

Lab 0: RV64 内核调试

1 实验目的

按照实验流程搭建实验环境，掌握基本的 Linux 概念与用法，熟悉如何从 Linux 源代码开始将内核运行在 QEMU 模拟器上，学习使用 GDB 跟 QEMU 对代码进行调试，为后续实验打下基础。

2 实验内容及要求

- 学习 Linux 基本知识
- 安装 Docker，下载并导入 Docker 镜像，熟悉 docker 相关指令
- 编译内核并用 GDB + QEMU 调试，在内核初始化过程中设置断点，对内核的启动过程进行跟踪，并尝试使用 GDB 的各项命令

请各位同学独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据本文档中的要求记录实验过程，最后删除文档末尾的附录部分，将文档导出并命名为“**学号姓名lab0.pdf**”，以 pdf 格式上传至学在浙大平台。

3 操作方法和实验步骤

3.1 安装 Docker 环境并创建容器 (25%)

请参考【[附录B.Docker使用基础](#)】了解相关背景知识。

3.1.1 安装 Docker 并启动

请参照 <https://docs.docker.com/get-docker/> 自行在本机安装 Docker 环境，安装完成后启动 Docker 软件。

3.1.2 下载并导入 Docker 镜像

为了便于开展实验，我们在 [镜像](#) 中提前安装好了实验所需的环境（RISC-V工具链、QEMU模拟器），相关环境变量也以设置完毕。**请下载该 Docker 镜像至本地。**

下载好的镜像包不需要解压，后面命令中直接使用。

接下来建议大家使用终端操作，而非使用桌面端等 UI 程序，这样每一步操作有迹可循，易于排查问题。

- Windows 用户：可以使用系统自带的 PowerShell 软件，命令提示符 (cmd) 软件不推荐使用。
- MacOS 用户：使用默认终端即可。
- Linux 用户：使用默认终端即可。

在执行每一条命令前，请你对将要进行的操作进行思考，给出的命令不需要全部执行，并且不是所有的命令都可以无条件执行，请不要直接复制粘贴命令去执行。

以下给出的指令中，**\$** 提示符表示当前运行的用户为普通用户，**#** 代表 Shell 中注释的标志，他们并非实际输入指令的一部分。

导入失败的同学请查看自己的 C 盘空间是否满。

```
# 进入 oslab.tar 所在的文件夹
$ cd path/to/oslab # 替换为你下载文件的实际路径

# 导入docker镜像
$ docker import oslab.tar oslab:2023

# 查看docker镜像
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
oslab	2023	9192b7dc0d06	47 seconds ago	2.89GB

请在此处添加你导入容器的执行命令及结果截图：

答：

```
riceshower@Hachimi:~$ docker import oslab.tar oslab:2023
sha256:2dcce6da10c98a0ea772a824a3abad46bec7d0671799aa3c13e44d7bf79e45d9
riceshower@Hachimi:~$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
oslab	2023	2dcce6da10c9	12 seconds ago	2.89GB

```
riceshower@Hachimi:~$
```

3.1.3 从镜像创建一个容器并进入该容器

请按照以下方法创建新的容器，并建立 volume 映射([参考资料](#))。建立映射后，你可以方便的在本地编写代码，并在容器内进行编译检查。未来的实验中同样需要用该方法搭建相应的实验环境，但不再作具体指导，请理解每一步的命令并自行更新相关内容。

什么是 volume 映射？其实就是把本地的一个文件夹共享给 Docker 容器用，无论你在容器内修改还是在本地环境下修改，另一边都能感受到这个文件夹变化了。

你也可以参照知识库中提供的，通过配置 VSCode 智能提示来直接连接到 Docker 容器内进行实验，如若此，请提供你使用软件直接在 Docker 容器内进行编辑的截图即可。下文的建立映射关系可以跳过。

如果你使用 VSCode 或其他具有直接连接 Docker 容器功能的软件，你也可以直接在 Docker 容器内进行编辑，而无需建立映射关系，如若此，请提供你使用软件直接在 Docker 容器内进行编辑的截图即可。

Windows 中的路径一般分 C, D, E 等多盘符，因此 Windows 下的路径一般为 xx盘符:\xx路径，例如 C:\Users\Administrator，而 Linux 下与 Windows 不同，Linux 只有一个根目录 /，例如 /home/oslab/lab1 表示 根目录下的 home 文件夹下的 oslab 文件夹下的 lab1 文件夹，在映射路径时请按照自己系统的路径描述方法填写。更多细节可自行搜索学习。

Linux 下一般默认 /home 文件夹用来存放用户文件，而别的路径用来存放系统文件，因此请在映射文件夹的时候映射到 /home 的文件夹目录下。/home/aaa 表示 aaa 用户的用户文件所在目录，同理 /home/oslab 表示 oslab 用户的用户文件所在目录。如果你使用的是虚拟机，请映射到 /home/自己用户名 的目录下，一般情况下 ~ 符号等价于 /home/当前用户名，详情请自行搜索 Linux 下 /home 目录含义。

一般来说, aaa 用户不能访问 bbb 用户的用户文件, 也就是不能访问/修改 /home/bbb 文件夹。但 Docker 容器中用的是 root 用户登录, 相当于 Windows 中的管理员权限, 因此可以访问 /home/oslab 下的文件。

指令仅做参考, 注意修改指令中的路径为你自己设置的路径。**如果你使用的是 Windows 系统, 建议**不要将本地新建的目录放在 C 盘等位置。避免后续指令权限问题。本地目录和映射的目录路径不需要相同。****

```
# 首先请在本地新建一个目录用作映射需要
$ cd /path/to/your/local/dir
$ mkdir os_experiment

# 创建新的容器, 同时建立 volume 映射
$ docker run -it -v
/path/to/your/local/dir/os_experiment:/home/oslab/os_experiment oslab:2023
/bin/bash
oslab@3c1da3906541:~$
```

请在此处添加一张你执行 Docker 映射的命令及结果截图:

答:

```
riceshower@Hachimi:~$ mkdir homework
riceshower@Hachimi:~$ mkdir homework/os_experiment
riceshower@Hachimi:~$ docker run -it -v /home/riceshower/homework/os_experiment:/home/oslab/os_experiment oslab:2023 /bin/bash
root@f8b3720181fc:/#
```

请解释该命令各参数含义:

- `docker run -it -v`
`/path/to/your/local/dir/os_experiment:/home/oslab/os_experiment oslab:2023`
`/bin/bash`

答: `-i` 意为允许在容器内输入, 即保持标准输入打开。 `-t` 让 Docker 为容器分配一个伪终端, 以便可以在终端中进行交互。 `-v` 用于将本地文件系统中的目录与容器中的目录进行挂载, `/path/to/your/local/dir/os_experiment` 是本地主机上的目录路径, `/home/oslab/os_experiment` 是容器内的目录路径。 `oslab:2023` 为要运行的 docker 镜像的名称以及标签。 `/bin/bash` 是在容器内要运行的命令, 启动一个 Bash shell。

3.1.4 测试映射关系

为测试映射关系是否成功, 你可以在本地映射目录中创建任意文件, 并在 Docker 容器中进行检查。

```
# 在你的本地映射目录中, 创建任意文件
$ cd /path/to/your/local/dir/os_experiment
$ touch testfile
$ ls
testfile
```

以上指令将在你的本地映射目录创建一个文件, 接下来在容器中执行指令进行检查。

```
# 在 Docker 容器中确认是否挂载成功
root@dac72a2cc625:/home/oslab/os_experiment$ ls
testfile
# 退出docker，退出后容器将变为关闭状态，再次进入时需要重新启动容器（不是重新创建容器）
root@dac72a2cc625:/home/oslab/os_experiment$ exit
```

可以看到创建的文件存在，证明映射关系建立成功，接下来你可以使用你喜欢的 IDE 在该目录下进行后续实验的编码了。

请在此处添加你测试映射关系的全指令截图：

答：

```
riceshower@Hachimi:~$ cd homework/os_experiment/
riceshower@Hachimi:~/homework/os_experiment$ touch testfile
riceshower@Hachimi:~/homework/os_experiment$ ls
testfile
riceshower@Hachimi:~/homework/os_experiment$ docker start f8b3720181fc
f8b3720181fc
riceshower@Hachimi:~/homework/os_experiment$ docker attach f8b3720181fc
root@f8b3720181fc:/# cd /home/oslab/os_experiment/
root@f8b3720181fc:/home/oslab/os_experiment# ls
testfile
root@f8b3720181fc:/home/oslab/os_experiment# exit
exit
```

其他常用docker指令如下，在后续的实验过程中将会经常使用这些命令：

```
# 查看当前运行的容器
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
95efacf34d2c	oslab:2023	"/bin/bash"	About a minute ago	Exited (0)	About a minute ago	oslab

```
# 查看所有存在的容器
$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
95efacf34d2c	oslab:2023	"/bin/bash"	About a minute ago	Exited (0)

```
# 启动处于停止状态的容器
$ docker start oslab
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
95efacf34d2c	oslab:2023	"/bin/bash"	2 minutes ago	Up 26 seconds	

```
# 进入已经运行的容器
$ docker attach oslab
root@95efacf34d2c:/#

# 在已经运行的docker中运行/bin/bash命令，开启一个新的进程
$ docker exec -it oslab /bin/bash
root@95efacf34d2c:/#
```

3.2 编译 Linux 内核 (25%)

请参考[【附录E.LINUX 内核编译基础】](#)了解相关背景知识。

```
# 以下指令均在容器中操作

# 进入实验目录
$ cd /home/oslab/lab0

# 查看当前目录文件
$ ls
linux  rootfs.ext4

# 创建目录，用来存储编译结果
$ mkdir -p build/linux

# 编译 Linux 内核
$ make -C linux \
    O=/home/oslab/lab0/build/linux \
    CROSS_COMPILE=riscv64-unknown-linux-gnu- \
    ARCH=riscv \
    CONFIG_DEBUG_INFO=y \
    defconfig \
    all \
    -j$(nproc)
```

有关 make 指令和 makefile 的知识将在 Lab1 进一步学习。这里简单介绍一下编译 Linux 内核各参数的含义。

<code>-C linux</code>	表示进入 linux 文件夹，并执行该目录下的 makefile 文件。因此，你执行该命令时应在 <code>/home/oslab/lab0</code> 路径下。
<code>O=.....</code>	指定变量 O 的值，O 变量在 linux makefile 里用来表示编译结果输出的路径
<code>CROSS_COMPILE=.....</code>	指定变量 CROSS_COMPILE 的值，linux makefile 中使用 CROSS_COMPILE 变量的值作为前缀选择编译时使用的工具链。例如本例子中， <code>riscv64-unknown-linux-gnu-gcc</code> 即是实际编译时调用的编译器。
<code>ARCH=.....</code>	指定编译的目标平台
<code>CONFIG_DEBUG_INFO=y</code>	同上，当该变量设置时，编译过程中将加入 <code>-g</code> 配置，这会使得编译结果是包含调试信息的，只有这样我们才可以比较好的进行调试。
<code>defconfig</code>	指定本次编译的目标，支持什么编译目标是 linux makefile 中已经定义好的，defconfig 就表示本次编译要编译出 defconfig 这个目标，该目标代表编译需要的一些配置文件。
<code>all</code>	指定本次编译的目标，目标是可以有多个的。这里的 all 并不表示编译所有目标，而是 makefile 中定义好的一个名称为 all 的编译目标。该目标代表 linux 内核。

<code>-C linux</code>	表示进入 linux 文件夹，并执行该目录下的 makefile 文件。因此，你执行该命令时应在 <code>/home/oslab/lab0</code> 路径下。
<code>-j\$(nproc)</code>	<code>-j</code> 表示采用多线程编译，后跟数字表示采用线程数量。例如 <code>-j4</code> 表示 4 线程编译。这里的 <code>\$(nproc)</code> 是 shell 的一种语法，表示执行 <code>nproc</code> 命令，并将执行的结果替换这段字符串。 <code>nproc</code> 命令会返回本机器的核心数量。

编译报错为 Error.137 的同学，可能是电脑性能不足，可以将最后的参数改为 `-j1`，降低资源消耗。


在docker的设置中可以调整给docker分配的资源数量，可以根据需要适当调整。



如果还不行的话，可以给电脑分配更多的虚拟内存解决，Linux 上可以创建 swap 分区，Windows 上可以在我的电脑，高级设置中修改虚拟内存大小。

请在此处添加一张你的编译完成的结果截图：

答：

 root@8ab9e30c1532: /home/oslab/lab0

```
CC      drivers/gpu/drm/radeon/r600_cs.o
CC      drivers/gpu/drm/radeon/evergreen_cs.o
CC      drivers/gpu/drm/radeon/r100.o
CC      drivers/gpu/drm/radeon/r300.o
CC      drivers/gpu/drm/radeon/r420.o
AR      drivers/gpu/drm/radeon/built-in.a
AR      drivers/gpu/drm/built-in.a
AR      drivers/gpu/built-in.a
AR      drivers/built-in.a
GEN      .version
CHK      include/generated/compile.h
LD      vmlinux.o
MODPOST vmlinux.symvers
MODINFO modules.builtin.modinfo
GEN      modules.builtin
LD      .tmp_vmlinux.kallsyms1
KSYM     .tmp_vmlinux.kallsyms1.o
LD      .tmp_vmlinux.kallsyms2
KSYM     .tmp_vmlinux.kallsyms2.o
LD      vmlinux
SYSMAP   System.map
MODPOST  Module.symvers
OBJCOPY  arch/riscv/boot/Image
CC [M]   fs/nfs/flexfilelayout/nfs_layout_flexfiles.mod.o
GZIP     arch/riscv/boot/Image.gz
LD [M]   fs/nfs/flexfilelayout/nfs_layout_flexfiles.ko
Kernel: arch/riscv/boot/Image.gz is ready
make[1]: Leaving directory '/home/oslab/lab0/build/linux'
make: Leaving directory '/home/oslab/lab0/linux'
root@8ab9e30c1532:/home/oslab/lab0#
```

3.3 使用 QEMU 运行内核 (25%)

请参考【[附录C.QEMU使用基础](#)】了解相关背景知识。

注意，QEMU的退出方式较为特殊，需要先按住 `ctrl+a`，放开后再按一次 `x`。

```
$ cd /home/oslab/lab0

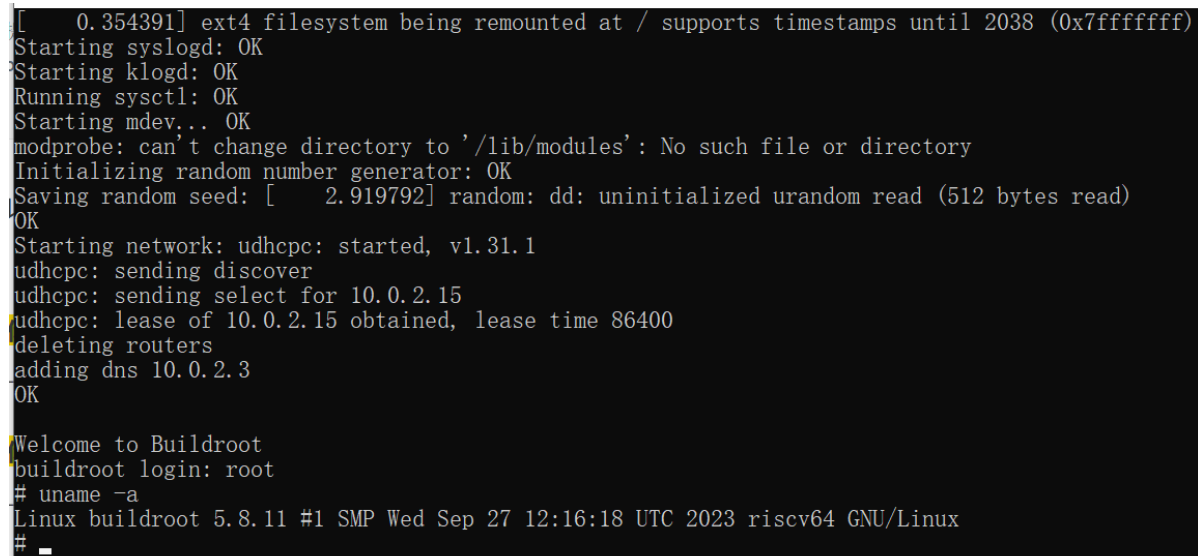
# 如果不在上面目录下执行的话，请手动修改 -kernel 和 file=.... 的文件路径，这里使用的是相对路径
$ qemu-system-riscv64 \
    -nographic \
    -machine virt \
    -kernel build/linux/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 \
    -append "root=/dev/vda ro console=ttyS0" \
    -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
    -netdev user,id=net0 -device virtio-net-device,netdev=net0

# 执行成功会提示登录，默认用户名为 root，密码为空，这里输入 root 进入即可
# Welcome to Buildroot
# buildroot login:
```

登录成功后，你可以在这个模拟运行的内核系统里到处看看。使用 `uname -a` 指令来确定你运行的系统是 riscv64 架构。

请在此处添加一张你成功登录后的截图：

答：



```
[ 0.354391] ext4 filesystem being remounted at / supports timestamps until 2038 (0x7fffffff)
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting mdev... OK
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator: OK
Saving random seed: [ 2.919792] random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: udhcpd: started, v1.31.1
udhcpd: sending discover
udhcpd: sending select for 10.0.2.15
udhcpd: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
# uname -a
Linux buildroot 5.8.11 #1 SMP Wed Sep 27 12:16:18 UTC 2023 riscv64 GNU/Linux
#
```

请在此处添加一张你运行 `**uname -a**` 指令后的结果截图：**

答：如上图。

3.4 使用 GDB 调试内核 (25%)

请参考【[附录D.GDB使用基础](#)】了解相关背景知识。学会调试将在后续实验中为你提供帮助，推荐同学们跟随[GDB调试入门指南](#)教程完成相应基础练习，熟悉 GDB 调试的使用。

首先请你退出上一步使用 QEMU 运行的内核，并重新使用 QEMU 按照下述参数模拟运行内核（**不是指在上一步运行好的 QEMU 运行的内核中再次运行下述命令！**）。


```
$ 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
```

-S: 表示启动时暂停执行，这样我们可以在 GDB 连接后再开始执行
 # -s: -gdb tcp::1234 的缩写，会开启一个 tcp 服务，端口为 1234，可以使用 GDB 连接并进行调试

上述命令由于 -s 的原因，执行后会直接停止，表现为没有任何反应。接下来再打开一个终端，进入同一个 Docker 容器，并切换到 lab0 目录，使用 GDB 进行调试。

```
# 进入同一个 Docker 容器
$ docker exec -it oslab /bin/bash

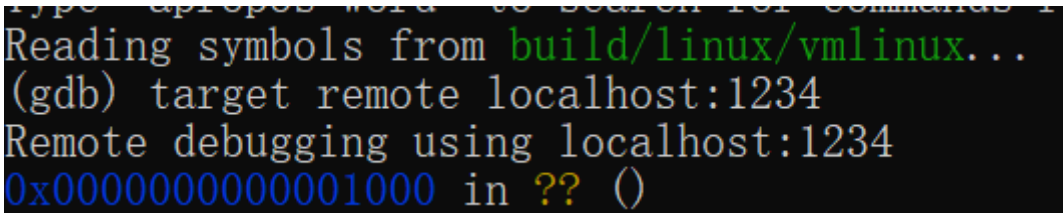
# 切换到 lab0 目录
$ cd /home/oslab/lab0/

# 使用 GDB 进行调试
$ riscv64-unknown-linux-gnu-gdb build/linux/vmlinux
```

顺序执行下列 GDB 命令，写出每条命令的含义并附上执行结果的截图。(可以全部执行后一起截图，不需要每个命令截一次图)

```
(gdb) target remote localhost:1234
```

- 含义：target remote 命令表示远程调试，而 1234 是上述 QEMU 执行时指定的用于调试连接的端口号。
- 执行结果：



```
Reading symbols from build/linux/vmlinux...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
```

```
(gdb) b start_kernel
(gdb) b *0x80000000
(gdb) b *0x80200000
(gdb) info breakpoints
(gdb) delete 2
(gdb) info breakpoints
```


- 含义：`b start_kernel` 命令表示在函数`start_kernel`处设置断点，`b *0x80000000` 命令表示在内存地址`0x80000000`处设置断点。`info breakpoints` 命令显示当前设置的断点信息。`delete 2` 命令删除编号为2的断点。
- 执行结果：

```
(gdb) b start_kernel
Breakpoint 1 at 0xfffffe000001714: file /home/oslab/lab0/linux/init/main.c, line 837.
(gdb) b *0x80000000
Breakpoint 2 at 0x80000000
(gdb) b *0x80200000
Breakpoint 3 at 0x80200000
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1         breakpoint       keep y   0xfffffe000001714 in start_kernel
                                     at /home/oslab/lab0/linux/init/main.c:837
2         breakpoint       keep y   0x0000000080000000
3         breakpoint       keep y   0x0000000080200000
(gdb) delete 2
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1         breakpoint       keep y   0xfffffe000001714 in start_kernel
                                     at /home/oslab/lab0/linux/init/main.c:837
3         breakpoint       keep y   0x0000000080200000
(gdb)
```

```
(gdb) continue
(gdb) delete 3
(gdb) continue
(gdb) step
(gdb) s
(gdb) (不做输入，直接回车)
(gdb) next
(gdb) n
(gdb) (不做输入，直接回车)
```

- 含义：`continue` 继续程序的执行直到遇到下一个断点或程序结束。`step` 和 `s` 命令 单步执行程序，进入函数内部。`next` 和 `n` 命令 单步执行程序，不进入函数内部，直接执行函数的下一行。不做输入，直接回车则直接执行上一条命令。
- 执行结果：

```
(gdb) continue
Continuing.

Breakpoint 3, 0x0000000080200000 in ?? ()
(gdb) delete 3
(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) step
set_task_stack_end_magic (tsk=<optimized out>) at /home/oslab/lab0/linux/kernel/fork.c:863
863      *stackend = STACK_END_MAGIC; /* for overflow detection */
(gdb) s
start_kernel () at /home/oslab/lab0/linux/init/main.c:838
838      smp_setup_processor_id();
(gdb)
smp_setup_processor_id () at /home/oslab/lab0/linux/arch/riscv/kernel/smp.c:38
38      cpuid_to_hartid_map(0) = boot_cpu_hartid;
(gdb) next
start_kernel () at /home/oslab/lab0/linux/init/main.c:841
841      cgroup_init_early();
(gdb) n
843      local_irq_disable();
(gdb)
844      early_boot_irqs_disabled = true;
```

```
(gdb) disassemble
(gdb) nexti
(gdb) n
(gdb) stepi
(gdb) s
```

- 含义：`disassemble` 命令显示当前位置的汇编代码。`nexti` 命令单步执行一个机器指令。`stepi` 命令单步执行一个机器指令，如果是函数调用，则进入函数内部执行一条指令。

- 执行结果：

```

    0x0000000000000000 early_boot_irqs_disabled = true;
(gdb) disassemble
Dump of assembler code for function start_kernel:
0xffffffff000001714 <+0>:      addi    sp,sp,-80
0xffffffff000001716 <+2>:      sd      ra,72(sp)
0xffffffff000001718 <+4>:      sd      s0,64(sp)
0xffffffff00000171a <+6>:      sd      s1,56(sp)
0xffffffff00000171c <+8>:      addi    s0,sp,80
0xffffffff00000171e <+10>:     sd      s2,48(sp)
0xffffffff000001720 <+12>:     sd      s3,40(sp)
0xffffffff000001722 <+14>:     sd      s4,32(sp)
0xffffffff000001724 <+16>:     sd      s5,24(sp)
0xffffffff000001726 <+18>:     sd      s6,16(sp)
0xffffffff000001728 <+20>:     auipc   a0,0x100a
0xffffffff00000172c <+24>:     addi    a0,a0,1560 # 0xffffffff00100bd40 <init_task>
0xffffffff000001730 <+28>:     auipc   ra,0x205
0xffffffff000001734 <+32>:     jalr    92(ra) # 0xffffffff00020678c <set_task_stack_end_magic>
0xffffffff000001738 <+36>:     jal    ra,0xffffffff000003730 <smp_setup_processor_id>
0xffffffff00000173c <+40>:     jal    ra,0xffffffff000008d4e <cgroup_init_early>
0xffffffff000001740 <+44>:     csrci   sstatus,2
=> 0xffffffff000001744 <+48>:     li      a5,1
0xffffffff000001746 <+50>:     auipc   a4,0x106f
0xffffffff00000174a <+54>:     sb      a5,-1786(a4) # 0xffffffff00107004c <early_boot_irqs_disabled>
0xffffffff00000174e <+58>:     jal    ra,0xffffffff000004606 <boot_cpu_init>
0xffffffff000001752 <+62>:     auipc   a1,0x9ff
0xffffffff000001756 <+66>:     addi    a1,a1,-1682 # 0xffffffff000a000c0 <linux_banner>
0xffffffff00000175a <+70>:     auipc   a0,0xb3d
0xffffffff00000175e <+74>:     addi    a0,a0,-850 # 0xffffffff000b3e408
0xffffffff000001762 <+78>:     auipc   ra,0x245
0xffffffff000001766 <+82>:     jalr    -510(ra) # 0xffffffff000246564 <printk>
0xffffffff00000176a <+86>:     addi    a0,s0,-72
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) nexti
0xffffffff000001746      844          early_boot_irqs_disabled = true;
(gdb) n
850          boot_cpu_init();
(gdb) stepi
boot_cpu_init () at /home/oslab/lab0/linux/arch/riscv/include/asm/current.h:31
31          return riscv_current_is_tp;
(gdb) s
2492          set_cpu_online(cpu, true);
(gdb)
```

- 请回答：`nexti` 和 `next` 的区别在哪里？`stepi` 和 `step` 的区别在哪里？`next` 和 `step` 的区别是什么？

答：`nexti`在汇编级别进行单步调试，执行当前机器指令，而`next`在源代码级别进行单步调试，执行当前行的C代码。`stepi`和`step`的区别同理。`next`和`step`的区别在于后者会进入函数内部执行，而前者不会。

```
(gdb) continue
# 这个地方会卡住，可以用 ctrl+c 强行中断
(gdb) quit
```

- 含义：`quit` 命令即退出gdb调试。

- 执行结果：

```
^C
Program received signal SIGINT, Interrupt.
arch_cpu_idle () at /home/oslab/lab0/linux/arch/riscv/kernel/process.c:33
33      local_irq_enable();
(gdb) quit
A debugging session is active.

      Inferior 1 [process 1] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/oslab/lab0/build/linux/vmlinux, process 1
Ending remote debugging.
[Inferior 1 (process 1) detached]
root@8ab9e30c1532:/home/oslab/lab0#
```

`vmlinux` 和 `Image` 的关系和区别是什么？为什么 QEMU 运行时使用的是 `Image` 而不是 `vmlinux`？

提示：一个可执行文件包括哪几部分？从 `vmlinux` 到 `Image` 发生了什么？

答：`vmlinux` 是 Linux 内核编译后生成的未经压缩的可执行内核镜像文件。它包含了完整的内核代码，符号表，调试信息以及其他调试用途的数据。这个文件通常很大，包含了所有的内核代码和符号信息，用于开发、调试和分析内核；而 `Image` 是经过压缩的 Linux 内核镜像文件。这个文件是可以直接在系统上运行的内核镜像，用于引导系统启动。在 Linux 内核编译过程中，`vmlinux` 会被压缩成 `Image` 文件，以便在启动时能够占用较少的空间。

在使用 QEMU 运行 Linux 内核时，通常需要使用经过压缩的内核镜像（即 `Image` 文件），因为 `Image` 是可以直接运行的内核镜像，而且相对于未压缩的 `vmlinux`，它占用的空间更小，方便传输和加载。

`Image` 包含的是经过压缩的内核，可以直接加载到虚拟机中运行。而 `vmlinux` 是未经压缩的，需要在调试器中进行调试，而不是直接运行在虚拟机中。

4 讨论和心得

在Ubuntu中安装完docker后，运行命令 `sudo systemctl start docker`，但却报错无法运行该命令。在查询之后才知道是因为没有以 `systemd` 作为 `init` 系统来引导（PID 1）。我的Ubuntu使用SysVinit 作为 `init` 系统，所以改用命令 `sudo service docker start`，正常启动docker服务。