Mixter: Towards Simultaneous Emulation and Virtualization

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Sciences

by

Thomas Hütter, B.Eng.

Registration Number 01120239

to the Department of Computer Sciences
at the Faculty of Natural Sciences
at the Paris Lodron University of Salzburg

Supervisor: Univ.-Prof Dr. Christoph Kirsch

Salzburg, August 22, 2017

Department of Computer Sciences • Faculty of Natural Sciences
Paris Lodron University of Salzburg
Jakob-Haringer-Straße 2 • A-5020 Salzburg • +43 (0) 662 8044-6328
https://informatik.uni-salzburg.at

Statement of Authentication

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Salzburg, August 22, 2017

Thomas Hütter, B.Eng.

Acknowledgments

Zu Beginn möchte ich mich bei Professor Christoph Kirsch für die Betreuung während meines gesamten Bachelor- und Masterstudiums bedanken. Aufgrund deiner Bemühungen habe ich in den letzten Jahren viele großartige Erfahrungen machen dürfen. Danke, dass du mir all diese Türen geöffnet hast.

Ich bin dankbar, dass ich so eine wundervolle Familie und tolle Freunde habe, die mir immer den Rücken gestärkt und meinen Blick nach vorne gerichtet haben.

Bettina, vielen Dank, dass du diesen Weg mit mir gehst und dir immer wieder Zeit genommen hast, mein Englisch zu verbessern.

Und ich danke speziell dir, Mama, für deine Unterstützung, deine Geduld und dein Verständnis über all die Jahre.

Vielen Dank!

Abstract

Machine emulation is software that reproduces the functionality of hardware, for example through some form of code interpretation. System virtualization aims at creating virtual instances of hardware on which it runs, mostly through some form of context switching and virtual memory. Emulation and virtualization of the same hardware is supposed to be functionally equivalent while emulation may be simpler to implement than virtualization if performance is less of a concern. This thesis presents the design and implementation of an experimental hypervisor called *mixter* which virtualizes the machine on which it runs through emulation and virtualization. *mixter* can in fact alternate between emulation and virtualization at runtime. Its purpose is to bring us closer to identifying methodologies for verifying the functional equivalence of emulation and virtualization. The software of *mixter* is implemented in *selfie*, a software system that is used for educational purposes.

Keywords: Verification, virtualization, emulation, selfie

Contents

CHAPTER 1

Introduction

Machine emulation and virtualization are two important concepts in computer science. Though, most users do not realize that these concepts are used in a number of well known applications. For example, emulators are used to emulate a smartphone in mobile application development or to emulate an old video console to play retro games; whereas, virtualization is mainly used in server applications.

A machine emulator (software) mimics a specific computer architecture (hardware) which may be different to the one that the emulator is executed on. For example, a smartphone is emulated on a computer when testing a smartphone application. Emulation can be achieved via a form of code interpretation which means that an emulator is capable of executing code that is written for the emulated hardware instruction by instruction. Hence, it is possible to execute software that was originally developed for another architecture.

Compared to emulators, in virtualization a hypervisor creates virtual instances of the machine on which it runs, mostly through a form of virtual memory. Instead of code interpretation, a hypervisor tells the underlying machine to execute the code for it which is called hosting. The underlying machine switches to the context of the code that needs to be executed and starts executing it. On the one hand, this way a performance gain is achieved since a hypervisor avoids code interpretation; on the other hand, the hosted program is restricted to the underlying hardware architecture.

Even though, the concepts of an emulator and a hypervisor are different it is supposed that emulation and virtualization of the same hardware are functionally equivalent when they execute the same code. This thesis presents *mixter*, an experimental hypervisor that is capable of simultaneously emulating and virtualizing a *MIPSter* machine. Here, simultaneously means that *mixter* alternates between both concepts at runtime. *mixter* is a first step towards identifying methodologies for verifying the functional equivalence of emulation and virtualization.

Dependent on whether *mixter* is emulating or virtualizing, the context information is stored in two different places. First, in case of emulation, *mixter* executes the code itself and stores the context information in its own address space. Second, in case of virtualization, the code is executed by the underlying machine of *mixter* and, therefore, the context information is stored in the address space of the underlying machine. Hence, the main task for alternating between emulation and virtualization is to operate on the same context data in both options.

A concept called context caching is introduced that allows *mixter* to operate on the same context data independent of emulation and virtualization. With each context switch an emulator caches the latest context information of the *mixter* instance on top of it. Therefore, the emulator is informed about the changes on the context information stored in the address space of *mixter*. Additionally, the changes made by the emulator on the cached context information are written back to the context in the address space of *mixter*.

The software of *mixter* is implemented as part of the *selfie* project. *selfie*[1] is a software project used as an educational platform to teach topics like compiler construction and operating systems. The project includes a self-compiling compiler (*starc*), a self-executing emulator (*mipster*), and a self-hosting hypervisor (*hypster*). The *mipster* emulator is an interpreter of *MIPSter* code, which is a subset of the MIPS32[8, 9, 10] architecture. Based on a microkernel within the *mipster* emulator, the *hypster* hypervisor is able to host any *MIPSter* code. *mixter* makes use of the existing structure of *selfie* and is integrated smoothly by following the same design principles, namely: self-referentiality, simplicity, and readability. Self-referentiality for *mixter* means that it can execute and host another instance of *mixter*.

With the implementation of context caching, the system calls to create a context and to map a virtual page to a physical frame are obsolete. In fact, *mixter* goes one step further by getting rid of all system calls except context switching. As a consequence, the microkernel is no longer located within the *mipster* emulator.

1.1 Outline

The thesis is structured in four chapters starting with the introduction. Prior to the explanation of *mixter* it is important to understand *selfie* and the concepts of emulation and virtualization which is why they are explained in greater detail. A more elaborate view on the chapters is given below.

Chapter 1 gives a brief introduction to emulation, virtualization, the *selfie* project as well as *mixter* itself; furthermore, an outline of the thesis is given.

Chapter 2 provides closer insight into the concept and implementation of *selfie* before *mixter* was introduced. It contains an overview of all the different parts of *selfie* focusing on the most relevant topics regarding *mixter*.

Chapter 3 is all about *mixter*. The concept, theoretical aspects, and the implementation will be covered in this chapter.

Chapter 4 will end this thesis with a conclusion and remaining future work.

[1]http://selfie.cs.uni-salzburg.at

CHAPTER 2

System

2.1 Overview

selfie[1] is a software project which is being developed by the Computational Systems Group at the Department of Computer Sciences at the University of Salzburg. It is used as an educational platform to teach various topics like operating systems, compiler construction and runtime systems. Even though *selfie*'s source code is just one single file with about 7000 lines of code it consists of four major parts:

- **starc** is a self-compiling compiler which is capable of compiling all of *selfie*'s source code. It takes any *C** code and generates appropriate *MIPSter* code for it where *C** is a Turing complete subset of *C* and *MIPSter* is a subset of an instruction set architecture called MIPS32.
- **mipster** is the name of the self-executing emulator within *selfie*. It can execute any *MIPSter* code compiled with *starc*.
- **hypster** is a self-hosting hypervisor and one of the main parts of *selfie*. Based on a microkernel within mipster, *hypster* maintains virtual machines that can host any *MIPSter* code as well as *selfie* itself.
- **libcstar** is a library written in *C** that provides for example bitwise operations. Therefore, one can use them even though they are not included in the *C** language subset itself.

Each part of *selfie* is fully self-referential (see Section 2.6) and therefore the whole project is self-referential as well. Going into every detail of this project would be far beyond the scope of this thesis. However, there is a draft of a free book[2] available that covers most of *selfie* in a detailed manner. Nevertheless, all functions of *selfie* that are relevant for this thesis will be covered in this Chapter 2.

[1]http://selfie.cs.uni-salzburg.at
[2]https://leanpub.com/selfie

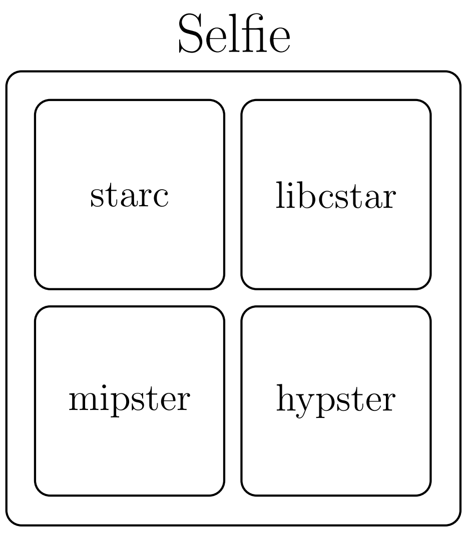


Figure 2.1: Different parts of selfie.

2.2 starc - the C^* compiler

It could be said that the idea and the very first part of the *selfie* project was born via lecturing compiler construction. For many years Professor Kirsch's students had to implement their own self-compiling compiler as part of the lecture. Facing the same task, *starc* was built with further demands on readability and simplicity. The source language as well as the target language were kept as minimalistic as possible, but were still sufficient to cover all functionality of *selfie*.

Before going into more detail on the first part of *selfie*, called *starc*, it is necessary to introduce some basic terms of compiler theory based on the lecture notes of Christoph Kirsch. [4]

Machine Code: *A sequence of instructions executed directly by a computer's central processing unit (CPU). [5]*

Compiler: *A computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object or machine code. The most common reason for converting source code is to create an executable program. [5]*

Single-pass: *The compiler passes through the source code only once.*

The transformation from source to target language in a single-pass compiler is processed via several different steps as shown in Figure 2.2.

A given file written in the source language serves as input for a compiler and is processed by the scanner. The scanner goes through the input file character by character and combines them to so called tokens. Comparing it to a text file a scanner would form words out of characters by splitting them, for example, by

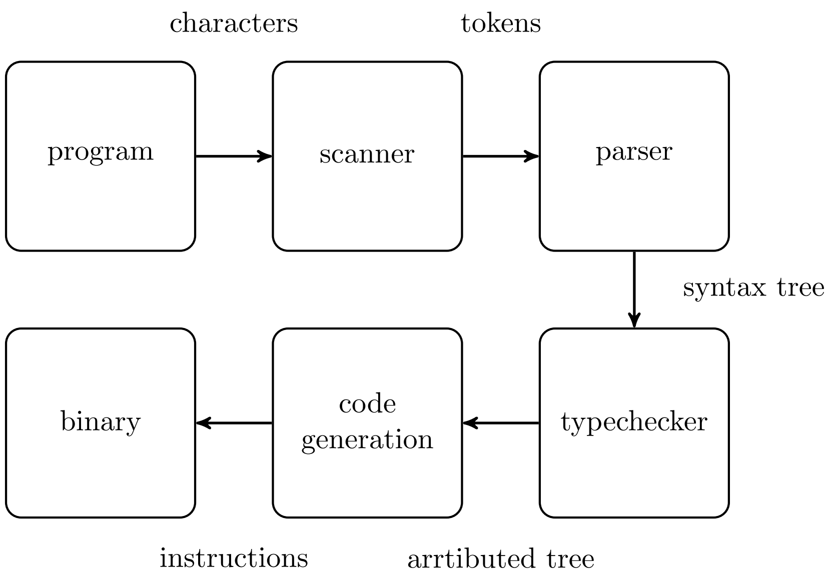


Figure 2.2: Steps taken during a single-pass compilation of a program.[4]

whitespace or punctuation marks. These tokens enable us to execute the next step called parsing.

Compared to the example of textual input this would be the step from words to sentences. Similar to a human language, a programming language is defined by a grammar. According to the given grammar of the input language parsing analyzes the syntax of the input tokens. There are many different ways of how parsing can be proceeded, still, *selfie* uses a $LL(1)$-Parser. This is a special form of a top-down parser where the input is parsed from left (therefore, the L) to right and the leftmost derivation is performed. The input and implementation language of *selfie*, called $C*$ (see Section 2.2.1), is designed in a way that the parser only needs to look at one token ahead, at most, in order to perform the analysis, this is where the 1 in $LL(1)$-Parser comes from.

The sense of type checking is self explanatory. An example of it would be to make sure that there is no string assigned to an integer variable.

The last step is called code generation. The compiler needs to generate code that is semantically equivalent to the input but written in the target language. In order to be able to create machine code the instructions have to be encoded with specified formats by the compiler. A detailed view on encoding can be found in Section 2.2.4.

2.2.1 The $C*$ programming language

As stated above one aim of *selfie* is to keep everything as minimalistic as possible while still providing full functionality. Therefore, a tiny subset of the well known programming language C was chosen as source language for the *starc* compiler (and the source code of *selfie* itself).

Without describing every detail of the grammar some interesting facts should be pointed out. As already mentioned above the grammar is $LL(1)$. Regarding data types C^* only provides two different ones, namely signed 32-bit integers and pointers to them. Consequently, the grammar does not provide any compositional data structures like arrays or structs. Any collection of data is grouped consecutive in memory and is accessed via pointers and offsets. In order to get the data of a given address it is important to have a dereference operator $*$ which is where the name C^* comes from. Furthermore, C^* comes with five built-in functions namely exit, malloc, open, read, write. These functions are implemented within *selfie* as so-called system calls. A detailed explanation can be found in Section 2.5.1. More details about the C^* programming language are described by Kirsch [6].

2.2.2 The MIPSter instruction set architecture

The same design criteria that are used for the source language are applied to define the target language of the compiler. Like before a minimalistic subset of a well known standard is used. *MIPSter* is an instruction set architecture (ISA) that provides 17 well chosen instructions of the MIPS32 standard[8, 9, 10] (*addiu*, *addu*, *beq*, *bne*, *divu*, *j*, *jal*, *jr*, *lw*, *mfhi*, *mflo*, *multu*, *nop*, *slt*, *subu*, *sw*, *syscall*). These instructions are again a tiny but powerful subset capable of implementing all of *selfie*.

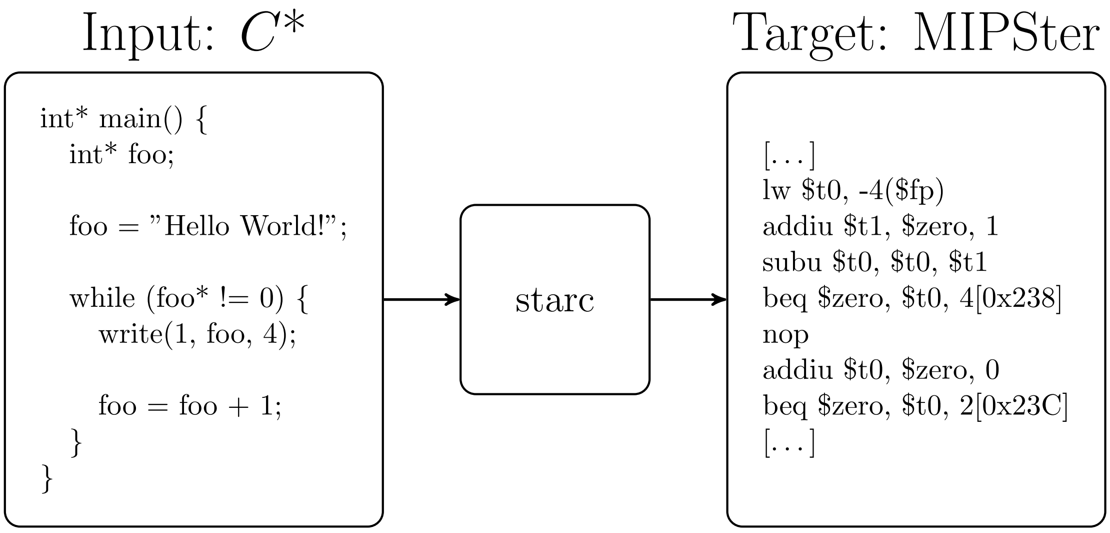
2.2.3 Compiling

After covering the basics of the source language and the target language, now, the focus is at the process of compilation which is schematically depicted in Figure 2.3. The *starc* compiler operates as a single-pass and recursive decent compiler and can take any given C^* code as input which is compiled to *MIPSter* code including *starc* and *selfie* itself. This is called *self-compiling* and is explained in Section 2.6.

2.2.4 Encoding

In order to create *MIPSter* machine code all instructions are encoded with a special format. Each *MIPSter* instruction can be categorized according to one of three different formats based on the MIPS CPU instruction formats[8]:

- **R-Format**:
  addu, divu, mfhi, mflo, multu, nop, slt, subu, syscall
- **I-Format**:
  addiu, beq, bne, lw, sw
- **J-Format**:
  j, jal



```
int* main() {
  int* foo;

  foo = "Hello World!";

  while (foo* != 0) {
    write(1, foo, 4);

    foo = foo + 1;
  }
}
```

```
[...]
lw $t0, -4($fp)
addiu $t1, $zero, 1
subu $t0, $t0, $t1
beq $zero, $t0, 4[0x238]
nop
addiu $t0, $zero, 0
beq $zero, $t0, 2[0x23C]
[...]
```

Figure 2.3: *starc* - the *C** compiler.

Figure 2.4 gives a detailed view of how instructions of all formats have to be encoded.

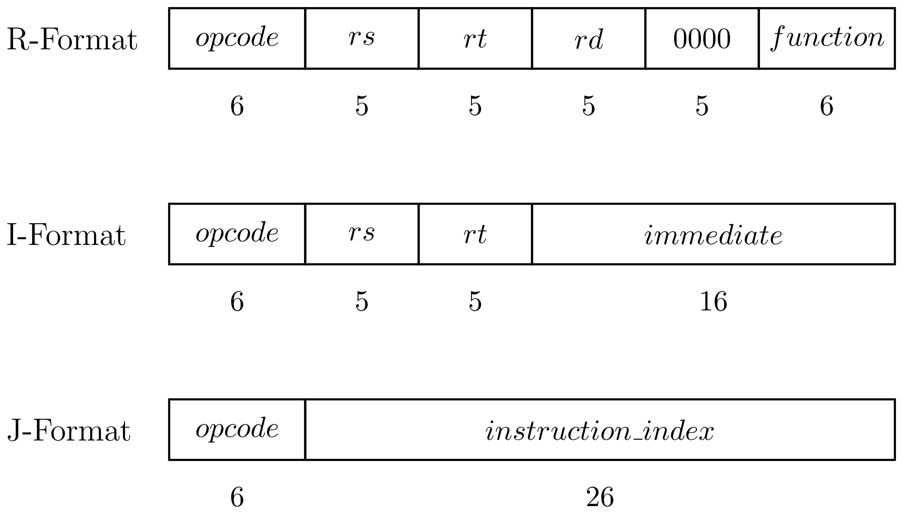


Figure 2.4: Different encoding formats of *MIPSter*.[8]

When encoding is performed the input is an instruction of the target language and the result is a binary number that holds all the information about this instruction. The example in Figure 2.5 shows each step in detail. First, the according number of the instruction has to be looked up, as well as the register numbers. The integer at the right is already an immediate so nothing has to be performed in the first step. Next, all the numbers of step one have to be translated in binary numbers of the size given by the according format (in the example format I). Last, all the numbers are appended one after another in the way that is defined in the format. This is done with shift and addition operators. Finally, a 32-bit integer that represents the instruction *ADDIU* $\$t0$, $\$zero$, 135 is created.

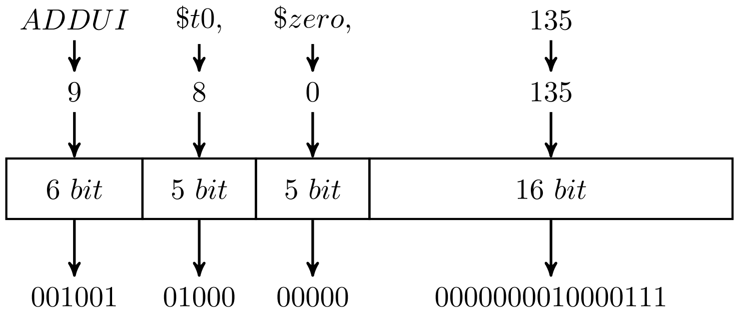


Figure 2.5: Example of encoding a *MIPSter* instruction.[8]

2.3 libcstar - the *C** library

As discussed in Section 2.2.1 the design of *C** is minimalistic. This is why a lot of common programming language features such as bit operations (for example shift left or shift right) are not part of *C**. Since there is no linker available within *selfie*, it also includes a *C** library named *libcstar* which provides several features that are shown in Figure 2.6. Another example is the printing of strings. The library *libcstar* provides a *print* function that is more convenient than the use of the *write* function for each character.

To access the functionality of *libcstar* in another *C** program, the library parts can easily be copied from the *selfie* source code.

2.4 The mipster emulator

2.4.1 Overview

As already mentioned above the emulator within *selfie* is called *mipster*. Before going into detail, it is necessary to have a look at emulators in general.

Emulator: *Software that enables one computer system (called the host) to behave like another computer system (called the guest). [5]*

App development is a well known example where emulators are used. Almost every integrated development environment (**IDE**) used for mobile application development comes with software that emulates common mobile devices and operating systems, for example Android SDK.

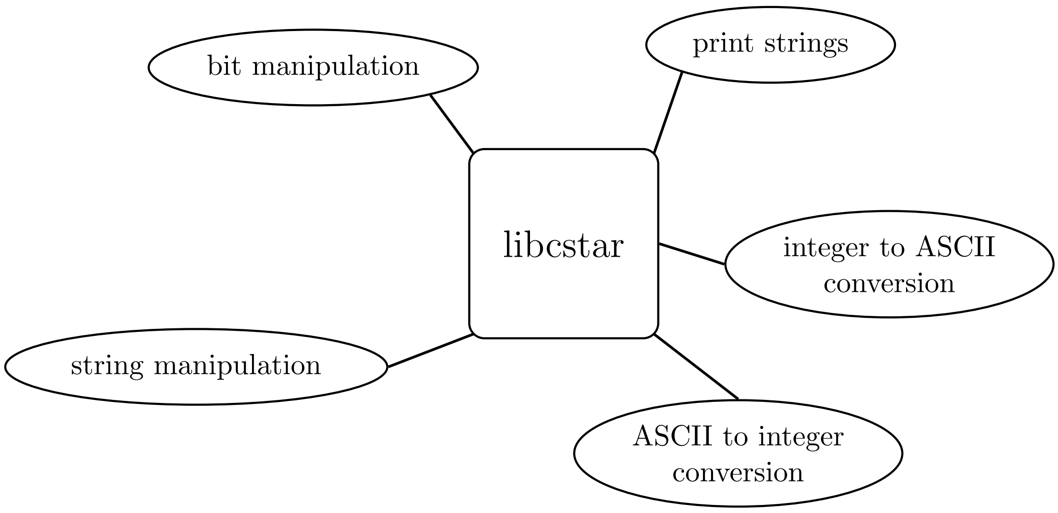


Figure 2.6: libcstar - *C** library.

Now a short overview of the guest system that is emulated by *mipster* is given. Since *starc* compiles *C** code to *MIPSter* code, a machine that can execute *MIPSter* instructions is needed. Furthermore, the machine consists of three different units:

- **Arithmetic unit**:
  The arithmetic unit is a collection of 32 so called general purpose registers, each of them can hold 32 bits. Table 2.1 gives detailed information about all those registers and their purposes.
- **Control unit**:
  This is a small number of registers that are absolutely necessary for the machine to interpret code, like the program counter (pc), the instruction register (ir), break (brk) and the page table pointer (pt). One could say that even some of the general purpose registers are part of the control unit, e.g. register number 31 which is used for return addresses.
- **Store**:
  Store contains all the memory that the machine provides to the guest system. All the information is stored in it. The memory is organized as depicted in Figure 2.7.

Program counter (pc): *A processor register that indicates where a computer is in its program sequence. In most processors, the* ***pc*** *is incremented after fetching an instruction, and holds the memory address of ("points to") the next instruction that would be executed. [...] [5]*

Instruction register (ir): *A processor register that holds the instruction that is to be executed. The process of writing an instruction into* ***ir*** *is called fetching.*

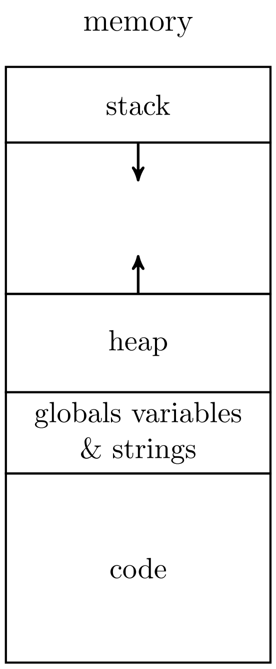


Figure 2.7: Memory layout of the emulated memory.[5]

Program break (brk): *A processor register that holds a pointer to the end of the static memory. The static memory contains the code, global variables and strings.*

Page table: *A page table is a data structure used to resolve virtual addresses to physical addresses. This is done by mapping one virtual page to one physical frame in the table. For each virtual memory address space a page table is needed.*

Page table pointer (pt): *A processor register that holds a pointer to the page table of the process currently executed.*

These 32 general and various special purpose registers alongside memory hold all the information, see Table 2.2. This information at a given time is called the machine state. Here "at a given time" means between two instructions. After executing another instruction the whole system and consequently the machine state as well may look completely different.

2.4.2 Interpretation

mipster can be invoked in two different ways as shown in Figure 2.8. On the one hand the emulator works as an unoptimized *MIPSter* interpreter, on the other hand it can also be used as a disassembler. This section covers all the important information about the interpreter while the disassembler is described in Section 2.4.3.

Register Number	Alternative Name	Description
0	$zero$	the value 0.
1	$\$at$	(assembler temporary) reserved by the assembler.
$2-3$	$\$v0-\$v1$	(values) from expression evaluation and function results.
$4-7$	$\$a0-\$a3$	(arguments) First four parameters for subroutine. Not preserved across procedure calls.
$8-15$	$\$t0-\$t7$	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls.
$16-23$	$\$s0-\$s7$	(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls.
$24-25$	$\$t8-\$t9$	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $\$t0-\$t7$ above. Not preserved across procedure calls.
$26-27$	$\$k0-\$k1$	reserved for use by the interrupt/trap handler
28	$\$gp$	global pointer. Points to the middle of the 64K block of memory in the static data segment.
29	$\$sp$	stack pointer. Points to last location on the stack.
30	$\$s8/\$fp$	saved value / frame pointer. Preserved across procedure calls.
31	$\$ra$	return address.

Table 2.1: General purpose registers in MIPS [3].

Value	Description
pc	Program counter
ir	Instruction register
pt	Page table pointer
brk	Break between code, data, and heap
regs	32 general purpose registers
reg_hi	Used for multiplication and division
reg_lo	Used for multiplication and division
memory	Emulated memory array

Table 2.2: Machine state of mipster.

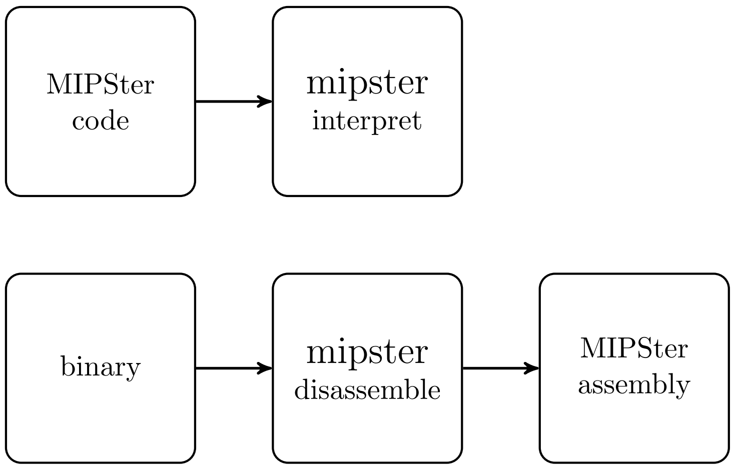


Figure 2.8: mipster can be invoked in two different ways.

An interpreter is a piece of software that simply executes one instruction after another. This is done in three steps:

- fetching an instruction,
- decode it and
- execute it.

Basically, this happens in a loop until the interpreter is interrupted by an exception (for example timeout, page fault, etc.). The next sections will contain detailed information about all of the steps mentioned above.

2.4.2.1 Fetching instructions

The information of the machine state is needed for executing a guest on *mipster* by executing one instruction after another. Before an instruction can be executed the emulator has to fetch it first. This is done with the help of two values from the machine state, namely the program counter and the page table pointer.
The program counter contains the address to the next instruction that is to be executed. Since this address is virtual, *mipster* has to translate it via a the given page table in pt.

2.4.2.2 Decode instructions

The data that is produced by fetching an instruction is a 32-bit integer value. According to the *MIPSter* standard formats for encoding the instruction is decoded now. Encoding an instruction is described in Section 2.2.4, basically decoding applies all these steps in reversed order.

When an instruction is decoded the first step is to extract the opcode. A brief look on Figure 2.4 shows that the opcode is at the same position for every format.

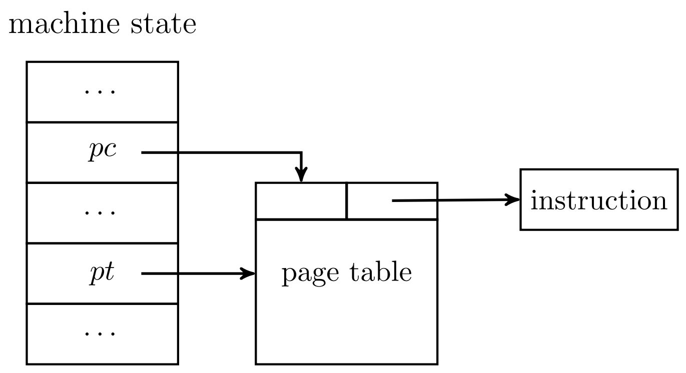


Figure 2.9: Fetching an instruction.

The reason for this becomes quiet clear in the process of decoding, because the opcode is necessary to choose the correct format before all other values are decoded. Knowing the correct format in combination with bit shifting allows us to decode the 32 bit integer into the according values needed for the instruction. The full concept of decoding is depicted in Figure 2.10.

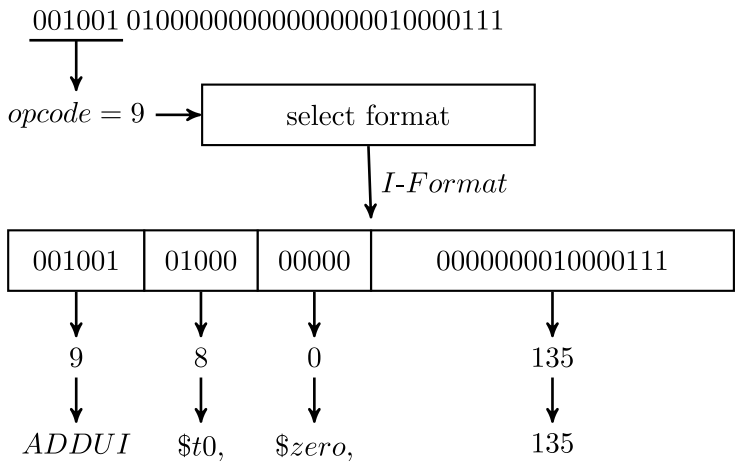


Figure 2.10: Decoding an instruction.

2.4.2.3 Execute instructions

The concept of execute is explained easily. For each *MIPSter* instruction the emulator has a concrete implementation of its functionality. Since all the parameters are given in the defined registers from decoding *mipster* only needs to execute on them. As an example Listing 1 contains *selfie*'s implementation of the jump (jr) instruction. The code is quiet simple. Except from debugging the only thing that happens is that the program counter is set to a given value from the parameter.

2.4.2.4 Traps

As already mentioned above, interpretation is done in a loop and may be interrupted by a trap. In *selfie* this is done by throwing an exception. There may be several reasons for an exception:

- **Page faults**:
  Whenever a virtual address is not mapped in the corresponding page table a page fault occurs.
- **Timer interrupts**:
  selfie implements a timer that interrupts the interpreter after a given amount of instructions.
- **Invalid address accesses**:
  If an invalid virtual address is accessed by the interpreted program.
- **Unknown instructions**:
  The actual interpreted instruction is not in the set of *MIPSter* instructions.

```
void fct_jr() {
  if (debug) {
    printFunction(function);
    print((int*) " ");
    printRegister(rs);
    if (interpret) {
      print((int*) ": ");
      printRegister(rs);
      print((int*) "=");
      printHexadecimal(*(registers+rs), 0);
    }
  }

  if (interpret)
    pc = *(registers+rs);

  if (debug) {
    if (interpret) {
      print((int*) " -> $pc=");
      printHexadecimal(pc, 0);
    }
    println();
  }
}
```

Listing 1: mipster's implementation of the jr call.

2.4.3 Disassembling

After discussing the interpreter of *mipster* in a detailed manner the second operation mode of *mipster* is briefly described here.

Disassembler: *A computer program that translates machine language into assembly language the inverse operation to that of an assembler. [5]*

```
0x1CBFC(~7088): 0x00081021: addu $v0,$zero,$t0
0x1CC00(~7088): 0x08007302: j 0x7302[0x1CC08]
0x1CC04(~7088): 0x00000000: nop
0x1CC08(~7089): 0x27DD0000: addiu $sp,$fp,0
0x1CC0C(~7089): 0x8FBE0000: lw $fp,0($sp)
0x1CC10(~7089): 0x27BD0004: addiu $sp,$sp,4
0x1CC14(~7089): 0x8FBF0000: lw $ra,0($sp)
0x1CC18(~7089): 0x27BD000C: addiu $sp,$sp,12
0x1CC1C(~7089): 0x03E00008: jr $ra
0x1CC20(~7089): 0x00000000: nop
```

Listing 2: Example output of *mipster*'s disassembler.

As shown in Figure 2.8 *mipster* can take any *MIPSter* machine code and outputs *MIPSter* assembly. Compared to machine code the output (see Listing 2) is easily readable for humans.

2.4.4 Machine context

The purpose of a machine context is to store the machine state of a specific instance. While executing an instance, the machine state of the emulated processor changes dynamically. It is important to save the current machine state before a context switch occurs, in order to be able to resume the execution of this instance when it is scheduled again. In fact, these are the two main steps of context switching; saving the current machine state in the according machine context and loading the values of the machine context that is to be executed next. A *selfie* instance creates a context for an instance on top of it. Furthermore, a duplication of each context is created in the address space of the emulator. Section 2.8.1 describes in detail which contexts are created in which address spaces.

However, a context is uniquely identified with an integer number assigned by the microkernel. Figure 2.11 shows the structure of a machine context and a brief description of all the fields. While fields like id, pointer to registers, or page table pointer remain the same during the whole execution; others, like the program counter may change after any instruction. The arrows at the next and prev field indicate pointers since all machine contexts are organized as a list.

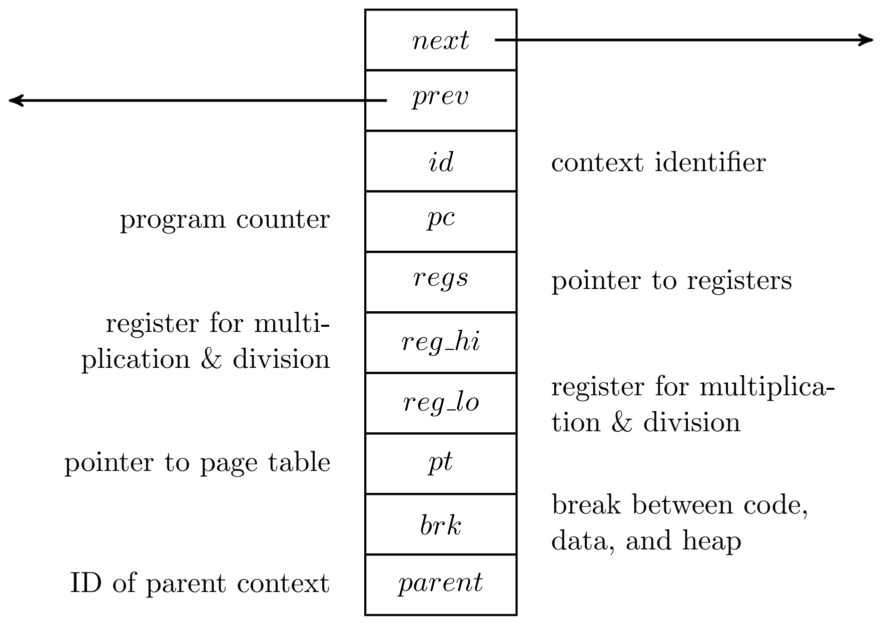


Figure 2.11: Structure of a machine context in selfie.

2.4.5 Microkernel

Microkernel: *A piece of software that provides as little functionality as possible which is necessary to implement an operating system.*

A special characteristic of *selfie* is the fact that the microkernel sits within the emulator. The microkernel provides a bunch of important functionality that is necessary in order to implement *hypster*, a hypervisor within *selfie*, see Section 2.5. Like all built-in library functions of *C** system calls are used to access the functionality of the microkernel. A detailed view on system calls can be found in Section 2.5.1. The implementation is inspired by the work of Jochen Liedtke[7]. The three main areas that the microkernel is taking care of are:

- **Machine contexts**:
  The first functionality concerns machine contexts. E.g., a *hypster* instance that runs on top of *mipster* creates a virtual machine to host another instance. Therefore, *hypster* needs to create a new machine context and tell the microkernel via system call to do so as well. The following list shows all the different system calls that the microkernel provides regarding machine contexts:
  - **ID**:
    Get the ID of the currently executing context.
  - **create**:
    Creates a context in the address space of the microkernel.

  - **delete**:
    Deletes a context in the address space of the microkernel.
  - **switch**:
    Switches from one context to another.

- **Virtual memory**:
  Similar to machine contexts, the information about virtual memory has to be passed down to *mipster*. Here there is only one system call:
  - **map**:
    Map a virtual page to a physical frame in the address space of the microkernel.
- **Status**:
  As part of the system state the emulator holds a value called machine status. It is used in case of an exception to hold the exception number as well as further information about the exception like the page number of a faulting page.
  - **get**:
    Return the machine status.

2.5 The hypster hypervisor

Since *hypster* is the hypervisor implemented in *selfie* the following definitions of virtualization and hypervisors need to be discussed before going into every detail.

Virtualization: *The act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, operating systems, storage devices, and computer network resources. [5]*

Hypervisor: *A piece of computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine. [5]*

Hosting: *Rather than executing a binary itself, a hypervisor asks the machine on which it runs to do so. This execution by the underlying machine is called hosting.*

Compared to *mipster* which emulates a machine, *hypster* hosts it. This means that *hypster* creates *MIPSter* virtual machines and instead of executing instructions itself it orders *mipster* to execute them, see Figure 2.12. Of course this has an enormous impact on the runtime. In fact, running multiple instances of *mipster* on top of each other leads to an exponential growth of runtime.

In order to be able to provide all the operating system functionality *hypster* needs to be able to access the microkernel in the emulator. The way this is implemented is called system calls and is explained in the next Section 2.5.1.

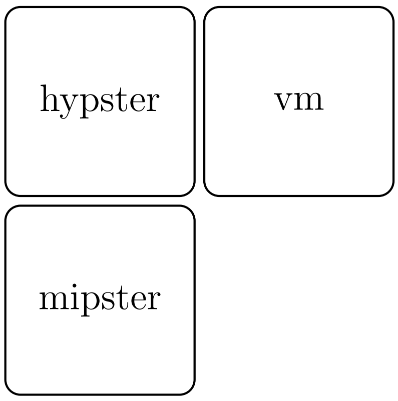


Figure 2.12: hypster - the hypervisor.

2.5.1 System calls

As already mentioned in Section 2.4.5 *hypster* is based on a microkernel that is located within *mipster*. Here "based on" means that the microkernel provides important functionality regarding machine contexts, status and virtual memory. In the *hypster* implementation system calls, also called hypster calls, are used to access these functions of the microkernel. The technique of system calls is also used for implementing the built-in library functions to bootstrap the compiler. The code and concept is almost identical for any of the two fields of application. Still, the built-in library functions are slightly simpler. In addition, it is important to take into consideration whether the hypster call was called within the emulator or on top of it. First, however it is of interest what it means when something is implemented as a system call.

Taking a closer look at the implementation of the five built-in library functions (exit, malloc, open, read and write) in the source code of *selfie* it becomes clear that there is a common pattern. There are two different procedures for each of them. One is called *emit*, the other is called *implement*. Listing 3 and Listing 4 show the according functions of the exit system call. A detailed explanation of all the details of a system call on the basis of the exit example is given in the following paragraphs. The concept is the same for all the other system calls as well.

The *emit* function is used to produce wrapper code that invokes the system call. Notice that this function is always invoked by the compiler. First, an entry for exit in the symbol table. Second, the arguments are handled by loading and removing them from the stack. Third, the system call itself is invoked. Every system call has its own number in order to be able to uniquely identify it; this number is first loaded before the call is finally invoked. Of course, exit never returns since the calling instance is exited. However, this is not the case for any of the other four library functions.

```
void emitExit() {
  createSymbolTableEntry(LIBRARY_TABLE, (int*) "exit", 0,
                         PROCEDURE, VOID_T, 0, binaryLength);

  emitIFormat(OP_LW, REG_SP, REG_A0, 0);

  emitIFormat(OP_ADDIU, REG_SP, REG_SP, WORDSIZE);

  emitIFormat(OP_ADDIU, REG_ZR, REG_V0, SYSCALL_EXIT);
  emitRFormat(0, 0, 0, 0, FCT_SYSCALL);

  // never returns here
}
```

Listing 3: Emit function of the exit system call invoked by the compiler.

The actual functionality of the exit call is coded in the *implement* procedure. This function differs from system call to system call. In the case of the exit call the *exitCode* is read from the specific register A0. A check guarantees that the *exitCode* is within the range of a signed 16-bit integer. To exit the caller instance an exception is thrown with the corresponding exception and *exitCode*. As explained in Section 2.4.2.4 the emulator stops interpreting and exits the caller instance.

```
void implementExit() {
  int exitCode;

  exitCode = *(registers+REG_A0);

  if (exitCode > INT16_MAX)
    exitCode = INT16_MAX;
  else if (exitCode < INT16_MIN)
    exitCode = INT16_MIN;

  throwException(EXCEPTION_EXIT, exitCode);

  // debugging code
}
```

Listing 4: Implement function of the exit system call invoked by the emulator.

So far, the general structure and implementation of a system call have been introduced. In case of hypster calls just a little more code has to be added. The reason for this procedure is that there is a difference between calling a hypster call within *mipster* or on top of it. Within *mipster* the compiler should not care about the emit function while generating code. Therefore, there is an additional procedure for *hypster* within *mipster*. Listing 5 shows the *hypster_create*() function which is only executed at boot level zero which means by the *selfie* instance that was invoked first. The *emit* and *implement* procedure remains the same as before. Even though, there are now two different implementations of each system call there is no difference in invocation for most of them. The only exception is the context switch system call where an invocation within *mipster* needs to call the interpreter on the target context. However, on top of *mipster* the interpreter is already activated, interpreting the actual context. The only thing to do is to switch to the target context.

```
int hypster_create() {
  return doCreate(selfie_ID());
}

int selfie_create() {
  if (mipster)
    return doCreate(selfie_ID());
  else
    return hypster_create();
}
```

Listing 5: Additional procedure for hypster calls.

2.6 Self-referentiality

A word that often occurs along with *selfie* and that has also been mentioned several times in this thesis by now is "self-referentiality". Instead of a definition its meaning is explained directly via the example of *selfie* and its components.

First, speaking of the compiler *starc* it can be said that it is self-compiling since it can compile any *C** code and, therefore, *selfie* and *starc* themselves.

An emulator like *mipster* is called self-executing if it can execute itself. In *selfie mipster* can run on top of another instance of *mipster*.

The same accounts for *hypster*. A hypervisor is called self-hosting if it can host itself. Again, *hypster* can run on top of itself and, therefore, is self-hosting.

Due to the self-referentiality of all parts of *selfie* the whole project is called self-referential. Furthermore, in *selfie* it is possible to run *hypster* on *mipster* and vice versa. Each of the steps above is shown in Figure 2.13.

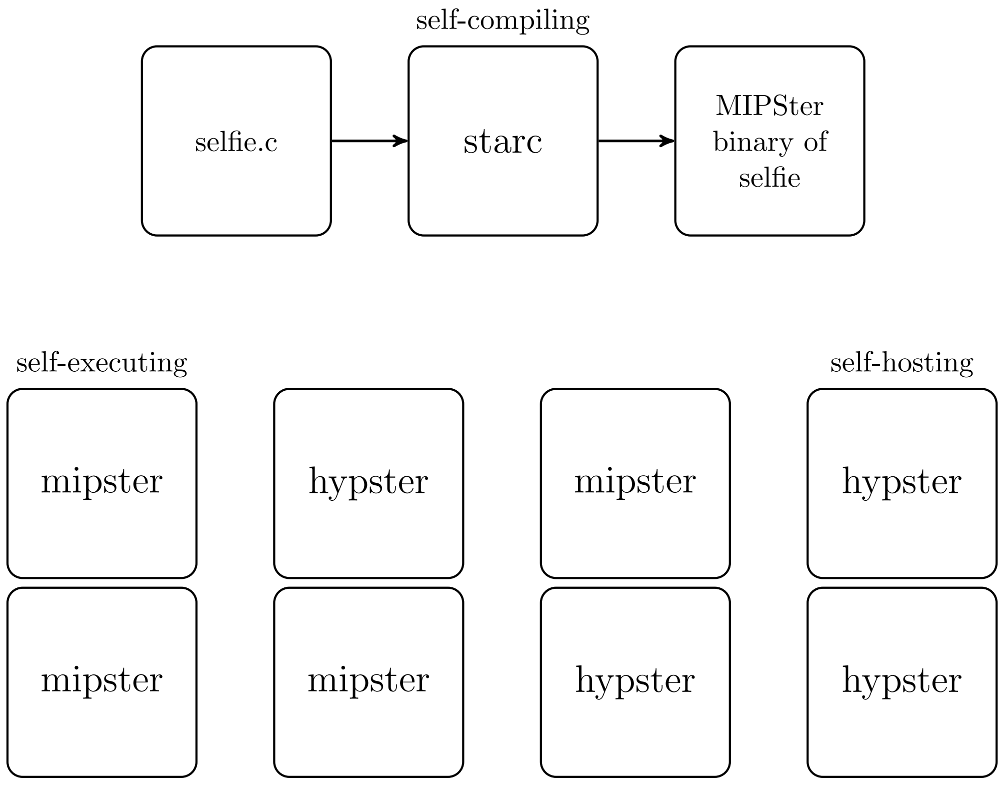


Figure 2.13: Self-referentiality

2.7 Compilation and execution

So far the focus was on the different parts of *selfie*, the concept and some implementation details. Now an explanation of how the source code can be built and executed is given.

At the beginning of this Chapter it has been mentioned that all of *selfie*'s code is within a single file and is written in the *C** programming language. Since *C** is a real subset of *C*, any common *C* compiler can be used to compile the whole project. The output is an executable version of *selfie*. Therefore, *starc*, *mipster* and *hypster* can be executed on this machine. Using parameters, the user can specify which parts of *selfie* should be executed in which order and further information on virtual memory size for *mipster* or *hypster* can be given. A complete list of the parameters can also be found in List 2.7 which has been taken from the official GitHub repository of *selfie*.[3]

- The **-c** option invokes the C* compiler on the given list of source files compiling and linking them into *MIPSter* code that is stored internally.
- The **-o** option writes *MIPSter* code produced by the most recent compiler invocation to the given binary file.

[3]https://github.com/cksystemsteaching/selfie

- The **-s** option writes *MIPSter* assembly of the *MIPSter* code produced by the most recent compiler invocation including approximate source line numbers to the given assembly file.
- The **-l** option loads *MIPSter* code from the given binary file. The **-o** and **-s** options can also be used after the **-l** option. However, in this case the **-s** option does not generate approximate source line numbers.
- The **-m** option invokes the *mipster* emulator to execute *MIPSter* code most recently loaded or produced by a compiler invocation. The emulator creates a machine instance with size MB of memory. The source or binary name of the *MIPSter* code and any remaining ... arguments are passed to the main function of the code. The **-d** option is similar to the **-m** option except that *mipster* outputs each executed instruction, its approximate source line number, if available, and the relevant machine state.
- The **-y** option invokes the *hypster* hypervisor to execute *MIPSter* code similar to the *mipster* emulator. The difference to *mipster* is that *hypster* creates *MIPSter* virtual machines rather than a *MIPSter* emulator to execute the code.
- The **-min** and **-mob** options invoke special versions of the *mipster* emulator used for teaching.

With the generated executable *starc* can be invoked to perform self-compilation as shown in Figure 2.14. According to the parameters given above the call would look like this:

```
$ ./selfie -c selfie.c -o selfie.m                    (2.7.1)
```

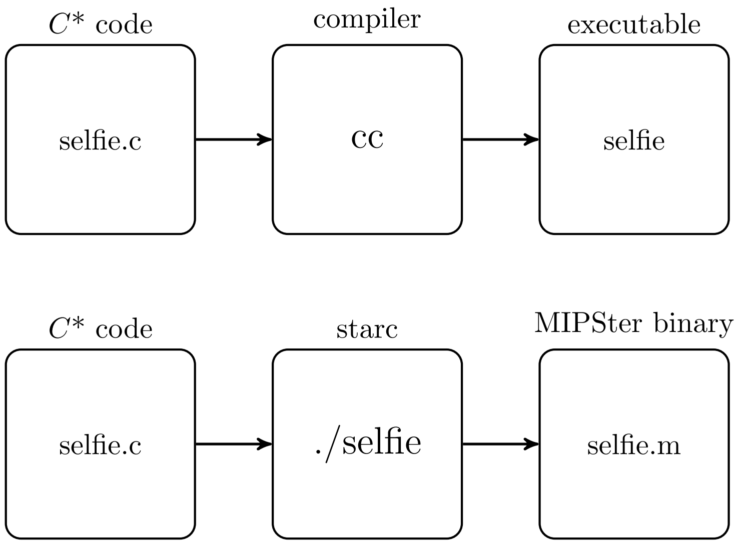


Figure 2.14: Building process of selfie.

Since *mipster* and *hypster* instances can be stacked in any possible way, complex calls with the **-m** and **-y** options can be built. Be careful, as explained in Section 2.5 running several instances of *mipster* on top of one another leads to exponential execution times.

More examples can be found on the GitHub page of *selfie*. Nevertheless, the following example call is given:

```
$ ./selfie -l selfie.m -m 3 -l hello.m -y 2                    (2.7.2)
```

In Example 2.7.2, a *mipster* instance is created, on top of it sits a *hypster* which hosts a *MIPSter* binary called *hello.m* in a virtual machine. This configuration is used in Section 2.8 to explain the memory layout that is created by *selfie*.

2.8 Memory organization

This Section covers probably the most important information that is needed to understand the core concept of this thesis. The two main questions that will be answered are:

- How is memory organized within *selfie*?
- Which information is stored in which address spaces?

2.8.1 Memory layout

The memory layout of a single *selfie* instance has already been briefly discussed in Section 2.4. This section provides more details and targets the question what the memory layout looks like if a *selfie* instance runs on top of another *selfie* instance.

mipster and *hypster* are both creating a virtual memory space that is used to run a program on top of it. When introducing *mipster*, the layout of the created virtual memory space was examined. In fact, the memory layout is organized entirely the same way by *hypster*. One question that remains is: what does the memory layout look like if multiple instances of *selfie* are run on top of each other?

To answer this question it is necessary to take a look at the concept of virtual memory. As stated above, *mipster* as well as *hypster* is creating a virtual memory space for each instance that they are emulating or hosting. Therefore, each instance can operate on a virtual address space starting from virtual address 0 up to a given memory size. This implicates that if instances are run on top of each other, like in Example 2.7.2, the virtual address spaces will become nested. The resulting memory layout from the example before is depicted in Figure 2.15.

To finally resolve a virtual address to a physical address a page table is needed. Such a page table exists for each virtual address space and maps virtual pages to physical frames. Figure 2.16 sketches the memory layout together with page tables of the example.

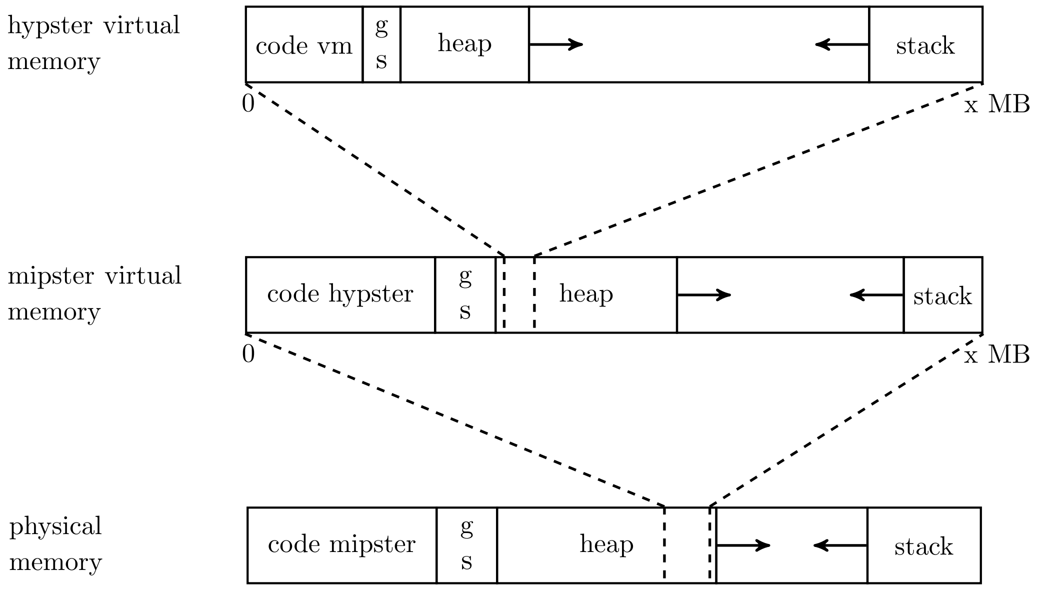


Figure 2.15: Memory layout of Example 2.7.2.

In *selfie* the emulator needs to know all these page tables in order to be able to resolve all addresses, since the emulator may interpret all the created instances on top of it. The page table information is passed via system calls by all these instances on top of the emulator, as already explained in Section 2.4.5 and Section 2.5.1. Therefore, in *selfie* there are duplicates at the emulator level not only of each page table but also of each machine context. This concept will be examined more closely in the following Section 2.8.2.

Another important fact that can be seen in Figure 2.16 is that contiguous virtual addresses are not contiguous in physical memory. Theoretically, two successive allocation calls could allocate one memory chunk at the beginning and one memory chunk at the end of the physical memory.

2.8.2 Duplication of machine contexts

As already stated above Figure 2.16 is not correct or at least does not show the full information. As explained in Section 2.4.5, the microkernel is implemented within *mipster* and handles machine contexts and virtual memory. Every *selfie* instance creates a machine context in its own address space for the instance running on top of it. Furthermore, a duplication of each machine context is created by the microkernel within the address space of the emulator. These duplications are used by the emulator to access the context information and to translate virtual addresses while interpreting an instance.

There are always two context creation calls for each instance. First, the microkernel is invoked to create a new context in the address space of the emulator. This has to be done first to get back a unique ID from the microkernel. After that an instance can create the context with the ID in its own address space. So, which machine contexts and page tables are created in which address spaces in Example 2.7.2?

mipster is invoked first. Therefore, *mipster* is said to be at boot level zero. Since the microkernel is part of the emulator, *mipster* is furthermore said to be at microkernel boot level. First, *mipster* creates a context of the instance running on top of it (*hypster*) at physical memory. Since the emulator created the context, no duplication is needed. Next, *hypster* is invoked to run on top of *mipster*. A context for the created virtual machine is needed. Following the steps from above *hypster* invokes the microkernel which creates a context at boot level zero and gives back the new ID. Last, *hypster* creates the context of the virtual machine in its own address space.

The result is visualized in Figure 2.17. Notice, that the address space of *mipster* is in the physical memory and the address space of *hypster* is in the virtual memory of *mipster*.

2.9 RISC-V port

During the work of this thesis another team finished working on a RISC-V[4] portation[5] of *selfie*. The RISC-V[1, 2] project was initiated at the University of California, Berkeley in 2010. The two major differences are targeting *starc* and *mipster*.

- **starc**: The compiler no longer targets *MIPSter*. Instead another subset of an open instruction set architecture called *RISCY* is the new target. The instruction set of *RISCY* is a well chosen subset of RISC-V instructions. Like MIPS, RISC-V is as the named suggests, a reduced instruction set architecture (RISC).
- **mipster**: The emulator needs to be changed as well of course. *rocstar* is the name of a *RISCY* emulator implemented within *selfie*. The porting process did not violate the self-reference property of *selfie*.

This is just additional information regarding the *selfie* project. As will be seen in Chapter 3 the implementation of *mixter* is absolutely independent of the underlying architecture. It can easily be ported to the RISC-V implementation. Nevertheless, the port was not part of this thesis and remains future work.

[4]https://riscv.org
[5]https://github.com/cksystemsteaching/selfie/tree/riscv

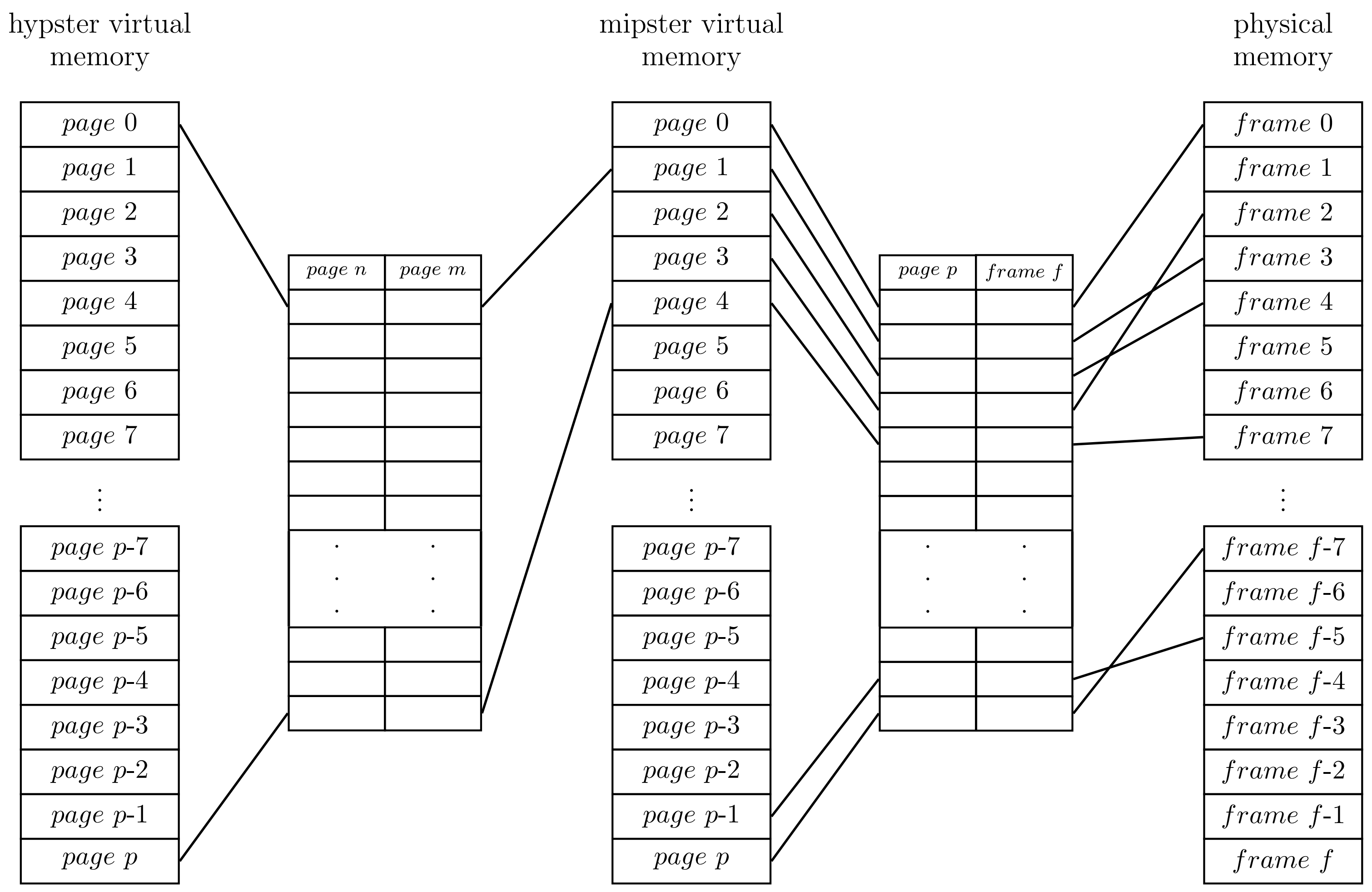


Figure 2.16: Page tables and virtual address spaces of Example 2.7.2.

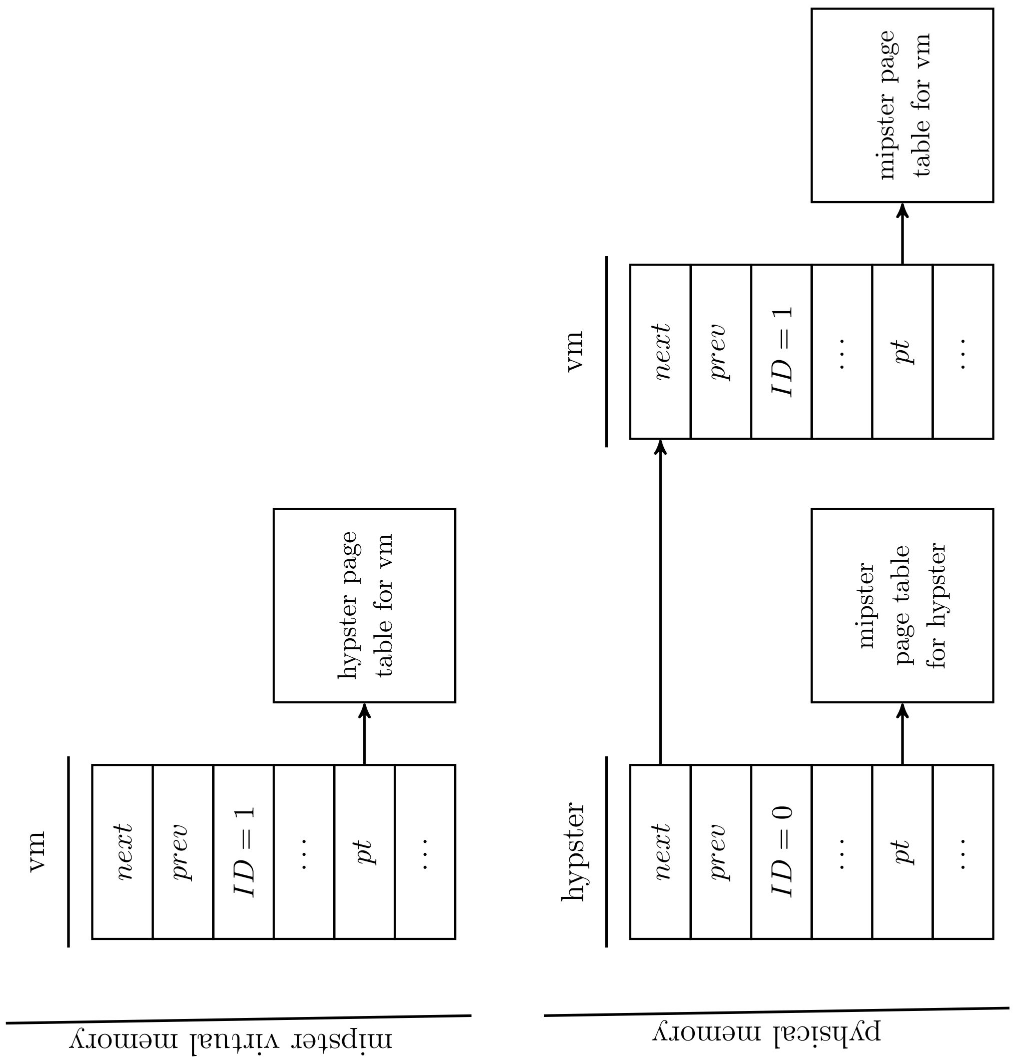


Figure 2.17: Machine contexts and page tables of Example 2.7.2.

CHAPTER 3

Implementation

3.1 Problem

So far, all the necessary information regarding *selfie* and its components have been discussed. Based on this system, *mixter* implements another new part of *selfie*. Compared to *mipster* and *hypster*, *mixter* should not be restricted on either emulating or hosting. In fact, *mixter* virtualizes a *MIPSter* machine through alternating between emulation and virtualization.

Figure 3.1 shows an example configuration of *selfie* executing a *mixter* instance on top of a *mipster* instance. The *mixter* emulator runs a program called *hello.c* which is written in the *C** programming language. Now, an explanation of emulation and virtualization in *mixter* based on the example of Figure 3.1 follows.

Emulation: *hello.c* is executed by *mixter*
In case of emulation, *mixter* acts as *mipster*. Therefore, the code of *hello.c* is interpreted by *mixter* as described in Section 2.4.2. Independent of emulation and virtualization, *mixter* creates a virtual address space to execute the *C** program and holds the page table for that virtual address space in its own address space which allows *mixter* to translate the virtual addresses. As a result, *mixter* is able to fetch and execute the code of *hello.c*. Furthermore, the context information is stored in a context created by *mixter* in its own address space.

Virtualization: *hello.c* is hosted by *mixter*
Notice that *mixter* needs to virtualize a *MIPSter* machine, therefore, it has to be executed on top of one. This can be achieved, for example, by running *mixter* on top of a *mipster* instance. The *mixter* instance acts conform to *hypster* when hosting another instance. Instead of interpreting the code like before with emulation, in virtualization, a context switch happens. This means that in the example of

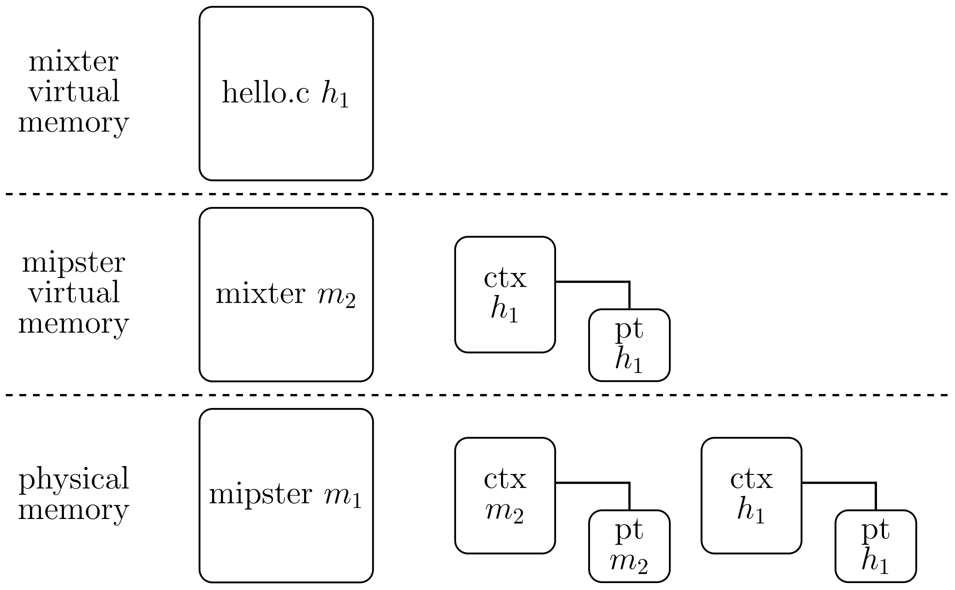


Figure 3.1: Address spaces, machine context and page table locations in a selfie configuration including mixter.

Figure 3.1, *mixter* tells the *mipster* instance below to execute *hello.c* for it. It is important that the emulator knows about the page table that is needed to translate the virtual address space where *hello.c* is located. Before the implementation of *mixter* as well as after, the emulator has a context and a page table in its address space for each *selfie* instance running on top of it. Therefore, the emulator is able to translate virtual addresses of all address spaces.

Even though the process of executing and hosting is different, emulation and virtualization of the same hardware is supposed to be functionally equivalent. The implementation of *mixter* is motivated by the task to find methods for verifying the functional equivalence of emulation and virtualization.

Another example configuration where *mixter* is used within *selfie* is depicted in Figure 3.2. Two *mixter* instances are illustrated that are running on top of a *mipster* instance. This figure shows another property of *mixter*, called self-referentiality. Like all other components of *selfie*, *mixter* should be able to run on top of another *mixter* instance.

It is important to know that *mixter* does not emulate and virtualize at the same time; instead, it can alternate between emulation and virtualization an instance at runtime. Figure 3.2 shows all possible execution scenarios where instances executed by *mixter* are underlined dotted and instances hosted by *mixter* are underlined dashed. Note that *hello.c* in the second line illustrates two different instances. First, hosted by *mixter*2 of the second line and second, executed by *mixter*2 of the third line.

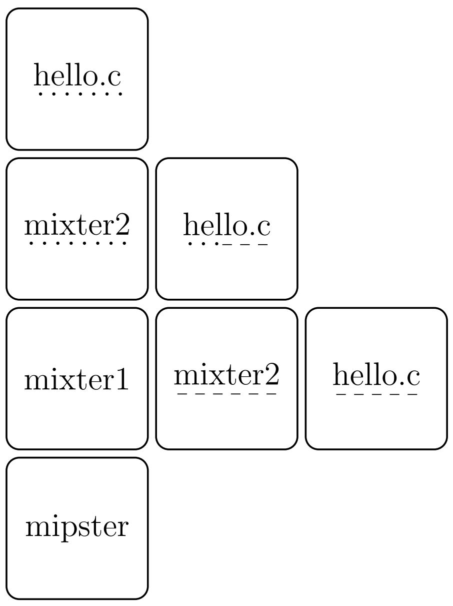


Figure 3.2: Example configuration of selfie using mixter.

As stated above, the context information is stored in different contexts in different address spaces depending on whether *mixter* is executing or hosting. The main task for introducing simultaneous emulation and virtualization is to operate in either way on the same context data. In the actual implementation of *selfie* a second context is created for each instance, via a system call, by the microkernel. Therefore, the emulator has all the necessary information to interpret all instances running on top of it. However, an instance on top of the emulator cannot access these contexts created by the microkernel. Therefore, another concept is needed that allows the emulator to access the contexts in the virtual address spaces of the instances running on top of it. These virtual contexts are located in different virtual address spaces. As a consequence, a technique is needed that allows the emulator to translate virtual addresses. A first attempt would be to translate a virtual address by looking up multiple page tables and translating the respective address address space by address space. When running several *selfie* instances on top of each other, each instance creates its own virtual memory space as shown in Figure 2.15. In this example, a virtual address in the *hypster* address space would be first translated into a virtual address of the *mipster* space. Further, the *mipster* virtual address is again translated into a physical address by another page table. This approach is complex and leads to an overhead for virtual memory accesses, especially for deeply nested virtual address spaces. In *mixter*, another concept to access virtual memory by the emulator, called context caching, is implemented. The details are described in Section 3.2.

Due to the implementation of context caching, the system calls to create a context and to map a page to a frame are obsolete and not used anymore. In fact, *mixter* goes one step further and gets rid of all other system calls except context switching. Hence, the microkernel is no longer part of the emulator in *mixter*. Concepts and implementation details regarding the extraction of the microkernel are explained in Sections 3.3, 3.4, 3.5, and 3.6.

3.2 Context caching

3.2.1 Problem

Context caching is used within *mixter* to enable the emulator to translate virtual addresses. As discussed in Section 2.8.2 there are always two machine contexts for each instance, one created by the microkernel in its own address space and one at the according virtual address space. This also includes all the data that are part of the machine context as well as registers and the page table.

A *mixter* instance is always running on top of at least one *mipster* instance. Consequently, an instruction is executed by the emulator in case that *mixter* is acting as a hypervisor. Still, *mixter* may act as emulator and executes the instruction itself. Therefore, it is necessary that in any case the system operates on the same context data which requires the emulator to access the virtual context of all instances running on top of it and translate their virtual addresses.

Basically, the emulator is able to translate every virtual address into a physical address. A virtual address would be translated address space by address space with multiple page tables until the physical address is reached as described in Section 3.1. Since this approach is complex and leads to a translation overhead when accessing virtual memory, *mixter* implements another technique called context caching.

The basic idea is simple, every time a context switch happens the emulator caches the virtual context of the context to be executed in a local copy, which is later on called "context restore". On the other hand after a change at a local copy the updates have to be written back to the virtual context, which is called "context save". So, the aim is to have consistent data between the locally cached context and the virtual context.

The following information of context may change throughout the execution and therefore have to be considered for saving and restoring:

- Program counter
- Lo register
- Hi register
- All other registers
- Program break

- Exception
- Faulting page
- Exit code
- all newly allocated pages (restore only)

The list of context information contains fields that was not introduced in Chapter 2, the reason is that this information was added during the implementation of *mixter*. All newly added fields will be introduced in the following sections with a detailed explanation of their purpose.

3.2.2 Implementation

The two main questions that occur while implementing context caching are where the context has to be cached and how?

As already stated above restoring the context is done with every context switch. Before the emulator loads the machine state of the context that is executed next it makes sure that it has the most recent data. The actual context switch is implemented by loading the values of the context and store them in the machine state variables (*pc*, *registers*, *loReg*, *hiReg*, *pt*). This can be seen in the implementation of the *doSwitch* function in Listing 1.

```
void doSwitch(int* toContext, int timeout) {
  int* fromContext;

  fromContext = currentContext;
  restoreContext(toContext);

  pc        = getPC(toContext);
  registers = getRegs(toContext);
  loReg     = getLoReg(toContext);
  hiReg     = getHiReg(toContext);
  pt        = getPT(toContext);

  if (getParent(fromContext) != MY_CONTEXT)
    *(registers+REG_V1) = (int) getVirtualContext(fromContext);
  else
    *(registers+REG_V1) = (int) fromContext;

  currentContext = toContext;

  // debugging code
  timer = timeout;
}
```

Listing 1: Restoring and switching the context in selfie.

Saving the context is done in two different positions in the code. First, after interpreting all the changes have to be written back to the virtual context, see Listing 2. Furthermore, this function shows how a context switch is implemented for *mipster*. *doSwitch* is implemented as shown in Figure 1. After that the emulator starts interpreting in *runUntilException*. In the last step, as already mentioned, the changes on the local context are saved in the virtual context.

```
int* mipster_switch(int* toContext, int timeout) {
  doSwitch(toContext, timeout);

  runUntilException();

  saveContext(currentContext);

  return currentContext;
}
```

Listing 2: Wrapper function for context switching of mipster.

Second, the context is saved in the *implementSwitch* function. As explained in Section 2.5.1 the implement function is invoked by the emulator, which saves the context of the currently executed instance. After that, the *doSwitch* function is called. The *cacheContext* function within the arguments of the *doSwitch* function tries to find the local context information of the context to be executed. In case there is no such context it allocates a new one for this instance.

```
void implementSwitch() {
  saveContext(currentContext);

  doSwitch(cacheContext((int*) *(registers+REG_A0)),
           *(registers+REG_A1));
}
```

Listing 3: Implement function of the context switch system call invoked by the emulator.

3.2.2.1 Save context

After clarifying where the context needs to be saved and restored, the remaining question is how the information is exchanged between local and virtual contexts?

This question will be answered with the implementation of the *saveContext* function shown in Listing 4. The procedure start with variable declaration and saving the machine context locally. Since it is not part of saving the context virtually it is commented out. The relevant code start with a condition that checks if the executing instance is the parent of the context that needs to be saved. In case that this conditions holds, the context does not need to be saved, since the parent already operates on the virtual context. On the other hand if the condition does not hold the local information should be copied to the virtual context. Therefore, the parent page table of the context that should be saved is looked up first. The parent page table is needed to access virtual addresses by translating them into physical addresses. For example, the parent page table is used to save the program counter which is shown in Line 11. *storeVirtualMemory* uses the page table of the parent and the virtual address of the program counter translates it and stores the local program counter in it. Most of the other context fields (LoReg, HiReg, ProgramBreak, Exception, FaultingPage, and ExitCode) are saved the same way.

While saving the registers it is important that not the pointer to the registers should be saved, but the value of each register. Therefore, after translating the virtual address to the registers a while-loop saves register by register, see Lines 18 – 22.

```
1  void saveContext(int* context) {
2    // variable declaration
3
4    // save machine state
5
6    if (getParent(context) != MY_CONTEXT) {
7      parentTable = getPT(getParent(context));
8
9      vctxt = getVirtualContext(context);
10
11     storeVirtualMemory(parentTable, PC(vctxt), getPC(context));
12
13     r = 0;
14     regs = getRegs(context);
15
16     vregs = (int*) loadVirtualMemory(parentTable, Regs(vctxt));
17
18     while (r < NUMBEROFREGISTERS) {
19       storeVirtualMemory(parentTable, (int) (vregs + r),
                          *(regs + r));
      r = r + 1;
    }

    storeVirtualMemory(parentTable, LoReg(vctxt),
                        getLoReg(context));
    // similar for HiReg, ProgramBreak, Exception,
    // FaultingPage, and ExitCode
  }
}
```

Listing 4: Information of a local context is saved in the virtual context.

3.2.2.2 Restore context

Rather than going through the source code of the *restoreContext* function two main concepts are pointed out.

First, Listing 5 shows how the program counter is restored from the virtual context. Similar to saving the context, the page table of the parent is necessary in order to resolve the virtual addresses to physical addresses. The process of restoring the context is similar to saving the context, but the other way around. *PC*(*vctxt*) gives the virtual address of the program counter of the virtual context. This address is translated by using the *parentTable* and the value is written in the locally cached context via *setPC*. Again, this is done for all other context fields (LoReg, HiReg, ProgramBreak, Exception, FaultingPage, and ExitCode) and the values of the registers are restored one by one in a loop.

```
setPC(context, loadVirtualMemory(parentTable, PC(vctxt)));
```

Listing 5: Restoring the program counter of a virtual context in the locally cached context.

Second, it has to be taken care of newly allocated pages. Following the approach of the 32 general purpose registers would lead to an obvious performance issue. The standard configuration of *selfie* uses a virtual memory size of 67108864 byte and the page size is 4096 bytes. Therefore, 16384 pages would have to be copied at every context switch.

In order to optimize the caching of the page table, *mixter* uses the fact that the page table is contiguous in memory and addresses are allocated bottom up and top down. The aim is to minimize the number of pages that have to be cached.

mixter introduces three new pointers to the context data, namely LoPage, MePage, and HiPage.

- The LoPage marks the page with the lowest address that was allocated bottom up since the last time the context was cached.
- The MePage marks the page with the highest address that was allocated bottom up since the last time the context was cached.
- The HiPage marks the next free page will be allocated top down.

With the help of these pointers caching the page table only maps pages between LoPage and MePage and between HiPage and the page with the lowest address top down. Here, bottom up means from low to high addresses and top down means from high to low addresses. Do not get confused by the figures which are drawn the other way around. Figure 3.3 and Figure 3.4 show an example of page table caching comparing the virtual and the local page table before and after caching.

Before caching, three new pages have been allocated bottom up and one new page was allocated top down. This can be seen in Figure 3.3 on the left hand side, since the difference between LoPage and MePage is three. HiPage would be the next free page for top down allocation, since there is a mapping at the HiPage address, one allocation happened.

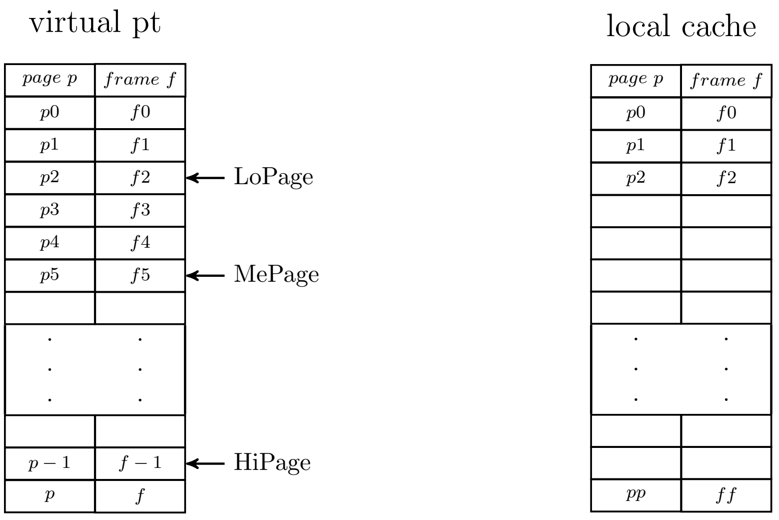


Figure 3.3: Example of caching a page table before caching.

Now, the algorithm of caching allocated pages is presented.

Starting at LoPage, for each page between LoPage and MePage the following steps are taken. First, the according frame of the actual page is loaded via the parent page table. Notice that the frame address is virtual and has to be translated via the parent page table itself. Second, the mapping of page to the translated frame is inserted in the local page table. After raising the page by one towards the MePage the same steps will be executed again until all pages up to MePage are allocated. Last, the LoPage pointer has to be set to the last allocated page.

The process for top down allocation is similar. Starting at the HiPage pointer, the frame of each page is loaded via the parent page table. As long as the frame is set to an address the page is mapped and therefore has to be mapped in the local cache. Like before, the virtual address of the frame has to be translated via the parent page table. After a page to frame mapping is inserted into the page table at the local copy of the context, the next lowest page of the virtual page table is handled and the process starts allover again. In the last step, the HiPage pointer is set to the next free page.

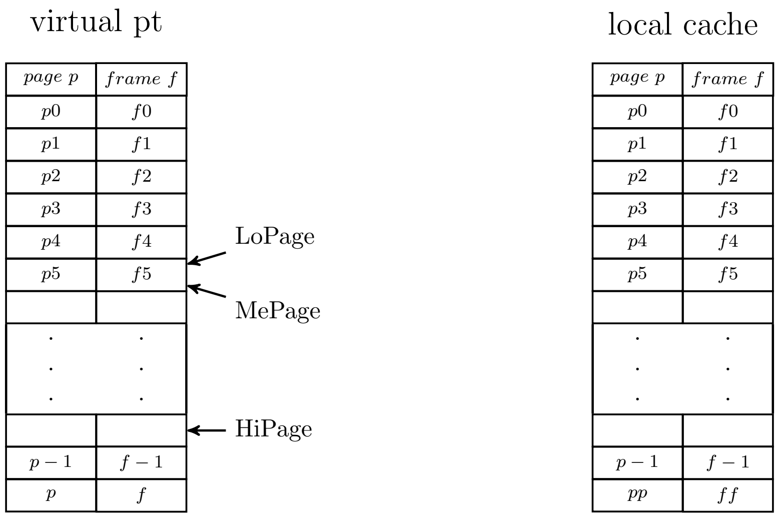


Figure 3.4: Example of caching a page table after caching.

Still missing is the handling of MePage which is not part of restoring the context. The pointer is set in *mapPage* a procedure that is used in *selfie* to map a given page to a given frame. The according code can be seen in the nested if statements in Listing 6. HiPage always remains the same until a context is cached. If a bottom up allocation happens, there are two cases of interest. On the one hand a page before LoPage could be allocated, then LoPage has to be set to that page to ensure that it will be cached at the next context switch. On the other hand a page above the actual MePage could be allocated, therefore the MePage has to be set to that page.

```
void mapPage(int* context, int page, int frame) {
  int* table;

  table = getPT(context);
  *(table + page) = frame;

  if (page != getHiPage(context)) {
    if (page < getLoPage(context))
```

```
      setLoPage(context, page);
    else if (getMePage(context) < page)
      setMePage(context, page);
  }

  // debugging code
}
```

Listing 6: Map a given page to a given frame in selfie.

3.2.3 Summary

mixter implements a concept called context caching that synchronizes the virtual context data and a local copy at the emulator address space. With every context switch the emulator restores all changes of the virtual context in the local context. After interpretation and for every context switch on top of the emulator, the local copy of the context is saved in the virtual context.

The implementation of context caching has two major implications for the microkernel of *selfie*:

- Memory management: The *map* system call was used to give the emulator the information about the page tables. Now, with the implementation of context caching a system similar to a memory management unit (MMU) is used within the emulator to translate virtual addresses. The page tables are fully cached within the emulator and therefore it has all necessary information. *map* is no longer used and removed.
- Context creation: Using context caching makes context creation via the microkernel obsolete. An instance on top of the emulator does no longer need to explicitly call the microkernel via a system call to create a duplication of a context. With every context switch the emulator ensures that it caches the latest information from a virtual context. Consequently, *selfie* got rid of the *context_create* system call.

Due to the implementation of context caching two of the system calls are already obsolete and removed. In fact, the implementation of *mixter* goes one step further by removing all system calls except context switching and therefore, extracting the microkernel from the emulator. The following sections describe all details regarding the system calls and the extraction of the microkernel.

3.3 Identifiers

3.3.1 Problem

selfie's microkernel design is inspired by the work of Jochen Liedtke[7]. In his work, he proposes a concept that identifies each task or thread with a unique identifier (uid). This concept was already implemented in original version of *selfie*. The microkernel within *mipster* holds a counter that is assigned for each created context. Whenever a new uid is needed, the actual counter value is returned as uid and after that the counter is incremented.

mixter comes up with a different approach. Instead of assigning an unique integer number to an instance, the virtual address of a context alongside a pointer to the parent context are used to identify an instance in *mixter*.

An important property of *mixter* is that an instance creates at most one virtual address space. As a consequence, the same virtual address with the same parent pointer cannot belong to different contexts. Without this property, the context identification concept of *mixter* would not work.

3.3.2 Implementation

Implementing the introduced concept of identifiers requires to add the identifier information to the context and set it appropriately.

3.3.2.1 Context information

Like ID before, the information about the identifier is part of the context in *mixter* as well. Therefore, ID can safely be removed and the virtual address of the context added. Notice that in Figure 2.11 from Chapter 2 the parent ID is already part of the context. However, since IDs are obsolete in *mixter* the relevant information for the identifier is a pointer to the parent context. Thus, the parent information has to be changed from an integer to a pointer.

3.3.2.2 Set identifier

Whenever an instance creates a new context in its address space, the identifier information is set within the context. Notice that there is a difference in the identifier information between the context in the virtual address space and the local copy at the emulator. While an instance sets the virtual address and the parent pointer to address 0 in the context created in the according address space, the local copy holds the virtual address of the context and a pointer to the local copy of the parent context. Therefore, an instance can check whether it is the parent of a context by comparing the parent pointer of the context to address 0. Only the parent that created the context in its address space will find address 0 as parent pointer. Theoretically, the emulator could also find address 0 at the local copy of the context in case that the local copy of the parent context is located at address 0 in physical memory. However, since lower addresses are most likely used by the underlaying operating system this case is improbable and neglected in *selfie*.

3.3.3 Summary

Instead of unique integer identifiers, as previously used in *selfie*, *mixter* changed the identifier of an instance to a combination of the virtual context address and the pointer to the parent context. The implication of these changes is that the microkernel acts no longer as a centralized system that creates unique identifiers for all instances.

Furthermore, as another consequence of the introduced identifier concept, *mixter* got rid of the ID system call. Calling ID asks the microkernel to return the identifier of the currently executed instance which is used to verify whether the currently executed instance is the parent of a given context. As explained in Section 3.3.2, *mixter* is able to verify whether an instance is the parent of a given context without invoking the microkernel. Therefore, the ID system call is obsolete and removed in *mixter*.

3.4 Additional context information

3.4.1 Problem

Regarding operating systems, the term context is used for the information that is needed to execute an instance. An instance could be either a process or a thread. A detailed view of which information a *selfie* context holds can be found in Section 2.4.4. During the implementation process of *mixter* the context information has changed a lot. For example, as discussed in Section 3.3.2 the ID of a context was replaced by the virtual address of the context and a pointer to the parent of the context.

Basically, a context consists of all the processor registers. In *mixter* the context is further used to hold some additional information. A *mixter* instance uses context caching to synchronize the virtual context data with a local copy of the context at the emulator address space, see Section 3.2. The following section will take a close look at the context implemented in *mixter*.

3.4.2 Implementation

Figure 3.5 shows the information that is stored in a context in the *mixter* implementation together with a brief description.

Compared to Figure 2.11 two context fields were removed (id and parent) and nine have been added (loPage, mePage, hiPage, exception, faultingPage, exitCode, parent, virtualContext, and name). Due to a new context identification concept

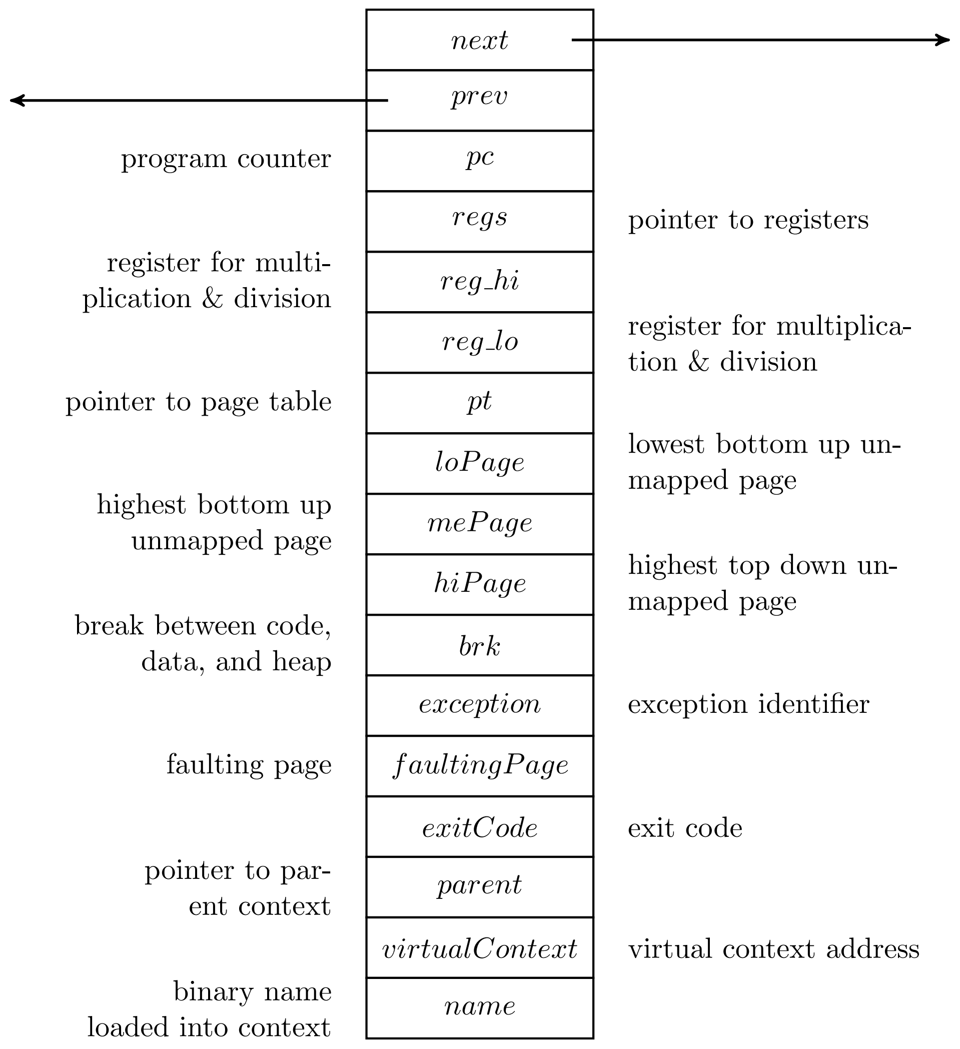

next
prev
program counter
pc
regs
pointer to registers
register for multiplication & division
reg_hi
reg_lo
register for multiplication & division
pointer to page table
pt
loPage
lowest bottom up unmapped page
highest bottom up unmapped page
mePage
hiPage
highest top down unmapped page
break between code, data, and heap
brk
exception
exception identifier
faulting page
faultingPage
exitCode
exit code
pointer to parent context
parent
virtualContext
virtual context address
binary name loaded into context
name

Figure 3.5: Structure of a context in mixter.

introduced in *mixter*, the id and parent field were removed from the context information, see Section 3.3. However, now follows a description of all the newly introduced context data fields:

- loPage, mePage, and hiPage:
  Pages in *selfie* are only mapped bottom up or top down. With the help of context caching a page that was mapped in a virtual page table will be mapped on the local copy of the emulator. Therefore, the information telling the emulator which pages should be mapped is synchronized via context data. The pointers loPage, mePage, and hiPage are used to mark the newly allocated pages. A detailed explanation of the caching process can be found in Section 3.2.
- exception:

Prior to the *mixter* implementation the machine status was part of the emulator and contained information telling the instances which exception occurred and what the exception parameter was. When an instance on top of the emulator wanted to access the value of the status a system call was needed. This was done by using the *status* system call. As a result the microkernel returned the current status of the emulator. Now, due to context saving which was implemented during the course of creating mixter, the emulator writes back the status value into the exception field of the virtual context. Therefore, instances on top of the emulator already know about the exception identifier.

- faultingPage:
  In the original *selfie* implementation the machine status included the exception identifier as well as exception parameter which may have contained the address of the faulting page. Now, exception identifier and the faulting page are two separate 32 bit integers and are both part of the context. Like exception, the information about the faulting page is distributed via context saving and therefore instances can access the data.

- exitCode:
  Another information that have to be available at each instance is the exit code. Again, this is achieved by adding the data to the context and context caching.

- parent and virtualContext:
  The virtual context address, together with a pointer to the parent context, introduces a new concept to identify a context and replace the ID context data. The name parent still stays the same, but rather than holding the ID of the parent context, a pointer to the parent is stored now. The concept is explained in detail in Section 3.3.

- name:
  In order to give correct debugging messages at each instance, the name of the loaded binary file becomes also part of the context data.

3.4.3 Summary

The main purpose of adding information to the context is to synchronize the context data between the virtual context and the local copy of the context. Since the context includes information about the exception number as well as the exception parameter which is used for the faulting page address, an instance on top of the emulator already knows about all the information the *status* system call would provide. As a result, the status call is obsolete and can safely be removed. This implies that an instance on top of the emulator is able to handle the exception of its children without using a system call.

3.5 Context switch

3.5.1 Problem

By using the techniques described in the last sections, namely context caching, context identifiers, and additional context information, *selfie* almost got rid of all hypster calls. The only one that is left is context switching.

Before a processor begins to interpret code, the context information, for example program counter, page table pointer, etc., of a given process p_1 is loaded into the processor registers. According to this context information the processor starts fetching and executing instructions. Notice that during the interpretation of code, the values in the processor registers may change. As a consequence, the processor registers have to be saved in the context of the executing process when the processor stops interpreting. In order to interpret another process p_2 the processor performs a context switch as follows. First, the processor has to save the actual value of all processor registers back into the context of process p_1. Therefore, the processor is able to start interpreting p_1 at the same state at which it had stopped before the context switch happened. Second, the context information of process p_2 is loaded into the processor registers. Now, the processor has the information needed to interpret process p_2.

In *selfie* context switching is implemented as a system call. Whenever a context switch occurs, the microkernel is invoked and performs the context switch mechanism as described above. Since the microkernel is located within the emulator, a context switch happens between interpreting two processes. Before the next instruction is interpreted, the context information is fully restored in the processor registers.

3.5.2 Proposal for solution

Getting rid of the context switch system call would require that context switching happens on top of the emulator while interpreting a process and not within the emulator by invoking the microkernel. Consider again two processes p_1 and p_2. The emulator is interpreting process p_1 and would like to switch to process p_2. The functionality of context switching is now part of the code of the processes p_1 and p_2 which are interpreted by the emulator. Applying the mechanism on top of the emulator would not work straight away. The *MIPSter* instruction set only allows to load one value at a time into a processor register. Therefore, the context information is loaded into the processor registers one by one which leads to an inconsistent state of the processor registers. Loading the program counter and the page table pointer into the according processor registers reveals the problem. There are two possibilities, loading the program counter first or loading the page table pointer first. Assuming that p_1 loads the program counter of p_2 first, the emulator would execute the next instruction of process p_2. While fetching, the instruction of process p_2 is translated with the page table of process p_1 which is incorrect. On the other hand, assuming p_1 loads the page table pointer first, the emulator would execute the next instruction of process p_1 but would try to translate it using the page table of process p_2. Again, the instruction could not be fetched.

As a consequence, at least the program counter and the page table pointer need to be exchanged atomically. This implies, that the *MIPSter* instruction set has to be extended by instructions that fulfill these requirements. However, this task is not implemented in *mixter* and remains future work.

3.6 Exceptions

3.6.1 Problem

So far, the focus of the implementation discussion was on hypster calls. As already mentioned, *selfie* includes five more system calls for the built-in library. The idea is that the parent should be concerned with handling the *exit*, *open*, *read*, *write*, and *malloc* calls; whereas before the microkernel needed to be invoked.

Therefore, *mixter* provides a technique using exceptions to handle the build-in library functions. Since the parent is in charge of handling exceptions, the microkernel basically throws an exception instead of executing the implement function for the caller. Therefore, a context switch to the parent happens. Due to context caching the parent has all the information that is needed to handle the exception from its children and can execute the according implement function itself instead of the microkernel.

3.6.2 Implementation

The system call is still invoked like before in the according emit function. For example, the emit function of exit stays entirely the same, see Listing 3. The instructions in Line 9 and Line 10 are similar for all system calls. First, the system call number is stored in register $v0$ and then, the system call is invoked. The difference lies in the way the interpreter handles the system call that has been invoked. This can be seen in Listing 7. Since the context switch call is still handled by the emulator, the difference can be seen clearly. Instead of calling the implement function, the emulator throws an exception. Since the parent is in charge of exception handling, a context switch to the parent happens.

For all kinds of instances, namely *mipster*, *hypster*, and *mixter*, additional functionality has to be added in the exception handling. If the exception is caused by a built-in library function, the exception handler has to check the system call number stored in register $v0$. This number clearly identifies all former system calls. To handle the system call, the according implement function is invoked.

```
void fct_syscall() {
  // debugging code

  if (interpret) {
    pc = pc + WORDSIZE;

    if (*(registers+REG_V0) == SYSCALL_SWITCH)
      implementSwitch();
    else
      throwException(EXCEPTION_SYSCALL, 0);
  }
}
```

Listing 7: Code how the interpreter handles a system call instruction.

Minor changes have been made to the implement function. Before, the implement function was only invoked by the emulator which has the relevant information about the calling instance in its registers. Due to context caching the parent can access the recent information about the child context in its memory. Therefore, all context accesses are changed in a way that is shown in Listing 8. Instead of reading the values from the emulated machine registers, the data is accessed by the parent via its child context.

```
// before: access via machine registers
size  = *(registers+REG_A2);
vaddr = *(registers+REG_A1);
fd    = *(registers+REG_A0);

// after: access via saved context information
size  = *(getRegs(context)+REG_A2);
vaddr = *(getRegs(context)+REG_A1);
fd    = *(getRegs(context)+REG_A0);
```

Listing 8: Changes needed for context data access in an implement function of a system call.

The implement functions of *open*, *read*, and *write* are implemented by calling their own function. For example, *read* is implemented by calling the *read* function. This results in a top down recursive exception handling explained in the following example.

Suppose a system where a *mixter* m_3 is running on top of another *mixter* m_2 and m_2 is running on top of a *mipster* m_1. Initially, m_3 wants to read from memory by calling the *read* function. As explained above, the microkernel throws an exception. m_2 is in charge to handle this exception and calls the implement function of *read*.

Since there is another instance below m_2 the microkernel throws an exception like before. Again, m_1 handles the exception by calling the implement function of *read*. Now, at the emulator m_1 a read on the physical memory is performed.

3.6.3 Summary

By using exceptions, a concept was introduced that allows the parent to handle all built-in library calls from its children. It is important to notice that the call in the end may still be handled by the emulator. Like in the *read* example from above, the physical read on memory is done by the emulator, but the call is handled parent by parent top down. As a result, the built-in library is no longer implemented via system calls.

All system calls except context switching are no longer used in *selfie*. As a consequence, the microkernel is no longer located within the emulator. All functionality of the former system calls is now handled by the parent instance or it is obsolete.

CHAPTER 4

Conclusion and future work

4.1 Conclusion

This thesis presented *mixter*, an experimental hypervisor that is capable of simultaneously emulating and virtualizing a *MIPSter* machine. In machine emulation, the functionality of a specific hardware is imitated by software. One way to implement machine emulation is through code interpretation. In system virtualization, virtual instances of the hardware on which it runs are created. This can be achieved by a form of context switching and virtual memory. However, emulation and virtualization of the same hardware are supposed to be functionally equivalent.

The purpose of *mixter* is to get closer towards identifying methods for the verification of the functional equivalence of emulation and virtualization. Even though *mixter* cannot verify the equivalence, it is capable of virtualizing the machine on which it runs by alternating between emulation and virtualization at runtime.

In both cases, no matter if emulation or virtualization, it is essential to operate on the same context data. Therefore, a concept called context caching was introduced. This concept is used to synchronize the context information of a context stored in a virtual address space and the according context stored at the emulator. Furthermore, with the implementation of *mixter selfie* got rid of all system calls except context switching. Again, this was achieved by introducing a new concept for identifying an instance, handling exceptions, enhancing context information and caching contexts. The next steps towards the verification remain future work and are briefly described in the following section.

The software of *mixter* is implemented as part of *selfie*, a software system that is used for educational purposes. The *selfie* system consists of a self-compiling compiler (*starc*), a self-executing emulator (*mipster*) and a self-hosting hypervisor (*hypster*). Similar to all other parts of *selfie*, *mixter* is said to be self-referential, since it can emulate and host another *mixter* instance on top of it.

4.2 Future work

As shown in Chapter 3 there is still one system call left in the implementation of *mixter*. Getting rid of the context switch system call cannot be achieved by any of the steps taken to implement *mixter*. Performing a context switch on top of the emulator leads to new challenges as explained in Section 3.5. During a context switch, the machine registers of the emulator may hold information of different contexts. As a consequence, the emulator is in an inconsistent state and thus is not able to proceed the execution. In order to solve these challenges it is most likely necessary to extend the *MIPSter* instruction set. However, one design criteria of *mixter* was to keep *selfie* as simple and minimalistic as possible. Therefore, removing the last system call remains to be implemented in the future.

A further step towards the verification of the functional equivalence of emulation and virtualization would be to execute each instruction twice. Instead of alternating between emulation and virtualization *mixter* should do both for each instruction. Therefore, the machine state could be compared at instruction level. Special care has to be taken during the execution of file operations like read, write, and open. Here, the instructions should be executed twice as well, but the effects on the file should only happen once.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Krste Asanovic Andrew Waterman. The RISC-V Instruction Set Manual: Volume I: User-Level ISA. SiFive Inc.; CS Division, EECS Department, University of California, Berkeley, 2.2 edition, May 2017.

[2] Krste Asanovic Andrew Waterman. The RISC-V Instruction Set Manual: Volume II: Privileged Architecture. SiFive Inc.; CS Division, EECS Department, University of California, Berkeley, 1.10 edition, May 2017.

[3] James Goodman and Karen Miller. A Programmer's View of Computer Architecture with Assembly Language Examples from the MIPS RISC Architecture. Saunders College Publishing, Philadelphia, PA, USA, 1993.

[4] Christoph Kirsch. Introduction to compiler construction. Lecture Notes.

[5] Christoph Kirsch. Selfie: Computer Science for Everyone. Leanpub, 1 edition, 2016. Work in progress.

[6] Christoph M. Kirsch. Selfie and the basics. In Proceedings of the 2017 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017. ACM, 2017.

[7] Jochen Liedtke. On micro-kernel construction, volume 29. ACM, 1995.

[8] MIPS Technologies, 1225 Charleston Road, Mountain View, CA. MIPS32 Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture, 0.95 edition, March 2001.

[9] MIPS Technologies, 1225 Charleston Road, Mountain View, CA. MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set, 0.95 edition, March 2001.

[10] MIPS Technologies, 1225 Charleston Road, Mountain View, CA. MIPS32 Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture, 0.95 edition, March 2001.