Unified resource allocator, not Operating System

Systemy Wbudowane

Wydział Informatyki Elektroniki i Telekomunikcaji

Pazdźiernik 2019 - Styczeń 2020

Mateusz Hurbol, Piotr Kotara

Spis treści

Rozdział 1

Wstęp

Celem tego przedsięwzięcia jest zbadanie procesu tworzenia systemu operacyjnego na architekturze ARMv8a w trybie Aarch64.

D0 uzyskania celu wykorzystana zostanie para mikrokomputerów - RaspberryPi 4B i RaspberryPi 3B posiadające odpowiednio procesory Cortex-A72 i Cortex-A53.

Jako język programowania wybraliśmy język Rust http://rustlang.org będący językiem programowania systemowego z gwarancjami bezpieczeństwa pamięci. Umożliwi on nam stworzenie bezpieczniejszego systemu niż gdybyśmy go pisali w języku C.

Celem końcowym projektu jest uzyskanie logicznej separacji procesów za pomocą wbudowanego w procesor układu MMU (Memory Management Unit).

1.1 Źródła

Projekt rozpoczęliśmy od zapoznania się z teorią tworzenia systemów operacyjnych jak i działania naszych platform sprzętowych. Poszukując podobnych projektów w Internecie natrafialiśmy głównie na projekty i poradniki oparte o język C lub nawet czyste Assembly. Wśród nich wyróżniały się trzy projekty, które stanowiły dla nas źródło dodatkowej wiedzy:
//TODO OPIS

1. Circle w języku C++
   (https://github.com/rsta2/circle.git)

2. Redox OS - system operacyjny napisany w języku Rust na maszyny x86_64
   (https://redox-os.org)

3. Poradnik dostępowy do sprzętu RaspberryPi V1
   (https://github.com/rust-embedded/rust-raspi3-OS-tutorials.git)

Rozdział 2

Droga do `fn main()`

2.1 Skrypt linkera

Strukturę pamięci podczas inicjalizacji naszego systemu określamy za pomocą skryptu linkera:

```
ENTRY(_boot_cores);

SECTIONS
{
```

Zaczynamy od offsetu do `0x80000`. Pod ten adres skacze bootloader znajdujący się na GPU. Następnie definiujemy sekcję `text`, w której znajduje sie kod naszego kernela.

```
    . = 0x80000;

    .text :
    {
        KEEP(*(.text.boot)) *(.text .text.*)
    }
```

Po tej sekcji znajduje się sekcja danych tylko do odczytu i sekcja danych.

```
    .rodata :
    {
        *(.rodata .rodata.*)
    }

    .data :
    {
        *(.data .data.*)
    }
```

Następnie znajduje się segment zawierający statycznie alokowane zmienne.

```
    .bss ALIGN(8):
    {
```

```
        __bss_start = .;
        *(.bss .bss.*)
        *(COMMON)
        __bss_end = .;
    }
    .end ALIGN(8):
    {
        __binary_end = .;
    }
    .vectors ALIGN(2048):
    {
        *(.vectors)
    }
    .heap ALIGN(4096):
    {
        *(.heap .heap.*)
    }

    /DISCARD/ : { *(.comment) *(.gnu*) *(.note*) *(.eh_frame*) }
}
```

2.2 Inicjalizacja rdzeni procesora

Nasza przygoda zaczyna się w pliku *boot.S*. Jest on w większości oparty o poradnik dostępowy do sprzętu (pozycja 3 w źródłach).

```
.section ".text.boot"

.global _boot_cores

_boot_cores:
    // read cpu id, stop slave cores
    mrs     x1, mpidr_el1
    and     x1, x1, #3
    cbz     x1, 2f
    // cpu id > 0, stop
1:  wfe
    b       1b
2:  // cpu id == 0
    // set stack before our code
    ldr     x1, =_boot_cores
    mov     sp, x1
    // jump to Rust code, should not return
    bl      reset
    // for failsafe, halt this core too
    b       1b
```

Prześledźmy go:

```
_boot_cores:
    // read cpu id, stop slave cores
    mrs     x1, mpidr_el1
    and     x1, x1, #3
    cbz     x1, 2f
    // cpu id > 0, stop
1:  wfe
    b       1b
```

Na początku pobieramy wartość rejestru `mpdir_el1` w którym na dwóch najmłodszych bitach znajduje się ID danego rdzenia. Maskujemy te bity za pomocą `and` i sprawiamy, tylko rdzeń i ID 0 nie wpadnie w nieskończoną pętlę.

```
    2:  // cpu id == 0
    // set stack before our code
    ldr     x1, =_boot_cores
    mov     sp, x1

    // jump to Rust code, should not return
    bl      reset

    // for failsafe, halt this core too
    b       1b
```

Rdzeniowi o ID = 0 ustawiamy teraz stack pointer na adres znajdującą się przed naszym kodem. Adres ten ustalamy w linkerze. Następnie skaczemy do funkcji `reset`.

2.3 Funkcja `reset`

Głównym zadaniem funkcji reset jest wyzerowanie segmentu `bss`.

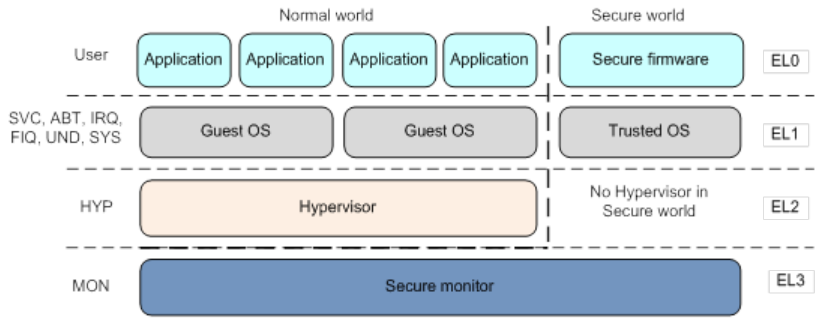
```
#[no_mangle]
pub unsafe extern "C" fn reset() -> ! {
    extern "C" {
        // Boundaries of the .bss section, provided by the linker script
        static mut __bss_start: u64;
        static mut __bss_end: u64;
    }

    // Zeroes the .bss section
    r0::zero_bss(&mut __bss_start, &mut __bss_end);
```

Następnie przekazuje ona wykonanie programu do funkcji zmieniającej Exception Level procesora.

```
    extern "Rust" {
        fn main() -> !;
    }
    mode::ExceptionLevel::drop_to_el1(main);
    //main();
}
```

2.4 Exception Levels w AArch64

W architekturze AArch64 wyróżniono 4 tak zwane poziomy wyjątków. Obrazują one różne tryby pracy procesora, od najbardziej uprzywilejowanego (EL3) do najmniej (EL0). Ich zastosowania przedstawiono na poniższym rysunku. W naszym systemie operacyjnym wykorzystujemy EL1 do zadań jądra systemu oraz EL0 dla zadań użytkownika.



Rysunek 2.1: Diagram Exception levels(Źródło:http://140.120.7.21/LinuxRef/DIYBigData/OdroidXU4/ArmPrivilegeLevel.html)

2.5 Funkcja drop_to_el1

Jest to funkcja odpowiedzialna za zmienienie poziomu procesora na EL1.

```
pub unsafe fn drop_to_el1(el1_entry : unsafe fn () -> !) -> ! {
        const STACK_START: u64 = 0x80_000;
        asm::initialize_timers_el1();
        match ExceptionLevel::get_current() {
            Self::User => halt(),
            Self::Kernel => halt(),

            Self::Hypervisor => {
                asm::set_el1_stack_pointer(STACK_START);
                ...
            },

            Self::Firmware => {
                asm::set_el1_stack_pointer(STACK_START);
                ...
            },
```

```
        asm::eret();
```

W naszym systemie własnościowy firmware używanych płytek zrzuca procesor do stanu EL2. Powyższa funkcja umożliwia jednak skok do EL1 zarówno z EL2 jak i EL3. Wykrywa ona też niepoprawne jej wywołanie (czyli z EL1 lub EL0). Zmiana Exception Level następuje poprzez wpisanie odpowiednich wartości do specjalnych rejestrów procesora. Procesor zmieni Exception Level przy użyciu rozkazu powrotu z przerwania, więc jako adres powrotu z przerwania ustawiamy adres naszej funkcji main (`kernel_entry`).

Rozdział 3

Co kryje się za `println!("Hello, World!")`

W naszym systemie operacyjnym główna część interakcji z użutkownikiem dokonuje się za pomocą UART. Na naszych płytkach wyprowadzony jest on na interfejs GPIO.

3.1 Inicjalizacja GPIO

Obsługa GPIO odbywa się za pomocą rejestrów kontrolera GPIO, które są zmapowane do pamięci wirtualnej. W celu bezpiecznego korzystania z nich wykorzystaliśmy dostępną w Rust abstrakcje do obsługi rejestrów bitowych. Poniżej fragment wspomnianej abstrakcji:

```
register_bitfields! {
    u32,
    GPFSEL1 [
        /// Pin 15
        FSEL15 OFFSET(15) NUMBITS(3) [
            Input = 0b000,
            Output = 0b001,
            RXD0 = 0b100, // UART0     - Alternate function 0
            RXD1 = 0b010  // Mini UART - Alternate function 5

        ],
...

pub const GPFSEL1: *const ReadWrite<u32, GPFSEL1::Register> =
    (MMIO_BASE + 0x0020_0004) as *const ReadWrite<u32, GPFSEL1::Register>;

pub const GPFSEL2: *const ReadWrite<u32, GPFSEL2::Register> =
    (MMIO_BASE + 0x0020_0008) as *const ReadWrite<u32, GPFSEL2::Register>;

pub const GPSET0: *const WriteOnly<u32, GPSET0::Register> =
    (MMIO_BASE + 0x0020_001C) as *const WriteOnly<u32, GPSET0::Register>;
...
```

Jak widzimy abstrakcja polega najpierw na zadeklarowaniu dozwolonych stanów rejestru a następnie na wskazaniu gdzie dany rejestr się zaczyna.

3.2 Inicjalizacja UART

Wykorzystywane płytki wyposażone są w dwa różne interfejsy UART, z czego jeden jest bardziej rozbudowany od drugiego. W naszym systemie wykorzystujemy ten o większych możliwościach, a jego inicjalizacja polega na podążaniu za poleceniami jego dokumentacji. Rejestry kontrolera UART również zmapowane są do pamięci wirtualnej.

3.2.1 trait Uart

Definiujemy w naszym systmie abstrakcję nad kontrolerem UART o następującym interfejsie:

```
trait Uart {
    pub const fn new() -> Uart {
        Uart
    }

    /// Returns a pointer to the register block
    fn ptr() -> *const RegisterBlock;

    ///Set baud rate and characteristics (115200 8N1) and map to GPIO
    pub fn init(&self, mbox: &mut mbox::Mbox) -> UartResult;

    /// Send a character
    pub fn send(&self, c: char);

    /// Receive a character
    pub fn getc(&self) -> char;

    /// Display a string
    pub fn puts(&self, string: &str);

    /// Display a binary value in hexadecimal
    pub fn hex(&self, d: u32);
}
```

3.3 Makra do komunikacji przez UART

```
pub fn _print(args: fmt::Arguments) {
    use core::fmt::Write;
    unsafe {
        UART.write_fmt(args).unwrap();
    }
}

#[macro_export]
macro_rules! print {
    ($($arg:tt)*) => ($crate::io::_print(format_args!($($arg)*)));
}

#[macro_export]
macro_rules! println {
    () => ($crate::print!("\n"));
    ($($arg:tt)*) => ($crate::print!("{}\n", format_args!($($arg)*)));
}
```

Rozdział 4

`free() || !free()` that's not a question

Rust zapewnia dużo Funkcjonalności w samej bibliotece Core, ale aby uzyskać dostęp do struktur takich jak Wektor Kolejka czy Hashmap-a należy Sięgnąć do biblioteki Alloc.

4.1 `use alloc::GlobalAllocator`

Po dodaniu do projektu biblioteki

```
extern crate alloc;
```

otrzymujemy bład

```
error: no global memory allocator found but one is required;
       link to std or add #[global_allocator]
       to a static item that implements the GlobalAlloc trait.
```

Ponieważ w projekcie wbudowanych nie możemy użyć biblioteki standardowej, musimy zaimplementować `trait GlobalAlloc`.

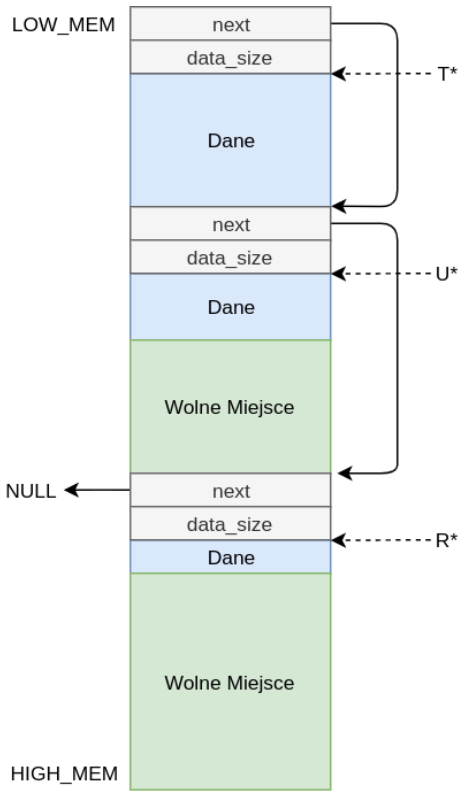
Spójrzmy na definicje co należy zaimplementować (Wyczyszczoną ze zbędnych elementów):

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
}
```

GlobalAlloc jest bardzo wrażliwą częścią programu co widać w tym że jest to `unsafe trait`, co jest wskazówką dla implementatora że należy dokładnie zastosować się do niezmienikow i założenie umieszczonych w dokumentacji. Jednak jego poprawne zaimplementowanie umożliwi nam skorzystanie z wielu bezpiecznych abstrakcji w dalszej części projektu.

Alokator jest zaimplementowany jako Linked-Lista utworzona w pewnej pamięci ciągłej, gdzie każda alokacja jest reprezentowana jako węzeł listy.

```
pub struct SystemAllocator
    heap_size: usize,
    first_block: core::cell::UnsafeCell<Block>,
}

pub struct Block {
    next: *mut Block,
    data_size: usize,
}
```



Rysunek 4.1: Pamięć Alokatora

4.2 fn alloc()

Przeanalizujmy funkcje alloc:

```
unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
    let mut previous = self.block_list();
    let mut current = (*previous).next;
```

Przygotowujemy wskaźniki którymi będziemy sie poruszać po liscie alokator

```
    while current != null_mut() {
```

Dopóki nie dojdziemy do końca listy

```
    if is_the_space_big_enough(previous, layout, current as usize) {
        // FOUND PLACE
        let mut new_block =
            align_address(previous as usize + (*previous).size_of(), layout.align())
                as *mut Block;
        (*new_block).next = current;
        (*new_block).data_size = layout.size();
        (*previous).next = new_block;
        let ptr = (new_block as usize + size_of::<Block>()) as *mut u8;
        return ptr;
    }
```

Jeżeli pomiędzy kolejnymi blokami jest dostatecznie dużo miejsca to wstaw tam nowy blok, wepnij go do listy i zwróć wskaźnik na nowo zalakowane miejsce.

```
        previous = current;
        current = (*current).next;
```

W przeciwnym wypadku Idz dalej po liście

```
    if is_the_space_big_enough(previous, layout, self.heap_end()) {
        // FOUND PLACE
        let mut new_block =
            align_address(previous as usize + (*previous).size_of(), 8)
                as *mut Block;
        (*new_block).next = null_mut();3,50
        (*new_block).data_size = layout.size();
        (*previous).next = new_block;
        let ptr = (new_block as usize + size_of::<Block>()) as *mut u8;
        return ptr;
    }
```

Gdy znajdziesz sie na końcu listy spróbuj wstawić Węzeł na końcu.

```
    return null_mut();
}
```

Jesli to również się nie udało zwróć `nullptr`.

4.3 fn dealloc()

Funkcja dealloc jest doklednie funkcja usuwania elementu z LinkedList-y.

```
unsafe fn dealloc(&self, ptr: *mut u8, _layout: Layout) {
    let block = ptr.offset(-(size_of::<Block>() as isize)) as *mut Block;
```

Przygotowywujemy wskaźniki robocze.

```
    let mut previous = self.block_list();
    let mut current = (*previous).next;
```

Wyszukujemy naszego bloku

```
    // TOTALY UNSAFE FOR NOW
    while current != block && current != null_mut() &&
          (current as u64) < (block as u64) {
        previous = current;
        current = (*current).next;
    }
```

Jeżeli udało się znaleźćbBlok to wypinamy go z listy

```
    if current != null_mut() {
        (*previous).next = (*current).next;
    }
}
```

Rozdział 5

I'm not sorry to interrupt you

Kontroler przerwań różni się z RPi3 od RPi4 (w tej drugiej występuje zewnętrzny kontroler przerwań GICv2). Mimo to nad obsługą poszczególnych kontrolerów wystawiamy wspólne API dla obu urządzeń

5.1 Struktura `ExceptionContext`

Jest to struktura przechowująca kontekst przerwania

```
#[repr(C)]
#[derive(Debug)]
pub struct ExceptionContext {
    // General Purpose Registers
    pub gpr: GPR,
    pub spsr_el1: u64,
    pub elr_el1: u64,
    pub esr_el1: u64,
    pub sp_el0: u64,
    pub far_el1: u64,
}
```

Pole `gpr`

Pole to jest strukturą przechowującą rejestry ogólnego przeznaczenia:

```
#[repr(C)]
#[derive(Debug)]
pub struct GPR {
    pub x: [u64; 30],
    pub lr : u64,
}
```

Pole `spsr_el1`

Jest to pole przechowujące zapisany rejestr statusowy przy przerwaniu do poziomu EL1 (Saved Process State Register)

Pole `elr_el1`

Jest to pole przechowujące adres powrotu z przewania do poziomu EL1 (Exception Link Register)

Pole `esr_el1`

Jest to pole przechowujące informacje o rodzaju przerwania do poziomu EL1 (Exception Syndrome Register)

Pole `sp_el0`

Jest to pole przechowujące wskaźnik stosu wykorzystywany na EL0.

Pole `far_el1`

Jest to pole przechowujące Modified Virtual Address miejsca przy którym wystąpił fault.

5.2 Makro exception_handler

Jest to assemblerowe makro służące do obsługi przychodzącego przerwania:

```
.macro exception_handler handler, exception_level, id
.balign 0x80

    sub     sp,  sp,  #16 * 18

    stp     x0,  x1,  [sp, #16 * 0]
    stp     x2,  x3,  [sp, #16 * 1]
    stp     x4,  x5,  [sp, #16 * 2]
    stp     x6,  x7,  [sp, #16 * 3]
    stp     x8,  x9,  [sp, #16 * 4]
    stp     x10, x11, [sp, #16 * 5]
    stp     x12, x13, [sp, #16 * 6]
    stp     x14, x15, [sp, #16 * 7]
    stp     x16, x17, [sp, #16 * 8]
    stp     x18, x19, [sp, #16 * 9]
    stp     x20, x21, [sp, #16 * 10]
    stp     x22, x23, [sp, #16 * 11]
    stp     x24, x25, [sp, #16 * 12]
    stp     x26, x27, [sp, #16 * 13]
    stp     x28, x29, [sp, #16 * 14]
```

Na początku zostaje zadeklarowana ramka stosu, do której następnie zapisywany jest kontekst procesora.

```
    mrs     x1,  SPSR_EL1
    mrs     x2,  ELR_EL1
    mrs     x3,  ESR_EL1
    mrs     x5, FAR_EL1
```

Następnie do rejestrów x1-x3 oraz x5 zapisywane są wartości rejestrów zawierających informacje o danym przerwaniu.

```
    .if \exception_level == 0
        mrs x4, SP_EL0
    .else
        add x4, sp, #16 * 18
    .endif
```

W zależności od poziomu przerwania podawanego jako argument makra do x4 zapisywany jest albo adres przed początkiem ramki stosu albo wartość sczytana z rejestru. `SP_EL0`.

```
    stp     x30, x1,  [sp, #16 * 15]
    stp     x2,  x3,  [sp, #16 * 16]
    stp     x4,  x5,  [sp, #16 * 17]

    mov     x0,  sp
    mov     x1, \id
    bl      \handler
```

Sczytane wartości zapisywane są do ramki stosu, do x0 zapisywany jest wskaźnik stosu, który wskazuje na dane umiejscowione zgodnie ze strukturą `ExeptionContext` a do x1 id przerwania (są to argumenty dla funkcji handlera, do której następnie następuje skok).

```
    .if \exception_level == 0
        b restore_context_el0
    .else
        b restore_context_el1
    .endif
.endm
```

Po powrocie z funkcji handlera w zależności od poziomu przerwania wykonywany jest skok do odpowiedniego makra assemblera służącego przywróceniu kontekstu procesora sprzed przerwania.

5.3 Handlery przerwań

Handler przerwania to funkcja przyjmująca dwa argumenty: referencje na strukturę `ExceptionContext` oraz id przerwania. Sygnatura handlera jest następująca:

```
#[no_mangle]
pub unsafe extern "C" fn sample_handler(context: &mut ExceptionContext, id: usize)
↪  -> ! {
```

Oba te argumenty dostarcza makro zapisania kontekstu procesora przy wejściu w przerwanie.

5.3.1 Linkowanie handlerów przerwań

Handlery przerwań zdefiniowane są w pliku `interrupts/handlers.rs`. Jednak aby zapewnić elastyczność oraz ułatwić debugowanie w linkerze umieszczono dyrektywy `PROVIDE`.

```
PROVIDE(current_el0_synchronous    = default_interupt_handler);
PROVIDE(current_el0_irq            = default_interupt_handler);
PROVIDE(current_el0_serror         = default_interupt_handler);

PROVIDE(current_elx_synchronous    = default_interupt_handler);
PROVIDE(current_elx_irq            = default_interupt_handler);
PROVIDE(current_elx_serror         = default_interupt_handler);

PROVIDE(lower_aarch64_synchronous = default_interupt_handler);
PROVIDE(lower_aarch64_irq         = current_elx_irq);
PROVIDE(lower_aarch64_serror      = default_interupt_handler);

PROVIDE(lower_aarch32_synchronous = default_interupt_handler);
PROVIDE(lower_aarch32_irq         = default_interupt_handler);
PROVIDE(lower_aarch32_serror      = default_interupt_handler);
```

Dzięki nim jeśli linker nie znajdzie w kodzie zdefiniowanych funkcji o nazwach po lewej stronie znaku równości podlinkuje w jej miejsce funkcje o nazwie z prawej strony znaku równości. Jak widzimy w przerażającej większości będzie to `default_interupt_handler`.

5.3.2 `default_interupt_handler`

Jest to domyślny handler przerwania wyoływany gdy nie został zdefiniowany specyficzny dla danego rodzaju przerwania handler.

```
#[no_mangle]
pub unsafe extern "C" fn default_interupt_handler(context: &mut ExceptionContext,
↪   id: usize) -> ! {
    super::disable_irqs();
    println!("Interupt Happened of ID-{}:\n  SP : {:#018x}\n {}", id, context as
↪   *mut ExceptionContext as u64, *context);
    println!("{:?}", boot::mode::ExceptionLevel::get_current());
    context.esr_el1 = 0;

    loop {}
}
```

Wyłącza on obsługę przerwań i informuje o ID przerwania które spowodowało jego wywołanie. Wyświetla również całą strukturę `ExceptionContext` i wchodzi w pustą nieskończoną pętle.

5.4 Tablica wektorów przerwań

Tablica wektorów przerwań zdefiniowana jest w pliku `interupts/vector_table.S`.

```
section .vectors, "ax"
.global __exception_vectors_start
__exception_vectors_start:
    exception_handler current_el0_synchronous 1 0 // 0x000
    exception_handler current_el0_irq         1 1 // 0x080
    fiq_dummy                                     // 0x100
    exception_handler current_el0_serror      1 2 // 0x180

    exception_handler current_elx_synchronous 1 3 // 0x200
    exception_handler current_elx_irq         1 4 // 0x280
    fiq_dummy                                     // 0x300
    exception_handler current_elx_serror      1 5 // 0x380

    exception_handler lower_aarch64_synchronous 0 6 // 0x400
    exception_handler lower_aarch64_irq         0 7 // 0x480
    fiq_dummy                                     // 0x500
    exception_handler lower_aarch64_serror      0 8 // 0x580

    exception_handler lower_aarch32_synchronous 0 9  // 0x600
    exception_handler lower_aarch32_irq         0 10 // 0x680
    fiq_dummy                                     // 0x700
    exception_handler lower_aarch32_serror      0 11 // 0x780
```

Podpięte są w niej wywołania opisanego wcześniej makra `exception_handler` z różnymi handlerami, poziomami przerwań oraz ich ID. Tablica wektorów znajduje sie między segmentem `bss` a stertą i jest wyrównany do 2048 bajtów o czym mówi skrypt linkera:

```
.bss ALIGN(8):
{
   (...)
}
.end ALIGN(8):
{
    __binary_end = .;
}
.vectors ALIGN(2048):
{
    *(.vectors)
}
.heap ALIGN(4096):
{
    *(.heap .heap.*)
}
```

5.5 Inicjalizacja przerwań

5.5.1 Inicjalizacja tablicy wektorów

Do ustawienia tablicy wektorów służy funkcja `set_vector_table_pointer` zdefiniowana następująco:

```
    #[inline(always)]
pub fn set_vector_table_pointer(address: u64) {
    unsafe {
        asm!("msr vbar_el1, $0" : :  "r"(address) : : "volatile");
    }
}
```

Wpisuje ona do rejestru `vbar_el1` adres początku tablicy przerwań przekazywany w parametrze funkcji. Wywoływana jest ona w funkcji `kernel_entry` przy inicjalizacji jądra systemu:

```
    (...)
    print!("Initializing Interupt Vector Table: ");
    unsafe {
        let exception_vectors_start: u64 = &__exception_vectors_start as *const _
↪   as u64;
        println!("{:x}", exception_vectors_start);
        interupt::set_vector_table_pointer(exception_vectors_start);
    }
    (...)
```

5.5.2 Włączanie/wyłączanie obsługi przerwań

Do przełączania obsługi przerwań wykorzystywane są rejestry procesora `daifset` i `daifclr`. Poszczególne bity tych rejestrów odpowiadają za włączanie obsługi poszczególnych typów przerwań. W naszym przypadku wykorzystujemy do obsługi tych rejestrów dwie funkcje:

```
#[inline(always)]
pub fn disable_irqs() {
    unsafe {
        asm!("msr daifset, #15" : : : : "volatile");
    }
}
#[inline(always)]
pub fn enable_irqs() {
    unsafe {
        asm!("msr daifclr, #15" : : : : "volatile");
    }
}
```

5.6 Przykład wykorzystania przerwań - timer

Przykładowym przypadkiem użycia przerwań jest konfiguracja przerwań zegarowych w celu późniejszego cyklicznego wywoływania schedulera w kontekście tego przerwania. Wykorzystujemy do tego zegar per-core stanowiący część procesorów w tej architekturze.

5.6.1 Abstrakcja timer'a

Do obsługi zegarów zdefiniowaliśmy odpowiedniego traita:

```
trait Timer {
    fn get_time() -> u64;
    fn interupt_after(ticks: u32) -> Result<(), Error>;
    fn enable();
    fn disable();
    fn get_frequency() -> u32;
}
```

Do obsługi opisanego wcześniej konkretnego zegara wykorzystujemy strukturę `ArmTimer` implementującą powyższy trait. Implementacje funkcji wymaganych przez trait w tym wypadku sprowadzają się do inline'owanych funkcji polegających na zapisaniu lub zczytaniu wartości do odpowiedniego rejestru; dla przykładu funkcja `enable()` zdefiniowana jest następująco:

```
impl super::Timer for ArmTimer {
    fn enable() {
        ArmTimer.route_clock.set(0x8);
        let val: u32 = 1;
        unsafe {
            asm!("msr cntv_ctl_el0, $0" : : "r"(val) : : "volatile");
        }
    }
```

5.6.2 Inicjalizacja przerwań zegarowych

W funkcji `kernel_entry` inicjalizowane są przerwania zegarowe. Na początku po inicjalizacji tablicy wektorów oraz włączeniu obsługi przerwań ustawiany jest czas do następnego przerwania i timer jest włączany.

```
println!("freq: {}", Timer::get_frequency());

interupt::enable_irqs();
Timer::interupt_after(Timer::get_frequency());
Timer::enable();
println!("Timer enabled");
```

5.6.3 Handler przerwań zegarowych

Przerwania timera zaliczają się do kategorii obsługiwanej przez handler `current_elx_irq`

```
#[no_mangle]
pub unsafe extern "C" fn current_elx_irq(_context: &mut ExceptionContext) {
        // super::disable_irqs();
    Timer::interupt_after(Timer::get_frequency());
    Timer::enable();
    super::enable_irqs();
    if is_scheduling.load(core::sync::atomic::Ordering::Relaxed) {
        return;
    }
    is_scheduling.store(true, core::sync::atomic::Ordering::Relaxed);
    scheduler::schedule();
    is_scheduling.store(false, core::sync::atomic::Ordering::Relaxed);
}
```

Najpierw ponownie ustawiamy zegar tak jak za pierwszym razem, włączamy obsługę przerwań i przechodzimy do wywołania schedulera pod warunkiem, że w zmiennej `is_scheduling` będącej atomową zmienną bool'owską jest wartość `false`. Dzięki temu zabraniamy zagnieżdżania się przerwań schedulera w sobie.

Rozdział 6

Scheduling time![1]

6.1 Struktura zadań systemu

W naszym systemie poszczególne zadania systemu operacyjnego reprezentowane są poprzez strukturę `TaskContext`

```
#[repr(C)]
#[derive(Debug)]
/// Structure containing context of a single task
pub struct TaskContext {
    /// General Purpose Registers
    pub(super) gpr: GPR,
    /// State of task
    pub(super) task_state: TaskStates,
    /// Counter meaning how many time quants has remained
    /// to this task during current cycle
    pub(super) counter: u32,
    /// Number of time quants given in one round
    pub(super) priority: u32,
    /// Currently unused
    preemption_count: u32,
    /// "Pointer" to kernel space task stack
    stack: Option<TaskStack>,
    /// "Pointer" to user space task stack
    user_stack: Option<TaskStack>,
    /// Is user task
    has_user_space: bool,
}
```

[1] Brought to you by scheduling gang

6.1.1 Pole gpr

Pole GPR jest to struktura przechowująca rejestry x19-x30 , sp oraz lr zdefiniowana następująco:

```
pub struct GPR {
    pub x19: [u64; 11],
    pub sp: u64,
    lr: u64,
}
```

6.1.2 Pole task_state

Pole task_state jest to enum określający aktualny stan zadania.

```
pub enum TaskStates {
    /// Task is created, by not started
    NotStarted = 0,
    /// Task is running and managed by scheduler
    Running = 1,
    /// Task is suspended and skipped by scheduler
    Suspended = 2,
    /// Task is dead and waiting to clean after it
    Dead = 3,
}
```

6.1.3 Pola counter i priority

Pola te zostaną omówione przy opisie algorytmu schedulera

6.1.4 Pola stack, user_stack

Są to struktury stosów dla zadań - zadanie z user space posiada oba wyżej wymienione stosy, natomiast zadanie z kernel space posiada tylko ten pierwszy.

6.1.5 Pole has_user_space

Pole określające przynależność zadania do user space (true) lub kernel space (false)

6.2 Struktura stosu zadania

6.2.1 Struktura `TaskStack`

```
pub struct TaskStack {
    ptr : *mut u8,
    size : usize,
}
```

Sama struktura reprezentująca stos jest bardzo prosta: składa się ona ze wskaźnika na jego początek oraz rozmiaru stosu. Bezpieczną alokacje, dealokacje stosu oraz informacje o podstawowych jego własnościach zapewniają poniższe metody:

6.2.2 Inicjalizacja stosu

Odbywa się ona poprzez wywołanie metody `new()`:

```
pub fn new(size : usize) -> Option<Self> {
    let layout = Layout::from_size_align(size, 16).ok()?;
    let ptr = unsafe { alloc_zeroed(layout) };
    if ptr.is_null() {
        return None;
    }
    Some(TaskStack {ptr,size})

}
```

Prześledźmy jej wykonanie:

- Podany rozmiar jest wyrównywany do 16 bitów
- Następuje próba zaalokowania odpowiedniej ilości ciągłej pamięci
- W zależności od powodzenia alokacji zwracane jest `Option` naszej struktury

6.2.3 Funkcje zwracające informacje o stosie

```
pub fn stack_base(&self) -> usize {
    self.ptr as usize + self.size - 16
}
pub fn stack_top(&self) -> usize {
    self.ptr as usize
}
pub fn size(&self) -> usize {
    self.size
}
```

Są to funkcje zwracające odpowiednio podstawę stosu, jego wierzchołek oraz rozmiar.

6.2.4 Dealokacja stosu

W celu poprawnej dealokacji stosu implementujemy na strukturze `TastStack` trait `Drop`:

```
impl Drop for TaskStack {
    fn drop(&mut self) {
        let layout = Layout::from_size_align(self.size, 16).unwrap();
        unsafe { dealloc(self.ptr, layout) };
    }
}
```

Tak jak przy alokacji wyrównujemy rozmiar do 16 bitów i dealokujemy obszar o tej długości począwszy od adresu na który wskazuje `ptr`.

6.3 Struktura schedulera oraz jego algorytm

6.3.1 Struktura Scheduler

```
pub struct Scheduler {
    tasks: Vec<TaskContext>,
    current_running_task: usize,
}
```

Ze strukturą schedulera skojarzone są dwa pola:

- vector wszystkich tasków rozpatrywanych przez scheduler
- indeks taska któremu aktualnie jest przydzielony procesor.

6.3.2 Zgłaszanie zadania do schedulera

Nowe zadanie zgłasza się do schedulera za pomocą metody `submit_task` zdefiniowanej następująco:

```
fn submit_task(&mut self, task_context: TaskContext) -> Result<(), TaskError> {
    if self.tasks.len() >= MAX_TASK_COUNT {
        return Err(TaskError::TaskLimitReached);
    }
    self.tasks.push(task_context);
    Ok(())
}
```

Przekazane zadanie jest dodawane do wektora wtedy, gdy działających zadań jest mniej niż limit.

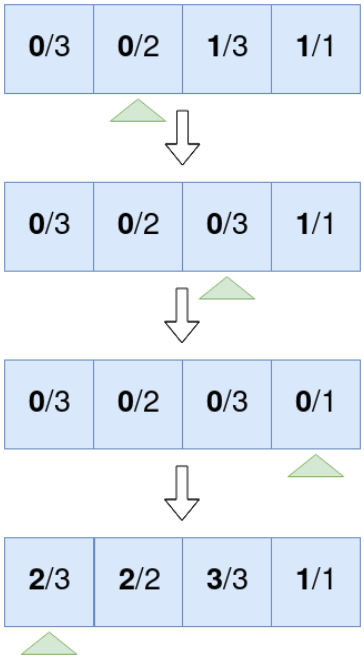
6.3.3 Algorytm schedulera

Cały algorytm schedulera polega na znalezieniu następnego zadania, które ma otrzymać procesor oraz zmianie kontekstu procesora na wybrane zadanie. Proces wyboru następnego zadania odbywa się w głównej pętli metody `schedule(&mut self)`:

```
let next_task_pid;
let tasks = &mut self.tasks;

'find_task: loop {
    for (i, task) in tasks.iter_mut().enumerate() {
        // get mutable reference for currently examined task

        match task.task_state {
            // if curr_task is in state that it
            //should be scheduled and task has unused quant of time
            TaskStates::Running | TaskStates::NotStarted if task.counter > 0 => {
                // decrease counter and mark it as next task
                task.counter -= 1;
                next_task_pid = i;
                break 'find_task;
            }
            // in other states ignore this task
            _ => {}
        }
    }
    for task in tasks.iter_mut() {
        task.counter = task.priority;
    }
}
self.change_task(next_task_pid)?;

Ok(next_task_pid)
```

W pętli przechodzimy po wszystkich zadaniach w tablicy `tasks`; jeśli dany task jest w stanie `TaskStates::Running` lub `TaskStates::NotStarted` oraz została mu niezerowa ilość kwantów czasu (pole `counter`) to to zadanie będzie następnym któremu przydzielony zostanie procesor. Zmniejszamy więc liczbe kwantów czasu o 1 i wychodzimy z pętli. W tym miejscu idealną jest przejrzysta składnia breaków z pętli w Rust. Umożliwia ona nazywanie poszczególnych pętli i określenie wprost poza którą wychodzimy.

W przypadku, gdy nie znaleziono żadnego takiego zadania scheduler przywraca każdemu z tasków count równy jego priorytetowi i ponawia przeszukiwanie.



Rysunek 6.1: Przykładowe następne kroki schedulera

6.3.4 Zmiana kontekstu procesora

Po wybraniu następnego zadania które ma uzyskać procesor, scheduler wykonuje metodę `self.change_task(next_task_pid)`

```
fn change_task(&mut self, next_task: usize) -> Result<(), TaskError> {
        let tasks = &mut self.tasks;

        if next_task >= tasks.len() {
            return Err(TaskError::InvalidTaskReference);
        }

        let prev_task_addr = &tasks[self.current_running_task]
            as *const TaskContext as u64;
        let next_task_addr = &tasks[next_task]
            as *const TaskContext as u64;

        self.current_running_task = next_task;

        unsafe {
            cpu_switch_to(prev_task_addr, next_task_addr);
        }

        Ok(())
    }
```

Funkcja ta wyciąga adresy w pamięci struktur `TaskContext` zadania poprzedniego oraz wybranego jako następny, aktualizuje zmienną `current_running_task` oraz wywołuje funckję `cpu_switch_to`. Jest

to funkcja napisana w assembly, która zapisuje kontekst procesora do struktury pod adresem `prev_task_addr` oraz ustawia kontekst procesora na ten zapisany w strukturze `next_task_addr`

6.3.5 Funkcja `cpu_switch_to`

```
  .globl cpu_switch_to
cpu_switch_to:
    mov    x8, x0
    mov    x9, sp
    stp    x19, x20, [x8], #16        // store callee-saved registers
    stp    x21, x22, [x8], #16
    stp    x23, x24, [x8], #16
    stp    x25, x26, [x8], #16
    stp    x27, x28, [x8], #16
    stp    x29, x9, [x8], #16
    str    x30, [x8]

    mov    x8, x1
    ldp    x19, x20, [x8], #16        // restore callee-saved registers
    ldp    x21, x22, [x8], #16
    ldp    x23, x24, [x8], #16
    ldp    x25, x26, [x8], #16
    ldp    x27, x28, [x8], #16
    ldp    x29, x9, [x8], #16
    ldr    x30, [x8]
    mov    sp, x9
    ret
```

Nasza funkcja składa się z dwóch zasadniczych części - zapisania rejestrów procesora do struktury, której adres przekazaliśmy w rejestrze `x0`, oraz wczytania do rejestrów wartości ze struktury pod adresem z `x1`. Stack pointer jest przechowywany w `x9` na czas wykonania funkcji.

6.4 Pierwsze uruchomienie schedulera

Przy inicjalizacji systemu nastąpić musi również inicjalizacja schedulera i pierwsza zmiana kontekstu procesora z kodu inicjalizującego na pierwszy task. Dzieje się tak poprzez wykonanie metody schedulera `start()`:

```
pub fn start(&mut self) -> Result<!, TaskError> {
      let tasks = &mut self.tasks;

      if tasks.len() == 0 {
          return Err(TaskError::ChangeTaskError);
      }

      let mut init_task = &mut tasks[0];
      init_task.counter = init_task.priority - 1;

      let init_task_addr = init_task as *mut TaskContext as u64;

      unsafe {
          cpu_switch_to_first(init_task_addr);
      }
  }
```

Następuje w niej sprawdzenie istnienia co najmniej jednego zadania, ręczne odjęcie jednego kwantu czasu od zerowego zadania oraz zmiana kontekstu procesora na nie. Dzieje się to za pomocą innej funkcji niż zamienianie kontekstu między zadaniami, ponieważ nie zależy nam na zachowaniu informacji o kontekście przed zmianą.

6.4.1 Funkcja `cpu_switch_to_first`

```
  .globl cpu_switch_to_first
cpu_switch_to_first:
    mov    x8, x0
    ldp    x19, x20, [x8], #16        // restore callee-saved registers
    ldp    x21, x22, [x8], #16
    ldp    x23, x24, [x8], #16
    ldp    x25, x26, [x8], #16
    ldp    x27, x28, [x8], #16
    ldp    x29, x9, [x8], #16
    ldr    x30, [x8]
    mov    sp, x9
    ret
```

Jest to okrojona wersja funkcji `cpu_switch_to`, która przyjmuje tylko jeden argument - adres struktury z której zczytać należy nowy kontekst.

Rozdział 7

Call me maybe[1]

7.1 Ogólny interfejs syscall'i

Stworzyliśmy zestaw uniwersalnych wrapperów do wywoływania przerwania do supervisora (instrukcji svc):

```
#[inline(never)]
pub unsafe fn syscall0(mut a: usize) -> usize {
    asm!("svc   0"
          : "={x0}"(a)
          : "{x8}"(a)
          : "x0", "x8"
          : "volatile");
    return a;
}

#[inline(never)]
pub unsafe fn syscall1(mut a: usize, b: usize) -> usize {
    asm!("svc   0"
          : "={x0}"(a)
          : "{x8}"(a), "{x0}"(b)
          : "x0", "x8"
          : "volatile");

    return a;
}
```

Wrappery tego typu są zdefiniowane analogicznie aż do `syscall5`. Poszczególne implementacje syscall'i są oparte na wywoływaniu odpowiednich wrapperów.

Koncepcją wrapperów jest przekazywanie rodzaju syscall'a w rejestrze `x8`.

[1] Here is my number: http://bitly.com/98K8eH

7.1.1 Typy syscall'i

Typ danego syscall'a określany jest przez ostatni argument wrappera. Jest on określany jako jedna z wartości enuma `Syscalls` rzutowana na `usize`:

```
#[repr(usize)]
#[derive(FromPrimitive, ToPrimitive, Debug)]
pub enum Syscalls {
    Print,
    NewTask,
}
```

7.1.2 Obsługa syscall'i

Syscall jest obsługiwany w handlerze przerwania synchronicznego:

```
#[no_mangle]
pub unsafe extern "C" fn lower_aarch64_synchronous(
    context: &mut ExceptionContext) -> () {
    // println!("{}",*context);
    match SynchronousCause::from_u64(context.esr_el1) {
        Some(_) => {
            // println!("{}",*context);
            let syscall_type: Option<Syscalls> = Syscalls::from_u64(context.gpr.x[8]);

            if syscall_type.is_none() {
                println!(
                    "[Task Fault] Unsupported Syscall number '{}' detected.",
                    context.gpr.x[8]
                );
                return;
            }
            let syscall_type = syscall_type.unwrap();
```

Najpierw następuje sprawdzenie wartości rejestru ESR_EL1, która mówi o przyczynie przerwania. Jeśli jest ona prawidłowa przystępujemy do rozpoznania typu syscall'a - w obu przypadkach w razie błędu zwracamy odpowiednie komunikaty. Następnie ma miejsce wywołanie odpowiedniego od typu syscall'a handlera:

```
match syscall_type {
            Syscalls::Print => handle_print_syscall(context),
            Syscalls::NewTask => handle_new_task_syscall(context),
}
```

W przypadku niedopasowania do żadnego handlera następuje wyświetlenie na ekranie podpiętym pod HDMI *THE TURQUOISE SCREEN OF ETERNAL DOOM*, wypisanie na UART stosownego komunikatu oraz wejść:

```
None => {
        let mut charbuffer = crate::framebuffer::charbuffer::CHARBUFFER.lock();
        if charbuffer.is_some() {
            let charbuffer = charbuffer.as_mut().unwrap();
            charbuffer.set_cursor((0,0));
            charbuffer.background = (0,0,180,255);
            charbuffer.puts("                                              *   * \n");
            charbuffer.puts(" THE TURQUOISE SCREEN OF ETERNAL DOOM!    |   \n");
            charbuffer.puts("                                            /\\/\\/\\/   \n");
            for i in 0..charbuffer.height - 10 {
                charbuffer.putc('\n');
            }
            charbuffer.update();

        }
        println!(
            "[Task Fault]\n\tReason: Unknown code '{:#018x}'
            \n\tProgram location:    '{:#018x}'
            \n\tAddress:             '{:#018x}'
            \n\tStack:               '{:#018x}\n",
            context.esr_el1,
            context.elr_el1,
            context.far_el1,
            context.sp_el0
        );
        loop {}
    }
```

7.2 Syscall `Print`

Syscall `print` wykorzystywany jest do obsługi czynności związanych z UARTem czy framebufferem.

7.2.1 Funkcja wołająca syscall

Do wywoływania syscalla `print` służy funkcja `write`:

```
pub fn write(msg: &str) {
    let bytes = msg.as_bytes();
    unsafe {
        syscall2(
            bytes.as_ptr() as usize,
            bytes.len(),
            Syscalls::Print as usize,
        );
    }
}
```

Jest to prosty wrapper na funkcję `syscall2` wydobywający wskaźnik na tekst z referencji na `str`.

Funkcja ta wykorzystywana jest w naszej implementacji makr typu `print!` o których więcej w rozdziale im dedykowanym.

7.2.2 Handler syscall'a

Syscall `Print` obsługiwany jest za pomocą funkcji `handle_print_syscall` wywoływanej z handlera przerwań synchronicznych.

```
fn handle_print_syscall(context: &mut ExceptionContext) {
    let ptr = context.gpr.x[0] as *const u8;
    let len = context.gpr.x[1] as usize;

    let data = unsafe { slice::from_raw_parts(ptr, len) };

    let string = from_utf8(data);
```

Na początku wyciągamy nasze argumenty przekazane od funkcji wołającej syscall i konwertujemy do właściwego typu danych

```
    if string.is_err() {
        println!(
            "[Syscall Fault (Write)] String provided doesen't apper to be correct
↪   UTF-8 string.
            \n\t -- Caused by: '{}'",
            string.err().unwrap()
        );
        return;
    }
    let string = string.unwrap();
```

Jeśli wiadomość zawierała znaki niezgodne ze standarem UTF-8 wyświetlamy stosowny komunikat i kończymy syscall'a.

```
    if charbuffer.is_some() { charbuffer.as_mut().unwrap().puts(string); }
    else{ println!("{}", string); }
```

W zależności od tego, czy charbuffer został zainicjalizowany wyświetlamy to albo na charbufferze, albo przekazujemy do makra `print!` w celu wypisania na UART.

7.3 Syscall NewTask

Syscall `new_task` wykorzystywany jest do tworzenia nowego zadania systemu operacyjnego.

7.3.1 Funkcja wołająca syscall

Do wywoływania syscalla `new_task` służy funkcja `new_task`:

```
pub fn new_task_syscall(start_function: extern "C" fn(), priority_difference: usize) {
    let function_ptr = start_function as usize;

    unsafe {
        syscall2(
            function_ptr,
            priority_difference as usize,
            Syscalls::NewTask as usize,
        );
    }
}
```

Jest to prosty wrapper na funkcję `syscall2` wydobywający wskaźnik na funkcję oraz przekazujący go do niej wraz z różnicą priorytetów (nie można stworzyć zadania z priorytetem większym niż obecne).

7.3.2 Handler syscall'a

Syscall `print` obsługiwany jest za pomocą funkcji `handle_print_syscall` wywoływanej z handlera przerwań synchronicznych.

```
fn handle_new_task_syscall(context: &mut ExceptionContext) {
    let start_function = unsafe { &*(&context.gpr.x[0] as *const u64 as *const
↪   extern "C" fn()) };
    crate::println!("KERNEL ADDRESS {:#018x}", *start_function as u64);

    let priority_difference = context.gpr.x[1] as u32;

    let curr_priority = 1;
    let new_priority = if curr_priority > priority_difference {
        curr_priority - priority_difference
```

```
    } else {
        1
    };
```

Na początku konwertujemy otrzymane parametry na poprawne typy danych i obliczamy nowy priorytet zadania.

```
    let task = scheduler::TaskContext::new(*start_function, new_priority,
↪   true).unwrap();

    match task.start_task() {
        Ok(_) => {},
        Err(e) => {
            println!(
            "[Syscall Fault (New Task)] System was unable to create new task.
            \n\t -- Caused by: '{:?}'",
            e);
        },
    }
}
```

Następnie tworzymy nowe zadanie systemu operacyjnego i obsługujemy ewentualne błędy.

7.4 Syscall TerminateTask

7.4.1 Funkcja wołająca syscall

```
    pub fn terminate_user_task(return_value: usize) -> ! {
    unsafe {
        syscall1(return_value, Syscalls::TerminateTask as usize);
    }
    loop {}
}
```

Jest to prosty wrapper na funkcję `syscall1` przekazujący wartość zwracaną przez proces.

Automatyczne wywołanie syscall'a przy zakończeniu działania zadania poziomu użytkownika. Jest ono uzyskiwane dzięki opokowaniu wywoływanej funkcji tworzącej user task w następujący sposób:

```
pub extern "C" fn user_task(task: extern "C" fn() -> usize) -> ! {
    let return_value = task();
    crate::userspace::syscall::terminate_user_task(return_value);
}
```

7.4.2 Handler syscall'a

Handler syscalla wywołuje funkcję `scheduler::end_task(scheduler::get_current_task_id())` w celu zakończenie wykonywania zadania oraz wywołuje scheduler.

7.5 Syscall GetTime

7.5.1 Funkcja wołająca syscall

```
pub fn get_time() -> u64 {
    unsafe {
        return syscall0(Syscalls::GetTime as usize) as u64;
    }
}
```

Jest to prosty wrapper na funkcję `syscall0`.

7.5.2 Handler syscall'a

```
fn handle_get_time_syscall(context: &mut ExceptionContext) {
    context.gpr.x[0] = timer::ArmQemuTimer::get_time();
}
```

Handler syscalla wywołuje funkcję `timer::ArmQemuTimer::get_time()` w celu pobrania aktualnego czasu.

7.6 Syscall Yield

7.6.1 Funkcja wołająca syscall

```
pub fn yield_cpu() -> u64 {
    unsafe {
        return syscall0(Syscalls::Yield as usize) as u64;
    }
}
```

Jest to prosty wrapper na funkcję `syscall0`.

7.6.2 Handler syscall'a

```
fn handle_yield_syscall(context: &mut ExceptionContext) {
    is_scheduling.store(true, core::sync::atomic::Ordering::SeqCst);
    scheduler::schedule();
    is_scheduling.store(false, core::sync::atomic::Ordering::SeqCst);
}
```

Handler syscalla wywołuje funkcję `schedule()` rezygnując z pozostałej przydzielonej części czasu procesora.

Rozdział 8

MMUUUUUUUUUUUUUUUUUUUU!

8.1 Memeory Management Unit

Architektura ARMv8-A w wersji 64 bit-owej **AArch64** wymaga od procesora posiadania osobnego układu kontroli pamięci wirtualnej.

Dzięki temu implementacja pamięci wirtualnej w UranOS-ie jest niezależna od konkretnego modelu mikrokomputera.

Zdecydowaliśmy się na granulacje pamięci o wielkosci 64kB gdyż zmniejsza to rozmiary pamieci wykorzytywanej do translacji.

8.1.1 Translation Table Descriptor

```
// A table descriptor, as per AArch64 Reference Manual Figure D4-15.
register_bitfields! {u64,
    STAGE1_TABLE_DESCRIPTOR [
        /// Physical address of the next page table.
        NEXT_LEVEL_TABLE_ADDR_64KiB OFFSET(16) NUMBITS(32) [], // [47:16]

        TYPE  OFFSET(1) NUMBITS(1) [
            Block = 0,
            Table = 1
        ],

        VALID OFFSET(0) NUMBITS(1) [
            False = 0,
            True = 1
        ]
    ]
}
```

8.1.2 Page Descriptor

Jako że Deskryptor strony jest bardziej skompilowany analiza jest podzielona na części.

```
// A level 3 page descriptor, as per AArch64 Reference Manual Figure D4-17.
register_bitfields! {u64,
    STAGE1_PAGE_DESCRIPTOR [
```

Bit określający czy dane w pamięci mogą być używane jako instrukcje w trybie EL0

```
        /// User execute-never.
        UXN      OFFSET(54) NUMBITS(1) [
            False = 0,
            True = 1
        ],
```

Bit określający czy dane w pamięci mogą być używane jako instrukcje w trybie EL1

```
        /// Privileged execute-never.
        PXN      OFFSET(53) NUMBITS(1) [
            False = 0,
            True = 1
        ],
```

2 Bajty określające najmniej znaczące bity adresu strony.

```
        /// Physical address of the page (lvl3).
        OUTPUT_ADDR_64KiB OFFSET(16) NUMBITS(32) [], // [47:16]
```

Bit określający czy dana strona zostala juz użyta, w Linuxie wykorzystywane do alokacji.

```
        /// Access flag.
        AF       OFFSET(10) NUMBITS(1) [
            False = 0,
            True = 1
        ],
```

2 Bity określające czy dana strona jest współdzielona z jakimś innym urządzeniem niż CPU.

```
        /// Shareability field.
        SH       OFFSET(8) NUMBITS(2) [
            OuterShareable = 0b10,
            InnerShareable = 0b11
        ],
```

2 Bity określający możliwości zapisu i odczytu do strony w zależności czy jesteśmy w trybie `EL0` lub `EL1`. `AP::RW_EL1_EL0` i `PXN::True` są wzajemnie wykluczające, w wypadku ich wzajemnego ustawienia efektywnie `PXN == PXN::False`.

```
        /// Access Permissions.
        AP       OFFSET(6) NUMBITS(2) [
            RW_EL1 = 0b00,
            RW_EL1_EL0 = 0b01,
            RO_EL1 = 0b10,
            RO_EL1_EL0 = 0b11
        ],
```

3 Bity określające typ strony z pośród 8 (programistyczne) zdefiniowanych archetypów pamięci.

```
        /// Memory attributes index into the MAIR_EL1 register.
        AttrIndx OFFSET(2) NUMBITS(3) [],
```

1 Bit który jest wykorzystywane przez MMU do określenia czy adres został już przetłumaczony czy nalezy kontynuawać translacje.

```
        TYPE     OFFSET(1) NUMBITS(1) [
            Block = 0,
            Table = 1
        ],
```

1 Bit którego ustawienie na `False` spowoduje powstanie przerwania synchronicznego `MMU Fault`. (Domyślne zachowanie dla nieprawidłowych adresów)

```
        VALID    OFFSET(0) NUMBITS(1) [
            False = 0,
            True = 1
        ]
    ]
}
```

8.2 Abstakcja

Jakkolwiek dobrze pola bitowe nie były by opisane, to stworzenie osobnych typów dla atrybutów pamięci pozwala myśleć o niej w przystępniejszy sposób.
Translation określa czy strona powinna zostać przetłumaczona 1:1 lub otrzymac konkretny adres.

```
#[derive(Copy, Clone)]
pub enum Translation {
    Identity,
    Offset(usize),
}
```

Atrybut określający czy mamy doczynienia z pamięcią RAM czy MMIO.

```
#[derive(Copy, Clone)]
pub enum MemAttributes {
    CacheableDRAM,
    Device,
}
```

Atrybut sposób dostępu do pamięci.

```
#[derive(Copy, Clone)]
pub enum AccessPermissions {
    ReadOnly,
    ReadWrite,
}
```

Oprócz powyższych atrybutów dodajemy bool-a informującego o możliwości traktowania jako kodu.

```
#[derive(Copy, Clone)]
pub struct AttributeFields {
    pub mem_attributes: MemAttributes,
    pub acc_perms: AccessPermissions,
    pub execute_never: bool,
}
```

Domyślnym typem pamięci jest "Read Write Non-executable RAM".

```
impl Default for AttributeFields {
    fn default() -> AttributeFields {
        AttributeFields {
            mem_attributes: MemAttributes::CacheableDRAM,
            acc_perms: AccessPermissions::ReadWrite,
            execute_never: false,
        }
    }
}
```

Sprawdźmy czy procesor umożliwia użycie Stron 64kB.

```
if !ID_AA64MMFR0_EL1.matches_all(ID_AA64MMFR0_EL1::TGran64::Supported) {
    crate::println!("64 KiB translation granule not supported");
    aarch64::halt();
}
```

Przygotuj archetypy pamięci.

```
memory::setup_mair();
```

Przygotuj tabele translacji.

```
memory::setup_transaltion_tables();
```

Wystaw adress tabel translacji dla kontrolera.

```
// Set the "Translation Table Base Register".
TTBR0_EL1.set_baddr(memory::get_translation_table_address());
```

Skonfiguruj przebieg translacji.

```
memory::configure_translation_control();
```

Manulanie wymuś wykonanie wszystkich zapisów do pamięci.

```
// Switch the MMU on.
//
// First, force all previous changes to be seen before the MMU is enabled.
barrier::isb(barrier::SY);
```

Włącz MMU i Cache

```
// Enable the MMU and turn on data and instruction caching.
SCTLR_EL1.modify(SCTLR_EL1::M::Enable + SCTLR_EL1::C::Cacheable +
↪ SCTLR_EL1::I::Cacheable);
```

Ponownie wymuś wykonanie wszystkich zapisów do pamięci.

```
// Force MMU init to complete before next instruction.
barrier::isb(barrier::SY);
```

Na tym etapie MMU jest włączone i zkonfigurowane, program może kontynuować.

8.2.1 fn memory::setup_mair()

```
pub unsafe fn setup_mair() {
    MAIR_EL1.write(

↪   MAIR_EL1::Attr1_HIGH::Memory_OuterWriteBack_NonTransient_ReadAlloc_WriteAlloc
            +
↪   MAIR_EL1::Attr1_LOW_MEMORY::InnerWriteBack_NonTransient_ReadAlloc_WriteAlloc
            + MAIR_EL1::Attr0_HIGH::Device
            + MAIR_EL1::Attr0_LOW_DEVICE::Device_nGnRE,
    );
}
```

8.2.2 fn memory::setup_transaltion_tables()

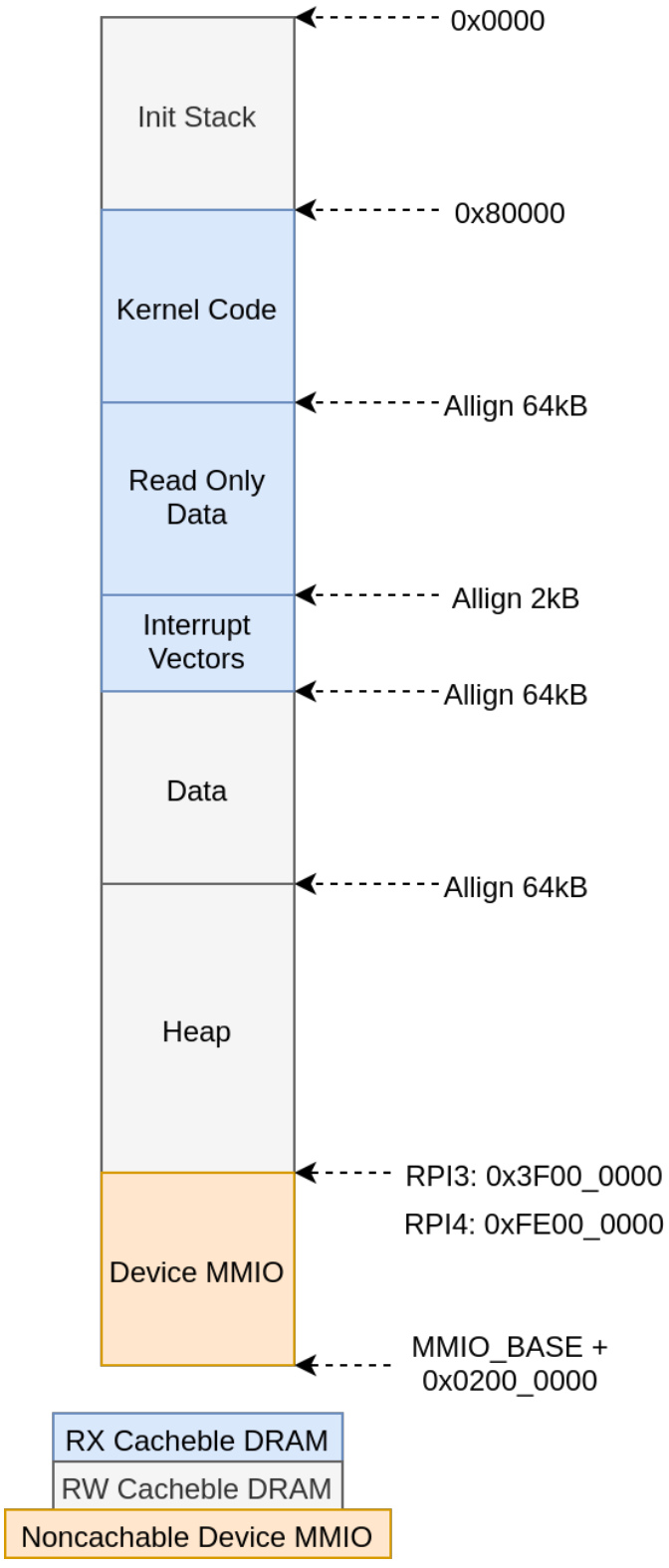
```
pub unsafe fn setup_transaltion_tables() -> Result<(), &'static str> {
    let mut tables = &mut TRANSLATION_TABLES;
    layout::LAYOUT.print_layout();
    for (l2_nr, l2_entry) in tables.level_2.iter_mut().enumerate() {
        *l2_entry = tables.level_3[l2_nr].base_addr().into();
        for (l3_nr, l3_entry) in tables.level_3[l2_nr].iter_mut().enumerate() {
            let virt_addr = (l2_nr << LOG_512_MIB) + (l3_nr << LOG_64_KIB);

            let (output_addr, attribute_fields) =
                layout::LAYOUT.get_virt_addr_properties(virt_addr)?;
            *l3_entry = PageDescriptor::new(output_addr, attribute_fields);
        }
    }
    Ok(())
}
```

8.2.3 fn memory::configure_translation_control()

```
/// Configure various settings of stage 1 of the EL1 translation regime.
pub unsafe fn configure_translation_control() {
    let ips = ID_AA64MMFR0_EL1.read(ID_AA64MMFR0_EL1::PARange);
    TCR_EL1.write(
        TCR_EL1::TBI0::Ignored
            + TCR_EL1::IPS.val(ips)
            + TCR_EL1::TG0::KiB_64
            + TCR_EL1::SH0::Inner
            + TCR_EL1::ORGN0::WriteBack_ReadAlloc_WriteAlloc_Cacheable
            + TCR_EL1::IRGN0::WriteBack_ReadAlloc_WriteAlloc_Cacheable
            + TCR_EL1::EPD0::EnableTTBR0Walks
            + TCR_EL1::T0SZ.val(32), // TTBR0 spans 4 GiB total.
    );
}
```

8.3 Mapa Pamięci

Jako demonstracje działania MMU wykorzystujemy, używamy różnych atrybutów pamięci w celu zabezpieczenia przed nadpisaniem danych tylko do odczytu czy kodu wykonywalnego. .



Rysunek 8.1: Mapa pamięci wirtualnej

Rozdział 9

"Don't question the *Concurent One*"

9.1 Rust i współbieżność

Każdy typ w Rust może być używany w kontekście jednego wątku. Jednak aby uzywac go w kontekście w którym może być dostępny z wielu wątków aby kompilator nas nie okrzyczał musi on implementowac `trait Sync` oraz `trait Send`

"Safe Rust" jako jezyk zabrania powstawania "undefined behaviour" (do ktorych zaliczają sie data race-y) są to to `trait`-y `unsafe`.

```
unsafe impl Send for T {}
unsafe impl Sync for T {}
```

9.2 Mutex

```
pub struct Mutex<T> {
    lock: AtomicBool,
    data: UnsafeCell<T>,
}

unsafe impl<T : Sync> Sync for Mutex<T> {}
unsafe impl<T : Send> Send for Mutex<T> {}
```

9.2.1 API Mutex-a

Wprowadzone w C++ pojęcie RAII czyli zasobów zwalnianych przez destruktor na koncu życia obiektów jest podstawą funkcji lock. Funkcja ta zwraca MutexGuard który automatycznie zwolni blokadę na mutexie

```
pub fn lock(&self) -> MutexGuard<T> {
        self.take_lock();
        MutexGuard {
            lock: &self.lock,
```

```
            data: &mut *self.data.get(),
        }
    }
```

Preferowanym sposobem jest jednak `fn sunc<F, R>` która przyjmuje funkcje anonimową, co pozwala na dokładniejszą kontrole czasu zwolnienia mutex-a.

```
pub fn sync<F, R>(&self, f: F) -> R
    where
        F: FnOnce(&mut T) -> R,
    {
        self.take_lock();

        let result = f(unsafe { &mut *self.data.get() });

        self.lock.store(false, Ordering::Release);
        result
    }
```

9.3 NullLock

Jest strukturą która udaje API `Mutex`-a ale nim nie jest, tzn. nie blokuje dostępu do danych. Moze być bezpiecznie użyty tylko w kontekście niewspółbieżnym.

9.4 `userspace::Mutex`

Mutex dla procesów użytkownika potrafi w momencie gdy zostaje zajęty, oddać czas procesor-a innemu procesorowi.
Róznice względem pomiędzy `Mutex`-em kernel-a (Spinlockiem) a `Mutex`-em użytkownika występują w funkcji `fn take_lock(&self)`

9.4.1 spinlock

```
asm::nop();
```

9.4.2 waitlock

```
super::syscall::yield_cpu();
```

Rozdział 10

Tworzenie własnych zadań w systemie UranOS

10.1 Hello world

Uruchamianie własnych zadań w systemie UranOS jest proste i szybkie. Całość naszego kodu będzie obracać się wokół pliku `src/init.rs`, czyli w miejscu, gdzie zdefiniowane jest główne zadanie naszego systemu. Na początku nasz kod zawieramy w funkcji, którą chcemy, aby wykonywało pojedyncze zadanie. W naszym przykładzie będzie to funkcja `hello_world`.

```
#[no_mangle]
pub extern "C" fn hello_world() {
    crate::uprintln!("Hello world!");
}
```

Jak zostało wspomniane w sekcji opisu syscall'i nie musimy kończyć zadania syscall'em `terminate_user_task`, gdyż jest on dla zadań użytkownika wywoływany automatycznie.

Po stworzeniu naszej funkcji musimy powiedzieć systemowi operacyjnemu aby utworzył nowe zadanie wykonujące nasz kod. Robimy to w funkcji `init`:

```
pub extern "C" fn init() {
    crate::userspace::syscall::new_task(hello_world, 0);
}
```

Tak skonfigurowany system uruchomi nam nasze pojedyncze zadanie.

10.2 Problem pięciu filozofów

W celu pokazania możliwości naszego systemu zaimplementowaliśmy przykład problemu pięciu filozofów.

Całość kodu naszego programu zawiera się w pliku `src/init.rs`

10.2.1 Zmienne statyczne

Na początku tworzymy statyczne struktury nazw, ID i widelców reprezentowanych jako mutexy:

```
static NAMES : [&'static str; 5] = [
    "Platon",
    "Aristoteles",
    "Konfucjusz",
    "Nitze",
    "Schopenhauer"
];
use core::sync::atomic::*;
static IDs : [AtomicBool; 5] = [
    AtomicBool::new(false),
    AtomicBool::new(false),
    AtomicBool::new(false),
    AtomicBool::new(false),
    AtomicBool::new(false)
];
type Fork = Mutex<()>;
static forks : [Fork; 5] = [
    Fork::new(()),
    Fork::new(()),
    Fork::new(()),
    Fork::new(()),
    Fork::new(())
];
```

10.2.2 Funkcja filozofa

Następnie definiujemy funkcję taska filozofa:

```
#[no_mangle]
pub extern "C" fn philisopher() {

    let my_id = || {
        loop {
            for (i, atomic) in IDs.iter().enumerate() {
                if let Ok(_) = atomic.compare_exchange(false, true,
↪   Ordering::SeqCst, Ordering::SeqCst){
                    return i;
                }
            }
        }
    };
    let left_fork = my_id;
    let rigth_fork = (left_fork + 1) % 5;
    crate::uprintln!("I'm {} and I have forks {} and {}", NAMES[my_id], left_fork,
↪   rigth_fork);
    let mut counter = 0;
    let mut times : u64 = 0;
    unsafe { loop {
        let start = crate::userspace::syscall::get_time();
        let (first, second) = if my_id % 2 == 0 {
            (forks[left_fork].lock(), forks[rigth_fork].lock())
        }
        else {
            (forks[rigth_fork].lock(), forks[left_fork].lock())
        };
        let end = crate::userspace::syscall::get_time();
        times += end - start;
        if counter > 10_000 {
            crate::uprintln!("[{}] Avg time {} us", NAMES[my_id], times  * 100 /
↪   (crate::userspace::syscall::get_frequency() as u64));

            counter = 0;
            times = 0;
        }
        else {
            counter += 1;
        }

        drop(first);
        drop(second);
    }}
```

```
}
```

Jak widzimy kod jest bardzo naturalny; nasze API do mutexów jest identyczne z API biblioteki standardowej, a makra nad obsługą UARTa pozwalają korzystać z pełni możliwości dobrze znanego makra `println`