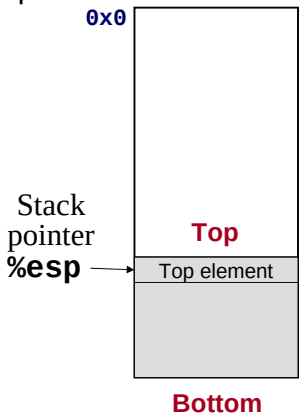
Procedure Calls

CS 217

IA32/Linux Stack

- Memory managed with stack discipline
- Register **%esp** stores the address of top element

Instructions	Functions
pushl src	Fetch data at src Decrement **%esp** by 4 **movl src, (%esp)**
popl dest	**movl (%esp), dest** Increment **%esp** by 4



Procedure Calls

- Calling a procedure involves following actions
  - pass arguments
  - save a return address
  - transfer control to callee
  - transfer control back to caller
  - return results

```
int add3(int a, int b, int c)
{
   return a + b + c;
}

foo(void) {
   int d;
   d = add3( 3, 4, 5 );
   return d;
}
```


Procedure Calls

- Requirements
  - Make a call to an arbitrary address
  - Return back after the call sequence
  - Handle nested procedure calls
  - Save and restore caller's registers
  - Pass an arbitrary number of arguments
  - Pass and return structures
  - Allocate and deallocate space for local variables
- Procedure call and return instruction sequences collaborate to implement these requirements

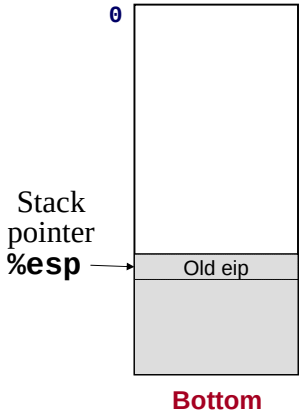
Procedure Calls

- Requirements
  - ➢ Make a call to an arbitrary address
  - ➢ Return back after the call sequence
  - ◦ Handle nested procedure calls
  - ◦ Save and restore caller's registers
  - ◦ Pass an arbitrary number of arguments
  - ◦ Pass and return structures
  - ◦ Allocate and deallocate space for local variables
- Procedure call and return instruction sequences collaborate to implement these requirements

Call and Return Instructions

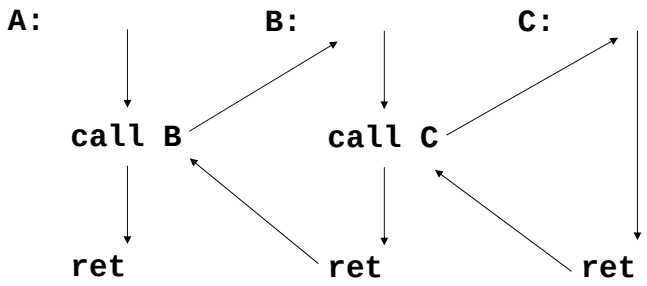
- Procedure call
  - ◦ Push the return address on the stack
  - ◦ Jump to the procedure location
- Procedure return
  - ◦ Pop the return address off the stack
  - ◦ Jump to the return address
- Why using a stack?

Instructions	Functions
`call addr`	`pushl %eip` `jmp addr`
`ret`	`pop %eip`



Nested Procedure Call

- A calls B, which calls C
- Must even work when B is A

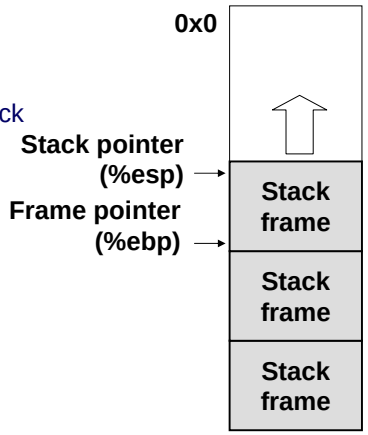


Procedure Calls

- Requirements
  - ◦ Make a call to an arbitrary address
  - ◦ Return back after the call sequence
  - ➢ Handle nested procedure calls
  - ➢ Save and restore caller's registers
  - ◦ Pass an arbitrary number of arguments
  - ◦ Pass and return structures
  - ◦ Allocate and deallocate space for local variables
- Procedure call and return sequences collaborate to implement these requirements

Procedure Stack Structure

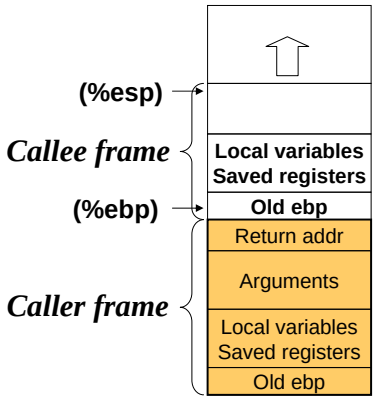
- Stack frame
  - Each procedure call has a stack frame
  - Deal with nested procedure calls
- Stack Pointer
  - Register **%esp**
  - Point to the top element of the stack
- Frame Pointer
  - Register **%ebp**
  - Start of current stack frame
- Why using a frame pointer?
  - Pop off the entire frame before the procedure call returns



Stack Frame in Detail

- Callee stack frame
  - Parameters for called functions
  - Local variables
    - If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address
    - Pushed by "`call`" instruction
  - Arguments for this call
- Before return, use "`leave`" instruction, which does

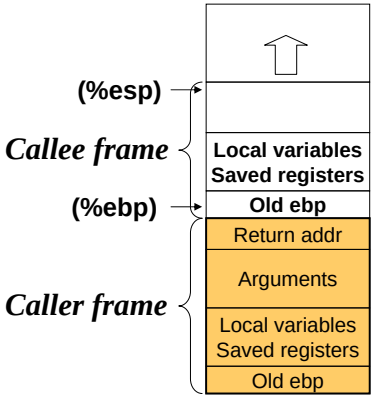
```
movl %ebp, %esp
popl %ebp
```



Procedure (Callee)

```
    .text
    .globl Foo
Foo:
    pushl %ebp
    movl  %esp, %ebp
    .
    .
    .
    leave
    ret
```

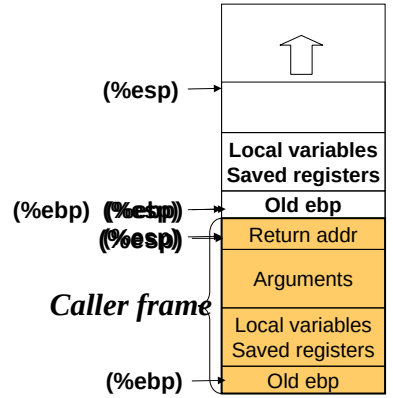
```
movl %ebp, %esp
popl %ebp
```



Procedure (Callee) in Action

```
    .text
    .globl Foo
Foo:
    pushl %ebp
    movl  %esp, %ebp
    .
    .
    .
    leave
    ret
```

```
movl %ebp, %esp
popl %ebp
```



Register Saving Options

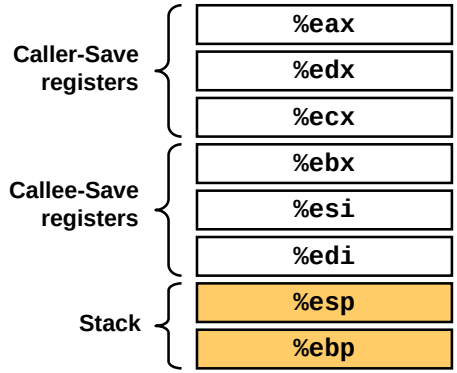
- Problem: a procedure needs to use registers, but
  - If you use the registers, their contents will be changed when returning to the caller
  - If we save registers on the stack, who is responsible?
- Caller Save
  - Caller saves registers in its frame before calling
- "Callee Save"
  - Callee saves registers in its frame before using

```
main:
    • • •
    movl $0x123, %edx
    call Foo
    addl %edx, %eax
    • • •
    ret
```

```
Foo:
    • • •
    movl 8(%ebp), %edx
    addl $0x456, %edx
    • • •
    ret
```


IA32/Linux Register Saving Convention

- Special stack registers
  - **%ebp, %esp**
- Callee-save registers
  - **%ebx, %esi, %edi**
  - Old values saved on stack prior to using
- Caller-save registers
  - **%eax, %edx, %ecx**
  - Save on stack prior to calling

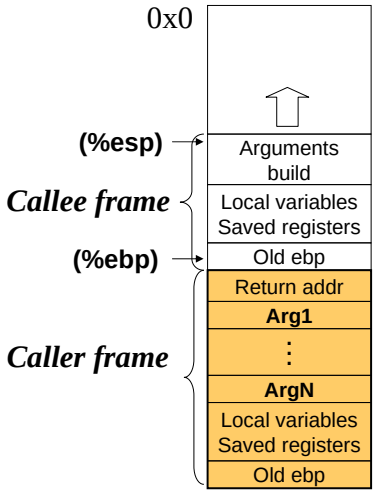


Procedure Calls

- Requirements
  - Set PC to arbitrary address
  - Return PC to instruction after call sequence
  - Handle nested procedure calls
  - Save and restore caller's registers
  - ➢ Pass an arbitrary number of arguments
  - ➢ Pass and return structures
  - ➢ Allocate and deallocate space for local variables
- Procedure call and return sequences collaborate to implement these requirements

Passing Arguments to Procedure

- Arguments are passed on stack in order
  - Push N-th argument first
  - Push 1st argument last
- Callee references the argument by
  - 1st argument: **8(%ebp)**
  - 2nd argument: **12(%ebp)**
  - **...**
- Passing result back by **%eax**
  - Caller is responsible for saving **%eax** register



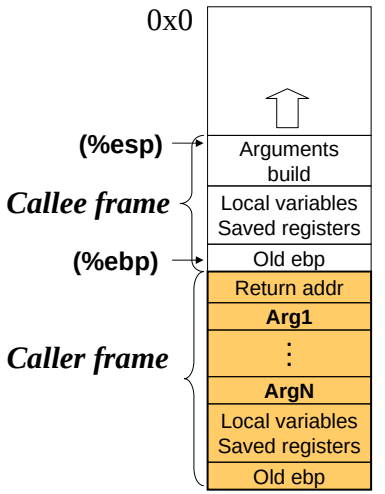
Example: Passing Arguments

```
int d;

int add3(int a, int b, int c)
{
    return a + b + c;
}


foo(void) {
    d = add3( 3, 4, 5 );
    return d;
}
```

```
        .text
        .globl add3
add3:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        addl    16(%ebp), %eax
        leave
        ret
        .globl foo
foo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $5
        pushl   $4
        pushl   $3
        call    add3
        movl    %eax, d
        leave
        ret
        .comm   d,4,4
```


Allocation for Local Variables

- Local variables are stored in a stack frame
- Allocation is done by moving the stack pointer **%esp**
  `subl $4, %esp`
- Reference local variable by using register **%ebp**
  `e.g. -4(%ebp)`



Example: Local Variables

```
int add3(int a, int b, int c)
{
    return a + b + c;
}


foo(void) {
    int d;
    d = add3( 3, 4, 5 );
    return d;
}
```

```
        .text
        .globl add3
add3:
        pushl %ebp
        movl %esp, %ebp
        movl 12(%ebp), %eax
        addl 8(%ebp), %eax
        addl 16(%ebp), %eax
        leave
        ret
        .globl foo
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        pushl $5
        pushl $4
        pushl $3
        call add3
        movl %eax, -4(ebp)
        leave
        ret
```


Summary

- Issues related to calling conventions
  - Stack frame for caller and callee
  - Use esp and ebp registers
  - Passing arguments on stack
  - Saving registers on stack (caller save and callee save)
  - Local variables on stack
  - Passing result in eax register
- Procedure call instructions
  `call, ret, leave`