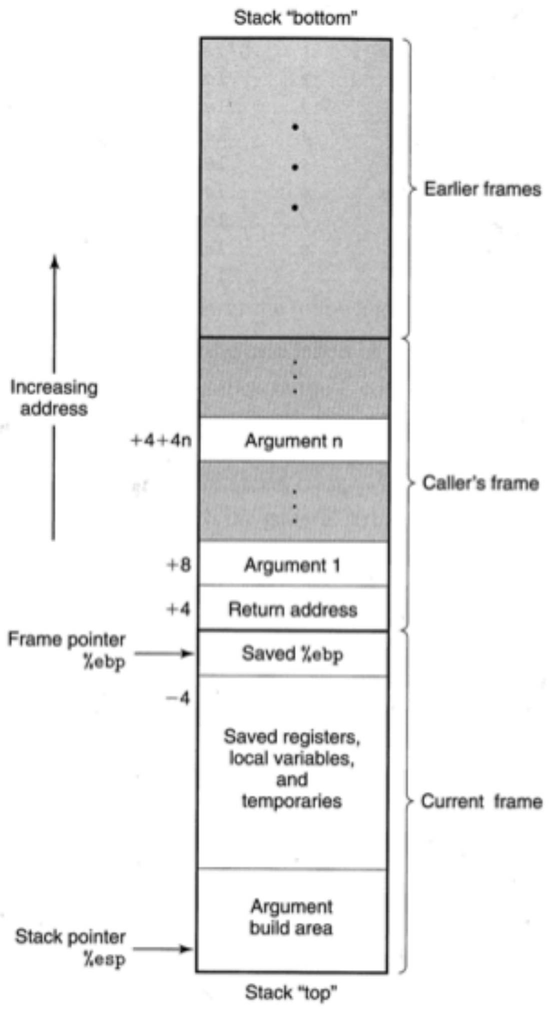
CS61 Fall 2011 Section 2
Additional Notes on Stack Layout

1 Stack layout

Remember, the exact layout of the stack frame is a convention and depends on hardware, OS, and compiler used. We'll look at the x86/Linux stack frame.

Figure 3.21
Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.



The *current* stack frame contains:

- The old value of `%ebp`
- Any saved registers
- Local variables (that don't fit in registers)
- Arguments to the function about to be called

The *caller's* (i.e. previous) stack frame contains:

- The return address (pushed by `call`)
- Arguments to the current function

2 Calling conventions

When a procedure invoked, the following things take place: Within the caller:

1. Any in-use caller-save registers are saved off
2. Arguments to the procedure are placed on the stack at `(%esp)`, `4(%esp)`, ...
3. `call` is executed, pushing the address of the next instruction on stack and transferring control to the callee

Within the callee:

4. The old `%ebp` is pushed on the stack
5. `%ebp` set to the current value of `%esp`
6. Additional stack space is allocated by decreasing `%esp` if the program has any locals, arguments to future calls, or temporaries
7. Any callee-save registers are saved off (if they will be used)
8. The body of the procedure is executed: arguments are accessed from the previous stack frame
9. Any callee-save registers are restored
10. The old `%esp` is restored from `%ebp` (if it was modified)
11. The old `%ebp` is restored by popping it from the stack

These last two steps can be combined by executing the `leave` instruction.

12. `ret` pops the return address off the stack (restoring `%esp` to its pre-call value) and transfers control back to the caller

2.1 Stack alignment

GCC adheres to an x86 programming guideline and makes sure that the total stack space used by a function is a multiple of 16 bytes (including 4 bytes for the saved `%ebp` and 4 bytes for the saved return address). As a result, GCC will allocate at least 24 bytes of stack even if the procedure uses less ($24+4+4 = 32 = 16 \times 2$). This maintains proper data alignment.