Procedures II
CSE 351 Autumn 2020

Instructor:
Justin Hsia

Teaching Assistants:
Aman Mohammed
Ami Oka
Callum Walker
Cosmo Wang
Hang Do
Jim Limprasert
Joy Dang
Julia Wang
Kaelin Laundry
Kyrie Dowling
Mariam Mayanja
Shawn Stanley
Yan Zhe Ong

http://xkcd.com/1270/
Administrivia

- Lab 2 due Friday (10/30)
  - Since you are submitting a text file (`defuser.txt`), there won’t be any Gradescope autograder output this time
  - Extra credit (bonus) needs to be submitted to the extra credit assignment

- Midterm (take home, 10/31–11/2)
  - Find groups of 5 for the group stage
    - Automatic assignment will happen at the end of Thursday (10/29)
  - Make notes and use the [midterm reference sheet](#)
  - Form study groups and look at past exams!
Reading Review

- Terminology:
  - Stack frame: return address, saved registers, local variables, argument build
  - Register saving conventions: callee-saved and caller-saved

- Questions from the Reading?
Polling 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;
}
```

```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?

A. 1  B. 2  C. 3  D. 4
Review Question

- In the following function, which instruction(s) pertain to the **local variables** and **saved registers** portions of its stack frame?

```assembly
call_incr2:
1    pushq  %rbx
2    subq  $16, %rsp
3    movq  %rdi, %rbx
4    movq  $351, 8(%rsp)
5    movl  $100, %esi
6    leaq  8(%rsp), %rdi
7    call  increment
8    addq  %rbx, %rax
9    addq  $16, %rsp
10   popq  %rbx
11   ret
```
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)
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\textsuperscript{st} arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} 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 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Initial Stack Structure**

Return addr <main+8>

- 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`

**Instruction Code**

```
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
```
Procedure Call Example (step 1)

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

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
- Unused
  - ←%rsp
  - ←%rsp+8

**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`.
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+?>`

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td><code>&amp;v1</code></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>100</td>
</tr>
<tr>
<td><code>%rax</code></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;
}
```

Stack Structure

- Return addr <main+8>
- 451
- Unused
- Return addr <call_incr+?>

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

Increment:

<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>

Increment:

```assembly
movq (%rdi), %rax  # x = *p
addq %rax, %rsi   # y = x + 100
movq %rsi, (%rdi) # *p = y
ret
```
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>
Procedure Call Example (step 6)

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

- **Return addr <main+8>**
  - `451`
  - `Unused` ← `%rsp` ← `%rsp+8`

Update `%rax` to contain `v1 + v2`

---

**Register Use(s)**

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

![Stack Structure Diagram]

### 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>

![Register Use(s) Table]
Procedure Call Example (step 8)

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

**Stack Structure**

![Stack Structure Diagram]

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

<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 9)

```c
long call_incr() {
    long v1 = 351;
    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)

```asm
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
```

<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>
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Register Saving Conventions

- When procedure `whoa` calls `who`:
  - `whoa` is the **caller**
  - `who` is the **callee**

- Can registers be used for temporary storage?

  ```
  whoa:
  ... 
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ... 
  ret
  ```

  ```
  who:
  ... 
  subq $18213, %rdx
  ... 
  ret
  ```

- No! Contents of register `%rdx` overwritten by `who`!

- This could be trouble – something should be done. Either:
  - **Caller** should save `%rdx` before the call (and restore it after the call)
  - **Callee** should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

- **“Caller-saved” registers**
  - It is the **caller**’s responsibility to save any important data in these registers before calling another procedure (i.e., the **callee** can freely change data in these registers)
  - **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call

- **“Callee-saved” registers**
  - It is the callee’s responsibility to save any data in these registers before using the registers (i.e., the **caller** assumes the data will be the same across the **callee** procedure call)
  - **Callee** saves values in its stack frame before using, then restores them before returning to **caller**
Silly Register Convention Analogy

1) Parents (caller) leave for the weekend and give the keys to the house to their child (callee)
   - Being suspicious, they put away/hid the valuables (caller-saved) before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party (computation), spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (callee-saved) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
x86-64 Linux Register Usage, part 1

- **%rax**
  - Return value
  - Also *caller*-saved & restored
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also *caller*-saved & restored
  - Can be modified by procedure

- **%r10, %r11**
  - *Caller*-saved & restored
  - Can be modified by procedure
x86-64 Linux Register Usage, part 2

- **%rbx, %r12, %r13, %r14**
  - **Callee**-saved
  - **Callee** must save & restore
- **%rbp**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match
- **%rsp**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>Argument #5 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6 - <strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%r10</code></td>
<td><strong>Caller saved</strong></td>
</tr>
<tr>
<td><code>%r11</code></td>
<td><strong>Caller Saved</strong></td>
</tr>
<tr>
<td><code>%r12</code></td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td><code>%r13</code></td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td><code>%r14</code></td>
<td><strong>Callee saved</strong></td>
</tr>
<tr>
<td><code>%r15</code></td>
<td><strong>Callee saved</strong></td>
</tr>
</tbody>
</table>
**Callee-Saved Example (step 1)**

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

---

**Initial Stack Structure**

- ... (Stack initial values)
- ret addr
- %rsp

**Resulting Stack Structure**

- ... (Stack after call)
- ret addr
- Saved %rbx
- 351
- Unused
- %rsp+8
- %rsp
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**Stack Structure**

- Rtn address
- Saved `%rbx`
- `351`
- Unused
- `%rsp+8`
- `%rsp`

**Pre-return Stack Structure**

- Rtn address
- `%rsp`
Why Caller *and* Callee Saved?

- We want *one* calling convention to simply separate implementation details between caller and callee.

- In general, neither caller-save nor callee-save is “best”:  
  - If caller isn’t using a register, caller-save is better  
  - If callee doesn’t need a register, callee-save is better  
  - If “do need to save”, callee-save generally makes smaller programs  
    - Functions are called from multiple places

- So… “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring.
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!
Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Recursive Function

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Compiler Explorer:
https://godbolt.org/z/naP4ax
- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!
Recursive Function: Base Case

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Trick because some AMD hardware doesn’t like jumping to ret
Recursive Function: **Callee Register Save**

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
</tbody>
</table>

**The Stack**

Need original value of `x` after recursive call to `pcount_r`.

“Save” by putting in `%rbx` *(callee saved)*, but need to save old value of `%rbx` before you change it.

**pcount_r:**

```
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Recursive Function: Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x (new)</td>
<td>Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

**The Stack**

```
%rsp →
saved %rbx
rtn <main+?>
...
```

**pcount_r:**

```
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Recursive Function: Call

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

**Register Use(s) Type**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Recursive call return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

**The Stack**

```
%rsp →
%rbx
saved %rbx
rtn <pcount_r+22>
...
...
```

**pcount_r:**

```
    movl    $0, %eax
    testq   %rdi, %rdi
    jne     .L8
    rep ret

  .L8:
    pushq   %rbx
    movq    %rdi, %rbx
    shrq    %rdi
    call    pcount_r
    andl    $1, %ebx
    addq    %rbx, %rax
    popq    %rbx
    ret
```
Recursive Function: Result

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1) + pcount_r(x>>1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x&amp;1</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

The Stack

```
%rsp →
saved %rbx
rtn <main+?>
...
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    rep ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```
Recursive Function: Completion

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x&1)+pcount_r(x>>1);
}
```

The Stack

```
%rsp →
  ... 
  rtn <main+?>
  saved %rbx
```

Register Use(s) Type

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Previous %rbx value</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

```
pcount_r:
  movl $0, %eax
  testq %rdi, %rdi
  jne .L8
  rep ret
  .L8:
  pushq %rbx
  movq %rdi, %rbx
  shrq %rdi
  call pcount_r
  andl $1, %ebx
  addq %rbx, %rax
  popq %rbx
  ret
```
GDB Demo

- Let’s examine the pcound_r stack frames on a real machine!
  - Using `pcount.c` from the course website

- You will need to use GDB to get through the Midterm
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Checking the current stack frames (`backtrace`)
  - Getting stack frame information (`info frame <#>`)
Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return address
  - Register saving conventions prevent one function call from corrupting another’s data
    - Unless the code explicitly does so (e.g. buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)

- Also works for mutual recursion (P calls Q; Q calls P)
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure needs to grow its stack frame when it:
  - Has too many local variables to hold in caller-saved registers
  - Has local variables that are arrays or structs
  - Uses & to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using caller-saved registers and then calls a procedure
  - Modifies/uses callee-saved registers
x86-64 Procedure Summary

- **Important Points**
  - Procedures are a combination of instructions and conventions
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions

- **Heavy use of registers**
  - Faster than using memory
  - Use limited by data size and conventions

- **Minimize use of the Stack**

```
Return
Addr
Saved
Registers
+    +
Local
Variables
Argument
Build

%rbp  →  (Optional)
%rsp  →

Args
ments
7+
Return
Addr
Saved
Registers
+    +
Local
Variables
Argument
Build

Caller
Frame
```