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
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
}
```

**Approximate Translation**

```java
target = JTab[x];
goto target;
```

**Jump Table**

- JTab:
  - Targ0
  - Targ1
  - Targ2
  - • • •
  - Targn-1

**Jump Targets**

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- • • •
- Targn-1: Code Block n-1

The addresses (8 bytes wide) are like an array of pointers.
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 $x \leq 7$:

```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;
}
```

Register | Use(s)
---|---
%rdi | 1st argument (x)
%rsi | 2nd argument (y)
%rdx | 3rd argument (z)
%rax | return value

Note compiler chose to not initialize \( w \)

Jump above – unsigned > catches negative default cases

\(-1 > 7U \rightarrow \text{jump to default case} \)

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
```
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?

\[
\begin{align*}
\text{pushq } &\%\text{rbp} \\
\text{subq } &\$0\times18, \%\text{rsp}
\end{align*}
\]

❖ For the following function, which registers do we know must be used?

```c
void* memset(void* ptr, int value, size_t num);
```

- Return value in \%rax
- Arguments in \%rdi, \%rsi, and \%rdx
- \%rsp changed by call & ret
- \%rip changed while executing instructions
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**: local variables and procedure context
- **Dynamic Data (Heap)**: variables allocated with new or malloc
- **Static Data**: static variables (including global variables)
- **Literals**: large literals/constants (e.g., "example")
- **Instructions**: program code

Memory Addresses:

- **High Addresses**: 0xF...F
- **Low Addresses**: 0x0...0

What Goes Here:
Memory Management

Who’s Responsible:
- **Address Space:**
  - **Instructions**:
    - Managed “statically”
    - (initialized when process starts)
  - **Literals**:
    - Managed “statically”
    - (initialized when process starts)
  - **Static Data**:
    - Managed “statically”
    - (initialized when process starts)
  - **Dynamic Data (Heap)**:
    - Managed “dynamically”
    - (by programmer)
  - **Stack**:
    - Managed “automatically”
    - (by compiler/assembly)
  - **Low Addresses**
    - **High Addresses**
      - Address Space: $0xF...F$
        - Stack
        - Dynamic Data (Heap)
        - Static Data
        - Literals
        - Instructions

- Memory Addresses
  - **Low Addresses**
    - $0x0...0$
  - **High Addresses**
    - $0xF...F$
Memory Permissions

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:
- Segmentation fault: impermissible memory access
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`  
**Stack Grows Down**

0x00...00
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;
}

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

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

**Disassembly**

```
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` *(special push)*
  1) Push return address on stack *(why? which address?)*
  2) Jump to `label`

> 0. Move %rsp down
> 1. Store ret addr at %rsp
> 2. Label \(\rightarrow\) %rip
Procedure Control Flow (Review)

- Use stack to support procedure call and return

**Procedure call:** `call label` *(special push)*

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` *(special pop)*

1. Pop return address from stack *(1) read ret addr at %rsp into %rip*
2. Jump to address *(2) move %rsp up*

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)

0000000000400540 <multstore>:

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

0000000000400550 <mult2>:

400550: movq %rdi, %rax
400557: ret
Procedure Return Example (step 1)

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

00000000000400550 <mult2>:
  400550: movq %rdi,%rax
  
  400557: ret
Procedure Return Example (step 2)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  
0000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: ret
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
  1. %rdi
  2. %rsi
  3. %rdx
  4. %rcx
  5. %r8
  6. %r9

- Return value
  %rax

Stack (Memory)

- Only allocate stack space when needed
- First 6 arguments are directly in registers:
  - Diane’s Silk Dress Costs $89

- Return value is in register %rax

- Stack grows downward, accessed by $8(%rsp)
  - Ret addr $0x00...00

- High Addresses
- Low Addresses
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

**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;
}
```

---

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
0000000000400541: movq %rdx,%rbx     # “Save” dest
    ...
0000000000400544: call 400550 <mult2> # mult2(x,y)
    # t in %rax
0000000000400549: movq %rax,(%rbx)    # Save at dest
    ...
```

---

```
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
0000000000400550: movq %rdi,%rax     # a
    ...
0000000000400553: imulq %rsi,%rax    # a * b
    # s in %rax
0000000000400557: 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

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

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

amI(...) {
  •
  if(...) {
    amI()
  }
  •
}

Procedure amI is recursive
(calls itself)

Example
Call Chain

whoa
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
  ↓
amI

1st call recurses twice
2nd call doesn’t recurse
based on condition
1) Call to whoa

```
whoa (...) {
  •
  •
  who();
  •
}
```
2) Call to who

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

2) Call to who

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

Stack

whoa

who

%rbp

“create” frame by manipulating %rsp

%rsp
3) Call to `amI (1)`

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

```
Stack

whoa

who

amI

amI

%rbp

%rsp

amI_1
```
4) Recursive call to `amI` (2)
5) (another) Recursive call to amI (3)

Stack

whoa

who

amI

amI

amI

amI

%rbp

%rsp

Stack
6) **Return from (another) recursive call to amI**

![Diagram showing recursive calls and stack frames]

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

- `amI(...)`
  - `if(){ amI() }

---

**Stack**

- `whoa`
- `who`
- `amI_1`
- `amI_2`
- `amI_3`

**Notes:**
- "deallocrate" stack frame by moving %rsp back up
- Data still exists, but you shouldn't use it.
7) Return from recursive call to amI

```
whoa(…)
{
    who(…)
    {
        amI(…)
        {
            •
            if()
            {
                amI()
            }
        }
    }
    •
}
```
8) Return from call to `amI`
9) *(second)* Call to `ami` (4)
10) Return from \textit{(second) call to amI}
11) Return from call to `who`

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

Call chain: main -> `whoa` -> `who` -> `whoa`

Stack:
- `main`
- `whoa`
- `who`
- `amI_4`
- `amI_2`
- `amI_3`

Total stack frames created: 7

Maximum stack depth: 6 frames