The Stack & Procedures
CSE 351 Summer 2020

Instructor: Porter Jones
Teaching Assistants: Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/571/
Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-15](https://tinyurl.com/CSE351-7-15)
- Unit Summary 1 due Friday (7/17) – 11:59pm
  - Can still use late days until 7/20
- Mid-quarter Survey due Friday (7/17) – 11:59pm
  - Submit via Canvas!
- hw8, hw9, hw10 now due Monday (7/20) – 10:30am
  - hw11 also due Monday (7/20)
  - See course schedule for original/suggested deadlines
- Lab 2 due Wednesday (7/22)
  - GDB Tutorial on Gradescope walks through first phase
Administrivia

- No midterm or final! But the midterm and final cheat sheets could be useful throughout the course
  - Can find them at the exams page of the course website
  - [https://courses.cs.washington.edu/courses/cse351/20su/exams/](https://courses.cs.washington.edu/courses/cse351/20su/exams/)

- These *do not* mimic a good unit summary
  - Too much like a cheat sheet (lots of listed facts w/out much organization and summary)
  - See unit summary spec for more details and good/bad examples of unit summaries
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:
```
call get_mpg
push %rbp
movq %rsp, %rbp

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

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory allocation
Java vs. C

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

✓ 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

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

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

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 faults? Accessing memory in a way that is not allowed.
**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:** $%rsp$

- Stack Pointer: $%rsp$
- Stack “Top”
- Stack Grows Down
- Low Addresses
  - $0x00...00$
- High Addresses
  - Increasing Addresses
- Stack “Bottom”
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 Pointer:** `%rsp`

Before:
- `%rsp`: 0x8010C0
- `%rcx`: 0x4a
- `%edx`: 0x7b

After:
- `%rsp`: 0x8010c8
- `%rcx`: 0x7b

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)

```
CALL
<create local vars>
...  <set up return val>
<destroy local vars>
ret
```
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/nQ6KbZ
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:
    
    ```
    400544: call 400550 <mult2>
    400549: movq %rax,(%rbx)
    ```
    Return address = 0x400549
- **Procedure return:** `ret`  
  1) Pop return address from stack  
  2) Jump to address

  ```
  jump to address = move address to %rip
  ```

  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)

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

0000000000400550 <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

Registers (NOT in Memory)

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

- Return value
  - rax

Stack (Memory)

- Only allocate stack space when needed

Diane’s Silk Dress Costs $89
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
```

The code snippet demonstrates the use of the `multstore` function, which multiplies two long integers and stores the result in a specified memory location. The `mult2` function computes the product of two long integers and returns the result. The assembly code shows the details of how the multiplication is performed and how the result is stored in memory.
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 whoa

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

```
Stack

main

whoa

```

```
whoa

```

```
who

```

```
who

```

```
amI

```

```
amI

```

```
amI

```

```
%rbp

```

```
%rsp

```

```
2) Call to who

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

Stack

```
whoa

who

amI

amI

%rbp

%rsp

create frame by moving rsp down
```
3) Call to amI (1)

whoa(…)
{
    who(…)
    {
        amI(…)
        {
            •
            if(){
                amI()
            }
            •
        }
        •
    }
}
4) Recursive call to amI (2)
5) (another) Recursive call to \texttt{amI} (3)
6) Return from (another) recursive call to \texttt{amI}
7) Return from recursive call to `amI`

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

Stack

```
whoa
who
amI
amI
amI_1
amI_2
amI_3
%rbp
%rsp
```
8) Return from call to `amI`

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

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

```
8)	Return	from	call
to	amI
```

**Stack**

```
whoa

who

amI

amI

%rbp

%rsp

amI

amI

amI

amI

amI

amI

amI

amI

amI
```

new stack frame will overwrite old data
9) (second) Call to `amI (4)`
10) Return from (second) call to \texttt{amI}

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

Stack

```
whoa
who
amI  amI
%rbp
%rsp
amI
amI
amI_4
amI_2
amI_3
```
11) Return from call to `who`

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

Stack

Total stack frames created: 7 frames
Max stack depth: 6 frames
Polling Question [Proc I – a]  

Vote only on 3rd question at http://pollev.com/pbjones

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

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

- `int randSum(int n) {`  
  ```c
  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
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)