`%rax`	Return value	Caller-saved	`%r8`	Argument #5	Caller-saved
`%rbx`		Callee-saved	`%r9`	Argument #6	Caller-saved
`%rcx`	Argument #4	Caller-saved	`%r10`		Caller-saved
`%rdx`	Argument #3	Caller-saved	`%r11`		Caller-saved
`%rsi`	Argument #2	Caller-saved	`%r12`		Callee-saved
`%rdi`	Argument #1	Caller-saved	`%r13`		Callee-saved
`%rsp`	Stack pointer	Callee-saved	`%r14`		Callee-saved
`%rbp`		Callee-saved	`%r15`		Callee-saved

Recursion

The great thing is that, with our stack frame layout and calling conventions, recursion works without any additional special considerations! As each recursive call is made, a new stack frame is created to hold that instance's local procedure context and the register saving conventions will prevent each instance from corrupting each other's data.

Something additional to look out for is now you have to write your recursive procedure from the perspective of *both* a caller and a callee. This is true for any procedure that calls another procedure (*i.e.* it is both called by a procedure and calls a procedure), but instead of being able to treat those interactions independently, a recursive function is guaranteed to use the same registers during its recursive calls. This means that in a recursive procedure, important values will need to be saved whether they are in caller-saved or callee-saved registers; the choice will just affect *when* the saving and restoring needs to happen.

Stack Frame Contents

To tie everything together now, let's re-examine what goes on the stack. In an "ideal" case, our minimal stack frame contains just a return address – this means all of the procedure's work is being done in the registers, which are much faster to access than memory.

A procedure *needs* to grow its stack frame in the following situations:

- It has too many local variables to hold in caller-saved registers
- It has local variables that can't fit in registers (*e.g.* arrays or structs)
- It uses the address-of operator (`&`) to compute the address of a local variable
- It calls another function that takes more than 6 arguments (*i.e.* needs an argument build)
- It uses data in caller-saved registers *across* (*i.e.* needed before and after) a procedure call
- It modifies/uses callee-saved registers