Function Calls and Stack

Philipp Koehn

16 April 2018

functions

Another Example

- C code with an undefined function

```
int main(void) {
  int a = 2;
  int b = do_something(a);
  return b;
}
```

Another Example

- C code with an undefined function

```
int main(void) {
  int a = 2;
  int b = do_something(a);
  return b;
}
```

- This can be successfully compiled into an object file

```
linux> gcc -w -Og -c function.c
```

Another Example

- C code with an undefined function

```
int main(void) {
  int a = 2;
  int b = do_something(a);
  return b;
}
```

- This can be successfully compiled into an object file

```
linux> gcc -w -Og -c function.c
```

- Only linker will complain about the non-existing function

```
linux> gcc -Og function.o
function.o: In function ‘main’:
function.c:(.text+0xf): undefined reference to ‘do_something’
collect2: error: ld returned 1 exit status
```

Separate Function Definition

- Definition of function in separate file do-something.c

```
int do_something(int x) {
  return x*x;
}
```

Separate Function Definition

- Definition of function in separate file do-something.c

```
int do_something(int x) {
  return x*x;
}
```

- Compilation

```
linux> gcc -w -Og -c do-something.c
```

Separate Function Definition

- Definition of function in separate file do-something.c

```
int do_something(int x) {
  return x*x;
}
```

- Compilation

```
linux> gcc -w -Og -c do-something.c
```

- Linking

```
linux> gcc -Og function.o do-something.o
linux> ./a.out
linux> echo $?
4
```

Assembly Code

- function.s

```
        movl        $2, %edi
        call        do_something
```

- do-something.s

```
        movl        %edi, %eax
        imull       %edi, %eax
        ret
```

- Convention
  - integer argument is in register %edi
  - return value is in register %eax

No Type Checking

- Change of data types in do-something.c

```
float do_something(float x) {
  return x*x;
}
```

- Still links

```
linux> gcc -w -Og -c do-something.c
linux> gcc -Og function.o do-something.o
```

- But fails in execution

```
linux> ./a.out
linux> echo $?
0
```

(should return 4)

Solution: Header Files

- Header file do-something.h

```
int do_something(int);
```

- Include it in both do-something.c and function.c

```
#include "do-something.h"
```

- Compiler will now complain if there is a mismatch

function calls in x86

Example: plus.c


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

int main(void) {
  return plus(37,10);
}
```

x86 (32-bit)

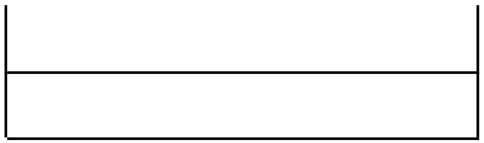
- Compile: `gcc -Og -S -m32 plus.c`

```
plus:
        movl    8(%esp), %eax
        addl    4(%esp), %eax
        ret
main:
        pushl   $10
        pushl   $37
        call    plus
        addl    $8, %esp
        ret
```

- Call values are pushed onto the stack: `pushl $10`
- Afterwards stack pointer is moved back up: `addl $8, %esp`
- Function reads directly from stack: `movl 8(%esp), %eax`
- Return value is in `%eax`

Stack Organization

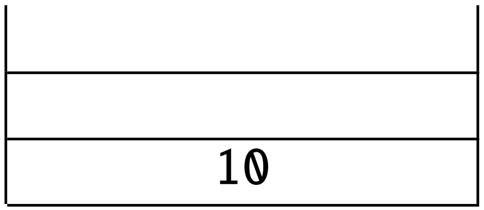
- Stack is filled downwards



%esp points here when main is called

Stack Organization

- Stack is filled downwards



%esp points here when main is called
second call value

Stack Organization

- Stack is filled downwards

	%esp points here when main is called
10	second call value
37	first call value

Stack Organization

- Stack is filled downwards

12		%esp points here when main is called
8	10	second call value
4	37	first call value
0	return address	%esp points here in plus

- Function has to read above the return address

function calls in x86-64

Register Conventions

- 32 bit x86 uses stack for call values (like 6502)
- Recall: MIPS had designated registers for call and return values
- 64 bit version of x86 also uses registers
- Note: all these are conventions, hardware always allows both options

Recall: Integer Registers

- 4 general purpose registers: %ax, %bx, %cx, %dx
- Stack pointer: %sp
- Base pointer: %bp
- Address registers: %si, %di

Recall: Integer Registers

- 4 general purpose registers: %ax, %bx, %cx, %dx
- Stack pointer: %sp
- Base pointer: %bp
- Address registers: %si, %di
- 32 bit registers: prefix with "e", e.g., %eax

Recall: Integer Registers

- 4 general purpose registers: %ax, %bx, %cx, %dx
- Stack pointer: %sp
- Base pointer: %bp
- Address registers: %si, %di
- 32 bit registers: prefix with "e", e.g., %eax
- 64 bit registers: prefix with "r", e.g., %rax

  8 additional registers added (%r8-%r15)

Recall: Integer Registers

- 4 general purpose registers: %ax, %bx, %cx, %dx
- Stack pointer: %sp
- Base pointer: %bp
- Address registers: %si, %di
- 32 bit registers: prefix with "e", e.g., %eax
- 64 bit registers: prefix with "r", e.g., %rax
  8 additional registers added (%r8-%r15)

x86-64

- Compile: `gcc -Og -S plus.c` (without `-m32`)

```
plus:
        leal        (%rdi,%rsi), %eax
        ret
main:
        movl        $10, %esi
        movl        $37, %edi
        call        plus
        ret
```

- Call values stored in registers (%esi,%edi)
- Function uses these directly
- Recall: %rdi is 64-bit view, %edi is 32-bit view of same register

lea

- lea: load effective address
- Carries out calculations typically done for memory lookup, e.g.,
  - leal (%rdi,%rsi), %eax
  - leal 4(%ebp), %eax
- But: stores result in register, makes no lookup
- Often abused to store result of math calc. in different register
- In example: addition of two register

Comparison

x86

```
plus:
  movl   8(%esp), %eax
  addl   4(%esp), %eax
  ret

main:
  pushl  $10
  pushl  $37
  call   plus
  addl   $8, %esp
  ret
```

x86-64

```
plus:
  leal  (%rdi,%rsi), %eax
  ret

main:
  movl  $10, %esi
  movl  $37, %edi
  call  plus
  ret
```

Use of registers more efficient

But: requires more attention to which registers may be overwritten

x86-64 Argument Conventions

- Arguments are stored in
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9
  - %xmm0-7
- Return value is in %rax
- These are Linux conventions, Windows conventions are different
- Caller has to preserve any register values that may be overwritten

Recursive Call

```
int main(void) {
  return fibonacci(10);
}

int fibonacci(int x) {
  if (x <= 1)
    return x;
  return fibonacci(x-2) + fibonacci(x-1);
}
```

x86-64 Assemby


```
fibonacci:
       cmpl     $1, %edi
       jle      .L3            ; special case <=1
       pushq    %rbp           ; function will preserve bp and bx
       pushq    %rbx
       subq     $8, %rsp       ; stack pointer must be multiple of 16
       movl     %edi, %ebx     ; save x from di in bx
       leal     -2(%rdi), %edi ; x-2
       call     fibonacci      ; f(x-2) -> eax
       movl     %eax, %ebp     ;        -> ebp
       leal     -1(%rbx), %edi ; x-1
       call     fibonacci      ; f(x-1) -> eax
       addl     %ebp, %eax     ; f(x-2) in ebp + f(x-1) in eax
       addq     $8, %rsp       ; restore sp
       popq     %rbx           ; restore bx and bp
       popq     %rbp
       ret
.L3:   movl     %edi, %eax     ; special case handling f(x) = x
       ret
```

Preserve Registers

- Function uses registers
  - bp to store result from first recursive call (f(x-2))
  - bx to store call value (x)

- These need to be stored on the stack

```
pushq       %rbp
pushq       %rbx
subq        $8, %rsp   ; stack pointer must be multiple of 16
```

- ... and retrieved

```
addq        $8, %rsp
popq        %rbx
popq        %rbp
```

Preserve Registers


```
fibonacci:
       cmpl     $1, %edi
       jle      .L3              ; special case <=1
       pushq    %rbp             ; function will preserve bp and bx
       pushq    %rbx
       subq     $8, %rsp         ; stack pointer must be multiple of 16
       movl     %edi, %ebx       ; save x from di in bx
       leal     -2(%rdi), %edi   ; x-2
       call     fibonacci        ; f(x-2) -> eax
       movl     %eax, %ebp       ;        -> ebp
       leal     -1(%rbx), %edi   ; x-1
       call     fibonacci        ; f(x-1) -> eax
       addl     %ebp, %eax       ; f(x-2) in ebp + f(x-1) in eax
       addq     $8, %rsp         ; restore sp
       popq     %rbx             ; restore bx and bp
       popq     %rbp
       ret
.L3:   movl     %edi, %eax       ; special case handling f(x) = x
       ret
```

Special Case


```
fibonacci:
      cmpl     $1, %edi
      jle      .L3              ; special case <=1
      pushq    %rbp             ; function will preserve bp and bx
      pushq    %rbx
      subq     $8, %rsp         ; stack pointer must be multiple of 16
      movl     %edi, %ebx       ; save x from di in bx
      leal     -2(%rdi), %edi   ; x-2
      call     fibonacci        ; f(x-2) -> eax
      movl     %eax, %ebp       ;        -> ebp
      leal     -1(%rbx), %edi   ; x-1
      call     fibonacci        ; f(x-1) -> eax
      addl     %ebp, %eax       ; f(x-2) in ebp + f(x-1) in eax
      addq     $8, %rsp         ; restore sp
      popq     %rbx             ; restore bx and bp
      popq     %rbp
      ret
.L3:  movl     %edi, %eax       ; special case handling f(x) = x
      ret
```

First Recursive Call

```
fibonacci:
      cmpl     $1, %edi
      jle      .L3            ; special case <=1
      pushq    %rbp           ; function will preserve bp and bx
      pushq    %rbx
      subq     $8, %rsp       ; stack pointer must be multiple of 16
      movl     %edi, %ebx     ; save x from di in bx
      leal     -2(%rdi), %edi ; x-2
      call     fibonacci      ; f(x-2) -> eax
      movl     %eax, %ebp     ;         -> ebp
      leal     -1(%rbx), %edi ; x-1
      call     fibonacci      ; f(x-1) -> eax
      addl     %ebp, %eax     ; f(x-2) in ebp + f(x-1) in eax
      addq     $8, %rsp       ; restore sp
      popq     %rbx           ; restore bx and bp
      popq     %rbp
      ret
.L3:  movl     %edi, %eax     ; special case handling f(x) = x
      ret
```

Second Recursive Call


```
fibonacci:
      cmpl     $1, %edi
      jle      .L3                ; special case <=1
      pushq    %rbp               ; function will preserve bp and bx
      pushq    %rbx
      subq     $8, %rsp           ; stack pointer must be multiple of 16
      movl     %edi, %ebx         ; save x from di in bx
      leal     -2(%rdi), %edi     ; x-2
      call     fibonacci          ; f(x-2) -> eax
      movl     %eax, %ebp         ;        -> ebp
      leal     -1(%rbx), %edi     ; x-1
      call     fibonacci          ; f(x-1) -> eax
      addl     %ebp, %eax         ; f(x-2) in ebp + f(x-1) in eax
      addq     $8, %rsp           ; restore sp
      popq     %rbx               ; restore bx and bp
      popq     %rbp
      ret
.L3:  movl     %edi, %eax         ; special case handling f(x) = x
      ret
```