МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій

Кафедра комп'ютерної інженерії та інформаційних систем

ЛАБОРАТОРНА РОБОТА №1

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. Створення макету проекту.

**Завдання:**

Реалізувати етапи 1-4 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 1.

На цьому етапі створюється базова структура проєкту, що буде використовуватися для розробки операційної системи.

Далі запускається невеликий фрагмент коду збірки, який виконується під час процесу збірки проєкту. Цей фрагмент коду використовує інструкцію WFE (Wait For Event), яка зупиняє виконання процесора до тих пір, поки не станеться якась подія. Після цього виконується інструкція B (Branch), яка переходить на початок проєкту з адресою 0x80000. Оскільки на цьому етапі ще немає жодного коду, який би виконувався, процесор буде зупинений на інструкції WFE, тобто всі ядра процесора будуть зупинені.

Основні команди при написанні коду:

* doc: створює документацію проєкту. Результат може зберігатися у вигляді HTML-файлів або іншого формату, залежно від налаштувань проєкту.
* qemu: запускає ядро проєкту в емуляторі QEMU. Це дозволяє перевірити, як буде працювати ядро на реальному обладнанні.
* clippy: запускає інструмент Clippy, який допомагає знайти можливі проблеми та помилки у коді.
* clean: видаляє всі файли, що були згенеровані в процесі збірки проєкту, тобто очищує середовище від результатів попередніх збірок.
* readelf: дозволяє переглянути вміст ELF-файлу, що був згенерований в результаті збірки проєкту.
* objdump: дозволяє переглянути збірку проєкту в шістнадцятковому форматі.
* nm: дозволяє переглянути символи, що були створені під час збірки проєкту.

Код реалізації:

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -d in\_asm -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -d in\_asm -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

# DOCKER\_IMAGE defined in include file (see top of this file).

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

.PHONY: all doc qemu clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

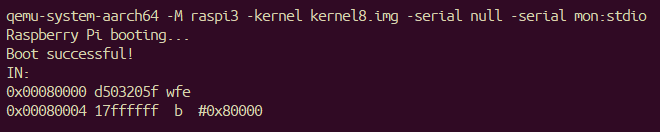
$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

Результат:



Етап 2.

На цьому етапі розширюється функціонал boot.s для першого виклику коду на мові програмування Rust. Спочатку виконується невелика робота по ініціалізації під час виконання перед тим, як здійснити перехід до Rust. Викликаний код Rust зупиняє виконання за допомогою виклику panic!(). Для цього було внесено зміни до початкового коду, додавши розділи .rodata, .got, .data, .bss та спеціальне місце для зв’язування аргументів часу завантаження, які потрібно прочитати \_start().

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -d in\_asm -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -d in\_asm -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

# DOCKER\_IMAGE defined in include file (see top of this file).

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

.PHONY: all doc qemu clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

Файл main.rs

#![feature(asm\_const)]

#![no\_main]

#![no\_std]

mod bsp;

mod cpu;

mod panic\_wait;

unsafe fn kernel\_init() -> ! {

panic!()

}

Файл panic\_wait.rs

use crate::cpu;

use core::panic::PanicInfo;

#[panic\_handler]

fn panic(\_info: &PanicInfo) -> ! {

cpu::wait\_forever()

}

Етап 3.

На цьому етапі в проект додаються глобальні макроси print!(), які дозволяють виводити повідомлення на консоль якнайшвидше. Це важливо для налагодження проекту та пошуку помилок. Функції друку наразі використовують властивість QEMU, що дозволяє користуватися Raspberry UART без налаштування належним чином. Далі, в проекті вводяться важливі доповнення, включаючи інтерфейс Traits для консольних команд, реалізацію інтерфейсу для емульованого QEMU UART, оновлений обробник panic\_handler, який використовує print!() для відображення повідомлень про помилки користувача, а також новий Makefile з цільовим файлом make test, який призначений для автоматичного тестування. Цей файл завантажує скомпільоване ядро в QEMU і перевіряє очікуваний вихідний рядок, згенерований ядром. На цьому етапі перевіряється рядок "Stopping here", який видається panic!() в кінці main.rs.

Код реалізації:

Файл panic\_wait.rs

use crate::{cpu, println};

use core::panic::PanicInfo;

fn panic\_prevent\_reenter() {

use core::sync::atomic::{AtomicBool, Ordering};

#[cfg(not(target\_arch = "aarch64"))]

compile\_error!("Add the target\_arch to above's check if the following code is safe to use");

static PANIC\_IN\_PROGRESS: AtomicBool = AtomicBool::new(false);

if !PANIC\_IN\_PROGRESS.load(Ordering::Relaxed) {

PANIC\_IN\_PROGRESS.store(true, Ordering::Relaxed);

return;

}

cpu::wait\_forever()

}

#[panic\_handler]

fn panic(info: &PanicInfo) -> ! {

// Protect against panic infinite loops if any of the following code panics itself.

panic\_prevent\_reenter();

let (location, line, column) = match info.location() {

Some(loc) => (loc.file(), loc.line(), loc.column()),

\_ => ("???", 0, 0),

};

println!(

"Kernel panic!\n\n\

Panic location:\n File '{}', line {}, column {}\n\n\

{}",

location,

line,

column,

info.message().unwrap\_or(&format\_args!("")),

);

cpu::wait\_forever()

}

Файл print.rs

use crate::console;

use core::fmt;

#[doc(hidden)]

pub fn \_print(args: fmt::Arguments) {

use console::interface::Write;

console::console().write\_fmt(args).unwrap();

}

#[macro\_export]

macro\_rules! print {

($($arg:tt)\*) => ($crate::print::\_print(format\_args!($($arg)\*)));

}

#[macro\_export]

macro\_rules! println {

() => ($crate::print!("\n"));

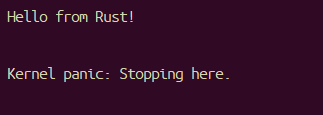
($($arg:tt)\*) => ({

$crate::print::\_print(format\_args\_nl!($($arg)\*));

})

}

Результат:



Етап 4.

На цьому етапі розроблюється механізм безпечного доступу до глобальних структур даних в ядрі операційної системи. Це забезпечується шляхом використання примітивів синхронізації ОС для реалізації псевдоблокування. Розробляється механізм синхронізації, який забезпечує безпечний доступ до глобальних структур даних, включаючи NullLock, що дозволяє демонструвати концепцію внутрішньої мінливості Rust.

Код реалізації:

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

EXEC\_TEST\_DISPATCH = ruby ../common/tests/dispatch.rb

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

DOCKER\_ARG\_DIR\_COMMON = -v $(shell pwd)/../common:/work/common

# DOCKER\_IMAGE defined in include file (see top of this file).

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

DOCKER\_TEST = $(DOCKER\_CMD) $(

.PHONY: all doc qemu clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

--section .rodata \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

.PHONY: test test\_boot

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

test\_boot test:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

test\_boot: $(KERNEL\_BIN)

$(call color\_header, "Boot test - $(BSP)")

@$(DOCKER\_TEST) $(EXEC\_TEST\_DISPATCH) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

test: test\_boot

endif

Файл synchronization.rs

use core::cell::UnsafeCell;

pub mod interface {

pub trait Mutex {

type Data;

fn lock<'a, R>(&'a self, f: impl FnOnce(&'a mut Self::Data) -> R) -> R;

}

}

pub struct NullLock<T>

where

T: ?Sized,

{

data: UnsafeCell<T>,

}

unsafe impl<T> Send for NullLock<T> where T: ?Sized + Send {}

unsafe impl<T> Sync for NullLock<T> where T: ?Sized + Send {}

impl<T> NullLock<T> {

/// Create an instance.

pub const fn new(data: T) -> Self {

Self {

data: UnsafeCell::new(data),

}

}

}

impl<T> interface::Mutex for NullLock<T> {

type Data = T;

fn lock<'a, R>(&'a self, f: impl FnOnce(&'a mut Self::Data) -> R) -> R {

// In a real lock, there would be code encapsulating this line that ensures that this

// mutable reference will ever only be given out once at a time.

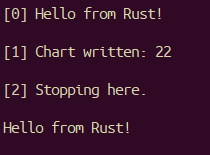
let data = unsafe { &mut \*self.data.get() };

f(data)

}

}

Результат:



ВИСНОВОК

На основі зроблених дій можна зробити наступні висновки:

Реалізація операційної системи для Raspberry Pi 4 відрізняється від реалізації для Raspberry Pi 3 наступним чином:

* Архітектура процесора: Raspberry Pi 4 використовує процесор ARM Cortex-A72, а Raspberry Pi 3 - ARM Cortex-A53. Це означає, що операційна система для Raspberry Pi 4 може використовувати покращений процесор, що дає змогу отримати більшу продуктивність та швидкість роботи.
* Кількість оперативної пам'яті: Raspberry Pi 4 доступний в трьох конфігураціях з різною кількістю оперативної пам'яті - 1 ГБ, 2 ГБ або 4 ГБ. Raspberry Pi 3 має тільки 1 ГБ оперативної пам'яті. Отже, операційна система для Raspberry Pi 4 може працювати з більшим обсягом оперативної пам'яті, що дозволяє запускати більш потужні додатки та більші обсяги даних.
* Версія підключення SD-карти: Raspberry Pi 4 підтримує стандарт SD Card Association версії 3.0, який забезпечує більшу швидкість передачі даних та підтримку більш великих об'ємів пам'яті. Raspberry Pi 3 підтримує стандарт версії 2.0. Отже, операційна система для Raspberry Pi 4 може працювати з більш швидким та потужним засобом зберігання даних.
* Підтримка бездротових технологій: Raspberry Pi 4 має вбудований модуль Wi-Fi та Bluetooth, що дозволяє бездротове підключення до мережі та інших пристроїв. Raspberry Pi 3 не має вбудованого модулю Wi-Fi та Bluetooth, тому для підключення до мережі потрібен окремий адаптер. Отже, операційна система для Raspberry Pi 4 може підтримувати бездротові технології.

Оскільки апаратне забезпечення різних версій Raspberry Pi різні, то для кожної версії можуть бути необхідні відмінні реалізації певних функцій та драйверів, що використовують це апаратне забезпечення. Наприклад, відео драйвер для Raspberry Pi 4 буде відрізнятися від відео драйвера для Raspberry Pi 3.

Отже, реалізація для Raspberry Pi 4 має додаткові можливості та функції порівняно з Raspberry Pi 3, але також може вимагати відмінну реалізацію деяких функцій через відрізняючеся апаратне забезпечення.