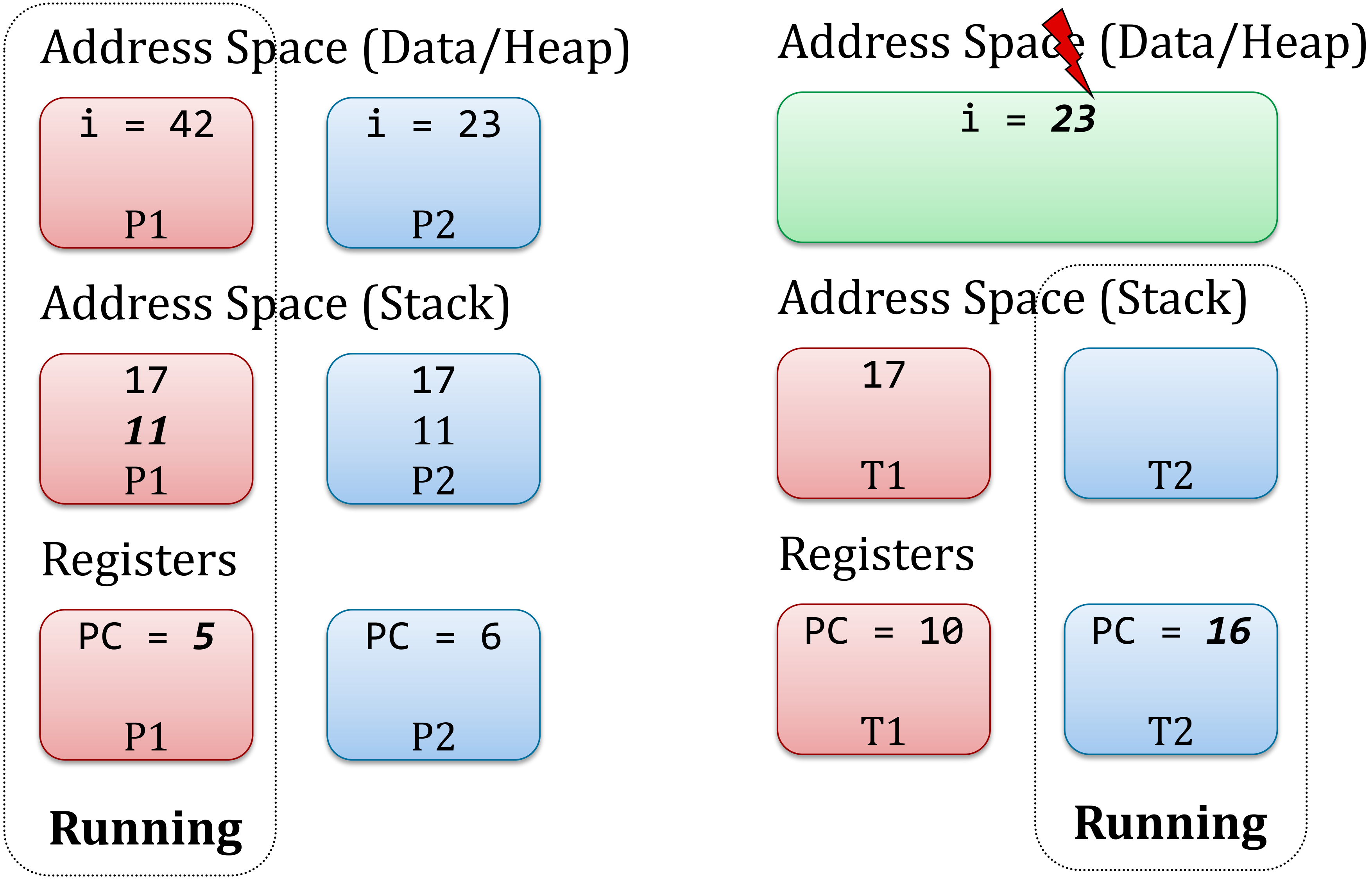
EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical &
Computer Engineering

Boston University

Process vs. Thread



Address Space (Data/Heap)

i = 42
P1

i = 23
P2

Address Space (Stack)

17
11
P1

17
11
P2

Registers

PC = ***5***
P1

PC = 6
P2

Running

Address Space (Data/Heap)

i = ***23***

Address Space (Stack)

17
T1

T2

Registers

PC = 10
T1

PC = ***16***
T2

Running

Thread Primitives (e.g., pthread_create)

```
$ man pthread_create
PTHREAD_CREATE(3)                Linux Programmer's Manual

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

Each thread has a separate stack!

The Stack …

Procedures

```
int f(int x) {return   x + 1; }
int g(int x) {return    f(x); }
int h(int x) {return f(x *2); }
```

Procedures (functions) are intrinsically linked to the stack

- Provides space for local variables
- Records where to return to
- Used to pass arguments (sometimes)

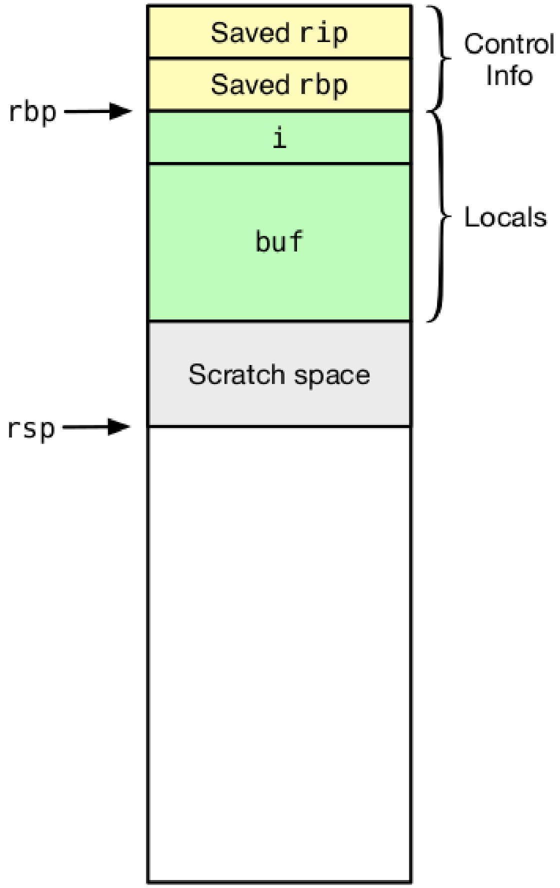
Implemented using stack frames

- Also known as activation records

Procedures: Calling and Returning

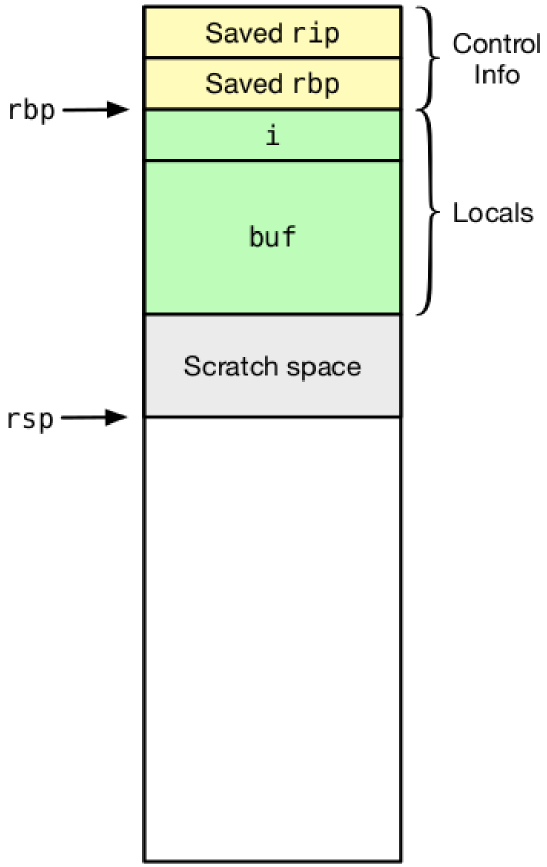
Instruction	Effect	Description
`call x`	rsp ← rsp - 8	Decrement rsp by 8
	Mem(rsp) ← Succ(rip)	Store successor
	rip ← Addr(x)	Jump to address
`ret`	rip ← Mem(rsp)	Pop successor into rip
	rsp ← rsp + 8	Increment rsp by 8

Stack Frame



```c
int auth(const char* user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof(buf));
    buf[sizeof(buf) - 1] = '\0';
    for (i = 0; i < sizeof(buf); i++)
        buf[i] ^= 0xe5;
    return !memcmp(buf, "secret", 6);
}
```

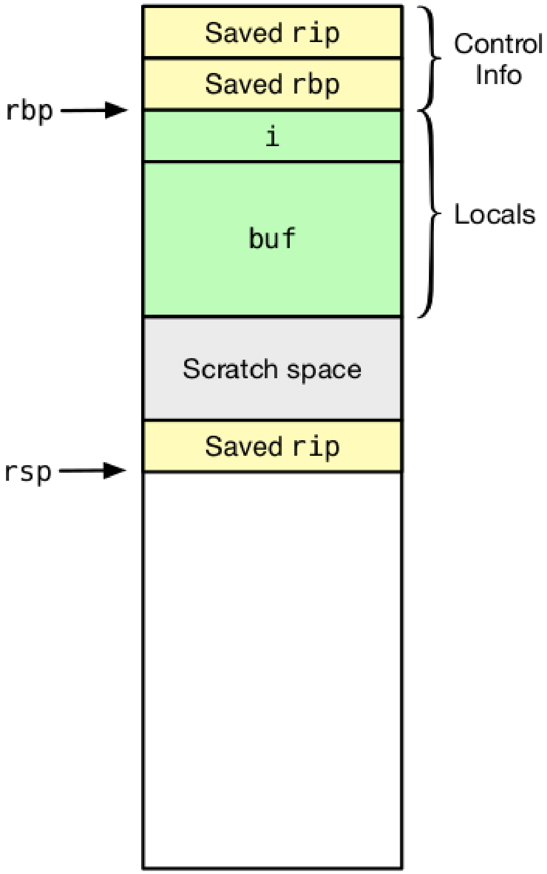
Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp−0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

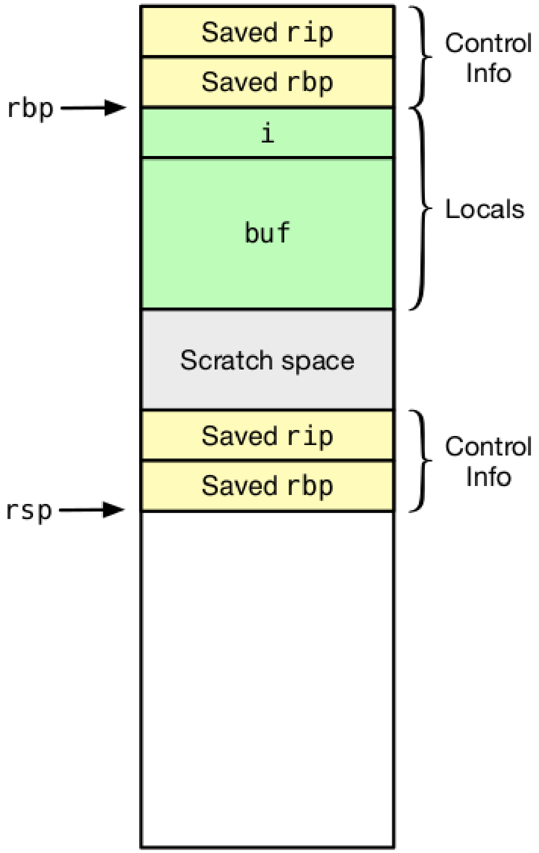
Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

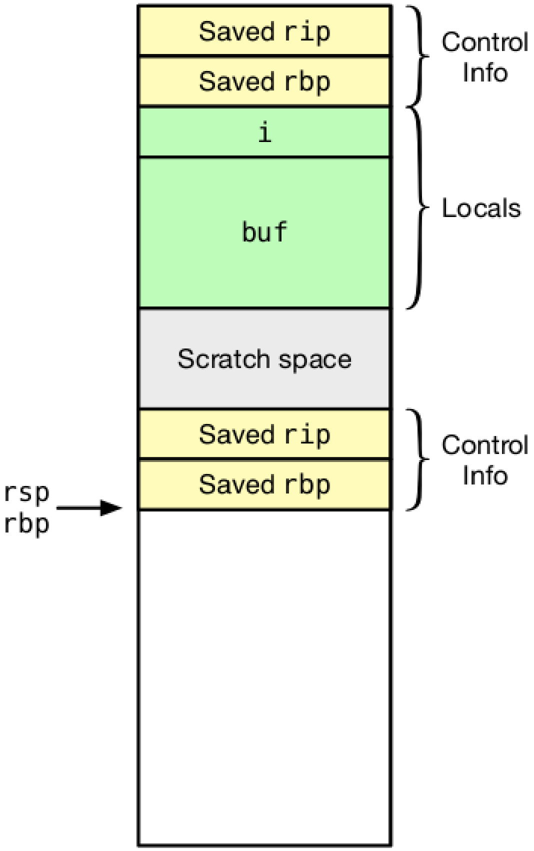
Stack Frame



```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

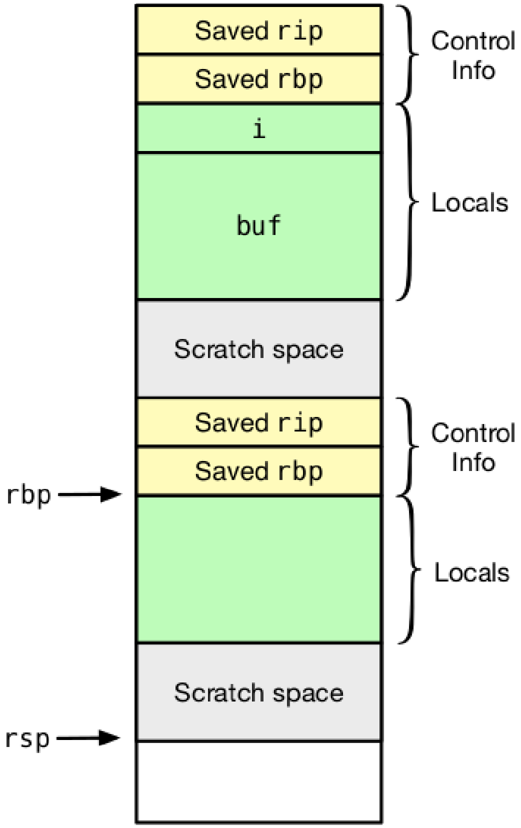
Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

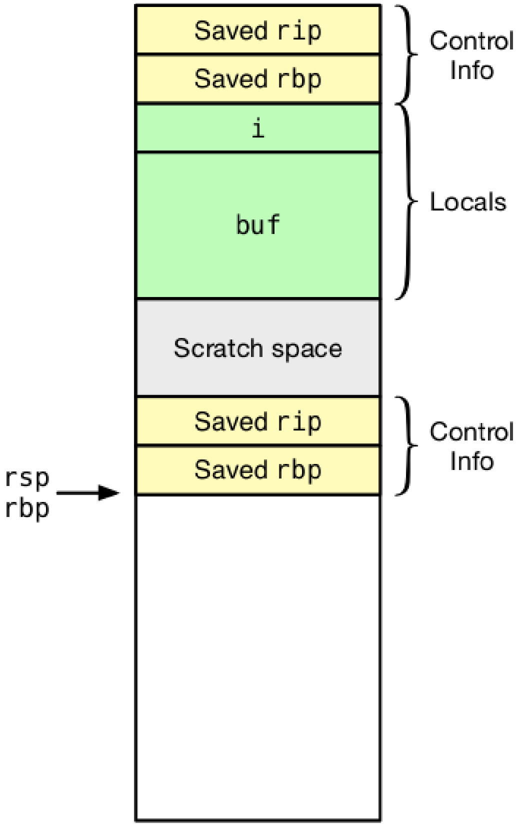
Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

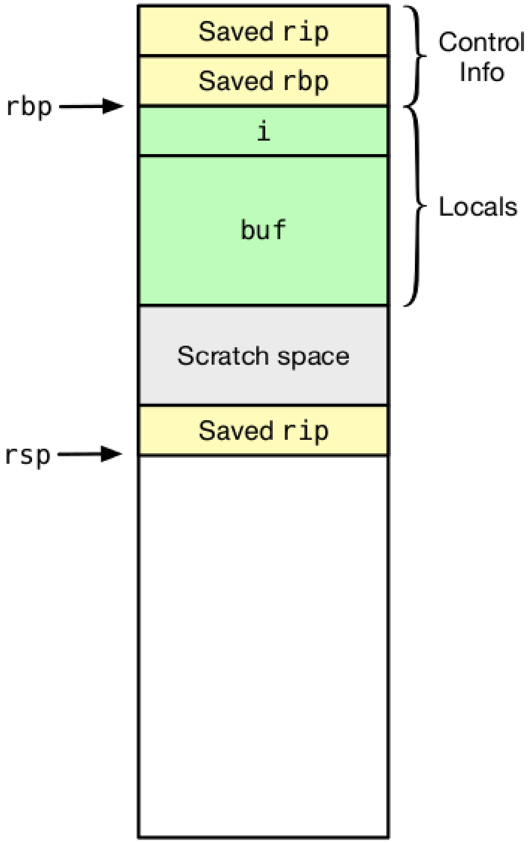
Stack Frame



```
auth:
      ...
      mov rdi, rax
      call strncpy
      mov byte [rbp-0x11], 0x00
      ...

strncpy:
      push rbp
      mov rbp, rsp
      sub rsp, 0x30
      ...
      add rsp, 0x30
      pop rbp
      ret
```

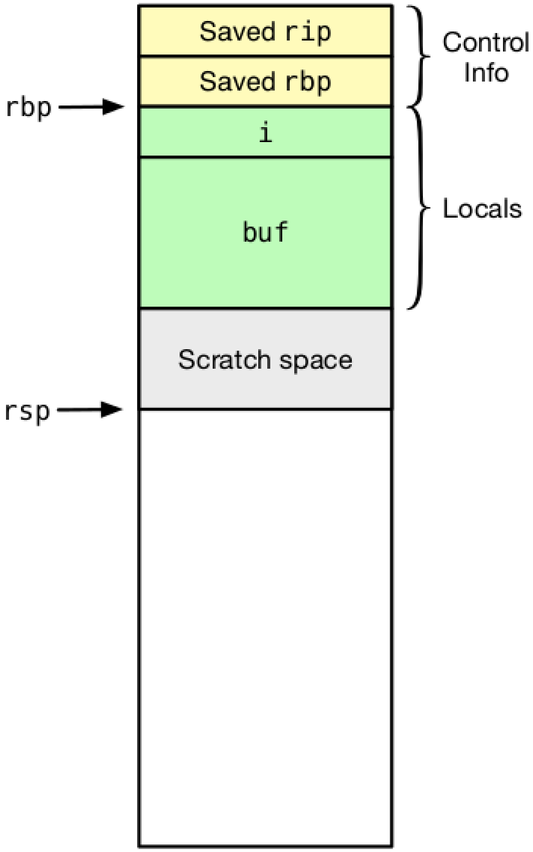
Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

Stack Frame



```
auth:
    ...
    mov rdi, rax
    call strncpy
    mov byte [rbp-0x11], 0x00
    ...

strncpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x30
    ...
    add rsp, 0x30
    pop rbp
    ret
```

Procedure Arguments

Standards (*calling conventions*) exist for argument passing

- Specify where arguments are passed (registers, stack)
- Specify the caller and callee's responsibilities
  - Who deallocates argument space on the stack?
  - Which registers can be clobbered, and who must save them?

Why do we need standards?

- There are many ways to pass arguments
- How would code compiled by different developers and toolchains interoperate?

Calling Conventions

We often speak of *callers* and *callees*

- Caller: Code that invokes a procedure
- Callee: Procedure invoked by another function

Conventions must specify how registers must be dealt with

- Could always save them, but that is inefficient (why?)
- Usually, some registers can be overwritten (clobbered), others cannot
- Registers that can be clobbered: *caller* saved
- Registers that must not be clobbered: *callee* saved

SysV AMD64 ABI

- **x86_64 calling convention used on Linux, Solaris, FreeBSD, Mac OS X**
  - This is what you'll see most often in this course
- **First six arguments passed in registers**
  - rdi, rsi, rdx, rcx, r8, r9
    - Except syscalls, rcx → r10
  - Additional arguments spill to stack
- **Return value in rax**

cdecl

We’ve been concentrating on x86_64, but cdecl is important to know

- Linux 32 bit calling convention

Arguments

- Passed on the stack
- Pushed *right to left* (reverse order)

Registers

- eax, edx, ecx are *caller* saved
- Remainder are *callee* saved

Return value in eax

Caller deallocates arguments on stack after return

SysV AMD64 ABI Example

```
int auth( const char * user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof (buf));

auth:
    push rbp                    ; save previous frame pointer
    mov rbp, rsp                ; set new frame pointer
    sub rsp, 0x30               ; allocate space for locals (i, buf)
    movabs rdx, 0x10            ; move sizeof(buf) to rdx
    lea rax, [rbp-0x20]         ; get the address of buf on the stack
    mov qword [rbp-0x08], rdi   ; move user pointer into stack
    mov rsi, qword [rbp-0x08]   ; move user pointer back into rsi
    mov rdi, rax                ; move buf into rdi
    call strncpy                ; call strncpy(rdi, rsi, rdx)
    ...
```

Questions?