Procedures I
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan
Administrivia

- Homework 2 due tonight
- Lab 2 due next Friday (10/27)
  - Ideally want to finish well before the midterm
- Homework 3 released next week
  - On midterm material, but due after the midterm

- **Midterm** (10/30, 5-6:30pm, KNE 120)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 70%
  - **Review session** (10/27) in EEB 105 from 5:30-7:30pm
**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:**

**High Addresses**
- Increasing Addresses
  - Stack Grows Down
  - Low Addresses
    - `0x00...00`

**Last In, First Out (LIFO)**
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:**

1. Move `%rsp` down (subtract)
2. Store `src` at `%rsp`

High Addresses

Increasing Addresses

Stack Grows Down

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

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst` (must be register)
  - *Increment* `%rsp` by 8

Example:
- `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`
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 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;
}
```

**Compiler Explorer:**
https://godbolt.org/g/cKKDZn
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:
    
    | Address  | Instruction      | Result          |
    |----------|------------------|-----------------|
    | 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

%rsp 0x120
%rip 0x400544
0x120
0x128
0x130
Procedure Call Example (step 2)

0000000000340540 <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

%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

- Diane’s Silk Dress Costs $89

- Return value
  - %rax

Stack (Memory)

- Only allocate stack space when needed

High Addresses

Low Addresses

0x00...00
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
```
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 pointer

- 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

```
yoo (...) {
  
  who();
  
}
```

```
yoo
    who
    ami
    ami
    ami
```

Stack

```
%rbp
%rsp
```
2) Call to who

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

Stack

- yoo
- who
- %rbp
- %rsp

Diagram:
- yoo
- who
- amI
- amI
3) Call to amI (1)
4) Recursive call to `amI (2)`

```
yoo(…)
{  who(…)
  { amI(…)
    { amI(…)
      {  
        if(){
        amI()
      }
      }
      amI()
    }
    amI()
  }
  who(); amI();
}
```

Stack:
- `yoo`
- `who`
- `amI_1`
- `amI_2`

Variables:
- `%rbp`
- `%rsp`
5) (another) Recursive call to amI (3)
6) Return from (another) recursive call to `amI`

```
yoo(…) {
  who(…) {
    amI(…) {
      amI(…) {
        • if() {
            amI()
          } else {
            amI()
          }
        }
      }
    }
  }
}
```
7) Return from recursive call to `amI`

```c
yoo(...) {
    who(...) {
        amI(...) {
            if()
                amI()
        }
    }
}  
```

Stack:
- `%rbp`
- `%rsp`
- `amI_3`
- `amI_2`
- `amI_1`
- `who`
- `yoo`
8) Return from call to amI

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

Stack:
- `yoo`
- `who`
- `amI` (3 frames)

Variables:
- `%rbp`
- `%rsp`
9) (second) Call to `amI` (4)
10) Return from (second) call to amI

```
10) Return from (second) call to amI

```

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

```

```
yoo
  who
    amI
      amI
          amI
```

```
Stack
  yoo
  who
    amI
    amI
    amI
```

```
%rbp
%rsp
```
11) Return from call to `who`

```c
yoo (...) {
  ...
  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)
Peer Instruction Question

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

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

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

Vote only on 3rd question at http://PollEv.com/justinh

A. 1  B. 2  C. 3  D. 4
Example: *increment*

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
Procedure Call Example (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything
  - Pushed onto stack by `call call_incr`

Initial Stack Structure

- Return addr <main+8>
Procedure Call Example (step 1)

```c
long call_incr() {  
  long v1 = 351;  
  long v2 = increment(&v1, 100);  
  return v1+v2;  
}
```

Stack Structure

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return addr &lt;main+8&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>351</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allocate space for local vars

- Setup space for local variables
  - Only `v1` needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

### Stack Structure

- Return addr `<main+8>`
- `351` ← `%rsp+8`
- `Unused` ← `%rsp`  

#### Call_incr:

- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Aside:** `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes one less byte to encode a `movl` than a `movq`.

- **Register** | **Use(s)**
  - `%rdi` | `&v1`
  - `%rsi` | 100
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr `<main+8>`
- 351
- Unused
- Return addr `<call_incr+?>` ←%rsp

- **State while inside** `increment`
  - Return address on top of stack is address of the `addq` instruction immediately following call to `increment`

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Call Example (step 4)

```c
long call_incr() {  
  long v1 = 351;  
  long v2 = increment(&v1, 100);  
  return v1+v2;  
}
```

```
call_incr:  
  subq  $16, %rsp  
  movq  $351, 8(%rsp)  
  movl  $100, %esi  
  leaq  8(%rsp), %rdi  
  call  increment  
  addq  8(%rsp), %rax  
  addq  $16, %rsp  
  ret
```

```
increment:  
  movq  (%rdi), %rax  # x = *p  
  addq  %rax, %rsi  # y = x+100  
  movq  %rsi, (%rdi)  # *p = y  
  ret
```

- State while inside `increment`
  - *After* code in body has been executed

**Stack Structure**

<table>
<thead>
<tr>
<th>Return addr</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;main+8&gt;</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>451</td>
<td>451</td>
</tr>
<tr>
<td>Unused</td>
<td>451</td>
</tr>
<tr>
<td>&lt;call_incr+?&gt;</td>
<td>351</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>

Stack Structure

Return addr `<main+8>`

451 ←%rsp+8

UNUSED ←%rsp
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

Stack Structure

- Return addr <main+8>
- %rdi &v1
- %rsi 451
- Unused

Update %rax to contain v1+v2

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr `<main+8>`
- 451
- Unused

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

---

State *just before* returning from call to `call_incr`

- **Stack Structure**
  - Return addr `<main+8>`

- **Register Use(s)**
  - `%rdi` &v1
  - `%rsi` 451
  - `%rax` 802
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**call_incr:***
  - `subq $16, %rsp`
  - `movq $351, 8(%rsp)`
  - `movl $100, %esi`
  - `leaq 8(%rsp), %rdi`
  - `call increment`
  - `addq 8(%rsp), %rax`
  - `addq $16, %rsp`
  - `ret`

**Final Stack Structure**

- State immediately *after* returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>