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

姓名：姜雨童
学号：3220103450
日期：2024.09.18

1 实验内容及原理

1.1 实验目的

- 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

1.2 实验环境

Ubuntu 22.04

```
jyt@fine:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
jyt@fine:~$
```

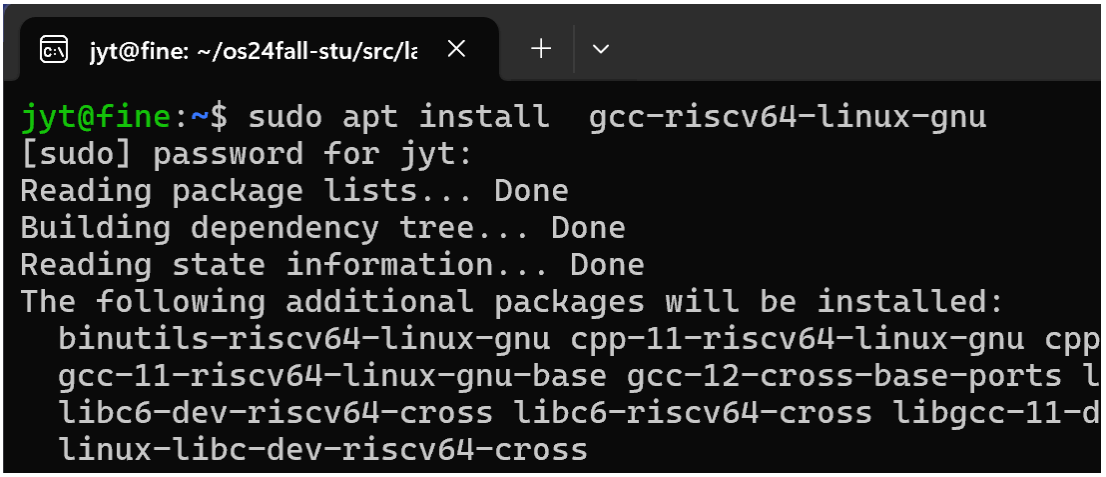
2 实验过程与代码实现

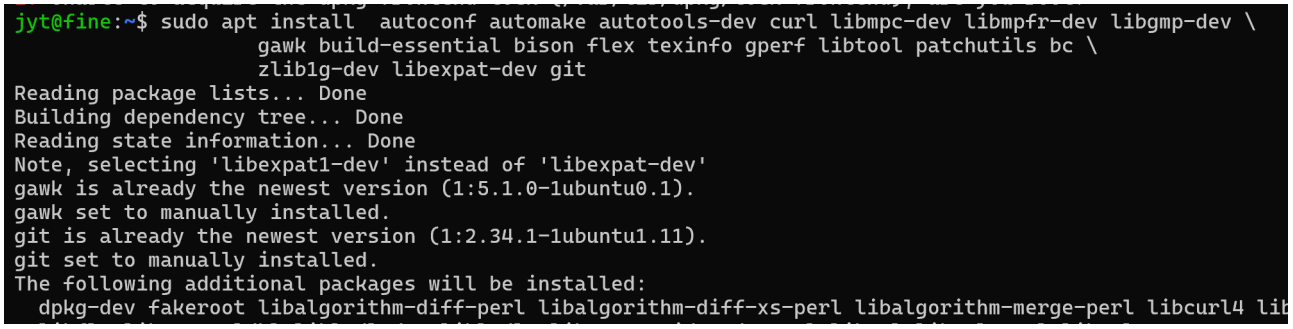
2.1 搭建实验环境

更新软件包列表并安装安装编译内核所需要的交叉编译工具链和用于构建程序的软件包：

```
$ sudo apt update
$ sudo apt install  gcc-riscv64-linux-gnu
$ sudo apt install  autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev \
          gawk build-essential bison flex texinfo gperf libtool patchutils bc \
          zlib1g-dev libexpat-dev git
```

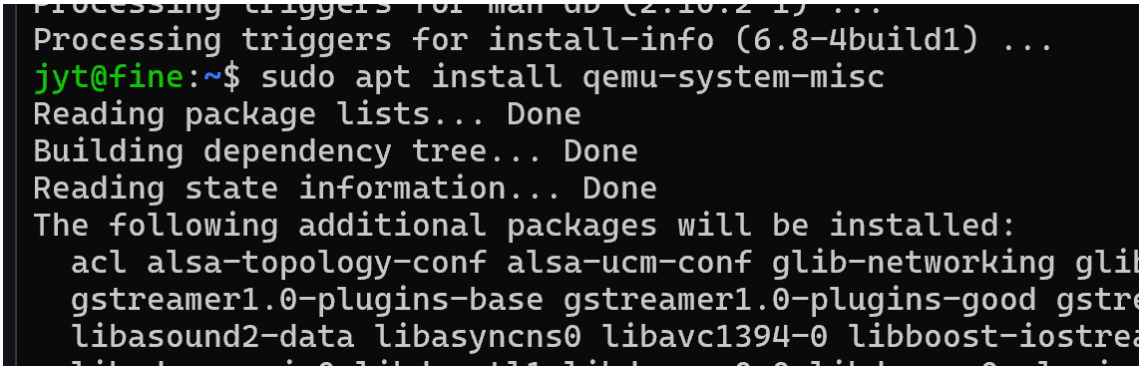
```
jyt@fine: ~
jyt@fine:~$ sudo apt update
[sudo] password for jyt:
Hit:1 http://mirrors.ustc.edu.cn/ubuntu jammy-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu jammy InRelease
Hit:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:4 http://archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
71 packages can be upgraded. Run 'apt list --upgradable' to see them.
jyt@fine:~$
```





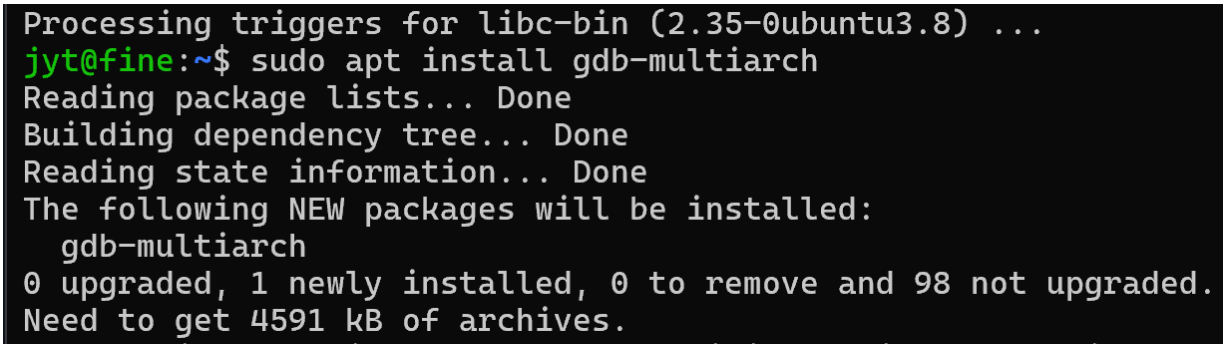
安装模拟器qemu，用以启动 riscv64 平台上的内核：

```
$ sudo apt install qemu-system-misc
```



安装gdb，用以调试qemu上运行的Linux内核：

```
$ sudo apt install gdb-multiarch
```



2.2 获取 Linux 源码和已经编译好的文件系统

利用wget从 https://www.kernel.org下载最新的Linux源码（截至实验时为`linux-6.11-rc7`），并解压：

```
$ wget https://git.kernel.org/torvalds/t/linux-6.11-rc7.tar.gz
$ tar -zxvf linux-6.11-rc7.tar.gz
```

```
jyt@fine:~$ wget
wget: missing URL
Usage: wget [OPTION]... [URL]...

Try `wget --help' for more options.
jyt@fine:~$ wget https://git.kernel.org/torvalds/t/linux-6.11-rc7.tar.gz
--2024-09-13 09:18:40--  https://git.kernel.org/torvalds/t/linux-6.11-rc7.tar.gz
Resolving git.kernel.org (git.kernel.org)... 145.40.73.55, 2604:1380:40e1:4800::1
Connecting to git.kernel.org (git.kernel.org)|145.40.73.55|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/snapshot/linux-6.11-rc7.tar.gz [following]
--2024-09-13 09:18:42--  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/snapshot/linux-6.11-rc7.tar.gz
Reusing existing connection to git.kernel.org:443.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/x-gzip]
Saving to: ‘linux-6.11-rc7.tar.gz’

linux-6.11-rc7.tar.gz              [           <=>                                 ]   7.31M  1.88MB/s
```

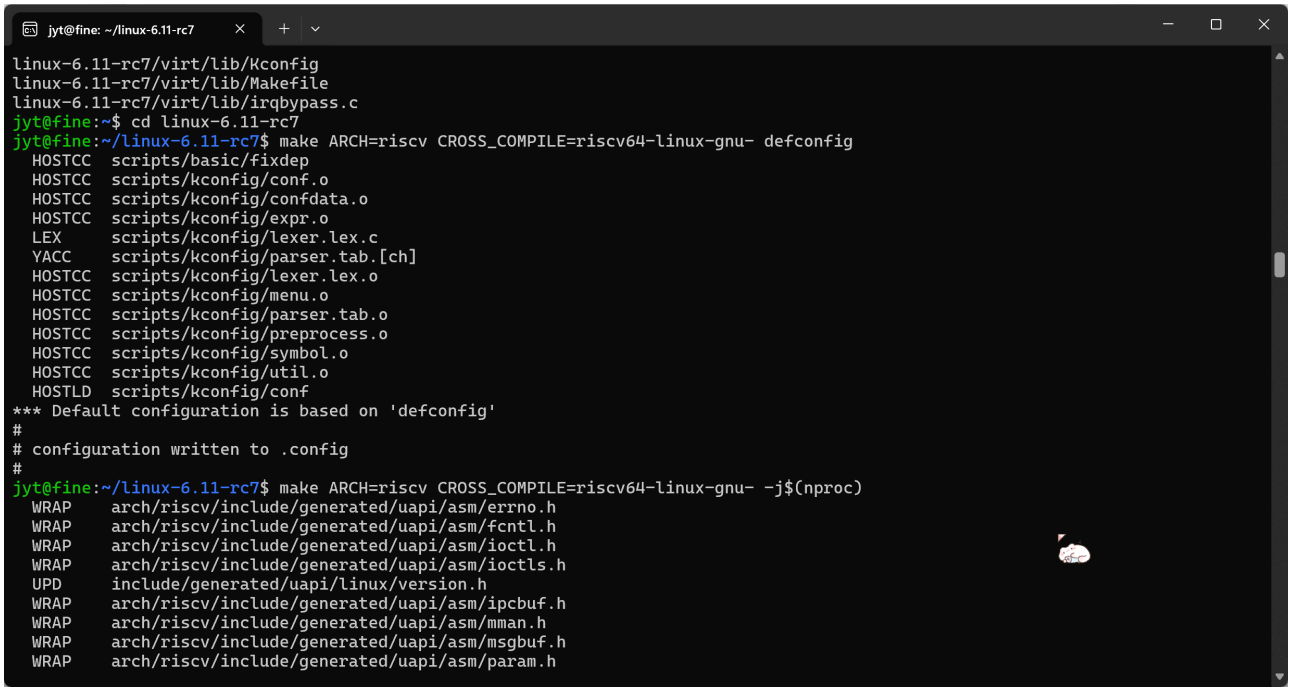
使用git工具clone实验提供的仓库：

```
$ git clone https://github.com/ZJU-SEC/os24fall-stu.git
$ cd os24fall-stu/src/lab0
$ ls
rootfs.img  # 已经构建完成的根文件系统的镜像
```

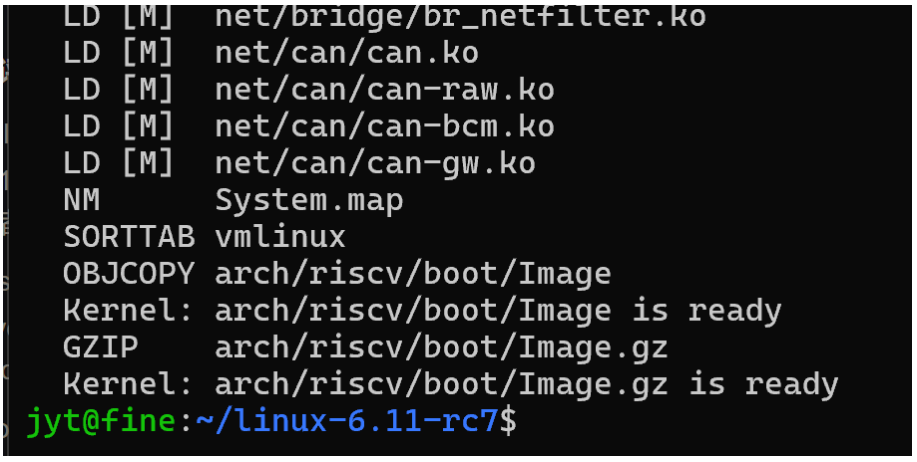
```
jyt@fine:~$ git clone https://github.com/ZJU-SEC/os24fall-stu.git
Cloning into 'os24fall-stu'...
remote: Enumerating objects: 82, done.
remote: Counting objects: 100% (82/82), done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 82 (delta 36), reused 60 (delta 18), pack-reused 0 (from 0)
Receiving objects: 100% (82/82), 1.97 MiB | 2.50 MiB/s, done.
Resolving deltas: 100% (36/36), done.
jyt@fine:~$ cd os24fall/src/lab0
-bash: cd: os24fall/src/lab0: No such file or directory
jyt@fine:~$ cd os24fall-stu/src/lab0
jyt@fine:~/os24fall-stu/src/lab0$ ls
rootfs.img
jyt@fine:~/os24fall-stu/src/lab0$
```

2.3 编译Linux内核

```
$ cd linux-6.11-rc7                                                  # 修改为实际路径
$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig         # 使用默认配置
$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j$(nproc)        # 编译
```



编译完毕：



2.4 使用 QEMU 运行内核

```
$ 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=../os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0
```

```
jyt@fine:~/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=../os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0

OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
```

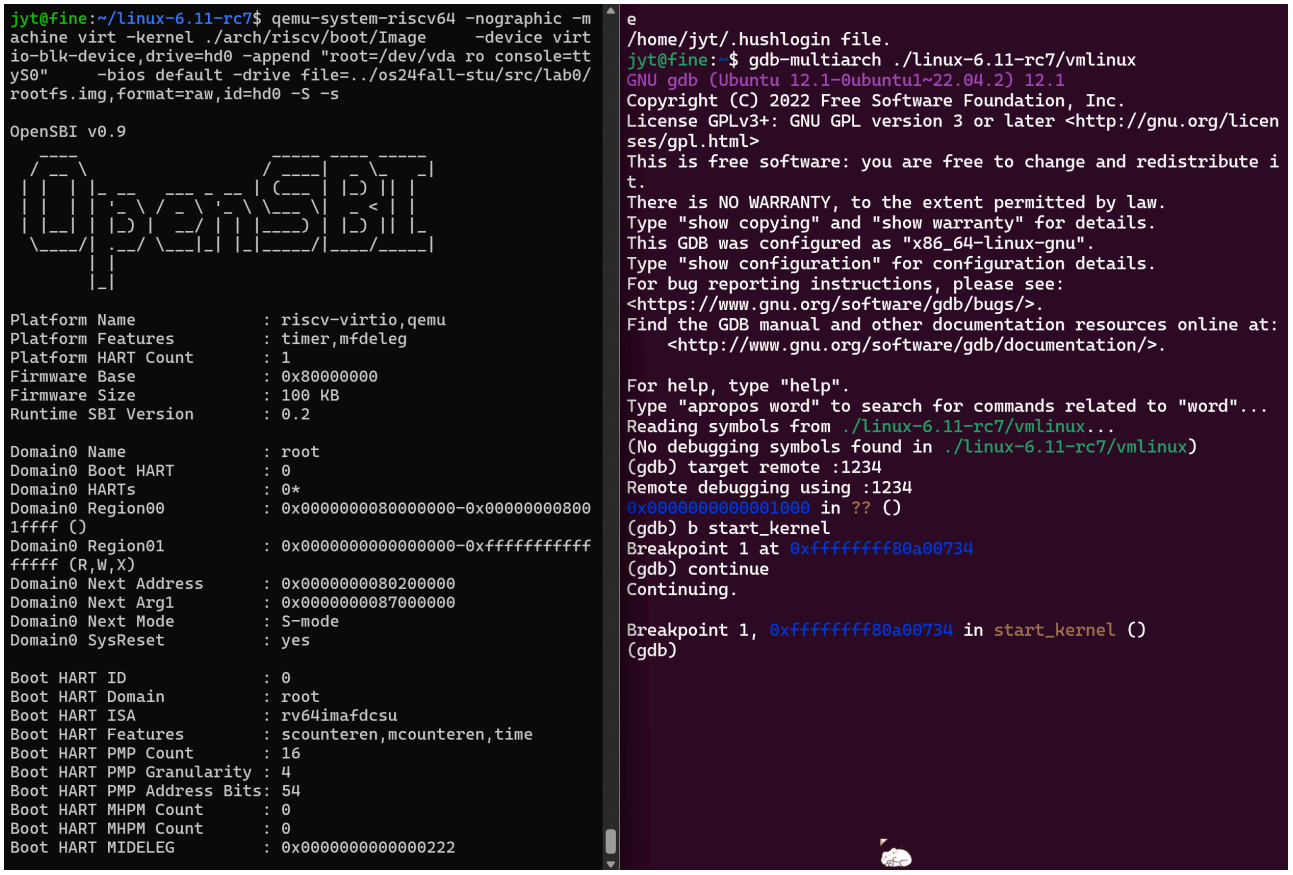
使用Ctrl+A，松开后按下X退出QEMU：

```
[    0.445859] devtmpfs: mounted
[    0.473934] Freeing unused kernel image (initmem) memory: 2256K
[    0.474586] Run /sbin/init as init process

Please press Enter to activate this console.
/ # QEMU: Terminated
```

2.5 使用 GDB 对内核进行调试

这一步开启两个终端，一个启动linux（图左），另一个使用gdb与QEMU远程通信（图右），进行调试，代码和结果如下：

```
# 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=../os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0 -S -s

# Terminal 2
$ gdb-multiarch ./linux-6.11-rc7/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
```

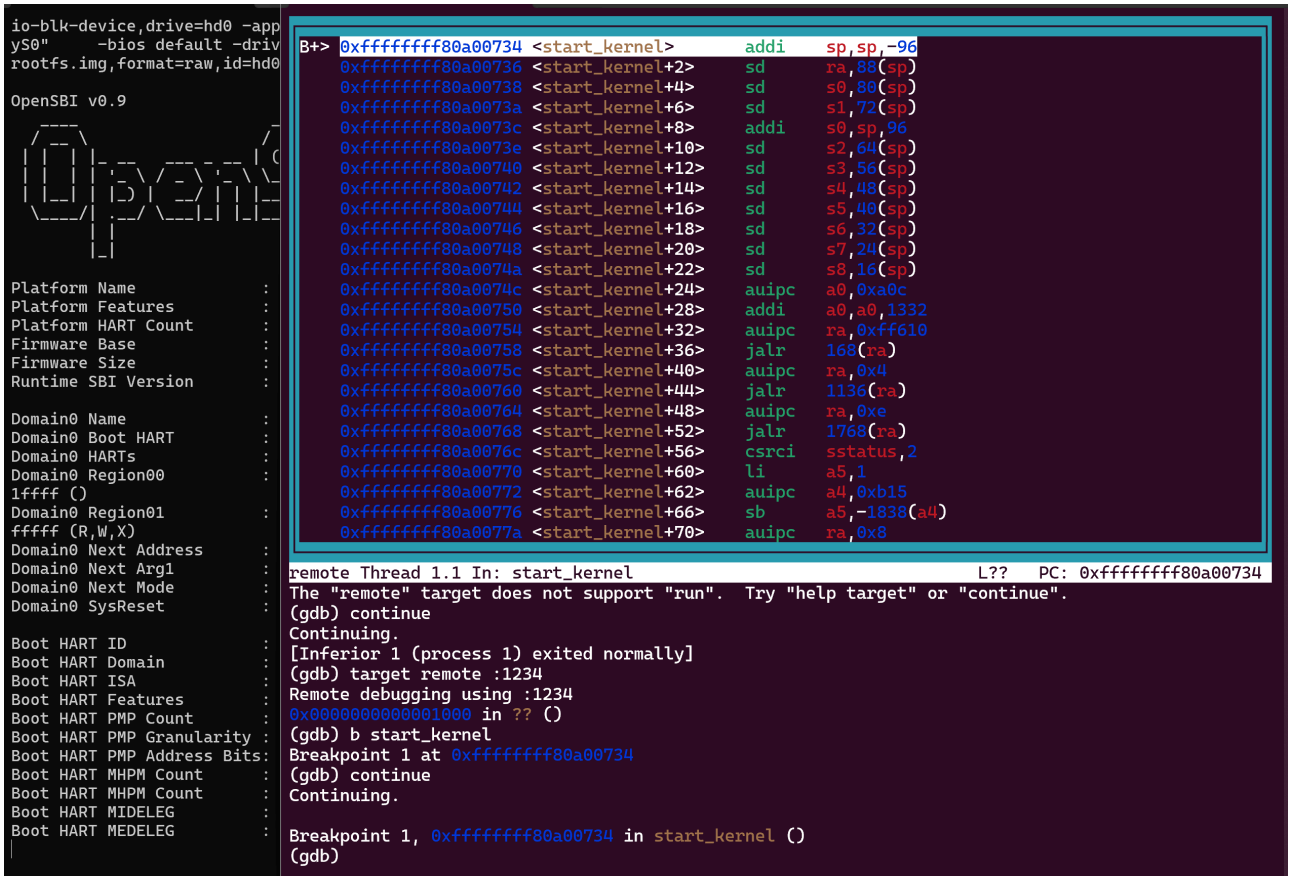


后续尝试了一些gdb命令：

- `b`（`break`的简写）：设置断点（见上图）
- `continue`：从断点后继续执行（见上图）
- `info`：提供各种信息，这里查看了断点信息（中间重跑了一遍上述代码，因此两个断点都在start_kernel处）

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff80a00734 <start_kernel>
        breakpoint already hit 2 times
2       breakpoint     keep y   0xffffffff80a00734 <start_kernel>
        breakpoint already hit 1 time
(gdb) bt
#0  0xffffffff80a00734 in start_kernel ()
#1  0xffffffff80001164 in _start_kernel ()
Backtrace stopped: frame did not save the PC
(gdb) f
#0  0xffffffff80a00734 in start_kernel ()
(gdb) f 1
#1  0xffffffff80001164 in _start_kernel ()
(gdb)
```

- `bt`（`backtrace`的简写）：查看函数的调用的栈帧和层级关系（见上图）
- `f`（`frame`的简写）：切换函数的栈帧，这里尝试切换了f0和f1（见上图）
- `layout asm`：显示汇编代码



- `next`：单步执行程序，逐行执行，不会进入函数内部

```
(gdb) next
Single stepping until exit from function start_kernel,
which has no line number information.
```

- `stepi`：执行单条指令

```
(gdb) stepi
0xffffffff80a0073c in start_kernel ()
(gdb) finish
Run till exit from #0  0xffffffff80a0073c in start_kernel ()
```

- `finish`：结束当前函数，返回到函数调用点（见上图）

3 实验中遇到的问题及解决方法

Q1：直接下载解压linux源码拖入相应文件夹，编译时报错`Permission denied`。

A1：利用wget下载源码，tar相关指令解压（代码见2.2）。

```
jyt@fine:~/os24fall-stu/src/lab0$ cd ../../../linux-6.10.10
jyt@fine:~/linux-6.10.10$ 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
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
  HOSTCC  scripts/kconfig/lexer.lex.o
  HOSTCC  scripts/kconfig/menu.o
  HOSTCC  scripts/kconfig/parser.tab.o
  HOSTCC  scripts/kconfig/preprocess.o
  HOSTCC  scripts/kconfig/symbol.o
  HOSTCC  scripts/kconfig/util.o
  HOSTLD  scripts/kconfig/conf
*** Default configuration is based on 'defconfig'
sh: 1: ./scripts/cc-version.sh: Permission denied
scripts/Kconfig.include:45: Sorry, this C compiler is not supported.
make[2]: *** [scripts/kconfig/Makefile:95: defconfig] Error 1
make[1]: *** [/home/jyt/linux-6.10.10/Makefile:695: defconfig] Error 2
make: *** [Makefile:240: __sub-make] Error 2
jyt@fine:~/linux-6.10.10$
```

Q2：使用QEMU运行内核时，报错`No such file or directory`。

A2：路径错误，将路径中`image`改成`Image`。

```
jyt@fine:~$ cd linux-6.11-rc7
jyt@fine:~/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=../os24fall-stu/src/lab0/rootfs.img,f
ormat=raw,id=hd0
./arch/riscv/boot/image: No such file or directory
qemu-system-riscv64: could not load kernel './arch/riscv/boot/image'
jyt@fine:~/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=../os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0

OpenSBI v0.9
```

4 思考题

1. 使用`riscv64-linux-gnu-gcc`编译单个`.c`文件

```
Please press Enter to activate this console. QEMU: Terminated
jyt@fine:~/linux-6.11-rc7$ cd ../
jyt@fine:~$ ls *.c
os_lab0_test.c
jyt@fine:~$ riscv64-linux-gnu-gcc -o os_lab0_test.exe os_lab0_test.c
jyt@fine:~$ ls *.exe
os_lab0_test.exe
jyt@fine:~$
```

os_lab0_test.c

```c
#include <stdio.h>
int main(void)
{
    int a, b;
    a = 10;
    b = 2 * a;
}
```

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

```
jyt@fine:~$ riscv64-linux-gnu-objdump -d os_lab0_test.exe

os_lab0_test.exe:     file format elf64-littleriscv


Disassembly of section .plt:

0000000000000540 <.plt>:
 540:   00002397                auipc   t2,0x2
 544:   41c30333                sub     t1,t1,t3
 548:   ac83be03                ld      t3,-1336(t2) # 2008 <__TMC_END__>
 54c:   fd430313                addi    t1,t1,-44
 550:   ac838293                addi    t0,t2,-1336
 554:   00135313                srli    t1,t1,0x1
 558:   0082b283                ld      t0,8(t0)
 55c:   000e0067                jr      t3

0000000000000560 <__libc_start_main@plt>:
 560:   00002e17                auipc   t3,0x2
 564:   ab8e3e03                ld      t3,-1352(t3) # 2018 <__libc_start_main@GLIBC_2.34>
 568:   000e0367                jalr    t1,t3
 56c:   00000013                nop

Disassembly of section .text:

0000000000000570 <_start>:
 570:   022000ef                jal     ra,592 <load_gp>
 574:   87aa                    mv      a5,a0
 576:   00002517                auipc   a0,0x2
 57a:   aba53503                ld      a0,-1350(a0) # 2030 <_GLOBAL_OFFSET_TABLE_+0x10>
 57e:   6582                    ld      a1,0(sp)
 580:   0030                    addi    a2,sp,8
```

主函数如下：

```
0000000000000628 <main>:
 628:   1101                    addi    sp,sp,-32
 62a:   ec22                    sd      s0,24(sp)
 62c:   1000                    addi    s0,sp,32
 62e:   47a9                    li      a5,10          a=10;
 630:   fef42423                sw      a5,-24(s0)
 634:   fe842783                lw      a5,-24(s0)
 638:   0017979b                slliw   a5,a5,0x1      b=2*a;
 63c:   fef42623                sw      a5,-20(s0)
 640:   4781                    li      a5,0
 642:   853e                    mv      a0,a5
 644:   6462                    ld      s0,24(sp)
 646:   6105                    addi    sp,sp,32
 648:   8082                    ret
jyt@fine:~$
```

3. 调试Linux时:

(a) 在 GDB 中查看汇编代码（不使用任何插件的情况下）

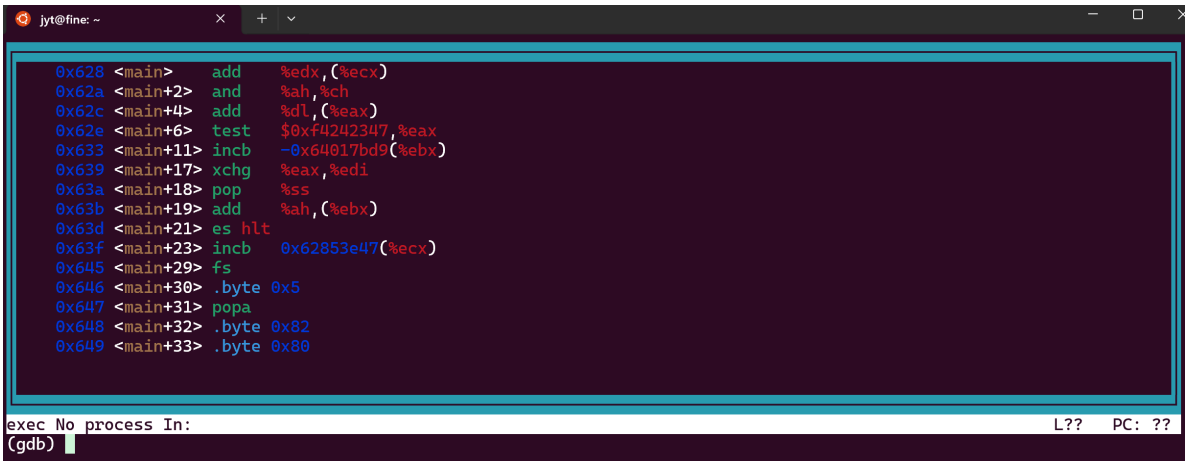
查阅资料后得知使用`disassemble`指令时，加入`/m`可使源码和汇编码一起排列，加入`/r`可以看到16进制代码。

`layout asm`指令显示汇编代码窗口。

```
$ gdb os_lab0_test.exe # 查看os_lab0_test对应汇编代码
(gdb) disassemble /m main
```

```
(gdb) disassemble /m main
Dump of assembler code for function main:
   0x00000628 <+0>:     add    %edx,(%ecx)
   0x0000062a <+2>:     and    %ah,%ch
   0x0000062c <+4>:     add    %dl,(%eax)
   0x0000062e <+6>:     test   $0xf4242347,%eax
   0x00000633 <+11>:    incb   -0x64017bd9(%ebx)
   0x00000639 <+17>:    xchg   %eax,%edi
   0x0000063a <+18>:    pop    %ss
   0x0000063b <+19>:    add    %ah,(%ebx)
   0x0000063d <+21>:    es hlt
   0x0000063f <+23>:    incb   0x62853e47(%ecx)
   0x00000645 <+29>:    fs
   0x00000646 <+30>:    .byte 0x5
   0x00000647 <+31>:    popa
   0x00000648 <+32>:    .byte 0x82
   0x00000649 <+33>:    .byte 0x80
End of assembler dump.
(gdb)
```

```
1 (gdb) layout asm
```




```
1 $ gdb-multiarch ./linux-6.11-rc7/vmlinux
2 (gdb) target remote :1234   # 连接 qemu
3 (gdb) layout asm # 查看linux内核的汇编代码
```

(b) 在 `0x80000000` 处下断点

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

```
exec No process In:
(gdb) b *0x80000000
Breakpoint 1 at 0x80000000
(gdb)
```

(c) 查看所有已下的断点

```
(gdb) i breakpoints
```

```
(gdb) i breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80000000
(gdb)
```

(d) 在 `0x80200000` 处下断点

```
(gdb) b *0x80200000
```

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

(e) 清除 `0x80000000` 处的断点

```
(gdb) delete 1
```

```
(gdb) delete 1
(gdb) i breakpoint
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x80200000
```

(f) 继续运行直到触发 `0x80200000` 处的断点

```
(gdb) continue
```

```
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb)
```

(g) 单步调试一次

```
(gdb) stepi
```

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

(h) 退出 QEMU

使用 `Ctrl` + `A`，松开后按下 `X` 退出QEMU：

```
Boot HART MHPM Count       : 0
Boot HART MIDELEG          : 0x0000000000000222
Boot HART MEDELEG          : 0x000000000000b109
QEMU: Terminated
jyt@fine:~$
```

4. 使用 `make` 工具清除 Linux 的构建产物

```
1 $ make clean
```

```
QEMU: Terminated
jyt@fine:~$ cd ./linux-6.11-rc7
jyt@fine:~/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
jyt@fine:~/linux-6.11-rc7$
```

5. `vmlinux` 和 `Image` 的关系和区别是什么?

   区别：`vmlinux` 是直接编译出来的原生内核镜像文件，未经压缩且包含调试信息，可以直接用于调试。可以用来开发/调试内核，但是不太适合将其部署到生产系统中。`Image` 同样是未压缩的，但是优化掉了一些信息（经过objcopy转换，去掉了一些符号表等信息）。它包含了实际运行在生产系统上的内核代码，但是一般不含调试信息，因此文件更小，适合部署到生产系统中。

   关系：二者都是未压缩的Linux内核镜像文件，构建Linux内核时会先生成 `vmlinux` 文件，然后将其转化成 `Image` 文件，以供实际使用。

5 心得体会

Lab0更多侧重于配置环境和熟悉相关操作，总体而言难度较小，踩坑较少。