CSE 351 Section 5
Stack & Procedures, Recursion
Administrivia

- **Homework 12**
  - Due Friday (10/28)

- **Homework 13**
  - Due Wednesday (11/2) – longer because it covers two lectures

- **Lab 2:**
  - Due this Friday (10/28)
  - Make sure you put each phase answer on a new line, and have an empty line after your last phase answer (don’t actually type ‘\n’)  
  - One late day covers Saturday and Sunday
Stack & Procedures

(%rsp is the MVP)
Memory Layout

- Stack is located at the top of our memory layout.
- Stack is placed upside down in memory, with higher addresses considered the “bottom” and lower addresses considered the “top.”
- There is a dedicated register %rsp that points to the current top of the stack.
Stack Frame Structure

- Return address
  - Pushed by `callq`; address of instruction after `callq`

- Callee-saved registers
  - Only if function modifies/uses them

- Local variables
  - Variables that fit in a register may not be allocated on the Stack
  - Unavoidable if variable is too big for a register (e.g., array)
  - Unavoidable if variable needs an address (i.e., uses `&var`)

- Caller-saved registers
  - Only if values are needed across a function call

- Argument build
  - Only if function calls a function with more than six arguments
Calling Conventions

First 6 arguments are ordered in registers:
1: %rdi, 2: %rsi, 3: %rdx, 4: %rcx, 5: %r8, 6: %r9

Registers are not part of memory/the stack.

Additional arguments are pushed to the stack by the caller before invoking callq
In reverse order: arg n pushed first, arg 7 last.
Part of the caller’s stack frame.

Return value
Placed in %rax.
Register Saving Conventions

“Caller-saved” registers:
- `%rax, %rcx, %rdx, %rsi, %rdi, %r8–%r11
- If caller needs to use their value(s) across a function call, then push onto the stack.
- Pushed just before function call; popped right after.

“Callee-saved” registers:
- `%rbx, %rbp, %r12–%r15
- If callee wants to change their value(s), then push onto the stack.
- Pushed at beginning of function; popped just before ret.
In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- **main** is actually started by a library routine, so it has arguments and a return address, too!

```assembly
main:
01    pushq   %rbx
02    subq    $16, %rsp
03    movl    %edi, %ebx
04    movl    $351, 12(%rsp)
05    movl    $1, (%rsp)
06    movl    $2, 4(%rsp)
07    movl    $3, 8(%rsp)
08    pushq   $7
09    movl    $6, %r9d
```
Stack Frame Example

- main uses %ebx, a callee-saved register, in Line 03, so it must save the old value on the stack before then.

```
main:
01        pushq   %rbx
02        subq    $16, %rsp
03        movl    %edi, %ebx
04        movl    $351, 12(%rsp)
05        movl    $1, (%rsp)
06        movl    $2, 4(%rsp)
07        movl    $3, 8(%rsp)
08        pushq   $7
09        movl    $6, %r9d
```
Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- `main` then allocates space on the stack for local variables:
  - We use `&x`, so `x` must go on the stack
  - `a[]` takes up 12 bytes, so must go on the stack

```plaintext
main:
01    pushq   %rbx
02    subq    $16, %rsp
03    movl    %edi, %ebx
04    movl    $351, 12(%rsp)
05    movl    $1, (%rsp)
06    movl    $2, 4(%rsp)
07    movl    $3, 8(%rsp)
08    pushq   $7
09    movl    $6, %r9d
```

![Stack diagram showing the allocation of space for local variables](image)

- `return address (libc_start_main)`
- `saved %rbx (callee-saved)`
- `space for local variables`
Stack Frame Example

- `main` then stores the initial values of the local variables in their corresponding locations:
  - `x` is placed at `12(%rsp)`
  - `a[0]` is placed at `(%rsp)`, with the other two elements directly following that

```
main:
01     pushq   %rbx
02     subq    $16, %rsp
03     movl    %edi, %ebx
04     movl    $351, 12(%rsp)
05     movl    $1, (%rsp)
06     movl    $2, 4(%rsp)
07     movl    $3, 8(%rsp)
08     pushq   $7
09     movl    $6, %r9d

Note: without an explicit reference to `&x`, `x` may not be stored on the stack depending on whether the compiler can optimize that memory access away
```
Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}
```

```c
int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

Note that, in this example, the compiler chose to save %rdi (caller-saved) in %rbx back in Line 03 instead of pushing the old value of %rdi to the stack here so no caller-saved registers are pushed to the stack.

main:
01 pushq %rbx
02 subq $16, %rsp
03 movl %edi, %ebx
04 movl $351, 12(%rsp)
05 movl $1, (%rsp)
06 movl $2, 4(%rsp)
07 movl $3, 8(%rsp)
08 pushq $7
09 movl $6, %r9d

Stack

return address
(libc_start_main)
saved %rbx
(callee-saved)

351 (x) 3 (a[2])
2 (a[1]) 1 (a[0])
Stack Frame Example

- main then prepares to call foo, which includes putting arg7 (7) on the stack.
  - arg1/%rdi is a pointer; note that &x is now 20 bytes above %rsp after arg7 is pushed
**Stack Frame Example**

- `main` calls `foo`, which pushes the return address to `main` on the stack and marks the beginning of a new stack frame.

```
:  
15  movl  $0, %eax
16  call  foo
17  addl  %ebx, %eax
18  addq  $24, %rsp
19  popq  %rbx
20  ret
foo:
21  movl  (%rdi), %eax
22  addl  8(%rsp), %eax
23  ret
```
Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}

int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- `foo` has a minimal stack frame and doesn’t put anything on the stack, however, it does access memory on the stack, since it sums `*arg1` and `arg7`.

```
15      movl    $0, %eax
16      call    foo
17      addl    %ebx, %eax
18      addq    $24, %rsp
19      popq    %rbx
20      ret

foo:
21      movl    (%rdi), %eax
22      addl    8(%rsp), %eax
23      ret
```
Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- **foo** returns to **main** using **ret**, which pops the return address to **main** from the stack into %rip.
  - **foo** has finished execution and its stack frame is now deallocated

```assembly
15        movl    $0, %eax
16        call    foo
17        addl    %ebx, %eax
18        addq    $24, %rsp
19        popq    %rbx
20        ret

foo:
21        movl    (%rdi), %eax
22        addl    8(%rsp), %eax
23        ret
```
In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}

int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let's look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- main returns the original value of %rdi, which was stored in %rbx, plus the return value of foo.

```assembly
15        movl    $0, %eax
16        call    foo
17        addl    %ebx, %eax
18        addq    $24, %rsp
19        popq    %rbx
20        ret

foo:
21        movl    (%rdi), %eax
22        addl    8(%rsp), %eax
23        ret
```
Stack Frame Example

In x86-64, stack can be broken down into stack frames of functions. Consider the following lines of code:

```c
int main(int argc, char* argv[]) {
    int x = 351;
    int a[] = {1, 2, 3};
    int y = foo(&x, 2, 3, 4, 5, 6, 7);
    return y + argc;
}
int foo(int* arg1, int arg2, ..., int arg7) {
    return *arg1 + arg7;
}
```

Let’s look at how the stack grows and shrinks as the code above executes in assembly.
Stack Frame Example

- **main** deallocateds the argument build and local variables simultaneously (we must work our way up the stack).
  - Note that these would have been split up if there had been caller-saved registers on the stack.

```
15    movl  $0, %eax
16    call  foo
17    addl  %ebx, %eax
18    addq  $24, %rsp
19    popq  %rbx
20    ret

foo:
21    movl  (%rdi), %eax
22    addl  8(%rsp), %eax
23    ret
```
Stack Frame Example

- main must restore the saved value of %rbx before returning.

```
15        movl    $0, %eax
16        call    foo
17        addl    %ebx, %eax
18        addq    $24, %rsp
19        popq    %rbx
20        ret

foo:
21        movl    (%rdi), %eax
22        addl    8(%rsp), %eax
23        ret
```
// Recursive function rfun
long rfun(char *s) {
    if (*s) {
        long temp = (long)*s;
        s++;
        return temp + rfun(s);
    }
    return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
    char *s = "CSE351";
    long r = rfun(s);
    printf("r: %ld\n", r);
}
Stack Exercise

// Recursive function rfun
long rfun(char *s) {
    if (*s) {
        long temp = (long)*s;
        s++;
        return temp + rfun(s);
    }
    return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
    char *s = "CSE351";
    long r = rfun(s);
    printf("r: %ld\n", r);
}

a) In terms of the C function, what value is being saved on the stack?

```
0000000000004005e6 <rfun>:
4005e6: 0f b6 07 movzbl (%rdi),%eax
4005e9: 84 c0 test %al,%al
4005eb: 74 13 je 400600 <rfun+0x1a>
4005ed: 53 push %rbx
4005ee: 48 0f be d8 movsbq %al,%rbx
4005f2: 48 83 c7 01 add $0x1,%rdi
4005f6: e8 eb ff ff ff callq 4005e6 <rfun>
4005fb: 48 01 d8 add %rbx,%rax
4005fe: eb 06 jmp 400606 <rfun+0x20>
400600: b8 00 00 00 00 mov $0x0,%eax
400605: c3 retq
400606: 5b pop %rbx
400607: c3 retq
```
Stack Exercise

// Recursive function rfun
long rfun(char *s) {
    if (*s) {
        long temp = (long)*s;
        s++;
        return temp + rfun(s);
    }
    return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
    char *s = "CSE351"
    long r = rfun(s);
    printf("r: %ld\n", r);
}

b) What is the return address to rfun that gets stored on the stack during the recursive calls (in hex)?

000000000004005e6 <rfun>:
4005e6: 0f b6 07 movzbl (%rdi),%eax
4005e9: 84 c0 test %al,%al
4005eb: 74 13 je 400600 <rfun+0x1a>
4005ed: 53 push %rbx
4005ee: 48 0f be d8 movsbq %al,%rbx
4005f2: 48 83 c7 01 add $0x1,%rdi
4005f6: e8 eb ff ff ff callq 4005e6 <rfun>
4005fb: 48 01 d8 add %rbx,%rax
4005fe: eb 06 jmp 400606 <rfun+0x20>
400600: b8 00 00 00 00 mov $0x0,%eax
400605: c3 retq
400606: 5b pop %rbx
400607: c3 retq
c) Assume main calls rfun with char *s = “CSE351” and then prints the result using the printf function, as shown in the C code above. Assume printf does not call any other procedure. Starting with (and including) main, how many total stack frames are created, and what is the maximum depth of the stack?

c) char *s = “CSE351”
Stack Exercise

00000000004005e6 <rfun>:

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffdb48</td>
<td>Unknown</td>
<td>%rsp when main is entered</td>
</tr>
<tr>
<td>0x7fffffffdb38</td>
<td>0x400616</td>
<td>Return address to main</td>
</tr>
<tr>
<td>0x7fffffffdb30</td>
<td>Unknown</td>
<td>Original %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb28</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb20</td>
<td>*s, “C”, 0x43</td>
<td>Saved %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb18</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb10</td>
<td>*s, *(s+1), “S”, 0x53</td>
<td>Saved %rbx</td>
</tr>
<tr>
<td>0x7fffffffdb08</td>
<td>0x4005fb</td>
<td>Return address</td>
</tr>
<tr>
<td>0x7fffffffdb00</td>
<td>*s, *(s+2), “E”, 0x45</td>
<td>Saved %rbx</td>
</tr>
</tbody>
</table>