The Stack & Procedures
CSE 351 Summer 2021

Instructor:
Justin Hsia
Mara Kirdani Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk

http://xkcd.com/571/
Gentle, Loving Reminders

- Mid-quarter Survey due Friday (7/16) – 8 pm
  - Submit via Canvas!

- hw9 due tonight, hw10 due Friday

- Lab 2 due Wednesday (7/21)
  - GDB Tutorial on website walks through first phase
  - Mara’s favorite lab!
As a note

- No midterm or final! But the midterm and final cheat sheets could be useful for the course
  - Can find them at the exams page of the course website
  - https://courses.cs.washington.edu/courses/cse351/21su/exams/
Learning Objectives

- Understanding this lecture means you can...
  - Draw stack diagrams, and track stack usage throughout function calls
  - Trace stack usage through recursive calls, with stack frames for each invocation
  - Trace stack frames and registers through a procedure call
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

```c
int Q(int i)
{
    int t = 3*i;
    int v[10];
    
    return v[t];
}
```
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: Addresses range from $0xF...F$ to $0x0...0$
  - Low Addresses: Addresses range from $0x0...0$ to $0xF...F$

- **Stack**
  - Grow towards each other to maximize use of space

- **Dynamic Data (Heap)**

- **Static Data**

- **Literals**

- **Instructions**
Memory Permissions

Segmentation fault: impermissible memory access

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
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`
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.
How are y’all feeling about stack structure, push, pop?
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?
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/ndro9E

---

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

---

00000000000400540 <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

00000000000400550 <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 *(1) read ret addr at %rsp into %rip*
  2) Jump to address *(2) move %rsp up*
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)

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
```

```
%rip 0x400549
%rsp 0x120
```

```
400544: call 400550 <mult2>
```

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

```
0x400550 <mult2>
0x400549
0x130
0x128
0x120
```
Feelings check: Procedure call/return?
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:
  1. %rdi
  2. %rsi
  3. %rdx
  4. %rcx
  5. %r8
  6. %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;
}
```

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

```assembly
00000000000400540 <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
    ...
```

```assembly
00000000000400550 <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-Based Languages

- 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(...) {
  
  if(...) {
    amI();
    amI();

  }

}
```

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

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

Stack diagram:
- `main`
- `whoa`

- `whoa` is the entry point.
- The stack frame for `whoa` includes:
  - `%rbp` (frame pointer)
  - `%rsp` (return stack pointer)

- The frame pointer is not necessary, as `whoa` does not use it.
- The stack frame contains:
  - The function name `whoa`.
  - Local variables.

- The stack frame is allocated on the stack.

- The `who()` function does not use the stack frame, as it only uses local variables.

- The diagram shows the stack allocation and the flow of execution.
2) Call to `who`

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

```
whoa

who

amI  amI

%rbp

%rsp

“create” frame by manipulating %rsp

Stack
```
3) Call to amI (1)

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

amI

Stack

%rbp

%rsp

whoa

who

amI

amI

amI

amI

amI

amI

amI

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

Stack

whoa

who

amI

amI

amI

%rbp

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

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

8) Return from call to `amI`
```
9) (second) Call to `amI (4)`
10) Return from (second) call to `amI`

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

Stack diagram:
- `whoa`
- `who`
- `amI`
- `amI4`
- `amI2`
- `amI3`
- `%rbp` (Register Base Pointer)
- `%rsp` (Register Stack Pointer)
11) Return from call to `whoa`

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

**Call Chain:**
```
main \(\uparrow\)
  whoa\(2\)
    who \(3\)
      ami \(5\)
        ami \(6\)
  ami \(4\)
```

**Stack:**
```
main
  whoa
    who
      ami
        ami
          ami
    ami
      ami
        ami
          ami
```

- **Total stack frames created:** 7
- **Maximum stack depth:** 6 frames
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 \texttt{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)
Checking in!

- 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?
Checking in!

- During the setup and execution of the following function, how many registers do we know must be modified?

```c
int foo(int a, int b, int c);
```

3, 4, 5, 6, Help!
Feelings check: Stack Frames!