

实时操作系统基本知识

- [实时操作系统基本知识](#)
 - [Linux实时操作系统](#)
 - [双内核操作系统基本知识](#)
 - [Xenomai操作系统](#)
 - [rros](#)
 - [rros编译](#)
 - [docker安装及拉取代码](#)
 - [rros编译过程](#)
 - [使用Qemu进行模拟同时使用gdb对进行Debug](#)
 - [编译等级](#)
 - [使用rust-gdb调试](#)
 - [使用vscode的调试](#)
 - [lab内容: 在内核中打印自己的信息](#)
 - [提交](#)
 - [引用](#)

Linux实时操作系统

Linux是一个通用操作系统，而不是一个实时操作系统。目前有多种方法可以让Linux成为一个实时操作系统或者具备实时操作系统的能力：

- 一个广泛使用的方案是让给实时操作系统提供兼容POSIX的API接口来改善生态问题，但是这个方法在面對大型项目的移植时的效果不好，也没有解决性能问题；
- 另一条技术路线是可以对Linux内核采用抢占补丁，但是这个方法只能让Linux成为一个软实时的操作系统，同时没有任何的隔离措施，不利于稳定性；
- 还可以采用虚拟机来同时在一个主机上同时运行多个内核，一个是较为简单的硬实时内核，另一个是Linux内核，这个方法的稳定性很强，但是由于采用了硬件虚拟化，实时性和性能受损，同时在Linux内核上运行的应用不具备实时性能，在实时内核上运行的应用不能享受到Linux生态的好处，两者之间的数据交互比较困难；
- 最后一条路线是采用双内核的方法，在一个内核空间里面同时运行两个内核，实时内核和Linux同时并行，实时内核的优先级更高，Linux内核作为一个idle任务在实时内核中调度，这个方法同时兼顾了实时性，性能和应用生态，实时任务可以同时利用Linux的生态和实时应用的能力，但是这个方法的缺点是稳定性不足，因为两个内核同处一个地址空间，所以如果Linux内核出现故障，没有任何的隔离措施，很容易导致操作系统崩溃；

我们的rros（rust-based real operating system）就是采用了双内核的技术路线，下面首先介绍一下双内核操作系统。

双内核操作系统基本知识

双内核操作系统主要分为两部分，一个是硬件虚拟层（HAL），主要完成中断虚拟化等相关工作；第二个是实时内核，主要负责处理实时请求，和Linux内核在逻辑上是并列的关系，但是优先级要比Linux内核高。

双内核的实现主要分为RTLinux，RTAI，Xenomai三个项目。双内核最早的实现RTLinux是在[1]这篇论文中提出的。硬件虚拟层是双内核路线中的关键一环，RTHAL是在RTLinux的论文中提出来的，被另一个实时操作系统

RTAI仿照实现，但是RTLinux的项目组后来申请了专利，RTAI被迫采用了其他硬件虚拟层技术ADEOS。ADEOS是在[2]这篇论文中提出的，采用不同方法换了RT-Linux提出的RTHAL，规避了专利问题。所以后来RTAI和Xenomai社区采用了ADEOS。

拓展阅读 ADEOS的实现细节在[3]中可以看到；
ADEOS和RTHAL两种硬件虚拟层技术的比较在[4]中可以看到；

RTLinux自从被同类竞品VxWorks收购后，已经从开源逐步走向关停。而Xenomai和RTAI两个项目在一段时间有过短暂的合并。但是后来因为开发的目标不同，两个项目又逐渐分离。rros主要仿照的就是Xenomai操作系统。

Xenomai操作系统

Xenomai目前已经进展到4.0了。Xenomai4.0主体是两个部分，硬件中断层dovetail和实时内核evl。dovetail主要是以代码树形式提供，直接修改了Linux内核的代码，作用是根据优先级将中断分发给cobalt内核和evl内核。实时内核evl则作为一个module插入到linux系统中，和Linux内核一起启动，启动后接管整个操作系统。

rros

rros采用了dovetail硬件中断层，用rust重写了实时内核evl，下面我们将会介绍rros的基本情况。

项目的代码树主体结构如下，相对于Linux代码树新增的文件用*标出，一些重要的目录或者文件我们加以解释：

```
.
├── arch
├── block
├── certs
├── COPYING
├── CREDITS
├── crypto
├── .config
├── Documentation
├── drivers
├── fs
├── gr
├── include
├── init
├── io_uring
├── ipc
├── Kbuild
├── Kconfig
├── kernel
├── acct.c
├── ...
├── rros
├── built-in.a
├── clock.rs
├── clock_test.rs
├── crossing.rs
├── double_linked_list_test.rs
```

这个文件中包含了Linux编译时Linux项目的文档

* 包含了运行rust-gdb命令的文件

Linux内核的主要代码

* rros实时内核的主要代码

```

├── factory.rs
├── fifo.rs
├── fifo_test.rs
├── file.rs
├── idle.rs
├── init.o
├── init.rs
├── libinit.rmeta
├── list.rs
├── list_test.rs
├── lock.rs
├── Makefile
├── memory.rs          内存子系统
├── modules.order
├── monitor.rs
├── net.rs
├── queue.rs
├── sched.rs          调度子系统
├── sched_test.rs
├── stat.rs
├── stat_test.rs
├── syscall.rs
├── test.rs
├── thread.rs         线程子系统
├── thread_test.rs
├── tick.rs           tick子系统
├── timeout.rs
├── timer.rs          时钟子系统
├── timer_test.rs
├── wait.rs
├── weak.rs
├── ...
├── workqueue.o
├── lib
├── LICENSES
├── MAINTAINERS
├── Makefile
├── mm
├── modules.builtin
├── modules.builtin.modinfo
├── modules.order
├── Module.symvers
├── net
├── README
├── rust              * rust-for-linux的代码
├── samples           包含了各个子系统的一些示例代码
├── scripts
├── security
├── sound
├── System.map
├── tools
├── usr
├── virt
├── vmlinux

```

```
|— vmlinux.a  
|— vmlinux.o
```

因为dovetail硬件中断层已经合入到代码树中，通过[git log](#)的历史记录是看不出来的，如果想要知道dovetail修改了哪些内容，可以从[patch-5.15.9-dovetail1.patch](#)这个patch中看到。

rust的支持是通过rust-for-linux (rfl) 项目[5]，rfl目前已经合入linux主线，由于历史原因，我们项目中rfl的支持是通过补丁的形式进行的，和目前主线上的rfl不兼容。rfl项目支持我们用rust写linux的驱动，我们的rros就是以驱动的形式加载到linux内核中的。

rros编译

为了方便大家做实验，我们以docker的形式提供一个可用的环境，大家只需要安装docker并拉取镜像，然后按照我们的编译说明进行编译，就可以做后续的实验了。

docker安装及拉取代码

docker在windows/linux/mac上都可以直接安装，主要参考[官方的文档](#)，具体步骤方法如下：

- 在windows上安装
 - 下载[docker desktop](#)，并点击安装。
 - 安装完docker后，如果提示因为wsl2没有安装不能正常启动的话，这是因为在windows上使用docker需要开启wsl2或者hyper-v相关的组件，我们这里采用wsl2，这部分内容参考微软的[官方说明](#)。
 - 以管理员身份打开 PowerShell (“开始”菜单 > “PowerShell” > 单击右键 > “以管理员身份运行”)，然后输入以下命令：

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart  
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

- 下载[wsl更新包](#)，并安装执行。
 - 重启电脑，并启动docker desktop，就可以正常启动了
- 在linux ubuntu/mac上安装
 - 这里考虑到linux上安装时可能没有图形化界面，所以下面用命令行说明
 - linux上运行docker的原理是使用kvm虚拟化技术，可以使用下列命令检测linux是否满足docker的条件

```
lsmod | grep kvm
```

正确的输出如下：

- 对linux软件包安装地址进行换源

```
sudo add-apt-repository "deb [arch=amd64]
https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb_release -cs)
stable"
sudo apt-get update
```

如果换源不成功，出现GPG error，可以参考这个[教程](#)。

- 安装docker

```
sudo apt-get install docker-ce
```

- docker换源

```
vim /etc/docker/daemon.json
# 加入下面的内容
# {
#   "registry-mirrors": ["https://akchsm1h.mirror.aliyuncs.com"]
# }
```

- 检查是否可以正常执行

```
docker run hello-world
```

安装完成后，打开命令行窗口，使用`docker pull 1543306408/rros_lab`命令来拉取rros docker的镜像image。接着使用`docker run -itd --security-opt seccomp=unconfined --name rros_lab 1543306408/rros_lab /bin/bash`来运行一个名为rros_lab的container。

最后我们利用vscode来完成后续实验。

- 如果你的docker是运行在本机上，而不是远程的Linux服务器，只需要在vscode应用市场中安装dev-container插件：
- 点击插件后，我们就可以看到我们运行起来的docker了，然后点击Attach in New Window进入我们的docker
- 然后在container中打开我们项目的文件夹
- 输入/data/bupt-rtos/rros

远端的Linux配置方法如下：

- 如果你的docker运行在远程的Linux服务器，需要安装remote-ssh插件
- 需要先配置一下ssh
- 将下图的ip_address换成Linux服务器的ip地址，ssh_port设置为对应ssh的服务端口（一般是22），user_name替换成Linux服务器的用户名字

- 最后打开配置好的远端服务器，之后和在本地vscode打开docker容器的步骤一致 如果发现打开插件后没有找到容器信息，有可能是因为所用账户没有docker的权限，在/etc/group中找到docker的用户组，添加自己的用户名。

rros编译过程

首先，rros的编译和linux编译的方法相似，都是通过Kconfig系统。对于rros比linux多出的dovetail和rfl子系统，都可以通过在主目录下执行配置menuconfig的命令进行开启，对于其他的子系统也可以在同时进行配置，配置好的结果会保存到.config文件中。为了方便大家做后面的实验，我们提供一个已经配置好的.config文件，不需要大家手动配置。并且这个.config中的选项经过了剪裁，所以操作系统编译的速度会大大加快。如果大家想要体会手动配置config的过程，可以参考下面的编译tips中第四点。

然后，可以用`make LLVM=1 -j80 >compile.txt 2>&1 && tail -10 compile.txt`对整个操作系统进行编译，这个命令后面的重定向是由于目前项目中有大量的warning没有被消除，所以我们最好把编译的结果保存到一个compile.txt文件中。如果代码中没有错误，这个命令会输出10行编译信息，如下图所示。

1. 下面的/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase路径和大家看到的/data/bupt-rtos/rros路径是等价的。
2. 如果你是从头编译的话，这个命令的输出结果如下

如果代码中有错误（大家可以试着改动一下kernel/rros/init.rs中的代码，比如注释init_core的代码），可以用finderr.py脚本对错误进行过滤，将脚本下载到项目工程/data/bupt-rtos/rros中，执行命令是`python3 finderr.py compile.txt`，最后利用`cat result`查看result文件中的错误信息。

最后，如果编译成功了，我们就可以在主目录下看到最新生成的vmlinux文件，然后用qemu去模拟运行这个操作系统，使用qemu运行操作系统的部分，会在[使用Qemu进行模拟](#)小节进行讲解。

编译tips：

1. 可以注意到make命令中使用了LLVM=1，这个参数会让编译过程中使用llvm而不是gcc，这是因为我们需要rfl项目的支持，而rfl需要通过llvm才能成功编译rust。所以我们在rros的大部分编译命令中都需要加入LLVM=1。
 2. rros目标的平台是在arm64，所以涉及到交叉编译的知识，交叉编译就是编译代码的环境和执行代码的环境不在一个平台上，比如在x86_64平台下，编译arm64的目标文件，我们在编译时通过使用两个环境变量来说明这两个信息，rros编译时会自动读取这两个环境变量来获得这部分信息。
 3. 一些docker环境中隐藏的细节：docker环境中配置了可以支持交叉编译环境的gcc，llvm，qemu，gdb，objdump等编译相关的工具，以及cmake，rust等开发相关的工具，有些软件是从源码编译安装的，因为ubuntu/centos的apt-get和yum包管理工具会限制这些软件的版本。
 4. config如何手动配置：config中需要手动配置的主要是分为三部分：如何开启dovetail和rfl，如何开启debug相关的选项，如何裁剪和rros内核无关的config，下面介绍前两部分。
- 执行`make LLVM=1 menuconfig`配置命令
 - Kernel Features中开启Dovetail interface
 - General setup中开启Rust support
 - debug相关的config

使用Qemu进行模拟同时使用gdb对进行Debug

编译等级

调试时可能需要将编译等级调低。我们所给的配置文件`.config`应该已经配置，若没有，你可以手动配置一下。

输入`menuconfig`

```
.config - Linux/arm64 5.13.0 Kernel Configuration

Linux/arm64 5.13.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

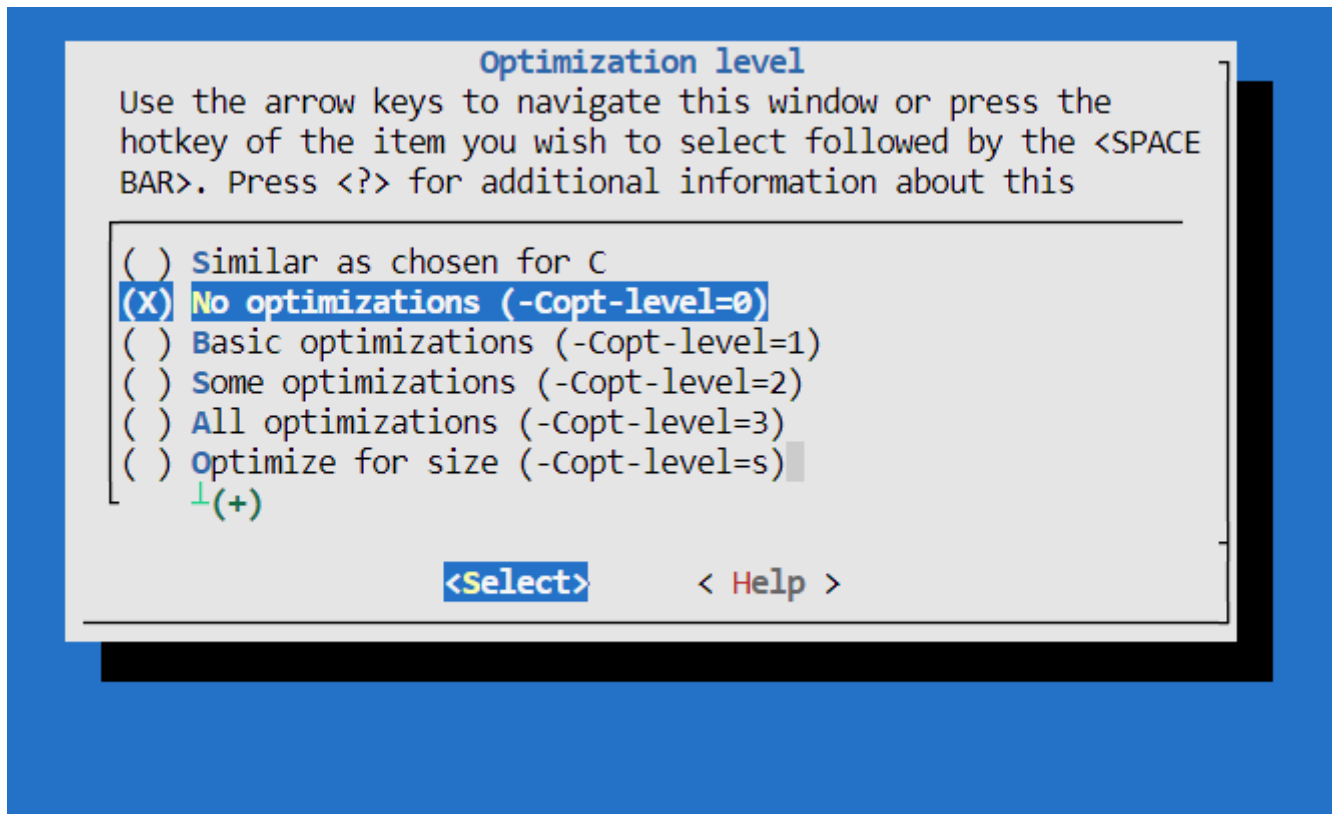
[*] General setup --->
[*] Support DMA zone
[ ] Support DMA32 zone
Platform selection --->
Kernel Features --->
Boot options --->
Power management options --->
CPU Power Management --->
Firmware Drivers --->
[ ] Virtualization ----
General architecture-dependent options --->
[*] Enable loadable module support --->
[ ] Enable the block layer ----
[ ] Executable file formats --->
↓(+)
```

如果出现问题，可能有两种情况：

- 把窗口拉大一点
- 看bash环境变量是否有`$CROSS_COMPILE`，`$ARCH`。这些在之前应该已经配置过。

```
# 在 ~/.bashrc 里添加
export CROSS_COMPILE=aarch64-linux-gnu-
export ARCH=arm64
```

把`kernel hacking > Rust Hacking > Optimization level > debug-level`调到最低



按空格确定向右选择Exit，按回车退出，选择保存Yes.

使用rust-gdb调试

在docker中我们已经安装好qemu了，所以可以直接使用。并且qemu启动所需要的文件系统已经在docker中准备完成。

我们只需要同时打开两个命令行窗口，然后左边运行qemu的命令，右边运行gdb的命令，我们就可以完成对rros的debug工作。

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096 -s -S
```

最后的两个标志 -s 表示启动gdb server，-S表示不要立刻执行指令，按c可以开始执行。

修改环境中的gr脚本为以下代码：

```
gdb-multiarch \
--tui vmlinux \
-ex "target remote :1234" \
-ex "set architecture aarch64" \
-ex "set language rust" \
-ex "set auto-load safe-path /"
```

这是由于目前docker环境中的rust-gdb不支持交叉平台编译。

然后运行此脚本（注意中间有空格）

```
./gr
```

如果不想debug，只想用qemu对操作系统进行模拟运行，那么只需要打开一个窗口，然后去掉-s -S这两个gdb相关的参数，运行下列命令即可

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
./arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096
```

先在qemu所在窗口执行上述命令，然后在gdb窗口执行上述命令，就可以成功运行

在gdb窗口中按c并回车，可以看到rros操作系统就可以正常执行了。

gdb的具体指令和上学期bos lab中的相关指令一致，具体内容可以回看上学期ppt。

下面给出了gdb的一些常见命令

命令	作用
c	跳到下一个断点
b 文件名:行号	设置断点
p 变量名	打印变量
x/	打印地址下的数据
finish	跳到当前函数的结尾
frame	查看栈帧
n	next，下一步，不进入函数
s	step in， 下一步，但是可能进入函数

举一个例子，如果想要在操作系统执行之前打一个断点，可以用b kernel/rros/init.rs:135在kernel/rros/init.rs这个文件的135行打一个断点，然后再按c就能执行到这一行了。

执行结束时，退出qemu时，先按ctrl+a+x；退出gdb时，按ctrl+d。

使用vscode的调试

也可以给vscode添加配置文件。

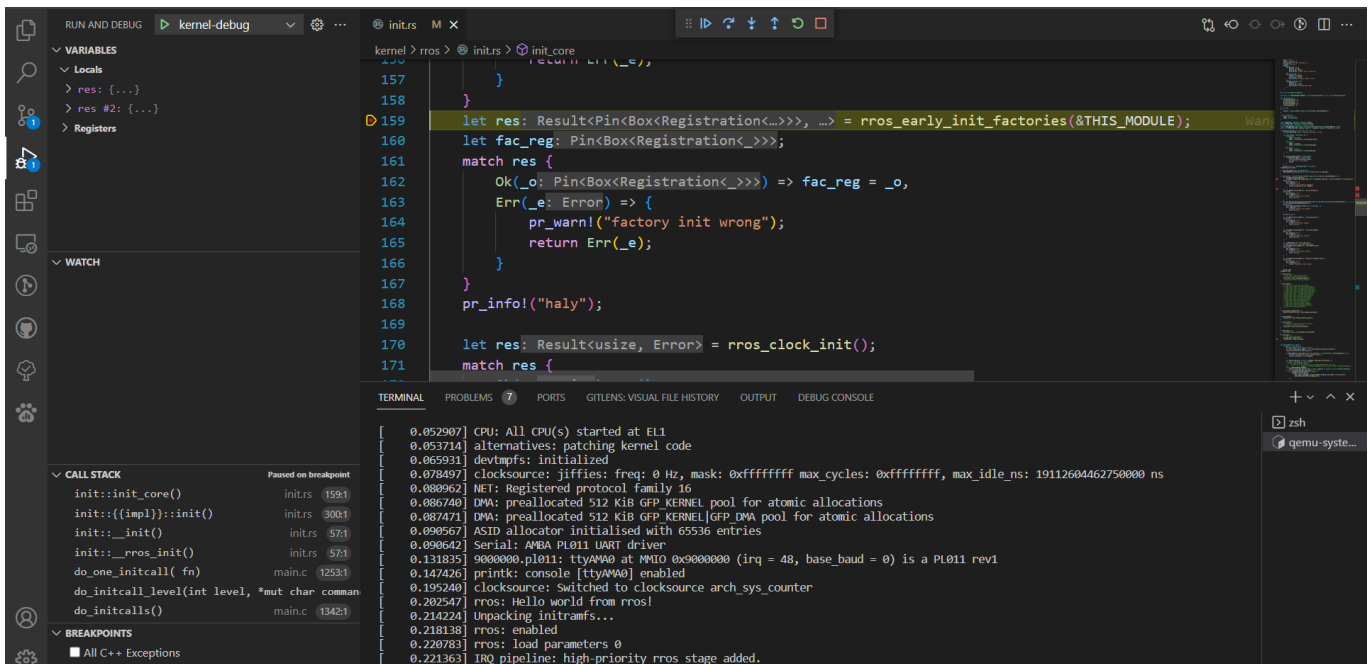
首先，同样还是在命令行启动调试的qemu：

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096 -s -S
```

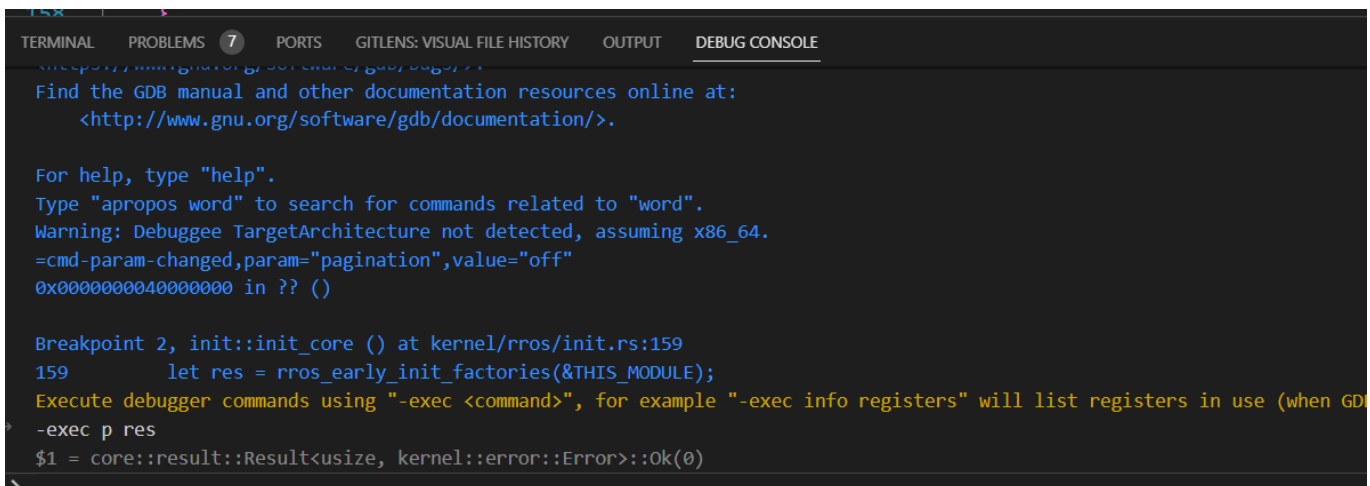
然后，在项目根目录的.vscode文件夹中，打开.vscode/launch.json(没有的话新建一个)，把下面的配置粘贴进去：

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "kernel-debug",
      "type": "cppdbg",
      "request": "launch",
      "miDebuggerServerAddress": "127.0.0.1:1234",
      "program": "${workspaceFolder}/vmlinux",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "logging": {
        "engineLogging": false
      },
      "MIMode": "gdb",
      // "miDebuggerPath" : "/root/.cargo/bin/rust-gdb",
      "miDebuggerPath": "/usr/bin/gdb-multiarch",
      "setupCommands": [
        {
          "description": "set language rust",
          "text": "set lang rust",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

在行号处点击断点，按F5开始调试



如果需要使用gdb命令，可以在下面DEBUG CONSOLE，输入-exec {gdb命令}执行



lab内容: 在内核中打印自己的信息

为了验证大家成功运行并编译内核，需要在内核启动时打印一个特定的字符串加自己的学号，修改位于 `kernel/rros/init.rs` 中的第252行，打印的内容为 `Hello world from rros! I'm 2020xxxxxx`，其中 `2020xxxxxx` 替换为自己的学号。如果你的实现正确，运行 `qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd ../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append "rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096 | grep "Hello world from rros! I'm 2022xxxxxx"`，就可以看到如下结果：

提交

本次实验需要提交自己的代码相对于commit号为 `ed16617d67a5a0cd91964c7decf428ac2c41486b` 的patch，以及一个文档，文档里面最少记录两个截图：成功编译内核的截图，成功用qemu运行内核并且使用gdb调试的截图。

引用

- [1] A linux-based real-time operating system
- [2] Adaptive domain environment for operating systems
- [3] Life with adoes
- [4] Study and Comparison of the RTHAL-based and ADEOS-based RTAI Real-time Solutions for Linux
- [5] <https://github.com/Rust-for-Linux/linux>