

浙江大学

本科实验报告

课程名称：操作系统

姓 名：蔡佳伟

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3220104519

指导教师：李环、柳晴

2024 年 9 月 11 日

浙江大学操作系统实验报告

实验名称: _____lab0:GDB & QEMU 调试 64 位 RISC-V LINUX_____

电子邮件地址: _____3220104519@zju.edu.cn_____手机: _____19550230334_____

实验地点: _____曹西 503_____实验日期: _____2024_____年 _____9_____月 _____11_____日

一、实验目的和要求

- 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

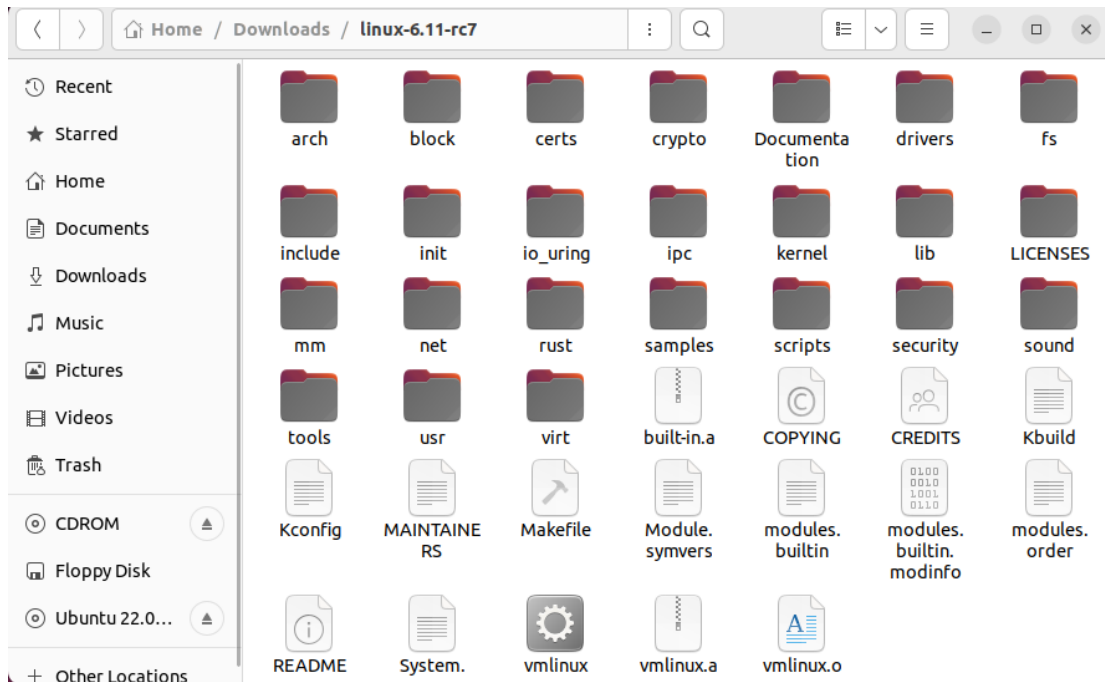
二、实验过程

1.实验环境

```
cjw@cjw-virtual-machine:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.4 LTS
Release:        22.04
Codename:       jammy
```

2.获取 linux 源码和 clone 仓库

```
cjw@cjw-virtual-machine: ~/Downloads/linux-6.11-rc7
cjw@cjw-virtual-machine:~$ cd OSlab0
cjw@cjw-virtual-machine:~/OSlab0$ git clone https://github.com/ZJU-SEC/os24fall-stu.git
Cloning into 'os24fall-stu'...
cjw@cjw-virtual-machine:~/OSlab0$ cd os24fall-stu/src/lab0
cjw@cjw-virtual-machine:~/OSlab0/os24fall-stu/src/lab0$ ls
rootfs.ing
```



可以看到我已经下载了 linux 内核源码并解压

3.编译 linux 内核

使用默认配置

指令：

```
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
```

```
cjw@cjw-virtual-machine:~$ cd Downloads
cjw@cjw-virtual-machine:~/Downloads$ cd linux-6.11-rc7
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
HOSTCC scripts/kconfig/confdata.o
HOSTCC scripts/kconfig/expr.o
```

进行编译，为防止内存耗尽，使用八线程编译

指令：

```
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j8
```

```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j8
WRAP arch/riscv/include/generated/uapi/asm/errno.h
UPD include/generated/uapi/linux/version.h
WRAP arch/riscv/include/generated/uapi/asm/ioctl.h
```

编译完成

4.使用 QEMU 运行内核

指令：

```
qemu-system-riscv64 -nographic -machine virt -kernel arch/riscv/boot/Image \
-device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
-bios default -drive file=../../OSlab0/os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0
```

```
cjw@cju-virtual-machine: ~/Downloads$ cd linux-6.11-rc7
cjw@cju-virtual-machine: ~/Downloads/linux-6.11-rc7$ qemu-system-riscv64 -nographic
-machine virt -kernel arch/riscv/boot/Image \
-device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
-bios default -drive file=../../OSlab0/os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0

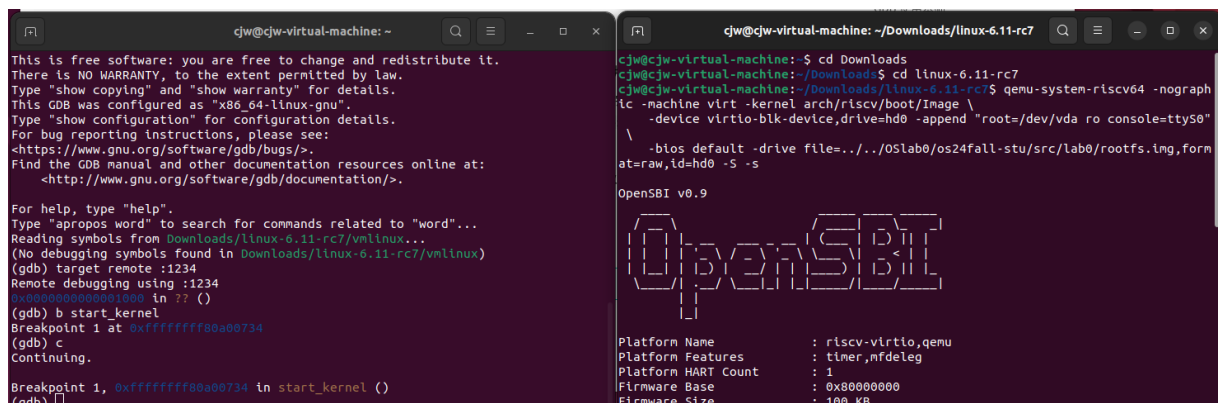
OpenSBI v0.9

Platform Name      : riscv-virtio,qemu
```

进入系统并成功退出（使用 `ctrl+A` 松开后再按下 `X`）

```
Please press Enter to activate this console.
/ #
/ # ls
bin          etc          lost+found  sbin         usr
dev          linuxrc      proc        sys
/ # QEMU: Terminated
```

5.使用 GDB 对内核进行调试



```
cjw@cju-virtual-machine: ~
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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 Downloads/linux-6.11-rc7/vmlinux...
(No debugging symbols found in Downloads/linux-6.11-rc7/vmlinux)
(gdb) target remote :1234
Remote debugging using :1234
0x0000000000000000 in ?? ()
(gdb) b start_kernel
Breakpoint 1 at 0xfffffff80a00734
(gdb) c
Continuing.

Breakpoint 1, 0xfffffff80a00734 in start_kernel ()
(gdb) ||

cjw@cju-virtual-machine: ~/Downloads/linux-6.11-rc7
cjw@cju-virtual-machine: ~/Downloads$ cd linux-6.11-rc7
cjw@cju-virtual-machine: ~/Downloads/linux-6.11-rc7$ qemu-system-riscv64 -nographic
-machine virt -kernel arch/riscv/boot/Image \
-device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
-bios default -drive file=../../OSlab0/os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0 -S -s

OpenSBI v0.9

Platform Name      : riscv-virtio,qemu
Platform Features  : timer,mfdeleg
Platform HART Count : 1
Firmware Base      : 0x80000000
Firmware Size      : 100 KB
```

可以看到我打开了两个终端，一个 Terminal 使用 QEMU 启动 Linux（右边），另一个 Terminal 使用 GDB 与 QEMU 远程通信（使用 `tcp:: 1234` 端口）（左边）。并使用了 `continue` 等命令。指令（Terminal 1）：

```
qemu-system-riscv64 -nographic -machine virt -kernel arch/riscv/boot/Image \
-device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
-bios default -drive file=../../OSlab0/os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0 -S -s
```

指令（Terminal 2）：

```
gdb-multiarch Downloads/linux-6.11-rc7/vmlinux
target remote :1234
```

使用 `break` 命令：设置断点，使程序在指定位置停下来

使用 `backtrace/bt`：显示函数调用栈信息，包括当前函数和调用链

```
Breakpoint 1, 0xfffffff80a00734 in start_kernel ()
(gdb) bt
#0  0xfffffff80a00734 in start_kernel ()
#1  0xfffffff80001164 in _start_kernel ()
Backtrace stopped: frame did not save the PC
```

使用 `frame/f`: 切换当前栈帧, 可以查看不同函数的局部变量和上下文信息

```
(gdb) f
#0  0xffffffff80a00734 in start_kernel ()
(gdb) f 1
#1  0xffffffff80001164 in _start_kernel ()
```

使用 `info`: 显示各种信息, 如断点, 变量, 栈帧等

```
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint      keep y   0xffffffff80a00734 <start_kernel>
breakpoint already hit 1 time
```

使用 `finish`: 执行程序, 直到当前函数返回

```
(gdb) stepi
0xffffffff80a00736 in start_kernel ()
(gdb) finish
Run till exit from #0  0xffffffff80a00736 in start_kernel ()
```

使用 `next`: 单步到程序源代码的下一行, 不进入函数

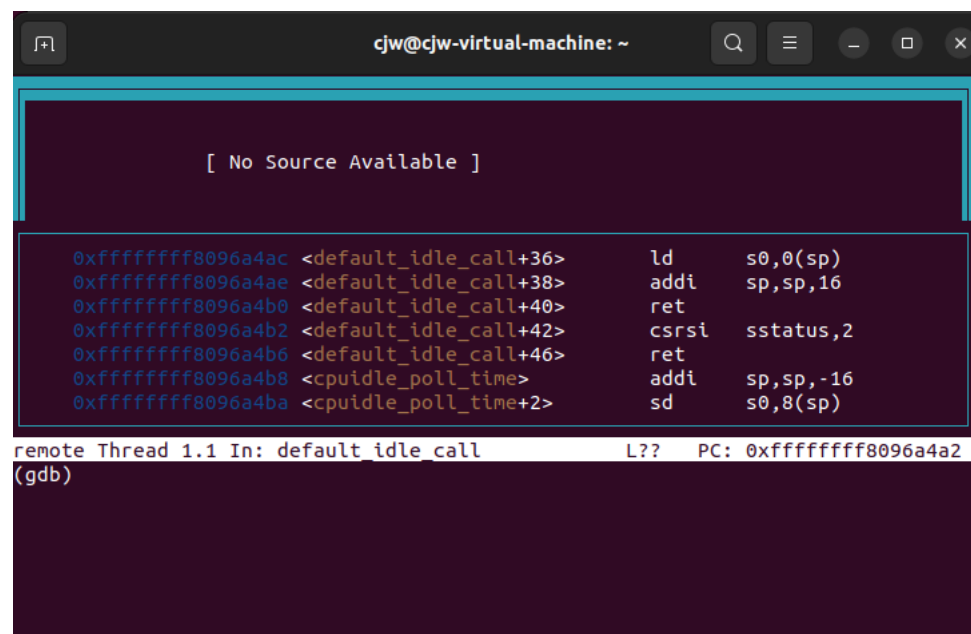
使用 `nexti`: 单步一条机器指令, 不进入函数

使用 `step`: 单步到下一个不同的源代码行 (会直接执行函数体)

使用 `stepi`: 单步一条机器指令

```
(gdb) next
Single stepping until exit from function arch_cpu_idle,
which has no line number information.
0xffffffff8096a4a2 in default_idle_call ()
```

使用 `layout`: 配置 GDB 界面布局



三、讨论和心得

在此次实验之前, 我没有接触过操作系统相关的知识, 对虚拟机的操作也非常不熟悉。实验过程中, 我补充了很多相关知识, 包括 `linux` 使用的基础操作, 常见指令的执行等。我认为这些基本的知识是非常重要的。在过程中也有几次失败: 比如 `gdb` 调试的时候不知道程序是否已经在运行, 错误使用了 `run` 等指令, 比如没有弄清楚各个操作需要在哪个目录下进行等。

我在寻找错误的过程中，也加深了自己对于本实验的理解，同时也了解了更多的知识。

四、思考题

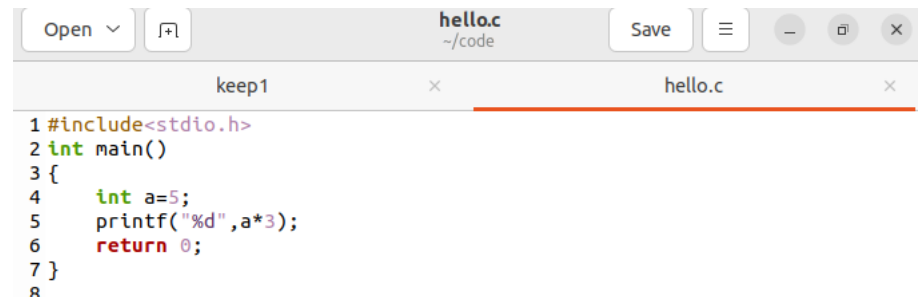
1.使用 riscv64-linux-gnu-gcc 编译单个.c 文件

指令：

```
riscv64-linux-gnu-gcc -o hello hello.c
```

```
cjw@cjw-virtual-machine:~/code$ riscv64-linux-gnu-gcc -o hello hello.c
```

其中 hello.c 如下：



```
1 #include<stdio.h>
2 int main()
3 {
4     int a=5;
5     printf("%d",a*3);
6     return 0;
7 }
8
```

之后会发现生成了 hello 文件

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

指令：

```
Riscv64-linux-gnu-objdump -d hello
```

```
cjw@cjw-virtual-machine:~/code$ riscv64-linux-gnu-objdump -d hello
hello:      file format elf64-littleriscv

Disassembly of section .plt:

Trash 000000570 <.plt>:
000000570: 00002397          auipc   t2,0x2
574: 41c30333          sub     t1,t1,t3
578: a983be03          ld      t3,-1384(t2) # 2008 <__TMC_END__>
57c: fd430313          addi    t1,t1,-44
580: a9838293          addi    t0,t2,-1384
584: 00135313          srli    t1,t1,0x1
588: 0082b283          ld      t0,8(t0)
58c: 000e0067          jr      t3

0000000000000590 <__libc_start_main@plt>:
590: 00002e17          auipc   t3,0x2
594: a88e3e03          ld      t3,-1400(t3) # 2018 <__libc_start_
```

拉到 main 函数部分，发现和 hello.c 相符

```

0000000000000668 <main>:
668: 1101      addi    sp,sp,-32
66a: ec06      sd      ra,24(sp)
66c: e822      sd      s0,16(sp)
66e: 1000      addi    s0,sp,32
670: 4795      li      a5,5
672: fef42623  sw      a5,-20(s0)
676: fec42783  lw      a5,-20(s0)
67a: 873e      mv      a4,a5
67c: 87ba      mv      a5,a4
67e: 0017979b  slliw   a5,a5,0x1
682: 9fb9      addw    a5,a5,a4
684: 2781      sext.w  a5,a5
686: 85be      mv      a1,a5
688: 00000517  auipc   a0,0x0
68c: 02050513  addi    a0,a0,32 # 6a8 <_IO_stdin_used+0x8
>
690: f11ff0ef  jal     ra,5a0 <printf@plt>
694: 4781      li      a5,0
696: 853e      mv      a0,a5
698: 60e2      ld      ra,24(sp)
69a: 6442      ld      s0,16(sp)

```

3.调试 Linux 部分:

在 gdb 中查看汇编代码

指令:

disassemble /m start_kernel

```

(gdb) disassemble /m start_kernel
Dump of assembler code for function start_kernel:
0xffffffff80a00734 <+0>:      addi    sp,sp,-96
0xffffffff80a00736 <+2>:      sd      ra,88(sp)
0xffffffff80a00738 <+4>:      sd      s0,80(sp)
0xffffffff80a0073a <+6>:      sd      s1,72(sp)
0xffffffff80a0073c <+8>:      addi    s0,sp,96
0xffffffff80a0073e <+10>:     sd      s2,64(sp)
0xffffffff80a00740 <+12>:     sd      s3,56(sp)
0xffffffff80a00742 <+14>:     sd      s4,48(sp)
0xffffffff80a00744 <+16>:     sd      s5,40(sp)
0xffffffff80a00746 <+18>:     sd      s6,32(sp)
0xffffffff80a00748 <+20>:     sd      s7,24(sp)
0xffffffff80a0074a <+22>:     sd      s8,16(sp)
0xffffffff80a0074c <+24>:     auipc   a0,0xa0c
0xffffffff80a00750 <+28>:     addi    a0,a0,1332 # 0xffffffff8140cc80

```

在 0x80000000 处下断点

指令:

break *0x80000000

查看所有已下的断点

指令:

info breakpoints

```

(gdb) break *0x80000000
Breakpoint 1 at 0x80000000
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint      keep y   0x0000000080000000

```

在 0x80200000 处下断点

```

(gdb) break *0x80200000
Breakpoint 2 at 0x80200000
(gdb) delete 2

```

清除 0x80000000 处的断点

指令:

delete 断点序号

刚刚清除有误, 重新清除

```
(gdb) break *0x80200000
Breakpoint 3 at 0x80200000
(gdb) delete 1
```

继续运行直到触发 0x80200000 处的断点

指令:

continue

单步调试一次

指令:

stepi

```
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000
(gdb) continue
Continuing.

Breakpoint 1, 0x0000000080200000 in ?? ()
(gdb) stepi
0x0000000080200002 in ?? ()
```

退出 QEMU

```
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG : 0x0000000000000222
Boot HART MEDELEG : 0x000000000000b109
QEMU: Terminated
cjlw@cjlw-virtual-machine:~/Downloads/linux-6.11-rc7$ a
```

4.使用 make 工具清除 Linux 的构建产物

指令:

make clean

```
QEMU: Terminated
cjlw@cjlw-virtual-machine:~/Downloads/linux-6.11-rc7$ make clean
CLEAN drivers/firmware/efi/libstub
CLEAN drivers/gpu/drm/radeon
CLEAN drivers/scsi
CLEAN drivers/tty/vt
CLEAN init
CLEAN kernel
CLEAN lib/raid6
CLEAN lib
CLEAN security/apparmor
CLEAN security/selinux
CLEAN usr
CLEAN .
CLEAN modules.builtin modules.builtin.modinfo .vmlinux.export.c
```

5.vmlinux 和 Image 的关系和区别是什么

vmlinux 和 Image 都是 Linux 内核的文件。vmlinux 是原始的、未压缩的 Linux 内核映像。Linux 前面的“vm”代表虚拟内存。在 Linux 中, 我们可以使用一部分硬盘空间作为虚拟内存, 因此得名“vm”。Image 是经过优化的 Linux 内核映像文件, 一般编译器链接生成的文件都是一个 ELF 格式的可执行文件, 对于内核来说也就是经过 LD 后生成 vmlinux, 然后利用 OBJCOPY 工具处理这个 EFL 文件, 去除其中的符号和重定位信息等等, 生成一个完全的二进制文件 Image。Image 文件可以被加载到计算机的内存中, 用于引导和运行系统。

