CS 24
Introduction to Computer Systems

4: Subroutines

Questions?

- How's the lab going?

- Lab 1 will probably be back to you soon.
- From the TAs:
  - GDB tutorial online.
  - GDB "mini-lecture" Wednesday eve at 8:30 in lab.

Last Time: GCD

```
int gcd(int a,int b) {
    // Euclidean Algorithm
    while (b != 0) {
        if (a > b) {
            a = a – b;
        } else {
            b = b – a;
        }
    }
    return a;
}
```

```
/* Assume a in %eax, b in %ecx */

loop:   cmp $0,%ecx
        je end
        cmp %ecx,%eax
        jle else
        sub %ecx,%eax
        jmp loop
else:   sub %eax,%ecx
        jmp loop
end:

/* Result is in %eax. */
```

Last Time: Faster GCD

```c
int gcd(int a,int b) {
    int t;
    while (b != 0) {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

```
/* Assume a in %eax, b in %ecx */

loop:   cmp $0,%ecx
        je end
        cdq
        idiv %ecx
        mov %ecx,%eax
        mov %edx,%ecx
        jmp loop
end:

/* Result is in %eax. */
```

Statically-allocated variables

```
        .data
a:      .space 4
b:      .space 4
        .text

# … stuff omitted …

loop:   cmp $0,b
        je end
        mov a,%eax
        cmp b,%eax
        jle else
        sub b,%eax
        mov %eax,a
        jmp loop
else:   mov b,%eax
        sub a,%eax
        mov %eax,b
        jmp loop
end:
```

- “.data” means put what follows in “data segment” of memory.
- “.space *n*” means *n* bytes of empty space.
- “.text” means put what follows in “text segment” (where code goes).
- Using label as operand means direct memory access.

Procedures

Calling a procedure:

- Put arguments where procedure's code can find them.
- Put "return address" where procedure's code can find it.
- Put everything else you're working on somewhere procedure's code won't overwrite it.
- Jump to beginning of procedure.
- (???)
- Execution resumes at return address, with result value where procedure left it (and all your other stuff untouched).

FORTRAN Procedures

- In old-school FORTRAN, all data statically allocated.
- Each procedure has statically-allocated space for parameters, return address, result (if any).
- To call:
  - copy parameters* to procedure’s parameter locations;
  - place return address in procedure’s return address location;
  - jump to procedure;
  - after return, get result from procedure’s result location.

* We are ignoring the fact that FORTRAN passes parameters by reference.

FORTRANish Function Call

```
            .data
gcd_a:      .space 4
gcd_b:      .space 4
gcd_ra:     .space 4
gcd_result: .space 4
            .text

# … stuff omitted …

gcd:        mov gcd_a,%eax
            mov gcd_b,%ecx
loop:       cmp $0,%ecx
            je end
            cmp %ecx,%eax
            jle else
            sub %ecx,%eax
            jmp loop
else:       sub %eax,%ecx
            jmp loop
end:        mov %eax,gcd_result
            jmp *gcd_ra
```

```
# … then, somewhere else …

            mov $12,gcd_a
            mov $18,gcd_b
            mov $return,gcd_ra
            jmp gcd
return:     mov gcd_result,%eax
```

- Calling code places return address in location gcd_ra.
- Function returns by indirect jump.
- For better performance, could pass information in registers.
(But on x86, will usually need memory.)

FORTRANish Function Call

```
            .data
gcd_a:      .space 4
gcd_b:      .space 4
gcd_ra:     .space 4
gcd_result: .space 4
            .text

# … stuff omitted …

gcd:        mov gcd_a,%eax
            mov gcd_b,%ecx
loop:       cmp $0,%ecx
            je end
            cmp %ecx,%eax
            jle else
            sub %ecx,%eax
            jmp loop
else:       sub %eax,%ecx
            jmp loop
end:        mov %eax,gcd_result
            jmp *gcd_ra
```

```
# … then, somewhere else …

            mov $12,gcd_a
            mov $18,gcd_b
            mov $return,gcd_ra
            jmp gcd
return:     mov gcd_result,%eax
```

Why will this strategy not work for C?

C needs to support recursion

FORTRANish Function Call

```
            .data
gcd_a:      .space 4
gcd_b:      .space 4
gcd_ra:     .space 4
gcd_result: .space 4
            .text

# … stuff omitted …

gcd:        mov gcd_a,%eax
            mov gcd_b,%ecx
loop:       cmp $0,%ecx
            je end
            cmp %ecx,%eax
            jle else
            sub %ecx,%eax
            jmp loop
else:       sub %eax,%ecx
            jmp loop
end:        mov %eax,gcd_result
            jmp *gcd_ra
```

```
# … then, somewhere else …

            mov $12,gcd_a
            mov $18,gcd_b
            mov $return,gcd_ra
            jmp gcd
return:     mov gcd_result,%eax
```

Why will this strategy not work for C?

C needs to support recursion.

Supporting Recursion

- Every invocation, or activation, of procedure needs its own copy of everything.
- Unknown number of activations alive at a time.

- *What do we do?*

Supporting Recursion

- Every invocation, or activation, of procedure needs its own copy of everything.
- Unknown number of activations alive at a time.

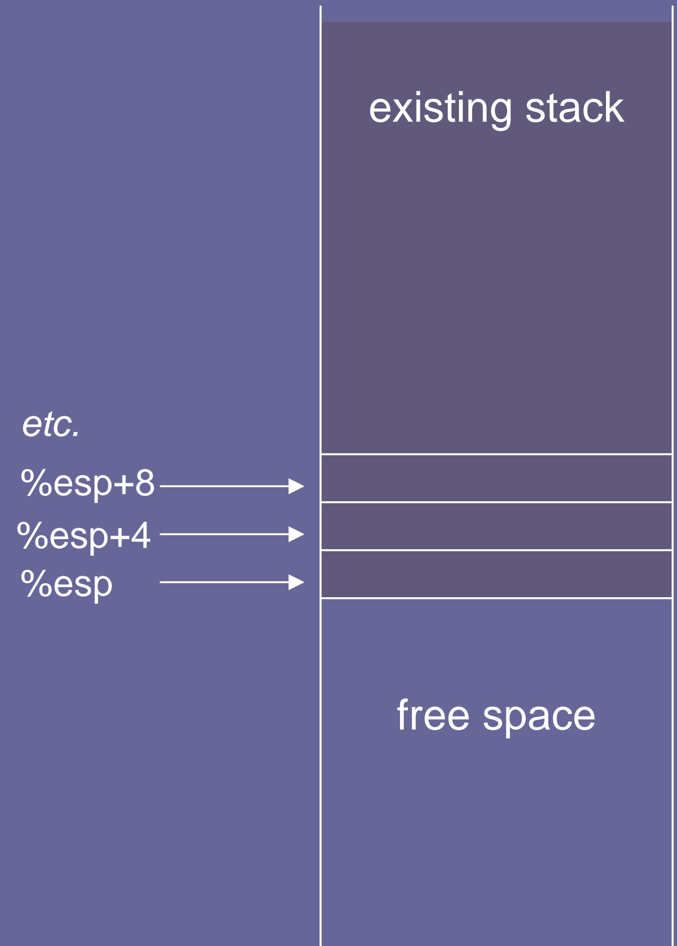
- *What do we do?*

  Use a (The) stack.
  - Procedure activations in program obey stack discipline: last one created is first one destroyed.
    - *Is this true in every programming language?*
  - Let each invocation create a separate “activation record”, store them on a stack.
  - This has been so important so long, it’s built into the hardware.

The Stack

- Portion of memory devoted to the stack.
- Stack grows “downward” – don’t ask why.
- Stack pointer, %esp, points to “fresh end”.
  - Most recently pushed.
  - “Top” of stack, but is really the bottom.

- To allocate *n* bytes of space:
  sub $*n*,%esp
  To deallocate:
  add $*n*,%esp
- Also:
  pushl (*32-bit value*)
  popl (*32-bit destination*)



Note: Always keep %esp divisible by 4.

C-ish Function Call

```
gcd:    mov 4(%esp),%eax
        mov 8(%esp),%ecx
loop:   cmp $0,%ecx
        je end
        cmp %ecx,%eax
        jbe else
        sub %ecx,%eax
        jmp loop
else:   sub %eax,%ecx
        jmp loop
end:    jmp *(%esp)
```

```
# … then, somewhere else …

        pushl $12
        pushl $18
        pushl $return
        jmp gcd
return: add $12,%esp
        # result is now in %eax.
```

- Caller pushes parameters, return address onto stack.
- Procedure gets parameters from stack using relative indirect addressing, *e.g.*, 4(%esp)
- Procedure leaves result in %eax.
- Caller deallocates stack space.

Call and Return Instructions

```
gcd:    mov 4(%esp),%eax
        mov 8(%esp),%ecx
loop:   cmp $0,%ecx
        je end
        cmp %ecx,%eax
        jbe else
        sub %ecx,%eax
        jmp loop
else:   sub %eax,%ecx
        jmp loop
end:    ret
```

```
# … then, somewhere else …

        pushl $12
        pushl $18
        call gcd

        add $8,%esp
        # result is now in %eax.
```

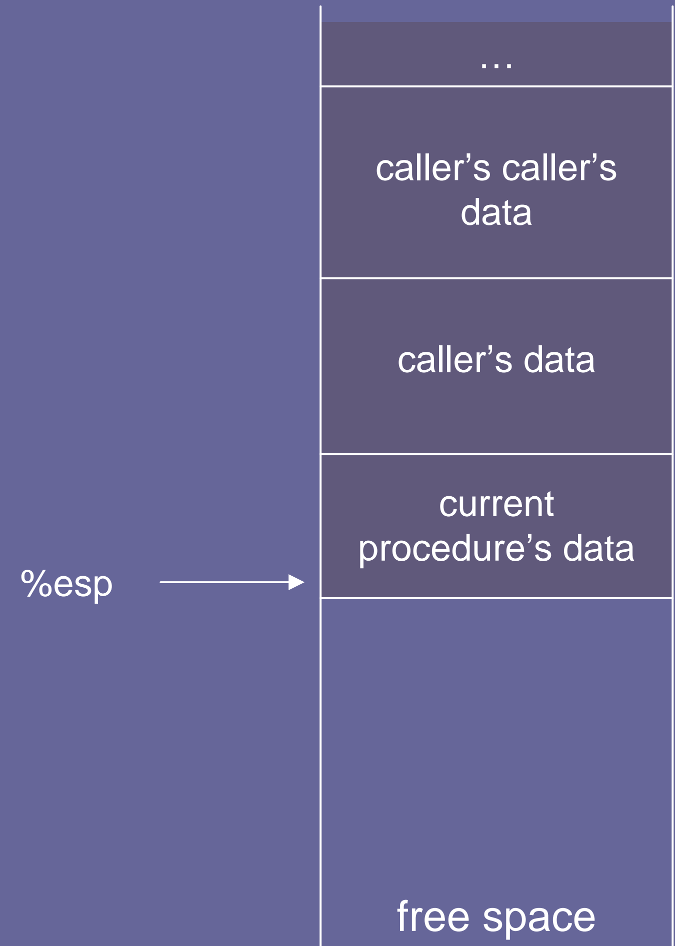
- call instruction pushes return address and jumps.
- ret instruction pops address and jumps to it.
- (Note, caller now only frees 8 bytes.)
- Code at right is now compatible with GCC.

Why we’re not done yet

- Got basic idea of functions.
- Still need to deal with:
  - Local variables.
  - What to do about registers.

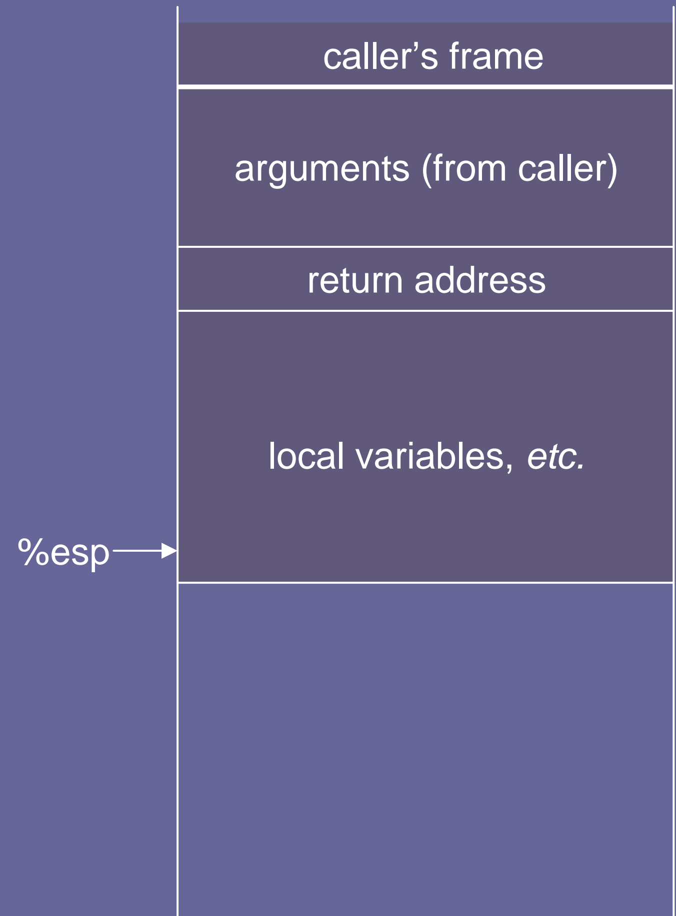
Stack Storage

- Stack provides per-invocation storage.
- Communication of arguments, return addresses…
- Local variables.
  - Declared in program text, or created by compiler.
  - Any temporary storage.
- Each live procedure activation has a frame.



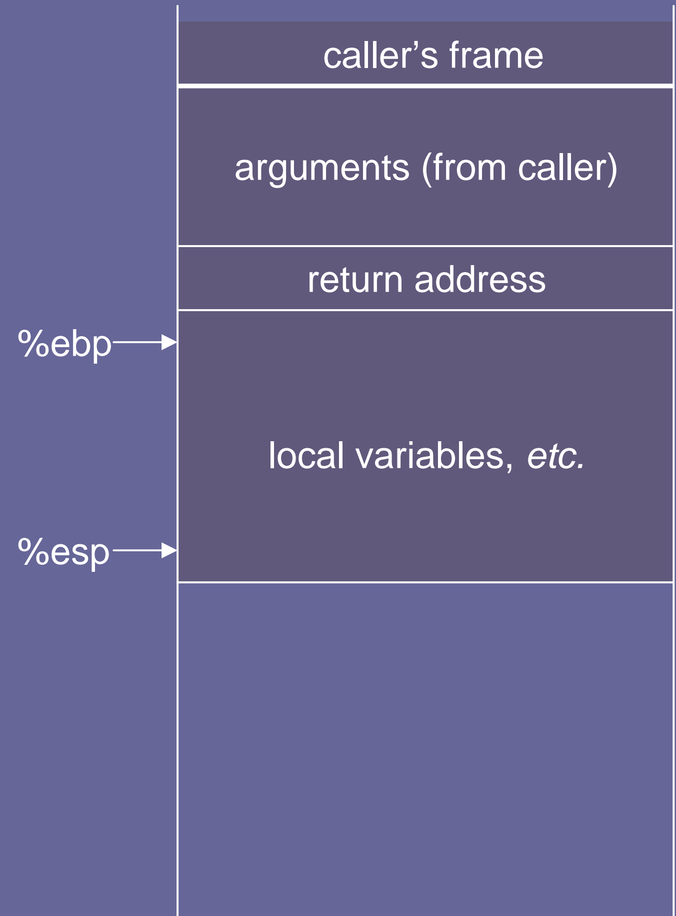
Typical Stack Frame

- Frame constructed during procedure call/entry:
  - Caller pushes arguments.
  - Caller executes `call`, which pushes return address.
  - Callee allocates local variable space.
- Frame destroyed during, after procedure exit.
  - Callee deallocates local variable space.
  - Callee executes `ret`, which pops return address.
  - Caller pops/deallocates arguments.



Frame Pointer

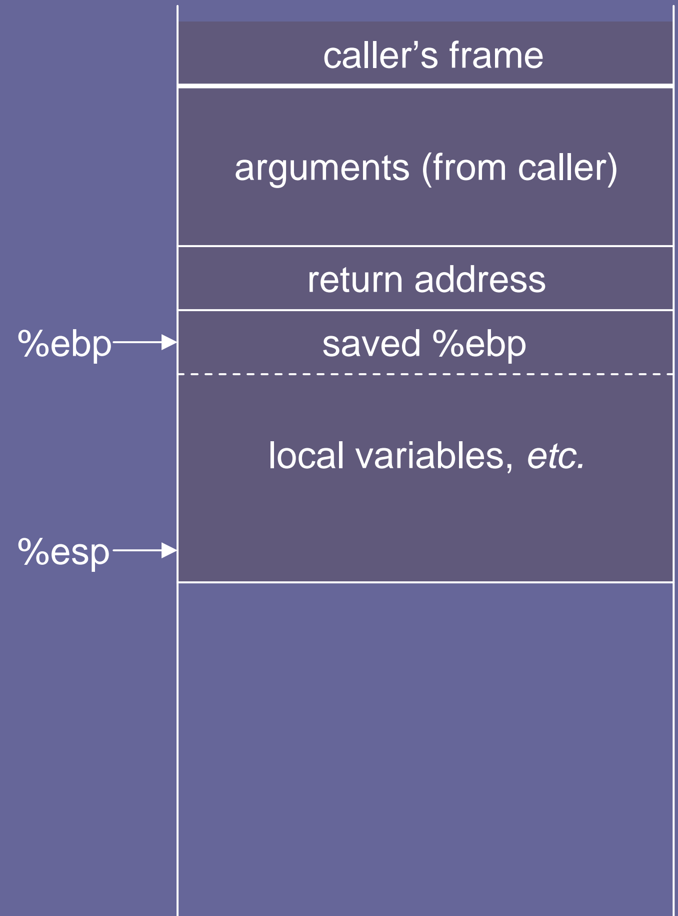
- For convenience, use frame pointer register (%ebp) to point to fixed place in frame.
- Access args, vars by offset from %ebp.
  - *i*th argument: 4*i*+1(%ebp)
  - *j*th local: -4*j*(%ebp)
- %esp now free to move as necessary.
  - Push args for “outgoing calls”.
  - alloca() function. (Look it up.)



Frame Pointer

- But caller was also using ebp!
  - Save caller's value in stack, just below return address.
- Function prologue/epilogue:

```
f:  pushl %ebp
    mov %esp,%ebp
    sub LOCALVARSIZE,%esp
    …
    mov %ebp,%esp
    popl %ebp
    ret
```



Registers?

- What about other registers?
- They’re “global” – every procedure gets to use them.
- What happens to them when calling a procedure?

  Two choices:
  - Callee allowed to overwrite.
    (Caller must save important values on stack.)
  - Callee required to preserve.
    (Callee saves caller’s values before using registers.)
- *Is one better? Is there a tradeoff? What should we do?*

Registers?

- What about other registers?
- They’re “global” – every procedure gets to use them.
- What happens to them when calling a procedure?

  Two choices:
  - Callee allowed to overwrite.
    (Caller must save important values on stack.)
  - Callee required to preserve.
    (Callee saves caller’s values before using registers.)
- *Is one better? Is there a tradeoff? What should we do?*

  Tradeoff: wasted time/space if unused values get saved.
  Caller doesn’t know which regs callee will use.
  Callee doesn’t know which regs caller was using.

Register Saving Policies

Since there's a tradeoff, use hybrid approach.

- %ebp and %esp take care of themselves.

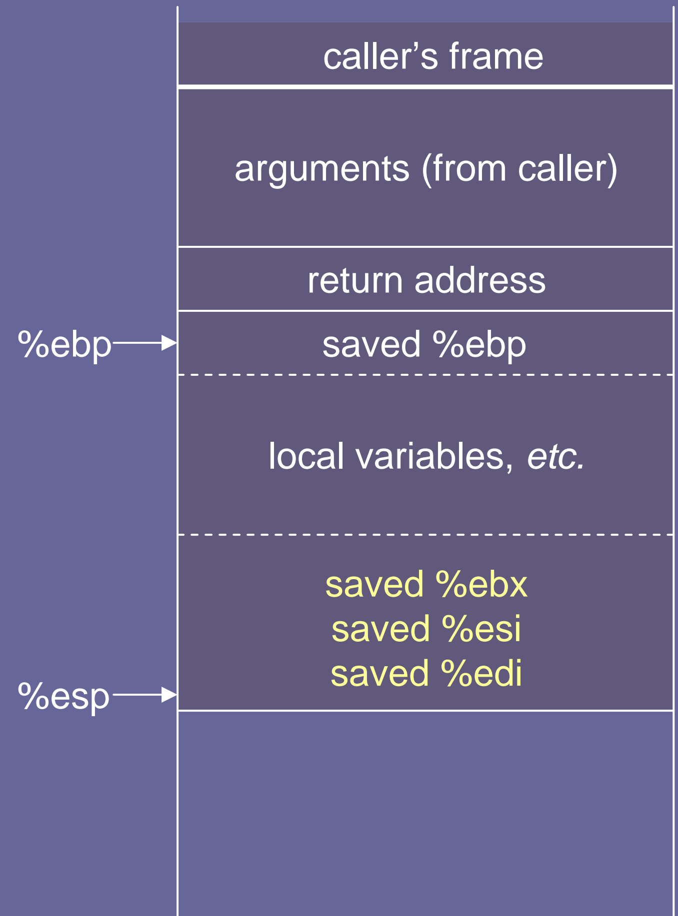
- %eax, %ecx, %edx are "scratch" registers.
  - Any procedure can overwrite without saving.
  - Therefore, must save important values before call. ("caller-saves")
    But don't bother if value doesn't have to survive.

- %ebx, %esi, %edi are "callee-saves" registers.
  - Procedures must preserve their values.
    Save to stack before overwriting, restore on return.
    (Don't bother saving if won't use register.)
  - Safe to leave data there when making calls.

Frame Pointer

- Function prologue/epilogue:

```
f:  pushl %ebp
    mov %esp,%ebp
    sub LOCALVARSIZE,%esp
    pushl %ebx          # if using
    pushl %esi          # if using
    pushl %edi          # if using
    …
    …
    popl %edi           # if pushed
    popl %esi           # if pushed
    popl %ebx           # if pushed
    mov %ebp,%esp
    popl %ebp
    ret
```



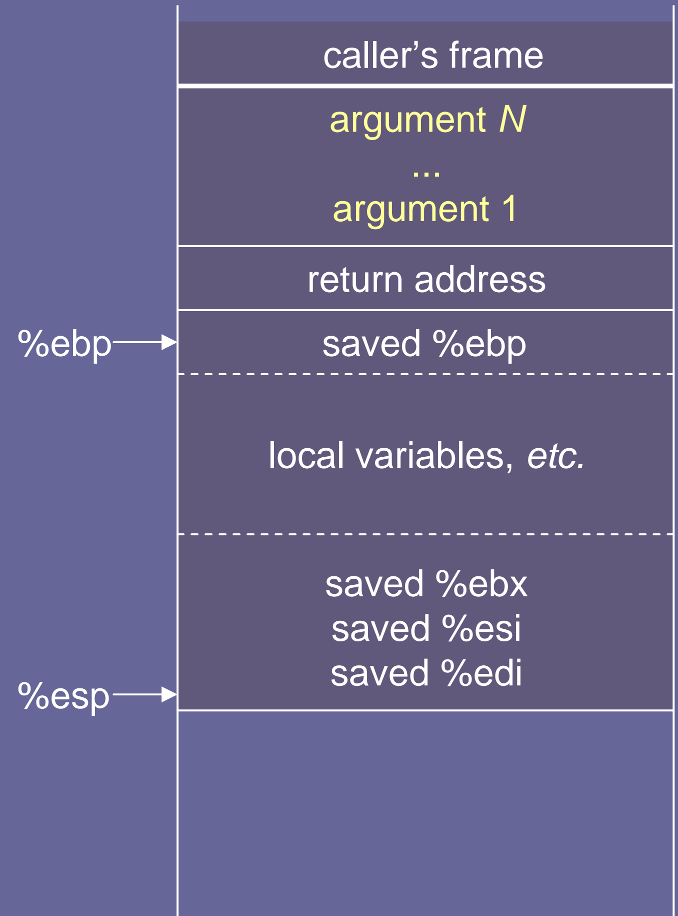
One more thing.

Never mentioned this before:

- Arguments are pushed in reverse order.

Any idea why?

- *Hint: motivated by a feature of C.*



One more thing.

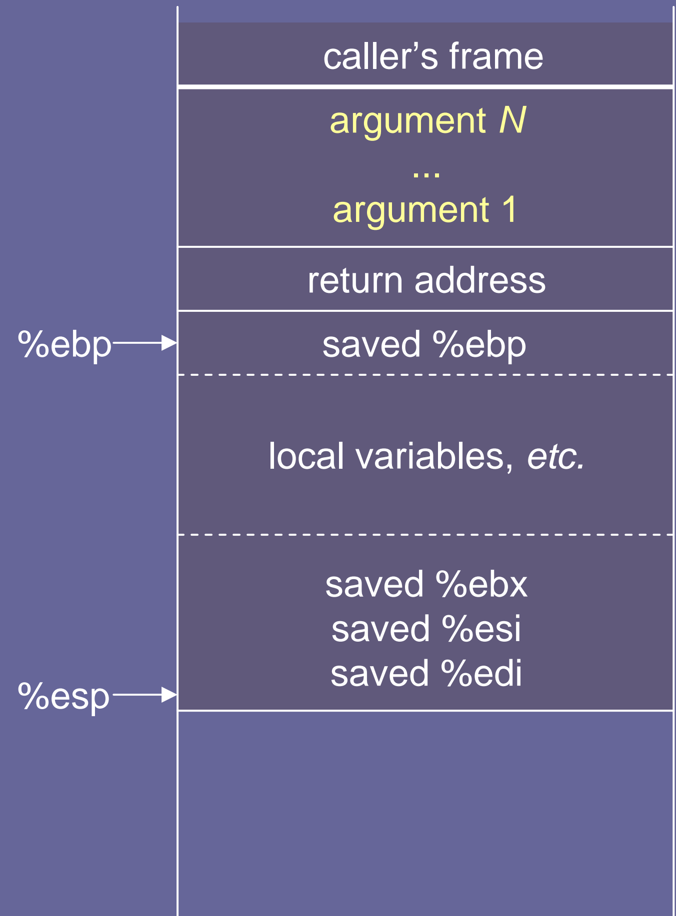
Never mentioned this before:

- Arguments are pushed in reverse order.

Any idea why?

- *Hint: motivated by a feature of C.*

Procedures with variable # of arguments. Helpful to have *first* argument at known location.



So much for procedures.

- Still some things not covered.
  - Those variable-argument procedures.
  - Passing/returning values smaller or larger than 32 bits.
  - Passing/returning floating-point values.
    - Haven’t talked about floating-point *at all*.