The Stack & Procedures
CSE 351 Winter 2020

Instructor: Ruth Anderson
Teaching Assistants: Jonathan Chen 
Josie Lee
Justin Johnson
Eddy (Tianyi) Zhou

Porter Jones
Jeffery Tian
Callum Walker

http://xkcd.com/571/
Administrivia

- Lab 2 due next Friday (2/07)
  - Ideally want to finish well before the midterm
  - Optional GDB Tutorial homework on Gradescope

- Midterm: 2/10, during lecture
  - You will be provided a fresh reference sheet
  - Find a study group! Look at past exams!
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:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100001111
```

OS:

```
Windows 10
OS X Yosemite
```
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

   \[ \text{All implemented with machine instructions!} \]
   - An x86-64 procedure uses only those mechanisms required for that procedure

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}

P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Procedures

- **Stack Structure**

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

- **Register Saving Conventions**

- **Illustration of Recursion**
Simplified Memory Layout

- **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:**
  - **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:

- **High Addresses**: Memory Addresses
- **Low Addresses**: Memory Addresses

Address Space: 0xF...F

Instructions

Literals

Static Data

Dynamic Data (Heap)

Stack
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:

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

Memory Addresses

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

Segmentation faults?
x86-64 Stack

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

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

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

**Stack “Bottom”**

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses

0x00...00
x86-64 Stack: Pop

- \texttt{popq \, dst}
  - Load value at address given by \%rsp
  - Store value at \textit{dst}
  - \textit{Increment} \%rsp by 8

- \textbf{Example:}
  - \texttt{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)
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?
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;
}
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:
    ```assembly
    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)

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

00000000000400550 <mult2>:
  400550: movq  %rdi,%rax
  •
  •
  400557: ret
```
Procedure **Call Example** (step 2)

0000000000400540 <multstore>:

- 
- 

400544: **call** 400550 <mult2>

400549: **movq** %rax, (%rbx)

- 
- 

0x400549

- 

0x118

0x120

0x128

0x130

0x400549

- 

%rip

0x400550

%rsp

0x118

0000000000400550 <mult2>:

400550: **movq** %rdi,%rax

- 

400557: **ret**
Procedure Return Example (step 1)

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

0000000000400550 <mult2>:
  •
  •
 400550: movq %rdi,%rax
  •
  •
 400557: ret

0x400550

0x400549

%rip 0x400557

%rsp 0x118

0x130
0x128
0x120
0x118
0x400549
Procedure **Return** Example (step 2)

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

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

%rip 0x400549
%rsp 0x120
0x120
0x128
0x130
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

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

```assembly
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
    ...
```

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

```assembly
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

whoa(...)
{
  •
  •
  who();
  •
}

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

Example Call Chain

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

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

Stack Diagram:
- `whoa`
- `%rbp`
- `%rsp`
- `who`
- `ami`
2) Call to who

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

Stack

```
whoa

who

ami ami

ami

ami

%rbp

%rsp

who

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

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

**Stack**

- whoa
- who
- amI
- amI
- amI
- amI
- amI
- amI
- amI

```
%rbp
%rsp
```
6) Return from (another) recursive call to `amI`
7) Return from recursive call to `amI`

```
whoa(...) {
    who(...) {
        amI(...) {
            •
            if() {
                amI()
            }
        }
        •
    }
    •
}
```
8) Return from call to amI

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

Stack

whoa

who

amI amI

%rbp

%rsp

amI

amI1

amI2

amI3
9) (second) Call to \texttt{amI} (4)

whoa(\ldots)
{
  \texttt{who(\ldots)}
  {
    \texttt{amI(\ldots)}
    {
      \texttt{if(\ldots)\texttt{amI(\ldots)}}
    }
  }
}

\texttt{amI(\ldots)}
{
  \texttt{if(\ldots)\texttt{amI(\ldots)}}
}

Stack

- whoa
- who
- amI
- amI

\texttt{amI}_1
\texttt{amI}_2
\texttt{amI}_3
\texttt{amI}_4

\%rbp
\%rsp
10) Return from (second) call to `amI`
11) Return from call to who

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

Stack:
- whoa
- who
- amI
- amI
- %rbp
- %rsp
- amI
- amI
- amI
- amI
- amI
- amI
- amI

Diagram:
- whoa
- who
- amI
- amI
- %rbp
- %rsp
- amI
- amI
- amI
- amI
- amI
- amI
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;
}
```

- **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
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)