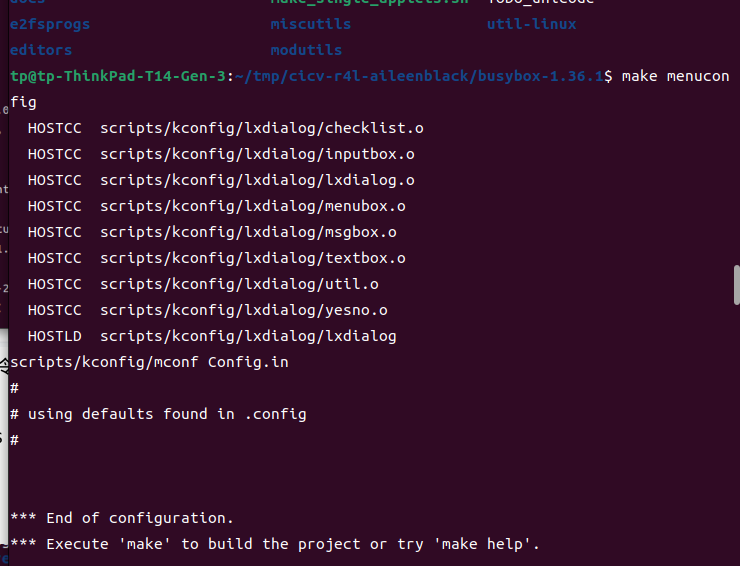
## **作业1：编译Linux内核**

前期环境配置

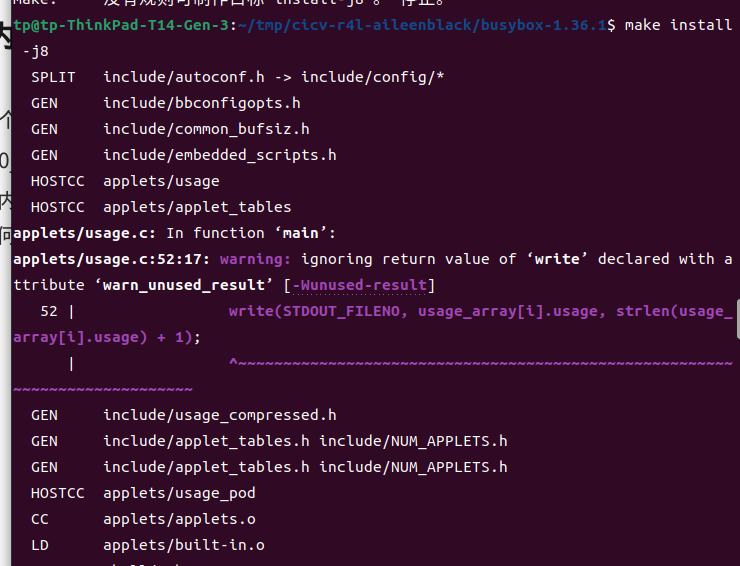
1. 下载对应的工具



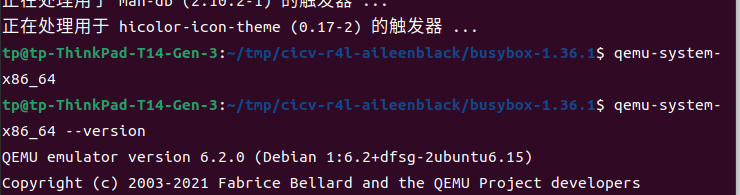
配置busybox config



安装busybox



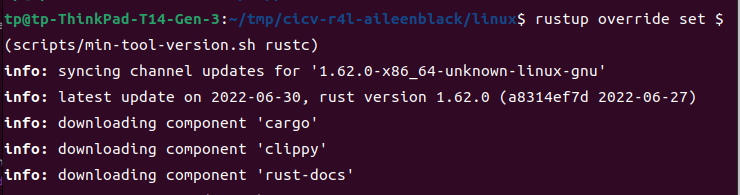
安装qemu



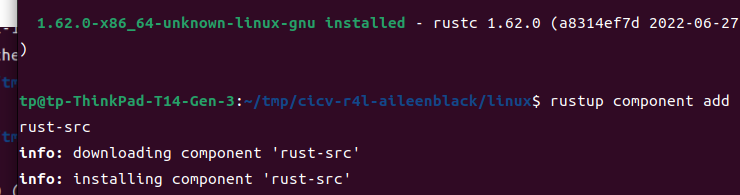
配置linux支持rust

执行

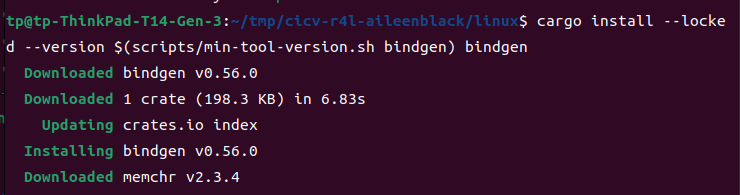
rustup override set $(scripts/min-tool-version.sh rustc)



rustup override set $(scripts/min-tool-version.sh rustc)



cargo install --locked --version $(scripts/min-tool-version.sh bindgen) bindgen



rustup component add rustfmt

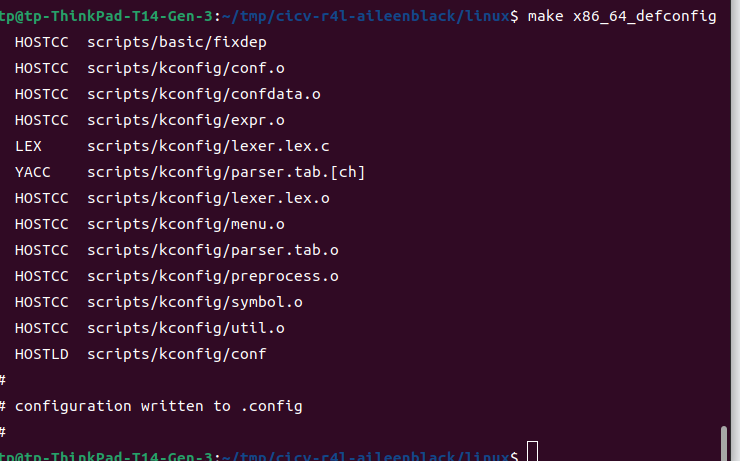
rustup component add clippy

make LLVM=1 rustavailable



进入linux文件夹，执行下面命令

make x86\_64\_defconfig



make LLVM=1 menuconfig

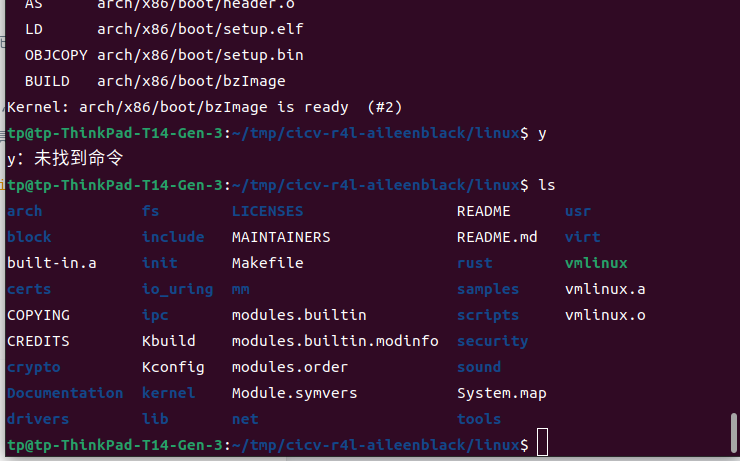
#set the following config to yes

General setup

        ---> [\*] Rust support

make LLVM=1 -j$(nproc)

等待一段时间，可以得到



## **作业2：对Linux内核进行一些配置**

1. 编译成内核模块，是在哪个文件中以哪条语句定义的？

Kbuild中obj-m := r4l\_e1000\_demo.o

2、该模块位于独立的文件夹内，却能编译成Linux内核模块，这叫做out-of-tree module，请分析它是如何与内核代码产生联系的？

1）. 内核符号表：内核符号表是一个重要的数据结构，用于存储内核中定义的变量、函数和其他符号的信息。内核模块通过符号表来与其他模块进行交互。

2）. 模块的编译和链接：内核模块通常是作为独立的源代码文件编写，并通过Makefile进行编译和链接。编译时，模块代码会与内核的符号表进行链接，以便能够访问内核中的符号。

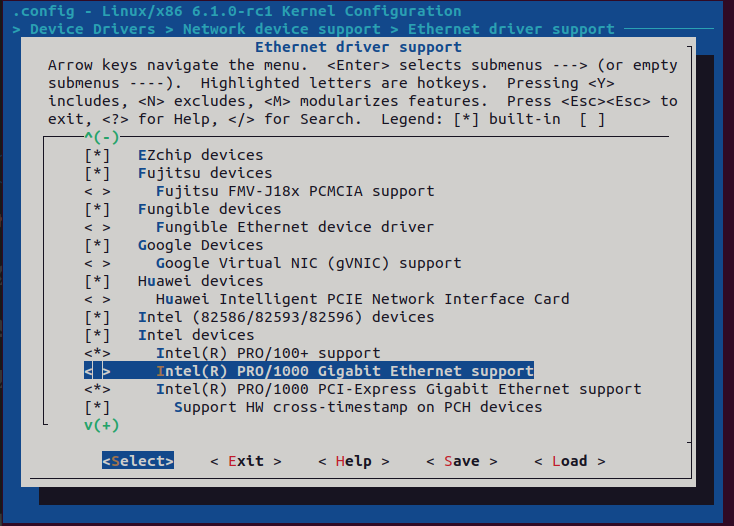
3.） 模块加载和卸载：在运行时，可以使用insmod命令加载一个内核模块，使其成为内核的一部分。加载模块时，内核会分配必要的内存空间，并执行模块的初始化函数。同样，可以使用rmmod命令卸载一个已加载的模块，释放相关的资源。

4）. 模块依赖关系：内核模块可以依赖于其他模块，以便使用其提供的功能。模块管理机制会自动处理模块之间的依赖关系，确保依赖的模块在正确的顺序进行加载和卸载。

禁用网卡

make LLVM =1 menuconfig

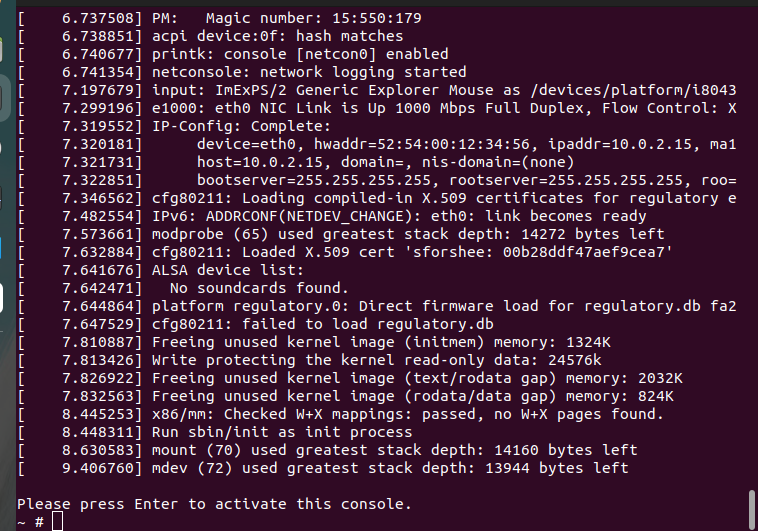
选择Device Drivers > Network device support >Ethernent driver support > Intel Devices Intel(R) PRO/1000 Gigabit Ethernent Support)选择N ，如下图所示：



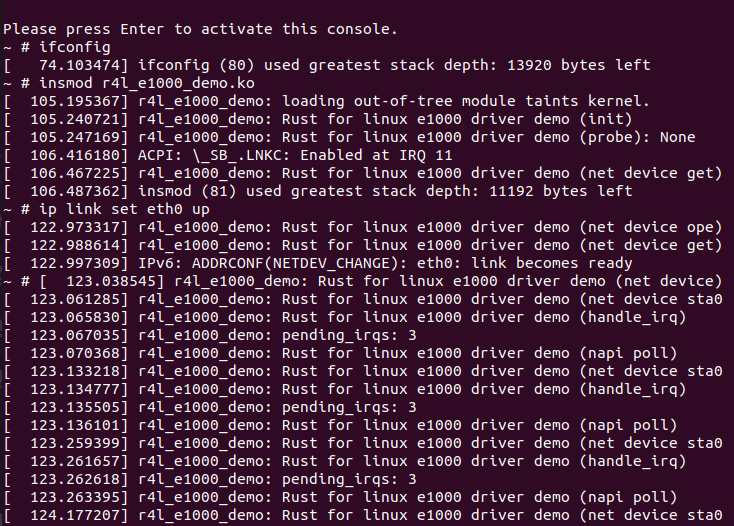
重新编译内核

执行chmod 777 ./build\_image.sh

运行./build\_image.sh



查看ip ifconfig



执行

insmod r4l\_e1000\_demo.ko

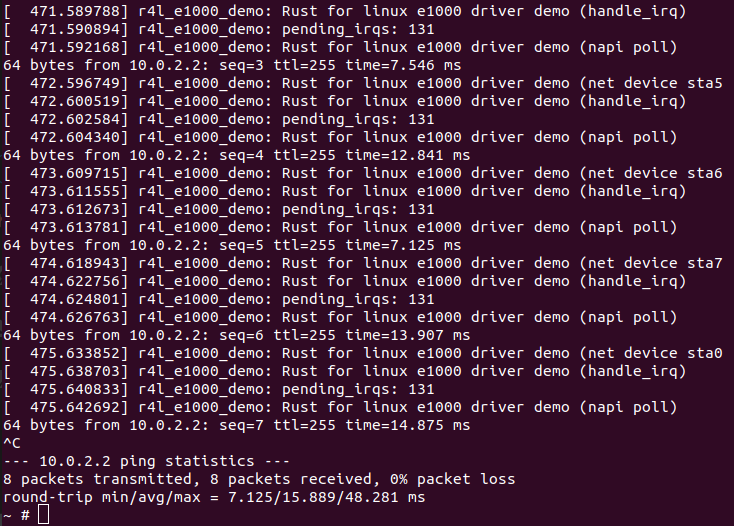
ip link set eth0 up

ip addr add broadcast 10.0.2.255 dev eth0

ip addr add 10.0.2.15/255.255.255.0 dev eth0

ip route add default via 10.0.2.1

ping 10.0.2.2



## **作业3：使用rust编写一个简单的内核模块并运行**

1. 进入源码的samples/rust目录下，根据所给代码添加文件

// SPDX-License-Identifier: GPL-2.0

//! Rust helloworld test sample.

use kernel::prelude::\*;

module!{

type:RustHelloWorld,

name:"rust\_helloworld",

author:"whocare",

description:"hello world module in rust",

license:"GPL",

}

struct RustHelloWorld{}

impl kernel::Module for RustHelloWorld{

fn init(\_name: &'static CStr,\_module: &'static ThisModule)->Result<Self>{

pr\_info!("Hello World from Rust module");

Ok(RustHelloWorld{})

}

}

1. 在KConfig中添加

config SAMPLE\_RUST\_HELLOWORLD

tristate "print helloworld"

help

This option builds the helloworld test cases for Rust.

If unsure, say N.

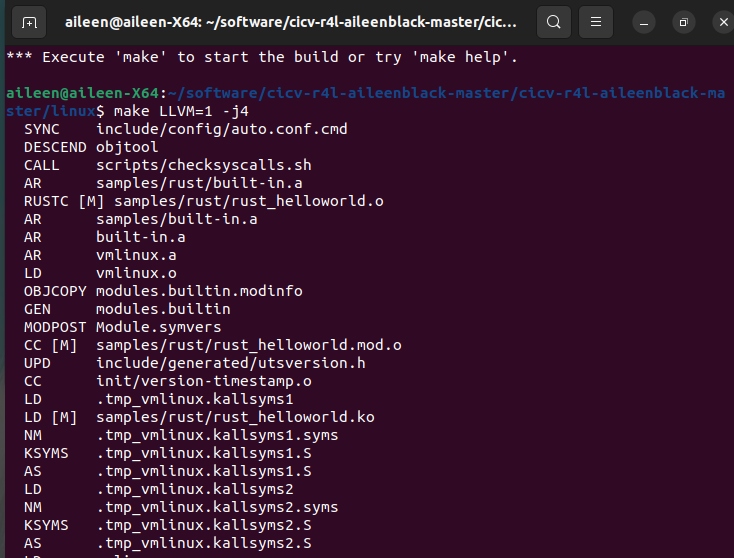
1. 在makefile中添加

obj-$(CONFIG\_SAMPLE\_RUST\_HELLOWORLD) += rust\_helloworld.o

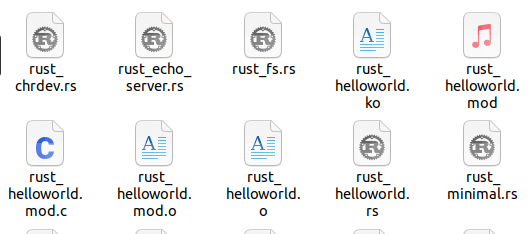
1. 运行make LLVM=1 menuconfig

选择Kernel hacking->Sample Kernel code->Rust Samples -><M> Print Helloworld in Hust Rust(new)

make LLVM=1 -j4



在sample/rust中得到文件

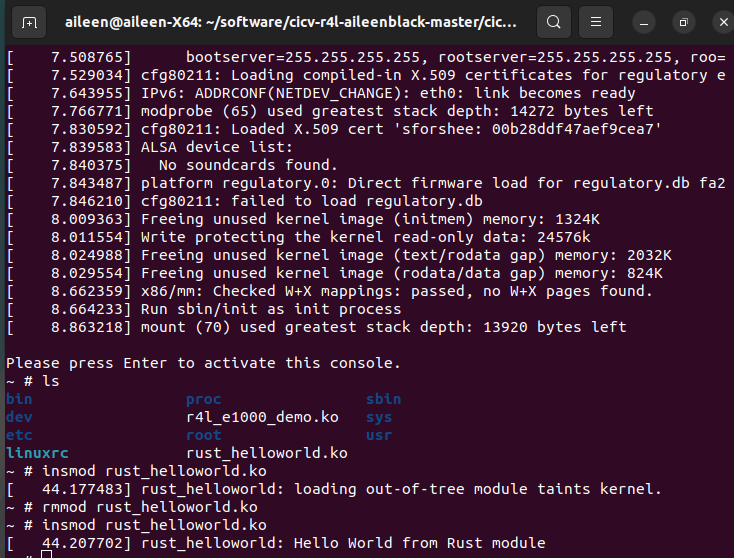


其中ko文件拷贝到src\_e1000/rootfs中，其余拷贝到src\_e1000中，修改src\_e1000的Kbuild文件，添加如下代码

obj-m := rust\_helloworld.o

运行build\_image.sh

加载rust\_helloworl.ko模块



作业4：为e1000网卡驱动添加remove代码

参照2禁用内核网卡驱动

内核pci.rs代码修改

impl Device {

/// Creates a new PCI device from the given pointer.

///

/// # Safety

///

/// `ptr` must be non-null and valid. It must remain valid for the lifetime of the returned

/// instance.

unsafe fn from\_ptr(ptr: \*mut bindings::pci\_dev) -> Self {

Self { ptr }

}

/// enables point-mastering for device

pub fn get\_point(&self)->\*mut bindings::pci\_dev{

self.ptr

}

/// enables bus-mastering for device

pub fn set\_master(&self) {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

unsafe { bindings::pci\_set\_master(self.ptr) };

}

/// get legacy irq number

pub fn irq(&self) -> u32 {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

unsafe { (\*self.ptr).irq }

}

/// Initialize device

pub fn enable\_device(&mut self) -> Result {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

let ret = unsafe { bindings::pci\_enable\_device(self.ptr) };

if ret != 0 {

Err(Error::from\_kernel\_errno(ret))

} else {

Ok(())

}

}

/// iter PCI Resouces

pub fn iter\_resource(&self) -> impl Iterator<Item = Resource> + '\_ {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

let pdev = unsafe { &\*self.ptr };

pdev.resource.iter().map(|x| Resource {

start: x.start,

end: x.end,

flags: x.flags,

})

}

/// Return BAR mask from the type of resource

pub fn select\_bars(&self, flags: u64) -> i32 {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

unsafe { bindings::pci\_select\_bars(self.ptr, flags) }

}

/// Reserve selected PCI I/O and memory resources

pub fn request\_selected\_regions(&mut self, bars: i32, name: &'static CStr) -> Result {

// SAFETY: By the type invariants, we know that `self.ptr` is non-null and valid.

let ret =

unsafe { bindings::pci\_request\_selected\_regions(self.ptr, bars, name.as\_char\_ptr()) };

if ret != 0 {

Err(Error::from\_kernel\_errno(ret))

} else {

Ok(())

}

}

/// Get address for accessing the device

pub fn map\_resource(&self, resource: &Resource, len: usize) -> Result<MappedResource> {

MappedResource::try\_new(resource.start, len)

}

}

对r4l\_e1000\_demo.rs修改

// SPDX-License-Identifier: GPL-2.0

//! Rust for linux e1000 driver demo

#![allow(unused)]

use core::iter::Iterator;

use core::sync::atomic::AtomicPtr;

use kernel::pci::Resource;

use kernel::prelude::\*;

use kernel::sync::Arc;

use kernel::{pci, device, driver, bindings, net, dma, c\_str};

use kernel::device::RawDevice;

use kernel::sync::SpinLock;

use kernel::driver::DeviceRemoval;

mod consts;

mod hw\_defs;

mod ring\_buf;

mod e1000\_ops;

use hw\_defs::{TxDescEntry, RxDescEntry};

use ring\_buf::{RxRingBuf, TxRingBuf};

use e1000\_ops::E1000Ops;

use consts::\*;

module! {

type: E1000KernelMod,

name: "r4l\_e1000\_demo",

author: "Myrfy001",

description: "Rust for linux e1000 driver demo",

license: "GPL",

}

/// The private data for this driver

struct NetDevicePrvData {

dev: Arc<device::Device>,

napi: Arc<net::Napi>,

e1000\_hw\_ops: Arc<E1000Ops>,

tx\_ring: SpinLock<Option<TxRingBuf>>,

rx\_ring: SpinLock<Option<RxRingBuf>>,

irq: u32,

\_irq\_handler: AtomicPtr<kernel::irq::Registration<E1000InterruptHandler>>

}

// TODO not sure why it is safe to do this.

unsafe impl Send for NetDevicePrvData {}

unsafe impl Sync for NetDevicePrvData {}

/// Represent the network device

struct NetDevice {}

impl NetDevice {

/// Alloc the tx descriptor. But doesn't need to alloc buffer memory, since the network stack will pass in a SkBuff.

fn e1000\_setup\_all\_tx\_resources(data: &NetDevicePrvData) -> Result<TxRingBuf> {

// Alloc dma memory space for tx desciptors

let dma\_desc = dma::Allocation::<hw\_defs::TxDescEntry>::try\_new(&\*data.dev, TX\_RING\_SIZE, bindings::GFP\_KERNEL)?;

// Safety: all fields of the slice members will be inited below.

let tx\_ring = unsafe{core::slice::from\_raw\_parts\_mut(dma\_desc.cpu\_addr, TX\_RING\_SIZE)};

tx\_ring.iter\_mut().enumerate().for\_each(|(idx, desc)| {

desc.buf\_addr = 0;

desc.cmd = 0;

desc.length = 0;

desc.cso = 0;

desc.css = 0;

desc.special = 0;

desc.sta = E1000\_TXD\_STAT\_DD as u8; // Mark all the descriptors as Done, so the first packet can be transmitted.

});

Ok(TxRingBuf::new(dma\_desc, TX\_RING\_SIZE))

}

/// Alloc the rx descriptor and the corresponding memory space. use `alloc\_skb\_ip\_align` to alloc buffer and then map it to

/// DMA address.

fn e1000\_setup\_all\_rx\_resources(dev: &net::Device, data: &NetDevicePrvData) -> Result<RxRingBuf> {

// Alloc dma memory space for rx desciptors

let dma\_desc = dma::Allocation::<hw\_defs::RxDescEntry>::try\_new(&\*data.dev, RX\_RING\_SIZE, bindings::GFP\_KERNEL)?;

// Safety: all fields of the slice members will be inited below.

let rx\_ring\_desc = unsafe{core::slice::from\_raw\_parts\_mut(dma\_desc.cpu\_addr, RX\_RING\_SIZE)};

// Alloc dma memory space for buffers

let dma\_buf = dma::Allocation::<u8>::try\_new(&\*data.dev, RX\_RING\_SIZE \* RXTX\_SINGLE\_RING\_BLOCK\_SIZE, bindings::GFP\_KERNEL)?;

let mut rx\_ring = RxRingBuf::new(dma\_desc, RX\_RING\_SIZE);

rx\_ring\_desc.iter\_mut().enumerate().for\_each(|(idx, desc)| {

let skb = dev.alloc\_skb\_ip\_align(RXTX\_SINGLE\_RING\_BLOCK\_SIZE as u32).unwrap();

let dma\_map = dma::MapSingle::try\_new(&\*data.dev, skb.head\_data().as\_ptr() as \*mut u8, RXTX\_SINGLE\_RING\_BLOCK\_SIZE, bindings::dma\_data\_direction\_DMA\_FROM\_DEVICE).unwrap();

desc.buf\_addr = dma\_map.dma\_handle as u64;

desc.length = 0;

desc.special = 0;

desc.checksum = 0;

desc.status = 0;

desc.errors = 0;

rx\_ring.buf.borrow\_mut()[idx] = Some((dma\_map, skb));

});

Ok(rx\_ring)

}

// corresponding to the C version e1000\_clean\_tx\_irq()

fn e1000\_recycle\_tx\_queue(dev: &net::Device, data: &NetDevicePrvData) {

let tdt = data.e1000\_hw\_ops.e1000\_read\_tx\_queue\_tail();

let tdh = data.e1000\_hw\_ops.e1000\_read\_tx\_queue\_head();

let mut tx\_ring = data.tx\_ring.lock\_irqdisable();

let mut tx\_ring = tx\_ring.as\_mut().unwrap();

let descs = tx\_ring.desc.as\_desc\_slice();

let mut idx = tx\_ring.next\_to\_clean;

while descs[idx].sta & E1000\_TXD\_STAT\_DD as u8 != 0 && idx != tdh as usize {

let (dm, skb) = tx\_ring.buf.borrow\_mut()[idx].take().unwrap();

dev.completed\_queue(1, skb.len());

skb.napi\_consume(64);

drop(dm);

drop(skb);

idx = (idx + 1) % TX\_RING\_SIZE;

}

tx\_ring.next\_to\_clean = idx;

}

}

#[vtable]

impl net::DeviceOperations for NetDevice {

type Data = Box<NetDevicePrvData>;

/// this method will be called when you type `ip link set eth0 up` in your shell.

fn open(dev: &net::Device, data: &NetDevicePrvData) -> Result {

pr\_info!("Rust for linux e1000 driver demo (net device open)\n");

dev.netif\_carrier\_off();

// init dma memory for tx and rx

let tx\_ringbuf = Self::e1000\_setup\_all\_tx\_resources(data)?;

let rx\_ringbuf = Self::e1000\_setup\_all\_rx\_resources(dev, data)?;

// TODO e1000\_power\_up\_phy() not implemented. It's used in case of PHY \*MAY\* power down,

// which will not be supported in this MVP driver.

// modify e1000's hardware registers, give rx/tx queue info to the nic.

data.e1000\_hw\_ops.e1000\_configure(&rx\_ringbuf, &tx\_ringbuf)?;

\*data.rx\_ring.lock\_irqdisable() = Some(rx\_ringbuf);

\*data.tx\_ring.lock\_irqdisable() = Some(tx\_ringbuf);

let irq\_prv\_data = Box::try\_new(IrqPrivateData{

e1000\_hw\_ops: Arc::clone(&data.e1000\_hw\_ops),

napi: Arc::clone(&data.napi),

})?;

// Again, the `irq::Registration` contains an `irq::InternalRegistration` which implemented `Drop`, so

// we mustn't let it dropped.

// TODO: there is memory leak now.

let req\_reg = kernel::irq::Registration::<E1000InterruptHandler>::try\_new(data.irq, irq\_prv\_data, kernel::irq::flags::SHARED, fmt!("{}",data.dev.name()))?;

data.\_irq\_handler.store(Box::into\_raw(Box::try\_new(req\_reg)?), core::sync::atomic::Ordering::Relaxed);

data.napi.enable();

dev.netif\_start\_queue();

dev.netif\_carrier\_on();

Ok(())

}

fn stop(\_dev: &net::Device, \_data: &NetDevicePrvData) -> Result {

pr\_info!("Rust for linux e1000 driver demo (net device stop)\n");

let irq\_handler\_ptr=\_data.\_irq\_handler.load(core::sync::atomic::Ordering::Relaxed);

if !irq\_handler\_ptr.is\_null(){

unsafe{

let irq=Box::from\_raw(irq\_handler\_ptr);

}

\_data.\_irq\_handler.store(core::ptr::null\_mut(),core::sync::atomic::Ordering::Relaxed);

}

Ok(())

}

fn start\_xmit(skb: &net::SkBuff, dev: &net::Device, data: &NetDevicePrvData) -> net::NetdevTx {

if skb.head\_data().len() > RXTX\_SINGLE\_RING\_BLOCK\_SIZE {

pr\_err!("xmit msg too long");

return net::NetdevTx::Busy

}

let mut tx\_ring = data.tx\_ring.lock\_irqdisable();

let mut tdt = data.e1000\_hw\_ops.e1000\_read\_tx\_queue\_tail();

let tdh = data.e1000\_hw\_ops.e1000\_read\_tx\_queue\_head();

let rdt = data.e1000\_hw\_ops.e1000\_read\_rx\_queue\_tail();

let rdh = data.e1000\_hw\_ops.e1000\_read\_rx\_queue\_head();

pr\_info!("Rust for linux e1000 driver demo (net device start\_xmit) tdt={}, tdh={}, rdt={}, rdh={}\n", tdt, tdh, rdt, rdh);

/\* On PCI/PCI-X HW, if packet size is less than ETH\_ZLEN,

\* packets may get corrupted during padding by HW.

\* To WA this issue, pad all small packets manually.

\*/

skb.put\_padto(bindings::ETH\_ZLEN);

// tell the kernel that we have pended some data to the hardware.

dev.sent\_queue(skb.len());

let mut tx\_ring = tx\_ring.as\_mut().unwrap();

let tx\_descs:&mut [TxDescEntry] = tx\_ring.desc.as\_desc\_slice();

let tx\_desc = &mut tx\_descs[tdt as usize];

if tx\_desc.sta & E1000\_TXD\_STAT\_DD as u8 == 0 {

pr\_err!("xmit busy");

return net::NetdevTx::Busy;

}

// alloc DMA map to skb

let ms:dma::MapSingle<u8> = if let Ok(ms) = dma::MapSingle::try\_new(&\*data.dev, skb.head\_data().as\_ptr() as \*mut u8, skb.len() as usize, bindings::dma\_data\_direction\_DMA\_TO\_DEVICE) {

ms

} else {

return net::NetdevTx::Busy;

};

tx\_desc.buf\_addr = ms.dma\_handle as u64;

tx\_desc.length = skb.len() as u16;

tx\_desc.cmd = ((E1000\_TXD\_CMD\_RS | E1000\_TXD\_CMD\_EOP) >> 24) as u8;

tx\_desc.sta = 0;

tx\_ring.buf.borrow\_mut()[tdt as usize].replace((ms, skb.into()));

// TODO memory fence here. we are testing it on an x86, so maybe left it out is ok.

tdt = (tdt + 1) % TX\_RING\_SIZE as u32;

data.e1000\_hw\_ops.e1000\_write\_tx\_queue\_tail(tdt);

net::NetdevTx::Ok

}

fn get\_stats64(\_netdev: &net::Device, \_data: &NetDevicePrvData, stats: &mut net::RtnlLinkStats64) {

pr\_info!("Rust for linux e1000 driver demo (net device get\_stats64)\n");

// TODO not implemented.

stats.set\_rx\_bytes(0);

stats.set\_rx\_packets(0);

stats.set\_tx\_bytes(0);

stats.set\_tx\_packets(0);

}

}

// since the ownership limitation, We can't use NetDevicePrvData as C code, so we need to define a new type here.

struct IrqPrivateData {

e1000\_hw\_ops: Arc<E1000Ops>,

napi: Arc<net::Napi>,

}

struct E1000InterruptHandler {}

impl kernel::irq::Handler for E1000InterruptHandler {

type Data = Box<IrqPrivateData>;

fn handle\_irq(data: &IrqPrivateData) -> kernel::irq::Return {

pr\_info!("Rust for linux e1000 driver demo (handle\_irq)\n");

let pending\_irqs = data.e1000\_hw\_ops.e1000\_read\_interrupt\_state();

pr\_info!("pending\_irqs: {}\n", pending\_irqs);

if pending\_irqs == 0 {

return kernel::irq::Return::None

}

data.napi.schedule();

kernel::irq::Return::Handled

}

}

/// the private data for the adapter

struct E1000DrvPrvData {

\_netdev\_reg: net::Registration<NetDevice>,

bars:i32,

dev\_ptr:\*mut bindings::pci\_dev,

irq: u32,

}

impl Drop for E1000DrvPrvData{

fn drop(&mut self){

pr\_info!("drop E1000DrvPrvData\n");

self.disable\_device();

self.release\_selected\_regions();

}

}

impl E1000DrvPrvData{

fn disable\_device(&self)->Result{

unsafe{ bindings::pci\_disable\_device(self.dev\_ptr)};

Ok(())

}

fn release\_selected\_regions(&self)->Result{

unsafe{bindings::pci\_release\_selected\_regions(self.dev\_ptr,self.bars)};

Ok(())

}

}

impl driver::DeviceRemoval for E1000DrvPrvData {

fn device\_remove(&self) {

pr\_info!("Rust for linux e1000 driver demo (device\_remove)\n");

}

}

unsafe impl Send for E1000DrvPrvData {}

unsafe impl Sync for E1000DrvPrvData {}

struct NapiHandler{}

impl net::NapiPoller for NapiHandler {

type Data = Box<NetDevicePrvData>;

fn poll(

\_napi: &net::Napi,

\_budget: i32,

dev: &net::Device,

data: &NetDevicePrvData,

) -> i32 {

pr\_info!("Rust for linux e1000 driver demo (napi poll)\n");

let mut rdt = data.e1000\_hw\_ops.e1000\_read\_rx\_queue\_tail() as usize;

rdt = (rdt + 1) % RX\_RING\_SIZE;

let mut rx\_ring\_guard = data.rx\_ring.lock();

let rx\_ring = rx\_ring\_guard.as\_mut().unwrap();

let mut descs = rx\_ring.desc.as\_desc\_slice();

while descs[rdt].status & E1000\_RXD\_STAT\_DD as u8 != 0 {

let packet\_len = descs[rdt].length as usize;

let buf = &mut rx\_ring.buf.borrow\_mut();

let skb = &buf[rdt].as\_mut().unwrap().1;

skb.put(packet\_len as u32);

let protocol = skb.eth\_type\_trans(dev);

skb.protocol\_set(protocol);

data.napi.gro\_receive(skb);

let skb\_new = dev.alloc\_skb\_ip\_align(RXTX\_SINGLE\_RING\_BLOCK\_SIZE as u32).unwrap();

let dma\_map = dma::MapSingle::try\_new(&\*data.dev, skb\_new.head\_data().as\_ptr() as \*mut u8, RXTX\_SINGLE\_RING\_BLOCK\_SIZE, bindings::dma\_data\_direction\_DMA\_FROM\_DEVICE).unwrap();

descs[rdt].buf\_addr = dma\_map.dma\_handle as u64;

buf[rdt] = Some((dma\_map, skb\_new));

descs[rdt].status = 0;

data.e1000\_hw\_ops.e1000\_write\_rx\_queue\_tail(rdt as u32);

rdt = (rdt + 1) % RX\_RING\_SIZE;

}

NetDevice::e1000\_recycle\_tx\_queue(dev, data);

data.napi.complete\_done(1);

1

}

}

struct E1000Drv {}

impl pci::Driver for E1000Drv {

// The Box type has implemented PointerWrapper trait.

type Data = Box<E1000DrvPrvData>;

kernel::define\_pci\_id\_table! {(), [

(pci::DeviceId::new(E1000\_VENDER\_ID, E1000\_DEVICE\_ID), None),

]}

fn probe(dev: &mut pci::Device, id: core::option::Option<&Self::IdInfo>) -> Result<Self::Data> {

pr\_info!("Rust for linux e1000 driver demo (probe): {:?}\n", id);

// Note: only support QEMU's 82540EM chip now.

// this works like a filter, the PCI device may have up to 6 bars, those bars have different types,

// some of them are mmio, others are io-port based. The params to the following function is a

// filter condition, and the return value is a mask indicating which of those bars are selected.

let bars = dev.select\_bars((bindings::IORESOURCE\_MEM | bindings::IORESOURCE\_IO) as u64);

// the underlying will call `pci\_enable\_device()`. the R4L framework doesn't support `pci\_enable\_device\_memory()` now.

dev.enable\_device()?;

// ask the os to reserve the physical memory region of the selected bars.

dev.request\_selected\_regions(bars, c\_str!("e1000 reserved memory"))?;

// set device to master mode.

dev.set\_master();

// get resource(memory range) provided by BAR0

let mem\_res = dev.iter\_resource().next().ok\_or(kernel::error::code::EIO)?;

let io\_res = dev.iter\_resource().skip(1).find(|r:&Resource|r.check\_flags(bindings::IORESOURCE\_IO)).ok\_or(kernel::error::code::EIO)?;

// TODO pci\_save\_state(pdev); not supported by crate now, only have raw C bindings.

// alloc new ethernet device, this line represent the `alloc\_etherdev()` and `SET\_NETDEV\_DEV()` in C version.

let mut netdev\_reg = net::Registration::<NetDevice>::try\_new(dev)?;

let netdev = netdev\_reg.dev\_get();

// map device registers' hardware address to logical address so the kernel driver can access it.

let mem\_addr = Arc::try\_new(dev.map\_resource(&mem\_res, mem\_res.len())?)?;

// get the io-port based address

let io\_addr = Arc::try\_new(pci::IoPort::try\_new(&io\_res)?)?;

// TODO implement C version `e1000\_init\_hw\_struct()`

// only pci-x need 64-bit, to simplify code, hardcode 32-bit for now.

dma::set\_coherent\_mask(dev, 0xFFFFFFFF)?;

// TODO ethtool support here.

// Enable napi, the R4L will call `netif\_napi\_add\_weight()`, the origin C version calls `netif\_napi\_add`

let napi = net::NapiAdapter::<NapiHandler>::add\_weight(&netdev, 64)?;

// TODO implement C version `e1000\_sw\_init()`

// TODO a lot of feature flags are assigned here in the C code, skip them for now.

let e1000\_hw\_ops = E1000Ops {

mem\_addr: Arc::clone(&mem\_addr),

io\_addr: Arc::clone(&io\_addr),

};

e1000\_hw\_ops.e1000\_reset\_hw()?;

// TODO: the MAC address is hardcoded here, should be read out from EEPROM later.

netdev.eth\_hw\_addr\_set(&MAC\_HWADDR);

// TODO: Some background tasks and Wake on LAN are not supported now.

let irq = dev.irq();

let common\_dev = device::Device::from\_dev(dev);

netdev.netif\_carrier\_off();

// SAFETY: `spinlock\_init` is called below.

let mut tx\_ring = unsafe{SpinLock::new(None)};

let mut rx\_ring = unsafe{SpinLock::new(None)};

// SAFETY: We don't move `tx\_ring` and `rx\_ring`.

kernel::spinlock\_init!(unsafe{Pin::new\_unchecked(&mut tx\_ring)}, "tx\_ring");

kernel::spinlock\_init!(unsafe{Pin::new\_unchecked(&mut rx\_ring)}, "rx\_ring");

netdev\_reg.register(Box::try\_new(

NetDevicePrvData {

dev: Arc::try\_new(common\_dev)?,

e1000\_hw\_ops: Arc::try\_new(e1000\_hw\_ops)?,

napi: napi.into(),

tx\_ring,

rx\_ring,

irq,

\_irq\_handler: AtomicPtr::new(core::ptr::null\_mut()),

}

)?)?;

//let mut\*

Ok(Box::try\_new(

E1000DrvPrvData{

// Must hold this registration, or the device will be removed.

\_netdev\_reg: netdev\_reg,

bars:bars,

dev\_ptr:dev.get\_point(),

irq:irq,

}

)?)

}

fn remove(data: &Self::Data) {

pr\_info!("Rust for linux e1000 driver demo (remove)\n");

//drop(data);

}

}

struct E1000KernelMod {

\_dev: Pin<Box<driver::Registration::<pci::Adapter<E1000Drv>>>>,

}

impl kernel::Module for E1000KernelMod {

fn init(name: &'static CStr, module: &'static ThisModule) -> Result<Self> {

pr\_info!("Rust for linux e1000 driver demo (init)\n");

let d = driver::Registration::<pci::Adapter<E1000Drv>>::new\_pinned(name, module)?;

// we need to store `d` into the module struct, otherwise it will be dropped, which

// means the driver will be removed.

Ok(E1000KernelMod {\_dev: d})

}

}

impl Drop for E1000KernelMod {

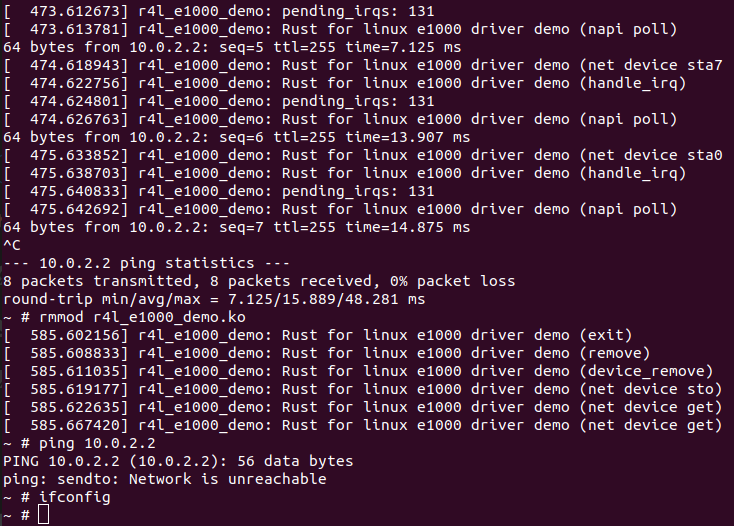
fn drop(&mut self) {

pr\_info!("Rust for linux e1000 driver demo (exit)\n");

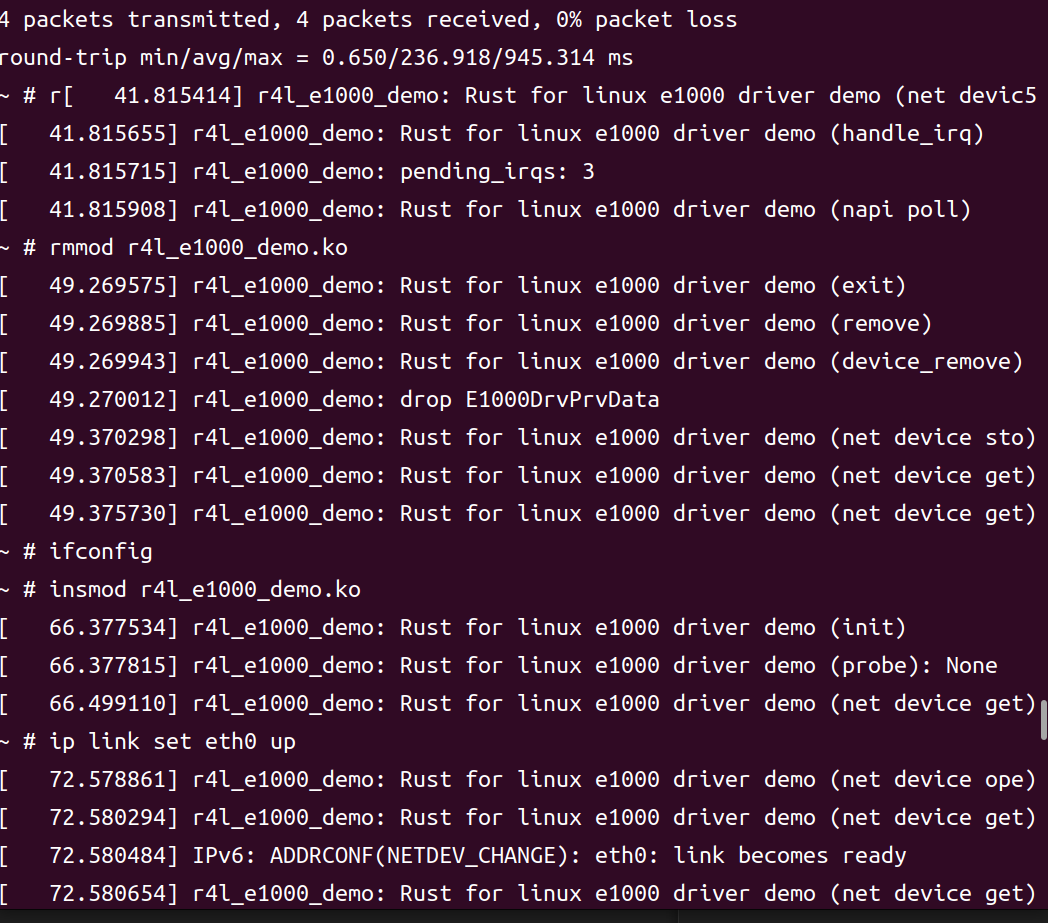
}

}

rmmod r4l\_e1000\_demo.ko后发现无法ping通



重新insmod

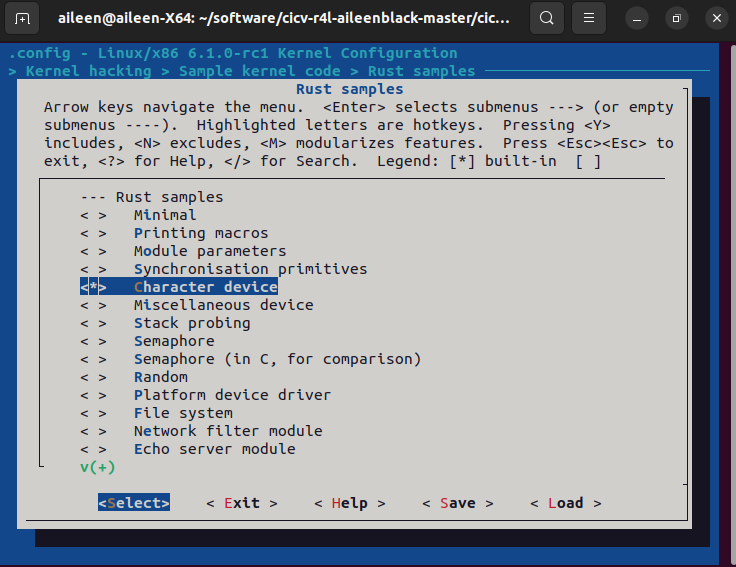


作业5：注册字符设备

更改配置

如下make LLVM =1 menuconfig 修改配置

kernel hacking -> Sample Kernel code ->Rust samples-> <\*> Character device (New)



代码修改

// SPDX-License-Identifier: GPL-2.0

//! Rust character device sample.

use core::result::Result::Err;

use kernel::prelude::\*;

use kernel::sync::Mutex;

use kernel::{chrdev, file};

const GLOBALMEM\_SIZE: usize = 0x1000;

module! {

type: RustChrdev,

name: "rust\_chrdev",

author: "Rust for Linux Contributors",

description: "Rust character device sample",

license: "GPL",

}

static GLOBALMEM\_BUF: Mutex<[u8;GLOBALMEM\_SIZE]> = unsafe {

Mutex::new([0u8;GLOBALMEM\_SIZE])

};

struct RustFile {

#[allow(dead\_code)]

inner: &'static Mutex<[u8;GLOBALMEM\_SIZE]>,

// inner: &'static Mutex<Vec<u8>>,

//inner:Mutex<Vec<u8>>,

}

#[vtable]

impl file::Operations for RustFile {

type Data = Box<Self>;

fn open(\_shared:&(), \_file: &file::File) -> Result<Box<Self>> {

Ok(

//unsafe {

Box::try\_new(RustFile {

inner: &GLOBALMEM\_BUF

//inner:\_shared.inner.clone(),

})?

// }

)

// Ok(())

}

fn write(\_this: &Self,\_file: &file::File,\_reader: &mut impl kernel::io\_buffer::IoBufferReader,\_offset:u64,) -> Result<usize> {

//Err(EPERM)

pr\_info!("character write\n");

let copy = \_reader.read\_all()?;

let len = copy.len();

let v=&copy;

//let mut det=&\*\_this.inner.lock();

let mut inner = \_this.inner.lock();

if len < GLOBALMEM\_SIZE{

let mut id=0;

while id<len{

inner[id]=v[id];

id+=1;

}

Ok(len)

}else{

Ok(0)

}

}

fn read(\_this: &Self,\_file: &file::File,\_writer: &mut impl kernel::io\_buffer::IoBufferWriter,\_offset:u64,) -> Result<usize> {

//Err(EPERM)

pr\_info!("character read\n", );

let offset = \_offset.try\_into()?;

let vec = \_this.inner.lock(); // 获取锁，避免脏读

let len = core::cmp::min(\_writer.len(), vec.len().saturating\_sub(offset));

\_writer.write\_slice(&vec[offset..][..len])?;

//pr\_info!("{}\n",vec[offset..][...len]);

Ok(len)

}

}

struct RustChrdev {

\_dev: Pin<Box<chrdev::Registration<2>>>,

}

impl kernel::Module for RustChrdev {

fn init(name: &'static CStr, module: &'static ThisModule) -> Result<Self> {

pr\_info!("Rust character device sample (init)\n");

let mut chrdev\_reg = chrdev::Registration::new\_pinned(name, 0, module)?;

// Register the same kind of device twice, we're just demonstrating

// that you can use multiple minors. There are two minors in this case

// because its type is `chrdev::Registration<2>`

chrdev\_reg.as\_mut().register::<RustFile>()?;

chrdev\_reg.as\_mut().register::<RustFile>()?;

Ok(RustChrdev { \_dev: chrdev\_reg })

}

}

impl Drop for RustChrdev {

fn drop(&mut self) {

pr\_info!("Rust character device sample (exit)\n");

}

}

make LLVM -j4编译

再进入src\_e1000运行./build\_image.sh

效果如下：

