Introduction aux Systèmes d'Exploitation

Unit 8: Invocation and Stack management

François Taïani

Invocation Mechanism

- Well known: procedures (C), methods (Java, C++)
- Goal: write once, call from everywhere
- Challenge: how to remember where to return?

Example

```c
#include <stdio.h>

void foo() {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar() {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```

Represent the program's structure as a call graph

... And the program's execution as a call tree

Invocation Mechanism

- Challenge: how to remember where to return?
  - → **foo** returns into **main** in 1st invocation
  - → but into **bar** for 2nd invocation
- Solution:
  - → Use a stack!

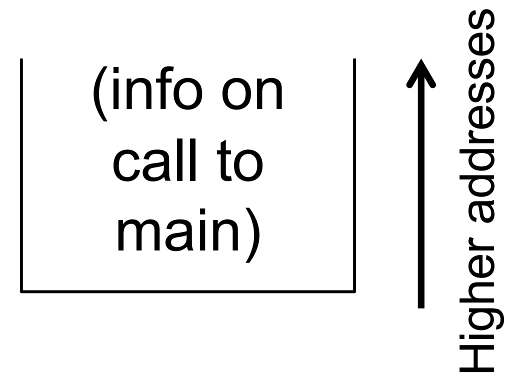
The Call Stack

- Special zone in process' memory ("stack segment")
  - ➔ LIFO principle
  - ➔ Grows downwards (on x86): from high to low addresses (most common, but reverse possible, cf. ARM)

- Role of the stack
  - ➔ **Remember** where to return to after invocation
  - ➔ Pass **parameters** (more on this, registers can be used too)
  - ➔ Store **local variables** (more on this)
  - ➔ Retrieve **returned values** (more on this)

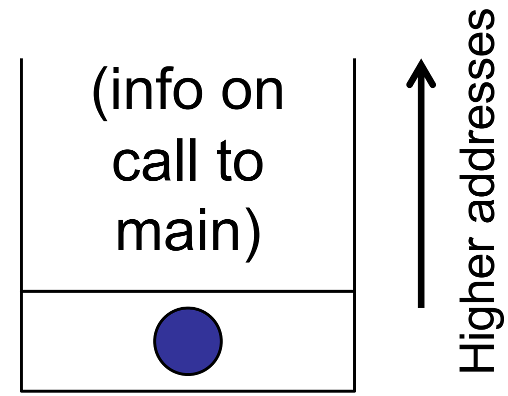
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



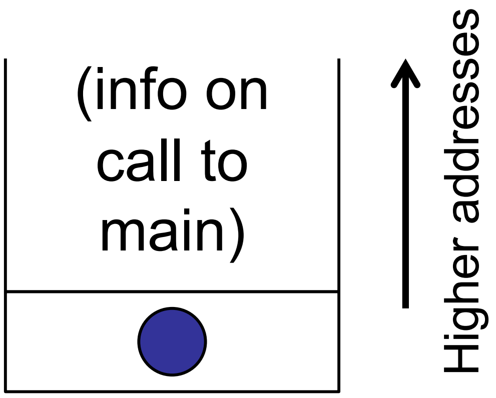
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



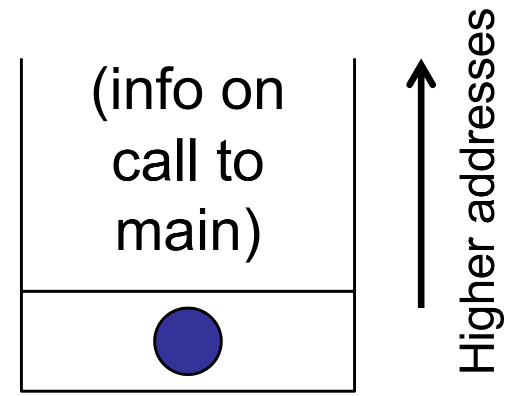
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



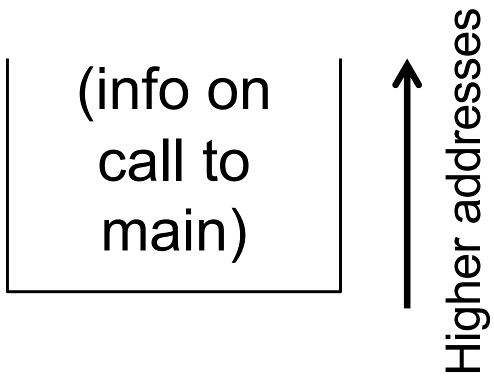
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



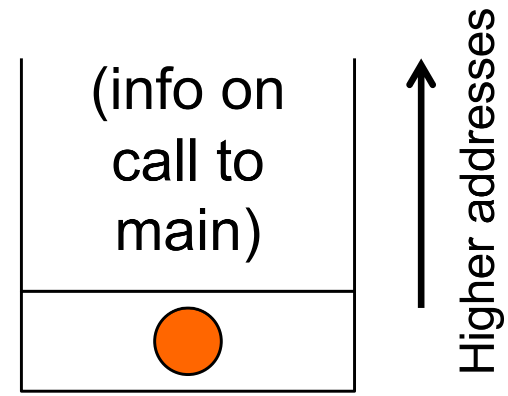
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



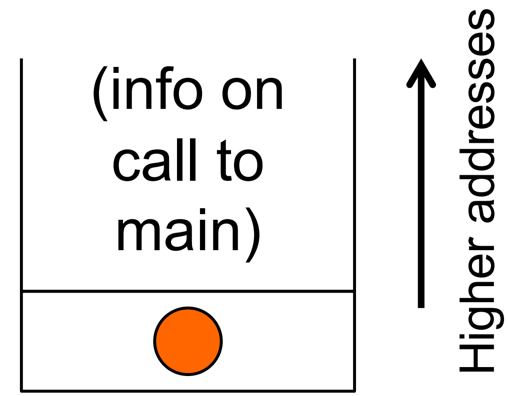
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



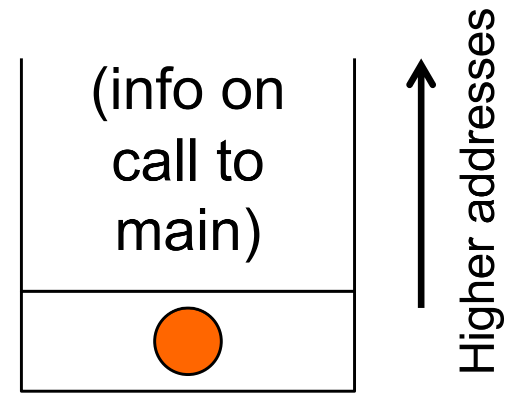
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



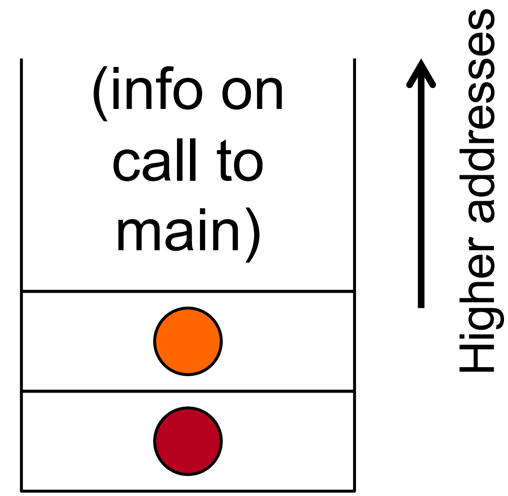
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



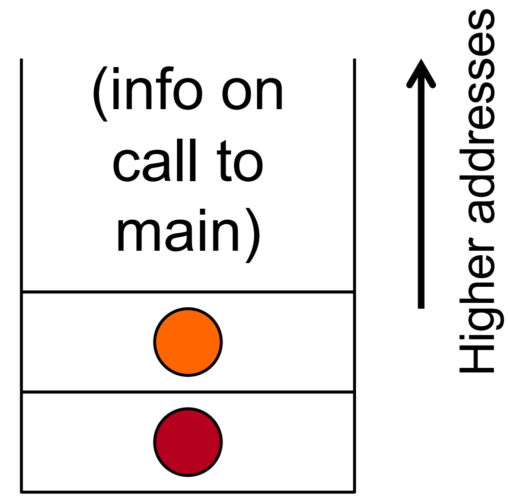
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



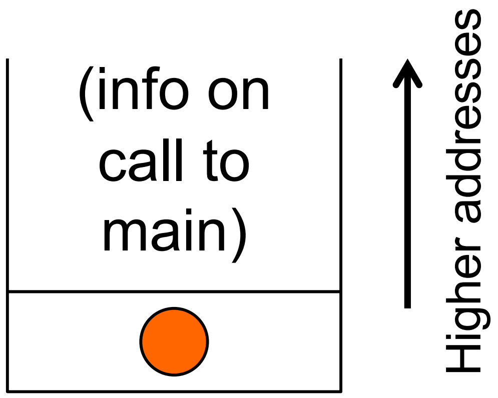
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



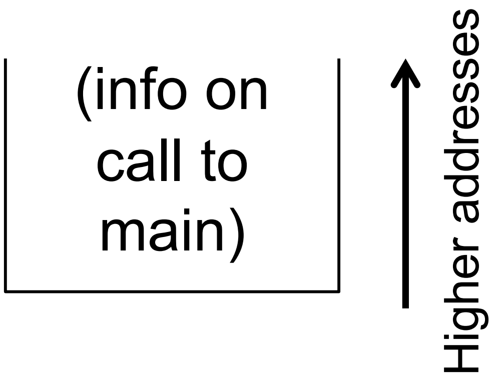
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



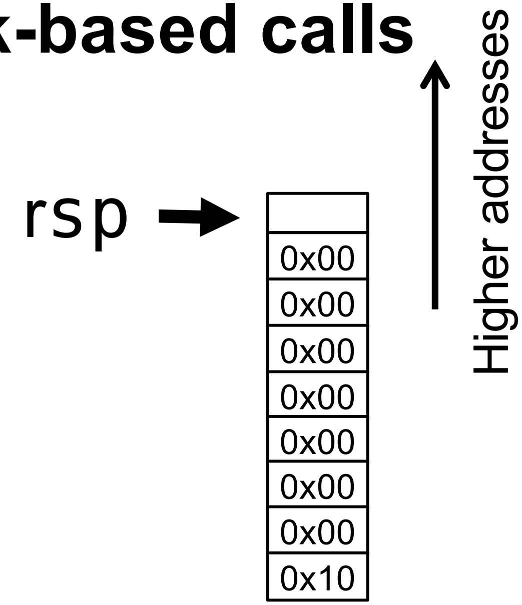
Example

```c
#include <stdio.h>

void foo(void) {
  printf("Executing foo\n");
  printf("Returning from foo\n");
} // End foo

void bar(void) {
  printf("Executing bar\n");
  printf("Calling foo from bar\n");
  foo();
  printf("Returning from bar\n");
} // End bar

int main( int argc, char** argv) {
  printf("Calling foo from main\n");
  foo();
  printf("Calling bar from main\n");
  bar();
  printf("Returning from main\n");
} // EndMain
```



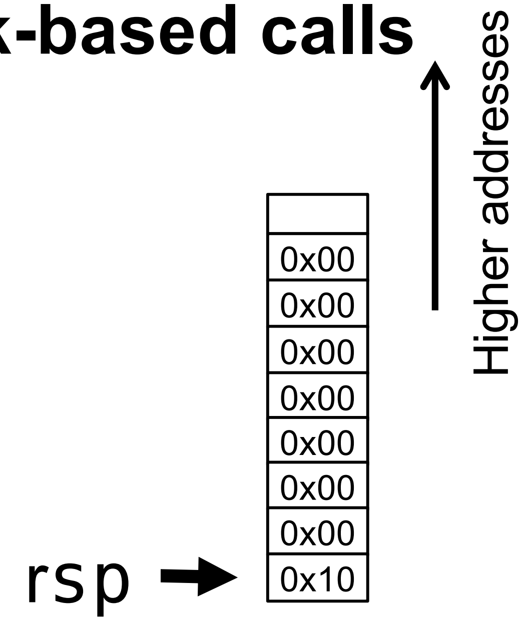
Stack in x86 Assembly

- X86-64: direct hardware support for **stack-based calls**
  - → case of most high-level processors
- Special register: **`rsp`**
  - → Points to top of the stack
- Two operations
  - → `push` **R/M/V**, for instance `push rax`
    - pushes operand (`rax` here) onto the stack onto the stack
    - `rsp` ← `rsp` – 8 (moves to lower addresses)



Stack in x86-64 Assembly

- X86-64: direct hardware support for **stack-based calls**
  - ➔ case of most high-level processors
- Special register: **rsp**
  - ➔ Points to top of the stack
- Two operations
  - ➔ `push` **R/M/V**, for instance `push rax`
    - pushes operand (`rax` here) onto the stack onto the stack
    - `rsp` ← `rsp` – 8 (moves to lower addresses)
  - ➔ `pop` **R/M**, for instance `pop rax`
    - pops **8 bytes** at top of stack, move them to operand
    - `rsp` ← `rsp` + 8 (moves to higher addresses)



rax

Calls in x86-64

Two special op to call functions

- `call` **`some_address`**
  - ➔ Pushes address of **next instruction** on stack
  - ➔ (→ `rsp` moves towards lower addresses, stack grows)
  - ➔ Jumps to **`some_address`**
- `ret`
  - ➔ Retrieves (pops) **return address** from stack
  - ➔ (→ moves `rsp` towards higher addresses, stack shrinks)
  - ➔ Jumps to **return address**

Example: calling foo

```
  call foo
  ...

foo:
  mov       rax, 1        ; system call for write
  mov       rdi, 1        ; file handle 1 is stdout
  mov       rsi, message  ; address of string to output
  mov       rdx, msgLen   ; number of bytes
  syscall                 ; invoke operating system to do the write
  ret
```

- The code after `foo:` is executed every time `call foo` is executed
- Note how `foo` is a label
  - ➔ Replaced by an address (an offset in the code segment)

Potential Problem

```
_start:
foo:
  mov       rax, 1         ; system call for write
  mov       rdi, 1         ; file handle 1 is stdout
  mov       rsi, message   ; address of string to output
  mov       rdx, msgLen    ; number of bytes
  syscall                  ; invoke operating system to do the write
  ret

  call foo
  mov       rax, 60        ; system call for exit
  mov       rdi, 0         ; exit code 0
  syscall                  ; invoke operating system to exit

GLOBAL    _start
```

- Will the above code work? Why?

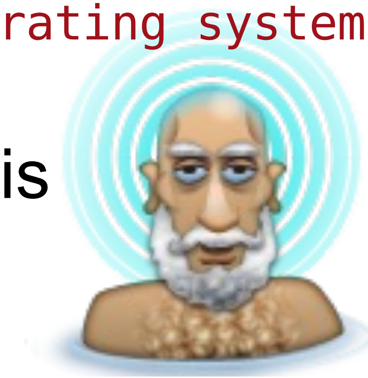
And a solution

- Either implement foo after the exit to OS or ...

entry point

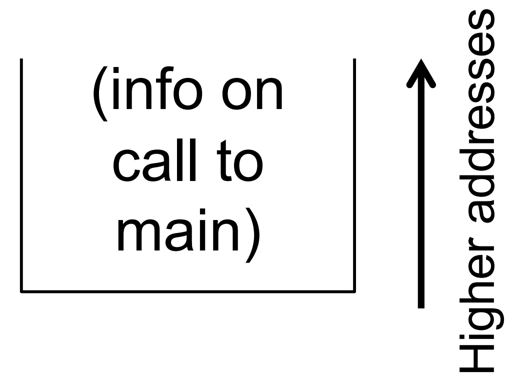
```
foo:
  mov        rax, 1         ; system call for write
  mov        rdi, 1         ; file handle 1 is stdout
  mov        rsi, message ; address of string to output
  mov        rdx, msgLen    ; number of bytes
  syscall                   ; invoke OS to do the write
  ret

_start:
  call foo
  mov        rax, 60        ; system call for exit
  mov        rdi, 0         ; exit code 0
  syscall                   ; invoke operating system to exit

GLOBAL    _start
```

Why should this now work?



Local Variables

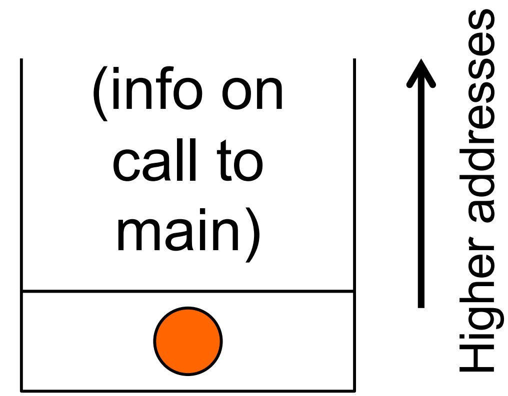
- Variables declared in procedure → allocated on stack




```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```

Local Variables

- Variables declared in procedure → allocated on stack

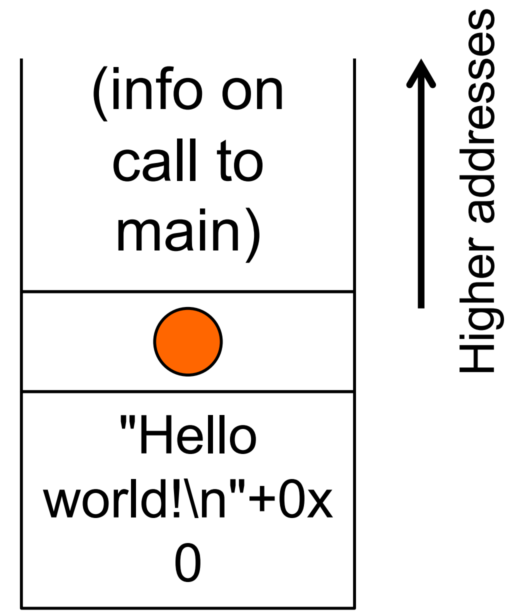



```c
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```

Local Variables

- Variables declared in procedure → allocated on stack

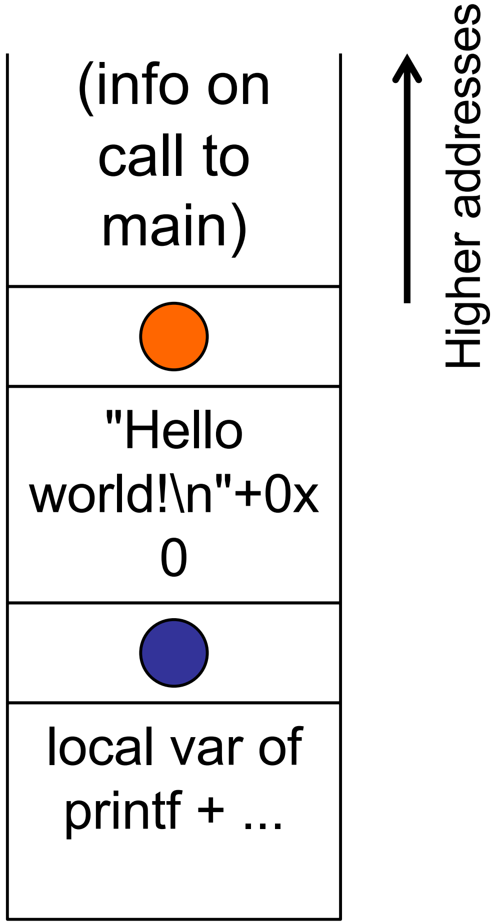
```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Local Variables

- Variables declared in procedure → allocated on stack

```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



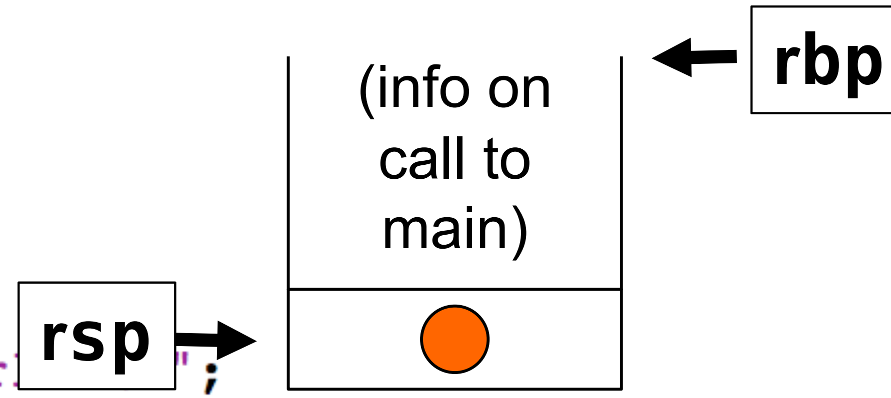
Base Pointer & Stack Frame

- Each function call: its own area on stack
  - ➔ Known as a **"stack frame"**
- Local variables create a problem for **`rsp`**
  - ➔ Additional data on top of current return address
  - ➔ **`rsp`** (top of stack) not always pointing to return address
- We need a second register = **rbp "base pointer"**
  - ➔ A.k.a frame pointer
  - ➔ Return address: just before where **rbp** points on the stack
  - ➔ Used to access local variables + debugging
- **rbp** needs to be saved after `call` / restored before `ret`
  - ➔ Responsibility of callee function

Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

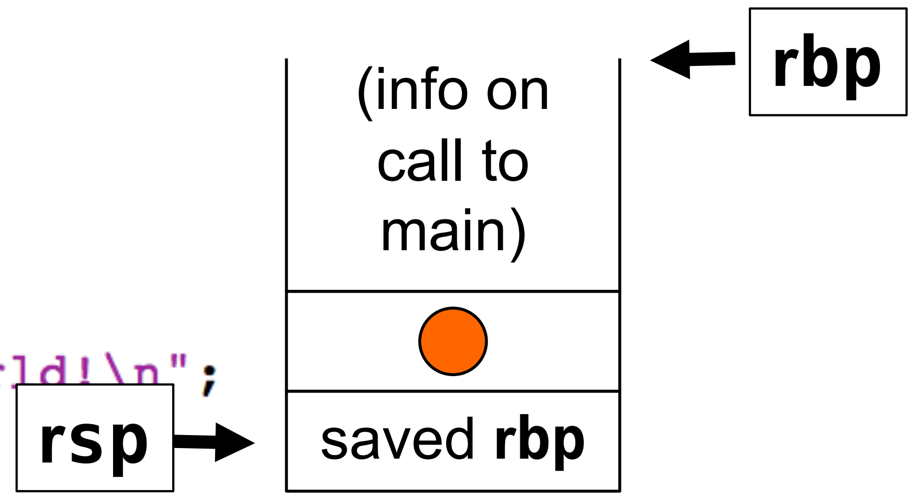
```
void foo(void) {
  char my_string[] = "Hello Wor";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

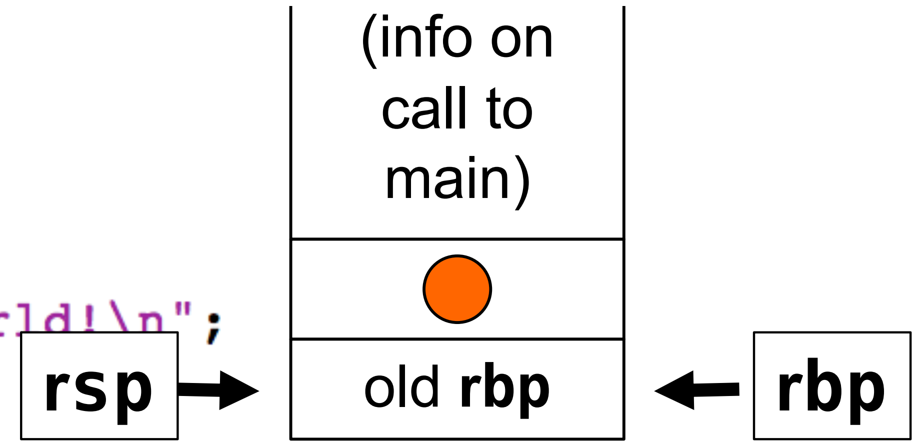
```c
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

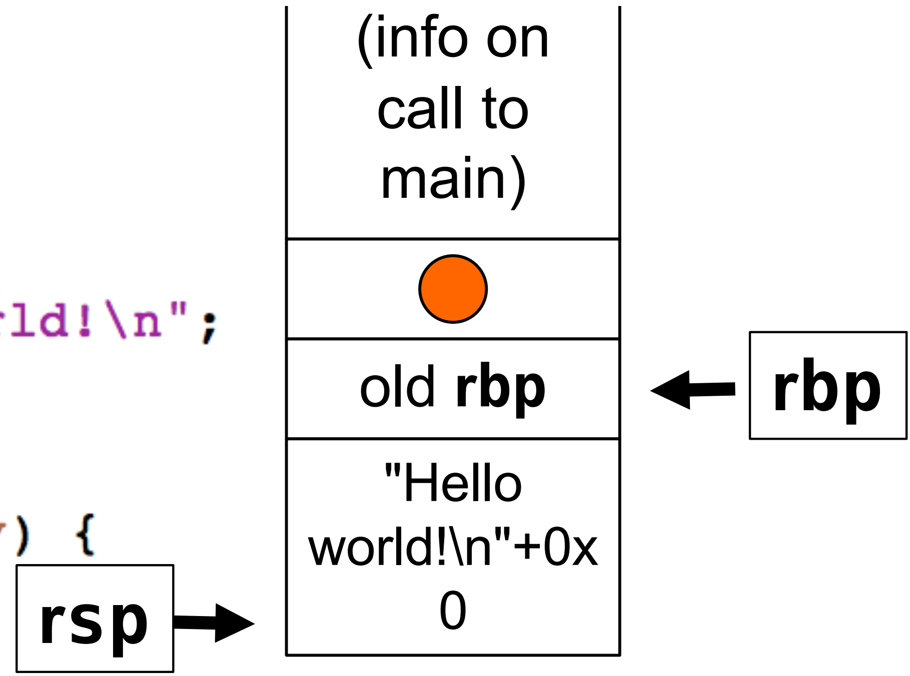
```
void foo(void) {
   char my_string[] = "Hello World!\n";
   printf(my_string);
} // End foo

int main( int argc, char** argv) {
   foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

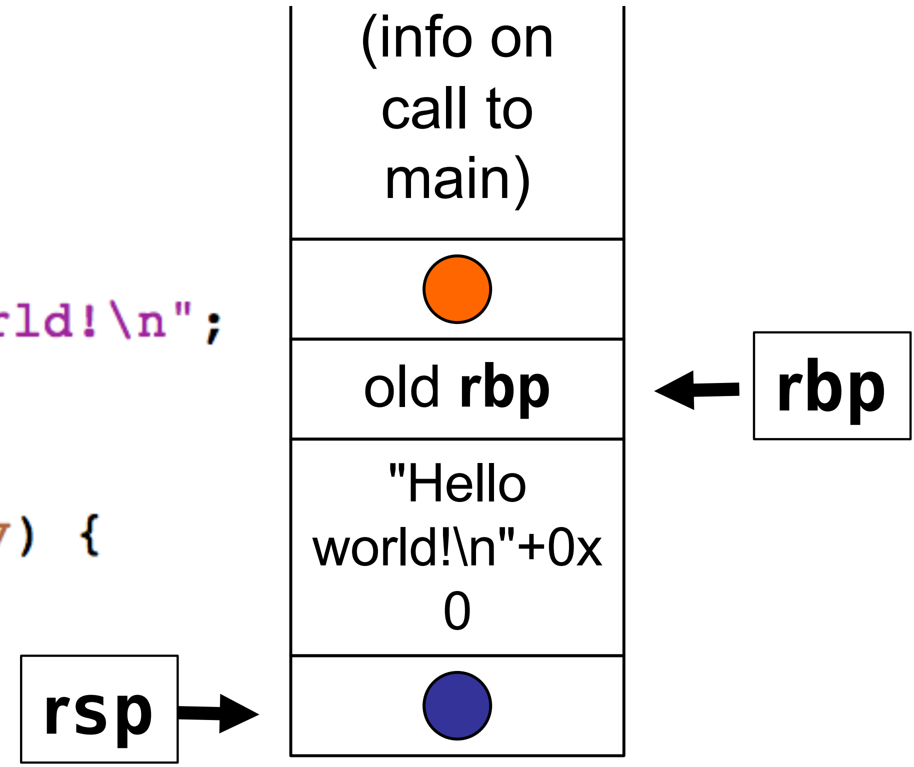
```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

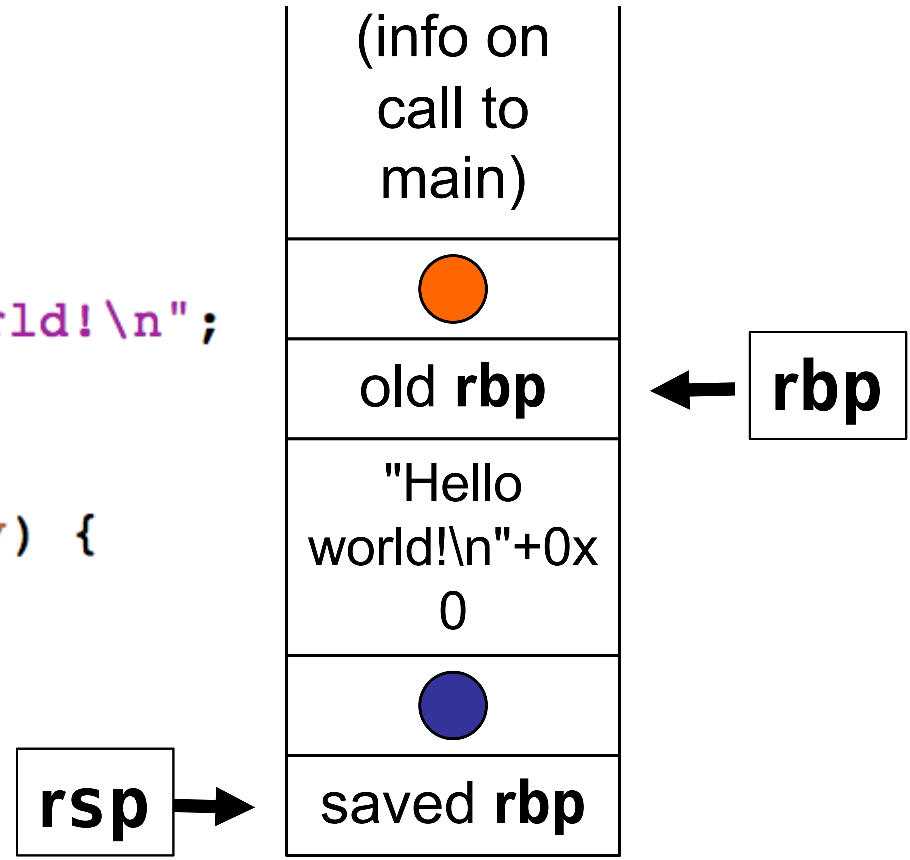
```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

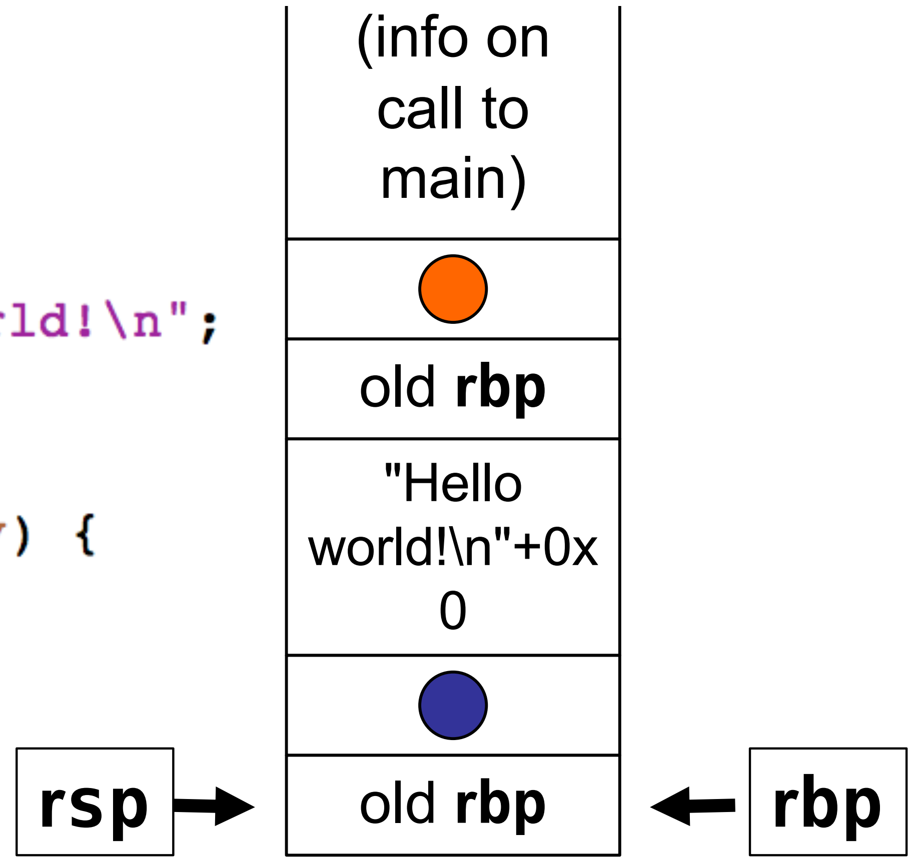
```c
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

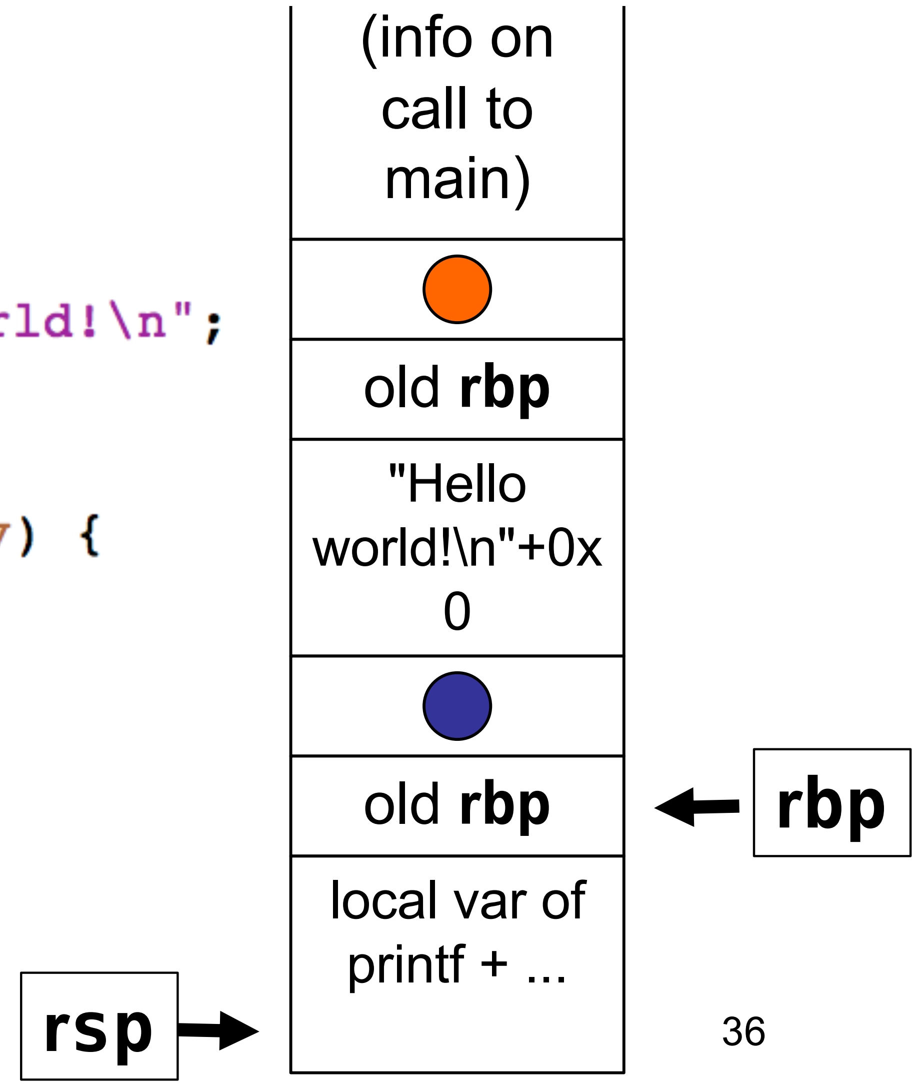
```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Base Pointer & Stack Frame

- Variables declared in procedure → allocated on stack

```
void foo(void) {
  char my_string[] = "Hello World!\n";
  printf(my_string);
} // End foo

int main( int argc, char** argv) {
  foo();
} // EndMain
```



Saving / Restoring rbp

- Uses `push` & `pop`

- Function **prologue** (just after `call`)
  - ```
    push    rbp
    mov     rbp,rsp
    sub     rsp, <space for local variables> ;
    ```
- Function **epilogue** (just before `ret`)
  - ```
    mov     rsp,rbp
    pop     rbp                                r
    ret
    ```
- On x86 > 8186 compound op : `enter` / `leave`
  - `enter` not used, too slow

Saving Registers

- When issuing a `call foo` code jumps to location 'foo'
  - ➔ 'foo' likely to use **registers**
  - ➔ **any register value** before `call` **might be overwritten**
  - ➔ need to **save registers** that must be preserved
  - ➔ Using `push` and `pop` (e.g. `push rax`)

- Who should save / restore what
  - ➔ The work can be done on the caller's or the callee's side
  - ➔ Or shared
  - ➔ Who does what is defined by **calling conventions**
    (also covers order of parameters on stack, etc., see later)

Passing Parameters

- 2 mechanisms to transmit parameters ("*transportation*")
  - Through some **registers** (default), or
  - Through the **stack** (when not enough regs)

- 2 kinds of parameters ("*change propagation*")
  - By **value** (changes not propagated back to caller)
    - Actual value directly passed in register or on stack
  - By **reference** (aka address) (change propagated back)
    - Register or stack contains an address
    - Actual value obtained after **indirection** (means 2 indirections if stack frame is used)
- 2 x 2 = 4 possibilities

By Value vs. By Reference

- Parameter passed by **value**
  - ➔ Value of parameter **copied** into reg **or** onto the stack
  - ➔ Change to parameter not visible on return
- Parameter passed by **reference**
  - ➔ Needs to pass an address (= a pointer in C)
  - ➔ Access to value uses indirection (“dereference”)
  - ➔ Changes to parameter will be visible on return
- Note: in Java
  - ➔ Everything is a reference, except basic types (values)

What kind of passing is this?

```
SECTION   .data
y:  db  65
...

SECTION   .text

foo:
  add  ax,2      ; x += 2
  ret

_start:
  mov  ax,[y]
  call foo
```

What kind of passing is this?

```
SECTION   .data
y:  db  65
...

SECTION   .text

foo:
  add  [rax], BYTE 2  ; x += 2
  ret

_start:
  mov  rax,y
  call foo
```

Returning Value

- Most frequent approach
  - ➔ using registers (**`rax`**, etc.)
- Alternative: on the stack
  - ➔ Reserve some space for the return value before calling

Parameters in C (Linux)

- Optimized approach : Using specific registers
  - ➔ Input in : rdi, rsi, rdx, rcx, r8, r9
  - ➔ Output in : rax (+ rdx if needed)
  - ➔ Stack used if parameters / return values do not fit
- Precise rules on how this works
  - ➔ Registers need to be saved / restored by callee or caller
  - ➔ Depends on language / ISA
  - ➔ Linux : specified in **ABI : Application Binary Interface**
    - rbx, rsp, rbp = callee-saved registers
    - The rest : “scratch”, might be overwritten (“clobbered”)

Passing Parameters by Register

- Example: Decompiling foo() from first example

```
00000000004004e4 <foo>:

void foo(void) {
  4004e4:       55                      push   rbp
  4004e5:       48 89 e5                mov    rbp,rsp
  printf("Executing foo\n");
  4004e8:       bf 4c 06 40 00          mov    edi,0x40064c
  4004ed:       e8 ee fe ff ff          call   4003e0 <puts@plt>
  printf("Returning from foo\n");
  4004f2:       bf 5a 06 40 00          mov    edi,0x40065a
  4004f7:       e8 e4 fe ff ff          call   4003e0 <puts@plt>
} // End foo
  4004fc:       c9                      leave
  4004fd:       c3                      ret
```

➔ edi is used to pass the @ of the string to be printed

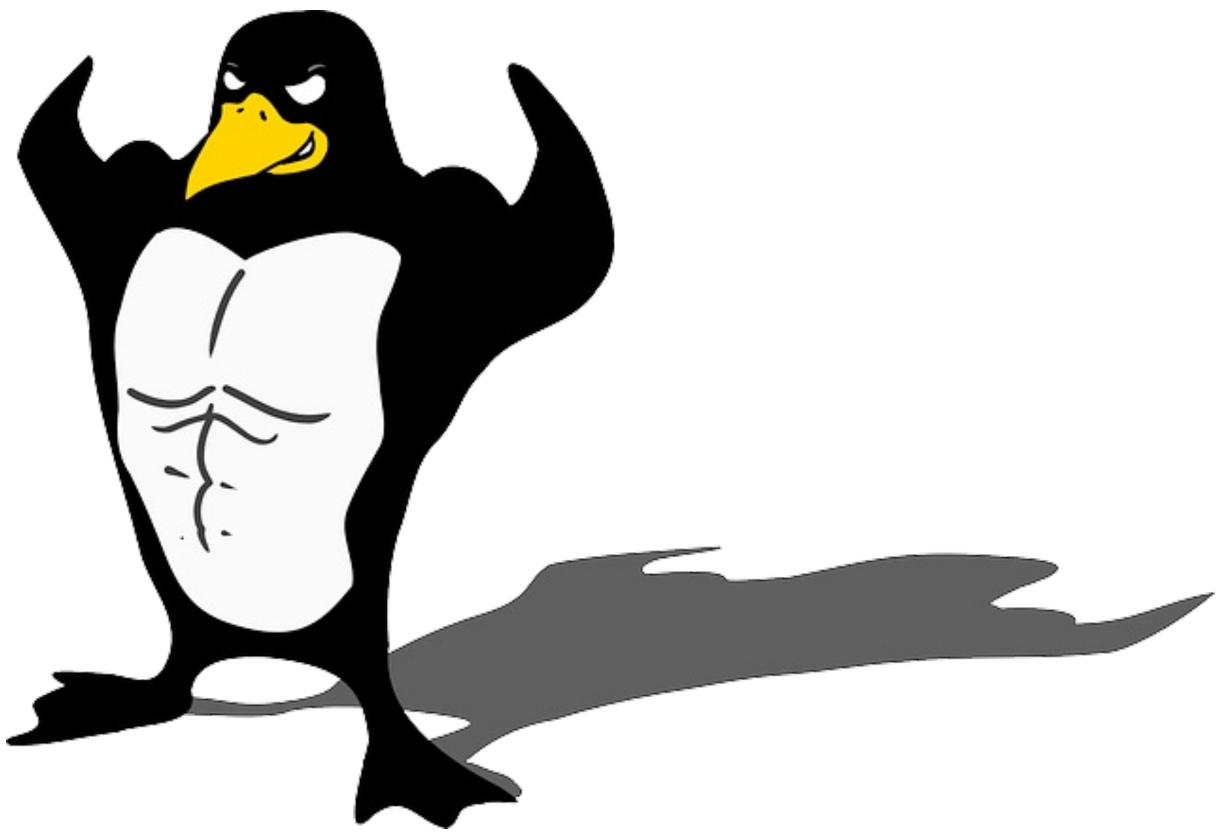
Summary

As the end of this session you should:

- Be able to explain the role of the stack
- Understand the role of `call` and `ret`
- Be able to understand and manipulate a stack with `rsp`, `pop`, `push`
- Be able to describe and analyse the structure of stack frames, and the role of `rbp`
- Explain how local variables are allocated in a frame
- Be able to explain how parameters are passed and returned (stack vs. registers, value vs. references)

Bonus Material

- Not exam material



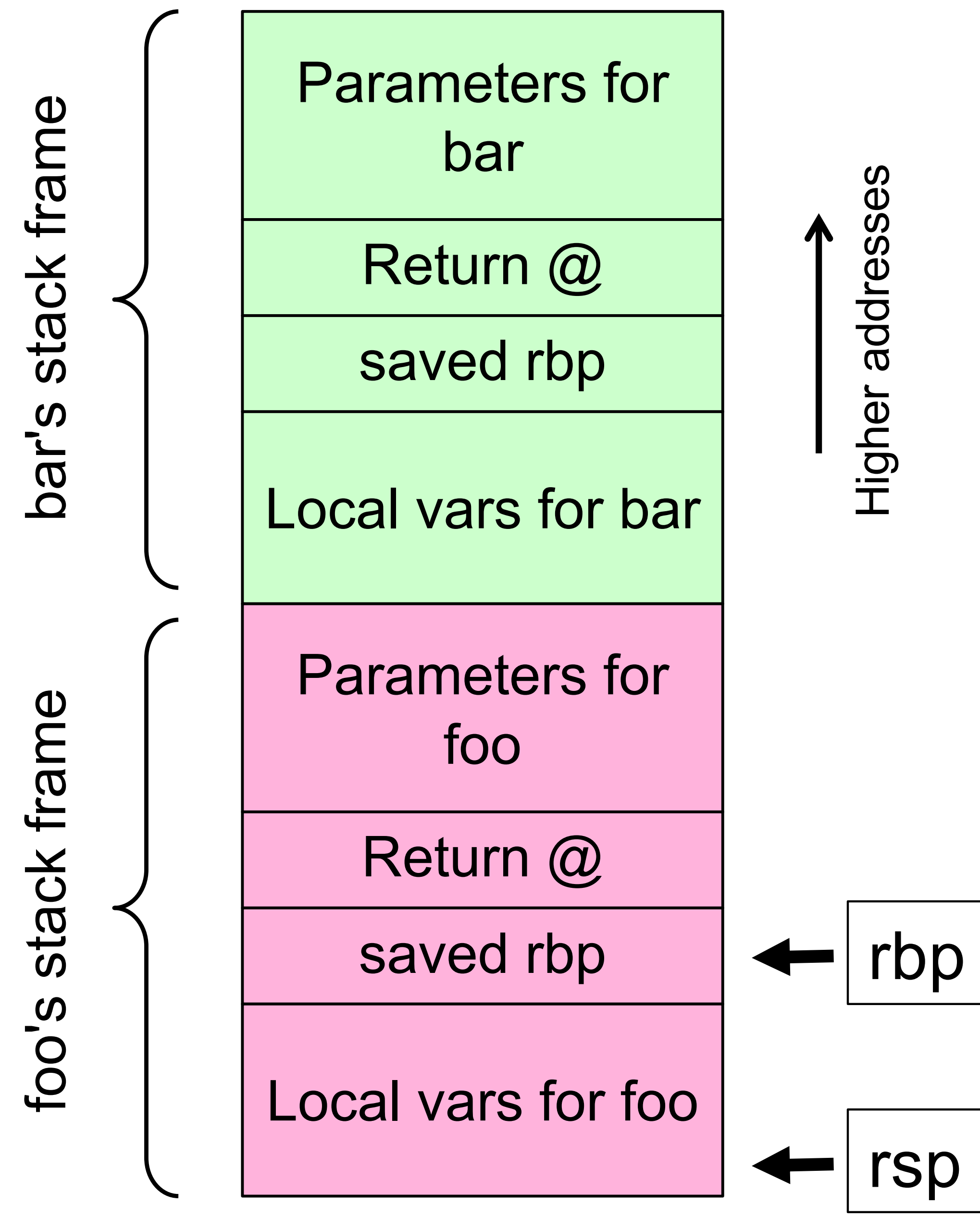
Using Stack to Pass Parameters

- Parameters pushed before the **call** (w/ **push**)
  - ➔ Parameter area liberated on return
    - Either with **ret <size to liberate>** (callee clean up)
    - Or by caller (caller clean up, C)

- Parameters and local values **accessed** using rbp + offset
  - ➔ "based" or "indexed" addressing mode: rbp[offset]
  - ➔ Positive offsets: parameters
  - ➔ Negative offsets: local variable

Stack Layout, Stack-based Passing

Stack growing from higher to lower addresses



Parameters passed on stack, no return value

Who is calling who?

Returning Value On the Stack

- Stack Layout