浙江大学实验报告

Lab 0: GDB + QEMU 调试 64 位 RISC-V LINUX

课程名称： 操作系统 实验类型： 综合

实验项目名称： Rinux环境搭建和内核编译

学生姓名： 汪辉 专业： 计算机科学与技术 学号： 3190105609

同组学生姓名： 个人实验 指导老师： 季江民

电子邮件：3190105609

实验地点： 曹西503 实验日期： 2021 年 9 月 23 日

1 实验目的

- 了解容器的使用
- 使用交叉编译工具, 完成Linux内核代码编译
- 使用QEMU运行内核
- 熟悉GDB和QEMU联合调试

2 实验环境

- Linux虚拟机环境下的Docker容器工具
- 下载安装实验环境镜像

3 实验内容

3.1 搭建 Docker 环境

安装 Docker 环境。然后按照以下步骤导入已经准备好的 Docker 镜像：

```
# 导入docker镜像
$ cat oslab.tar | docker import - oslab:2021

# 查看docker镜像
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
oslab        latest    b2b39a3bcd81   404 days ago    3.62GB

# 从镜像创建一个容器
$ docker run --name oslab -it oslab:2021 bash   # --name:容器名称 -i:交互式操作 -t:终端
root@132a140bd724:/#                            # 提示符变为 '#' 表明成功进入容器 后面的字符串根据容器而生成，为容器id
root@132a140bd724:/# exit (or CTRL+D)           # 从容器中退出 此时运行docker ps，运行容器的列表为空
```

```
# 启动处于停止状态的容器
$ docker start oslab          # oslab为容器名称
$ docker ps                   # 可看到容器已经启动
CONTAINER ID   IMAGE         COMMAND       CREATED              STATUS
PORTS     NAMES
132a140bd724   oslab:2021   "bash"        About a minute ago   Up 1 second
      oslab

# 挂载本地目录
# 把用户的 home 目录映射到 docker 镜像内的 have-fun-debugging 目录
$ docker run --name oslab -it -v ${HOME}:/have-fun-debugging oslab:2021 bash
 # -v 本地目录:容器内目录
```

3.2 编译 linux 内核

```
$ pwd
path/to/lab0/linux
$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig    # 生成配置
$ make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc)   # 编译
```

> 使用多线程编译一般会耗费大量内存，如果 `-j` 选项导致内存耗尽 (out of memory)，请尝试调低线程数，比如 `-j4`, `-j8` 等。

3.3 使用QEMU运行内核

```
$ qemu-system-riscv64 -nographic -machine virt -kernel
path/to/linux/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"
\
    -bios default -drive file=rootfs.img,format=raw,id=hd0
```

运行结果如下：

退出 QEMU 的方法为：使用 Ctrl+A，**松开**后再按下 X 键即可退出 QEMU。

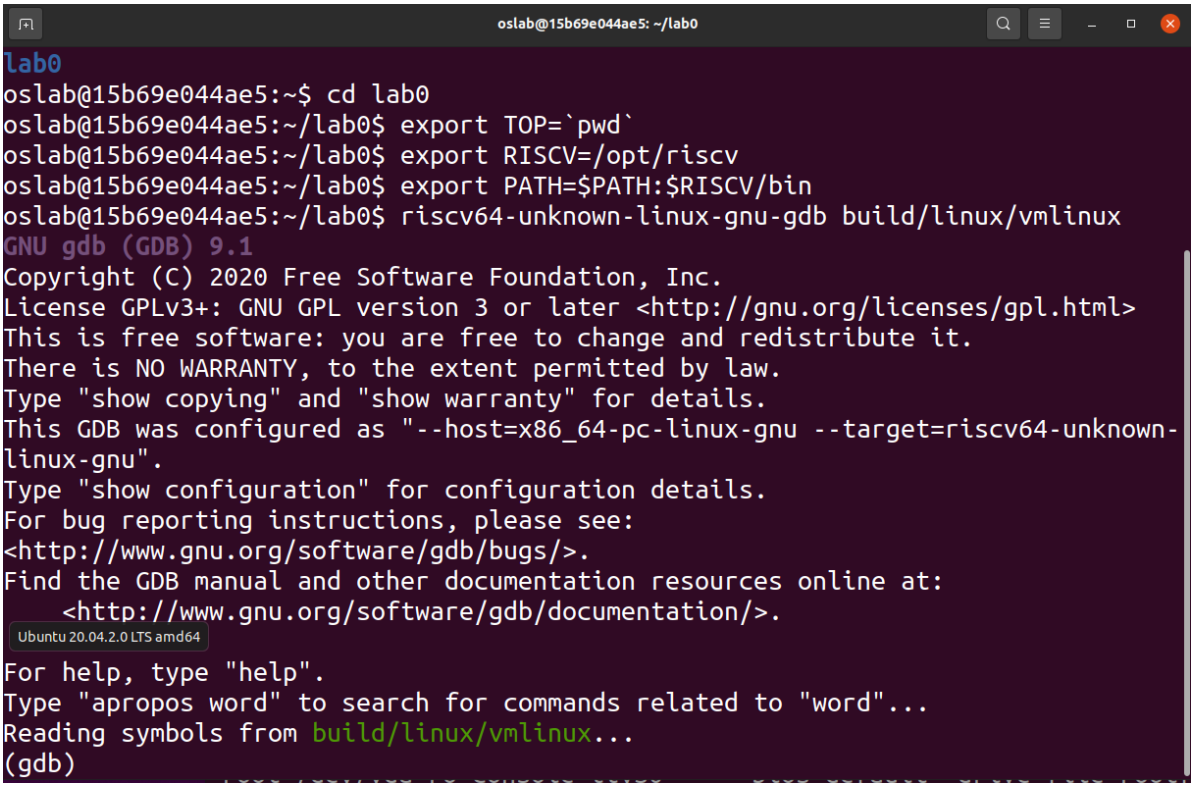
3.4 使用 GDB 对内核进行调试

这一步需要开启两个 Terminal Session，一个 Terminal 使用 QEMU 启动 Linux，另一个 Terminal 使用 GDB 与 QEMU 远程通信（使用 tcp::1234 端口）进行调试。

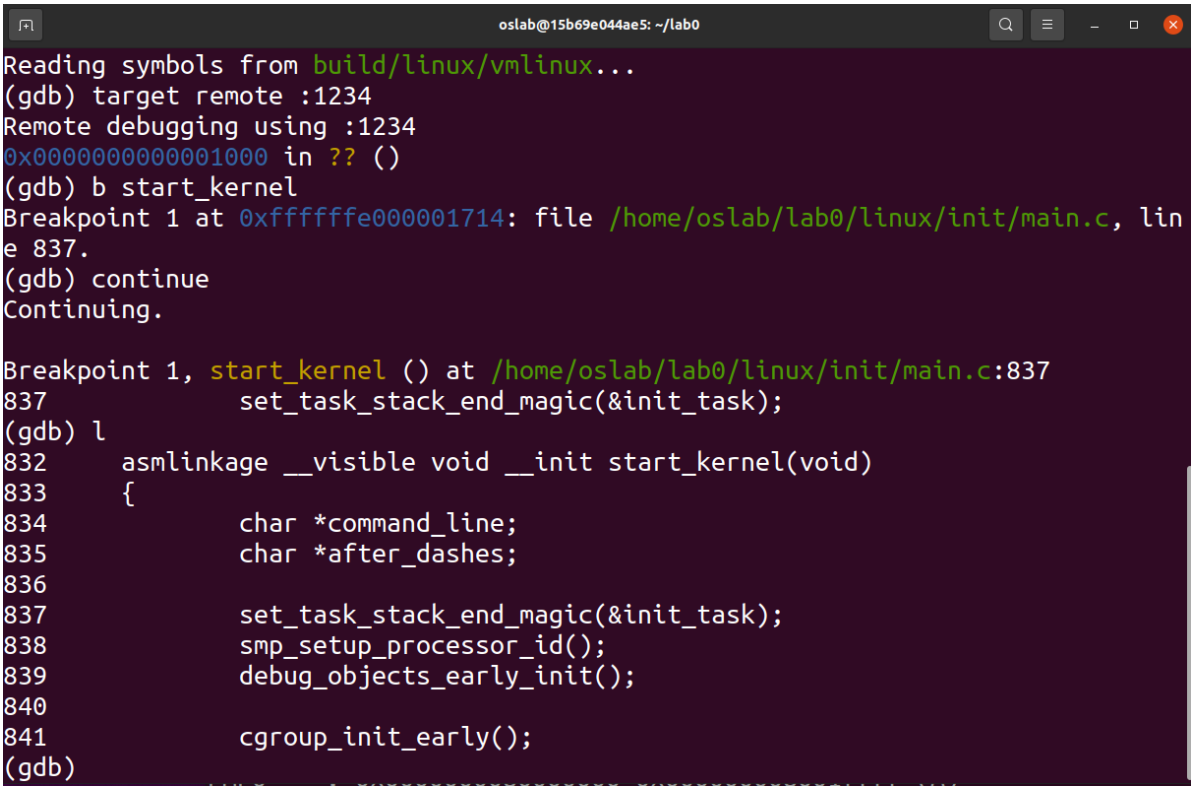
```
# Terminal 1
$ qemu-system-riscv64 -nographic -machine virt -kernel
path/to/linux/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"
\
    -bios default -drive file=rootfs.img,format=raw,id=hd0 -S -s

# Terminal 2
$ riscv64-unknown-linux-gnu-gdb path/to/linux/vmlinux
(gdb) target remote :1234    # 连接 qemu
(gdb) b start_kernel         # 设置断点
(gdb) continue               # 继续执行
(gdb) quit                   # 退出 gdb
```

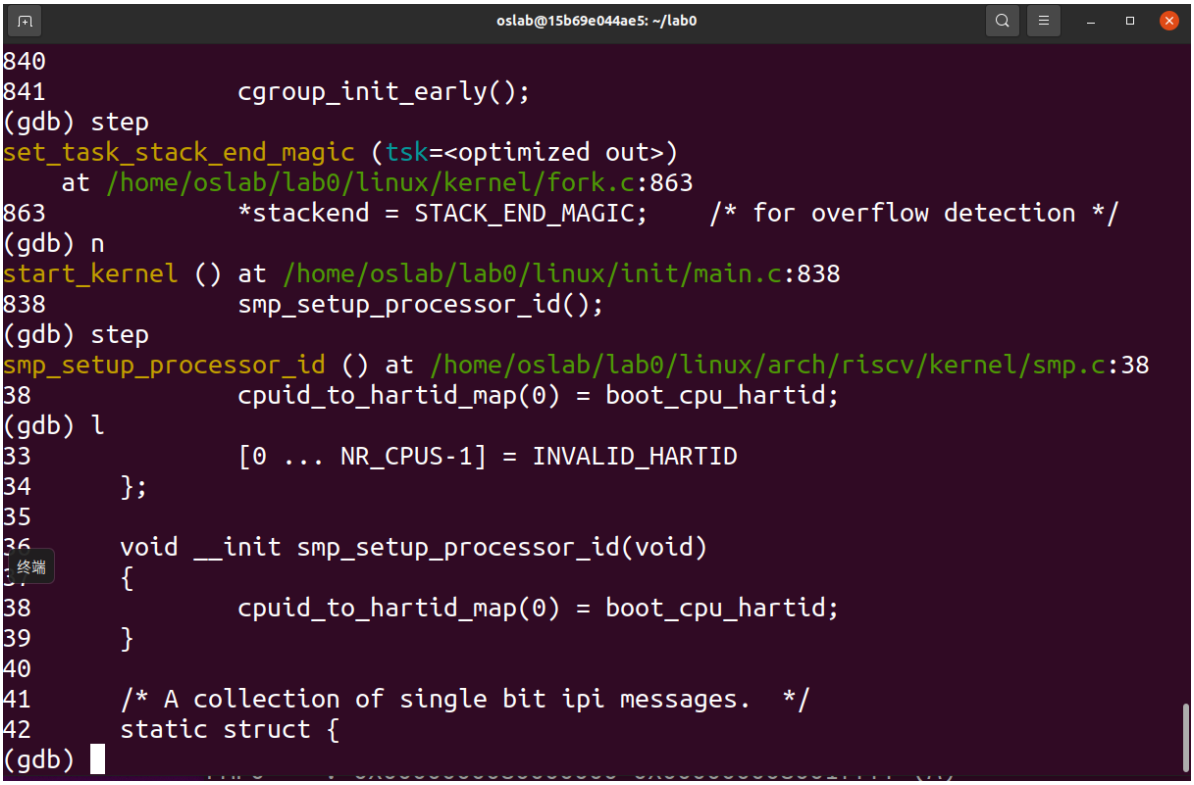
在新的终端启动gdb调试：



设置断点以及查看代码段：



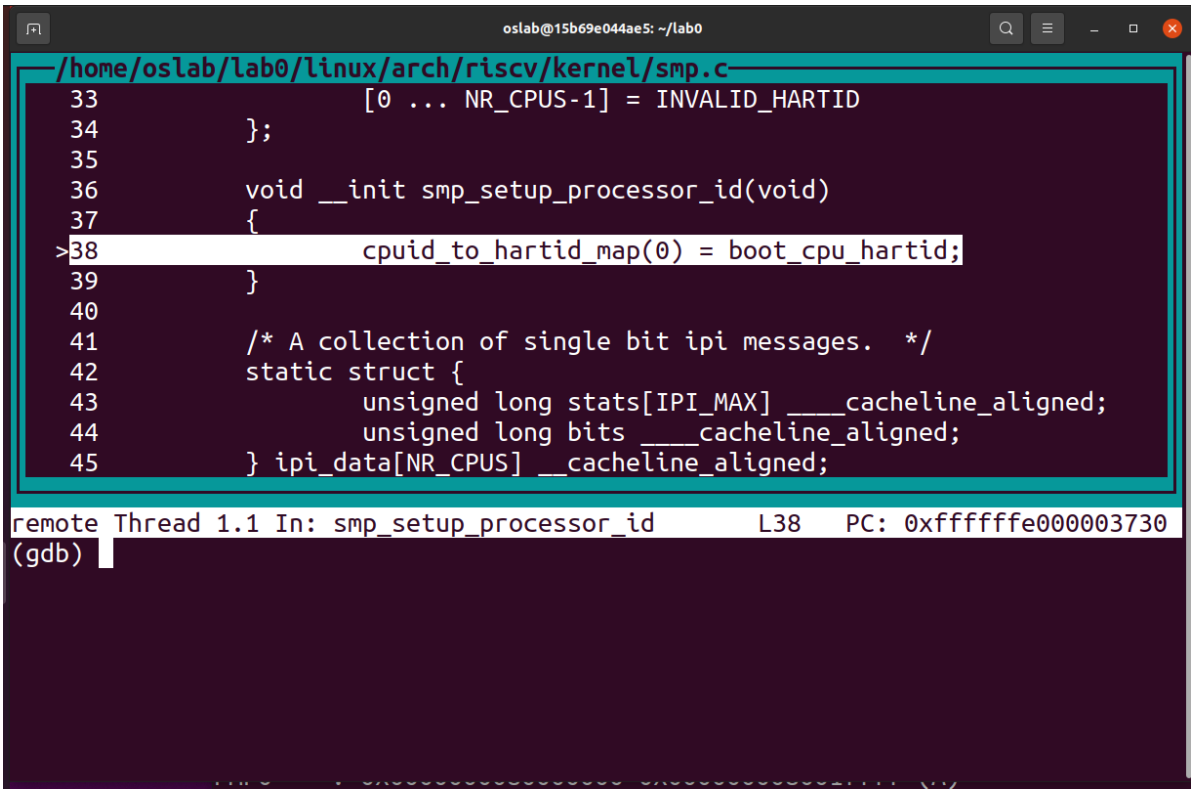
进入子函数`smp_setup_processor_id()`:



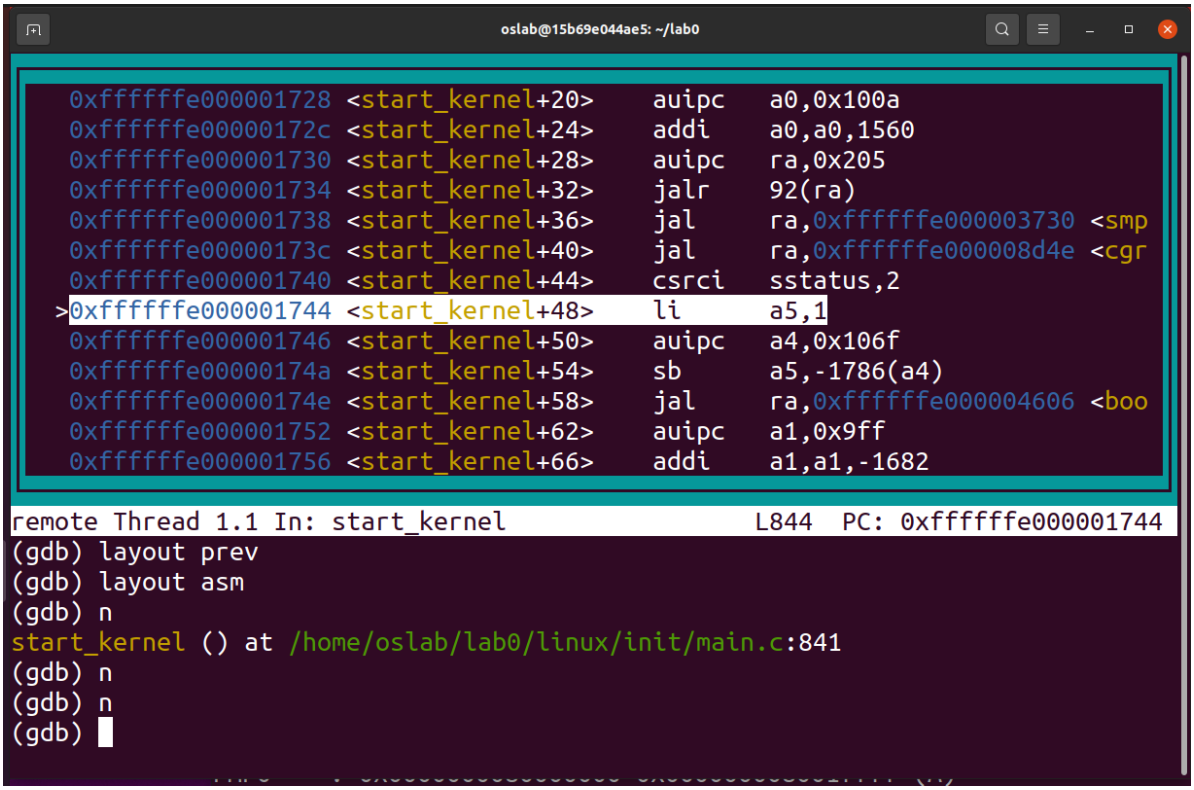
查看栈信息:

```
(gdb) bt
#0  smp_setup_processor_id ()
    at /home/oslab/lab0/linux/arch/riscv/kernel/smp.c:38
#1  0xffffffe00000173c in start_kernel ()
    at /home/oslab/lab0/linux/init/main.c:838
#2  0xffffffe00000108e in _start_kernel ()
    at /home/oslab/lab0/linux/arch/riscv/kernel/head.S:247
Backtrace stopped: frame did not save the PC
(gdb) frame 0
#0  smp_setup_processor_id ()
    at /home/oslab/lab0/linux/arch/riscv/kernel/smp.c:38
38              cpuid_to_hartid_map(0) = boot_cpu_hartid;
(gdb)
```

显示源代码窗口：



查看汇编代码：



查看当前断点：

```
remote Thread 1.1 In: start_kernel                 L844   PC: 0xffffffe000001744
Not confirmed.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffe000001714 in start_kernel
                                                   at /home/oslab/lab0/linux/ini
t/main.c:837
        breakpoint already hit 1 time
(gdb)
```

清除断点：

```
(gdb) clear 1
No breakpoint at 1.
(gdb) clear 837

(gdb) info breakpoints
Deleted breakpoint 1 No breakpoints or watchpoints.
(gdb)
```

结束当前函数运行：

```
(gdb) finish
Run till exit from #0  start_kernel ()
    at /home/oslab/lab0/linux/init/main.c:844

Program received signal SIGINT, Interrupt.
arch_cpu_idle () at /home/oslab/lab0/linux/arch/riscv/kernel/process.c:33
(gdb)
```

4 心得体会

- 学习使用gdb调试工程和代码的基本步骤和方法，了解基础的Linux的shell指令。
- 了解Linux内核的架构，以及编译的原理和过程。