The Stack & Procedures
CSE 410 Winter 2017

Instructor: Justin Hsia
Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui
Administivia

- Homework 3 released today, due next Thu (2/9)
- Lab 2 deadline pushed to Monday (2/13)
  - Definitely want to start before the Midterm

- **Midterm** (2/10) in lecture
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Aiming for average of 75%

- **Midterm review session** (2/7) in BAG 261 from 5-7:30pm
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;
}
```

```asm
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
0000000000400541:  movq   %rdx,%rbx    # Save dest
0000000000400544:  call   400550 <mult2> # mult2(x,y)
0000000000400549:  movq   %rax,(%rbx)   # Save at dest
    ...
```

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

```asm
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

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

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

amI (...)
{
    ...
    if (...){
        amI();
    }
    ...
}
```

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

```c
yoo (...) {
    ...
    who () ;
    ...
}
```
2) Call to who

```c
yoo(...) {
  who(...) {
    •
    amI();
    •
    amI();
  }
}
```
3) Call to `amI` (1)
4) Recursive call to `amI` (2)

```
yoo(...)
{
  who(...)
  {
    amI(...)
    {
      amI(...)  
      {
        if()
        {
          amI()
        }
      }
    }
    .
  }
  .
}
```
5) *(another) Recursive call to amI (3)*

Stack

```plaintext
Stack

%rbp

yoo

who

amI

amI

amI

amI

amI

%rsp
```

Code:

```c
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        if(){
          amI()
        }
      }
    }
  }
}
```
6) Return from (another) recursive call to amI

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

Stack diagram:

```
Stack:
  yoo
  who
  amI1
  amI2
  amI3

%rbp -> %rsp
```
7) Return from recursive call to amI
8) Return from call to `amI`
9) (second) Call to `amI` (4)
10) **Return from (second) call to amI**

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

The stack frame shows the call to `amI` from the second call to `who` inside `yoo`. The stack pointer (`%rsp`) points to the return address at the bottom of the stack, indicating the next instruction to be executed after returning from the `amI` function.
11) Return from call to who

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

Stack:
- `%rbp` pointer
- `%rsp` pointer
- `yoo`
- `who`
- `amI_4`
- `amI_2`
- `amI_1`
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):

```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;
}
```

- **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
Example: increment

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

increment:
```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

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

long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}

Initial Stack Structure

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

Return address on stack is the address of instruction immediately following the call to "call_incr"
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr <main+8>
- 410
- Unused

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 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Set up parameters for call to increment

```
call_incr:
    subq $16, %rsp
    movq $410, 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.
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**call_incr:**
- `subq $16, %rsp`
- `movq $410, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**increment:**
- `movq (%rdi), %rax`
- `addq %rax, %rsi`
- `movq %rsi, (%rdi)`
- `ret`

**Stack Structure**

- Return addr `<main+8>`
- 410
- Unused
- Return addr `<call_incr+?>`

- 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 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq  $16, %rsp
    movq  $410, 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

<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>510</td>
</tr>
<tr>
<td>%rax</td>
<td>410</td>
</tr>
</tbody>
</table>

Stack Structure

- Return addr `<main+8>`
- 510
- Unused
- Return addr `<call_incr+?>` ← %rsp
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 410;
    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

Stack Structure:

- Return addr `<main+8>`
- `%rsp` ← `%rsp+8`
- `%rdi` ← `&v1`
- `%rsi` ← `510`
- `%rax` ← `410`
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure

```asm
subq $16, %rsp
movq $410, 8(%rsp)
movl $100, %esi
leaq 8(%rsp), %rdi
call increment
addq 8(%rsp), %rax
addq $16, %rsp
ret
```

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>510</td>
</tr>
<tr>
<td>%rax</td>
<td>510+410</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Stack Structure

- Return addr `<main+8>`
- 410
- 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>510</td>
</tr>
<tr>
<td>%rax</td>
<td>920</td>
</tr>
</tbody>
</table>

De-allocate space for local vars
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

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

---

### Stack Structure

<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>510</td>
</tr>
<tr>
<td>%rax</td>
<td>920</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- **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>510</td>
</tr>
<tr>
<td>%rax</td>
<td>920</td>
</tr>
</tbody>
</table>
Lab 2 Demo

- Let’s look at that binary bomb!