Machine-Level Programming III: Procedures

CSci 2021: Machine Architecture and Organization
February 26th-28th, 2020
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron

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

These Slides
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
    - Illustration of Recursion

x86-64 Stack
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp contains lowest in-use stack address
  - address of “top” element

x86-64 Stack: Push
- pushq Src
  - Fetch operand at Src
  - Decrement %rsp by 8
  - Write operand at address given by %rsp

x86-64 Stack: Pop
- popq Dest
  - Read value at address given by %rsp
  - Increment %rsp by 8
  - Store value at Dest (usually a register)
Today

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustration of Recursion

Code Examples

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

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

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

Control Flow Example #1

```
0000000000400548 <multstore>: 
            0x400548:  callq 400550 <mult2>  # mult2(a, y)
            0x400549:  mov %rax,(%rbx)  # Save at dest
            0x400550:  mov %rdi,%rax  # a
            0x400551:  imul %rsi,%rax  # a * b
            0x400552:  mov %rax,%rdi  # a
            0x400553:  addq %edi,%eax  # a + b
            0x400554:  mov %eax,%rax  # s
            0x400555:  mov %rax,%rdi  # s
            0x400556:  addq %edi,%eax  # a + b
            0x400557:  mov %eax,%rax  # s
            0x400558:  mov %rax,%rdi  # s
            0x400559:  addq %edi,%eax  # a + b
            0x40055a:  mov %eax,%rax  # s
            0x40055b:  mov %rax,%rdi  # s
            0x40055c:  addq %edi,%eax  # a + b
            0x40055d:  mov %eax,%rax  # s
            0x40055e:  mov %rax,%rdi  # s
            0x40055f:  addq %edi,%eax  # a + b
            0x400560:  mov %eax,%rax  # s
            0x400561:  mov %rax,%rdi  # s
            0x400562:  addq %edi,%eax  # a + b
            0x400563:  mov %eax,%rax  # s
            0x400564:  mov %rax,%rdi  # s
            0x400565:  addq %edi,%eax  # a + b
            0x400566:  mov %eax,%rax  # s
            0x400567:  mov %rax,%rdi  # s
            0x400568:  addq %edi,%eax  # a + b
            0x400569:  mov %eax,%rax  # s
            0x40056a:  mov %rax,%rdi  # s
            0x40056b:  addq %edi,%eax  # a + b
            0x40056c:  mov %eax,%rax  # s
            0x40056d:  mov %rax,%rdi  # s
            0x40056e:  addq %edi,%eax  # a + b
            0x40056f:  mov %eax,%rax  # s
            0x400570:  mov %rax,%rdi  # s
            0x400571:  addq %edi,%eax  # a + b
            0x400572:  mov %eax,%rax  # s
            0x400573:  mov %rax,%rdi  # s
            0x400574:  addq %edi,%eax  # a + b
            0x400575:  mov %eax,%rax  # s
            0x400576:  mov %rax,%rdi  # s
            0x400577:  addq %edi,%eax  # a + b
            0x400578:  mov %eax,%rax  # s
            0x400579:  mov %rax,%rdi  # s
            0x40057a:  addq %edi,%eax  # a + b
            0x40057b:  mov %eax,%rax  # s
            0x40057c:  mov %rax,%rdi  # s
            0x40057d:  addq %edi,%eax  # a + b
            0x40057e:  mov %eax,%rax  # s
            0x40057f:  mov %rax,%rdi  # s
            0x400580:  addq %edi,%eax  # a + b
            0x400581:  mov %eax,%rax  # s
            0x400582:  mov %rax,%rdi  # s
            0x400583:  addq %edi,%eax  # a + b
            0x400584:  mov %eax,%rax  # s
            0x400585:  mov %rax,%rdi  # s
            0x400586:  addq %edi,%eax  # a + b
            0x400587:  mov %eax,%rax  # s
            0x400588:  mov %rax,%rdi  # s
            0x400589:  addq %edi,%eax  # a + b
            0x40058a:  mov %eax,%rax  # s
            0x40058b:  mov %rax,%rdi 
```
Control Flow Example #4

0000000000400550 <mult2>:
400550:  mov %rdi,%rax
•
400557:  retq

0000000000400540 <multstore>:
•
400544: callq 400550 <mult2>
•
400549: mov    %rax,(%rbx)

Today

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustrations of Recursion & Pointers

Procedure Data Flow

Registers
- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9
- Return value
  - %rax

Stack
- Only allocate stack space when needed

Data Flow Examples

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

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

Overview: GDB without source code

- GDB can also be used just at the instruction level

<table>
<thead>
<tr>
<th>Source-level GDB</th>
<th>Binary-level GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>step/next</td>
<td>stepi/nexti</td>
</tr>
<tr>
<td>break &lt;line number&gt;</td>
<td>break *&lt;address&gt;</td>
</tr>
<tr>
<td>list</td>
<td>disas</td>
</tr>
<tr>
<td>print &lt;variable&gt;</td>
<td>print with registers &amp; casts</td>
</tr>
<tr>
<td>print &lt;data structure&gt;</td>
<td>examine</td>
</tr>
<tr>
<td>info local</td>
<td>info reg</td>
</tr>
<tr>
<td>software watch</td>
<td>hardware watch</td>
</tr>
</tbody>
</table>
Disassembly and stepping

- The **disas** command prints the disassembly of instructions
  - Give a function name, or defaults to current function, if available
  - Or, supply range of addresses <start>, <end> or <start>, ++<length>
  - If you like TUI mode, "layout asm"
  - Shortcut for a single instruction: x/i <addr>, x/i $rip
  - disasm/r shows raw bytes too

- **stepi** and **nexti** are like **step** and **next**, but for instructions
  - Can be abbreviated **si** and **ni**
  - **stepi** goes into called functions, **nexti** stays in current one
  - **continue**, **return**, and **finish** work as normal

Binary-level breakpoints

- All breakpoints are actually implemented at the instruction level
  - info br will show addresses of all breakpoints
  - Sometimes multiple instructions correspond to one source location
  - To break at an instruction, use break *<address>
    - Address usually starts with 0x for hex
  - The **until** command is like a temporary breakpoint and a continue
    - Works the same on either source or binary

Binary-level printing

- The **print** command still mostly uses C syntax, even when you don't have source
  - Registers available with $ names, like $rax, $rip
  - Often want p/x, for hex

- Use casts to indicate types
  - p (char)$r10
  - p (char *)$rbx

- Use casts and dereferences to access memory
  - p *(int *)$rcx
  - p *(int *)($rbx + 1)
  - p *(int *)($rbx + 4)

Examining memory

- The examine (x) command is a low-level tool for printing memory contents
  - No need to use cast notation
  - x/<format> <address>
    - Format can include repeat count (e.g., for array)
    - Many format letters, most common are x for hex or d for decimal
    - Size letter b/h/w/g means 1/2/4/8 bytes

Example: x/20xg 0x404100
  - Prints first 20 elements of an array of 64-bit pointers, in hex

More useful printing commands

- info reg prints contents of all integer registers, flags
  - In TUI: layout reg, will highlight updates
  - Float and vector registers separate, or use info all-reg
- info frame prints details about the current stack frame
  - For instance, "saved rip" means the return address
- backtrace still useful, but shows less information
  - Just return addresses, maybe function names

Hardware watchpoints

- To watch memory contents, use print-like syntax with addresses
  - watch *(int *)0x404170
- GDB's "Hardware watchpoint" indicates a different implementation
  - Much faster than software
  - But limited in number
  - Limited to watching memory locations only
- Watching memory is good for finding memory corruption
Today

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustration of Recursion

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

```
Example Call Chain

... yoo(...) {...
    ... who();
    ... ...
...
... amI();
    ...
    ...

Procedure amI() is recursive
```

Stack Frames

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

- Management
  - Space allocated when enter procedure
    - "Set-up" code, also called "prolog"
    - Includes push by call instruction
  - Deallocated when return
    - "Finish" code, also called "epilog"
    - Includes pop by ret instruction

Example

```
Stack

%rbp

%rsp

Stack "Top"
```

Example

```
Stack

%rbp

%rsp

Stack "Top"
```
Example

Stack

Example

Stack

Example

Stack

Example

Stack

Example

Stack

Example

Stack
Example: incr

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

Register Use(s)

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

Example: Calling incr #1

Initial Stack Structure

```
...                  %rax
Rtn address
```

Resulting Stack Structure

```
...                  %rax
15213    8(%rsp)
Rtn address
```

**x86-64/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 (optional)

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

**Register Use(s)**

- `%rdi`: Argument p
- `%rsi`: Argument val, y
- `%rax`: x, Return value

**Call Instruction**

```
call incr
```

**Local Variables**

- `%rdi`: Argument p
- `%rsi`: Argument val
- `%rax`: Return value

**Register Build (Optional)**

- `%rbp`: Frame pointer
- `%rsp`: Stack pointer

**Arguments**

- `%rdi`: Frame pointer
- `%rsi`: Saved registers
- `%rax`: Old frame pointer (optional)

**Return Address**

- `%rdi`: Frame pointer
- `%rsi`: Saved registers
- `%rax`: Old frame pointer (optional)
Example: Calling incr #2

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

Example: Calling incr #3

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

Example: Calling incr #4

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

Example: Calling incr #5

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

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 overwritten by who
  - This could be trouble = something should be done!
  - Need some coordination
x86-64 Linux Register Usage #1 (scratch)

- %rax
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- %rdi, ... , %r9
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- %r10, %r11
  - Caller-saved
  - Can be modified by procedure

Return value

Arguments

Callee-saved temporaries

x86-64 Linux Register Usage #2 (preserved)

- %rbx, %r12, %r13, %r14
  - Caller-saved
  - Caller must save & restore
- %rbp
  - Caller-saved
  - Caller must save & restore
  - May be used as frame pointer
  - Can mix & match
- %rsp
  - Special form of callee save
  - Restored to original value upon exit from procedure

Callee-saved Temporaries

Special

Callee

Temporaries

Pre-return Stack Structure

Resulting Stack Structure

Initial Stack Structure

Rtn address

%rsp+8

%rbx

Saved %rbx

Resulting Stack Structure

%rbx

Rtn address

%rbp

Saved %rbp

Rtn address

%rbx

Pre-return Stack Structure

Rtn address

%rbx

Initial Stack Structure

Rtn address

%rbx

Resulting Stack Structure

%rbx

Rtn address

%rbx

Today

- Procedures
  - Stack Structure
  - Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
  - Illustration of Recursion

Recursive Function

```c
defintion

long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```
Recursive Function Terminal Case

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

<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>
</tbody>
</table>

Recursive Function Call Setup

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>x &amp; 1</td>
<td>Rec. argument</td>
</tr>
<tr>
<td>rbx</td>
<td>x &amp; 1</td>
<td>Call. saved</td>
</tr>
</tbody>
</table>

Recursive Function Call

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

<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>Call. 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 popcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + popcount_r(x >> 1);
}
```

<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>Call. saved</td>
</tr>
<tr>
<td>rax</td>
<td></td>
<td>Return value</td>
</tr>
</tbody>
</table>

Recursive Function Completion

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

<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>
</tbody>
</table>

```c
movl  $0, %rax
movl  $1, %rbx
andl  %1, %rbx
andl  %0, %rbx
movq  %rbx, %rbx
shrq  %rdi, %rbx
call  popcount_r
addq  %rbx, %rax
popq  %rbx
```

```
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
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - 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

Discussion interlude

- **Does a recursive function always have to save one or more registers on the stack?**
  - If yes, why?
  - If no, what's an example of a function that doesn't need to?

Recursive function examples

- **void loop(void) { loop(); }**

- **int fact(unsigned n, int prod) {**
  - if (n == 0)
    - return prod;
  - else
    - return fact(n - 1, n * prod);
  **}**

- But if storing a value across a call, the stack is needed
  - If caller-save, need to save because callee will use it
  - If callee-save, need to save caller's value
  - Changing the calling convention would not help

x86-64 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 %rax
  - **Pointers are addresses of values**
    - On stack or global

- **Return Addr**
- **Saved Registers + Local Variables**
- **Argument Build**