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 (must be 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

```c
void multstore:
    retq
```

Control Flow Example #2

```c
void multstore:
    retq
```

Control Flow Example #3

```c
void multstore:
    retq
```
Control Flow Example #4

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

Today

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

Data Flow Examples

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

Announcements

- Exercise set 1 solutions set posted
  - Probably will not turn back until after midterm, so look over the solutions now
- Hands-on assignment 1 graded
  - Grades will be on Moodle [shortly]
  - Moodle forum has a general feedback post
- Midterm 1 preparations
  - Test will start promptly at 3:35pm on Monday, try to arrive at least a few minutes before
  - Two historical sample exams are on the course web site, solutions Friday
  - Recommended writing implement is a mechanical pencil and a good eraser
  - Friday in class will review material covered by the exam
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 *)$rbx`
- **Use casts and dereferences to access memory**
  - `p *(int *)$rbx`
  - `p *(char **)$r8`
  - `p *(int *)($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

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
  - Callie returns before caller does
- Stack allocated in Frames
  - State for single procedure instantiation

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

Today

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

Call Chain Example

Example Call Chain

Example

Procedure an1() is recursive
Example

```
who(…)
•
•
•
who();
•
•
•
```

Stack

```
who
%rbp
%rsp
```

Example

```
who(…)
•
•
•
who();
•
•
•
```

Stack

```
who
%rbp
%rsp
```

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’s Frame
  - Arguments
  - Return Address
  - Old %rbp

- Frame pointer %rbp (Optional)

Example: incr

```
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)
```

Register | Use(s)
---|---
%rdi | Argument p
%rsi | Argument val, y
%rax | Return value
Example: Calling incr #1

```
long call_incr() {
    long vl = 15213;
    long v2 = incr(vl, 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 krdx overwritten by who
  - This could be trouble => something should be done!
  - Need some coordination

Example: Calling incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr4(v1, 3000);
    return v1+v2;
}
```

Example: Calling incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr4(v1, 3000);
    return v1+v2;
}
```

Example: Calling incr #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr4(v1, 3000);
    return v1+v2;
}
```

Example: Calling incr #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr4(v1, 3000);
    return v1+v2;
}
```
Register Saving Conventions

- When procedure \( \text{yoo} \) calls \( \text{who} \):
  - \( \text{yoo} \) is the caller
  - \( \text{who} \) is the callee

- Can register be used for temporary storage?

  - Conventions
    - "Caller Saved", a.k.a. "scratch"
      - Caller saves temporary values in its frame before the call
    - "Callee Saved", a.k.a. "preserved"
      - Callee saves temporary values in its frame before using
      - Callee restores them before returning to caller

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

x86-64 Linux Register Usage #2 (preserved)

- %rbx, %r12, %r13, %r14
  - Callee-saved
  - Callee must save & restore
- %rbp
  - Callee-saved
  - Callee 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 Example #1

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

Callee-Saved Example #2

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

Today

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

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

Recursive Function Register Save

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

Recursive Function Call

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

Recursive Function Terminal Case

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

Recursive Function Result

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

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?
  - Talk with your neighbors, then put your answer on ChimeIn

https://chimein.cla.umn.edu/course/view/2021

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