UNIVERSITY OF BAMBERG

Alternative Approaches for Virtual Memory Management

BY

Max Meidinger

Chair of Practical Computer Science, esp. Systems Programming

September 2024

v20200913

Links to this document:
https://doi.org/10.20378/irb-48428 (initial version)
https://github.com/UBA-PSI/psi-thesis-guide (most recent version)

Abstract

Virtual memory (VM) is a fundamental component of modern computer systems, widely adopted across various computing environments, from embedded devices to large-scale data centers. While it initially served to automate memory management by transparently swapping pages between main memory and secondary storage, it now plays a crucial role in ensuring system security, process isolation, and overall flexibility. However, the increasing overhead associated with traditional VM systems — originally designed for resource-constrained environments — has led to performance bottlenecks, particularly in systems with growing memory demands and applications with poor spatial locality. The ever-increasing depth of conventional hierarchical page tables further exacerbates these challenges.

To address these limitations, alternative approaches like *Inverted Page Tables* are used, but they have their own set of problems. The plethora of designs and approaches to optimizing these designs suggest general performance problems. This thesis proposes an approach that aims to eliminate the need for page tables and main memory accesses altogether by using specialized mapping functions instead of costly page table walks. These mappings promise faster address translation, as they omit the cost associated with the page table accesses, while providing flexibility to system designers by defining them in software. This allows tailoring the virtual memory system to specific use cases.

The thesis details the theoretical foundations and practical implementation of a platform designed to facilitate the exploration and experimentation with such mapping functions. It also presents an initial simplified VM system prototype that demonstrates the feasibility of the approach.

Contents

1 Introduction

Virtual memory provides a multitude of features which vastly simplify the life of application programmers [JM98b]. Computer systems of all scales, ranging from small embedded devices to huge data centers use virtual memory [BL17]. While originally being used to automate the task of swapping pages of processes between main memory and secondary storage transparently to the processes [JM98b], it now is the foundation of security, reliability, process isolation and flexibility [JM98a; Wal99].

With ever increasing memory sizes of systems and memory requirements of applications, the overhead of virtual memory systems, being developed for systems that only had scarce resources available [Hal+23], degrades performance and significantly increases power consumption [ZSM20]. Orthodox hierarchical page table organizations [TB14] can get as deep as 5 levels in commodity hardware [Int17]. Not only does this add a level of indirection for each page table walk, this cost is even higher in virtualized systems, potentially accounting for up to 5% - 90% of the runtime [YT16].

Alternative designs like inverted page tables realize page table mappings using hash functions [TB14]. While reducing the number of additional memory references, these approaches have problems on their own: Some features of virtual memory systems become harder to implement, exacting additional performance penalties [YT16] and page table lookups become a lot more expensive when following collision chains [JM98c].

Chapter 3 will show, that there are a lot of different approaches to optimize the virtual memory system. This shows that there is little agreement on how the virtual memory system is best implemented [JM98c]. However, most of the optimization approaches still rely on page table structures to do the bookkeeping of the mappings.

This thesis explores the idea of getting rid of page table structures all together and base the mapping of virtual to physical addresses on specially crafted functions (for example non-cryptographic hash functions [Mit+21]). Hash functions implemented by simple arithmetic instructions are orders of magnitude faster than the multiple memory accesses required by a page table walk [TB14]. Allowing them to be defined in software gives the operating system a lot of flexibility to fit the implementation of the mapping function to the custom needs of applications.

This thesis will describe the theory and implementation of a platform that facilitates the definition and the experimentation with such mapping functions and shows a first simple mapping function that realizes a simplified virtual memory system without any page tables.

Chapter 2 provides an overview of the fundamentals of virtual memory systems, software-managed virtual memory systems and some basics of the virtual memory system of the RISC-V ISA.

Chapter 3 takes a look at previous work which approaches the topic of optimizing virtual memory systems as well.

Chapter 4 describes the theoretical development of the idea presented in this paper and thus provides the foundation for the implementation.

Chapter 5 describes the implementation. It delves into the specifics of the programming platforms and outlines the implementation process step by step. This chapter also includes an overview of the debugging techniques used for verification and troubleshooting of the implementation.

Chapter 6 critically examines the current state of the theoretical development and implementation, analyzing the results in light of typical requirements for other virtual memory systems. It also includes a discussion on further deepening the approach.

2 Fundamentals

This chapter introduces some essential concepts and mechanisms which form the basis of the following chapters. It first gives an overview of *Virtual Memory (VM)*, its core requirements, tradeoffs and implementations. Then the hardware components and caches used to accelerate VM systems are presented. An overview of purely software-managed systems follows with a comparison of the general trade-offs between software-managed and hardware-managed Virtual Memory systems. Finally, some specifics of the memory system of the chosen implementation platform, *RISC-V*, are shown.

2.1 Virtual Memory

Virtual Memory was first introduced in the Atlas System [Fot61] to automate the task of swapping pages between main and secondary memory. The idea was to make programs completely unaware of real, physical memory by providing an abstraction layer called virtual memory [Den96]. With virtual memory, programs appear to have the whole memory space of the machine at their disposal. It also hides all the other processes and their memory from them. The task of managing a programs memory, no matter where in its virtual address space it is, and putting it somewhere in physical memory now falls to the operating system [Den70]. This change to the operating systems role in managing program memory was not only useful but also necessary. The size of programs was increasing faster than the size of main memory; while single programs still fit in memory, operating systems made it possible to run multiple programs at once, collectively exceeding the available physical memory [TB14].

2.1.1 Virtual and physical addresses

The terms of *Virtual* and *Physical* addresses are used a lot in the course of this thesis. Virtual addresses refer to those that are used by the program to reference memory object in its address space. They are only valid in the programs own address space. Thus, it is possible for programs to use the same virtual addresses, that point to different physical addresses. Virtual addresses are translated to physical addresses, which are actual addresses to memory locations on main memory. The *Virtual Memory System* is tasked with performing this translation, the creation of mappings, and keeping book of those mappings [Den96].

Some memory systems which use inverted page tables also use the notion of *effective* addresses. These form an addition layer of indirection that allows the sharing of pages [JM98b]. Effective addresses will not be discussed here any further.

2.1.2 Memory System Requirements

In modern systems, *Virtual Memory* does a lot more than swapping pages between main memory and secondary storage and providing new pages of memory on-demand to programs. It is the foundation of a number of requirements to the memory system that are taken for granted [JM97]. These requirements are as follows:

Address Space Protection / Isolation It should not be possible for one process to access the data of another process, unless explicitly shared. [JM98a]

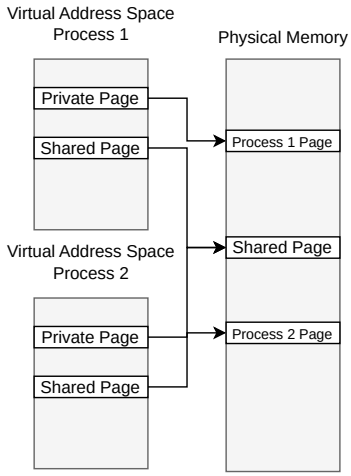

Figure 2.1: Page Sharing VM systems allow for the same physical page to be mapped into different virtual address spaces

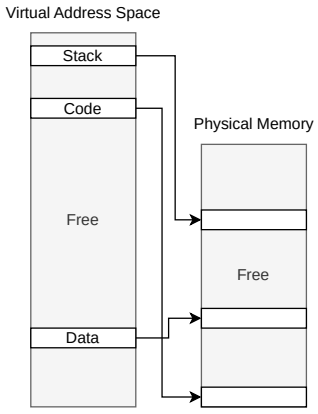

Figure 2.2: Flexibility Program segments can be dispersed anywhere around the virtual address space; the Virtual Memory System has to place the pages into actual physical memory.

Shared Memory Sharing memory allows programms to work on the same physical data with potentially differing virtual addresses [JM98a]. Shared memory is used as a high-throughput mechanism for inter-process communication or for working on a shared data structure [TB14].

Large Address Spaces Programs tend to require more and more memory. Swapping memory can help to support programs bigger than the actual size of main memory [TB14], but programs may also require more memory than the theoretical limit set by hardware and software. Modern architectures use bigger addresses to support programs that require a lot of memory [JM97; JM98a].

Fine-grained Protection From a security perspective, it is often not desirable to e.g. allow the code segment of a program to be writeable and the data section to be exectuable.*Virtual Memory Systems* provide read-only, read-write and exectue-only protection on a page granularity[JM97]. Illegal references may trigger exceptions, which allow the operating system to deal with the program [JM98a].

Superpages Structures may be bigger than a single page and may thus occupy multiple entries of the translation caches while only refering to one object. To avoid displacing other caches entries, *Virtual Memory Systems* use superpages to provide space for bigger objects to increase the reach of cache entries [JM97].

Flexibility Programmers should not have to think about the management of the resources required to do their job, that is the operating systems job [TB14]. As such, it should be possible to place the programs code and data anywhere in the programs virtual address space to make the programmers job as easy as possible and to increase the flexibility of the programs [JM98b].

Sparsity Big addresses and fine granularity result in a huge address space with a sparse population for programs that do not require a lot of pages [TB14]. A lot of programs do not require the full virtual address space and only occupy a small set of pages. Bookkeeping of smaller address spaces should be as cheap as possible.

There are different ways how memory systems implement these requirements. The two most common implementations are based on hierarchical/multi-level/radix [JM98b; TB14; YT16] page tables and inverted page tables [JM98b; JM98c]. These implementations are summarized in the following.

Figure 2.3: Sparsity / Large Address Spaces Virtual Memory Systems need to efficiently realize huge address spaces with only a few pages being used.

2.1.3 Implementation of Virtual Memory

Every virtual memory system has to realize a mapping from virtual addresses of each processes private, virtual address space to physical addresses that index data in main memory. This section will provide an overview of how most commonly used virtual memory implementation satisfy this requirement.

The naive approache is to simply have a big array in main memory. This array can be indexed using the virtual addresses, at which the physical address to that virtual address is placed. In a 32 bit address space with 4 KB pages, this requires 20 bits per page table entry. This adds up to

$$20 * 2^{20} bit = 20971520/8 Byte = 2.5 MB$$

To properly isolate the processes from each other, every process needs to have one of those arrays. To realize fine-grained protection of pages, there would also need to be some bits per array entry for read/write/execute rights. With 64-bit computers, the space requirements for such page tables would be even higher.

2.1.3.1 Hierarchical/radix/multi-level page tables

To reduce the memory cost of managing pages, most virtual memory systems use a multi-level page table, also known as a hierarchical page table. However, its structure is less like a table and more like a tree, where the nodes are tables of page table entries (PTEs) [TB14]. Here the virtual page number is divided into several parts. Each part of the Virtual Page Number (VPN) is used to index a smaller table. The indexed PTE then points to the next smaller page table, which in turn is indexed by the next part of the VPN. Depending on the implementation this can involve up to 5 further indirections. RISC-V, the architecture the implementation of this paper is based on, offers a 3-level page table design [PW17]. The deconstruction of the virtual address and construction of the physical address of this scheme is shown in 2.4. For bigger address space, RISC-V also offers 4 and 5-level schemes.

Traversing the page table to find the mapping requires an additional memory reference for each level [JM98a], and since finding the mapping is on the critical path of every memory operation, in the worst case, if all

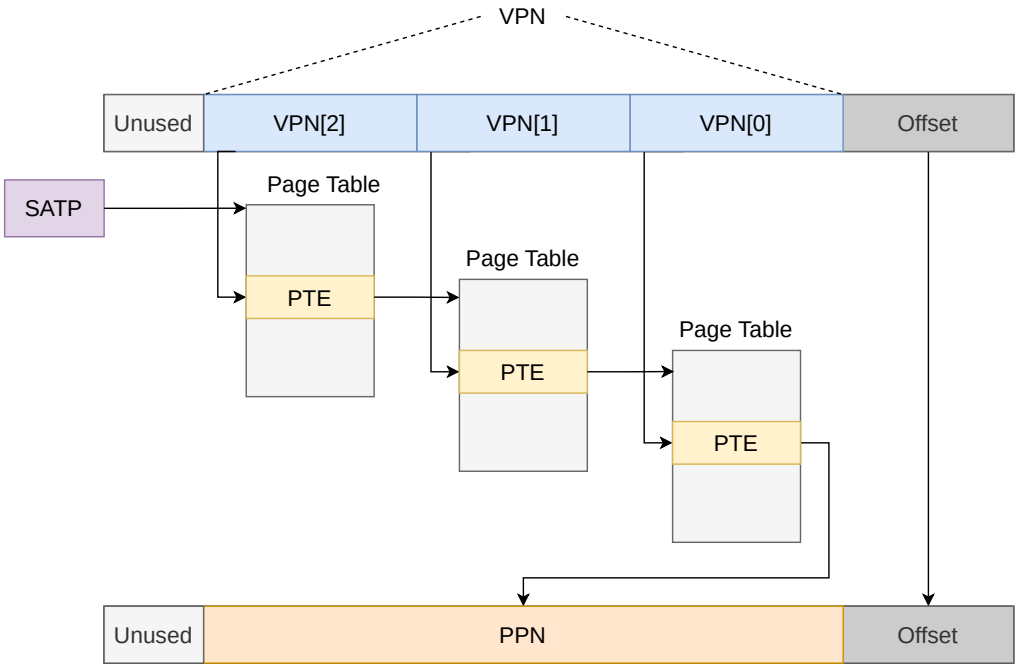

Figure 2.4: Three-step page walk with a RISC-V Sv39 Page Table Tree: The value in the `satp` register is the base of the root page table; `VPN[2]` is the index into the root page table; the indexed `PTE` points to the next page table. This traversal continues until the bottom of the page table is reached (or until a valid PTE is found). The last `PTE` contains the `PPN` of the physical address which can then be combined with the offset bits to make the full physical address [PW17]

caches miss, up to 5 memory accesses may be required (in a 5-level paging scheme) just to find the mapping for a single memory access.

2.1.3.2 Inverted page tables

An alternative paging scheme approaches the problem from the opposite direction and provides a PTE for each physical frame instead of one entry per virtual page. A physical page frame is a page-aligned space in physical memory for a page. The number of physical frames is determined by the size of main memory divided by the page size. This has the enormous advantage that the bookkeeping only needs to keep record of as many pages as the physical memory can support. In contrast, hierarchical page tables need to keep track of as many pages as there can be in the virtual address space of all processes combined. That is a huge memory overhead compared to inverted page tables [JM98c]. The page table design also has the advantage that, in the best case, significantly fewer main memory accesses are required. In the simple design shown in figure 2.5, the corresponding page table entry can be found with just two memory lookups [Ska+20].

To get a deterministic mapping from virtual to physical address, *Inverted Page Tables* are indexed using a hash calculated from the VPN. Since the domain of the hash functions (virtual address space) is much bigger than its codomain (physical page frames), collisions can happen. These

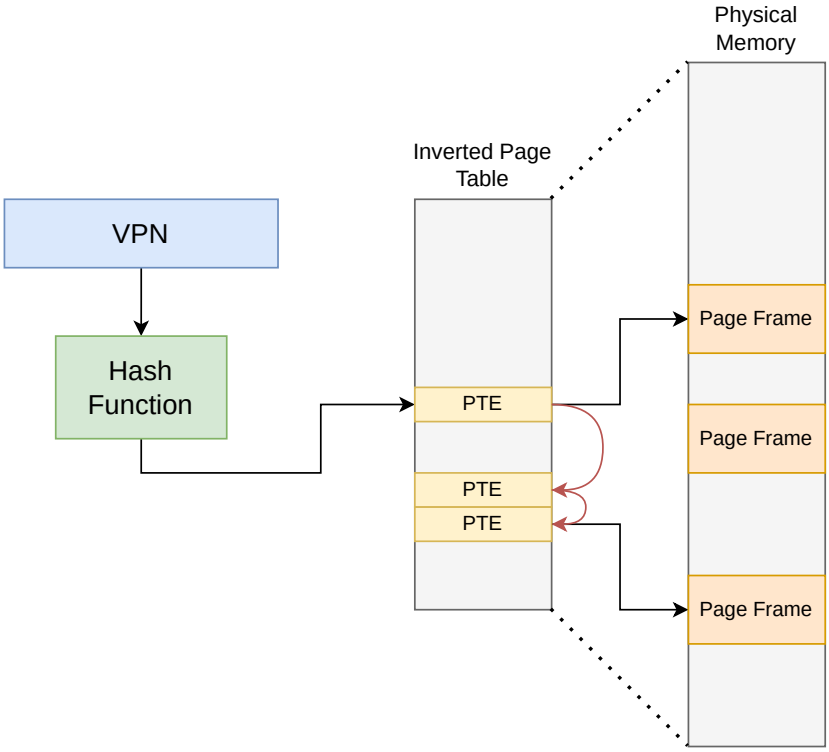

Figure 2.5: An inverted page table has an entry for every physical page frame, reducing memory accesses to a minimum of one. Collisions in the hash table (red arrows) can make the access much more expensive.

collisions are stored in a linked list. Since the collision chains can potentially become infinitely long, the access time of the inverted page table is theoretically unbounded [TB14].

Hierarchical page tables have the key advantage that they always require a fixed number of memory accesses to determine the PTE.

Typical inverted page table designs often include a so-called hash anchor table, which is then indexed by the calculated hash. Entries in the hash anchor table point to entries in the inverted page table containing the actual PTEs. If the hash anchor is twice the size of the page table, the average collision chain length can be halved [JM98b]. However, at least one additional memory reference is required in any case [JM98b]. Alternative designs for inverted page tables allow the table to be dynamically resized to avoid hash collisions. But resizing the whole page table is very expensive and should be avoided[Ska+20].

Inverted page tables significantly reduce the average access time to the PTEs, but they make it more difficult to support other features like superpages and memory sharing. The Power Architecture, for example, supports this through a two-stage translation process [YT16].

There is no clear winner in the debate between hashed and multi-level page tables. Superiority of one design over the other also appears to

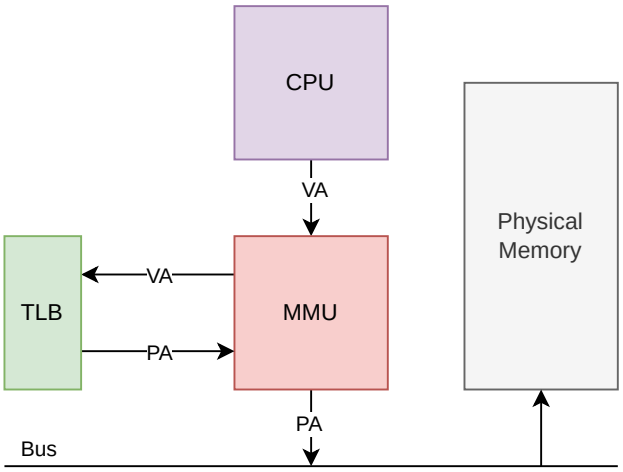

Figure 2.6: A simplified architecture of CPU, MMU and TLB: User-level programs running on the `CPU` try to access main memory with virtual addresses; virtual addresses get transparently translated to physical addresses by the `MMU` by either looking up the address in the TLB or by performing a page table lookup with the hardware-supported page table design

depend on the computer architecture [BCR] and the specific implementation of the design [YT16]. Commercial hardware supports a variety of designs that are not standardized and, in some cases, differ significantly [JM98c].

Modern Intel processors now support radix designs with a depth of up to 5 levels, which still use 4KB pages to maintain compatibility.

2.2 Memory Management Hardware

To further accelerate the translation of virtual to physical addresses, most modern computers use additional [Den70] hardware components . These consist of a hardware page table walker (MMU) and a translation cache, commonly referred to as a Translation Lookaside Buffer (TLB) [JM98a].

MMU The Memory Management Unit (MMU) takes on the task of address translation for the computer. It sits between the processor, which primarily works with virtual addresses, and the memory bus, which is accessed with physical addresses. When the processor accesses a certain virtual address, the MMU performs a page table walk to determine the physical address corresponding to the virtual address. During this process, the processor is effectively frozen [JM98a].

TLB Since it would be very costly to traverse the page table in hardware for every load or store memory access, there is a cache for the translations, the Translation Lookaside Buffer (TLB). This cache contains the most recent translations from virtual to physical addresses. The MMU can

first check the TLB, which can be searched in parallel and thus extremely quickly [Dre07; JM98a].

Address Space Identifies With Address Space Identifiers (ASIDs), RISC-V provides a way to utilize virtual memory more efficiently: Since every process has its own virtual address space, translations that are still present in the TLB may not be valid after a context switch.

SFENCE.VMA RISC-V provides one instruction that acts primarily as a memory barrier: It prevents reordering of instructions accessing memory accross the `sfence.vma` instruction. This is important when the page tables are switched, e.g. when the kernel is entered from user mode, because the translations will change. The instruction also acts as a flushing operation for TLB entries. With both of the optional register arguments set to zero, the instruction will flush all entries. Setting the first register to a specific address will only flush translations containing that address. The second register specifies the address space that is supposed to be flushed, given an ASID.

This mechanism allows for a precise control over flushing of TLB entries, which can improve the memory system performance [Wat+24].

2.3 Sofware-based Virtual Memory System

Software-based Virtual Memory Systems are generally more flexible than hardware-based systems as the mapping can be controlled in software and structures (such as the PTE) are not bound to the hardware [JM98b]. However, all of the approaches listed here still use a page table structure to keep track of mappings. Main differences are in the details of the implemented page table structure and in the way the software has to create the mappings. This freedom allows adjusting the virtual memory system to fit more specific use cases, as hardware-based systems are design to service a wide range of different applications[**citation needed**]. In the following, a number of software-based Virtual Memory Systems are presented.

2.3.1 MIPS

MIPS [16] allows software-management of TLBs by raising an exception on TLB miss and by providing instructions to write TLB entries from software.

Ultrix The Ultrix page table is a two-tiered page table with two exception handlers for the TLB miss exception. One of the handlers services user-level misses, the other one handles kernel-level misses. Kernel level misses may occur when the page containing a PTE is not loaded and causes a TLB miss [JM98c].

Guarded Page Tables In his dissertation, Jochen Liedtke presented an innovative approach to handeling virtual memory [Lie96]. The approach is based on hierarchical page tables, and is shown to provide an efficient solution particularly in systems with large, sparse address spaces. The design allows skipping over PTE entries to reach valid PTEs faster, without having to traverse every single level of the tree. Gernot Heißer showcased a implementation of that design in [Hei99].

PA-RISC PA-RISC uses a inverted page table design. Similar to Ultrix, the inverted page table is searched in the TLB miss handler when a TLB miss occurs. The TLB miss handler calculates the hash from the virtual address to locate the head of the hash collision chain and then searches linearily through the chain to find the mapping [JM98c].

NOTLB A comparative study conducted by Jacob et al. [JM98c] looks at a software-managed Virtual Memory Design that uses no TLB at all. The design uses a two-level page table structure similar to the ULTRIX/MIPS design and organizes its address space into disjunct segments. Instead of a TLB miss handler, there is a Cache-Miss handler that services misses in the virtual L2 cache. Similar to the Ultrix design, there are two handlers to handle nested cache misses.

2.4 Hardware Virtual Memory vs Software Virtual Memory

There are several considerations that should be taken into account when comparing hardware and software-managed translations.

Fixed Paging Structures In hardware-managed virtual memory, the structures for page tables and page table entries are fixed by the microarchitecture. As a result, the operating system cannot tailor memory management to its purposes and use case, and it is stuck with the fixed design. This also complicates the portability of system software, as there is no standard for these memory management structures. Despite there being no significant performance differences among the various designs, there is no standardization [JM98c].

Pipeline Freezing / Flushing On a TLB miss, with a hardware-managed TLB, only the pipeline freezes (at least for instructions dependent on the memory access). However, with a software-managed TLB, control is handed back to the operating system via an exception, which notes that the required address is not in the TLB (TLB miss exception). The jump back to the operating system causes a context switch, requiring the state of the current process to be saved. During this, the reorder buffer is flushed, and the pipeline is heavily disrupted. Switching to the kernel can also lead to further data and instruction misses in the long term as the kernel entry likely overwrites some cache lines that the running process needed.

Complexity Choosing between hardware and software virtual memory is choosing where to shift the complexity. Systems that do only support software managed virtual memory are simpler on the hardware side, because the burden of managing the memory is shifted to the software, which in turn will be more complex. This goes in both directions [JM98b].

[JM98c] concludes in a comparative study of various hardware and software memory designs that hardware-based approaches are generally more performant, but software-based designs are certainly viable if the caches are large enough to reduce the number of cache misses. Especially in terms of flexibility, software-based approaches have a significant advantage, as the VM system can be fully defined by the operating system.

2.5 RISC-V Basics

The implementation presented in this work runs on a RISC-V platform. Therefore, it is necessary to go over some basic concepts of the RISC-V platform. Particularly relevant here are the virtual memory system, the exception/trap mechanism, and the control and status registers (CSRs), which form the foundation for extending RISC-V. The information presented in the following section is taken from both the RISC-V Reader [PW17] and the RISC-V ISA Specification [Wat+24].

2.5.1 Sv39 Virtual Memory

The RISC-V ISA specifies a simple, modular and scalable page-based virtual memory system [PW17]. It supports different page sizes as well as different addressing modes: Sv32 for 32-bit systems; Sv39 for 64-bit systems with a 3-level page table. For bigger memory needs, there are the Sv48 and Sv57 addressing modes, adding one and two levels to the page table and supporting 256 TB and 128 PB respectively. The following will only look at specifics for the Sv39 addressing mode.

The page table is an orthodox hierarchical multi-level page table, where each Page Table Entry (PTE) points to a PTE in the next level. Each level of the page table contains 512 PTEs with 64-bit in size, bringing a page table to exactly 4 KB in size. The format of a PTE is shown in figure 2.7.

Figure 2.7: RISC-V Sv39 Page Table Entry

The translation process (which was already shown in Figure 2.4) splits the virtual address (See figure 2.9) into 4 logical parts: Three parts are parts of the Virtual Page Number (VPN), which are used to index the different page tables that are being traversed. If the page table walk succeeds, the last PTE will contain a Physical Page Number (PPN), which will then be combined with the offset to create a physical address (See figure 2.8).

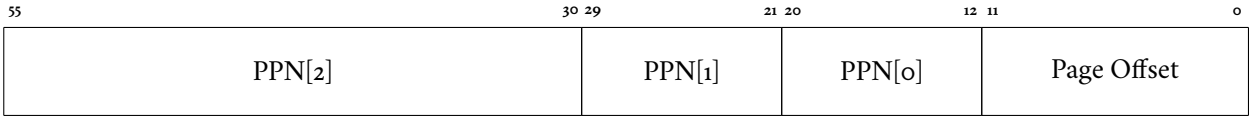

Figure 2.8: RISC-V Sv39 Physical Address

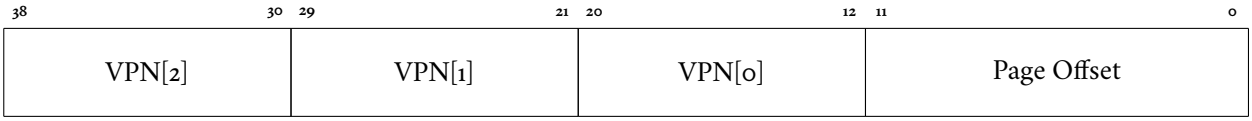

Figure 2.9: RISC-V Sv39 Virtual Address

Essential to the configuration of the virtual memory system in RISC-V is the `satp` register (See figure 2.10). It configures whether virtual memory is enabled at all or if direct mapping is used for all addresses. It is also used to set the addressing mode. Beyond the configuration of the virtual memory system, it contains the base address pointing to the root page table used for translations.

2.5.2 Traps

Traps are part of the RISC-V privileged architecture. They provide a mechanism to respond to external events and unusual runtime events, known as exceptions [PW17]. The term ”trap” is an umbrella term that is further divided into interrupts — asynchronous events — and exceptions — synchronous events. Exceptions are particularly of interest here, as a TLB miss occurs during the execution of an instruction, meaning it happens synchronously with the processor’s clock. However, it is important to keep interrupts in mind when working with Qemu and xv6 source code, because on one hand, the Qemu code handles exceptions and interrupts within the same functions, and on the other hand, xv6, or RISC-V in general, uses a unified vector for handling both interrupts and exceptions [Wat+24].

There are six central registers for triggering and handling exceptions. These exist both for Supervisor Mode (prefixed with ”s”) and Machine Mode (prefixed with ”m”). Whether the machine mode or supervisor mode version of the register should be used for an exception depends on the mode in which the exception is handled. In the following, all registers are presented with the M-Mode prefix only .

Exception Vector The hardware thread (hart) experiencing an exceptional state must know where the kernel routine is located to handle the exception. The `BASE` field of the register contains a 4-byte aligned physical address to which the program counter is set in the case of an exception. The `MODE` field allows switching between `direct` and `vectored` modes. In `MODE=Direct`, the PC is set to BASE for all traps, whereas in `MODE=Vectored`, the PC is set to $BASE + 4 * CAUSE$ for asynchronous interrupts.

Context Information To properly handle the exception, some context information is required. The **mcause** register contains the exception

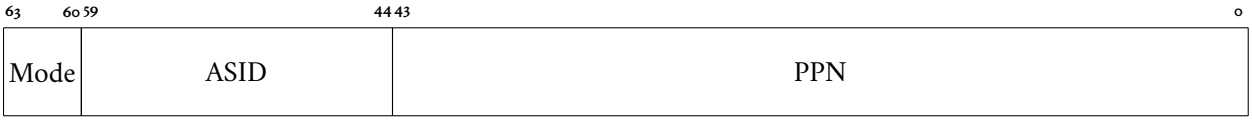

Figure 2.10: RISC-V Sv39 satp CSR

code of the exception; **mepc** contains the program counter of the instruction that triggered the exception; and **mtval** holds exception-specific information, such as the virtual address that triggered a page fault exception. **mstatus** contains general information about the current hardware state.

Delegation Normally, all exceptions are handled in machine mode; however, in some cases, it may be useful to handle the exception in a lower privilege mode. With the bitfield in the **medeleg** register, individual exceptions can be chosen to be delegated to the next-lower privilege mode.

Exception Code Each exception is assigned a unique number, the exception code [PW17]. This can be found in the `mcause` register when handling the exception.

Hardware Trap Mechanisms When a trap is risen, the hardware does the following: [CKM11]

1. interrupts are disabled by clearing the `MIE` bit i **mstatus**
2. `pc` is copied to `mepc`
3. the current mode is saved to the `MPP` field in mstatus
4. `mcause` is set to the proper exception code
5. the mode is set to the machine mode
6. `pc` is set to stvec
7. execution continues at the new `pc`

2.5.3 Contol and Status Registers

The RISC-V ISA provides a 12-bit encoding space for 4096 CSRs. A CSR address is logically split into four parts: The top two bits `csr[11:10]` specify whether the CSR is read/write or read-only; `csr[9:8]` encode the minimum priviledge level that is allowed to access the CSR; `csr[7:4]` may be partially used to define a specific use for a range of CSRs. E.g. CSRs with an address between `0x7B0` and `0x7BF` shall be used for Debug-mode-only CSRs [Wat+24]. The format of the CSR addresses is also depicted in figure 2.11.

Each legal CSR address identifies a CSR. The size of the CSRs identified by the CSR address depends on the values of the SXLEN and UXLEN fields in the `mstatus` register. Currently, the specification [Wat+24] allows for 32-bit, 64-bit and 128-bit. RISC-V provides dedicated instructions for

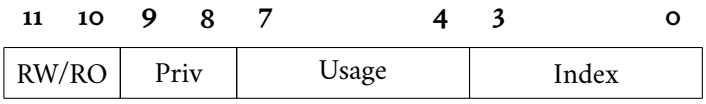

Figure 2.11: RISC-V CSR address format `csr[11:10]` define whether the CSR is read-write or write-only; `csr[9:8]` defines the minimum priviledge level that is allowed to access the CSR; `csr[7:4]` sets a directory within the CSR space for the usage of the CSR; `csr[3:0]` define a index

read/write and bit manipulation with both register values and immediates.

3 Related Work

This work can be broadly classified to be in the field of Virtual Memory optimization. There is a large body of literature that deals with optimizing the Virtual Memory system, as it lies on the critical path of every memory operation.

There are different approaches or perspectives to addressing the system (this is not an exhaustive list):

- page table structures and their optimization, e.g., inverted or hierarchical
- caching of pages, PTEs in the TLB, and cache indexing
- cache replacement strategies
- cache sizes, associativity of caches
- Page Walk Caches (PWCs) to store partial translation results

[Lie96] shows an innovative design leveraging the flexibility of software-managed address translation. The design of Guarded Page Tables (GPTs) is based on hierarchical page tables, but allows skipping over levels of the table to reach translation results faster. This proves to be very effective for increasing the performance of virtual memory systems, as the biggest penalty comes from page tables walks needing to reference memory for every level in the page table. Gernot Heiser showcases a practical implementation of GPTs in the L4/MIPS system [Hei99].

[JM97] explores software-managed address translation and analyses the efficiency of a PowerPC implementation of the presented design they call *softvm*. They show that software-managed address translation can achieve better performance and at the same time simplify hardware by dispensing with translation caches and the hardware state-machine for walking the page table. The approach is based on handling virtually indexed and tagged cache misses in software. With sufficiently-sized virtual caches the system can go for long periods without requiring translations. Not unlike this paper, they extend the PowerPC architecture by two new instructions to write entries to the cache. However, their design uses software page table walks to find translations on cache-miss, while the approach presented in this paper presents a software-based TLB fill mechanism with segmented memory allocation.

[BCR] examines the design space of translation caches in MMUs. These caches are not the same as TLBs: TLBs contain full translation results,

while the translation caches considered here contain partial translations. In the case of a cache hit, these partial translations allow the system to skip individual steps when traversing the page table tree, thereby saving one or more memory accesses. Otherwise, each level of translation requires a memory reference. These caches are also referred to as Page Walk Caches (PWC) [YT16].

The specific translation cache designs of AMD and Intel platforms are examined and compared to three other designs proposed by the authors. Barr et al. conclude that radix page tables, by caching entries at higher page table levels, can outperform inverted page table designs.

This work focuses on optimizing the memory path by eliminating page tables and using a software-controlled TLB, and it does not further consider caches aside from the TLB.

[YT16] challenges the results of [BCR] and argues that the obtained results are based on a suboptimal implementation of the inverted page table.

They conclude that a well-optimized inverted page table can outperform a radix page table equipped with PWCs. However, they also address the conceptual disadvantages of inverted page tables. For example, it is more difficult to implement superpages.

The work takes a closer look at the differences between various page table designs and the requirements of a memory system (such as superpages or page sharing). These requirements are also important for the design presented here. However, this work aims to avoid using page table structures altogether.

[Par+22] identifies that today's memory capacities far exceed the coverage of TLBs, causing memory-hungry applications to suffer from frequent page table walks (PTWs).

Two approaches are presented to reduce the associated costs: The first approach aims to reduce the number of memory references per PTW by combining two levels of the page table into one. The second approach modifies the cache replacement policy so that cache entries containing PTEs are more likely to remain in the cache during periods with many TLB misses, allowing PTWs to run directly from the cache instead of being loaded from main memory.

They show a 2.3% performance improvement from flattening the page table tree, 6.2% through cache prioritization, and a combined performance improvement of 9.2%. Both approaches focus on optimizing access to page table structures and are separate from this work, as this work aims to eliminate these structures entirely.

[Ska+20] presents a novel page table design called *Elastic Cuckoo Page Tables*. The design exploits memory-level parallelism to enable fully parallel page table lookups. At the core of the design is the Elastic Cuckoo Hashing algorithm, which allows multiple hashing locations for a given element and also enables efficient, gradual resizing of the hash table. Skarlatos et al. demonstrate an application execution speedup of 3-18% using the Elastic Cuckoo Page Table design.

[ZSM20] proposes an approach that shifts responsibilities of memory management in parts back to the applications: All applications get a fixed-size chunks of physical memory and then have to manage allocations across these blocks. Thus the task of memory management with the chunks falls to compilers, language runtimes and the applications themselves. This approach does not require any address translation and thus gets completely rid of the overhead associated with the virtual memory system. Common features of virtual memory, like memory space protection can still be implemented with physical memory protection mechanisms present on commodity hardware. Overall, this approach trades a reduction for complexity of the hardware with increased complexity on the software side.

The design presented in this paper resorts to bigger segments per process, but generally aims to keep the memory management responsibilities with the operating system. It tries to get rid of the page table structure in favor of mapping functions.

[Hal+23] argues that prevailing memory management system designs are becoming increasingly inefficient given modern systems that have very big main memories and workloads that require large in-memory data sets. The paper presents a novel approach addressing the limitations of current methods with *Morsels*. These Morsels are self contained memory object, spanning entire page table sub-trees, thus allowing efficient remapping, sharing and reducing memory overhead. They reuse existing interfaces and work on top of existing page table structures to provide a supplementary layer to improve the efficiency of memory-intensive applications.

The related work presented here focuses on optimizing current page table designs and their hardware support. This work will explore the feasibility of implementing virtual memory without any page table whatsoever using a specialized mapping function to generate virtual to physical mappings. This promises to reduce the overhead of TLB misses caused by expensive main memory access.

4 Theory

This chapter presents the theoretical foundation of the the implementation presented in the next chapter. It will start with a short elaboration on the general idea followed by a description of the programming platform. It then presents the theory behind the different components of the implementation.

4.1 Function-based Virtual Memory

The goal of this work is to provide a proof of concept of an alternative approach to virtual memory management that reduces the overhead associated with traditional page tables walks. Instead of suffering the penalty of up to five memory accesses on a TLB miss, we want to explore using simple functions for the mapping.

Figure 4.1 shows the usual architecture of a system with a hierarchical page table. This architecture provides only limited flexibility in interacting with the TLB, especially when it comes to TLB writes. Architectures like MIPS allow direct manipulation of TLB structures and thus provide a good framework for implementing new approaches for virtual memory management.

The architecture is only one half of the design. To test the ideas for mapping functions, a system utilizing the memory system and defining the functions is necessary. The xv6 educational operating system provides a good platform for experimentation as it is very lightweight and provides a Unix-like interface and structures.

xv6 is designed for RISC-V, but RISC-V does not allow for flexible software-management of the TLB structure. Thus part of this chapter will describe the theoretical foundations for modifying the QEMU/RISC-V emulator to transform the memory system architecture to something more similar to MIPS. The final architecture for the memory sytem will look like depicted in figure 4.2

To adapt the RISC-V emulator to fit the architectural design, a number of changes to the implementation of the emulator are necessary:

- Extending the Qemu emulator to trigger an exception on a TLB miss
- A machine-mode trap handler in xv6 that handles the TLB miss exception

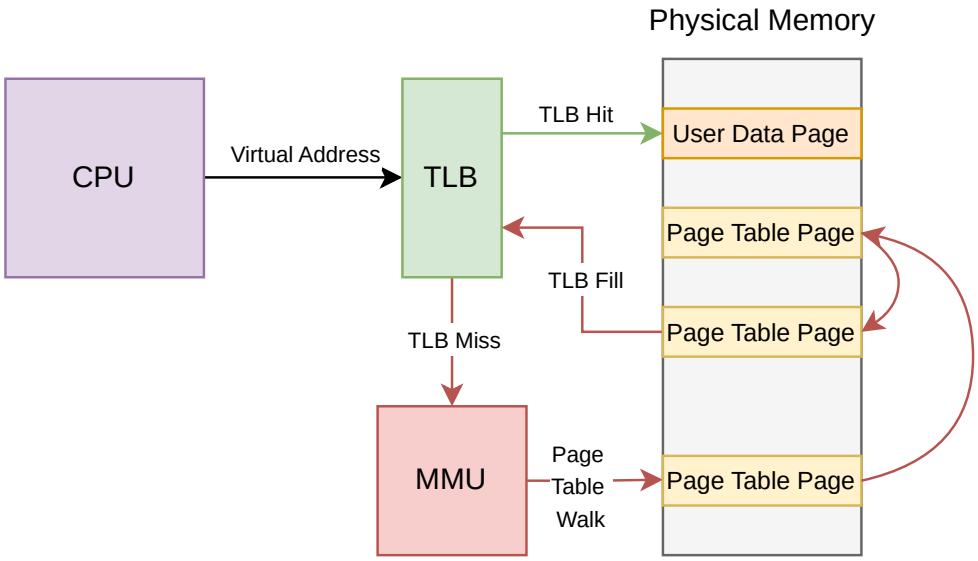

Figure 4.1: This figure shows what usually happens when the TLB misses: the miss will invoke the hardware state machine page table tree walker; the walker traverses the page table tree and if a valid PTE is found, the mapping is added to the TLB. The processor then executes the failing instruction again which will then result in a TLB hit

- A way to write TLB entries using special instructions

4.2 Platform

The chosen platform is the xv6 operating system [24a]. It is a teaching operating system used by MIT operating systems courses to teach the basics of operating systems. xv6 implements the basic Unix Version 6 interfaces, but does so in a very simplified fashion. There is a x86 version and a RISC-V version. For this project, the RISC-V version was chosen. xv6's simplicity and the accompanying handbook [CKM11] make it a good choice for a first proof of concept.

xv6 will be run on the QEMU [24b] emulator. The QEMU RISC-V emulator implements all the RISC-V features and extensions that xv6 needs.

Since using actual RISC-V hardware is not possible within the scope of implementing a new exception, an emulator is used to implement and utilize the necessary changes to the ISA. The QEMU emulator was chosen for this purpose, as it is very comprehensive and performant, and also emulates a TLB.

4.3 TLB Miss Exception

Operating systems running on RISC-V hardware are not directly notified of TLB misses. A TLB miss can only be inferred when a page fault occurs.

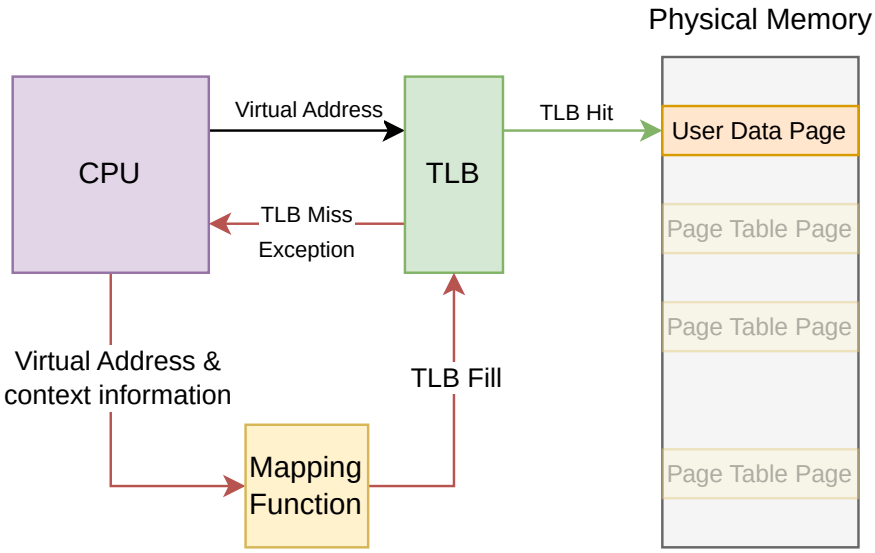

Figure 4.2: Instead of emulating a hardware page table walk in software on a TLB Miss exception, a mapping function is invoked that calculates the PTE using arithmetic primitives and tries to avoid memory references.

Otherwise, TLB misses will be handled by the MMU. To manage a TLB in software the operating system needs to be made aware that a TLB miss occured. The natural mechanism to do this is the exception mechanism.

RISC-V Exceptions Unlike MIPS [Hei99], RISC-V does not raise a TLB miss exception on TLB miss. Instead, the *Virtual Address Translation Process* is initiated: The MMU will walk through the page table tree and may, depending on the PTEs, either throw different kinds of page faults, or successfully add the missing entry to the TLB. At this point the faulting instruction will either be repeated, if the TLB fill was successful, or a page fault exception will be invoked [Wat+24]. The kernel then has to handle that page fault with help of the contents of the `satp` and `mtval` (or `stval`) registers.

The behavior of a TLB Miss exception should be pretty similar: If the TLB hits, the program will just continue as usual. If the TLB misses, a exception should be risen, providing context information about the miss in registers. The invokation of the hardware page table walker is not necessary.

Adding a new Exception to RISC-V Extending the emulator to throw a new exception is explained in the Implementation chapter. On the theoretical side, it is important to chose a exception code from the code ranges designated for custom use. These are $24-31$ and $48-63$ [Wat+24]. For the implementation we chose 24 or $0x18$.

4.4 Exception Handling

Throwing a exception is only one half of the puzzle. The operating system also has to handle the exception properly.

L4/MIPS TLB Exception Handling MIPS vectors its exception vectors in a continuous area in memory, starting at a base address. That gives the TLB miss handler 32 instructions to server the TLB Miss. Otherwise it has to jump somewhere with more space. The *Fast TLB Miss* handler is able to service the miss using only the two kernel-reserved registers `k0` and `k1` and another register, which has to be saved first [Hei99]. Thus the memory footprint of handler is really low.

xv6 Exception Handler xv6's exception handler does not actually handle most of the exceptions. For most of the exceptions, but notably also the different page fault exceptions, xv6 will simply kill the offending process [CKM11].

TLB Miss Exception Handler To simplify things, we will let the TLB Miss exception handler run in machine mode only. The usual mode for a operating system kernel would be the supervisor mode. However, this mode also uses address translation. This could create a situation in which we would have to deal with nested exception.

Catching the Exception M-mode exceptions will set the program counter to the address specified in the `mtvec` register. Originally, xv6 only serviced the asynchronous timer interrupt in machine mode. Every other trap is forwarded to supervisor mode. The trap vectoring mode can also be changed by setting the `MODE` field of the `mtvec` register [Wat+24]. The alternative mode forwards each interrupt to its own address, set at a fixed offset from the `BASE` address.

Every exception will still be forwarded to the `BASE` address set in the `mtvec` register, regardless of the vectoring mode.

The value of the `mcause` register identifies the reason for the exception. This value can be used in something like a switch-case statement to call specific handlers for the different exception causes. This switch-case is not really necessary in xv6, since the only exception that can enter the machine-mode exception handler is the TLB Miss exception.

Before running any code that modifies general-purpose registers, the current state of the registers and the program counter should be saved to memory. These need to be restored after the exception handler code has finished. This is essential to keep the process, that was just running when the exception triggered, in an consistent state when execution resumes.

4.5 TLB Filling

Not only does the operating system need to be made aware of TLB misses, the operating system also needs to be able to write to the TLB to actually create mappings that will be used after the handler finishes. RISC-V also does not provide any means to write to the TLB out of the box. But the RISC-V Control and Status Registers (CSRs) allow for a great deal of extensibility [PW17]. With CSRs, it is possible to implement custom behavior, which can then be accessed using instructions of the `CSRR` instruction group. To make an informed decision on how the CSR format for TLB writing may look like, we will first look at the TLB and the TLB instructions of *MIPS*. Then we will take a look at what the RISC-V ISA specifies about TLB, how QEMU implements TLBs and finally we will look at a the CSR format that is used for the implementation.

4.5.1 MIPS TLBs

The MIPS64 instruction set manual [16] shows a number of different instructions concerned with invalidation, probing, flushing, reading and writing (indexed and random). The most interesting instruction for a first design would be the `TLBWR` instruction for writing a TLB entry at a random index. With a similar instruction in RISC-V, we can already implement a purely software-controlled virtual memory system. The other types of TLB instructions that MIPS provides are not strictly necesarry, except for flushing. Without being able to flush existing translations from the TLB, user mode processes may try to access physical mappings stemming from other processes.But the RISC-V priviledged Architecture already provides this functionality with the `sfence.vma` instruction [PW17].

Replacement Strategies An advantage of software-managed TLBs is that the operating system can implement custom TLB replacement policies, that may even change depending on workload, programs running and other circumstances. The default replacement strategy for the MIPS `tlbwr` instruction is to simply use the value of the `C0_RANDOM` register as the index for the next TLB entry to be replaced. The name of that register is missleading, because it is not actually a random value, but it is rather decremented on each instruction[Hei99]. It is not clear whether this is a sensible replacement strategy, but it can be used to ensure that the same TLB slot is not used for every `tlbwr` if the implementation does not provide for any further replacement strategy. Some TLB entries can be protected from this "random" replacement by setting a value in the `C0_WIRED` register. The value in this register represents a lower bound, protecting all TLB entries that lie below it. This is useful to keep some mappings in the TLB that are valid all the time.

Having a protected space of TLB entries can especially be useful for global mappings. xv6 employs such a global mapping for every process and the kernel with the trampoline page.

MIPS - TLBWR The arguments for the instruction need to be written in some other registers - `EntryHi`, `EntryLo0`, `EntryLo1` and `PageMask`. The

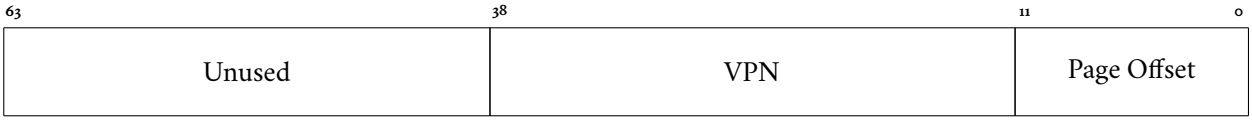

Figure 4.3: TLBH CSR Format The format is derived from the format of Sv39 virtual addresses and also works for Sv48 and Sv57 virtuall addresses by keeping the most significant bits unused

contents of those registers directly correspont to the TLB entry, which creates quite large TLB entries [Hei99].

4.5.2 TLB CSRs

The minimum information the hardware needs to create a mapping is the faulting physical address and the virtual address it maps to. The mapping has a granularity of 4 KB pages. Thus the lower 12 bits of both the physical and the virtual address are not needed. This still leaves $2 * (64 - 12) = 104$ bits for the mapping. The virtual address cannot be easily shortened anymore. Taking some of the most significant bits would shorten the virtual memory space of the programs. The physical address could still be shortened to only have enough bits to cover the physical address space used by xv6. But this would still end up taking more than 64 bit and thus more than one CSR.

So instead of trying to save as much space as possible, two CSRs are used. This also leaves enough room for more experimentation with the format. They are called `tlbwh` (TLB Write High) and `tlbwl` (TLB Write Low). The emulator will expect the physical address 2.8 in `tlbwh` and a the virtual address in form of a PTE 2.7 in the `tlbwl` CSR.

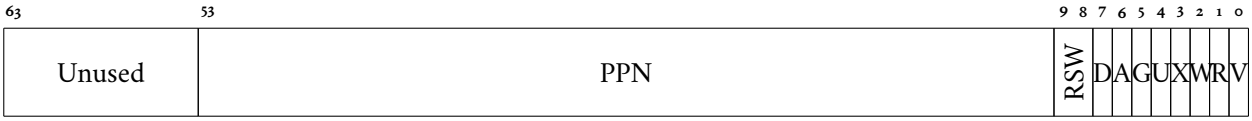

Figure 4.4: TLBL CSR Format The format is derived from the Sv39 PTEs. Reserved bits at the top are omitted. Access rights are retained to be set in the TLB

4.6 Mapping Functions

4.6.1 Segmented Mapping

The first attempt at a stateless paging design uses a design that determines memory allocation at compile time. The resulting mapping is similar to a segmented memory design.

Segmeneted Memory *Memory Segmentation* splits available memory into logical segments following the structure of a program. Thus there may be a segement for the programs code, the data, stack and so on. For bookkeeping, there is a segment table keeping track of the base address of the segment and the size of the segment. The operating system uses the base address to calculate the physical address from the logical address

consisting of segment number and offset. The segment size is used to ensure that accesses beyond the segment result in an error [TB14].

The main advantages of this memory design are, that the memory layout follows the logical structure of the programs and that the segments are protected from undesired access (like writing to the code segment). Typical segmented memory designs also allow for dynamic segment sizes [TB14].

Tableless Segmented Virtual Memory As a typical segmented memory design splits the programs memory into logical units, this design uses segmentation to split the whole physical memory into logical parts. These logical parts however do not reflect the structure of a program, but distinct process address spaces: Each segment of main memory can hold one process. To keep the design as light-weight as possible, each segments has a fixed size, which is determined at compile time of the operating system. This is less flexible but omits the need of storing the segment size for each segment, as they are all the same size. To calculate a physical address from a virtual one, all the mapping function needs is the ASID (segment number) and the offset into the address space. To check whether accesses reach over the the segment, only the static segment size is necessary.

This approach also protect logical units (process address spaces) from each other. This is made sure by flushing the TLB on context switch.

Program Layout The segmented memory design assumes that programs place their data at the low end of their address space and grow upwards. Placement of memory pages at higher addresses than the maximum per-address space memory size does not work. This is a serious restriction to the flexibility of the programmer. Conventional processes load the stack segment at the very top of the virtual address space and the heap segment at the bottom to grant both of them as much space to grow as possible [TB14]. This is still possible with segmentation, but requires context information at compile time about the maximum size of a segment.

xv6 processes do however satisfy these restrictions. Figure 4.6 shows the memory layout of an xv6 process.

Trapframe and Trampoline xv6 pins special pages to the top end of a processes address space: The trampoline and trapframe pages. The trampoline page contains code for context switches and kernel entry and needs to be mapped into every processes address space. The trapframe is used to save the state of the process on context switch and is thus essential to be present for every process.

In xv6's original memory system, both of these pages would be pinned to the maximum virtual address. With the segmented design, the address space only comprises the range of

$$[0; MEM\_START + ASID * MAX\_AS\_SIZE]$$

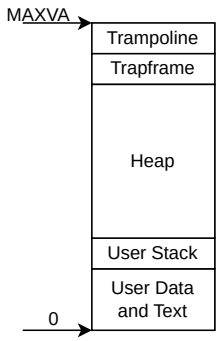

Figure 4.6: Virtual memory layout of xv6 processes. Taken from the xv6 book [CKM11].

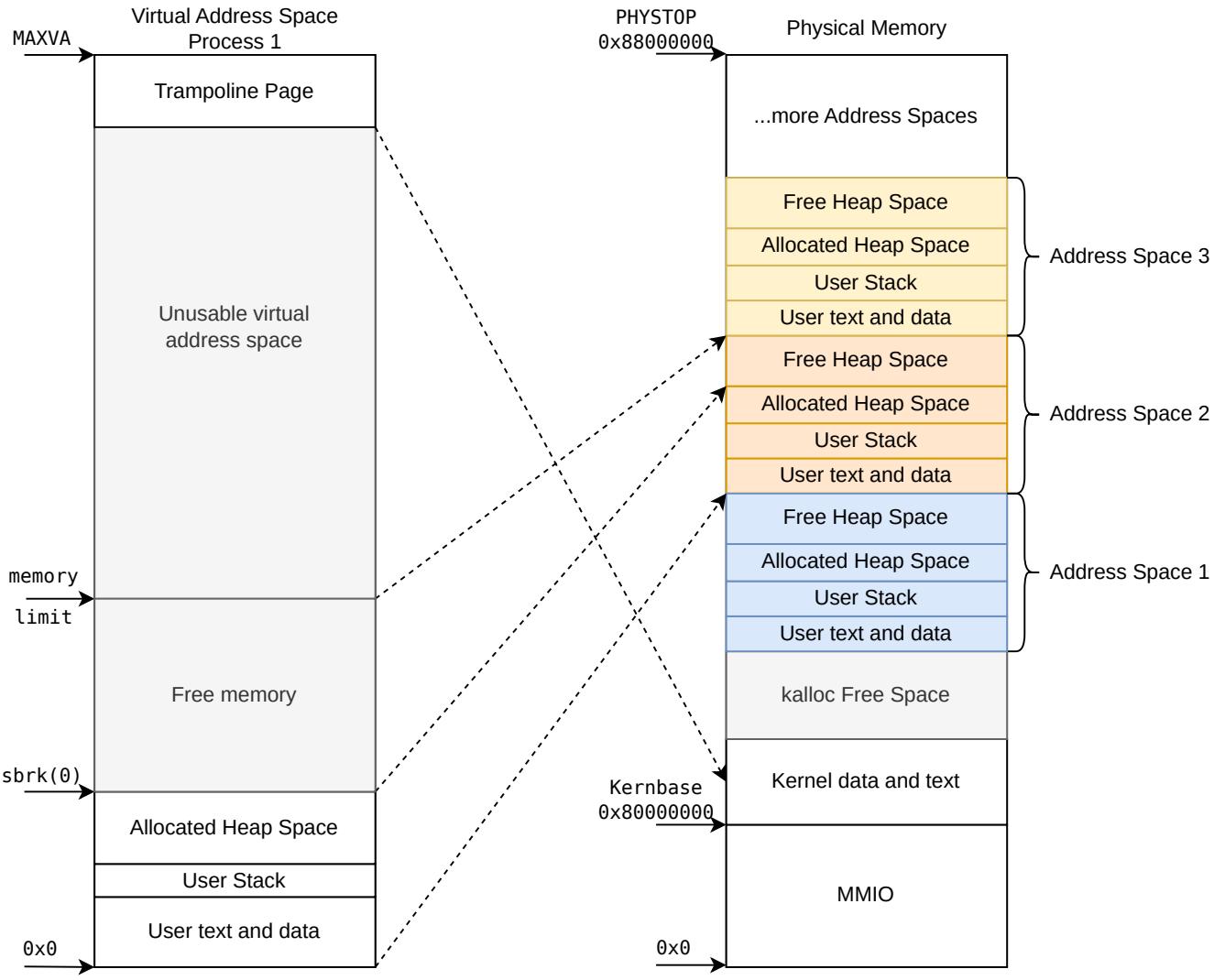

Figure 4.5: Segmented Memory Layout

with `MEM_START` being the start of the `RAM`, `ASID` being the *Address Space Identifier* of the process and `MAX_AS_SIZE` being the maximum size of one address space. `MAX_AS_SIZE` is determined by the size of available physical memory devided by the maximum number of processes that can simultaneously run.

The address space of a process would thus not be big enough to include the usual addresses for the trapframe and trampoline pages. To not change the memory layout of xv6 processes, a special rule is put into the mapping function to explicitly check for the usual addresses of trapframe and trampoline and then map them to appropiate physical addresses. The trampoline page can be the same for every process, but the trapframe needs to be unique per process and is thus pinned to the very top of the physical segment of each process.

Address Calculation The calculation of physical addresses from physical ones is very straight forward. All that is required is the processes ASID

and the offset into the address space. The ASID is saved in the process control block (PCB). But because accessing it may be expensive when frequent TLB misses occur, the ASID is encoded into the `satp` register. `satp` is normally used to provide the base address of the root page table to the memory management hardware, but since there is not page table to be used in the memory manager, the register is free to be used for additional context info in the exception handler.

Looking back at figure 2.10 shows that there is a ASID field in `satp` anyways, so the `PPN` field does not have to be touched. But the implementation may chose to use a different number of address spaces than there are supported by the `ASID` field to allow more processes to be run at the same time. But with 16 bit, $2^16 = 65536$ address spaces can be supported, which should be more than enough .

This optimization is based on the assumption, that TLB misses (and thus calls to the exception handler) occur more frequently than context switches, which should be granted when processes use more than one page and have all of their TLB entries flushed at context switch. So the access of the ASID is moved from the frequent TLB miss exception handler to the infrequent context switch.

5 Implementation

This chapter summarizes the implementation of the *softtlb* page-table-less virtual memory system. softtlb is a term used secludedly in this paper and refers to the software handling of TLB misses via a exception handler. The ordering of the sections in this chapter reflect the implementation process of the softtlb design and a mapping function implemented on top of that design. It consists of the following steps:

1. **TLB miss exception and Exception Triggerer** The first step is about the implementation of the TLB miss exception in the QEMU RISC-V emulator. On the xv6-side, a user-mode program is added to trigger a TLB miss exception from the shell.
2. **Exception Handling and TLB Writing** The second step implements the handling of the TLB miss exception by implementing a machine mode exception handler. The RISC-V emulation is extended by two new CSRs that facilitate writing TLB from the software-side of things.
3. **Software Page Table Walk for all Addresses** This step removes the restriction in the QEMU emulator to only throw exeptions for a specific address. In the exception handler, a page table walk is implemented to now create virtual to physical mappings for all addresses.
4. **Segmented Memory Design using software-defined TLB Filling** In this step, the xv6 virtual memory system is completely replaced. The new design gets rid of the page table and only uses information present in registers to create virtual-to-physical mappings and to fill the tlb.

Each section elaborates on both the xv6 side and the QEMU side of the implementation. The final section describes debugging techniques that were used or can otherwise be useful for similar implementations.

5.1 TLB miss exception and Exception Triggerer

There are two requisites the hardware needs to fulfill in order to do a software-managed TLB fill: to be a way to signal to the operating system that a TLB miss occurred and there needs to be some sort of instruction that can be used to write TLB entries. This step is about the former: Changing the QEMU RISC-V emulator to throw a newly defined *TLB miss exception* whenever the TLB misses. Changing the whole system to start throwing a TLB miss exception on every virtual address would

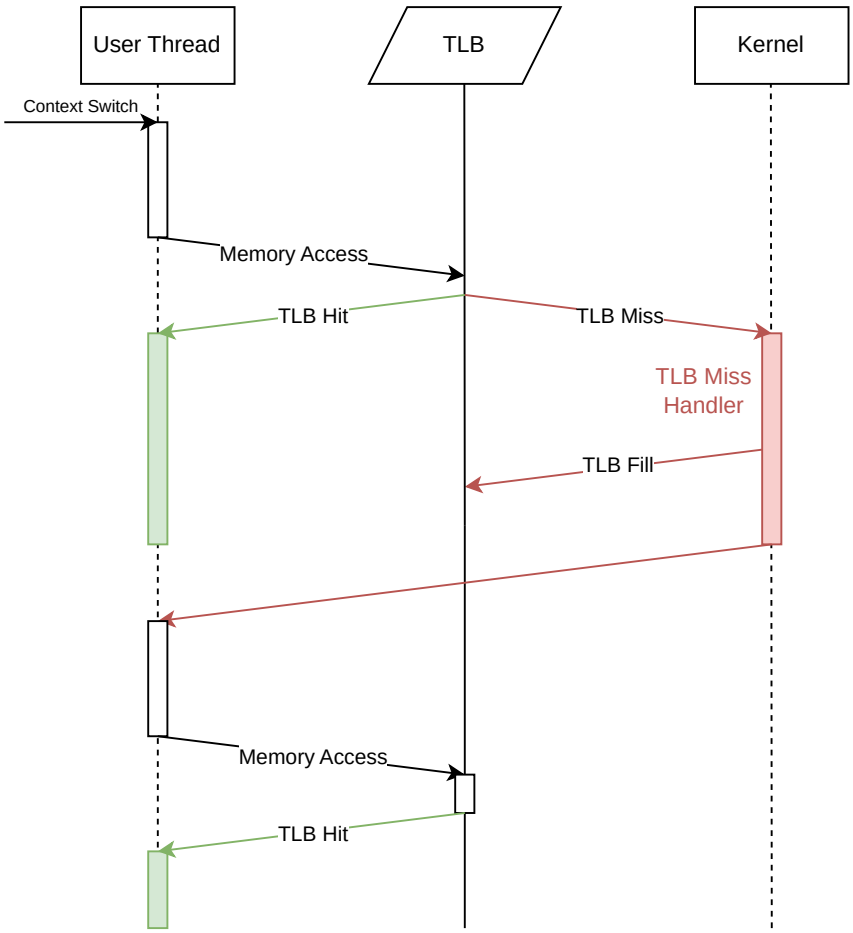

Figure 5.1: Sequence of action in case of a TLB Hit and Miss

make it very hard to debug both first tries at implementing a handler for the exception and the exception throwing code in QEMU itself. And not only would the exception be thrown as soon as virtual memory is activated in `xv6-riscv:kernel/main.c`, the exception would be thrown as soon as exceptions are activated and a memory access happens, because QEMU also uses the `fill_tlb` routine to fill the TLB with direct virtual-to-physical mappings when no virtual memory is used. This speeds up the execution of the dynamically-translated code, as it is able to directly lookup addresses in the TLB using a fast path []. For this step of the implementation, the QEMU memory system emulation must be changed to throw a TLB miss exception when a TLB lookup misses. To keep the system running as normal, this will be done for only one hardcoded address, that is usually not used by xv6. On the xv6 side, we need a user-level program, that accesses the specific address and thus prompts the emulated hardware to throw a TLB miss exception.

5.1.1 Address Selection

The choice for an address to be used for testing the TLB miss exception throwing is easy: As shown in Figure 5.2, the physical memory map of xv6 has an area of “Free memory” that is managed by the physical memory

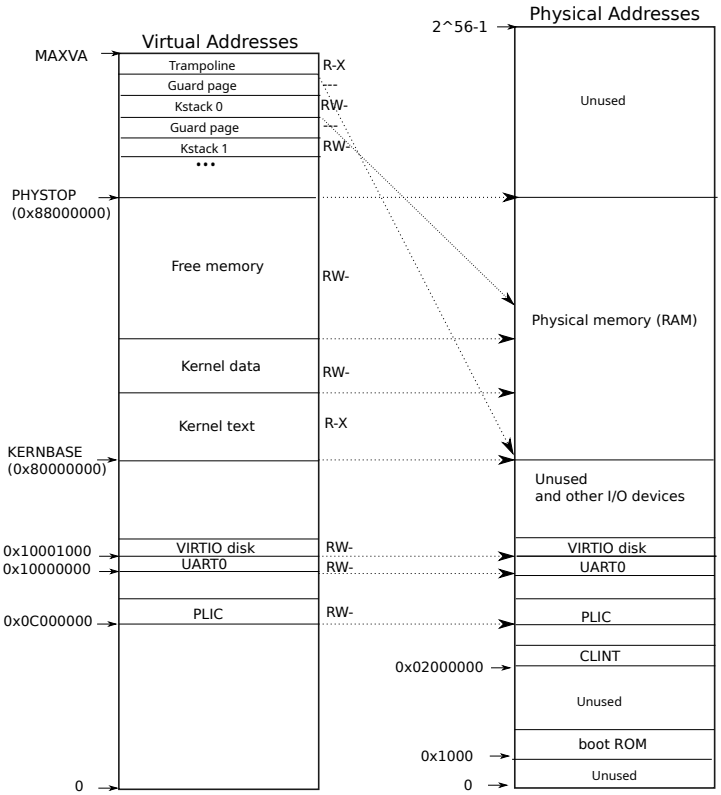

Figure 5.2: The xv6 memory layout and how the kernel virtual address space is mapped on the physical address space. Taken from the xv6 book [CKM11].

allocator. The memory allocator always gives out the next page starting from low to high addresses when a new page is requested by kernel routines. The address `0x84fff000` was chosen as a testing address. Note that the physical memory allocator in `kalloc.c` will actually touch on to this (or any address in the range from `0x80000000` to `0x88000000`), when initializing the linked list used to keep track of the free pages [CKM11].

5.1.2 TLB miss exception in QEMU

To properly test the implementation, the `tlb_fill` function was replaced to throw the TLB_MISS exception for one specified, page-aligned address and to continue on normally for every other address. The implementation is outlined in Listing 5.1. as normal

To draw inspiration on how to implement a TLB miss exception in QEMU, you can take a look at how page fault exceptions are thrown. Whenever a page fault exception is triggered, the TLB is checked first to see if there is a mapping for the input virtual address [24b]. Additionally, RISC-V cores provide the faulting address of the page fault exception in the `mtval` register [Wat+24]. The faulting address will also be necessary for handling the TLB miss exception.

Adding a new exception to the QEMU emulator requires changes at a number of places. In the following, the relevant code locations in the QEMU source [24b] are shown. This may be completely different

Below are…
The following shows…

Listing 5.1: Alternative Implementation for the RISC-V tlb_fill function with a special case to start testing TLB Miss Handler implementations. In line 11, a conditional branch is used to only trigger the exception when neither the Virtual Memory (as set in the `satp` `MODE` field) is bare nor the priviledge mode is the machine mode. If the virtual address is the hardcoded one, a TLB miss exception is thrown, otherwise the original functions is called, which will perform a page table walk to find the mapping.

```
1  bool my_riscv_cpu_tlb_fill(CPUState *cs, vaddr address, int size,
2      MMUAccessType access_type, int mmu_idx,
3      bool probe, uintptr_t retaddr)
4  {
5      RISCVCPU *cpu = RISCV_CPU(cs);
6      CPURISCVState *env = &cpu->env;
7      int mode = mmuidx_priv(mmu_idx);
8      int vm = get_field(env->satp, SATP64_MODE);
9      bool ret = false;
10
11     if(!(vm == VM_1_10_MBARE || mode == PRV_M) &&
12             address == (uint64_t)0x84fff000) {
13         ret = riscv_cpu_tlb_miss_exception(cs,address,size,access_type, mmu_idx, probe, retaddr);
14     } else {
15         ret = riscv_cpu_tlb_fill(cs,address,size,access_type, mmu_idx, probe, retaddr);
16     }
17     return ret;
18 }
```

for other targets, as the exception code is mostly target specific and this implementation only looked at the RISC-V target.

- `target/riscv/cpu_bits.h` contains all CPU-definitions specific to the RISC-V target. There is also a enum called `RISCVException` which contains the number-codes for all RISC-V exceptions. In choosing a appropiate number for a new exception, ~~one should consult~~ the Privileged Architecture Specification [Wat+24]. There are specific exception code ranges that are designated for custom use. E.g. the codes 24–32 and 48–63.
- `target/riscv/cpu_helper.c` `riscv_cpu_do_interrupt` is the target-specific function for triggering interrupts. Here it suffices to add the new exception enum item to the switch case, when the new exception is similar in behavior to exising exceptions. Here the new exception is simply supposed to jump into an exception handler in the kernel. A lot of exceptions like page faults share that behavior.
- Finally, if the exception should be delegatable to supervisor mode or user mode, the n-th bit, with n being the exception code, needs to be set in the `DELEGABLE_EXCPS` definition in `target/riscv/csr.c`. This enables the kernel to delegate the exception to another priviledge level by setting the appropiate bit in the `medeleg` and `sedeleg` CSRs.

The code shown in Listing 5.1 will finally trigger the function shown in Listing 5.2. After executing this function, QEMU will trigger a TLB miss exception as soon as it gets back to the main execution loop [24b].

5.1.3 Exception Triggerer

To properly test the changes introduced to the QEMU emulator, there needs to be some way to trigger a TLB exception. By implementing this as a user-level program, the exception can be triggered using the xv6 shell.

Listing 5.2: Setup-Code for raising a TLB Exception. The `cs->exception_index` variable needs to be set to the custom `TLB Exception` enum value. The `env->badaddr` variable will end up in the `mtval` register. The address will be page-aligned first, by zeroing out the lowest 12 bits. This is used to encode the `mmu_idx` into the faulting address. Why this is necessary is explained in Section 5.2

```
|
|   static void raise_tlb_exception(CPURISCVState *env, target_ulong address,
|                           MMUAccessType access_type,
|                           /*unnecessary?*/ bool pmp_violation,
|                           bool first_stage, bool two_stage,
|                           bool two_stage_indirect, uint8_t mmu_idx) {
|       CPUState *cs = env_cpu(env);
|
|       cs->exception_index = RISCV_EXCP_TLB_MISS;
|       env->badaddr = ( address & ~( (1 << 12) - 1)) | mmu_idx;
|       env->two_stage_lookup = two_stage;
|       env->two_stage_indirect_lookup = two_stage_indirect;
|   }
```

Adding a new user-level Program to xv6 only needs you to add a new `.c` file to the user subfolder and to add the name of the generated binary (name of `.c` file prefixed with a _) to the list of user binaries in the makefile. The new `.c` file only needs a `main` function and should also include the `user.h` file to gain access to some preimplemented function and system call wrappers [24a].

The final exception triggerer may look something like this:

,

Listing 5.3: Exception Triggerer Trying to load from a hardcoded address prompts the emulated hardware to trigger a TLB miss exception.

```
|   #include "kernel/types.h"
|   #include "user/user.h"
|
|   void do_tlb_exc(void) {
|       __asm__("li s2, 0x84fff000\n\t \
|              lw s4, 0(s2)\n\t");
|       register int *foo asm ("s4");
|       printf("%x\n", foo);
|       return;
|   }
|
|   int main(int argc, char *argv[]) {
|       do_tlb_exc();
|       //exit(0);
|   }
```

The program first loads the hardcoded address into a register and then tries to load a word from this address. If the implementation of the `TLB miss exception` was done correctly, the process will trap to the kernel and the kernel will print out an error message, as it does for all exceptions that either have unknown exception numbers or do not have a exception handler implemented [CKM11]. If the exception was not properly implemented, the kernel would report a load page fault exception.

5.2 Exception Handling and TLB Writing

Now that the new *TLB miss exception* can be triggered by a user-level program, there needs to be an exception handler in the kernel that will create virtual-physical mappings and add them to the TLB. This section will first go into a general description to add new CSRs to the RISC-V QEMU emulation and will then elaborate on the specific implementation for the TLB CSRs. The section ends with the implementation of the exception handler.

5.2.1 Adding CSRs to RISC-V/QEMU

Following code locations are relevant for CSRs in the RISC-V/QEMU emulation source code [24b]:

- `disas/riscv.c` Contains a big switch case with all the CSR number to CSR name mappings. Name and number of new CSRs need to be added there.
- `target/riscv/cpu_bits.h` contains definitions for all CSR numbers. While it is not strictly necessary to add another definition for new CSRs here, readability and maintainability of the code increases if a more descriptive definition name is used instead of a magic constant.
- `target/riscv/cpu_cfg.h` contains a structure called `RISCVCPUConfig`. Every emulated RISC-V hart has this structure to expose all the extensions that the hart supports. The structure has a boolean flag for every extension that is currently supported by the emulator. New extensions should get their own flag in this struct. Similar entries also need to be added to the `isa_edata_arr` and `riscv_cpu_extensions` arrays in `target/riscv/cpu.c`.
- `target/riscv/csr.c` contains the implementation for all CSRs. The `riscv_csr_operations csr_ops[]` array is essential for adding callback functions to CSR numbers. For every new CSR, a struct of the type `riscv_csr_operations` must be added to that array using the CSR number as an index. This struct is comprised of multiple function pointers, which deal with
  - Checking if the hart implements the CSR
  - Reading from the CSR
  - Writing to the CSR
  - Combined read/write
  - 128 bit read/writes

As previously mentioned, the CSRs have some index ranges for new, custom CSRs. For the implementation of TLB write CSRs, the indexes `0xBEE` and `BFF` have been selected. Using these constants and the steps above, two new instructionscan be realized.

5.2.2 CSR Callback Implementation

Apart from the steps above to

Apart from the above mentioned steps to add new CSRs to the emulator, the main logic of the implementation is in the callbacks referenced in `target/riscv/csr.c`. The implementation of these callbacks is strongly dependent on the structure of the data that is written to the CSRs. Fundamentally, these callbacks act as a bridge between the exposed ISA and the implementation of that instruction set in software. As previously mentioned, two new CSRs will be needed to implement the TLB-writing. ~~The implementations of the write-callbacks look as follows:~~

The write-callback implementations look like this:

```
1  static RISCVException write_tlbh(CPURISCVState *env, int csrno, target_ulong new_val)
2  {
3      env->tlbh = new_val;
4      return RISCV_EXCP_NONE;
5  }
```

nothing more than store

The implementation of the `tlbh` CSR write, does ~~not do anything else but saving~~ the value that is written to it ~~to~~ in the environment of the CPU. This is because the theory specifies, that the TLB entry will only be written to the TLB when the write to the `tlbl` CSR has succeeded.

```
1  static RISCVException write_tlbl(CPURISCVState *env, int csrno, target_ulong pte)
2  {
3
4      target_ulong tlb_size = TARGET_PAGE_SIZE;
5
6      CPUState *cpu = env_cpu(env);
7      vaddr addr = env->tlbh;
8      hwaddr paddr = ((pte & ~(PTE_RESERVED)) >> 10) << 12;
9
10     int mmu_idx = addr & (tlb_size - 1);
11
12     int prot = pte & (PTE_R | PTE_W | PTE_X | PTE_V );
13
14     addr &= ~(tlb_size - 1);
15     paddr &= ~(tlb_size - 1);
16
17     tlb_set_page(cpu, addr, paddr, prot, mmu_idx, tlb_size, false);
18
19     env->tlbh = 0;
20     env->tlbl = 0;
21
22     return RISCV_EXCP_NONE;
23 }
```

The value written to the `tlbl` CSR adheres to the same format as the RISC-V Sv39 PTEs (As shown in section 2.5.1).

To get the page-aligned physical address and to get rid of the access bits stored in the lower 10 bits, the value will first be right-shifted by ten and then left shifted by 12 bits. The top most bits are specified to be all zero, as explained in the fundamentals chapter [Wat+24].

In line 10, the `mmu_idx` is extracted from the lowest 11 bits of the virtual address. This is necesarry, because QEMU uses up to 16 different MMU modes with dedicated TLBs [24b]. Whenever QEMU performs a TLB lookup, it does so in a specific MMU mode. This MMU mode is clear when a TLB entry is retrieved, it is however not clear when a TLB entry is written. To still fill the correct TLB, the `mmu_idx` is transfered to the exception handler as part of the faulting address in `mtval` and then back to the emulator via the TLB write CSRs.

The following lines deal with extracting the protection bits from the PTE and with page-aligning the virtual and physical addresses. Finally, a preexisting QEMU function is invoked to add a new entry to the emulated TLB and the CSR values are cleared. The return value `RISCV_EXCP_NONE` indicates that nothing out of the ordinary happend.

This is all that needs to be done to add CSRs for TLB filling to the QEMU RISC-V emulator.

5.2.3 TLB miss exception Handler

With capabilities to write TLB entries in place, an effective exception handler can be implemented.

xv6 Machine-Mode Trap Handler The xv6 machine-mode trap handler only deals with the timer interrupt. All other interrupts and exceptions are delegated to the supervisor mode. This allows the trap handler to be very small and very specific to the timer interrupt [CKM11]. It thus only needs to store two registers to memory to make enough room in the register file to reset the timer and invoke the scheduler. Adding another trap to be handled adds more complexity, as the trap number needs to be checked and the code needs to branch to the right routines.

But it can also be completely avoided to touch the timer code at all. xv6 uses trap vectoring mechanism in *direct* mode. In direct mode, all traps jump to the same address. Using the *vectored* mode makes all *exceptions* jump to the address set in the `mtvec BASE` field but all *interrupts* are set the program counter to `BASE` plus four times the interrupt cause [Wat+24].

as little disruption to

So to add machine mode exception handlers with disrupting the existing code ~~as little~~ as possible only requires changing the trap vectoring mode to `vectored` mode and moving the timer interrupt code to the correct offset.

```
1 -   w_mtvec((uint64)timervec);
2 +   w_mtvec((uint64)mtvec_vector_table | 0x1);
```

modifications to change

The changes ~~for changing~~ the mode are trivial. Only the one bit indicating the `vectored` mode needs to be set. Placing the timer vector at the correct offset from the default trap handler vector can be achieved by filling the space between the base address and the timer vector with no-operations. The same could be achieved with linker directives. The interrupt number for the timer interrupt is `0x7`, so thats puts the timer interrupt vector at a positive `0x1c` offset from the base address.

RISC-V provides 4 bytes for every interrupt request (IRQ) and the default trap handler. This is not enough to implement proper trap handling, so these 4 bytes are typically spent jumping to a more elaborate trap handling routine.

Now any machine-mode exception (that is not delegated to a lower-priviledge mode) sets the program counter to the address of the `mtvec_vector_table`

Listing 5.4: Vectored Trap Handler Routine

```
mtvec_vector_table:
IRQ_0:
        j default_exception_handler
        nop
IRQ_1:
        j default_vector_handler
        nop
IRQ_2:
        j default_vector_handler
        nop
IRQ_3:
        j default_vector_handler
        nop
IRQ_4:
        j default_vector_handler
        nop
IRQ_5:
        j default_vector_handler
        nop
IRQ_6:
        j default_vector_handler
        nop
IRQ_7: # timer handler
        j timervec
```

label. The next jump brings the execution into the default exception handler. The timer interrupt will jump directly to the `IRQ_7` label.

Store/Load Sandwich The idea behind the *softtlb* design is to not use as many of the five memory references modern architectures need for a page table walk [Int17]. But for a first proof of concept, it is helpful ~~to~~ not to have to think about which registers to use and ~~to~~ not to optimize away all unnecessary unnecesarry memory references before the design even works. Therefore That is why the first thing the default exception handler will do is to store the current state of the register file. This is essential so that, once returned to the context that triggered the TLB miss, execution can continue as if nothing happened. After storing the state and running the exception handler, the state must again be loaded be reloaded into the registers.

Most of the load/stores have been omitted to not flood the listing. In between the store and load blocks, the `machine_default_exception_handler` C function is called. This is where the implementation differentiates between the different exception numbers and calls exception handlers specific to the exceptions. In xv6, all other exceptions but the new TLB miss exception are handled in supervisor mode, so there are no handlers for other exceptions.

Finally, the `tlb_handle_miss()` functions is called with the faulting address and the `satp` register value as arguments. The initial implementation simply calls the new `tlbh` and `tlbl` CSR instructions with and then returns.

Note that this implementation does not yet use the format for the `tlbh` and `tlbl` CSRs specified in the theory section . Merely the faulting address and the intended mapping are written to the registers.

5.2.4 Testing

Listing 5.7 already shows a part of the setup to test that the mapping actually worked. When the mapping worked, it should be possible to

Listing 5.5: Store/Load Sandwich

```
default_exception_handler:

    csrrw a0, mscratch, a0
    sd sp, 40(a0)

    ld sp, 48(a0)
    csrrw a0, mscratch, a0

    # make room to save registers.
    addi sp, sp, -256

    # save the registers to the hart stack
    sd ra, 0(sp)
    sd gp, 16(sp)
    ...
    sd t5, 232(sp)
    sd t6, 240(sp)

    addi s0, sp, 256 # Set Frame Pointer

    # ??
    # csrr t0, sscratch
    # sd t0, 8(sp)

    # TODO Pass args to fx

    # csrr a0, mtval
    # csrr a1, satp

    call machine_default_exception_handler


    # restore registers from hart stack
    ld ra, 0(sp)
    ld gp, 16(sp)
    ...
    ld t5, 232(sp)
    ld t6, 240(sp)

    addi sp, sp, 256

    csrrw a0, mscratch, a0

    sd sp, 48(a0)
    ld sp, 40(a0)

    csrrw a0, mscratch, a0
    mret
```

Listing 5.6: Exception Switch-Case Statement

```
    void machine_default_exception_handler() {
        switch (r_mcause()) {
            case NONE:
            case INST_ADDR_MIS:
            case INST_ACCESS_FAULT:
            case ILLEGAL_INST:
            case BREAKPOINT:
            case LOAD_ADDR_MIS:
            case LOAD_ACCESS_FAULT:
            case STORE_AMO_ADDR_MIS:
            case STORE_AMO_ACCESS_FAULT:
            case U_ECALL:
            case S_ECALL:
            case INST_PAGE_FAULT:
            case LOAD_PAGE_FAULT:
            case STORE_PAGE_FAULT:
            default:
                unhandled_exc(r_mcause(), r_mtval());
                break;
            case TLB_MISS:
                tlb_handle_miss(r_mtval(), r_satp());
                break;
        }
    }
```

read a value from the mapped page. The user-level exception triggerer already tries to do just that. With the mapping working, the output of the program looks as follows:

Listing 5.7: Simple TLB miss exception handler for a single fixed address

```
void tlb_handle_miss(uint64 addr, uint64 usatp) {
  //uint64 *paddr = kalloc();
  w_tlbh(addr);
  w_tlbl((uint64)addr+(uint64)0x1000);
  return;
}
```

The value `0xDEADBEEF` that was read from the mapped page is not a coincidence: For testing purposes, the page `0x85000000` was filled with `0xDEADBEEF` bit patterns.

The address for the physical ”testing” page was deliberately chosen to be `0x85000000` and not `0x84fff000` (like all the hardcoded testing addresses) to make sure that there is no accidental physical mapping happening.

5.3 Software Page Table Walk for all Addresses

After implementation step 2, we now have a modified QEMU RISC-V emulator, which will throw the new TLB miss exception when the TLB misses. We have also extended the xv6 operating system to catch said exception and to create a virtual-physical mapping for the faulting address. The mapping is created by writing new TLB entries using newly implemented TLB CSRs.

Currently, all of this only works for one specific ”testing” address. This step now elaborates on how the scheme is extended to be used for all addresses. To not change too much at once, the page table walk that would typically be done in hardware is implemented as part of the TLB miss exception handler.

5.3.1 TLB miss exception for all Addresses

The changes introduced to QEMU are simple: Essentially, only the condition checking if the given address is the ”testing” address needs to be changed. The final implementation looks as follows:

```
    bool my_riscv_cpu_tlb_fill(CPUState *cs, vaddr address, int size,
                               MMUAccessType access_type, int mmu_idx,
                               bool probe, uintptr_t retaddr)
    {
        RISCVCPU *cpu = RISCV_CPU(cs);
        CPURISCVState *env = &cpu->env;
        int mode = mmuidx_priv(mmu_idx);
        int vm = get_field(env->satp, SATP64_MODE);

        //No address translation in m-mode
        if(vm == VM_1_10_MBARE || mode == PRV_M) {
            int tlb_size = 4096; //TODO Get from PMP?
            hwaddr pa = address;
            int prot = PAGE_READ | PAGE_WRITE | PAGE_EXEC;
            tlb_set_page(cs, address & ~(tlb_size - 1), pa & ~(tlb_size - 1),
                         prot, mmu_idx, tlb_size, true);
            return true;
        }
        riscv_cpu_tlb_miss_exception(cs,address,size,access_type, mmu_idx, probe, retaddr);
        return true;
    }
```

Note that in comparison to the previous implementation (as shown in figure 5.1), the call to the original `riscv_cpu_tlb_fill` function is not called anymore. Either a TLB miss exception is raised, or the TLB is filled with an identity mapping if either the current priviledge mode is the machine mode or the virtual memory system is configured to use identity mappings only [Wat+24].

5.3.2 Software Page Table Walk

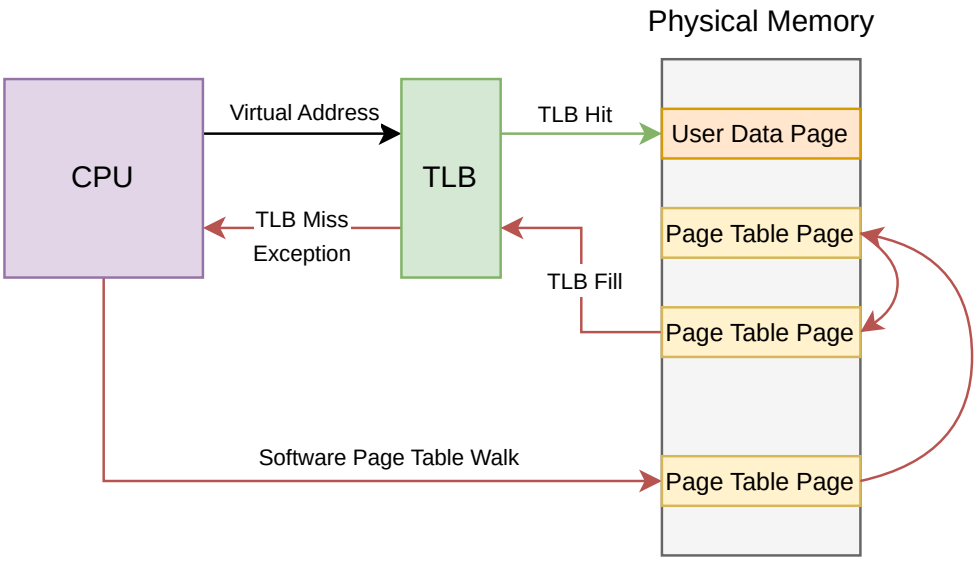

Figure 5.3: Instead of transfering control over to the MMU to fill in the missing mapping, the TLB miss exception invokes a software page table walker

Now that the TLB miss exception is thrown for every faulting address, it also needs to be handled for every address. Even though the goal is to explore alternatives to virtual memory using page tables, we will use a software implementation of the hardware page table walk. This is to verify that the general setup including exception invokation, exception handling and TLB writing works as expected. Listing 5.8 shows the implementation.

If the `walk_pt()` function returns 0, the virtual address lead to no valid PTE. This is a page fault and would raise a page fault fitting the faulting instructions type of access [TB14]. As xv6 does not handle page faults other than by killing the program [CKM11] this paper will not go into more detail of raising and triggering page faults for the `softtlb` design.

terminating of

not provide further revarding the generation

Listing 5.8: TLB miss exception Handler with Page Table Walk The `walk_pt()` function walks the Sv39 page table with the base address encoded in the `satp` register. If a PTE with the valid bit set is found, the function returns the address encoded in the PTE. Otherwise, the function returns `0`.

```
#define SATP2PA(satp) ((satp << 12) & ~(0xffull << 44))

pte_t *
walk_pt(uint64 satp, uint64 va)
{
  uint64 *pt = (uint64*)SATP2PA(satp);
  if(va ≥ MAXVA)
    panic("walk");
  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pt[PX(level, va)];
    if(*pte & PTE_V) {
      pt = (uint64*)PTE2PA(*pte);
    } else {
      return 0;
    }
  }
  return &pt[PX(0, va)];
}

void tlb_handle_miss(uint64 addr, uint64 satp) {
    w_tp(r_mhartid());
    pte_t *pte = walk_pt(satp, addr);
    w_tlbh(addr);
    uint16 prot = PTE_R | PTE_W | PTE_U | PTE_V;
    w_tlbl(*pte | prot);
    return;
}
```

5.4 Segmented Memory Design using software-defined TLB Filling

After step 3, xv6's virtual memory system runs completely without using the (emulated) page table walking state machine: Instead of invoking the page table walker, a TLB miss triggers a exception. This exception is handled by the operating system kernel and a TLB entry for the faulting address is generated. Yet at this point, we have not really won anything. We only moved the page table walking to the software side, which would certainly be slower at traversing the page table than a real MMU [JM98c]. And that even if we ignore the expensive context switch that needs to be performed when the exception handler is invoked.

While we did not gain any performance, there is now the opportunity to modify the virtual memory system completely in software. We do not need to adhere to the rigid structures [TB14] of the memory management hardware anymore. We gained a great deal of flexibility.

The rest of this section explores a first idea of how a table-based virtual memory system can be replaced by essentially a simple function that does not require expensive memory accesses.

The initial idea, as presented in the theory chapter, is a segmented memory design. It splits the total available memory into parts of equal size, also called segments. Each new process acquires one of those segments, if there are any left. If none are left, process creation fails.

5.4.1 Address Spaces

As with segmentation, the available physical memory is split into evenly sized portionsfrom now called address space. When a new process is started it tries to acquire an address space by checking whether there is a free one and then claiming that free address space. The number of address spaces will thus limit the number of concurrently running programs.

To keep track of which process acquired which address space, the kernel keeps an integer array of process IDs. This array has a size equal to the number of address spaces (NAS). For a newly created process to acquire an address space, the kernel will (synchronized by a spinlock [CKM11]) iterate through the array to find a an array slot with the value 0. It then writes the process ID into that array. The process creation fails if there are no more array slots set to 0. The address space ID (ASID) acquired by the process is also written to the processes control block (PCB).

NAS is set via a preprocessor definition and thus set at compilation time. More elaborate implementations could also allow dynamic resizing of address spaces to provide space for more processors.

Using the ASID, the address range for a process can be calculated very easily. The macros in listing ?? are used in different parts of the code to calculate different addresses related to an Address Space.

```
1 | #define MAX_AS_MEM ((((PHYSTOP - (PGROUNDUP((uint64)end))) / (NAS + 1)) >> 12 )<< 12)
2 | #define AS_START(asid) (PGROUNDUP((uint64)end) + ((asid) * MAX_AS_MEM))
3 | #define AS_N(index, asid) (PROC_START(asid) + (index * (0x1000)))
4 | #define AS_END(asid) ((PGROUNDUP((uint64)end) + ((asid + 1) * MAX_AS_MEM))-0x1000)
```

5.4.2 Mapping function for Segmented Memory

The ASID of a process is the foundation for calculating the virtual to physical mapping in the exception handler. The only other information required is the faulting address.

The ASID can be used to calculate the actual physical address at which the Address Space starts. The faulting address, which would, according to the xv6 process memory model, be an address in the low end of the virtual memory space, will provide the offset into the Address Space.

Listing ?? shows the implementation for the TLB miss exception handler.

5.4.3 Special Mappings

xv6 needs a trampoline page mapped into the address space of the process for system calls to work properly (see Chapter). In both the kernel and in the user processes, it is mapped to the highest possible, virtual page. If a address on this page is encountered, the handler will map it to the physical address of the trampoline page. The handler takes the same approach for the TRAPFRAME page set just below the trampoline.

```c
#include "defs.h"
#include "param.h"
#include "types.h"
#include "riscv.h"
#include "memlayout.h"

#define SATP2ASID(satp) ((satp << 4) >> 48)

extern char trampoline[];
paddr get_mapping(vaddr addr, uint16 asid) {

  //special case for trampoline, which is in every address space at the same address
  // (except for the physical one!)
  if(PGROUNDDOWN(addr) == TRAMPOLINE) {
    return (uint64)trampoline;
  }
  if(asid == 0) {
    //Kernel -> direkt mapping
    //TODO special mappings
    return addr;
  } else {
    //mappings for processes
    //TODO trapframe and trampoline mappings
    if(PGROUNDDOWN(addr) == TRAPFRAME) {
      return TRAPFRAME_FROM_ASID(asid);
    }

    //Base case
    uint64 vpage = PGROUNDDOWN(addr);
    if(vpage >= MAX_AS_MEM - (0x1000 * 4)) {
      //Error, address out of range
      panic("tlb_manager: address to high!");
    }
    return AS_START(asid) + vpage;
  }
  return 0;
}

void tlb_handle_miss(vaddr addr, uint64 satp) {
  w_tp(r_mhartid()); //fix problems with locks based on cpuid()

  //TODO Locks!
  //TODO buffering printer, only prints completed lines
  //uint16 mmuid = addr & 0xfff;
  vaddr addr_no_mmuid = addr & ~0xfff;
  //printf("tlb_manager: addr=%p satp=%p mmuid=%d\n", addr_no_mmuid, satp, mmuid);
  paddr pa = get_mapping(addr_no_mmuid, SATP2ASID(satp));

  w_tlbh(addr);
  //TODO all access rights for now
  uint16 prot = PTE_R | PTE_W | PTE_X | PTE_V | ((SATP2ASID(satp) != 0) ? PTE_U : 0);
  w_tlbl(PA2PTE(pa) | prot);
  return;
}
```

5.4.4 Further Changes to the OS

Other than the exception handler, other parts of the OS need to be changed to fit the new memory management scheme.

Physical Memory Allocator Demand paging of the virtual memory system was the main customer of the physical memory allocator. But some modules like the virtio disk and pipes also rely on being able to access physical pages. So it is not possible to get rid of the physical allocator completely. Here, its managed memory is merely reduced by assigning it the memory area of the Address Space with index 0.

`copyin()` `copyout()` The copyin() and copyout() functions are used to copy data between user space and kernel. They traverse the page table tree to acquire the physical location for the data to be either copied in or copied out. Both of those function rely on the `walkaddr()` function to

functions

do the walking for them. The implementation can be changed to call the `tlb_manager.c:walk_addr()` function instead.

Process Loading

sbrk() The `sbrk()` system call is used by the program to increase the size of the programs data segment. Typically, this call would demand a new page from the physical memory allocator and then add a new PTE to the page table. This system call normally returns the previous program break if it was successful (so basically a pointer to the new program area) and `-1` if it was not successful. Instead this design checks if the program size would increase to be bigger than the maximum program size and then return `-1` if it is and the previous program break otherwise.

5.5 Debugging

When making simultaneous changes to both the Qemu emulator and xv6, it can be difficult to determine where bugs originate from. This section first describes how to debug Qemu and xv6 individually and then outlines a combined debugging process. This can be useful, for example, when trying to trace the code path between software and emulated hardware. This is not a guide on using GDB but rather a collection of specific tricks that were useful for debugging the code in this work. Perhaps they will also be helpful in other scenarios.

5.5.1 xv6

The Makefile in the xv6-riscv source repository has a rule `qemu-gdb` to start xv6 in Qemu with a GDB server. In another terminal, you can then start GDB with the binary you want to debug and connect to the gdbstub using `target remote:<port>`. A comment in the Makefile suggests that this rule is intended for user-mode debugging, but in reality, nothing prevents you from also debugging the kernel, which is particularly useful when making changes to the memory system.

Debugging user mode When debugging a user-mode program, you often want to start at the `main` function. If you set a breakpoint at this function, a breakpoint is set at a rather low virtual address. If you set the breakpoint before even starting the kernel, you might stop immediately after starting. This happens because you're currently in the Qemu boot ROM, which is mapped by Qemu into the emulated memory layout in the range `0x0 - 0x1000`. The debugger does not differentiate between virtual and physical addresses and simply looks at what the program counter (PC) contains.

Debugging the kernel After replacing the virtual memory system, an interesting effect was observed when debugging the kernel: As soon as the `satp` register is set in the `kvminithart()` function, GDB can no

longer display the code at the current address (or any following ones). This is due to the value of the satp register. After the VM system was redesigned, it no longer contains the PPN. Only the MODE and ASID fields are relevant for the new system. There is no longer a page table that the PPN could point to. This behavior suggests that the gdbstub implemented by Qemu performs a hardware-emulated page walk to obtain the physical addresses and, consequently, the code. This, of course, does not work without a page table. One way to debug the kernel properly in GDB again would be to rebuild the kernel page table as before. However, since all the kernel mappings are direct mappings, it is sufficient to place a PTE for the address 0x80000000, where the kernel code begins, behind the PPN in satp.

A whole page was allocated in the code as follows: However, it would

```
1 |   uint64 *pt = (uint64*)kalloc();
2 |   for(uint64 i = 0 ; i < 512; i++) {
3 |     pt[i] = 0xffffffffffffffff;
4 |   }
5 |   pt[2] = ((0x80000000 >> 2)| PTE_V | PTE_X | PTE_W | PTE_R);
```

also be sufficient to allocate only the space for a single PTE. The PPN must be set so that at least the entry at index 2 is present. Interpreting the address 0x80000000 as a virtual address yields a value of 2 for the VPN[2] field. By marking this top-level page table entry as valid, the memory at this address is treated as a 1 GB superpage. This is enough to cover the kernel code, and the code can once again be viewed as usual during debugging.

Verifying Addresses in Binaries Implementing the vectored trap vectoring mode required moving the interrupt handler code for the timer interrupt to a specific offset from the address specified in the mtvec BASE field. To verify the correct placment of the label, the unix nm tool is very useful: It lists the symbols of a object file and their addresses. Looking back at the code in listing 5.7, the label IRQ_7 is supposed to come exactly 0x1c bytes after the mtvec_vector_table label.

The following call confirms the correct placement of the symbols:

```
1 |   $ nm kernel/kernel | grep 'mtvec_vector_table\|IRQ_7'
2 |   00000000800092cc t IRQ_7
3 |   00000000800092b0 T mtvec_vector_table
```

5.5.2 QEMU Monitor

Mit dem Qemu monitor lassen sich allerlei informationen über die laufende emulation anzeigen lassen. Besonders interessant für diese Arbeit wäre der monitor für die aktuell im emulierten TLB enthaltene Mappings. Gerade für RISC-V war dieser allerdings nicht verfügbar.

DEUTSCH?

5.5.3 QEMU Record/replay

QEMUs Redord/replay feature allows a deterministic replay of a machines execution by recording all non-deterministic events that happend. This can be especially helpful to investigate bugs that occur as a result of an interrupt or require special preconditions or timings to occur. With the Record/replay feature, it is easy to recreate the situation in which a bug or unexpected behavior occured. For this project, especially the mouse and keybord input and the hardware clock are interesting non-deterministic events that influence the execution of the system [].

5.5.4 Double GDB Setup

6 Evaluation

The previous chapters show the successful and transparent replacement of the xv6 virtual memory system with a mapping function based TLB miss handler. However, the design and implementation have a number of shortcomings. This chapter will discuss these shortcomings of the presented mapping function in light of typical requirements to virtual memory systems. It will also attempt to qualitatively analyze the cost of the exception handler implementing the mapping function and compare it to conventional page table based schemes. The chapter will also discuss some shortcomings of the implementation and give some ideas on how to alleviate the problems.

6.1 Software-managed TLBs

Managing TLBs in software grants a lot of flexibility, but this flexibility does not come for free. While the general trade-offs between software and hardware TLB managements have already been discussed in the Fundamentals chapter 2, this section will go into more detail of the trade-offs of this specific design and implementation.

6.1.1 Context Switch

The fast TLB miss handler presented by Gernot Heiser for the L4/MIPS implementation only uses three registers: Two of those registers are reserved for the kernel and the other one has to be saved in memory to later be restored [Hei99]. While the processor still has to run the instructions, this is only minimally invasive for the state of the kernel.

This implementation performs a complete save of the processor state, including most of the 32 registers that RISC-V offers. This allows running C code from the exception handler without having to think about what registers are mangled by the code, which is very useful for trying different implementations. The trade-off here lies between ease of programmability and performance.

6.1.2 Pipeline Flush

Exception handlers not only change the state of the processor that is exposed to the programmer, they also disturb the transparent state consisting of the instruction pipeline and the reorder buffer. Running instructions from the context of the exception handler (coupled with RISC-V handling memory faults precisely [Wat+24]) requires a flush of both reorder buffer and instruction pipeline [JM98a]. Instructions that have already been partially executed in the pipeline have to be restarted, imposing extra cost on the software handler [JM98c].

6.1.3 Exception Handler

The key difference of this exception handler and exception handlers of other software-based virtual memory designs is that this one does not need to access memory to generate mappings (apart from the register state save). Looking back at listing **??** it is apparent that there still are some memory references: Mostly the function call of the `get_mapping()` function would generate code that writes to the stack. But the function call is only for readability of the code and could also be inlined. Otherwise the implementation is based on calculation for determining the memory offset from the ASID and bitwise operations. This is an advantage over radix page table systems that may need up to five memory references in contemporary architectures [Int17].

6.2 Segmented Memory Design

The segmented mapping function design has some fundamental problems that restrict

Fragmentation

Limited Process count

6.3 Memory System Requirements

For a quantitative assessment of the design, the functional requirements to virtual memory system from chapter 2 are revisited.

Address Space Protection / Isolation Since all memory accesses from virtual addresses go through the TLB, a proper isolation of processes in the TLB exception handler guarantees isolation of processes. ASIDs are used as the foundation of address calculation to provide each process with a distinct physical memory space. Synchronization on the data structure managing ASIDs makes sure that no two processes can have the same ASID while both are alive.

Large Address Space Segmentation restricts the maximum size each process can have. One could implement dynamic resizing of address spaces or allow processes to hold multiple ASIDs to allow processes to increase their memory over the statically determined maximum limit. That would however increase the complexity of the mapping function. Another idea would be to have address spaces with different (predetermined) sizes that could then be assigned according to a programs memory demands.

Superpages The current state of the design does not really deal in pages but rather in complete address spaces. As such, super pages are not supported.

Flexibility Pages in the virtual memory space can be placed anywhere as long as they are within the maximum address space size. And they must not clash with the heap space that will always be expanding from low to high addresses.

Sparsity Sparsity in page table based virtual memory systems is about efficiently supporting huge address spaces with only a small numbers of pages being actually used. The bookkeeping should use as little additional memory as possible.

The stateless design requires no tables to store the bookkeeping for the mapping - there is no bookkeeping. However, huge address spaces are not supported and the internal fragmentation if very high.

Swapping One of the most important tasks of a virtual memory system is to automate the swapping of memory pages between main memory and secondary storage. xv6 does not provide a good starting point to experiment with new virtual memory systems that also fulfil that requirement, as xv6 does not implement page swapping. Programs are completely held in memory and also loaded completely into memory on execution [CKM11]. As such, page table faults of any kind will result in xv6 killing the process.

6.4 Cost Analysis

A naive approach to measuring the performance of the virtual memory system is to implement a xv6 user-level program that will result in a number of TLB misses and thus prompt the triggering of the TLB refill mechanism. The execution time of that program can be measured and the run first with the QEMU-emulated MMU, the software page table walker in the TLB miss handler, and the mapping function. It would however be hard

In the implementation, the cost introduced by memory accesses is very high, because the whole state of the register file is saved to memory. That is an unacceptable cost for a operation that is on the critical path of every

TLB miss. It is however possible to omit most of these memory accesses in favor of a light-weight TLB miss handler similar to the fast TLB miss handler in [Hei99]. The main difference being that the TLB miss handler in [Hei99] performs a page table walk and has thus access main memory. An optimized implementation of the mapping function can work with only a small number of register.

To omit all memory accesses from the handler, some registers may be reserved to be only used by kernel code, as MIPS does it with two registers. However, compilers would have to be made aware of that. Depending on the application, reserving registers may still result in a loss of performance because the availability of registers is imperative to the compilers ability to optimize code [EH13].

Even when all memory references are eliminated from the handler code, it still impacts both the pipeline and the reorder buffer [JM97]. This can not happen with a hardware-based TLB refill mechanism, as it would typically freeze the pipeline and not touch the reorder buffer [BL17]. Instructions independent from the memory access could still continue execution while the TLB miss is being serviced [JM98b].

[JJ01] proposes a solution that promises to alleviate the cost related to the calling of software interrupt handlers that disturb the pipeline. The paper describes a method to integrate exception handlers directly into the reorder buffer, avoiding the need to flush the pipeline when handling a precise interrupt. The exception handler for TLB misses using a mapping function can be very small and may thus benefit from the idea.

6.5 Potential Implementation Improvements

TLB Flushing xv6 always flushes the complete TLB on process switch [CKM11]. However, when switching from a process with ASID n to a process with ASID m, it suffices to flush the TLB entries belonging to ASID n only. The `sfence.vma` instruction already allows for such fine-grained control of the TLB. The memory fencing effect of the instruction would not change that way [Wat+24].

Finer TLB Control As shown in chapter 4, MIPS [16] provides a very fine control of its TLB entries. Such fine control would allow the TLB handler to implement custom replacement policies which could adapt the system to current work loads and biasing cache replacement [Par+22].

7 Conclusion

This paper presented the theory behind and the modifications of the xv6 operating system running on the QEMU/RISC-V platform to transform the fundamental control flow of the virtual memory path. The result is a simple platform forming a foundation to start experimenting with mapping functions that replace the typical page table structure used by commodity hardware.

The paper also presented a mapping function implemented on that platform, realizing a (quite restricted) virtual memory system, that works completely without using any bookkeeping of mappings. The changes are completely transparent to the system call interface of the operating system and thus do not require changes to programs running on the xv6 operating system.

However, the evaluation showed clear deficits when compared to a conventional VM system regarding functional requirements to a memory system. Also, the mapping function has not been evaluated quantitatively and might suffer from performance issues on specific workloads. Also, the general problems with software-managed VM systems still persist in the mapping function design.

As this thesis provides a platform for further testing with function based VM systems, more work might go into further research on how these functions could look like and how the functional requirements to VM systems could be realized with this approach. If some functions prove effective at providing proper virtual memory support, hardware designs to implement such functions based on simple arithmetic operations may be examined. Hardware support for VM mapping functions would alleviate costs associated with context switches and pipeline disruptions.

Exploration on good functions for memory mapping could start by looking into hash functions used by current inverted paging approaches, which already show attributes which are very useful for distributing pages across a smaller physical space while keeping spatial locality and combating collisions.

A Unused paragraph dump

References

[] *A Deep Dive into QEMU: TCG Memory Accesses*. QEMU internals. https://airbus-seclab.github.io/qemu_blog/tcg_p3.html (visited on 07/16/2024) (page 30).

[BCR] TW Barr, AL Cox, and S Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 48–59 (pages 8, 15, 16).

[BL17] A Bhattacharjee and D Lustig. *Architectural and operating system support for virtual memory*. Morgan & Claypool Publishers, 2017 (pages 1, 50).

[CKM11] R Cox, MF Kaashoek, and R Morris. Xv6, a simple Unix-like teaching operating system. 2011 (pages 13, 20, 22, 25, 31, 33, 36, 40, 42, 49, 50).

[Den70] PJ Denning. Virtual memory. In: *ACM Computing Surveys (CSUR)* 2(3):(1970), 153–189 (pages 3, 8).

[Den96] PJ Denning. Virtual memory. In: *ACM Computing Surveys (CSUR)* 28(1):(1996), 213–216 (page 3).

[Dre07] U Drepper. What every programmer should know about memory. In: *Red Hat, Inc* 11(2007):(2007), 2007 (page 9).

[EH13] K Elphinstone and G Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 133–150 (page 50).

[Fot61] J Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. In: *Communications of the ACM* 4(10):(1961), 435–436 (page 3).

[Hal+23] A Halbuer et al. Morsels: Explicit virtual memory objects. In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. 2023, pp. 52–59 (pages 1, 17).

[Hei99] G Heiser. Anatomy of a High-Performance Microkernel. In:(1999) (pages 10, 15, 21–24, 47, 50).

[Int17] Intel. *5-Level Paging and 5-Level EPT White Paper*. Intel. 2017. https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html (visited on 08/07/2024) (pages 1, 37, 48).

[JM97] B Jacob and T Mudge. Software-Managed Address Translation. In: Third International Symposium on High-Performance Computer Architecture. IEEE Comput. Soc. Press, 1997, pp. 156–167 (pages 4, 15, 50).

[JM98a] B Jacob and T Mudge. Virtual Memory in Contemporary Microprocessors. In: *IEEE Micro* 18(4):(1998), 60–75 (pages 1, 4, 5, 8, 9, 48).

[JM98b] B Jacob and T Mudge. Virtual memory: Issues of implementation. In: *Computer* 31(6):(1998), 33–43 (pages 1, 4, 5, 7, 9, 11, 50).

[JM98c] BL Jacob and TN Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In: *ACM SIGPLAN Notices* 33(11):(1998), 295–306 (pages 1, 5, 6, 8–11, 41, 48).

[JJ01] A Jaleel and B Jacob. In-line interrupt handling for software-managed TLBs. In: *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*. IEEE. 2001, pp. 62–67 (page 50).

[Lie96] J Liedtke. On the realization of huge sparsely occupied and fine grained address spaces. PhD thesis. Berlin Institute of Technology, 1996. https://d-nb.info/949253553 (pages 10, 15).

[16] *MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual*. 2016. (Visited on 08/08/2024) (pages 9, 23, 50).

[24a] *Mit-Pdos/Xv6-Riscv*. MIT PDOS, Aug. 24, 2024. https://github.com/mit-pdos/xv6-riscv (visited on 08/24/2024) (pages 20, 33).

[Mit+21] A Mittelbach et al. Non-cryptographic Hashing. In: *The Theory of Hash Functions and Random Oracles: An Approach to Modern Cryptography*:(2021), 303–334 (page 1).

[Par+22] CH Park et al. Every walk'sa hit: making page walks single-access cache hits. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 128–141 (pages 16, 50).

[PW17] D Patterson and A Waterman. *The RISC-V Reader: An Open Architecture Atlas*. 1st. Strawberry Canyon, 2017 (pages 5, 6, 11–13, 23).

[24b] *QEMU Source*. QEMU, Sept. 6, 2024. https://github.com/qemu/qemu (visited on 09/06/2024) (pages 20, 31, 32, 34, 35).

[] *Record/Replay — QEMU Documentation*. https://www.qemu.org/docs/master/system/replay.html (visited on 09/04/2024) (page 46).

[Ska+20] D Skarlatos et al. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1093–1108 (pages 6, 7, 16).

[TB14] AS Tanenbaum and H Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014 (pages 1, 3–5, 7, 25, 40, 41).

[Wal99] S Wales. VIRTUAL MEMORY IN A 64-BIT MICROKERNEL. PhD thesis. Citeseer, 1999 (page 1).

[Wat+24] A Waterman et al. The RISC-V Instruction Set Manual: Volume II: Privileged Architecture version 20240411. In:(2024). https://riscv.org/technical/specifications/ (pages 9, 11–13, 21, 22, 31, 32, 35, 36, 40, 48, 50).

[YT16] I Yaniv and D Tsafrir. Hash, don't cache (the page table). In: *ACM SIGMETRICS Performance Evaluation Review* 44(1):(2016), 337–350 (pages 1, 5, 7, 8, 16).

[ZSM20] D Zagieboylo, GE Suh, and AC Myers. The cost of software-based memory management without virtual memory. In: *arXiv preprint arXiv:2009.06789*:(2020) (pages 1, 17).

Declaration of Authorship

Ich erkläre hiermit gemäß § 9 Abs. 12 APO, dass ich die vorstehende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung der Bachelorarbeit ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Bamberg, den _____ _____

Max Meidinger