

OS in 1000 Lines Notes

Katie Baek

Boston College

Notes from coding OS in 1000 lines

November 28, 2025

OS in 1000 Lines Notes

Katie Baek

November 28, 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" [6]. This book includes the instructions, plus any explanations or questions that I had while reading the instructions.

Contents

1	RISC-V 101	2
1.1	But what is an ISA?	2
1.1.1	Why does the OS care about the ISA?	2
1.1.2	How is the ISA different from the ABI?	2
1.2	RISC-V Assembly	2
1.3	Registers in RISC-V	3
1.4	Memory access	3
1.5	Stack	3
1.6	Privileged instructions	3
1.7	Inline assembly	3
2	Overview	4
2.1	Features	4
2.2	QEMU	5
2.2.1	So QEMU is not a hypervisor?	5
2.3	QEMU Monitor	5
3	Boot	5
3.1	Supervisor Binary Interface (SBI)	5
3.2	Booting OpenSBI	5
3.3	Linker script	6
3.3.1	Why do you need one during boot?	7
3.4	Minimal kernel	7
3.5	First kernel debugging	8
4	Hello World!	9
4.1	Hello via SBI	9
4.2	printf function	10
4.2.1	What are the builtin features?	11
4.2.2	How are the builtin features used in printf?	12
5	C Standard Library	12
5.1	Memory operation	12
	References	12

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.

RISC-V a free and open standard ISA, based on Reduced Instruction Set Computer principles (designed to simply instructions given to computers to accomplish tasks).

1.1 But what is an ISA?

The Instruction Set Architecture (ISA) defines the language that the computer can "think" in (defines its programmable interface):

1. Instruction set
2. Register names and meanings
3. Calling conventions
4. Trap/exception behavior
5. Privilege levels
6. Memory model
7. Page table formats
8. How the system calls work
9. How interrupts are delivered

A device that interprets instructions defined by an ISA is an implementation of the ISA. If a devices implements an ISA, then the software that that devices runs must also comply with that same ISA. A program which was compiled for the x86 ISA can't necessarily run on the ARM ISA and that is because the way that they encode instructions are different.

1.1.1 Why does the OS care about the ISA?

The ISA is everything to the OS. As stated above, the ISA defines its programmable interface. The OS must be able to do things like handle traps, interrupts, and exceptions. It must also set up virtual memory. AKA the OS must obey the rules of the ISA in order to function.

1.1.2 How is the ISA different from the ABI?

First off, what is Application Binary Interface (ABI)? The ABI is sort like a contract that compiled binaries follow in order to interact with each other [2]. This way, new updates to a kernel or some other program don't break the functionality of the old ones and things are backwards compatible. Having knowledge of the ABI isn't necessary to building the OS (I think), but I've ran into the ABI enough times when researching that it was worth finally writing it down.

1.2 RISC-V Assembly

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

```
1  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**

1.3 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

1.4 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.

1.5 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:

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

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

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

1.6 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 stores CPU settings. We use the following:

1.7 Inline assembly

Inline assembly code for C looks like this:

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

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

Listing 1: sample inline assembly code

Where its written in the form of:

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

Listing 2: format of inline assembly code

Part	Description
<code>__asm__</code>	Indicates its inline assembly
<code>__volatile__</code>	Tell the compiler not to optimize the "assembly" code
<code>"assembly"</code>	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.

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

```
1  __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. **Exception Handler:** 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

2.2 QEMU

QEMU is a machine emulator which can emulate a variety of different machines. This means that it provides a virtual model of a computer (CPU, memory, and emulated devices). Since we are writing an OS for the RISC-V ISA, we are using the machine `virt`. `virt` does not exist in the real world and so it is only used in virtual machines, but it emulates a 32-bit CPU which implements the RISC-V ISA along with other devices that could be used in a computer [3].

2.2.1 So QEMU is not a hypervisor?

Yes, QEMU is a machine emulator and *not* a hypervisor. A hypervisor is able to execute instructions on the host CPU. QEMU just emulates the execution and so it is not a hypervisor. The emulator pretends to be hardware while the hypervisor lets the guest use the real hardware safely. However, both are similar in that they allow you to create that virtual machine abstraction. Within this virtual machine abstraction, we are aiming to develop an OS. This OS does not know that it's in a VM and it does not know that the machine is being emulated.

2.3 QEMU Monitor

The QEMU monitor is how you can interact with QEMU itself. It is used to:

1. Remove or insert removable media images (CD-ROM or floppy disks)
2. Freeze/unfreeze the VM or save/restore its state from a disk file
3. Inspect the VM state without an external debugger

3 Boot

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

What happens before the OS starts? In PCs, BIOS initializes 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 "initializing 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 initialization, and boots the kernel. In other words, the SBI performs tasks on the hardware/controls the hardware on behalf of the OS. It can be analogous to a program making sys calls to the kernel.

The SBI is specific to RISC-V. It defines how an S-mode operating system (like the one that we're developing) requests privileged operations from M-mode firmware. This is a design choice for RISC-V specifically, as, again, it is based on the *reduced* instruction set computer principles. The S-mode is the mode in which the kernel operates, M-mode is the mode in which OpenSBI operates, and the U-mode is the mode in which the user operates. Since the kernel is in S-mode, it cannot make the privileged calls, where privileged instructions include things that involve the hardware (like printing to console). If the kernel wants to make that call, then OpenSBI traps the instruction, does the printing, then hands control back to the kernel. OpenSBI is software which complies with the RISC-V SBI and allows the kernel to interact with the firmware in a standardized manner.

3.2 Booting OpenSBI

The following shell code starts OpenSPI:

```

1  #!/bin/bash
2  set -xue
3
4  # QEMU file path
5  QEMU=qemu-system-riscv32
6
7  # Start QEMU
8  $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

```

1 ENTRY(boot)
2
3 SECTIONS {
4   . = 0x80200000;
5
6   .text :{
7     KEEP(*(.text.boot));
8     *(.text .text.*);
9   }
10
11  .rodata : ALIGN(4) {
12    *(.rodata .rodata.*);
13  }
14
15  .data : ALIGN(4) {
16    *(.data .data.*);
17  }
18
19  .bss : ALIGN(4) {
20    __bss = .;
21    *(.bss .bss.* .sbss .sbss.*);
22    __bss_end = .;
23  }
24
25  . = ALIGN(4);
26  . += 128 * 1024;

```

```

27  __stack_top = .;
28 }

```

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 [\[4\]](#)

3.4 Minimal kernel

```

1 typedef unsigned char uint8_t;
2 typedef unsigned int  uint32_t;
3 typedef uint32_t size_t;
4
5 extern char __bss[], __bss_end[], __stack_top[];
6
7 void *memset(void *buf, char c, size_t n) {
8     uint8_t *p = (uint8_t *)buf;
9     while (n--)
10         *p++ = c;
11     return buf;
12 }
13
14 void kernel_main(void) {
15     memset(__bss, 0, (size_t) __bss_end - (size_t) __bss);
16
17     for(;;);
18 }
19
20 __attribute__((section(".text.boot")))
21 __attribute__((naked))
22 void boot(void) {
23     __asm__ __volatile__(
24         "mv sp, %[stack_top]\n"
25         "j kernel_main\n"
26         :
27         : [stack_top] "r" (__stack_top)
28     );
29 }

```

Listing 6: kernel code

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

1. `__attribute__((section(".text.boot")))` controls the placement of the function in the linker script. Since OpenSBI jumps to `0x80200000` automatically, the boot code needs be to put there
2. `__attribute__((naked))` instructs the compiler not to generate any unnecessary code before and after the function body. This ensures that the inline assembly code is the exact function body.

In the `boot` function, in line (24), the `stack pointer` (`sp`) is set to the end address of the stack area thats defined in the linker script. Then, it jumps to the `kernel_main` function.

In line (5), `extern` is used because these variables come from the linker script, and not `kernel.c`. The `char` and `[]` that they are addresses, you can take the pointer value of it, and that its type is a pointer to char, which is a byte.

In the `main` function, the `.bss` section is initialized to zero using `memset`. Then, the function enters an infinite loop. This prevents the execution from leaving the controlled code.

3.5 First kernel debugging

Because the kernel enters an infinite loop, there are no indications that the kernel is running correctly. However, you can look at values of registers to see if anything has gone wrong. To do this, open the QEMU monitor and execute the `info registers` command

```
(qemu) info registers

CPU#0
V      = 0
pc      80200048
mhartid 00000000
mstatus 80006080
mstatush 00000000
hstatus 00000000
vsstatus 00000000
mip      00000020
mie       00000008
mideleg   00001666
hideleg   00000000
medeleg   00f0b509
hedeleg   00000000
mtvec     800004e0
stvec     80200000
vstvec    00000000
mepc      80200000
sepc      00000000
vsepc     00000000
mcause    00000003
scause    00000000
vscause   00000000
mtval     80010724
stval     00000000
htval     00000000
mtval2    00000000
mscratch  80046000
sscratch  00000000
satp      00000000
x0/zero   00000000 x1/ra    8000e63e x2/sp    8022004c x3/gp    00000000
x4/tp     80046000 x5/t0    00000001 x6/t1    00000002 x7/t2    00001000
x8/s0     80045f40 x9/s1    00000001 x10/a0   8020004c x11/a1   8020004c
x12/a2    00000000 x13/a3   00000019 x14/a4   00000000 x15/a5   00000001
x16/a6    00000001 x17/a7   00000005 x18/s2   80200000 x19/s3   00000000
x20/s4    87e00000 x21/s5   00000000 x22/s6   80006800 x23/s7   00000001
x24/s8    00002000 x25/s9   80042308 x26/s10  00000000 x27/s11  00000000
x28/t3    80020ad1 x29/t4   80045f40 x30/t5   0000008c x31/t6   00000000
fcsr      00000000
f0/ft0    ffffffff00000000 f1/ft1    ffffffff00000000 f2/ft2    ffffffff00000000 f3/ft3    ffffffff00000000
f4/ft4    ffffffff00000000 f5/ft5    ffffffff00000000 f6/ft6    ffffffff00000000 f7/ft7    ffffffff00000000
f8/fs0    ffffffff00000000 f9/fs1    ffffffff00000000 f10/fa0   ffffffff00000000 f11/fa1   ffffffff00000000
f12/fa2   ffffffff00000000 f13/fa3   ffffffff00000000 f14/fa4   ffffffff00000000 f15/fa5   ffffffff00000000
f16/fa6   ffffffff00000000 f17/fa7   ffffffff00000000 f18/fs2   ffffffff00000000 f19/fs3   ffffffff00000000
f20/fs4   ffffffff00000000 f21/fs5   ffffffff00000000 f22/fs6   ffffffff00000000 f23/fs7   ffffffff00000000
f24/fs8   ffffffff00000000 f25/fs9   ffffffff00000000 f26/fs10  ffffffff00000000 f27/fs11  ffffffff00000000
f28/ft8   ffffffff00000000 f29/ft9   ffffffff00000000 f30/ft10  ffffffff00000000 f31/ft11  ffffffff00000000
```

4 Hello World!

Instead of having to look at registers to confirm whether something is working or not, want to make it more obvious by printing something to the debug console.

4.1 Hello via SBI

To use functions by the SBI, we use the `ecall` function found in line (17) of Listing 8. But before that inline assembly code, we have to ask the compiler to place a few values in specified registers. Those registers are the ones reserved for the kernel. That is accomplished in lines (7) - (14) in Listing 8. Specifically, the `register` and `__asm__`("register name") does this.

When `ecall` is evoked, the CPU's execution mode switches from kernel mode (S-Mode) to OpenSBI mode (M-Mode) and OpenSBI's processing handler is invoked. Once its finished, it switches back to kernel mode and execution resumes after the `ecall` instruction.

```
1 #pragma once
2
3 struct sbiret {
4     long error;
5     long value;
6 };
```

Listing 7: (kernel.h) using `ecall`

```
1 #include "kernel.h"
2 #include "common.h"
3
4 extern char __bss[], __bss_end[], __stack_top[];
5
6 struct sbiret sbi_call(long arg0, long arg1, long arg2, long arg3, long arg4, long
    arg5, long fid, long eid) {
7     register long a0 __asm__("a0") = arg0;
8     register long a1 __asm__("a1") = arg1;
9     register long a2 __asm__("a2") = arg2;
10    register long a3 __asm__("a3") = arg3;
11    register long a4 __asm__("a4") = arg4;
12    register long a5 __asm__("a5") = arg5;
13    register long a6 __asm__("a6") = fid;
14    register long a7 __asm__("a7") = eid;
15
16    __asm__ __volatile__(
17        "ecall"
18        : "=r"(a0), "=r"(a1)
19        : "r"(a0), "r"(a1), "r"(a2), "r"(a3), "r"(a4), "r"(a5), "r"(a6), "r"(a7)
20        : "memory"
21    );
22    return (struct sbiret){.error = a0, .value=a1};
23 }
24
25 void putchar(char ch) {
26     sbi_call(ch, 0, 0, 0, 0, 0, 0, 1);
27 }
28
29 void kernel_main(void) {
30     memset(__bss, 0, (size_t) __bss_end - (size_t) __bss);
31
32     printf("\n\nHello %s\n", "World!");
33     for (int i = 0; s[i] != '\0'; i++) {
```

```

34     putchar(s[i]);
35 }
36
37 for(;;) {
38     __asm__ __volatile__("wfi");
39 }
40 }

```

Listing 8: (kernel.c) using ecall

In order to write to the console, need to use the OpenSBI call `Console Putchar`, which puts a character to the console. To make the call, we select function 0 and extension 1 (by putting 0 in register `a6` and 1 in register `a7`) [5]. In addition, SBI functions must return a pair of values in `a0` and `a1` where the value in `a0` is the error code. This is why we have the `struct sbiret` definition in Listing 7. In addition, `Console Putchar` is guaranteed not to change registers `a2` to `a7`.

4.2 printf function

Now, we can implement the `printf` function which, under the hood, simply prints the characters supplied to the function (using `Console Putchar`). In this `printf`, we are only going to support 3 format specifiers:

1. `%d` (decimal)
2. `%x` (hexadecimal)
3. `%s` (string)

```

1 #pragma once
2
3 #define va_list __builtin_va_list
4 #define va_start __builtin_va_start
5 #define va_end __builtin_va_end
6 #define va_arg __builtin_va_arg
7
8 void printf(const char *fmt, ...);

```

Listing 9: (common.h) header file for printf

```

1 #include "common.h"
2
3 void putchar(char ch);
4
5 void printf(const char *fmt, ...) {
6     va_list vars;
7     va_start(vars, fmt);
8
9     while (*fmt) {
10         if (*fmt == '%') {
11             fmt++;
12             switch (*fmt) {
13                 case '\0': // at the end of the format string
14                     putchar('%');
15                     goto end;
16                 case '%':
17                     putchar('%');
18                     break;
19                 case 's': {
20                     const char *s = va_arg(vars, const char *);
21                     while (*s) {
22                         putchar(*s);

```

```

23     s++;
24 }
25 break;
26 }
27 case 'd': {
28     int value = va_arg(vargs, int);
29     unsigned magnitude = value;
30     if (value < 0) {
31         putchar('-');
32         magnitude = -magnitude;
33     }
34
35     unsigned divisor = 1;
36     while (magnitude / divisor > 9)
37         divisor *= 10;
38
39     while (divisor > 0) {
40         putchar('0' + magnitude / divisor);
41         magnitude %= divisor;
42         divisor /= 10;
43     }
44     break;
45 }
46 case 'x': {
47     unsigned value = va_arg(vargs, unsigned);
48     for (int i = 7; i >= 0; i--) {
49         unsigned nibble = (value >> (i * 4)) & 0xf;
50         putchar("0123456789abcdef"[nibble]);
51     }
52 }
53 }
54 } else {
55     putchar(*fmt);
56 }
57 fmt++;
58 }
59 end:
60     va_end(vargs);
61 }

```

Listing 10: (common.c) C file for printf

4.2.1 What are the builtin features?

In Listing 9, all of the `define` macros map the standard variadic-names to compiler intrinsics. The standard variadic names are typically defined in `<stdarg.h>` and allows functions to accept an indefinite number of arguments [1]. However, since we're writing our own kernel, `<stdarg.h>` is not available to use so instead we use Clang's built-ins.

1. `__builtin_va_list`: represents the internal structure that stores the argument pointer
2. `__builtin_va_start(ap, last)`: initializes a `va_list` to start reading args after the parameter `last`
3. `__builtin_va_arg(ap, type)`: Fetches the next argument from the list and converts it to `type`
4. `__builtin_va_end(ap)`: cleans up

The arguments need to be *variadic* because `printf` can take any number of format specifiers. The various `va_...` are used to loop through the arguments.

4.2.2 How are the builtin features used in printf?

Typically, function arguments are stored in registers. Since the number of arguments is variadic, the arguments are stored in memory, then loaded into the registers using the builtin features.

In Listing 10, line (6) declares a variable named `vargs` of type `va_list`. This stores the current position in the list of arguments but does not hold anything yet. Line (7) initializes the `va_list` so you can read arguments from it. It finds where the variable arguments begin in the stack/registers, sets `vargs` to point immediately after `fmt` and prepares `vargs` so that calls to `va_arg(vargs, type)` will fetch each additional argument. After each call to `av_arg(vargs, type)`, `vargs` is automatically incremented to "point" to the next variable.

5 C Standard Library

```
1 typedef int bool;
2 typedef unsigned char uint8_t;
3 typedef unsigned short uint16_t;
4 typedef unsigned int uint32_t;
5 typedef unsigned long long uint64_t;
6 typedef uint32_t paddr_t;
7 typedef uint32_t vaddr_t;
8
9 #define true 1
10 #define false 0
11 #define NULL ((void *) 0)
12 #define align_up(value, align) __builtin_align_up(value, align)
13 #define is_aligned(value, align) __builtin_is_aligned(value, align)
14 #define offsetof(type, member) __builtin_offsetof(type, member)
15
16 void *memset(void *buf, char c, size_t n);
17 void *memcpy(void *dst, const void *src, size_t n);
18 char *strcpy(char *dst, const char *src);
19 int strcmp(const char *s1, const char *s2);
```

Listing 11: new additions to common.h

A few useful macros (in Listing 11):

1. `paddr_t`: A type representing physical memory address
2. `vaddr_t`: A type representing virtual memory addresses
3. `align_up`: Round up `value` to the nearest multiple of `align` and `align` must be a power of 2
4. `is_aligned`: Checks if `value` is a multiple of `align`
5. `offsetof`: Returns the offset of a member within a structure (how many bytes from the start of the structure)

5.1 Memory operation

References

- [1] The Open Group. *The Open Group Base Specifications Issue 7, 2018 Edition (IEEE Std 1003.1-2017)*. 2018. URL: <https://pubs.opengroup.org/onlinepubs/9699919799/>.
- [2] Lu Pan. *Application Binary Interface from the Ground Up*. 2021. URL: <https://uvdn7.github.io/abi/>.
- [3] QEMU. *'virt' Genetic Virtual Platform*. 2025. URL: <https://www.qemu.org/docs/master/system/riscv/virt.html>.

- [4] Alex Bennee. *Anatomy of a Boot, a QEMU perspective*. URL: <https://www.qemu.org/2020/07/03/anatomy-of-a-boot/>.
- [5] Stephen Marz. *OpenSBI Calls*. URL: <https://courses.stephenmarz.com/my-courses/cosc562/risc-v/opensbi-calls/>.
- [6] Seiya Nuta. *OS in 1000 Lines*. URL: <https://operating-system-in-1000-lines.vercel.app/en/>.