浙江大学

课程名称	:	操作系统原理与实践
题　　目	:	Lab 0: GDB & QEMU 调试 64 位 RISC-V LINUX
授课教师	:	申文博
助　　教	:	王鹤翔、陈淦豪、许昊瑞
姓　　名	:	潘潇然
学　　号	:	3220106049
地　　点	:	32舍367

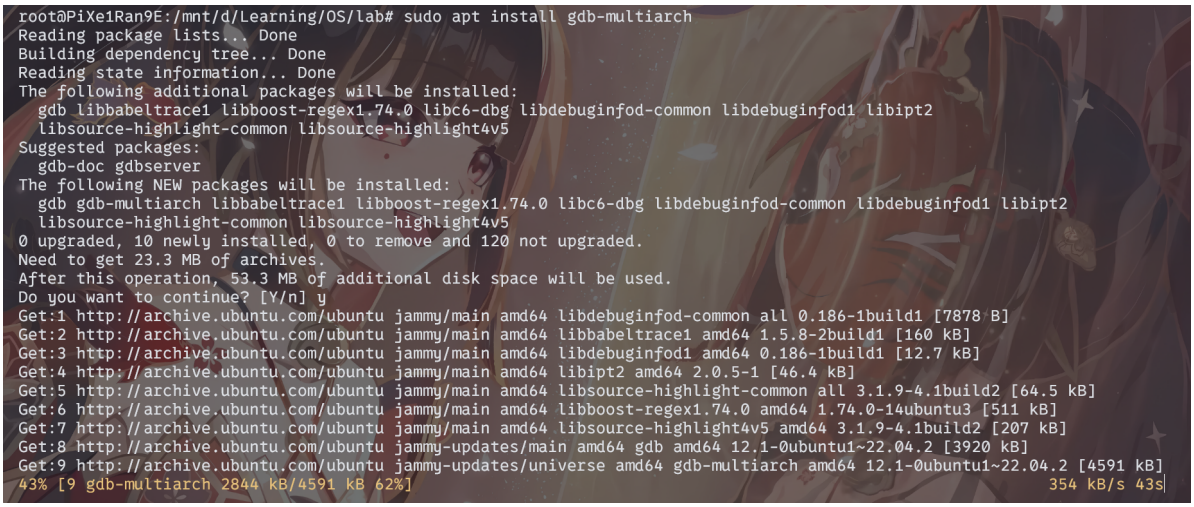
一、实验过程与步骤

1. 实验环境搭建

- 首先`sudo apt update`以更新环境内的软件包列表
- 之后运行`sudo apt install  gcc-riscv64-linux-gnu`和`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`以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

- 运行`sudo apt install qemu-system-misc`以安装qemu

- 运行`sudo apt install gdb-multiarch`以安装gdb进行调试工作



2. 获取 Linux 源码和已经编译好的文件系统

- 为了正常编译Linux内核，我们需要在linux内用wget获取最新Linux源码，同时不能放置在`/mnt`目录，此处放置在`/usr`目录，运行`sudo wget https://git.kernel.org/torvalds/t/linux-6.11-rc7.tar.gz`

- 之后解压文件`sudo tar -xzvf linux-6.11-rc7.tar.gz`，这里由于速度太快没截到图
- git clone实验指导仓库，之后观察到在`lab0`目录下有`rootfs.img`镜像

```
root@PiXe1Ran9E:/usr# git clone https://github.com/ZJU-SEC/os24fall-stu.git
Cloning into 'os24fall-stu'...
remote: Enumerating objects: 78, done.
remote: Counting objects: 100% (78/78), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 78 (delta 33), reused 66 (delta 21), pack-reused 0 (from 0)
Receiving objects: 100% (78/78), 1.97 MiB | 88.00 KiB/s, done.
Resolving deltas: 100% (33/33), done.
root@PiXe1Ran9E:/usr# cd os24fall-stu/src/lab0
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab0# ls
rootfs.img
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab0#
```

3. 编译Linux内核

- 使用默认配置(`defconfig`)

```
root@PiXe1Ran9E:/usr/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
  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'
#
# configuration written to .config
#
```

- 为避免内存耗尽，同时避免过高并行度导致编译出错，因此使用8线程编译，`make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j8`

```
root@PiXe1Ran9E:/usr/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/fcntl.h
  WRAP    arch/riscv/include/generated/uapi/asm/ioctl.h
  WRAP    arch/riscv/include/generated/uapi/asm/ioctls.h
  WRAP    arch/riscv/include/generated/uapi/asm/ipcbuf.h
  WRAP    arch/riscv/include/generated/uapi/asm/mman.h
  WRAP    arch/riscv/include/generated/uapi/asm/msgbuf.h
  WRAP    arch/riscv/include/generated/uapi/asm/param.h
  WRAP    arch/riscv/include/generated/uapi/asm/poll.h
  WRAP    arch/riscv/include/generated/uapi/asm/posix_types.h
  WRAP    arch/riscv/include/generated/uapi/asm/resource.h
  HOSTCC  scripts/dtc/dtc.o
  WRAP    arch/riscv/include/generated/uapi/asm/sembuf.h
```

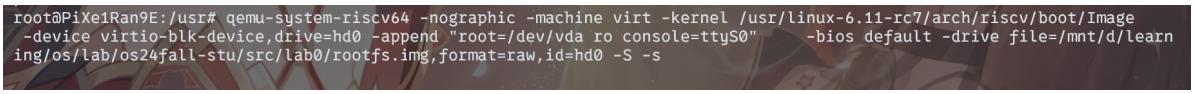
4. 使用QEMU运行内核

- 运行下图命令以运行内核，注意修改镜像文件的路径

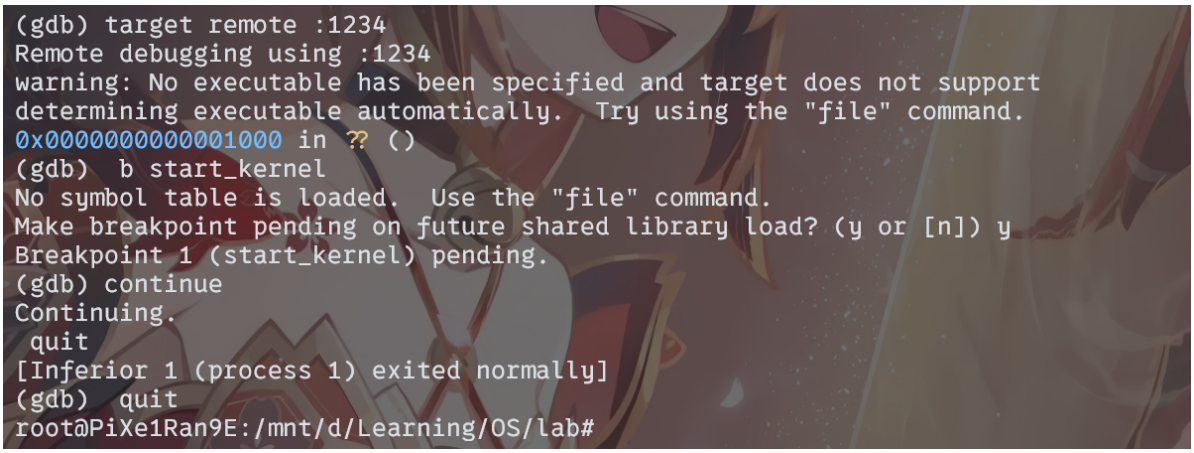
```
root@PiXe1Ran9E:/usr# qemu-system-riscv64 -nographic -machine virt -kernel /usr/linux-6.11-rc7/arch/riscv/boot/Image
 -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"     -bios default -drive file=/mnt/d/learn
ing/os/lab/os24fall-stu/src/lab0/rootfs.img,format=raw,id=hd0

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
```

5. 使用 GDB 对内核进行调试

- 这里需要打开两个terminal，一个使用QEMU启动Linux，另一个使用GDB与QEMU远程通信
- 首先打开一个terminal运行QEMU，注意与上一步不同的是，需要在命令最后加上`-s -s`，其中`-s`表示QEMU在启动后需要等待GDB连接，而不是立刻开始执行CPU，`-s`是`-gdb tcp::1234`的简写，告诉QEMU在tcp端口1234上启动一个GDB服务器。因此运行后不会立即产生任何输出



- 之后开启另一个terminal，先输入`gdb-multiarch /usr/linux-6.11-rc7/vmlinux`启动gdb，之后利用gdb进行连接qemu，设置断点等操作，之后继续执行，最后退出gdb



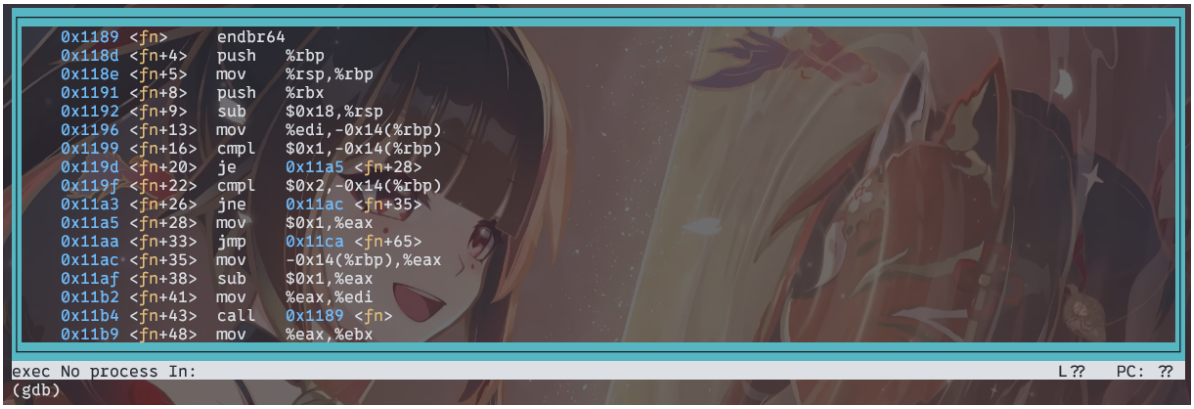
6. 接下来，对一个具体的C语言程序利用gdb进行调试，练习gdb各项基本指令

- 首先编写一个有嵌套调用的程序，这里简单以一个计算斐波那契数列的函数作为例子

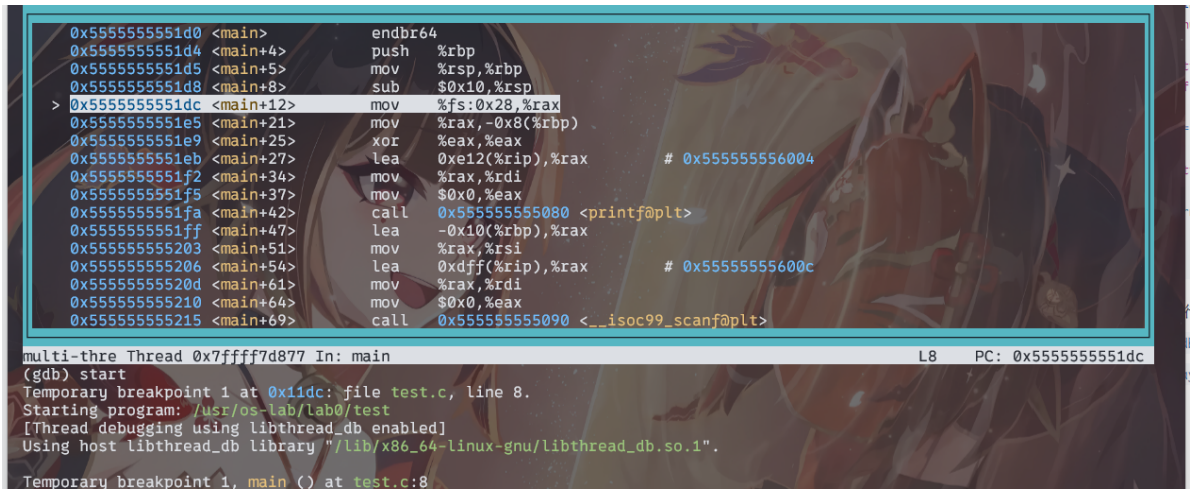
```c
#include <stdio.h>

int fn(int n) {
  if (n == 1 || n == 2) return 1;
  return fn(n - 1) + fn(n - 2);
}

int main() {
  int num, sum;

  printf("Input: ");
  scanf("%d", &num);

  sum = fn(num);

  printf("Sum: %d\n", sum);

  return 0;
}
```

- 之后运行`gcc -g test.c -o test`对此程序进行编译

- 运行`gdb test`进入调试
- 首先`layout asm`查看汇编代码

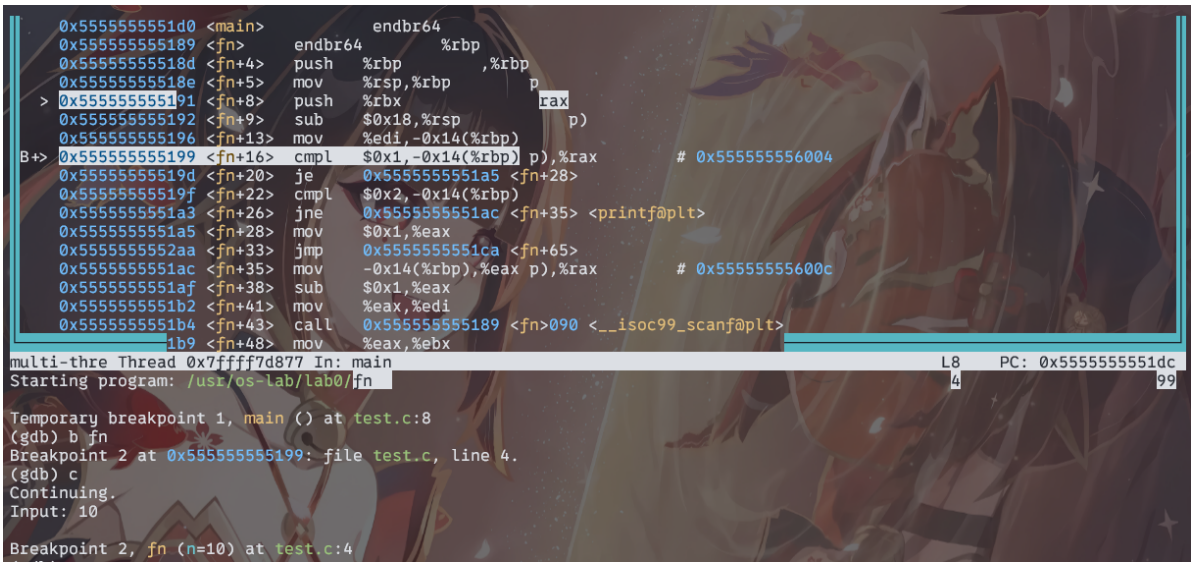


- 之后`start`开始运行程序，将会停止在`main`函数开头

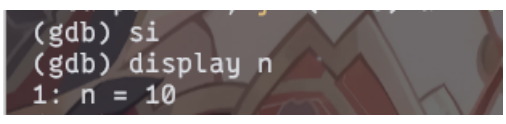


- `b fn`在`fn`函数入口设立断点

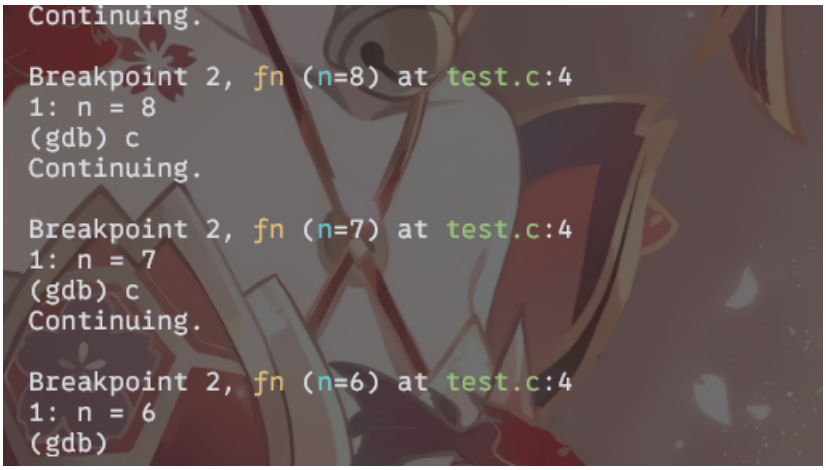
- `c`(continue)执行到程序第一次遇到`fn`



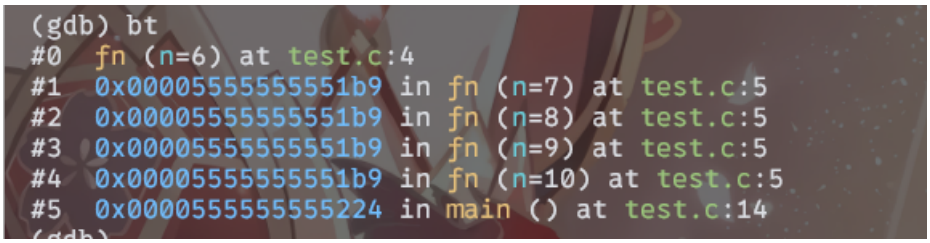
- `display n`查看`n`值



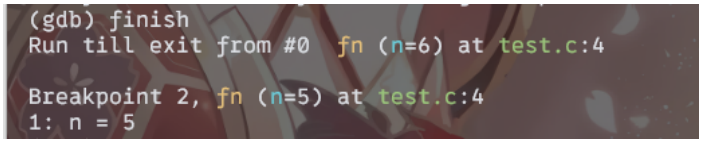
- 之后我们可以通过反复`c`，嵌套调用`fn`函数，可以观察到`n`的值在不断变化



- 这时可以运行`bt`查看函数的调用的栈帧和层级关系



- 最后运行`finish`，可以执行到函数末尾，而`n=6`的末尾自然就是执行`n=5`



二、实验心得体会

这个实验总体还是比较顺利的，主要的问题就是一开始没有注意常见问题中提到的不能在`/mnt`目录下进行Linux源码的编译，导致编译了一两节课还没编译完，后面才发现此问题。这个实验做下来，整体对wsl的文件结构更熟悉了一些，同时也更熟悉了gdb的基本使用。

三、思考题

1. 为了后续操作，首先写一个简单的`.c`文件，主要内容是一个简单的从1到10的循环，并输出对应数字

```c
#include <stdio.h>

int main() {
   int i;
   for (i = 1; i <= 10; i++)  printf("%d\n", i);
   return 0;
}
```

2. 之后使用`riscv64-linux-gnu-gcc`编译此文件，之后利用`riscv64-linux-gnu-objdump`反编译前一步得到的编译产物并将输出重定向到`new.txt`文件中。其中`riscv64-linux-gnu-gcc`将C语言编译生成适用于RISC-V64位架构并在Linux上运行的可执行文件，`>`表示重定向

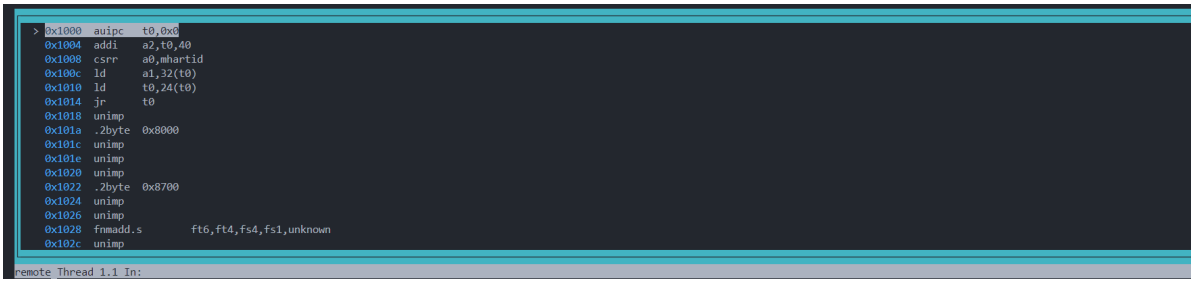
```
root@PiXe1Ran9E:/usr/os-lab/lab0# riscv64-linux-gnu-gcc -o new new.c
root@PiXe1Ran9E:/usr/os-lab/lab0# riscv64-linux-gnu-objdump -d new > new.txt
```

3. 之后我们可以打开`new.txt`查看汇编代码，可以观察到以下汇编代码完成了循环条件的判断和`printf`

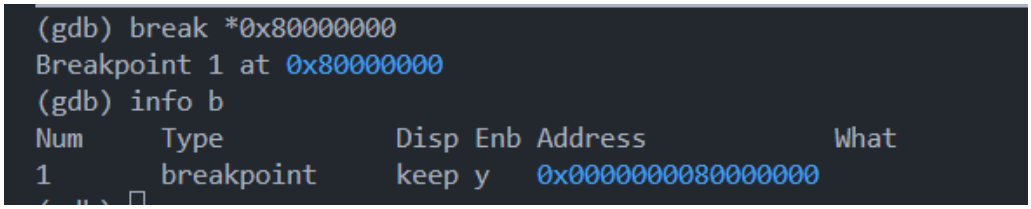
```
678:   fec42783          lw  a5,-20(s0)
67c:   85be               mv  a1,a5
67e:   00000517           auipc   a0,0x0
682:   03a50513           addi    a0,a0,58 # 6b8 <_IO_stdin_used+0x8>
686:   f1bff0ef         jal ra,5a0 <printf@plt>
68a:   fec42783          lw  a5,-20(s0)
68e:   2785               addiw   a5,a5,1
690:   fef42623          sw  a5,-20(s0)
694:   fec42783          lw  a5,-20(s0)
698:   0007871b           sext.w  a4,a5
69c:   47a9               li  a5,10
69e:   fce7dde3          bge a5,a4,678 <main+0x10>
```

4. 接下来进行Linux调试，首先需要按实验步骤5打开两个terminal，将gdb与qemu连接，之后运行下述指令

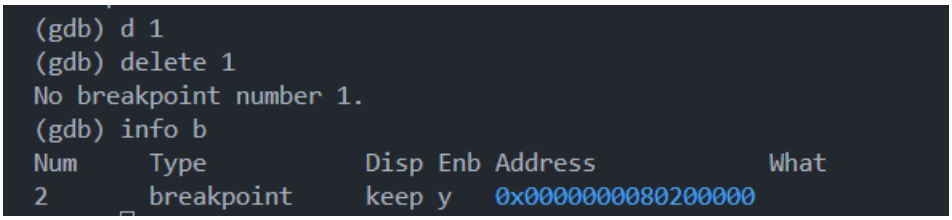
- 运行`layout asm`显示汇编代码



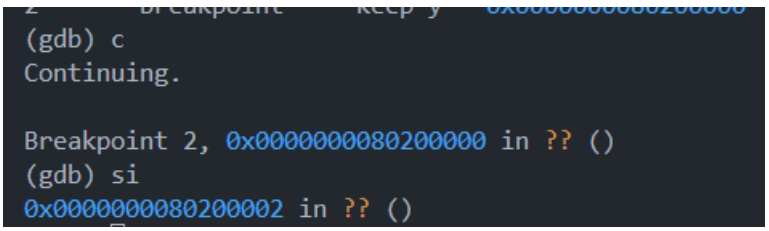
- 在`0x80000000`处下断点并查看所有已下的断点，发现成功设置断点



- 在`0x80200000`处下断点后清除`0x80000000`处的断点，发现成功删除



- 继续运行直到触发`0x80200000`处的断点，并单步调试一次。发现成功在断点处停止，同时单步调试成功



-

- 最后`quit`退出qemu

5. 使用 `make` 工具清除 Linux 的构建产物，我们进入`linux-6.11-rc7`目录，运行`make clean`

```
root@PiXe1Ran9E:/usr/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
```

6. `vmlinux`和`image`的联系和区别

- 区别：
  - `vmlinux`(其中`vm`指的是`virtual memory`，Linux支持虚拟内存)是编译Linux内核得到的最原始的内核文件，是`elf`格式(Executable and Linkable Format)的文件，未经压缩，文件比较大，多存放在PC上。`vmlinux`主要用于加载和运行Linux内核。计算机启动时，Boot Loader会加载`vmlinux`文件，首先将其复制到系统内存中，之后开始执行代码。`vmlinux`包含了Linux的所有代码和数据结构，还包含调试符号等信息(如函数名称、变量名称等)
  - `image`是Linux内核镜像文件，但仅包含可执行的二进制数据
- 联系：用户对Linux内核源码进行编译，会先生成`vmlinux`，之后由于此文件过大，要经过`objcopy`处理成只包含二进制数据的内核代码，去除掉不需要的文件信息，即`image`