Function Calls

COS 217

Reading: Chapter 4 of “Programming From the Ground Up”
(available online from the course Web site)
Goals of Today’s Lecture

• Finishing introduction to assembly language
  ○ EFLAGS register and conditional jumps
  ○ Addressing modes

• Memory layout of the UNIX process
  ○ Data, BSS, roData, Text
  ○ Stack frames, and the stack pointer ESP

• Calling functions
  ○ Call and ret commands
  ○ Placing arguments on the stack
  ○ Using the base pointer EBP
Detailed Example

count=0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}

```assembly
movl $0, %ecx

.loop:
cmpl $1, %edx
jle .endloop
addl $1, %ecx
movl %edx, %eax
addl %eax, %edx
addl %eax, %edx
addl $1, %edx
addl $1, %ecx
jmp .loop

.endloop:
```
Setting the EFLAGS Register

• **Comparison** `cmpl` compares two integers
  - Done by subtracting the first number from the second
    - Discarding the results, but setting the eflags register
  - Example:
    - `cmpl $1, %edx` (computes %edx – 1)
    - `jle .endloop` (looks at the sign flag and the zero flag)

• **Logical operation** `andl` compares two integers
  - Example:
    - `andl $1, %eax` (bit-wise AND of %eax with 1)
    - `je .else` (looks at the zero flag)

• **Unconditional branch** `jmp`
  - Example:
    - `jmp .endif` and `jmp .loop`
EFLAGS Register & Condition Codes

Reserved (set to 0)

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Flag Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Identification flag</td>
</tr>
<tr>
<td>30</td>
<td>Virtual interrupt pending</td>
</tr>
<tr>
<td>29</td>
<td>Virtual interrupt flag</td>
</tr>
<tr>
<td>28</td>
<td>Alignment check</td>
</tr>
<tr>
<td>27</td>
<td>Virtual 8086 mode</td>
</tr>
<tr>
<td>26</td>
<td>Resume flag</td>
</tr>
<tr>
<td>25</td>
<td>Nested task flag</td>
</tr>
<tr>
<td>24</td>
<td>I/O privilege level</td>
</tr>
<tr>
<td>23</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>22</td>
<td>Direction flag</td>
</tr>
<tr>
<td>21</td>
<td>Interrupt enable flag</td>
</tr>
<tr>
<td>20</td>
<td>Trap flag</td>
</tr>
<tr>
<td>19</td>
<td>Sign flag</td>
</tr>
<tr>
<td>18</td>
<td>Zero flag</td>
</tr>
<tr>
<td>17</td>
<td>Auxiliary carry flag</td>
</tr>
<tr>
<td>16</td>
<td>Parity flag</td>
</tr>
<tr>
<td>15</td>
<td>Carry flag</td>
</tr>
</tbody>
</table>

Carry flag
Identification flag
Virtual interrupt pending
Virtual interrupt flag
Alignment check
Virtual 8086 mode
Resume flag
Nested task flag
I/O privilege level
Overflow flag
Direction flag
Interrupt enable flag
Trap flag
Sign flag
Zero flag
Auxiliary carry flag or adjust flag
Parity flag
Carry flag
Data Access Methods

- **Immediate addressing:** data stored in the instruction itself
  - `movl $10, %ecx`

- **Register addressing:** data stored in a register
  - `movl %eax, %ecx`

- **Direct addressing:** address stored in instruction
  - `movl 2000, %ecx`

- **Indirect addressing:** address stored in a register
  - `movl (%eax), %ebx`

- **Base pointer addressing:** includes an offset as well
  - `movl 4(%eax), %ebx`

- **Indexed addressing:** instruction contains base address, and specifies an index register and a multiplier (1, 2, or 4)
  - `movl 2000(%ecx,1), %ebx`
Effective Address

\[
\text{Offset} = \left( \begin{array}{c}
\text{eax} \\
\text{ebx} \\
\text{ecx} \\
\text{edx} \\
\text{esp} \\
\text{ebp} \\
\text{esi} \\
\text{edi}
\end{array} \right) + \left( \begin{array}{c}
\text{eax} \\
\text{ebx} \\
\text{ecx} \\
\text{edx} \\
\text{esp} \\
\text{ebp} \\
\text{esi} \\
\text{edi}
\end{array} \right) \times \left( \begin{array}{c}
1 \\
2 \\
3 \\
4
\end{array} \right) + \left( \begin{array}{c}
\text{None} \\
\text{8-bit} \\
\text{16-bit} \\
\text{32-bit}
\end{array} \right)
\]

- **Displacement**
  \[\text{movl foo, } %eax\]

- **Base**
  \[\text{movl (} %eax, %ebx \text{)}\]

- **Base + displacement**
  \[\text{movl foo(} %eax, %ebx \text{)}\]
  \[\text{movl 1(} %eax, %ebx \text{)}\]

- **(Index \times scale) + displacement**
  \[\text{movl (,} %eax, 4, %ebx \text{)}\]

- **Base + (index \times scale) + displacement**
  \[\text{movl foo(,} %eax, 4, %ebx \text{)}\]
A Simple Assembly Program

```
.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

# Program ends with an
# “exit()” system call
# to the operating system
movl $1, %eax
movl $0, %ebx
int $0x80
```
Main Parts of the Program

• Break program into sections (.section)
  ○ Data, BSS, RoData, and Text

• Starting the program
  ○ Making _start a global (.global _start)
    – Tells the assembler to remember the symbol _start
    – … because the linker will need it
  ○ Identifying the start of the program (_start)
    – Defines the value of the label _start

• Exiting the program
  ○ Specifying the exit() system call (movl $1, %eax)
    – Linux expects the system call number in EAX register
  ○ Specifying the status code (movl $0, %ebx)
    – Linux expects the status code in EBX register
  ○ Interrupting the operating system (int $0x80)
Function Calls

• Function
  ○ A piece of code with well-defined entry and exit points, and a well-defined interface

• “Call” and “Return” abstractions
  ○ Call: jump to the beginning of an arbitrary procedure
  ○ Return: jump to the instruction immediately following the “most-recently-executed” Call instruction

• The jump address in the return operation is dynamically determined
Implementing Function Calls

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

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

R: # Function R
  ...
  jmp ??? # Return

What should the return instruction in R jump to?
Implementing Function Calls

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

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

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

Convention: At Call time, store return address in EAX
Problem: Nested Function Calls

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

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

R: # Function 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

- A return address needs to be saved for as long as the function invocation continues
- Return addresses are used in the reverse order that they are generated: Last-In-First-Out
- The number of return addresses that may need to be saved is not statically known
- Saving return addresses on a Stack is the most natural solution
Stack Frames

- Use stack for all temporary data related to each active function invocation
  - Return address
  - Input parameters
  - Local variables of function
  - Saving registers across invocations

- Stack has one Stack Frame per active function 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

%ESP → Bottom

main’s Stack Frame
main begins executing
main calls P

%ESP →

P’s
Stack Frame

main’s
Stack Frame

Bottom
main begins executing
main calls P
P calls Q
main begins executing
main calls P
P calls Q
Q calls P
High-Level Picture

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

%ESP

Q’s
Stack Frame
P’s
Stack Frame
main’s
Stack Frame
Bottom
High-Level Picture

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

%ESP
0

Bottom

main’s Stack Frame
P’s Stack Frame
Q’s Stack Frame
R’s Stack Frame
High-Level Picture

main begins executing
main calls P
P calls Q
Q calls P
P returns
Q calls R
R returns
High-Level Picture

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

main’s 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
main returns
Function Call Details

- Call and Return instructions
- Argument passing between procedures
- Local variables
- Register saving conventions
Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
</table>
| pushl src   | subl $4, %esp  
|            | movl src, (%esp) |
| popl dest   | movl (%esp), dest  
|            | addl $4, %esp |
| call addr   | pushl %eip  
|            | jmp addr |
| ret         | pop %eip |
## Call and Return Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</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>

After call, the ESP points to the old EIP.
# Call and Return Instructions

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

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

• Caller pushes input parameters before executing the Call instruction

• Parameters are pushed in the reverse order
  ○ Push N<sup>th</sup> argument first
  ○ Push 1<sup>st</sup> argument last
  ○ So that the first argument is at the top of the stack at the time of the Call

%ESP before pushing arguments
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

```plaintext
%ESP
\downarrow
0
```

- Arg N
- Arg ...
- Arg 1
Input Parameters

- Caller pushes input parameters before executing the Call instruction
- Parameters are pushed in the reverse order
  - Push $N^{th}$ argument first
  - Push $1^{st}$ 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 $N^{th}$ argument first
  ○ Push 1$^{st}$ argument last
  ○ So that the first argument is at the top of the stack at the time of the Call
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 function 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 N\textsuperscript{th} argument first
  - Push 1\textsuperscript{st} argument last
  - So that the first argument is at the top of the stack at the time of the Call

After the function call is finished, the caller pops the pushed arguments from the stack
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
  ```
**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).
Before returning, Callee must restore EBP to its old value.

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

• Executes

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

• Executes

```assembly
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
  ```
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)
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
  ◦ 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
- **Callee-save registers**
  - %ebx, %esi, %edi
  - Old values saved on stack prior to using
- %esp, %ebp handled as described earlier
- Return value is passed from Callee to Caller in %eax
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;
}
```

In general, one may need to push callee-save registers onto the stack

Add the three arguments
```
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax
```

Put the sum into d
```
movl %eax, -4(%ebp)
```

Return value is already in eax

In general, one may need to pop callee-save registers

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

Return
```
ret
```
A Simple Example

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

**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**

- **No need to save caller-save registers either**

- **Push arguments in reverse order**
  - `pushl $5`
  - `pushl $4`
  - `pushl $3`
  - `call add3`

- **Return value is already in eax**

- **Restore old ebp and discard stack frame**
  - `movl %ebp, %esp`
  - `popl %ebp`

- **Return**
  - `ret`
Conclusion

• Invoking a function
  ◦ Call: call the function
  ◦ Ret: return from the instruction

• Stack Frame for a function invocation includes
  ◦ Return address,
  ◦ Procedure arguments,
  ◦ Local variables, and
  ◦ Saved registers

• Base pointer EBP
  ◦ Fixed reference point in the Stack Frame
  ◦ Useful for referencing arguments and local variables