Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Aksel Hauge Slettemark

# A Plan 9 port to RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
June 2021

NTNU
Norwegian University of
Science and Technology

Aksel Hauge Slettemark

# A Plan 9 port to RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

# Abstract

The Plan 9 operating system has been ported to many instruction set architectures (ISAs) since its introduction in the 1980s. The RISC-V family of ISAs is an emerging open standard suitable for a wide range of computing systems. To test the claims of Plan 9s portability, a port of Plan 9 to supervisor-mode on 32-bit RISC-V is implemented. The port has no device drivers, but implements most of the necessary RISC-V specific functionality. The claims of Plan 9s portability are judged as being valid, as the port did not necessitate any changes to the portable parts of the Plan 9 source code. Furthermore, RISC-V is found to be suitable as a target for a Plan 9 port. RISC-Vs privilege model provides convenient mechanisms for privilege level separation and abstraction of higher privilege levels.

# Sammendrag

Operativsystemet Plan 9 har blitt tilpasset til å kjøre på mange instruksjonssettarkitekturer (ISAer) siden det ble introdusert på 1980-tallet. ISA-familien RISC-V er en stadig mer populær åpen standard egnet for et bredt spektrum av datasystemer. For å teste påstander om Plan 9s tilpasningsevne til nye instruksjonssettarkitekturer implementeres en tilpasning av Plan 9 for *supervisor mode* på 32-bit RISC-V. Implementasjonen har ingen utstyrsdrivere, men implementerer mesteparten av den nødvendige RISC-V-spesifikke funksjonaliteten. Påstandene om Plan 9s tilpasningsevne viser seg å være gyldige, ettersom tilpasningen ikke nødvendiggjorde noen endringer i den arkitekturuavhengige delen av Plan 9s kildekode. RISC-V viser seg å være en egnet ISA for Plan 9. RISC-Vs privilegiemodell gir praktiske mekanismer for separasjon av privilegiumsnivå og abstraksjon av de høyere privilegiumsnivåene.

# Preface

This project is the continuation of a specialization project. Because the specialization project report [1] is not generally available, and because this project is a direct continuation, some chapters are re-stated with varying degrees of modifications. This is considered standard practice when basing the master's thesis on the specialization project at NTNU. The adapted chapters are listed below.

- **Chapter 1 – Introduction** The first two paragraphs are adapted with modifications.
- **Section 2.2.2 – Compiling on and for Plan 9** The first two paragraphs are adapted with modifications.
- **Sections 4.1 to 4.4** The underlying work described in these sections was first performed during the specialization project, but has since seen modifications.
- **Section 4.2 – Kernel source setup** This section is adapted with modifications.
- **Section 4.4 – Calling SBI functions** This section is adapted with modifications.

# Acknowledgements

I would like to thank my supervisor Michael Engel for his invaluable support, advice, and genuine interest in the outcome of this project.

# Contents

# Figures

# Tables

# Code Listings

# Acronyms

**ABI** application binary interface. ix, 3, 4, 10, 14, 38

**AEE** application execution environment. 3, 4

**CSR** control and status register. viii, 3–6, 15, 18, 21–25, 29

**EID** SBI extension ID. 4

**FID** SBI function ID. 4

**hart** RISC-V hardware thread. 3, 12, 21, 22

**ISA** instruction set architecture. ii, 1, 3, 6, 8, 10, 30, 31, 34

**M-mode** machine-mode. 3, 12

**MMIO** Memory-mapped I/O. 13

**MMU** memory management unit. 7, 8, 18, 20, 21, 25

**PC** program counter. 4, 8, 11, 14, 22, 23, 26–29

**PTE** page table entry. viii, ix, 5, 18, 19, 31

**RISC** reduced instruction set computer. 1

**RV32** RISC-V 32-bit. viii, 2–4, 6, 9, 22

**RV32GC** RISC-V 32-bit with extensions G and C. 9, 12

**RV64** RISC-V 64-bit. 2, 9

**RV64GC** RISC-V 64-bit with extensions G and C. 9

**S-mode** supervisor-mode. ii, 2–5, 12, 15, 22, 23, 28, 33

**SB** static base. 10, 11, 14, 22

**SBI** supervisor binary interface. xi, 3, 4, 12, 15, 33

**SEE** supervisor execution environment. 3, 4, 12

**SP** stack pointer. 10, 14, 22, 27–29

**TLB** translation lookaside buffer. 21

**U-mode** user-mode. 3–5, 12, 15, 22, 26, 28, 29

# Chapter 1

# Introduction

Plan 9 from Bell Labs [2] is an operating system developed by the Computing Science Research Center at Bell Labs. Started in the late 1980s, Plan 9 replaced Unix (also of Bell Labs) as Bell Labs' primary research operating system. Unix is important in the history of portability. Being the first operating system implemented in the C programming language (also of Bell Labs), the first Unix port [3] was a major milestone in operating system development.

Plan 9 is a distributed operating system, using the `9P` protocol [2] to access files locally and across machines. Files represent much more than just storage, and `9P` is at the core of Plan 9s distributed nature. As `9P` is agnostic to the instruction set architecture (ISA) of the other end of the communication, and because Plan 9 comes with a suite of cross compilers for various architectures, a distributed system can be heterogeneous with respect to the instruction set. An operating system kernel naturally contains a lot of architecture-dependent code. Plan 9 separates the architecture-independent (portable) and architecture-dependent code. The portable code requires a number of functions whose implementation is architecture-dependent to be implemented.

RISC-V [4, 5] is an emerging family of ISAs originating at the University of California, Berkeley. As its name suggests, it is based on reduced instruction set computer (RISC) principles, and its specification is published under a Creative Commons license. The use of the ISA is unrestricted, and companies implementing their own designs, hardware or otherwise, are not subject to licensing fees. It is a modular standard intended to be suitable for teaching, embedded systems, personal computers, data centers, virtualization, and everything between.

This report aims to validate the claims the authors of Plan 9 make about its portability, exploring the suitability of RISC-V as a target for Plan 9 along the way. At the time of writing, it has been 30 years since Presotto *et al.* [6] presented Plan 9 as a "general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks". Since then, it has been ported to several new architectures, such as PowerPC. RISC-V is considerably newer than PowerPC and is, much like Plan 9, an exercise in simplicity. If a RISC-V port of Plan 9 is possible without modifying the portable parts of the Plan 9 kernel, this is an affirmation of the claims of the Plan 9 authors, as well as a demonstration of the suitability of RISC-V for such purposes.

To test the portability of Plan 9 and the suitability of RISC-V as a target for a Plan 9 port, a port of Plan 9 running in supervisor-mode (S-mode) on a single-core 32-bit RISC-V implementation is implemented. This is thought to be the simplest case, but is sufficient to demonstrate the capabilities of portable Plan 9 code and RISC-V. Even though RV32 is not a strict subset of RV64 [4], we see no reason RV64 should be less capable of running Plan 9.

# Chapter 2

# Technologies

## 2.1 RISC-V

This chapter gives an introduction to the RISC-V instruction set architecture (ISA) and the aspects of it which are most relevant to the Plan 9 port. 32-bit RISC-V is used for the implementation, so this chapter is written with that in mind, although most of the information is equally valid for 64-bit RISC-V. In particular, the width of control and status registers (CSRs) and the available virtual memory schemes are different. The Sv32 virtual memory scheme described in Section 2.1.4 is only available for RV32.

### 2.1.1 Privilege modes

The RISC-V instruction set manual [5] defines several privilege levels called modes a RISC-V hardware thread (hart) can operate in. The most privileged mode is machine-mode (M-mode), and it is the only mandatory mode in a hardware RISC-V implementation. To isolate user processes from each other and prevent privileged operations, an implementation can add the user-mode (U-mode), which is the least privileged mode. Supervisor-mode (S-mode) is an optional mode with a privilege level between M-mode and U-mode. On an implementation with all three modes a Unix-like operating system would typically run in S-mode [5], supported by firmware running in M-mode providing a supervisor execution environment (SEE) [7]. The interface between S-mode and M-mode is called supervisor binary interface (SBI), and is described in Section 2.1.2. Similarly, the OS running in S-mode provides an application execution environment (AEE) to the user code running in U-mode. The application binary interface (ABI), which is OS-specific, defines a set of system calls to interact with the AEE/S-mode operating system.

### 2.1.2 Supervisor binary interface

The purpose of the supervisor binary interface (SBI) is to make S-mode software portable across different RISC-V implementations such as different hardware platforms and hypervisors, by abstracting away the platform-specific details [7]. SBI, much like RISC-V itself, is a modular standard. The modularity comes in the form of SBI extension IDs (EIDs). An EID, optionally in combination with a SBI function ID (FID), identifies a specific SBI function that can be invoked. Calling into SBI is quite similar to calling a regular function, and generally follows the same calling convention as the standard ABI [8] in terms of register usage. Instead of jumping to a different program code location, the `ECALL` instruction is used, trapping and transferring control to the supporting supervisor execution environment (SEE). The `ECALL` instruction is used to perform environment calls from all privilege modes to a higher mode [4, 5]. In the case of S-mode software and SBI that environment is the SEE, but the instruction is also used for environment calls from U-mode to the application execution environment (AEE), enabling system calls.

### 2.1.3 Control and Status Registers

The RISC-V specification [5] defines a number of control and status registers (CSRs). These registers are identified by a 12-bit address separate from the normal address space and are accessed using special instructions. Each CSR is associated with a specific privilege level, but software running at a higher privilege level or the hardware can access it regardless. The associated privilege level is prefixed to the CSR name, like `mstatus`, `sstatus`, and `ustatus`. Some CSRs are read-only and used to read the current state, for instance the `mcycle` CSR, which is the machine cycle counter register. Other CSRs can be written to and are used to handle things such as enabling/disabling interrupts, setting the interrupt handler address, etc. The RISC-V implementation provides details about exceptions and interrupts to the trap handler by setting certain CSRs such as `sepc`, `scause`, and `stval` for the S-mode trap handler. In this case, `sepc` is used both for communicating the program counter at the time of the trap to the trap handler and in reverse by letting the trap handler use it to specify the address at which execution should be resumed.

### 2.1.4 The Sv32 virtual memory scheme

The RISC-V standard defines several virtual memory schemes [5]. Of them, only the Sv32 scheme is supported by RV32. Sv32 is based on a two-level page table. Level 1 is the top level, and level 0 is the last. Virtual addresses are divided into a 10-bit VPN[1] that is used as an index in the top-level table, a 10-bit VPN[0] that is used as an index in the last level table, and a 12-bit page offset that is the same in the virtual address and the physical address. This can be seen in Figure 2.1.

| 31 22 | 21 12 | 11 0 |
|---|---|---|
| VPN[1] | VPN[0] | page offset |

**Figure 2.1:** Sv32 virtual address. Adapted from Waterman and Asanović [5].

Normal pages are $2^{12}$B $= 4$ KiB large, but *megapages* can be mapped at the top level and are 4 MiB large [5]. Each page table entry (PTE) is 4 bytes, and each page table contains 1024 entries, making the page table page-sized. The table must also be page-aligned, meaning the lower 12 bits of the memory address of the start of the table must be all 0. The structure of a PTE can be seen in Figure 2.2. Bits 9 and 8 (RSW) are reserved for use by S-mode software. The full name and function of each flag is displayed in Table 2.1. PPN[i] denotes a part of a physical address, and the relation between a physical address and its PPNs can be seen in Figure 2.3. Sv32 supports 34-bit physical addresses, as can be seen from the bit width of Figure 2.3. A valid PTE in the top-level table, using its PPNs, either points to a mapped megapage, or the start of the last level page table. A valid PTE in the last level page table points to the start of a mapped 4 KiB page.

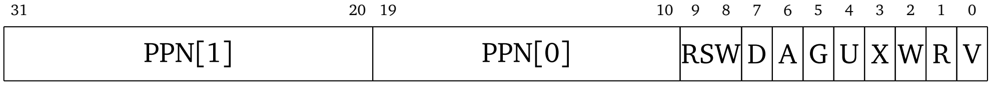

**Figure 2.2:** Sv32 page table entry. Adapted from Waterman and Asanović [5].

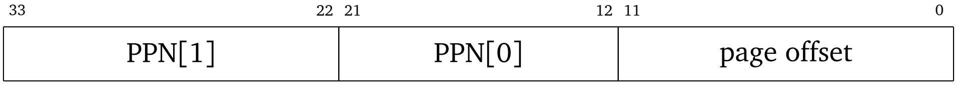

**Figure 2.3:** Sv32 physical address. Adapted from Waterman and Asanović [5].

**Table 2.1:** Full name and function of Sv32 PTE flags.

| Flag | Name | Function |
|---|---|---|
| D | Dirty | The page has been written to since flag was last cleared |
| A | Accessed | The page has been read since flag was last cleared |
| G | Global | The page mapping is valid in all address spaces |
| U | User | The page is accessible to U-mode |
| X | Execute | The page is executable |
| W | Write | The page is writable |
| R | Read | The page is readable |
| V | Valid | The PTE is valid |

To enable virtual memory the `satp` CSR is written with a 1 in the MODE field and the PPNs of the top-level page table in the lower bits [5]. The bit layout can be seen in Figure 2.4. Bit 31 is the MODE field. ASID is an address space identifier, but its mechanism will not be used in this thesis.

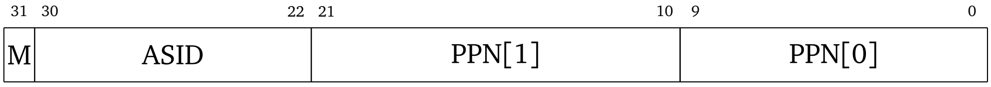

**Figure 2.4:** The `satp` control and status register (CSR) on RV32. Adapted from Waterman and Asanović [5].

## 2.2 Plan 9

A brief introduction to Plan 9 was given in Chapter 1. This section will focus on the technological aspects of Plan 9 that are relevant for a port to RISC-V.

### 2.2.1 Plan 9 C

Plan 9 is written in its own version of the C language. The details are thoroughly explained by Thompson [9]. The two most consequential for our purposes are that the preprocessor directive `#if` is not supported, and the concept of unnamed substructures. Unnamed substructures allow struct members without names, meaning only the type is provided. An example that is ubiquitous throughout the Plan 9 kernel is including unnamed `Lock` structs in various other structs. The outer struct may then be passed to functions expecting a `Lock` argument, and the compiler will automatically pass the inner unnamed `Lock`. Named members of the inner struct can be accessed directly as if they were a member of the outer struct.

### 2.2.2 Compiling on and for Plan 9

Each instruction set architecture supported by Plan 9 has its own toolchain with an assembler, a C compiler, and a linker. The linker is often referred to as a loader in various Plan 9 literature, but this is a misnomer when considering the modern use of the term. All toolchains in Plan 9 are cross-architecture and are identified by a single number or letter. For instance, the toolchain for `i386` is identified by the number 8, and so the assembler program is called `8a`, the compiler is called `8c`, and the linker is called `8l`. As all the toolchains are cross-architecture, the 8 toolchain runs on all architectures supported by Plan 9 itself, not just `i386`.

Even though three standalone programs are created for every architecture, a lot of architecture-independent code is shared between the different toolchains. For the compilers, it amounts to a little over 12000 lines of shared source files, as seen in Table 2.2. For the ARM compiler, `5c`, the number of lines in architecture-specific source files is just under 7500. The assemblers and linkers rely less on sharing code. Instead, code is duplicated and modified when a new toolchain is written. All the assemblers and the shared C compiler code use a version of Yacc [10] to parse their respective input languages.

All three components of the toolchains are unique in some way. The compilers accept a custom version of the C programming language, as described in Section 2.2.1. The assemblers use a custom syntax that generally looks quite differ-

**Table 2.2:** Line numbers of shared source files for C compilers in sys/src/cmd/cc/*.(h|c|y). Copied from Slettemark [1].

| File | Lines |
|---|---|
| acid.c | 303 |
| bits.c | 89 |
| com64.c | 619 |
| com.c | 1462 |
| compat.c | 47 |
| dcl.c | 1636 |
| dpchk.c | 494 |
| funct.c | 400 |
| lex.c | 1561 |
| mac.c | 3 |
| omachcap.c | 8 |
| pgen.c | 591 |
| pickle.c | 268 |
| pswt.c | 199 |
| scon.c | 606 |
| sub.c | 2032 |
| cc.h | 782 |
| cc.y | 1183 |
| total | 12283 |

ent from what is used with mainstream toolchains such as the GNU Compiler Collection. Instead, the assembly syntaxes for the different architectures on Plan 9 are quite similar to each other. The Plan 9 assembler manual [11] describes the syntax and important architecture-dependent differences between the assemblers. Both the compilers and assemblers output a binary encoded representation of assembly-like instructions and leave the selection of concrete instructions up to the linker. The toolchain components are typically not invoked manually. Instead, they are driven by `mk` and `mkfiles`, which provide similar functionality to Make and Makefiles.

### 2.2.3 Portability

The Plan 9 kernel code is located in `sys/src/9/`. Each architecture-specific kernel has its own directory, for instance `pc` for `i386`, `ppc` for PowerPC and so on. The portable code shared between the kernels is located in the `port` directory. Port contains portable code for process management, scheduling, memory management (except MMU calls), communication protocols, and mostly complete implementations of all system calls, and more [6]. Some system calls rely on architecture-specific functions, particularly those that deal with spawning/forking processes and manipulating program stacks.

### Toolchain

Plan 9, with its custom C dialect and custom executable header format, requires a custom toolchain for every new instruction set architecture. Porting the toolchain is a substantial task. Luckily, a Plan 9 toolchain for RISC-V is already released by Miller [12]. The Miller toolchain for RISC-V will be presented in Section 3.1.

### Virtual memory

Virtual memory is a requirement for running Plan 9. The portable code in the kernel is naturally not able to manipulate architecture-specific page tables. The `portfns.h` header defines several MMU-related functions that must be implemented. These are `putmmu`, `flushmmu`, `mmurelease`, and `mmuswitch`.

### Trap handling

Trap-setup and -handling is not portable. Saving and restoring registers requires architecture-specific assembly code. Setting up a stack and static base, jumping to C code, and returning from trap handling are all highly non-portable procedures. Different architectures also have very different mechanisms for communicating the trap cause. The non-portable trap handlers rely heavily on portable code to do the heavy lifting regarding system calls, scheduling, keeping track of process memory, and more.

### Various low-level functions

Some low-level functions like atomic test-and-set (`tas`), functions that read or modify the PC such as `setlabel` and `gotolabel`, and functions for reading machine state such as clock cycle counters are not portable and must in most cases be written in assembly. Setting up clock interrupts, and the trap handling itself is not portable, but the actual scheduling algorithm is.

# Chapter 3

# Previous work

This chapter describes the previous work done specifically towards a RISC-V port of Plan 9. RISC-V ports of other operating systems are not described. All the work covered is done by Richard Miller, who is also considered the first person to port Unix [3], a major milestone in the history of portable operating systems.

## 3.1 Toolchain by Richard Miller

Miller [12] presents a RISC-V toolchain for Plan 9 that supports RISC-V 32-bit with extensions G and C (RV32GC) and RISC-V 64-bit with extensions G and C (RV64GC). The release[1] assigns the letter `i` to the RV32GC version and the letter `j` to the RV64GC version, as is required by the naming scheme explained in Section 2.2.2. The `G` extension is shorthand for extensions `IMAFD`, `Z_icsr`, and `Z_ifencei` [4, Chapter 27]. These extensions are sufficient for our purposes. Miller [12] refers to the RV32 architecture as `riscv` and to RV64 as `riscv64`.
On March third 2021, Miller uploaded an updated version of the toolchain containing some architecture-specific library code for Plan 9, some updated documentation, and some minor changes to the C compiler and linker. The March 3rd version of the `i` toolchain for RV32GC consisting of the assembler `ia`, the C compiler `ic`, and the linker `il` will be discussed in this chapter and used for the implementation. Like the other Plan 9 toolchains, the selection of specific RISC-V instructions is performed by the linker.

### 3.1.1 Noteworthy non-standard behaviour

Being a Plan 9 toolchain, it is both written in and made for compiling the Plan 9 version of C. By default, the object header format is the Plan 9 format. Other aspects that may be considered non-standard or surprising are laid out in this section.

[1] `http://9p.io/sources/contrib/miller/riscv.tar`

### Assembler and registers

The assembler accepts a syntax that is very unlike what is seen in mainstream toolchains such as GCC but rather more aligned with the other Plan 9 assemblers. It uses neither the register names x0-x31 nor the register names used in the standard ABI [8]. Table 3.1 summarizes the difference in naming between the register numbers, the standard ABI names, and the Plan 9 toolchain names.

**Table 3.1:** Register mnemonics in the standard ABI [8] compared to Plan 9 toolchain by Miller [12].

| Register | Standard ABI Name | Plan 9 Name | Plan 9 Function |
|---|---|---|---|
| x0 | zero | R0 | Zero constant |
| x1 | ra | R1 | Link Register |
| x2 | sp | R2 | Stack pointer |
| x3 | gp | R3 | Static base (SB) |
| x4 | tp | R4 | Loader temporary |
| x5-7 | t0-2 | R5-7 | |
| x8 | s0/fp | R8 | First function argument/return value |
| x9 | s1 | R9 | |
| x10-17 | a0-7 | R10-17 | |
| x18-27 | s2-11 | R18-27 | |
| x28-31 | t3-6 | R28-31 | |

The assembler does not directly expose the instructions in the RISC-V ISA. For instance, when encountering MOVW, the linker will output a sw (store word) instruction if the source is a register and the target is a memory address, lw (load word) in the opposite case, or yet another instruction if both the source and destination are registers. If the source is a constant value, it will generate different instructions depending on whether the constant fits in the 12 bits available for immediate values. When using MOVW to store a word to main memory using a base register and an offset that requires too many bits to represent, the linker will generate instructions to place the target address in R4. The fact that R4 is modified is not visible in the assembly source file.

### C compiler

The C compiler by Miller [12] does not follow the standard RISC-V calling convention as laid out by the RISC-V ISA manual [4]. Instead, it uses a fairly simple scheme where the first function argument is placed in R8, and any additional arguments are on the stack. R8 is also used to pass the return value.

### Static Base

The C compiler and assembler both reference global symbols relative to a special value called the static base (SB). In the Miller toolchain, this value is always kept in the R3 register. To load this value into the register the special pseudo-instruction MOV $setSB(SB), R3 is used, which instructs the linker to generate instructions to store or otherwise construct the SB value in R3.

## 3.2 Other contributions by Richard Miller

With his toolchain release, Miller [12] also included several other files and library patches that are useful for compiling for RISC-V on Plan 9. Some parts of libc must be implemented in assembly, and some parts are usually implemented in assembly for performance reasons. The release includes every part of libc necessary to compile the Plan 9 user-space. This also includes a patch to 9syscall that inserts architecture-specific code for performing system calls. In the case of RISC-V, this means placing the arguments and issuing the ECALL instruction. The headers u.h and ureg.h were also included for the riscv and riscv64 architectures. u.h contains typedefs for various primitive types, such as u32int and macro definitions for variadic functions (varargs). ureg.h contains the definition of the Ureg struct, which is typically used for saving registers and execution state such as the program counter on traps.

Miller [12] also includes patched versions of libmach and mkfile.proto, which makes it possible for the host Plan 9 installation to recognize and conveniently compile for the new architectures.

# Chapter 4

# Implementation

This project targets the QEMU [13] emulator. Specifically, version 5.0.0 of the `qemu-system-riscv32` emulator with the `virt` machine preset emulating a single RISC-V hardware thread (hart). QEMU is configured with 256 MiB of RAM. The Miller [12] toolchain described in Section 3.1 is used. The kernel is based on the January 10th 2015 release of the fourth edition of Plan 9. While the kernel is built in a Plan 9 environment, an operating system that supports QEMU is required to run the compiled kernel. The ported operating system exclusively runs in S-mode, and U-mode for its processes. A completely unmodified, off-the-shelf M-mode firmware provides the supervisor binary interface (SBI) and acts as the supervisor execution environment (SEE).

The full implementation can be found in my GitHub repository[1]. The exact version delivered with this document is tagged with the name `thesis_state`. Appendix A is a step-by-step guide for replicating the development environment used, and includes instructions to produce a patch containing the Miller [12] toolchain and this kernel implementation.

## 4.1 OpenSBI

OpenSBI [14] is an open-source implementation of the RISC-V SBI [7]. It includes support for QEMU and will serve as our M-mode firmware, and provide the supervisor execution environment (SEE) for our kernel which will run exclusively in S-mode. OpenSBI is cross-compiled with GCC for RV32GC with the `ilp32` ABI targeting the `generic` OpenSBI platform. QEMU starts in M-mode, executing the `fw_jump` version of the OpenSBI firmware which jumps to the beginning of the Plan 9 kernel in S-mode.
The kernel is placed at memory address `0x80400000`. Memory between `0x80000000` (ram zero) and `0x80200000` is assumed to be reserved. This range is 2 MiB and is a very conservative reservation. In reality, OpenSBI typically uses around 128

[1] `https://github.com/aslettemark/plan9_riscv`

KiB. Since we are not using a device tree with exact information and to err on the side of caution, it is assumed that 2 MiB is reserved, just like the Linux kernel assumes. Figure 4.1 summarizes the layout of the emulated physical memory. The area between `0x80200000` and `0x80400000` is used as a manually managed memory area for when compile-time known addresses are convenient, such as when writing assembly code.

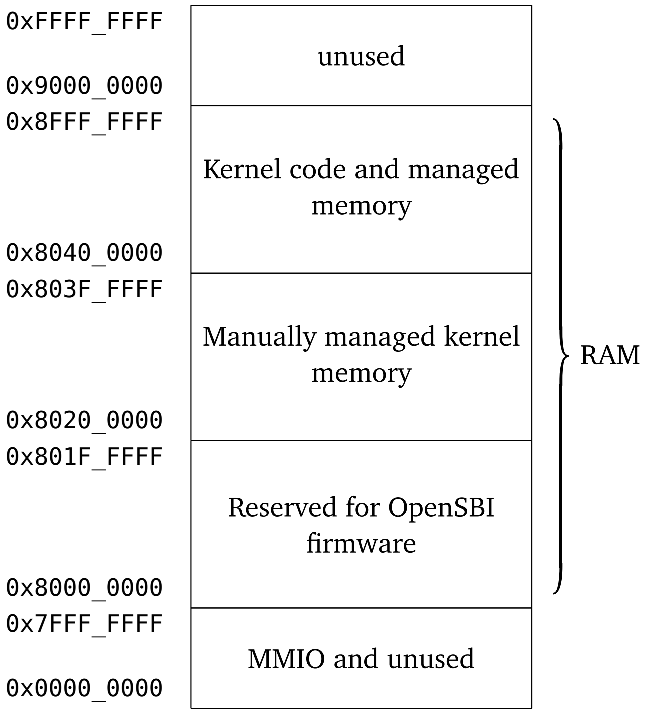

**Figure 4.1:** The emulated "physical" memory layout.

## 4.2 Kernel source setup

To get started, a new folder with a kernel configuration file and `mkfile` need to be created in `sys/src/9/`. This implementation will be called `qrv32`, short for QEMU RISC-V 32-bit. Some headers, `fns.h`, `mem.h`, and `dat.h` need to be filled with various macros and definitions related to memory addresses. All hard-coded memory addresses are defined in `mem.h`. Some structs need to be defined in `dat.h`, like the `Mach`, `Lock`, `Conf`, and `Proc` structs. Each of these structs has some mandatory fields that are used by other parts of the kernel, such as performance counters in `Mach`, and some fields that may be needed in architecture-specific code. All the non-portable functions the portable code expects must be implemented or written as stubs to satisfy the linker. The kernel is then compiled and linked as an `ELF` file with load address `0x80400000`, which is the address the OpenSBI firmware is programmed to jump to.

## 4.3 Entering C

The kernel entry point is the `_start` symbol defined in the assembly source file `l.s`. To enter C code, we need to set the stack pointer (SP) register to support the stack. We also need to set the special static base (SB) register as described in Section 3.1.1. A 8 KiB memory region starting at address `0x80200000` is reserved for the stack. Because the stack grows downwards, and the stack pointer points to the next available memory location, the value is modified to point to the top of the stack minus 4 B. The assembly code for jumping to the C function `main(void)` is shown in Code listing 4.1.

**Code listing 4.1:** Entering C.

```
TEXT _start(SB), $-4
	MOVW $setSB(SB), R3

	MOVW $(KSTACK_LOW_END), R2
	ADD $(KSTKSIZE), R2, R2
	ADD $-4, R2

	JAL R1, main(SB)
	RET
```

## 4.4 Calling SBI functions

To be able to call SBI functions we must define a routine for using the `ECALL` instruction [4]. The interface for `ECALL` is defined in terms of the standard calling convention, and requires arguments in registers `a0`, `a1`, and `a2`, and the SBI extension ID in register `a7` [7]. As can be seen in Table 3.1 registers `a0` to `a2`, and `a7` correspond to `R10` through `R12`, and `R17` in the Plan 9 toolchain by Miller [12]. It also uses `R8` as the register for first argument and return value, contrary to the standard RISC-V ABI [8] which uses different registers. The `sbi_ecall` procedure translates between the two calling conventions, issues the `ECALL` instruction, and moves the return value to `R8`. The code is shown in Code listing 4.2. The number `1` between the function name and `$-4` means the function will not be profiled, and is present for syscall-like functions as recommended by the Plan 9 assembler manual [11]. The value `$-4` is a special value that makes sure the linker does not automatically reserve area on the stack or generate any PC save and restore instructions [11].

**Code listing 4.2:** Using the ECALL instruction to interact with SBI.

```
TEXT sbi_ecall(SB), 1, $-4
	MOVW R8, R10 // __a0
	MOVW 4(FP), R11 // __a1
	MOVW 8(FP), R12 // __a2
	MOVW 12(FP), R17 // __num (a7)
	ECALL
	MOVW R10, R8 // a0 to ret. val
	RET
```

To use the sbi_ecall routine for printing text we need to use SBI extension ID 0x01 (sbi_console_putchar) as defined in the SBI specification [7]. The C code for printing null-terminated C strings to console is shown in Code listing 4.3.

**Code listing 4.3:** Printing to console.

```
#define SBI_CONSOLE_PUTCHAR 0x1
extern int opensbi_ecall(unsigned int a0,
                        unsigned int a1,
                        unsigned int a2,
                        unsigned int sbi_ext_id);

void console_print(char *str) {
    while (*str) {
        opensbi_ecall(*str, 0, 0, SBI_CONSOLE_PUTCHAR);
        str++;
    }
}
```

## 4.5 Plan 9 initialization

When starting Plan 9, various parts of the kernel must be initialized. We begin by clearing the BSS and Mach struct with memset. The global variable m is set to point to the Mach struct, and it is populated with data about the number of processors. The global variable up which is a pointer to the Proc struct representing the currently scheduled process is set to nil. up is widely used throughout both the portable and non-portable parts of the kernel. A series of argument-less portable Plan 9 initialization functions are called to initialize Plan 9s internal data structures related to page handling and allocation, such as xinit, initseg and pageinit.

## 4.6 MMU

To give each process its own address space, we need to use virtual memory. Functions for using the Sv32 virtual memory scheme as described in Section 2.1.4 will be implemented. Sv32 supports 32-bit virtual addresses and 34-bit physical addresses. As the portable parts of the Plan 9 kernel source code represents physical addresses using pointer-sized unsigned integers, there is no way to utilize the two extra bits of address space without significant changes.
In addition to using Sv32, the implementation also sets the SUM bit in the sstatus CSR to be able to access pages mapped for U-mode. This allows the operating system to read memory like system call arguments from user programs, but still prevents S-mode software from executing code from a U-mode-accessible page [5].

The virtual memory layout is summarized in Figure 4.2. The addresses at the region borders are defined mem.h. The QEMU virt machine for RISC-V has memory starting at address `0x80000000`. All used addresses above this base are identity mapped to form the kernel address space. User programs place all their segments in the range `0x00000000` to `0x80000000`. `ESEG` is a temporary stack used by the `sysexec` function in the portable code to set up a new stack when a process uses the `exec` system call.

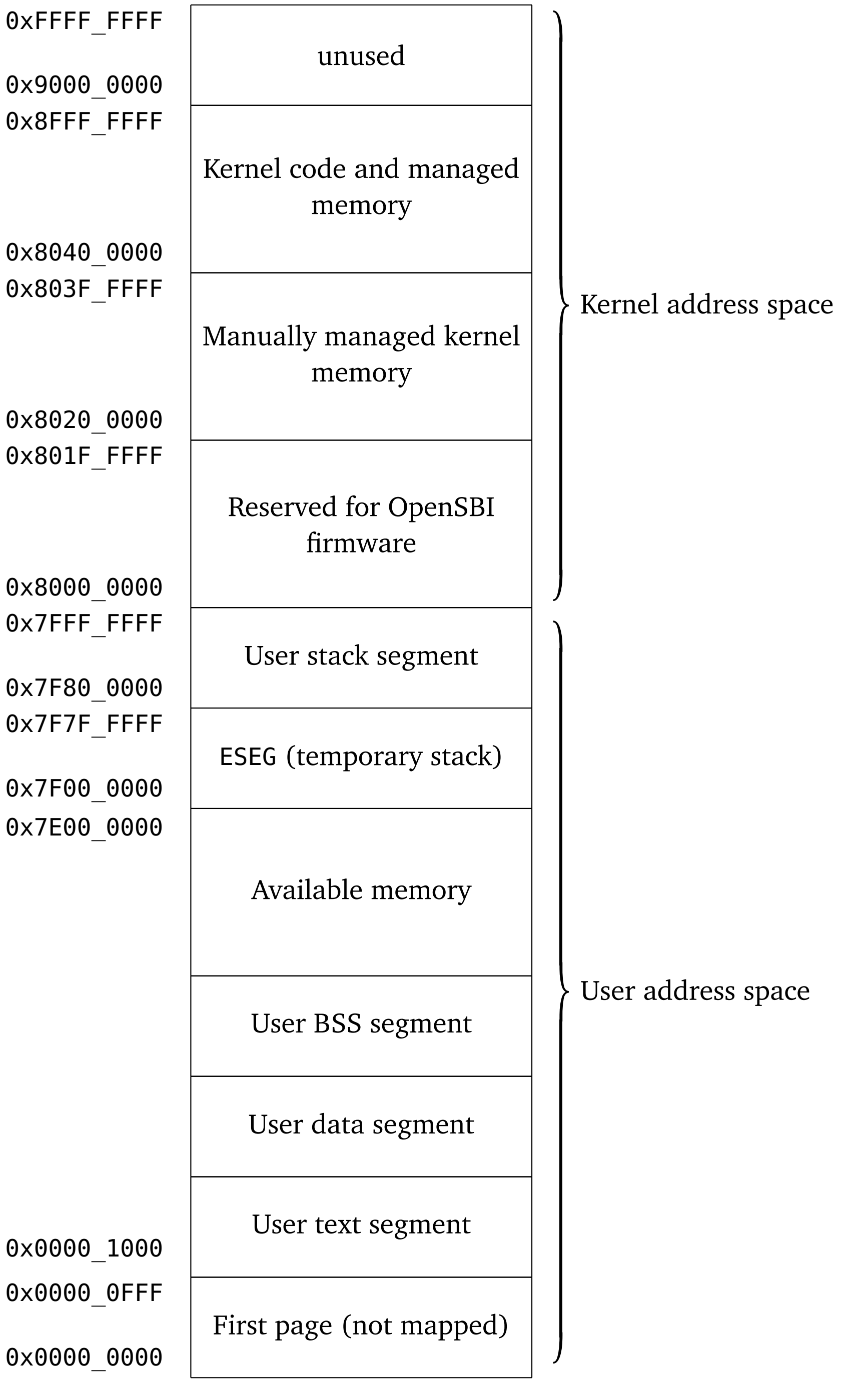

**Figure 4.2:** The virtual memory layout.

### 4.6.1 Initializing virtual memory

Initializing virtual memory is handled by the `mmuinit` function, which is called from `main` when booting the kernel. The function is displayed in Code listing 4.4. First, the top-level page table is cleared with all 0s. `ROOT_PAGE_TABLE` is a memory address in the manually-managed region of kernel memory as described in Section 4.1 and Figure 4.1. An identity mapping of all memory in the kernel segment is performed with read, write, and execute flags set. Notice that the user flag as described in Table 2.1 is not set, so it is safe to leave this mapping in place even when executing user code. The `map_single_page` function will be described in Section 4.6.2. A value for the `satp` CSR is then constructed to enable virtual memory and register the top level page table address as described in Section 2.1.4. `write_satp` and `set_sstatus_sum_bit` are tiny assembler functions that write to their respective CSRs.

**Code listing 4.4:** The `mmuinit` function.

```c
#define BY2PG (4 * 1024)
typedef u32int PTE;
PTE *toplevel_pagetable = UINT2PTR(ROOT_PAGE_TABLE);

void mmuinit() {
        memset(toplevel_pagetable, 0, BY2PG);

        for (u32 i = 0; i < MEMSIZE; i += BY2PG) {
                map_single_page(
                    RAMZERO + i,
                    RAMZERO + i,
                    (PTEREAD | PTEWRITE | PTEEXECUTE),
                    toplevel_pagetable
                );
        }

        uintptr satp = (1 << 31) | (ROOT_PAGE_TABLE >> 12);
        write_satp(satp);
        set_sstatus_sum_bit();
}
```

### 4.6.2 Mapping pages

The portable part of the Plan 9 kernel (port) maps addresses in the MMU by calling the `putmmu` function that is implemented once per architecture. Port keeps track of which pages are mapped in each process, so it already knows the physical address (`pa`) and virtual address (`va`) that should be mapped. Our implementation is listed in Code listing 4.5. The `interpret_fixfault_flags` function called from `putmmu` translates from the portable PTE flags that the portable code sets before calling `putmmu` to the actual flags as defined in the RISC-V specification. Plan 9 uses internal flags such as `PTERONLY` (read-only) that are not a direct match with a flag in every architecture. When applied to Sv32, `READONLY` implies the absence of the `PTEWRITE` flag, so some degree of interpretation of the internal flags is needed.

`interpret_fixfault_flags` also adds the `PTEUMODE` flag to make the mapping available to user-space.

We begin in the `map_single_page` function in Code listing 4.5 by extracting the VPNs and PPNs of the virtual and physical address to be mapped. Line 11 checks if there is an entry in the top-level page table pointing to a level 0 (L0) page table for this virtual address. If there is, we extract the physical address of the L0 table based on the PTE on line 18. Recall that kernel space identity-maps the available memory, meaning we can use this physical address even though we are currently operating with virtual memory enabled. We index the L0 table and check if there is already an entry with the `valid` bit set, as this would be an error. The PTE is then constructed on line 22 using the PPNs and entered into the L0 table on the following line. Going back to the case where there is no L0 table, we begin by allocating a new zero-initialized page to back the L0 table on line 26. We then construct and enter the entry for the L0 table as before. On line 35, a PTE for linking the newly constructed L0 table to the top-level page table is constructed. This implementation does not utilize the *megapage* mechanism.

**Code listing 4.5:** The `putmmu` and `map_single_page` functions.

```
1   #define PTE2PA(pte) (((pte) >> 10) << 12)
2
3   void map_single_page(u32 va, u32 pa, u32 flags, PTE *l1_table) {
4           u32 vpn1 = (va >> 22) & 0x3FF;
5           u32 vpn0 = (va >> 12) & 0x3FF;
6
7           u32 ppn1 = (pa >> 22) & 0x3FF;
8           u32 ppn0 = (pa >> 12) & 0x3FF;
9
10          PTE entry = l1_table[vpn1];
11          if (entry & PTEVALID) {
12                  u32 mask = PTEEXECUTE | PTEWRITE | PTEREAD;
13                  if (entry & mask) {
14                          panic("Top␣level␣remap␣va␣=␣0x%p\n", va);
15                  }
16
17                  // Now we know entry is a pointer to next level
18                  PTE *l0_table = UINT2PTR(PTE2PA(entry));
19                  if (l0_table[vpn0] & PTEVALID) {
20                          panic("L0␣remap␣va␣=␣0x%p\n", va);
21                  }
22                  PTE leaf_entry = (ppn1 << 20) | (ppn0 << 10) | flags | PTEVALID;
23                  l0_table[vpn0] = leaf_entry;
24          } else {
25                  // Create new l0 table and entry
26                  Page *page = newpage(1, 0, 0);
27
28                  // Fill the entry in the l0 table
29                  PTE leaf_entry = (ppn1 << 20) | (ppn0 << 10) | flags | PTEVALID;
30                  PTE *l0_table = UINT2PTR(page->pa);
31                  l0_table[vpn0] = leaf_entry;
32
33                  // Write the new page to root level table
34                  u32 ppn_full = page->pa;
35                  PTE l1_entry = (ppn_full >> 2) | PTEVALID;
36                  l1_table[vpn1] = l1_entry;
```

```
        }
}

void putmmu(uintptr va, uintptr pa, Page* page) {
        u32 flags = pa & 0xFFF;
        pa = page->pa;

        u32 our_actual_flags = interpret_fixfault_flags(flags);

        map_single_page(va, pa, our_actual_flags, toplevel_pagetable);
}
```

### 4.6.3 Flushing the MMU and switching between processes

As the system should support more than one process, the virtual memory mappings need to be flushed on several occasions. Plan 9 requires us to implement the `flushmmu`, `mmurelease`, and `mmuswitch` functions. The kernel address space is always kept mapped, so only the mappings for virtual addresses between `0x00000000` and `0x80000000` are cleared. The code for flushing the user-space addresses is shown in Code listing 4.6. Note that the pages backing the L0 tables are not deallocated but instead wiped. Plan 9 uses portable data structures to keep track of which physical pages belong to each process. Therefore the MMU implementation is free to wipe its entries when flushing. This keeps the implementation very simple and avoids duplicating the data structures that keep track of page ownership and mappings for processes. However, it comes at the cost of performance, as process switching typically will cause many page faults.

**Code listing 4.6:** The `flushmmu` function.

```
void flush_userspace() {
        u32 va = UZERO;

        while (va < KZERO) {
                u32 vpn1 = (va >> 22) & 0x3FF;
                u32 vpn0 = (va >> 12) & 0x3FF;

                PTE entry = toplevel_pagetable[vpn1];
                if (entry & PTEVALID) {
                        PTE *l0_table = UINT2PTR(PTE2PA(entry));
                        memset(l0_table, 0, BY2PG);
                }

                // Skip over all 1024 page entries in one l0 table
                va += BY2PG * 1024;
        }
}

void flushmmu() {
        int s = splhi();
        flush_userspace();
        sfence_vma();
        splx(s);
}
```

The implementation is fairly straightforward, iterating over the entries in the top-level page table by incrementing a virtual address. This particular implementation assumes the kernel space boundary is *megapage*-aligned because it skips over entire level 0 tables at a time. `splhi` and `splx` disables interrupts and restores the previous interrupt state. `sfence_vma` is a low-level assembly function that consists of a supervisor memory-management fence instruction. The mnemonic used in the RISC-V specification [5] is `SFENCE.VMA`. It forces the RISC-V implementation to synchronize with main memory memory-management data structures. This means that a RISC-V implementation with a translation lookaside buffer (TLB) for caching MMU entries must invalidate its entries. The `SFENCE.VMA` instruction can be used in conjunction with the address space identifier (ASID) mechanism to only invalidate a subset of entries. As stated in Section 2.1.4 this project does not use the ASID mechanism, so the `sfence_vma` function is not parameterized with an ASID and instead forces a full invalidation. `SFENCE.VMA` is not implemented in the Miller [12] assembler, but is emitted using a `WORD` macro manually crafted according to the RISC-V specification [5].

In `mmuswitch`, which is called on a process switch, it is only necessary to perform a user-space flush as previously described. The MMU page mappings are wiped, but the portable parts of the Plan 9 kernel keep track of the pages for us, and the process pages will be mapped again by the fault handler and `putmmu` if the original process is re-scheduled in the future. Likewise, `mmurelease` only performs a user-space flush.

## 4.7 Trap handling

In accordance with the RISC-V specification [4] *exception* refers to an unusual condition occurring associated with an instruction, while *interrupt* refers to an external asynchronous event that may cause a transfer of control. Traps or trapping refers to the transfer of control to the trap handler caused by either an exception or an interrupt.

### 4.7.1 Initialization

Trap initialization is very simple on RISC-V. The address of the trap handling function must be written to the `stvec` CSR. As can seen in Figure 4.3 the lower two bits of `stvec` encode a mode. This means the address of the trap handler must be aligned with the lower 2 bits set to 0. A mode value of 00 means direct mode, and 01 means vectored mode [5]. Direct mode means all traps jump to the handler registered in the CSR. Vectored mode means a subset of trap causes will cause the hart to jump directly to an offset from the registered address. Direct mode is used in this implementation, so we simply write the address of the function `stvec_asm`, which is described in the next section, to the `stvec` control and status register.

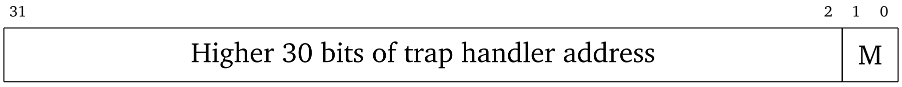

**Figure 4.3:** The `stvec` control and status register (CSR) for RV32.

### 4.7.2 Low-level trap handler

When an exception or interrupt happens, the PC is set to point to the `stvec_asm` function as shown in Code listing 4.7. This section describes the steps taken by this function and what must be done in C to determine the trap cause and handle it accordingly.

**Saving the registers**

First, the registers must be saved to an `Ureg` struct pointed to by `UREGADDR`. The `Ureg` definition by Miller [12] is used. Register `R4` is moved to the `sscratch` CSR for temporary storage while `R4` is used to index main memory. The `UREG_field` macro addresses relative to `R0`, which is hard-wired to zero, and does not explicitly use `R4`. As was discovered during this implementation, the offset used in the macro is too large to fit in a single instruction. This means the linker generates instructions to construct the address in `R4` as explained in Section 3.1.1, which cannot be seen by reading the assembly instructions. Therefore `R4` is skipped on line 9, and it is finally saved to main memory on lines 15 and 16. Some additional `Ureg` fields are then filled by reading supervisor-level trap-related CSRs.

**Selecting a stack location and entering C**

The `sstatus` CSR is then inspected to check if the trap was from supervisor-mode (S-mode). Specifically, bit 8 (`sstatus.SPP`) is 0 if the trap occurred while in user-mode (U-mode), or 1 if the hart was in S-mode [5]. If the trap was from U-mode, we get the location of the allocated kernel stack for the current process on lines 45 to 47 and add the kernel stack size to it. If the trap handler triggers a page fault while handling a trap from U-mode, such as when reading or writing process memory during a system call, we need to use a different stack to fix the page fault. We detect this condition by checking if the system was running in S-mode when the trap occurred. Aside from programming errors and assuming we correctly disable interrupts while in the kernel, these should be the only traps occurring while in S-mode. In either case a stack pointer is loaded in `R2`.
Before entering C code we have to set the static base (SB) as described in Section 3.1.1 and already done once in Section 4.3. This is done on line 56 in Code listing 4.7. We are then ready to call into C to perform the rest of the trap handling. The implementation of `c_trap` will be explained shortly. For now, it suffices to know that it returns a pointer to the `Ureg` struct we should restore registers from. Recall that return values are passed in `R8`.

### Restoring registers and exiting the trap handler

After returning to assembly, we start the restoration process by reading the stored PC from Ureg, and writing it to the sepc CSR. This is the address we want to resume execution from when exiting the trap handler. sepc was automatically written with the current PC when the trap occurred, but we want to load the saved value because the C handler might have changed it and because sepc might have been overwritten if we handled another trap in the middle of handling the one we are currently returning from. All the registers except R8 are then restored using the pointer in R8, and then finally R8 itself is restored. Finally, we issue the SRET instruction to return to the previous privilege level as indicated by sstatus.SPP and resume execution with the current registers from the PC stored in sepc. Note that sstatus.SPP is set to 0 by the implementation when executing SRET [5]. This ensures we return to the correct privilege mode even when a nested trap has been handled from S-mode. Also, note that the assembler does not implement SRET. Instead, it is a macro for a manually constructed WORD based on the instruction listing in the RISC-V specification [5].

**Code listing 4.7:** The assembly trap handler.

```
#define UREG_field(x) (UREGADDR + 4*(x))(R0)

TEXT stvec_asm(SB), $-4
        MOVW R4, CSR(sscratch)

        MOVW R1, UREG_field(1)
        MOVW R2, UREG_field(2)
        MOVW R3, UREG_field(3)
        // not R4
        MOVW R5, UREG_field(5)
        [...]
        MOVW R30, UREG_field(30)
        MOVW R31, UREG_field(31)

        MOVW CSR(sscratch), R1
        MOVW R1, UREG_field(4)

        MOVW CSR(sepc), R1
        MOVW R1, UREG_field(0)

        MOVW CSR(sstatus), R1
        MOVW R1, UREG_field(32)

        MOVW CSR(sie), R1
        MOVW R1, UREG_field(33)

        MOVW CSR(scause), R1
        MOVW R1, UREG_field(34)

        MOVW CSR(stval), R1
        MOVW R1, UREG_field(35)

        // Are we handling a trap from S-mode?
        // Faults may occur in S-mode while handling syscalls using the
        // per-process kernel stack, so we have a separate stack for this

        MOVW CSR(sstatus), R10

        MOVW $(0x100), R9
        AND R9, R10, R10
        BEQ R9, R10, use_smode_stack

        /* Fall through */
use_process_kstack: // Load mach->proc->kstack
        MOVW $(MACHADDR), R9
        MOVW 12(R9), R10 // proc* in R10
        MOVW 4(R10), R2
        ADD $(KSTACK - 4), R2
        JMP goto_c

use_smode_stack: // Use the S-mode trap stack
        MOVW $(INTR_STK_TOP), R2

        /* Fall through */
goto_c:
        MOVW $setSB(SB), R3
        JAL R1, c_trap(SB)

        // The Ureg address we should recover from is now in R8

        // Load (optionally) modified pc value from Ureg
        MOVW (0)(R8), R1
        MOVW R1, CSR(sepc)

        // Recover regs
        MOVW (4 * 1)(R8), R1
        MOVW (4 * 2)(R8), R2
        [...]
        MOVW (4 * 7)(R8), R7
        // not R8 yet
        MOVW (4 * 9)(R8), R9
        [...]
        MOVW (4 * 30)(R8), R30
        MOVW (4 * 31)(R8), R31

        MOVW (4 * 8)(R8), R8

        SRET
```

### Handling the trap in C

The c_trap function is quite simple. It is shown in Code listing 4.8. First, a copy of Ureg must be saved. The location chosen is the bottom of the stack address range for the trap handling. This means we must decide between two stack areas, as was done in the assembly handler. This is shown on lines 7 to 11 in the listing. ureg is then copied using memmove, and the rest of the trap handling works with the copy from now on. The cause field of ureg is then inspected to decide how to handle the trap. Recall that cause is written with the value of the scause CSR on trap entry.

**Code listing 4.8:** The `c_trap` function.

```
Ureg *c_trap() {
        Ureg *ureg = (Ureg *) UREGADDR;
        uintptr spp = (ureg->status & 0x100);
        u32 user_trap = !(spp >> 8);
        Ureg *ureg_copy_location;

        if (spp) {
                ureg_copy_location = UINT2PTR(INTR_STK_LOW_END);
        } else {
                ureg_copy_location = (Ureg *) up->kstack;
        }

        memmove(ureg_copy_location, ureg, sizeof(Ureg));
        ureg = ureg_copy_location;

        switch (ureg->cause) {
                case ErrInstrPageFault:
                        faultriscv(ureg, ureg->tval, user_trap, 1);
                        break;
                case ErrLoadPageFault:
                        faultriscv(ureg, ureg->tval, user_trap, 1);
                        break;
                case ErrStorePageFault:
                        faultriscv(ureg, ureg->tval, user_trap, 0);
                        break;
                case UECALL:
                        syscall(ureg);
                        break;

                default:
                        print("Trap␣cause␣not␣yet␣handled\n");
                        printureg(ureg);
                        spin();
                        break;
        }

        return ureg;
}
```

### 4.7.3 Page faults

Page fault handling happens mostly in portable code that calls into the architecture-specific MMU implementation described in Section 4.6. The `faultriscv` function is listed in Code listing 4.9. It is called from the trap handler and contains mostly bookkeeping. The most important part is the call to `fault` on line 11, which calls into the portable part of the kernel. It, in turn, then calls into `putmmu` if there is a page to map. The virtual address (`va`) parameter comes from `ureg.tval`, which is written with the value of the `stval` CSR. Its value is the address that caused the fault and needs to be mapped. Recall that `up` is a pointer to the currently scheduled process.

**Code listing 4.9:** The `faultriscv` function.

```
static void faultriscv(Ureg *ureg, uintptr va, int user, int read) {
        char buf[ERRMAX];

        if(up == nil) {
                printureg(ureg);
                panic("fault:␣nil␣up␣in␣faultriscv,␣accessing␣%#p", va);
        }
        int insyscall = up->insyscall;
        up->insyscall = 1;

        int n = fault(va, read);
        if(n < 0){
                if(!user){
                        printureg(ureg);
                        panic("fault:␣kernel␣accessing␣%#p", va);
                }

                snprint(buf, sizeof(buf), "sys:␣trap:␣fault␣%s␣va=%#p",
                        read ? "read": "write", va);
                postnote(up, 1, buf, NDebug);
        }
        up->insyscall = insyscall;
}
```

### 4.7.4 System calls

System calls are handled mostly in portable code. Before issuing an `ECALL` instruction in U-mode to perform a system call a system call number is placed in `R8`. This is handled by 9syscall, a part of libc for Plan 9. The 9syscall implementation by Miller [12] as described in Section 3.2 is used. The system call number is used to index the `systab` array, which is a table of function pointers to system call handler functions that all live in the portable part of the kernel. Code listing 4.10 shows the implementation of `syscall`. The arguments and the system call number are extracted on lines 7 and 9. The system call table is indexed and the retrieved function called on line 27, returning a value. The return value is written to `ureg` on line 39, meaning the value will be in the return register when registers are restored for the user process. The program counter in `ureg` is incremented by 4 bytes to skip over the `ECALL` instruction that was used to perform this system call. Note that even when using the compressed instruction format, `ECALL` is always 4 bytes.

**Code listing 4.10:** The `syscall` function.

```
1  void syscall(Ureg *ureg) {
2          m->syscall++;
3          up->insyscall = 1;
4          up->pc = ureg->pc;
5          spllo();
6          uintptr sp = ureg->sp;
7          Sargs *sargs = (Sargs *) (sp + BY2WD);
8
9          u32 syscallnr = ureg->r8;
```

```
        syscallfmt(syscallnr, ureg->pc, (va_list)(sargs));

        u32 ret = -1;
        char *e = nil;
        if(!waserror()){
                if(syscallnr >= nsyscall){
                        print("bad␣syscall␣nr␣%d␣pc␣%#p\n", syscallnr, ureg->pc);
                        postnote(up, 1, "sys:␣bad␣sys␣call", NDebug);
                        error(Ebadarg);
                }

                if(sp < (USTKTOP-BY2PG) || sp > (USTKTOP-sizeof(Sargs)-BY2WD)) {
                        validaddr(sp, sizeof(Sargs)+BY2WD, 0);
                }
                up->s = *(sargs);
                up->psstate = sysctab[syscallnr];

                ret = systab[syscallnr](up->s.args);
                poperror();
        } else {
                /* failure: save the error buffer for errstr */
                e = up->syserrstr;
                up->syserrstr = up->errstr;
                up->errstr = e;
        }

        up->insyscall = 0;
        splhi();

        ureg->r8 = ret;

        // Skip over ECALL instruction
        ureg->pc += 4;
}
```

Some system calls rely on architecture-specific functions. The `exec` system call, used to replace the currently running program with a new executable, relies on the `execregs` function. Code listing 4.11 shows the implementation. The stack pointer is calculated based on the amount of memory pushed to the new stack by the portable code. Then the number of args is pushed to the stack, and `ureg` is updated with the correct PC and SP. This code is very similar between architectures but not perfectly portable as, for instance, the stack pointer semantics can be different. The return value is the start of user/kernel shared data, as defined by the `Tos` struct (top of stack). It is used for bookkeeping and communication of per-process profiling data.

**Code listing 4.11:** The `execregs` function.

```
long execregs(ulong entry, ulong ssize, ulong nargs) {
        Ureg *ureg = (Ureg *) up->kstack;

        ulong *sp = (ulong*)(USTKTOP - ssize);
        *--sp = nargs;

        ureg->pc = entry;
        ureg->sp = PTR2UINT(sp);

        return USTKTOP - sizeof(Tos);
}
```

Similar to exec, forking processes also requires manually crafting the Ureg struct to ensure proper behaviour when a process resumes. This has to be implemented in the forkchild function. This was not implemented as it was not in the "critical path" to booting the system.

## 4.8 Crafting the first process

The first process is hand-crafted in the userinit function. An abbreviated version of the function is shown in Code listing 4.12. First, a Proc struct to represent the process is allocated and initialized with default values using the portable newproc function. A kernel stack (kstack) is also allocated and attached to the process. On lines 10 to 12 the PC and SP are set to the init0 function and top of the kernel stack. This is because the process will start out as a kernel process (kproc) executing the init0 function in S-mode. The user stack segment is created and initialized on lines 14 to 19. user_sp is a global variable that will be used when it is time to enter U-mode. The user text segment is created and initialized with a single mapped page on lines 21 to 26. The mapped page has the virtual address 0x00001000 (UTZERO) as was previously displayed in Figure 4.2. On line 29, the code for the user program (qrv32_initcode) to be executed is copied to the mapped text page. The qrv32_initcode program will be explained in Section 4.8.2. The process is then marked as ready to be scheduled. When it is scheduled, execution will start in init0, which will switch to user mode and jump to the user program code we just copied.

**Code listing 4.12:** The userinit function.

```
1   void userinit(void) {
2           /* no processes yet */
3           up = nil;
4
5           Proc *proc = newproc();
6           [...]
7
8           kstrdup(&proc->text, "*init*");
9
10          proc->sched.pc = PTR2UINT(init0);
11          proc->sched.sp = PTR2UINT(proc->kstack + KSTACK
12                                  -sizeof(up->s.args) - sizeof(uintptr));
13
14          Segment *seg = newseg(SG_STACK, USTKTOP-USTKSIZE, USTKSIZE/BY2PG);
15          seg->flushme++;
16          proc->seg[SSEG] = seg;
17          Page *pg = newpage(1, 0, USTKTOP-BY2PG);
18          segpage(seg, pg);
19          user_sp = stackspace(pg->pa);
20
21          seg = newseg(SG_TEXT, UTZERO, 1);
22          proc->seg[TSEG] = seg;
23          pg = newpage(1, 0, UTZERO);
```

```
        memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
        segpage(seg, pg);
        uintptr text_pa = seg->map[0]->pages[0]->pa;

        assert(sizeof(qrv32_initcode) < BY2PG);
        memmove(UINT2PTR(text_pa), qrv32_initcode, sizeof(qrv32_initcode));

        ready(proc);
}
```

### 4.8.1 Jumping to user mode

The `init0` function is very simple. It consists mainly of Plan 9 bookkeeping and setting environment variables. At the end it calls the function `touser` with `user_sp` from Code listing 4.12 as a parameter. The `touser` function is shown in Code listing 4.13. It begins by moving the stack passed as an argument from `R8` to `R2` (SP). `R8` is then re-used to hold the user program entry point, which is then written to the `sepc` CSR. The program entry is offset by 32 bytes from the start of the text segment because that's the size of the program header. With the user PC in `sepc`, the stack pointer in `R2`, and `sstatus.SPP` set to 0 the `SRET` instruction is used as if we were returning from trap handling, entering U-mode.

**Code listing 4.13:** The `touser` function.

```
TEXT touser(SB), 1, $-4
        MOVW R8, R2

        MOVW $(UTZERO + 0x20), R8
        MOVW R8, CSR(sepc)

        SRET // sstatus.SPP should be 0 at this point, so SRET returns to user mode
```

### 4.8.2 The first user program

The user-space program we just entered is called `qrv32_initcode`. It's purpose is to start the portable `boot` program with an `exec` system call, transforming itself into a `boot` process. It is done this way because it is easier to manually craft and jump into such a simple process, compared to the whole `boot` program. This is done almost identically in the Plan 9 kernels for other architectures, except they support more sophisticated passing of parameters to `boot`. The code in Code listing 4.14 and Code listing 4.15 is compiled together independently of the kernel as a user-space program. When compiling the kernel, the program is read byte-by-byte and written to a new file called `qrv32_initcode.h` that defines the byte array `qrv32_initcode`. Recall that this array was copied to the user text segment of the first process at the beginning of Section 4.8. The `startboot` function in Code listing 4.15 calls `exec` with a hard-coded command, transforming the program if succeeding, or exiting with an error string if unsuccessful.

**Code listing 4.14:** Jumping to `startboot` in the first user-space program.

```
TEXT _main(SB), $-4
        MOVW $setSB(SB), R3
        JAL R1, startboot(SB)
```

**Code listing 4.15:** The `startboot` function in the first user-space program.

```
/*
 * IMPORTANT!  DO NOT ADD LIBRARY CALLS TO THIS FILE.
 * The entire text image must fit on one page
 */

#include <u.h>
#include <libc.h>

char *boot_cmd[] = {"/boot/boot", "boot", nil};

void startboot()
{
        char buf[200];

        exec(boot_cmd[0], boot_cmd);

        rerrstr(buf, sizeof buf);
        buf[sizeof buf - 1] = '\0';
        _exits(buf);
}
```

### 4.8.3 The boot program

The program transforms into the boot program. It is a portable program that performs a lot of setup of the user space by connecting to a file server, starting a shell, authenticates, and more, depending on how the system is configured. Starting the boot program is the last non-portable operation necessary to boot the system. The boot program performs a number of system calls but eventually exits when it cannot bind the `/net` file. No drivers have been ported to this kernel, so this is as far as we get. Porting drivers have more to do with the specific hardware, rather than the instruction set architecture.

# Chapter 5

# Results

The results in this thesis are the working RISC-V Plan 9 implementation itself and the fact that no changes were necessary in the portable parts of the kernel. RISC-V is found to be a suitable architecture for a Plan 9 kernel. Likewise, Plan 9 is found to hold up to almost 30 years of evolution in the instruction set architecture space. The claims made about Plan 9s portability by Presotto *et al.* [6] are validated by porting Plan 9 to an architecture that would not even be invented for almost 20 years without changing anything in the portable source code. Some specific observations about the process and tools are elaborated on in this chapter.

## 5.1 Virtual memory

In Section 4.6 it is noted that Plan 9 would require changes in the portable code to take advantage of the Sv32 virtual memory schemes ability to address 34 bits of physical memory. This is not regarded as something that breaks Plan 9s portability, as it is an optional mechanism that software may make use of, not something that the architecture imposes on the software. It is not reasonable to expect the software to utilize every optional mechanism an architecture provides to qualify as being portable, even though support for this particular mechanism could be quite useful.

The portability mechanism for page table entry (PTE) flags is a bit awkward. As elaborated on in Section 4.6.2, the correct PTE flags must be set based on the semi-portable flags given by the portable code. They are considered semi-portable because, while they can be a one-to-one match with some architecture-dependent flags, they carry a semantic meaning that may not directly match a specific architecture-dependent flag. Instead of going through this trouble, the portable code could just pass the segment type to the architecture-specific code and let it determine the appropriate flags.

## 5.2 Processes and system calls

Code listing B.1 in Appendix B shows the trace for all system calls performed by the first process. This is included to demonstrate the kernels ability to support the user space. In total, 36 system calls are performed and served before the boot process exits due to the `/net` device file not existing.

The architecture-specific functionality needed to support the `fork` and `rfork` system calls was not implemented. Neither were timer interrupts. These two relatively small additions together would arguably make the implementation "complete", although as with any operating system, there is an endless supply of devices to make drivers for. Neither of the two changes are considered likely to require changes in the portable source code.

## 5.3 Toolchain usage

In Section 4.7.2 an issue with the use of the `R4` register in the Miller toolchain [12] came up. The cause of the issue won't be re-stated here, but the effect is "hidden" use of the `R4` register, which is highly unexpected when writing assembly code. Workarounds and avoiding the problem is possible if the programmer is aware of the issue. The issue is seen as a sign of immaturity in the toolchain and something that should be re-designed to get more predictable results.

# Chapter 6

# Conclusion and future work

In short, the conclusion is that the claims made about Plan 9s portability [6] when it was first introduced still hold up. Even if there are some minor aspects that, in hindsight, could be improved, Plan 9 manages to keep almost everything that could realistically be written as portable code portable. Supervisor-mode RISC-V with SBI is considered a suitable target for Plan 9.

Portability aside, Plan 9 does not come across as a viable operating system for general-purpose computing in the current year. Today, developers expect to be able to use standard tools such as GCC or LLVM-based toolchains regardless of target architecture. While the Plan 9 C compilers are relatively, compared to mainstream optimizing compilers, low-effort to implement, the C custom language they support, along with the scripting language `rc`, remain the only well-supported programming languages available on Plan 9. As there are already operating systems running everywhere, even on tiny embedded devices, the issue is not only getting an operating system up and running quickly, but support for the user-space and application programmers have come to expect and rely on. Plan 9 falls short in this regard.

## 6.1 Future work

The implementation is currently very minimal and missing functionality for supporting the `rfork` system call as well as timer interrupts. These will have to be implemented for the port to be of any practical use. To boot a fully-functional Plan 9 user-space some drivers are needed. At the very least, a driver providing network capabilities is needed to integrate a machine running this implementation as a `cpuserver` in a Plan 9 network. To function as a full terminal, drivers for graphical interaction are needed. A 64-bit version using the 64-bit toolchain from Miller [12] would also be interesting, particularly for general purpose computing or server computing purposes. In this space, a multi-core port is also of interest. On the other end of the scale, a toolchain and kernel for the RV32E standard could be interesting if RV32E sees wider adoption in the future. RV32E is the 32-bit embedded base ISA. It is a subset of the RV32I base ISA and only has 16 integer registers, compared to RV32Is 32 [4].

# Bibliography

[1] A. Slettemark, 'Porting Plan 9 to RISC-V,' Department of Computer Science, NTNU – Norwegian University of Science and Technology, Project report in TDT4501, Dec. 2020.

[2] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey *et al.*, 'Plan 9 from Bell Labs,' *Computing systems*, vol. 8, no. 3, pp. 221–254, 1995.

[3] R. Miller, 'The First Unix Port,' in *Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association, 1998, p. 25.

[4] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*. RISC-V Foundation, Dec. 2019.

[5] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, Jun. 2019.

[6] D. Presotto, R. Pike, K. Thompson and H. Trickey, 'Plan 9, a distributed system,'

[7] P. Dabbelt and A. Patra, 'RISC-V Supervisor Binary Interface Specification,' [Accessed 28th May 2021]. [Online]. Available: `https://github.com/riscv/riscv-sbi-doc/blob/master/riscv-sbi.adoc`.

[8] P. Dabbelt, S. O'Rear, K. Cheng, A. Waterman, M. Clark, A. Bradbury *et al.*, 'RISC-V ELF psABI Specification,' [Accessed 27th May 2021]. [Online]. Available: `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`.

[9] K. Thompson, 'Plan 9 C compilers,' in *Proceedings of the Summer 1990 UKUUG Conference*, 1990, pp. 41–51.

[10] S. C. Johnson *et al.*, 'Yacc: Yet another compiler-compiler,' in *Computing Science Technical Report No. 32*, Bell Laboratories Murray Hill, NJ, 1975.

[11] R. Pike, 'A Manual for the Plan 9 assembler,' [Accessed 28th May 2021]. [Online]. Available: `https://9p.io/sys/doc/asm.html`.

[12] R. Miller, ‘A Plan 9 C Compiler for RV32GC and RV64GC,’ *London Open Source Meetup for RISC-V*, Oct. 2020, [Accessed 28th May 2021]. [Online]. Available: `https://riscv.org/news/2020/10/a-plan-9-c-compiler-for-rv32gc-and-rv64gc/`.

[13] F. Bellard, ‘QEMU, a fast and portable dynamic translator,’ in *USENIX Annual Technical Conference, FREENIX Track*, USENIX Association, 2005, pp. 41–46.

[14] *OpenSBI*. RISC-V International, [Accessed 5th June 2021]. [Online]. Available: `https://github.com/riscv/opensbi`.

[15] B. Ford and R. Cox, ‘Vx32: Lightweight user-level sandboxing on the x86.,’ in *USENIX Annual Technical Conference*, 2008, pp. 293–306.

# Appendix A

# Replicating the development environment

This appendix describes how to set up the development environment used for the implementation. There are many different ways to get a Plan 9 environment. This setup uses the 9vx environment on a x86_64 Linux host machine. 9vx is a virtual environment enabled by the vx32 sandboxing library [15].

## A.1 Running 9vx

First, 9vx must be downloaded and extracted from the official site[1]. Disregard the Plan 9 file system included in the download. If, as was the case for myself, the `9vx.Linux` executable does not work, 9vx can be compiled from source. The source code is distributed with the vx32 [15] library, which can be downloaded from its official site[2]. Add the 9vx executable to your `PATH`. Then, a Plan 9 ISO file (CD image) must be downloaded from the Plan 9 site[3]. Extract it to a directory, using for instance `7z x plan9.iso`. You can now run Plan 9 from this directory using `9vx -r . -u glenda`. If everything went as expected, you should now have a graphical Plan 9 environment running the rio windowing system.

## A.2 Getting the Miller toolchain and RISC-V kernel source

Start by cloning my public GitHub repository[4]. If you want to get the exact version delivered with this thesis, `git checkout` the `thesis_state` tag. It's now time to create a patch. The command used to create a patch file called `riscv.patch` including both the Miller [12] toolchain and the RISC-V kernel is shown in Code listing A.1.

[1] https://swtch.com/9vx/

[2] https://pdos.csail.mit.edu/~baford/vm/

[3] https://9p.io/plan9/download.html

[4] https://github.com/aslettemark/plan9_riscv

**Code listing A.1:** Git command to generate patch file.

```
git format-patch upstream..HEAD --stdout > riscv.patch
```

Navigate back to the root directory where you extracted the Plan 9 file system in the previous section. Execute `git apply path/to/riscv.patch`. You can now start Plan 9 using `9vx -r . -u glenda`. Inside Plan 9, navigate to `/sys/src` and execute `mk release` to update your user-space with the new commands and library updates. This might take several minutes. It is recommended to enable scrolling in the terminal. After your user-space has compiled, execute `objtype=riscv mk libs` to compile system libraries for RISC-V. You should now be ready to compile the kernel.

Navigate to `/sys/src/9/qrv32`. Execute `mk`. The kernel should now compile successfully. The output file name is `9qrv32`. This is the kernel that QEMU will run later.

## A.3 Compiling OpenSBI

Back on the host machine, start by getting a cross-compiler for 32-bit RISC-V. My cross-compiler is GCC version 9.2.0 with the `ilp32` (soft float) ABI. Set the environment variable `CROSS_COMPILE` to the prefix allowing a Makefile to use the different programs in the toolchain. For example, mine is `/opt/riscv32gc_ilp32/bin/riscv32-unknown-linux-gnu-`.

The OpenSBI source code is obtained by cloning the official GitHub repository[5]. The most recent tested version for this project is commit hash `54d7def6c254058f9458a0e26205b3c93a48bb42` (May 24, 2021). With your cross-compiler installed, compile OpenSBI firmwares for QEMU by running `make PLATFORM=generic`. The firmware we want to use should now be located at `build/platform/generic/firmware/fw_jump.bin`.

## A.4 Running with QEMU

The `qemu-system-riscv32` emulator from QEMU version 5.0.0 was used. The full command to run the compiled kernel using the previously compiled OpenSBI `fw_jump` firmware is shown in Code listing A.2. Substitute your exact path for the `-bios` and `-kernel` options. This configuration lets you use Ctrl-C to exit the kernel, which is very convenient. To read the machine state in the QEMU monitor connect using telnet with the command `telnet localhost 55555`.

[5] `https://github.com/riscv/opensbi`

**Code listing A.2:** QEMU command for running kernel with OpenSBI firmware.

```
qemu-system-riscv32 -M virt -m 256M -nographic
    -bios [..]/build/platform/generic/firmware/fw_jump.bin
    -kernel [..]/sys/src/9/qrv32/9qrv32
    -monitor telnet:127.0.0.1:55555,server,nowait
```

# Appendix B

# System call trace for the first process

Code listing B.1 displays the trace for all system calls performed. The format is process id (pid), program name, system call name, and arguments. String arguments are displayed, reading from the character pointer. The formatting is done by syscallfmt.c in port/.

**Code listing B.1:** Trace of all performed system calls.

```
1 *init* Open 10bc 0x11e0/"#c/cons" 0x0
1 *init* Open 10bc 0x11e0/"#c/cons" 0x1
1 *init* Open 10bc 0x11e0/"#c/cons" 0x1
1 *init* Bind 10d4 0x11b8/"#c" 0x11c8/"/dev" 0x2
1 *init* Bind 10d4 0x11c4/"#ec" 0x11d0/"/env" 0x2
1 *init* Bind 10d4 0x11bc/"#e" 0x11d0/"/env" 0x6
1 *init* Bind 10d4 0x11c0/"#s" 0x11d8/"/srv" 0x4
1 *init* Exec 10c8 0x11f8/"/boot/boot" 0x11f8/"/boot/boot" 0x1203/"boot"
1 boot Close 8aa4 0
1 boot Close 8aa4 1
1 boot Close 8aa4 2
1 boot Bind 8ab0 0x13868/"#c" 0x1386b/"/dev" 0x1
1 boot Open 8a2c 0x13870/"/dev/cons" 0x0
1 boot Open 8a2c 0x1387a/"/dev/cons" 0x1
1 boot Open 8a2c 0x13884/"/dev/cons" 0x1
1 boot Bind 8ab0 0x1388e/"#ec" 0x13892/"/env" 0x0
1 boot Bind 8ab0 0x13897/"#e" 0x1389a/"/env" 0x5
1 boot Bind 8ab0 0x1389f/"#s" 0x138a2/"/srv/" 0x4
1 boot Open 8a2c 0x7ffffec4/"/env/debugboot" 0x0
1 boot Open 8a2c 0x7ffffec4/"/env/nousbboot" 0x0
1 boot Open 8a2c 0x13ab1/"#e/cputype" 0x0
1 boot Pread 8a16 0 0x101a8 63 -1
1 boot Close 8aa4 0
1 boot Brk ce64 0x16950
1 boot Stat 89b6 0x14e68/"#u/usb/ctl" 0x15994 115
1 boot Open 8a2c 0x13b37/"#e/nobootprompt" 0x0
1 boot Open 8a2c 0x13b8d/"#e/bootargs" 0x0
1 boot Pwrite ce4e 1  0x7ffffbc0/"root.is.from.(tcp)[tcp]:." 25 -1
1 boot Pread 8a16 0 0x7ffffd1c 255 -1
1 boot Bind 8ab0 0x14ec8/"#I" 0x100a8/"/net" 0x2
1 boot Errstr 8a82 0x7ffffe74 128
```

```
1 boot Pwrite ce4e 2  0x7ffffd18/"boot:.bind.#I:.’/net’.file.does.not.exist." 42 -1
1 boot Open 8a2c 0x10550/"#c/pid" 0x0
1 boot Pread 8a16 0 0x7ffffe34 20 -1
1 boot Close 8aa4 0
1 boot Exits 8ac8 0x15978/"bind #I: ’/net’ file does not exist"
```

NTNU

Kunnskap for en bedre verden