МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №4

по дисциплине: Операционные системы
тема: «Разработка драйвера для ОС Linux (Ubuntu)»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
доц. Островский Алексей Мичеславович
асс. Четвертухин Виктор Романович

Белгород 2024 г.

Цель работы: Изучить основы разработки драйверов для ядра Linux с использованием языка программирования C, включая настройку окружения, создание драйвера и его тестирование.

Условие индивидуального задания: Создать драйвер виртуального устройства /dev/queue, реализующего глобальную очередь для строковых данных. Реализовать команды enqueue, dequeue и peek.

Ход выполнения работы

Большинство современных дистрибутивов, в том числе и Ubuntu, всё ещё плохо ладит с драйверами, написанными на Rust. Есть множество проблем: устаревшая версия ядра Линукс, не поддерживающая Rust; отсутствие необходимых флагов при сборке ядра в стандартном дистрибутиве; ядро компилируется при помощи GCC, в то время как компилятор Rust работает с LLVM; требование отключить версионирование драйверов при активации драйверов на Rust, что может повлечь за собой некорректную работу системы при использовании внешних драйверов дистрибутива и других программ.

Поэтому было принято решение собрать собственное ядро Линукс с поддержкой драйверов на Rust при помощи средств LLVM.

Стоит также отметить, что в лабораторной работе был рассмотрен способ встраивания драйвера, называемый out-of-tree. При таком способе драйверы подключаются в ядро во время его работы. Классический способ использования драйверов - написание их в ядре и дальнейшая компиляция ядра. Именно этим способом мы и будем компилировать наш драйвер, так как нам всё равно понадобится компилировать ядро а также могут возникнуть сложности с подключением способом out-of-tree.

Один из форков уже настроен и готов к работе, в нём исключены ненужные компоненты (что существенно уменьшает время компиляции) а также он поддерживает драйверы Rust. Ссылка на ядро Линукс: https://github.com/jackos/linux

В ядре мы должны активировать наш драйвер через следующее меню:

После чего можно собрать ядро и запустить его:

Сообщение об инициализации нашего драйвера в консоли:

Перейдём к самому драйверу:

shared.h

```
//! Virtual Device Module
use kernel::prelude::*;
use kernel::file::{File, Operations};
use kernel::{miscdev};
use kernel::io_buffer::{IoBufferReader, IoBufferWriter};
use kernel::sync::smutex::Mutex;
use kernel::sync::{Ref, RefBorrow};

module! {
    type: OS4Lab,
    name: b"lab4_os",
    license: b"GPL",
}
```

```rust
struct Device {
    queue: Mutex<Vec<Vec<u8>>>,
    begin_index: Mutex<usize>,
    end_index: Mutex<usize>,
    count: Mutex<usize>
}


struct OS4Lab {
    _dev: Pin<Box<miscdev::Registration<OS4Lab>>>,
}

impl kernel::Module for OS4Lab {
    fn init(_name: &'static CStr, _module: &'static ThisModule) -> Result<Self> {
        pr_info!("--------------------------\n");
        pr_info!("initialize lab4_os module!\n");
        pr_info!("now you can queue strings\n");
        pr_info!("watching for changes...\n");
        pr_info!("--------------------------\n");

        let mut new_queue = Vec::<Vec::<u8>>::new();
        for i in 0..512 {
            new_queue.try_push(b"".try_to_vec()?);
        }

        let reg = miscdev::Registration::new_pinned(
            fmt!("queue"),
            Ref::try_new(Device {
                queue: Mutex::new(new_queue),
                count: Mutex::new(0),
                begin_index: Mutex::new(0),
                end_index: Mutex::new(0)
            })
            .unwrap(),
        )?;
        Ok(Self { _dev: reg })

    }
}

/*

/dev/queue enqueue <value to queue>  - поставить в очередь (write)

/dev/queue dequeue                   - удаляет значение очереди на вершине (write)


*/
#[vtable]
impl Operations for OS4Lab {
    // Тип данных, передаваемый в open
    type OpenData = Ref<Device>;
    // Тип данных, возвращаемый open
    type Data = Ref<Device>;
```

```
    fn open(context: &Ref<Device>, _file: &File) -> Result<Ref<Device>> {
        pr_info!("Queue device was opened\n");
        Ok(context.clone())
    }


//  /dev/queue peek                     - пишет значение на вершине очереди (read)
    fn read(
        data: RefBorrow<'_, Device>,
        _file: &File,
        writer: &mut impl IoBufferWriter,
        offset: u64,
    ) -> Result<usize> {
        pr_info!("File for queue device was read\n");
        let offset = offset.try_into()?;

        // Получить мьютексы для устройства
        // Порядок важен, чтобы не получить Deadlock
        let queue = data.queue.lock();
        let count = data.count.lock();
        let begin_index = data.begin_index.lock();
        let end_index = data.end_index.lock();

        if *count == 0 {
            pr_info!("Queue is empty - nothing to peek!\n");
            return Ok(0);
        }

        let vec = &(*queue)[*begin_index];

        let len = core::cmp::min(writer.len(), vec.len().saturating_sub(offset));
        writer.write_slice(&vec[offset..][..len])?;
        Ok(len)
    }

/*
    /dev/queue enqueue <value to queue>  - поставить в очередь (write)

    /dev/queue dequeue                  - удаляет значение очереди на вершине (write)
*/
    fn write(
        data: RefBorrow<'_, Device>,
        _file: &File,
        reader: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        pr_info!("File for queue device was written\n");
        // Будем считывать все данные без буферизации
        let copy = reader.read_all()?;
        let len = copy.len();

        // Получить мьютексы для устройства.
        // Порядок важен, чтобы не получить Deadlock
```

```
        let mut queue = data.queue.lock();
        let mut count = data.count.lock();
        let mut begin_index = data.begin_index.lock();
        let mut end_index = data.end_index.lock();

        if copy.starts_with(b"enqueue ") {
            pr_info!("Enqueue operation started queue len: {}, count: {}\n", (*queue).len(), *count);
            let copy = copy.strip_prefix(b"enqueue ");

            if (*queue).len() == *count {
                pr_info!("Enqueue overflow\n");
                return Ok(len);
            }

            if let Some(valid_copy) = copy {
                (*queue)[*end_index] = valid_copy.try_to_vec()?;
                *end_index = (*end_index + 1) % (*queue).len();
                *count += 1;
                pr_info!("Enqueue args count: {} begin: {} end: {}\n", *count, *begin_index, *end_index);
            } else {
                pr_info!("Enqueue somehow failed\n");
                return Ok(len);
            }
        } else if copy.starts_with(b"dequeue") {
            pr_info!("Dequeue operation started\n");
            if *count == 0 {
                return Ok(len);
            }

            *begin_index = (*begin_index + 1) % (*queue).len();
            *count -= 1;
        } else {
            return Ok(len);
        }

        Ok(len)
    }
}
```

В нём описываем модуль, девайс а также функции девайса - открыть, прочитать, написать. Контекст устройства в эти методы передаётся при помощи RefBorrow, в этом объекте все данные иммутабельны. Обойти это ограничение позволяет Mutex, который позволяет dereference (разыменовать) наш объект и в дальнейшем взаимодействовать с ним. Вместе с этим происходит и захват мьютекса. Возвращать мьютекс в Rust нельзя - это делается автоматически при выходе из scope, в котором был вызван lock.

Для удобства работы очередь была реализована на статической памяти в 512 элементов.

Для вставки строки в очередь используем write. Строка, идущая после ”enqueue ”будет вставлена в очередь. Для удаления также будем использовать write, для этого в драйвер нужно послать ”dequeue”. peek будет выполняться при попытке считать данные из драйвера - он будет возвращать элемент на вершине очереди.

Результат выполнения программы:

Результат прогона тестов:

Вывод: в ходе лабораторной работы изучили основы разработки драйверов для ядра Linux с использованием языка программирования Rust, включая настройку окружения, создание драйвера и его тестирование. Поддержка драйверов на Rust всё ещё очень сырая. Кроме вышеозвученных причин можно также отметить отсутствие std для модулей написанных на Rust. Большинство стандартных функций просто недоступны, взаимодействие происходит через create kernel, а его функционал крайне узкий (просмотрите, например, документацию к kernel::alloc::kvec::Vec, очень много функций для вектора просто нет). Rust for Linux всё ещё активно развивается и дополняется, однако уже сегодня можно писапть простые и безопасные драйверы на Rust.