IA32 Procedures
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses

- Register `%esp` contains lowest stack address
  - address of “top” element
IA32 Stack: Push

- **pushl** **Src**
  - Fetch operand at **Src**
  - Decrement \( \%esp \) by 4
  - Write operand at address given by \( \%esp \)
IA32 Stack: Pop

Stack Pointer: %esp

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Grows Down
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**:  
- Address of the next instruction right after call
- Example from disassembly

```
804854e:  e8 3d 06 00 00  call  8048b90
<main>
8048553:  50           pushl  %eax
```
- Return address = 0x8048553

**Procedure return**: `ret`
- Pop address from stack
- Jump to address
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

call 8048b90

%esp 0x108
%esp 0x104
%eip 0x804854e
%eip 0x8048b90

%eip: program counter
Procedure Return Example

8048591:  c3  ret

ret

0x110
0x10c
0x108 123
0x104 0x8048553

%esp 0x104  %esp 0x108
%eip 0x8048591  %eip 0x8048553

%eip:  program counter
Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be “Reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- **Stack discipline**
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does

- **Stack allocated in Frames**
  - state for single procedure instantiation
Call Chain Example

Procedure `amI()` is recursive
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - “Set-up” code
- Deallocated when return
  - “Finish” code
Example

```c
yoo(...) {
  •
  •
  who();
  •
  •
}
```

Stack

```
%ebp
%esp
```

```c
yoo
who
amI
amI
amI
```
Example

```c
yoo() {
  who(...) {
    ...
    amI();
    ...
    amI();
    ...
  }
}
```

Stack

```
yoo
who
%ebp
%esp
```
Example

```
yoo()
{
    who(...)
    {
        ami(...)
        {
            ...
            ami();
        }
    }
}
```

Stack

```
%ebp
%esp
```
Example

```
yoo() {
  who(...) {
    amI(...) {
      amI(...) {
        amI();
        amI();
      }
    }
  }
}
```
Example

```c
yoo()
{
    who(...)
    {
        amI(...)
        {
            amI(...)
            {
                amI(...)
                {
                    amI();
                    amI()
                    amI()
                }
            }
        }
    }
}

amI()
{
    amI()
    amI()
    amI()
}
```

Stack

- `yoo`
- `who`
- `amI`
- `amI`
- `%ebp`
- `%esp`
Example

```c
yoo( )
{
  who(…)
  {
  amI(…)
  {
    amI(…)
    {
      amI()
      {
        amI()
      }
    }
  }
}
```

Stack

```
yoo
who
amI
amI
%ebp
%esp
```
Example

```c
yoo() {
    who(...) {
        ami(...) {
            •
            •
            ami();
            •
        }
    }
}

ami() {
    •
    •
}
```

Stack

- yoo
- who
- ami

 `%ebp`  
 `%esp`
Example

```c
yoo (...) {
    who (...) {
        • • •
        amI ();
        • • •
        amI ();
        • • •
    }
    • • •
    who ();
    • • •
    amI ();
    • • •
}
```

Stack

```
%ebp ─── yoo

%esp ─── who
```

Diagram: [Diagram of stack and function calls]
Example

```c
yoo() {
  who(...) {
    amI(...) {
      •
      •
      amI();
      •
      •
    }
  }
}
```

Stack

```
%ebp → %esp → yoo → who → amI → amI
```
Example

```
void yoo() {
    who(...) {
        • • •
        ami();
        • • •
        ami();
        • • •
    }
}
```

Stack

```
%ebp

%esp

who

yoo
```
Example

```c
yoo(...) {
    •
    •
    who();
    •
    •
}
```
IA32/Linux Stack Frame

- **Current Stack Frame (“Top” to Bottom)**
  - “Argument build:”
    Parameters for function about to call
  - Local variables
    If can’t keep in registers
  - Saved register context
  - Old frame pointer

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call

```
IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - “Argument build:”
    Parameters for function about to call
  - Local variables
    If can’t keep in registers
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
```
Revisiting swap

```c
int course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}

void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```assembly
call_swap:
  . . .
  subl $8, %esp
  movl $course2, 4(%esp)
  movl $course1, (%esp)
  call swap
  . . .
```

Resulting Stack

```
| &course2 |
| &course1 |
| Rtn adr |
```

subl %esp
call %esp
Revisiting swap

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx

    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)

    popl %ebx
    popl %ebp
    ret
```
swap Setup #1

Entering Stack

•
•
•
&course2
&course1
Rtn adr

Resulting Stack

%ebp

•
•
•

%ebp
YP
xp
Rtn adr
Old %ebp

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Setup #2

Entering Stack

\[
\begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
& \text{course2} \\
& \text{course1} \\
\text{Rtn adr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\end{array}
\]

Resulting Stack

\[
\begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
& \text{YP} \\
& \text{xp} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\end{array}
\]

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\end{array}
\]

\[
\text{swap:}
\]

\[
\begin{array}{c}
\text{pushl \%ebp} \\
\text{movl \%esp,\%ebp} \\
\text{pushl \%ebx} \\
\end{array}
\]
swap Setup #3

Entering Stack

- %ebp
- &course2
- &course1
- Rtn adr

Resulting Stack

- %ebp
- %esp
- Rtn adr
- Old %ebp
- Old %ebx
- yp
- xp

swap:

pushl %ebp
movl %esp, %ebp
pushl %ebx
swap Body

Entering Stack

Resulting Stack

Offset relative to %ebp

&course2

12

yp

%ebp

&course1

8

xp

%esp

Rtn adr

4

Rtn adr

Old %ebp

Old %ebx

%esp

%ebp

movl 8(%ebp),%edx  # get xp
movl 12(%ebp),%ecx  # get yp

...
swap Finish

Stack Before Finish

Resulting Stack

- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx
Disassembled swap

08048384 <swap>:

8048384:  55
8048385:  89 e5
8048387:  53
8048388:  8b 55 08
804838b:  8b 4d 0c
804838e:  8b 1a
8048390:  8b 01
8048392:  89 02
8048394:  89 19
8048396:  5b
8048397:  5d
8048398:  c3

push %ebp
mov %esp,%ebp
push %ebx
mov 0x8(%ebp),%edx
mov 0xc(%ebp),%ecx
mov (%edx),%ebx
mov (%ecx),%eax
mov %eax,(%edx)
mov %ebx,(%ecx)
pop %ebx
pop %ebp
ret

Calling Code

80483b4:  movl $0x8049658,0x4(%esp)  # Copy &course2
80483bc:  movl $0x8049654,(%esp)    # Copy &course1
80483c3:  call 8048384 <swap>      # Call swap
80483c8:  leave                   # Prepare to return
80483c9:  ret                      # Return
Today

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the **caller**
  - `who` is the **callee**

- Can register be used for temporary storage?

```assembly
yoo:
  ...
  movl $15213, %edx
  call who
  addl %edx, %eax
  ...
  ret

who:
  ...
  movl 8(%ebp), %edx
  addl $18243, %edx
  ...
  ret
```

- Contents of register `%edx` overwritten by `who`
- This could be trouble ➔ something should be done!
  - Need some coordination
Register Saving Conventions

- When procedure **you** calls **who**:
  - **you** is the caller
  - **who** is the callee

- Can register be used for temporary storage?

- Conventions
  - “Caller Save”
    - Caller saves temporary values in its frame before the call
  - “Callee Save”
    - Callee saves temporary values in its frame before using
IA32/Linux+Windows Register Usage

- `%eax, %edx, %ecx`  
  - Caller saves prior to call if values are used later

- `%eax`  
  - also used to return integer value

- `%ebx, %esi, %edi`  
  - Callee saves if wants to use them

- `%esp, %ebp`  
  - special form of callee save  
  - Restored to original values upon exit from procedure
Today

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

 Registers

- %eax, %edx used without first saving
- %ebx used, but saved at beginning & restored at end

pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shrl %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax

.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return (x & 1) + pcount_r(x >> 1);
}

- **Actions**
  - Save old value of %ebx on stack
  - Allocate space for argument to recursive call
  - Store x in %ebx

```
pcount_r:
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
  ...
```
Recursive Call #2

/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

- Actions
  - If x == 0, return
    - with %eax set to 0

movl  $0, %eax
testl  %ebx, %ebx
je   .L3
    .L3:
    ret
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

- **Actions**
  - Store \(x \gg 1\) on stack
  - Make recursive call

- **Effect**
  - \(%eax\) set to function result
  - \(%ebx\) still has value of \(x\)

```
    movl  %ebx, %eax
    shrl  %eax
    movl  %eax, (%esp)
call   pcount_r
    ...
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

Assume
- %eax holds value from recursive call
- %ebx holds x

Actions
- Compute (x & 1) + computed value

Effect
- %eax set to function result
Recursive Call #5

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

### Actions
- Restore values of `%ebx` and `%ebp`
- Restore `%esp`

```
L3:
addl$4, %esp
popl%ebx
popl%ebp
ret
```
Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out

- **Also works for mutual recursion**
  - P calls Q; Q calls P
**Pointer Code**

**Generating Pointer**

```c
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

**Referencing Pointer**

```c
/* Increment value by k */
void incrk(int *ip, int k) {
    *ip += k;
}
```

- **add3** creates pointer and passes it to **incrk**
Creating and Initializing Local Variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- **Variable localx must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as -4(%ebp)**

First part of add3

```
add3:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp  # Alloc. 24 bytes
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp)# Set localx to x
```
Creating Pointer as Argument

```
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Use leal instruction to compute address of localx

Middle part of add3

```
movl $3, 4(%esp)  # 2^{nd} arg = 3
leal -4(%ebp), %eax# &localx
movl %eax, (%esp)  # 1^{st} arg = &localx
call incrk
```

- Use leal instruction to compute address of localx
Retrieving local variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Retrieve localx from stack as return value

Final part of add3

```assembly
movl -4(%ebp), %eax  # Return val = localx
leave
ret
```
IA 32 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - Recursion (& mutual recursion) handled by normal calling conventions
    - Can safely store values in local stack frame and in callee-saved registers
    - Put function arguments at top of stack
    - Result return in `%eax`

- **Pointers are addresses of values**
  - On stack or global