# COEN 175

Lecture 15: Stack-Allocated Variables

# Parameter Passing

- Who evaluates the arguments to the called function?
  - The caller evaluates them since they are passed by value.
- How are arguments communicated to the callee?
  - Some arguments may be placed in registers.
  - Other arguments will be placed on the stack.
- Who evaluates the return value?
  - The callee evaluates the return value.
- How is the return value communicated to the caller?
  - The value is typically placed in a register (assuming it fits).

# Calling Conventions

- The **calling conventions** of the system dictate how the registers and the stack are used.
  - Which registers are used to pass arguments?
  - What if an argument is too large for a single register?
  - Where are extra arguments placed on the stack?
  - Which register is used to return a value?
  - Where is a return value placed if it does not fit in a register?
  - Which registers can be modified by the callee?
- These conventions depend on the OS used.
  - Unix and Microsoft systems have different conventions.

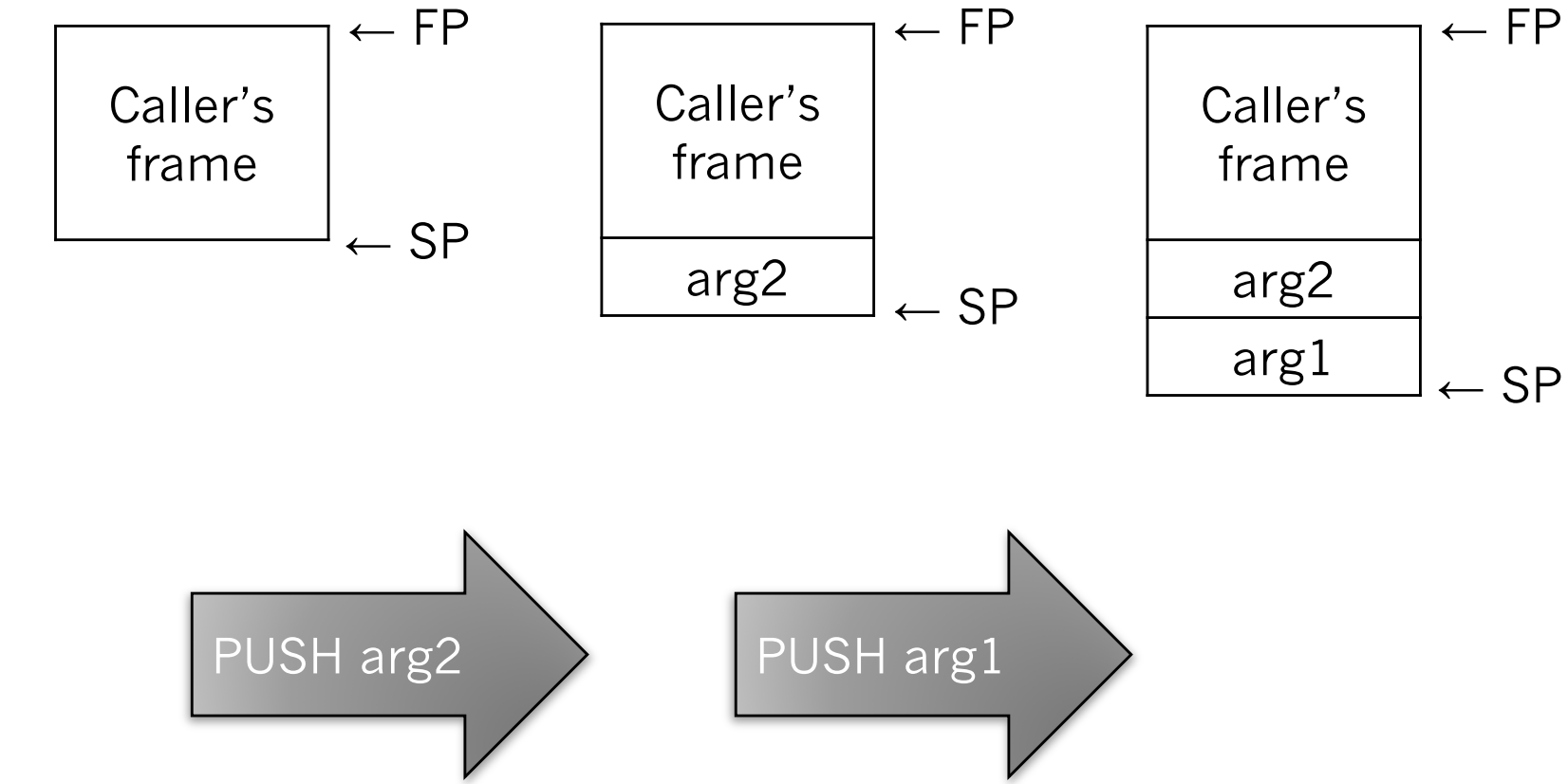
# Intel Register Names

- The original 16-bit Intel register names:
  - `AX` – accumulator
  - `BX` – base register
  - `CX` – count register
  - `DX` – data register
  - `SI` – source index
  - `DI` – destination index
- The 32-bit versions are called `EAX`, `EBX`, etc.
- The 64-bit versions are called `RAX`, `RBX`, etc.
  - The additional 8 registers are simply `R8–R15`.
  - The 32-bit versions of these are called `R8d–R15d`.

# Linux Calling Conventions

- No arguments are passed in registers.
- All arguments are placed on the stack.
  - Arguments are pushed on the stack from **right to left**.
  - Arguments must be a multiple of 4 bytes.
  - The stack must be kept aligned on a 4-byte boundary. (Note that OS X requires 16-byte alignment!)
  - The first argument is +8 bytes from the base pointer.
- An integer return value is placed in `EAX`.
  - A floating value is returned differently using the floating-point co-processor stack.

# Stack-Based Parameters

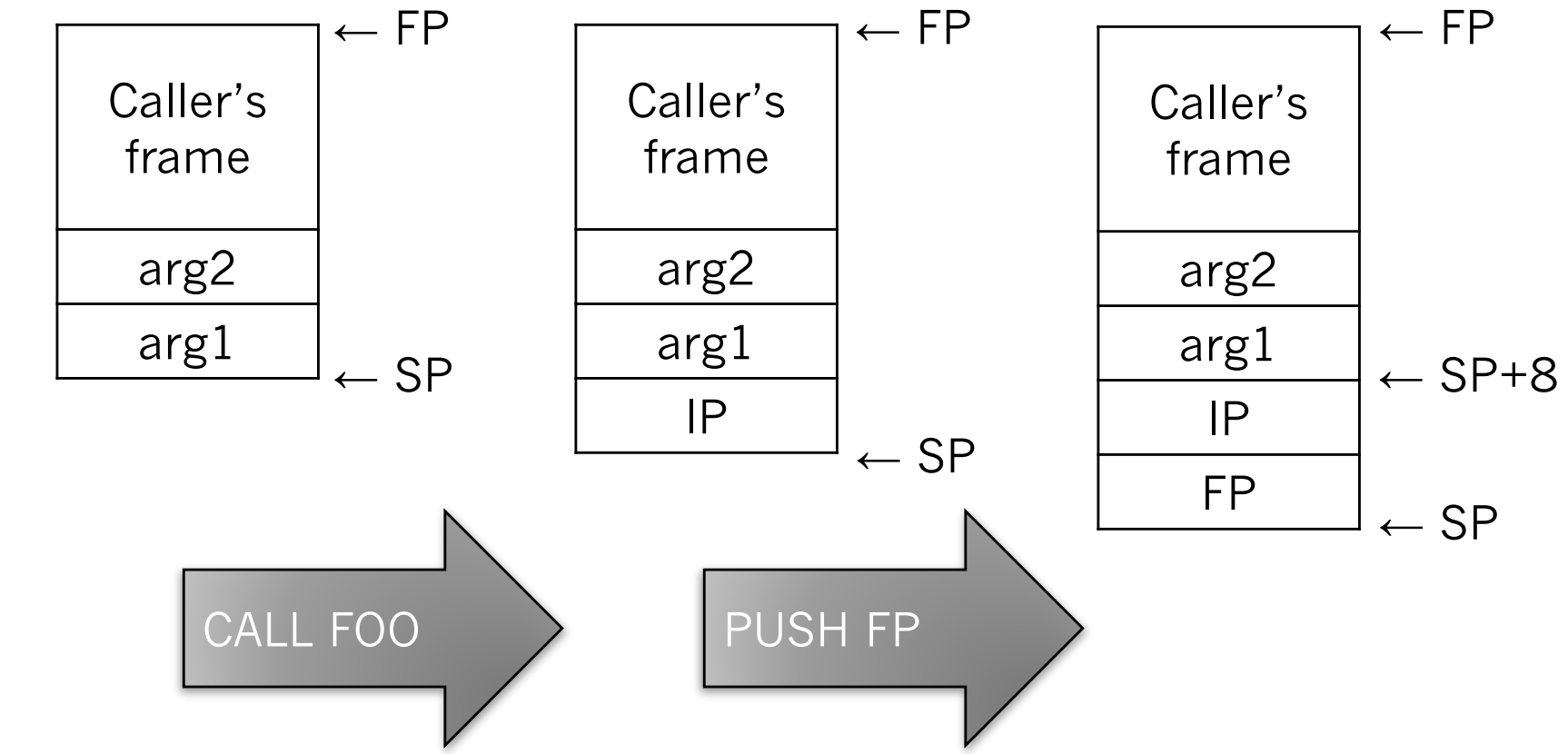



Before call to `foo(arg1,arg2)`

Push rightmost arg

Push next argument

# Stack-Based Parameters




After arguments  Saved instruction pointer  Saved frame pointer

# Referencing Locals

- Local variables are stored within the stack frame, which could be located anywhere in memory.
- Thus, we cannot use absolute or fixed addressing.
- Instead, we must use relative addressing.
  - Also called base-offset addressing.
- Relative to which base register? FP or SP?
  - SP might move during function execution.
  - FP will not move during function execution.
  - So, FP is a better choice.

# Assigning Offsets

- Local variables will be referenced via negative offsets from the frame pointer.
- For simplicity, local variables are typically laid out within the frame in declaration order:
  - Keep track of the last offset in a global variable `offset`.
  - Assign the next local variable, `x`, its offset as follows:

```
offset = offset - x.size
x.offset = offset
```

  - For example, if the first variable requires 4 bytes, it would be at offset -4. If it required 8 bytes, it would be at offset -8.

# Example: Offsets

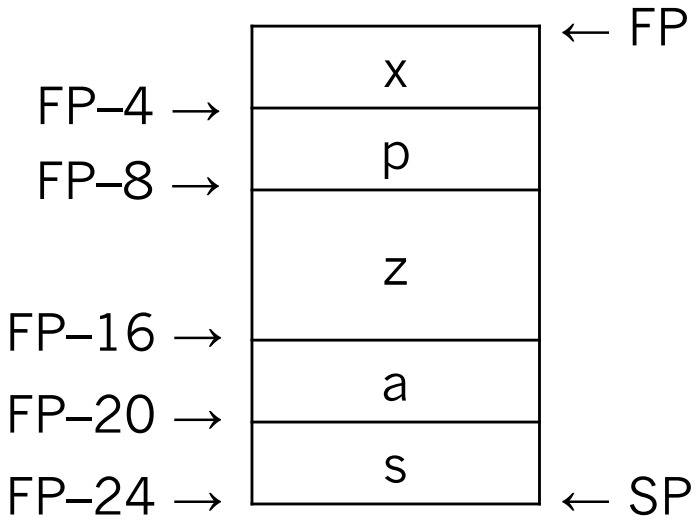
- Assign offsets to each variable:

```
int q, a[10];                 // none: q and a are static

int f(int i, int j) {         // i = +8, j = +12
    int x, *p;                // x = -4, p = -8
    double z;                 // z = -16
    char a[4], *s;            // a = -20, s = -24
}
```

# Example: Offsets

- Assign offsets to each variable:

```
int q, a[10];

int f(int i, int j) {
    int x, *p;
    double z;
    char a[4], *s;
}
```




- What if `f()` were to call another function?
- What would happen to any registers in use?

# Register Usage

- There is only one set of registers that must be shared amongst all functions.
- Most calling conventions divide up the registers:
  - **Caller-saved** – the register's value must be saved on the stack before a function call and restored after the call;
  - **Callee-saved** – the register's value must be saved before it is first used and restored before returning;
  - Any argument and return registers must be caller-saved.
- Intel 32-bit convention: `EAX`, `ECX`, and `EDX` are caller-saved; `EBX`, `ESI`, and `EDI` are callee-saved.

# Another Example: Offsets

- Assign offsets to each variable:

```
int f(int i, int j) {       // i = +8, j = +12
    char x, y, *p;          // x = -1, y = -2, p = -6
    double z;               // z = -14
    char a[5], *s;          // a = -19, s = -23
}
```

- Some of our objects are not **aligned**.
  - Not all architectures require alignment.
  - An unaligned memory access will trigger a **bus error**.
  - Intel does not require alignment, but aligned accesses run faster as unaligned accesses require multiple fetches.

# Another Example: Alignment

- Assign offsets to each variable:

```
int f(int i, int j) {       // i = +8, j = +12
    char x, y, *p;          // x = -1, y = -2, p = -8
    double z;               // z = -16
    char a[5], *s;          // a = -21, s = -28
}
```

- Assume memory accesses should be aligned:
  - A 4-byte object should be aligned on a 4-byte boundary;
  - An 8-byte object should be aligned on an 8-byte boundary;
  - A byte object is always aligned.

# Another Example: Alignment

- Assign offsets to each variable:

```
int f(int i, int j) {
    char x, y, *p;
    double z;
    char a[5], *s;
}
```