Abstract and Bio

A SIMPLE RISC-V MULTITASKING OS FOR LEARNING

2021-12-03 @ Ohio Linux Fest, Columbus, Ohio

Abstract:

Explore the line between hardware and software by writing code with absolute control over the cpu and peripherals. We'll explore how to do this using a completely free and open source simulator (Renode), toolchain (GCC), and instruction set (RISC-V). Using assembly, we'll initialize parts of the system such as CPU interrupts and privilege levels. Finally we'll review the assembly code for doing a context switch, the key software to which enables multitasking via timesharing.

Bio:

Zak Kohler is a Chemical Engineer by training but a hacker at heart. He started programming in 3rd grade and has never let up. His first foray in open source was in early high school, and he discovered Linux and Free Software at university. Electronics is his second love and he fuses the two by playing with early computer hardware, modern microcontrollers and FPGAs. When zak isn't messing with computers he can be found growing plants, drawing, and exploring the world on foot.

A SIMPLE RISC-V MULTITASKING OS FOR LEARNING

2021-12-03 - Ohio Linuxfest - Zak Kohler

Zak Kohler

(y2kbugger)

- Chemical Engineering, University of Akron 2010
- Materials engineer turned software developer

- Relevant Interests:
  - programming, electronics, retrocomputing
- Irrelevant Interests:
  - running, punk rock, cheesemaking

Outline

Motivation

Explain baremetal, risc-v, renode

Operating system background

KohlerOS, my operating system

- Startup
- Shell
- Multitasking, the context switch
- System calls

Motivation

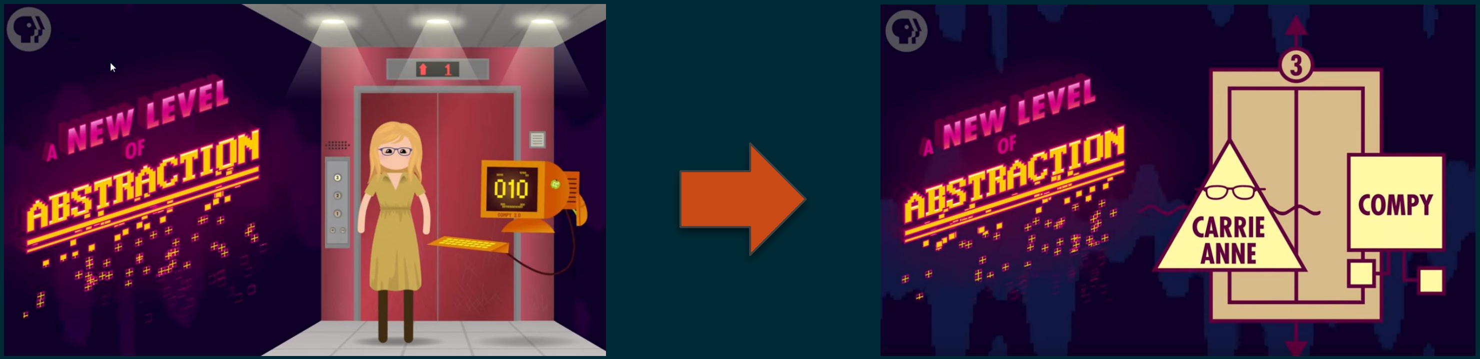
Learning from a toy Operating System

This talk seeks to explore the line between hardware and software using a minimal simulated environment to write and play with a toy operating system.

Abstraction

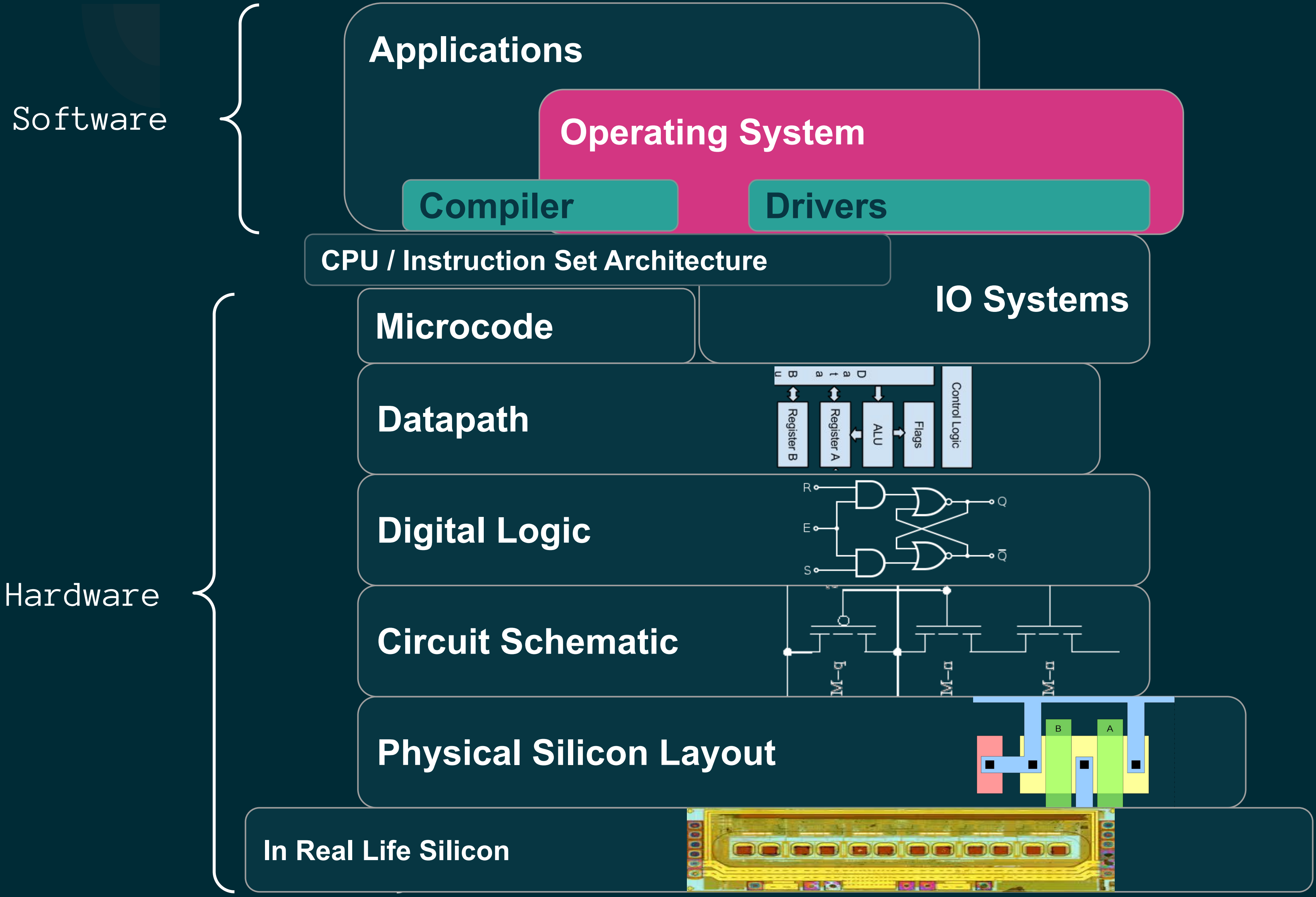
Simple interfaces *allow* complex systems

You don't think about radios each time you send a text via sms, this is normally considered a good thing.



Video Series on Computer Science "Up another level of abstraction"
https://www.youtube.com/watch?v=tpIctyqH29Q

Abstractions in Computers



Software

Applications

Operating System

Compiler

Drivers

CPU / Instruction Set Architecture

IO Systems

Hardware

Microcode

Datapath

Digital Logic

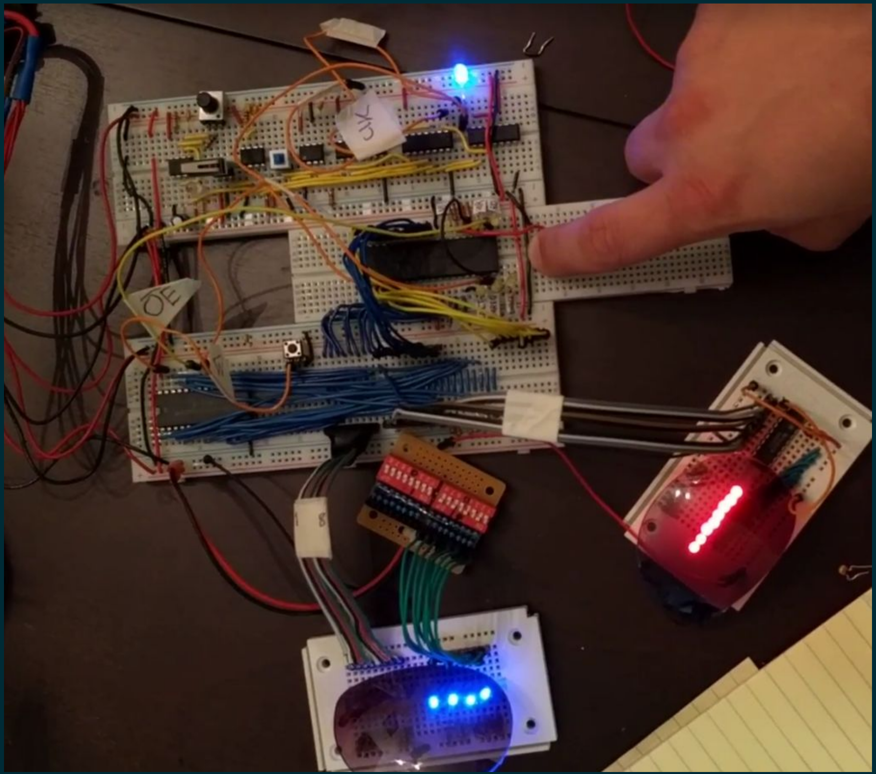
Circuit Schematic

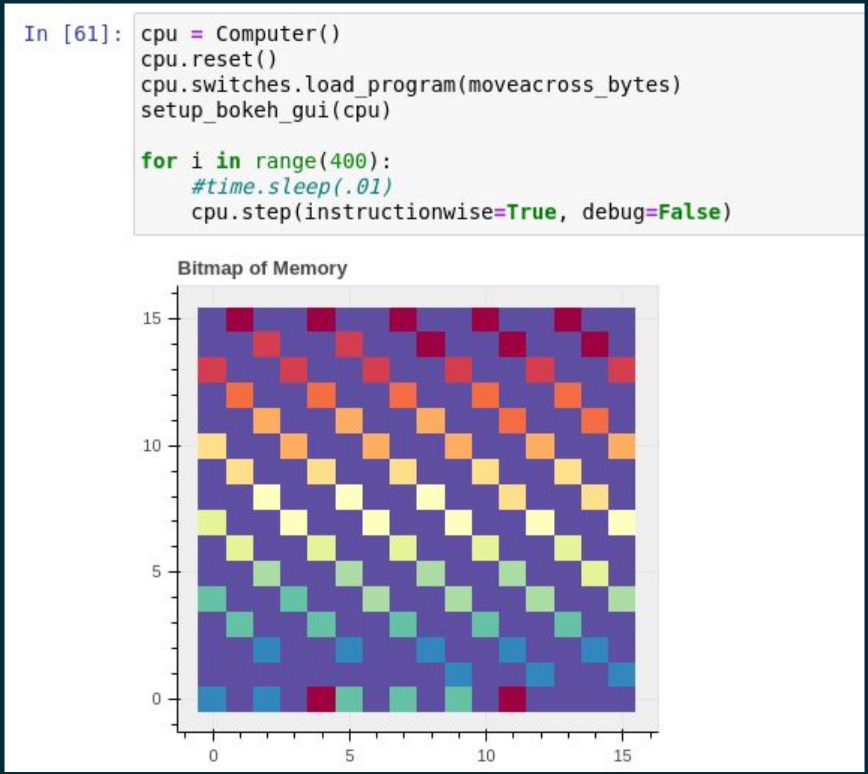
Physical Silicon Layout

In Real Life Silicon

In pursuit of killing the magic I've been playing with hardware and toy CPU emulators

This talk is a(nother) step towards building a IRL CPU





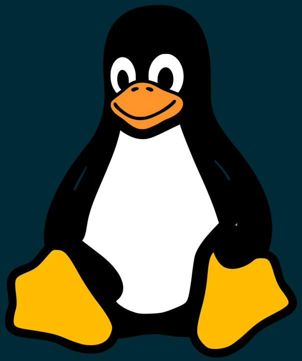
```
In [61]: cpu = Computer()
         cpu.reset()
         cpu.switches.load_program(moveacross_bytes)
         setup_bokeh_gui(cpu)

         for i in range(400):
             #time.sleep(.01)
             cpu.step(instructionwise=True, debug=False)
```

What is baremetal?

In the non-embedded world, when you compile and link a C program into an executable you are doing so with the intention of running it within a specific operating system.

In contrast when you compile baremetal or `–freestanding` you are telling the compiler that you intend to run this without relying on an operating system.

What is baremetal?



This could be used, for example, to write an operating system. Alternatively it can be used to access the hardware of a system directly on an embedded system. Doing so sacrifices higher level luxuries such as memory management, standard IO, thread/process control, etc.

Because of this, sometimes it makes sense to run on a type of minimal OS optimized for embedded, e.g. a real time operating system (RTOS).

What is RISC-V?

Wikipedia

> RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use.

Akin to x86, arm, mips, alpha, 6502, powerpc

What is Renode

Renode is a simulator designed for embedded firmware, including networks of devices.

Traditional alternatives such as QEMU aren't as optimized for the embedded space.

Video game console emulators target a specific platform.

Operating Systems

First operating system: Human operators

- Batch Processed
- Each program has full control of the entire mainframe.

The role of operating systems

The job of an operating system is to abstract away hardware. They do this by implementing standard interface for tasks that talk to real world.

- Input/Output
- Timesharing
- Memory Management
- File systems
- User/Machine Mode

Timesharing

Imagine if this guy had to manually set the program counter each time a human interrupted with some input…or change out one job for another thousands of times per second.

My OS

The job of an operating system is to abstract away hardware. They do this by implementing standard interface for tasks that talk to real world.

- ~~Input/Output~~
- Timesharing
- ~~Memory Management~~
- ~~File systems~~
- ~~User/Machine Mode~~

Monitor/Shell

The shell's special job:

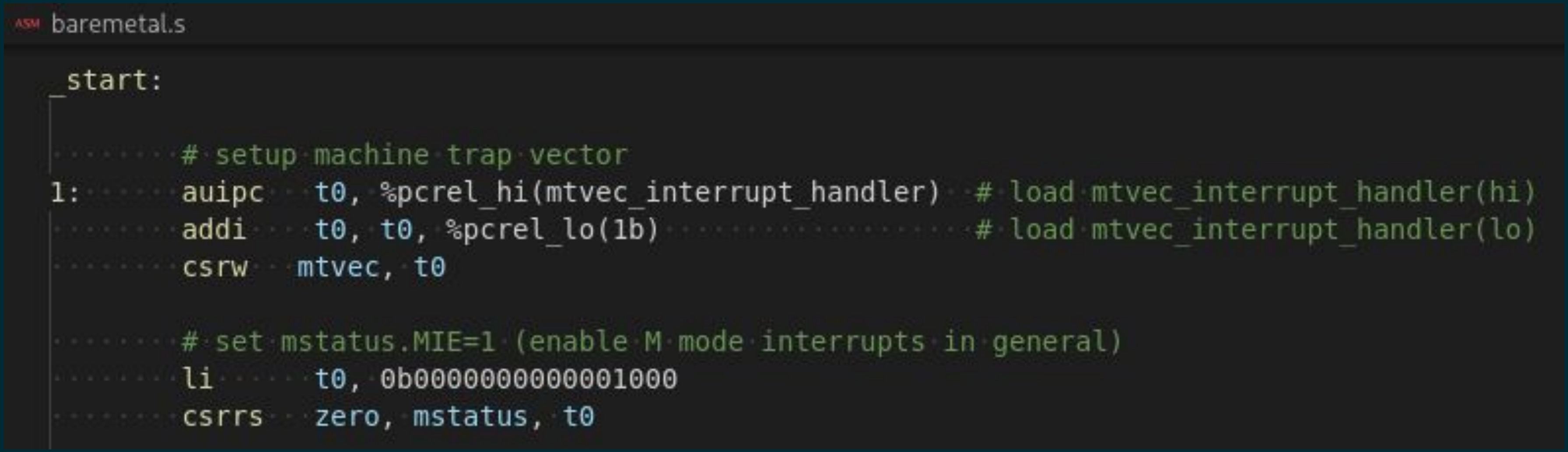
launch and control other processes.

```
kos> ?
Programs:
    h: hello world
    e: hello ecall world
    l: laugh
    f: laugh forever
    c: count forever
Shell commands:
    ?: show this help
    @: list stopped processes
    ^: restart all stopped processes in background
    !: stop all background processes.
  C-Z: stop foreground process
  C-C: kill foreground process
```

If the shell launches programs, how does the shell get launched?

Early Startup

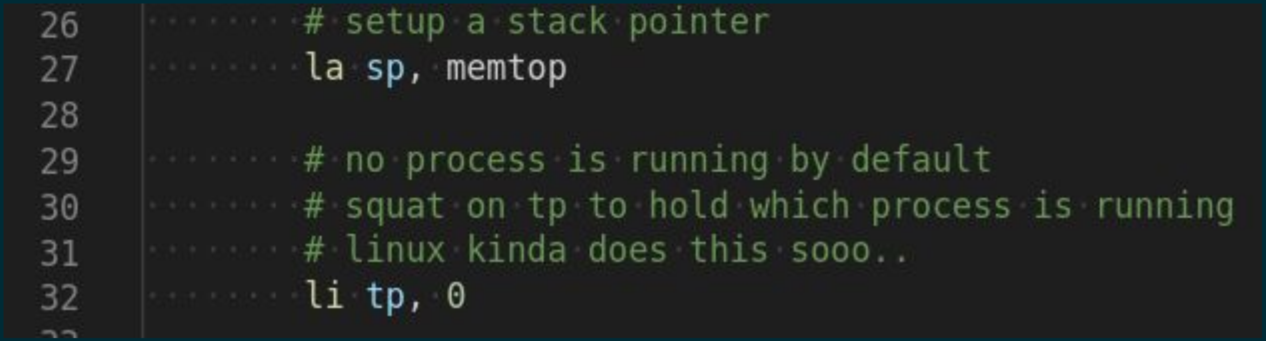
Setup interrupts



```
_start:

        # setup machine trap vector
1:      auipc   t0, %pcrel_hi(mtvec_interrupt_handler)  # load mtvec_interrupt_handler(hi)
        addi    t0, t0, %pcrel_lo(1b)                   # load mtvec_interrupt_handler(lo)
        csrw   mtvec, t0

        # set mstatus.MIE=1 (enable M mode interrupts in general)
        li      t0, 0b0000000000001000
        csrrs   zero, mstatus, t0
```

Early Startup

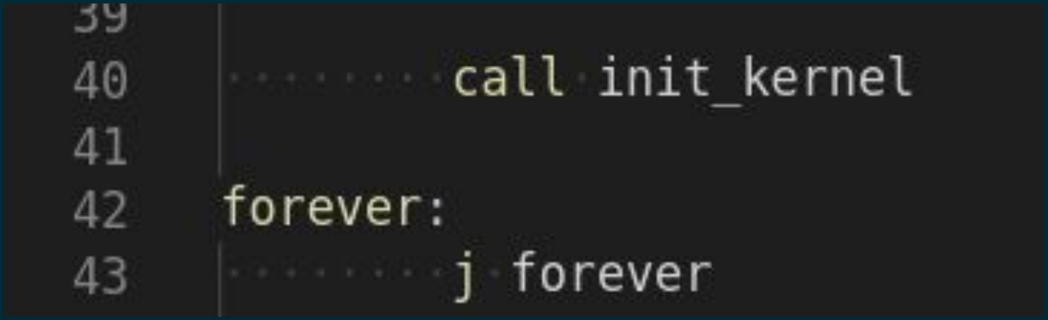
Initialize stack and process pointers



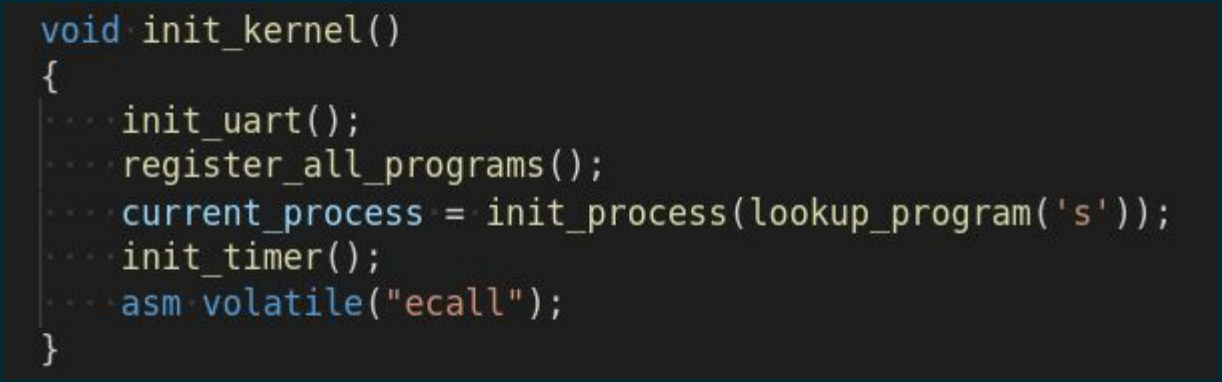
```
        # setup a stack pointer
        la sp, memtop

        # no process is running by default
        # squat on tp to hold which process is running
        # linux kinda does this sooo..
        li tp, 0
```

Early Startup

Jump to C to initial rest of kernel



```
        call init_kernel

forever:
        j forever
```



```
void init_kernel()
{
    init_uart();
    register_all_programs();
    current_process = init_process(lookup_program('s'));
    init_timer();
    asm volatile("ecall");
}
```

Monitor/Shell

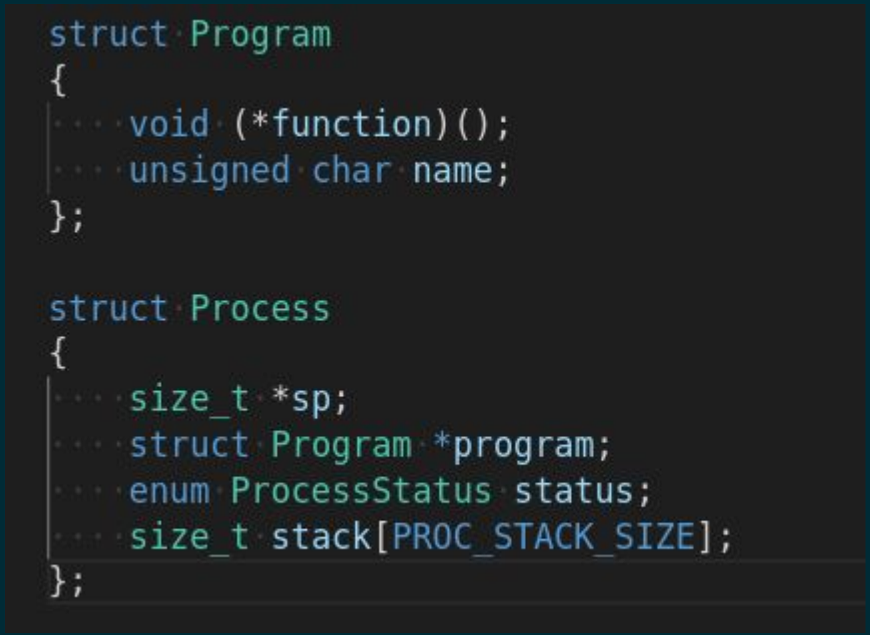
```
kos> ?
Programs:
    h: hello world
    e: hello ecall world
    l: laugh
    f: laugh forever
    c: count forever
Shell commands:
    ?: show this help
    @: list stopped processes
    ^: restart all stopped processes in background
    !: stop all background processes.
  C-Z: stop foreground process
  C-C: kill foreground process
```

Programs and Processes

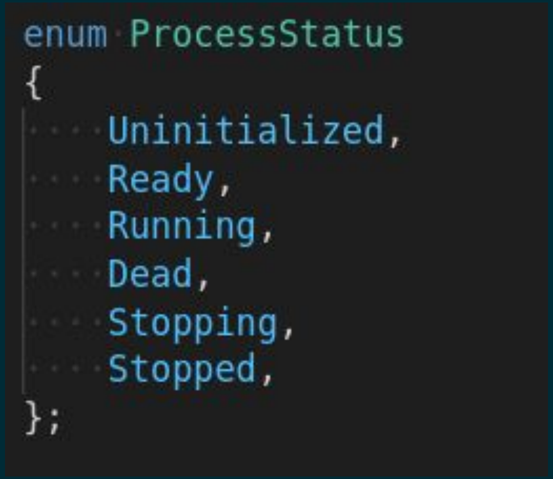
Programs are simply C functions registered with a name.

A running program is a Process.

Scheduling is just a naive round-robin for all "Ready" Processes.

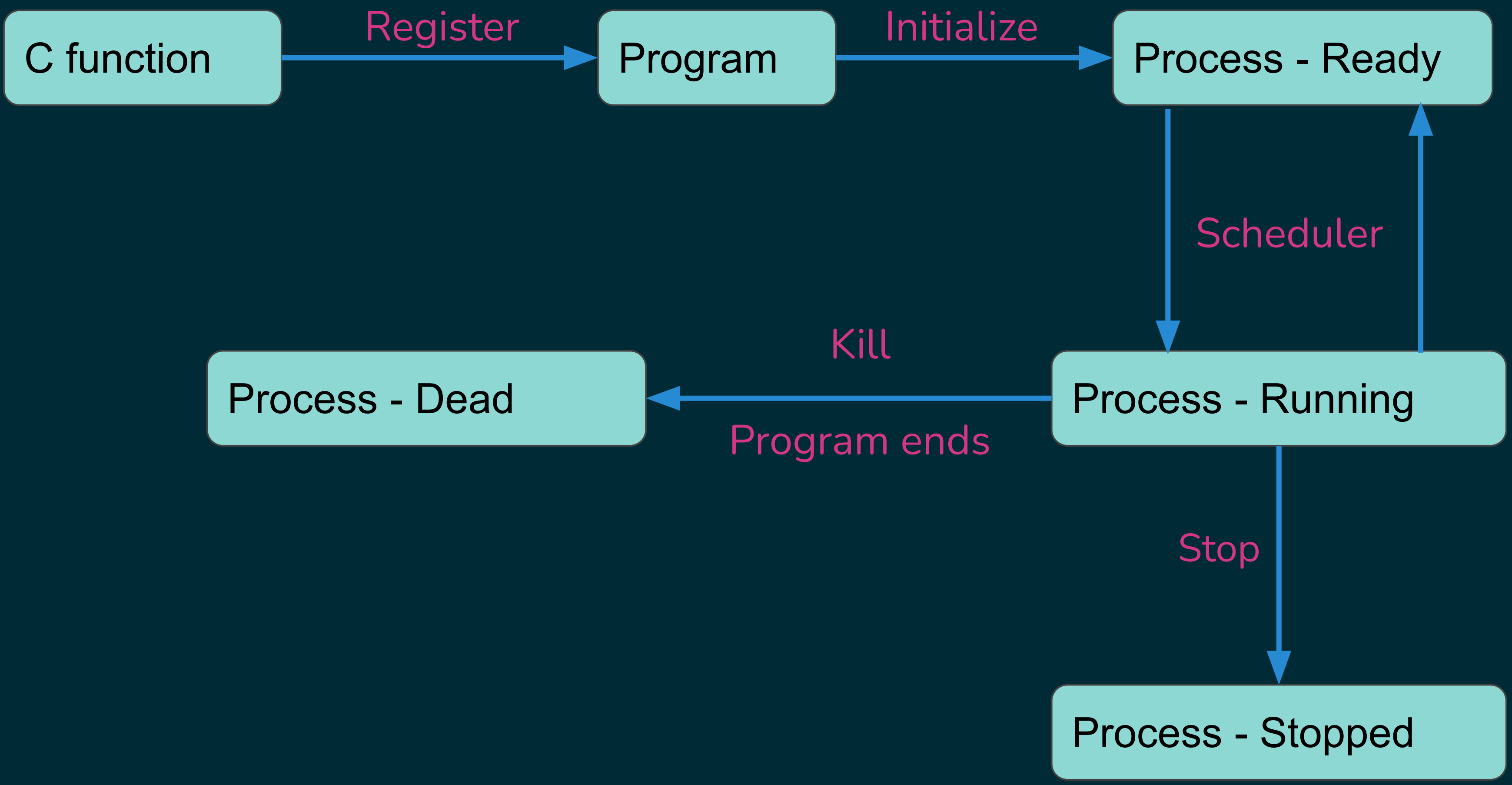


```c
struct Program
{
    void (*function)();
    unsigned char name;
};

struct Process
{
    size_t *sp;
    struct Program *program;
    enum ProcessStatus status;
    size_t stack[PROC_STACK_SIZE];
};
```



```c
enum ProcessStatus
{
    Uninitialized,
    Ready,
    Running,
    Dead,
    Stopping,
    Stopped,
};
```

Lifecycle of a kOS process



Launch a simple program using the shell.

Processes are written as C functions, but this is out of convenience. We could add an arbitrary binary loader to the shell.

DEMO

Ok, that's not very interesting

It just looks as if the shell is dispatching to a C function based on user input.

Next I will give a demo to prove that this is not the case.

DEMO

Context switch

enable preemptive multitasking

The context switch

A software construct that

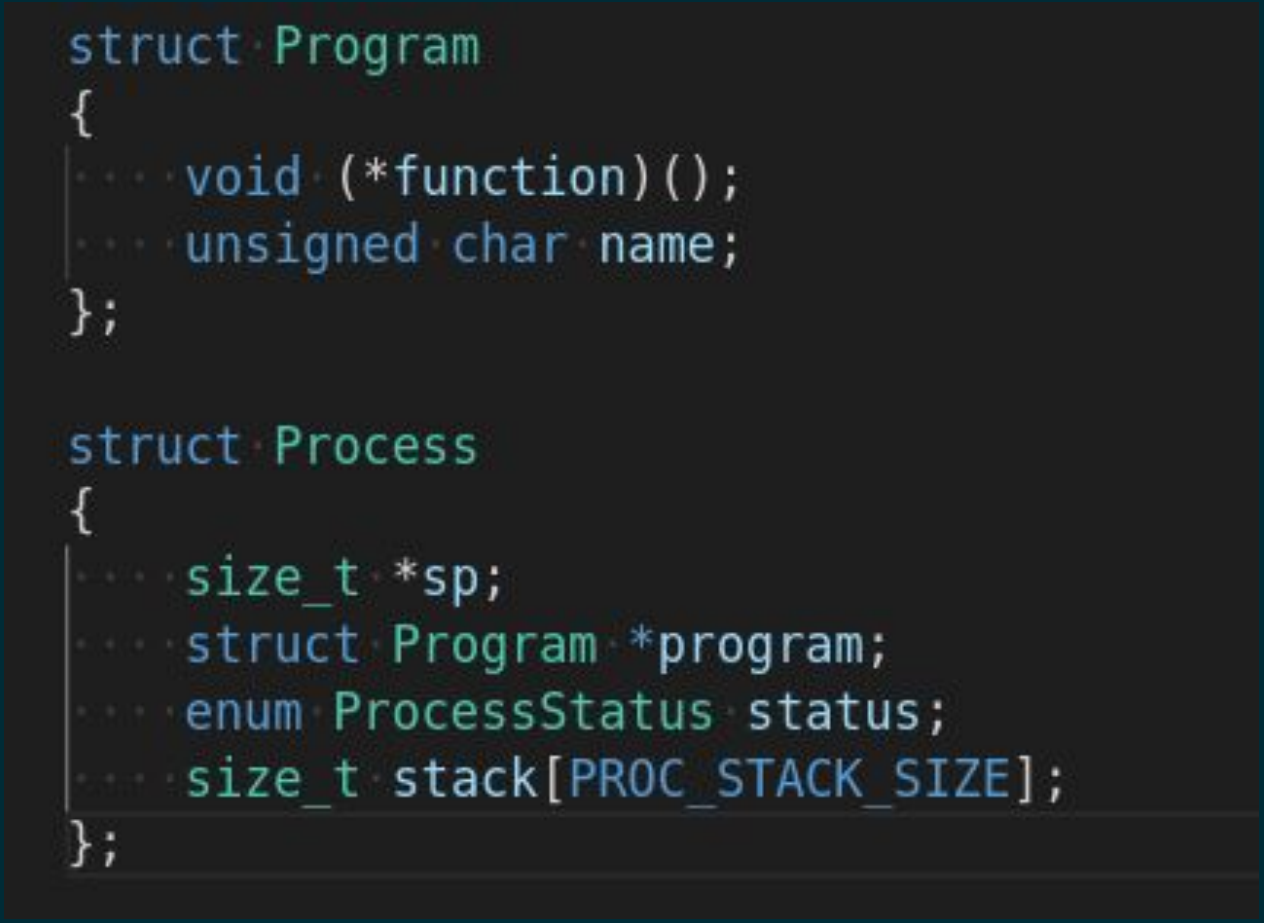
- Cannot be written in C
- Is CPU architecture dependent
- Is triggered by a hardware timer interrupt

The basic stages are:

- Backs up state of CPU registers
- Restores a different state
- and then continues execution.

Show Code

Programs and Processes



```c
struct Program
{
    void (*function)();
    unsigned char name;
};

struct Process
{
    size_t *sp;
    struct Program *program;
    enum ProcessStatus status;
    size_t stack[PROC_STACK_SIZE];
};
```

System Call,
Software Interrupt,
ecall

Software Interrupts

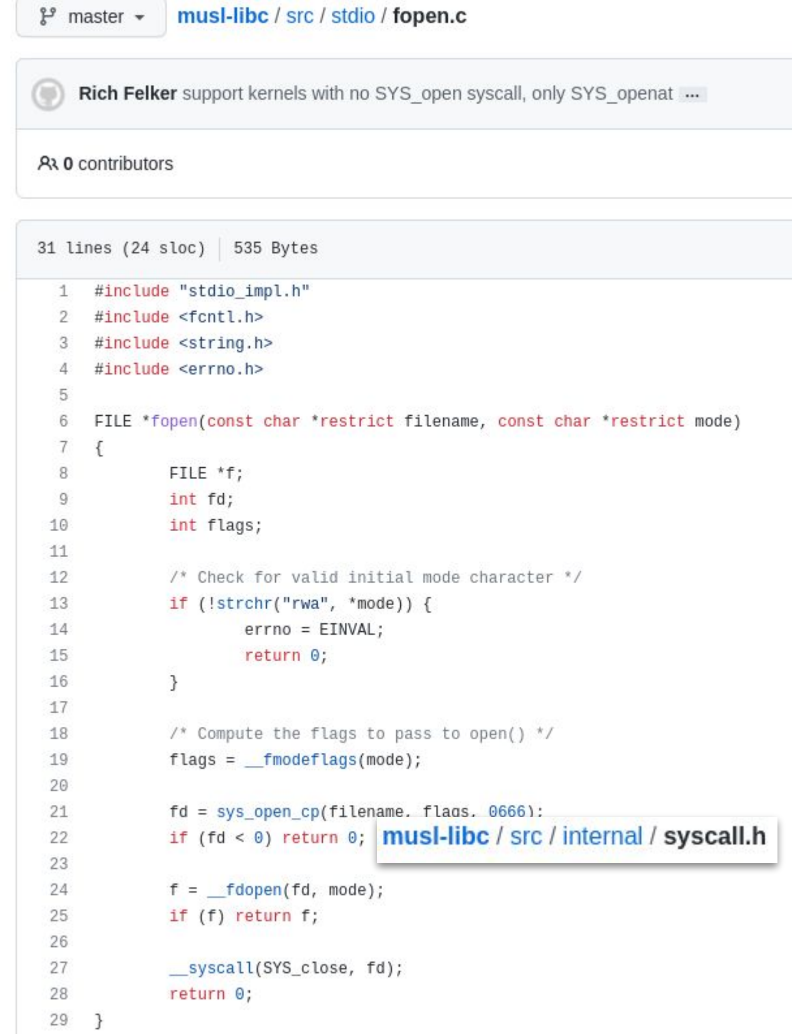
- In an operating system using privilege modes, processes are prevented from accessing the hardware directly.
- In order to talk to hardware a process must talk to the kernel.

Q: How can a process talk to the kernel?

A: Software interrupts

Show Code

How a real standard library uses system calls (musl)



master ▾ **musl-libc / src / stdio / fopen.c**

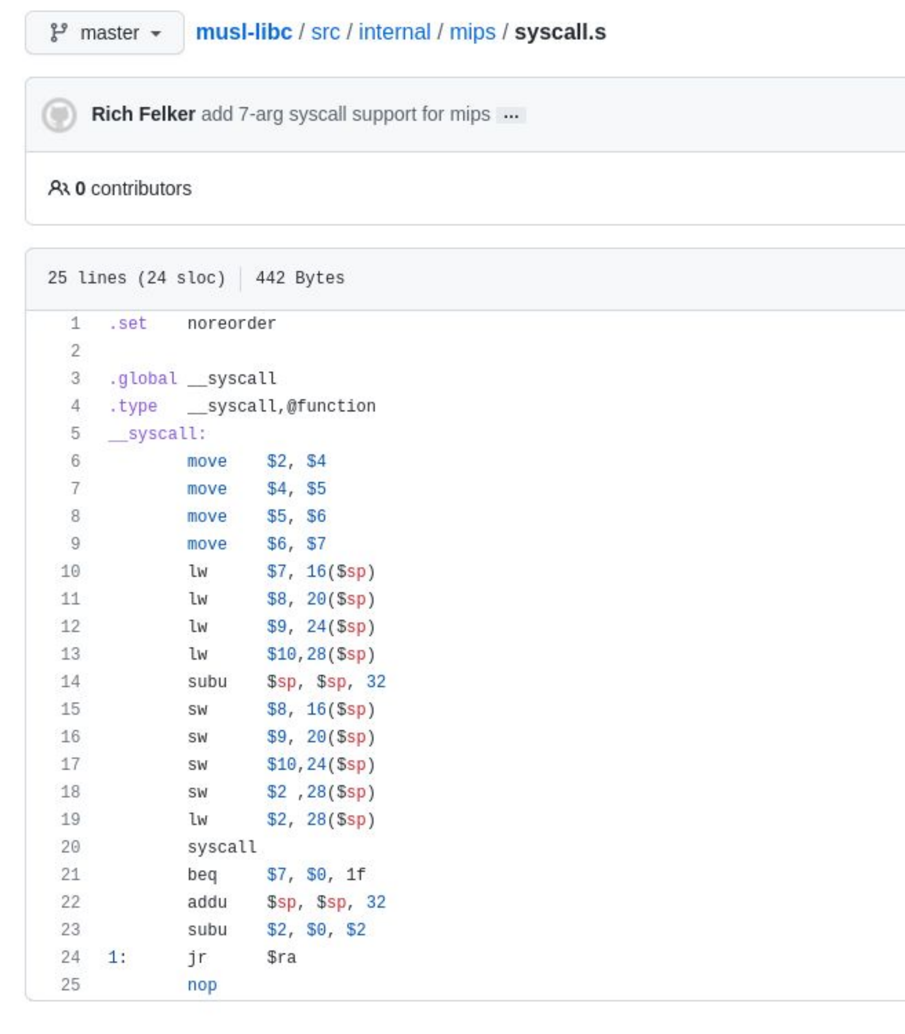
Rich Felker support kernels with no SYS_open syscall, only SYS_openat …

0 contributors

31 lines (24 sloc) | 535 Bytes

```c
#include "stdio_impl.h"
#include <fcntl.h>
#include <string.h>
#include <errno.h>

FILE *fopen(const char *restrict filename, const char *restrict mode)
{
        FILE *f;
        int fd;
        int flags;

        /* Check for valid initial mode character */
        if (!strchr("rwa", *mode)) {
                errno = EINVAL;
                return 0;
        }

        /* Compute the flags to pass to open() */
        flags = __fmodeflags(mode);

        fd = sys_open_cp(filename, flags, 0666);
        if (fd < 0) return 0;

        f = __fdopen(fd, mode);
        if (f) return f;

        __syscall(SYS_close, fd);
        return 0;
}
```

musl-libc / src / internal / syscall.h



master ▾ **musl-libc / src / internal / mips / syscall.s**

Rich Felker add 7-arg syscall support for mips …

0 contributors

25 lines (24 sloc) | 442 Bytes

```asm
.set    noreorder

.global __syscall
.type   __syscall,@function
__syscall:
        move    $2, $4
        move    $4, $5
        move    $5, $6
        move    $6, $7
        lw      $7, 16($sp)
        lw      $8, 20($sp)
        lw      $9, 24($sp)
        lw      $10,28($sp)
        subu    $sp, $sp, 32
        sw      $8, 16($sp)
        sw      $9, 20($sp)
        sw      $10,24($sp)
        sw      $2 ,28($sp)
        lw      $2, 28($sp)
        syscall
        beq     $7, $0, 1f
        addu    $sp, $sp, 32
        subu    $2, $0, $2
1:      jr      $ra
        nop
```

Wrap Up

Motivation

Explain baremetal, risc-v, renode

Operating system background

My operating system

- Startup
- Shell
- Multitasking, the context switch
- System calls

Further Learning

Topics Not Covered

- Memory management
- Race conditions
- CPU modes/rings
- File Systems
- Drivers
  - UART
  - Timer
- RTOS
- Deploying to hardware

Want to try this out for yourself?

Last year I covered the basics of using Renode to run baremetal risc-v code.

https://blog.y2kbugger.com/baremetal-riscv-renode-1.html

Thank You!

@y2kbugger on twitter

https://blog.y2kbugger.com

Questions??