Bachelor Thesis

RISC-V S-Mode-Hosted Bare-Metal Selfie

by

Martin Fischer

Submitted to the Department of Computer Sciences
at the Paris Lodron University of Salzburg
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

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

September 2020

Abstract

Modern operating systems like Linux run on a great variety of different machine architectures and have complicated mechanisms for process and resource management. Since this makes understanding their internals a tedious task, we present a simple preemptive multitasking kernel that targets a 64-bit RISC-V platform and which is tailored to run binaries compiled by selfie through the implementation of its Application Binary Interface. It runs in supervisor-mode, executes spatially isolated processes in user-mode, and communicates through the Supervisor Binary Interface with OpenSBI, a hardware abstraction layer, running in machine-mode; all while sharing core parts of its code base with a library operating system that is described in another bachelor thesis. The main difficulties that arise through the use of memory virtualization are discussed and solutions for them are presented. The kernel is able to execute on top of real RISC-V hardware in the form of a SiFive HiFive Unleashed.

Contents

1 Introduction

We live in a world controlled by software. The majority of modern computing devices execute programs consisting of thousands, sometimes even millions of lines of code. Every day this number grows substantially due to the barriers of entry in computer programming becoming increasingly lower. One of the reasons why this is the case, is that operating systems provide a huge abstraction layer of the underlying hardware platforms so that the required knowledge for using computers shrinks drastically. They must manage an immense amount of processes that execute code on those machines, have to offer secure and efficient execution environments, and have to provide a seemingly unbounded supply of resources. Some of the aforementioned properties are achieved through the help of hardware features, while others are accomplished by implementing various sophisticated algorithms. Therefore, these operating systems pose as very complex systems.
We provide a multitasking kernel implementation that targets a 64-bit RISC-V[1] platform and is stripped of all those intricate mechanisms for process and resource management and instead focus on simplicity while still providing a resemblance of a modern operating system through the use of advanced hardware support for paging, trap handling, and preemption. It operates on top of a hardware abstraction layer in the form of OpenSBI[2] and manages processes that run binaries compiled by selfie[3] in a less privileged execution mode. Internal functionalities like the file system have been constructed to make it possible to share them between this implementation and a library operating system. Due to strict adherence to the official specifications, our kernel is able to run on native hardware in the form of a SiFive HiFive Unleashed[4].

1.1 Collaboration

This thesis is based on the collaboration with Marcell Haritopoulos, with whom this kernel and an additional library operating system have been designed and implemented. They share their code for setting up the C runtime, the file system, and for SBI communication and provide the same interface in the form of five library functions. In his thesis, he talks about the components that our implementations have in common. Both of them should be read together in order to create a complete understanding of our work.

[1] `https://riscv.org/`
[2] `https://github.com/riscv/opensbi/`
[3] `https://github.com/cksystemsteaching/selfie`
[4] `https://www.sifive.com/boards/hifive-unleashed`

2 Prerequisites

2.1 Selfie

Selfie is an ongoing project of the Computational Systems Group[5] at the Department of Computer Sciences at the University of Salzburg, Austria [13]. It is used in teaching the basic principles of compiler construction, operating systems and hypervisors [13, 11]. Selfie consists of around 10 000 lines of code within one single file and contains:

- a self-compiling compiler for C*, a subset of C, including a small library called libcstar, that compiles down to RISC-U, a tiny subset of RISC-V's 64-bit Base Integer Instruction Set and its Standard Extension for Integer Multiplication and Division [13, 16],
- a self-emulating RISC-U emulator named mipster, and
- a self-hosting RISC-U hypervisor called hypster [13].

Selfie interfaces with its environment through the use of the following five system calls:

- `exit` to enable a program to exit,
- `read`, `write`, and `openat` that provide I/O related functionality in the form of reading from, writing to, and opening files, and
- `brk` in order to allow dynamic memory allocation [13].

In the past, selfie has been extended to support an official RISC-V toolchain consisting of spike[6], a RISC-V ISA simulator, and pk[7], a small kernel that can be used to run RISC-V ELF binaries by delegating I/O related system calls to the underlying environment (e.g. Linux) [1]. The aim of our work is to take this support even further by allowing selfie-compiled binaries to run on native RISC-V hardware by providing our own implementation of a small operating system kernel tailored to run those executables. Additionally, we explore native hardware functionalities, like for example memory virtualization, that are emulated in mipster [13] and are essential for today's operating systems.

2.2 Supervisor Binary Interface and OpenSBI

Due to a blend of for example hardware-specific interfaces with the operating systems that run on them, the development of new hardware platforms can be quite difficult [17], if they should be able to run this existing software. For this reason, [17] defines multiple different implementation stacks in order to standardize and ease the development of new RISC-V technologies. One of those stacks is presented in Figure 2.1. Here, we have the standard way of communication between an application

[5] `http://www.cs.uni-salzburg.at/~ck`
[6] `https://github.com/riscv/riscv-isa-sim`
[7] `https://github.com/riscv/riscv-pk`

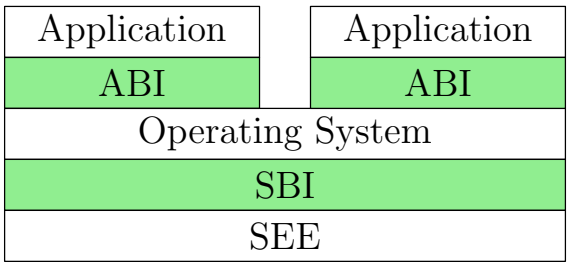

Figure 2.1: A RISC-V software stack supporting an operating system (recreated from [17])

and an operating system through the use of an application binary interface (ABI). Additionally, the operating system communicates with the supervisor execution environment (SEE) through the use of a supervisor binary interface (SBI) [17]. In our case, the SEE is the combination of a bootloader and platform-specific firmware [2] provided by OpenSBI version 0.7. OpenSBI implements version 0.2 of the official RISC-V Supervisor Binary Interface Specification[8], whose functionality is used within this work. The SBI consists of several extensions with different functionalities [4], all of which are supported by OpenSBI [2]. Since certain features that we use within our implementation require more privileges than we have available at our hand, we rely on the functionalities defined by [4] that are accessible to us through the use of environment calls [4]. As [4] specifies, SBI calls for a specific function are performed by loading its extension ID into `a7` and its extension-specific function ID into `a6` before executing an `ecall` instruction. Additionally, return values are passed in `a0` and `a1` with `a0` containing an error code and `a1` returning a possible return value. Here, non-zero values in `a0` indicate an error.

2.3 Executable and Linking Format

The Executable and Linking Format (ELF) is a file format that can be used for instance for executables or shared objects [9]. The format is used in its 64-bit variation within selfie in order to create executable files [13]. For this reason, we will only discuss the 64-bit but not the 32-bit version of ELF's structure. ELF files consist of a mandatory ELF header that is succeeded by a program-header table and/or a section-header table [9]. Since selfie makes no use of section headers [13], we will refrain from discussing them and only mention their existence within the ELF header for the sake of completeness.

2.3.1 ELF Header

The following explanation of the ELF header and its contents that are described within this section is based on the information available in [9].
The ELF header begins at offset zero within the file and has the structure presented in Figure 2.2. The meaning of each field is as follows:

- `e_ident` is an array containing interpretation information about the file. It consists of the following entries:

[8] `https://github.com/riscv/riscv-sbi-doc/`

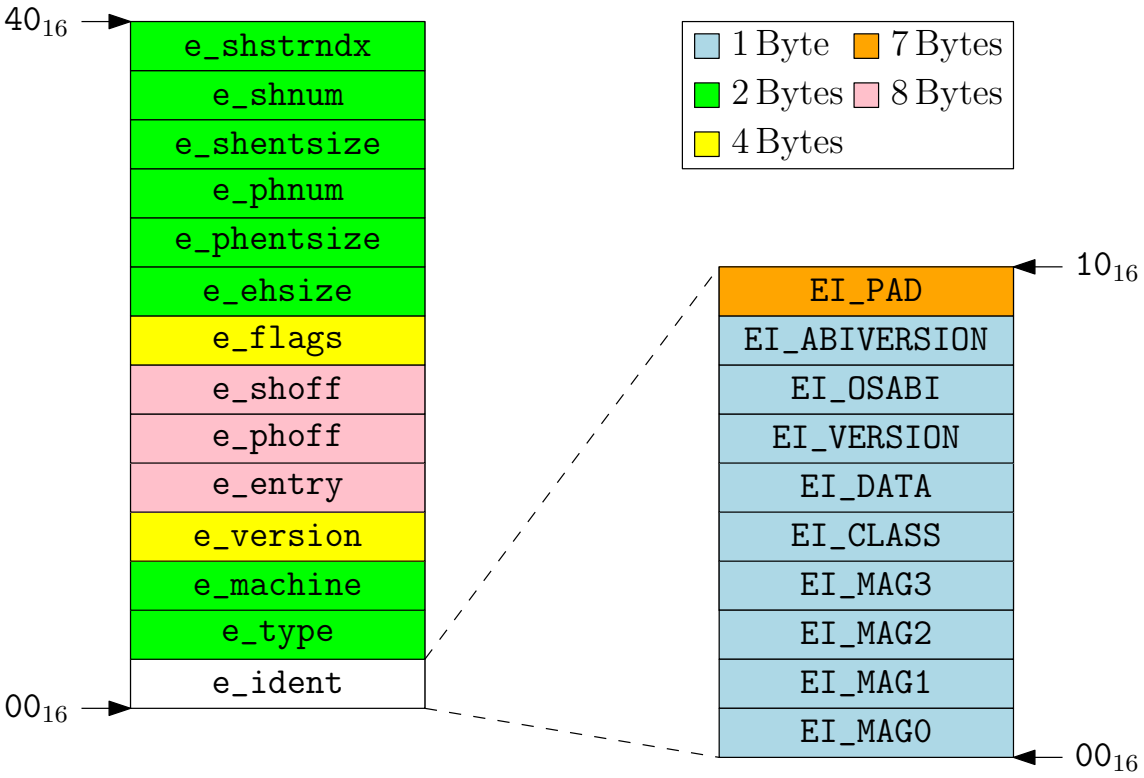

Figure 2.2: ELF header structure

  - `EI_MAG`x: This is the magic number used to identify ELF files. It is composed of the four characters `0x7F`, `'E'`, `'L'`, and `'F'`.
  - `EI_CLASS`: This entry contains information whether this binary utilizes a 32-bit or a 64-bit architecture.
  - `EI_DATA`: This byte is used to differentiate between little-endian and big-endian systems.
  - `EI_VERSION`: Here, the version number of the ELF specification used for this binary is specified. At the moment, the only legal value for this byte is 1 [9, 7].
  - `EI_OSABI`: This byte contains an ID for the operating system and the corresponding ABI that this binary has been compiled for.
  - `EI_ABIVERSION`: Here, an ABI version dependent on the value of `EI_OSABI` can be specified.
  - `EI_PAD`: The remaining seven bytes of `e_ident` are padding bytes reserved for future use and are set to 0.

- `e_type` represents whether this file is for example an executable or a shared object.
- `e_machine` specifies the machine architecture this binary has been compiled for.
- `e_version` identifies the file version. Only 1 is a valid version number for this entry [9, 7].

- `e_entry` is the virtual address of a process's entry point.
- `e_phoff` is the offset of the program-header table in bytes.
- `e_shoff` saves the section-header table's offset in bytes.
- `e_flags` is an entry for processor-specific flags.
- `e_ehsize` specifies the ELF header's size in bytes.
- `e_phentsize` and `e_phnum` hold the size of each program header in the program-header table and the number of program headers in this table respectively.
- `e_shentsize` and `e_shnum` specify the size of each entry in the section-header table and the amount of entries respectively.
- `e_shstrndx` contains the index of the section header entry used to save the section name string table.

2.3.2 Program Header

This section's explanation of program headers within ELF files has been sourced from [9].
If the file includes a program-header table, it contains one or more program headers with the structure shown in Figure 2.3. Each of them describes a segment within

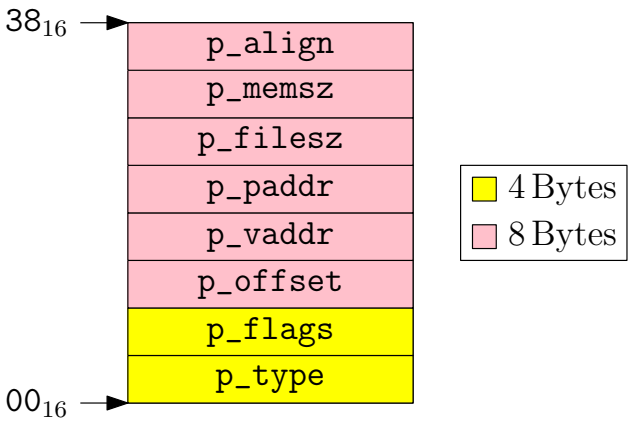

Figure 2.3: ELF program-header entry

the file. The usage of the entries within a program header is as follows:

- `p_type` specifies what type of segment this program header describes. Examples are loadable segments or segments containing information about an interpreter that should be used to execute this ELF binary.
- `p_flags` marks a segment as a combination of the properties of being executable, writable, and readable.
- `p_offset` describes the file offset of the actual segment this program header identifies.

- `p_vaddr` represents the virtual address at which this segment is loaded into memory.
- `p_paddr` is reserved for saving the segment's physical address within memory.
- `p_filesz` specifies the number of bytes this segment takes up in the file.
- `p_memsz` is the number of bytes this segment takes up in memory.
- `p_align` defines the alignment of segments within memory and within the ELF file.

3 RISC-V Machine Model

RISC-V is an ISA family that is heavily based on the use of extensions which allows many specialized implementation designs. Due to this reason, and the fact that many parts of the specification are still under development, we define our machine model as an RV64GC implementation with additional support for S-mode and U-mode. Additionally, we require the Bare and Sv39 addressing modes as they are specified in [16] and [17].
In this chapter, we aim to explain parts of our machine model, that we see as essential in understanding the development of a kernel.

3.1 Privilege Levels

If there was no isolation between an operating system and user applications running on the same machine, there would be nothing to stop a malfunctioning or rogue user application from for example manipulating the kernel's memory [15]. For this reason, there often exists hardware support in the form of modes or privilege levels to distinguish between operating system components and the user applications they manage [15].
As noted in [17], RISC-V specifies three different privilege levels: machine-mode (M-mode), supervisor-mode (S-mode), and User-mode (U-mode), where M-mode is the highest and U-mode the lowest privilege level. Furthermore, they state, that while M-mode has maximal possible access to the machine implementation, S-mode and especially U-mode have restrictions regarding for example the access to certain registers or the execution of certain instructions. Since only M-mode is mandatory [17], we must specify that our model of a RISC-V machine implements all three privilege levels.

3.2 Control and Status Registers

In simple implementations of a computer, during runtime, there may be no ways of direct interaction between the machine and software that runs on top of it. While it may be feasible to run specialized single-purpose software in such environments, it would be impossible to run a modern operating system on such hardware. Part of this reason is that today's operating systems heavily rely on communication with the underlying hardware implementation. This is one of the motives behind the existence of Control and Status Registers (CSRs), where each of them has a special purpose by allowing software to provide information to the hardware or vice versa. RISC-V provides room for up to 4096 possible CSRs, most of which are currently undefined [17]. Since not all of the CSRs are accessible to every single privilege level [17], a certain degree of isolation is provided. While some of those registers are used for things such as signaling floating-point exceptions or debugging support [17], we are only going to focus on CSRs related to timing and S-mode specific CSRs.

3.3 Traps

RISC-V defines a trap as a control transfer to a specified trap handler that can be caused by either an exception induced by an anomaly linked to a specific instruction, or an interrupt, which represents an asynchronous event [16]. Traps can be divided into horizontal and vertical traps, where the former ones are traps where the machine remains at the same privilege level, while the latter ones increase the privilege level [17] (e.g. a system call from a process running in U-mode to an S-mode kernel). As horizontal traps remain entirely unused in our implementation, they will not be discussed in this work.

3.4 Memory Virtualization

For 64-bit RISC-V, [17] defines three possible memory addressing modes: Bare, Sv39, and Sv48. There, Bare refers to a memory addressing scheme with non-existing virtual-address spaces, i.e. no address translation is performed. For Sv39 and Sv48 on the other hand, both are stated to provide a page-based virtual addressing scheme with virtual addresses of lengths 39 bits and 48 bits respectively being translated to physical addresses with a length of 56 bits. Due to a virtual memory size of 512 GiB being sufficient for our needs, we are going to focus on Sv39 as a memory virtualization technique.
As mentioned in [17], Sv39 page tables are constructed as radix-trees with three levels. Even though they define Sv39 to also support megapages and gigapages with sizes of 2 MiB and 1 GiB respectively, we will focus on the standard 4 KiB pages due to the larger page-sizes remaining unused in our implementation. Since every page table within the tree holds 512 page-table entries (PTEs), each of a size of 8 B [17], they all fit exactly into 4 KiB pages. While the virtual addresses themselves have lengths of 39 bits, bits 63–39 have to be equal to bit 38 (similar to sign extension) in order for them to be actually valid [17]. This leads to the lowest 256 GiB and the highest 256 GiB of the virtual-address space being addressable, while the space in between consists of invalid virtual addresses. We refer to the addressable space as *lower half* and *higher half* respectively (see Figure 3.1). A 39-bit virtual address

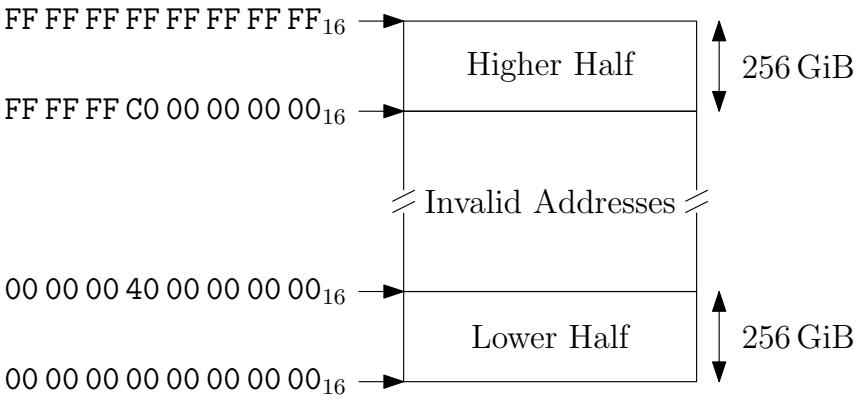

Figure 3.1: Sv39 memory areas

is split up into a 27-bit virtual page number (VPN) and a 12-bit offset within a page. Furthermore, in order to enable the three-leveled page-table structure, the

VPN is divided further into the VPN[2], the VPN[1], and the VPN[0], with each of them taking up 9 bits [17]. Those parts are used to perform a page-table walk where each of them represents a page-table entry's index within a page table, so that a VPN can be translated to a physical page number (PPN) [17]. A page-table walk for a given virtual address works as presented in Figure 3.3: (1) VPN[2] is used to retrieve the PPN of the mid-level page-table from the root page-table, (2) VPN[1] is used to retrieve the leaf page-table's PPN from the mid-level page-table, (3) the page frame's PPN is acquired by retrieving the PTE at index VPN[0] in the leaf page table, and (4) the offset is added to the page frame's base address, so that the corresponding physical address is formed [17].

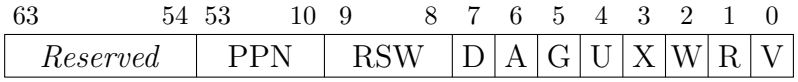

Figure 3.2: Sv39 page-table entry (adapted from [17])

As can be seen in Figure 3.2, every page-table entry consists of 10 bits that are reserved for future use, 44 bits for the PPN that this PTE points to, the 2-bit RSW field that can be freely used by a kernel, and eight additional 1-bit flags: (1) the dirty (D) bit and accessed (A) bit which can be used to for example implement swapping, (2) the global (G) bit that marks a mapping as global (i.e. it exists in every virtual-address space), (3) the U bit which indicates that a page is accessible from U-mode, (4) the X, W, and R bits that mark a page as executable, writable, and/or readable respectively, and (5) the valid (V) bit that is utilized to show that this entry is actually valid [17].

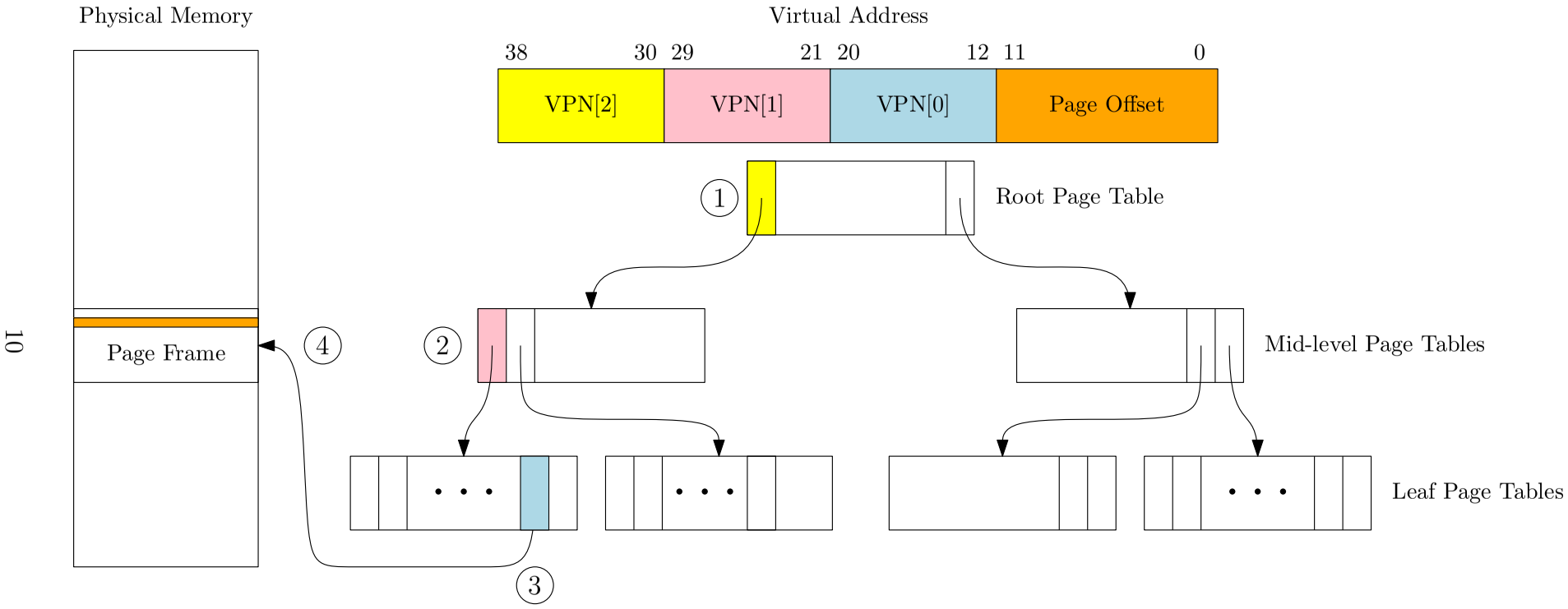

Figure 3.3: Sv39 page-table walk

4 Implementation

Many modern operating systems like Linux or the BSDs all have one thing in common: a huge complexity. While there are many simple educational operating systems like for example Xv6[9] or Pintos[10] available, we decided to start from the ground up and implement our own operating system kernel targeting the RISC-V architecture. Our software stack consists of OpenSBI running in M-mode, our kernel executing from S-mode, and user processes residing in U-mode (see Figure 4.1). Preemptive multitasking functionality is given, together with support for spatial isolation through the use of paging.

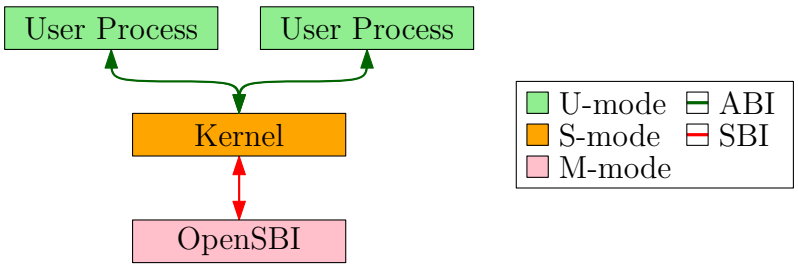

Figure 4.1: Our implementation's software stack

4.1 Context Management

Modern operating systems are preemptive and provide multitasking functionality. One problem that arises with those features is that if there were no preventive measures being taken by the operating system, a process's state would be in danger of being modified if another program is executed on the same machine. To avoid this issue, an operating system defines the notion of contexts that keep track of a certain machine state, which allows it to provide the aforementioned properties by itself.
While our definition of a context includes that of being a representation of parts of a machine state, we also save all the relevant information for controlling a process in a `context` structure (see Listing 4.1).

```
1 struct context {
2     uint64_t id;
3     struct pt_entry* pt;
4     uint64_t program_break;
5     struct registers saved_regs;
6     struct memory_boundaries legal_memory_boundaries;
7     FILEDESC open_files[MAX_NUM_FILES];
8 };
```

Listing 4.1: Structure used to save context information

[9] https://pdos.csail.mit.edu/6.828/2020/xv6.html
[10] https://web.stanford.edu/~ouster/cgi-bin/cs140-spring20/index.php

The machine state is saved both directly and indirectly: Since the 32 general-purpose registers and the program counter are the only registers used by RISC-U [13], they can be saved directly in a dedicated `registers` structure within the `context` structure. Due to a C compiler being free to insert padding bytes between members in structures [8], handling them in assembly code by for example iterating over their members proves to be a dangerous task. For this reason, the `registers` structure, which needs to be addressed at certain points in assembly code, is marked with the `packed` attribute. This GNU C extension instructs the compiler to not insert padding bytes [6]. The process's memory on the other hand, does not have such a small upper bound, in fact with up to 512 GiB for the Sv39 paging mechanism it is rather large, so that storing it directly in the `context` is infeasible. For this reason, a pointer `pt` to the process's root page table is used as an indirect way of keeping track of its entire memory.
Memory allocation is done by the kernel through the `brk` system call. Therefore the kernel needs to save a process's program break and additionally needs to implement methods to determine whether memory accesses are legal or not. To support the latter functionality, a dedicated `memory_boundaries` structure (see Listing 4.2) is embedded into a `context`. It saves the boundaries of the three regions *lo*, *mid*, and *hi*. Here, the lo region provides boundaries for the memory containing the code segment, data segment, and the heap, while the mid region identifies the stack. The hi region on the other hand, is not used to identify where user memory lies, but instead describes where the kernel's stack and its trampoline code, both of which are mapped into each user process's virtual memory, reside within the user process's virtual-address space (see Figure 4.2).

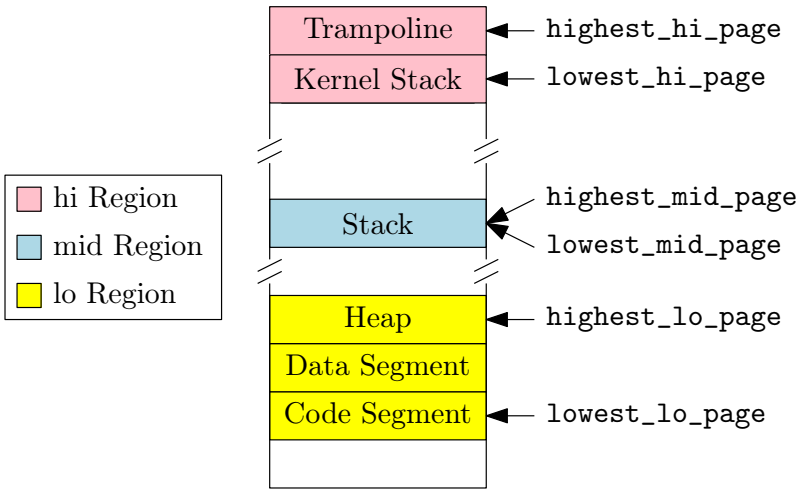

Figure 4.2: Memory regions within a virtual address space

Since processes access files by issuing an `openat` system call, which causes them to receive a file descriptor, the kernel needs to keep track of all the files that a program has opened. Our way of doing this, is to embed an array `open_files` of file descriptors directly into the `context` structure. It has the length of the maximum number of files that can be saved in the file system.

```
1 struct memory_boundaries {
2   uint64_t lowest_lo_page;
3   uint64_t highest_lo_page;
4   uint64_t lowest_mid_page;
5   uint64_t highest_mid_page;
6   uint64_t lowest_hi_page;
7   uint64_t highest_hi_page;
8 };
```

Listing 4.2: Structure used to divide a virtual-address space into its lo, mid, and hi regions

Last but not least, in order to uniquely identify each context, an ID field is put into the `context` structure.
As the only form of dynamic memory allocation within our kernel is the page allocator, it is not possible to allocate memory chunks of variable sizes. For this reason, a `context` cannot be allocated during runtime, but must be embedded into the binary at compile time instead. We choose to reserve space for the kernel context and for a predefined amount of freely available contexts. In order to provide an allocation and freeing mechanism, together with a simple implementation of a round-robin scheduler, each context is wrapped into a `context_manager` (see Listing 4.3).

```
1 struct context_manager {
2   struct context context;
3   bool is_used;
4   struct context_manager* prev_scheduled;
5   struct context_manager* next_scheduled;
6 };
```

Listing 4.3: Structure used to manage contexts and schedule processes

Here, the Boolean value `is_used` represents an indicator for the context allocation mechanism to determine if a `context` is freely available or not. The pointers `prev_scheduled` and `next_scheduled` are used to form a circular doubly-linked-list that is traversed as a part of the round-robin scheduler.
Since the operating system needs to be able to kill a process after it for example exited or because it caused a segmentation fault, a killing mechanism is implemented. When a process is killed, its entire page table, the pages it used, and the `context_manager` structure that contains its `context` are freed, so that the consumed resources can be reused later on. If all processes have been killed, the kernel has nothing to manage anymore and will start to panic.

4.2 Memory Virtualization

RV64 uses the `satp` CSR to control memory virtualization (see Figure 4.3). As [17] writes, the `MODE` field identifies the address-translation scheme used for memory virtualization, where 0 stands for Bare and 8 for Sv39. Furthermore, the `ASID` field makes it possible to specify an address-space identifier that can be used for more

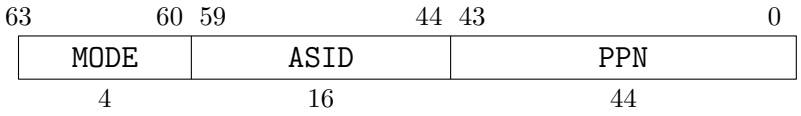

Figure 4.3: Structure of the `satp` CSR (recreated from [17])

efficient TLB usage. For the sake of simplicity, this feature remains unused in our implementation. Therefore, we always write 0 into it. The last field, `PPN`, contains the PPN of the root page-table used to perform memory translation [17].
When we talk about an *`satp` value*, we refer to a 64-bit value that has the form of the `satp` CSR and contains the identifier for Sv39 in its `MODE` field, 0 in its `ASID` field, and the PPN of a root page-table in its `PPN` field.

4.2.1 Page Allocation

Since processes are able to allocate memory dynamically, the operating system's job is it to provide a supply of usable memory. This means that for a virtual-memory system, it has to provide ways of mapping new pages. Conveniently, both page tables and pages are of sizes 4 KiB each, which allows us to implement both page-table and page allocation within one simple page allocator. It uses a pre-allocated pool of available pages in combination with a bump pointer. The issue with this implementation is that the page allocator can run out of pages. This is especially then a problem, if for example a process has been killed, and therefore all of its consumed pages are not used anymore. For this reason, we use a free list in conjunction with the bump allocator. Since it is possible to modify the contents of a freed page without affecting another context, the first doubleword within a free page is used to save the PPN of the next free page in the list (see Figure 4.4). Our page-allocation algorithm is described in Algorithm 4.1. Here, 0 is reserved as an invalid return value since the free page pool will never start at address 0.

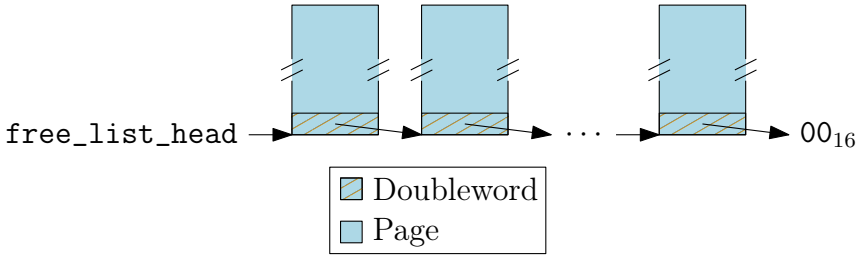

Figure 4.4: Free list used within the page allocator

If a virtual-address space needs to be freed, it is necessary to free the mapped pages before the page tables that store the references to them, or else the references to the pages would be lost before they could be freed. The same applies for page tables that are referenced by other page tables. For this reason, we implement a simple depth-first algorithm that frees all subtrees first (see Algorithm 4.2).

Algorithm 4.1: Page allocator

```
   Data: PPN bump-pointer
   Output: PPN of available page frame
 1 if page frame in free list then
 2 |   dequeue page frame;
 3 |   return PPN of page frame;
 4 end
 5 if page frame left in pool then
 6 |   increase PPN bump-pointer;
 7 |   return PPN of page frame;
 8 end
 9 return 0;
```

Algorithm 4.2: Freeing virtual memory

```
   Input: root page-table
 1 foreach mid-level page-table in root page-table do
 2 |   foreach leaf page-table in mid-level page-table do
 3 |   |   foreach page in leaf page-table do
 4 |   |   |   free page;
 5 |   |   end
 6 |   |   free leaf page-table;
 7 |   end
 8 |   free mid-level page-table;
 9 end
10 free root page-table;
```

4.2.2 Page Mapping

Mapping new pages provided by the page allocator into a virtual-address space proves to be an undertaking where many small details have to considered in order to provide correct data to the memory management unit. Here, the two main tasks within our implementation are (1) to identify when one or more new page tables need to be mapped into the tree and (2) to mark both PTEs that point to page tables and PTEs that point to pages correctly. Since PTEs employ the V bit to indicate whether a PTE is valid or not [17], we can use this flag to decide whether a new page table needs to be mapped or not: If an entry in the root page-table is invalid, we need to map a mid-level page-table, and if an entry in a mid-level page-table is not valid, a leaf page-table needs to be mapped. Since the V bit must be cleared for invalid PTEs, we allocate zeroed pages for page tables in those cases. Those PTEs that point to other page tables must have their X, W, and R bits set to 0, since this indicates that a PTE contains information about a page table [17]. If on the other hand we create a PTE that points to a page, we always mark it as executable, writable, and readable. While this may be an insecure approach, it reduces complexity. Additionally, ELF files emitted by selfie do not include any section information but describe the whole binary in one single program header with its executable, writable and readable flags set [13], which would make support for pages with different flags redundant anyway.
Since we do not implement swapping or similar mechanisms, we only follow the rules and recommendations specified in [17]: We set the A and D bits within PTEs that refer to page tables to 0 due to mandated forward compatibility an in the case of PTEs that refer to pages on the other hand, we set them to 1 in order to improve performance.
As we do not use the functionality of global mappings, we always set the G bit to 0. Upon creating a new PTE, it is marked as valid by setting its V bit. Additionally, we also set the U bit for all PTEs that refer to pages that are supposed to be accessible from U-mode.
When selfie compiles a C* program, the library function `zalloc` is embedded into the resulting binary. Since here it is assumed that allocated memory is always zeroed, `zalloc` falls back to `malloc` without explicitly zeroing the memory [13]. For this reason, we also always zero newly mapped pages.

4.2.3 Kernel and User Memory Virtualization

When working with virtual memory, it has to be constructed with a certain structure in mind. This is necessary in order to enable the kernel to bootstrap, to allow for context switches, and to handle system calls in an efficient manner. The kernel's virtual memory contains an identity mapping for all the page frames that it uses. This is done to aid in bootstrapping and to handle system calls where user memory needs to be accessed in an easy fashion. Additionally, in order to for our context switching functionality to work, the trampoline and the kernel's stack are mapped again into the highest pages of the kernel's higher half. This results in the memory map presented in Figure 4.5. In order to strip the kernel of the complexity of determining if the trampoline spans over page boundaries, we align it to a 4 KiB boundary using the assembler directive `p2align` defined in [3]. Due to its nature of

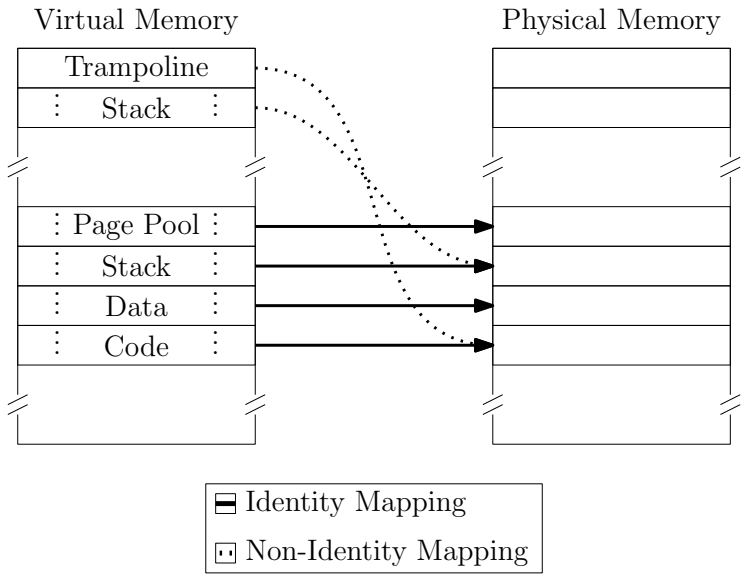

Figure 4.5: Kernel memory map

consisting of less than 100 instructions, the trampoline fits into a page quite easily, which means that it is very simple to map it into a virtual-address space.
For user processes, we follow the approach of mapping the user space into the lower half and the kernel space into the higher half of its virtual memory. In order to provide the expected memory model for programs compiled with selfie as defined in [12], we replicate it in the lower half (see Figure 4.6). The only parts of the kernel that need to be mapped into other virtual-address spaces are the trampoline and its stack. For our context switching algorithm to work, they follow the same rules as for the kernel's virtual-address space of being mapped into the highest pages of the higher half. Additionally, the kernel space mappings have the U bit within their corresponding PTEs set to 0 in order to provide spatial isolation between a user process and the kernel.

4.3 ELF Loading

Binary files compiled by selfie include a minimal ELF header containing the necessary information in order to load and run those executables [13]. In fact, as our own experiments utilizing a SiFive HiFive Unleashed have shown, those ELF files are compatible with Linux running on native RISC-V hardware. In order to enable our kernel to also have this fundamental functionality of loading and executing selfie-compiled programs natively, we implement an incomplete ELF loader tailored to our needs.
By looking at the values that selfie puts into the ELF-header fields (see Table 4.1) and into its only program header (see Table 4.2), we can determine what functionality our ELF loader has to support. Apart from performing sanity checks like for example the verification of ELF's magic number or the machine architecture, the ELF loader needs to be able to (1) set the entry point to `e_entry`, (2) to read the program header from offset `e_phoff`, and (3) to read `p_filesz` bytes from offset `p_offset` within the file into the memory starting at virtual address `p_vaddr` of

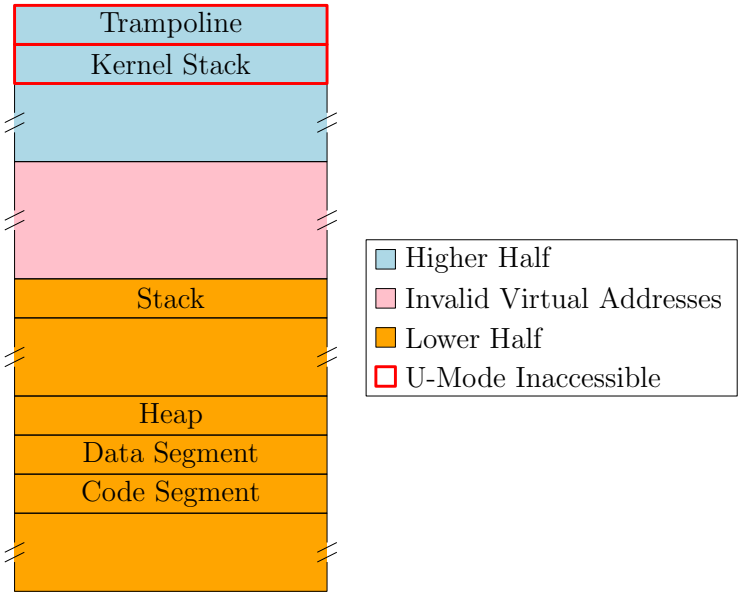

Figure 4.6: Mappings within a user process

Header entry		Value
e_ident	EI_MAGx	0x7F, 'E', 'L', 'F'
	EI_CLASS	ELFCLASS64 (2)
	EI_DATA	ELFDATA2LSB (1)
	EI_VERSION	EV_CURRENT (1)
	EI_OSABI	ELFOSABI_SYSV (0)
	EI_ABIVERSION	0
	EI_PAD	0
e_type		ET_EXEC (2)
e_machine		EM_RISCV (243)
e_version		EV_CURRENT (1)
e_entry		0x10000
e_phoff		0x40
e_shoff		0x00
e_flags		0
e_ehsize		64
e_phentsize		56
e_phnum		1
e_shentsize		0
e_shnum		0
e_shstrndx		0

Table 4.1: ELF header generated by selfie [13, 7]

Header entry	Value
p_type	PT_LOAD (1)
p_flags	PF_R | PF_W | PF_X (7)
p_offset	0x1000
p_vaddr	0x10000
p_paddr	0x00
p_filesz	Length of generated binary
p_memsz	Length of generated binary
p_align	4096

Table 4.2: Program header generated by selfie [13, 7]

the corresponding process. A solution for (1) is provided by setting the process's program counter to the virtual address specified in `e_entry`. Since for fulfilling the requirements (2) and (3) we do not want to rely on our knowledge of the hardcoded entries that selfie puts into the ELF header and its only program header entirely, but want to resemble an incomplete version of an ELF loader instead, we iterate over all `e_phnum` entries in the program-header table and respect possible differences between `p_filesz` and `p_memsz`. Therefore, for each program header, we calculate and map the number of pages needed to read the segment of size `p_filesz` into the process's virtual memory and copy the segment's content from offset `p_offset` within the file into those pages. Additionally, we determine the delta δ between that number of pages and the amount of pages we need to provide `p_memsz` bytes of virtual memory for the entire segment and map the δ pages after the space where the segment's file content resides. Finally, in order to save correct context information, we set the boundaries for the context's lo region to the lowest and the highest page that have been mapped when loading all segments.

4.4 Context Switching

The machine is in a constant cycle of executing code in user space, receiving a trap, handling this trap within the kernel, and then switching back into user space (see Figure 4.7). Here, there exist different contexts that must never be mixed up with each other. For this reason, a context switch is performed before entering and after exiting the trap handler.
RISC-V only allows one page table to be set. This means that regardless of being in U-mode or S-mode, the page table referenced in the `satp` CSR is the active one. Also, there is no mechanism provided to switch the active page table and to change the program counter atomically. This implies that code for switching between contexts needs to be mapped into a user process's virtual memory, and also has to be mapped at the same location into the kernel's virtual-address space. We choose to map two kernel components into every user page-table: the kernel's stack which contains information necessary for context switching, and the trampoline, the actual code that performs context switches. The program counter's critical movements during the context-switching cycle are as presented in Figure 4.8: (1) a trap occurs which causes the machine to switch from U-mode to S-mode and to jump into the trampoline, (2) the active page table is switched from the user page table

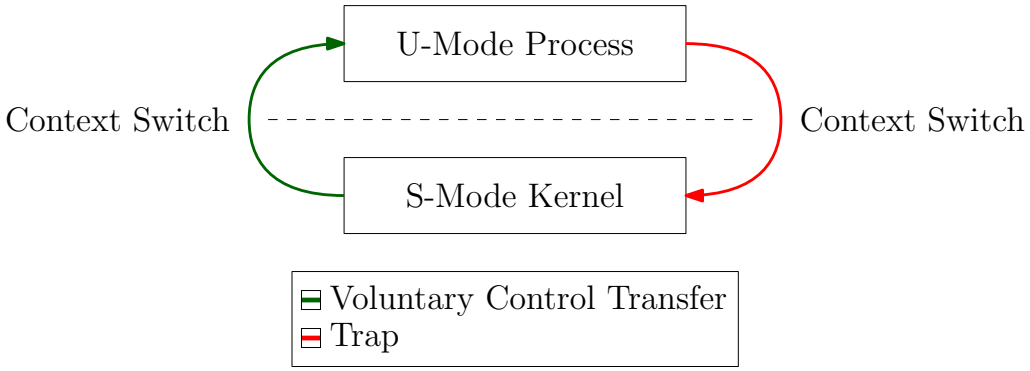

Figure 4.7: Context-switching cycle

to the kernel page table, (3) the kernel starts executing the trap handler, (4) the trap has been handled and the kernel jumps back to the context-switching code in the trampoline, (5) the active page table is switched from the kernel page table to the user page table, (6) the kernel puts the machine back into U-mode.

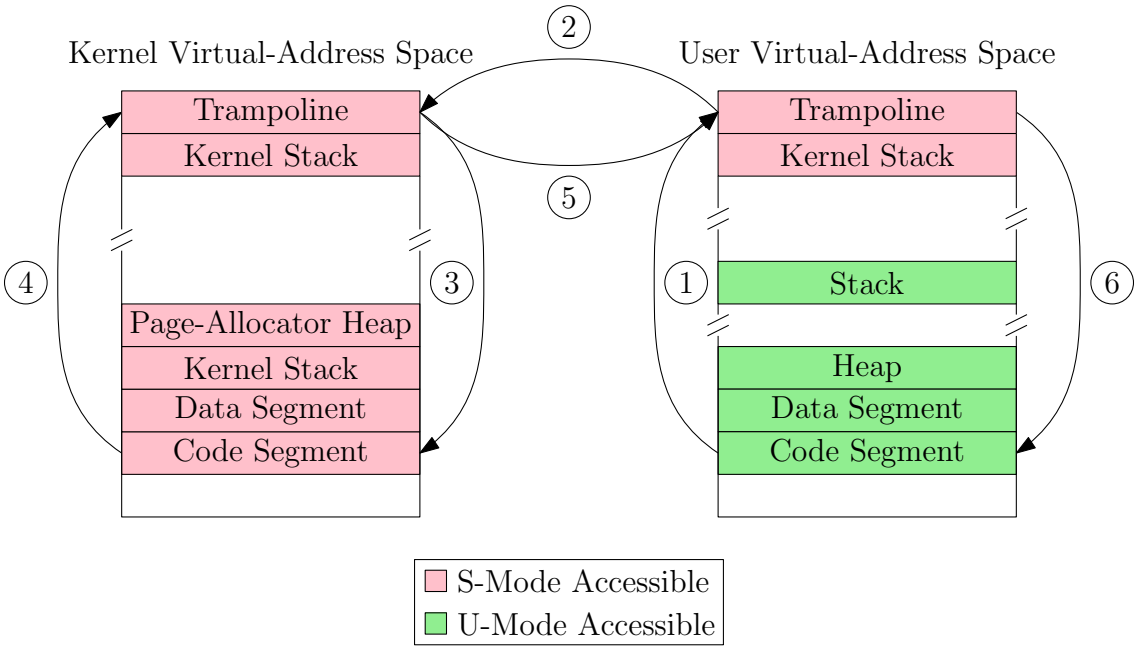

Figure 4.8: Program counter movement upon trap handling

When writing the context switching code in C, important state information would be lost due to the compiler generating code which would overwrite the contents of another context's registers. To be able to save all this relevant state information, the entire context switching code is implemented in handwritten assembly code.

4.4.1 Trampoline Entry

The `sscratch` CSR can be used freely by a kernel but normally it holds a pointer to the kernel context. Furthermore, it is a register whose content can be swapped with another register's content upon entering a trap handler, in order to have an

initial general purpose register that can be worked with [17]. When the trampoline is entered, it expects that `sscratch` contains the kernel's stack pointer. Additionally, the kernel stack is expected to provide a 8 B buffer to save a register, the `satp` value that corresponds to the kernel's page table, the kernel's global pointer, and the address of the trap handler per se (see Figure 4.9).

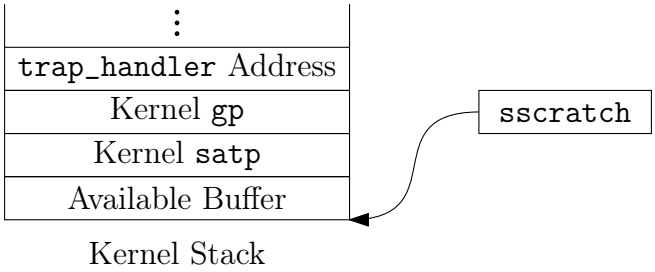

Figure 4.9: Expected `sscratch` and kernel stack setup when entering the trampoline

Now it is possible to change to the kernel's virtual-address space. To do so, we first have to swap the contents of `sscratch` and `sp` atomically, which allows us to access the kernel stack. Since data cannot be read directly from memory into a CSR, we first save the contents of `s0` into the dedicated buffer on the stack. This way, we have a free register that we can load the kernel's `satp` value into. Afterwards, the `satp` CSR is written and the TLB is flushed by issuing an `sfence.vma` instruction [17] as presented in Listing 4.4.

```
1 csrrw sp, sscratch, sp // swap sp and sscratch
2 sd s0, 0(sp)           // save s0 onto stack
3 ld s0, 8(sp)           // load kernel satp value
4 csrw satp, s0          // switch to kernel page-table
5 sfence.vma zero, zero  // flush TLB
```

Listing 4.4: Switching to the kernel's virtual-address space at the beginning of the trampoline

Since we always write 0 into the `ASID` field in the `satp` CSR and do not use global mappings within page tables, we follow the recommendation of [17] and execute `sfence.vma` with $rs1 =$ `zero` and $rs2 =$ `zero`.
Now, a new frame is set up and space for a `registers` structure is allocated on the stack. In order to save every register except for the user context's `sp`, `s0`, and `pc`, they are written directly into the structure. Before they can be saved, `sp` and `s0` have to be read from `sscratch` and the buffer on the kernel's stack respectively. When the trap happened, the user process's `pc` was saved into the `sepc` CSR. We read it from there and save it into the `registers` structure. Since all necessary context information has been saved now, the kernel's global pointer can be set by loading its value from the stack and by writing it into the `gp` register. Due to the `registers` structure being a parameter for the actual trap handler, it must be passed following the way defined in [5], which specifies that in our case its address has to be passed in `a0`. The current machine configuration is as presented in Figure 4.10.

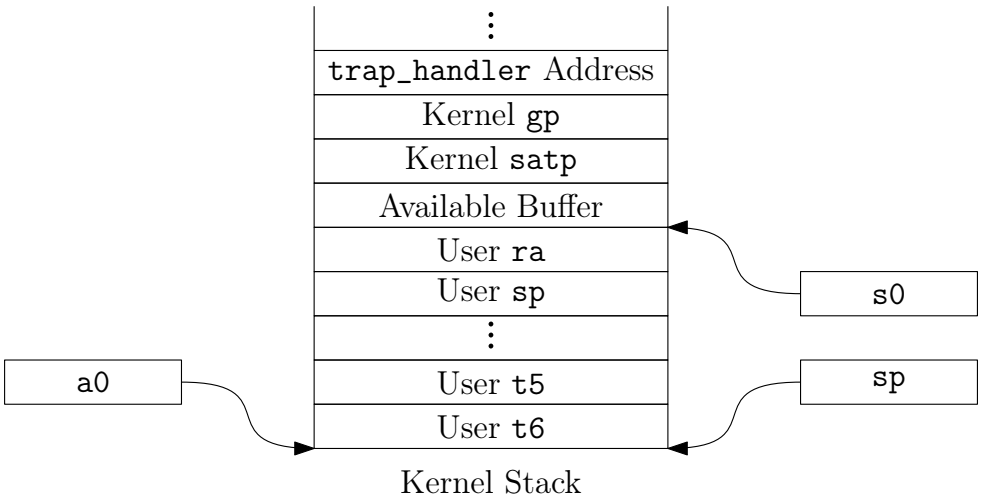

Figure 4.10: Machine configuration before entering the trap handler

Normally, when programming in RISC-V assembler language, function calls are performed through the use of the `call` pseudo instruction [3], which uses PC-relative addressing [16]. This poses a problem insofar that (1) the compiler does not know that we map the trampoline code into the highest part of the virtual-address space and execute it from there, and (2) that the 32-bit offset used in the corresponding instruction sequence would not be enough to cover the distance between the trampoline and the trap handler. For this reason, the trap handler's absolute address is saved on the kernel's stack. This way we can load it from there and perform a `jalr` instruction to jump into the trap handler.

4.4.2 Trampoline Exit

After the trap has been handled, the `registers` structure on the stack contains the register values of the context the machine is supposed to switch to. Additionally, the trap handler returned the `satp` value of this context in `a0` [5] (see Figure 4.11).

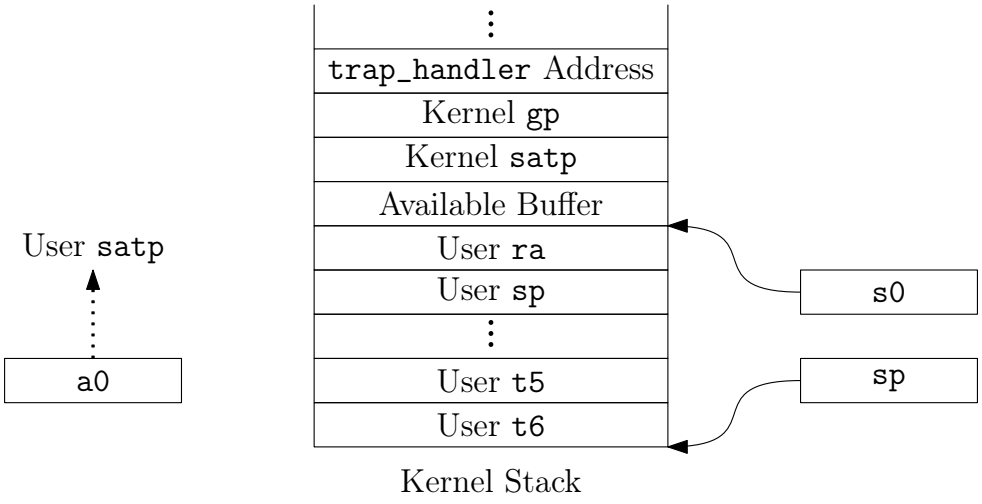

Figure 4.11: Machine configuration after exiting the trap handler

Now, the active page table is switched to the user page table. The value of `s0` is loaded into `sscratch` in order to provide the conditions that the trampoline expects to encounter when a trap is received. Afterwards the contents of the `registers` structure are loaded from the stack into each register and the user context's program counter is loaded into `sepc`. In order to continue execution in the user process, an `sret` instruction is performed, which in this case causes the machine to set the privilege level to U-mode and to set the `pc` to the value in `sepc` [17]. This closes the cycle shown in Figure 4.7.

4.5 Trap Handling

After receiving a trap and switching to the kernel's context, the actual trap needs to be taken care of. This is the main task of the trap handler. There exist many different reasons for why a trap can happen and in order to handle certain traps, additional information is needed. For this reason, [17] defines multiple CSRs that are important for us in this regard:

- `sstatus` is an S-mode specific subset of the `mstatus` CSR and contains information about the current processor state.
- `scause` contains a code to identify the reason for a trap after a trap into S-mode has been taken. Its structure consists of the most significant bit that flags whether the trap is an interrupt or not, and the lowest 63 bits that contain an exception code.
- `sepc` saves the virtual address of the instruction that was interrupted/is associated with the exception when a trap into S-mode is taken.
- `stval` is used to save additional information for some traps that are taken into S-mode. If a trap does not provide this information, this CSR is set to zero.

Since our kernel is designed to never cause any traps that would cause it to end up in its own trap handler, it would be a malfunction if this was the case. For this reason, we rely on the SPP bit within `sstatus`. It indicates the privilege level prior to the trap, where it is set to 0 if the trap was caused in U-mode and 1 otherwise [17]. Since traps never cause a control transfer from a higher privilege level to a lower privilege level [17], we can guarantee that a set SPP bit means that the trap was caused by our kernel running in S-mode. To signal such an error, the kernel fails gracefully by panicking and providing debugging information.
Due to the fact that the trampoline uses a `registers` structure on the stack as a buffer to save the user process's registers and because it passes this structure as a parameter to the trap handler, we must copy all of the structure's content into the appropriate `context` which is retrieved by querying the scheduler.
Now, the actual trap handling can begin after the trap's type has been determined from the information provided by `scause`. The traps we will cover are presented in Table 4.3 and will be discussed soon.

After the trap has been handled, the machine has to be prepared for the next context switch that is performed as a part of the trampoline. To do so, the next

Trap type	Interrupt bit	Exception code
Supervisor timer interrupt	1	5
Environment call from U-mode	0	8
Instruction page fault	0	12
Load page fault	0	13
Store/AMO page fault	0	15

Table 4.3: Traps handled within our implementation (adapted from Table 4.2 in [17])

process is scheduled and its register's contents are loaded into the buffer that the trap handler received as a parameter.
The next step is to set a new timer interrupt. Timer interrupts are triggered if the memory-mapped `mtime` register, a real-time counter, contains a value greater than or equal to the value in `mtimecmp`, which is also a memory-mapped register [17]. This means that a future timer interrupt is set up by adding an appropriate time slice to the current value in `mtime` and by loading this value into `mtimecmp`. The problem herein lies in the fact that these memory mappings are platform specific and would therefore require knowledge of the current device's memory map on our side. Fortunately, `mtime` is exposed through the `time` CSR and can be read with the help of the `rdtime` instruction [17, 16]. For `mtimecmp` this is not the case. This functionality is accessible through the SBI though [4]. Therefore, we can easily set a timer interrupt by reading the current time with `rdtime`, adding the desired time slice to it, and by calling into OpenSBI. If OpenSBI returns an error value for this call, the kernel starts to panic. Last but not least, as the trampoline expects to receive the `satp` value of the process that is scheduled to be run next, the trap handler returns this value.

4.5.1 System Calls

When it is detected that a trap was caused by an `ecall` instruction executed by a user process, the kernel needs to be able to identify which system call was performed and has to retrieve the correct parameters. Selfie uses Linux's system call ABI [13, 10]. Here, the system call ID is loaded into `a7`, while `a0` to `a5` hold the parameters. The register usage for passing the parameters of all five system calls is shown in Table 4.4. These five system calls are handled in the following ways:

System call	a0	a1	a2	a3
`exit`	Exit code	*N/A*	*N/A*	*N/A*
`read`	File descriptor	Pointer to buffer	Size	*N/A*
`write`	File descriptor	Pointer to buffer	Size	*N/A*
`openat`	Directory file descriptor	Filename	Flags	Mode
`brk`	New program break	*N/A*	*N/A*	*N/A*

Table 4.4: Register usage for system call parameters [13]

If a program makes an `exit` call, the kernel retrieves its exit code, prints information about the process exiting to the console, and kills it afterwards.

The difficulty in handling calls to `read` and `write` lies in the existence of the passed pointer to a buffer. This stems from the fact that (1) user input can never be trusted and (2) that the pointer is an address within the user process's virtual-address space. In order to understand the problem that comes with (2), one needs to think about two properties of virtual memory: a virtual address may not correspond to the same physical address and memory that is contiguous within virtual memory may not be contiguous in the physical world (see Figure 4.12). In order to read from/write to the

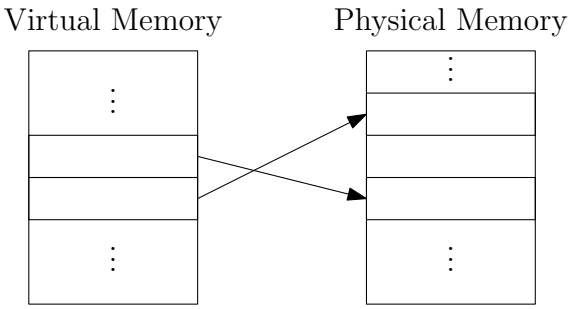

Figure 4.12: Address translations and non-contiguity of physical memory

buffer within the user process's virtual memory, we slightly adapt the algorithm used within selfie where the memory is read from/written to in doubleword granularity and where each resulting address is checked for validity [13]. The algorithm used to implement a `read` system call is described in Algorithm 4.3. The overall algorithm used for `write` calls is the same but with the obvious changes to be made. While this may not be a very efficient algorithm, it may at least be considered as effective.

When handling `openat` system calls, both the directory file descriptor in `a0` and the mode in `a3` are ignored due to our file system not implementing the concept of directories and advanced modes not being needed. With the pointer to a string containing the filename, we have a similar problem as with `read` and `write`: it is a pointer to memory within another virtual-address space. To copy the string into a buffer on the kernel's stack in its own virtual address space, we again perform sanity checks and copy parts of the string's content at every doubleword boundary. Since there would be no value added by providing the exact algorithm, it is omitted in this work. If the retrieval of the filename was successful, the internal kernel function for opening a file is called and the corresponding file descriptor is loaded into the user process's `a0` register. If on the other hand the string containing the filename could not be copied, -1 is loaded into the user process's `a0` register.

Since we want the behavior of our `brk` implementation to be same as in selfie, we again reuse its algorithm for the implementation of this system call [13]: If the new program break is (1) not less than the current program break, (2) below the user process's stack pointer, and (3) aligned to a doubleword boundary, it is set as the user process's program break and is loaded into its `a0` register. If on the other hand at least one of the aforementioned requirements is not fulfilled, the previous

Algorithm 4.3: Implementation of a `read` system call (adapted from [13])

```
1  fd = content of user process's a0;
2  vbuffer = content of user process's a1;
3  size = content of user process's a2;
4  read_total = 0;
5  to_read = 8;
6  failed = false;
7  buffer;
8  actually_read;
9  while size > 0 do
10 |  if size < to_read then
11 |  |  to_read = size;
12 |  end
13 |  if vbuffer is mapped and in lower half then
14 |  |  buffer = translation of vbuffer to physical address;
   |  |  /* kread tries to read to_read bytes from the file with
   |  |     the file descriptor fd into buffer and returns the
   |  |     actual number of bytes read                          */
15 |  |  actually_read = kread(fd, buffer, to_read);
16 |  |  if actually_read == to_read then
17 |  |  |  read_total = read_total + actually_read;
18 |  |  |  size = size − actually_read;
19 |  |  |  if size > 0 then
20 |  |  |  |  vbuffer = vbuffer + 8;
21 |  |  |  end
22 |  |  else
23 |  |  |  if 0 < actually_read then
24 |  |  |  |  read_total = read_total + actually_read;
25 |  |  |  end
26 |  |  |  size = 0;
27 |  |  end
28 |  else
29 |  |  failed = true;
30 |  |  break;
31 |  end
32 end
33 if failed then
34 |  load −1 into user process's a0;
35 else
36 |  load read_total into user process's a0;
37 end
```

program break is returned in the user process's `a0` register.

4.5.2 Page Faults

RISC-V uses page faults to for example indicate that an unmapped page has been accessed or to signal certain illegal memory accesses like trying to read from a U-mode inaccessible page while running in U-mode [17]. Here, it differentiates between three different categories of page faults: instruction page faults, load page faults, and store/AMO page faults that are caused by instruction fetches, load operations, and store operations or atomic memory operations (AMO) respectively [16, 17]. We on the other hand, do not need to differentiate between all three of them, but can put them into two groups instead, where the first one consists of instruction page faults and the seconds one of load page faults and store/AMO page faults.
As already mentioned before, we map the entire code of an ELF file when loading it. Therefore, it should not be possible for a program compiled by selfie to cause an instruction page fault. This stems from the fact that C* does not implement for example jumps to arbitrary positions or function pointers [13]. Since C* is an unsafe language though, it would be possible for a program to manipulate a return address that is saved on the stack [12], and therefore to cause a jump to any address within memory. Now if an instruction page fault were to occur, this would mean that the user process tried to execute code from an unmapped page or from a page within its hi region. In both of these cases it is justified to terminate the process, which is the exact thing we do.
Due to our kernel implementing a lazy mapping scheme for dynamically allocated memory and stack memory, the load and store/AMO page faults on the other hand need to be handled with greater caution. Therefore, we need to determine whether the memory access was actually legal or not. To do so, we differentiate between the six categories of possible load and store/AMO page faults presented in Figure 4.13: (1) legal accesses to lo memory, (2) legal accesses to mid memory, (3) illegal accesses to hi memory, (4) legal heap growth bounded by the `highest_lo_page` and the program break, (5) legal stack growth bounded by the `lowest_mid_page` and the stack pointer, and (6) illegal memory accesses or accesses to invalid virtual addresses. How each of these categories of page faults are handled, is shown in Figure 4.14.

4.5.3 Miscellaneous Traps

From the traps that are neither environment calls nor page faults, we only expect to receive supervisor timer interrupts. If such a timer interrupt is detected, it is simply seen as a sign to schedule the next process and is in no need of any additional treatment. All the other traps that are delegated from M-mode to S-mode, and therefore to our kernel, are never expected to occur. For this reason, if one of those traps is received, the process that caused them is killed.

4.6 Bootstrapping

Early bootstrapping is initiated by setting up the page allocator's PPN bump-pointer with the physical page number of the first page frame after the stack.

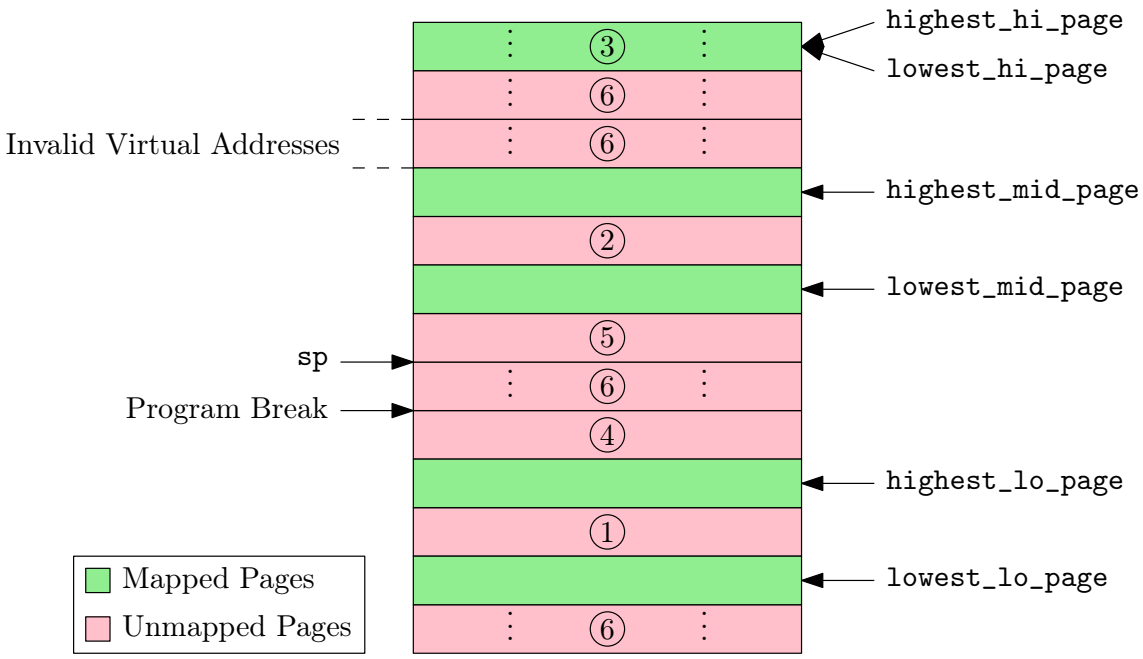

Figure 4.13: Categories of load page faults and store/AMO page faults for different memory accesses

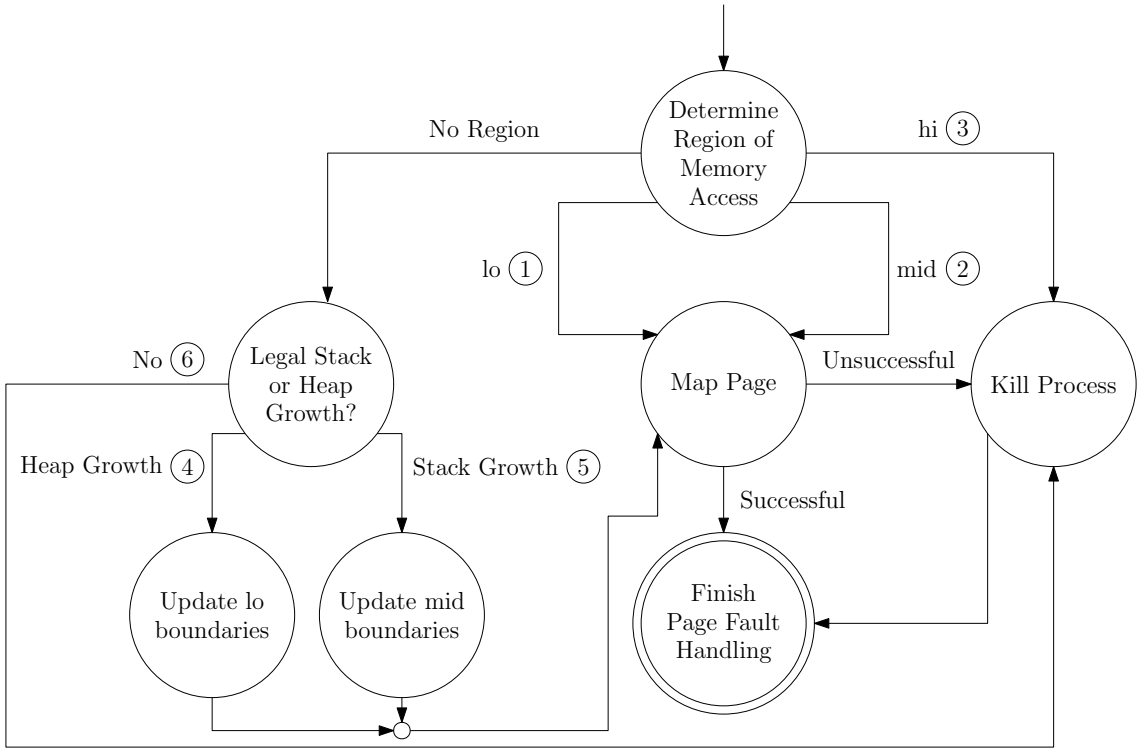

Figure 4.14: Load and store/AMO page fault handling algorithm

Following that, the kernel environment is set up in several steps: The kernel context is initialized with for the kernel relevant information consisting of its context ID, the pointer to its page table, its program break and its memory boundaries. Before enabling paging for the kernel, its page table must be set up correctly. This is done by identity-mapping the kernel and its stack which lies directly after it in physical memory. Since the trampoline and the kernel's stack also need to reside in the highest pages of every virtual-address space's higher half, it is necessary to map the page frames containing them to the appropriate pages. Last but not least, the free page pool that is used within the page allocator has to be mapped. The additional pages needed for the page tables that are used to map all of the aforementioned content are also mapped into the kernel's virtual memory.
At the moment, the machine does not know the location where it needs to jump to if a trap occurs. This information is provided through the `stvec` register [17]. Here it is possible to use a vectored mode for providing different trap handlers that are each executed depending on the content of `scause` in order to improve performance. Since we do not use this advanced feature, we just load the trampoline's address in the higher half masked with the Direct mode for trap handling into `stvec`.
Currently, our kernel is cooperative instead of preemptive. One of the steps towards the latter property is to enable supervisor timer interrupts by setting the appropriate bit in the `sie` CSR [17].
Now, paging is activated by loading the kernel's `satp` value into the `satp` CSR [17]. Since the kernel's stack is only mapped into the higher half of every user process, it is also necessary to change the stack pointer's value so that it points to the duplicate mapping within the higher half (see Figure 4.15).

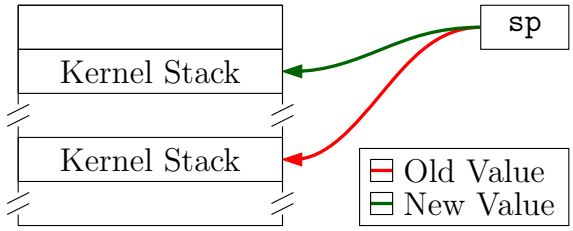

Figure 4.15: Migration of the kernel's stack

A kernel is of no use if there are no processes that it manages. Therefore, we spawn an init process after the kernel has set itself up. The first step in this procedure is to search for the init's ELF file corresponding to its hardcoded filename. If the file could not be found, the kernel can not proceed any further and will start to panic.
Now, a `context` will be allocated and initialized. This means that all registers are set to zero, except for the `sp` register, which is initialized with the start address for user stacks. Since there is nothing pushed onto the stack yet, this corresponds to the first invalid Sv39 address (i.e. the first address after the end of the lower half). One stack page is mapped and the mid region's boundaries are set. As it is required for the trampoline to work, its code and the kernel stack are mapped into the top of the higher half and the boundaries for the hi region are set up. Nothing is loaded into the lo region when initializing a `context`. For this reason, its boundaries and

the program break are all set to zero.
A state container in the form of a `context` is now available. At the moment, there exists no code within its virtual memory, that would allow us to start execution. To solve this problem, the retrieved ELF file is passed to the ELF loader. Again, if there occurred an error while loading the ELF file, a kernel panic is caused. The next step is to pass the program arguments by loading the necessary information onto the init's stack. We set the stack up exactly like selfie does (see Figure 4.16) [13]. For the environment list, only a null-terminator is pushed onto the stack. Since selfie uses the stack to pass function parameters [13], this setup fulfills the requirements of [8]. Here, it is stated that if arguments are passed to the `main` function, the first argument `argc` is the number of arguments and the second argument `argv` an array of pointers to strings, `argv[0]` to `argv[argc-1]` are pointers to the string arguments, and `argv[argc]` contains a null-pointer. Since for copying the strings we

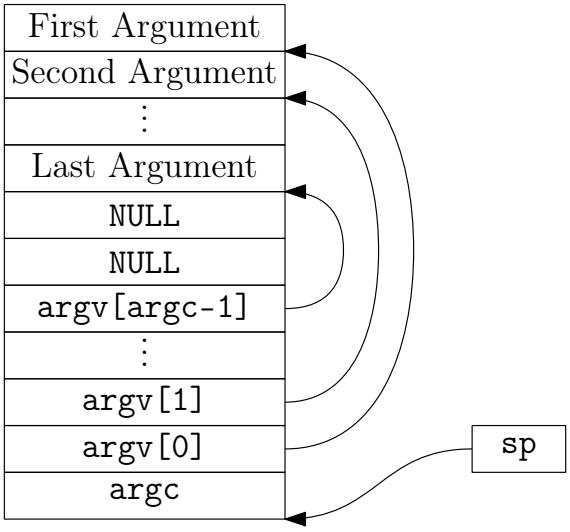

Figure 4.16: Stack argument setup

again have the problem of different virtual-address spaces, we adapt the algorithm used in selfie. This means that we copy the argument strings into the init's virtual memory in the granularity of doublewords. If more pages are needed in order to set up its stack, they are mapped into its virtual-address space. Due to the possibility of the page allocator running out of available pages, the process of copying the arguments can fail. If this happens, the kernel panics.
As one of the final preparations before the context switch to the init process is initiated, a timer interrupt is set through an SBI call and if this call fails, the kernel will panic. If now control would be transferred to the init process, the trampoline code would fail upon its first execution. This stems from the fact that the machine is not set up in the exact way the trampoline expects it to be. For this reason, we manipulate it directly through assembly code. This assembly routine receives the init's `satp` value and a pointer to its `registers` structure as parameters. With this information available, the machine is set up as presented in Figure 4.11. Since now the machine is set up in the way the trampoline's exit code expects it to be, it is possible to jump to this location. The trampoline's exit code starts with the switch to the scheduled user process's virtual-address space. If the kernel would

start executing this code in its lower half, unknown things would happen since then the machine would start trying to execute code from the same location within the user process's virtual-address space. For this reason, it is necessary to execute the trampoline's exit code from the higher half. This is done by calculating its address and by performing a jump to this address. Now, the initial context switch will be finalized with the help of the trampoline's exit code and the cycle described in Figure 4.7 will begin.

5 Conclusion and Outlook

Now is the time for a quick overview. We followed the recommended RISC-V software stack for the creation of an operating system on a machine supporting all three privilege levels. Hereby, we utilized the official SBI that is implemented by OpenSBI running in M-mode. Our kernel is designed to run in S-mode and to provide preemptive multitasking functionalities by controlling processes running in U-mode. It can execute ELF binaries of C* programs that have been compiled by selfie and provides them with runtime support by implementing selfie's ABI consisting of the `exit`, `read`, `write`, `openat`, and `brk` system calls. We showed the difficulties that come from having paging enabled for user processes, as well as for the kernel and designed memory layouts that allow us to bootstrap, context switch, and to handle traps. Starting only with a working C environment, we set up our kernel step by step and spawned a process running selfie that successfully executed until its `exit` call.
Since our kernel follows the official RISC-V specifications and has been compiled with those targets in mind, it is able to run on QEMU[11] version 5.0 and actual hardware in the form of a SiFive HiFive Unleashed. Even though the latter platform only implements version 2.1 of the unprivileged and version 1.10 of the privileged specification [14], we only used features that did not experience any breaking changes between those versions.

Sadly, after spending a serious amount of time working on this project, we had to put it to a stop at some point. Due to this, two topics that we wanted to look into were left behind. Firstly, the SiFive HiFive Unleashed is a very expensive development board. A cheaper RV64 alternative with support for the required privilege levels and Sv39 exists in the form of platforms utilizing the Kendryte K210[12] chip. It implements an older version of the privileged specification that is incompatible with the one used by us. We hope that in the future, support for this chip can be implemented in order to reach a wider audience.
Secondly, the implementation of a hypervisor or the port of hypster to native hardware would be an interesting undertaking. Since this is a very complicated topic, it would be exciting to know if and how this could be done, maybe even with support for RISC-V's Hypervisor Extension, which is still under development[13] and is currently available in version 0.6.1.

[11] `https://www.qemu.org/`
[12] `https://canaan.io/product/kendryteai`
[13] `https://github.com/riscv/riscv-isa-manual/releases`

References

[1] Christian Barthel. "Porting Selfie to RISC-V: Native Toolchain Support". Bachelor Thesis. June 2017. URL: https://github.com/cksystemsteaching/selfie/blob/master/theses/bachelor_thesis_barthel.pdf.

[2] Western Digital Corporation et al. *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. Repository. URL: https://github.com/riscv/opensbi/.

[3] Palmer Dabbelt, Michael Clark, and Alex Bradbury. *RISC-V Assembly programmer's Manual*. Specification. Accessed on 2020-09-11. 2020. URL: https://github.com/riscv/riscv-asm-manual.

[4] Palmer Dabbelt and Atish Patra. *RISC-V Supervisor Binary Interface Specification*. Specification. Version 0.2. 2020. URL: https://github.com/riscv/riscv-sbi-doc.

[5] Palmer Dabbelt et al. *RISC-V ELF psABI Specification*. Specification. Accessed on 2020-09-08. 2020. URL: https://github.com/riscv/riscv-elf-psabi-doc.

[6] Free Software Foundation, Inc. *Using the GNU Compiler Collection (GCC)*. Documentation. Version 10.1.0. 2020. URL: https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc/.

[7] *GNU C Library: elf.h*. Software Library. Version 2.32. 2020. URL: https://sourceware.org/git/?p=glibc.git;a=blob;f=elf/elf.h;hb=3de512be7ea6053255afed6154db9ee31d4e557a.

[8] *Information technology – Programming languages – C*. en. Standard ISO/IEC 9899:2018. Geneva, CH: International Organization for Standardization, June 2018. URL: https://www.iso.org/standard/74528.html.

[9] Michael Kerrisk, ed. *Linux Programmer's Manual: elf – format of Executable and Linking Format (ELF) files*. Apr. 2020. URL: https://www.kernel.org/doc/man-pages/.

[10] Michael Kerrisk, ed. *Linux Programmer's Manual: syscall – indirect system call*. Apr. 2020. URL: https://www.kernel.org/doc/man-pages/.

[11] Christoph Kirsch. "Selfie and the Basics". In: *Proc. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* ACM. Oct. 2017, pp. 198–213. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133857.

[12] Christoph Kirsch and Sara Seidl. *Selfie: Introduction to the Implementation of Programming Languages, Operating Systems, and Processor Architecture*. Slides. Accessed on 2020-09-11. 2020. URL: http://selfie.cs.uni-salzburg.at/slides/.

[13] Christoph Kirsch et al. *Selfie*. Repository. Accessed on 2020-09-22. 2020. URL: https://github.com/cksystemsteaching/selfie.

[14] SiFive, Inc. *SiFive FU540-C000 Manual*. Version v1p0. 2018. URL: https://sifive.cdn.prismic.io/sifive%2F834354f0-08e6-423c-bf1f-0cb58ef14061_fu540-c000-v1.0.pdf.

[15] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th. Wiley, 2012.

[16] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Specification. Version 20191213. RISC-V Foundation. Dec. 2019.

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