МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій

Кафедра комп'ютерної інженерії та інформаційних систем

ЛАБОРАТОРНА РОБОТА №4

з дисципліни «Теорія і технології проектування спеціалізованих операційних систем»

ЛРКІ. 180102.22.01.02 ПЗ

Галузь знань \_\_\_\_\_\_\_\_\_\_12 – Інформаційні технології\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Спеціальність \_\_\_\_\_\_\_\_\_123 –Комп’ютерна інженерія\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Виконав:

студент 1 курсу, групи КІ2м-22-1 Біньковський Я.В.

(Підпис)

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лисенко С.М.

(Підпис)

Хмельницький – 2023 рік

**Тема:**

Проєктування спеціалізованої RTOS для Інтернету речей на платформі Raspberry Pi. Реалізація функцій обробки виняткових ситуацій. Розмежування адресного простору ядра і користувача. Реалізація функцій обробки виняткових ситуацій. Комплексне тестування на реальній платформі

**Завдання:**

Реалізувати етапи 10-16 проєктування спеціалізованої RTOS для платформи Raspberry Pi3. Протестувати поточну версію ядра та зробити відповідні висновки. Пояснити відмінності реалізації для платформи Raspberry Pi4. Оформити звіт з виконаної роботи.

**Виконання завдання:**

Етап 10.

На наступному етапі, використовуючи загальний підхід, відбувається налаштування системи трансляції пам'яті (MMU). Важливі кроки на цьому етапі включають:

1. Визначення архітектурно-незалежних типів дескрипторів для створення високорівневої структури даних, яка описує віртуальну пам'ять ядра. Ці типи дозволяють описувати атрибути різних областей пам'яті, такі як доступ на читання/запис, виконання, кешування тощо. Дескриптори не залежать від фактичних дескрипторів апаратного MMU.
2. У частині BSP (Board Support Package), конкретно у файлі src/bsp/raspberrypi/memory/mmu.rs, створюється статичний екземпляр KernelVirtualLayout, який містить описані вище дескриптори. Цей екземпляр робиться доступним через функцію bsp::memory::mmu::virt\_mem\_layout().
3. У частині aarch64, конкретно у файлі src/\_arch/aarch64/memory/mmu.rs, знаходиться фактичний драйвер MMU для архітектури AArch64. Цей драйвер отримує високорівневий KernelVirtualLayout з BSP і відображає його за допомогою гранул розміром 64 КБ.
4. У файлі memory/mmu.rs загального коду ядра, екземпляр KernelVirtualLayout з BSP використовується для створення останнього екземпляра ядерних таблиць трансляції пам'яті. Ці таблиці заповнюються шляхом виклику функції KERNEL\_TABLES.populate\_tt\_entries(), яка використовує інформацію про віртуальні адреси, отриману з virt\_mem\_layout(), і перетворює загальні дескриптори ядра в фактичні 64-бітні записи, необхідні для апаратного забезпечення AArch64 MMU.
5. Налаштування регістра MAIR\_EL1

Код реалізації:

Файл Cargo.toml

[package]

name = "mingo"

version = "0.10.0"

authors = ["Andre Richter <andre.o.richter@gmail.com>"]

edition = "2021"

[profile.release]

lto = true

[features]

default = []

bsp\_rpi3 = ["tock-registers"]

bsp\_rpi4 = ["tock-registers"]

[[bin]]

name = "kernel"

path = "src/main.rs"

[dependencies]

tock-registers = { version = "0.8.x", default-features = false, features = ["register\_types"], optional = true }

[target.'cfg(target\_arch = "aarch64")'.dependencies]

aarch64-cpu = { version = "9.x.x" }

Файл translation\_table.rs

#[cfg(target\_arch = "aarch64")]

#[path = "../../\_arch/aarch64/memory/mmu/translation\_table.rs"]

mod arch\_translation\_table;

pub use arch\_translation\_table::KernelTranslationTable;

Файл mmu.rs

#[cfg(target\_arch = "aarch64")]

#[path = "../\_arch/aarch64/memory/mmu.rs"]

mod arch\_mmu;

mod translation\_table;

use crate::common;

use core::{fmt, ops::RangeInclusive};

pub use arch\_mmu::mmu;

#[allow(missing\_docs)]

#[derive(Debug)]

pub enum MMUEnableError {

AlreadyEnabled,

Other(&'static str),

}

pub mod interface {

use super::\*;

pub trait MMU {

unsafe fn enable\_mmu\_and\_caching(&self) -> Result<(), MMUEnableError>;

fn is\_enabled(&self) -> bool;

}

}

pub struct TranslationGranule<const GRANULE\_SIZE: usize>;

pub struct AddressSpace<const AS\_SIZE: usize>;

#[allow(missing\_docs)]

#[allow(dead\_code)]

#[derive(Copy, Clone)]

pub enum Translation {

Identity,

Offset(usize),

}

#[allow(missing\_docs)]

#[derive(Copy, Clone)]

pub enum MemAttributes {

CacheableDRAM,

Device,

}

#[allow(missing\_docs)]

#[derive(Copy, Clone)]

pub enum AccessPermissions {

ReadOnly,

ReadWrite,

}

#[allow(missing\_docs)]

#[derive(Copy, Clone)]

pub struct AttributeFields {

pub mem\_attributes: MemAttributes,

pub acc\_perms: AccessPermissions,

pub execute\_never: bool,

}

#[allow(missing\_docs)]

pub struct TranslationDescriptor {

pub name: &'static str,

pub virtual\_range: fn() -> RangeInclusive<usize>,

pub physical\_range\_translation: Translation,

pub attribute\_fields: AttributeFields,

}

pub struct KernelVirtualLayout<const NUM\_SPECIAL\_RANGES: usize> {

max\_virt\_addr\_inclusive: usize,

inner: [TranslationDescriptor; NUM\_SPECIAL\_RANGES],

}

impl fmt::Display for MMUEnableError {

fn fmt(&self, f: &mut fmt::Formatter<'\_>) -> fmt::Result {

match self {

MMUEnableError::AlreadyEnabled => write!(f, "MMU is already enabled"),

MMUEnableError::Other(x) => write!(f, "{}", x),

}

}

}

impl<const GRANULE\_SIZE: usize> TranslationGranule<GRANULE\_SIZE> {

pub const SIZE: usize = Self::size\_checked();

pub const SHIFT: usize = Self::SIZE.trailing\_zeros() as usize;

const fn size\_checked() -> usize {

assert!(GRANULE\_SIZE.is\_power\_of\_two());

GRANULE\_SIZE

}

}

impl<const AS\_SIZE: usize> AddressSpace<AS\_SIZE> {

pub const SIZE: usize = Self::size\_checked();

pub const SIZE\_SHIFT: usize = Self::SIZE.trailing\_zeros() as usize;

const fn size\_checked() -> usize {

assert!(AS\_SIZE.is\_power\_of\_two());

Self::arch\_address\_space\_size\_sanity\_checks();

AS\_SIZE

}

}

impl Default for AttributeFields {

fn default() -> AttributeFields {

AttributeFields {

mem\_attributes: MemAttributes::CacheableDRAM,

acc\_perms: AccessPermissions::ReadWrite,

execute\_never: true,

}

}

}

impl fmt::Display for TranslationDescriptor {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

let start = \*(self.virtual\_range)().start();

let end = \*(self.virtual\_range)().end();

let size = end - start + 1;

let (size, unit) = common::size\_human\_readable\_ceil(size);

let attr = match self.attribute\_fields.mem\_attributes {

MemAttributes::CacheableDRAM => "C",

MemAttributes::Device => "Dev",

};

let acc\_p = match self.attribute\_fields.acc\_perms {

AccessPermissions::ReadOnly => "RO",

AccessPermissions::ReadWrite => "RW",

};

let xn = if self.attribute\_fields.execute\_never {

"PXN"

} else {

"PX"

};

write!(

f,

" {:#010x} - {:#010x} | {: >3} {} | {: <3} {} {: <3} | {}",

start, end, size, unit, attr, acc\_p, xn, self.name

)

}

}

impl<const NUM\_SPECIAL\_RANGES: usize> KernelVirtualLayout<{ NUM\_SPECIAL\_RANGES }> {

pub const fn new(max: usize, layout: [TranslationDescriptor; NUM\_SPECIAL\_RANGES]) -> Self {

Self {

max\_virt\_addr\_inclusive: max,

inner: layout,

}

}

pub fn virt\_addr\_properties(

&self,

virt\_addr: usize,

) -> Result<(usize, AttributeFields), &'static str> {

if virt\_addr > self.max\_virt\_addr\_inclusive {

return Err("Address out of range");

}

for i in self.inner.iter() {

if (i.virtual\_range)().contains(&virt\_addr) {

let output\_addr = match i.physical\_range\_translation {

Translation::Identity => virt\_addr,

Translation::Offset(a) => a + (virt\_addr - (i.virtual\_range)().start()),

};

return Ok((output\_addr, i.attribute\_fields));

}

}

Ok((virt\_addr, AttributeFields::default()))

}

pub fn print\_layout(&self) {

use crate::info;

for i in self.inner.iter() {

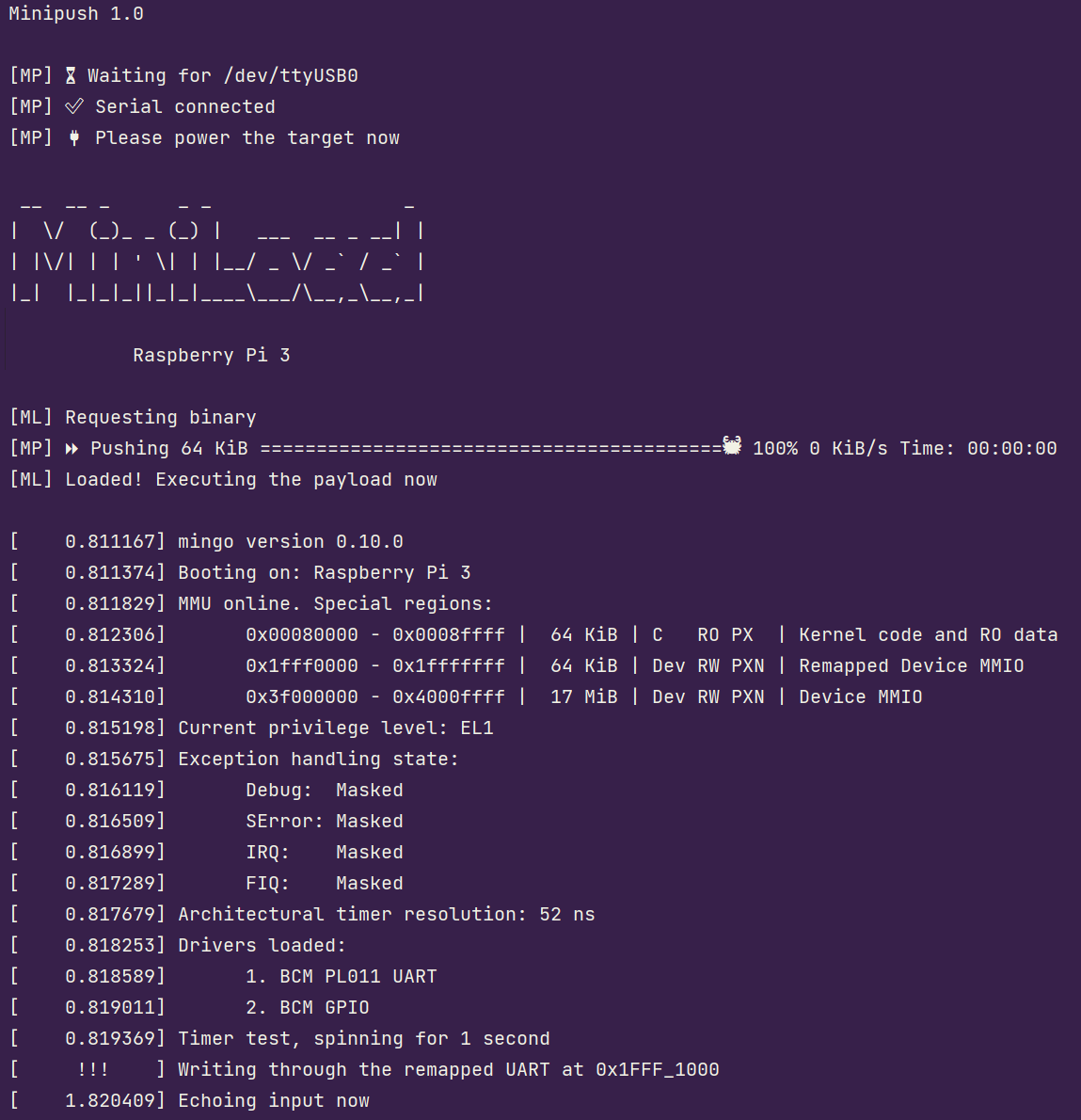
info!("{}", i);

}

}

}

Результат:



Етап 11

В даному етапі розглядається архітектура AArch64 (64-бітна архітектура ARM) і процес обробки винятків на цій архітектурі. Розглядаються чотири типи винятків в AArch64: синхронний, запит на переривання (IRQ), запит швидкого переривання (FIQ) і системна помилка (SError).

Під час виникнення винятку процесор переходить на той самий або вищий рівень винятків, але ніколи на нижчий рівень виконання (EL). Статус програми зберігається в регістрі SPSR\_ELx, а бажана адреса повернення зберігається в регістрі ELR\_ELx. Вибір покажчика стека для обробки винятку залежить від рівня винятків, на якому відбувається обробка.

AArch64 має 16 векторів винятків, кожний зі своїм зміщенням. Базова адреса векторів винятків зберігається в регістрі VBAR\_EL1. Код обробника винятку зазвичай зберігає контекст виконання, включаючи загальні цільові регістри, на стеку перед виконанням коду обробника. Після завершення обробки винятку контекст відновлюється, щоб процесор міг продовжити з місця, де він зупинився.

У реалізації системного програмного забезпечення на архітектурі AArch64 часто використовують поєднання мови програмування Rust і мови асемблера. Збереження та відновлення контексту під час обробки винятків виконується за допомогою макросів і коду на мові асемблера, оскільки це є одним з місць, де використання асемблера є необхідним.

Файл exception.rs

use aarch64\_cpu::{asm::barrier, registers::\*};

use core::{arch::global\_asm, cell::UnsafeCell, fmt};

use tock\_registers::{

interfaces::{Readable, Writeable},

registers::InMemoryRegister,

};

global\_asm!(include\_str!("exception.s"));

#[repr(transparent)]

struct SpsrEL1(InMemoryRegister<u64, SPSR\_EL1::Register>);

struct EsrEL1(InMemoryRegister<u64, ESR\_EL1::Register>);

#[repr(C)]

struct ExceptionContext {

gpr: [u64; 30],

lr: u64,

elr\_el1: u64,

spsr\_el1: SpsrEL1,

esr\_el1: EsrEL1,

}

fn default\_exception\_handler(exc: &ExceptionContext) {

panic!(

"CPU Exception!\n\n\

{}",

exc

);

}

#[no\_mangle]

extern "C" fn current\_el0\_synchronous(\_e: &mut ExceptionContext) {

panic!("Should not be here. Use of SP\_EL0 in EL1 is not supported.")

}

#[no\_mangle]

extern "C" fn current\_el0\_irq(\_e: &mut ExceptionContext) {

panic!("Should not be here. Use of SP\_EL0 in EL1 is not supported.")

}

#[no\_mangle]

extern "C" fn current\_el0\_serror(\_e: &mut ExceptionContext) {

panic!("Should not be here. Use of SP\_EL0 in EL1 is not supported.")

}

#[no\_mangle]

extern "C" fn current\_elx\_synchronous(e: &mut ExceptionContext) {

if e.fault\_address\_valid() {

let far\_el1 = FAR\_EL1.get();

if far\_el1 == 8 \* 1024 \* 1024 \* 1024 {

e.elr\_el1 += 4;

return;

}

}

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn current\_elx\_irq(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn current\_elx\_serror(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch64\_synchronous(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch64\_irq(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch64\_serror(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch32\_synchronous(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch32\_irq(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[no\_mangle]

extern "C" fn lower\_aarch32\_serror(e: &mut ExceptionContext) {

default\_exception\_handler(e);

}

#[rustfmt::skip]

impl fmt::Display for SpsrEL1 {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

writeln!(f, "SPSR\_EL1: {:#010x}", self.0.get())?;

let to\_flag\_str = |x| -> \_ {

if x { "Set" } else { "Not set" }

};

writeln!(f, " Flags:")?;

writeln!(f, " Negative (N): {}", to\_flag\_str(self.0.is\_set(SPSR\_EL1::N)))?;

writeln!(f, " Zero (Z): {}", to\_flag\_str(self.0.is\_set(SPSR\_EL1::Z)))?;

writeln!(f, " Carry (C): {}", to\_flag\_str(self.0.is\_set(SPSR\_EL1::C)))?;

writeln!(f, " Overflow (V): {}", to\_flag\_str(self.0.is\_set(SPSR\_EL1::V)))?;

let to\_mask\_str = |x| -> \_ {

if x { "Masked" } else { "Unmasked" }

writeln!(f, " Exception handling state:")?;

writeln!(f, " Debug (D): {}", to\_mask\_str(self.0.is\_set(SPSR\_EL1::D)))?;

writeln!(f, " SError (A): {}", to\_mask\_str(self.0.is\_set(SPSR\_EL1::A)))?;

writeln!(f, " IRQ (I): {}", to\_mask\_str(self.0.is\_set(SPSR\_EL1::I)))?;

writeln!(f, " FIQ (F): {}", to\_mask\_str(self.0.is\_set(SPSR\_EL1::F)))?;

write!(f, " Illegal Execution State (IL): {}",

to\_flag\_str(self.0.is\_set(SPSR\_EL1::IL))

)

}

}

impl EsrEL1 {

#[inline(always)]

fn exception\_class(&self) -> Option<ESR\_EL1::EC::Value> {

self.0.read\_as\_enum(ESR\_EL1::EC)

}

}

#[rustfmt::skip]

impl fmt::Display for EsrEL1 {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

writeln!(f, "ESR\_EL1: {:#010x}", self.0.get())?;

write!(f, " Exception Class (EC) : {:#x}", self.0.read(ESR\_EL1::EC))?;

let ec\_translation = match self.exception\_class() {

Some(ESR\_EL1::EC::Value::DataAbortCurrentEL) => "Data Abort, current EL",

\_ => "N/A",

};

writeln!(f, " - {}", ec\_translation)?;

write!(f, " Instr Specific Syndrome (ISS): {:#x}", self.0.read(ESR\_EL1::ISS))

}

}

impl ExceptionContext {

#[inline(always)]

fn exception\_class(&self) -> Option<ESR\_EL1::EC::Value> {

self.esr\_el1.exception\_class()

}

#[inline(always)]

fn fault\_address\_valid(&self) -> bool {

use ESR\_EL1::EC::Value::\*;

match self.exception\_class() {

None => false,

Some(ec) => matches!(

ec,

InstrAbortLowerEL

| InstrAbortCurrentEL

| PCAlignmentFault

| DataAbortLowerEL

| DataAbortCurrentEL

| WatchpointLowerEL

| WatchpointCurrentEL

),

}

}

}

impl fmt::Display for ExceptionContext {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

writeln!(f, "{}", self.esr\_el1)?;

if self.fault\_address\_valid() {

writeln!(f, "FAR\_EL1: {:#018x}", FAR\_EL1.get() as usize)?;

}

writeln!(f, "{}", self.spsr\_el1)?;

writeln!(f, "ELR\_EL1: {:#018x}", self.elr\_el1)?;

writeln!(f)?;

writeln!(f, "General purpose register:")?;

#[rustfmt::skip]

let alternating = |x| -> \_ {

if x % 2 == 0 { " " } else { "\n" }

};

for (i, reg) in self.gpr.iter().enumerate() {

write!(f, " x{: <2}: {: >#018x}{}", i, reg, alternating(i))?;

}

write!(f, " lr : {:#018x}", self.lr)

}

}

use crate::exception::PrivilegeLevel;

pub fn current\_privilege\_level() -> (PrivilegeLevel, &'static str) {

let el = CurrentEL.read\_as\_enum(CurrentEL::EL);

match el {

Some(CurrentEL::EL::Value::EL2) => (PrivilegeLevel::Hypervisor, "EL2"),

Some(CurrentEL::EL::Value::EL1) => (PrivilegeLevel::Kernel, "EL1"),

Some(CurrentEL::EL::Value::EL0) => (PrivilegeLevel::User, "EL0"),

\_ => (PrivilegeLevel::Unknown, "Unknown"),

}

}

pub unsafe fn handling\_init() {

extern "Rust" {

static \_\_exception\_vector\_start: UnsafeCell<()>;

}

VBAR\_EL1.set(\_\_exception\_vector\_start.get() as u64);

barrier::isb(barrier::SY);

}

Файл main.rs

#![allow(clippy::upper\_case\_acronyms)]

#![allow(incomplete\_features)]

#![feature(asm\_const)]

#![feature(const\_option)]

#![feature(core\_intrinsics)]

#![feature(format\_args\_nl)]

#![feature(int\_roundings)]

#![feature(nonzero\_min\_max)]

#![feature(panic\_info\_message)]

#![feature(trait\_alias)]

#![feature(unchecked\_math)]

#![no\_main]

#![no\_std]

mod bsp;

mod common;

mod console;

mod cpu;

mod driver;

mod exception;

mod memory;

mod panic\_wait;

mod print;

mod synchronization;

mod time;

unsafe fn kernel\_init() -> ! {

use memory::mmu::interface::MMU;

exception::handling\_init();

if let Err(string) = memory::mmu::mmu().enable\_mmu\_and\_caching() {

panic!("MMU: {}", string);

}

if let Err(x) = bsp::driver::init() {

panic!("Error initializing BSP driver subsystem: {}", x);

}

driver::driver\_manager().init\_drivers();

kernel\_main()

}

fn kernel\_main() -> ! {

use console::console;

use core::time::Duration;

info!(

"{} version {}",

env!("CARGO\_PKG\_NAME"),

env!("CARGO\_PKG\_VERSION")

);

info!("Booting on: {}", bsp::board\_name());

info!("MMU online. Special regions:");

bsp::memory::mmu::virt\_mem\_layout().print\_layout();

let (\_, privilege\_level) = exception::current\_privilege\_level();

info!("Current privilege level: {}", privilege\_level);

info!("Exception handling state:");

exception::asynchronous::print\_state();

info!(

"Architectural timer resolution: {} ns",

time::time\_manager().resolution().as\_nanos()

);

info!("Drivers loaded:");

driver::driver\_manager().enumerate();

info!("Timer test, spinning for 1 second");

time::time\_manager().spin\_for(Duration::from\_secs(1));

info!("");

info!("Trying to read from address 8 GiB...");

let mut big\_addr: u64 = 8 \* 1024 \* 1024 \* 1024;

unsafe { core::ptr::read\_volatile(big\_addr as \*mut u64) };

info!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

info!("Whoa! We recovered from a synchronous exception!");

info!("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

info!("");

info!("Let's try again");

info!("Trying to read from address 9 GiB...");

big\_addr = 9 \* 1024 \* 1024 \* 1024;

unsafe { core::ptr::read\_volatile(big\_addr as \*mut u64) };

info!("Echoing input now");

console().clear\_rx();

loop {

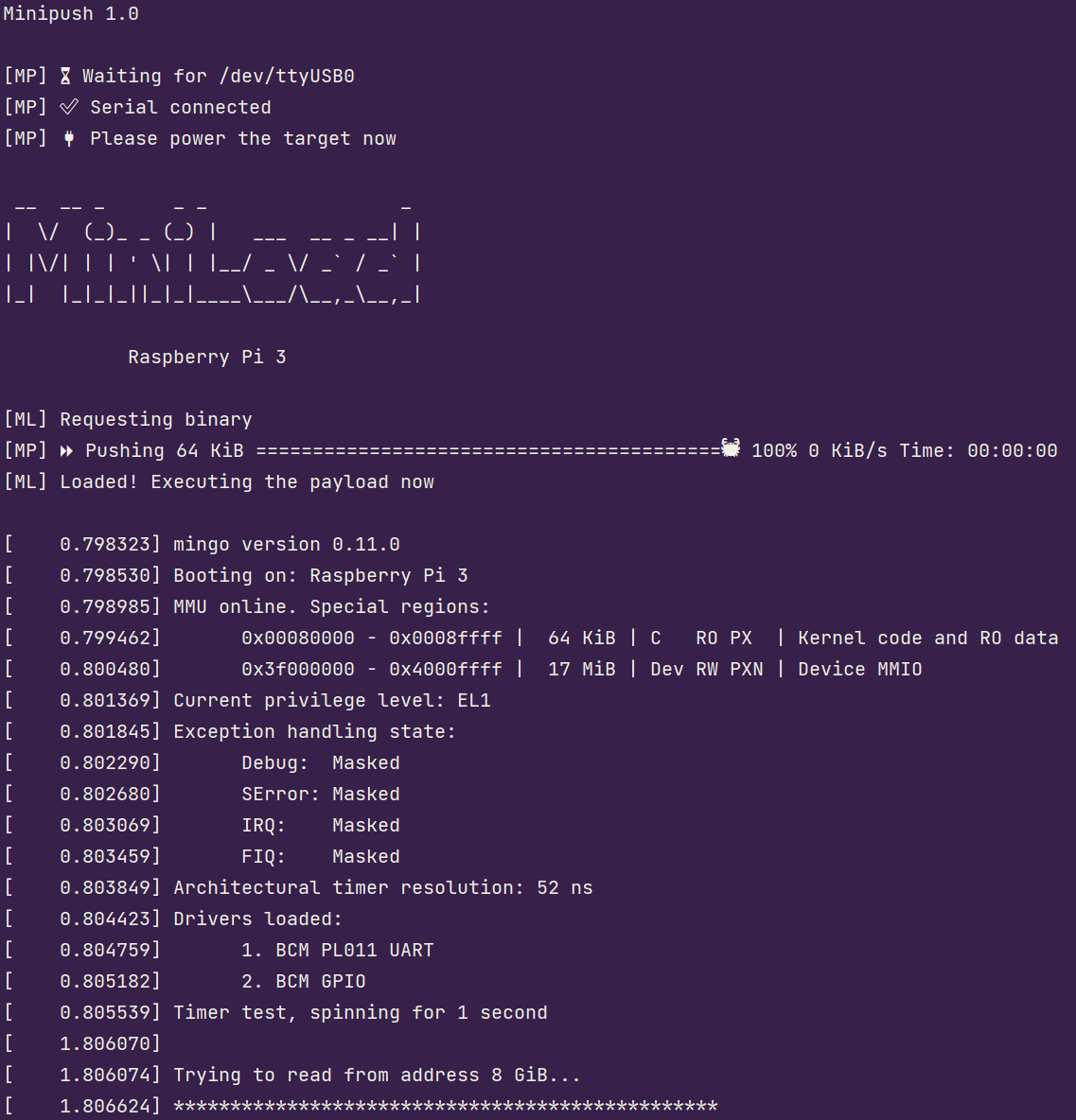
let c = console().read\_char();

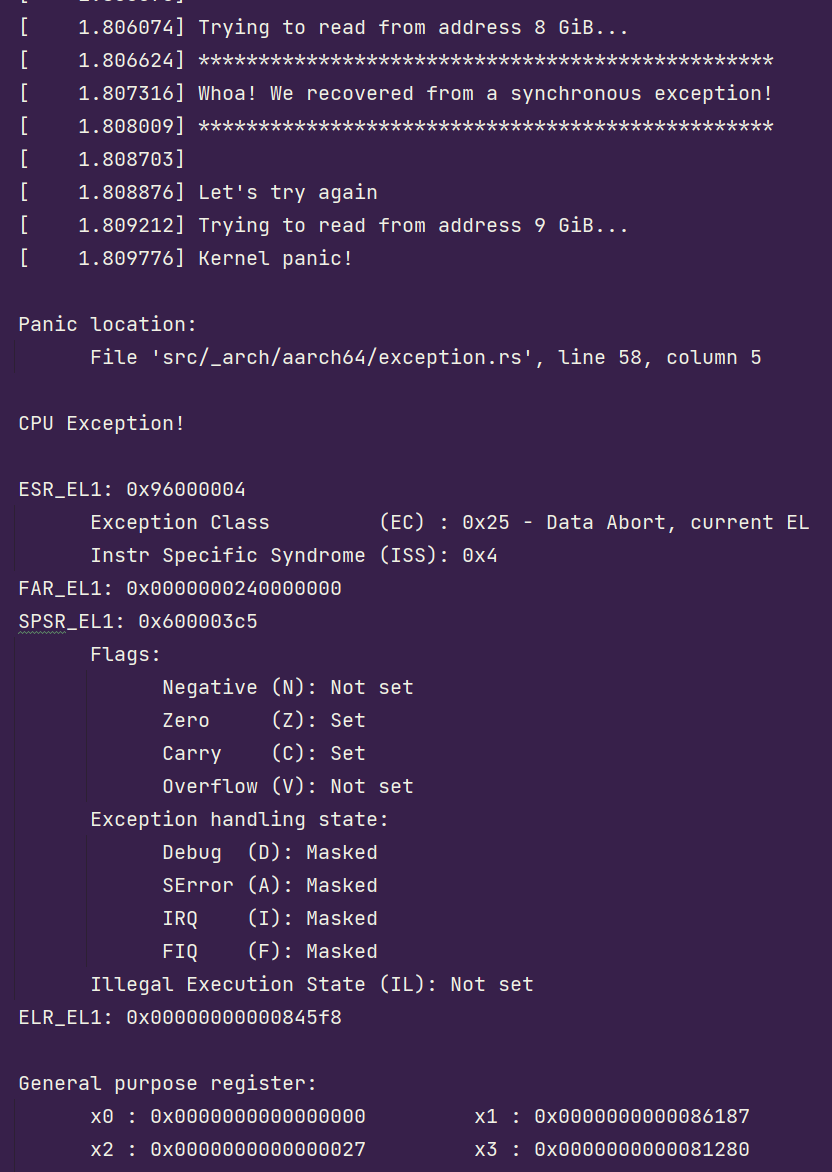
console().write\_char(c);

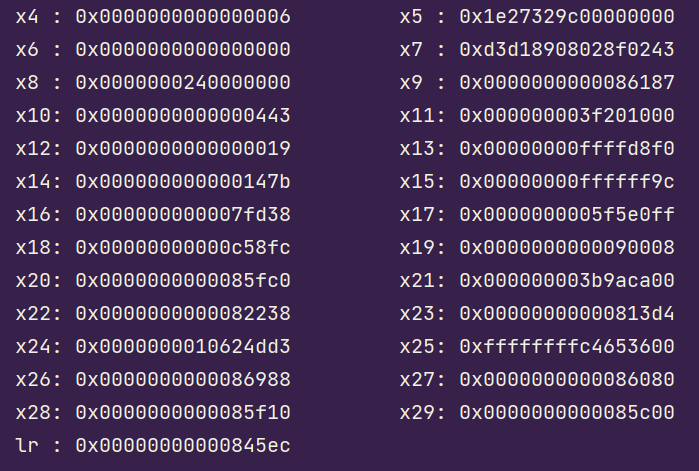
}

}

Результат:







Етап 12

На цьому етапі розробляється інтегроване тестування ядра. Інтегроване тестування використовує власне створене середовище тестування, яке використовує функцію custom\_test\_frameworks в мові програмування Rust. Це середовище дозволяє виконувати модульні тести та інтеграційні тести з використанням QEMU. Також можлива автоматизація тестування вводу-виводу через консоль ядра, яка доступна через UART.

На цьому етапі розширюється наявний базовий тест завантаження ядра, додаванням нових засобів тестування. Ці нові засоби включають класичні модульні тести, тести інтеграції (автономні тести, що зберігаються у каталозі $CRATE/tests/) та тести консольного вводу-виводу. Інтеграційні тести дозволяють тестувати зовнішні впливи, такі як консольний ввід.

Проте тестування коду Rust з атрибутом #![no\_std], як у випадку ядра, не є простою задачею. Використання стандартної системи тестування Rust з макросом атрибута #[test] та запуском команди cargo test може призвести до помилок компіляції через залежності від стандартної бібліотеки. Тому для тестування використовується нестабільна функція custom\_test\_frameworks у Rust. Замість анотування функцій #[test], використовується атрибут #[test\_case]. Також необхідно написати функцію test\_runner, яка виконує всі функції, анотовані #[test\_case].

Для запуску модульних тестів використовується ціль make test\_unit, яка виконує команду cargo test. Для запуску інтеграційних тестів використовується ціль make test\_integration.

Також на цьому етапі розглядається процес виконання модульних тестів у ядрі з використанням засобів тестування, наданих мовою програмування Rust та інфраструктурою Cargo.

1. UnitTest - це структура, яка містить інформацію про модульний тест. Вона містить назву тесту та вказівник на класичну функцію, яка є функцією модульного тестування.
2. test\_runner - це функція, яка друкує ім'я відповідного тесту та викликає функцію модульного тестування.
3. Функція main() - це точка входу для тестового виконання. Вона не викликається напряму, оскільки ми створюємо ядро, а не процес простору користувача. Замість цього, функція main() створюється автоматично під час виконання cargo test, і її виклик відбувається вручну.
4. Атрибут #![reexport\_test\_harness\_main = "test\_main"] - це атрибут, який оголошує ім'я головної функції test\_main, щоб ми могли викликати її вручну. Цей атрибут необхідний, оскільки ми маємо власну функцію \_start(), яка запускає наступний ланцюжок викликів під час завантаження ядра.
5. Функція bsp::console::qemu\_bring\_up\_console() - це виклик функції, яка ініціалізує периферійне обладнання, що реалізує консольний інтерфейс ядра. У даному випадку, для модульних тестів нічого реального не потрібно, тому функція порожня. Але для іншого обладнання, що імітується QEMU, ця функція може бути необхідною.

Використання атрибуту #[kernel\_test]: Це процедурний макрос, який створює структуру UnitTest з атрибутованих функцій. Цей макрос автоматично перетворює назву функції в ім'я члена структури UnitTest і заповнює член test\_func замиканням, яке виконує тіло приписаної функції.

Збереження модульних тестів у відповідних файлах: Модульні тести зберігаються в файлах з розширенням .rs у відповідній папці tests в проекті.

Інтеграційні тести: Інтеграційні тести знаходяться в папці $CRATE/tests/. Кожен файл .rs в цій папці компілюється у свій власний виконуваний файл тестового ядра і виконується окремо за допомогою команди cargo test. Код у файлах інтеграційних тестів включає бібліотечну частину ядра (libkernel) за допомогою операторів use. Точкою входу для кожного інтеграційного тесту повинна бути функція kernel\_init().

Тестова оснастка: Під час виконання команди cargo test, використовується тестова оснастка, яка є згенерованою функцією main(). Вона дозволяє розділити тестовий код на окремі частини. Ця функція автоматично додається до інтеграційних тестів.

Консольні тести: Для тестування взаємодії користувача/ядра через послідовну консоль використовується інфраструктура тестової оснастки, яка має сценарій обгортки dispatch.rb. Цей сценарій надає можливість вводу-виводу на консоль та очікування певних відповідей. Для кожного інтеграційного тесту перевіряється наявність відповідного файлу супроводу (.rb), який містить підтести консолі вводу-виводу. Ці підтести виконуються за допомогою класу ConsoleIOTest, який спочатку створює QEMU і підключається до емуляції послідовної консолі QEMU, а потім виконує консольні підтести.

Таким чином, на цьому етапі розроблюються модульні тести з використанням макросу #[kernel\_test], інтеграційні тести, які перевіряють взаємодію між різними компонентами системи, тестова оснастка для об'єднання тестів та консольні тести для перевірки взаємодії з користувачем через консоль.

Файл cpu.rs

#[cfg(feature = "test\_build")]

use qemu\_exit::QEMUExit;

#[cfg(feature = "test\_build")]

const QEMU\_EXIT\_HANDLE: qemu\_exit::AArch64 = qemu\_exit::AArch64::new();

#[cfg(feature = "test\_build")]

pub fn qemu\_exit\_failure() -> ! {

QEMU\_EXIT\_HANDLE.exit\_failure()

}

#[cfg(feature = "test\_build")]

pub fn qemu\_exit\_success() -> ! {

QEMU\_EXIT\_HANDLE.exit\_success()

}

Файл lib.rs

pub fn test\_runner(tests: &[&test\_types::UnitTest]) {

println!("Running {} tests", tests.len());

for (i, test) in tests.iter().enumerate() {

print!("{:>3}. {:.<58}", i + 1, test.name);

(test.test\_func)();

// Failed tests call panic!(). Execution reaches here only if the test has passed.

println!("[ok]")

}

}

Файл Cargo.toml

[[test]]

name = "00\_console\_sanity"

harness = false

[[test]]

name = "02\_exception\_sync\_page\_fault"

harness = false

[[test]]

name = "03\_exception\_restore\_sanity"

harness = false

Файл 00\_console\_sanity.rb

require 'console\_io\_test'

class TxRxHandshakeTest < SubtestBase

def name

'Transmit and Receive handshake'

end

def run(qemu\_out, qemu\_in)

qemu\_in.write\_nonblock('ABC')

expect\_or\_raise(qemu\_out, 'OK1234')

end

end

#![feature(format\_args\_nl)]

#![no\_main]

#![no\_std]

use libkernel::{bsp, console, cpu, exception, print};

#[no\_mangle]

unsafe fn kernel\_init() -> ! {

use console::console;

exception::handling\_init();

bsp::driver::qemu\_bring\_up\_console();

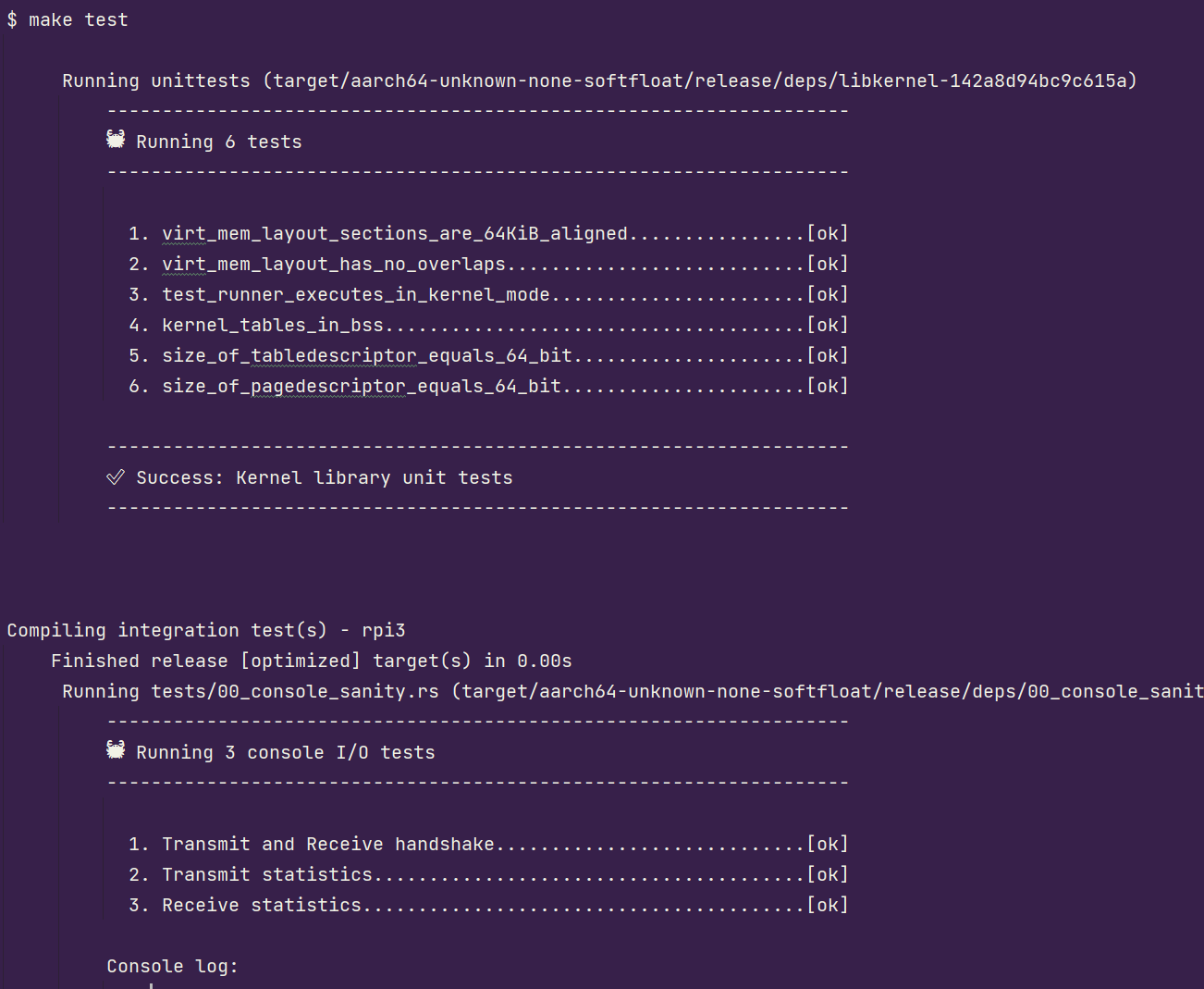
assert\_eq!(console().read\_char(), 'A');

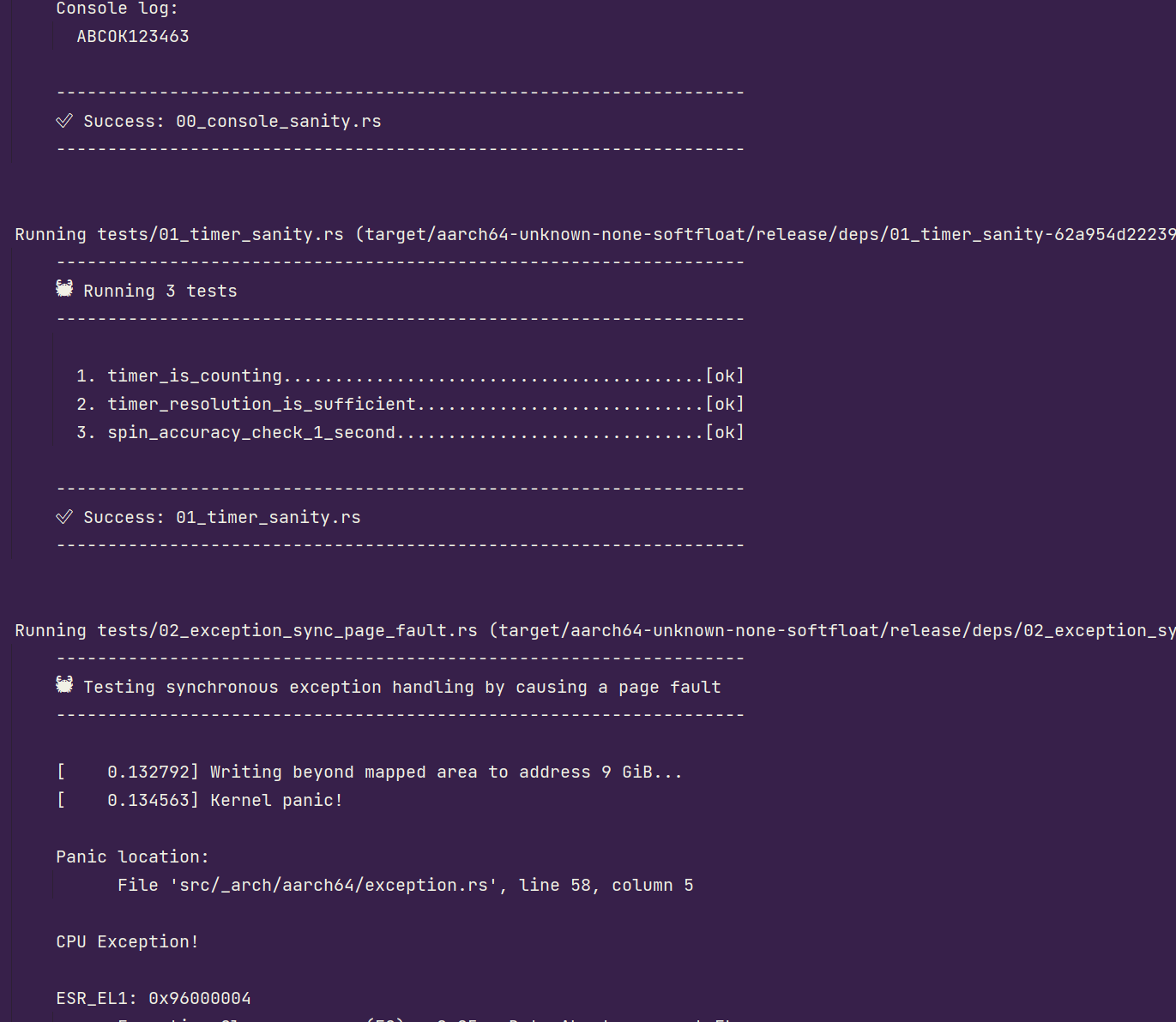
assert\_eq!(console().read\_char(), 'B');

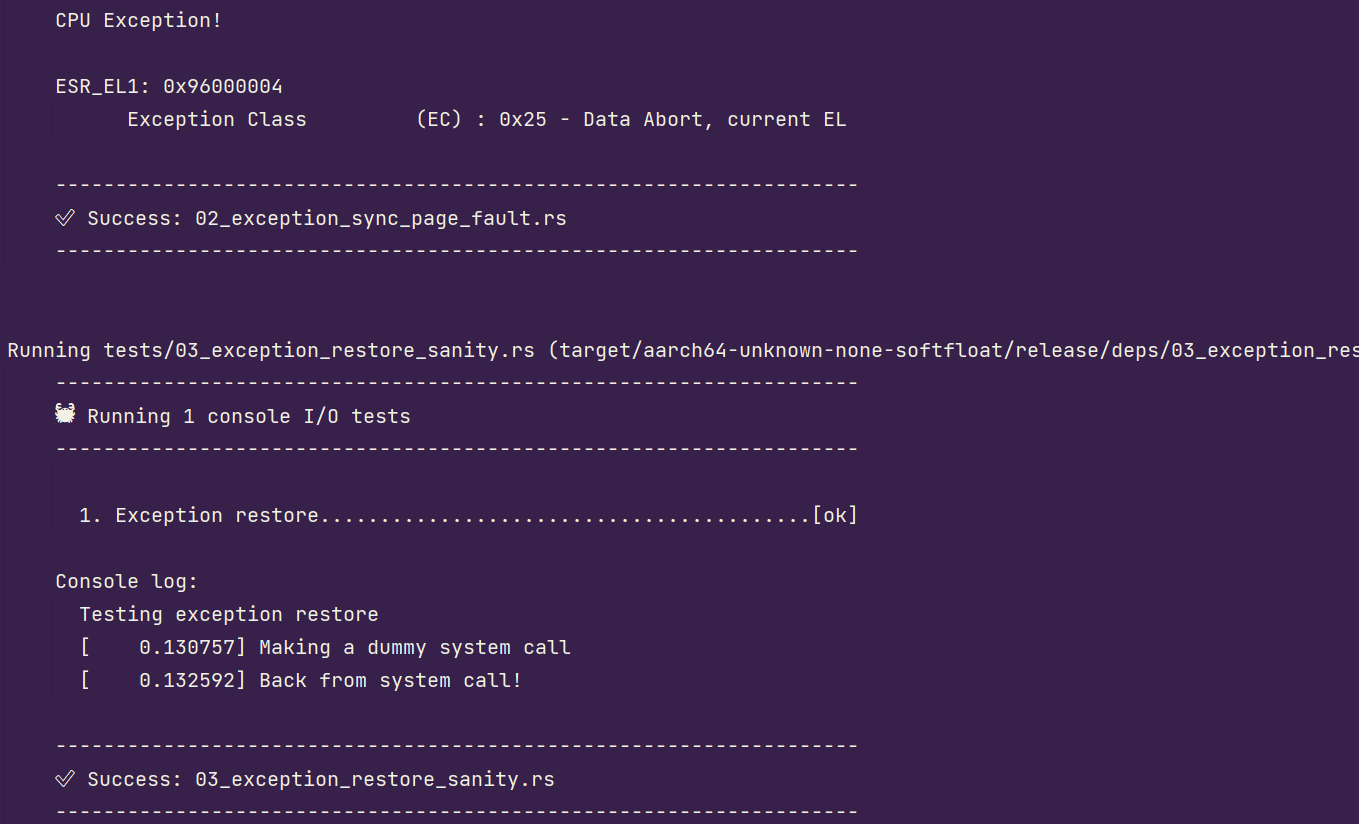
assert\_eq!(console().read\_char(), 'C');

print!("OK1234");

Результати:







Етап 13

На цьому етапі розробляються драйвери пристроїв для контролерів переривань на Raspberry Pi 3 (Broadcom користувацький контролер) та Pi 4 (ARM Generic Interrupt Controller v2, GICv2). Це включає реалізацію обробки периферійних переривань, зокрема обробку переривання UART.

У розробці використовується підхід модульності шляхом використання ознаки IRQManager для взаємодії між компонентами. Реалізовано підтримку обробки асинхронних переривань типу IRQ для архітектури AArch64. Метою є заміна активного опитування UART на режим очікування, використовуючи переривання UART IRQ RX для вказівки отримання нових символів.

Розробляється загальна ідіома програмного інтерфейсу IRQManager, яка визначає функціональні можливості для реєстрації обробників переривань, увімкнення та обробки переривань, а також виводу списку зареєстрованих обробників. Ця загальна ідіома дозволяє створити чисту абстракцію та модульний код.

Також на цьому етапі вирішуються питання захисту розділів коду від повторного введення. Для цього використовуються псевдоблокування NullLock замість реальних Spinlock, оскільки ядро все ще працює на єдиному ядрі.

Унікальне визначення IRQ включає в себе асоційований тип IRQNumberType, який вирішує проблему однозначної ідентифікації IRQ, враховуючи їх властивості, такі як створені програмним або апаратним забезпеченням, приватні або спільні. Для BCM контролера Raspberry Pi 3, який має різні типи IRQ (локальний та периферійний), використовується асоційований тип IRQNumberType для забезпечення унікальності та ідентифікації кожного типу переривання.

Загалом, розробка драйверів пристроїв для обробки переривань на Raspberry Pi включає створення обробників переривань, реєстрацію їх у системі, налаштування контролерів переривань та обробку переривань згідно з вимогами конкретних пристроїв. Це дозволяє програмі взаємодіяти з різноманітними зовнішніми пристроями, такими як UART, GPIO, SPI, I2C та інші, шляхом обробки їх переривань.

Зважаючи на те, що інформація про розробку драйверів на Raspberry Pi може змінюватися з часом, рекомендую відвідати офіційний веб-сайт Raspberry Pi або форум спільноти Raspberry Pi, де можна знайти оновлену інформацію та ресурси для розробки драйверів для Raspberry Pi.

Для керування перериваннями використовується контролер переривань GICv2.

Схема номерів IRQ у GICv2 на Raspberry Pi 4 відрізняється від інших схем. Номери IRQ 0..31 призначені для приватних IRQ. Вони поділяються на дві категорії: SW-generated (SGIs 0..15) та HW-generated (PPIs, Private Peripheral Interrupts, 16..31).

Номери 32..1019 призначені для спільних апаратно згенерованих переривань (SPI, Shared Peripheral Interrupts). В цій схемі використовується просте ціле число як унікальний ідентифікатор для кожного IRQ. Важливо зазначити, що немає перекривань між номерами IRQ.

Для того, щоб драйвер контролера переривань міг керувати обробкою переривань, йому потрібно знати, де знаходяться обробники та як їх викликати. Для цього використовується ознака IRQHandler в exception::asynchronous, яку має реалізувати будь-яка сутність ПЗ, яка бажає обробляти переривання.

Першим драйвером, який реалізує цю ознаку, є драйвер PL011Uart. На цьому етапі налаштовуються переривання RX IRQ і RX Timeout IRQ для PL011Uart. PL011Uart стверджуватиме свою лінію переривання, коли виконується одна з умов: рівень заповнення RX FIFO дорівнює або перевищує налаштований рівень тригера або рівень заповнення RX FIFO більший за нуль, але менший за налаштований рівень заповнення, і символи не витягувалися протягом певного часу.

Для реєстрації та включення обробників переривань в контролері переривань GICv2 використовуються наступні кроки:

1. Отримання доступу до реєстру контролера переривань GICv2. Це може виконуватися шляхом отримання вказівника на відповідну структуру даних, яка представляє контролер переривань.
2. Налаштування обробників переривань. Кожен обробник переривань повинен бути зареєстрований у контролері переривань з вказанням його вектора переривань (номеру IRQ) та вказівника на функцію-обробник. Це зазвичай робиться шляхом виклику функції реєстрації обробника переривань, яка передає необхідну інформацію контролеру переривань.
3. Включення обробки переривань. Це виконується шляхом встановлення бітів у відповідних регістрах контролера переривань, які вказують, які переривання мають бути увімкнені.
4. Налаштування пріоритетів переривань. Кожне переривання може мати пріоритет, що вказує на його важливість. Вищий пріоритет означає, що переривання буде оброблено першим. Контролер переривань має регістри, які дозволяють налаштувати пріоритети переривань.
5. Обробка переривань. Після включення обробки переривань та налаштування обробників, контролер переривань буде автоматично викликати відповідні функції-обробники при спрацьовуванні переривань.

Один зі способів досягнення захисту певних функцій від повторного входу та паралельного доступу до певних ресурсів полягає у використанні токена IRQContext, який вимагається від користувача, що викликає функцію handle\_pending\_irqs(). Цей токен дозволяє запобігти випадковому виклику цієї функції з інших місць. Однак, для того, щоб цей механізм працював, користувач повинен створити токен IRQContext лише в контексті обробки переривань. Це гарантує, що функції захищені від повторного входу.

Процес захисту певних ресурсів від паралельного доступу. У попередній роботі були підготовлені деякі ресурси до захисту від паралельного доступу, використовуючи NullLocks. Тепер ці NullLocks оновлюються до IRQSafeNulllocks, щоб захистити ресурси від повторного входу.

Для реалізації цих механізмів безпеки та реентерабельності використовуються спеціальні структури, такі як IRQManager і InitStateLock. Крім того, описується структура верхнього рівня, що складається з декількох компонентів, таких як GICD (дистриб'ютор), GICC (інтерфейс процесора) та HandlerTable.

В описі реалізації драйвера GICv2 (Pi 4) розглядаються особливості приватних і спільних номерів переривань. Приватні номери переривань використовуються для індивідуального преривання кожного процесора, тоді як спільні номери переривань використовуються для преривання всіх процесорів.

Крім того, описується робота GICD (дистриб'ютора) і GICC (інтерфейсу процесора). GICD відповідає за розподіл переривань до процесорів, включаючи вибір пріоритету переривань, маскування та розміщення в черзі. GICC забезпечує інтерфейс для обробки переривань на процесорі, включаючи прийом, обробку і відправку переривань.

Механізм реєстрації та обробки обробників переривань за допомогою структури HandlerTable. Ця структура містить вказівники на функції обробки переривань для кожного номера переривання. При отриманні переривання, відповідний обробник викликається для обробки цього переривання.

Використання структури InitStateLock для синхронізації ініціалізації драйвера. Ця структура забезпечує доступ до ініціалізаційних даних тільки одному потоку в момент часу, що допомагає уникнути конфліктів та непередбачуваних станів.

Узагальнюючи, розглянута реалізація безпеки та реентерабельності в контексті обробки переривань забезпечує захист від повторного входу і паралельного доступу до ресурсів. Використання спеціальних структур, таких як IRQManager, IRQContext, IRQSafeNulllocks, HandlerTable і InitStateLock, допомагає забезпечити коректну роботу обробки переривань в системі.

Реалізація:

Файл gicv2.rs

mod gicc;

mod gicd;

use crate::{

bsp::{self, device\_driver::common::BoundedUsize},

cpu, driver, exception, synchronization,

synchronization::InitStateLock,

};

type HandlerTable = [Option<exception::asynchronous::IRQHandlerDescriptor<IRQNumber>>;

IRQNumber::MAX\_INCLUSIVE + 1];

[`exception::asynchronous::interface::IRQManager`].

pub type IRQNumber = BoundedUsize<{ GICv2::MAX\_IRQ\_NUMBER }>;

pub struct GICv2 {

gicd: gicd::GICD,

gicc: gicc::GICC,

handler\_table: InitStateLock<HandlerTable>,

}

impl GICv2 {

const MAX\_IRQ\_NUMBER: usize = 300; // Normally 1019, but keep it lower to save some space.

pub const COMPATIBLE: &'static str = "GICv2 (ARM Generic Interrupt Controller v2)";

pub const unsafe fn new(gicd\_mmio\_start\_addr: usize, gicc\_mmio\_start\_addr: usize) -> Self {

Self {

gicd: gicd::GICD::new(gicd\_mmio\_start\_addr),

gicc: gicc::GICC::new(gicc\_mmio\_start\_addr),

handler\_table: InitStateLock::new([None; IRQNumber::MAX\_INCLUSIVE + 1]),

}

}

}

use synchronization::interface::ReadWriteEx;

impl driver::interface::DeviceDriver for GICv2 {

type IRQNumberType = IRQNumber;

fn compatible(&self) -> &'static str {

Self::COMPATIBLE

}

unsafe fn init(&self) -> Result<(), &'static str> {

if bsp::cpu::BOOT\_CORE\_ID == cpu::smp::core\_id() {

self.gicd.boot\_core\_init();

}

self.gicc.priority\_accept\_all();

self.gicc.enable();

Ok(())

}

}

impl exception::asynchronous::interface::IRQManager for GICv2 {

type IRQNumberType = IRQNumber;

fn register\_handler(

&self,

irq\_handler\_descriptor: exception::asynchronous::IRQHandlerDescriptor<Self::IRQNumberType>,

) -> Result<(), &'static str> {

self.handler\_table.write(|table| {

let irq\_number = irq\_handler\_descriptor.number().get();

if table[irq\_number].is\_some() {

return Err("IRQ handler already registered");

}

table[irq\_number] = Some(irq\_handler\_descriptor);

Ok(())

})

}

fn enable(&self, irq\_number: &Self::IRQNumberType) {

self.gicd.enable(irq\_number);

}

fn handle\_pending\_irqs<'irq\_context>(

&'irq\_context self,

ic: &exception::asynchronous::IRQContext<'irq\_context>,

) {

let irq\_number = self.gicc.pending\_irq\_number(ic);

if irq\_number > GICv2::MAX\_IRQ\_NUMBER {

return;

}

self.handler\_table.read(|table| {

match table[irq\_number] {

None => panic!("No handler registered for IRQ {}", irq\_number),

Some(descriptor) => {

// Call the IRQ handler. Panics on failure.

descriptor.handler().handle().expect("Error handling IRQ");

}

}

});

self.gicc.mark\_comleted(irq\_number as u32, ic);

}

fn print\_handler(&self) {

use crate::info;

info!(" Peripheral handler:");

self.handler\_table.read(|table| {

for (i, opt) in table.iter().skip(32).enumerate() {

if let Some(handler) = opt {

info!(" {: >3}. {}", i + 32, handler.name());

}

}

});

}

}

Файл peripheral\_ic.rs

use super::{PendingIRQs, PeripheralIRQ};

use crate::{

bsp::device\_driver::common::MMIODerefWrapper,

exception, synchronization,

synchronization::{IRQSafeNullLock, InitStateLock},

};

use tock\_registers::{

interfaces::{Readable, Writeable},

register\_structs,

registers::{ReadOnly, WriteOnly},

};

register\_structs! {

#[allow(non\_snake\_case)]

WORegisterBlock {

(0x00 => \_reserved1),

(0x10 => ENABLE\_1: WriteOnly<u32>),

(0x14 => ENABLE\_2: WriteOnly<u32>),

(0x18 => @END),

}

}

register\_structs! {

#[allow(non\_snake\_case)]

RORegisterBlock {

(0x00 => \_reserved1),

(0x04 => PENDING\_1: ReadOnly<u32>),

(0x08 => PENDING\_2: ReadOnly<u32>),

(0x0c => @END),

}

}

type WriteOnlyRegisters = MMIODerefWrapper<WORegisterBlock>;

type ReadOnlyRegisters = MMIODerefWrapper<RORegisterBlock>;

type HandlerTable = [Option<exception::asynchronous::IRQHandlerDescriptor<PeripheralIRQ>>;

PeripheralIRQ::MAX\_INCLUSIVE + 1];

pub struct PeripheralIC {

wo\_registers: IRQSafeNullLock<WriteOnlyRegisters>,

ro\_registers: ReadOnlyRegisters,

handler\_table: InitStateLock<HandlerTable>,

}

impl PeripheralIC {

pub const unsafe fn new(mmio\_start\_addr: usize) -> Self {

Self {

wo\_registers: IRQSafeNullLock::new(WriteOnlyRegisters::new(mmio\_start\_addr)),

ro\_registers: ReadOnlyRegisters::new(mmio\_start\_addr),

handler\_table: InitStateLock::new([None; PeripheralIRQ::MAX\_INCLUSIVE + 1]),

}

}

fn pending\_irqs(&self) -> PendingIRQs {

let pending\_mask: u64 = (u64::from(self.ro\_registers.PENDING\_2.get()) << 32)

| u64::from(self.ro\_registers.PENDING\_1.get());

PendingIRQs::new(pending\_mask)

}

}

use synchronization::interface::{Mutex, ReadWriteEx};

impl exception::asynchronous::interface::IRQManager for PeripheralIC {

type IRQNumberType = PeripheralIRQ;

fn register\_handler(

&self,

irq\_handler\_descriptor: exception::asynchronous::IRQHandlerDescriptor<Self::IRQNumberType>,

) -> Result<(), &'static str> {

self.handler\_table.write(|table| {

let irq\_number = irq\_handler\_descriptor.number().get();

if table[irq\_number].is\_some() {

return Err("IRQ handler already registered");

}

table[irq\_number] = Some(irq\_handler\_descriptor);

Ok(())

})

}

fn enable(&self, irq: &Self::IRQNumberType) {

self.wo\_registers.lock(|regs| {

let enable\_reg = if irq.get() <= 31 {

&regs.ENABLE\_1

} else {

&regs.ENABLE\_2

};

let enable\_bit: u32 = 1 << (irq.get() % 32);

enable\_reg.set(enable\_bit);

});

}

fn handle\_pending\_irqs<'irq\_context>(

&'irq\_context self,

\_ic: &exception::asynchronous::IRQContext<'irq\_context>,

) {

self.handler\_table.read(|table| {

for irq\_number in self.pending\_irqs() {

match table[irq\_number] {

None => panic!("No handler registered for IRQ {}", irq\_number),

Some(descriptor) => {

descriptor.handler().handle().expect("Error handling IRQ");

}

}

}

})

}

fn print\_handler(&self) {

use crate::info;

info!(" Peripheral handler:");

self.handler\_table.read(|table| {

for (i, opt) in table.iter().enumerate() {

if let Some(handler) = opt {

info!(" {: >3}. {}", i, handler.name());

}

}

});

}

}

Результат:



Етап 14:

На цьому етапі відбуваються наступні дії:

1. Налаштування віртуальної пам'яті: Використовуючи платформо-залежний код, встановлюється віртуальна пам'ять та вирішуються питання, пов'язані зі сторінковим перекладом адрес (paging), дескрипторами сторінок і таблицями сторінок.
2. Remap MMIO (переміщення вхідно-вихідних адрес): За допомогою механізму віртуальної пам'яті, проводиться перенаправлення (розташування) пам'яті ввідно-вивідних пристроїв (MMIO) до віртуального адресного простору ядра ОС. Це означає, що пам'ять MMIO тепер можна доступатися через віртуальні адреси, що спрощує роботу з пристроями.
3. Реалізація драйверів MMIO: Драйвери для ввідно-вивідних пристроїв (наприклад, UART або GPIO) реалізуються для взаємодії з цими пристроями через MMIO. Це означає, що ОС тепер може взаємодіяти з пристроями, використовуючи віртуальні адреси та відповідні драйвери.

В цілому, на цьому етапі здійснюється налаштування віртуальної пам'яті та забезпечення роботи з ввідно-вивідними пристроями через віртуальний адресний простір. Це дозволяє ОС ефективно керувати пристроями і забезпечує зручний спосіб взаємодії з ними.

Реалізація:

Файл translation\_table.rs

use crate::{

bsp,

memory::{

self,

mmu::{

arch\_mmu::{Granule512MiB, Granule64KiB},

AccessPermissions, AttributeFields, MemAttributes, MemoryRegion, PageAddress,

},

Address, Physical, Virtual,

},

};

use core::convert;

use tock\_registers::{

interfaces::{Readable, Writeable},

register\_bitfields,

registers::InMemoryRegister,

};

register\_bitfields! {u64,

STAGE1\_TABLE\_DESCRIPTOR [

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

]

]

}

register\_bitfields! {u64,

STAGE1\_PAGE\_DESCRIPTOR [

UXN OFFSET(54) NUMBITS(1) [

False = 0,

True = 1

],

PXN OFFSET(53) NUMBITS(1) [

False = 0,

True = 1

],

OUTPUT\_ADDR\_64KiB OFFSET(16) NUMBITS(32) [], // [47:16]

AF OFFSET(10) NUMBITS(1) [

False = 0,

True = 1

],

SH OFFSET(8) NUMBITS(2) [

OuterShareable = 0b10,

InnerShareable = 0b11

],

AP OFFSET(6) NUMBITS(2) [

RW\_EL1 = 0b00,

RW\_EL1\_EL0 = 0b01,

RO\_EL1 = 0b10,

RO\_EL1\_EL0 = 0b11

],

AttrIndx OFFSET(2) NUMBITS(3) [],

TYPE OFFSET(1) NUMBITS(1) [

Reserved\_Invalid = 0,

Page = 1

],

VALID OFFSET(0) NUMBITS(1) [

False = 0,

True = 1

]

]

}

#[derive(Copy, Clone)]

#[repr(C)]

struct TableDescriptor {

value: u64,

}

#[derive(Copy, Clone)]

#[repr(C)]

struct PageDescriptor {

value: u64,

}

trait StartAddr {

fn phys\_start\_addr(&self) -> Address<Physical>;

}

#[repr(C)]

#[repr(align(65536))]

pub struct FixedSizeTranslationTable<const NUM\_TABLES: usize> {

lvl3: [[PageDescriptor; 8192]; NUM\_TABLES],

lvl2: [TableDescriptor; NUM\_TABLES],

initialized: bool,

}

impl<T, const N: usize> StartAddr for [T; N] {

fn phys\_start\_addr(&self) -> Address<Physical> {

Address::new(self as \*const \_ as usize)

}

}

impl TableDescriptor {

pub const fn new\_zeroed() -> Self {

Self { value: 0 }

}

pub fn from\_next\_lvl\_table\_addr(phys\_next\_lvl\_table\_addr: Address<Physical>) -> Self {

let val = InMemoryRegister::<u64, STAGE1\_TABLE\_DESCRIPTOR::Register>::new(0);

let shifted = phys\_next\_lvl\_table\_addr.as\_usize() >> Granule64KiB::SHIFT;

val.write(

STAGE1\_TABLE\_DESCRIPTOR::NEXT\_LEVEL\_TABLE\_ADDR\_64KiB.val(shifted as u64)

+ STAGE1\_TABLE\_DESCRIPTOR::TYPE::Table

+ STAGE1\_TABLE\_DESCRIPTOR::VALID::True,

);

TableDescriptor { value: val.get() }

}

}

impl convert::From<AttributeFields>

for tock\_registers::fields::FieldValue<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>

{

fn from(attribute\_fields: AttributeFields) -> Self {

let mut desc = match attribute\_fields.mem\_attributes {

MemAttributes::CacheableDRAM => {

STAGE1\_PAGE\_DESCRIPTOR::SH::InnerShareable

+ STAGE1\_PAGE\_DESCRIPTOR::AttrIndx.val(memory::mmu::arch\_mmu::mair::NORMAL)

}

MemAttributes::Device => {

STAGE1\_PAGE\_DESCRIPTOR::SH::OuterShareable

+ STAGE1\_PAGE\_DESCRIPTOR::AttrIndx.val(memory::mmu::arch\_mmu::mair::DEVICE)

}

};

desc += match attribute\_fields.acc\_perms {

AccessPermissions::ReadOnly => STAGE1\_PAGE\_DESCRIPTOR::AP::RO\_EL1,

AccessPermissions::ReadWrite => STAGE1\_PAGE\_DESCRIPTOR::AP::RW\_EL1,

};

desc += if attribute\_fields.execute\_never {

STAGE1\_PAGE\_DESCRIPTOR::PXN::True

} else {

STAGE1\_PAGE\_DESCRIPTOR::PXN::False

};

desc += STAGE1\_PAGE\_DESCRIPTOR::UXN::True;

desc

}

}

impl PageDescriptor {

pub const fn new\_zeroed() -> Self {

Self { value: 0 }

}

pub fn from\_output\_page\_addr(

phys\_output\_page\_addr: PageAddress<Physical>,

attribute\_fields: &AttributeFields,

) -> Self {

let val = InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(0);

let shifted = phys\_output\_page\_addr.into\_inner().as\_usize() >> Granule64KiB::SHIFT;

val.write(

STAGE1\_PAGE\_DESCRIPTOR::OUTPUT\_ADDR\_64KiB.val(shifted as u64)

+ STAGE1\_PAGE\_DESCRIPTOR::AF::True

+ STAGE1\_PAGE\_DESCRIPTOR::TYPE::Page

+ STAGE1\_PAGE\_DESCRIPTOR::VALID::True

+ (\*attribute\_fields).into(),

);

Self { value: val.get() }

}

fn is\_valid(&self) -> bool {

InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(self.value)

.is\_set(STAGE1\_PAGE\_DESCRIPTOR::VALID)

}

}

impl<const AS\_SIZE: usize> memory::mmu::AssociatedTranslationTable

for memory::mmu::AddressSpace<AS\_SIZE>

where

[u8; Self::SIZE >> Granule512MiB::SHIFT]: Sized,

{

type TableStartFromBottom = FixedSizeTranslationTable<{ Self::SIZE >> Granule512MiB::SHIFT }>;

}

impl<const NUM\_TABLES: usize> FixedSizeTranslationTable<NUM\_TABLES> {

#[allow(clippy::assertions\_on\_constants)]

pub const fn new() -> Self {

assert!(bsp::memory::mmu::KernelGranule::SIZE == Granule64KiB::SIZE);

assert!(NUM\_TABLES > 0);

Self {

lvl3: [[PageDescriptor::new\_zeroed(); 8192]; NUM\_TABLES],

lvl2: [TableDescriptor::new\_zeroed(); NUM\_TABLES],

initialized: false,

}

}

#[inline(always)]

fn lvl2\_lvl3\_index\_from\_page\_addr(

&self,

virt\_page\_addr: PageAddress<Virtual>,

) -> Result<(usize, usize), &'static str> {

let addr = virt\_page\_addr.into\_inner().as\_usize();

let lvl2\_index = addr >> Granule512MiB::SHIFT;

let lvl3\_index = (addr & Granule512MiB::MASK) >> Granule64KiB::SHIFT;

if lvl2\_index > (NUM\_TABLES - 1) {

return Err("Virtual page is out of bounds of translation table");

}

Ok((lvl2\_index, lvl3\_index))

}

#[inline(always)]

fn set\_page\_descriptor\_from\_page\_addr(

&mut self,

virt\_page\_addr: PageAddress<Virtual>,

new\_desc: &PageDescriptor,

) -> Result<(), &'static str> {

let (lvl2\_index, lvl3\_index) = self.lvl2\_lvl3\_index\_from\_page\_addr(virt\_page\_addr)?;

let desc = &mut self.lvl3[lvl2\_index][lvl3\_index];

if desc.is\_valid() {

return Err("Virtual page is already mapped");

}

\*desc = \*new\_desc;

Ok(())

}

}

impl<const NUM\_TABLES: usize> memory::mmu::translation\_table::interface::TranslationTable

for FixedSizeTranslationTable<NUM\_TABLES>

{

fn init(&mut self) {

if self.initialized {

return;

}

for (lvl2\_nr, lvl2\_entry) in self.lvl2.iter\_mut().enumerate() {

let phys\_table\_addr = self.lvl3[lvl2\_nr].phys\_start\_addr();

let new\_desc = TableDescriptor::from\_next\_lvl\_table\_addr(phys\_table\_addr);

\*lvl2\_entry = new\_desc;

}

self.initialized = true;

}

fn phys\_base\_address(&self) -> Address<Physical> {

self.lvl2.phys\_start\_addr()

}

unsafe fn map\_at(

&mut self,

virt\_region: &MemoryRegion<Virtual>,

phys\_region: &MemoryRegion<Physical>,

attr: &AttributeFields,

) -> Result<(), &'static str> {

assert!(self.initialized, "Translation tables not initialized");

if virt\_region.size() != phys\_region.size() {

return Err("Tried to map memory regions with unequal sizes");

}

if phys\_region.end\_exclusive\_page\_addr() > bsp::memory::phys\_addr\_space\_end\_exclusive\_addr()

{

return Err("Tried to map outside of physical address space");

}

let iter = phys\_region.into\_iter().zip(virt\_region.into\_iter());

for (phys\_page\_addr, virt\_page\_addr) in iter {

let new\_desc = PageDescriptor::from\_output\_page\_addr(phys\_page\_addr, attr);

let virt\_page = virt\_page\_addr;

self.set\_page\_descriptor\_from\_page\_addr(virt\_page, &new\_desc)?;

}

Ok(())

}

}

#[cfg(test)]

pub type MinSizeTranslationTable = FixedSizeTranslationTable<1>;

#[cfg(test)]

mod tests {

use super::\*;

use test\_macros::kernel\_test;

#[kernel\_test]

fn size\_of\_tabledescriptor\_equals\_64\_bit() {

assert\_eq!(

core::mem::size\_of::<TableDescriptor>(),

core::mem::size\_of::<u64>()

);

}

#[kernel\_test]

fn size\_of\_pagedescriptor\_equals\_64\_bit() {

assert\_eq!(

core::mem::size\_of::<PageDescriptor>(),

core::mem::size\_of::<u64>()

);

}

}

Файл mmu.rs

use crate::{

bsp, memory,

memory::{mmu::TranslationGranule, Address, Physical},

};

use aarch64\_cpu::{asm::barrier, registers::\*};

use core::intrinsics::unlikely;

use tock\_registers::interfaces::{ReadWriteable, Readable, Writeable};

/// Memory Management Unit type.

struct MemoryManagementUnit;

pub type Granule512MiB = TranslationGranule<{ 512 \* 1024 \* 1024 }>;

pub type Granule64KiB = TranslationGranule<{ 64 \* 1024 }>;

#[allow(dead\_code)]

pub mod mair {

pub const DEVICE: u64 = 0;

pub const NORMAL: u64 = 1;

}

static MMU: MemoryManagementUnit = MemoryManagementUnit;

impl<const AS\_SIZE: usize> memory::mmu::AddressSpace<AS\_SIZE> {

pub const fn arch\_address\_space\_size\_sanity\_checks() {

assert!((AS\_SIZE % Granule512MiB::SIZE) == 0);

assert!(AS\_SIZE <= (1 << 48));

}

}

impl MemoryManagementUnit {

fn set\_up\_mair(&self) {

MAIR\_EL1.write(

MAIR\_EL1::Attr1\_Normal\_Outer::WriteBack\_NonTransient\_ReadWriteAlloc +

MAIR\_EL1::Attr1\_Normal\_Inner::WriteBack\_NonTransient\_ReadWriteAlloc +

MAIR\_EL1::Attr0\_Device::nonGathering\_nonReordering\_EarlyWriteAck,

);

}

fn configure\_translation\_control(&self) {

let t0sz = (64 - bsp::memory::mmu::KernelVirtAddrSpace::SIZE\_SHIFT) as u64;

TCR\_EL1.write(

TCR\_EL1::TBI0::Used

+ TCR\_EL1::IPS::Bits\_40

+ 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::A1::TTBR0

+ TCR\_EL1::T0SZ.val(t0sz)

+ TCR\_EL1::EPD1::DisableTTBR1Walks,

);

}

}

pub fn mmu() -> &'static impl memory::mmu::interface::MMU {

&MMU

}

use memory::mmu::MMUEnableError;

impl memory::mmu::interface::MMU for MemoryManagementUnit {

unsafe fn enable\_mmu\_and\_caching(

&self,

phys\_tables\_base\_addr: Address<Physical>,

) -> Result<(), MMUEnableError> {

if unlikely(self.is\_enabled()) {

return Err(MMUEnableError::AlreadyEnabled);

}

if unlikely(!ID\_AA64MMFR0\_EL1.matches\_all(ID\_AA64MMFR0\_EL1::TGran64::Supported)) {

return Err(MMUEnableError::Other(

"Translation granule not supported in HW",

));

}

self.set\_up\_mair();

TTBR0\_EL1.set\_baddr(phys\_tables\_base\_addr.as\_usize() as u64);

self.configure\_translation\_control();

barrier::isb(barrier::SY);

SCTLR\_EL1.modify(SCTLR\_EL1::M::Enable + SCTLR\_EL1::C::Cacheable + SCTLR\_EL1::I::Cacheable);

barrier::isb(barrier::SY);

Ok(())

}

#[inline(always)]

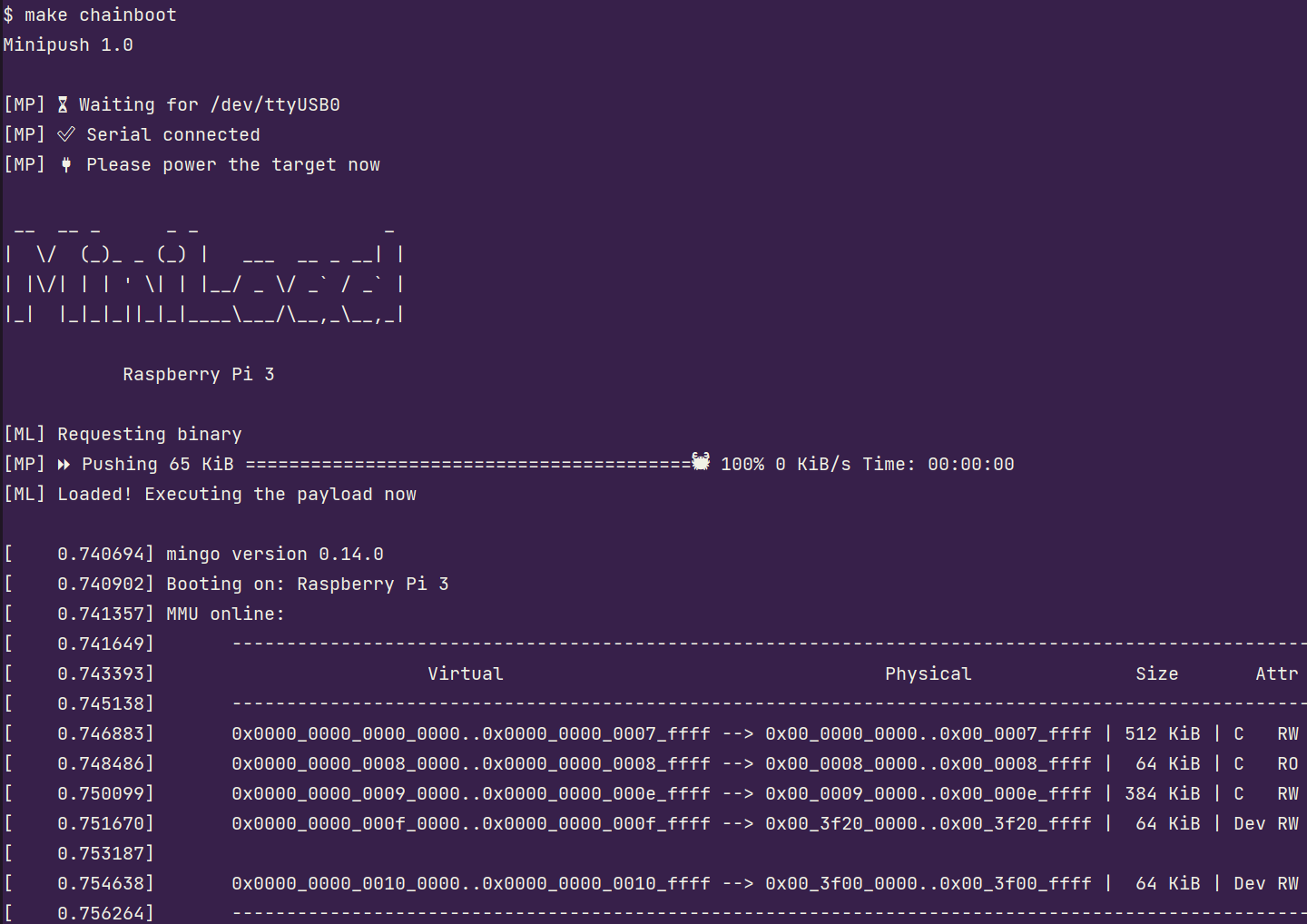
fn is\_enabled(&self) -> bool {

SCTLR\_EL1.matches\_all(SCTLR\_EL1::M::Enable)

}

}

Результат:



Етап 15:

На цьому етапі відбуваються наступні дії:

1. Попередній обрахунок таблиць (precomputed tables): Використовуючи попередньо обчислені таблиці, які залежать від архітектури пристрою, вирішуються питання, пов'язані зі сторінковим перекладом адрес (paging). Це дозволяє ефективно виконувати операції з віртуальною пам'яттю, зменшуючи затрати на обчислення в процесі виконання програми.
2. Побудова таблиць сторінок: За допомогою попередньо обрахованих таблиць, будуються таблиці сторінок, які використовуються для відображення віртуальних адресів на фізичну пам'ять. Це дозволяє ефективно виконувати переклад адрес і забезпечує відповідність між віртуальними і фізичними адресами.
3. Вдосконалення використання пам'яті: Застосовуються оптимізації для покращення використання пам'яті, зокрема використання кеш-пам'яті для швидкого доступу до таблиць сторінок та підтримка розріджених таблиць сторінок для ефективного використання пам'яті.

В цілому, на цьому етапі проводиться оптимізація використання віртуальної пам'яті шляхом попереднього обрахунку таблиць та вдосконалення використання пам'яті. Це дозволяє покращити продуктивність і ефективність ОС на Raspberry Pi, зменшуючи затрати на обчислення та використання пам'яті.

Реалізація:

Файл translation\_table.rs

use crate::{

bsp,

memory::{

self,

mmu::{

arch\_mmu::{Granule512MiB, Granule64KiB},

AccessPermissions, AttributeFields, MemAttributes, MemoryRegion, PageAddress,

},

Address, Physical, Virtual,

},

};

use core::convert;

use tock\_registers::{

interfaces::{Readable, Writeable},

register\_bitfields,

registers::InMemoryRegister,

};

register\_bitfields! {u64,

STAGE1\_TABLE\_DESCRIPTOR [

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

]

]

}

register\_bitfields! {u64,

STAGE1\_PAGE\_DESCRIPTOR [

UXN OFFSET(54) NUMBITS(1) [

False = 0,

True = 1

],

PXN OFFSET(53) NUMBITS(1) [

False = 0,

True = 1

],

OUTPUT\_ADDR\_64KiB OFFSET(16) NUMBITS(32) [], // [47:16]

AF OFFSET(10) NUMBITS(1) [

False = 0,

True = 1

],

SH OFFSET(8) NUMBITS(2) [

OuterShareable = 0b10,

InnerShareable = 0b11

],

AP OFFSET(6) NUMBITS(2) [

RW\_EL1 = 0b00,

RW\_EL1\_EL0 = 0b01,

RO\_EL1 = 0b10,

RO\_EL1\_EL0 = 0b11

],

AttrIndx OFFSET(2) NUMBITS(3) [],

TYPE OFFSET(1) NUMBITS(1) [

Reserved\_Invalid = 0,

Page = 1

],

VALID OFFSET(0) NUMBITS(1) [

False = 0,

True = 1

]

]

}

#[derive(Copy, Clone)]

#[repr(C)]

struct TableDescriptor {

value: u64,

}

#[derive(Copy, Clone)]

#[repr(C)]

struct PageDescriptor {

value: u64,

}

trait StartAddr {

fn virt\_start\_addr(&self) -> Address<Virtual>;

}

#[repr(C)]

#[repr(align(65536))]

pub struct FixedSizeTranslationTable<const NUM\_TABLES: usize> {

lvl3: [[PageDescriptor; 8192]; NUM\_TABLES],

lvl2: [TableDescriptor; NUM\_TABLES],

initialized: bool,

}

impl<T, const N: usize> StartAddr for [T; N] {

fn virt\_start\_addr(&self) -> Address<Virtual> {

Address::new(self as \*const \_ as usize)

}

}

impl TableDescriptor {

pub const fn new\_zeroed() -> Self {

Self { value: 0 }

}

pub fn from\_next\_lvl\_table\_addr(phys\_next\_lvl\_table\_addr: Address<Physical>) -> Self {

let val = InMemoryRegister::<u64, STAGE1\_TABLE\_DESCRIPTOR::Register>::new(0);

let shifted = phys\_next\_lvl\_table\_addr.as\_usize() >> Granule64KiB::SHIFT;

val.write(

STAGE1\_TABLE\_DESCRIPTOR::NEXT\_LEVEL\_TABLE\_ADDR\_64KiB.val(shifted as u64)

+ STAGE1\_TABLE\_DESCRIPTOR::TYPE::Table

+ STAGE1\_TABLE\_DESCRIPTOR::VALID::True,

);

TableDescriptor { value: val.get() }

}

}

impl convert::From<AttributeFields>

for tock\_registers::fields::FieldValue<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>

{

fn from(attribute\_fields: AttributeFields) -> Self {

let mut desc = match attribute\_fields.mem\_attributes {

MemAttributes::CacheableDRAM => {

STAGE1\_PAGE\_DESCRIPTOR::SH::InnerShareable

+ STAGE1\_PAGE\_DESCRIPTOR::AttrIndx.val(memory::mmu::arch\_mmu::mair::NORMAL)

}

MemAttributes::Device => {

STAGE1\_PAGE\_DESCRIPTOR::SH::OuterShareable

+ STAGE1\_PAGE\_DESCRIPTOR::AttrIndx.val(memory::mmu::arch\_mmu::mair::DEVICE)

}

};

desc += match attribute\_fields.acc\_perms {

AccessPermissions::ReadOnly => STAGE1\_PAGE\_DESCRIPTOR::AP::RO\_EL1,

AccessPermissions::ReadWrite => STAGE1\_PAGE\_DESCRIPTOR::AP::RW\_EL1,

};

desc += if attribute\_fields.execute\_never {

STAGE1\_PAGE\_DESCRIPTOR::PXN::True

} else {

STAGE1\_PAGE\_DESCRIPTOR::PXN::False

};

desc += STAGE1\_PAGE\_DESCRIPTOR::UXN::True;

desc

}

}

impl convert::TryFrom<InMemoryRegister<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>> for AttributeFields {

type Error = &'static str;

fn try\_from(

desc: InMemoryRegister<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>,

) -> Result<AttributeFields, Self::Error> {

let mem\_attributes = match desc.read(STAGE1\_PAGE\_DESCRIPTOR::AttrIndx) {

memory::mmu::arch\_mmu::mair::NORMAL => MemAttributes::CacheableDRAM,

memory::mmu::arch\_mmu::mair::DEVICE => MemAttributes::Device,

\_ => return Err("Unexpected memory attribute"),

};

let acc\_perms = match desc.read\_as\_enum(STAGE1\_PAGE\_DESCRIPTOR::AP) {

Some(STAGE1\_PAGE\_DESCRIPTOR::AP::Value::RO\_EL1) => AccessPermissions::ReadOnly,

Some(STAGE1\_PAGE\_DESCRIPTOR::AP::Value::RW\_EL1) => AccessPermissions::ReadWrite,

\_ => return Err("Unexpected access permission"),

};

let execute\_never = desc.read(STAGE1\_PAGE\_DESCRIPTOR::PXN) > 0;

Ok(AttributeFields {

mem\_attributes,

acc\_perms,

execute\_never,

})

}

}

impl PageDescriptor {

pub const fn new\_zeroed() -> Self {

Self { value: 0 }

}

pub fn from\_output\_page\_addr(

phys\_output\_page\_addr: PageAddress<Physical>,

attribute\_fields: &AttributeFields,

) -> Self {

let val = InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(0);

let shifted = phys\_output\_page\_addr.into\_inner().as\_usize() >> Granule64KiB::SHIFT;

val.write(

STAGE1\_PAGE\_DESCRIPTOR::OUTPUT\_ADDR\_64KiB.val(shifted as u64)

+ STAGE1\_PAGE\_DESCRIPTOR::AF::True

+ STAGE1\_PAGE\_DESCRIPTOR::TYPE::Page

+ STAGE1\_PAGE\_DESCRIPTOR::VALID::True

+ (\*attribute\_fields).into(),

);

Self { value: val.get() }

}

fn is\_valid(&self) -> bool {

InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(self.value)

.is\_set(STAGE1\_PAGE\_DESCRIPTOR::VALID)

}

fn output\_page\_addr(&self) -> PageAddress<Physical> {

let shifted = InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(self.value)

.read(STAGE1\_PAGE\_DESCRIPTOR::OUTPUT\_ADDR\_64KiB) as usize;

PageAddress::from(shifted << Granule64KiB::SHIFT)

}

fn try\_attributes(&self) -> Result<AttributeFields, &'static str> {

InMemoryRegister::<u64, STAGE1\_PAGE\_DESCRIPTOR::Register>::new(self.value).try\_into()

}

}

impl<const AS\_SIZE: usize> memory::mmu::AssociatedTranslationTable

for memory::mmu::AddressSpace<AS\_SIZE>

where

[u8; Self::SIZE >> Granule512MiB::SHIFT]: Sized,

{

type TableStartFromBottom = FixedSizeTranslationTable<{ Self::SIZE >> Granule512MiB::SHIFT }>;

}

impl<const NUM\_TABLES: usize> FixedSizeTranslationTable<NUM\_TABLES> {

#[allow(clippy::assertions\_on\_constants)]

const fn \_new(for\_precompute: bool) -> Self {

assert!(bsp::memory::mmu::KernelGranule::SIZE == Granule64KiB::SIZE);

assert!(NUM\_TABLES > 0);

Self {

lvl3: [[PageDescriptor::new\_zeroed(); 8192]; NUM\_TABLES],

lvl2: [TableDescriptor::new\_zeroed(); NUM\_TABLES],

initialized: for\_precompute,

}

}

pub const fn new\_for\_precompute() -> Self {

Self::\_new(true)

}

#[cfg(test)]

pub fn new\_for\_runtime() -> Self {

Self::\_new(false)

}

#[inline(always)]

fn lvl2\_lvl3\_index\_from\_page\_addr(

&self,

virt\_page\_addr: PageAddress<Virtual>,

) -> Result<(usize, usize), &'static str> {

let addr = virt\_page\_addr.into\_inner().as\_usize();

let lvl2\_index = addr >> Granule512MiB::SHIFT;

let lvl3\_index = (addr & Granule512MiB::MASK) >> Granule64KiB::SHIFT;

if lvl2\_index > (NUM\_TABLES - 1) {

return Err("Virtual page is out of bounds of translation table");

}

Ok((lvl2\_index, lvl3\_index))

}

#[inline(always)]

fn page\_descriptor\_from\_page\_addr(

&self,

virt\_page\_addr: PageAddress<Virtual>,

) -> Result<&PageDescriptor, &'static str> {

let (lvl2\_index, lvl3\_index) = self.lvl2\_lvl3\_index\_from\_page\_addr(virt\_page\_addr)?;

let desc = &self.lvl3[lvl2\_index][lvl3\_index];

Ok(desc)

}

#[inline(always)]

fn set\_page\_descriptor\_from\_page\_addr(

&mut self,

virt\_page\_addr: PageAddress<Virtual>,

new\_desc: &PageDescriptor,

) -> Result<(), &'static str> {

let (lvl2\_index, lvl3\_index) = self.lvl2\_lvl3\_index\_from\_page\_addr(virt\_page\_addr)?;

let desc = &mut self.lvl3[lvl2\_index][lvl3\_index];

if desc.is\_valid() {

return Err("Virtual page is already mapped");

}

\*desc = \*new\_desc;

Ok(())

}

}

impl<const NUM\_TABLES: usize> memory::mmu::translation\_table::interface::TranslationTable

for FixedSizeTranslationTable<NUM\_TABLES>

{

fn init(&mut self) -> Result<(), &'static str> {

if self.initialized {

return Ok(());

}

for (lvl2\_nr, lvl2\_entry) in self.lvl2.iter\_mut().enumerate() {

let virt\_table\_addr = self.lvl3[lvl2\_nr].virt\_start\_addr();

let phys\_table\_addr = memory::mmu::try\_kernel\_virt\_addr\_to\_phys\_addr(virt\_table\_addr)?;

let new\_desc = TableDescriptor::from\_next\_lvl\_table\_addr(phys\_table\_addr);

\*lvl2\_entry = new\_desc;

}

self.initialized = true;

Ok(())

}

unsafe fn map\_at(

&mut self,

virt\_region: &MemoryRegion<Virtual>,

phys\_region: &MemoryRegion<Physical>,

attr: &AttributeFields,

) -> Result<(), &'static str> {

assert!(self.initialized, "Translation tables not initialized");

if virt\_region.size() != phys\_region.size() {

return Err("Tried to map memory regions with unequal sizes");

}

if phys\_region.end\_exclusive\_page\_addr() > bsp::memory::phys\_addr\_space\_end\_exclusive\_addr()

{

return Err("Tried to map outside of physical address space");

}

let iter = phys\_region.into\_iter().zip(virt\_region.into\_iter());

for (phys\_page\_addr, virt\_page\_addr) in iter {

let new\_desc = PageDescriptor::from\_output\_page\_addr(phys\_page\_addr, attr);

let virt\_page = virt\_page\_addr;

self.set\_page\_descriptor\_from\_page\_addr(virt\_page, &new\_desc)?;

}

Ok(())

}

fn try\_virt\_page\_addr\_to\_phys\_page\_addr(

&self,

virt\_page\_addr: PageAddress<Virtual>,

) -> Result<PageAddress<Physical>, &'static str> {

let page\_desc = self.page\_descriptor\_from\_page\_addr(virt\_page\_addr)?;

if !page\_desc.is\_valid() {

return Err("Page marked invalid");

}

Ok(page\_desc.output\_page\_addr())

}

fn try\_page\_attributes(

&self,

virt\_page\_addr: PageAddress<Virtual>,

) -> Result<AttributeFields, &'static str> {

let page\_desc = self.page\_descriptor\_from\_page\_addr(virt\_page\_addr)?;

if !page\_desc.is\_valid() {

return Err("Page marked invalid");

}

page\_desc.try\_attributes()

}

fn try\_virt\_addr\_to\_phys\_addr(

&self,

virt\_addr: Address<Virtual>,

) -> Result<Address<Physical>, &'static str> {

let virt\_page = PageAddress::from(virt\_addr.align\_down\_page());

let phys\_page = self.try\_virt\_page\_addr\_to\_phys\_page\_addr(virt\_page)?;

Ok(phys\_page.into\_inner() + virt\_addr.offset\_into\_page())

}

}

#[cfg(test)]

pub type MinSizeTranslationTable = FixedSizeTranslationTable<1>;

#[cfg(test)]

mod tests {

use super::\*;

use test\_macros::kernel\_test;

#[kernel\_test]

fn size\_of\_tabledescriptor\_equals\_64\_bit() {

assert\_eq!(

core::mem::size\_of::<TableDescriptor>(),

core::mem::size\_of::<u64>()

);

}

#[kernel\_test]

fn size\_of\_pagedescriptor\_equals\_64\_bit() {

assert\_eq!(

core::mem::size\_of::<PageDescriptor>(),

core::mem::size\_of::<u64>()

);

}

}

Файл mmu.rs

use crate::{

bsp, memory,

memory::{mmu::TranslationGranule, Address, Physical},

};

use aarch64\_cpu::{asm::barrier, registers::\*};

use core::intrinsics::unlikely;

use tock\_registers::interfaces::{ReadWriteable, Readable, Writeable};

struct MemoryManagementUnit;

pub type Granule512MiB = TranslationGranule<{ 512 \* 1024 \* 1024 }>;

pub type Granule64KiB = TranslationGranule<{ 64 \* 1024 }>;

#[allow(dead\_code)]

pub mod mair {

pub const DEVICE: u64 = 0;

pub const NORMAL: u64 = 1;

}

static MMU: MemoryManagementUnit = MemoryManagementUnit;

impl<const AS\_SIZE: usize> memory::mmu::AddressSpace<AS\_SIZE> {

pub const fn arch\_address\_space\_size\_sanity\_checks() {

assert!((AS\_SIZE % Granule512MiB::SIZE) == 0);

assert!(AS\_SIZE <= (1 << 48));

}

}

impl MemoryManagementUnit {

fn set\_up\_mair(&self) {

MAIR\_EL1.write(

MAIR\_EL1::Attr1\_Normal\_Outer::WriteBack\_NonTransient\_ReadWriteAlloc +

MAIR\_EL1::Attr1\_Normal\_Inner::WriteBack\_NonTransient\_ReadWriteAlloc +

// Attribute 0 - Device.

MAIR\_EL1::Attr0\_Device::nonGathering\_nonReordering\_EarlyWriteAck,

);

}

fn configure\_translation\_control(&self) {

let t0sz = (64 - bsp::memory::mmu::KernelVirtAddrSpace::SIZE\_SHIFT) as u64;

TCR\_EL1.write(

TCR\_EL1::TBI0::Used

+ TCR\_EL1::IPS::Bits\_40

+ 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::A1::TTBR0

+ TCR\_EL1::T0SZ.val(t0sz)

+ TCR\_EL1::EPD1::DisableTTBR1Walks,

);

}

}

pub fn mmu() -> &'static impl memory::mmu::interface::MMU {

&MMU

}

use memory::mmu::MMUEnableError;

impl memory::mmu::interface::MMU for MemoryManagementUnit {

unsafe fn enable\_mmu\_and\_caching(

&self,

phys\_tables\_base\_addr: Address<Physical>,

) -> Result<(), MMUEnableError> {

if unlikely(self.is\_enabled()) {

return Err(MMUEnableError::AlreadyEnabled);

}

if unlikely(!ID\_AA64MMFR0\_EL1.matches\_all(ID\_AA64MMFR0\_EL1::TGran64::Supported)) {

return Err(MMUEnableError::Other(

"Translation granule not supported in HW",

));

}

self.set\_up\_mair();

TTBR0\_EL1.set\_baddr(phys\_tables\_base\_addr.as\_usize() as u64);

self.configure\_translation\_control();

barrier::isb(barrier::SY);

SCTLR\_EL1.modify(SCTLR\_EL1::M::Enable + SCTLR\_EL1::C::Cacheable + SCTLR\_EL1::I::Cacheable);

barrier::isb(barrier::SY);

Ok(())

}

#[inline(always)]

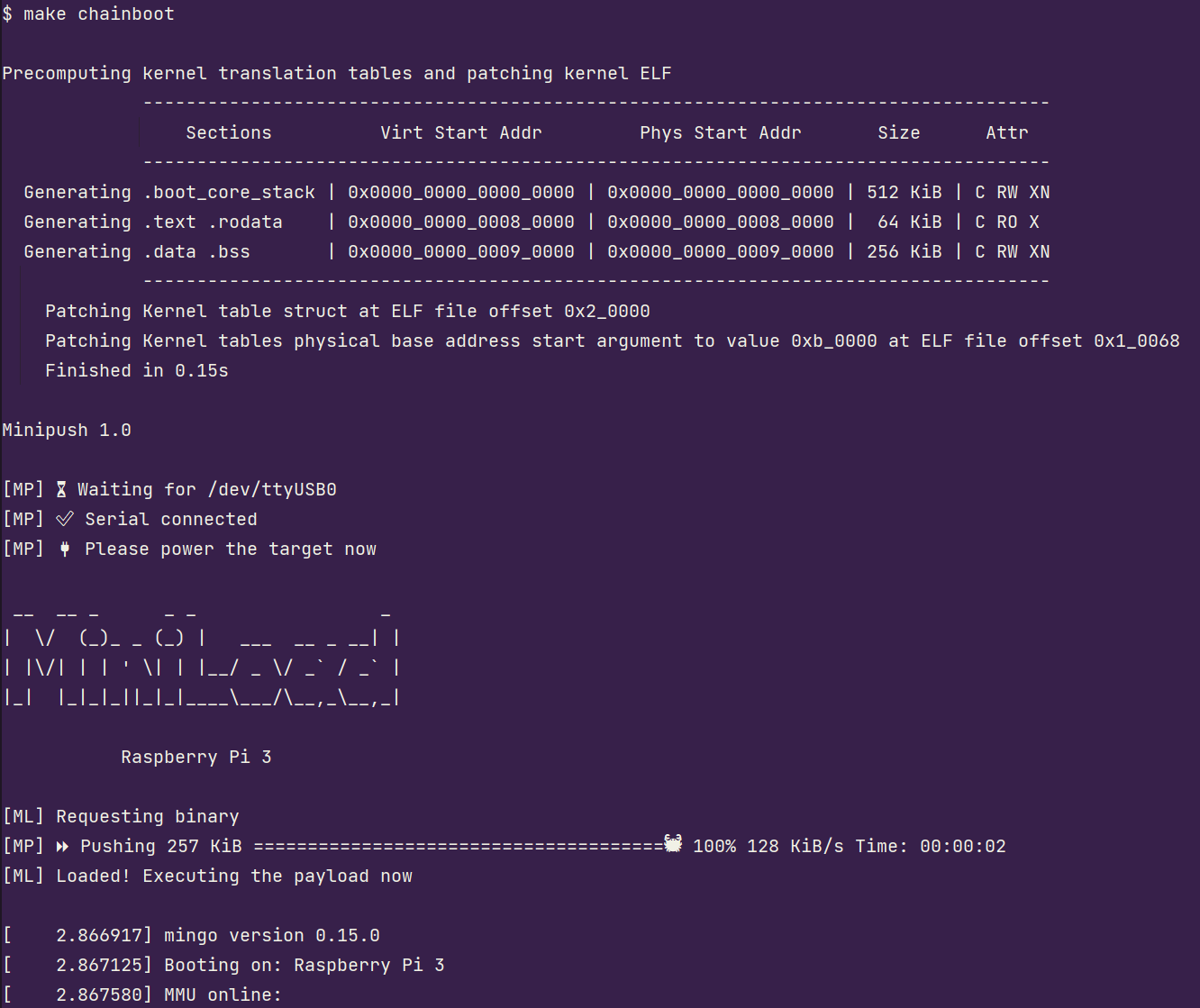
fn is\_enabled(&self) -> bool {

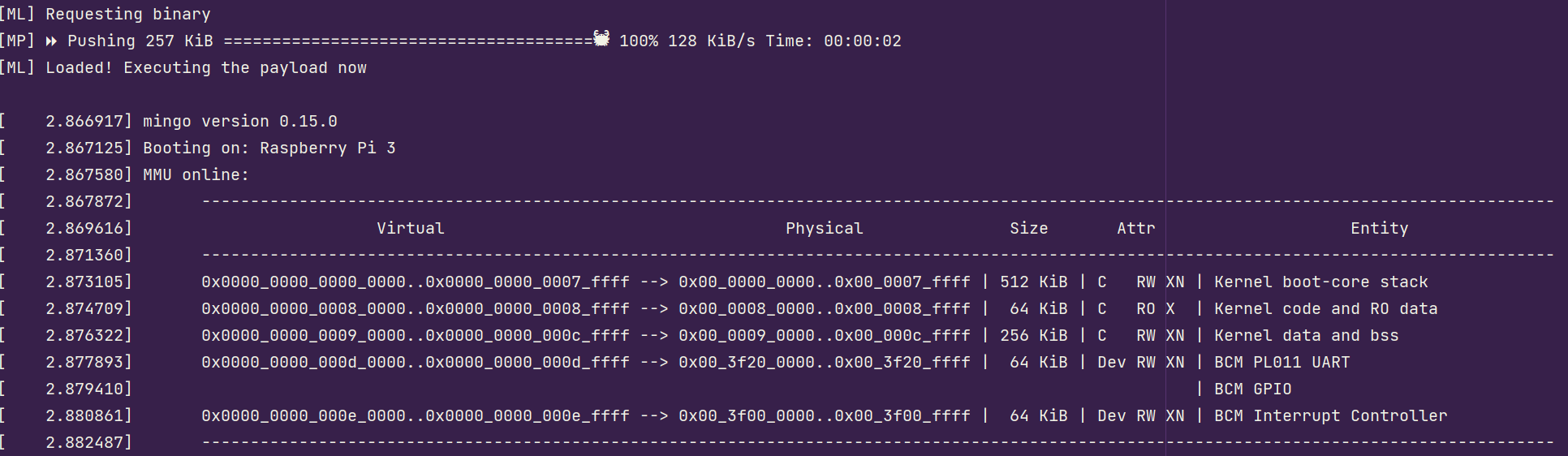
SCTLR\_EL1.matches\_all(SCTLR\_EL1::M::Enable)

}

}

Результат:





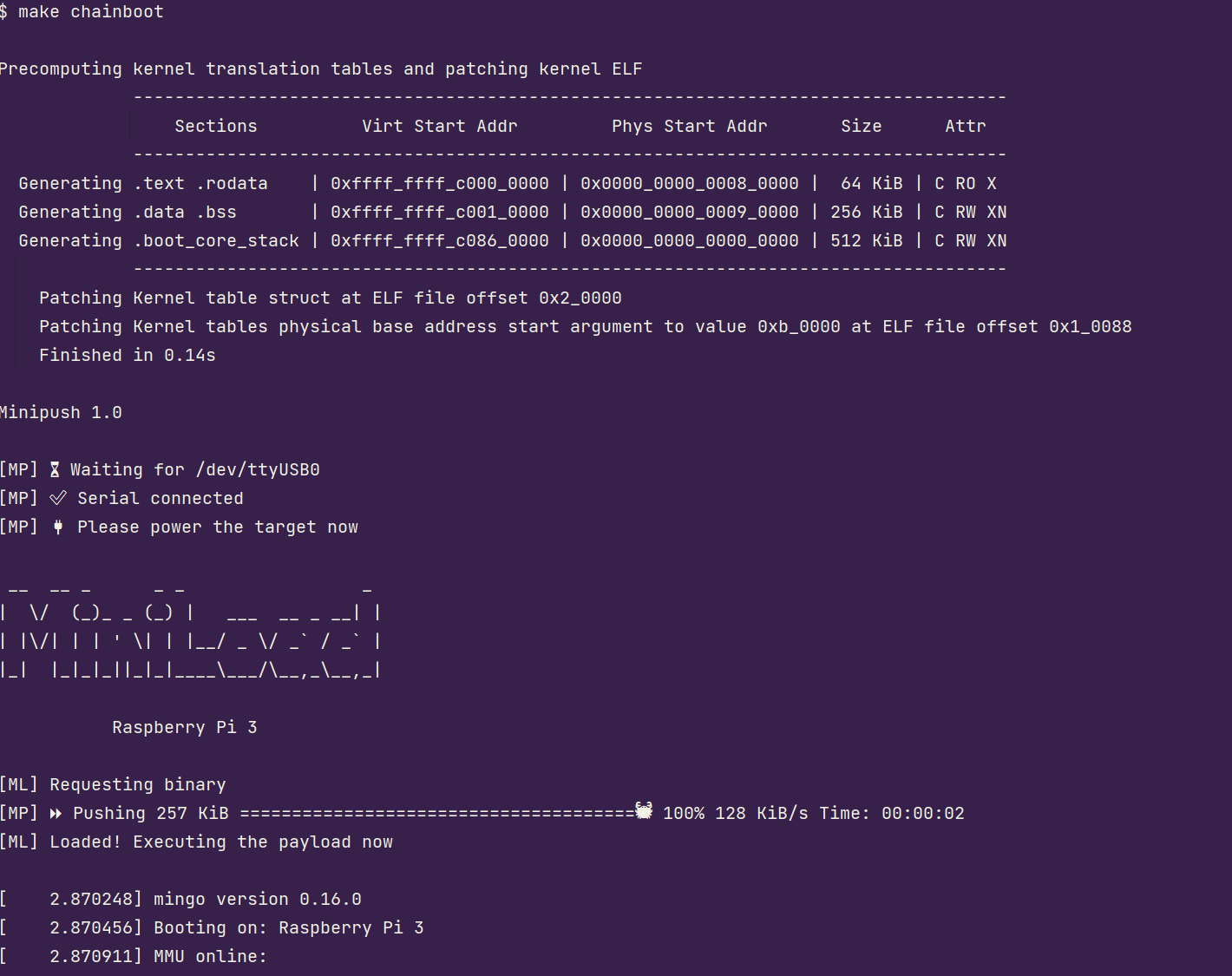
Етап 16:

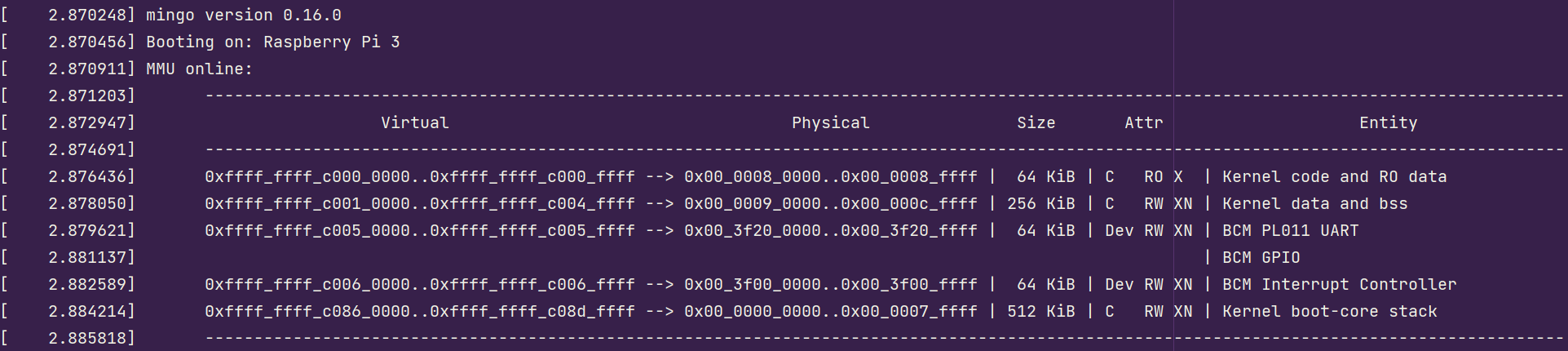
На цьому етапі відбуваються наступні дії:

1. Розташування ядра у вищій половині віртуального адресного простору: Ядро операційної системи розташовується у вищій половині віртуального адресного простору, що дозволяє відокремити його від користувацьких процесів. Це забезпечує захист і безпеку системи, оскільки доступ до вищої половини адресного простору може бути обмежений лише для привілейованих операцій.
2. Налаштування сторінкового перекладу: Використовуючи вищу половину віртуального адресного простору для ядра, потрібно відповідним чином налаштувати сторінковий переклад адрес. Це означає налаштування таблиць сторінок та відображення фізичних адрес на вищу половину віртуального адресного простору.
3. Забезпечення безпеки та ізоляції: Використання вищої половини віртуального адресного простору для ядра допомагає забезпечити безпеку та ізоляцію від користувацьких процесів. Це дозволяє обмежити доступ користувачів до критичних системних ресурсів і даних, зменшуючи потенційні загрози безпеці.

На даному етапі основна увага зосереджена на налаштуванні вищої половини віртуального адресного простору ядра операційної системи для забезпечення безпеки, ізоляції та оптимального розташування ядра у пам'яті Raspberry Pi.

Результат:





**Висновок:**

Під час розгляду кроків 10 до 16 у проектах "rust-raspberrypi-OS-tutorials" було реалізовано операційні системи для плат Raspberry Pi 3 та Raspberry Pi 4. Під час цих кроків були розглянуті теми віртуальної пам'яті, обробки винятків та інтегрованого тестування.

Одна з головних відмінностей реалізації для Raspberry Pi 3 та Raspberry Pi 4 полягає у рівні підтримки та архітектурних особливостях цих плат. Raspberry Pi 3 має процесор ARM Cortex-A53 з 32-бітною архітектурою ARMv8-A, тоді як Raspberry Pi 4 має процесор ARM Cortex-A72 з 64-бітною архітектурою ARMv8-A. Це вимагало адаптації коду та використання різних функцій та реєстрів, щоб забезпечити сумісність та оптимальну роботу на кожній платі.

Кроки 10 та 11 відповідають за встановлення вихідних основних функцій для розробки операційних систем. У кроці 12 реалізоване інтегроване тестування для перевірки правильності роботи системи.

Кроки 13 та 14 стосуються обробки винятків та переривань, зокрема для зовнішніх пристроїв та периферійних ІС. Це важливо для забезпечення взаємодії з різними пристроями та обробки непередбачуваних ситуацій.

У кроці 15 реалізована віртуальна пам'ять з попередньо обрахованими таблицями перекладу. Це дозволяє використовувати оптимізований механізм перекладу адресів, що покращує продуктивність та швидкість доступу до пам'яті.

У кроці 16 реалізована підтримка вищих половин ядра, що дозволяє використовувати великі адресні простори та підтримувати багатозадачність.

Загалом, реалізація операційних систем для Raspberry Pi 3 та Raspberry Pi 4 вимагає адаптації коду та використання специфічних функцій та реєстрів для кожної плати, що дозволяє досягти оптимальної роботи на кожній з них.