The Stack & Procedures
CSE 351 Autumn 2022

Instructor: Justin Hsia

Teaching Assistants: Angela Xu
Assaf Vayner
David Dai
James Froelich
Paul Stevans

Arjun Narendra
Carrie Hu
Dominick Ta
Jenny Peng
Renee Ruan

Armin Magness
Clare Edmonds
Effie Zheng
Kristina Lansang
Vincent Xiao

http://xkcd.com/571/
Relevant Course Information

❖ Lab 2 due next Friday (10/28)
  ▪ Can start in earnest after today’s lecture!
  ▪ See GDB Tutorial and Phase 1 walkthrough in Section 4 Lesson

❖ Midterm (take home, 11/3–11/5)
  ▪ Make notes and use the midterm reference sheet
  ▪ Form study groups and look at past exams!
x86 Control Flow

❖ Condition codes
❖ Conditional and unconditional branches
❖ Loops
❖ Switches
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - Jump table
  - Indirect jump instruction

```c
long switch_ex (long x, long y, long z) {
    long w = 1;
    switch (x) {
        case 1:
            w = y*z; break;
        case 2:
            w = y/z;
        /* Fall Through */
        case 3:
            w += z; break;
        case 5:
        case 6:
            w -= z; break;
        case 7:
            w = y%z; break;
        default:
            w = 2;
    }
    return w;
}
```
## Jump Table Structure

### Switch Form

```java
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
        ...
    case val_n-1:
        Block n-1
}
```

### Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Targ0:</td>
</tr>
<tr>
<td></td>
<td>Targ1</td>
</tr>
<tr>
<td></td>
<td>Targ2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Targn-1</td>
<td>Code Block 0</td>
</tr>
<tr>
<td></td>
<td>Code Block 1</td>
</tr>
<tr>
<td></td>
<td>Code Block 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Targn-1:</td>
</tr>
<tr>
<td></td>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>

### Approximate Translation

```java
target = JTab[x];
goto target;
```
Jump Table Structure

C code:

```c
switch (x) {
    case 1: <code> break;
    case 2: <code>
    case 3: <code> break;
    case 5:
    case 6: <code> break;
    case 7: <code> break;
    default: <code>
}
```

Use the jump table when x ≤ 7:

```c
if (x <= 7)
    target = JTab[x];
goto target;
else
    goto default;
```
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

Note compiler chose to not initialize w

`switch_ex:`

- `movq %rdx, %rcx`
- `cmpq $7, %rdi`  # x:7
- `ja .L9`        # default
- `jmp *.L4(%rdi,8)`  # jump table

Jump above – unsigned > catches negative default cases

Take a look!

https://godbolt.org/z/r8qY7Ec1T
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

**Jump table**

```
.section .rodata
.align 8
.L4:
    .quad .L9    # x = 0
    .quad .L8    # x = 1
    .quad .L7    # x = 2
    .quad .L10   # x = 3
    .quad .L9    # x = 4
    .quad .L5    # x = 5
    .quad .L5    # x = 6
    .quad .L3    # x = 7
```

**switch_ex:**
- `movq %rdx, %rcx`
- `cmpq $7, %rdi`  # x:7
- `ja .L9`     # default
- `jmp *.L4(%rdi,8)`  # jump table

**Indirect jump**
Assembly Setup Explanation

❖ Table Structure
  ▪ Each target requires 8 bytes (address)
  ▪ Base address at .L4

❖ Direct jump: jmp .L9
  ▪ Jump target is denoted by label .L9

❖ Indirect jump: jmp * .L4(%rdi,8)
  ▪ Start of jump table: .L4
  ▪ Must scale by factor of 8 (addresses are 8 bytes)
  ▪ Fetch target from effective address .L4 + x*8
    • Only for 0 ≤ x ≤ 7
The Hardware/Software Interface

- **Topic Group 2: Programs**
  - x86-64 Assembly, **Procedures, Stacks, Executables**

- How are programs created and executed on a CPU?
  - How does your source code become something that your computer understands?
  - How does the CPU organize and manipulate local data?
Reading Review

❖ Terminology:

- Stack, Heap, Static Data, Literals, Code
- Stack pointer (%rsp), push, pop
- Caller, callee, return address, call, ret
  - Return value: %rax
  - Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Stack frames and stack discipline

❖ Questions from the Reading?
Review Questions

❖ How does the stack change after executing the following instructions?
  pushq %rbp
  subq $0x18, %rsp

❖ For the following function, which registers do we know must be used?
  void* memset(void* ptr, int value, size_t num);
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

❖ All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Simplified Memory Layout (Review)

Address Space:

- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

What Goes Here:

- local variables and procedure context
- variables allocated with `new` or `malloc`
- `static` variables (including global variables)
- large literals/constants (e.g., "example")
- program code

High Addresses

Memory Addresses

Low Addresses

0xF...F

0x0...0
Memory Management

Address Space:

- **Stack**: Managed “automatically” (by compiler/assembly)
- **Dynamic Data (Heap)**: Managed “dynamically” (by programmer)
- **Static Data**: Managed “statically” (initialized when process starts)
- **Literals**: Managed “statically” (initialized when process starts)
- **Instructions**: Managed “statically” (initialized when process starts)

Who’s Responsible:

- **Address Space**:
  - High Addresses
  - Low Addresses

Memory Addresses

Instructions

Literals

Static Data

Dynamic Data (Heap)

Stack
Memory Permissions

- Segmentation fault: impermissible memory access

Address Space:

- Stack: writable; not executable
- Dynamic Data (Heap): writable; not executable
- Static Data: writable; not executable
- Literals: read-only; not executable
- Instructions: read-only; executable

Permissions:

- High Addresses: 0xF...F
- Low Addresses: 0x0...0
x86-64 Stack (Review)

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register %rsp contains lowest stack address
  - %rsp = address of top element, the most-recently-pushed item that is not-yet-popped

Stack Pointer: %rsp

Stack “Bottom”

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses

0x00...00
x86-64 Stack: Push (Review)

- **pushq src**
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - **Decrement** %rsp by 8
  - Store value at address given by %rsp

- **Example:**
  - **pushq %rcx**
    - Adjust %rsp and store contents of %rcx on the stack

**Stack Pointer:** %rsp → -8

**Stack “Top”**

**Stack “Bottom”**
x86-64 Stack: Pop (Review)

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst`
  - **Increment** `%rsp` by 8

- **Example:**
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g., no arguments)
Procedure Call Overview

❖ The *convention* of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn’t follow these conventions?
Code Example (Preview)

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

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

Compiler Explorer: [https://godbolt.org/z/Yb3GeoaMa](https://godbolt.org/z/Yb3GeoaMa)

```
0000000000400540 <multstore>:
  400540: push %rbx # Save %rbx
  400541: movq %rdx,%rbx # Save dest
  400544: call 400550 <mult2> # mult2(x,y)
  400549: movq %rax,(%rbx) # Save at dest
  40054c: pop %rbx # Restore %rbx
  40054d: ret # Return
```

```
0000000000400550 <mult2>:
  400550: movq %rdi,%rax # a
  400553: imulq %rsi,%rax # a * b
  400557: ret # Return
```
Procedure Control Flow (Review)

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  1) Push return address on stack (*why? which address?*)
  2) Jump to `label`
Procedure Control Flow (Review)

- Use stack to support procedure call and return

- **Procedure call**: `call label`
  1) Push return address on stack (why? which address?)
  2) Jump to `label`

- Return address:
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:

```
400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
```

  Return address = 0x400549

- **Procedure return**: `ret`
  1) Pop return address from stack
  2) Jump to address

  next instruction happens to be a move, but could be anything
Procedure Call Example (step 1)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  
0000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: ret
Procedure Call Example (step 2)

00000000000400540 <multstore>:
- 
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
- 

00000000000400550 <mult2>:
  400550: movq %rdi, %rax
- 
  400557: ret

%rsp 0x118
%rip 0x400550

0x400549
Procedure Return Example (step 1)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  
  400550 <mult2>:
  400550: movq %rdi,%rax
  
  400557: ret

%rip 0x400557
%rsp 0x118
**Procedure Return Example (step 2)**

```
00000000000400540 <multstore>:
  .
  .
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  .
  .

00000000000400550 <mult2>:
  400550: movq %rdi,%rax
  .
  .
  400557: ret
```

- `%rsp` 0x120
- `%rip` 0x400549
- 0x130
- 0x128
- 0x120
Procedures

❖ Stack Structure

❖ **Calling Conventions**
  ▪ Passing control
  ▪ **Passing data**
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Procedure Data Flow (Review)

Registers (NOT in Memory)
- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack (Memory)
- %rax
- %rsi
- %rdx
- %rcx
- %r8
- %r9

Diane’s Silk Dress Costs $89

• Only allocate stack space when needed
x86-64 Return Values

- By convention, values returned by procedures are placed in %rax
  - Choice of %rax is arbitrary

1) **Caller** must make sure to save the contents of %rax before calling a **callee** that returns a value
  - Part of register-saving convention

2) **Callee** places return value into %rax
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them

3) Upon return, **caller** finds the return value in %rax
Data Flow Examples

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

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

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
400541:    movq    %rdx,%rbx    # Save dest
400544:    call    400550 <mult2>    # mult2(x,y)
    # t in %rax
400549:    movq    %rax,(%rbx)    # Save at dest
    ...
```

```
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550:    movq    %rdi,%rax    # a
400553:    imulq    %rsi,%rax    # a * b
    # s in %rax
400557:    ret    # Return
```
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, most modern languages
  - Code must be re-entrant
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments, local variables, return address

- Stack allocated in frames
  - State for a single procedure instantiation

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
Call Chain Example

Procedure `amI` is recursive (calls itself)
1) Call to whoa

```c
whoa(...) {
    •
    •
    who();
    •
    •
}
```

Stack

```c
whoa
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
```
2) Call to who

whoa(…)
{
  who(…)
  {
    •
    amI();
    •
    amI();
  }
}

Stack

whoa

who

amI amI

%rbp

%rsp

amI

amI
3) Call to amI (1)

```
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      •
      if(){
        amI()
      }
    }
    •
  }
}
```

Stack

- whoa
- who
- amI
- amI
- %rbp
- %rsp
- amI₁
4) Recursive call to `amI` (2)

```
whoa(…)
{
    who(…)
    {
        amI(…)
        {
            if()
            {
                amI()
            }
        }
        .
    }
    .
}
```

Stack:
- `whoa`
- `who`
- `amI_1`
- `amI_2`
5) (another) Recursive call to amI (3)
6) Return from (another) recursive call to amI
7) Return from recursive call to `amI`

![Diagram of recursive function calls and stack](image)
8) Return from call to `amI`
9) (second) **Call to amI (4)**

The diagram illustrates the call stack for the function `amo(...)`, which calls `who(...)`, which in turn calls `amI(...)`. The stack frame for `amI(...)` includes a call to `amI()` within an `if()` condition. The stack also shows the return addresses for each function call, with the `%rbp` and `%rsp` registers indicating the stack pointers. The diagram also highlights the `amI_1`, `amI_2`, and `amI_3` frames on the stack.
10) **Return from (second) call to amI**
11) Return from call to who

```c
whoa(...) {
    •
    •
    who();
    •
}
```

Stack diagram:
- `%rbp`
- `%rsp`
- `whoa`
- `who`
- `amI`
- `amI`
- `amI4`
- `amI2`
- `amI3`