Port of Mimiker Operating System for AArch64 Architecture

(Port systemu operacyjnego Mimiker na architekturę AArch64)

Paweł Jasiak

Praca licencjacka

Promotorzy: mgr Krystian Bacławski
dr Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

20 czerwca 2021

Abstract

Many processor families are available today. In my work, I will present the process of preparing an operating system port for a new family of processors on the example of a Mimiker port for the AArch64 architecture. As this is the first port of that system, it required some extra work to separate the architecture-dependent parts from the independent ones. The port requires modification of many critical parts of the kernel: handling processor exceptions, managing the address space of processes, exchanging data between kernel threads and user programs. In addition, it was needed to prepare a set of tools that support the selected architecture, such as a compiler and a hardware debugger. The result of this work is the ability to run the Mimiker system for the first time on widespread hardware - the Raspberry Pi 3.

Obecnie dostępnych jest wiele rodzin procesorów. W mojej pracy przedstawię jak wygląda proces przygotowania portu systemu operacyjnego na nową rodzinę procesorów na przykładzie portu systemu Mimiker na architekturę AArch64. Jako, że jest to pierwszy port tego systemu, wymagał on dodatkowej pracy związanej z rozdzieleniem części zależnych od architektury od niezależenych. Port wymaga modyfikacji wielu krytycznych części jądra: obsługa wyjątków procesora, zarządzanie przestrzenią adresową procesów, wymiana danych pomiędzy wątkami jądra a programami użytkownika. Ponadto należało przygotować zestaw narzędzi, które wspierają wybraną architekturę jak kompilator oraz sprzętowy debuger. Rezultatem tej pracy jest możliwość uruchomienia po raz pierwszy systemu Mimiker na ogólnodostępnym sprzęcie – Raspberry Pi 3.

to Wiktor, who showed me the way

Contents

Chapter 1

Introduction

Nowadays we have multiple architectures of CPU. The most popular for customers are `x86_64` and `AArch64`. The first one is used in most personal computers since 2003. The second one was created in 2011 and is used mostly for smartphones and IoT devices but last year Apple migrated their devices to that architecture. It is expected that ARM architecture will become even more popular in the following years.

From a software engineer's point of view target architecture for software usually doesn't matter – we have a lot of abstractions over hardware and operating systems. But somebody needs to create these abstractions.

In my thesis, I will guide the reader through the abstractions that need to be created in an operating system for a new architecture.

I assume that the reader knows basic concepts from the standard course of computer system architecture and basic facts about Unix kernel design.

The structure of this thesis is as follows: this chapter provides an introduction to Mimiker operating system and explains what is a port. Chapter 2 describes the most important facts about Raspberry Pi 3 board, it is a target of my port. In Chapter 3 I explain the most important changes in MIPS and machine-independent code of Mimiker. Chapter 4 describes implementation details for AArch64 port. In Chapter 5 there are instructions how to run Mimiker of Raspberry Pi 3 and the challenges of switching from an emulator to physical hardware. Chapter 5 is a summary of my work and proposals for the future.

1.1 CPU architecture

The central processing unit is the most essential part of every computer. It performs basic operations like arithmetic for the rest of devices. Each CPU provides an interface for software called instruction set architecture – ISA. ISA is standardized for each CPU family. Each program that runs on the CPU is translated (directly on indirectly) from its original source code to binary code which encodes instruction understandable for a given CPU.

But CPU architecture is something more than an instruction set. CPU also provides a standardized way (for each family) for things like interrupt handling and virtual memory management.

In my thesis, I will show the most essential parts of CPU architecture that kernel developers should become aware of.

1.2 Mimiker

Mimiker is a research operating system inspired by the world of Unix, and in particular by its *BSD flavour. The main effort of the project is currently improving its kernel to the point it supports more Unix userspace programs [16].

It is a fully open-source project developed at the University of Wrocław.

Originally Mimiker was written for the Malta board which contains CPU belonging to the MIPS family.

MIPS architecture was introduced in 1986 but it lost popularity in last years. As a result in March 2021, MIPS announced that the development of the MIPS architecture had ended.

Unfortunately, it is very difficult to get a working Malta board, so we decided to rewrite Mimiker to a new architecture. AArch64 is our choice because it is easily accessible for everyone by cheap boards like Raspberry Pi which have become very popular in recent years.

The result of my thesis is fully working port for that board and run Mimiker on a physical machine for the first time ever.

1.3 What is a port?

As mentioned in section 1.1 each program needs to be translated into binary code that is understandable by the CPU. We also know that different families of CPUs can have different instruction sets and different interfaces for hardware management. When a kernel is compiled it can't be run on different CPUs, because they do not understand binary encoding of instructions. Even if we generate the correct encoding of instructions the hardware interface could be different. Port of an operating system is a special version of kernel and user-space programs adapted to specific architecture with drivers for devices used by the new hardware.

Each port should contain only a minimal subset of machine-dependent code which is used by the rest of the kernel through the hardware abstraction layer (HAL), the reason behind that is simple – more lines of code means more potential defects.

HAL is a set of interfaces that are needed to be implemented by device drivers and by a port that allows writing generic code in the kernel. This set also implies how to write a new driver for a given system. It speeds up the process of implementation support for new hardware because we have already provided high-level interface of our driver.

Portability is a very important feature of code. Portable code can be easily reused on newer hardware – this saves the developer's time and reduces resources needed to create a new version of the code. Portable systems are more likely to gain popularity.

1.4 What is needed for a port?

First of all, we need a toolchain that allows creating binaries for a given architecture. In our case, it will be the gnu toolchain with GCC compiler. We also need a set of tools for testing and debugging code. As a software debugger, we use gdb. The emulator of Raspberry Pi 3 of our choice is QEMU. For more information about toolchain see 5.1.

We also need to find which subsystems need to be rewritten for supporting new architecture. Here is a short introduction to the most important parts.

Kernel bootstrapping is the first phase of kernel initialization. We need to configure the CPU and enable MMU before jumping to machine-independent part of the code. It is the most sensitive part of machine-dependent code. We need to do that only once during system startup but every decision made during bootstrapping will affect the behavior of the hardware. For example, we can define how the CPU should react to unaligned accesses.

Interaction with MMU is required for the virtual memory concept. In our case, it is handled by the pmap module which is described at 4.3.

We need to be able to react for external events e.g. key pressing on keyboard. They are passed as CPU exceptions. It is described at 4.6.

Context switching allows us to run multiple programs in parallel. It is machine-dependent because each CPU can have a different set of general-purpose registers. It is described at 4.7.

Kernel without user-space processes is useless. Basic interaction with them needs to be machine-dependent because of different ISAs between CPUs. It also uses exception handling – as a way for system calls – which is also specific for CPU.

These interaction are described at 4.2.

A system that can only use CPU can't do anything interesting. External devices are important to interact with the world. Since we have working terminals in Mimiker we want to implement a driver for UART for communicating with the system. The second important device is the timer – without it we do not have any possibility for time measurement which is important for process management. For drivers implementation see 4.8.1.

Chapter 2

RPI 3

Raspberry Pi is a small single-board computer developed by the Raspberry Pi Foundation in association with Broadcom [25]. It is a simple, easily accessible platform. In my work, I base on Raspberry Pi 3 with BCM2837 board [21].

Here I will describe basic information about the hardware required for the porting process. It includes facts about the memory management unit in AArch64 architecture, CPU exceptions needed for external events handling, timer interface, and PL011 UART device.

2.1 CPU

In this section, I introduce important facts about quad-core ARM Cortex-A53 CPU used by Raspberry Pi 3 [22].

That CPU implements ARMv8-A 64-bit instruction set [2] which implements 64-bit extension of the ARM architecture called AArch64. In this work, we will be using these terms interchangeably.

2.1.1 MMU

The memory management unit is responsible for translating addresses from virtual to physical memory.

In modern operating systems each process has its own separate address space. From a process point of view, no other process exists and every address belongs to that process. The only requirement for using given address of memory is system call for kernel that makes given address valid. In that scenario, multiple processes can have the same chunk of memory, but they shouldn't share that memory. For that purpose, we have the concept of virtual memory. For each memory address used by a process, we have a mapping from virtual address to a physical address. To achieve that we need support from hardware and it is the role of the MMU. MMU does address translation and the kernel manages where virtual memory region of the process should be mapped to physical memory. For more information about role of memory management unit and virtual memory see [14].

For that purpose, we use page table data structure.

Page Table layout

For general information about how page table works see [14]. In my thesis, I'm using 4 levels page table (levels from 0 to 3) where the third level contains final mapping into physical memory.

We use 4 levels because our CPU supports that and we want to be able to use as much memory as possible. The newer version of Raspberry Pi supports up to 8 GiB of RAM which is not addressable by 2 levels page table used by the MIPS version of Mimiker.

bits [47:39]	bits [38:30]	bits [29:21]	bits [20:12]	bits [11:0]
Level 0 index	Level 1 index	Level 2 index	Level 3 index	Offset on page

Table 2.1: Virtual address format

Each PDE (page directory entry) has 512 64-bits entries and needs exactly one page (4096 bytes). We can see that in this way it is possible to address 256TiB by a single page table.

Level 0	Level 1	Level 2	Level 3
256TiB	512GiB	1GiB	2MiB

Table 2.2: Page table mapping size

This table describes how much memory can be addressed using a page table on a given level. Let's say that we have single page at second level. The page has 4096 bytes of memory. A single entry in the page table always has 8 bytes (in our CPU) which means we have 512 entries. The same goes for the third level. A single entry describes one page which is 4096 bytes of memory. Finally, we have $512 \times 512 \times 4096$ bytes of memory which is 1GiB.

In addition to mapping, page table contains permission bits for each page. We're interested in the following bits which are provided by the memory management unit in our CPU. They are translated to user-friendly format in 2.3:

- *AF* [10] - access permission; without that bit, every access to page triggers CPU exception; it is used by Mimiker for tracing accesses for every page; for more see 4.3.5;

- *USER* [6] - unprivileged permission; without that bit, every access to page from exception level 0 triggers CPU exception;
- *RO* [7] - read only; with that bit every write access to page triggers CPU exception;
- *UXN* [54] - unprivileged execution never; with that bit, execution access to page from exception level 0 triggers CPU exception; for information about exception levels see 2.1.2;
- *PXN* [53] - privileged execution never; with that bit, execution access to the page from higher exception levels triggers CPU exception;

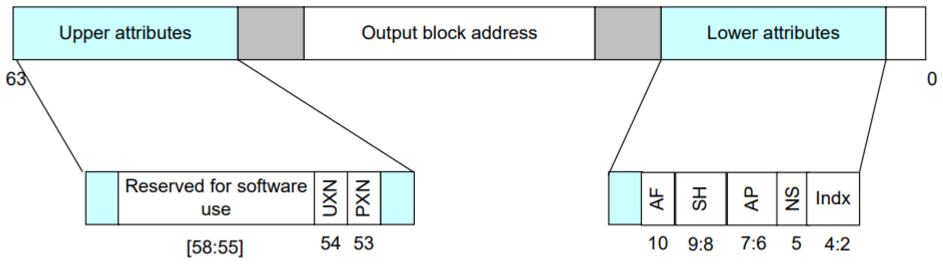


Figure 2.1: Page table entry

For more details see [3].

access	*AF*	*USER*	*RO*	*UXN* & *PXN*
user read	1	1	*	*
user write	1	1	0	*
user exec	1	1	*	0
kernel read	1	*	*	*
kernel write	1	*	0	*
kernel exec	1	*	*	0

Table 2.3: Protection map

This table describes which access bits need to be set in the page table entry for successful memory translation by MMU. Other configurations cause memory fault.

Two different page tables can be active at the same time. The addresses of page tables are located at `ttbr0` and `ttbr1` CPU registers. The first page table is dedicated to user-space, the second one for kernel-space.

CPU decides which page table will be used based on the highest bits of virtual address. Of course, CPU must be in the correct exception level to use that page table.

2.1.2 Exception levels

In ARMv8 execution takes place on one of the exception levels. A higher exception level means fewer privileges for executed code. It is a common practice to give a program access only to necessary resources. In Mimiker we use exception levels for separate user-space and kernel-space threads.

When an exception occurs, CPU jumps to a special procedure defined separately for each kind of exception and for each exception level. We can implement interrupt handlers or system calls using this mechanism.

There is four exception levels on AArch64

- EL0 – application; usually it is an exception level where user-space lives
- EL1 – kernel; usually it is an exception level where kernel-space lives
- EL2 – hypervisor; used by the hypervisor
- EL3 – firmware; reserved by low-level firmware and security code

These exception levels determine privileges for memory and registers access.

Mimiker uses EL3 & EL2 levels only to configure EL1 at the beginning of kernel bootstrap code. Next, we will switch to EL1 where the kernel lives and to EL0 for user-space programs.

2.1.3 Exceptions

In AArch64 we have two main kinds of exceptions 4.2.

- synchronous
- asynchronous

For each type of exception, we need to define a special function called exception handler that will be executed when the exception occurs.

These functions need to be known when the exception occurs. There is an exception vector structure that contains these functions and this vector is stored at `vbar` register.

Interrupts

Interrupts are asynchronous events generated from the outside world. A good example is the timer tick. When we configure the timer we say that in x time units we want to generate an interrupt. It can be used not only for time measurement, but also for scheduling. After x ticks, we want to decide if we want to change the running thread to another thread.

It is important that interrupt can appear any point at time, even in the middle of another instruction. So we can't do any special assumptions about the state of CPU.

Traps

Traps are synchronous exceptions generated by special instructions. For example, if we want to transfer control to kernel-space from user-space we can use `svc` instruction which generates trap – in fact, we use that instruction for the implementation of system calls in Mimiker.

Aborts

Aborts are synchronous exceptions generated by instructions but unlike traps, they are not intended. hey can occur as a result of wrong access to memory..

2.1.4 Timer

Our CPU provides an ARM timer. We do not need any advanced features of timers so let's discuss only the basics.

We have the following registers

- `cntpct_el0` – it contains current value of timer
- `cntp_ctl_el0` – this register controls if timer is enabled
- `cntp_cval_el0` – compare register; if current value of timer will be greater of equal then timer sends interrupt to CPU

With that knowledge, we can implement a driver for that timer 4.8.2. It is required for task scheduling. We want to give a time slice for a process when it can run on CPU. After that, we want to decide which process should be run next. For that we use timer interrupts which trigger the scheduling subsystem.

2.2 UART

On Raspberry Pi 3 there is PL011 UART [15].

This universal asynchronous receiver/transmitter provides:

- separate 16x8 transmit and 16x12 receive FIFO memory
- programmable baud rate generator
- standard asynchronous communication bits
- false start bit detection
- line break generation and detection
- support of the modem control functions CTS and RTS
- programmable hardware flow control
- fully-programmable serial interface characteristics:
  - data can be 5, 6, 7 or 8 bits
  - even, odd, stick or no-parity bit generation and detection
  - 1 or 2 stop bit generation
  - baud rate generation

This UART generates two interrupts:

- UARTRXINTR – the transmit interrupt
- UARTRTINTR – the receive interrupt

These interrupts are necessary to implement a terminal over that UART. For now terminal over UART is the only way to run interactive user sessions with shell in Mimiker.

The last important part is the location of registers used to control PL011. All of them are listed in [15].

Chapter 3

Preparing the system for porting process

In this chapter, I will describe the most important changes that have been done before we could implement AArch64 support for Mimiker.

It starts with a short dictionary of the most frequently used C constructions in the Mimiker codebase. Next, I will go through changes in the CPU context representation, the memory mapping subsystem, kernel address sanitizer routines, and new hardware abstraction layout over UART devices.

3.1 Dictionary

We use some specific constructions in our code. Here I will introduce those that might not be obvious.

The following macros reads and writes data to special registers using `msr` and `mrs` instructions.

```
READ_SPECIALREG(reg);
WRITE_SPECIALREG(reg, val);
```

`WITH_INTR_DISABLED` disables interrupts under the next scope. They will be enabled at the end of scope. It also takes care of nested calls – we maintain an internal counter.

```
WITH_INTR_DISABLED {
    ...
}
```

`WITH_MTX_LOCK` acquires mutual exclusion lock (mutex) for everything under the next scope. This lock will be auto-released.

```
WITH_MTX_LOCK(&lock) {
    ...
}
```

`SCOPED_MTX_LOCK` acquires mutex for current scope. This lock will be auto-released at the end of scope.

```
SCOPED_MTX_LOCK(&lock);
```

The same but for spin locks.

```
WITH_SPIN_LOCK(&lock) {
    ...
}
```

Wait on given conditional variable with given lock.

```
cv_wait(&cv, &lock);
```

Wake a single thread waiting on a conditional variable.

```
cv_signal(&cv);
```

Macro for `for` that iterates over a list.

```
TAILQ_FOREACH(var, head, name);
```

Insert an element at the end of the list.

```
TAILQ_INSERT_TAIL(head, elm, name);
```

Remove an element from the list.

```
TAILQ_INSERT_TAIL(head, elm, name);
```

For detailed API for lists see NetBSD manpages [23]. For mutexes see [26], for spin locks see [27] and for conditional variables see [28].

3.2 CPU context

CPU context contains CPU registers. We need to know how to manipulate context mostly for two actions.

- context switch
- exception handler

The context switch is a procedure of switching currently running thread into another. Each thread has a given time slice when it can be run on the CPU. After that period scheduler must decide which thread can replace the running thread and the dispatcher performs that change. It is a very important for modern operating systems. These days we are running hundreds of threads on our daily systems. For example, my system was running 637 threads in time of writing and I have only 8 available cores. But from the user point of view, every process is running simultaneously by the operating system.

The second action is when a CPU exception occurs. Then we need to save the context of the running thread, go to the exception handler in the kernel, handle that exception and at the end restore the context of the interrupted thread.

At the beginning of my work there were a several different CPU context representations:

- `struct sigcontext`

- `struct __ucontext`
- `struct mcontext`
- `struct ctx`
- `struct exc_frame`
- `jmp_buf`
- `sigjmp_buf`

Each of them has a similar structure that contains all CPU registers as individual fields or grouped by arrays, for example:

```
struct sigcontext {
  int sc_onstack;   /* sigstack state to restore */
  sigset_t sc_mask; /* signal mask to restore */
  int sc_pc;        /* pc at time of signal */
  int sc_regs[32];  /* processor regs 0 to 31 */
  int sc_mullo;
  int sc_mulhi;     /* mullo and mulhi registers... */
  int sc_fpused;    /* fp has been used */
  int sc_fpregs[33];/* fp regs 0 to 31 and csr */
  int sc_fpc_eir;   /* floating point exception instruction reg */
};
```

Listing 1: sigcontext structure

Now all of them are unified into one:

```
typedef uint64_t __greg_t;
typedef __greg_t __gregset_t[_NGREG];

typedef struct {
  union __freg {
    uint8_t __b8[16];
    uint16_t __h16[8];
    uint32_t __s32[4];
    uint64_t __d64[2];
  } __qregs[_NFREG] __aligned(16);
  uint32_t __fpcr; /* FPCR */
  uint32_t __fpsr; /* FPSR */
} __fregset_t;

typedef struct mcontext {
  __gregset_t __gregs; /* General Purpose Register set */
  __fregset_t __fregs; /* FPU/SIMD Register File */
  __greg_t __spare[8]; /* future proof */
} mcontext_t;

typedef struct ctx {
  __gregset_t __gregs;
} ctx_t;

❶struct __ucontext {
  unsigned int uc_flags;  /* properties */
  ucontext_t *uc_link;    /* context to resume */
  sigset_t uc_sigmask;    /* signals blocked in this context */
  stack_t uc_stack;       /* the stack used by this context */
  mcontext_t uc_mcontext; /* machine state */
};
```

Listing 2: context representation (mix of `ucontext.h` [31] and `mcontext.h` [32])

Now we use only `ucontext_t` ❶ for context representation. It is not the most optimal solution but it is the simplest solution – we do not need to convert one representation into another and code is much more clean.

This change also allows us to fix problems with nonlocal jumps which were broken because of different representation of context between user-space and kernel-space in `setcontext` system call.

3.2.1 FPU context

FPU context contains only floating point registers. We do not use any of them inside kernel code but we want to support them in user-space so we need to take care about them during context switching.

On the other hand we do not want to save & restore FPU context with every exception or context switching. We want to do that only if it is really needed. For each thread we have special flags that express in which state is FPU for given thread:

- `TDP_FPUCTXSAVED` – FPU context was saved by `ctx_switch`.
- `TDP_FPUINUSE` – FPU is in use and its context should be saved & restored on demand.

When new user-space thread is spawned FPU is disabled. First access to floating point unit triggers exception and kernel enables FPU for that thread and set `TDP_FPUINUSE` flag.

We do not need to save FPU context during exception handling because usually we will go back to user-space with the same thread and we have a guarantee that kernel doesn't touch any FPU register. But it is possible that we do context switch to other user thread. In that scenario we need to save FPU context for old thread during `ctx_switch` procedure. Finally when we go back to user-space in `user_exc_leave` we check if `TDP_FPUCTXSAVED` is set and restore FPU context if it is needed.

3.3 Pmap

In this section I will describe the most important changes in pmap module.

The physical-mapping module (pmap) manages machine-dependent address translation and page tables that are used either directly or indirectly by the MMU.

When new virtual memory is allocated by kernel for kernel-space or user-space it is need to be mapped into physical memory. Without that every access to memory triggers memory fault exception. Pmap manages page tables and tlb if its needed. But memory mapping is not all. Pmap gives also possibility to modify access permissions for given page - it is a very important feature for modern virtual memory subsystem and it is necessary for sharing memory between different process in user-space. The most known feature that use that is copy-on-write. When one process calls `fork` system call then all pages are marked as read-only. When any of processes (parent or child) try to write to memory then memory fault occurs. Kernel allocates new page and copies the content of old page into new. At the end pmap changes mapping of old virtual address into new physical page and clears caches and tlb for that address. For more details see [11].

For information about this module see 4.3.

3.3.1 Access emulation

On Malta board we do not have hardware tracking of accesses to pages. For that purpose we emulate these functionality. The same situation takes place on AArch64 4.3.5. Implementation of `pmap_emulate_bits` is similar for both supported architectures. For this reason I will comment only implementation for AArch64. Here is a implementation for MIPS. The reader can compare both of them.

```c
int pmap_emulate_bits(pmap_t *pmap, vaddr_t va, vm_prot_t prot) {
  paddr_t pa;

  WITH_MTX_LOCK (&pmap->mtx) {
    if (!pmap_extract_nolock(pmap, va, &pa))
      return EFAULT;

    pte_t pte = pmap_pte_read(pmap, va);

    if ((prot & VM_PROT_READ) && !(pte & PTE_SW_READ))
      return EACCES;

    if ((prot & VM_PROT_WRITE) && !(pte & PTE_SW_WRITE))
      return EACCES;

    if ((prot & VM_PROT_EXEC) && (pte & PTE_SW_NOEXEC))
      return EACCES;
  }

  vm_page_t *pg = vm_page_find(pa);
  assert(pg != NULL);

  WITH_MTX_LOCK (&pv_list_lock) {
    /* Kernel non-pageable memory? */
    if (TAILQ_EMPTY(&pg->pv_list))
      return EINVAL;
  }

  pmap_set_referenced(pg);
  if (prot & VM_PROT_WRITE)
    pmap_set_modified(pg);

  return 0;
}
```

Listing 3: Emulate access and reference bits MIPS (`pmap.c` [33])

3.3.2 Growkernel

I introduced `pmap_growkernel` function which increases virtual address space of kernel. This change was needed to make KASAN work on AArch64 architecture but it also works on MIPS. For more information about origin of that change see 3.4.

To increase virtual address space of kernel we need to know the old end of virtual address space ❶ (`vm_kernel_end` variable), new end ❸ and location of kernel page table ❷. We only need to add directory entries into kernel page table ❺ (if they do not exist) and send information to KASAN about new area of memory that can be used by kernel memory allocator ❻. `L1_SPACE_SIZE` ❹ is a number of bytes that are covered by single page directory entry. Note that the difference between MIPS and AArch64 4.3.6 version of that function is related to size of pointers (4 vs 8 bytes).

```
void pmap_growkernel(vaddr_t maxkvaddr) {
  assert(mtx_owned(&vm_kernel_end_lock));
❶assert(maxkvaddr > vm_kernel_end);

❷pmap_t *pmap = pmap_kernel();
  vaddr_t va;

❸maxkvaddr = roundup2(maxkvaddr, L1_SPACE_SIZE);

  WITH_MTX_LOCK (&pmap->mtx) {
   ❹for (va = vm_kernel_end; va < maxkvaddr; va += L1_SPACE_SIZE) {
      if (!is_valid_pde(PDE_OF(pmap, va)))
       ❺pmap_add_pde(pmap, va);
    }
  }

  /*
   * kasan_grow calls pmap_kenter which acquires pmap->mtx.
   * But we are under vm_kernel_end_lock from kmem so it is safe to call
   * kasan_grow.
   */
❻kasan_grow(maxkvaddr);

  vm_kernel_end = maxkvaddr;
}
```

Listing 4: `pmap_growkernel` MIPS (`pmap.c` [33])

3.4 KASAN

KASAN is a kernel address sanitizer. It is a tool that allow us to detect different types of memory error in runtime. Julian Pszczołowski is an author of original

KASAN implementation for Mimiker which detects stack overflow, buffer overflow and use-after-free errors [13].

Main idea of address sanitizer is creating special contiguous memory area for describing whole used memory. That area is called shadow map. For each byte of memory available for program we have one bit of memory in shadow map that describes if it is safe to use. For each instruction that access memory we add additional call to special function which checks in shadow map if we are doing valid access. For more information about please refer to Julian's thesis.

This implementation had very important assumption that we can allocate shadow map for whole memory at the begin of kernel life. It was justified for MIPS where we have 32 bit pointers so maximal memory that is available for CPU is 4GB of RAM so we need 512MB of RAM for shadow map. In fact we do not need so much memory and we used only 16MB for shadow map which is reasonable value for Malta board.

This assumption doesn't work for AArch64 architecture where pointers have 64 bits so we do not have any possibility to describe whole memory (32TiB for shadow map). So we need to split initialization of shadow map into two phases but it still is a contiguous area in memory – in our case it is located at the end of kernel virtual address space.

3.4.1 Kernel bootstrap

First we need to create shadow map for initial memory mapping. This mapping contains kernel code and static variables.

We need to create valid entries in page table for shadow map ❷ which is located at `KASAN_MD_SHADOW_START`. Shadow map is filled with proper values later during initialization of KASAN, but need to be done here since general memory allocator will be available after KASAN initialization. Here we use primitive physical memory allocator ❶.

```
size_t kasan_sanitized_size =
    roundup2(va - KASAN_MD_SANITIZED_START,
             SUPERPAGESIZE * KASAN_SHADOW_SCALE_SIZE);
size_t kasan_shadow_size = kasan_sanitized_size / KASAN_SHADOW_SCALE_SIZE;
va = KASAN_MD_SHADOW_START;
/* Allocate physical memory for shadow area */
❶paddr_t pa = (paddr_t)bootmem_alloc(kasan_shadow_size);
/* How many PDEs should we use? */
int num_pde = kasan_shadow_size / SUPERPAGESIZE;
for (int i = 0; i < num_pde; i++) {
  /* Allocate a new PT */
  pte = bootmem_alloc(PAGESIZE);
  pde[PDE_INDEX(va)] = PTE_PFN((paddr_t)pte) | PTE_KERNEL;
  for (int j = 0; j < PT_ENTRIES; j++) {
  ❷pte[PTE_INDEX(va)] = PTE_PFN(pa) | PTE_KERNEL;
    va += PAGESIZE;
    pa += PAGESIZE;
  }
}
```

Listing 5: Shadow map bootstrap MIPS (`boot.c` [34])

3.4.2 Grow kernel

We know how to create initial shadow map but in that way we can't use more sanitized memory and we do not want to use unsanitized memory in kernel when KASAN is running. From previous section 3.3 we know how to increase kernel virtual address space.

At the end of `pmap_growkernel` `kasan_grow` is called by pmap subsystem. It is responsible for increasing shadow map. For each missing page in shadow map ❶ we allocate new page using abstractions from machine-independent subsystems ❷. That page is mapped by pmap into kernel page table ❸. Finally the memory area that was added into virtual address space of kernel is marked as invalid ❹. It will be marked as valid latter by kernel memory allocator. This function implements key feature for KASAN on AArch64.

```
void kasan_grow(vaddr_t maxkvaddr) {
  assert(mtx_owned(&vm_kernel_end_lock));
  maxkvaddr = roundup2(maxkvaddr, PAGESIZE * KASAN_SHADOW_SCALE_SIZE);
  assert(maxkvaddr < KASAN_MD_MAX_SANITIZED_END);
  vaddr_t va = kasan_va_to_shadow(_kasan_sanitized_end);
  vaddr_t end = kasan_va_to_shadow(maxkvaddr);

  /* Allocate and map shadow pages to cover the new KVA space. */
❶for (; va < end; va += PAGESIZE) {
  ❷vm_page_t *pg = vm_page_alloc(1);
  ❸pmap_kenter(va, pg->paddr, VM_PROT_READ | VM_PROT_WRITE, 0);
  }

  if (maxkvaddr > _kasan_sanitized_end) {
  ❹kasan_mark_invalid((const void *)(_kasan_sanitized_end),
                       maxkvaddr - _kasan_sanitized_end,
                       KASAN_CODE_FRESH_KVA);
    _kasan_sanitized_end = maxkvaddr;
  }
}
```

Listing 6: Increase shadow map (`kasan.c` [35])

3.5 Infrastructure for testing

The important part of development is testing. Without tests it is impossible to track progress of project like operating system. Unfortunately it is not easy to run tests for kernel. Usually we can run program and collect it is output or exit code and generate report based on that values but we have an assumption that this program is running on some system. Here we run program directly on hardware or on emulator.

Previously we used fact that we have multiple UARTs on Malta board and one of them was generating diagnostic output for testing purpose. We were using `socat` to grab output of that UART and decide if test passed or not. But on Raspberry Pi 3 we do not have enough UARTs for that so we decided to redesign our approach.

We introduced special function `ktest_success`, which is a dead end in kernel. We use `gdb` to set breakpoint on this function. If we stop inside that function we kill emulator and quit from `gdb` with exit code `0`, otherwise we quit with different exit code. Of course we do something more in case of failure e.g. we print kernel log, stack trace of every thread and much more.

These changes reduced our complicated logic and eliminated race condition on one of UART which was a source of errors in our continuous integration.

3.6 Generic UART interface

A general terminal interface that we support in Mimiker is tty. In this section I will describe generic UART interface created as a glue between UART drivers and tty machine-independent layer. The terminal subsystem is responsible for processing incoming characters for processes. That processing requires managing multiple queues of characters on different layers. Here we will only discuss UART layer. For more information about tty see [12].

At the beginning of my work big part of the tty layer have lived inside UART driver. It was understandable - we had only one terminal exposed to user-space. With new platform we gain a new UART driver 2.2 and we want to share as much code as we can between drivers.

I proposed an abstraction over UART. The abstraction is a set of functions for interaction with hardware and generic structure for describing current state of UART.

3.6.1 Low level interface

```
typedef uint8_t (*uart_getc_t)(void *state);
typedef bool (*uart_rx_ready_t)(void *state);
typedef void (*uart_putc_t)(void *state, uint8_t byte);
typedef bool (*uart_tx_ready_t)(void *state);
typedef void (*uart_tx_enable_t)(void *state);
typedef void (*uart_tx_disable_t)(void *state);

typedef struct uart_state {
  spin_t u_lock;
  ringbuf_t u_rx_buf; /* Software receiver queue. */
  ringbuf_t u_tx_buf; /* Software transmitter queue. */
  tty_thread_t u_ttd;
  void *u_state; /* Private state - mostly memory and irq resources. */
} uart_state_t;
```

Listing 7: Generic low level UART interface (`uart.h` [36])

- `uart_getc_t` returns single byte from UART.
- `uart_rx_ready_t` returns true if receiver hardware queue is ready.
- `uart_putc_t` puts byte into UART.
- `uart_tx_ready_t` returns true if transmitter hardware queue is ready.
- `uart_tx_enable_t` enables transmitter interrupt.

- `uart_tx_disable_t` disables transmitter interrupt.

This low level interface can be used to implement generic interrupt handler for UART and generic worker for TTY subsystem. If you are interested in how the implementation of these methods looks, see 4.8.3.

3.6.2 Generic interrupt handler

Generic interrupt handler is responsible for responding to two kinds of events:

- a new character has been put into hardware queue (`TTY_THREAD_RXRDY`);
- a new character can be sent by device (`TTY_THREAD_TXRDY`);

If a new character has been put into hardware queue ❶, we put it into the software receive queue and notify tty subsystem ❸. If a new character can be sent by device ❹ we send as many characters from queue as we can ❺. Finally, if software queue is empty ❻ we send notification to tty layer about that.

```
intr_filter_t uart_intr(void *data /* device_t* */) {
  device_t *dev = data;
  uart_state_t *uart = dev->state;
  tty_thread_t *ttd = &uart->u_ttd;
  intr_filter_t res = IF_STRAY;

  WITH_SPIN_LOCK (&uart->u_lock) {
    /* data ready to be received? */
   ❶if (uart_rx_ready(dev)) {
     ❷(void)ringbuf_putb(&uart->u_rx_buf, uart_getc(dev));
     ❸ttd->ttd_flags |= TTY_THREAD_RXRDY;
      cv_signal(&ttd->ttd_cv);
      res = IF_FILTERED;
    }

    /* transmit register empty? */
    if (uart_tx_ready(dev)) {
      uint8_t byte;
     ❹while (uart_tx_ready(dev) && ringbuf_getb(&uart->u_tx_buf, &byte))
       ❺uart_putc(dev, byte);
      if (ringbuf_empty(&uart->u_tx_buf)) {
        /* If we're out of characters and there are characters
         * in the tty's output queue, signal the tty thread to refill. */
        if (ttd->ttd_flags & TTY_THREAD_OUTQ_NONEMPTY) {
         ❻ttd->ttd_flags |= TTY_THREAD_TXRDY;
          cv_signal(&ttd->ttd_cv);
        }
        /* Disable TXRDY interrupts - the tty thread will re-enable them
         * after filling tx_buf. */
        uart_tx_disable(dev);
      }
      res = IF_FILTERED;
    }
  }

  return res;
}
```

Listing 8: UART interrupt handler (`uart.c` [37])

3.6.3 UART-TTY thread

The role of UART-TTY thread is very similar to role of UART interrupt handler but on the upper layer. Once we are signalled ❶ by UART interrupt, we check what we need to do. If the receive queue is not empty ❷ we move ❸ characters from that queue into upper layer of terminal subsystem. If the output queue is not empty ❹ we move characters from terminal's queue to the outgoing buffer ❺.

```
static void uart_tty_thread(void *arg) {
  device_t *dev = arg;
  uart_state_t *uart = dev->state;
  tty_thread_t *ttd = &uart->u_ttd;
  tty_t *tty = ttd->ttd_tty;
  uint8_t work, byte;

  while (true) {
    WITH_SPIN_LOCK (&uart->u_lock) {
      /* Sleep until there's work for us to do. */
      while ((work = ttd->ttd_flags & TTY_THREAD_WORK_MASK) == 0)
      ❶ cv_wait(&ttd->ttd_cv, &uart->u_lock);
      ttd->ttd_flags &= ~TTY_THREAD_WORK_MASK;
    }
    WITH_MTX_LOCK (&tty->t_lock) {
    ❷ if (work & TTY_THREAD_RXRDY) {
        /* Move characters from rx_buf into the tty's input queue. */
        while (uart_getb_lock(uart, &byte))
        ❸ if (!tty_input(tty, byte))
            klog("dropped character %hhx", byte);
      }
    ❹ if (work & TTY_THREAD_TXRDY)
      ❺ uart_tty_fill_txbuf(dev);
    }
  }
}
```

Listing 9: UART TTY thread (`uart_tty.c` [38])

Chapter 4

Mimiker on AArch64

In this chapter we will focus on changes in Mimiker related to AArch64 and Raspberry Pi 3.

It includes most of work related to AArch64 and Raspberry Pi 3 specific code for Mimiker. I will go through interaction between kernel-space and user-space, memory mapping module, kernel address sanitizer machine-dependent part, first instructions of kernel, exception handling and finally basic set of drivers.

It is recommended that the reader uses the source code viewer when reading this chapter which is available at [16].

These changes were first developed for QEMU emulator and then adjusted to physical hardware.

All assembly code presented in this chapter belongs to ARMv8 Instruction Set [2] with the following calling convention:

- `x31` – stack pointer;
- `x30` – link register;
- `x19` – `x28` – callee-saved registers;
- `x9` – `x18` – caller-saved registers;
- `x8` – indirect return value address;
- `x0` – `x7` – function arguments and their results;

4.1 How to run emulator?

A special toolchain is required to build Mimiker. The most significant part is GCC – GNU Compiler Collection – in version 11 in time of writing. Building of toolchain is fully automated by `make` scripts in Mimiker repository [17]. For full list of required packages see `Dockerfile` in root directory of project. It contains description of image used by CI system for automatic tests. More detailed instructions are available at 5.1. There exists development environment with `ssh` connection. For access please contact with administrators of Mimiker website [16]. If you have account on development machine you can skip installation of toolchain.

The simplest command for build is:

```
make BOARD=rpi3
```

It produces ramdisk with user-space programs – `initrd.cpio`, `sysroot` directory with debugging symbols for each user-space binary, `sys/mimiker.elf` – ELF with kernel and `sys/mimiker.img` – final kernel image.

For running interactive session use:

```
./launch --board=rpi3 -d init=/bin/ksh
```

This command runs `tmux` terminal multiplexer with three windows:

1. QEMU logs;
2. Mimiker console – our equivalent of TTY – with running `ksh`;
3. gdb session;

By default execution stops on first instruction of machine-independent part of kernel initialization and manual `continue` command is required by gdb.

4.2 Interaction with user-space

In modern general purpose operating systems kernel without user programs is useless. In this chapter we will focus on machine-dependent communication between kernel and user programs.

4.2.1 Copy

The `copy` functions are designed to copy contiguous data from one address to another like `memcpy` but they are safe to use for copying data between kernel-space and user-space. It means that write or read from unmapped address will not cause the kernel panic. The `copy` functions return 0 on success and error otherwise.

`copyin` copies `len` bytes of data from the user-space address `uaddr` to the kernel-space address `kaddr`.

```
int copyin(const void *uaddr, void *kaddr, size_t len);
```

`copyout` copies `len` bytes of data from the kernel-space address `kaddr` to the user-space address `uaddr`.

```
int copyout(const void *kaddr, void *uaddr, size_t len);
```

`copyinstr` copies a NULL-terminated string, at most `len` bytes long, from user-space address `uaddr` to kernel-space address `kaddr`. The number of bytes actually copies, including the terminating `NULL`, is returned in `done` (if `done` is not `NULL`).

```
int copyinstr(const void *uaddr, void *kaddr, size_t len, size_t *done);
```

`copystr` copies a NULL-terminated string, at most `len` bytes long, from kernel-space address `kfaddr` to kernel-space address `kdaddr`. The number of bytes actually copied, including the terminating `NULL`, is returned in `done` (if `done` is not `NULL`).

```
int copystr(const void *kfaddr, void *kdaddr, size_t len, size_t *done);
```

The `copy` functions return `0` on success. If a bad address is encountered they return `EFAULT`.

All of the `copy` functions need to do following things:

- validate arguments;
- set a flag what to do in case of fault;
- copy data;

First one is simple – kernel need to validate if `uaddr` is placed in user space which is a contiguous space between `USER_SPACE_START` and `USER_SPACE_END`.

For second one we have a special flag in thread structure. In case of fault we check if `td::td_onfault` flag is set. If it is we jump into special `copyerr` routine that returns `EFAULT` from `copy` context. It is needed because kernel doesn't have control over arguments from user-space. Let's think about `read`

```
ssize_t read(int fd, void *buf, size_t count);
```

It is a common mistake to pass wrong pointer as argument to syscall e.g.

```
char *read_data(FILE *fp, size_t count) {
❶char *buf = malloc(count);
  read(fileno(stdin), buf, count);
  return buf;
}
```

Listing 10: heap buffer overflow

It is possible that `malloc` returns `NULL` ❶. As a result we trigger memory fault in kernel during copy data to user-space but kernel cannot fail. So we set special flag to handle memory errors from functions which copy data between kernel-space and user-space.

For third one we use generic `bcopy` implementation.

Let's see the final implementation of `copyin`. We check if user-space pointers point to user-space ❶. Then we set flag in thread structure ❷ – this is an information that the thread is inside copy function. Finally `bcopy` [24] is called ❸. On success we return directly from `copyin` ❹ with 0.

```
ENTRY(copyin)
        sub     sp, sp, #16
        str     lr, [sp]

        # len > 0
        cbz     x2, 1f
❶
        # (uintptr_t)us < (uintptr_t)(us + len)
        add     x3, x0, x2
        cmp     x0, x3
        b.hi    copyerr

        # (uintptr_t)(us + len) <= USER_SPACE_END
        ldr     x4, =USER_SPACE_END
        cmp     x3, x4
        b.hi    copyerr

       ❷set_onfault x3, x4, copyerr
       ❸bl      bcopy
        clr_onfault x3
❹
1:      mov     x0, xzr

        ldr     lr, [sp]
        add     sp, sp, #16
        ret
END(copyin)
```

Listing 11: copyin (`copy.S` [39])

4.2.2 Syscall handler

System call (syscall) is a way in which a user program communicates with kernel. From user-space programmer view most of syscalls are normal functions provided by *libc*. That functions generate exceptions in order to changing context from user-space to kernel-space but they also must pass their arguments. We know that we have calling convention for normal function but that convention can be different for system calls.

user-space In *Mimiker* (on *AArch64*) we have the following convention:

- each syscall can have at most six arguments;
- arguments are passed via registers from `x0` to `x5`;
- syscall number is encoded in exception;

so from user-space side it is very simple – we only call assembly which generates exception ❶. At the end of syscall wrapper we also need to update `errno` variable ❷ according to *C* standard – it is a responsibility of `__sc_error`.

```
#define SYSCALL(name, num)
        ENTRY(name);
       ❶svc num;
       ❷cbnz x1, _C_LABEL(__sc_error);
        ret;
        END(name)
```

Listing 12: syscall entry from user-space (`syscall.h` [40])

kernel-space In kernel-space we need a trampoline from machine-dependent exception code to machine-independent specific syscall implementation. We have following generic syscall definition:

```
typedef int syscall_t(proc_t *p, void *args, register_t *result);

typedef struct sysent {
  int nargs;       /* number of args passed to syscall */
  syscall_t *call; /* syscall implementation */
} sysent_t;
```

Listing 13: system call entry (`sysent.h` [41])

And for each system call we are casting args into specific argument type e.g.:

```
❶#define SCARG(p, x) ((p)->x.arg)
❷#define SYSCALLARG(x) union { register_t _pad; x arg; }

typedef struct {
  SYSCALLARG(int) fd;
  SYSCALLARG(void *) buf;
  SYSCALLARG(size_t) nbyte;
} read_args_t;

static int sys_read(proc_t *p, read_args_t *args, register_t *res) {
  int fd = SCARG(args, fd);
  void *u_buf = SCARG(args, buf);
  size_t nbyte = SCARG(args, nbyte);
  int error;

  uio_t uio = UIO_SINGLE_USER(UIO_READ, 0, u_buf, nbyte);
  if ((error = do_read(p, fd, &uio)))
    return error;
  *res = nbyte - uio.uio_resid;
  return 0;
}
```

Listing 14: read system call (`syscalls.c` [42])

It is very important for us that each argument has size of the register. For that purpose we use combination of `SCARG` ❶ and `SYSCALLARG` ❷. We need that for simplicity in trampoline code.

trampoline We can divide syscall trampoline into following steps:

- copy arguments from user context into args structure ❶;
- find `sysent_t` of called syscall ❷;
- call syscall function ❸;
- jump to exception return code;

For arguments copying we use buffer where we are storing subsequent arguments for args structure ❶. This is the reason why arguments have size of the register.

```
static void syscall_handler(register_t code, ctx_t *ctx,
                            syscall_result_t *result) {
  register_t args[SYS_MAXSYSARGS];
  const int nregs = 6;

❶memcpy(args, &_REG(ctx, X0), nregs * sizeof(register_t));

  if (code > SYS_MAXSYSCALL) {
    args[0] = code;
    code = 0;
  }

❷sysent_t *se = &sysent[code];
  size_t nargs = se->nargs;

  thread_t *td = thread_self();
  register_t retval = 0;

❸int error = se->call(td->td_proc, (void *)args, &retval);

  result->retval = error ? -1 : retval;
  result->error = error;
}
```

Listing 15: system call trampoline (`trap.c` [43])

4.2.3 crt0

C standard [1] says that:

- The function called at program startup is named `main`. The implementation declares no prototype for this function. It shall be defined with a return type of `int` and with no parameters or with two parameters or in some other implementation-defined manner.
- If they are declared, the parameters to the `main` function shall obey the following constraints:
  - The value of `argc` (first argument) shall be nonnegative.
  - `argv[argc]` (second argument) shall be a null pointer.
  - If the value of *argc* is greater than zero, the array embers `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup.
  - If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the *program name*. If the value of `argc` is greater than one,

the strings pointer to by `argv[1]` through `argv[argc - 1]` represent the *program parameters.*

– The parameters `argc` and `argv` and the strings pointer to by the `argv` array shall be modifiable by the program, and retain ther last-stored values between program startup and program termination.

But there is a lot of things to do before `main` e.g. calling functions marked as `__attribute__((constructor))` or *libc* init. So the real startup for program is `_start` from (in our case) `crt0.S`.

To understand implementation of `_start` we need to know how the initial stack layout of process looks. *Mimiker* puts program arguments and environment variables onto initial process stack. Let's use previous names for variables and additionally let's call `envp` as a table of environment variables, `m` as a number of environment variables and `n` as a number of *program arguments.*

`envp[m - 1]`
⋮
`envp[1]`
`envp[0]`
	stack segment high address
	each of `envp[i]` is a NULL-terminated string

`argv[n - 1]`
⋮
`argv[1]`
`argv[0]`
	each of `argv[i]` is a NULL-terminated string

`envp`	NULL-terminated environment vector storing pointers to `envp[0..m]`
`argv`	NULL-terminated argument vector storing pointers to `argv[0..n]`
`argc`	a single `uint32` declaring the number of arguments (`n`)
program stack ⋮	stack segment low address

Table 4.1: user stack layout

```
ENTRY(_start)
        # Grab argc from stack.
        ldr     w0, [sp, #0]

        # Prepare argv.
        add     x1, sp, #8

        # Prepare envp, it starts at argv + argc + 1
        lsl     x2, x0, #3
        add     x2, x1, x2
        add     x2, x2, #8

        # Jump to start in crt0-common.c
        # void ___start(int argc, char **argv, char **envp)
        b       ___start
END(_start)
```

Listing 16: crt0 (`crt0.S` [44])

The rest of the program initialization is in machine-independent `___start` provided by *libc*.

4.2.4 Signals

Signals [8] are a form of inter process communication used in Unix-like operating systems. Signal is an asynchronous notification sent to a process or to a thread. When a signal is sent kernel interrupts execution of process and signal handler (if it is set) is being executed. Signal can be sent directly by kernel e.g. as a result of memory fault or by other process using `kill` system call [9]. User-space program can be interrupted by signal at almost any time.

```
static void sighandler(int sig, ksiginfo_t *info, ucontext_t *ctx);
```

Machine-dependent part of signals is calling signal handler. In Mimiker we copy following data into user thread stack

- `sigcode` procedure; it is a special code that calls `sigreturn` system call for going back from signal to a context of interrupted thread;
- `struct ksiginfo` a bunch of information about signal for user;
- `struct ucontext` a context of interrupted thread

`sigcode` procedure is needed because signal handler can destroy any register so we can't just return from function. We need to go back to kernel to fix context of thread. Our solution is to copy simply procedure that calls special syscall ❶ into stack and changing return address of syscall handler to that procedure ❸. In this way we can go back to kernel and fix context.

```
ENTRY(sigcode)
        mov     x0, sp        /* address of ucontext_t to restore */
      ❶ svc     #SYS_sigreturn
        brk     #0
EXPORT(esigcode)
END(sigcode)
```

Listing 17: sigcode.S (`sigcode.S` [45])

We also need to allocate memory for that data on user stack ❶ and copy their addresses into registers according to ABI ❷.

```
int sig_send(signo_t sig, sigset_t *mask, sigaction_t *sa,
             ksiginfo_t *ksi) {
  thread_t *td = thread_self();
  mcontext_t *uctx = td->td_uctx;

  ucontext_t uc;
  mcontext_copy(&uc.uc_mcontext, uctx);
  uc.uc_sigmask = *mask;

❶register_t sc_code = sig_stack_push(uctx, sigcode, esigcode - sigcode);
  register_t sc_info = sig_stack_push(uctx, ksi, sizeof(ksiginfo_t));
  register_t sc_uctx = sig_stack_push(uctx, &uc, sizeof(ucontext_t));

❷_REG(uctx, ELR) = (register_t)sa->sa_handler;
  _REG(uctx, X0) = sig;
  _REG(uctx, X1) = sc_info;
  _REG(uctx, X2) = sc_uctx;
❸_REG(uctx, LR) = sc_code;

  return 0;
}
```

Listing 18: signal.c (`signal.c` [46])

4.2.5 Nonlocal goto

Here I want to describe AArch64 implementation of non local jumps.

Like on MIPS we have six different functions related to non local jumps.

`_setjmp` - which saves context of running thread into jump buffer. We need to save callee saved registers. They are `x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29` ❶. Link register `lr` ❷. Stack pointer `sp` ❸. We also need to save all floating point registers ❺ and flags ❹ for the kernel that this context contains valid CPU and FPU registers.

```
ENTRY(_setjmp)
        /* Copy saved registers */
       ❶str     x19, [x0, UC_REGS_X19]
        stp     x20, x21, [x0, UC_REGS_X20]
        stp     x22, x23, [x0, UC_REGS_X22]
        stp     x24, x24, [x0, UC_REGS_X24]
        stp     x26, x27, [x0, UC_REGS_X26]
        stp     x28, x29, [x0, UC_REGS_X28]
       ❷str     lr, [x0, UC_REGS_LR]

        /* sp register is special */
        mov     x1, sp
       ❸str     x1, [x0, UC_REGS_SP]

        /* save flags */
        ldr     x1, [x0, UC_FLAGS]
        orr     x1, x1, (_UC_FPU | _UC_CPU)
       ❹str     x1, [x0, UC_FLAGS]

        /* FPU status */
        mrs     x1, fpcr
        str     w1, [x0, UC_FPREGS_FPCR]
        mrs     x1, fpsr
        str     w1, [x0, UC_FPREGS_FPSR]

       ❺stp     q0, q1, [x0, UC_FPREGS_Q0]
        stp     q2, q3, [x0, UC_FPREGS_Q2]
        stp     q4, q5, [x0, UC_FPREGS_Q4]
        stp     q6, q7, [x0, UC_FPREGS_Q6]
        stp     q8, q9, [x0, UC_FPREGS_Q8]
        stp     q10, q11, [x0, UC_FPREGS_Q10]
        stp     q12, q13, [x0, UC_FPREGS_Q12]
        stp     q14, q15, [x0, UC_FPREGS_Q14]
        stp     q16, q17, [x0, UC_FPREGS_Q16]
        stp     q18, q19, [x0, UC_FPREGS_Q18]
        stp     q20, q21, [x0, UC_FPREGS_Q20]
        stp     q22, q23, [x0, UC_FPREGS_Q22]
        stp     q24, q25, [x0, UC_FPREGS_Q24]
        stp     q26, q27, [x0, UC_FPREGS_Q26]
        stp     q28, q29, [x0, UC_FPREGS_Q28]
        stp     q30, q31, [x0, UC_FPREGS_Q30]

        mov     x0, xzr
        ret
END(_setjmp)
```

Listing 19: _setjmp (`_setjmp.S` [47])

`_longjmp` - which is an opposite to `_setjmp`. It reads every registers written by _setjmp.

```
ENTRY(_longjmp)
        ldr     x19, [x0, UC_REGS_SP]
        mov     sp, x19
        ldr     x19, [x0, UC_REGS_X19]
        ldp     x20, x21, [x0, UC_REGS_X20]
        ldp     x22, x23, [x0, UC_REGS_X22]
        ldp     x24, x25, [x0, UC_REGS_X24]
        ldp     x26, x27, [x0, UC_REGS_X26]
        ldp     x28, x29, [x0, UC_REGS_X28]
        ldr     lr, [x0, UC_REGS_LR]

        /* FPU status */
        ldr     w2, [x0, UC_FPREGS_FPCR]
        /* msr ignores upper 32 bits for fpcr & fpsr */
        msr     fpcr, x2
        ldr     w2, [x0, UC_FPREGS_FPSR]
        msr     fpsr, x2

        ldp     q0, q1, [x0, UC_FPREGS_Q0]
        ldp     q2, q3, [x0, UC_FPREGS_Q2]
        ldp     q4, q5, [x0, UC_FPREGS_Q4]
        ldp     q6, q7, [x0, UC_FPREGS_Q6]
        ldp     q8, q9, [x0, UC_FPREGS_Q8]
        ldp     q10, q11, [x0, UC_FPREGS_Q10]
        ldp     q12, q13, [x0, UC_FPREGS_Q12]
        ldp     q14, q15, [x0, UC_FPREGS_Q14]
        ldp     q16, q17, [x0, UC_FPREGS_Q16]
        ldp     q18, q19, [x0, UC_FPREGS_Q18]
        ldp     q20, q21, [x0, UC_FPREGS_Q20]
        ldp     q22, q23, [x0, UC_FPREGS_Q22]
        ldp     q24, q25, [x0, UC_FPREGS_Q24]
        ldp     q26, q27, [x0, UC_FPREGS_Q26]
        ldp     q28, q29, [x0, UC_FPREGS_Q28]
        ldp     q30, q31, [x0, UC_FPREGS_Q30]

        mov     x0, x1
        ret
END(_longjmp)
```

Listing 20: _longjmp (_setjmp.S [47])

longjmp - in addition to _longjmp it sets signal mask ❶.

```c
void longjmp(jmp_buf env, int val) {
  ucontext_t *sc_uc = (void *)env;
  ucontext_t uc;
  memset(&uc, 0, sizeof(ucontext_t));

  if (_REG(sc_uc, SP) == 0)
    goto err;

  if (val == 0)
    val = 1;

  uc.uc_flags =
    _UC_CPU | ((sc_uc->uc_flags & _UC_STACK) ? _UC_SETSTACK : _UC_CLRSTACK);

❶sigprocmask(SIG_SETMASK, &sc_uc->uc_sigmask, NULL);

  uc.uc_link = 0;

  _REG(&uc, X0) = val;

  _REG(&uc, X19) = _REG(sc_uc, X19);
  _REG(&uc, X20) = _REG(sc_uc, X20);
  _REG(&uc, X21) = _REG(sc_uc, X21);
  _REG(&uc, X22) = _REG(sc_uc, X22);
  _REG(&uc, X23) = _REG(sc_uc, X23);
  _REG(&uc, X24) = _REG(sc_uc, X24);
  _REG(&uc, X25) = _REG(sc_uc, X25);
  _REG(&uc, X26) = _REG(sc_uc, X26);
  _REG(&uc, X27) = _REG(sc_uc, X27);
  _REG(&uc, X28) = _REG(sc_uc, X28);
  _REG(&uc, X29) = _REG(sc_uc, X29);

  _REG(&uc, SP) = _REG(sc_uc, SP);
  _REG(&uc, LR) = _REG(sc_uc, LR);
  _REG(&uc, PC) = _REG(sc_uc, PC);
  _REG(&uc, SPSR) = _REG(sc_uc, SPSR);
  _REG(&uc, TPIDR) = _REG(sc_uc, TPIDR);

  if (sc_uc->uc_flags & _UC_FPU) {
    memcpy(&uc.uc_mcontext.__fregs, &sc_uc->uc_mcontext.__fregs,
           sizeof(__fregset_t));
    uc.uc_flags |= _UC_FPU;
  }

  setcontext(&uc);
err:
  longjmperror();
  abort();
}
```

Listing 21: longjmp (`longjmp.c` [48])

`setjmp` - in addition to `_setjmp` it saves current signal mask.

```
ENTRY(setjmp)
        sub     sp, sp, CALLFRAME_SIZ
        stp     lr, x19, [sp]
        /* Save env in safe register. */
        mov     x19, x0

        /* Save current signalmask at ucontext::uc_sigmask.
         * If set is NULL, then the signal mask is unchanged (i.e., how is
         * ignored), but the current value of the signal mask is
         * nevertheless returned in oldset (if it is not NULL). */
        add     x2, x19, UC_MASK /* &env->uc_sigmask */
        mov     x1, xzr
        bl      sigprocmask
        cmp     x0, xzr
        bne     botch

        /* Save stack_t at ucontext::uc_stack
         * By specifying ss as NULL, and old_ss as a non-NULL value, one
         * can obtain the current settings for the alternate signal stack
         * without changing them. */
        add     x1, x19, UC_STACK /* &env->uc_stack */
        /* We know that x0 is equal to 0 here. */
        bl      sigaltstack
        cmp     x0, xzr
        bne     botch

        /* stack_t::ss_flags is a int */
        ldr     w0, [x19, UC_STACK+SS_FLAGS]
        and     w0, w0, SS_ONSTACK
        cmp     w0, wzr
        beq     1f

        /* ucontext_t::uc_flags is a int */
        ldr     w0, [x19, UC_FLAGS]
        orr     w0, w0, _UC_STACK
        str     w0, [x19, UC_FLAGS]

1:
        /* restore jpmbuf */
        mov     x0, x19
        ldp     lr, x19, [sp]

        add     sp, sp, CALLFRAME_SIZ

        b       _setjmp
botch:
        bl      abort
END(setjmp)
```

Listing 22: setjmp (`setjmp.S` [49])

And finally `sigsetjmp` and `siglongjmp`. They are `setjmp` and `longjmp` that can be used inside signal handlers. They are only dispatchers which call the rest functions based on arguments.

```
/* int sigsetjmp(jmp_buf buf, int savesigs) */
ENTRY(sigsetjmp)
        cmp     x1, xzr
        bne     1f
        str     x1, [x0, UC_FLAGS]
        b       _setjmp

1:      mov     x1, _UC_SIGMASK
        str     x1, [x0, UC_FLAGS]
        b       setjmp
END(sigsetjmp)

/* void siglongjmp(sigjmp_buf env, int val) */
ENTRY(siglongjmp)
        ldr     x2, [x0, UC_FLAGS]
        and     x2, x2, _UC_SIGMASK
        cmp     x2, _UC_SIGMASK
        beq     longjmp
        b       _longjmp
END(siglongjmp)
```

Listing 23: sigsetjmp and siglongjmp (`sigsetjmp.S` [50])

4.3 pmap

The physical-mapping module (pmap) manages machine-dependent address translation and access tables that are used either directly or indirectly by the MMU.

When new virtual memory is allocated by the kernel for kernel-space or user-space it needs to be mapped into physical memory. Without that every access to memory triggers memory fault exception. Pmap manages page tables and tlb if its needed. But memory mapping is not all. Pmap also gives possibility to modify access permissions for given virtual memory address - it is very important feature for modern virtual memory subsystem and it is necessary for sharing memory between different process in user-space. The most known feature, which uses that is copy-on-write. When one process calls `fork` system call then all pages are marked as read-only. When any of processes (parent or child) try to write to memory then memory fault occurs. Kernel allocate new page and copies the content of old page into the new. At the end pmap change mapping of old virtual address into new physical page and clear caches and tlb for that address. For more details about copy-on-write see [11].

4.3.1 Interface

pmap_t is a structure that manages actual virtual address space of process.

```
typedef struct pmap {
  mtx_t mtx;                      /*protects all fields in this structure*/
  asid_t asid;                    /*address space identifier*/
  paddr_t pde;                    /*directory page table physical address*/
  vm_pagelist_t pte_pages;        /*pages we allocate in page table*/
  TAILQ_HEAD(, pv_entry) pv_list; /*all pages mapped by this physical map*/
} pmap_t;
```

Listing 24: Physical map definition (pmap.c [51])

Return true if va belongs to given pmap. It is used mostly for sanity-checks in our codebase.

```
bool pmap_address_p(pmap_t *pmap, vaddr_t va);
```

Return true if the range start to end belongs to given pmap. It is also used for sanity-checks.

```
bool pmap_contains_p(pmap_t *pmap, vaddr_t start, vaddr_t end);
```

Return first address that belongs to given pamp.

```
vaddr_t pmap_start(pmap_t *pmap);
```

Return last address that belongs to given pamp.

```
vaddr_t pmap_end(pmap_t *pmap);
```

Bootstrap kernel pmap. It sets address space id for page table allocated during early bootstrapping, initializes mutexes for pmap module and initializes lists of used pages used by kernel page table.

```
void init_pmap(void);
```

Allocate new page table, address space id and bootstrap new pmap with that page table. We call this function every time a new process is created directly by kernel or by system call.

```
pmap_t *pmap_new(void);
```

Delete pmap structure, free pages which belong to given `pmap` and free address space id.

```
void pmap_delete(pmap_t *pmap);
```

Map page to given virtual address in `pmap` with given protection and cache flags. It is used mostly for mapping pages into address spaces of user threads. Kernel should use `pmap_kenter` for that purpose.

```
void pmap_enter(pmap_t *pmap, vaddr_t va, vm_page_t *pg,
                vm_prot_t prot, unsigned flags);
```

Find where a given virtual address is mapped in `pmap`.

```
bool pmap_extract(pmap_t *pmap, vaddr_t va, paddr_t *pap);
```

Remove mapping from given `pmap`.

```
void pmap_remove(pmap_t *pmap, vaddr_t start, vaddr_t end);
```

Map page to given virtual address in kernel pmap with given protection and cache flags.

```
void pmap_kenter(vaddr_t va, paddr_t pa, vm_prot_t prot,
                 unsigned flags);
```

Find where a given virtual address is mapped in kernel pmap.

```
bool pmap_kextract(addr_t va, paddr_t *pap);
```

Remove mapping from kernel pmap.

```
void pmap_kremove(vaddr_t start, vaddr_t end);
```

Change protection of mapping. It can be called as a result of `mprotect` system call or by copy-on-write mechanism in kernel [11].

```
void pmap_protect(pmap_t *pmap, vaddr_t start, vaddr_t end,
                  vm_prot_t prot);
```

Remove page from every pmap.

```
void pmap_page_remove(vm_page_t *pg);
```

Clear given page.

```
void pmap_zero_page(vm_page_t *pg);
```

Copy given page.

```
void pmap_copy_page(vm_page_t *src, vm_page_t *dst);
```

Software tracking of referenced and modified bits. For more information see 4.3.5.

```
bool pmap_clear_referenced(vm_page_t *pg);
bool pmap_clear_modified(vm_page_t *pg);
bool pmap_is_referenced(vm_page_t *pg);
bool pmap_is_modified(vm_page_t *pg);
void pmap_set_referenced(vm_page_t *pg);
void pmap_set_modified(vm_page_t *pg);
int pmap_emulate_bits(pmap_t *pmap, vaddr_t va, vm_prot_t prot);
```

Activate mapping from given `pmap`. It is called when we want to change active address space.

```
void pmap_activate(pmap_t *pmap);
```

Return pmap for given virtual address.

```
pmap_t *pmap_lookup(vaddr_t addr);
```

Return kernel pmap.

```
pmap_t *pmap_kernel(void);
```

Return active user pmap.

```
pmap_t *pmap_user(void);
```

Increase usable kernel virtual address space to at least `maxkvaddr`. More details are available at 3.3.2.

```
void pmap_growkernel(vaddr_t maxkvaddr);
```

Here I describe the most important internals of that module. For high level overview please see FreeBSD manpages [5].

4.3.2 Protection map

`vm_prot_map` is a representation of access bits from 2.3.

```c
static const pte_t pte_common = L3_PAGE | ATTR_SH_IS;
static const pte_t pte_noexec = ATTR_XN | ATTR_SW_NOEXEC;

static const pte_t vm_prot_map[] = {
  [VM_PROT_NONE] = pte_noexec | pte_common,
  [VM_PROT_READ] =
    ATTR_AP_RO | ATTR_SW_READ | ATTR_AF | pte_noexec | pte_common,
  [VM_PROT_WRITE] =
    ATTR_AP_RW | ATTR_SW_WRITE | ATTR_AF | pte_noexec | pte_common,
  [VM_PROT_READ | VM_PROT_WRITE] = ATTR_AP_RW | ATTR_SW_READ |
    ATTR_SW_WRITE | ATTR_AF | pte_noexec | pte_common,
  [VM_PROT_EXEC] = ATTR_AF | pte_common,
  [VM_PROT_READ | VM_PROT_EXEC] =
    ATTR_AP_RO | ATTR_SW_READ | ATTR_AF | pte_common,
  [VM_PROT_WRITE | VM_PROT_EXEC] =
    ATTR_AP_RW | ATTR_SW_WRITE | ATTR_AF | pte_common,
  [VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXEC] =
    ATTR_AP_RW | ATTR_SW_READ | ATTR_SW_WRITE | ATTR_AF | pte_common,
};
```

Listing 25: protection map (`pmap.c` [51])

4.3.3 Walk

These functions are responsible for walking through page table. They use direct map ❶ which maps all physical memory into a contiguous area of virtual memory. The difference is that `pmap_ensure_pte` always returns a valid pointer to page table entry – new entries are allocated as needed ❷.

```
static pte_t *pmap_lookup_pte(pmap_t *pmap, vaddr_t va) {
  pde_t *pdep;
  paddr_t pa = pmap->pde;

  /* Level 0 */
❶pdep = (pde_t *)PHYS_TO_DMAP(pa) + L0_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep)))
    return NULL;

  /* Level 1 */
  pdep = (pde_t *)PHYS_TO_DMAP(pa) + L1_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep)))
    return NULL;

  /* Level 2 */
  pdep = (pde_t *)PHYS_TO_DMAP(pa) + L2_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep)))
    return NULL;

  /* Level 3 */
  return (pde_t *)PHYS_TO_DMAP(pa) + L3_INDEX(va);
}

static pte_t *pmap_ensure_pte(pmap_t *pmap, vaddr_t va) {
  pde_t *pdep;
  paddr_t pa = pmap->pde;

  /* Level 0 */
  pdep = (pde_t *)PHYS_TO_DMAP(pa) + L0_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep))) {
   ❷pa = pmap_alloc_pde(pmap, va);
    *pdep = pa | L0_TABLE;
  }

  /* Level 1 */
  pdep = (pde_t *)PHYS_TO_DMAP(pa) + L1_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep))) {
    pa = pmap_alloc_pde(pmap, va);
    *pdep = pa | L1_TABLE;
  }

  /* Level 2 */
  pdep = (pde_t *)PHYS_TO_DMAP(pa) + L2_INDEX(va);
  if (!(pa = PTE_FRAME_ADDR(*pdep))) {
    pa = pmap_alloc_pde(pmap, va);
    *pdep = pa | L2_TABLE;
  }

  /* Level 3 */
  return (pde_t *)PHYS_TO_DMAP(pa) + L3_INDEX(va);
}
```

Listing 26: page table walk (`pmap.c` [51])

4.3.4 Activation

This function activates given page table for user access. Pointer to level 0 of page table must be stored in `ttbr0` register with address space identifier (ASID) ❶. `EPD0` bit must be cleared in `tcr` register ❷.

```
void pmap_activate(pmap_t *umap) {
  SCOPED_NO_PREEMPTION();

  PCPU_SET(curpmap, umap);

  uint64_t tcr = READ_SPECIALREG(TCR_EL1);

  if (umap == NULL) {
    WRITE_SPECIALREG(TCR_EL1, tcr | TCR_EPD0);
  } else {
    uint64_t ttbr0 = ((uint64_t)umap->asid << ASID_SHIFT) | umap->pde;
  ❶WRITE_SPECIALREG(TTBR0_EL1, ttbr0);
  ❷WRITE_SPECIALREG(TCR_EL1, tcr & ~TCR_EPD0);
  }
}
```

Listing 27: activate virtual address space (`pmap.c` [51])

4.3.5 Access emulation

Since we do not use hardware tracking of `AF` (access permission) and `DBM` (dirty page) we need to manage them from software. We do not do that because not every CPU supports that and we want to be compatible with solutions from MIPS version. After mapping page table entry doesn't contain `AF` bit so first access to that page triggers page fault. When we handle that exception we use `pmap_emulate_bits` for checking permissions for that access ❶. If they are sufficient we set needed bits in page table entry (`AF` and some permission bits) in `pmap_set_referenced` and `pmap_set_modified` ❷. In other case error is returned.

```
int pmap_emulate_bits(pmap_t *pmap, vaddr_t va, vm_prot_t prot) {
  paddr_t pa;

  WITH_MTX_LOCK (&pmap->mtx) {
    if (!pmap_extract_nolock(pmap, va, &pa))
      return EFAULT;

    pte_t pte = *pmap_lookup_pte(pmap, va);

   ❶if ((prot & VM_PROT_READ) && !(pte & ATTR_SW_READ))
      return EACCES;

    if ((prot & VM_PROT_WRITE) && !(pte & ATTR_SW_WRITE))
      return EACCES;

    if ((prot & VM_PROT_EXEC) && (pte & ATTR_SW_NOEXEC))
      return EACCES;
  }

  vm_page_t *pg = vm_page_find(pa);
  assert(pg != NULL);

  WITH_MTX_LOCK (pv_list_lock) {
    /* Kernel non-pageable memory? */
    if (TAILQ_EMPTY(&pg->pv_list))
      return EINVAL;
  }

 ❷pmap_set_referenced(pg);
  if (prot & VM_PROT_WRITE)
    pmap_set_modified(pg);

  return 0;
}
```

Listing 28: Emulate access and reference bits (`pmap.c` [51])

4.3.6 Growkernel

Because address space on AArch64 is much bigger than on mips we can't describe whole virtual memory in various subsystems. It is the reason why `pmap_growkernel` exists. When `kmem` (kernel memory allocator) fails with out of memory error it calls `pmap_growkernel` to extend memory available for kernel space. Then new memory range is put to `vmem` (virtual memory allocator) and `kmem` call is restarted. That function is similar to MIPS 3.3.2 version so code comments are omitted.

```
void pmap_growkernel(vaddr_t maxkvaddr) {
  assert(mtx_owned(&vm_kernel_end_lock));
  assert(maxkvaddr > vm_kernel_end);

  pmap_t *pmap = pmap_kernel();
  vaddr_t va;

  maxkvaddr = roundup2(maxkvaddr, L2_SIZE);

  WITH_MTX_LOCK (&pmap->mtx) {
    for (va = vm_kernel_end; va < maxkvaddr; va += L2_SIZE)
      pmap_ensure_pte(pmap, va);
  }

  kasan_grow(maxkvaddr);

  vm_kernel_end = maxkvaddr;
}
```

Listing 29: `pmap_growkernel` (`pmap.c` [51])

4.4 KASAN

Thanks to changes described in 3.4 we only need to chose where the shadow map is located and build initial shadow map.

For shadow map I have chosen `0xffffff0000000000` ❹. It is located at the end of kernel space and only direct mapping is in higher addresses. Note that BSD systems follow the reverse order.

That address needs to be passed to C compiler directly because accesses to stack are sanitized directly by gcc code which doesn't use our functions. As a result we need to do one additional change in boot code – all functions in virtual addresses need to use virtually mapped stack. It is a reason why `aarch64_init` returns `_boot_stack`.

Here is a code that build initial shadow map:

```
❶size_t kasan_sanitized_size =
 ❷2 * SUPERPAGESIZE + roundup2(va - KASAN_MD_SANITIZED_START,
                        SUPERPAGESIZE * KASAN_SHADOW_SCALE_SIZE);
size_t kasan_shadow_size =
  kasan_sanitized_size / KASAN_SHADOW_SCALE_SIZE;
❸vaddr_t kasan_shadow_end = KASAN_MD_SHADOW_START + kasan_shadow_size;
❹va = KASAN_MD_SHADOW_START;
*(vaddr_t *)AARCH64_PHYSADDR(&_kasan_sanitized_end) =
  KASAN_MD_SANITIZED_START + kasan_sanitized_size;
❺pa = (paddr_t)bootmem_alloc(kasan_shadow_size);

❻while (va < kasan_shadow_end) {
  if (l0[L0_INDEX(va)] == 0)
    l0[L0_INDEX(va)] = (pde_t)bootmem_alloc(PAGESIZE) | L0_TABLE;

  pde_t *l1k = (pde_t *)PTE_FRAME_ADDR(l0[L0_INDEX(va)]);
  if (l1k[L1_INDEX(va)] == 0)
    l1k[L1_INDEX(va)] = (pde_t)bootmem_alloc(PAGESIZE) | L1_TABLE;

  pde_t *l2k = (pde_t *)PTE_FRAME_ADDR(l1k[L1_INDEX(va)]);
  if (l2k[L2_INDEX(va)] == 0)
    l2k[L2_INDEX(va)] = (pde_t)bootmem_alloc(PAGESIZE) | L2_TABLE;

  pde_t *l3k = (pde_t *)PTE_FRAME_ADDR(l2k[L2_INDEX(va)]);

 ❼for (int j = 0; va < kasan_shadow_end && j < PT_ENTRIES; j++) {
   ❽l3k[L3_INDEX(va)] = pa | ATTR_AP_RW | ATTR_XN | pte_default;
    va += PAGESIZE;
    pa += PAGESIZE;
  }
}
```

Listing 30: Build initial shadow map (`boot.c` [52])

First we need to calculate size of the current kernel space ❶. With that knowledge we calculate end of shadow area ❸ and physical memory is allocated ❺. Then the mapping between virtual and physical memory is created ❽ in page table for `shadow map`. The outer loop ❻ does page table walk in each iteration and the inner loop ❼ fills page table entries.

Additional superpages ❷ are workaround for bug in machine-independent par of memory management subsystem which is not a part of that port.

4.5 Boot

In this section we will discuss what is going on from first instruction to jumping to machine-independent part of code.

First thing we need to know is that first instruction is located at `0x200000` – it is kernel entry point.

4.5.1 start.S

As we can see the start is not complicated. First we have magic header needed by bootloader ❶. Next we check current CPU number ❷ – nowadays Mimiker is not ready to run on multiprocessor machine – if we are CPU0 we can execute code otherwise we are in the loop forever. We need to save a pointer to device tree blob ❸. That binary blob stores serialized information about devices present in machine and kernel's command line. The specification of device tree is available at [30]. Then initial stack is prepared ❹ and we jump to C code – `aarch64_init`. That code configures CPU and returns temporary stack ❺. Next we jump to `board_stack` which configures final stack for kernel and finally to `board_init` which is a trampoline for machine-independent code.

```
_ENTRY(_start)
        /* Based on locore.S from FreeBSD. */
        b       1f
       ❶.long   0
        .quad   IMAGE_OFFSET
        .quad   IMAGE_SIZE
        .quad   IMAGE_FLAGS
        .quad   0
        .quad   0
        .quad   0
        .long   0x644d5241 /* Magic "ARM\x64" */
        .long   0
1:
        /* Get CPU number. */
       ❷MRS     x3, MPIDR_EL1
        AND     x3, x3, #3
        CMP     x3, #0
        BNE     .

        /* Save pointer to dtb. */
       ❸mov     x19, x0
        /* Setup initial stack. */
        ADR     x3, __boot_stack_end
       ❹MOV     sp, x3

        BL      aarch64_init

       ❺mov     sp, x0
        /* Restore dtb pointer. */
        mov     x0, x19

        BL      board_stack
        MOV     sp, x0

        B       board_init
_END(_start)
```

Listing 31: First kernel instructions (`start.S` [53])

4.5.2 boot.c

`aarch64_init` is a dispatcher which calls functions that configure CPU.

AArch64 has four different exception levels 2.1.2. In our case exception level 0 is where user-space lives and exception level 1 is where kernel lives. But at the beginning we are not in exception level 1 so we need to drop ourselves to level 1 and it is exactly what `drop_to_el1` does ❶.

Next we need to clear `.bss` section of our binary ❷.

Finally we can build initial page table for kernel and configure MMU to use that page table ❸. For more details see 2.1.1, 4.3 and source code.

```
__boot_text void *aarch64_init(void) {
❶drop_to_el1();
  configure_cpu();
❷clear_bss();

  /* Set end address of kernel for boot allocation purposes. */
  _bootmem_end = (void *)align(AARCH64_PHYSADDR(__ebss), PAGESIZE);

❸enable_mmu(build_page_table());
  return &_boot_stack[PAGESIZE];
}
```

Listing 32: Early machine-dependent initialization (`boot.c` [52])

4.5.3 board stack

The responsibility of `board_stack` function is preparing final stack for the first kernel thread. This stack is preallocated in `thread0` structure ❶. We process device tree blob ❷ and extract important data: memory size, kernel's command line, location of initrd. These information are stored on kernel stack.

```
void *board_stack(paddr_t dtb) {
  dtb_early_init(dtb, fdt_totalsize(PHYS_TO_DMAP(dtb)));

❶kstack_t *stk = &thread0.td_kstack;

  thread0.td_uctx = kstack_alloc_s(stk, mcontext_t);

  /*
   * NOTE: memsize, rd_start, rd_size, cmdline + 2 = 6
   */
  char **kenvp = kstack_alloc(stk, 6 * sizeof(char *));
❷process_dtb(kenvp, stk, (void *)PHYS_TO_DMAP(dtb));
  kstack_fix_bottom(stk);
  init_kenv(kenvp);

  return stk->stk_ptr;
}
```

Listing 33: Build kernel stack (`board.c` [54])

The last thing is `board_init` which configures machine-independent part of kernel using data from dtb and jumps to kernel machine-independent entry.

```
__noreturn void board_init(void) {
  init_kasan();
  init_klog();
  rpi3_physmem();
  intr_enable();
  kernel_init();
}
```

Listing 34: Jump to machine-independent code (`board.c` [54])

4.6 Exception handler

Exceptions are a form of exceptional control flow, implemented partly by the hardware and partly by the operating system. An exception is an abrupt change in the control flow in response to some change in the processor's state.

We have four classes of exceptions.

class	cause	async/sync	return behaviour
interrupt	signal from I/O device	async	always returns to next instruction
trap	intentional exception	sync	always returns to next instruction
fault	potentially recoverable error	sync	might return to current instruction
abort	nonrecoverable error	sync	never returns

Table 4.2: classes of exceptions

For more high level information about exceptions see [10].

In Mimiker we only handle four types of exceptions.

- Synchronous EL1h
- IRQ EL1h
- Snchronous 64-bit EL0
- IRQ 64-bit EL0

There are more types of exceptions but we are running only on 0 & 1 exception modes so we do not care about others. For more information about exception modes see [7] and 2.1.

IRQ handlers are simple. We need to save context of running thread ❶ and jump to main interrupt handler in C code. `save_ctx` and `load_ctx` are macros that save and restore context of thread. Additional parameter (1 or 0) is a information if context belongs to user-space of kernel-space. The same code is used for kernel exception handler.

```
❶save_ctx 1
mov     x0, sp
bl      intr_root_handler
load_ctx 1
eret
```

Listing 35: irq exception handler (`evec.S` [55])

Things are more complicated for user exception handler. Again we need to save CPU context ❶. Here we also need to take care about FPU context when we return to user-space. We need to check if FPU is in use ❷. Based on that information we enable FPU ❸ for CPU and if given thread already has saved FPU context ❹ it is restored ❺. It is exactly the same as in mips 3.2.1.

```
        .cfi_signal_frame
       ❶save_ctx 0
        mov     x0, sp
        bl      user_trap_handler
user_exc_leave:
        /* disable interrupts */
        msr     daifset, #DAIF_I

        /* load thread_t::tdp_flags */
        /* thread_t::tdp_flags is a volatile unsigned - use 32-bit */
        load_pcpu x1
        ldr     x1, [x1, #PCPU_CURTHREAD]
        ldr     w3, [x1, #TD_PFLAGS]

       ❷and     w2, w3, #TDP_FPUINUSE
        cmp     w2, wzr
        beq     .skip_fpu_restore

        /* enable FPU */
        mrs     x2, cpacr_el1
        and     x2, x2, ~CPACR_FPEN_MASK
        orr     x2, x2, CPACR_FPEN_TRAP_NONE
       ❸msr     cpacr_el1, x2

       ❹and     w2, w3, #TDP_FPUCTXSAVED
        cmp     w2, wzr
        beq     .skip_fpu_restore

        /* clear TDP_FPUCTXSAVED flag */
        and     w2, w3, ~TDP_FPUCTXSAVED
        str     w2, [x1, #TD_PFLAGS]

        /* restore FPU context */
        ldr     x1, [x1, #TD_UCTX]
       ❺load_fpu_ctx x1, 2

.skip_fpu_restore:
        load_ctx 0
        eret
```

Listing 36: user exception handler (`evec.S` [55])

4.7 Context switching

Context switching is one of the most important parts of operating system. Without them we can't run multiple programs in parallel.

```
long ctx_switch(thread_t *from, thread_t *to);
```

This procedure changes running thread on current CPU from `from` to `to`. Context switching is a very sensitive function so interrupts need to be disabled before that procedure ❶. Then we need to check if FPU must be saved by us ❷. It is true when we switch from thread that was in user-space with activated FPU. After that CPU context of current thread is saved ❸. In the next step active virtual memory space is changed to the used by `to` thread ❹, and finally CPU context of that thread is loaded ❺.

For the reason why FPU context is only saved here see 3.2.1.

```
        # ctx_switch must be called with interrupts disabled
        mrs     x2, daif
      ❶ and     x2, x2, #PSR_I
        cmp     x2, xzr
        bne     .ctx_save
        hlt     #0
        # save context of @from thread
.ctx_save:
        ldr     w3, [x0, #TD_PFLAGS]
        and     w2, w3, #TDP_FPUINUSE|TDP_FPUCTXSAVED
        mov     w4, #TDP_FPUINUSE
      ❷ cmp     w2, w4
        bne     .skip_fpu_save

        orr     w3, w3, #TDP_FPUCTXSAVED
        str     w3, [x0, #TD_PFLAGS]
        /* enable FPU and save context */
        /* thread_t::tdp_flags is a volatile unsigned - use 32-bit */
        mrs     x2, cpacr_el1
        and     x2, x2, ~CPACR_FPEN_MASK
        orr     x2, x2, CPACR_FPEN_TRAP_NONE
        msr     cpacr_el1, x2

        ldr     x2, [x0, #TD_UCTX]
        save_fpu_ctx x2, 3
        /* disable FPU */
        mrs     x2, cpacr_el1
        and     x2, x2, ~CPACR_FPEN_MASK
        msr     cpacr_el1, x2
.skip_fpu_save:
        sub     sp, sp, #CTX_SIZE
      ❸ SAVE_CTX
        mov     x2, sp
        str     x2, [x0, #TD_KCTX]
.ctx_resume:
        # switch stack pointer to @to thread
        ldr     x2, [x1, #TD_KCTX]
        mov     sp, x2
        # update curthread pointer to reference @to thread
        load_pcpu x2
        str     x1, [x2, #PCPU_CURTHREAD]
        # switch user space if necessary
        mov     x0, x1
      ❹ bl      vm_map_switch
        # restore @to thread context
        LOAD_CTX
      ❺ add     sp, sp, #CTX_SIZE
        ret
```

Listing 37: context switch (`switch.S` [56])

4.8 Device tree

In this section I will describe minimal subset of drivers needed to boot Mimiker on RPi3.

We can think about devices as a tree. There is a one root which is an ancestor of all devices. Nodes are responsible for resources, like management of interrupts, for other devices e.g. USB controller. Leaves are final devices in our infrastructure e.g. keyboard.

4.8.1 Rootdev

Rootdev is a fake device. Purpose of rootdev is being an ancestor of all other devices. But for simplicity interrupt controller is integrated with rootdev device.

It provides methods for dispatching interrupts, enabling interrupts and disabling interrupts.

It is an interrupt dispatcher. Firstly CPU local interrupts are handled ❶. Next it handles interrupts from peripherals ❷.

```
static void rootdev_intr_handler(ctx_t *ctx, device_t *dev, void *arg) {
  assert(dev != NULL);
  rootdev_t *rd = dev->state;

  /* Handle local interrupts. */
❶bcm2835_intr_handle(rootdev_local_handle,
                     BCM2836_LOCAL_INTC_IRQPENDINGN(0),
                     &rd->intr_event[BCM2836_INT_BASECPUN(0)]);

  /* Handle GPU0 interrupts. */
❷bcm2835_intr_handle(rootdev_arm_base,
                     (BCM2835_ARMICU_OFFSET + BCM2835_INTC_IRQ1PENDING),
                     &rd->intr_event[BCM2835_INT_GPU0BASE]);

  /* Handle GPU1 interrupts. */
  bcm2835_intr_handle(rootdev_arm_base,
                     (BCM2835_ARMICU_OFFSET + BCM2835_INTC_IRQ2PENDING),
                     &rd->intr_event[BCM2835_INT_GPU1BASE]);

  /* Handle base interrupts. */
  bcm2835_intr_handle(rootdev_arm_base,
                     (BCM2835_ARMICU_OFFSET + BCM2835_INTC_IRQBPENDING),
                     &rd->intr_event[BCM2835_INT_BASICBASE]);
}
```

Listing 38: rootdev interrupt handler (`bcm2835_rootdev.c` [57])

This is a handler for given set of interrupts. Each interrupt is represented by single bit where 1 means that interrupt is present and 0 means that it is absent. Single set is represented by 32-bit register located in physical memory ❶. These registers are mapped to virtual memory during rootdev initialization. We iterate over pending interrupts ❷ and handle them one by one ❸.

```
static void bcm2835_intr_handle(bus_space_handle_t irq_base,
                                bus_size_t offset,
                                intr_event_t **events) {
❶uint32_t pending = bus_space_read_4(rootdev_bus_space, irq_base, offset);

  while (pending) {
  ❷int irq = ffs(pending) - 1;
    /* XXX: some pending bits are shared between BASIC and GPU0/1. */
    if (events[irq])
    ❸intr_event_run_handlers(events[irq]);
    pending &= ~(1 << irq);
  }
}
```

Listing 39: bcm2835 interrupt handler (`bcm2835_rootdev.c` [57])

Again, to enable interrupt we need dispatcher very similar to interrupt handler.

```
static void rootdev_enable_irq(intr_event_t *ie) {
  int irq = ie->ie_irq;
  assert(irq < NIRQ);

  if (irq < BCM2836_NIRQ) {
    /* Enable local IRQ. */
    enable_local_irq(irq);
  } else if (irq < BCM2835_INT_GPU1BASE) {
    /* Enable GPU0 IRQ. */
    enable_gpu_irq(irq - BCM2835_INT_GPU0BASE, BCM2835_INTC_IRQ1ENABLE);
  } else if (irq < BCM2835_INT_BASICBASE) {
    /* Enable GPU1 IRQ. */
    enable_gpu_irq(irq - BCM2835_INT_GPU1BASE, BCM2835_INTC_IRQ2ENABLE);
  } else {
    /* Enable base IRQ. */
    enable_gpu_irq(irq - BCM2835_INT_BASICBASE, BCM2835_INTC_IRQBENABLE);
  }
}
```

Listing 40: (`bcm2835_rootdev.c` [57])

To enable single interrupt we need to set suitable bit in register ❶.

```
static void enable_local_irq(int irq) {
  assert(irq < BCM2836_INT_NLOCAL);
  uint32_t reg = bus_space_read_4(rootdev_bus_space, rootdev_local_handle,
                                  BCM2836_LOCAL_TIMER_IRQ_CONTROLN(0));
❶bus_space_write_4(rootdev_bus_space, rootdev_local_handle,
                    BCM2836_LOCAL_TIMER_IRQ_CONTROLN(0), reg | (1 << irq));
}
```

Listing 41: (`bcm2835_rootdev.c` [57])

Disabling interrupts looks analogous.

For more information about interrupts see 2.1.3.

4.8.2 Timer

Timer is necessary if we want to run periodic tasks e.g. scheduler. Here I want to show simple implementation of driver for timer described at 2.1.4.

For start it is need to get current value of timer ❶. Then it is possible to set next tick time ❷ and at the end, timer can be enabled ❸.

```
static int arm_timer_start(timer_t *tm, unsigned flags __unused,
                           const bintime_t start __unused,
                           const bintime_t period) {
  arm_timer_state_t *state = ((device_t *)tm->tm_priv)->state;
  state->step = bintime_mul(period, tm->tm_frequency).sec;

  WITH_INTR_DISABLED {
  ❶uint64_t count = READ_SPECIALREG(cntpct_el0);
  ❷WRITE_SPECIALREG(cntp_cval_el0, count + state->step);
  ❸WRITE_SPECIALREG(cntp_ctl_el0, CNTCTL_ENABLE);
  }

  return 0;
}
```

Listing 42: (`timer.c` [58])

To stop timer it is only needed to set value of `cntp_ctl_el0` register.

```
static int arm_timer_stop(timer_t *tm) {
  WRITE_SPECIALREG(cntp_ctl_el0, CNTCTL_DISABLE);
  return 0;
}
```

Listing 43: (`timer.c` [58])

To get current time we need to read current value of timer ❶ and convert them to the form used by machine-independent part of Mimiker.

```
static bintime_t arm_timer_gettime(timer_t *tm) {
❶uint64_t count = READ_SPECIALREG(cntpct_el0);
  bintime_t res = bintime_mul(tm->tm_min_period, (uint32_t) count);
  bintime_t high_bits = bintime_mul(tm->tm_min_period,
                                    (uint32_t) (count >> 32));
  bintime_add_frac(&res, (high_bits.frac << 32));
  res.sec += (high_bits.sec << 32) + (high_bits.frac >> 32);
  return res;
}
```

Listing 44: (`timer.c` [58])

The most important part is interrupt handler. Here we triggers machine-dependent actions ❶ based on current time and at the end time of next tick is updated ❷.

```
static intr_filter_t arm_timer_intr(void *data /* device_t* */) {
  arm_timer_state_t *state = ((device_t *)data)->state;

❶tm_trigger(&state->timer);

  uint64_t prev = READ_SPECIALREG(cntp_cval_el0);
❷WRITE_SPECIALREG(cntp_cval_el0, prev + state->step);

  return IF_FILTERED;
}
```

Listing 45: timer interrupt handler (`timer.c` [58])

4.8.3 PL011

For PL011 device described at 2.2 we only need to implement the following functions:

It checks if receiver hardware queue is ready.

```c
static bool pl011_rx_ready(void *state) {
  pl011_state_t *pl011 = state;
  return (bus_read_4(pl011->regs, PL01XCOM_FR) & PL01X_FR_RXFE) == 0;
}
```

Listing 46: (pl011.c [59])

Puts character in uart. Transmitter hardware queue must be ready.

```c
static uint8_t pl011_getc(void *state) {
  pl011_state_t *pl011 = state;
  return bus_read_4(pl011->regs, PL01XCOM_DR);
}
```

Listing 47: (pl011.c [59])

It checks if transmitter hardware queue is ready.

```c
static bool pl011_tx_ready(void *state) {
  pl011_state_t *pl011 = state;
  return (bus_read_4(pl011->regs, PL01XCOM_FR) & PL01X_FR_TXFF) == 0;
}
```

Listing 48: (pl011.c [59])

It enables transmitter interrupt.

```c
static void pl011_tx_enable(void *state) {
  pl011_state_t *pl011 = state;
  set4(pl011->regs, PL011COM_CR, PL011_CR_TXE);
}
```

Listing 49: (pl011.c [59])

It disables transmitter interrupt.

```c
static void pl011_tx_disable(void *state) {
  pl011_state_t *pl011 = state;
  clr4(pl011->regs, PL011COM_CR, PL011_CR_TXE);
}
```

Listing 50: (pl011.c [59])

So all functions are simple wrappers for reading, setting, clearing bits.

4.9 Summary

In this chapter we have seen the most important pieces of code used for AArch64 port and Raspberry Pi 3 drivers. We have started with glue between user-space and kernel-space which is used by every single program. Next we have gone through MMU related code. It allows us to use virtual memory as a abstraction over resources. After that we have seen kernel bootstrapping and kernel address sanitizer initialization. At the end of core kernel code we became more familiar with exception handlers and context switching. Finally we have seen drivers for Raspberry Pi 3 that implement our hardware abstraction layer.

Chapter 5

Mimiker on Raspberry Pi 3

In this chapter I will show how to run Mimiker on Raspberry Pi 3 board with ARM-8 Cortex-A53 CPU.

Everything was tested with Debian 10 as a build machine [18], Raspberry Pi 3 Model 3B, MicroSD card and Segger J-Link EDU as hardware debugger [20].

5.1 Installation

5.1.1 Toolchain

There is a `toolchain` directory in a repository. There is a `Makefile` for each directory so you only need to run `make` from console. After that `deb` packages with toolchain will be built. Please be patient – compilation of toolchain is a long process.

These packages need to be installed by `dpkg` [19] command and currently they are supported only on Debian [18].

5.1.2 Configuration

Mimiker has a few build options that need to be set before compilation. These are:

- `BOARD` – build image for given board
- `CLANG` – use clang instead of gcc as a C compiler
- `LOCKDEP` – build with lock dependency validator
- `KASAN` – build with kernel address sanitizer
- `KGPROF` – build with kernel profiler

`BOARD` need to be set to `rpi3` in `config.mk`. For now only `KASAN` is well tested.

5.1.3 Compilation

You only need to run `make` command.

5.1.4 Final installation

I highly recommend to use sd card image of raspbian operating system [29]. That image already contains necessary firmware and configuration files on boot partition.

Copy kernel (`mimiker.img`) and initrd to memory card. Then you need to modify `kernel`, `arm_64bit`, `initramfs` and set `kernel_address` in `config.txt` to `0x200000` according to [6].

Memory card should be formatted in standard way – only one partition formatted as FAT32 is needed.

Note that due to the problems discussed at 5.2.4, 5.2.2 and active development, without automatic tests on Rasbperry Pi 3, there is a possibility that Mimiker will crash after launch.

5.1.5 Debugging

Hardware debugger

For debugging I have been using Segger J-Link EDU [20].

It implements JTAG (Joint Test Action Group) standard for verifying designs and testing printed circuit boards after manufacture.

I have been using that tool with OpenOCD software. Before we can start with software debugging we need to connect J-Link to Raspberry Pi 3. You can use the following diagram 5.1 from [7]:

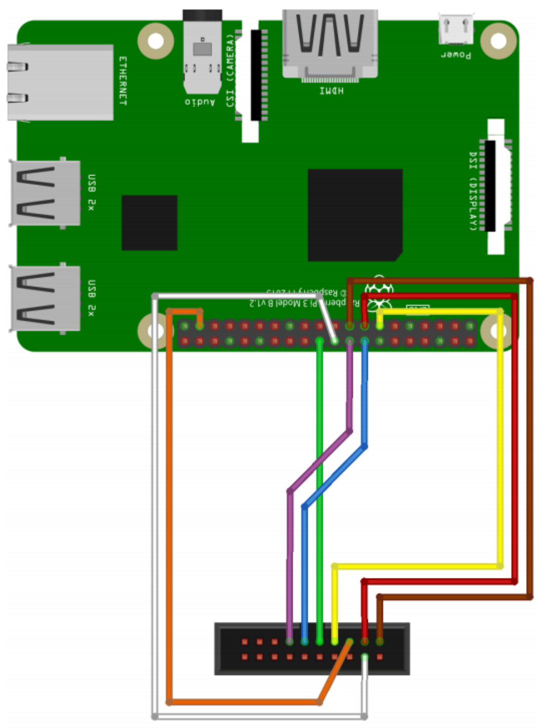


Figure 5.1: J-Link connection diagram for Raspberry Pi 3 [7]

Here is a photo of my setup 5.2:

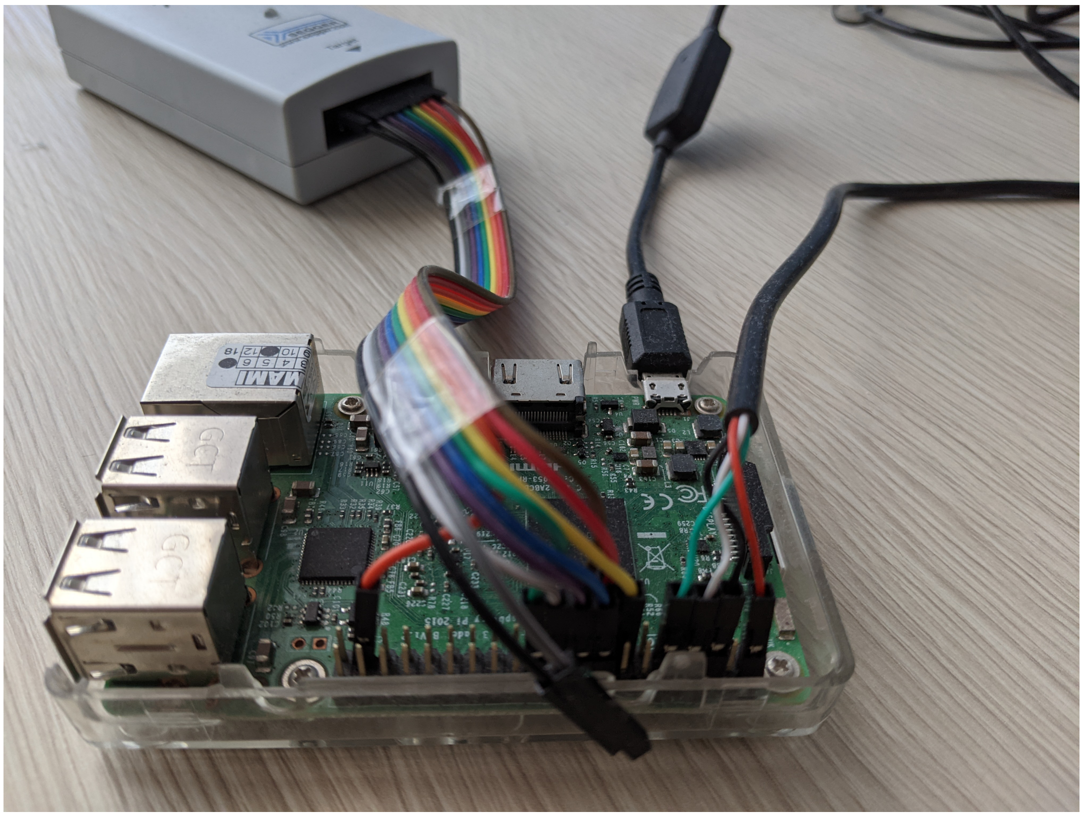


Figure 5.2: Raspberry Pi 3 with J-Link and UART

It includes additional UART connected on right side and power source via micro USB.

OpenOCD

For debugging you can use OpenOCD (it is included in our toolchain).

```
openocd -c "" -f jlink.cfg -f rpi3_64.cfg
```

With the following configuration for `jlink.cfg`:

```
adapter driver jlink
```

and following configuration for `rpi3_64.cfg`:

```
transport select jtag
reset_config trst_and_srst
adapter speed 1000
jtag_ntrst_delay 500
if { [info exists CHIPNAME] } {
  set _CHIPNAME $CHIPNAME
} else {
  set _CHIPNAME rpi3
}
if { [info exists DAP_TAPID] } {
   set _DAP_TAPID $DAP_TAPID
} else {
   set _DAP_TAPID 0x4ba00477
}
jtag newtap $_CHIPNAME cpu -expected-id $_DAP_TAPID -irlen 4
dap create $_CHIPNAME.dap -chain-position $_CHIPNAME.cpu
set _TARGETNAME_0 $_CHIPNAME.cpu0
set _TARGETNAME_1 $_CHIPNAME.cpu1
set _TARGETNAME_2 $_CHIPNAME.cpu2
set _TARGETNAME_3 $_CHIPNAME.cpu3
set _CTINAME_0 $_CHIPNAME.cti0
set _CTINAME_1 $_CHIPNAME.cti1
set _CTINAME_2 $_CHIPNAME.cti2
set _CTINAME_3 $_CHIPNAME.cti3
# The ARM Cross-Trigger Interface (CTI)
cti create $_CTINAME_0 -dap $_CHIPNAME.dap -ap-num 0 -baseaddr 0x80018000
target create $_TARGETNAME_0 aarch64 -dap $_CHIPNAME.dap -coreid 0 \
    -dbgbase 0x80010000 -cti $_CTINAME_0
cti create $_CTINAME_1 -dap $_CHIPNAME.dap -ap-num 0 -baseaddr 0x80019000
target create $_TARGETNAME_1 aarch64 -dap $_CHIPNAME.dap -coreid 1 \
    -dbgbase 0x80012000 -cti $_CTINAME_1
cti create $_CTINAME_2 -dap $_CHIPNAME.dap -ap-num 0 -baseaddr 0x8001A000
target create $_TARGETNAME_2 aarch64 -dap $_CHIPNAME.dap -coreid 2 \
    -dbgbase 0x80014000 -cti $_CTINAME_2
cti create $_CTINAME_3 -dap $_CHIPNAME.dap -ap-num 0 -baseaddr 0x8001B000
target create $_TARGETNAME_3 aarch64 -dap $_CHIPNAME.dap -coreid 3 \
    -dbgbase 0x80016000 -cti $_CTINAME_3
$_TARGETNAME_0 configure -event reset-assert-post "aarch64 dbginit"
$_TARGETNAME_0 configure -event gdb-attach { halt }
$_TARGETNAME_1 configure -event reset-assert-post "aarch64 dbginit"
$_TARGETNAME_1 configure -event gdb-attach { halt }
$_TARGETNAME_2 configure -event reset-assert-post "aarch64 dbginit"
$_TARGETNAME_2 configure -event gdb-attach { halt }
$_TARGETNAME_3 configure -event reset-assert-post "aarch64 dbginit"
$_TARGETNAME_3 configure -event gdb-attach { halt }
```

After that you can use `gdb` for remote debugging.

```
aarch64-mimiker-elf-gdb sys/mimiker.elf
    -ex 'set architecture aarch64'
    -ex 'file sys/mimiker.elf'
    -ex 'target extended-remote localhost:3333'
    -ex 'monitor reset init'
    -ex 'monitor targets rpi3.cpu0'
    -ex "monitor load_image sys/mimiker.img 0x200000 bin"
    -ex 'monitor reg pc 0x200000'
    -ex "load sys/mimiker.elf"
    -ex 'source .gdbinit'
```

For more information about debugging without operating system support and explanation of used commands see [7].

5.2 Challenges

Here I want to mention the most bothersome problems I encountered when running Mimiker on Raspberry Pi 3. Most of them are caused by differences between QEMU emulator and real hardware. QEMU doesn't emulate every single detail of Rapsberry Pi 3 so most of errors can't be detected by our CI system. They require installation and debugging on physical hardware which is a more complicated process than development in virtualized environment.

5.2.1 Boot process

On QEMU emulator we can boot directly from ELF (Executable and Linkable Format) but on physical machine it is not working. Kernel image needs to be a binary blob. We can achieve that by `objcopy` command:

```
objcopy -O binary mimiker.elf mimiker.img
```

5.2.2 Destroying `x0`

The most disturbing error that I have found is first instruction of kernel code:

```
0000000000200000 <_start>:
  200000:       14000010        b       200040 <_start+0x40>
        ...
  200010:       000af75c        .word   0x000af75c
  200014:       00000000        .word   0x00000000
  200018:       00000002        .word   0x00000002
        ...
  200038:      644d5241     .word   0x644d5241
  20003c:      00000000     .word   0x00000000
  200040:      aa0003f3     mov     x19, x0
  200044:      d53800a3     mrs     x3, mpidr_el1
  200048:      92400463     and     x3, x3, #0x3
  20004c:      f100007f     cmp     x3, #0x0
  200050:      54000001     b.ne    200050 <_start+0x50>
// b.any
  200054:      1000c3e3     adr     x3, 2018d0 <_bootmem_end>
  200058:      9100007f     mov     sp, x3
  20005c:      940004f7     bl      201438 <aarch64_init>
  200060:      9100001f     mov     sp, x0
  200064:      aa1303e0     mov     x0, x19
  200068:      9400050e     bl      2014a0 <__board_stack_veneer>
  20006c:      9100001f     mov     sp, x0
  200070:      14000512     b       2014b8 <__board_init_veneer>
```

Before first instruction `x0` contains address of atags or dtb. But for unknown reason first branch instruction destroys `x0` and put address of `pc` in that register. We can live without that by hardcoding that address in kernel but it is not the best solution.

5.2.3 Address alignment

During early initialization of MMU we set `SCTLR_SA0`, `SCTLR_SA` and `SCTLR_A` bits of `sctlr_el1`. They are responsible for checking alignment of kernel stack, user stack and memory access. These bits are not implemented by QEMU. As a result we get alignment exceptions on Raspberry Pi 3 because our implementation of `memcpy` doesn't meet these requirements. That issue has been resolved.

5.2.4 Cache control

In original implementation of Mimiker for MIPS architecture we didn't care about cache control. QEMU doesn't support cache and we never tried to run Mimiker on physical Malta board.

It also wasn't a problem for AArch64 implementation for QEMU emulator. Everything works without any support for caches. Unfortunately on real hardware details are different, now caches matter. It looks like running multiple threads in different address spaces causes cache mismatch for user-space processes. It means that one process uses caches of other process which was running before on the same core. It is a real problem because we can't test that in virtualized environment so our tests are useless for that kind of bugs.

Chapter 6

Summary

Adapting operating system to new architecture is a long journey. It requires knowledge of most parts of the kernel. Working with emulated environment is not the same as working with real hardware because emulator usually doesn't implement all details of hardware. Hardware will not forgive mistakes which could be ignored by emulator. First port is also a challenge because it needs to separate machine-dependent part from kernel and requires to create abstraction over hardware which will be machine-independent.

In my thesis I have prepared toolchain and infrastructure for developing Mimiker on AArch64 architecture and Rapsberry Pi 3 board. I have separated machine-dependent part of MIPS code from kernel and written the following components:

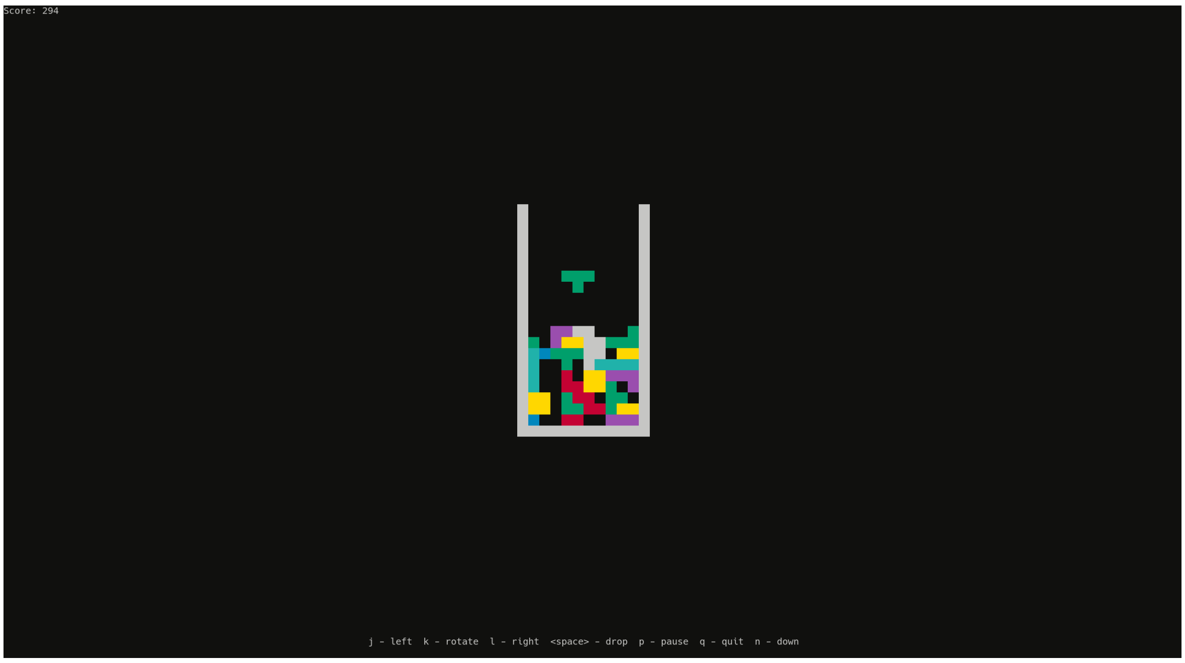
- kernel bootstrapping
- context switching
- exception handler
- pmap
- KASAN
- copy routines
- syscall handler
- timer driver
- UART driver

The results of that work are available in Mimiker repository [17]. Now we have:

- original MIPS implementation

- fully working AArch64 port
- drivers needed for run on Raspberry Pi 3 board
- automated tests in CI for AArch64 which pass
- integrated kernel address sanitizer
- infrastructure for future work on other architectures

The nice side effect is that everybody with basic knowledge can run Mimiker at home.



```
Score: 294

j - left  k - rotate  l - right  <space> - drop  p - pause  q - quit  n - down
```

Tetris game on Raspberry Pi 3 build.

The well known Tetris 6 game is an example of program that uses complex abstractions (for example system calls and terminal subsystem). Now we can run that program on AArch64 build with full functionality.

6.1 Future work

There are two big achievements of my work that can be further developed in the future.

First is a support for CPU that has many cores. It gives opportunity to add SMP support for Mimiker. It requires changes in multiple kernel subsystems. The most affected one will be scheduler but it is not the only one that needs to be changed. The VFS subsystem has substantial problems with locking – running multiple processes that use file system causes deadlocks. Locking mechanism also needs to be adapted – e.g. spinlocks assume that there is only a single CPU core.

Second one is bootstrapping Mimiker on a physical board. We know that QEMU is not perfect and hides some of our bugs. We may invest time in tests infrastructure that uses multiple Raspberry Pi 3 boards connected to development serve with setup similar to the one used in my work. It should give an opportunity to track changes in machine-dependent subsystems of Mimiker without QEMU's quirks.

Now that the port on AArch64 is as mature as the MIPS version, we can start implementing drivers for the rest of the devices available in Raspberry Pi 3. Drivers for USB and video will make it possible to use Raspberry Pi 3 as a multimedia device with Mimiker operating system.

Bibliography

[1] *C99 standard (ISO/IEC 9899:1999)*

[2] *ARM Cortex-A Series Version: 1.0 Programmer's Guide for ARMv8-A*

[3] *ARMv8-A Address Translation Version: 1.0*

[4] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, Second Edition

[5] FreeBSD manpages, *pmap(9)*
`https://www.freebsd.org/cgi/man.cgi?query=pmap&apropos=0&sektion=0&manpath=FreeBSD+13.0-current&arch=default&format=html`

[6] Raspberry Pi Documentation, *config.txt*
`https://www.raspberrypi.org/documentation/configuration/config-txt/`

[7] Michał Barnaś, *Hardware Abstraction Layer For ARMv8 Processors*

[8] NetBSD manpages, *signal(7)*
`https://man.netbsd.org/signal.7`

[9] NetBSD manpages, *kill(2)*
`https://man.netbsd.org/kill.2`

[10] Bryant, O'Hallaron, *Computer Systems A Programmer's Perspective*, Second Edition

[11] Charles D. Cranor, *Design and implementation of the UVM virutal memory system*

[12] Jakub Piecuch, *Implementation of the Terminal Subsystem and Job Control in the Mimiker Operating System*

[13] Julian Pszczołowski, *Integrating the Kernel Address Sanitizer into the Mimiker Operating System*

[14] Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems*, Fourth Edition

[15] Boradcom *BCM2837 ARM Peripherals*

[16] Mimiker web page, `https://mimiker.ii.uni.wroc.pl`

[17] Mimiker repository, `https://github.com/cahirwpz/mimiker`

[18] Debian, *The Universal Operating System*
`https://www.debian.org/`

[19] dpkg suite *Debian manpages*
`https://manpages.debian.org/buster/dpkg/dpkg.1.en.html`

[20] segger web page
`https://www.segger.com/products/debug-probes/j-link/`

[21] BCM2837 documentation
`https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md`

[22] Arm Cortex-A53 MPCore Processor Technical Reference Manual
`https://developer.arm.com/documentation/ddi0500/j/`

[23] NetBSD manpages, *queue(3)*
`https://man.netbsd.org/queue.3`

[24] NetBSD manpages, *bcopy(3)*
`https://man.netbsd.org/bcopy.3`

[25] Broadcom webpage
`https://www.broadcom.com/`

[26] Mimiker mutex API
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/mutex.h?r=bfb3bae9`

[27] Mimiker spin lock API
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/spinlock.h?r=27b8c19a`

[28] Mimiker conditional variable API
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/condvar.h?r=71604845`

[29] Raspberry Pi operating system images
`https://www.raspberrypi.org/software/operating-systems/`

[30] Devicetree Specification
`https://github.com/devicetree-org/devicetree-specification`

[31] Mimiker source code – `include/sys/ucontext.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/ucontext.h?r=4ba50916`

[32] Mimiker source code – `include/aarch64/mcontext.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/aarch64/mcontext.h?r=731f9b87`

[33] Mimiker source code – `sys/mips/pmap.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/mips/pmap.c?r=d5439d54`

[34] Mimiker source code – `sys/mips/boot.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/mips/boot.c?r=2609772a`

[35] Mimiker source code – `sys/kern/kasan.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/kern/kasan.c?r=fc47d4fd`

[36] Mimiker source code – `include/dev/uart.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/dev/uart.h?r=2609772a`

[37] Mimiker source code – `sys/drv/uart.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/drv/uart.c?r=2609772a`

[38] Mimiker source code – `sys/kern/uart_tty.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/kern/uart_tty.c?r=2609772a`

[39] Mimiker source code – `sys/aarch64/copy.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/copy.S?r=8dda89f3`

[40] Mimiker source code – `include/aarch64/syscall.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/aarch64/syscall.h?r=6da6392f`

[41] Mimiker source code – `include/sys/sysent.h`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/include/sys/sysent.h?r=4cae32de`

[42] Mimiker source code – `sys/kern/syscalls.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/kern/syscalls.c?r=8ae97262`

[43] Mimiker source code – `sys/aarch64/trap.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/trap.c?r=db7eaf68`

[44] Mimiker source code – `lib/csu/aarch64/crt0.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/lib/csu/aarch64/crt0.S?r=4cb7508a`

[45] Mimiker source code – `sys/aarch64/sigcode.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/sigcode.S?r=9c185874`

[46] Mimiker source code – `sys/aarch64/signal.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/signal.c?r=9c185874`

[47] Mimiker source code – `lib/libc/gen/aarch64/_setjmp.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/lib/libc/gen/aarch64/_setjmp.S?r=f0de79b8`

[48] Mimiker source code – `lib/libc/gen/aarch64/longjmp.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/lib/libc/gen/aarch64/longjmp.c?r=1d93d219`

[49] Mimiker source code – `lib/libc/gen/aarch64/setjmp.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/lib/libc/gen/aarch64/setjmp.S?r=96506ee9`

[50] Mimiker source code – `lib/libc/gen/aarch64/sigsetjmp.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/lib/libc/gen/aarch64/sigsetjmp.S?r=f0de79b8`

[51] Mimiker source code – `sys/aarch64/pmap.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/pmap.c?r=fd5537f8`

[52] Mimiker source code – `sys/aarch64/boot.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/boot.c?r=fd5537f8`

[53] Mimiker source code – `sys/aarch64/start.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/start.S?r=fd5537f8`

[54] Mimiker source code – `sys/aarch64/board.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/board.c?r=fd5537f8`

[55] Mimiker source code – `sys/aarch64/evec.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/evec.S?r=1f315016`

[56] Mimiker source code – `sys/aarch64/switch.S`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/switch.S?r=96506ee9`

[57] Mimiker source code – `sys/drv/bcm2835_rootdev.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/drv/bcm2835_rootdev.c?r=2609772a`

[58] Mimiker source code – `sys/aarch64/timer.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/aarch64/timer.c?r=2609772a`

[59] Mimiker source code – `sys/drv/pl011.c`
`https://mimiker.ii.uni.wroc.pl/source/xref/mimiker/sys/drv/pl011.c?r=2609772a`