Last Time
- Processes: What, Why, (How Created)
- Stack Frames (kind of)

Today
- More Stack Frames
- Kernel/Userspace
- System Calls
- Traps
- Fork Again?

Main's stack frame:

```c
int main(void)
{
    uint64_t x = 0;
    uint64_t arg = 8;
    x = f(&arg);
    printf("x: %lu\n", x);
    printf("dereference q: %lu\n", *q);
    return 0;
}

uint64_t f(uint64_t* ptr)
{
    uint64_t x = 0;
    x = g(*ptr);
    return x + 1;
}
```
1. Support usual function semantics
   \[ \longrightarrow \text{Pass arguments to function} \]
   \[ \longrightarrow \text{Return value} \]
   \[ \frac{1}{2} \longrightarrow \text{Return control flow} \]
   \[ \longrightarrow \text{Variable scope} \]
   \[ \longrightarrow \text{Stack frames} \]

### Assembly Code

```
15  movq  $0, -8(%rbp)
16  movq  $8, -16(%rbp)
17  leaq  -16(%rbp), %rdi
18  call  f
19  movq  %rax, -8(%rbp)
20  pushq %rbp
21  movq  %rsp, %rbp
22  subq  $32, %rsp
23  movq  %rdi, -24(%rbp)
24  movq  $0, -8(%rbp)
25  ...
26  ...
27  ...
28  ...
29  ...
30  ...
31  ...
32  ...
33  ...
34  ...
35  ...
36  ...
37  ...
38  ...
39  ...
40  ...
41  ...
42  ...
43  ...
44  ...
45  ...
46  ...
47  movq  %r10, %rax
48  movq  %rbp, %rsp
49  popq  %rbp
50  ret
```

### Stack-related Instructions
- \%RSP = 0x-ffe8
- \%RDI = 0x-ffe8
- PROLOG: CREATE f's STACK FRAME
- EPILOG: SET return value, DESTROY stack frame, RETURN control to main

### Diagram

- Stack frames
- Memory layout
- Function call
- Variable scope
- Return address on stack
Calling Convention

- How to pass arguments
  rdi, rsi, rdx, rcx, r8, r9, on stack

- How to return values
  rax

- Register values on return
  Caller saves/volatile: includes rax
  Callee saves: includes rbp, rsp

Problem with Pointers to Stack Variables

```c
38 uint64_t g(uint64_t a)
39 {
40   uint64_t x = 2*a;
41   q = &x;
42   return x;
43 }
```

More Generally
- Pointers are just like any other value
Pointers are just like any other - just some bit pattern

- Interpretation determined by how it is used
- A slight problem when calling functions provided by the kernel

- `fork()`
- `open(char*, int)`
- `read(int, char*, size_t)`
- `write(int, char*, size_t)`

Pointers to memory owned by the application. Must be read by the kernel.

```c
const char *f = "LO3.txt";
int fd = open(f, O_RDWR);
```
User $\rightarrow$ Kernel Transitions

1. Support usual function semantics
   - Pass arguments
   - Return value
   - Return control flow
   - Variable scope
   - Kernel memory protection

Traps

- Mechanisms for kernel $\leftrightarrow$ userspace transitions
  - Syscall
  - Exceptions
- Interrupts

Creating a process
- fork
- execve