

# 浙江大学

## 本科实验报告

课程名称: 操作系统

---

姓 名:

---

学 院: 计算机科学与技术学院

---

系: 计算机科学与技术系

---

专 业: 计算机科学与技术

---

学 号:

---

指导教师: 夏莹杰

---

2022 年 9 月 25 日

浙江大学操作系统实验报告

实验名称: GDB + QEMU 调试 64 位 RISC-V LINUX

电子邮件地址:

手机:

实验地点: 曹西 503

实验日期: 2022 年 9 月 22 日

## 一、实验目的和要求

1. 配置 Linux 环境
2. 安装 docker, 并载入实验环境镜像
3. 下载 linux kernel 源码, 并在 docker 容器中进行编译
4. 加载实验环境镜像到容器中
5. 使用 gdb 调试内核运行情况

## 二、实验过程

1. 搭建 Docker 环境

① 安装 Docker 环境: 此处我直接选用了 docker 官方阿里云映像的安装脚本, 即 `$curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun`。等待一段时间, 安装好后即可使用 docker 命令。

```
zarin@ZaricCarpathia:~$ docker

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default
                        "/home/zarin/.docker")
```

② 使用 `docker load` 命令载入实验环境镜像 `oslab.tar`。使用 `docker images` 命令确认实验镜像已载入成功。

```

zarin@ZaricCarpathia:~$ sudo docker load < oslab.tar
[sudo] zarin 的密码:
36ffdc4c77: Loading layer 75.15MB/75.15MB
8b59f73b2a6e: Loading layer 3.577GB/3.577GB
Loaded image: oslab:2021
zarin@ZaricCarpathia:~$ sudo docker images
REPOSITORY          TAG             IMAGE ID         CREATED          SIZE
oslab                2021            8c15472cd111    7 months ago    3.63GB
hello-world         latest         feb5d9fea6a5    12 months ago   13.3kB

```

- ③ 从该镜像创建一个容器 oslab，并把 \${HOME} 目录挂载到容器中的 /debug 目录中。可通过 ls 等命令查看容器内部目录结构，确认挂载成功。

```

zarin@ZaricCarpathia:~$ sudo docker run --name oslab -it -v ${HOME}:/debug oslab:2021 bash
[sudo] zarin 的密码:
root@61868c8ae428:/# ls
bin      dev      lib      libx32  opt      riscv-glibc  sbin     tmp
boot     etc      lib32    media   proc      root          srv       usr
debug    home     lib64    mnt     riscv-elf  run           sys       var
root@61868c8ae428:/#







```

## 2. 获取 Linux 源码和已经编译好的文件系统



- ① 从 kernel.org 下载获取 linux 源码。

 linux-5.19.9.tar.xz 131.7 MB

- ② 使用 git 工具 clone 课程仓库，以获得根文件系统的镜像。

/ os22fall-stu		:	Q	🔍	▼	☰
名称					大小	
 docs					4 个项目	
 src					1 个项目	
 LICENSE					35.1 kB	
 mkdocs.yml					1.1 kB	
 README.en.md					826 字节	
 README.md					81 字节	

- ③ 将 linux kernel 源码解压到 os22fall-stu/src/lab0 下。

/ os22fall-stu / src / lab0		:	Q	☐	▼	☰
名称			大小			
 linux			39个项目			
 rootfs.img			16.8 MB			

### 3. 编译 Linux 内核

- ① 从终端进入 oslab 容器。

```
zarin@ZaricCarpathia:~$ sudo docker start oslab
oslab
zarin@ZaricCarpathia:~$ sudo docker exec -it oslab bash
root@61868c8ae428:/#
```

- ② 进入 debug/os22fall-stu/src/lab0/linux 目录。

```
root@61868c8ae428:/debug/os22fall-stu/src/lab0/linux# ls
COPYING      LICENSES      arch      fs      kernel  scripts  virt
CREDITS      MAINTAINERS  block     include  lib      security
Documentation Makefile      certs     init     mm       sound
Kbuild       Module.symvers crypto      io_uring  net      tools
Kconfig      README       drivers    ipc      samples  usr
```

- ③ 使用交叉编译工具编译 linux kernel，等待直到完成。

```
root@61868c8ae428:/debug/os22fall-stu/src/lab0/linux# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig
HOSTCC  scripts/basic/fixdep
HOSTCC  scripts/kconfig/conf.o

root@61868c8ae428:/debug/os22fall-stu/src/lab0/linux# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j4
HOSTCC  scripts/dtc/flattree.o
HOSTCC  scripts/dtc/dtc.o
```

### 4. 使用 QEMU 运行内核

- ① 回到上一层目录（即 lab0），使用 qemu 命令运行内核。启动终端，可以通过 ls 等命令验证内核是否已成功编译并启动。



- ③ 继续进行其他指令的操作，以熟悉 gdb 调试过程。这些指令及其结果可详见思考题。

```
[ 1.202278] VFS: Mounted root (ext4 filesystem) readonly on device 254:0.
[ 1.204882] devtmpfs: mounted
[ 1.230549] Freeing unused kernel image (initmem) memory: 2172K
[ 1.232133] Run /sbin/init as init process

Please press Enter to activate this console.

(gdb) ni
0xffffffff808080ba in start_kernel ()
(gdb) n
Single stepping until exit from function start_kernel,
which has no line number information.
```

## 三、讨论和心得

由于实验指导书中在容器内和容器外的命令都用的是\$提示符，我一开始以为后续操作需要在自己的 linux 系统中自行安装工具链进行。等走到编译的步骤时，因为没有指定的编译器（riscv64-unknown-linux-gnu-），在思考许久之后才明白过来这个编译器是置在容器内的。所以以后希望能把指导书里容器内和容器外的提示符区分一下，比如容器内用#……

除此之外，实验过程还是很顺畅的。

## 四、思考题

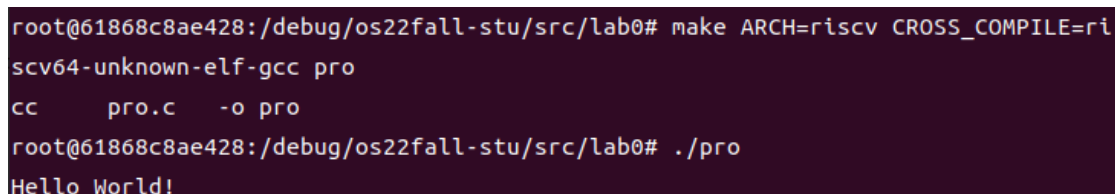
1. 使用 riscv64-unknown-elf-gcc 编译单个 .c 文件

- ① 编写程序如下：



```
1 #include <stdio.h>
2
3 signed main(){
4     int a=1;
5     int b=2;
6     int c=a;
7     a=b;
8     b=c;
9     printf("Hello World!\n");
10    return 0;
11 }
```

- ② 从终端进入容器，在容器内键入命令如下（采用交叉编译，因为直接编译无法在容器内进行）：



```
root@61868c8ae428:/debug/os22fall-stu/src/lab0# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-elf-gcc pro
cc    pro.c  -o pro
root@61868c8ae428:/debug/os22fall-stu/src/lab0# ./pro
Hello World!
```

- ③ 键入运行命令后程序正常执行正常输出，可见步骤成功。

2. 使用 riscv64-unknown-elf-objdump 反汇编 1 中得到的编译产物

- ① 重新直接编译 pro.c:

```
root@61868c8ae428:/debug/os22fall-stu/src/lab0# riscv64-unknown-elf-gcc -o pro p
ro.c
```

- ② 使用 riscv64-unknown-elf-objdump 进行反编译:

```
root@61868c8ae428:/debug/os22fall-stu/src/lab0# riscv64-unknown-elf-objdump -d p
ro > pro.asm
```

- ③ 可以查看生成的输出文件，以确认反编译过程正确。

```
66 0000000000001014a <main>:
67 1014a: 1101          addi    sp,sp,-32
68 1014c: ec06          sd      ra,24(sp)
69 1014e: e822          sd      s0,16(sp)
70 10150: 1000          addi    s0,sp,32
71 10152: 4785          li      a5,1
72 10154: fef42623      sw      a5,-20(s0)
73 10158: 4789          li      a5,2
74 1015a: fef42423      sw      a5,-24(s0)
75 1015e: fec42783      lw      a5,-20(s0)
76 10162: fef42223      sw      a5,-28(s0)
77 10166: fe842783      lw      a5,-24(s0)
78 1016a: fef42623      sw      a5,-20(s0)
79 1016e: fe442783      lw      a5,-28(s0)
80 10172: fef42423      sw      a5,-24(s0)
81 10176: 67c9          lui     a5,0x12
82 10178: 62878513      addi    a0,a5,1576 # 12628 <__errno+0x8>
83 1017c: 1c0000ef      jal     ra,1033c <puts>
84 10180: 4781          li      a5,0
85 10182: 853e          mv      a0,a5
86 10184: 60e2          ld      ra,24(sp)
87 10186: 6442          ld      s0,16(sp)
88 10188: 6105          addi    sp,sp,32
89 1018a: 8082          ret
```

### 3. 调试 Linux 时:

- ① 在 GDB 中查看汇编代码

在 gdb 中输入 layout asm 即可。

```
>0xfffffffff8000351e <arch_cpu_idle+14> csrsi sstatus,2
0xfffffffff80003522 <arch_cpu_idle+18> ld s0,8(sp)
0xfffffffff80003524 <arch_cpu_idle+20> addi sp,sp,16
0xfffffffff80003526 <arch_cpu_idle+22> ret
0xfffffffff80003528 <__show_regs> addi sp,sp,-32
0xfffffffff8000352a <__show_regs+2> sd s0,16(sp)
0xfffffffff8000352c <__show_regs+4> sd s1,8(sp)
0xfffffffff8000352e <__show_regs+6> sd ra,24(sp)
0xfffffffff80003530 <__show_regs+8> addi s0,sp,32
0xfffffffff80003532 <__show_regs+10> mv s1,a0
0xfffffffff80003534 <__show_regs+12> auipc a0,0xd6c
0xfffffffff80003538 <__show_regs+16> addi a0,a0,-1996
0xfffffffff8000353c <__show_regs+20> auipc ra,0x31a

remote Thread 1.1 In: arch_cpu_idle L?? PC: 0xfffffffff8000351e
(gdb)
```

- ② 在 0x80000000 处下断点

直接输入 b \*0x80000000 即可。

```
(gdb) b *0x80000000
Breakpoint 1 at 0x80000000
```

- ③ 查看所有已下的断点  
直接输入 `info break` 即可。

```
(gdb) info break
Num      Type           Disp Enb Address              What
1        breakpoint     keep y   0x0000000080000000
```

- ④ 在 `0x80200000` 处下断点  
直接输入 `b *0x80200000` 即可。

```
(gdb) b *0x80200000
Breakpoint 2 at 0x80200000
```

- ⑤ 清除 `0x80000000` 处的断点  
直接输入 `clear *0x80000000` 即可。

```
(gdb) clear *0x80000000
Deleted breakpoint 1
```

- ⑥ 继续运行直到触发 `0x80200000` 处的断点  
直接输入 `continue` 即可。

```
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG : 0x0000000000000222
Boot HART MEDELEG : 0x00000000000b109
(gdb) continue
Continuing.
Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb)
```

- ⑦ 单步调试一次  
直接输入 `ni(next instruction)`即可。

```
(gdb) ni
0x0000000080200002 in ?? ()
```

- ⑧ 退出 QEMU  
先在 `gdb` 中按 `ctrl+C` 强制中断。

```
^C
Program received signal SIGINT, Interrupt.
0xffffffff8000351e in arch_cpu_idle ()
```

然后在 `qemu` 中先按 `ctrl+A` 然后按 `X` 即可。

```
/ # QEMU: Terminated
```

#### 4. 使用 `make` 工具清除 Linux 的构建产物

- ① 进入编译文件夹。

```
root@61868c8ae428:/debug/os22fall-stu/src/lab0# cd linux
```

- ② 运行 `make clean` 即可。



```
root@61868c8ae428:/debug/os22fall-stu/src/lab0/linux# make clean
CLEAN    drivers/firmware/efi/libstub
CLEAN    drivers/gpu/drm/radeon
CLEAN    drivers/scsi
CLEAN    drivers/tty/vt
CLEAN    kernel
CLEAN    lib
CLEAN    usr
CLEAN    vmlinux.symvers modules-only.symvers modules.builtin modules.builtin.m
odinfo
```

5. vmlinux 和 Image 的关系和区别是什么？

vmlinux 是 elf 格式，支持虚拟内存，是最原始的 kernel 文件，需要用户自主运行。Image 是纯二进制文件，可以直接在裸机上运行。Image 是 vmlinux 通过 OBJCOPY 拷贝得到的。

## 五、附录

用到的参考资料：

[Ubuntu Docker 安装 | 菜鸟教程 \(runoob.com\)](#)