Procedure Calls

Young W. Lim

2020-01-25 Mon

Outline

Based on

1. "Self-service Linux: Mastering the Art of Problem Determination",

Mark Wilding

1. "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

Stack frame

- procedure calls
  - passing procedure arguments
  - storing return informations
  - saving registers for later restoration
  - local storage
- stack frame:
  - the portion of the stack allocated for a single procedure call

Descending full stack

- Descending stack
  - stack grows toward lower addresses
  - `push` decreases `%esp` (growing stack)
  - `pop` increases `%esp` (shrinking stack)
- Full stack
  - contains a valid data at `%esp` address

Stack frame pointers

- Frame Pointer (%ebp)
  - the highest address of a stack frame
  - bottom of a stack frame
- Stack Pointer (%esp)
  - the lowest address of a stack frame
  - top of a stack frame
- read access via %ebp
  - the stack pointer can move while the procedure is executing
  - most information is accessed relative to the frame pointer

Stack frame structures (1)

- suppose procedure `P` (caller) calls procedure `Q` (callee)

the stack frame for P (caller)	- argument values to Q - return address to P
the stack frame for Q (callee)	- P's frame pointer (%ebp) - saved registers - local variables - temporaries - Q's arguments to other functions

Stack frame structures (2)

- the stack frame for P (caller)
  - the argument to Q are contained within the stack frame for P
  - the return address within P is pushed on the stack forming the end of P's stack frame
- the stack frame for Q (callee)
  - starts with the saved value of the frame pointer for P
  - followed by copies of any other saved values of registers (callee saved)
  - local variables

Local variables

- procedure `Q` also uses the stack for any local variables
  that cannot be stored in registers
  - when there are not enough registers
    to hold all of the local data
  - when the local variables are arrays or structures
    and hence must be accessed by array or structure references
  - the address operator & is applied to one of the local variables
    and hence we must be able to generate an address for it
- `Q` will use the stack frame for storing arguments
  to any procedure it calls

Caller's Viewpoint

——— H.I.G.H. A.D.D.R.E.S.S. ———

- frame pointer (%ebp)
- saved registers
- local variables
- temporaries

– –

- arguments for a funcion call to the callee
- return address
- stack pointer (%esp)

——— L.O.W. A.D.D.R.E.S.S. ———

local variables > function arguments > return address

Callee's Viewpoint

——— H.I.G.H. A.D.D.R.E.S.S. ———

- %ebp+c: argument 2 from the caller
- %ebp+8: argument 1 from the caller
- %ebp+4: return address of the caller

- -

- frame pointer (%ebp) : caller's %ebp stored
- saved registers of the callee
- local variables of the callee
- temporaries of the callee

——— L.O.W. A.D.D.R.E.S.S. ———

function arguments > return address > caller's %ebp > local variables

Stack Frames & Heap

```
------------------- H.I.G.H. A.D.D.R.E.S.S. -------------------
STACK (stack frame grows toward lower addresses)
    ............................................................
    stack Frame #1       v v v v
    ............................................................
    stack Frame #2       v v v v
    ............................................................
                         v v v v
    ............................................................
    stack Frame #n       v v v v
----------------------------------------------------------------



----------------------------------------------------------------
                         ^ ^ ^ ^
                         ^ ^ ^ ^
                         ^ ^ ^ ^
- HEAP (heap grows toward higher address)
----------------------------------------------------------------
```

Stack Frames & Memory Map

STACK (toward lower addresses)
HEAP (toward higher addresses)
Global Variables (BSS Segment)
Static Variables (Data Segment)
Machine Code (Text Segment)

Procedure Instructions

Procedure Call	`call` label	direct call
	`call` *operand	indirect call
Procedure Return	`leave`	stack preparation
	`ret`	return from call

Direct / indirect call / jump

- direct call / jump
  - `call label` or `jmp label`
- indirect call / jump
  - `call *%eax` or `jmp *%eax`
    uses the value in register %eax as the call/jump target
  - `call *(%eax)` or `jmp *%eax`
    reads the call/jump target from memory
    using the value in %eax as the read address

call label	direct call
call *operand	indirect call
jmp label	direct jump
jmp *operand	indirect jump

Operand Addressing Modes

Imm		M[Imm]	Absolute
Imm	(Eb)	M[Imm + R[Eb]]	Base + displace
Imm	(Eb, Ei)	M[Imm + R[Eb] + R[Ei]]	Indexed
Imm	(, Ei, s)	M[Imm + R[Ei]*s]	Scaled Indexed
Imm	(Eb, Ei, s)	M[Imm + R[Eb] + R[Ei]*s]	Scaled Indexed
	(Ea)	M[R[Ea]]	Indirect
	(Eb, Ei)	M[R[Eb] + R[Ei]]	Indexed
	(, Ei, s)	M[R[Ei]*s]	Scaled Indexed
	(Eb, Ei, s)	M[R[Eb] + R[Ei]*s]	Scaled Indexed

`call` Instruction

- `call` label : direct call (without memory reference)
- `call` *operand : indirect call (with memory reference)
  - operand address modes : `Imm (Eb, Ei, s)`
    offset Imm (base reg Eb, index reg Ei, scale factor s)

- *return address*: the address of the instruction immediately following the call instruction

`call` instruction

1. `pushl` *return addr* : push a return address
2. `jmp` *procedure* : jump to the start the called function

ret Instruction

- stack pointer must points to the return address

ret instruction

1. popl *return addr*
   pops the return address from the stack
2. jmp *return addr*
   jump to the return address location

leave Instruction

- prepare the stack for returning

leave instruction

- mov %ebp, %esp
  set stack pointer to the beginning of callee's stack
- pop %ebp
  restore saved %ebp
  set the stack pointer to the end of caller's stack

Return Value

- to return the value of any function that returns an integer or pointer register %eax is used

Procedure Instruction Summary

`call`	push a return address	`pushl` *return addr*
	jump to a procedure	`jmp` *procedure*
`ret`	pops a retrun address	`popl` *return addr*
	jump to this address	`jmp` *return addr*
`leave`	set SP to BP	`movl %ebp, %esp`
	restore BP	`popl %ebp`

Setup and finish code in a procedure

`call`	push a return address	`pushl` *return addr*
	jump to a procedure	`jmp` *procedure*
setup	save old %ebp	`pushl %ebp`
	set %esp to %ebp	`movl %esp, %ebp`
	...	...
	function body	function body
	...	...
finish	restore %esp	`movl %ebp, %esp`
(`leave`)	restore %ebp	`popl %ebp`
`ret`	pops a retrun address	`popl` *return addr*
	jump to this address	`jmp` *return addr*

IA32 conventions for register usage

- the callee should not overwrite some registers that the caller is going to use later

%eax	Caller save register
%ebx	Callee save register
%ecx	Caller save register
%edx	Caller save register
%esi	Callee save register
%edi	Callee save register
%ebp	Frame Pointer
%esp	Stack Pointer

Caller save registers	Callee save registers
%eax	%ebx
%ecx	%esi
%edx	%edi

IA32 conventions for register usage

Caller Save Registers	%eax %ecx %edx	the callee can overwrite these registers
Callee Save Registers	%ebx %esi %edi	the callee must save these registers before using and restore them before returning

Example 1 (1)

- example code 1

```
int P() {
  int x = f();

  Q(x);
  return x;
}
```

- procedure P wants the value
  it has computed for `x = f()`
  to remain valid across the call to `Q(x)`
  then to return `x`

Example 1 (2)

- if `x` is in a caller save register,
  then `P` (the caller) must save the value `x`
  before calling `Q(x)`
  and restore `x` *after* `Q` *returns*

- if `x` is in a callee save register,
  and `Q` must save the value `x`
  before using the register
  and restore `x` *before returning*

- in either case,
  - saving : pushing the register value onto the stack
  - restoring : popping from the stack back to the register

Example 2 (1)

- example code 2

```
int P (int x)
{
  int y = x*x;          // y is computed here
  int z = Q(y);         // y is passed as an argument

  return y + z;         // y is accessed here also
}
```

- P compute `y=x*x` before calling `Q(y)`,
  but it must also ensure that
  the value of `y` is availiable
  in `return y+z` after `Q` returns

Example 2 (2)

- two ways to ensure that the value of `y` is availiable in `return y+z` after `Q` returns
  1. Caller `P` saves `y` in its own stack frame
  2. Callee `Q` saves `y` in a callee save register
- most commonly, gcc uses the latter conventions, since it tends to reduce the toal number of stack accesses

Example 2 (3)

1. Caller P saves y in its own stack frame
   - before calling Q(y),
     P can store the value of y=x*x in its own stack frame
   - when Q returns, in z=Q(y)
     P can then retrieve the value of y from the stack

Example 2 (4)

2. Callee `Q` saves `y` in a callee save register
   - `P` can store the value of `y=x*x` in a callee save register
   - if `Q` or any procdures called by `Q` wants to use this register, it must save the register value in its stack frame and restore the value before it returns.
   - thus, when `Q(y)` returns to `P`, the value of `z=Q(y)` will be in the callee save register,
   - either because the register was never altered or because it was saved and restored

GCC Example for a procedure call

- the beginning part of an assembly code

```
pushl %edi                  ; callee save %edi
pushl %esi                  ; callee save %esi
pushl %ebx                  ; callee save %ebx
movl  24(%ebp), %eax        ; caller save %eax
imull 15(%ebp), %eax
leal  0(,%eax,4), %ecx      ; caller save %ecx
addl  8(%ebp), %ecx
movl  %ebx, %edx            ; caller save %edx
```

- the callee save register (%edi, %esi, %ebx)
  - to use the callee save registers in the procedure, they should be save on its stack frame and be restored before returning to the caller
- the caller save register (%eax, %ecx, %edx)
  - these can be modified without saving nor restoring

Procedure definition example code

caller P source code

```
int P() {
  int a1 = 55;
  int a2 = 77;
  int sum = Q( &a1, &a2 );
  int diff = a1 - a2;

  return sum * diff;
}
```

callee Q source code

```
int Q(int *xp, int *yp) {
  int x = *xp;
  int y = *yp;

  *xp = y;
  *yp = x;
  return x+y;
}
```

Stack Frames contents for P & Q

before calling Q

```
         +---------+-----------|
%ebp ->  | %ebp+0  | saved %ebp |
         +---------+------------+
         | %ebp-4  | a2         |
         +---------+------------+
         | %ebp-8  | a1         |
         +---------+------------+
         | %ebp-12 | &a2        |
         +---------+------------+
%esp ->  | %ebp-16 | &a1        |
         +---------+-----------|
         |         |            |
         +---------+-----------|
         |         |            |
         +---------+-----------|
         |         |            |
         +---------+-----------|
```

in the body of Q

```
         +---------+-----------|
         | %ebp+24 | saved %ebp |
         +---------+------------+
         | %ebp+20 | a2         |
         +---------+------------+
         | %ebp+16 | a1         |
         +---------+------------+
         | %ebp+12 | &a2        |
         +---------+------------+
         | %ebp+ 8 | &a1        |
         +---------+-----------|
         | %ebp+ 4 | return adr |
         +---------+-----------|
%ebp ->  | %ebp+ 0 | saved %ebp |
         +---------+-----------|
%esp ->  | %ebp- 4 | saved %ebx |
         +---------+-----------|
```

Calling code of the caller P (1)

- the stack frame for P includes storage for local variables a1 and a2, at position %ebp-8 and %ebp-4
- Q retrieves its arguments &a1 and &a2 from the stack frrame for P

caller P code

```
int P() {
  int a1 = 55;
  int a2 = 77;
  int sum = Q( &a1, &a2 );
  int diff = a1 - a2;

  return sum * diff;
}
```

before calling Q

```
        +---------+-----------|
%ebp -> | %ebp+0  | saved %ebp |
        +---------+-----------+
        | %ebp-4  | a2         |
        +---------+-----------+
        | %ebp-8  | a1         |
        +---------+-----------+
        | %ebp-12 | &a2        |
        +---------+-----------+
%esp -> | %ebp-16 | &a1        |
        +---------+-----------|
```

Calling code of the caller P (2)

calling Q

```
; compute &a2 (addr of %ebp-4)
leal  -4(%ebp), %eax

; push &a2
pushl %eax

; compute &a1 (addr of %ebp-8)
leal  -8(%ebp), %eax

; push &a1
pushl %eax

; call Q() function
call  Q
```

before calling Q

```
        +---------+-----------|
%ebp -> | %ebp+0  | saved %ebp |
        +---------+-----------+
        | %ebp-4  | a2         |
        +---------+-----------+
        | %ebp-8  | a1         |
        +---------+-----------+
        | %ebp-12 | &a2        |
        +---------+-----------+
%esp -> | %ebp-16 | &a1        |
        +---------+-----------|
```

Calling code of the caller P (3)

- the local variable a1 and a2 must be stored on the stack since the addresses &a1 and &a2 need to be computed using leal instruction
- local variables (a2, a1) and arguments (&a2, &a1) are pushed on the stack in the order

calling Q

```
leal  -4(%ebp), %eax      ; compute &a2 (the address value of %ebp-4)
pushl %eax                ; push &a2
leal  -8(%ebp), %eax      ; compute &a1 (the address value of %ebp-8)
pushl %eax                ; push &a1
call  Q                   ; call Q() function
```

Function code of the callee Q

the compiled code for a function has 3 parts

1. the setup part
   the stack frame is initialized
2. the body part
   the actual computation of the procedure is performed
3. the finish part
   the stack state is restored and the procedure returns

Setup code for the callee Q

Setup code for the callee Q

```
Q:
  ; %ebp : frame pointer of P

  ; save this old %ebp
  pushl %ebp

  ; set %ebp as a new frame pointer
  movl  %esp, %ebp

  ; save %ebx
  pushl %ebx
```

- %ebx is used in the callee Q
- %ebx is a callee save register
- %ebx is pushed on the stack

Stack frame of the callee Q

```
             +---------+-----------|
             | %ebp+24 | saved %ebp |
             +---------+-----------+
             | %ebp+20 | a2         |
             +---------+-----------+
             | %ebp+16 | a1         |
             +---------+-----------+
             | %ebp+12 | &a2        |
             +---------+-----------+
             | %ebp+ 8 | &a1        |
             +---------+-----------|
             | %ebp+ 4 | return adr |
             +---------+-----------|
%ebp ->      | %ebp+ 0 | saved %ebp |
             +---------+-----------|
%esp ->      | %ebp- 4 | saved %ebx |
             +---------+-----------|
```

Body code for the callee Q (1)

Body Code for Q

```
;        %edx holds xp
movl  8(%ebp), %edx
;        %ecx holds yp
movl 12(%ebp), %ecx
;        %ebx holds x
movl   (%edx), %ebx
;        %eax holds y
movl   (%ecx), %eax

;        assign y to *xp
movl     %ecx, (%edx)
;        assign x to *yp
movl     %ebx, (%ecx)
;        %eax holds x+y
addl     %ebx,  %eax
```

- return value is at %eax

Stack frame of the callee Q

```
         +---------+-----------|
         | %ebp+24 | saved %ebp |
         +---------+-----------+
         | %ebp+20 | a2         |
         +---------+-----------+
         | %ebp+16 | a1         |
         +---------+-----------+
         | %ebp+12 | &a2        |
         +---------+-----------+
         | %ebp+ 8 | &a1        |
         +---------+-----------|
         | %ebp+ 4 | return adr |
         +---------+-----------|
%ebp ->  | %ebp+ 0 | saved %ebp |
         +---------+-----------|
%esp ->  | %ebp- 4 | saved %ebx |
         +---------+-----------|
```

Body code for the callee Q (2)

Body Code for Q

```
;        %edx holds xp
movl  8(%ebp), %edx
;        %ecx holds yp
movl 12(%ebp), %ecx
;        %ebx holds x
movl   (%edx), %ebx
;        %eax holds y
movl   (%ecx), %eax

;        assign y to *xp
movl     %ecx, (%edx)
;        assign x to *yp
movl     %ebx, (%ecx)
;        %eax holds x+y
addl     %ebx,  %eax
```

- return value is at %eax

callee Q source code

```
int Q(int *xp, int *yp) {
  int x = *xp;
  int y = *yp;

  *xp = y;
  *yp = x;
  return x+y;
}
```

Finish code for the callee Q

Finish code for Q

```
;    restore %ebx
popl %ebx

;    restore %esp
movl %ebp, %esp

;    restore %ebp
popl %ebp

;    return to the caller
ret
```

Stack frame of the callee Q

```
               +---------+-----------|
               | %ebp+24 | saved %ebp |
               +---------+-----------+
               | %ebp+20 | a2         |
               +---------+-----------+
               | %ebp+16 | a1         |
               +---------+-----------+
               | %ebp+12 | &a2        |
               +---------+-----------+
               | %ebp+ 8 | &a1        |
               +---------+-----------|
               | %ebp+ 4 | return adr |
               +---------+-----------|
%ebp ->        | %ebp+ 0 | saved %ebp |
               +---------+-----------|
%esp ->        | %ebp- 4 | saved %ebx |
               +---------+-----------|
```

direct / indirect procedure code

direct procedure

```
int foo(int a) {
  return a;
}

int direct() {
  int i, b = 0;

  for (i = 0; i < INT_MAX; ++i) {
      b = foo(b);
  }

  return b;
}
```

indirect procedure

```
int indirect(int (*fn)(int)) {
  int i, b = 0;

  for (i = 0; i < INT_MAX; ++i) {
    b = fn(b);
  }

  return b;
}
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

main procedure code

main procedure

```
int foo(int a) {
  return a;
}


int main(int argc, char *argv[]) {
  if (argc == 2 && argv[1][0] == 'd') {
    return direct();
  }
  else {
    return indirect(&foo);
  }
}
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Assembly code

direct version

```
_foo:
    movl    4(%esp), %eax
    ret

_direct_version:
    subl    $4, %esp
    movl    $2147483647, %edx
    xorl    %eax, %eax
L3:
    movl    %eax, (%esp)
    call    _foo
    subl    $1, %edx
    jne L3
    addl    $4, %esp
    ret
```

indirect version

```
_indirect_version:
    pushl   %esi
    pushl   %ebx
    xorl    %eax, %eax
    movl    $2147483647, %ebx
    subl    $20, %esp
    movl    32(%esp), %esi
L8:
    movl    %eax, (%esp)
    call    *%esi
    subl    $1, %ebx
    jne L8
    addl    $20, %esp
    popl    %ebx
    popl    %esi
    ret
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct / indirect call

direct / indirect call

- direct call
```
call      _foo
```
- indirect call
```
call      *%esi
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (3)

```
_foo:
    movl    4(%esp), %eax          ; Copy argument from stack into eax,
                                   ; which is normally used to store
                                   ; the return value from a function
                                   ; in x86.
    ret
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (4)

```
_direct_version:
    subl    $4, %esp                ; Allocate 4 bytes of stack space.
                                    ; This space will be used to hold
                                    ; the argument when we call foo().

    movl    $2147483647, %edx       ; edx is the 'i' variable of the
                                    ; for loop. Initialized to MAX_INT

    xorl    %eax, %eax              ; eax is the 'b' variable. That xor
                                    ; will set eax to 0.
L3:
    movl    %eax, (%esp)            ; Copy 'b' onto the stack space
                                    ; reserved to hold the argument
                                    ; for foo().
    call    _foo                    ; Call the function
    subl    $1, %edx                ; i--
    jne L3                          ; if (result of subtract above != 0) goto L3;
    addl    $4, %esp
    ret
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (5)

```
_indirect_version:
    pushl   %esi
    pushl   %ebx
    xorl    %eax, %eax
    movl    $2147483647, %ebx
    subl    $20, %esp
    movl    32(%esp), %esi
L8:
    movl    %eax, (%esp)
    call    *%esi
    subl    $1, %ebx
    jne L8
    addl    $20, %esp
    popl    %ebx
    popl    %esi
    ret
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (6)

- differences between the direct and indirect versions
  - the direct version uses 3 instructions to setup
    before it gets to the for-loop.
    the indirect version uses 6.
  - the loop itself is 4 instructions in both cases,
    but the direct version uses 3 registers (eax, esp and edx)
    while the indirect version uses 4 (eax, esp, esi, and ebx).
    If there were no more registers free,
    the indirect version would have to add extra code
    to move variables on and off the stack.

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (7)

- The extra setup overhead doesn't matter much, unless the loop count is tiny.
- But the extra register use does matter.
- In real code, register contention is often a problem - it is more of a problem on x86 than instruction sets with more registers, but I don't think we should ignore this cost in any case.

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (8)

- To investigate the cost, the code is changed to use additional copies of foo().
- timing the resulting executable, the indirect version is 3.4x slower.

direct procedure ver 2

```
int foo(int a) {  return a; }

int bar(int a) {  return a; }

int baz(int a) {  return a; }

int direct_version() {
  int i, b = 0;
  for (i = 0; i < INT_MAX; ++i) {
      b = foo(b) + bar(b) + baz(b);
  }
  return b;
}
```

indirect procedure ver 2

```
int indirect_version
    (int (*fn)(int),
     int (*fn2)(int),
     int (*fn3)(int)) {
  int i, b = 0;

  for (i = 0; i < INT_MAX; ++i) {
    b = fn(b) + fn2(b) + fn3(b);
  }

  return b;
}
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Direct and indirect call examples (9)

main procedure ver 2

```
int main(int argc, char *argv[]) {
  if (argc == 2 && argv[1][0] == 'd') {
    return direct_version();
  }
  else {
    return indirect_version(&foo, &bar, &baz);
  }
}
```

https://gist.github.com/rianhunter/0be8dc116b120ad5fdd4#file-call_overhead-c-L17

Fibonacci Sequence

```
int fibo(int n) {
  int prev, val;

  if (n <= 2) return 1;
  prev = fibo(n-2);
  val  = fibo(n-1);
  return prev + val;
}
```

- multiple outstanding calls
- each call has its own local variables
- allocated only when the procedure is called
- deallocated when it returns

Stack Frames for the caller and the callee

```
%ebp+8 : n
%ebp+4 : return address
%ebp+0 : saved %ebp
...
...
%ebp-20: saved %esi
%ebp-24: saved %ebp
```

after initial setup

```
%ebp+8 : n
%ebp+4 : return address
%ebp+0 : saved %ebp
...
...
%ebp-20: saved %esi
%ebp-24: saved %ebp
...
...
%ebp-40: n-2
```

just before the 1st recusive call

Setup Code for fibo()

```
fibo:
  pushl %ebp
  movl  %esp, %ebp
  subl  $16, %esp
  pushl %esi
  pushl %ebx
```

```
Set up code
%ebp: frame pointer
alloc 16 bytes on stack
save %esi (-20)
save %ebx (-24)
```

Body Code for fibo()

```
movl   8(%ebp), %ebx
cmpl   $2,      %ebx
jle    .L24
addl   $-12,    %esp
leal  -2(%ebx), %eax
pushl  %eax
call   fibo
movl   %eax,    %esi
addl   $-12,    %esp
leal  -1(%ebx), %eax
pushl  %eax
call   fibo
addl   %esi,    %eax
jmp    .L25
```

Finish Code for Q()

```
popl %ebx               restore %ebx
movl %ebp, %esp         restore %esp
popl %ebp               restore %ebp
ret                     return to the caller
```