Lab0: GDB+QEMU调试64位RISC-V LINUX实验报告

1. 实验简介

通过在QEMU上运行Linux来熟悉如何从源代码开始将内核运行在QEMU模拟器上，并且掌握使用GDB跟QEMU进行联合调试，为后续实验打下基础。

2. 实验环境

- Ubuntu20.04.1 LTS
- Docker

3. 实验过程

3.1 安装docker

我根据实验指导中给出的链接在Ubuntu虚拟机上完成了Docker的安装

1. 首先创建repository
   - 通过更新`apt`包的索引使之能够使用基于HTTPS的repository

     ```
     $ sudo apt-get update
     ```

   - 添加Docker的官方GPG key

     ```
     $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
     add -
     ```

     并验证fingerprint
   - 通过输入如下命令完成stable repository的建立

     ```
     $ sudo add-apt-repository \
        "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
        $(lsb_release -cs) \
        stable"
     ```

2. 安装Docker Engine
   - 更新`apt`包的索引，并下载最新版本的Docker Engine和containerd

     ```
      $ sudo apt-get update
      $ sudo apt-get install docker-ce docker-ce-cli containerd.io
     ```

3. 通过运行`hello-world` image来验证Docker Engine安装完成

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

3.2 搭建docker环境

1. 通过命令 `sudo usermod -aG docker wang` 用户加入docker group，使得之后的指令不需要在最前面加上 `sudo` .完成后可验证不加 `sudo` 也能运行 `hello-world` image

```
wang@wang-virtual-machine:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

2. 导入docker镜像，并通过命令 `docker image ls` 来查看docker镜像

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ cat oslab.tar | docker import - oslab:2020
sha256:37619cb29f3fab0a729524b5c1c2ab3c7f64aaf7db331936e01aa706789151e8
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
oslab               2020                37619cb29f3f        2 minutes ago       2.89GB
hello-world         latest              bf756fb1ae65        8 months ago        13.3kB
```

3. 从镜像中创建一个容器，exit退出后通过 `docker ps` 命令可看到没有正在运行的容器

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker run -it oslab:2020 /b
in/bash
root@8ad1a264e953:/# exit
exit
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS              PORTS               NAMES
```

4. 利用命令 `docker ps -a` 查看所有存在的容器

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
8ad1a264e953        oslab:2020          "/bin/bash"         38 seconds ago      Exited (0) 15 seconds ago                       modest_feynman
30658d585d97        hello-world         "/hello"            8 minutes ago       Exited (0) 8 minutes ago                        jolly_bhabha
893e8ae58d1d        hello-world         "/hello"            12 minutes ago      Exited (0) 11 minutes ago                       jovial_robinson
16c5508ad043        hello-world         "/hello"            15 minutes ago      Exited (0) 15 minutes ago                       stoic_chandrasekhar
8f8286e97de9        hello-world         "/hello"            38 minutes ago      Exited (0) 38 minutes ago                       jovial_gould
5080666d0344        hello-world         "/hello"            39 minutes ago      Exited (0) 39 minutes ago                       exciting_visvesvaraya
```

5. 启动处于停止状态的容器，我们可根据容器id的前四位来标识该容器

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker start 8ad1
8ad1
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
8ad1a264e953        oslab:2020          "/bin/bash"         About a minute ago  Up 8 seconds                            modest_feynman
```

6. 进入该容器

```
wang@wang-virtual-machine:~/Documents/OS/lab/lab0$ docker exec -it 8ad1a264e953 /bin/bash
root@8ad1a264e953:/# pwd
/
```

3.3 编译Linux内核

1. 进入/home/oslab/lab0，并设置`TOP`，`RISCV`,`PATH`等环境变量

```
root@8ad1a264e953:/home/oslab/lab0# pwd
/home/oslab/lab0
root@8ad1a264e953:/home/oslab/lab0# export TOP=`pwd`
root@8ad1a264e953:/home/oslab/lab0# export RSICV=/opt/riscv
root@8ad1a264e953:/home/oslab/lab0# export PATH=$PATH:$RISCV/bin
```

2. 在实验目录下新建`linux`目录，并在该目录中编译linux内核

```
root@8ad1a264e953:/home/oslab/lab0# ls build
linux
root@8ad1a264e953:/home/oslab/lab0# make -C linux O=$TOP/build/linux \
> CROSS_COMPILE=riscv64-unknown-linux-gnu- \
> ARCH=riscv CONFIG_DEBUG_INFO=y \
> defconfig all -j$(nproc)
```

3.4 使用QEMU运行内核

输入如下命令使用QEMU运行linux内核，并输入用户名root即可进入内核，默认初始目录为/root

```
root@8ad1a264e953:/home/oslab/lab0# qemu-system-riscv64 -nographic -machine virt -kernel build/linux/arch/riscv/boot/Image \
> -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
> -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
> -netdev user,id=net0 -device virtio-net-device,netdev=net0

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2
```

3.5 使用gdb对内核进行调试

1. 在qemu中运行的linux内核中输入exit退出至用户登录界面，`ctrl + A`抬起后，输入`x`返回至docker

```
Welcome to Buildroot
buildroot login: QEMU: Terminated
root@8ad1a264e953:/home/oslab/lab0#
```

2. 运行另一个终端，进入同一个docker容器

```
wang@wang-virtual-machine:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
8ad1a264e953        oslab:2020          "/bin/bash"         7 hours ago         Up 7 hours                              modest_feynman
wang@wang-virtual-machine:~$ docker exec -it 8ad1a264e953 /bin/bash
root@8ad1a264e953:/#
```

3. 在终端1中通过输入如下代码运行linux内核

```
root@8ad1a264e953:/home/oslab/lab0# qemu-system-riscv64 -nographic -machine virt -kernel build/linux/arch/riscv/boot/Image \
> -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
> -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
> -netdev user,id=net0 -device virtio-net-device,netdev=net0 -S -s
```

4. 可以看到在新打开的终端2的docker image内之前终端1中设置的环境变量`$TOP`等都不存在，因此需要先设置环境变量，然后运行gdb进行调试

```
root@8ad1a264e953:/home/oslab/lab0# echo $TOP

root@8ad1a264e953:/home/oslab/lab0# export export TOP=`pwd`
root@8ad1a264e953:/home/oslab/lab0# echo $TOP
/home/oslab/lab0
root@8ad1a264e953:/home/oslab/lab0# export RISCV=/opt/riscv
root@8ad1a264e953:/home/oslab/lab0# export PATH=$PATH:$RISCV/bin
root@8ad1a264e953:/home/oslab/lab0# riscv64-unknown-linux-gnu-gdb build/linux/vmlinux
GNU gdb (GDB) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/linux/vmlinux...
(gdb)
```

5. 连接qume，并设置断点

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) b start_kernel
Breakpoint 1 at 0xffffffe000001714: file /home/oslab/lab0/linux/init/main.c, line 837.
```

6. continue继续执行后，遇到在main.c中的断点

```
(gdb) continue
Continuing.

Breakpoint 1, start_kernel () at /home/oslab/lab0/linux/init/main.c:837
837             set_task_stack_end_magic(&init_task);
```

而在终端1中可以看到程序停在了此处

```
root@8ad1a264e953:/home/oslab/lab0# qemu-system-riscv64 -nographic -machine virt -kernel build/l
inux/arch/riscv/boot/Image \
> -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
> -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
> -netdev user,id=net0 -device virtio-net-device,netdev=net0 -S -s

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

7. 通过`i frame`可显示函数堆栈帧信息

```
(gdb) i frame
Stack level 0, frame at 0xffffffe001004000:
 pc = 0xffffffe000001714 in start_kernel (/home/oslab/lab0/linux/init/main.c:837);
    saved pc = 0xffffffe00000108e
 called by frame at 0xffffffe001004000
 source language c.
 Arglist at 0xffffffe001004000, args:
 Locals at 0xffffffe001004000, Previous frame's sp is 0xffffffe001004000
Could not fetch register "ustatus"; remote failure reply 'E14'
```

8. 输入`info functions`可以输出所有可执行文件的函数名称，结果如图：

```
All defined functions:

File /home/oslab/lab0/linux/arch/riscv/kernel/cacheinfo.c:
21:     const struct attribute_group *cache_get_priv_group(struct cacheinfo *);
116:    int init_cache_level(unsigned int);
117:    int populate_cache_leaves(unsigned int);
14:     void riscv_set_cacheinfo_ops(struct riscv_cacheinfo_ops *);
116:    static void _init_cache_level(void *);
117:    static void _populate_cache_leaves(void *);

File /home/oslab/lab0/linux/arch/riscv/kernel/cpu.c:
53:     int riscv_of_parent_hartid(struct device_node *);
15:     int riscv_of_processor_hartid(struct device_node *);
95:     static void *c_next(struct seq_file *, void *, loff_t *);
105:    static int c_show(struct seq_file *, void *);
87:     static void *c_start(struct seq_file *, loff_t *);
101:    static void c_stop(struct seq_file *, void *);

File /home/oslab/lab0/linux/arch/riscv/kernel/cpu_ops.c:
36:     void cpu_set_ops(int);
24:     void cpu_update_secondary_bootdata(unsigned int, struct task_struct *);

File /home/oslab/lab0/linux/arch/riscv/kernel/cpu_ops_sbi.c:
68:     static int sbi_cpu_prepare(unsigned int);
56:     static int sbi_cpu_start(unsigned int, struct task_struct *);

File /home/oslab/lab0/linux/arch/riscv/kernel/cpu_ops_spinwait.c:
15:     static int spinwait_cpu_prepare(unsigned int);
24:     static int spinwait_cpu_start(unsigned int, struct task_struct *);

File /home/oslab/lab0/linux/arch/riscv/kernel/cpufeature.c:
51:     bool __riscv_isa_extension_available(const unsigned long *, int);
62:     void riscv_fill_hwcap(void);
33:     unsigned long riscv_isa_extension_base(const unsigned long *);

File /home/oslab/lab0/linux/arch/riscv/kernel/irq.c:
13:     int arch_show_interrupts(struct seq_file *, int);
19:     void init_IRQ(void);

File /home/oslab/lab0/linux/arch/riscv/kernel/module-sections.c:
13:     unsigned long module_emit_got_entry(struct module *, unsigned long);
32:     unsigned long module_emit_plt_entry(struct module *, unsigned long);
90:     int module_frob_arch_sections(Elf64_Ehdr *, Elf64_Shdr *, char *, struct module *);

File /home/oslab/lab0/linux/arch/riscv/kernel/module.c:
296:
--Type <RET> for more, q to quit, c to continue without paging--
```

9. 使用`finish`命令执行完调试函数并打印返回值退出

```
(gdb) finish
Run till exit from #0  start_kernel () at /home/oslab/lab0/linux/init/main.c:837
```

10. 使用`backtrace`命令可以输出caller的frame，每行一个frame

```
(gdb) b start_kernel
Breakpoint 1 at 0xffffffe000001714: file /home/oslab/lab0/linux/init/main.c, line 837.
(gdb) continue
Continuing.

Breakpoint 1, start_kernel () at /home/oslab/lab0/linux/init/main.c:837
837             set_task_stack_end_magic(&init_task);
(gdb) backtrace
#0  start_kernel () at /home/oslab/lab0/linux/init/main.c:837
#1  0xffffffe00000108e in _start_kernel () at /home/oslab/lab0/linux/arch/riscv/kernel/head.S:247
Backtrace stopped: frame did not save the PC
```

11. 使用`list`命令可以输出源代码（如函数，文件某一行等）

```
(gdb) list start_kernel
832     asmlinkage __visible void __init start_kernel(void)
833     {
834             char *command_line;
835             char *after_dashes;
836
837             set_task_stack_end_magic(&init_task);
838             smp_setup_processor_id();
839             debug_objects_early_init();
840
841             cgroup_init_early();
```

12. 使用`display`命令可以自动输出表达式的值

```
(gdb) continue
Continuing.

Breakpoint 1, start_kernel () at /home/oslab/lab0/linux/init/main.c:837
837             set_task_stack_end_magic(&init_task);
(gdb) display command_line
1: command_line = 0xffffffe00001bb80 <early_init_dt_scan_nodes+74> "\242`\002dA\001\202\200\001\021\"\350&\344\006\354"
```

13. 使用`quit`命令即可退出GDB

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1] will be detached.

Quit anyway? (y or n) y
Detaching from program: , process 1
Ending remote debugging.
[Inferior 1 (process 1) detached]
root@8ad1a264e953:/#
```

3. 遇到的问题

- 在编译Linux内核时，因为没有读懂实验指导中的指令意义，将`export TOP=`pwd``的反引号打成了引号，变成`export TOP='pwd'`这使得环境变量`TOP`由指令`pwd`的结果变成了字符串`'pwd'`。这导致后来在进行make时`./build/linux`是空的
- 在打开新的终端运行docker的镜像时，需要重新设置环境变量
- 关闭qume的方法为：先通过`Ctrl + A`抬起，然后输入`X`退出

4. 心得体会

本次实验是操作系统的第一个实验，实验手册非常详细，使得自己做时搭建环境的过程还是比较顺利的，相对来说难度不大。通过本次实验，我熟悉了linux内核编译的方法，并对gdb调试有了一个基本的了解，感觉操作系统这门课还是很有意思的，希望能让自己得到锻炼。