Mechanisms in Procedures

Passing Control
- To beginning of procedure code
- Back to return point

Passing Data
- Procedure arguments
- Return value

Memory Management
- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required.

\[
P(...) \{
  \ldots
  y = Q(x);
  printf(y);
  \ldots
\}
\]

int Q(int i)
{
  int t = 3*i;
  int v[10];
  \ldots
  return v[t];
}\n
x86-64 Stack

Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp contains lowest stack address—address of “top element”

Pushing
- pushq Src
- Decrement %rsp by 8
- Write operand at address given by %rsp

\[
P(...) \{
  \ldots
  y = Q(x);
  printf(y);
  \ldots
\}
\]

int Q(int i)
{
  int t = 3*i;
  int v[10];
  \ldots
  return v[t];
}\n
\[
P (...) \{
  \ldots
  y = Q ( x ) ;
  printf ( y ) ;
  \ldots
\}
\]

int Q ( int \ i )
{
  int t = 3 \ast i ;
  int v [ 1 0 ] ;
  \ldots
  return v [ t ] ;
}\n
CS429 Slideset 9: 1 Instruction Set Architecture IV

CS429 Slideset 9: 2 Instruction Set Architecture IV

CS429 Slideset 9: 3 Instruction Set Architecture IV

CS429 Slideset 9: 4 Instruction Set Architecture IV
**Popping**
- popq Dest
- Read operand at address given by %rsp
- Increment %rsp by 8
- Write to Dest

```c
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

**Code Examples (2)**

```c
long mult2 (long a, long b) {
    long s = a * b;
    return s;
}
```

**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 next instruction right after call

**Procedure return:** ret
- Pop address from stack
- Jump to address
Control Flow Example #1

Poised to execute call in multstore.

```
0000000000400540 <multstore>:
  400544: callq 400580 <mult2>
  40054D: mov %rax, (%rbx)
...

0000000000400580 <mult2>:
  400580: mov %rdi, %rax
  400587: retq
```

```
0x130
0x128
0x120

%rsp 0x120
%rip 0x400544
```

Control Flow Example #2

After call in multstore. Now in mult2.

```
0000000000400540 <multstore>:
  ...
  400544: callq 400580 <mult2>
  40054D: mov %rax, (%rbx)
...

0000000000400580 <mult2>:
  400580: mov %rdi, %rax
  400587: retq
```

```
0x130
0x128
0x120
0x118
0x40054D

%rsp 0x118
%rip 0x400580
```

Control Flow Example #3

Execute through mult2 to retq.

```
0000000000400540 <multstore>:
  ...
  400544: callq 400580 <mult2>
  40054D: mov %rax, (%rbx)
...

0000000000400580 <mult2>:
  400580: mov %rdi, %rax
  400587: retq
```

```
0x130
0x128
0x120
0x118
0x40054D

%rsp 0x118
%rip 0x400587
```

Control Flow Example #4

After retq in mult2. Back in multstore.

```
0000000000400540 <multstore>:
  ...
  400544: callq 400580 <mult2>
  40054D: mov %rax, (%rbx)
...

0000000000400580 <mult2>:
  400580: mov %rdi, %rax
  400587: retq
```

```
0x130
0x128
0x120
0x118
0x40054D

%rsp 0x120
%rip 0x40054D
```
Procedure Data Flow

Registers: First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

Mnemonic to remember the order: “Diane’s silk dress cost $89.” Args past 6 are pushed on the stack in reverse order.

Return value

%rax

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 a limited time: from call to return
- Callee returns before caller does

Stack allocated in Frames
- State for a single procedure instantiation

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 a limited time: from call to return
- Callee returns before caller does

Stack allocated in Frames
- State for a single procedure instantiation

Call Chain Example

Code Structure

Procedure aml is recursive.

Stack Frames

In IA32, almost every procedure call generated a stack frame. In x86-64, most calls do not generate an explicit frame.

Contents
- Return information
- Local storage (if needed)
- Temporary space (if needed)

Management
- Space allocated when enter procedure
  - “Set-up” code
  - Includes push by call instruction
- Deallocated when returning
  - “Finish” code
  - Includes pop by ret instruction
Example (1)

```c
/* Executing here: */
yoo(...)
{
    ...
    who();
    ...
}
```

Stack

```
%rbp  →  
%rsp  →  yoo
```

Example (2)

```c
/* Executing here: */
yoo(...)
{
    ...
    who();
    ...
}
```

Stack

```
%rbp  →  
%rsp  →  who
```

Example (3)

```c
/* Executing here: */
yoo(...)
{
    ...
}
```

Stack

```
%rbp  →  
%rsp  →  who
```

Example (4)

```c
/* Executing here: */
amI(...)
{
    ...
    amI();
    ...
}
```

Stack

```
%rbp  →  
%rsp  →  amI
```
Example (5)

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

who(...)
{
    ...
}

/* Executing here: */
amI(...)
{
    ...
    amI();
    ...
}
```

Example (6)

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

who(...)
{
    ...
}

/* Executing here: */
amI(...)
{
    ...
    amI();
    ...
}
```

Example (7)

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

who(...)
{
    ...
}

/* Executing here: */
amI(...)
{
    ...
    amI();
    ...
}
```

Example (8)

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

/* Executing here: */
who(...)
{
    ...
    amI();
    ...
}
```
### x86-64/Linux Stack Frame

#### Current Stack Frame
- "Argument build:" parameters for function about to call
- Local variables, if can’t keep in registers
- Saved register context
- Old frame pointer (optional)

#### Caller Stack Frame
- Return address (pushed by call instruction)
- Arguments for this call
Building a Stack Frame

IA32 routines almost always constructed an explicit stack frame; x86-64 routines usually don’t. If you do build a stack frame:

```
proc:
pushq %rbp    # save caller’s frame base
movq %rsp, %rbp  # set callee’s frame base
...           # body of routine proc
movq %rbp, %rsp  # discard callee’s frame
popq %rbp      # reset caller’s frame base
ret
```

Reserve %rbp for this purpose if you’re doing this.

What does this do?

Example: Calling incr #1

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```

call_incr:

```
subq $16, %rsp
movq $15213, 8(%rsp)
movl $3000, %esi
leaq 8(%rsp), %rdi
call incr
addq 8(%rsp), %rax
addq $16, %rsp
ret
```

Initial Stack Structure

```
Initial Stack Structure
Rtn addr ← %rsp
```

Resulting Stack Structure

```
Resulting Stack Structure
Rtn addr ← %rsp
```

Example: Calling incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```

call_incr:

```
subq $16, %rsp
movq $15213, 8(%rsp)
movl $3000, %esi
leaq 8(%rsp), %rdi
call incr
addq 8(%rsp), %rax
addq $16, %rsp
ret
```

Stack Structure

```
Stack Structure
Rtn addr ← %rsp
```

This example doesn’t use explicit frames!

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

Register Use(s)

```
%rdi | Argument p
%rsi | Argument val, y
%rax | x, return value
```
Example: Calling incr #3

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```

call_incr:
- `subq $16, %rsp`
- `movq $15213, 8(%rsp)`
- `movl $3000, %esi`
- `leaq 8(%rsp), %rdi`
- `call incr`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

Stack Structure

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>18213</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>

Example: Calling incr #4

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```

call_incr:
- `subq $16, %rsp`
- `movq $15213, 8(%rsp)`
- `movl $3000, %esi`
- `leaq 8(%rsp), %rdi`
- `call incr`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

Stack Structure

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>18213</td>
<td></td>
</tr>
</tbody>
</table>

Update Stack Structure

Updated Stack Structure

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>18213</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>

Example: Calling incr #5

```c
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1 + v2;
}
```

call_incr:
- `subq $16, %rsp`
- `movq $15213, 8(%rsp)`
- `movl $3000, %esi`
- `leaq 8(%rsp), %rdi`
- `call incr`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

Updated Stack Structure

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>

Final Stack Structure

<table>
<thead>
<tr>
<th>Rtn addr</th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Saving Conventions

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

Can register be used for temporary storage?

- Contents of register `%rdx` are overwritten by `who`
- This could be trouble; something should be done!
- Need to coordinate between caller and callee.
**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 Saved”: Caller saves temporary values in its frame before the call
- “Callee Saved”:
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller

---

**x86-64 Linux Caller-saved Registers**

%rax
- Return value
- Also caller-saved
- Can be modified by procedure

%rdi, %rsi, %rdx, %rcx, %r8, %r9
- Arguments
- Also caller-saved
- Can be modified by procedure

%r10, %r11
- Caller-saved
- Can be modified by procedure

---

**x86-64 Linux Callee-saved Registers**

%rbx, %r12, %r13, %r14, %r15
- Callee-saved
- Callee must save and restore

%rbp
- Callee-saved
- Callee must save and restore
- May be used as frame pointer
- Can mix and match

%rsp
- Special form of callee-saved
- Restored to original value upon exit from procedure

---

**Callee-Saved Example #1**

```c
long call_incr2( long x ) {
    long v1 = 15213;
    long v2 = incr( &v1, 3000 );
    return x + v2;
}
```

---

**Initial Stack Structure**

- Rtn address ← %rsp

**Resulting Stack Structure**

- Rtn address
- Saved %rbx ← %rsp+8
- Unused ← %rsp
Callee-Saved Example #2

```c
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x + v2;
}
```

Resulting Stack Structure

<table>
<thead>
<tr>
<th>Rtn address</th>
<th>← %rsp+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>← %rsp</td>
</tr>
</tbody>
</table>

Pre-return Stack Structure

| Rtn address | ← %rsp  |

Recursive Function Terminal Case

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Recursive Function Register Save

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Recursive Function Call Setup

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

### Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x &gt;&gt; 1</td>
<td>Rec. argument</td>
</tr>
<tr>
<td>%rax</td>
<td>x &amp; 1</td>
<td>Caller-saved</td>
</tr>
</tbody>
</table>

Recursive Function Call

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

### Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>Recursive call return value</td>
</tr>
</tbody>
</table>

Recursive Function Result

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

### Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>Return value</td>
</tr>
</tbody>
</table>

Recursive Function Completion

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

### Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
<td>Return value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... ← %rsp
Observations About Recursion

Handled without Special Consideration
- Stack frames mean that each function call has private storage
  - Saved registers and local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another’s data
  - Unless the C code explicitly does so (e.g., buffer overflow)
- 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

Important Points
- Stack is the right data structure for procedure call / return; if P calls Q, then Q returns before P

Recursion (and mutual recursion) are 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 %rax

Pointers are addresses of values (on stack or global)