Procedure Calls

CS 217

Procedure Calls

- Procedure: a piece of code with a well-defined interface, and well-defined entry and exit points
- Call: jump to the beginning of an arbitrary procedure
- Return: jump back to the caller
- The jump address in the return operation is dynamically determined
  - Jump to the instruction immediately following the Call instruction in the Caller

Implementing Procedure Calls

P: # Proc P
   ...
   jmp R # Call R
Rtn_point1:
   ...

Q: # Proc Q
   ...
   jmp R # Call R
Rtn_point2:
   ...

What should the return instruction in R jump to?

Implementing Procedure Calls

P: # Proc P
   ...
   jmp %eax # Return

R: # Proc R
   ...

movl $Rtn_point1, %eax
jmp R # Call R
Rtn_point1:
   ...

Q: # Proc Q
   ...
   jmp %eax # Return

movl $Rtn_point2, %eax
jmp R # Call R
Rtn_point2:
   ...

Convention: At Call time, store return address in %eax
Problem: Nested Procedure Calls

P: # Proc P
   movl $Rtn_point1, %eax
   jmp Q     # Call Q
Rtn_point1:
   ...

Q: # Proc Q
   movl $Rtn_point2, %eax
   jmp R     # Call R
Rtn_point2:
   ...
   jmp %eax # Return

R: # Proc R
   ...
   jmp %eax # Return

• Problem if P calls Q, and Q calls R
• Return address for P to Q call is lost

Need to use a Stack

• P calls Q, Q calls R, R calls S, S calls P again
• A return address needs to be saved for as long as the procedure invocation continues
• Return addresses are used in Last-In-First-Out order
• Stack is a natural solution

Stack Frames

• Use stack for all temporary data related to each active procedure invocation
  
  • Return address
  • Procedure arguments
  • Local variables of procedures
  • Saving registers across invocations

Stack Frame

• Stack has one Stack Frame for each active procedure invocation

High-Level Picture

• At Call time, push a new Stack Frame on top of the stack
• At Return time, pop the top-most Stack Frame
**High-Level Picture**

main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
Q returns
P returns

%ESP → main’s Stack Frame

**Procedure Call Details**

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

**Call and Return Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td></td>
</tr>
<tr>
<td>subl $4, %esp</td>
<td></td>
</tr>
<tr>
<td>movl src, (%esp)</td>
<td></td>
</tr>
<tr>
<td>popl dest</td>
<td></td>
</tr>
<tr>
<td>movl (%esp), dest</td>
<td></td>
</tr>
<tr>
<td>addl $4, %esp</td>
<td></td>
</tr>
<tr>
<td>call addr</td>
<td></td>
</tr>
<tr>
<td>pushl %eip</td>
<td></td>
</tr>
<tr>
<td>jmp addr</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>pop %eip</td>
<td></td>
</tr>
</tbody>
</table>

%ESP → before Call
### Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
<tr>
<td>ret</td>
<td>pop %eip</td>
</tr>
</tbody>
</table>

Return instruction assumes that the return address is at the top of the stack.

### Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl src</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl src, (%esp)</td>
</tr>
<tr>
<td>popl dest</td>
<td>movl (%esp), dest</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call addr</td>
<td>pushl %eip</td>
</tr>
<tr>
<td></td>
<td>jmp addr</td>
</tr>
<tr>
<td>ret</td>
<td>pop %eip</td>
</tr>
</tbody>
</table>

Return instruction assumes that the return address is at the top of the stack.

### Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions
• Caller pushes input parameters before executing the Call instruction
• Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

Callee can address arguments relative to ESP: Arg 1 as 4(%esp)
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

After the procedure call is finished, the caller pops the pushed arguments from the stack

Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push Nth argument first
  - Push 1st argument last
  - So that the first argument is at the top of the stack at the time of the Call

After the procedure call is finished, the caller pops the pushed arguments from the stack

Input Parameters

- Why push parameters in reverse order?
- Allows variable number of parameters
- Arg 1 is always at 4(%esp)
- Arg 1 describes the other parameters

printf("A string %s and a number %d", str, num);

Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions
**Base Pointer: EBP**

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing
  - `pushl %ebp`
  - `movl %esp, %ebp`

**Base Pointer: EBP**

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing
  - `pushl %ebp`
  - `movl %esp, %ebp`

**Base Pointer: EBP**

- As Callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and other local variables
- Need to save old value of EBP before using EBP
- Callee begins by executing
  - `pushl %ebp`
  - `movl %esp, %ebp`

  Regardless of ESP, Callee can address Arg 1 as 8(%ebp)

**Base Pointer: EBP**

- Before returning, Callee must restore EBP to its old value
- Executes
  - `movl %ebp, %esp`
  - `popl %ebp`
  - `ret`
Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
  
  - Executes
    
    ```
    movl %ebp, %esp
    popl %ebp
    ret
    ```

Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
  
  - Executes
    
    ```
    movl %ebp, %esp
    popl %ebp
    ret
    ```

Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
  
  - Executes
    
    ```
    movl %ebp, %esp
    popl %ebp
    ret
    ```

Procedure Call Details

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions
**Allocation for Local Variables**

- Local variables of the Callee are also allocated on the stack.
- Allocation done by moving the stack pointer.
- Example: allocate two integers
  - `subl $4, %esp`
  - `subl $4, %esp`
  - (or equivalently, `subl $8, %esp`)
- Reference local variables using the base pointer
  - `-4(%ebp)`
  - `-8(%ebp)`

**Procedure Call Details**

- Call and Return instructions
- Argument passing between procedures
- Base pointer management (EBP)
- Local variables
- Register saving conventions

**Use of Registers**

- Problem: Callee may use a register that the caller is also using
  - When callee returns control to caller, old register contents may be lost. BAD!
  - Someone must save old register contents and later restore
- Need a convention for who saves and restores which registers

**GCC/Linux Convention**

- Caller-save registers
  - `%eax, %edx, %ecx`
  - Save on stack prior to calling
  - Restore after return
- Callee-save registers
  - `%ebx, %esi, %edi`
  - Old values saved on stack prior to using
  - Restored after using
- `%esp, %ebp` handled as described earlier
- Return value is passed from Callee to Caller in `%eax`
A Simple Example

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

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

A Simple Example

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

add3:
    # Save old ebp, and set-up
    # new ebp
    pushl %ebp
    movl %esp, %ebp
    # Allocate space for d
    subl $4, %esp
    # In general, one may need
    # to push Callee-save
    # registers onto the stack
    movl %ebp, %esp
    # Return
    ret
```

A Simple Example

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

```c
foo:
    # Save old ebp, and set-up
    # new ebp
    pushl %ebp
    movl %esp, %ebp
    # No local variables
    # No need to save Callee-
    # save registers as we
    # don’t use any registers
    call add3
    # Return value is already
    # in eax
    # Restore old ebp and
    # discard stack frame
    movl %ebp, %esp
    popl %ebp
    # Return
    ret
```

System Calls

```asm
.section .data
    # pre-initialized variables
    # go here

.section .bss
    # zero-initialized variables
    # go here

.section .rodata
    # pre-initialized constants
    # go here

.section .text
.globl _start
_start:
    # Program starts executing
    # here
    # Body of the program goes
    # here
    .section .rodata
    # Program ends with an
    # "exit()" system call
    # to the operating system
    movl $1, %eax
    int $0x80
```
Summary

• Temporary data related to procedure invocations is stored on the stack.
• Stack Frame for a procedure invocation includes return address, procedure arguments, local variables and saved registers.
• Call and Ret instructions implement procedure calls.
• Base pointer EBP is used as a fixed reference point in the Stack Frame.
• Arguments and local variables are addressed relative to the base pointer.