操作系统原理实验（1）：实现最小内核

原创

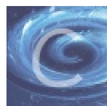
麓山君陌 于 2020-10-14 20:58:11 发布 2628 收藏 9

分类专栏： 操作系统原理实验

版权声明：本文为博主原创文章，遵循 CC 4.0 BY 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/qq_40851744/article/details/109083128

版权

 操作系统原理实验 专栏收录该内容

5 篇文章 3 订阅

订阅专栏

一、实验目的

安全性实验包含两个实验项目(参见表4.3)，其中1个为必修，1个为选修。自主存取控制实验为设计型实验项目，审计实验为验证型实验项目。

二、实验过程&错误

内容（一）：安装工具链（Mac）

步骤1：下载Rust，输入`curl https://sh.rustup.rs -sSf | sh`

问题1-1：显示未安装curl

```
junmo@ubuntu:~/shlab实验文件-20190521/shlab-handout$ curl https://sh.rustup.rs -sSf|sh
程序"curl"尚未安装。 您可以使用以下命令安装：
sudo apt install curl
```

解决方法1-1：输入`sudo apt install curl`安装curl

```
junmo@ubuntu:~/shlab实验文件-20190521/shlab-handout$ sudo apt install curl
```

现象1-1：停止在选择界面，有三个选项可以选择

```
junmo@ubuntu:~/shlab实验文件-20190521/shlab-handout$ curl https://sh.rustup.rs -sSf|sh
info: downloading installer

Welcome to Rust!

This will download and install the official compiler for the Rust programming
language, and its package manager, Cargo.

It will add the cargo, rustc, rustup and other commands to Cargo's bin
directory, located at:

  /home/junmo/.cargo/bin

This can be modified with the CARGO_HOME environment variable.

Rustup metadata and toolchains will be installed into the Rustup home
directory, located at:

  /home/junmo/.rustup

This can be modified with the RUSTUP_HOME environment variable.

This path will then be added to your PATH environment variable by modifying the
profile file located at:

  /home/junmo/.profile

You can uninstall at any time with rustup self uninstall and these changes will
be reverted.

Current installation options:

   default host triple: i686-unknown-linux-gnu
     default toolchain: stable
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
```

https://blog.csdn.net/qq_40851744

问题1-2：选择安装版本

解决方法1-2：这里直接输入1

```
     default toolchain: stable
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>1
```

现象1-2：当出现进度条的时候表示正在安装，保持网络通畅，等待即可

```
  4.8 MiB /   4.8 MiB (100 %) 780.8 KiB/s in  3s ETA:  0s
info: installing component 'rust-docs'
 11.3 MiB /  11.3 MiB (100 %) 523.2 KiB/s in 18s ETA:  0s
info: default toolchain set to 'stable'

  stable installed - rustc 1.37.0 (eae3437df 2019-08-13)


Rust is installed now. Great!

To get started you need Cargo's bin directory ($HOME/.cargo/bin) in your PATH
environment variable. Next time you log in this will be done automatically.

To configure your current shell run source $HOME/.cargo/env
junmo@ubuntu:~$
```

问题1-3：如何证明安装成功

解决方法1-3：输入`rustc --version`，这里是查看当前rust的版本的意思，如果出现当前的版本号，表示安装成功

```
junmo@ubuntu:~$ rustc --version
```

现象1-3：安装成功

```
junmo@ubuntu:~$ rustc --version
rustc 1.37.0 (eae3437df 2019-08-13)
junmo@ubuntu:~$
```

步骤2：安装nightly版本，输入`rustup install nightly`

```
junmo@ubuntu:~$ rustup install nightly
```

问题2-1：网络问题导致安装出错

```
junmo@ubuntu:~$ rustup install nightly
info: syncing channel updates for 'nightly-i686-unknown-linux-gnu'
info: latest update on 2019-09-22, rust version 1.39.0-nightly (ed8b708c1 2019-0
9-21)
info: downloading component 'rustc'
 79.1 MiB /  79.1 MiB (100 %) 848.0 KiB/s in  1m 31s ETA:  0s
info: downloading component 'rust-std'
 46.6 MiB / 175.0 MiB ( 27 %) 914.9 KiB/s in 52s ETA:  2m 23serror: component do
wnload failed for rust-std-i686-unknown-linux-gnu
info: caused by: could not download file from 'https://static.rust-lang.org/dist
/2019-09-22/rust-std-nightly-i686-unknown-linux-gnu.tar.xz' to '/home/junmo/.rus
tup/downloads/87040c8b874620916857028 3b03a69e19a3ca0b3eaf01644adddbbf375b4b4b1.p
artial'
info: caused by: error reading from socket
info: caused by: timed out
```

解决方法2-1：确保网络通畅，最好要求1m/s以上

现象2-1：安装成功

```
  nightly-i686-unknown-linux-gnu installed - rustc 1.39.0-nightly (ed8b708c1 201
9-09-21)

info: checking for self-updates
```

步骤3：默认使用nightly版本，输入`rustup default nightly`

现象3：默认完成

```
junmo@ubuntu:~$ rustup default nightly
info: using existing install for 'nightly-i686-unknown-linux-gnu'
info: default toolchain set to 'nightly-i686-unknown-linux-gnu'

  nightly-i686-unknown-linux-gnu unchanged - rustc 1.39.0-nightly (ed8b708c1 201
9-09-21)
```

步骤4：安装bootimage，xbuild和rust-src等工具，依次输入：

```
cargo install bootimage --version “ˆ0.7.3”
cargo install cargo-xbuild
rustup component add rust-src
rustup component add llvm-tools-preview
```

问题4-1：安装bootimage出错

```
junmo@ubuntu:~$ cargo install bootimage --version"^0.7.3"
error: Found argument '--version^0.7.3' which wasn't expected, or isn't valid in
 this context
        Did you mean --version?

USAGE:
    cargo install <crate>... --version <VERSION>

For more information try --help
junmo@ubuntu:~$
```

解决方法4-1：这里是pdf输入的代码有错误，应该将双引号去掉，即输入

```
cargo install bootimage --version ˆ0.7.3
```

```
junmo@ubuntu:~$ cargo install bootimage --version ^0.7.3
    Updating crates.io index
```

现象4-1：修改后bootimage安装成功

```
  Installing /home/junmo/.cargo/bin/cargo-bootimage
   Installed package `bootimage v0.7.7` (executables `bootimage`, `cargo-bootima
ge`)
```

现象4-2：安装xbuild成功

```
junmo@ubuntu:~$ cargo install cargo-xbuild
    Updating crates.io index
```

```
  Installing /home/junmo/.cargo/bin/cargo-xtest
   Installed package `cargo-xbuild v0.5.17` (executables `cargo-xb`, `cargo-xbu
ld`, `cargo-xc`, `cargo-xcheck`, `cargo-xclippy`, `cargo-xdoc`, `cargo-xr`, `ca
go-xrun`, `cargo-xrustc`, `cargo-xt`, `cargo-xtest`)
```

现象4-3：安装rust-src成功

```
junmo@ubuntu:~$ rustup component add rust-src
info: downloading component 'rust-src'
  2.5 MiB /   2.5 MiB (100 %) 581.0 KiB/s in  5s ETA:  0s
info: installing component 'rust-src'
```

现象4-4：安装llvm-tools-preview成功

```
junmo@ubuntu:~$ rustup component add llvm-tools-preview
info: downloading component 'llvm-tools-preview'
726.3 KiB / 726.3 KiB (100 %) 588.1 KiB/s in  2s ETA:  0s
info: installing component 'llvm-tools-preview'
junmo@ubuntu:~$
```

步骤5：安装QEMU，输入`/usr/bin/ruby -e “$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)” brew install qemu`

问题5-1：安装qemu出错，缺少了一个空格

```
junmo@ubuntu:~$ /usr/bin/ruby -e "$(curl-fsSL https://raw.githubusercomtent.com/
Homebrew/install/master/inst all)" brew install qemu
curl-fsSL：未找到命令
bash: /usr/bin/ruby: 没有那个文件或目录
```

解决方法5-1：应该在curl和-fsSL中间加上一个空格

问题5-2：安装qemu出错，缺少文件夹

```
junmo@ubuntu:~$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercomtent.com
/Homebrew/install/master/inst all)" brew install qemu
curl: (6) Could not resolve host: raw.githubusercomtent.com
curl: (6) Could not resolve host: all
bash: /usr/bin/ruby: 没有那个文件或目录
```

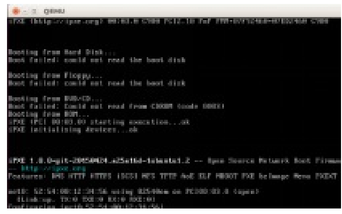
解决方法5-2：换一种方式安装qemu，即输入`sudo apt-get install qemu`

```
junmo@ubuntu:~$ sudo apt-get install qemu
[sudo] junmo 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  linux-headers-4.4.0-21 linux-headers-4.4.0-21-generic
  linux-image-4.4.0-21-generic linux-image-extra-4.4.0-21-generic
使用'sudo apt autoremove'来卸载它(它们)。
将会同时安装下列软件：
```

现象5-1：安装完成

```
正在处理用于 libc-bin (2.23-0ubuntu10) 的触发器 ...
正在处理用于 systemd (229-4ubuntu21.21) 的触发器 ...
正在处理用于 ureadahead (0.100.0-19) 的触发器 ...
junmo@ubuntu:~$
```

现象5-2：证明安装完成，在终端输入qemu，会弹出qemu窗口，如果没有弹出，可以先输入`sudo ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu`将qemu与qemu-system-i386进行链接即可



内容（二）：创建裸机程序

步骤1：创建一个新的cargo项目，输入`cargo new blog_os`

现象1-1：创建成功

```
junmo@ubuntu:~$ cargo new junmo_os
     Created binary (application) `junmo_os` package
```

步骤2：禁用标准库链接，打开src文件夹下面的main.rs文件，输入如下的代码

```
#![no_std]

fn main() {
    println!("Hello, world!");
}
```

步骤3：使用cargo build编译，即输入`cargo build`

```
junmo@ubuntu:~$ cargo build
```

问题3-1：报错，找不到.toml文件

```
junmo@ubuntu:~$ cargo build
error: could not find `Cargo.toml` in `/home/junmo` or any parent directory
```

解决方法3-1：转到新建的文件夹下面后重新编译

```
junmo@ubuntu:~$ cd junmo_os
junmo@ubuntu:~/junmo_os$ cargo build
```

问题3-2：报错，找不到println!函数（这里是因为println!宏是标准库的一部分，而我们的项目不再依赖于标准库，所以找不到，我们可以将main.rs中这段语句移除，再进行编译）
解决方法3-2：移除main.rs中有关于println的语句

```
#![no_std]

fn main() {
}
```

问题3-3：报错，panic错误，因为编译器缺少一个panic处理函数和一个语言项

```
junmo@ubuntu:~/junmo_os$ cargo build
   Compiling junmo_os v0.1.0 (/home/junmo/junmo_os)
error: `#[panic_handler]` function required, but not found

error: language item required, but not found: `eh_personality`

error: aborting due to 2 previous errors

error: could not compile `junmo_os`.

To learn more, run the command again with --verbose.
```

解决方法3-3：在main.rs文件中加入panic函数，需要自己定义，输入如下代码：

```
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> !{
        loop{}
}

fn main() {
}
```

问题3-4：报错，语言项错误：

```
junmo@ubuntu:~/junmo_os$ cargo build
   Compiling junmo_os v0.1.0 (/home/junmo/junmo_os)
error: language item required, but not found: `eh_personality`

error: aborting due to previous error

error: could not compile `junmo_os`.

To learn more, run the command again with --verbose.
junmo@ubuntu:~/junmo_os$
```

解决方法3-4：我们需要在toml中设置panic的参数，即在toml文件中加入如下语句：

```
[package]
name = "junmo_os"
version = "0.1.0"
authors = ["junmo"]
edition = "2018"

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

问题3-5：报错，start错误

```
junmo@ubuntu:~/junmo_os$ cargo build
   Compiling junmo_os v0.1.0 (/home/junmo/junmo_os)
error: requires `start` lang_item

error: aborting due to previous error

error: could not compile `junmo_os`.

To learn more, run the command again with --verbose.
junmo@ubuntu:~/junmo_os$
```

解决方法3-5：我们的程序遗失了start函数，这是程序的进入点，所以我们的程序无法正常运行，所以我们需要重新写一个crt0库和定义他的入口点，实际上直接将main.rs函数修改如下即可：

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> !{
        loop{}
}

#[no_mangle]
pub extern "C" fn _start() -> !{
        loop{}
}
```

问题3-6：报错，链接器错误

```
junmo@ubuntu:~/junmo_os$ cargo build
   Compiling junmo_os v0.1.0 (/home/junmo/junmo_os)
error: linking with `cc` failed: exit code: 1
  |
  = note: "cc" "-Wl,--as-needed" "-Wl,-z,noexecstack" "-m32" "-L" "/home/junmo/.rustup/toolchains/nightly-i686-unknown-linux-gnu/lib/rustlib/i686-unknown-linux-gnu/lib" "/home/junmo/junmo_os/target/debug/deps/junmo_os-49a41b65e9fb5a0f.yv4jbug8cqw5seb.rcgu.o" "-o" "/home/junmo/junmo_os/target/debug/deps/junmo_os-49a41b65e9fb5a0f" "-Wl,--gc-sections" "-pie" "-Wl,-zrelro" "-Wl,-znow" "-nodefaultlibs" "-L" "/home/junmo/junmo_os/target/debug/deps" "-L" "/home/junmo/.rustup/toolcha
```

解决方法3-6：首先我们要查看当前的目标三元组，即输入`rustc --version --verbose`

```
junmo@ubuntu:~/junmo_os$ rustc --version --verbose
rustc 1.39.0-nightly (ed8b708c1 2019-09-21)
binary: rustc
commit-hash: ed8b708c1a6bf6d94f860eeb6a6b0b442c380d7f
commit-date: 2019-09-21
host: i686-unknown-linux-gnu
release: 1.39.0-nightly
LLVM version: 9.0
```

添加thumbv7em，即输入`rustup target add thumbv7em-none-eabihf`，这行命令将为目标下载一个标准库和core库，之后就能为这个目标构建独立式可执行程序

```
junmo@ubuntu:~/junmo_os$ rustup target add thumbv7em-none-eabihf
info: downloading component 'rust-std' for 'thumbv7em-none-eabihf'
  4.7 MiB /   4.7 MiB (100 %)   1.0 MiB/s in  9s ETA:  0s
info: installing component 'rust-std' for 'thumbv7em-none-eabihf'
junmo@ubuntu:~/junmo_os$
```

构建独立式可执行程序，并传递–target参数，即输入`cargo build --target thumbv7em-none-eabihf`

```
junmo@ubuntu:~/junmo_os$ cargo build --target thumbv7em-none-eabihf
   Compiling junmo_os v0.1.0 (/home/junmo/junmo_os)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21s
junmo@ubuntu:~/junmo_os$
```

最终得到的代码如下：（main.rs）&（.toml程序）

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> !{
        loop{}
}

#[no_mangle]
pub extern "C" fn _start() -> !{
        loop{}
}
```

```
[package]
name = "junmo_os"
version = "0.1.0"
authors = ["junmo"]
edition = "2018"

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

现象3：程序运行结果

```
junmo@ubuntu:~/junmo_os$ cargo build --target thumbv7em-none-eabihf
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
```

内容（三）：构建最小内核

步骤1：构建目标配置清单文件（.json文件）

在操作系统（xx_os，这里是junmo_os文件夹）文件夹下面新建一个文件，然后命名为x86_64-blog_os.json，当然名称可以随便改。

内容如下，将如下代码直接复制进去即可：

```
{  "llvm-target": "x86_64-unknown-none",
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "arch": "x86_64",
  "target-endian": "little",
  "target-pointer-width": "64",
  "target-c-int-width": "32",
  "os": "none",
  "executables": true,
  "linker-flavor": "ld.lld",
  "linker": "rust-lld",
  "panic-strategy": "abort",
  "disable-redzone": true,
  "features": "-mmx,-sse,+soft-float"}
```

步骤2：编写内核（main.rs程序）
直接将main.rs文件修改如下即可：

```
// src/main.rs

#![no_std] // 不链接Rust标准库
#![no_main] // 禁用所有Rust层级的入口点

use core::panic::PanicInfo;

/// 这个函数将在panic时被调用
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}

#[no_mangle] // 不重整函数名
pub extern "C" fn _start() -> ! {
    // 因为编译器会寻找一个名为`_start`的函数，所以这个函数就是入口点
    // 默认命名为`_start`
    loop {}
}
```

步骤3：将json文件加入–target选项，并尝试编译我们的内核，即输入`cargo build --target x86_64-blog_os.json`
问题3-1：报错，rust编译器找不到core或compiler_builtins包

```
junmo@ubuntu:~/junmo2_os$ cargo build --target x86_64-blog_os.json
   Compiling junmo2_os v0.1.0 (/home/junmo/junmo2_os)
error[E0463]: can't find crate for `core`
  |
  = note: the `x86_64-blog_os-17931489391417382l9` target may not be installed

error: aborting due to previous error

For more information about this error, try `rustc --explain E0463`.
error: could not compile `junmo2_os`.

To learn more, run the command again with --verbose.
```

解决方法3-1：安装cargo xbuild库，即输入`cargo install cargo-xbuild`

```
junmo@ubuntu:~/junmo2_os$ cargo install cargo-xbuild
    Updating crates.io index
```

问题3-2：报错，cargo安装错误

```
junmo@ubuntu:~/junmo2_os$ cargo install cargo-xbuild
    Updating crates.io index
error: binary `cargo-xb` already exists in destination as part of `cargo-xbuild v0.5.17`
binary `cargo-xbuild` already exists in destination as part of `cargo-xbuild v0.5.17`
binary `cargo-xc` already exists in destination as part of `cargo-xbuild v0.5.17`
binary `cargo-xcheck` already exists in destination as part of `cargo-xbuild v0.
```

解决方法3-2：出错原因是因为未安装源代码，输入`rustup component add rust-src`，将rust-src进行更新

```
junmo@ubuntu:~/junmo2_os$ rustup component add rust-src
info: component 'rust-src' is up to date
```

现象3：使用xbuild重新编译后成功，即输入`rustup component add rust-src`

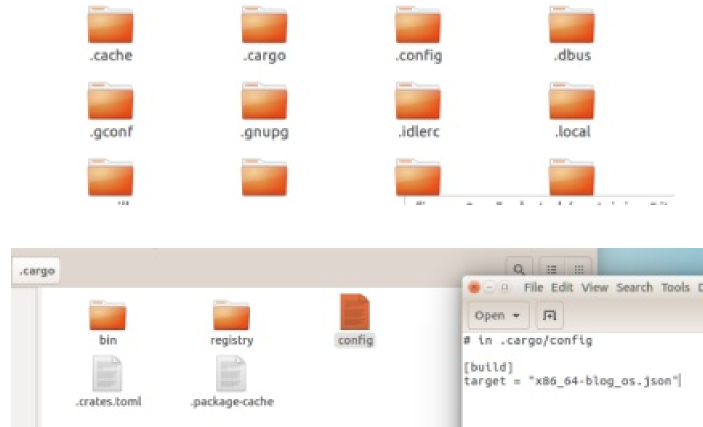
```
junmo@ubuntu:~/junmo2_os$ cargo xbuild --target x86_64-blog_os.json
    Updating crates.io index
  Downloaded compiler_builtins v0.1.19
   Compiling compiler_builtins v0.1.19
   Compiling core v0.0.0 (/home/junmo/.rustup/toolchains/nightly-i686-unknown-linux-gnu/lib/rustlib/src/rust/src/libcore)
   Compiling rustc-std-workspace-core v1.99.0 (/home/junmo/.rustup/toolchains/nightly-i686-unknown-linux-gnu/lib/rustlib/src/rust/src/tools/rustc-std-workspace-core)
   Compiling alloc v0.0.0 (/tmp/xargo.zGA3lSectc0S)
    Finished release [optimized] target(s) in 43.26s
   Compiling junmo2_os v0.1.0 (/home/junmo/junmo2_os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.54s
```

步骤4：设置默认目标：

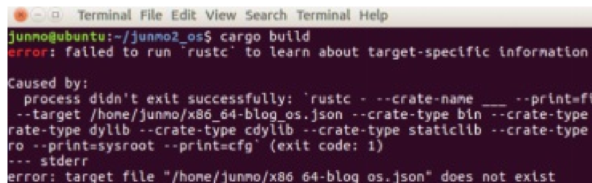
为了避免每次使用cargo xbuild时传递–target参数，我们可以覆写默认的编译目标。我们创建一个名为.cargo/config的cargo配置文件，并添加内容：

问题4-1：这个配置文件在哪里？

解决方法4-1：按下Ctrl+H显示隐藏文件，在HOME文件夹（最开始的文件夹）下面找到名为.cargo的文件夹，打开后创建空白新文件，并命名为config，然后添加代码

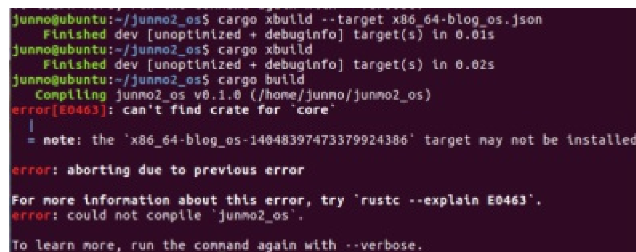


问题4-2：输入cargo build后报错



解决方法4-2：首先需要将json文件拿出来，放到主文件夹下面

另外，这里的命令有错误，应该是输入cargo xbuild才对，否则将会报错。改成xbuild后一切正常（在下面看到，输入xbuild结果和最初的结果是一样的，但build不行），如下：



步骤5：向屏幕打印字符。这里需要将main文件和toml文件修改为如下情况

```rust
// src/main.rs

#![no_std] // 不链接Rust标准库
#![no_main] // 禁用所有Rust层级的入口点

use core::panic::PanicInfo;

static HELLO: &[u8] = b"Hello World!";

#[no_mangle]
pub extern "C" fn _start() -> ! {
    let vga_buffer = 0xb8000 as *mut u8;

    for (i, &byte) in HELLO.iter().enumerate() {
        unsafe {
            *vga_buffer.offset(i as isize * 2) = byte;
            *vga_buffer.offset(i as isize * 2 + 1) = 0xb;
        }
    }

    loop {}
}

/// 这个函数将在panic时被调用
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

```toml
[package]
name = "junmo2_os"
version = "0.1.0"
authors = ["junmo"]
edition = "2018"

# in Cargo.toml

[dependencies]
bootloader = "0.6.0"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

在这段代码中，我们预先定义了一个字节字符串（byte string）类型的静态变量（static variable），名为HELLO。我们首先将整数0xb8000转换（cast）为一个裸指针（raw pointer）。这之后，我们迭代HELLO的每个字节，使用enumerate获得一个额外的序号变量i。在for语句的循环体中，我们使用offset偏移裸指针，解引用它，来将字符串的每个字节和对应的颜色字节——0xb代表淡青色——写入内存位置。

步骤6：输入`cargo bootimage`

将会执行三步：首先编译我们的内核为一个ELF（Executable and Linkable Format）文件；然后编译引导程序为独立的可执行文件；最后将内核ELF文件按字节拼接（append by bytes）到引导程序的末端。

问题6-1：报错，toml不完整

```
junmo@ubuntu:~/junmo2_os$ cargo bootimage
Building kernel
   Compiling nodrop v0.1.13
   Compiling bit_field v0.9.0
   Compiling zero v0.1.2
   Compiling os v0.1.3
error[E0461]: couldn't find crate `core` with expected target triple x86_64-blog_os-14848397473379924386
  |
  = note: the following crate versions were found:
          crate `core`, target triple x86_64-blog_os-17931409391417382l9: /home/junmo/junmo2_os/target/sysroot/lib/rustlib/x86_64-blog_os/lib/libcore-db29a351b984920c.rlib
          crate `core`, target triple x86_64-blog_os-17931409391417382l9: /home/junmo/junmo2_os/target/sysroot/lib/rustlib/x86_64-blog_os/lib/libcore-db29a351b984920c.rmeta
```

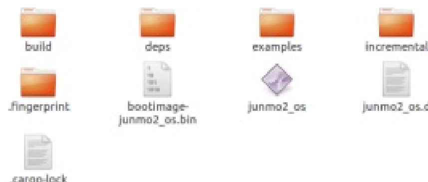
解决方法6-1：这里是因为toml缺少了最初的对panic参数进行复制的语句，所以出错，添加上即可

```toml
[package]
name = "junmo2_os"
version = "0.1.0"
authors = ["junmo"]
edition = "2018"

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html


# in Cargo.toml

[dependencies]
bootloader = "0.6.0"
```

现象6-1：修改成功后输入`cargo bootimage`的结果是很长一串，这里是最后一小段

```
   Compiling pulldown-cmark v0.0.3
   Compiling rustc_version v0.2.3
   Compiling rand v0.4.6
   Compiling raw-cpuid v6.1.0
   Compiling tempdir v0.3.7
   Compiling skeptic v0.5.0
   Compiling x86_64 v0.7.2
   Compiling fixedvec v0.2.3
    Finished release [optimized + debuginfo] target(s) in 53.49s
```

在HOME文件夹下面生成了一个bin文件

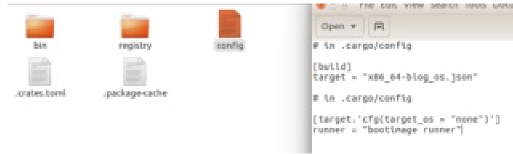


步骤7：启动内核，输入`qemu-system-x86_64 -drive format=raw,file=bootimage-blog_os.bin`
问题7-1：在哪个文件夹下输入？
解决方法7-1：在`HOME/JUNMO2_OS/TARGET/X86_64-BLOG_OS/DEBUG`文件夹下面，即bin文件所在的文件夹下打开终端并输入指令，注意需要按照生成的bin文件的文件名来修改输入的命令，如我这里就修改为了junmo2_os

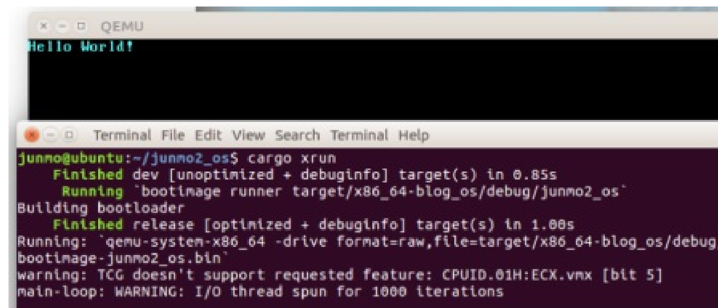
```
junmo@ubuntu:~/junmo2_os/target/x86_64-blog_os/debug$ qemu-system-x86_64 -drive
format=raw,file=bootimage-junmo2_os.bin
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
```

现象7-1：启动成功后如图，输出hello world语句

步骤8：使用cargo run
在cargo配置文件（config文件）中设置runner配置项，即输入如下代码;



这样就可以在终端中输入`cargo xrun`来编译内核并在qemu中启动，注意是在所创立的os文件夹下面打开终端，这里是junmo2_os文件夹下
现象8-1：成功启动qemu并打印hello world语句



三、实验重难点

（1）裸机程序：

要编写一个操作系统内核，我们需要不基于任何操作系统特性的代码。这意味着我们不能使用线程、文件、堆内存、网络、随机数、标准输出，或其它任何需要操作系统抽象和特定硬件的特性；这其实讲得通，因为我们正在编写自己的操作系统和硬件驱动。
实现这一点，意味着我们不能使用Rust标准库的大部分；但还有很多Rust特性是我们依然可以使用的。比如说，我们可以使用迭代器、闭包、模式匹配、Option、Result、字符串格式化，当然还有所有权系统。
为了用Rust编写一个操作系统内核，我们需要创建一个独立于操作系统的可执行程序。这样的可执行程序常被称作独立式可执行程序或裸机程序。
1、裸机程序main.rs程序详解：

```
#! [ no_std ] // don ’ t l i n k the Rust standard l i b r a r y
当前的包依然隐式地与标准库（std）链接，通过no_std来禁用这种链接
#! [ no_main ] // disable a l l Rust l e v e l entry points
重新定义函数的入口点，告诉rust编译器我们不适用预定义的入口点，这样我们就可以移除main函数。
use core : : panic : : PanicInfo ;
panicinfo参数包含了panic发生的文件名、代码行数和可选的错误信息，这个函数从不返回，所以他被标记为发散函数，发散函数的返回类型成为never类型，记为！。
#[ no_mangle ] // don ’ t mangle the name of t h i s function
使用no_mangle来标记start函数，来对他禁用名称重整，这确保rust编译器输出一个名为_start的函数，否则编译器最终生成名为_ZN3blog_os4_start7hb173fedf945531caE的函数，无法让链接器正确辨别。
pub extern ”C” fn _start () > ! {
将函数标记为extern “C”，告诉编译器这个函数应当使用c语言的调用约定，而不是rust语言的调用约定。函数名为_start是因为大多数系统默认使用这个名字作为入口点名称。
// t h i s function i s the entry point , since the linker looks for a function
// named ‘ _start ‘ by default
loop {}
}
这个函数的返回值类型为!，它定义了一个发散函数，或者说一个不允许返回的函数，因为这个入口点将不被任何函数调用，而直接被操作系统或者引导程序调用，所以作为函数返回的替换，这个入口点应该调用操作系统提供的exit系统调用函数。当一个独立式可执行程序返回时，不会留下任何需要做的事情，现在，我们只需要添加一个无限循环，用来满足对返回值类型的需求。
/// This function i s called on panic .
#[ panic_handler ]
属性定义了一个函数，他会在panic发生时被调用。标准库中提供了panic处理函数，但在当前的no_std环境中，我们需要定义一个自己的panic处理函数
fn panic ( _info : &PanicInfo ) > ! {
loop {}
}
```

2、裸机程序.toml文件：

```
[ package ]
name = ” rui_os ”
version = ” 0.1.0 ”
authors = [ ” rui <rui@hnu . edu . cn>” ]
edition = ”2018”
这里是操作系统的一些基本信息，可以随意改动
# the p r o f i l e used for ‘ cargo build ‘
[ p r o f i l e . dev ]
panic = ” abort ” # disable stack unwinding on panic
将dev配置的panic策略设为abort，dev配置适用于cargo build，现在编译器应该不再要求我们提供eh_personality语言项实现。禁用panic时栈展开
 # the p r o f i l e used for ‘ cargo build’ --release
[ p r o f i l e . r e l e a s e ]
panic = ” abort ” # disable stack unwinding on panic
将release配置的panic策略设为abort，releas配置适用于cargo build --release，现在编译器应该不再要求我们提供eh_personality语言项实现。禁用panic时栈展开
# See more keys and their d e f i n i t i o n s at https : //doc . rust lang . org/cargo/reference/manifest . html
[ dependencies ]
```

（2）实现最小内核：

基于x86架构，使用Rust语言，编写一个最小化的64位内核。我们将从上一章构建的独立式可执行程序开始，构建自己的内核；它将能够向显示器打印字符串，并被打包为一个能够引导启动的磁盘映像。
当我们启动电脑时，主板ROM内存储的固件将会运行：它将负责电脑的上电自检，可用内存的检测，以及CPU和其它硬件的预加载。这之后，它将寻找一个可引导的存储介质，并开始引导启动其中的内核。
BIOS启动：当电脑启动时，主板上特殊的闪存中存储的BIOS固件将被加载。BIOS固件将会上电自检、初始化硬件，然后它将寻找一个可引导的存储介质。如果找到了，那电脑的控制权将被转交给引导程序：一段存储在存储介质的开头的、512字节长度的程序片段。大多数的引导程序长度都大于512字节——所以通常情况下，引导程序都被切分为一段优先启动、长度不超过512字节、存储在介质开头的第一阶段引导程序，和一段随后由其加载的、长度可能较长、存储在其它位置的第二阶段引导程序。引导程序必须决定内核的位置，并将内核加载到内存。引导程序还需要将CPU从16位的实模式，先切换到32位的保护模式，最终切换到64位的长模式：此时，所有的64位寄存器和整个主内存才能被访问。引导程序的第三个作用，是从BIOS查询特定的信息，并将其传递到内核；如查询和传递内存映射表。推荐bootimage工具——它能够自动而方便地为你的内核准备一个引导程序。

（3）实现最小内核main.rs程序详解：

```
#! [ no_std ] // don ’ t l i n k the Rust standard l i b r a r y
当前的包依然隐式地与标准库（std）链接，通过no_std来禁用这种链接
#! [ no_main ] // disable a l l Rust l e v e l entry points
重新定义函数的入口点，告诉rust编译器不使用预定义的入口点，就可以移除main函数。
use core : : panic : : PanicInfo ;
panicinfo参数包含了panic发生的文件名、代码行数和可选的错误信息，这个函数从不返回，所以他被标记为发散函数，发散函数的返回类型成为never类型，记为！。
s t a t i c HELLO: &[u8 ] = b”Hello World ! ” ;
预先定义了一个字节字符串类型的静态变量，名为HELLO。
#[ no_mangle ] // don ’ t mangle the name of t h i s function
使用no_mangle来标记start函数，来对他禁用名称重整，这确保rust编译器输出一个名为_start的函数，否则编译器无法正确辨别。
pub extern ”C” fn _start () > ! {
将函数标记为extern “C”，告诉编译器这个函数应当使用c语言的调用约定，而不是rust语言的调用约定。函数名为_start是因为大多数系统默认使用这个名字作为入口点名称。
l e t vga_buffer = 0xb8000 as *mut u8 ;
首先将整数0xb80000转换为一个裸指针
for ( i , &byte ) in HELLO. i t e r ( ) . enumerate () {
unsafe { *vga_buffer . o f f s e t ( i as i s i z e * 2) = byte ;
*vga_buffer . o f f s e t ( i as i s i z e * 2 + 1) = 0xb ; }
}
迭代HELLO的每个字节，使用enumerate获得一个额外的序号变量i，在for语句的循环体中，使用offest偏移裸指针，解引用它，来将字符串的每个字节和对应的颜色字节—0xb代表淡青色—写入内存位置。所有的裸指针内存操作都被一个unsafe语句块包围。这是因为，此时编译器不能确保我们创建的裸指针是有效的；一个裸指针可能指向任何一个你内存位置；直接解引用并写入它，也许会损坏正常的数据。使用unsafe语句块时，程序员其实在告诉编译器，自己保证语句块内的操作是有效的。
loop {}
}
/// This function i s called on panic .
#[ panic_handler ]
属性定义了一个函数，他会在panic发生时被调用。标准库中提供了panic处理函数，但在当前的no_std环境中，我们需要定义一个自己的panic处理函数
fn panic ( _info : &PanicInfo ) > ! {
loop {}
}
```

（4）实现最小内核.toml文件详解：

```
[ package ]
name = ” rui_os ”
version = ” 0.1.0 ”
authors = [ ” rui <rui@hnu . edu . cn>” ]
edition = ”2018”
这里是操作系统的一些基本信息，可以随意改动
# the p r o f i l e used for ‘ cargo build ‘
[ p r o f i l e . dev ]
panic = ” abort ” # disable stack unwinding on panic
将dev配置的panic策略设为abort，dev配置适用于cargo build，现在编译器应该不再要求我们提供eh_personality语言项实现。禁用panic时栈展开
 # the p r o f i l e used for ‘ cargo build’ --release
[ p r o f i l e . r e l e a s e ]
panic = ” abort ” # disable stack unwinding on panic
将release配置的panic策略设为abort，releas配置适用于cargo build --release，现在编译器应该不再要求我们提供eh_personality语言项实现。禁用panic时栈展开
# See more keys and their d e f i n i t i o n s at https : //doc . rust lang . org/cargo/reference/manifest . html
[ dependencies ]
bootloader = ” 0.6.0 ”
为引导程序添加一个依赖项，用来启动我们的内核。
5、实现最小内核配置文件.json详解:
{
”llvm-target ” : ”x86_64 unknown none” ,
”data-layout ” : ”e m: e i64 :64 f80 :128 n8:16:32:64 S128” ,
本配置项定义了不同的整数、浮点数、指针类型的长度
”arch” : ”x86_64” ,
” target-endian” : ” l i t t l e ” ,
” target-pointer-width” : ”64” ,
rust用作条件变编译的配置项
” target-c-int width” : ”32” ,
”os ” : ”none” ,
” executables ” : true ,
” linker-flavor ” : ” ld . l l d ” ,
” linker ” : ” rust l l d ” ,
这两句不适用平台默认提供的链接器，使用跨平台的LLD链接器
”panic-strategy ” : ” abort ” ,
表示编译目标不支持panic时的栈展开，直接在panic时终止
” disable-redzone ” : true ,
禁用一个与红区（redzone）有关的栈指针优化
” features ” : ” -mmx,-sse ,+ sofe-f l o a t ”
启用或禁用某个目标CPU特征，通过在前面加-号，将mms和sse特征禁用，这两个特征决定了是否支持单指令多数据流相关指令；添加前缀+号，启用了soft-float特征
}
当机器启动时，引导程序将会读取并解析拼接在其后的ELF文件。这之后，它将把程序片段映射到分页表中的虚拟地址，清零BSS段，还将创建一个栈。最终它将读取入口点地址——我们程序中_start函数的位置——并跳转到这个位置。
```

四、实验心得体会

这次实验我做了很长时间，因为一方面这个实验本身有一定的难度，另一方面我一开始也有点小看了这个实验，才导致这个实验做了将近一个周才真正做完。其中我遇到了各种各样的困难，光是截图就截了74张，其中有关于pdf的步骤的截图也就只占了50张左右，剩下的都是我遇到的各种各样问题的截图。足以看到做这个实验的时候我遇到了多么大的困难。
但还好，我都克服了，还找到了pdf中的三个错误（两个是代码的拼写错误，一个是安装qemu的时候换了一种更简单的安装方法），对pdf做出了自己的修正。光是这一点我就感觉很有成就感了。
而且，我发现这次的实验和我正在做的《操作系统实验课—30天自制操作系统》的联系比较大。30天自制操作系统也是想要生成bin映像文件来实现操作系统，不过两个之间又有不同。30天自制操作系统对于操作系统制作的过程似乎并没有太在意，更在意的是如何构建出一个gui界面（学到第三天的时候的见解），而这次的实验可以看到有很多的命令行指令，好像更偏向于如何从无到有的建立起一个操作系统。所以，两门课都很重要，都要学好。
这次的实验教会了我很多，包括如何实现一个最小内核，如何创建裸机程序。而且，感觉实现了最小内核之后，后面应该就是添加各种各样的系统调用指令和函数，将操作系统的其他基础操作进行实现，应该会更有意思，加油吧。