The Stack & Procedures
CSE 351 Autumn 2019

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu
Diya Joy
Maurice Montag
Suraj Jagadeesh
Antonio Castelli
Ivy Yu
Melissa Birchfield
Cosmo Wang
Kaelin Laundry
Millicent Li

http://xkcd.com/571/
Administraivia

- Lab 2 due next Friday (10/25)
  - Ideally want to finish well before the midterm

- Midterm (10/28, 5:30-6:40 pm, KNE 130)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 75%
Switch Statement
Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 0, 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
}
```

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
```

Approximate Translation

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

C code:

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

Use the jump table when \( x \leq 6 \):

```c
if (x <= 6)
    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

### Take a look!
https://godbolt.org/z/aY24el

**Jump above** – unsigned > catches negative default cases
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 .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6
```

Switch statement example:

```c
switch_eg:
    movq %rdx, %rcx
    cmpq $6, %rdi    # x:6
    ja .L8          # default
    jmp *.L4(,%rdi,8) # jump table
```
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes (address)
  - Base address at \( .L4 \)

- **Direct jump**: \( \text{jmp} \ .L8 \)
  - Jump target is denoted by label \( .L8 \)

- **Indirect jump**: \( \text{jmp} \ *.L4(,%rdi,8) \)
  - 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 \( 0 \leq x \leq 6 \)

Jump table

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

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:
```
011101000011000
100011010000010000000010
100010011100010
11000001111110100001111
```

Computer system:

Memory & data
Integers & floats
x86 assembly

Executables
Arrays & structs

Processes
Virtual memory

Memory allocation
Java vs. C

OS:
```
Windows 10
OS X Yosemite
```

CSE351, Autumn 2019
Mechanisms required for procedures

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

2) Passing data
   a) Procedure arguments
   b) Return value

3) Memory management
   a) Allocate during procedure execution
   b) Deallocate upon return

  ❖ All implemented with machine instructions!
   a) 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

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
Memory Management

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

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

Memory Addresses

High Addresses 0xF...F

Low Addresses 0x0...0
Memory Permissions

- **Segmentation faults?**

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:

- writable; not executable

Memory Addresses

- High Addresses
  - $0xF \ldots F$

- Low Addresses
  - $0x0 \ldots 0$
x86-64 Stack

- 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`
x86-64 Stack: Push

- `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`
x86-64 Stack: Pop

- `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`

Stack **“Top”**

Stack **“Bottom”**

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;
}
```

Compiler Explorer: [https://godbolt.org/z/nQ6KbZ](https://godbolt.org/z/nQ6KbZ)

```asm
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
```

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

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

- 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

- 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:
    ```plaintext
    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
Procedure Return Example (step 1)

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

0000000000400550 <mult2>:
  400550: movq %rdi,%rax
  •
  •
  400557: ret
Procedure **Return Example** (step 2)

```
0000000000400540 <multstore>:
  •
  •
0x120 0x128 0x130
0x120

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

0000000000400550 <mult2>:
  •
  •
0x400550: movq  %rdi,%rax
  •
  •
400557: ret
```

Diagram:
- Stack frame with `%rsp` at 0x120 and `%rip` at 0x400549.
- Call instruction at 400544 to `mult2`.
- Move instruction at 400549 to store the result in the stack.
Procedures

- Stack Structure

- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data

- Register Saving Conventions

- Illustration of Recursion
Procedure Data Flow

Registers (NOT in Memory)

- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack (Memory)

- 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

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;
}
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

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

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

Stack:
- `whoa`
- `who`
3) Call to `amI` (1)
4) Recursive call to `amI` (2)
5) *(another)* Recursive call to **amI (3)**

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

```
Stack

whoa
who
amI
amI
amI
%rbp
%rsp

amI_1
amI_2
amI_3
```
6) **Return from (another) recursive call to amI**

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

Diagram:

- `whoa` function
- `who` function
- `amI` function
- Stack
  - `whoa`
  - `who`
  - `amI1`
  - `amI2`
  - `amI3`
  - `%rbp`
  - `%rsp`
7) Return from recursive call to `amI`
8) Return from call to amI

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

Stack

![Stack diagram](image)
9) (second) Call to `amI` (4)
10) Return from (second) call to amI
11) Return from call to who

```c
whoa(...) {
  •
  •
  who();
  •
}
```
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context
    - (when reusing registers)
  - Local variables
    - (If can’t be kept in registers)
  - “Argument build” area
    - (If callee needs to call another function - parameters for function about to call, if needed)
Polling Question

Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```c
int main() {
    int i, x = 0;
    for (i=0; i<3; i++)
        x = randSum(x);
    printf("x = %d\n", x);
    return 0;
}
```

```c
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```

- **Higher/larger address**: `x` or `y`?
- How many total stack frames are created?
- What is the maximum depth (# of frames) of the Stack?  
  
  A. 1  B. 2  C. 3  D. 4
Slides that expand on the simple switch code in assembly. These slides expand on material covered today and the previous lecture, so while you don’t need to read these, the information is “fair game.”
Jump Table

declaring data, not instructions

8-byte memory alignment

Jump table

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

data is 64-bits wide

```
switch(x) {
    case 1:      // .L3
        w = y*z;
        break;
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    case 5:
    case 6:      // .L7
        w -= z;
        break;
    default:     // .L8
        w = 2;
}
```
Code Blocks \((x == 1)\)

```
switch(x) {
    case 1:    // .L3
        w = y*z;
        break;
    . . .
}
```

Register Use(s)

<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>

```
.L3:
    movq %rsi, %rax  # y
    imulq %rdx, %rax  # y*z
    ret
```
Handling Fall-Through

```c
long w = 1;
... 
switch (x) {
  ... 
  case 2:    // .L5
    w = y/z;
    /* Fall Through */
  case 3:    // .L9
    w += z;
    break;
  ... 
}
```

```c
  case 2:
    w = y/z;
    goto merge;
```

```c
  case 3:
    w = 1;
merge:
    w += z;
```

More complicated choice than “just fall-through” forced by “migration” of `w = 1`;

- Example compilation trade-off
Code Blocks \((x == 2, x == 3)\)

```c
long w = 1;

...switch (x) {
  ...case 2: // .L5
    w = y/z;
    /* Fall Through */
    case 3: // .L9
    w += z;
    break;
  ...
}
```

<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>

.L5: # Case 2:
    movq %rsi, %rax # y in rax
    cqto
    idivq %rcx # y/z
    jmp .L6 # goto merge

.L9: # Case 3:
    movl $1, %eax # w = 1

.L6: # merge:
    addq %rcx, %rax # w += z
    ret
Code Blocks (rest)

```c
switch (x) {
    ...
    case 5: // .L7
        w -= z;
        break;
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

<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>

.L7:  # Case 5,6:
    movl $1, %eax  # w = 1
    subq %rdx, %rax # w -= z
    ret

.L8:  # Default:
    movl $2, %eax  # 2
    ret