Bachelor's Thesis

# RISC-V Bare-Metal Library Operating System for Selfie

by
Marcell Haritopoulos

Bachelor's thesis submitted to the
Paris Lodron University of Salzburg
Department of Computer Science
in partial fulfillment of the requirements
for the degree of Bachelor of Science

Supervisor: Univ.-Prof. Dr.-Ing. Christoph M. Kirsch

Salzburg, on February 2021

## Abstract

Despite its young age, the field of Computer Science is multifaceted and complex, at the expense of the entry barrier. Selfie, an educational project, aims to close the gap on the fundamentals. It runs on a hosted environment, but was not able to run on bare metal RISC-V hardware.

We introduce a simple library operating system for RISC-V that implements the system call interface of Selfie and supports (nearly) all applications, that are written in Selfie's tiny C subset `C*` and are using its syscall interface, without any modifications, including Selfie itself. The OS is linked to the application as a static library and is responsible for maintaining a C environment. System calls are performed as mere function invocations instead of raising an exception using `ecall`. Files are stored in a static read-only file system. I/O system calls are implemented in a general way so that related projects may share the same semantics.

# Contents

# 1 Introduction

The field of Computer Science is relatively young, yet it computing devices are omnipresent: From pacemakers and smartwatches to smartphones up to cloud computing. A lot of research has been conducted and many concepts and theories were introduced. Sadly, this also affects the barrier of entry for the discipline. Some educational projects aim to level this barrier by teaching the fundamentals of Computer Science. Selfie itself was meant to run as a hosted application on top of an operating system. We now wanted to port Selfie from an hosted environment to a quasi bare-metal environment, preferably with little to no changes to the main source itself, implemented as simple as possible.

In this bachelor's thesis, we present a small and simple library operating system for 64-bits RISC-V architecture. It implements all system calls necessary to host the majority of C* applications. We used established conventions and projects around the RISC-V ecosystem to ensure compatibility with future ISA implementations.

Before we can explain in detail how we realized the system, we first have to introduce concepts and projects relevant for this thesis.

## 1.1 Collaboration

This bachelor's thesis has been realized in close collaboration with Martin Fischer, BSc. This thesis introduces a common code base consisting of generalized I/O system functions, bootstrapping code, static in-memory file system and SBI interfacing as well as a shared build system. In his thesis, Martin discusses a preemptive kernel that builds upon this base and implements virtual memory and preemptive context switching support. Selfie's syscall interface is implemented using environmental calls. Compiled binaries emitted by Selfie are supported by the kernel's ELF loader. It is recommended to read Martin's thesis to gain a better understanding of RISC-V's privileged architecture and to deduce how both implementations differ.

## 1.2 The Selfie project

Selfie[1] is an educational project of the Computational Systems Group[2] of the Computer Science department at the University of Salzburg. It is used in undergraduate and graduate courses to teach the basics of operating systems, compilers, emulators and hypervisors [cf. 10]. More specifically, in ~11.2k lines of code in a single file, Selfie features:

- C*, a tiny subset of the C language and an accompanying library called `libcstar`. C*'s goal is to maintain code simplicity and readability while keeping the complexity of the self-referential compiler implementation low. The library contains functions for bit- and string manipulation, number-string conversions as well as formatted printing functions [cf. 11, p. 199+204]

[1]`https://selfie.cs.uni-salzburg.at/`
[2]`http://www.cs.uni-salzburg.at/~ck/`

- RISC-U, a tiny subset of the RV64IM ISA. It contains 14 instruction necessary to archive self-execution: `lui, addi, sd, ld, add, sub, mul, remu, divu, sltu, beq, jal, jalr` and `ecall` [cf. 10]
- `starc`, a self-compiling compiler from C* to RISC-U [cf. 10]
- `mipster`, a self-executing RISC-U emulator [cf. 10]
- `hypster`, a self-hosting hypervisor providing virtual RISC-U machines on behalf of an upper instance (even for an upper hypster instance) [cf. 10]
- A small operating system that proxies system calls `open`, `read`, `write`, `exit` and `brk` to the boot level below (or the hosting OS on boot level 0) [cf. 10]
- A profiler, a disassembler and a conservative garbage collector[3],

At the time of implementing the OS, Selfie relied on integral type `uint64_t` and required a 64-bit machine. Starting with commit `6af2320`, Selfie is able to run on 32-bit systems and emulate 32-bit machines, but as these changes have been merged recently, we have not considered them in this document and build upon the previous constraints.

## 1.3 Library Operating System

A library operating system, sometimes called *libOS*, are libraries "that hide low-level resources behind traditional operating system abstractions"[6, p. 11]. Applications that build upon a libOS do not perform system calls using an instruction to jump to a higher-privileged trap handler, but rather perform ordinary calls into and are linked against an (optimized) libOS [cf. 6, p. 11f].

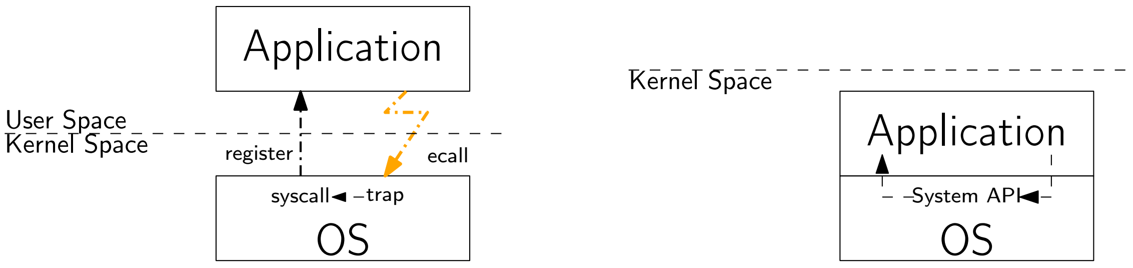


Figure 1.1: Application-OS interfacing of usual (left) and library (right) OS'

The library OS that is linked to the application may be tailored to the latter which may ease complexity or increase performance. At link-time, the linker may be able to perform better optimizations on the library OS binary, like moving symbols in such a way that the smallest possible addressing mode may be used. Due to the OS and the application being linked together, they reside in the same address space.

[3]Gregor Bachinger, "Conservative Garbage Collection in Kernel and Mutator Space": `https://github.com/cksystemsteaching/selfie/blob/master/theses/bachelor_thesis_bachinger.pdf`

Library OS' also benefit from compile-time static type-safety. [cf. 12, p. 1ff]

Engler introduced the *exokernel* architecture in his paper. Its goal is to multiplex and protect system resources for (untrusted) libOS applications while allowing them to manage the resources [cf. 6, p. 11].

Madhavapeddy et al. introduced a variant of library OS', *unikernels*, that tackles their hardware compatibility issues by targeting a standard hypervisor. It is intended for use in cloud computing [cf. 12, p. 1].

## 1.4 Hosted and Freestanding Environment

The C standard differentiates between two different execution environments: hosted and freestanding. [cf. 9, p. 12]

Regardless of execution environment, both environments must ensure that static data is initialized when the corresponding initial function is called. On "program termination", the execution environment regains control [cf. 9, p. 12].

A freestanding environment must be provided by a conforming C implementation. The startup and termination routine [cf. 17, p. 6] as well as the name and signature of the initially called function is implementation-specific[cf. 9, p. 12]. Only a small subset of the standard headers is deemed required, namely "`<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>` and `<stdnoreturn.h>`"[9, p. 8].

A hosted environment is not required, but must abide constraints on program startup and execution, if provided. On program startup, the hosted execution environment is expected to call function `main` which expects either no or two arguments, `int argc` and `char* argv[]`, and returns int[cf. 9, p. 13]. If `argc` and `argv` were declared, they must also meet the following constraints:

- `argc` must be non-negative
- `argv[argc]` should be a `NULL` pointer
- If `argc` is greater than 0, `argv[0]` should represent the application name, while all upper `argv` strings shall represent arguments.
- `argc`, `argv` and the argument strings should be modifyable by the application

[cf. 9, p. 13]

On termination, if `main` returns an int-compatible value, the hosted execution environment behaves as if `exit` with the given exit code has been passed. If the end of main has been reached without any return statement, value 0 is implied. If the return value is not int-compatible, the exit code is unspecified[cf. 9, p. 14].

"A conforming hosted implementation shall accept any strictly conforming program"[9, p. 8], including the entire set of C standard functions, macros, typedefs and objects[cf. 9, p. 13].

The hosted environment is meant for applications that are *hosted* by an operating system, while an application in the freestanding environment “may take place without any benefit of an operating system”[9, p. 12]. For kernel development, freestanding environment shall be used.

## 1.5 RISC-V

RISC-V is a free and open Instruction Set Architecture (ISA) which originated from University of California, Berkeley as a research and educational project but is now hoped to be established as a free and open standard industry ISA. Not only should it be fully virtualizable, but should also be capable of being implemented natively in hardware. It is implemented in a microarchitecture- and implementation technology-agnostic way, i.e. it is not optimized for a specific combination, but should be realized efficiently with any combination [cf. 20, p. 1ff].

RISC-V enables variable length instruction encoding. As of writing this thesis, 16-bit and 32-bit instruction length are frozen (with a subset of the 32-bit encoding space reserved for longer-sized instruction prefixes), while instruction lengths up to 176 bits are described but not frozen. [cf. 20, p. 8f]

The ISA is designed in a modular way. A implementation consists of a mandatory base ISA (RV32I, RV64I base integer ISAs, RV32E reduced embedded ISA, draft RV128I integer ISA) and optional standard or vendor extensions[cf. 20, p. 4ff+149ff]. The base ISA consists of a minimal set of instructions that provide a “reasonable target for compilers, assemblers, linkers, and operating system”[20, p. 4] and allows to specialize the ISA for a use case with the addition of necessary extensions[cf. 20, p. 4].

### 1.5.1 Boot Process

Upon power-on, the usual boot process on RISC-V consists of five stages:

- Boot ROM:
  Runs from on-chip rom and initializes critical facilities like clocks or power sources [cf. 13, p. 6].

- Loader:
  Runs from on-chip SRAM. It is responsible for setting up the DDR memory and loading the next stage(s) before passing on control [cf. 13, p. 6].

- Runtime:
  Runs from DDR memory and provides runtime services. It may also provide SoC security features or a secure enclave [cf. 13, p. 6].

- Bootloader:
  Runs from DDR memory. A bootloader is the last full-blown stage that provides support for more advanced boot options than the tiny loader stage, including file system and networking support. It also provides facilities to configure boot parameters [cf. 13, p. 6].

- OS:
  Runs from DDR memory [cf. 13, p. 6]. This is the actual operating system that manages resources and providing a sane abstraction above the machine [cf. 18, p. 3ff].

The Boot ROM and the bootloader are responsible for loading (and optionally validating) their respective next stages and pass control to them. The loader does not only load the runtime, but is also responsible for loading the bootloader. Control is then transferred to the runtime, which initializes itself and then transfers control to the bootloader [cf. 13, p. 6].

The first three stages run in M-mode [cf. 13, p. 6], the most privileged mode for "usually inherently trusted"[19, p. 3] code with low-level access to the machine [cf. 19, p. 3]. Before passing control to the bootloader and subsequently the OS, the runtime demotes the privilege level to S-mode [cf. 13, p. 6].

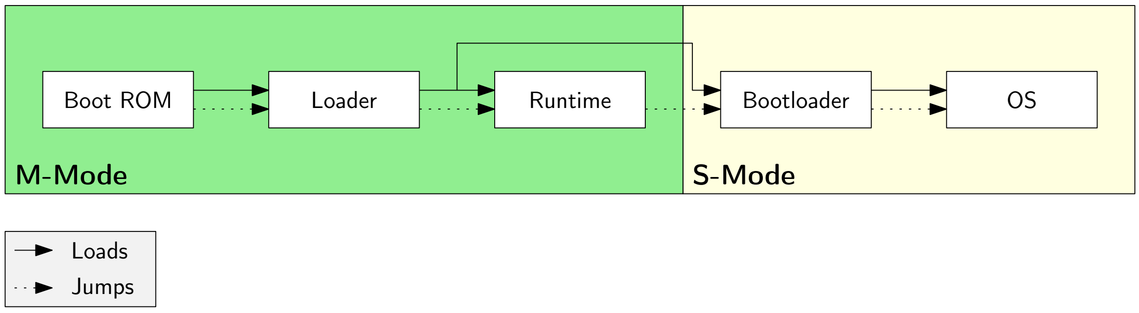


Figure 1.2: Boot stages[4]

Typically, there are two ways to include the runtime into the firmware: Either by embedding it into the loader (e.g. into EDK2, an open-source UEFI implementation), or by just seperately loading it [cf. 13, p. 14+19]. In the latter case, the runtime has three ways to jump to the next stage:

- The runtime may be merged with the (payload) bootloader stage, so that the loader only must load one single binary and the offset to which the runtime must jump to is known at link-time. Unfortunately, this means that the runtime-bootloader package has to be rebuilt if any of these components change [cf. 13, p. 15].

- Instead, the loader and the runtime may use the same address to load the next stage and, after initialization, jump to. Using this strategy, the bootloader may be replaced independently from the runtime and vice-versa [cf. 13, p. 16].

- The loader may provide dynamic booting information by preparing data for the runtime before jumping into it. This allows the loader to alter the runtime parameters like the next stage's address. While it makes the boot more flexible, it adds complexity to the loader because it must be aware of the data to pass to the runtime [cf. 13, p. 17].

[4]Based on the figure from [13, p. 6]

### 1.5.2 SBI / OpenSBI

The RISC-V Supervisor Binary Interface (SBI) is a calling convention between an Supervisor and a Supervisor Execution Environment (SEE), where a SEE can either be an a M-mode runtime firmware for an OS or hypervisor in (H)S-mode, or the aforementioned HS-mode hypervisor for a guest OS in VS-mode. It provides functionality to write portable RISC-V supervisor software, i.e. kernels and hypervisors, and access to hardware resources only accessible on M-mode privilege level [cf. 13, p. 4].

Like RISC-V itself, the SBI specification is organized in a modular way: The functionality of the interface is divided into separate extensions which themselves may host functions. Each extension and function has a signed 32-bit identifier. A mandatory base extension provides information about the SBI implementation and the machine, while optional extensions may extend the SEE. The calling convention described in the SBI specification mandates that supervisors must provide the extension ID in register `a7` and may additionally provide the function ID in `a6`, if the extension defines any. An `ecall` is performed to transfer control to the elevated runtime. The SEE returns an status/error code in register `a0` and a function-specific value in register `a1` [cf. 4].

OpenSBI is an open-source reference implementation of a M-mode runtime that fully implements the SBI specification. It consists of a platform-agnostic `libsbi.a` that implements the interface and hooks into a per-platform static library `libplatsbi.a` that implements platform-specific functions[5]. OpenSBI can be compiled into a standalone binary that is loaded by the loader or may be linked into the loader. It supports all three ways described in section 1.5.1 to jump to the next stage, named `FW_PAYLOAD`, `FW_JUMP` and `FW_DYNAMIC`, respectively [cf. 14].

### 1.5.3 Code Models

Memory for the symbols are allocated by the toolchain on link-stage. To interact with them, loads and stores to their corresponding memory regions must be generated. On RISC-V, being a load-store architecture, "only load and store instructions access memory and arithmetic instructions only operate on CPU registers" [20, p. 24]. Because it is expensive to implement addressing modes in hardware[cf. 3], RISC-V's load-store instructions do only support a single hardware-implemented mode: `base + offset`. The address to load from / store to is calculated by adding the value of the `base` register to the 12-bit `offset` (signed) immediate [cf. 20, p. 24+67]. More advanced addressing modes may be realized in software by the toolchain by combining other instructions.

"The code model determines which software addressing mode is used" [3] and determines the software addressing model emitted by the compiler and later on patched by the linker when the actual symbol address can be determined. Each code model poses different limitations on the code, though.

The RISC-V psABI defines three different code models: Small, medium and compact [cf. 5].

[5] `https://github.com/riscv/opensbi/blob/v0.7/include/sbi/sbi_platform.h#L74-L161`

The *small* code model relies on the `lui` instruction [cf. 5] to load the upper 20 bits of the 32 bit address into a register using an immediate [cf. 20, p 19+36]. The lower 12 bits are provided either on load-store as offset value or using `addi` when calculating the actual address is required [cf. 5] (e.g. referencing or runtime array indexing). On RV32I, the entire 4 GiB physical address space is addressable. On RV64I, the `lui` instruction sign-extends the immediate's most-significant bit to the upper 32 bits [cf. 20, p 36]. As this limits the immediate address to 31 bits plus sign bit, it is only possible to address the lowest 2 GiB (`0x0000_0000_0000_0000` ... `0x0000_0000_7FFF_FFFF`) and the highest 2 GiB (`0xFFFF_FFFF_8000_0000` ... `0xFFFF_FFFF_FFFF_FFFF`) on the 64-bit architecture. Due to using fixed link-time immediate values, the code model is not position-independent.

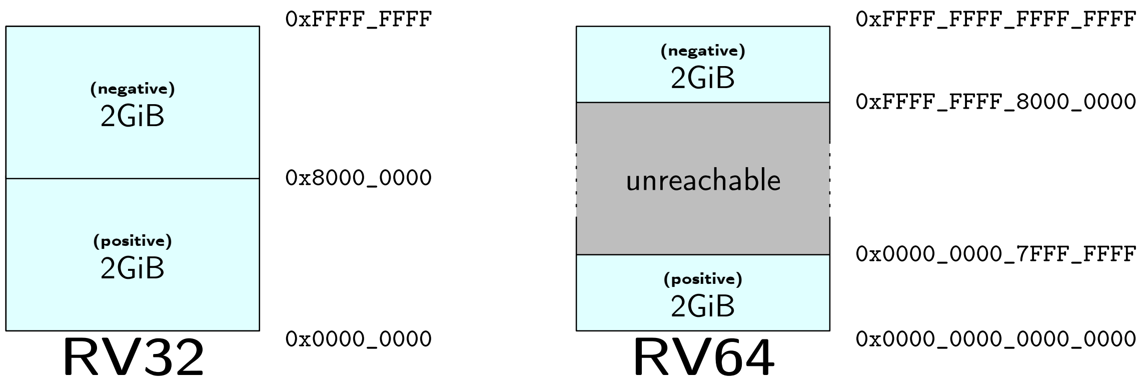


Figure 1.3: Small code model

The *medium* code model does not rely on (fully) fixed immediates, but instead calculates a symbol's position relative to the program counter [cf. 5]. Instruction `auipc` (Add Upper Immediate to Program Counter) loads the current value of the PC into a destination register. It is possible to specify a signed 32-bit offset from the current PC: The upper 20 bits (sign-extended in RV64I to the upper 32 bits) may be passed to `auipc` while the lower 12 bits, as in the small code model, may be provided to load-store instructions or `addi` [cf. 20, p. 19+36f]. Due to the 32-bit signed immediate offset, the application may address symbols that are positioned in a $\pm 2$ GiB window relative to the current PC.

Due to address calculation being relative to the program counter instead of relying on fixed link-time immediates, the medium code model is position independent.

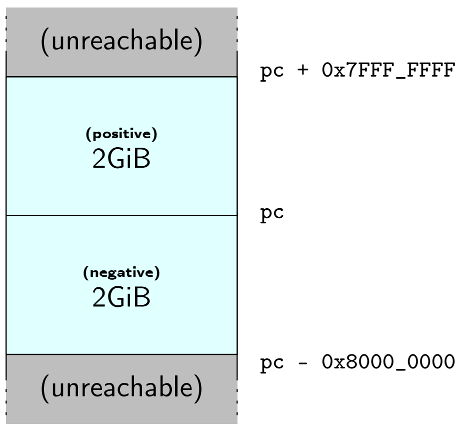


Figure 1.4: Medium code model

The *compact* code model is necessary for applications whose symbols are farther apart than the medium model can support. This memory model is able to address the entire 64-bit address space. Instead of relying on 32 bit offsets on `lui`/`auipc` and load/store/`addi`, the linker maintains a Global Offset Table (GOT) which stores a 64-bit offset to the symbol. The entry in the GOT itself is referred to by a signed 32-bit offset relative to the global pointer (gp). The offset is loaded using a `lui`-`addi` pair. Because the entry to the GOT is gp-relative and the GOT entry being an offset by itself, the compact code model is position independent [cf. 5].

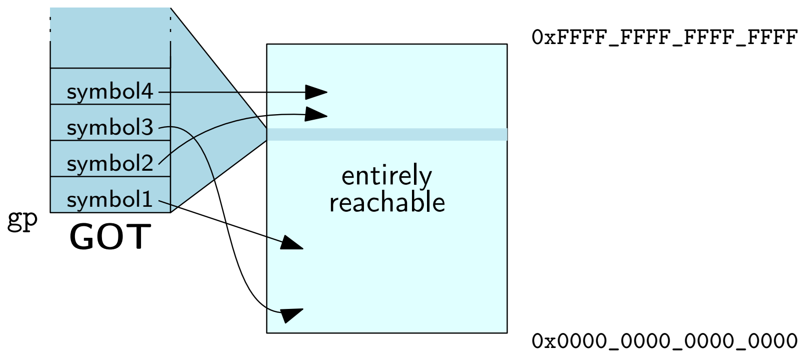


Figure 1.5: Compact code model

GCC and clang both support the small and medium code models by specifying flags `-mcmodel=medlow` or `-mcmodel=medany`, respectively. By default, the low code model is used for any base architecture [6] [cf. 17, p. 377].

### 1.5.4 Linker Relaxation

While the code models allow the compiler to generate code capable of addressing a large region of memory, it is not always necessary to have such large offsets if a symbol is nearby. Linker relaxation is an optimization that aims to reduce required instructions to address symbols [cf. 1, p. 2]. As the name suggests, it is performed at link time, contrary to typical optimization methods that are performed at compile

[6] `https://clang.llvm.org/docs/ClangCommandLineReference.html#riscv`

time[cf. 2].

At compile time, the compiler cannot resolve the the address a symbol will be placed in and instead emits instructions according to the code model and marks them for relocation. The relocation information contains the type of relocation and the symbol involved. Additionally, these relocations are marked as eligible for relaxation using type `R_RISCV_RELAX`[cf. 1, p. 3f].

At link time, the linker positions code and data, and their associated symbols, to their final locations. At this point, it is possible to patch the instructions marked for relocation with the actual offsets. Additionally, for relocations marked for relaxation, the linker asserts whether the offset can be encoded for more compact instructions with equivalent semantics. If this succeeds, the linker replaces the initially compiled instruction(s) (as specified by the code model) with the shortest instruction that can still represent the offset. If no compact representation is possible, only relocation is performed [cf. 2].

To archive better relaxation success than just with program counter-relative relaxation, the psABI reserves one register to act as a `global pointer`. The linker allocates a section called `.sdata` (small data) and on program startup, the global pointer must be initialized to “0x800 bytes past the start of the .sdata section”[2] and is treated like a constant register. It is necessary to add 0x800 to the GP because load-store and `addi` instructions allow to specify a **signed** 12-bit offset. By moving the GP up, all data up to the beginning of `.sdata` may be addressed GP-relative [cf. 2].

The linker allows to relax five different relocation types:

- `R_RISCV_CALL` and `R_RISCV_CALL_PLT` relocations for calls ( `auipc` and `jalr`)
- `R_RISCV_HI20` and `R_RISCV_LO12_I`/`R_RISCV_LO12_I` for *small* code model absolute offset relocations on I-type and S-type instructions (`lui` and `addi`/`ld`/`sd`/...)
- `R_RISCV_PCREL_HI20` and `R_RISCV_PCREL_LO12_I`/`R_RISCV_PCREL_LO12_I` for *medium* code model PC-relative offset relocations on I-type and S-type instructions (`auipc` and `addi`/`ld`/`sd`/...)
- Thread-local storage references
- `R_RISCV_ALIGN` for GNU assembler `.align` directives in `.text` section.

[cf. 2]

Including the compressed ISA extension, the linker may try to emit relaxed instructions using a combination of U-, I-, S-, CI-, CR-, CIW-, CL- and CS-type instructions[cf. 2]. While the *normal* instructions are 32 bits long, the compressed instructions are only 16 bits long, but are accordingly constrained: CR- and CI-type are limited to 8 addressable destination registers. Immediates are limited in length and are sometimes scaled[cf. 20, p. 16+99f].

Thus, with a single code model and the ISA’s only addressing mode, it is possible to generate four different addressing modes with variable length:

- 2 bytes: symbol within -64..63 bytes from GP (or 0) (signed 7-bits offset)

- 4 bytes: symbol within ±2 MiB from GP (or 0) (signed 12-bits offset)
- 6 bytes: symbol within ±64 MiB from PC (or 0) (signed 17-bits offset)
- 8 bytes: symbol within ±2 GiB from PC (or 0) (signed 32-bits offset)

[cf. 2]

Using linker relaxation, RISC-V archives variable length addressing, which "is usually something reserved for CISC processors"[2] by the use of hardware-implemented addressing modes.

```
build/qemu/library/bootstrap_library.o:     file format elf64-littleriscv


Disassembly of section .text:

[...]

000000000000000e <kernel_environ_init>:
   e:   1141                    addi    sp,sp,-16
  10:   e406                    sd      ra,8(sp)
  12:   e022                    sd      s0,0(sp)
  14:   0800                    addi    s0,sp,16
  16:   00000097                auipc   ra,0x0
                        16: R_RISCV_CALL        initial_stack_start
                        16: R_RISCV_RELAX       *ABS*
  1a:   000080e7                jalr    ra # 16 <kernel_environ_init+0x8>
  1e:   87aa                    mv      a5,a0
  20:   873e                    mv      a4,a5
  22:   00000797                auipc   a5,0x0
                        22: R_RISCV_PCREL_HI20  heap_head
                        22: R_RISCV_RELAX       *ABS*
  26:   00078793                mv      a5,a5
                        26: R_RISCV_PCREL_LO12_I        .L0
                        26: R_RISCV_RELAX       *ABS*
  2a:   e398                    sd      a4,0(a5)
  2c:   0001                    nop
  2e:   60a2                    ld      ra,8(sp)
  30:   6402                    ld      s0,0(sp)
  32:   0141                    addi    sp,sp,16
  34:   8082                    ret

[...]
```

Listing 1.1: Excerpt from `riscv64-elf-objdump -dSr bootstrap_library.o` before linker relaxation

```
[...]

0000000080200f7c <kernel_environ_init>:
    80200f7c:   1141                    addi    sp,sp,-16
    80200f7e:   e406                    sd      ra,8(sp)
    80200f80:   e022                    sd      s0,0(sp)
    80200f82:   0800                    addi    s0,sp,16
    80200f84:   8c2ff0ef                jal     ra,80200046 <initial_stack_start>
    80200f88:   87aa                    mv      a5,a0
    80200f8a:   873e                    mv      a4,a5
    80200f8c:   49018793                addi    a5,gp,1168 # 80289c90 <heap_head>
    80200f90:   e398                    sd      a4,0(a5)
    80200f92:   0001                    nop
    80200f94:   60a2                    ld      ra,8(sp)
    80200f96:   6402                    ld      s0,0(sp)
    80200f98:   0141                    addi    sp,sp,16
    80200f9a:   8082                    ret

[...]
```

Listing 1.2: Excerpt from `riscv64-elf-objdump -dS selfie.elf` after linker relaxation

## 1.6 Tested Platforms

As the main goal of this thesis was to get Selfie running on a bare metal environment, we wanted to test our kernels on real RISC-V silicon implementations, not just software emulators or programmed FPGAs. Due to Selfie's constraints, a 64-bit processor with Multiplication Extension is mandatory. The preemptive kernel also mandates the use of a processor that features all three privilege levels (M, S and U) as well as Sv39 virtual memory support. We did look for pre-assembled development boards instead of a plain chip because we wanted to focus on the software instead of electronics.

The RISC-V Organization maintains a list of available boards[7] and SoCs[8] on their website. Most cores did not qualify because they were either RV32I, FPGA implementations, or were not available on a development board. After researching available SoCs, we decided to test the implementations on two hardware platforms:

The HiFive Unleashed is "the first RISC-V-based, Linux-capable development board"[9]. It is powered by the FU540 SoC which has 4 IMADFC cores and a IMAC core as "monitor core"[10]. The board features 8 GiB memory. The HiFive Unleashed has been discontinued in favor of its successor, HiFive Unmatched. Unfortunately, both boards are expensive.

The Sipeed MAIX bit is a small, affordable board featuring a dual-core Kendryte

[7]`https://riscv.org/exchange/`

[8]`https://riscv.org/exchange/cores-socs/`

[9]`https://www.cnx-software.com/2018/02/04/sifive-introduces-hifive-unleashed-risc-v-linux-develop`

[10]`https://sifive.cdn.prismic.io/sifive%2F834354f0-08e6-423c-bf1f-0cb58ef14061_fu540-c000-v1.0.pdf`

K210 clocked at 400 MHz[11]. Both cores implement the IMAFDC extensions, have 32 KiB I-cache and D-cache and are backed by 8 MiB shared memory[12]. Due to difficulties with the vendor's flashing tool `kflash`, it is recommended to use a version modified by GitHub user `loboris`, `ktool`[13]. The SoC peripherals are not well documented by its datasheet, but unofficial documentation is available by GitHub user `laanwj`[14].

Furthermore, we opted to test our implementation on QEMU with 128 MiB memory. QEMU, being a emulator, is immediately and freely available to everybody who is interested in trying out our implementation. Furthermore, it provides debugging support courtesy of its `gdbstub` feature[15].

All three platforms are fully supported by the library operating system.

# 2 Implementation

The library operating is implemented as a tiny layer between the machine and a C* application. Because it implements Selfie's system call interface, the OS is compatible with all C* applications that do not depend on `stdin` or file write operations without modifications to them, including Selfie.

The OS requires a working M-mode runtime that exposes the SBI interface and provides base and legacy extensions. The OS has been tested with OpenSBI and the build system enables the OS to be packaged as payload to an OpenSBI build, if the loader cannot load the two stages separately. Because we do not need the flexibility provided by a bootloader, we omitted this boot stage for less dependencies and complexity. The library OS should not be functionally affected when it is loaded by a bootloader instead.

The library OS itself and the linked application both run in S-mode. We do not use any paging mechanism as physical addressing is sufficient in our case.

The system calls are implemented in such a way that it can be used for both the library OS and the preemptive OS by the mere addition of little glue code.

## 2.1 Compilation & Linkage

For typical hosted applications using hosted toolchains, compilation is as easy as specifying the source files and the desired output file name. The toolchain has already been configured to comply to an platform-specific ABI and include system libaries and linker scripts. For freestanding kernel development, manual configuration is mandatory. Like other larger software that is non-trivial to build, a build system is beneficial to use so that compilation, dependencies and generation of resources can be orchestrated.

This section introduces the project's build system and which specific compiler configuration is required to build it.

[11]`https://www.seeedstudio.com/Sipeed-MAix-BiT-for-RISC-V-AI-IoT-p-2872.html`
[12]`https://s3.cn-north-1.amazonaws.com.cn/dl.kendryte.com/documents/kendryte_datasheet_20181011163248_en.pdf`
[13]`https://github.com/loboris/ktool/tree/0345aa90`
[14]`https://github.com/laanwj/k210-sdk-stuff/tree/b847235/doc`
[15]`https://wiki.qemu.org/Features/gdbstub`

### 2.1.1 Build System

To fit into the existing build system of Selfie, we use GNU make for our build system. A RV64 toolchain (hosted `riscv64-linux-gnu-` or freestanding `riscv64-elf-`), a corresponding binutils package, the host's toolchain, `gettext` and `xxd` are expected in the search path. While the Makefile does use GCC by default, it is possible to use clang with target triple `riscv64-elf`. Furthermore, `curl` and `tar` are required if the binaries shall be packaged with OpenSBI. Optionally, QEMU may be used to test builds made for it.

The Makefile maintains three lists of source files: `SBI_WRAPPER_SRCS_COMMON` lists the sources that are shared across both OS targets, while variables `SBI_WRAPPER_SRCS_LIBRARY` and `SBI_WRAPPER_SRCS_KERNEL` represent additional source files for the library OS and the preemptive OS, respectively.

Both kernel variants are organized in build "profiles" that consist of the profile name, list of sources and macro definitions for the build. For both profiles, macro `KERN_TYPE` is set to the macro's name. Make macro `add-build-profile` adds profile variables and inserts the profile into list `ALL_PROFILES`. While it is possible to distinguish profiles in source, they should be used sparingly for profile-specific code in common source files and use other abstractions instead.

To support different platforms with diverse capabilities and properties, the Makefile stores platform configuration in `board` definitions. It keeps track of the board name, loader jump address, OpenSBI `FW_PAYLOAD` offset and the corresponding platform name in OpenSBI. The last two values are mandatory only if OpenSBI `FW_PAYLOAD` packaging is needed. Macro `add-board` adds board variables and inserts the board into list `ALL_BOARDS`.

```
$(eval $(call add-board,qemu,0x80000000,0x200000,qemu/virt))
$(eval $(call add-board,fu540,0x80000000,0x200000,sifive/fu540))
$(eval $(call add-board,k210,0x80000000,0x1A000,kendryte/k210))

$(eval $(call add-build-profile,kernel,$(SBI_WRAPPER_SRCS_COMMON)
↪   $(SBI_WRAPPER_SRCS_KERNEL),KERN_TYPE=kernel))
$(eval $(call add-build-profile,library,$(SBI_WRAPPER_SRCS_COMMON)
↪   $(SBI_WRAPPER_SRCS_LIBRARY),KERN_TYPE=library))
```

Listing 2.1: Profile and board configuration

Using variables `ALL_BOARDS` and `ALL_PROFILES`, make macro `add-all-targets` dynamically creates build rules and evaluates them to put them in effect.

For each board and profile combination, the Makefile generates three build outputs: The main compile output `selfie-`*`{board}`*`-`*`{profile}`*`.elf` is created by linking all object files and applying the linker script. It is intended for when the loader does explicitly load the image alongside a runtime image. Some loaders may not be able to handle ELF files and expect the executable as fat binary. Target `selfie-`*`{board}`*`-`*`{profile}`*`.bin` takes the emitted ELF file and emits a fat binary using `objcopy`. If the loader is not capable of loading the runtime and the next stage separately, it is necessary to bundle the application with OpenSBI as runtime.

Target `selfie-opensbi-{board}-{profile}.elf` takes the fat binary output and configures an OpenSBI build with mode `FW_PAYLOAD`, the board's configured offset and the path to the binary as payload.

Object files are not shared, each board-profile combination rebuilds the full source set. This is required due to potential different behavior courtesy of conditional builds using board-profile defines.

Additionally to the build targets, the Makefile defines some pseudo targets:

- `debug` adds compile flag `-g3` to enable full debugging information, embedded into the ELF[cf. 17, p. 131f], and defines `DEBUG` macro for conditional code, e.g. for debugging output. GDB allows for using this information either on-chip with OpenOCD[16] or virtually with QEMU.

- Pseudo target `list-targets` creates a list of all targets and collective pseudo targets generated by all possible board-profile configurations.

- Pseudo target `test` builds both kernel variants for the `qemu` platform and tries to execute both build outputs using `qemu-system-riscv64` with 128 MiB memory. If the binaries run successfully, they instruct QEMU to terminate. If it is not successful, QEMU may not exit and needs to be terminated using `[Ctrl]+[A] [X]`.

- Pseudo target `opensbi` is responsible for downloading and inflating OpenSBI v0.7 sources. The target need not be used directly, but are declared as order-only prerequisite for the targets packaging OpenSBI

- Pseudo target `clean` removes all build artifacts, while `distclean` additionally removes the OpenSBI sources.

### 2.1.2 Compiler Flags

The Makefile sets the default compiler flags to `-mabi=lp64d -march=rv64imafdc -mcmodel=medany -ffreestanding -Iinclude -Wall -Wextra -MMD -MP`

As described in section 1.5, RISC-V is structured in a modular way. To specify the ISA extensions the compiler may utilize, flag `-march` is used. All three tested implementations are `RV64IMAFDC` (= `RV64GC`), which is why `-march=rv64imadfc` is used. For architecture `RV64G`, the psABI specifies that the default ABI shall be `LP64D`, meaning that `long`s and pointers are 64 bits wide while `int`s are 32 bits wide. Floating points up to 64 bit width are passed in registers according to the hardware floating point calling convention, whenever possible[cf. 5]. Selfie itself only requires the `M` multiplication extension atop the base `RV64I` instruction set due to its self-referential nature, thus it would suffice to use architecture `rv64im` and ABI `LP64`.

`-mcmodel=medany` is responsible for compiling the library OS with the *medium* code model, as described in section 1.5.3. The default `medlow` code model may be sufficient for application code hosted in its own virtual address space and a fixed entry point, but is not feasible for the library OS. The lower addresses of physical

[16]`https://github.com/riscv/riscv-openocd/tree/v2018.12.0`

memory are usually reserved for memory-mapped peripheral registers. On all three tested platforms, main memory starts at address `0x8000 0000`[17] [18] [19], which makes it impossible to use `medlow`. Furthermore, due to `medany` calculating the symbol address in a PC-relative way, binaries are position independent. The loader may put the binary to a different address and communicate the jump address to OpenSBI using `FW_DYNAMIC`, as described in section 1.5.1.

The kernel's header files are stored in directory `include`. Flag `-Iinclude` adds this directory to the search path[cf. 17, p. 230f].

GCC features a large amount of warnings that may prevent unwanted behaviors. To enable most reasonable warnings, flags `-Wall -Wextra` are used[cf. 17, p. 78ff].

Flags `-MMD -MP` cause the preprocessor to additionally emit Makefile rules that reflect dependencies influencing the resulting object file. For the kernel, this usually includes the relevant source file and nested header includes[cf. 17, p. 218].

`-ffreestanding` instruct the compiler to compile the source in freestanding mode, as described in section 1.4.

### 2.1.3 Linker Flags

The linker flag build upon the compilation flags (not all compile flags are effective for linkage, though) and extend them with `-nostdlib -lgcc -Wl,--build-id=none -T selfie.ld` by default. The Makefile uses `$(CC)` for linkage. GCC and Clang pass their invocation further down to `riscv64-elf-ld` or `riscv64-linux-elf-ld`.

For hosted systems, the linker defaults to including some system libraries, like the C standard library, and startup files that prepare an environment and initialize the standard library by default. As they are tailored to a hosted system, flag `-nostdlib` is required to instruct the linker [cf. 17, p. 226ff] not to include them into the build. While the system libraries are not strictly necessary, it is required to provide custom startup code. This is described in section 2.2.1.

GCC features a low-level runtime library, `libgcc.a`, which it generates calls into "whenever it needs to perform some operation that is too complicated to emit inline code for"[16, p. 9]. It consists of integer, fixed point and soft floating point functions for processors that are lacking some operations. `libgcc.a` is also excluded from linking by flag `-nostdlib` and must be added explicitly to prevent unresolved symbol errors with flag `-lgcc` [cf. 16, p. 9].

Additionally, the compiler may also create calls to C standard library functions `memcpy`, `memmove`, `memset` and `memcmp`, required e.g. for struct copying, and must therefore be implemented by the freestanding code [cf. 17, p. 6]. Source file `tinycstd.c` provides a minimal and functionally incomplete subset of the C stdlib (and BSD extensions) for the kernel, including the functions required by GCC, `strlen`, `strncmp`, `strchr`, `strlcpy`, `printf`, `puts` and `putc`.

Per default, the non-freestanding (`riscv64-linux-gnu-` prefix) linker embeds a "build ID" into a separate binary section "that is always the same in an identical

[17]`https://github.com/qemu/qemu/blob/stable-5.0/hw/riscv/virt.c#L67`
[18]`https://github.com/laanwj/k210-sdk-stuff/blob/0445bc55/doc/memory_map.md`
[19]`https://sifive.cdn.prismic.io/sifive/b5e7a29c-d3c2-44ea-85fb-acc1df282e21_FU540-C000-v1p3.pdf`, p. 35

output file”[20]. It is placed as the first section of the binary if not differently specified by the linker script, breaking the fat binary output. To prevent the build ID to be inserted into the binary, linker flag `--build-id=none` has to be used [20].

### 2.1.4 Linker script

Flag `-T selfie.ld` instructs the linker to use the specified linker script.

For each board-profile combination, the script is generated because it depends on the board configuration. The template is located at `payload_template.ld` and expects variables `$SBI_START` and `$PAYLOAD_OFFSET` which are replaced using gettext tool `envsubst`.

```
$$(TARGET_$(1)_$(2)_DIR)/selfie.ld:
	SBI_START=$$(BOARD_$(1)_PAYLOAD_START)
	↪ PAYLOAD_OFFSET=$$(BOARD_$(1)_PAYLOAD_OFFSET) envsubst
	↪ '$$$$SBI_START$$$$PAYLOAD_OFFSET' <payload_template.ld
	↪ >$$@
```

Listing 2.2: Generation of linker script in `defines.mk`

The linker script template has been adapted from OpenSBI[21] as well as the default linker script shipped with `riscv64-elf-binutils`[22].

Initially, the script declares that RISC-V binaries shall be emitted and the entry point is symbol `_start`. Then, the section ordering is specified and linker-defined symbols are declared:

In the beginning, the script sets the current position to the board’s entry point and offset mandatory for OpenSBI. Section `.text` is then composed of the entry point and the rest of `.text`. Next, constant `.rodata` and initialized `.data` sections follow. Section `.sdata` is the *small* variant of the `.data` section and comes after the latter. This section hosts symbols that shall be accessible using GP-relative addressing instructions. Symbols `__global_pointer$` (for the linker) and `__SDATA_BEGIN__` (for the bootstrapping code) are both initialized to the beginning of `.sdata` plus 0x800. Directly after the this section, small `.bss`, `.sbss` follows. Like the small data section, this section hosts symbols that will need GP-relative and thus shorter addressing instructions.

The script defines symbols `_`*`<section>`*`_start` and `_`*`<section>`*`_end` for all sections that denote their respective ranges. Furthermore, symbols `_payload_start` and `_payload_end` contain the first and the last address of the binary, respectively.

The size of all sections (except `.sdata` and `.sbss`) are aligned to word size. Also, sections `.text`, `.rodata`, `.data` and `.bss` are aligned to page boundaries (each of

[20]`https://sourceware.org/binutils/docs-2.35/ld/Options.html#Options`
[21]`https://github.com/riscv/opensbi/blob/v0.7/firmware/fw_base.ldS`
[22]Generated on build by `https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=ld/emulparams/elf32lriscv-defs.sh;h=bc464918;hb=7e46a74aa;` installed at `/usr/riscv64-elf/lib/ldscripts/elf64lriscv.x`

these sections start a new page). The small sections do not start a new page because they must be tightly kept together as to not waste any GP-relative memory.

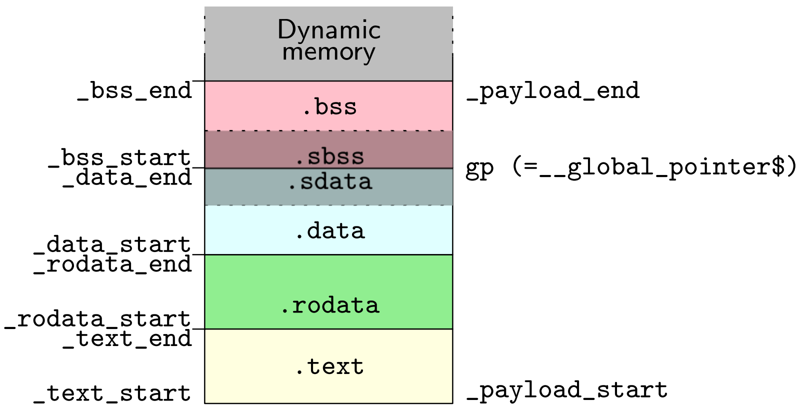


Figure 2.1: Section positioning and linker symbols within binary

## 2.2 Bootstrapping

While booting, the loader loads the library OS into memory, depending on the platform either as separate binary or combined with OpenSBI as payload. In any case, OpenSBI passes control to the next stage, that is our library OS, which is then responsible for setting up an environment for the application.

### 2.2.1 ASM entry point and C runtime setup

Initially, execution of the library OS starts at symbol `_start`, which is defined in file `asm/crt.S`. It is responsible for setting up an environment for C code and initializing basic kernel objects and afterwards passing execution into the kernel's C entry point. Parts of OpenSBI's assembler entry point were adapted [23].

#### 2.2.1.1 Global pointer

Before any symbol can be used, the global pointer must be initialized. The global pointer is assigned to register `x3` [cf. 5] and is determined at link time. The linker populates symbol `__SDATA_BEGIN__` with the value required for the global pointer. To exclude the set-up code itself from linker relaxation, assembler directive `.option norelax` must be specified before the load is performed [cf. 1, p. 7]. Directives `.option push` and `.option pop` are used to beforehand save and afterwards restore previous assembler configuration, respectively. Failing to set the global pointer correctly before referencing a symbol may result hard-to-detect errors due to rogue access for *some* symbols.

#### 2.2.1.2 BSS

Statically allocated, but uninitialized memory resides in the `.bss` section. This section is not materialized in the executable but its start and end address are specified. The linker script defines symbols `_bss_start` and `_bss_end` for this purpose. After

[23] `https://github.com/riscv/opensbi/blob/v0.7/firmware/fw_base.S#L48`

the kernel has been loaded, the content of the section is undefined, thus it is the bootstrapping process' responsibility to zero the region. This is done by initializing registers a4 and a5 with the start and end address, respectively. On the address given by a4, dword zero is written into memory. a4 is then incremented by 8 bytes and compared against a5. If the former is lower than the latter, there is still memory to zero and it branches to the store instruction, otherwise it progresses the program counter by the size of the instruction.

#### 2.2.1.3 Trap Handling

All user- and supervisor-level interrupts are disabled and cleared using registers sip (**S**upervisor **I**nterrupt **P**ending) and sie (**S**upervisor **I**nterrupt **E**nable)[cf. 19, p. 58f]. Timer interrupts are not necessary due to the library OS being a cooperative OS and external interrupts are not required because no other peripherals on the die are used. A trap handler is installed to symbol hang_machine into register stvec [cf. 19, p. 57] regardless to gracefully handle unexpected exceptions like access faults. As the two least significant bits are used for the vector mode, the trap handler must be 4-byte aligned[cf. 19, p. 57f], which can be forced by assembler directive .p2align 2 (power-two alignment) [24].

#### 2.2.1.4 Memory Layout

Bootstrapping code also needs to set up the memory layout before entering the C entry point. Usually, the loaded application image, that is its .text (binary code), .(ro)data (initialized data) and .bss, are located in the lower parts of the address space. As the heap grows upwards and the stack grows downwards, they are placed at the initial program break and the end of the address space, respectively, so that both regions may arbitrarily grow. Unfortunately, it is not trivially possible to determine how much memory a device features so that the highest physical (RAM) memory address can be determined. On embedded systems, a device tree is used to describe peripherals of a device that may not be dynamically discoverable. Usually, a "boot program" (bootloader, hypervisor) is responsible for providing a reference to the device tree to a "client program" (kernel, chained bootloader, hypervisor) [cf. 7, p. 3-6]. Relevant in this context is that at least one /memory node is required and are used to describe addresses and associated sizes of physical memory. However, to keep the library OS simple, we decided against implementing a flattened device tree parser to determine the physical memory layout and instead fixated the stack's size. As the heap is still dynamically growing, we opted to place the fixed-size stack between the kernel's .bss section and the heap. Macro NUM_STACK_PAGES determines the amount of 4 KiB pages that are reserved for the stack. ASM function uint64_t* initial_heap_head() is used to calculate the stack's highest address by adding the size of the stack (NUM_STACK_PAGES * $2^{12}$) to linker-defined symbol _payload_end. Up to 1024 64 bit words can be stored in the stack with the default size. As specified by the RISC-V psABI, a descending full stack[25] is used, thus the initial stack pointer has to point to the first non-stack address[cf. 5]. Due

[24] https://sourceware.org/binutils/docs/as/P2align.html#P2align

[25] http://www-mdp.eng.cam.ac.uk/web/library/enginfo/mdp_micro/lecture5/lecture5-4-2.html

to the heap being the last section in the memory layout, it may grow dynamically until it runs out of (contiguous) physical memory and causing an access fault.

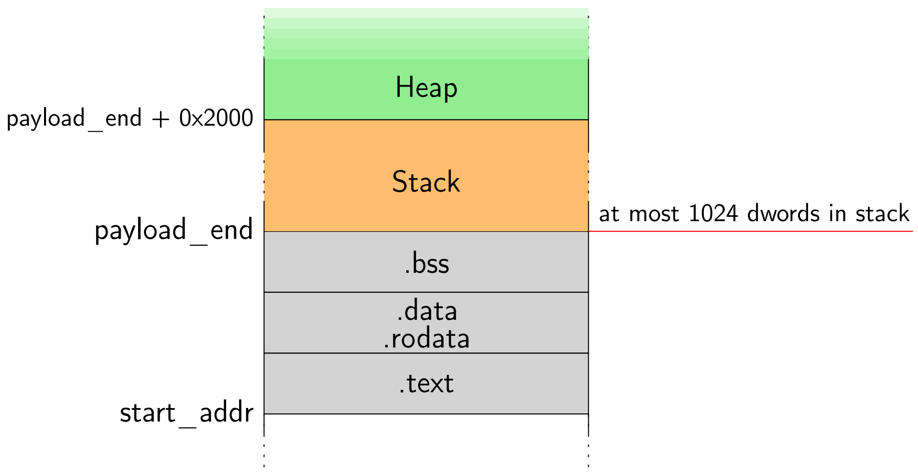


Figure 2.2: Memory layout of the library OS

### 2.2.2 C kernel entry

The kernel's C entry performs further initialization before passing control to the linked application. As necessary initialization procedures and process creation are distinct, the generalized bootstrapping code calls into three functions that must be defined by variant-specific bootstrapping code:

- `void early_init()`:
  This function is called before any generalized initialization takes place. It may be used to set up kernel objects and mechanisms that may be necessary for further initialization, e.g. kernel memory allocators or trap handlers. The library OS' bootstrapper defines an empty function.

- `void kernel_environ_init()`:
  This function is called after all generalized initialization took place and shall be used to put the kernel into an fully operational state, ready to accept processes. The library OS' bootstrapper uses this function to initialize the program break.

- `int start_init_process(uint64_t argc, const char** argv)`: This function is called after the kernel has been fully initialized and shall start its init process. For the library OS, this function merely calls function `main` with given argument count and vector and returns its result. No further preparations are necessary as no contexts, process isolation or image loading, due to the application already being linked to the kernel, are involved.

#### 2.2.2.1 Passing control and arguments to the application

The argument vector that is passed to the application cannot be read from the console. Instead, it is provided at compile-time. As required by the C standard, the last element of the argument vector array must be a `NULL` pointer [cf. 9, p. 13].

After the application returns from `main`, the bootstrapping code emits the exit code and performs a shutdown using the legacy SBI function `void sbi_shutdown(void)` [cf. 4].

## 2.3 Static file system

In order to support I/O operations `read`, `write` and `open` on multiple files, the operating system requires file system support. Selfie requires read file access to perform meaningful operations. However, Selfie itself does not necessarily require support for file write operations as it may use compiled binaries directly for emulation.

Usually, OS' organize access to storage in different layers:

1. Physical hardware (mass storage)
   Usually, the primarily used storage medium are secondary storage, e.g. hard disks and nonvolatile memory [cf. 15, p. 449]. These secondary storage are connected to the machine via a system or I/O bus. Each storage device contains a device controller that is responsible for actually driving the hardware and performing the I/O operations while abstracting away the complexity into a more comprehensible (standardized) interface. It contains device-specific logic and also handles error cases [cf. 18, p. 28f] [cf. 15, p. 456], e.g. moving the read-write head to the correct position in the case of HDDs or wear leveling and managing the bad blocks reserve in the case of SSDs.

   On the host side, a "host-bus adapters", attached to the host's bus, are responsible for communication with the device controller. Commands from the OS are sent to the host-bus adapter, which forwards the commands to the device controller where the command is then executed [cf. 15, p. 456].

2. Interrupt Handler
   The controllers may execute commands asynchronously, without blocking the CPU, and raise an interrupt if data is ready, the command finished or an error occurred. After an interrupt has been raised, the CPU catches it and passes control to the interrupt handler[cf. 15, p. 494ff], which dumps the machine state, handles the interrupt, unblocks the initiator of the command and schedules the next context [cf. 18, p. 356ff]. Because they are awkward to handle, interrupts should be isolated from the rest of the system so that "as little of the operating system as possible knows about them"[18, p. 356].

3. Controller driver
   The memory-mapped registers and I/O ports that are used to control the device as well as the command set may differ widely: A SATA device expects other commands to perform a read than a microSD card [cf. 18, p. 358].

   To hide the hardware differences from the upper I/O layer and keep it general, we want to provide a common ground, a standardized interface. Concrete implementations of this interface may be provided in the form of device drivers. [cf. 15, p. 501ff] While the system calls may differ between operating systems, there is a common definition of a type characteristic: "block I/O, character-stream I/O, memory-mapped file access, and network sockets" [15, p. 503]

Naturally, each of these different devices or device classes require device-specific drivers [cf. 15, p. 358].

4. Device-Independent I/O Subsystem
   Building upon the generalized driver interfaces, the device-independent part of the I/O subsystem is able to provide services. Among others, it is responsible for scheduling, buffering, error handling and power management [cf. 15, p. 508ff].

Using the I/O subsystem, it is possible to perform operations on persistent secondary storage, in particular block read and write operations. However, raw I/O operations are cumbersome to perform, especially on large systems with many users and process with lots of information. For example, it is necessary to coordinate where information are located and how they can be found, provide a way to protect files of different users or to determine which blocks are free. To solve these, another layer of abstraction is applied atop the I/O subsystem: the file system, which is the part of the OS that manages a collection of "files" [cf. 18, p. 264], is responsible of mapping them to physical storage location and providing a (generalized) interface for them [cf. 15, p. 529ff]. Files, one of the most important OS concepts, "are logical units of information created by processes"[18, p. 264]. They are the smallest unit of logical secondary storage and their content may take any form [cf. 15, p. 530].

Both in-hardware platforms that were used feature an microSD card slot to extend the board with secondary storage. While some simplifications can be applied to the I/O layers, like dropping interrupt handling in favor of polling and keeping the device-independent subsystem minimal by dropping power management, buffering and scheduling in favor of reading affected blocks on-demand and discarding them after returning the requested data and using blocking FIFO scheduling, it would still introduce a lot of complexity for an minimal educational system. In our case, we would have to implement a driver for the SPI bus controllers of both hardware platforms, a SD card driver (over SPI) and a file system. For QEMU, we would also have to implement an additional block device driver, e.g. "Virtio over MMIO"[26].

To keep complexity of the file system implementation low, we decided against write support and implemented a read-only file system. Furthermore, the file system is contained inside the binary. Before the boot chain can transfer execution control to the next stage (e.g. chained bootloader, supervisor or kernel), it must be transferred into memory from non-volatile storage first. As part of pre-initialized data, the compiler puts the content of packaged files into the `.rodata` section and will therefore be loaded by the one of the previous boot stages. So instead of performing block read operations on non-volatile storage ourselves, we take advantage of a stage's built-in loading capability and use it to load all files into memory at once.

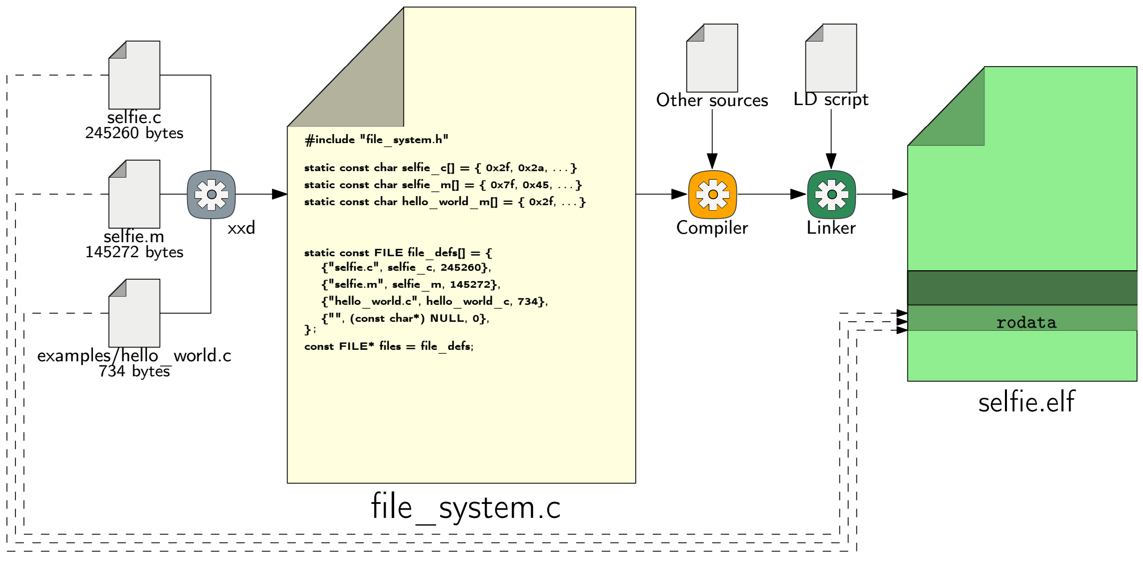
[26]`https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.pdf`, p. 53

### 2.3.1 Packaging

The build system contains targets related to the static file system in the `filesystem.mk` sub-makefile. It defines a rule for file `files_package.c` which is declared as prerequisite source for the library OS.

The target `files_package.c` declares prerequisites to the contents of the `$(SBI_WRAPPER_FILES_PACKAGE)` variable which contains the path to all files that shall be included into the static file system. The files are represented as `static const char[]` arrays in the resulting file. To accomplish this, the Makefile uses the `xxd -i` command/flag combination to emit a hexadecimal dump of the file in the form of a C byte array and appends its output to the output file. The file array's identifier is determined by the base file name with all "non-word" characters replaced by underscores (`sed 's/\W/_/g'`).

Also, the Makefile generates an array of `file descriptions` named `files` which encapsulates all file arrays a file's base name, pointer to the previously generated data array as well as the size of the data array, in bytes. The array augments the file arrays with metadata and represents the glue between the data and the file system functions. The file description array is terminated by an empty item `{"", (const char*) NULL, 0}` (empty file name, 0 bytes long, data pointer set to `NULL`).



```
#include "file_system.h"

static const char selfie_c[] = { 0x2f, 0x2a, ...}
static const char selfie_m[] = { 0x7f, 0x45, ...}
static const char hello_world_m[] = { 0x2f, ...}


static const FILE file_defs[] = {
    {"selfie.c", selfie_c, 245260},
    {"selfie.m", selfie_m, 145272},
    {"hello_world.c", hello_world_c, 734},
    {"", (const char*) NULL, 0},
};
const FILE* files = file_defs;
```

Figure 2.3: Filesystem packaging process

As previously stated, the file descriptions contain a file's base name instead of a path. Three other options for the file name were considered but discarded:

- **Absolute path**:
  Using this approach, it is possible to store in-repo and out-of-repo files. Unfortunately, this approach does not produce stable names as the path depends on the build directory location.

- **Nearest common ancestor:**
  Instead of representing the full absolute path, the nearest common ancestor of

each file to package is determined. The description's file name is determined starting from this common ancestor. But, like before, the description name is still not stable as the name may be extended or truncated when files are added or removed to the files package.

- **Relative path - based on repository root**:
  The description's file name is based on selfie's repository path. While this approach produces stable file names and allows multiple files with the same name, files out-of-repo require special handling to prevent names containing double dots and the selfie repository's path must be determined.

The decision for the base name was made to simplify the packaging process as well as access to said files. Contrary to all other approaches, the chosen approach waives the possibility to package two files with the same name in favor of simplicity.

Pseudo target `clean-fs-target` removes the generated `files_package.c` file. It is meant to be used as an prerequisite for the `clean` and `distclean` pseudo targets.

### 2.3.2 File Access

Source files `filesystem.h / filesystem.c` provides a single function `const KFILE* find_file(const char* filename)` to find file descriptions based on its name by linearly iterating over the `files` array. If the file has been found, the function returns the description. If the end of array has been reached, `NULL` will be returned.

Available files may iterated by directly accessing the `files` array.

This function and global array are sufficient to support the read-only file system.

## 2.4 System calls

Selfie exposes system calls `exit`, `read`, `write`, `openat` and `brk`. In order to support C* applications, the library OS must implement these calls as well.

### 2.4.1 Generalized I/O

The generalized I/O interface is a layer between the OS' syscall interface and the read-only static file system. It is responsible for finding files that were requested to be opened, keeping track of the read progress as well as forwarding written data to the serial.

As aforementioned in chapter 1.1, effort was made to generalize the implementation of I/O system call functions. Both OS implementations use this generalized interface by using some glue code. This ensures that both the library OS and the preemptive OS share the same I/O codebase and thus expose the same behavior.

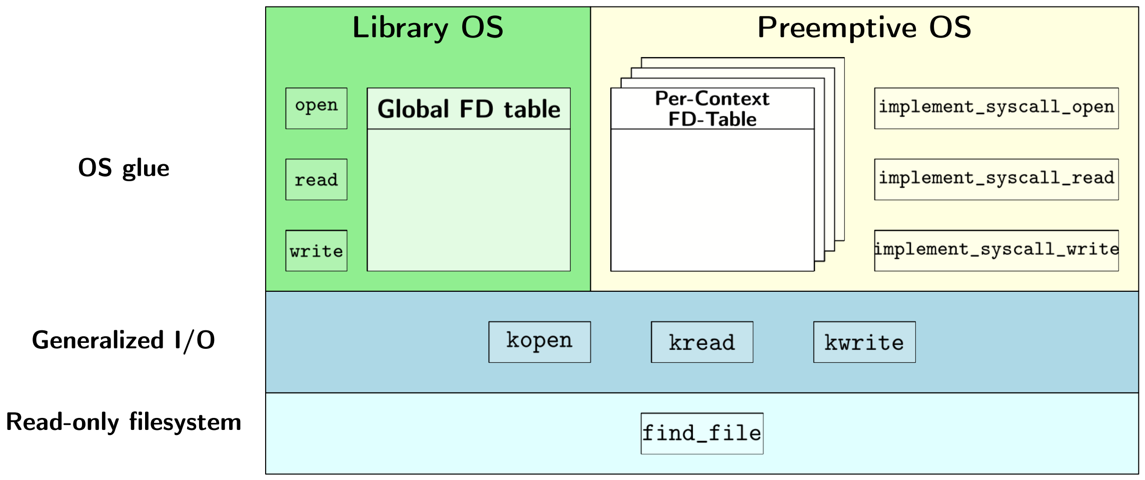


Figure 2.4: Layers of the I/O system calls

The generalized functions are prefixed by "k" (`kopen`, `kread`, `kwrite`) and take two additional parameters: a pointer to a file descriptor array, `open_files`, as well as the size of the array, `num_fds`.

```
ssize_t kread(int fd, char* buf, size_t count, FILEDESC*
↪   open_files, size_t num_fds);
ssize_t kwrite(int fd, const char* buf, size_t count, FILEDESC*
↪   open_files, size_t num_fds);
int kopen(const char* filename, int flags, FILEDESC* open_files,
↪   size_t num_fds);
```

Listing 2.3: Generalized system call declarations from `syscalls.h`

The file descriptor `struct` contains necessary context information, that is, a reference to the opened file pointer as well as the current seek position in file, in bytes.

#### 2.4.1.1 kopen

Function `kopen` is responsible for finding a file in the read-only file system as well as allocating and initializing an unused file descriptor slot.

The function tries to find the name wit the specified path using `find_file`. If the file does not exists, it returns `NULL` and prompts `kopen` to return -1. From the passed file descriptor list, `kopen` tries to find the next free slot and returns its index. If no file descriptor slots are available, the operation fails.

After a free file descriptor slot has been found, the function populates the fields of the `FILEDESC` structure. Field `file` is refers to the previously found file reference in the read-only file system, while file `pos` is set to 0 to start reading from the beginning of the file.

```c
typedef struct FILEDESC {
  const KFILE* file;
  size_t pos;
} FILEDESC;
```

Listing 2.4: Definition of `struct FILEDESC`

POSIX declares that the file descriptors of the standard streams `stdin`, `stdout` and `stderr` shall be defined as 0, 1 and 2, respectively [8, p. 2017]. These file descriptor IDs are never populated by the implementation.

Before doing its work, the function validates the flag combination that has been passed. Write operations are not supported and yield a negative return code.

Selfie does not feature a preprocessor and thus cannot use a host's header files. Instead, it uses fixed constant 32768 (0x8000) as `open` mode on all OS'. On Windows, macOS and Linux, `O_RDONLY` is 0. Additionally, Windows requires `O_BINARY` flag (0x8000) to be set because it differentiates between text files and binary files whereas UNIX and unixoid systems treat all files like binary files and process them as-is. On macOS, this flag has no effect. On Linux, flag `O_LARGEFILE`[27] occupies this bitmask. It is used to support files larger than 2 GiB, but does not have any effect on read-only operation. Like on all other OS', Selfie will pass open flags 0x8000 to our OS. Thus, our implementation must accept the flag instead of returning an error. The generalized I/O layer does not distinguish between text and binary files and will ignore the flag, if set.

All access modes other than `O_RDONLY` and `O_BINARY`/`O_LARGEFILE` (e.g. `O_WRONLY` and `O_RDWR`[28]) and all file creation flags (e.g. `O_CREAT`, `O_APPEND`, `O_TRUNC`, ...) are declined and prompts our implementation to return `-1`.

#### 2.4.1.2 kread

Function `kopen` provides the facilities to read from previously opened files similar to the POSIX definition of `read` for regular files [8, p. 1771].

Upon entry, the function checks whether the passed file descriptor is in a valid range and whether it has been opened. If at least one condition does not apply, -1 is returned to denote an error.

At first, the function determines how much bytes of data is left to read from the file by subtracting the current descriptor position from the length of the file. If requested amount of bytes is greater than the left count of bytes to read, the former is set to the value of the latter to prevent out-of-bounds reads. The data is copied by using function `memcpy` (as described in section 2.1.3 above). After the copying process has finished, the position of the descriptor is increased by the amount of bytes copied.

If any of the standard file descriptors is passed to the function, -1 is returned because reads from the serial interface has not been implemented and is not used

[27] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/asm-generic/fcntl.h?h=v5.10#n51`

[28] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/fcntl.h?h=v5.10#n9`

by Selfie without support for special files (e.g. `/dev/stdin`) that may be given as command line argument. Selfie itself does not use `STDIN_FILENO`.

#### 2.4.1.3 kwrite

Using the serial interface, an application can provide feedback to the user by emitting messages.

The shared OS base defines `intmax_t console_puts(const char* str, size_t len)`, a function that takes a string buffer and prints `len` characters to the serial output. Each individual character is emitted by calling `void sbi_console_putchar(int ch)`, defined in the RISC-V SBI specification 0.2.0 [cf. 4].

Function `kwrite` checks whether the passed file allows write operations. As files on the static file system may only be opened in read-only mode, these must not be written to and -1 will be returned. If the file descriptor is any of the standard file descriptors, the data will be passed to function `console_puts`. This is because usually all three file descriptors refer to the same file, namely `/dev/pts/0`, on Linux.

### 2.4.2 I/O glue code

The actual system call interface performs calls to the generalized I/O interface and provides the file descriptor array to use.

The glue code must only do necessary steps to perform the actual code to the generalized functions as to not modify the common operation's semantics.

The generalized I/O functions assume physical addressing and thus require further consideration. The preemptive OS switches to a kernel identity virtual address space before invoking system calls, so all pages allocated to user processes are mapped to the kernel's address space. The buffer address provided by the process is relative to the process' virtual address space, requiring address translation to kernel space. Because the read/write buffer may cross page frame boundaries due to pages only being assigned on initial access, the syscall implementation has to split calls to the generalized functions in such a way that memory of potential foreign page frames is not overridden.

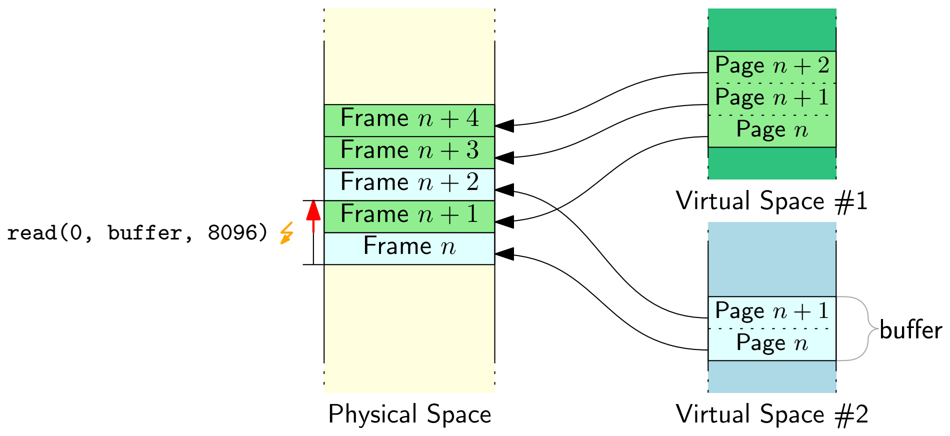


Figure 2.5: Invalid handling of I/O system calls

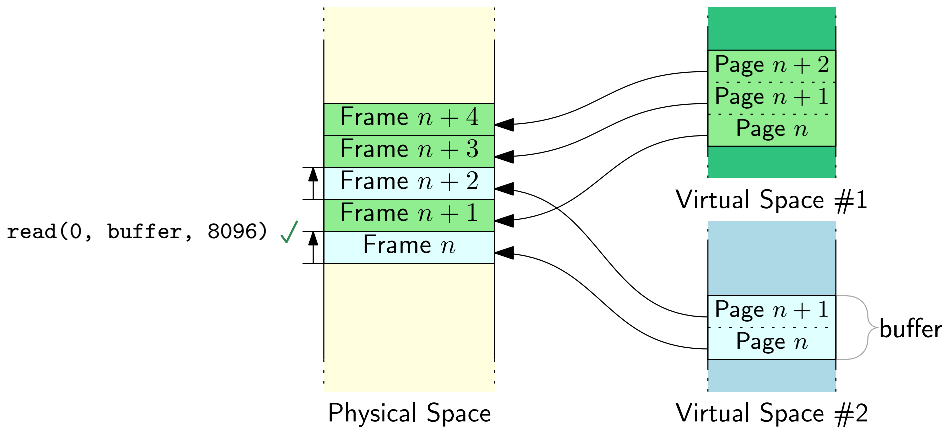


Figure 2.6: Page-aware handling of I/O system calls

Due to the library OS not being dependent on a virtual addressing mechanism, its glue code can forward the calls to the generalized functions without further considerations.

For each process, the glue code must provide a distinct array of file descriptors. For single-process kernels like our library OS, a global array passed to each call is sufficient.

The amount of file descriptor slots and thus the count of simultaneously opened files is specified by `#define NUM_FDS` in file `config.h`.

```c
static FILEDESC open_files[NUM_FDS];
void* heap_head;

ssize_t read(int fd, char* buf, size_t count) {
  return kread(fd, buf, count, open_files, NUM_FDS);
}

intmax_t write(int fd, const char* buf, size_t count) {
  return kwrite(fd, buf, count, open_files, NUM_FDS);
}

uint64_t open(char* filename, uint64_t flags, uint64_t mode) {
  UNUSED_VAR(mode);
  return kopen(filename, flags, open_files, NUM_FDS);
}
```

Listing 2.5: Library OS I/O glue

### 2.4.3 exit

`exit` is a system call that "stops" execution of the linked application by passing control back to the kernel and never returning back to the user code. The exit code cannot be queried by a "parent" process, as there is no concept of processes in the library OS, and will be emitted to the serial interface solely as information for the user.

After the linked application signaled its intention to exit, the library OS itself has no meaningful code left to execute and will try to shut down the system using SBI function `void sbi_shutdown()`. Depending on the platform, this call may have no effect and simply return. QEMU's implementation of OpenSBI provides a shutdown function that terminates the emulator, whereas FU540 and K210 shutdown functions are no-ops [cf. 14].

In case `sbi_shutdown` returns, it is not possible to gracefully shut the system down. Instead, the system call hangs the system by keeping it in an infinite `wfi` (wait for interrupt) loop until the user physically powers the system down.

```c
void shutdown() {
  sbi_ecall_sbi_shutdown();

  printf("shutdown failed - hanging machine...\n");
  hang_machine();
}
```

Listing 2.6: Function `shutdown()` from `diag.c`

```asm
  .section .entry, "ax", %progbits
  .p2align 2
  .global hang_machine
hang_machine:
  wfi
  j hang_machine
```

Listing 2.7: ASM symbol `hang_machine` in `asm/crt.S`

### 2.4.4 malloc / brk

For non-trivial applications, static and dynamic stack allocation do not suffice to operate. On one hand, for static allocation it is necessary to specify the size that shall be reserved in the `.bss` segment at compile time. The user may pass arbitrary data via the argument vector or files and if the memory required depends on the input data, it is infeasible to pre-allocate at compile time. An important aspect of our execution environment is the stack. Besides storing the return address and previous frame pointer to accomplish nested function calls, memory for local variables (functions and scopes) is reserved on the stack. C99 and some compiler-specific language extensions allow dynamically sized array in the form of *flexible array member* [cf. 9, p. 115].

However, C* does not allow flexible array members, and local variables cannot be passed outside of their scope, thus dynamic stack allocation is not sufficient as well. Also, due to the fact that Selfie itself heavily depends on heap allocations to allocate its compilation and emulation structures, it is necessary to implement dynamic allocation on the heap.

Usually, two parts are necessary to accomplish dynamic heap *allocation* :

- The library function `malloc` is responsible for returning a pointer to a contiguous block of memory large enough to meet the specified size. Usually, an allocator algorithm is used to keep heap fragmentation low but still responding to requests reasonably fast. If the heap ran out of space, the allocator requests more heap space from the OS.
- This is accomplished by system call `brk`. It expects a pointer as argument and modifies the program break accordingly. If the new program break is greater than the current one, the system must assign memory to the process' address space to assure enough memory is available to cover the program break.

To keep the implementation simple we used a bump allocator, that is it increases the program break by the amount of requested bytes and returns a pointer to the previous value of the program break. Allocation is possible in constant time and allocated memory is densely packed if no memory is freed.

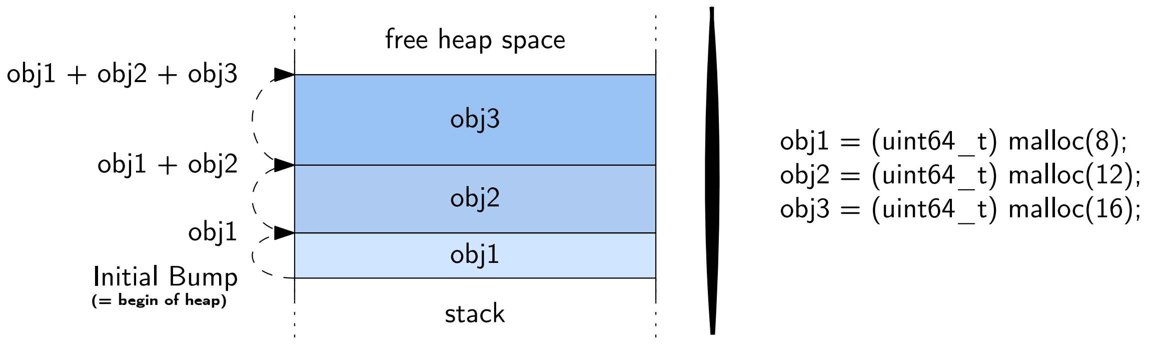


Figure 2.7: Bump pointer allocation

Because the library OS uses physical addressing, it is not necessary to manage virtual memory via a `brk` system call - our malloc implementation also covers increasing the program break accordingly.

Like Selfie, the OS does not support `free`ing the memory previously allocated by `malloc`. This leads to deliberate memory leaks that simplify the implementation. Selfie itself does not explicitly free `malloc`'d memory, but may instead use its conservative garbage collector to reclaim memory [29].

## 3 Experimental Evaluation

In order to get a feel how the library OS compares to a full-fledged hosted environment, we evaluated the amount of instructions that are executed on the library OS versus the same task on a linux-hosted binary linked against the glibc.

[29]`https://github.com/cksystemsteaching/selfie/blob/8ef40b71/theses/bachelor_thesis_bachinger.pdf`

We tested Selfie on commit `06cd083` in three different configurations:

- An Intel i7-7700HQ machine (GCC 10.2.0-4, glibc 2.32-5)
- RV64 Debian on QEMU (GCC 10.2.1-3, glibc 2.31-9)
  Based on the 2021-01-10 10:21:28 image from the Debian Quick Image Baker[30]. Unlike recommended by the documentation, QEMU's included OpenSBI[31] has been used and the kernel/initramfs were provided directly without using S-mode U-Boot using the following invocation:

  ```
  qemu-system-riscv64 -machine virt -cpu rv64 -m 1G
    -device virtio-blk-device,drive=hd
    -drive file=image.qcow2,if=none,id=hd
    -device virtio-net-device,netdev=net
    -netdev user,id=net,hostfwd=tcp::2222-:22
    -kernel kernel -initrd initrd
    -object rng-random,filename=/dev/urandom,id=rng
    -device virtio-rng-device,rng=rng -nographic
    -append "root=LABEL=rootfs console=ttyS0"
  ```

- The library OS on QEMU (GCC 10.2.0-1)

The count of executed instructions were recorded using GDB. For both linux-hosted environments, the usual system debugger was used, which is capable of stepping through the target's user space, including glibc (`malloc` and the syscall wrappers), but excluding kernel space and thus system calls. For the library OS, a cross debugger running on the host was attached to QEMU's GDB stub, which is able to step though the entire virtual machine's memory, including OpenSBI.

On all three variants, instructions beginning from symbol `_start` until symbol `exit_selfie` were recorded. On the library OS, this means that kernel initialization and system calls are included in the metric. However, functions `sbi_ecall_sbi_putchar` and `zero_mem` were excluded from the statistics, the former because it is merely a C convenience wrapper for an `ecall` into OpenSBI and we do not want to include its runtime services as much as we did not want to include Linux in the hosted environments, the latter due to the fact that Linux does provide zeroed pages to a process if it has been freshly allocated (i.e. not been free'd before) and zeroing memory in doubleword-granularity increases the instruction count dramatically.

Also, Selfie function `zero_memory` has been emptied. This function is used by `zalloc` to zero out memory allocated by the OS. Linux does zero newly allocated memory (but not reused memory; this point is irrelevant for us due to missing `free` calls) as well as our library OS, so additional zeroing by Selfie is not necessary and will significantly increase the instruction count and duration to measure it.

All binaries were built with the highest optimization level, `-O3`, and debugging information mandatory for symbol positions, `-g`. Additionally, on the hosted systems, binaries with shared and static linkage to glibc were built. Usually, the binaries are

[30]`https://people.debian.org/~gio/dqib/`

[31]`https://wiki.qemu.org/ChangeLog/5.1#RISC-V`

built with shared linkage, but the statically linked binaries are more similar to what we have with the statically linked library OS.

The flags were explicitly added to the make invocation to the `CFLAGS`, i.e. `make "CFLAGS=-Wall -Wextra -O3 -g -m64 -D'uint64_t=unsigned long long'" selfie` for shared linkage and `make "CFLAGS=-Wall -Wextra -O3 -g -m64 -D'uint64_t=unsigned long long' -static" selfie` for static linkage and Selfie's Makefile and `make "CFLAGS=-mabi=lp64d -march=rv64imafdc -mcmodel=medany -ffreestanding -Iinclude -Wall -Wextra -MMD -MP -O3 -g" selfie-qemu-library.elf` for the OS' Makefile. The tests were recorded with arguments `-c examples/hello-world.c -m 1`.

The changes to the repository in order to clear Selfie's function `zero_memory` and to provide the correct CLI arguments from the library OS' bootstrapping process were archived by the following patch:

```diff
diff --git a/machine/bootstrap.c b/machine/bootstrap.c
index c2779d3..d0aef2b 100644
--- a/machine/bootstrap.c
+++ b/machine/bootstrap.c
@@ -16,15 +16,9 @@ void bootstrap() {
   char* args[] = {
     "./" INIT_FILE_PATH,
     "-c",
-    "selfie.c",
+    "hello-world.c",
     "-m",
-    "2",
-    "-l",
-    "selfie.m",
-    "-y",
     "1",
-    "-c",
-    "hello-world.c",
     (char*) 0,
   };
   int argc = 0;
diff --git a/selfie.c b/selfie.c
index bdbc5b5..d1edd7e 100644
--- a/selfie.c
+++ b/selfie.c
@@ -2704,18 +2704,6 @@ uint64_t round_up(uint64_t n, uint64_t m) {
 }

 void zero_memory(uint64_t* memory, uint64_t size) {
-  uint64_t i;
-
-  size = round_up(size, SIZEOFUINT64) / SIZEOFUINT64;
-
-  i = 0;
-
-  while (i < size) {
-    // erase memory by setting it to 0
-    *(memory + i) = 0;
-
-    i = i + 1;
-  }
 }

 uint64_t* smalloc(uint64_t size) {
```

Listing 3.1: Diff on top of Selfie's repository for evaluation

There is no trivial way of measuring how much instructions were executed by the application. In a hosted environment, the OS cannot deduce how much instructions were executed by comparing the pre- and post-context-switch state. On QEMU, it is potentially possible to capture the instruction count using plugins. However, this is not feasible for the hosted environment counters. Also, the ISA counter `instret`[cf. 19, p. 11] also is not well suited as effort would have to be made by Linux to correctly reduce the amount of instructions occupied by context switching or general interrupt handling and our OS to exclude OpenSBI and `zero_mem`. Instead, GDB is able to step on instruction-level using `stepi`[32] and features scripting capabilities, which allows to define a user command to step instruction-wise and count how many instructions were encountered from command invocation (e.g. after breaking on a symbol) up to another symbol.

```
define do_count
set $count = (unsigned long long)0
while ($pc != $arg0)
    if ($pc == zero_mem)
        fini
    else
        if ($pc == sbi_ecall_sbi_putchar)
            fini
        else
            stepi
            set $count=$count+1
        end
    end
end
print $count
end
```

Listing 3.2: GDB script for recording the instruction count [33]

```
target remote :9000
add-symbol-file build/qemu/library/selfie.elf
monitor system_reset

set pagination off
break _start
c
do_count exit_selfie
```

Listing 3.3: Additional commands for GDB for the library OS

[32]`https://sourceware.org/gdb/onlinedocs/gdb/Continuing-and-Stepping.html`
[33]Based on an StackOverflow answer of Mark Plotnick, licensed under the CC BY-SA 3.0 (`https://stackoverflow.com/a/21639842`)

```
set pagination off
break _start
r -c examples/hello-world.c -m 1
do_count exit_selfie
```

Listing 3.4: Additional commands for GDB for the hosted binaries

| Target | Instruction Count | |
|---|---|---|
| | Dynamic linkage | Static linkage |
| Host | 592,518 | 553,487 |
| RV64 Linux | 642,347 | 566,517 |
| RV64 LibOS | *(N/A)* | 561,370 |

Table 3.1: Recorded instruction counts for all three targets

It is not surprising that the binaries executed on the Intel host need less instructions than their RISC-V equivalents, with x86_64 being a CISC instruction set. The statically linked binaries need less instructions than their dynamically linked counterparts. Considering the overhead by the dynamic linker to resolve shared object references, this seems plausible. In particular, a lot of cycles were spent in code related to `dl-addr.c` .

The library OS needed 5,147 fewer instructions than the static RV64 Linux binary. Notably, this number also includes the OS code for the static file system, I/O syscalls (excluding console printing), memory allocation (excluding zeroing memory) and kernel initialization.

While the library OS required less instructions to execute, it is also worth noting that, while both Linux/glibc and the library OS implement features necessary for Selfie, the former is more general while the latter is only limited to Selfie's syscall interface.

# 4 Open points / Caveats

While the current implementation of does work on emulated machines and actual RISC-V hardware, there is still room to improve the usability and to make it future-proof. Some of these open points affect the library OS only, but some more fundamental points involve both OS variants and must be realized in a generalized way. All of these points must be researched further and evaluated whether they would introduce too much complexity or not.

## 4.1 Specific to the Library OS

### 4.1.1 Trap Support

Supporting RISC-V traps may help to triage bugs that occurred during execution. Access faults and illegal instructions in particular may be important indicators. As of right now, the library OS sets the trap handler to `start_hang` (section 2.2.1.3) which silently hangs the machine. However, the machine would provide information relevant for tracking down errors. The hart sets register `sepc` to the faulting program counter and may provide additional information in register `stval`, like the virtual address that caused an access fault or the instruction that was deemed illegal. The type of is denoted by register `scause` [cf. 19, p. 60-63]

### 4.1.2 Stack smash detection

GCC features compiler flags `-fstack-protector` and `-fstack-protector-strong` which add canary word(s) to the start of the stack frame which are populated upon function entry and checked before the function returns [cf. 17, p. 212]. If the canary has been overridden and does not match, the stack has been smashed and a non-returning call to `void __stack_chk_fail()` is made [cf. 16, p. 554].

### 4.1.3 Memory Protection

While the library OS does not use or need virtual addressing, its memory protection facilities may be helpful. Using identity mapping, the OS may map its `.text` section read-only and executable, while all data sections may be mapped with read-write bits set. By not mapping page frames that were not yet allocated by `malloc`, rogue heap accesses may be detected. Also, the code would be protected from being overridden.

## 4.2 Regarding both OS variants

### 4.2.1 Serial Driver on OS Level

Starting with RISC-V SBI specification 0.2.0, the console get and put functions were moved to the *optional* legacy SBI extension namespace [cf. 4]. While they still work as of OpenSBI 0.7 with the three tested boards, it cannot be expected that these functions will work with future versions or new platforms.

For long-term serial support, the OS must provide its own driver to the serial interface instead of relying on OpenSBI to handle console output.

### 4.2.2 Device Tree Parsing

As mentioned in section 2.2.1.4, the device tree specification defines node `/memory` which describes where physical memory is located in the physical address space and how large the region is. This information may be used by the bump pointer allocator instead of relying that memory is a single contiguous block of memory. By detecting that the OS is out of memory, it may also provide diagnostic messages instead of relying on the machine to produce PMP access faults[cf. 19, p. 50] on out-of-memory.

In accordance to OS serial support, the OS may discover compatible serial interfaces and its parameters using the device tree [cf. 7, p. 39f].

To take advantage of the device tree, it is necessary to implement a flattened device tree parser, as described in [7, p. 44-49].

### 4.2.3 Hypervisor support

On bootlevel zero, hypster, Selfie's hypervisor, basically behaves like mipster, because this is the level that was usually executed on a different ISA (typically arm64 or x86_64). For boot level one and up, `starc` provides a masked definition of `hypster_switch` that is responsible for switching virtual machines by performing a `switch` system call. In these boot levels, we are in the domain of a running mipster emulation where we can be sure that the system call is supported and we are running on a RISC-U "machine".

Because both OS variants run on actual RISC-V machines, hypster on boot level 0 should work. However, we must trap into the environment call so that (virtualized) system calls may be handled by hypster and the `switch` system call can actually be implemented. Using *horizontal* traps [cf. 19, p. 4], we stay in S-mode and must handle the trap in our kernel. For the library OS, we probably have to find a way to dump the machine state into a Selfie context (possibly using `sscratch`[cf. 19, p. 60] register) and pass control to `handle_exception`. While this allows us to re-use selfie's facilities, it would tightly couple the library OS with Selfie and require changes to the latter.

Another way to realize hypster on boot level zero could use *vertical*[cf. 19, p. 4] traps to switch into the more privileged M-mode and perform the trap handling for the hypster clients and `switch` system call for hypster itself in OpenSBI. This would require maintaining a patch set for OpenSBI that is applied to the downloaded source, but would allow us to keep changes to Selfie minimal, if any.

Of course it would be interesting to use designated ISA extension capabilities for this task. There seems to be a hypervisor extension planned[cf. 19, p. 17] and worked on[34], but as of writing this thesis, it has not been ratified yet.

# 5 Conclusion

We presented a library operating system for 64-bits RISC-V implementing Selfie's system call interface. The OS and the application run in S-mode and share the same address space (common text, data, heap and stack sections). Runtime SBI services are provided by OpenSBI. We also demonstrated a simple, static, read-only in-memory file system and runtime initialization necessary to bootstrap the C environment and the kernel before passing execution to the linked Selfie. The system call layer has been generalized to enable the integration into other projects, like the preemptive kernel.

While we tested the library operating system on QEMU and two hardware platforms only, we strongly believe it is compatible with any board containing a RV64IM chip given a runtime with SBI base and (for now) legacy extensions is available. Now that RISC-V is slowly gaining traction and more, even end user-focused boards get released, we hope that more people will get involved with the ISA and will find our small educational operating system helpful with their endeavors on bare-metal.

[34]`https://github.com/riscv/riscv-isa-manual/blob/master/src/hypervisor.tex`

# List of Figures

# List of Tables

# List of Listings

## References

[1] Shiva Chen and Hsiangkai Wang. "Compiler Support For Linker Relaxation in RISC-V". In: *RISC-V Workshop Taiwan Proceedings.* Andes Technology. Mar. 13, 2019. URL: https://riscv.org/proceedings/2019/03/risc-v-workshop-taiwan-proceedings/.

[2] Palmer Dabbelt. *All Aboard, Part 3: Linker Relaxation in the RISC-V Toolchain.* SiFive, Inc. Aug. 28, 2017. URL: https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain (visited on 01/17/2021).

[3] Palmer Dabbelt. *All Aboard, Part 4: The RISC-V Code Models.* SiFive, Inc. Sept. 11, 2017. URL: https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models (visited on 01/17/2021).

[4] Palmer Dabbelt and Atish Patra. *RISC-V Supervisor Binary Interface Specification.* Oct. 5, 2019. URL: https://github.com/riscv/riscv-sbi-doc/blob/v0.2.0/riscv-sbi.adoc (visited on 01/24/2021).

[5] Palmer Dabbelt et al., eds. *RISC-V ELF psABI specification.* RISC-V Foundation. Nov. 7, 2020. URL: https://github.com/riscv/riscv-elf-psabi-doc/blob/f02dd7906a376f972be7450fb3a2cac1d24e3345/riscv-elf.md.

[6] Dawson R. Engler. "The exokernel operating system architecture". PhD thesis. Massachusetts Institute of Technology, Dec. 11, 1998. DOI: 1721.1/16713.

[7] Rob Herring, Grant Likely, and Sean Hudson, eds. *Devicetree Specification - Release v0.3.* Linaro, Ltd. Feb. 13, 2020. URL: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.3.

[8] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[9] *ISO/IEC 9899:201x - Committee Draft N1570.* Apr. 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[10] Christoph Kirsch et al. *Selfie Repository.* Jan. 22, 2021. URL: https://github.com/cksystemsteaching/selfie/tree/de92c8e (visited on 01/23/2021).

[11] Christoph M. Kirsch. "Selfie and the Basics". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Onward! 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 198–213. ISBN: 9781450355308. DOI: 10.1145/3133850.3133857. URL: https://doi.org/10.1145/3133850.3133857.

[12] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: https://doi.org/10.1145/2451116.2451167.

[13] Anup Patel. “OpenSBI Deep Dive”. In: *RISC-V Workshop Zurich Proceedings*. Western Digital Research. June 11, 2019. URL: `https://riscv.org/proceedings/2019/06/risc-v-workshop-zurich-proceedings/` (visited on 01/10/2021).

[14] Anup Patel et al. *OpenSBI Repository*. Western Digital Corporation. Apr. 20, 2020. URL: `https://github.com/riscv/opensbi/tree/v0.7` (visited on 01/23/2021).

[15] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 10th. John Wiley & Sons, Inc, Apr. 2018. ISBN: 978-1-119-75313-1.

[16] Richard M. Stallman and GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, Inc. May 7, 2020. URL: `https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc.pdf` (visited on 01/20/2021).

[17] Richard M. Stallman and GCC Developer Community. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, Inc. May 7, 2020. URL: `https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gccint.pdf` (visited on 01/20/2021).

[18] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2015. ISBN: 978-0-13-359162-0.

[19] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation. June 8, 2019.

[20] Andrew S. Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. RISC-V Foundation. Dec. 13, 2019.