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 discipline
- Register saving conventions
- Creating pointers to local variables

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

15-213, F'08
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

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

call 8048b90

0x100 0x100
0x10c 0x10c
0x108 123 0x108 123
0x104 0x8048553

%esp 0x108 %esp 0x104
%eip 0x804854e %eip 0x8048b90
%eip is program counter

Procedure Return Example

8048591: c3  ret

0x100 0x100
0x10c 0x10c
0x108 123 0x108 123
0x104 0x8048553

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

Code Structure

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

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

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

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

Frame Pointer `%ebp`

Stack Pointer `%esp`

Stack "Top"

```c
who(...) {
    ...
    amI();
    ...
    amI();
    ...
}
```
Stack Operation

```
amI(...) {
  yoo
  who
  amI()
  amI
}
```
Stack Operation

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

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
int zip1 = 15213;
int zip2 = 91125;

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

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

Call Chain

```c
yoo
    who()
    amI
    amI
    amI
    amI
```

---

Revisiting swap

```
call_swap:      . . .
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap    . . .
```

```
resulting Stack

<table>
<thead>
<tr>
<th>&amp;zip2</th>
<th>&amp;zip1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rtn adr</td>
<td>%esp</td>
</tr>
</tbody>
</table>
```

---

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

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %eax,%ecx
    popl %ebp
    ret
```

---

**Set Up**
- pushl %ebp
- movl %esp,%ebp
- pushl %ebx
- movl 12(%ebp),%ecx
- movl 8(%ebp),%edx
- movl (%ecx),%eax
- movl (%edx),%ebx
- movl %eax,(%edx)
- movl %ebx,(%ecx)
- movl -4(%ebp),%ebx
- movl %eax,%ecx
- popl %ebp
- ret

**Body**

**Finish**
**Swap Setup #1**

**Entering Stack**

- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr

**Resulting Stack**

- %ebp
- %esp
- yp
- xp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

**Swap:**

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

**Resulting Stack**

- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

---

**Swap Setup #2**

**Entering Stack**

- %ebp
- %esp
- yp
- xp
- &zip2
- &zip1
- Rtn adr

**Resulting Stack**

- %ebp
- %esp
- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

**Swap:**

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

**Resulting Stack**

- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

---

**Swap Setup #3**

**Entering Stack**

- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr

**Resulting Stack**

- %ebp
- %esp
- yp
- xp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

**Swap:**

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

**Resulting Stack**

- %ebp
- %esp
- &zip2
- &zip1
- Rtn adr
- Old %ebp

---

**Effect of Swap Setup**

**Entering Stack**

- %ebp
- %esp

**Resulting Stack**

- %ebp
- %esp
- Offset (relative to %ebp)

- &zip2
- yp
- 12

- &zip1
- xp
- 8

- Rtn adr
- 4

- %ebp
- %esp
- Old %ebp
- Old %ebx

**Swap:**

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

**Body**
### swap Finish #1

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Observation**

- Saved & restored register %ebx

### swap Finish #2

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Observation**

- Saved & restored register %ebx

### swap Finish #3

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Observation**

- Saved & restored register %ebx

### swap Finish #4

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 0      |    |    |         | &zip2
| &zip1 |

**Observation**

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

080483a4 <swap>:
80483a4:  55          push   %ebp
80483a5:  89 e5       mov   %esp,%ebp
80483a7:  53          push   %ebx
80483a8:  8b 55 08    mov   0x8(%ebp),%edx
80483ab:  8b 4d 0c    mov   0xc(%ebp),%ecx
80483ae:  8b 1a       mov   (%edx),%ebx
80483b0:  8b 01       mov   (%ecx),%eax
80483b2:  89 02       mov   %eax,(%edx)
80483b4:  89 19       mov   %ebx,(%ecx)
80483b6:  5b          pop    %ebx
80483b7:  c9          leave
80483b8:  c3          ret

Register Saving Conventions

When procedure \textit{yoo} calls \textit{who}:
\begin{itemize}
\item \textit{yoo} is the \textit{caller}, who is the \textit{callee}
\end{itemize}

Can Register be Used for Temporary Storage?

\begin{itemize}
\item Contents of register %edx overwritten by \textit{who}
\end{itemize}

Calling Code

8048409:  e8 96 ff ff ff   call 80483a4 <swap>
804840e:  8b 45 f8       mov 0xfffffffff8(%ebp),%eax

IA32/Linux Register Usage

Integer Registers

\begin{itemize}
\item Two have special uses %ebp, %esp
\item Three managed as callee-save %ebx, %esi, %edi
\item Old values saved on stack prior to using
\item Three managed as caller-save %eax, %edx, %ecx
\item Do what you please, but expect any callee to do so, as well
\item Register %eax also stores returned value
\end{itemize}
Recursive Factorial

```c
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

![Stack Diagram]

Rfact Body

```assembly
.globl rfact
.type rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
.align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Recursion

```assembly
movl 8(%ebp),%ebx  # ebx = x
```

```assembly
cmpl $1,%ebx      # Compare x : 1
jle .L78          # If <= goto Term
```

```assembly
leal -1(%ebx),%eax  # eax = x-1
pushl %eax          # Push x-1
```

```assembly
call rfact          # rfact(x-1)
```

```assembly
imull %ebx,%eax     # rval * x
```

```assembly
jmp .L79            # Goto done
```

.L78:  # Term:

```assembly
movl $1,%eax        # return val = 1
```

.L79:  # Done:

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

Return from Call

Assume that rfact(x-1) returns (x-1)! in register %eax

Rfact Completion

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

Pointer Code

Recursive Procedure

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

- Pass pointer to update location

Top-Level Call

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

Creating & Initializing Pointer

Initial part of sfact

lsenfact:
pushl %ebp # Save %ebp
movl %esp,%ebp # Set %ebp
subl $16,%esp # Add 16 bytes
movl 8(%ebp),%edx # edx = x
movl $1,-4(%ebp) # val = 1

Using Stack for Local Variable

- Variable val must be stored on stack
  - Need to create pointer to it
- Compute pointer as – 4(%ebp)
- Push on stack as second argument

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;
}
```

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

- Twice the number of registers
  - Accessible as 8, 16, 32, or 64 bits
### x86-64 Register Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Used for linking</td>
</tr>
<tr>
<td>%r12</td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r15</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

```c
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

### x86-64 Locals in the Red Zone

```c
/* 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];
}
```

- Can hold all information within small window beyond stack pointer

### Avoiding Stack Pointer Change

- Can hold all local information in registers

```c
void swap_a(long *xp, long *yp)
{
    movq (%rdi), %rax
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rax, (%rsi)
    ret
}
```
**x86-64 NonLeaf without Stack Frame**

- No values held while swap being invoked
- No callee save registers needed

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

```
swap_ele_se:
    movslq %esi,%rsi       # Sign extend i
    leaq (%rdi,%rsi,8), %rdi # &a[i]
    leaq 8(%rdi), %rsi     # &a[i+1]
    call  swap              # swap()
    incq  scount(%rip)      # scount++;
    ret
```

**x86-64 Call using Jump**

- 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]);
}
```

```
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()
```

**x86-64 Stack Frame Example**

- 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];
}
```

```
swap_ele_su:
    movq %rbx, -16(%rsp)  # Save %rbx
    movslq %esi,%rbx      # Save %esi
    movq %r12, -8(%rsp)   # Save %r12
    movq %rdi, %r12       # Save %rdi
    leaq (%rdi,%rbx,8), %rdi
    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 8(%rsp), %r12    # Restore %r12
    addq $16, %rsp        # Deallocate stack frame
    ret
```

**Understanding x86-64 Stack Frame**

```c
void swap
    (long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

```
swap:
    movslq %esi,%rsi       # Sign extend i
    leaq (%rdi,%rsi,8), %rdi # &a[i]
    leaq 8(%rdi), %rsi     # &a[i+1]
    call  swap              # swap()
```
Stack Operations

```assembly
movq %rbx, -16(%rsp)  # Save %rbx
movq %r12, -8(%rsp)   # Save %r12
subq $16, %rsp       # Allocate stack frame
```

```assembly
rtn Ptr

%rsp
%rbx
%r12
```

```assembly
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