

Lab 0: GDB + QEMU调试64位RISC-V Linux

赵伊蕾 学号: 3200104866

1 实验目的

- 安装Ubuntu、Docker
- 编译Linux内核
- 使用GDB、QEMU调试Linux内核

2 编译Linux内核

我的电脑是Unix系统的，为了更好体验操作系统，我在虚拟机上安装了Ubuntu然后下载Docker容器。使用git下载实验的框架，并且在 lab0 路径下下载Linux源代码进行编译。

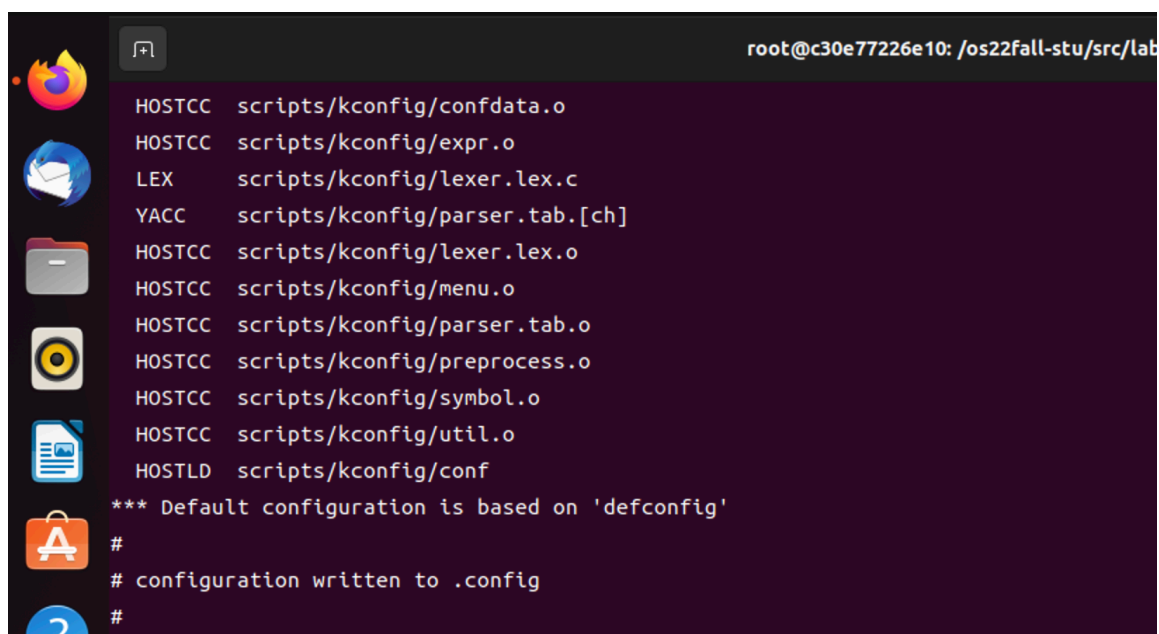
2.1 Linux源码形成配置

```
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig # 生成配置
```

这一步是对Linux的源代码形成配置。这一步指令正确执行之前，需要先保证Ubuntu环境之下已经配置好 make 指令、QEMU 环境、gcc 环境。

```
$ sudo apt-get install qemu
$ sudo apt-get install -y gcc make qemu build-essential module-assistant gcc-multilib
g++-multilib
$ sudo apt-get install -y gdb-multiarch
```

只有在Ubuntu安装好这些环境之后才能成功编译Linux。



2.2 编译内核

```
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc) # 编译
```

这一步是对Linux内核中的所有文件进行编译，形成 `.ko` 文件。考虑到在虚拟机下编译Linux，虚拟机的内核只有4位，所以为了防止内存耗尽，调低线程数位 `-j4`。以下为成功编译的结果。

```
LD [M] net/netfilter/xt_conntrack.ko
LD [M] net/netfilter/xt_ipvs.ko
LD [M] net/netfilter/xt_mark.ko
LD [M] net/netfilter/xt_nat.ko
LD [M] net/netfilter/xt_tcpudp.ko
LD [M] net/sched/cls_cgroup.ko
LD [M] net/xfrm/xfrm_algo.ko
LD [M] net/xfrm/xfrm_user.ko
root@c30e77226e10:/os22fall-stu/src/lab0/linux#
```

2.3 使用QEMU运行内核

QEMU是模拟处理器软件，可以在 x86 平台上执行不同架构下的程序。本实验用QEMU实现RISC-V架构的程序模拟，并且实现GDB与QEMU远程通信调试。

```
$ 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
```

QEMU参数含义：

- `-nographic`: 不使用图形窗口，使用命令行
- `-machine`: 指定要 emulate 的机器，可以通过命令 `qemu-system-riscv64 -machine help` 查看可选择的机器选项
- `-kernel`: 指定内核 image
- `-append cmdline`: 使用cmdline作为内核的命令行
- `-device`: 指定要模拟的设备，可以通过命令 `qemu-system-riscv64 -device help` 查看可选择的设备，通过命令 `qemu-system-riscv64 -device <具体的设备>,help` 查看某个设备的命令选项
- `-drive, file=<file_name>`: 使用 `file_name` 作为文件系统
- `-s`: 启动时暂停CPU执行
- `-s: -gdb tcp::1234` 的简写
- `-bios default`: 使用默认的 OpenSBI firmware 作为 bootloader

尤其是当GDB与QEMU要在两个Terminal中进行调试的时候，需要加上 `-s -s` 参数。

下图是QEMU成功运行内核的提示：

```
root@c30e77226e10:/os22fall-stu/src/lab0# qemu-system-riscv64 -nographic -machine virt -kernel linux/h/riscv/boot/Image -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" -bios fault -drive file=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
```

以下是编译成功，进入QEMU的提示与标志。

```
[ 0.870826] usbhid: USB HID core driver
[ 0.874275] NET: Registered PF_INET6 protocol family
[ 0.893744] Segment Routing with IPv6
[ 0.894554] In-situ OAM (IOAM) with IPv6
[ 0.895299] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[ 0.899678] NET: Registered PF_PACKET protocol family
[ 0.902358] 9pnet: Installing 9P2000 support
[ 0.903526] Key type dns_resolver registered
[ 0.906977] debug_vm_pgtable: [debug_vm_pgtable]: Validating architecture page table
[ 0.931740] Legacy PMU implementation is available
[ 1.007987] EXT4-fs (vda): mounted filesystem with ordered data mode. Quota mode: disabled.
[ 1.010759] VFS: Mounted root (ext4 filesystem) readonly on device 254:0.
[ 1.018024] devtmpfs: mounted
[ 1.115437] Freeing unused kernel image (initmem) memory: 2176K
[ 1.117705] Run /sbin/init as init process

Please press Enter to activate this console.

Please press Enter to activate this console.
/ #
```

2.4 使用GDB与QEMU远程通信调试

```
# 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
```

先启动QEMU，由于参数 `-s -s` QEMU会在启动时候暂停CPU的执行，所以在进入程序之前暂停下来。GDB 则可以通过新的终端，与QEMU连接进行远程通信，然后发送指令进行调试。

GDB连接、测试的代码如下：

```
(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
```

以下是我自己使用GDB测试的一些步骤与结果

- 设置、查询断点

GDB先使用(tcp:1234 port)建立通信, 然后设置一些断点, 可以通过start_kernel设置也可以通过手动对某地址添加断点。下图左侧的Terminal是用来发出GDB指令, 对远程QEMU进行调试; 右侧的Terminal是QEMU环境, 会显示调试的结果。

```
(gdb) b start_kernel
(gdb) b *0x80000000 #在0x80000000地址处设置断点
(gdb) info breakpoints #查询断点
```

```
(gdb) b *0x80000000
Breakpoint 1 at 0x80000000
(gdb) b *0xffffffff808006b8
Cannot access memory at address 0xffffffff808006b8
(gdb) b *0x80200000
Breakpoint 2 at 0x80200000
(gdb) info breakpoints
Num      Type      Disp Enb Address          What
1        breakpoint keep y  0x0000000080000000
2        breakpoint keep y  0x0000000080200000
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb) info breakpoints
Num      Type      Disp Enb Address          What
1        breakpoint keep y  0x0000000080000000
2        breakpoint keep y  0x0000000080200000
        breakpoint already hit 1 time
(gdb)

Domain0 HARTs           : 0*
Domain0 Region00       : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01       : 0x0000000000000000-0xffffffffffffffff (R
Domain0 Next Address    : 0x0000000080200000
Domain0 Next Arg1       : 0x0000000087000000
Domain0 Next Mode       : S-mode
Domain0 SysReset        : yes

Boot HART ID           : 0
Boot HART Domain        : root
Boot HART ISA           : rv64imafdcsu
Boot HART Features      : scounteren,mcounteren,time
Boot HART PMP Count     : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count    : 0
Boot HART MHPM Count    : 0
Boot HART MIDELEG       : 0x0000000000000222
Boot HART MEDELEG       : 0x000000000000b109
```

- 继续执行程序

GDB设置好断点之后可以通过 `continue` 指令让程序运行到下一断点为止。 `continue` 指令也可以简写为 `c`

```
(gdb) continue # 继续执行
```

```
Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb) info breakpoints
Num      Type      Disp Enb Address          What
1        breakpoint keep y  0x0000000080000000
2        breakpoint keep y  0x0000000080200000
        breakpoint already hit 1 time
(gdb) continue
Continuing.

cture page table helpers
[ 0.793002] Legacy PMU implementation is available
[ 0.880845] EXT4-fs (vda): mounted filesystem with ordered data mode. Quota
ode: disabled.
[ 0.881954] VFS: Mounted root (ext4 filesystem) readonly on device 254:0.
[ 0.885683] devtmpfs: mounted
[ 0.953485] Freeing unused kernel image (initmem) memory: 2176K
[ 0.954952] Run /sbin/init as init process

Please press Enter to activate this console.
```

- 删除断点

删除断点的方式比较简单，输入断点所对应的编号即可。例如我们一开始查询发现有1和2两个断点，执行了 `delete 1` 之后就只剩下2断点了。

```
(gdb) delete #num_of_breakpoint#
```

```
Num      Type      Disp Enb Address      What
1        breakpoint keep y  0x0000000080000000
2        breakpoint keep y  0x0000000080200000
(gdb) delete 1
(gdb) info breakpoints
Num      Type      Disp Enb Address      What
2        breakpoint keep y  0x0000000080200000
(gdb)
```

- backtrace指令

GDB的 `backtrace` 指令的作用是打印函数调用栈，可以把函数调用流程中的每一步执行的代码的地址。

```
(gdb) backtrace
```

```
(gdb) backtrace
#0  0x0000000000000100 in ?? ()
```

- 继续运行到触发 `0x80200000`

在地址 `0x80200000` 处设置断点，然后执行 `continue` 运行到该断点处。我们也可以看下面 `display` 指令的结果，显示了从当前位置往后的所有汇编代码。这也可以证明刚才我们执行到 `0x80200000` 地址。

- 查看汇编代码

```
(gdb) display /20i $pc
```

`display` 指令是从当前位置开始，输出后面所有的汇编代码。

```

Breakpoint 2, 0x0000000080200000 in ?? ()
1: x/20i $pc
=> 0x80200000: li      s4,-13
    0x80200002: j       0x802010cc
    0x80200006: nop
    0x80200008: unimp
    0x8020000a: addi    s0,sp,8
    0x8020000c: unimp
    0x8020000e: unimp
    0x80200010: sw      s0,32(s0)
    0x80200012: addi    sp,sp,13
    0x80200014: unimp
    0x80200016: unimp
    0x80200018: unimp
    0x8020001a: unimp
    0x8020001c: unimp
    0x8020001e: unimp
    0x80200020: c.slli64      zero
    0x80200022: unimp
    0x80200024: unimp
    0x80200026: unimp
--Type <RET> for more, q to quit, c to continue without paging--c
    0x80200028: unimp
(gdb)

```

- 单步运行

- Next 指令

GDB的next指令通常被用来单步/多步运行测试代码。

```
(gdb) next count    #count步长，默认下为1
```

```

Breakpoint 1, 0x0000000080000000 in ?? ()
(gdb) next
Cannot find bounds of current function
(gdb)

```

- Step 指令

GDB的Step指令与Next指令的功能相同，都是单步执行程序，不同之处在于，当 step 命令所执行的代码行中包含函数时，会进入该函数内部，并在函数第一行代码处停止执行。

```
(gdb) next count    #count步长，默认下为1
```

- Until 指令

GDB的Until指令可以指定运行到具体某一地址然后截止。

```

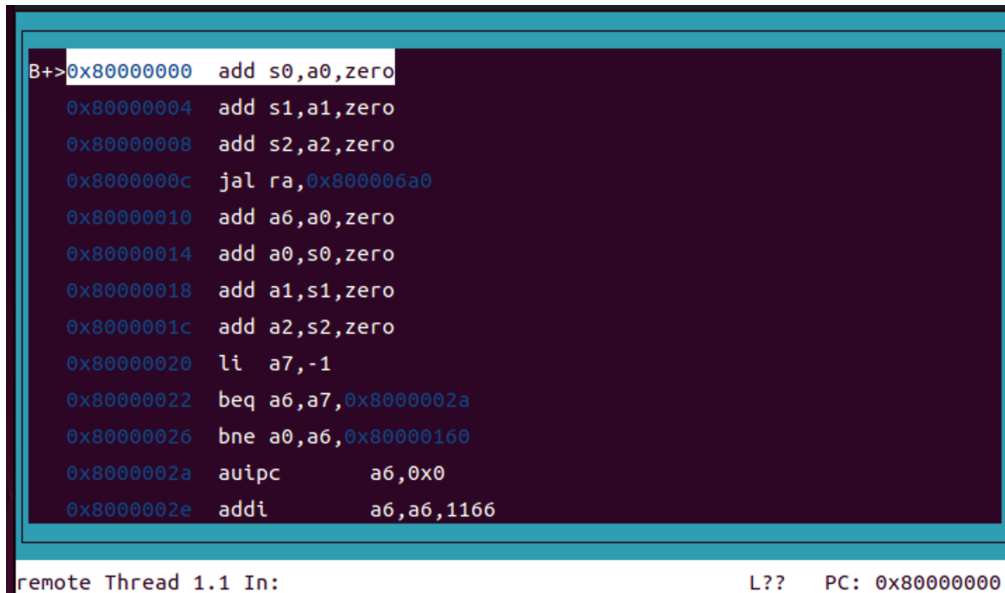
(gdb) until          #可以简写为u
(gdb) until location

```

- layout指令

`layout` 指令的作用是分割窗口，使得可以一边查看代码，以便进行测试。

```
(gdb) layout src          #显示源代码窗口
(gdb) layout asm          #显示汇编窗口
(gdb) layout regs         #显示源代码/汇编和寄存器窗口
(gdb) layout split        #显示源代码和汇编窗口
(gdb) layout next         #显示下一个layout
(gdb) layout prev         #显示上一个layout
```



```
B+>0x80000000 add s0,a0,zero
0x80000004 add s1,a1,zero
0x80000008 add s2,a2,zero
0x8000000c jal ra,0x800006a0
0x80000010 add a6,a0,zero
0x80000014 add a0,s0,zero
0x80000018 add a1,s1,zero
0x8000001c add a2,s2,zero
0x80000020 li a7,-1
0x80000022 beq a6,a7,0x8000002a
0x80000026 bne a0,a6,0x80000160
0x8000002a auipc a6,0x0
0x8000002e addi a6,a6,1166

remote Thread 1.1 In: L?? PC: 0x80000000
```

GDB与C语言调试

为了更好显示GDB对于C语言程序调试的方便性，我在 `lab0` 文件夹下创建了简单的 `.c` 文件，用GDB进行调试。编写简单的C语言文件，其中包括循环、输出等基本操作以探究程序运行的结构。具体代码如下：

```
#include<stdio.h>
int main(){

    int i = 0;
    for(i; i<10;i++)
        printf("%d ",i);
}
```

- 对该文件进行编译

```
riscv64-unknown-elf gcc test.c
```

```
root@c30e77226e10:/os22fall-stu/src/lab0# riscv64-unknown-elf-gcc test.c
```

编译通过!

- 对已经编译好的文件进行反汇编

```
riscv64-unknown-elf-objdump
```

得到汇编代码如下：

```
000000000001c23c <__clzdi2>:
 1c23c: 03800793      li    a5,56
 1c240: 00f55733      srl   a4,a0,a5
 1c244: 0ff77713      zext.b a4,a4
 1c248: e319         bnez   a4,1c24e <__clzdi2+0x12>
 1c24a: 17e1         addi   a5,a5,-8
 1c24c: fbf5         bnez   a5,1c240 <__clzdi2+0x4>
 1c24e: 6775         lui    a4,0x1d
 1c250: 04000693      li    a3,64
 1c254: 8e9d         sub    a3,a3,a5
 1c256: 00f55533      srl   a0,a0,a5
 1c25a: 08870793      addi   a5,a4,136 # 1d088 <__clz_tab>
 1c25e: 97aa         add    a5,a5,a0
 1c260: 0007c503      lbu    a0,0(a5)
 1c264: 40a6853b      subw   a0,a3,a0
 1c268: 8082         ret
root@c30e77226e10:/os22fall-stu/src/lab0#
```