The Stack & Procedures
CSE 351 Winter 2020

Instructor: Ruth Anderson
Teaching Assistants: Jonathan Chen
                    Josie Lee
                    Eddy (Tianyi) Zhou
                Justin Johnson
                Jeffery Tian
                Porter Jones
                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
110000011111110100001111
```

**OS:**
- Windows 10
- OS X Yosemite

**Memory & data**
- Integers & floats
- x86 assembly

**Procedures & stacks**
- Executables
- Arrays & structs
- Processes
- Virtual memory
- Memory allocation

**Java vs. C**
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

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

What Goes Here:

Memory Addresses

High Addresses

Low Addresses

0xF...F

0x0...0
Memory Management

Address Space:

- **Stack**
  - Managed "automatically" (by compiler/assembly)
  - grow towards each other to maximize use of space

- **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:

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

Address Space:

- **Address Space**: 0x0...0
- **Address Space**: 0xF...F
Memory Permissions

- High Addresses
- Memory Addresses
- Low Addresses

Address Space:
- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

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

Segmentation faults?

accessing memory in a way that you are not allowed to
**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 - 8

1. move %rsp down (subtract)
2. store src at %rsp

Memory

Stack “Bottom”

Stack “Top”

Increasing Addresses

High Addresses

Low Addresses

0x00...00
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 $%rsp$

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/nQ6KbZ](https://godbolt.org/z/nQ6KbZ)

```
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

- 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`
Procedure Control Flow

- 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
2) Jump to address

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

```assembly
00000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  
00000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: ret
```

Memory layout:
- `%rip` 0x400544
- `%rsp` 0x120
- Stack:
  - 0x120
  - 0x128
  - 0x130
- CPU registers:
  - `%rax`
  - `%rdi`
  - `%rbx`
Procedure Call Example (step 2)

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

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

Procedure Call Example (step 2)
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>:
  
  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**

**Registers (NOT in Memory)**
- First 6 arguments
  - %rdi: *Diane’s Silk Dress* Costs $89
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9
- Return value
  - %rax

**Stack (Memory)**
- Only allocate stack space when needed
- High Addresses
- Low Addresses 0x00...00
- Stack grows downward
- Accessed by 8(%rsp)
- Pushed first
- Pushed last

[Diagram showing stack and registers]
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
```

lined up nicely so we didn't have to manipulate arguments
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

Example Call Chain

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

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

Stack diagram:
- `main` procedure
- `whoa` procedure
- `who()` function
- `%rbp` pointer
- `%rsp` pointer
- Arrows indicate function calls and return stack frames.
2) Call to who

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

Stack

%rbp → whoa
%rsp → who

“create” frame by manipulating %rsp
3) Call to amI (1)
4) Recursive call to \texttt{amI (2)}
5) (another) Recursive call to \texttt{amI} (3)

Stack

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

%rbp

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

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

Stack

```
whoa
who
amI
amI
amI
amI_1
amI_2
amI_3
```

“deallocate stack frame by moving %rsp back up

Data still exists, but you shouldn't use it
7) Return from recursive call to amI

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

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

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

**Stack**

- `whoa`
- `who`
- `amI`
- `amI`
- `%rbp`
- `%rsp`
9) (second) Call to amI (4)
10) Return from (second) call to `amI`
11) Return from call to who

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

Total stack frames created: 7

Maximum stack depth: 6 frames
Polling Question

- Answer the following questions about when \texttt{main()} is run (assume \texttt{x} and \texttt{y} stored on the Stack):

  - Higher/larger address: \texttt{x} or \texttt{y}?  
  - How many total stack frames are created?  
  - What is the maximum depth (# of frames) of the Stack?

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

int randSum(int n) {
    int y = rand() % 20;
    return n + y;
}
\end{verbatim}

Vote only on 3\textsuperscript{rd} question at [http://pollev.com/rea](http://pollev.com/rea)
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)