CMPT 295
Machine-Level Programming
Lecture 15 – Function Call and Stack- Passing Control
Last Lecture

- In x86-64 assembly, there are no iterative statements.
- To alter the execution flow, compiler generates code sequence that implements these iterative statements (while, do-while and for loops) using branching method:
  - `cmp*` instruction
  - `jX` instructions (jump)
- 2 loop patterns:
  - “coding the false condition first” -> while loops (hence for loops)
  - “jump-in-middle” -> while, do-while (hence for loops)
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `lea` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
- Array
- Floating-point operations
What happens when a function (caller) calls another function (callee)?

- Control is passed (PC is set) …
  - To the beginning of the code in callee function
  - Back to where callee function was called in caller function
- Data is passed …
  - To callee function via function parameter(s)
  - Back to caller function via return value
- Memory is …
  - Allocated during callee function execution
  - Deallocated upon return to caller function
- Above mechanisms implemented with machine instructions as well as set of conventions

```c
void who(...) {
    int sum = 0;
    ...
    y = amI(x);
    sum = x + y;
    return;
}

int amI(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}
```
Remember from Lecture 2: Closer look at memory

- Seen as a linear array of bytes
- 1 byte (8 bits) smallest addressable unit of memory
- Each byte has a unique address
- Computer reads a “word size” worth of bits at a time
- Compressed view of memory

<table>
<thead>
<tr>
<th>M[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size - 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0x0040</td>
</tr>
<tr>
<td>0x0038</td>
</tr>
<tr>
<td>0x0030</td>
</tr>
<tr>
<td>0x0028</td>
</tr>
<tr>
<td>0x0020</td>
</tr>
<tr>
<td>0x0018</td>
</tr>
<tr>
<td>0x0010</td>
</tr>
<tr>
<td>0x0008</td>
</tr>
<tr>
<td>0x0000</td>
</tr>
</tbody>
</table>
Memory Layout

- **Stack**
  - Runtime stack, e.g., local variables

- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `new()`

- **Data**
  - Statically allocated data, e.g., global vars, static vars, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only
where does everything go?

Memory Allocation Example

```c
#include ...

char hugeArray[1 << 31]; /* 2^31 = 2GB */
int global = 0;

int useless(){ return 0; }

int main ()
{
    void *ptr1, *ptr2;
    int local = 0;
    ptr1 = malloc(1 << 28); /* 2^28 = 256 MB*/
    ptr2 = malloc(1 << 8); /* 2^8 = 256 B*/

    /* Some print statements ... */
}
```
Closer look at function call pattern

- A function may call a function, which may call a function, which may call a function, ...

- When a function (callee) terminates and returns, its most recent caller resumes which eventually terminates and returns and its most recent caller resumes ...

- Does this pattern remind you of anything?

```
who(...) {
    ...
    ...
    are();
    ...
    ...
}

are(...) {
    ...
    you();
    ...
    you();
    ...
    ...
}

you(...) {
    ...
    ...
    you();
    ...
    ...
    ...
}
Stack

Definition:
A stack is a last-in-first-out (LIFO) data structure with two characteristic operations:

- push(data)
- data = pop()
Closer look at stack

- x86-64 assembly language has stack-specific instructions and registers
- `%rsp`
  - Points to address of last used byte on stack
  - Initialized to some value at startup
  - Stack grows towards low memory address
- `pushq src`
- `popq dest`
x86-64 stack instruction: **push**

- **pushq src**
  - Fetch value of operand *src*
  - Decrement %rsp by 8
  - Write value at address given by %rsp
x86-64 stack instruction: \texttt{pop}

\begin{itemize}
\item \texttt{popq dest}
\item Read value at address given by \%\texttt{rsp}
\item Increment \%\texttt{rsp} by 8
\item Store value at operand \texttt{dest} (must be register)
\end{itemize}

\begin{itemize}
\item \texttt{... we pop once.}
\end{itemize}
Passing control mechanism
x86-64 instruction: call and ret

- **call func**
- **push PC**
- **jmp func** (set PC to func)

After 1 call ...

Effect: return address, i.e., the address of the instruction after call func (held in PC) is pushed onto the stack
Passing control mechanism

x86-64 instruction: **call** and **ret**

- **ret**
- **popq PC**
- **jmp PC**

After returning from the call ...

Effect: return address, i.e., the address of instruction after `call func`, is pop’ed from the stack and stored in PC

Increasing memory addresses

Stack grows down
Example

```c
long mult2(long a, long b) {
    long s = a * b;
    return s;
}

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
    return;
}
```

```
0000000000400550 <mult2>:
400550:  mov    %rdi,%rax    # a
400553:  imul   %rsi,%rax    # a * b
400557:  retq             # Return

0000000000400540 <multstore>:
400540: push   %rbx         # Save %rbx
400541: mov    %rdx,%rbx    # Save dest
400544: callq  400550 <mult2> # mult2(x,y)
400549: mov    %rax,(%rbx)   # Save at dest
40054c: pop    %rbx         # Restore %rbx
40054d: retq             # Return
```
Example – Steps 1 and 2

0000000000400540 <multstore>:
400540: push %rbx  # Save %rbx
400541: mov %rdx,%rbx  # Save dest
400544: callq 400550 <mult2>  # mult2(x,y)
400549: mov %rax,%rbx  # Save at dest
40054c: pop %rbx  # Restore %rbx
40054d: retq  # Return

0000000000400550 <mult2>:
400550: mov %rdi,%rax  # a
400553: imul %rsi,%rax  # a * b
400557: retq  # Return

%rdi %rbx %rsp 0x120
%rsi %rax %rip 0x400540
%rdx
Example – Steps 3 and 4

000000000000400540 <multstore>:
400540: push %rbx  # Save %rbx
400541: mov %rdx,%rbx  # Save dest
400544: callq 400550 <mult2>  # mult2(x,y)
400549: mov %rax,(%rbx)  # Save at dest
40054c: pop %rbx  # Restore %rbx
40054d: retq  # Return

000000000000400550 <mult2>:
400550: mov %rdi,%rax  # a
400553: imul %rsi,%rax  # a * b
400557: retq  # Return

%rdi  %rbx  %rsp  0x118
%rsi  %rax  %rip  0x400544
%rdx

Stack
Top
ret address
Example – Steps 5 and 6

00000000000400540 <multstore>:
  400540: push %rbx  # Save %rbx
  400541: mov %rdx,%rbx  # Save dest
  400544: callq 400550 <mult2>  # mult2(x,y)
  400549: mov %rax,(%rbx)  # Save at dest
  40054c: pop %rbx  # Restore %rbx
  40054d: retq  # Return

00000000000400550 <mult2>:
  400550:  mov %rdi,%rax  # a
  400553:  imul %rsi,%rax  # a * b
  400557:  retq  # Return

%rdi %rbx %rsp 0x110
%rsi %rax %rip 0x400553
%rdx
Example – Steps 7, 8 and 9

0000000000400540 <multstore>:
400540: push %rbx          # Save %rbx
400541: mov %rdx,%rbx      # Save dest
400544: callq 400550 <mult2> # mult2(x,y)
400549: mov %rax,(%rbx)    # Save at dest
40054c: pop %rbx           # Restore %rbx
40054d: retq               # Return

0000000000400550 <mult2>:
400550:  mov %rdi,%rax     # a
400553:  imul %rsi,%rax    # a * b
400557:  retq             # Return
Summary

- Function call mechanisms: passing control and data, managing memory
- Memory layout
- Stack (local variables …)
- Heap (dynamically allocated data)
- Data (statically allocated data)
- Text / Shared Libraries (program code)
- Stack is the data structure used for function call / return
  - If multstore calls mult2, then mult2 returns before multstore
- x86-64 stack register and instructions: stack pointer \texttt{rsp}, \texttt{push} and \texttt{pop}
- x86-64 function call instructions: \texttt{call} and \texttt{ret}
Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
  - Assembly language basics: data, move operation
  - Memory addressing modes
  - Operation leaq and Arithmetic & logical operations
  - Conditional Statement – Condition Code + cmovX

- Loops

- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data

- Array

- Floating-point operations