IA32 Stack
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element

IA32 Stack Pushing
Pushing
- pushl Src
- Fetch operand at Src
- Decrement %esp by 4
- Write operand at address given by %esp

IA32 Stack Popping
Popping
- popl Dest
- Read operand at address given by %esp
- Increment %esp by 4
- Write to Dest

IA32 Stack
- Stack discipline
- Register saving conventions
- Creating pointers to local variables

x86-64
- Argument passing in registers
- Minimizing stack usage
- Using stack pointer as only reference

IA32 Stack
- Stack discipline
- Register saving conventions
- Creating pointers to local variables
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: 50  pushl %eax
  - Return address = 0x8048553

Procedure return:
- ret  Pop address from stack; Jump to address

Procedure Call Example

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

Procedure Return Example

- Example from disassembly
  8048591: c3  ret

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

Code Structure

```plaintext
yoo(…)
{
  ...
  who();
  ...
}

who(…)
{
  ...
  amI();
  ...
}

amI(…)
{
  ...
  ...
}
```

- Procedure amI recursive

Stack Frames

Contents
- Local variables
- Return information
- Temporary space

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

Pointers
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame

Stack Operation

Stack Operation
Stack Operation

amI(...) {
    •
    • amI();
    •
}

Call Chain

Stack Operation

amI(...) {
    •
    • amI();
    •
}

Call Chain

Stack Operation

amI(...) {
    •
    • amI();
    •
}

Call Chain

Stack Operation

amI(...) {
    •
    • amI();
    •
}

Call Chain

Stack Operation

amI(...) {
    •
    • amI();
    •
}

Call Chain
Stack Operation

```
amI(...) {
  ...
  amI();
  ...
}
```

Call Chain

```
   yoo
   ↓ who
   ↓ amI
   ↓ amI
```

Frame Pointer %ebp
Stack Pointer %esp

Stack Operation

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

Call Chain

```
   who
   ↓ amI
   ↓ amI
   ↓ amI
```

Frame Pointer %ebp
Stack Pointer %esp

Stack Operation

```
amI(...) {
  ...
  amI();
  ...
}
```

Call Chain

```
   yoo
   ↓ who
   ↓ amI
   ↓ amI
```

Frame Pointer %ebp
Stack Pointer %esp

Stack Operation

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

Call Chain

```
   who
   ↓ amI
   ↓ amI
   ↓ amI
```

Frame Pointer %ebp
Stack Pointer %esp
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}

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

void call_swap()
{
    swap(&zip1, &zip2);
    swap(&zip2, &zip1);
}
```

Resulting Stack

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

IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)
- Parameters for function about to call
  - "Argument build"
- 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
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
**swap Setup #1**

Entering Stack

Resulting Stack

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\text{\$zip2} \\
\text{\$zip1} \\
\text{Rtn adr}
\end{array}
\]

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\text{\texttt{yp}} \\
\text{\texttt{xp}} \\
\text{Rtn adr} \\
\text{Old \%ebp}
\end{array}
\]

\[
\text{swap:} \\
pushl \%ebp \\
movl \%esp,\%ebp \\
pushl \%ebx
\]

Effect of swap Setup

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\text{\texttt{yp}} \\
\text{\texttt{xp}} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\text{Old \%ebx}
\end{array}
\]

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%esp} \\
\text{\texttt{yp}} \\
\text{\texttt{xp}} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\text{Old \%ebx}
\end{array}
\]

\[
\text{movl 12(\%ebp),%ecx \# get yp} \\
\text{movl 8(\%ebp),%edx \# get xp}
\]

\text{Body
swap Finish #1

**Observation**
- Saved & restored register %ebx

swap Finish #2

Observation
- Saved & restored register %ebx

swap Finish #3

Observation
- Saved & restored register %ebx
- Didn't do so for %eax, %ecx, or %edx

swap Finish #4

Observation
- Saved & restored register %ebx
Register Saving Conventions

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

Can Register be Used for Temporary Storage?

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

`who`:
```
  * * *
  movl 8(%ebp), %edx
  addl $91125, %edx
  * * *
  ret
```

- Contents of register `%edx` overwritten by `who`

Register Saving Conventions

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

Can Register be Used for Temporary Storage?

Conventions
- “Caller Save”
  - Caller saves temporary in its frame before calling
- “Callee Save”
  - Callee saves temporary in its frame before using

IA32/Linux Register Usage

Integer Registers
- Two have special uses
  - `%ebp`, `%esp`
- Three managed as callee-save
  - `%ebx`, `%esi`, `%edi`
  - Old values saved on stack prior to using
- Three managed as caller-save
  - `%eax`, `%edx`, `%ecx`
  - Do what you please, but expect any callee to do so, as well
- Register `%eax` also stores returned value

Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

Registers
- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end
### Rfact Stack Setup

**Entering Stack**

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

#### Callers
- `%ebp` pre
- `%ebx` pre
- `%esp`

### Rfact Body

#### Recursion
- `movl $8(%ebp),%ebx` # ebx = x
- `cmpl $1,%ebx` # Compare x : 1
- `jle .L78` # If <= goto Term
- `leal -1(%ebx),%eax` # eax = x-1
- `pushl %eax` # Push x-1
- `call rfact` # rfact(x-1)
- `imull %ebx,%eax` # rval * x
- `jmp .L79` # Goto done
- `.L78:` # Term:
  - `movl $1,%eax` # return val = 1
  - `.L79:` # Done:

#### Registers
- `%ebx` Stored value of x
- `%eax` Returned value from rfact(x-1)
- Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call

### Rfact Recursion

- `leal -1(%ebx),%eax`
- `pushl %eax`

### Rfact Result

#### Return from Call
- `imull %ebx,%eax`

#### Register
- Assume that rfact(x-1) returns (x-1)! in register `%eax`
**Rfact Completion**

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

**Pointer Code**

**Recursive Procedure**

```c
void s_helper (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

**Top-Level Call**

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

**Creating & Initializing Pointer**

**Initial part of sfact**

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

**Passing Pointer**

**Calling s_helper from sfact**

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```
### Using Pointer

```c
void s_helper(int x, int *accum) {
    int z = *accum * x;
    *accum = z;
}
```

- Register `%ecx` holds `x`
- Register `%edx` holds `accum`

- Assume memory initially has value `V`
- Use access `%edx` to reference memory

### IA 32 Procedure Summary

**The Stack Makes Recursion Work**
- Private storage for each *instance* of procedure call
  - Instantiations don’t clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

**IA32 Procedures Combination of Instructions + Conventions**
- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - `%ebp` and `%esp`
- Stack frame organization conventions

### x86-64 General Purpose Registers

<table>
<thead>
<tr>
<th><code>%rax</code></th>
<th><code>%eax</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rbx</code></td>
<td><code>%ebx</code></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td><code>%ecx</code></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td><code>%edx</code></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td><code>%esi</code></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td><code>%edi</code></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td><code>%esp</code></td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><code>%ebp</code></td>
</tr>
</tbody>
</table>

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits

### x86-64 Register Conventions

<table>
<thead>
<tr>
<th><code>%rax</code></th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rbx</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack Pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>%r8</code></th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%r9</code></td>
<td>Argument #6</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>Used for linking</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

Arguments passed to functions via registers
- If more than 6 integral parameters, then pass rest on stack
- These registers can be used as caller-saved as well

All References to Stack Frame via Stack Pointer
- Eliminates need to update %ebp

Other Registers
- 6+1 callee saved
- 2 or 3 have special uses

x86-64 Long Swap

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

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required

Avoiding Stack
- Can hold all local information in registers

x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

Avoiding Stack Pointer Change
- Can hold all information within small window beyond stack pointer

x86-64 NonLeaf without Stack Frame

long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_se(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    scount++;
}

- No values held while swap being invoked
- No callee save registers needed
When swap executes `ret`, it will return from `swap_ele`. Possible since swap is a “tail call.”

```c
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

```assembly
swap_ele:
    movslq %esi,%rsi       # Sign extend i
    leaq (%rdi,%rsi,8), %rdi # &a[i]
    leaq 8(%rdi), %rsi     # &a[i+1]
    jmp swap               # swap()
```

Keeps values of `a` and `i` in callee save registers. Must set up stack frame to save these registers.

```c
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}
```

```assembly
swap_ele_su:
    movq %rbx, -16(%rsp)   # Save %rbx
    movslq %esi,%rbx      # Extend & save i
    movq %r12, -8(%rsp)   # Save %r12
    movq %rdi, %r12       # Save a
    leaq (%rdi,%rbx,8), %rdi # &a[i]
    subq $16, %rsp        # Allocate stack frame
    leaq 8(%rdi), %rsi    # &a[i+1]
    call swap             # swap()
    movq (%r12,%rbx,8), %rax # a[i]
    addq %rax, sum(%rip)  # sum += a[i]
    movq (%rsp), %rbx    # Restore %rbx
    movq (%rsp), %r12    # Restore %r12
    addq $16, %rsp       # Deallocate stack frame
    ret
```

Understanding x86-64 Stack Frame

```assembly
swap_ele_su:
    movq %rbx, -16(%rsp)   # Save %rbx
    movslq %esi,%rbx      # Extend & save i
    movq %r12, -8(%rsp)   # Save %r12
    movq %rdi, %r12       # Save a
    leaq (%rdi,%rbx,8), %rdi # &a[i]
    subq $16, %rsp        # Allocate stack frame
    leaq 8(%rdi), %rsi    # &a[i+1]
    call swap             # swap()
    movq (%r12,%rbx,8), %rax # a[i]
    addq %rax, sum(%rip)  # sum += a[i]
    movq (%rsp), %rbx    # Restore %rbx
    movq (%rsp), %r12    # Restore %r12
    addq $16, %rsp       # Deallocate stack frame
    ret
```

Stack Operations

```assembly
movq %rbx, -16(%rsp)  # Save %rbx
movq %r12, -8(%rsp)   # Save %r12
subq $16, %rsp        # Allocate stack frame
movq (%rsp), %rbx    # Restore %rbx
movq 8(%rsp), %r12   # Restore %r12
addq $16, %rsp       # Deallocate stack frame
```
Interesting Features of Stack Frame

Allocate Entire Frame at Once
- All stack accesses can be relative to %rsp
- Do by decrementing stack pointer
- Can delay allocation, since safe to temporarily use red zone

Simple Deallocation
- Increment stack pointer

---

x86-64 Procedure Summary

Heavy Use of Registers
- Parameter passing
- More temporaries

Minimal Use of Stack
- Sometimes none
- Allocate/deallocate entire block

Many Tricky Optimizations
- What kind of stack frame to use
- Calling with jump
- Various allocation techniques