OS in 1000 Lines Notes

Katie Baek

Boston College

Notes from coding OS in 1000 lines

November 19, 2025

OS in 1000 Lines Notes

Katie Baek

November 19, 2025

Abstract

In the spirit of grokking the OS, we are attempting to write our own mini-OS using the guide "OS in 1000 Lines"[2]. We are expecting to finish within the next few weeks, then move onto the hypervisor.

Contents

1 RISC-V 101

The chosen CPU for this book is RISC-V, specifically, the 32-bit RISC-V. This is because there is well-written documentation and it has been rising in popularity in the recent years. The underlying machine is the QEMU virtual machine. Although this does not exist in the real world, it's simple and similar to a lot of real devices.

1.1 RISC-V Assembly

In order to write an OS, you need to write some assembly. Here is an example of assembly code:

```
addi a1 a0 123
```

The first instruction *addi* is the instruction name (opcode). The entire line tells the computer to add the value **123** to the value **a1** and store it in the register **a0**

1.2 Registers in RISC-V

Below is a table of common registers in RISC-V and their given alias that we will use to refer to them in code.

Register	ABI Name (alias)	Description
pc	pc	Program counter (where the next instruction is)
x0	zero	Hardwired zero (always reads as zero)
x1	ra	Return address
x2	sp	Stack pointer
x5-x7	t0-t2	Temporary registers
x8	fp	Stack frame pointer
x10-x11	a0-a1	Function arguments/return values
x12-x17	a2-a7	Function arguments
x18-x27	s0-s11	Temporary registers saved across calls
x28-x31	t3-t6	Temporary registers

1.3 Memory access

Most data is stored in memory, then moved to registers when its needed and so there are instructions which do this:

```
1 lw a0, (a1)
2 sw a0, (a1)
```

In line 1, you read a word from the address that is stored in `a1` and store it in `a0`. The equivalent C code is `a0 = *a1`. You are doing the same thing in line 2, but storing the word instead of loading it. The `(...)` is like a pointer dereference.

1.4 Stack

The stack is a LIFO (last in first out) memory space used for function calls and local variables. It grows downwards, and the stack pointer, `sp` points to the ”top” of the stack (but functionally the bottom since again, it grows downwards). To push onto the stack, decrement the stack pointer and store the value:

```
addi sp, sp -4 // Move the stack pointer down by 4 bytes
sw a0, (sp) // Store a0 to the stack
```

To pop from the stack, load the value and increment the stack pointer:

```
lw a0, (sp) // Load a0 from the stack
addi sp, sp 4 // Move the stack pointer up by 4 bytes
```

1.5 Privileged instructions

With CPU instructions, there are privileged instructions that applications in user mode cannot execute. The **CSR (Control and Status Register)** is a register that sores CPU settings. We use the following:

Opcode and operands	**Overview**
`csrr rd, csr`	Read from CSR
`csrw csr, rs`	Write to CSR
`csrrw rd, csr, rs`	Read from and write to CSR at once
`sret`	Return from trap handler
`sfence.vma`	Clear Translation Lookaside Buffer (TLB)

1.6 Inline assembly

Inline assembly code for C looks like this:

```
uint32_t value;
__asm__ __volatile__("csrr %0, sepc" : "=r(value)");
```

Listing 1: sample inline assembly code

Where its written in the form of:

```
__asm__ __volatile__("assembly" : output operands : input
    operands : clobbered registers);
```

Listing 2: format of inline assembly code

In the code 1, this reads the value of `sepc` CSR using the `scrr` instruction and assigns it to the `value` variable.

Part	Description
`__asm__`	Indicates its inline assembly
`__volatile__`	Tell the compiler not to optimize the "assembly" code
`"assembly"`	Assembly code written as a string literal
output operands	C variables to store the results of the assembly
input operands	C expressions to be used in the assembly
clobbered registers	Registers whose contents are destroyed in the assembly. If forgotten, the C compiler won't preserve the contents of these registers and would cause a bug.

```
 __asm__ __volatile__("csrw sscratch, %0" : : "r"(123))
```

Listing 3: different sample inline assembly code

In the code 3, this writes `123` to the `sscratch` CSR, using the `csrw` instruction. The C compiler automatically sets `123` to the register `a0` so that the assembly can be executed as written which is nice because then the dev doesn't have to worry about it.

2 Overview

2.1 Features

We will implement the following features:

1. **Multitasking**: Switch between processes to allow multiple applications to share the CPU
2. **Execption Hanlder**: Handle events requiring OS intervention, such as illegal instructions
3. **Paging**: Provide an isolated memory address space for each application
4. **System calls**: Allow applications to call kernel features
5. **Device drivers**: Abstract hardware functionalities, such as disk read-/write
6. **File system**: Manage files on disk
7. **Command-line shell**: User interface for humans

3 Boot

What happens when the computer is turned on? The CPU initalizes itself and starts executing the OS. The OS then initalizes the hardware and starts the applications. This process is called "booting".

What happens before the OS starts? In PCs, BIOS initalizes the hardware, displays the splash screen, and loads the OS from the disk. In QEMU virt machine, **OpenSBI** is the equivalent of BIOS. This is the CPU "initalizing itself".

3.1 Supervisor Binary Interface (SBI)

The SBI is an API for OS kernels, but defines what the firmware (OpenSBI) provides to an OS. In QEMU, OpenSBI starts by default, performs hardware-specific initalization, and boots the kernel.

3.2 Booting OpenSBI

The following shell code starts OpenSPI:

```
#!bin/bash
set -xue

# QEMU file path
QEMU=qemu-system-riscv32

# Start QEMU
$QEMU -machine virt -bios default -nographic -serial mon:
    stdio --no-reboot
```

Listing 4: script to start OpenSBI

With this shell script, you can boot OpenSBI. Specifically, the `-bios default` option tells QEMU to use the default firmware, which is OpenSBI. In addition, at this point, QEMU's standard input and standard output is connected to the virtual machine's serial port, and the characters that you type in are being sent to OpenSBI but there is no one to read the characters. This is because there is literally no OS. In order to do anything, we have to interact with QEMU (which is the software emulator and a type-2 hypervisor).

Press Ctrl+A then C to switch to the QEMU debug console (QEMU monitor)

Here are the other options which are used in the run script:

1. `-machine vert`: Start a `vert` machine
2. `-bios default`: Use the default firmware (OpenSBI)
3. `-nographic`: Start QEMU without a GUI window

4. `-serial mon:stdio`: Connect QEMU's standard IO to the virtual machine's serial port. Specifying `mon:` allows switching to the QEMU monitor by pressing Ctrl + A then C
5. `--no-reboot`: If the virtual machine crashes, stop the emulator without rebooting (useful for debugging)

3.3 Linker script

```
ENTRY(boot)

SECTIONS {
  . = 0x80200000;

  .text :{
    KEEP(*(.text.boot));
    *(.text .text.*);
  }

  .rodata : ALIGN(4) {
    *(.rodata .rodata.*);
  }

  .data : ALIGN(4) {
    *(.data .data.*);
  }

  .bss : ALIGN(4) {
    __bss = .;
    *(.bss .bss.* .sbss. .sbss.*);
    __bss_end = .;
  }

  . = ALIGN(4);
  . += 128 * 1024;
  __stack_top = .;
}
```

Listing 5: the linker script

The linker script is a file which defines the memory layout of executable files. Based on the layout, the linker assigns memory address to functions and variables. AKA this tells the linker exactly where in memory all parts of the program should live. You need this because during the **boot** process, there is no OS, no loader, and no runtime environment to relocate the code for you. Specifically, the linker script controls:

1. Where the code is placed in memory
2. Where global variables, stacks, and boot code lives

3. The address of symbols used by startup code
4. How the final binary is organized

3.3.1 Why do you need one during boot?

When a computer boots, the CPU starts executing from a well-defined hardware reset address. In this case, it is `0x80200000`. The boot code *must* be there or else the CPU would start executing whatever is at that address. In addition, it also defines the variables `__bss`, `__bss_end`, and `__stack_top` with specific addresses.

The boot code needs to ensure that the hardware is in a state to find and load the kernel, which is then going to take things over from there [1]

3.4 Minimal kernel

The kernel starts executing from the `boot` function (which was specified as the entry point in the linker script).

```c
typedef unsigned char uint8_t;
typedef unsigned int uint32_t;
typedef uint32_t size_t;

extern char __bss[], __bss_end[], __stack_top[];

void *memset(void *buf, char c, size_t n) {
  uint8_t *p = (uint8_t *)buf;
  while (n--)
    *p++ = c;
  return buf;
}

void kernel_main(void) {
  memset(__bss, 0, (size_t) __bss_end - (size_t) __bss);

  for(;;);
}

__attribute__((section(".text.boot")))
__attribute__((naked))
void boot(void) {
  __asm__ __volatile__(
    "mv sp, %[stack_top]\n"
    "j kernel_main\n"
    :
    : [stack_top] "r" (__stack_top)
  );
}
```

Listing 6: kernel code

References

[1] Alex Bennee. *Anatomy of a Boot, a QEMU perspective*. URL: https://www.qemu.org/2020/07/03/anatomy-of-a-boot/.

[2] Seiya Nuta. *OS in 1000 Lines*. URL: https://operating-system-in-1000-lines.vercel.app/en/.