Procedures II
CSE 351 Summer 2020

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

http://xkcd.com/1790/
Administrivia

❖ Questions doc: https://tinyurl.com/CSE351-7-17

❖ Unit Summary 1 due tonight (7/17) – 11:59pm
  ▪ Can still use late days until 7/20

❖ Mid-quarter Survey due tonight (7/17) – 11:59pm
  ▪ Submit via Canvas!

❖ hw8, hw9, hw10, hw11 due Monday (7/20) – 10:30am

❖ hw12 due Wednesday (7/22)

❖ Lab 2 due Wednesday (7/22)
  ▪ GDB Tutorial on Gradescope walks through first phase
11) Return from call to who

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

Stack:

```
whoa

who

amI

amI

%rbp

%rsp

who

amI

amI

amI_4

amI_2

amI_3
```
Polling Question [Proc I – a]

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

Vote only on 3rd question at [http://pollev.com/pbjones](http://pollev.com/pbjones)
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;
}
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>1\textsuperscript{st} arg (p)</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>2\textsuperscript{nd} arg (val), y</td>
</tr>
<tr>
<td><code>%rax</code></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_call_incr`

Initial Stack Structure

```
Return addr <main+8>
```

- %rsp
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

**Stack Structure**

- Return addr `<main+8>`
- `351` (unused)
- `old %rsp` ← `%rsp+8`
- `%rsp` ← `%rsp`
Procedure Call Example (step 2)

```
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
```

```
Set up parameters for call to increment

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.
```

```
<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>
</tbody>
</table>
```
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

**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
- addq %rax, %rsi
- movq %rsi, (%rdi)
- ret
Procedure Call Example (step 4)

```
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:

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

- `ret`

register use:

<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)

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

```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
```

<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;
}
```

Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>451</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>

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

<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>
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`

```
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>` ← `%rsp`

- 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;
}
```

### Final Stack Structure

```
... ← %rsp
```

- 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)

#### 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**
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, %r15`
  - **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>Description</th>
<th>Saved By</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #3 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #4 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 - Caller saved</td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
<td></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;
}
```

```
call_incr2:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```
Callee-Saved Example (step 2)

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

**Stack Structure**

```
call_incr2:
    pushq  %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

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

**Pre-return Stack Structure**

```
call_incr2:
    pushq  %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

- **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/xFCrsw](https://godbolt.org/z/xFCrsw)

- 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);
}
```

Trick because some AMD hardware doesn’t like jumping to ret

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

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.
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);
}
```

### The Stack

```
...  
%rsp  
saved %rbx  
rt� <main+?>  
```

### Register Use(s) and 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>

### Code Snippet

```
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

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

The Stack

Register | Use(s)                     | Type
---------|----------------------------|------
%rax     | Recursive call return value| Return value
%rbx     | x (old)                    | Callee saved

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

/* 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

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

The Stack

<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>

```
int 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
    ret
```
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