Procedure Calls

Young W. Lim

2022-04-12 Tue

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

TOC: Stack frame background

- Stack operation
  - Stack frame
  - Descending full stack
  - Stack frame pointers
- Stack frame structures
  - Stack frame structures
  - Local variables
  - `P` Caller's viewpoint
  - `Q` Callee's viewpoint
  - Stack Frames and heap
  - Stack frames and memory map

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 changer 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

P 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

L.O.W. A.D.D.R.E.S.S.

local variables > function arguments > return address

Q 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

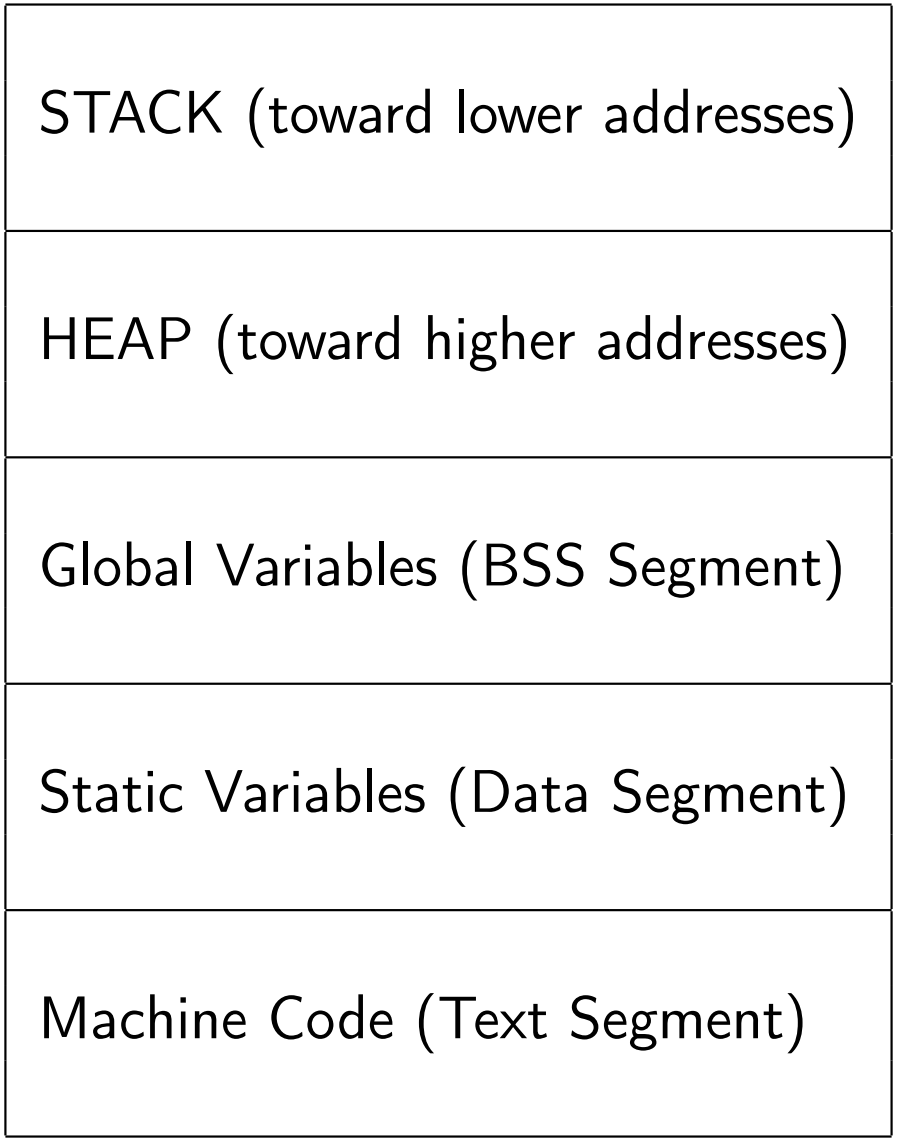
– – – – – – L.O.W A.D.D.R.E.S.S. – – – – – – – – – –

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

Stack frames & heap

STACK (stack frame grows toward _lower_ addresses)	
	stack Frame #1 ↓ ↓ ↓ ↓
	stack Frame #2 ↓ ↓ ↓ ↓
	....
	stack Frame #n ↓ ↓ ↓ ↓
HEAP (heap grows toward _higher_ address) ↑ ↑ ↑ ↑	

Stack frames & memory map



TOC: Transferring Control

- Procedure instructions
- Direct / indirect call / jump
- Operand addressing modes
- `call` instruction
- `ret` instruction
- `leave` instruction
- Return value
- Procedure instruction summary
- Setup and finish code in a procedure
- Register usage conventions for IA32

Procedure Instructions

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

Direct and indirect call / jump

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

- 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

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`

Call, setup, body, and finish code

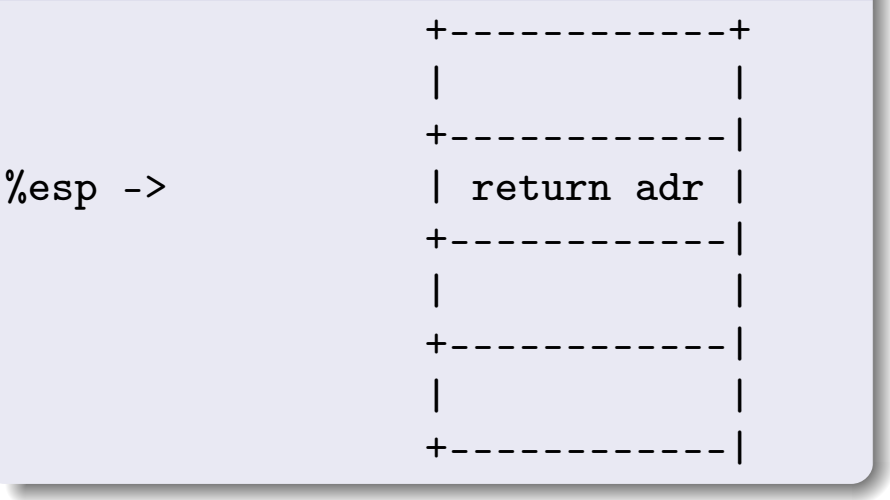
`call`	push a return address	`pushl` *return addr*
	jump to a procedure	`jmp` *procedure*

setup	save old %ebp	`pushl %ebp`
	set %esp to %ebp	`movl %esp, %ebp`
	...	...
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*

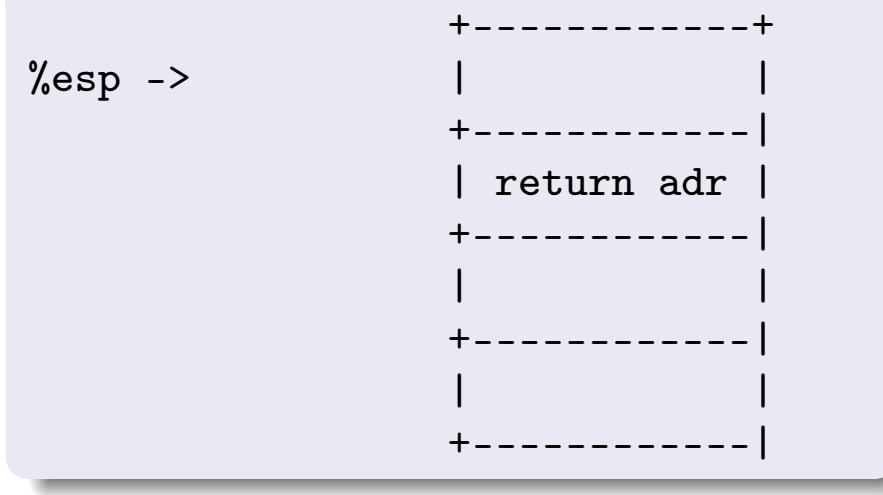
call and ret (the 2nd part of finish code)

call	push a return address	pushl *return addr*
	jump to a procedure	jmp *procedure*
finish	pops a retrun address	popl *return addr*
(ret)	jump to this address	jmp *return addr*

call



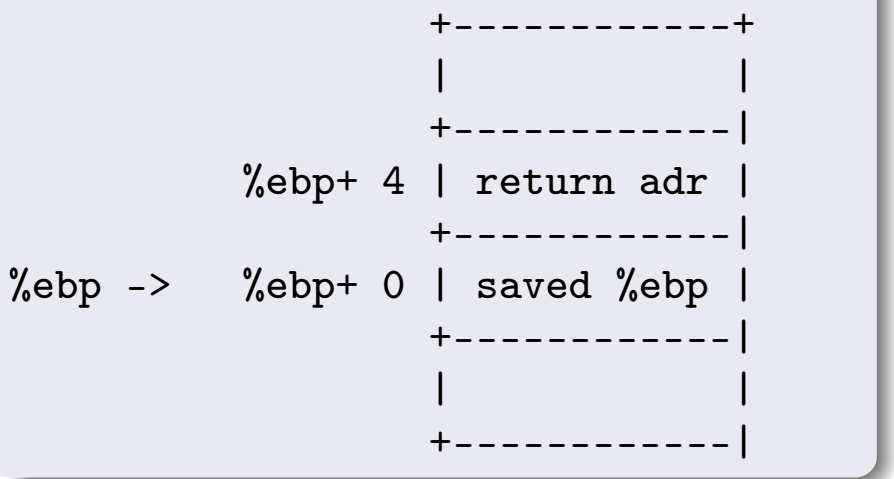
ret



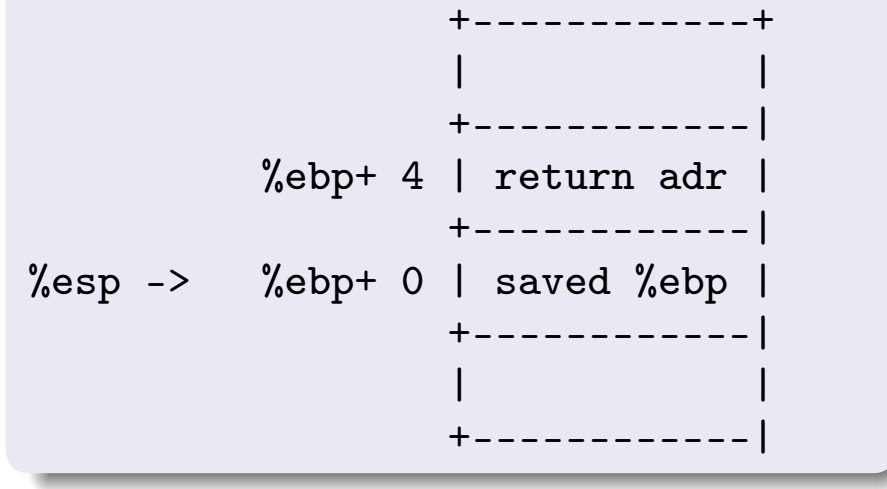
setup and leave (the 1st part of finish code)

setup	save old %ebp	`pushl %ebp`
	set %esp to %ebp	`movl %esp, %ebp`
finish	restore %esp	`movl %ebp, %esp`
(`leave`)	restore %ebp	`popl %ebp`

setup



leave



TOC: Register usage

- Register usage conventions
- Call Example 1
- Call Example 2
- Call Example 3

Register usage conventions for IA32 (1)

- 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

Register usage conventions for IA32 (2)

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

Register usage examples 1 & 2

example code 1

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

  Q();

  return x;
}
```

- x is not used as an argument
  1. caller save register
  2. callee save register

example code 2

```
int P (int x)
{
  int y = x*x;
  int z = Q(y);

  return y + z;
}
```

- y is used as an argument
  1. a stack frame
  2. a callee save register

Register usage example 1 - (1)

example code 1

```
int P() {
  int x = f();          // x is computed here

  Q();                  // x is *not* passed as an argument
  return x;             // x is accessed here also
}
```

- procedure P wants the value
  it has computed for x = f()
  to remain valid across the call to Q()
  then to return x
- but x is not used as an argument to Q()

Register usage example 1 - (2)

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

- if `x` is in a callee save register (`%ebx, %esi, %edi`),
  and `Q()` (the callee) 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

Register usage 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
- y is used as an argument : Q(y)

Register usage 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 (`%ebx, %esi, %edi`)

- most commonly, gcc uses the latter conventions, since it tends to reduce the toal number of stack accesses

Register usage example 2 - (3)

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

%ebp + 4(n+1)	argument n
%ebp + 8	argument 1
%ebp + 4	return address

Register usage 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 (`%eax, %ecx, %edx`)
   - 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 (`%eax, %ecx, %edx`)
   - either because the register was never altered
     or because it was saved and restored

GCC Example for a procedure call (1)

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 16(%ebp), %eax        ; caller save %eax
movl  24(%ebp), %ebx        ; callee save %ebx
leal  0(,%eax,4), %ecx      ; caller save %ecx
addl  8(%ebp), %ecx         ; caller save %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*

GCC Example for a procedure call (2)

- just three callee save registers
  (%edi, %esi, and %ebx)
  are *saved* on the stack
- the program modifies these
  and three other caller save registers
  (%eax, %ecx, and %edx)
- at the end of the procedure,
  the values of callee save registers
  (%edi, %esi, and %ebx)
  are *restored* using `popl` instruction,
- while the other three caller save registers
  (%eax, %ecx, and %edx)
  are *left* in their *modified* states

GCC Example for a procedure call (3)

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 16(%ebp), %eax        ; caller save %eax
movl  24(%ebp), %ebx        ; callee save %ebx
leal  0(,%eax,4), %ecx      ; caller save %ecx
addl  8(%ebp), %ecx         ; caller save %ecx
movl  %ebx, %edx            ; caller save %edx
```

%ebp + 4*6	24(%ebp)	A
%ebp + 4*5		
%ebp + 4*4	16(%ebp)	B
%ebp + 4*3		
%ebp + 4*2	8(%ebp)	C
%ebp + 4*1	ret addr	

GCC Example for a procedure call (4)

the beginning part of an assembly code

```
pushl %edi                  ;
pushl %esi                  ;
pushl %ebx                  ;
movl  24(%ebp), %eax        ; A -> %eax
imull 16(%ebp), %eax        ; A * B -> %eax
movl  24(%ebp), %ebx        ; A -> %ebx
leal  0(,%eax,4), %ecx      ;
addl  8(%ebp), %ecx         ;
movl  %ebx, %edx            ;
```

- `imull S, D` : D * S $\rightarrow$ D
- `leal  S, D` : &S $\rightarrow$ D

TOC: Procedure definition example

- Summary
  - Procedure control summary
  - Procedure definition example code
  - Stack frames contents for `P` & `Q`
- Calling code
  - Calling code of the caller `P`
- Callee code
  - Function code of the callee `Q`
  - Setup code for the callee `Q`
  - Body code for the callee `Q`
  - Finish code for the callee `Q`

Procedure Control Summary (1)

caller `call`

- `pushl` *return addr*
- `jmp` *procedure*

callee *setup*

- `pushl %ebp`
- `movl %esp, %ebp`

callee `leave`

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

callee `ret`

- `popl` *return addr*
- `jmp` *return addr*

Procedure Control Summary (2)

caller	call	- `pushl` *return addr*
		- `jmp` *procedure*
	setup	- `pushl %ebp`
		- `movl %esp, %ebp`
callee	leave	- `movl %ebp, %esp`
		- `popl %ebp`
	ret	- `popl` *return addr*
		- `jmp` *return addr*

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;
}
```

- local variables : a1, a2
- arguments : &a1, &a2

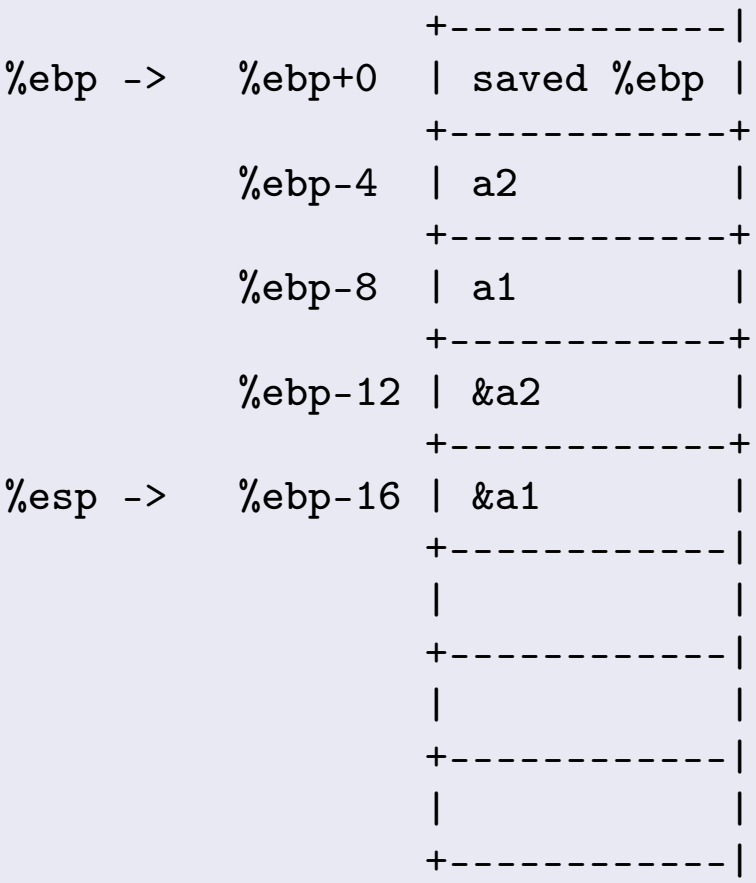
callee Q source code

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

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

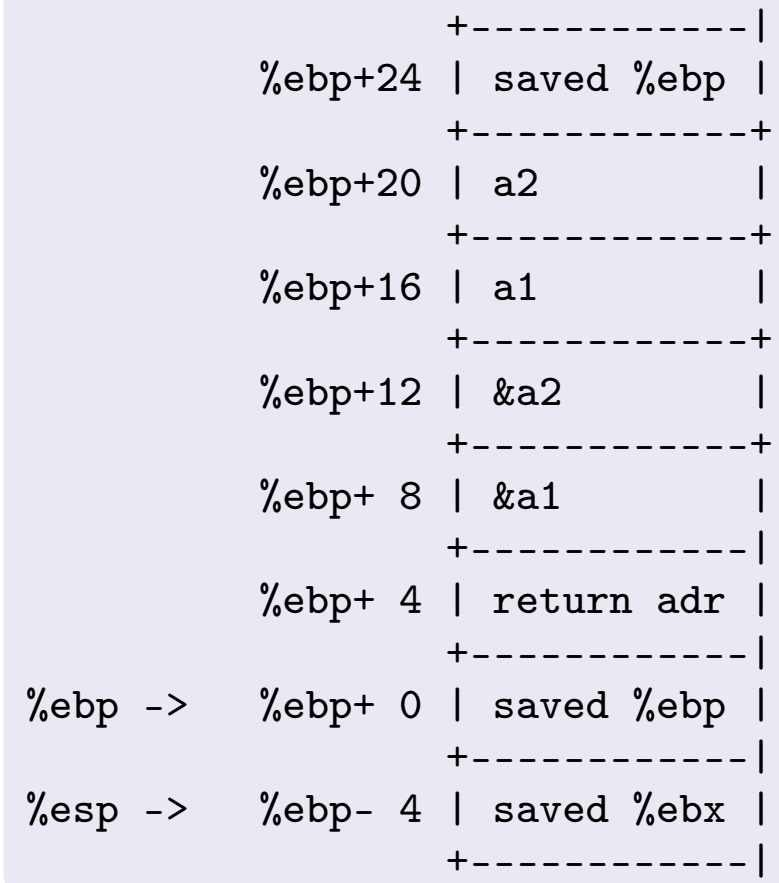
- parameters : xp, yp
- local variables : x, y

Stack Frames contents for P & Q

before calling to Q



in the body of Q



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;
}
```

- local variables : a1, a2
- arguments : &a1, &a2

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)
; push &a2 - 2nd argument

leal  -4(%ebp), %eax
pushl %eax

; compute &a1 (addr of %ebp-8)
; push &a1 - 1st argument

leal  -8(%ebp), %eax
pushl %eax

; call Q(&a1, &a2) function
call  Q
```

before calling Q

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

- &ea2 = %ebp-4
- &ea1 = %ebp-8

Calling code of the caller P (3)

- the local variable a1 and a2 must be stored on the stack since their 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 (= %ebp-4)
pushl %eax               ; push &a2
leal  -8(%ebp), %eax     ; compute &a1 (= %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    %eax, (%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;
}
```

- parameters
  - xp in %edx
  - yp in %ecx
- local variables
  - x in %ebx
  - y in %eax

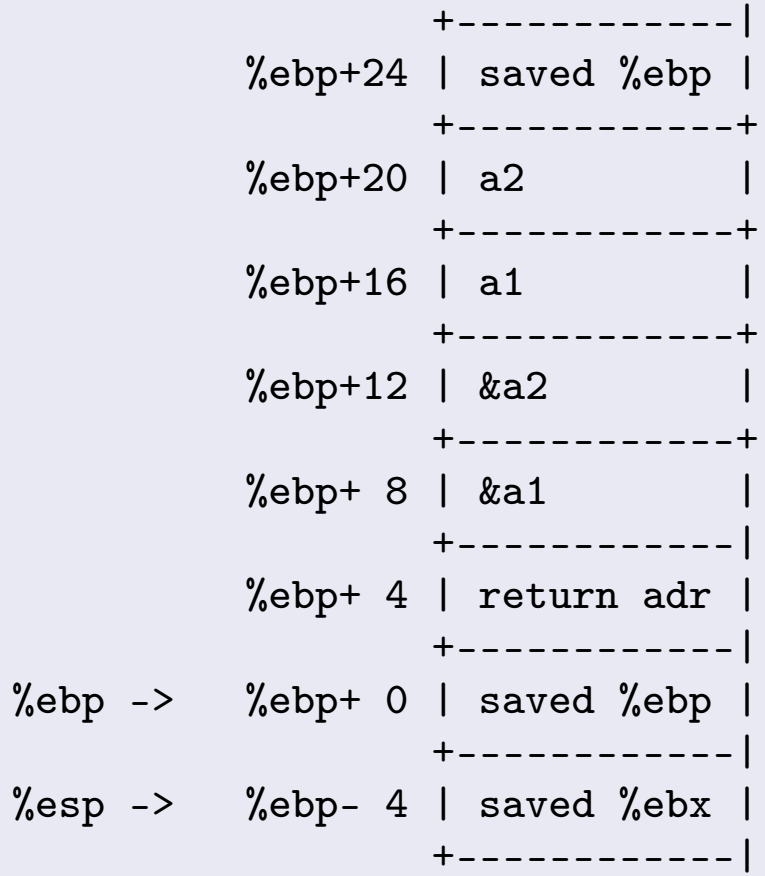
Finish code for the callee Q

Finish code for Q

```
;    restore %ebx
popl %ebx

;    restore %esp
movl %ebp, %esp

;    restore %ebp
popl %ebp

;    return to the caller
ret
```

- at the %ebp location
  the old %ebp was stored

Stack frame of the callee Q



TOC: Direct / indirect call examples

- Source and assembly code
- Stack operations
- Assembly analyses

TOC: Source and assembly codes

- Source codes (1) `foo` and `main`
- Source codes (2) `direct` and `indirect`
- Direct / indirect function call comparison
- (1) `direct` and `indirect` assemblies
- (2) `direct` source and assembly
- (3) `indirect` source and assembly

Source codes (1) foo and main

foo source code

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

- foo is called directly in direct function
- &foo is passed as an argument then foo is called in indirect function

main procedure

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

- direct has no argument
- indirect has one argument

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

Source codes (2) direct and indirect

direct source code

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

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

  return b;
}
```

- no parameter
- function foo

indirect source code

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

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

  return b;
}
```

- one parameter :
  function pointer fn

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

Direct / indirect function call comparison

direct function call

c

- `foo(int a)`
- `b = foo(b);`

assembly

- `call _foo`
- call label

indirect function call

c

- `int (*fn)(int)`
- `b = fn(b);`

assembly

- `call *%esi`
- call *operand

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

(1) direct and indirect assemblies

direct assembly

```
_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
```

- no parameter
- call _foo

indirect assembly

```
_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
```

- one parameter
- call *%esi

(2) direct source and assembly

direct source

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

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

  return b;
}
```

- direct : no parameter
- foo(b)

direct assembly

```
_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
```

- call _foo

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

(3) indirect source and assembly

indirect source

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

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

  return b;
}
```

- indirect : one argument function pointer fn
- fn(b)

indirect assembly

```
_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
```

- call *%esi

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

TOC: stack operations

- `foo` stack operation
- `direct` stack operation
- `indirect` stack operation

foo stack operation

foo stack code

- source

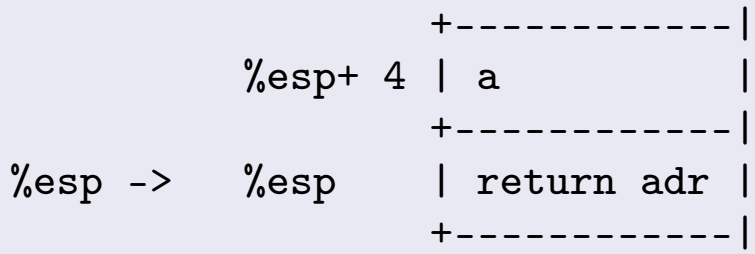
```
int foo(int a) {
  return a;
}
```

- stack related assembly

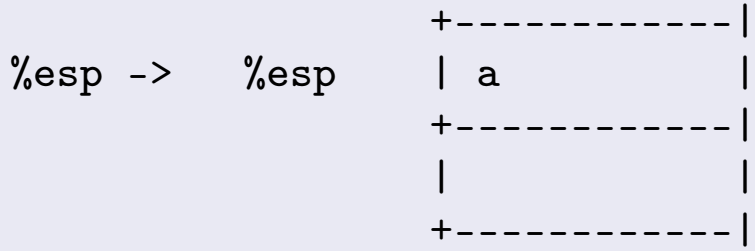
```
_foo:
    movl    4(%esp), %eax
    ret
```

foo stack operation

- just after call _foo



- just after ret



direct stack operation (1)

direct stack code

- source

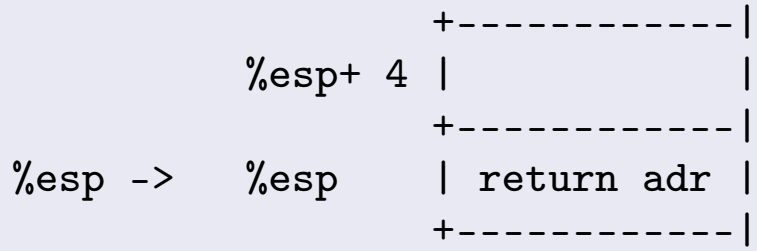
```
int direct() {
  int i, b = 0;
  for (i = 0; i < INT_MAX; ++i) {
      b = foo(b);
  }
  return b;
}
```

- stack related assembly

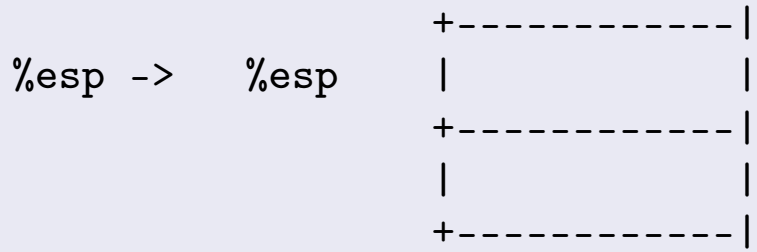
```
_direct:
    subl    $4, %esp
    ...     ...
L3: movl    %eax, (%esp)
    call    _foo
    addl    $4, %esp
    ...     ...
    ret
```

direct stack operation

- just after call _direct



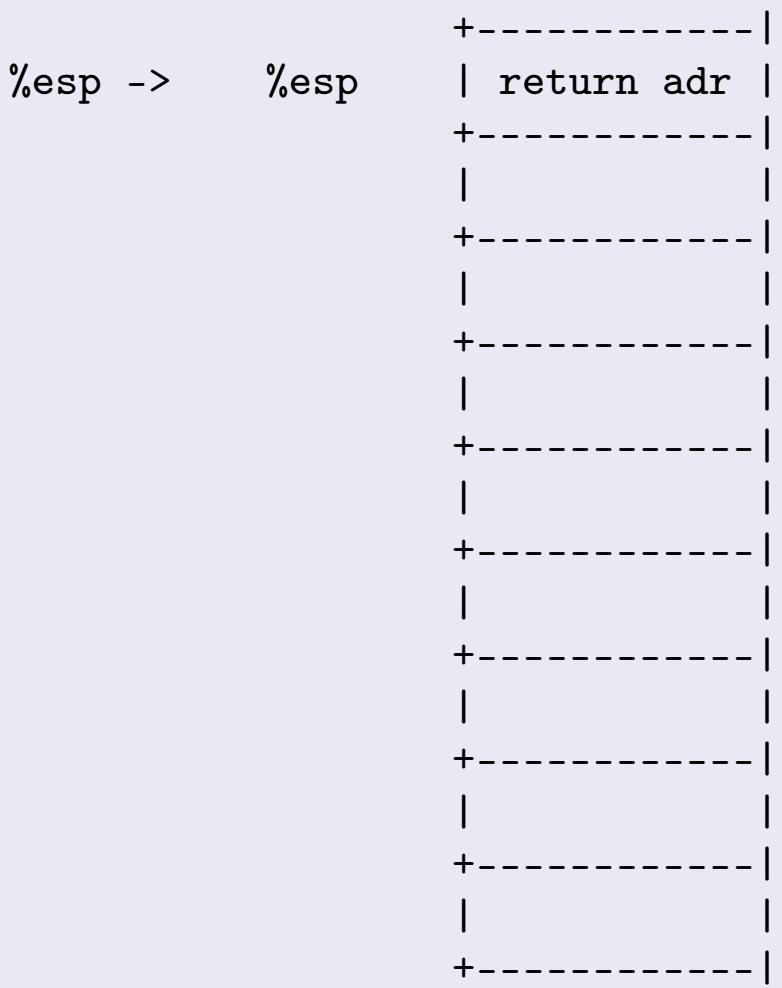
- just after ret



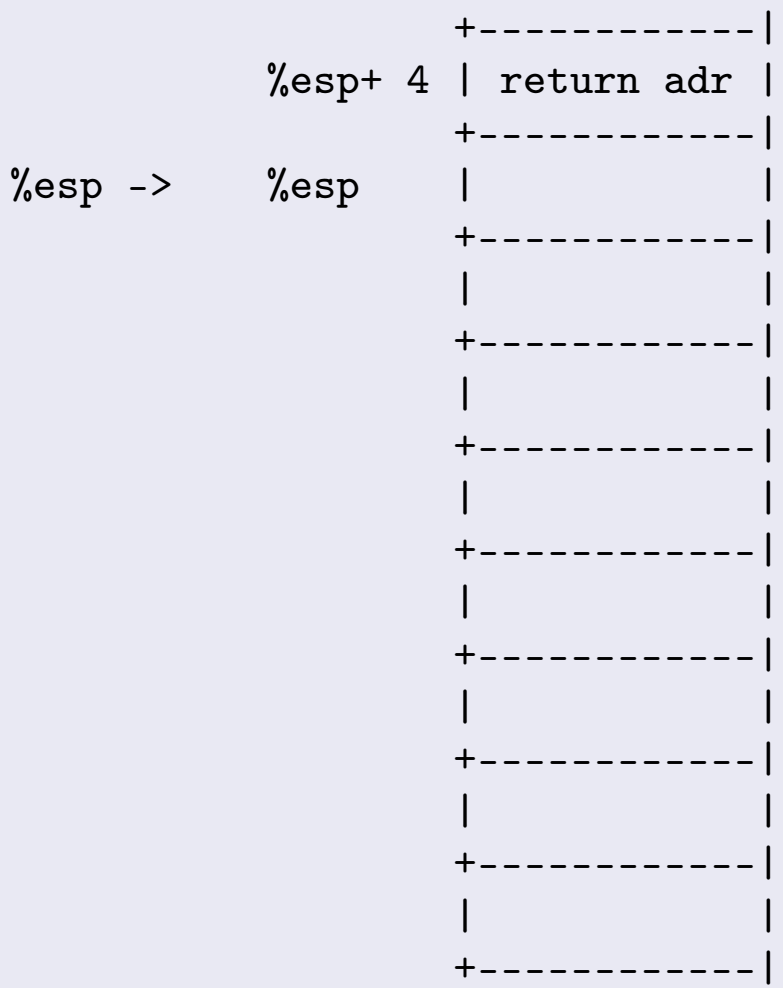
- direct() has no argument

direct stack operation (2)

just after call _direct

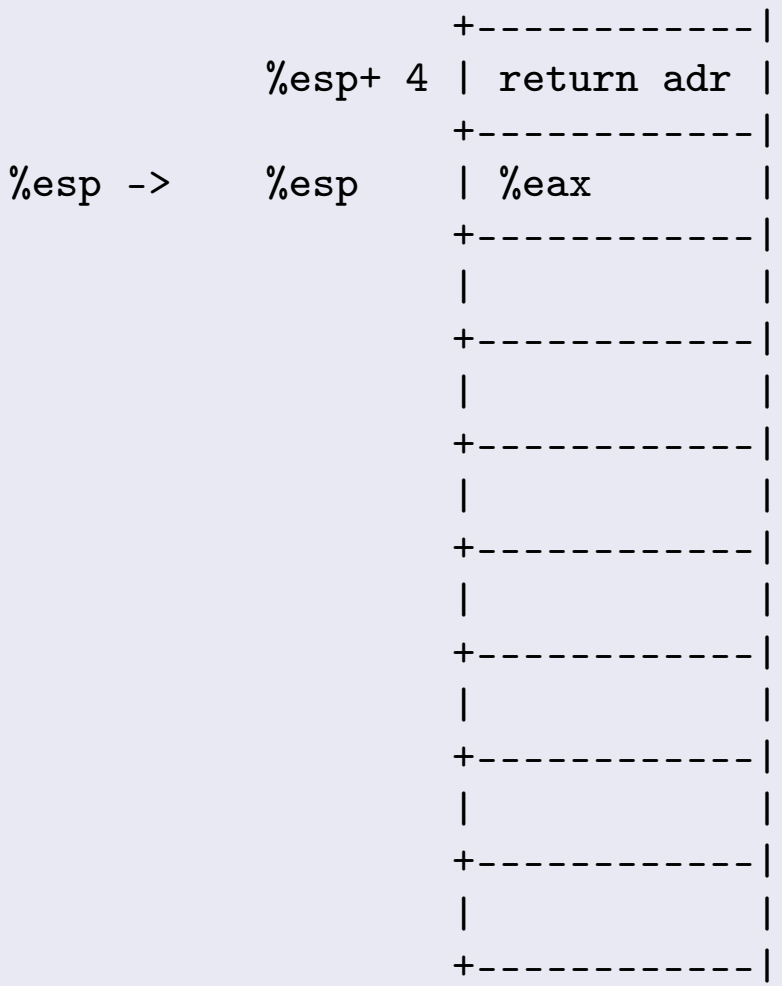


subl $4, %esp

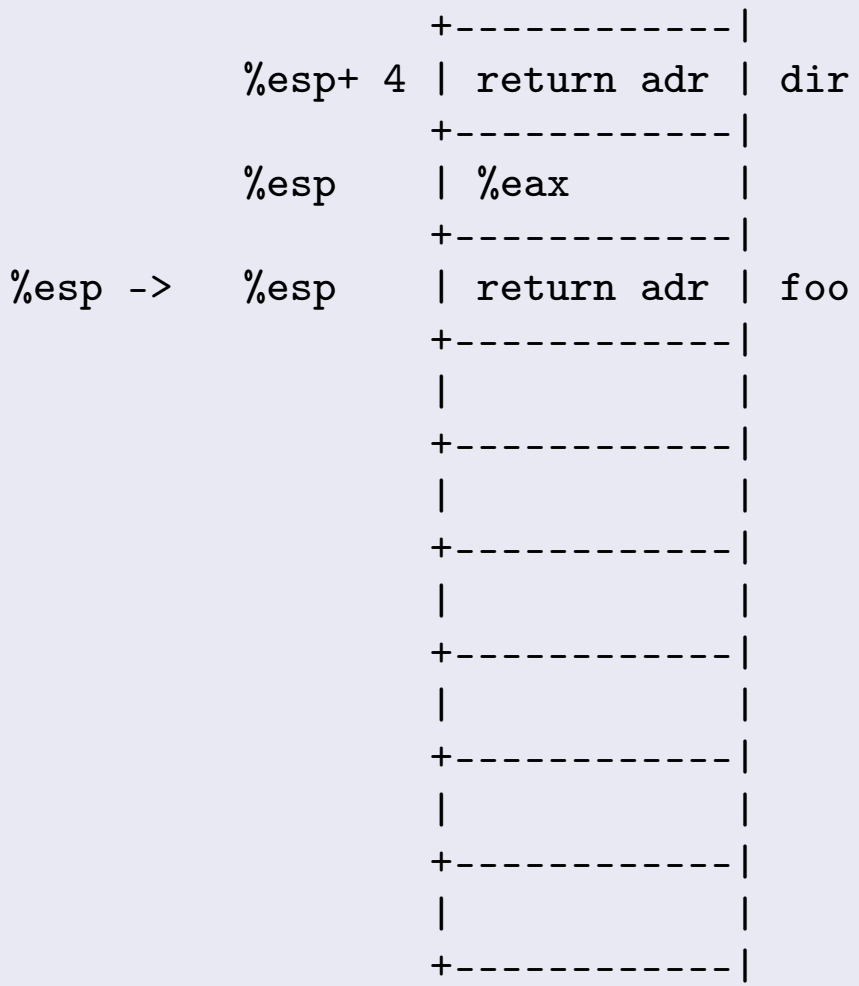


direct stack operation (3)

movl %eax, (%esp)

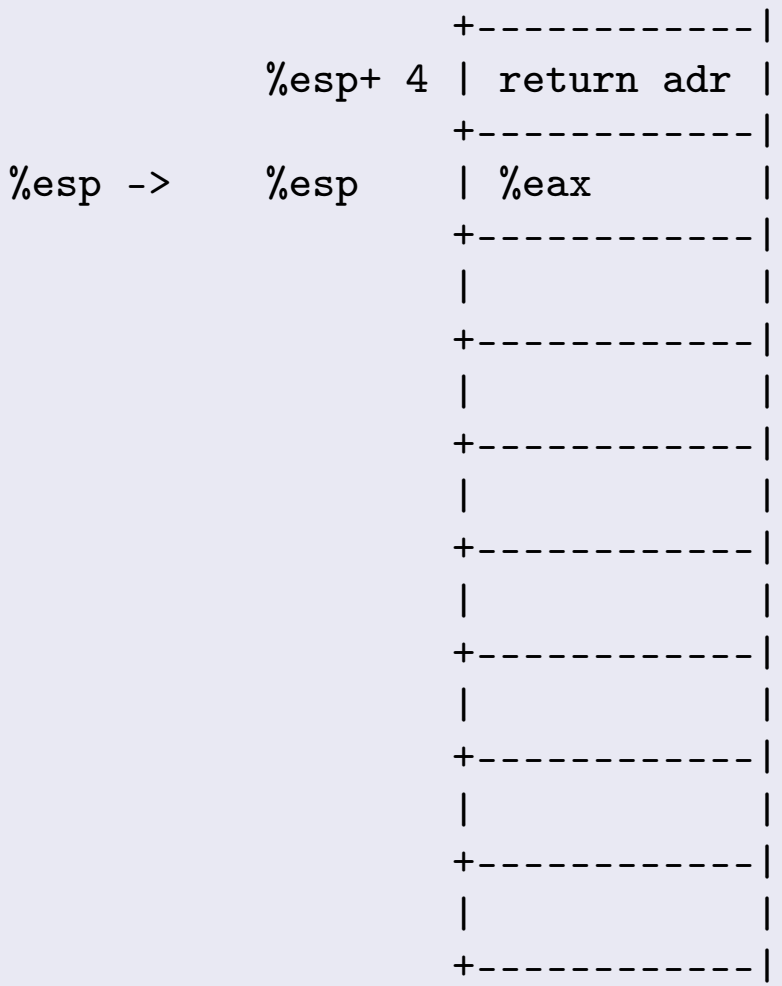


just after call _foo

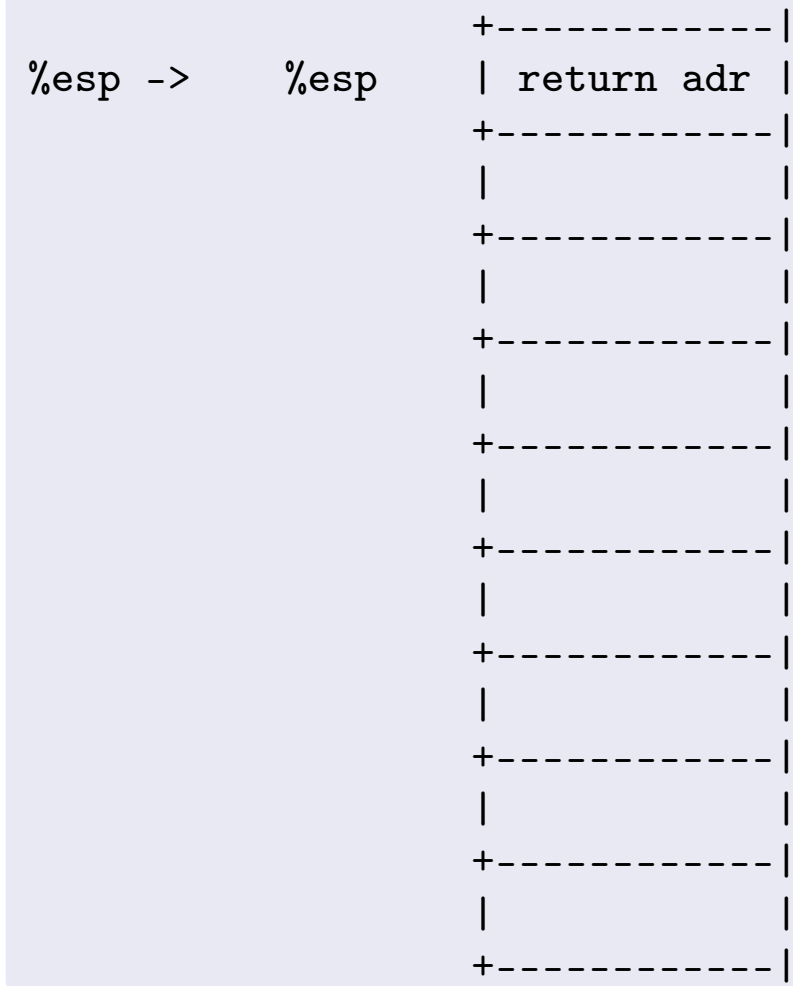


direct stack operation (4)

just after ret



addl $4, %esp



indirect stack operation (1)

indirect stack operation

- indirect source

```
int indirect(int (*fn)(int)) {
  int i, b = 0;
  for (i = 0; i < INT_MAX; ++i)
    b = fn(b);
  return b;
}
```

- indirect stack operation

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

indirect stack operation

- just after call _indirect

```
                  +------------|
          %esp+ 4 | fn         |
                  +------------|
%esp ->   %esp    | return adr |
                  +------------|
```

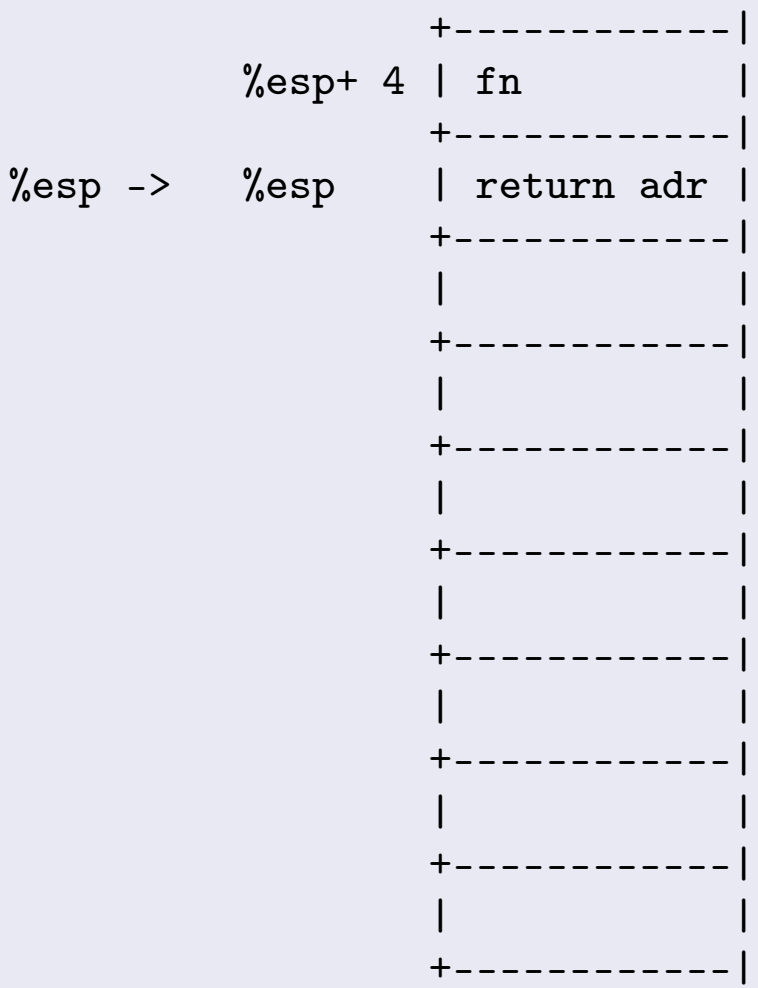
- just after ret

```
                  +------------|
%esp ->   %esp    | fn         |
                  +------------|
                  |            |
                  +------------|
```

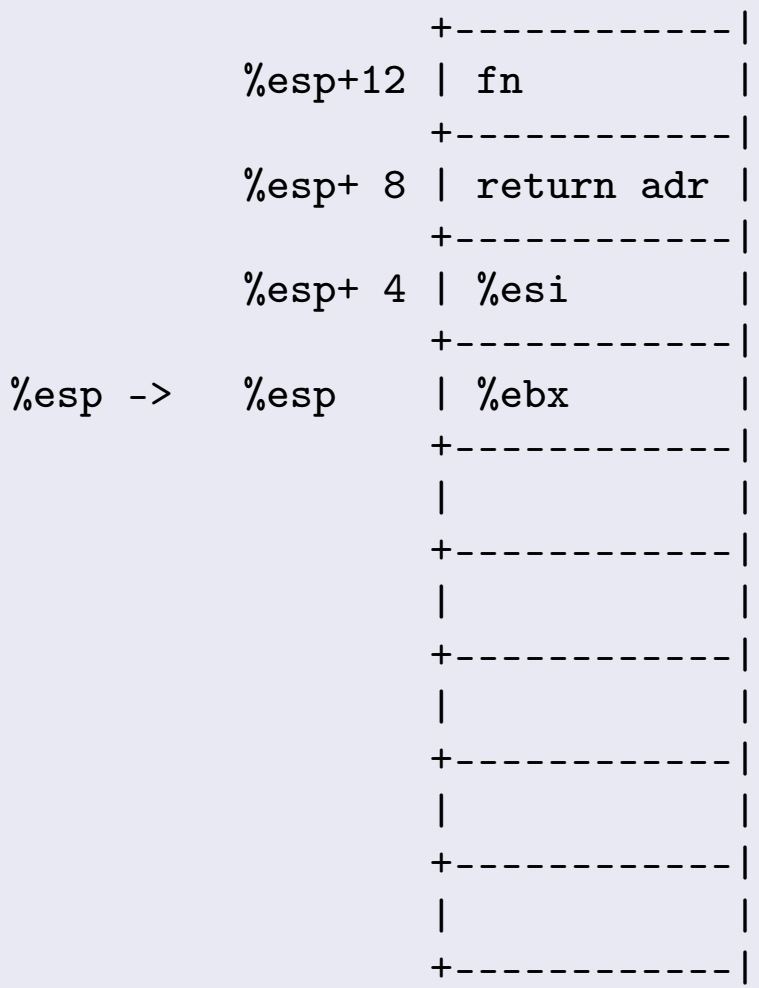
- indirect(foo) has one argument : fn
  int (*fn) (int)

indirect stack operation (2)

just after the call to indirect

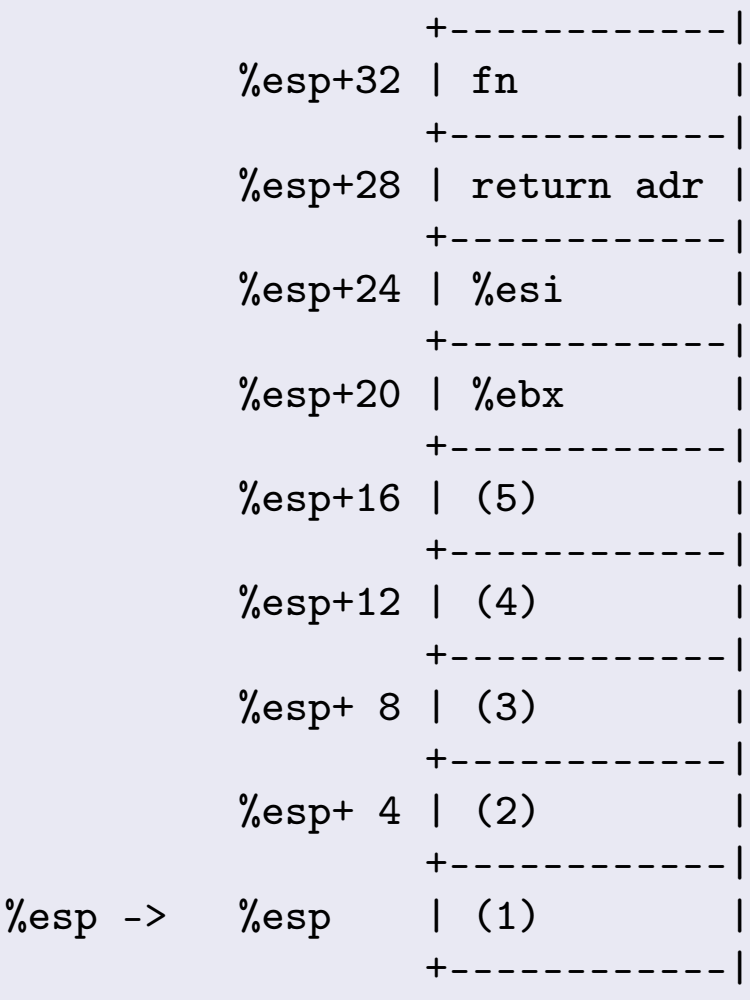


pushl %esi, pushl %ebx

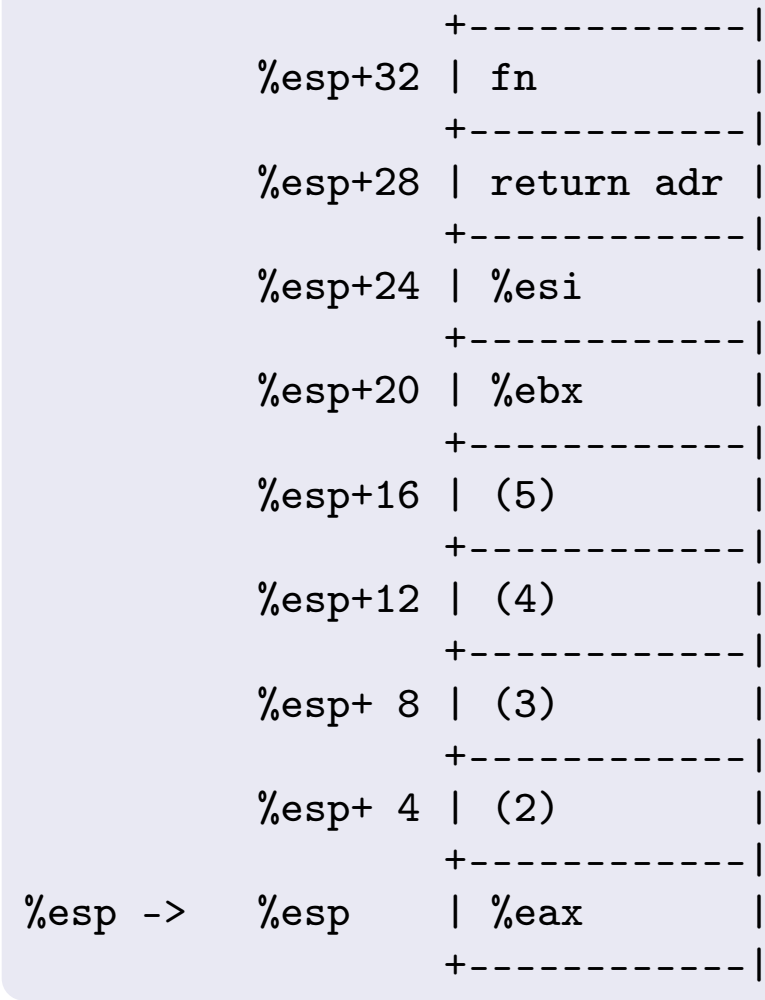


indirect stack operation (3)

subl $20, %esp

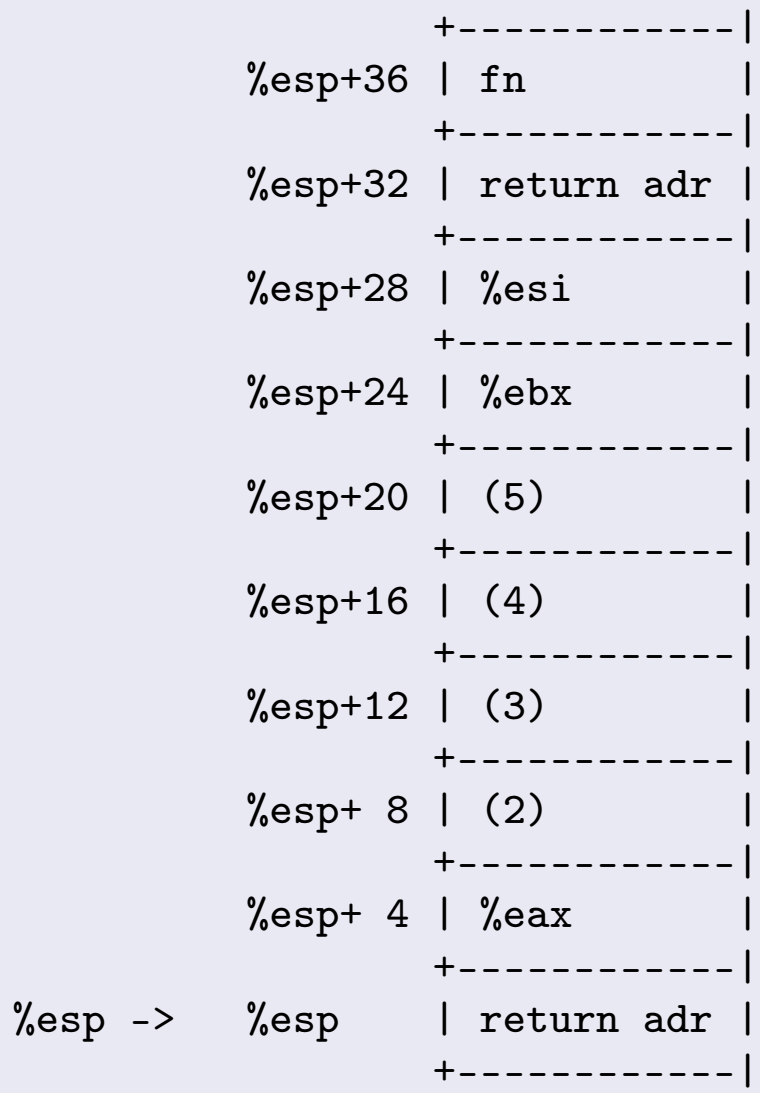


movl %eax (%esi)

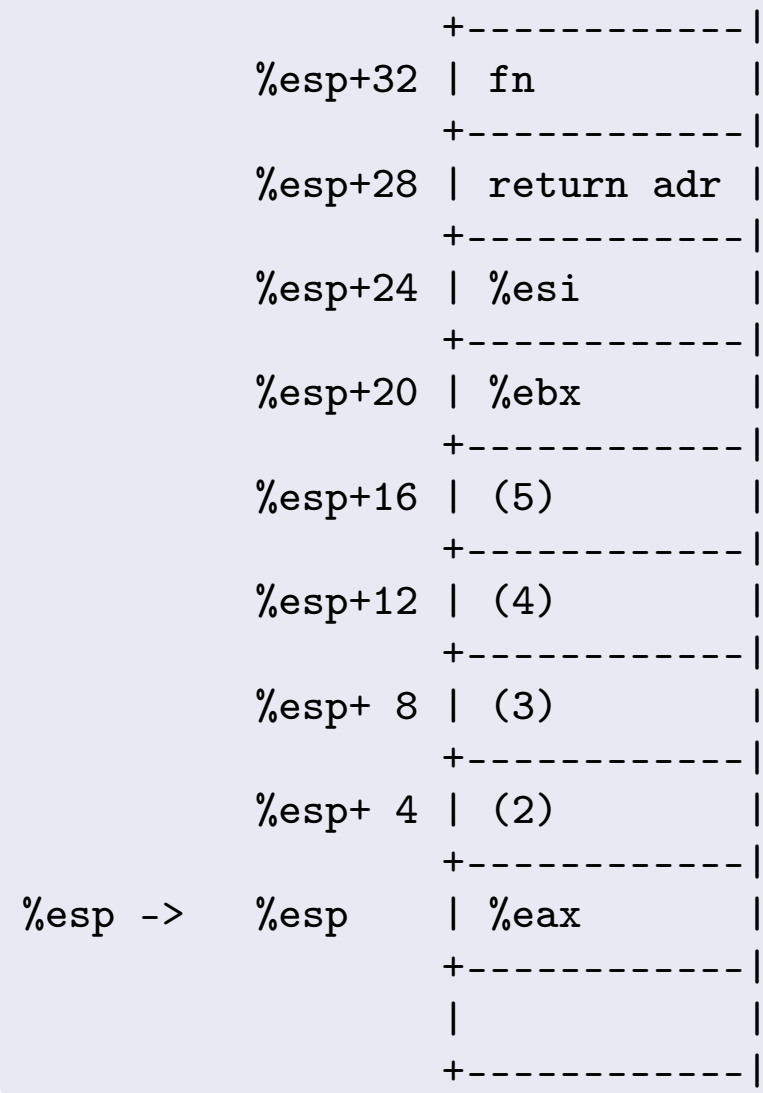


indirect stack operation (4)

just after call *%esi

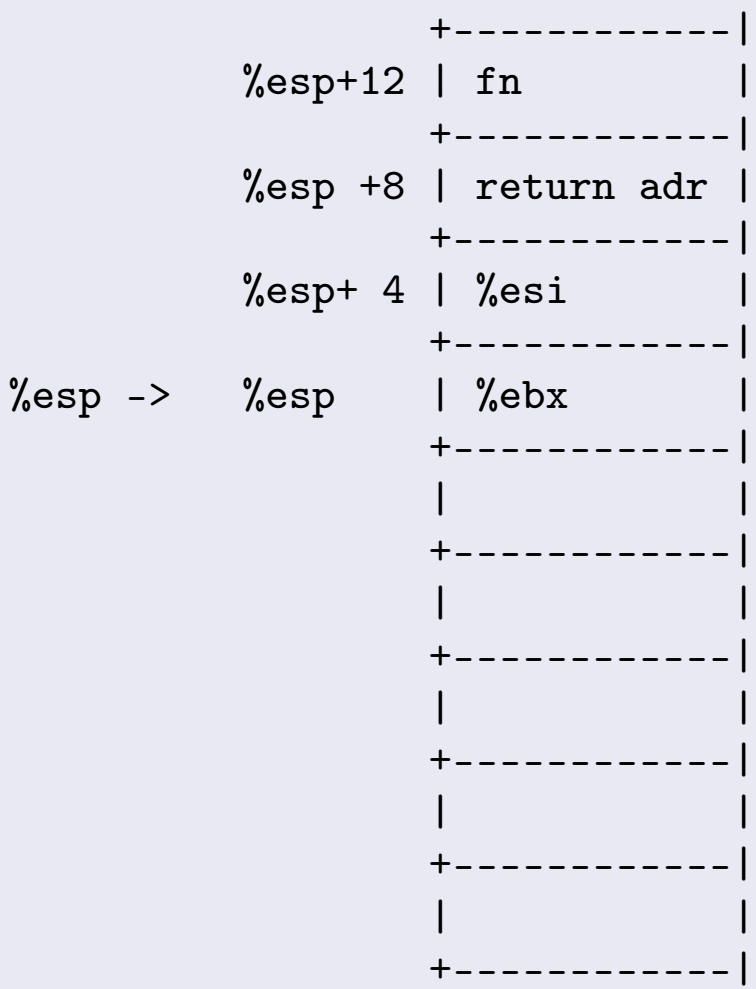


just after ret

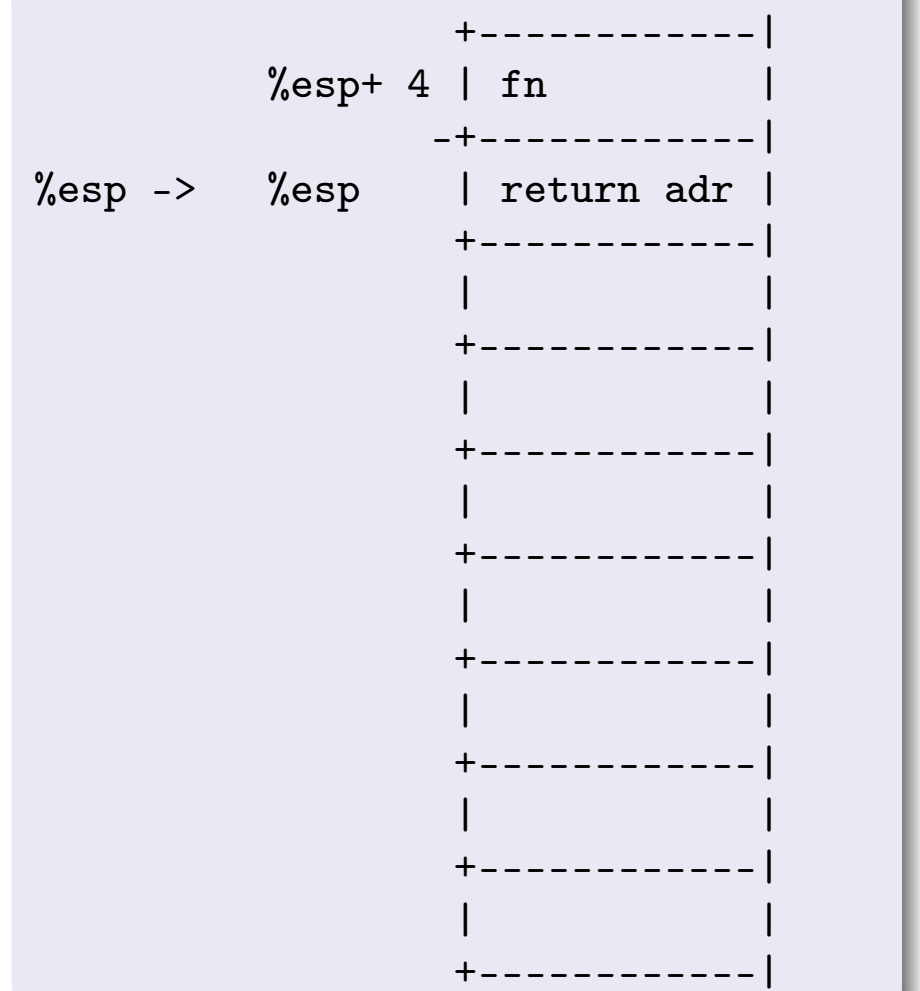


indirect stack operation (5)

addl $20, %esp



popl %ebx, popl %esi



TOC: assembly analyses

- `foo` assembly analysis
- `direct` assembly analysis
- `indirect` assembly analysis
- Differences (1) setup code
- Differences (2) loop codes
- Differences (3) costs

foo assembly analysis

foo assembly code

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

- argument 4(%esp)
- return value %eax

foo source code

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

- argument a becomes return value a

```
                +------------|
        %esp+ 4 | a          |   argument a becomes return value a
                +------------|
%esp ->   %esp  | return adr |
                +------------|
```

`movl 4(%esp), %eax`	copy argument from stack at %esp+4 into %eax to store the return value from a function

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

direct assembly analysis (1)

direct assembly code (1)

```
_direct:
    subl    $4, %esp
    movl    $2147483647, %edx
    xorl    %eax, %eax
```

direct source code

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

subl	$4, %esp	allocate 4 bytes of stack space 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 xor will set %eax to 0.

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

direct assembly analysis (2)

direct assembly code (2)

```
L3: movl    %eax, (%esp)  <---
    call    _foo              :
```

direct source code

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

return b;
```

```
                +------------|
        %esp+ 4 | %eax       |   argument %eax becomes return value %eax
                +------------|
%esp ->   %esp  | return adr |
                +------------|
```

movl	%eax, (%esp)	copy 'b' onto the stack space reserved to hold the argument for foo().
call	_foo	

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

direct assembly analysis (3)

direct assembly code (3)

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

direct source code

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

return b;
```

subl	$1, %edx	%edx is the i variable decrement %edx by one
addl	$4, %esp	dellocate 4 bytes of stack space that held the argument when we call foo()

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

indirect assembly analysis (1)

indirect assembly code (1)

```
_indirect:
    pushl   %esi
    pushl   %ebx
    xorl    %eax, %eax
    movl    $2147483647, %ebx
```

indirect source code

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

return b;
```

pushl	%esi	%esi is used for the argument fn
pushl	%ebx	%ebx is used for the loop variable i
xorl	%eax, %eax	%eax is the b variable xor will set %eax to 0.
movl	$2147483647, %ebx	%ebx is the i variable of the for loop Initialized to MAX_INT

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

indirect assembly analysis (2)

indirect assembly code (2)

```
_indirect:
    ...
    subl    $20, %esp
    movl    32(%esp), %esi
```

indirect source code

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

return b;
```

subl	$20, %esp	allocate 5*4 bytes of stack space to hold the argument when we call foo()
movl	32(%esp), %esi	copy the word at %esp+32 to %esi, where the function pointer fn is stored, which the argument of indirect

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

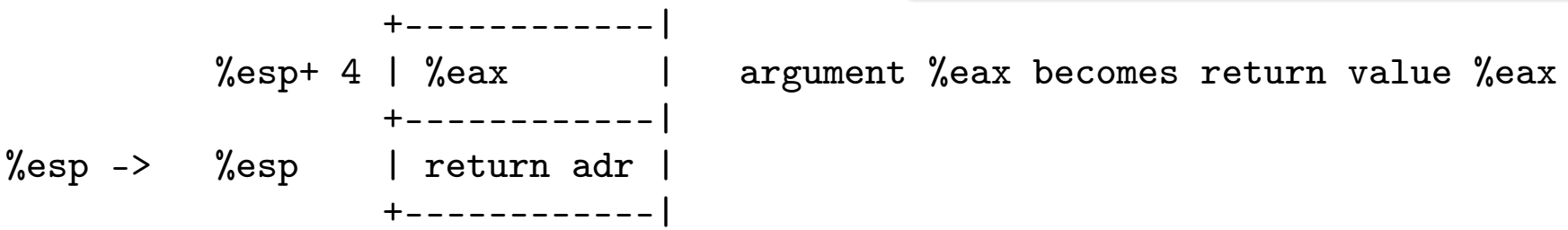
indirect assembly analysis (3)

indirect assembly code (3)

```
L8: movl    %eax, (%esp)  <---
    call    *%esi             :
```

indirect source code

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

return b;
```



movl	%eax, (%esp)	copy 'b' onto the stack space reserved to hold the argument for foo().
call	*%esi	%esi hold fn, the function pointer the argument of indirect

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

indirect assembly analysis (4)

indirect assembly code (4)

```
L8: movl    %eax, (%esp)  <---
    call    *%esi           :
    subl    $1, %ebx        :
    jne L8                  ----
    addl    $20, %esp
    popl    %ebx
    popl    %esi
    ret
```

indirect source code

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

return b;
```

subl	$1, %ebx	%ebx is the i variable decrement %ebx by one
addl	$20, %esp	deallocate 20 bytes of stack space to hold the argument when we call foo()
popl	%ebx	pop %ebx from the stack
popl	%esi	pop %esi from the stack

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

Differences (1) setup codes

direct assembly code

```
_direct:
    subl    $4, %esp
    movl    $2147483647, %edx
    xorl    %eax, %eax
```

indirect assembly code

```
_indirect:
    pushl   %esi
    pushl   %ebx
    xorl    %eax, %eax
    movl    $2147483647, %ebx
    subl    $20, %esp
    movl    32(%esp), %esi
```

- direct : 3 instructions
  - %edx is used for the loop variable i
- indirect : 6 instructions
  - %esi is used for the argument fn
  - %ebx is used for the loop variable i

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

Differences (2) loop codes

direct assembly code

```
L3: movl    %eax, (%esp)  <---
    call    _foo              :
    subl    $1, %edx          :
    jne     L3            ----
```

indirect assembly code

```
L8: movl    %eax, (%esp)  <---
    call    *%esi             :
    subl    $1, %ebx          :
    jne L8                ----
```

- direct
  - 4 instructions
  - use 3 registers (%eax, %esp, %edx)
- indirect
  - 4 instructions
  - use 4 registers (%eax, %esp, %esi, %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

Differences (3) costs

- The extra setup overhead doesn't matter much, unless the loop count is huge
  - `direct` : 3 instructions in the setup code
  - `indirect` : 6 instructions in the setup code
- But the extra register use does matter.
  - `direct` : 3 registers (`%eax`, `%esp`, `%edx`)
  - `indirect` : 4 registers (`%eax`, `%esp`, `%esi`, `%ebx`)
  - register contention is often a problem

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