

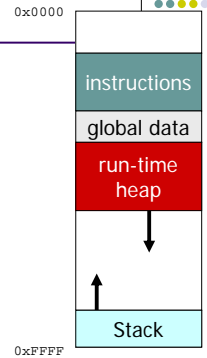
## Machine-Level Programming III: Control Flow, Stack frame

- Topic
  - Stack frames

1

## Programs & Memory

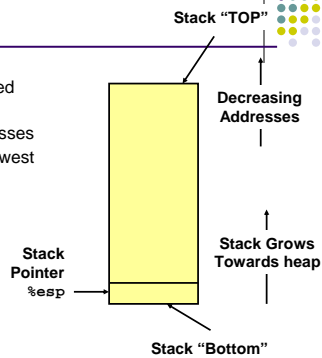
- The Von Neumann:
  - programs & data are stored in memory
- Recall assembly code converted to object code (**Assembler**)
- Labels are nothing more than address locations of instructions
- Here: `mov %eax, %edx`
- ...
- `jmp Here`



2

## IA32 Stack

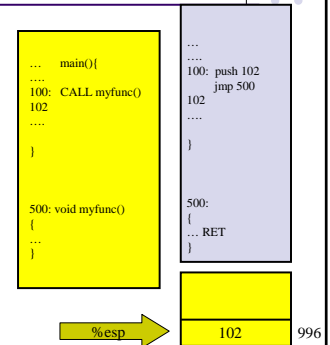
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element
- Push element on to stack
- Pop the top most element



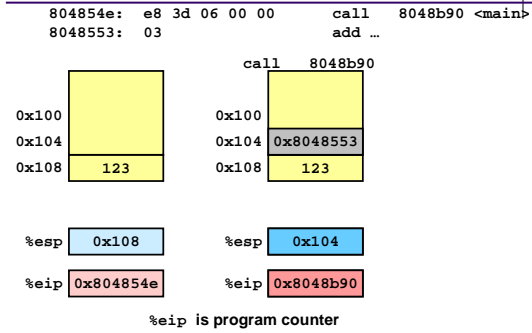
3

## Use of stack

- For storing return address of calling procedure
- When a procedure is called
- Program needs to jump to the starting address of the procedure
- After execution of the procedure, return to the next statement following the call statement
- `CALL .. Push 0102`
- `RET ... POP %eip`



## Procedure Call Example



5

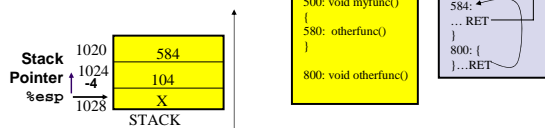
## Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call:
  - call label    Push return address on stack; Jump to label
- Return address value
  - Address of instruction beyond call
  - Example from disassembly
    - 804854e: e8 3d 06 00 00 call 8048b90 <main>
    - 8048553: 03            add ...
    - Return address = 0x8048553
- Procedure return:
  - ret            Pop address from stack; Jump to address

6

## Nested procedures

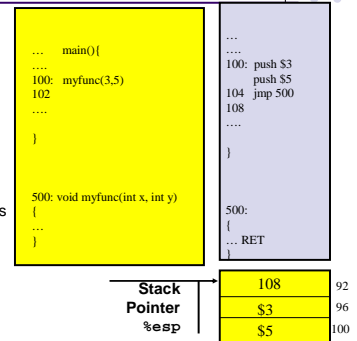
- On each call, push the return address on stack
- On RET, pop the address to return from the innermost procedure call
- RET → pop %eip



7

## Passing parameters

- What if the procedure has parameters?
- Use stack to store the values
- First,
  - Push parameters onto stack
- Where are the parameters?
- %esp + 4, %esp + 8
- %esp points to top of stack
- Second, Push return address
- Top of stack still contains the return address



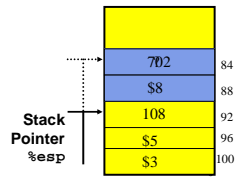
## Locating Parameters?

```

... main()
... myfunc(3,5)
108
...
}
500: void myfunc(int x, int y)
{
700 otherfunc(8)
702
...
RET
}
800: void otherfunc(int z)
{
...
RET
}

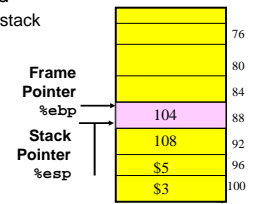
```

What happens on RET from the innermost procedure?  
 Pop decrements stack pointer  
 Because of parameters on stack, one POP does not point to the right location for the next level of the call sequence!!  
 $\%esp + 4, \%esp + 8$   
 Need to remember where the last call stack was



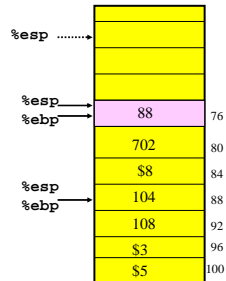
## Stack frames

- To preserve the offset for parameters for every called procedure
  - maintain another register called frame pointer
- $\%ebp$  is the frame pointer
- Every time a procedure is entered
  - push  $\%ebp$  old frame pointer on to stack
  - mov  $\%esp, \%ebp$
  - $\%ebp$  points to current SP
- When executing within myfunc
  - $\%ebp? \%esp?$
  - $\%ebp+8?, \%ebp+12?$



## Stack frames

- When myfunc calls otherfunc()
  - Push parameters and return address on stack
- Before entering otherfunc()
  - push  $\%ebp$  old frame pointer
  - mov  $\%esp, \%ebp$
  - $\%ebp$  points to current SP
- When executing within otherfunc()
  - $\%ebp? \%esp?$
  - $\%ebp+8?$
- On RET (clean up)
  - mov  $\%ebp, \%esp$
  - pop  $\%ebp$



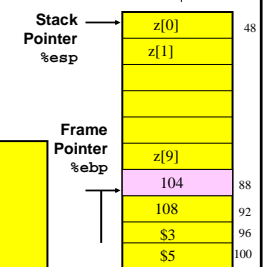
## What about local variables within procedures?

- Also goes on stack!
- The called procedure pushes local variables onto stack
- Where are the parameters?
  - $\%ebp + 8, \%ebp + 12$
- Where are the local variables?
  - $\%ebp-4, \%ebp-8, \dots$
- z is a local variable in myfunc()

```

... main()
... myfunc(3,5)
102
...
}
500: void myfunc(int x, int y)
{ int z[9];
  z[0]=x+y;
}

```



## Stack is used for local variables

- z[0] is stored in -40(%ebp)
- On RET from myfunc
- Clean up or LEAVE
- On RET (clean up)
  - mov %ebp, %esp
  - pop %ebp

```

...      main()
...
100:     push $3
...      push $5
102:     jmp 500
...
}
}

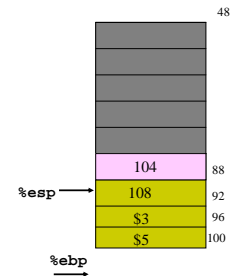
500:     void myfunc(int x, int y)
{ int z[9];
  z[0]=x+y;
  ...
}

500:     { mov $(%ebp), %eax
...      add 12(%ebp), %eax
...      mov %eax, -40(%ebp)
...      RET
}
    
```

13

## Cleaning the stack of arguments

- After mov %ebp, %esp
- pop %ebp
- RET instruction
- pop %eip
- %eip will point to the return address
- %esp is still pointing to the argument stack of the previous frame
- Two ways to clean up



14

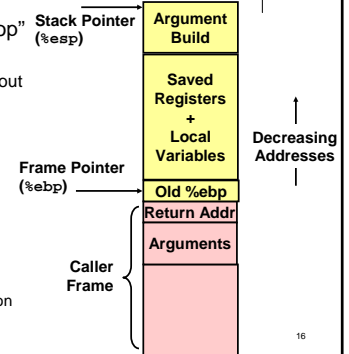
## Cleaning the stack of arguments

- Cleaning done by the called procedure
  - Use RET n instructions
  - RET n → pop %eip, and add %esp, \$n
  - Called procedure needs to know the size of the argument
- Cleaning done by the calling procedure
- After returning from the procedure
- addl sizeofargument, %esp

15

## x86 Stack Frame

- Current Stack Frame ("Top" Stack Pointer to Bottom)
  - Argument Build
  - Saved Registers + Local Variables
  - Old %ebp
  - Return Addr
  - Arguments
- Caller Stack Frame
  - Return address
  - Pushed by call instruction
  - Arguments for this call



16