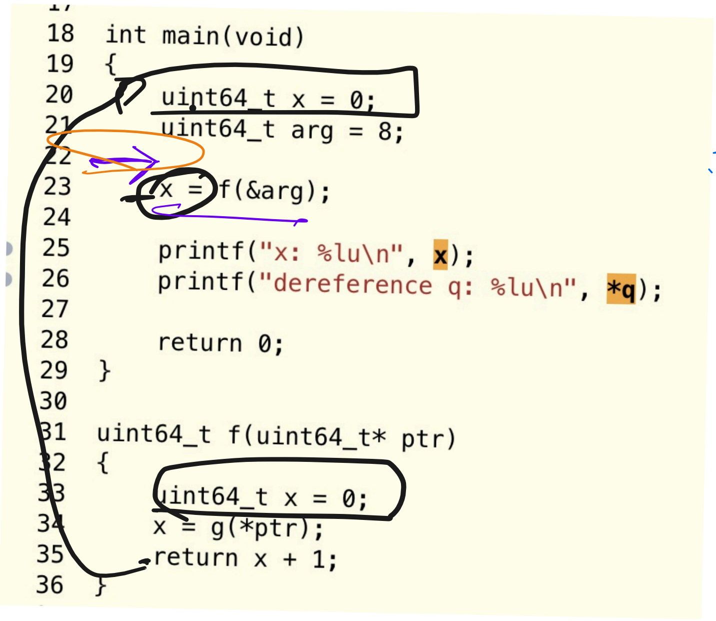
# Last Time

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

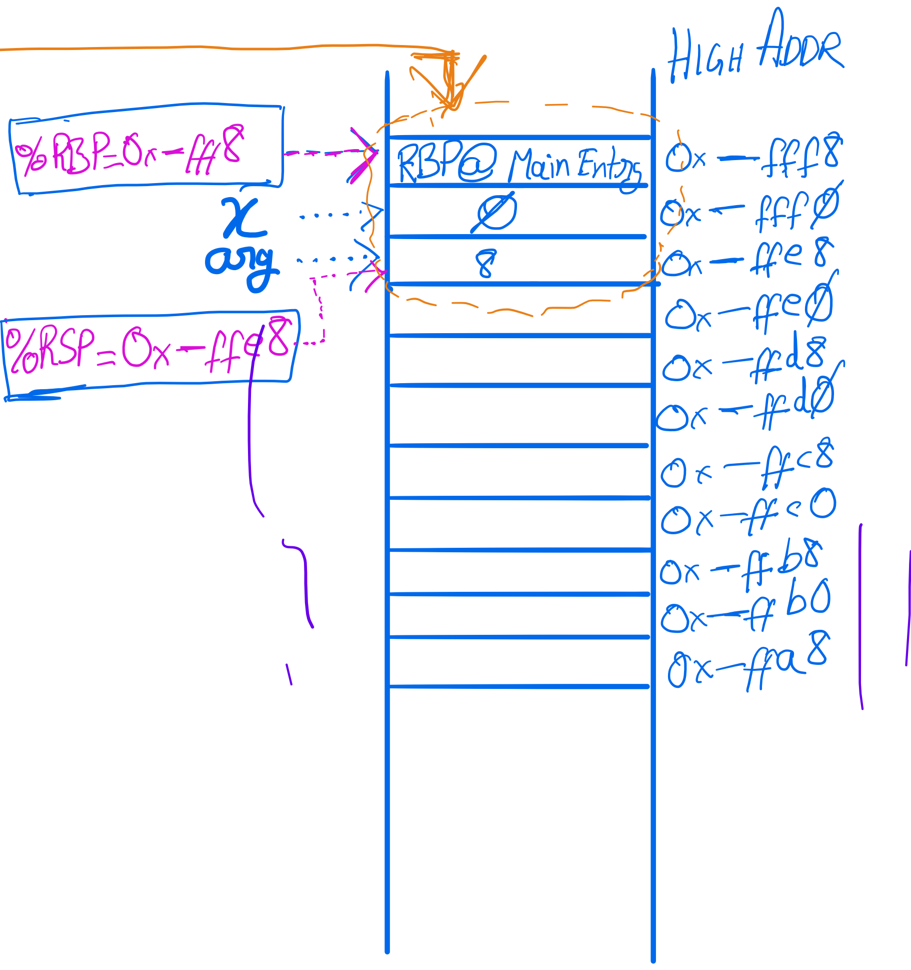
# Today

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

main's stack frame



```
 17
 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;
}
```



# Goals

① Support usual function semantics

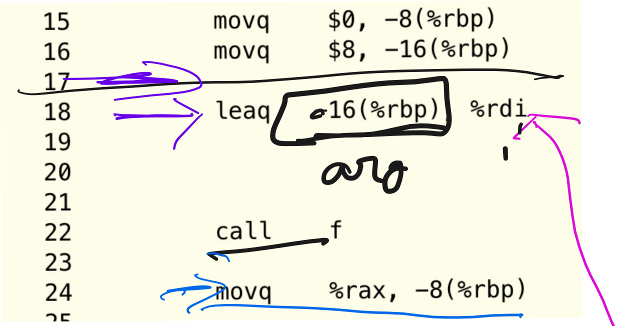
↳ Pass arguments } Calling convention

→ Return value }

1/2 → Return control flow } Return address on stack

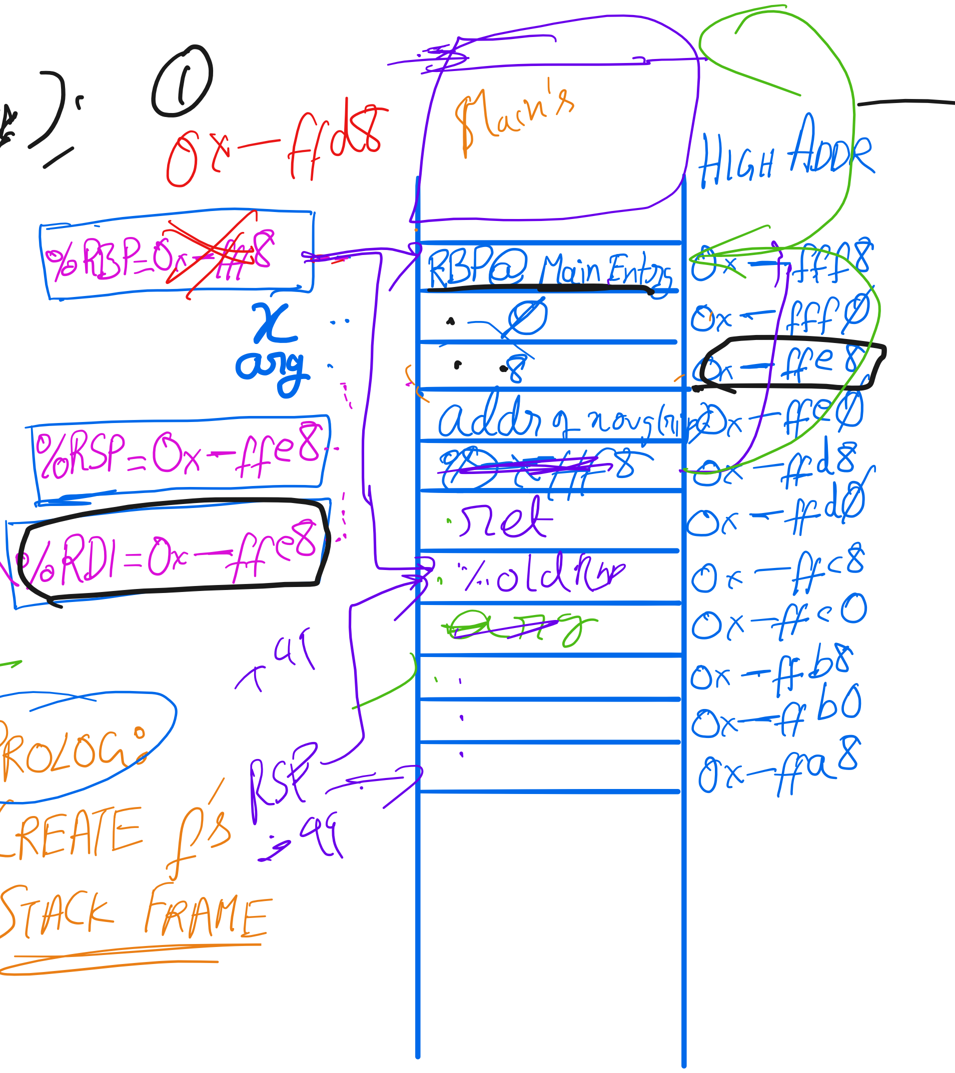
→ Variable scope } Stack frames

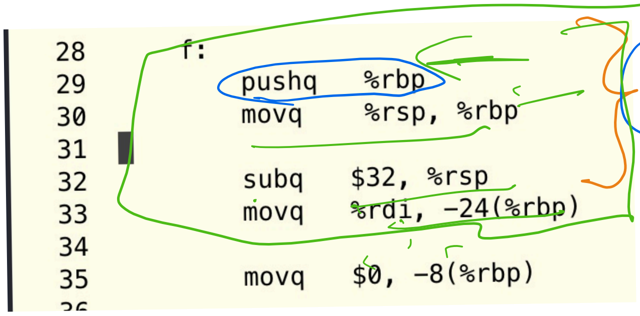
x = f(&arg); ①



```
        movq    $0, -8(%rbp)
        movq    $8, -16(%rbp)

        leaq    -16(%rbp), %rdi



        call    f

        movq    %rax, -8(%rbp)
```

arg



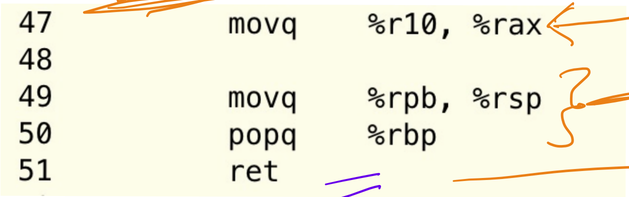


```
f:
        pushq   %rbp
        movq    %rsp, %rbp

        subq    $32, %rsp
        movq    %rdi, -24(%rbp)

        movq    $0, -8(%rbp)
```

PROLOG: CREATE f's STACK FRAME

o o o

EPILOG



```
        movq    %r10, %rax

        movq    %rpb, %rsp
        popq    %rbp
        ret
```

Set return value

Destroy stack frame

Return control to main

≡ popq %rip

# Calling Convention

- How to pass arguments

  rdi, rsi, rdx, rcx, r8, r9, on stack

- How to return values

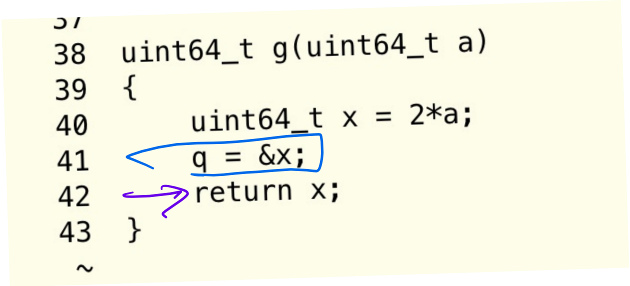
  rax

- Register values on return

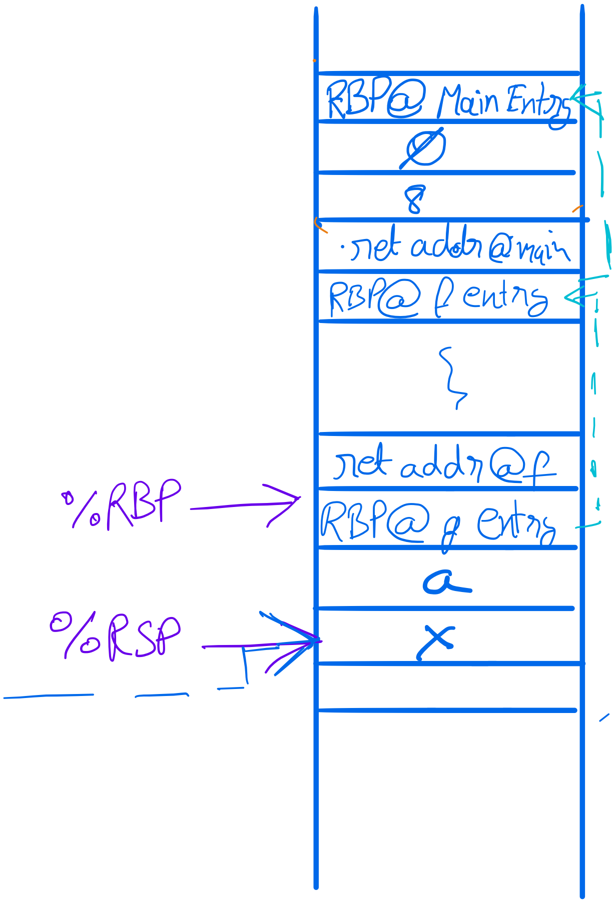
  Caller saves/volatile: includes rax

  Callee saves: includes rbp, rsp

## Problem With Pointers To Stack Variables



```
uint64_t g(uint64_t a)
{
      uint64_t x = 2*a;
      q = &x;
      return x;
}
~
```



## More Generally

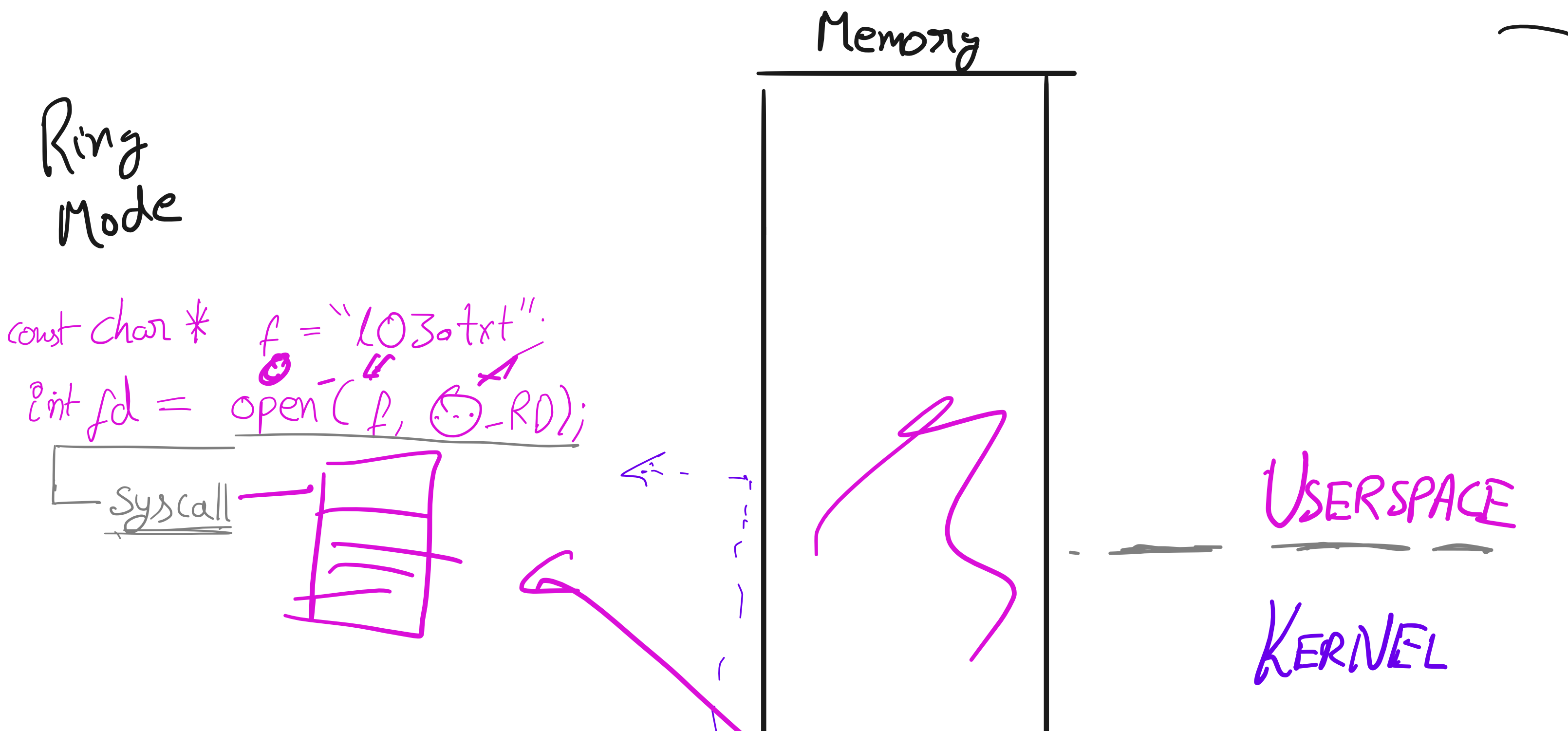
- Pointers Are Just Like Any Other Value

POINTERS ARE JUST LIKE ANY OTHER

JUST SOME BIT PATTERN

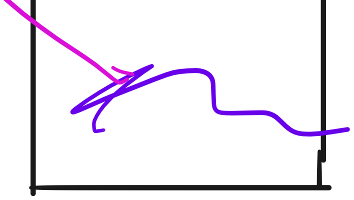
- INTERPRETATION DETERMINED BY HOW IT IS USED
- A SLIGHT PROBLEM WHEN CALLING FUNCTIONS PROVIDED BY THE KERNEL
    - fork( )
    - open(char*, int)
    - read(int, char*, size_t)
    - write(int, char*, size_t)

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



```
const char * f = "L03.txt";
int fd = open(f, O_RD);
int open(char*, int) {

    return fd;
}
```

iret



# User ⟷ Kernel Transitions

1. Support usual function semantics
   - ↳ Pass arguments } Calling convention
   - → Return value
   - → Return control flow } Information on stack
   - → Variable scope } Stack frames OR —
   - → Kernel memory protection } Hardware

## TRAPS

- Mechanisms for kernel ⟷ userspace transitions

- Syscall

- exceptions

- interrupts

Creating a process

- fork

- execve