

# 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：

```
PS C:\Users\admin> docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-world   latest    9c7a54a9a43c   4 months ago   13.3kB
PS C:\Users\admin> d:
PS D:\> cd .\Subjects\OS\
PS D:\Subjects\OS> ls

    目录：D:\Subjects\OS

Mode                LastWriteTime         Length Name
----                -
-a----             2023/9/25      10:29         28127 3210102037_徐铭_lab0.md
-a----             2023/9/23      16:16         42305 Docker_GDB_指令.pptx
-a----             2023/9/23      23:52       2963060224 oslab.tar
-a----             2023/9/25       8:47          1182 OSNotes.md

PS D:\Subjects\OS> docker import oslab.tar oslab:2023
sha256:885f4309c1ece8ce8dc18ed72662e5d313474949007cd8bb426b856ec42c1216
PS D:\Subjects\OS> docker iamge ls
docker: 'iamge' is not a docker command.
See 'docker --help'
PS D:\Subjects\OS> docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
oslab         2023      885f4309c1ec   24 seconds ago  2.89GB
hello-world   latest    9c7a54a9a43c   4 months ago   13.3kB
PS D:\Subjects\OS>
```

### 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 volumn 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。

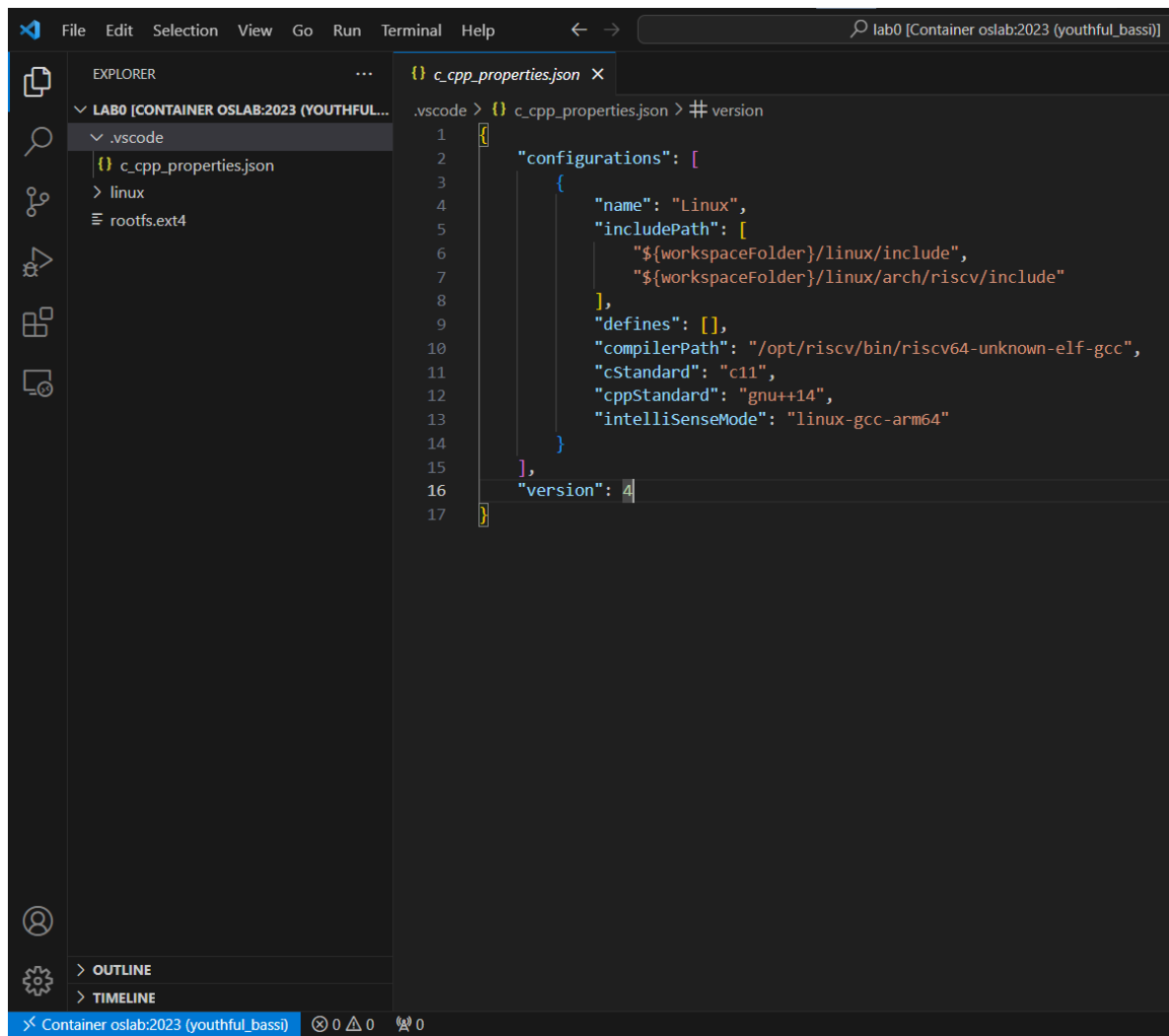
指令仅做参考，注意修改指令中的路径为你自己设置的路径。**如果你使用的是 Windows 系统，建议\*\*不要将本地新建的目录放在 C 盘等位置。避免后续指令权限问题。本地目录和映射的目录路径不需要相同。\*\***

```
# 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：这里我是用vscode智能提示直接连接到docker容器内部进行操作的，截图如下



注意这里，Container oslab:2023(youthful\_bassi)表示这是在容器内部

后来我在新的容器中试了一下挂载目录：

```
PS D:\Subjects\OS> docker run -it -v /d/Subjects/OS/os_experiment:/home/oslab/os_experiment --name os_lab oslab:2023 /bin/bash
root@58414b7bd670:/# cd ~
root@58414b7bd670:/# ls
root@58414b7bd670:/# cd ..
root@58414b7bd670:/# ls
bin boot dev etc home include lib lib64 libexec media mnt opt proc root run sbin share srv sys tmp usr var
root@58414b7bd670:/# cd /home/oslab
root@58414b7bd670:/home/oslab# ls
lab0
```

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：

- **docker run**：创建一个新的容器并启动；
- **-i**：input，以交互模式运行容器，也就是保持标准输入始终打开；
- **-t**：pseudo-terminal，让docker绑定一个伪终端到标准输入上，通常与“-i”同时使用；
- **-v**：volume，本意是卷/目录的意思，这里表示创建一个容器的数据卷，并将宿主机的一个目录挂载到这个卷上；
- **/path/to/your/local/dir/os\_experiment**：表示宿主机要挂载的目录；
- **/home/oslab/os\_experiment**：容器中的数据卷；
- **oslab:2023**：用于创建容器的镜像；
- **/bin/bash**：用来与容器进行交互的将被绑定到标准输入上的伪终端；

### 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。

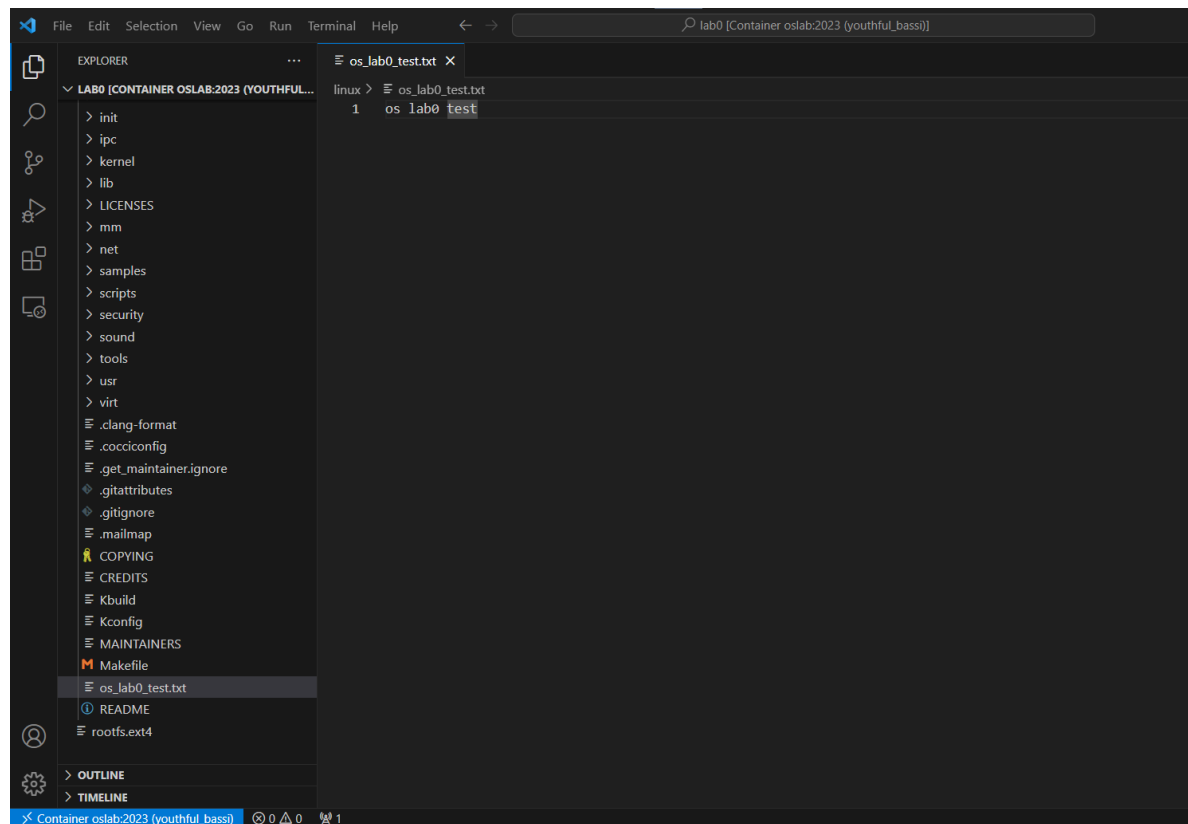
```
# 在 Docker 容器中确认是否挂载成功
root@dac72a2cc625:/home/oslab/os_experiment$ ls
testfile
# 退出docker，退出后容器将变为关闭状态，再次进入时需要重新启动容器（不是重新创建容器）
root@dac72a2cc625:/home/oslab/os_experiment$ exit
```

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：由于我使用的是VSCode连接，因此这里在VSCode中创建新文件，检查Docker容器中是否有相应改动：

在VSCode中创建新txt文件 `os_lab0_test`，并输入内容 `os lab0 test`



在bash中检查容器，发现有该文件存在，且内容一致，说明VSCode智能提示设置成功；

```
root@7ae1fd98f0e7:/home/oslab/lab0/linux/init# cd ..
root@7ae1fd98f0e7:/home/oslab/lab0/linux# ls
COPYING  Documentation  Kconfig  MAINTAINERS  README  block  crypto  fs  init  kernel  mm  os_lab0_test.txt  scripts  sound  usr
CREDITS  Kbuild        LICENSES  Makefile    arch   certs  drivers  include  ipc  lib  net  samples  security  tools  virt
root@7ae1fd98f0e7:/home/oslab/lab0/linux# cat os_lab0_test.txt
os lab0 testroot@7ae1fd98f0e7:/home/oslab/lab0/linux#
```

在新容器中的目录挂载测试结果：

```
PS D:\Subjects\OS\os_experiment> echo "testfile" > testfile
PS D:\Subjects\OS\os_experiment> ls
```

目录：D:\Subjects\OS\os\_experiment

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a----	2023/9/26 19:10	22	testfile

```
root@58414b7bd670:/home/oslab# cd os_experiment/
root@58414b7bd670:/home/oslab/os_experiment# ls
testfile
```

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
--------------	-------	---------	---------	--------	-------	-------

# 查看所有存在的容器

```
$ docker ps -a
```

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

# 启动处于停止状态的容器

```
$ docker start oslab
```

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					
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 /home/oslab/lab0 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，riscv64-unknown-linux-gnu-gcc 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 -g 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>-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>all</code>	指定本次编译的目标，目标是可以有多个的。这里的 <code>all</code> 并不表示编译所有目标，而是 makefile 中定义好的一个名称为 <code>all</code> 的编译目标。该目标代表 linux 内核。
<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。

image.png

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：编译过程的截图很长，这里就放一张最后编译完成的截图：

```
root@7aef1d98f0e7: /home/oslab/lab0# make
CC      drivers/gpu/drm/drm_hdcp.o
CC      drivers/gpu/drm/drm_radeon/r600_cs.o
CC      drivers/gpu/drm/drm_radeon/evergreen_cs.o
CC      drivers/gpu/drm/drm_managed.o
CC      drivers/gpu/drm/drm_gem_shmem_helper.o
CC      drivers/gpu/drm/drm_radeon/r100.o
CC      drivers/gpu/drm/drm_panel.o
CC      drivers/gpu/drm/drm_of.o
CC      drivers/gpu/drm/drm_pci.o
CC      drivers/gpu/drm/drm_debugfs.o
CC      drivers/gpu/drm/drm_debugfs_crc.o
CC      drivers/gpu/drm/drm_radeon/r300.o
CC      drivers/gpu/drm/drm_panel_orientation_quirks.o
CC      drivers/gpu/drm/drm_radeon/r420.o
CC      drivers/gpu/drm/drm_radeon/rs600.o
AR      drivers/gpu/drm/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
GZIP     arch/riscv/boot/Image.gz
CC [M]   fs/nfs/flexfilelayout/nfs_layout_flexfiles.mod.o
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@7aef1d98f0e7: /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：

```
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: [ 6.200766] 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
#
#
#
# |
```

请在此处添加一张你运行 `uname -a` 指令后的结果截图：

答：

```
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: [ 6.200766] 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 Mon Sep 25 14:21:46 UTC 2023 riscv64 GNU/Linux
#
```

### 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：

```
root@7ae1fd98f0e7: /home/os × root@7ae1fd98f0e7: /home/os × + v
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

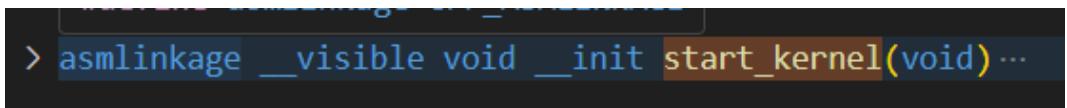
PS C:\Users\admin> docker exec -it youthful_bassi /bin/bash
root@7ae1fd98f0e7:/# cd /home/oslab/lab0
root@7ae1fd98f0e7:/home/oslab/lab0# riscv64-unknown-linux-gnu-gdb build/linux/vmlinux
GNU gdb (GDB) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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 "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 build/linux/vmlinux...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000000100 in ?? ()
(gdb)
```

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

- 含义:

- b start\_kernel: b是breakpoint的简写, 这里是设置断点的意思, 通过查看main.c源文件可知, 这里是将断点设置在函数start\_kernel处;



- b \*0x80000000: 在运行程序后地址为0x80000000处设置断点;
- b \*0x80200000: 在运行程序后地址为0x80200000处设置断点;
- info breakpoints: 查看当前的已设置的所有断点信息, 每个断点按设置先后顺序有一个标号;
- delete 2: 删除标号为2的断点;

- 执行结果:

```

PS C:\Users\admin> docker exec -it youthful_bassi /bin/bash
root@7ae1fd98f0e7:/# cd /home/oslab/lab0
root@7ae1fd98f0e7:/home/oslab/lab0# riscv64-unknown-linux-gnu-gdb build/linux/vmlinux
GNU gdb (GDB) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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 "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 build/linux/vmlinux...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) b start_kernel
Breakpoint 1 at 0xffffffe000001714: 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   0xffffffe000001714 in start_kernel at /home/oslab/lab0/linux/init/main.c:837
2        breakpoint      keep y   0x0000000000000000
3        breakpoint      keep y   0x0000000000000000
(gdb) delete 2
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint      keep y   0xffffffe000001714 in start_kernel at /home/oslab/lab0/linux/init/main.c:837
3        breakpoint      keep y   0x0000000000000000
(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：程序继续执行直到下一个断点
  - step：执行下一条指令，且如果是函数调用，进入函数内部；简写为s；
  - next：执行下一条指令，不进入函数内部；简写为n；
  - (不做输入，直接回车)：默认执行GDB上一条执行的命令；
- 执行结果：

```

(gdb) info breakpoints
Num    Type             Disp Enb Address              What
1      breakpoint      keep y   0xffffffff000001714 in start_kernel at /home/oslab/lab0/linux/init/main.c:837
3      breakpoint      keep y   0x0000000080200000
(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)

```

```

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

```

- 含义：
  - disassemble：反汇编一段代码，不带参数，默认的反汇编范围是所选择帧的pc附近的函数
  - nexti：汇编层面上的执行下一条机器码指令，不进入函数；
  - stepi：汇编层面上的执行下一条机器码指令，且如果遇到子函数会进入函数内部；
- 执行结果：

```

(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>:     csinci  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
0xffffffff00000176e <+90>:     auipc   s4,0x106f
0xffffffff000001772 <+94>:     addi    s4,s4,-1798 # 0xffffffff001070068 <initrd_end>
0xffffffff000001776 <+98>:     jal     ra,0xffffffff0000033b4 <setup_arch>
0xffffffff00000177a <+102>:    ld      s3,0(s4)
0xffffffff00000177e <+106>:    auipc   s2,0x106f
0xffffffff000001782 <+110>:    addi    s2,s2,-1806 # 0xffffffff001070070 <initrd_start>
0xffffffff000001786 <+114>:    beqz    s3,0xffffffff0000017d4 <start_kernel+192>
0xffffffff00000178a <+118>:    addi    s1,s3,-12
0xffffffff00000178e <+122>:    li      a2,12
0xffffffff000001790 <+124>:    auipc   a1,0xb3d
--Type <RET> for more, q to quit, c to continue without paging--nexti

```

```

--Type <RET> for more, q to quit, c to continue without paging--nexti
0xffffffff000001794 <+128>:    addi    a1,a1,-896 # 0xffffffff000b3e410
0xffffffff000001798 <+132>:    mv      a0,s1
0xffffffff00000179a <+134>:    auipc   ra,0x47f
0xffffffff00000179e <+138>:    jalr    2030(ra) # 0xffffffff000480f88 <memcmp>
0xffffffff0000017a2 <+142>:    bnez    a0,0xffffffff0000017d4 <start_kernel+192>
0xffffffff0000017a4 <+144>:    lw      a1,-20(s3)
0xffffffff0000017a8 <+148>:    ld      a2,0(s2)
0xffffffff0000017ac <+152>:    slli    a5,a1,0x20
0xffffffff0000017b0 <+156>:    srli    a5,a5,0x20
0xffffffff0000017b2 <+158>:    sub     s1,s1,a5
0xffffffff0000017b4 <+160>:    addi    s1,s1,-8
0xffffffff0000017b6 <+162>:    bgeu    s1,a2,0xffffffff0000017d0 <start_kernel+188>
0xffffffff0000017ba <+166>:    sub     a2,s3,a2
0xffffffff0000017be <+170>:    auipc   a0,0xb3d
0xffffffff0000017c2 <+174>:    addi    a0,a0,-926 # 0xffffffff000b3e420
0xffffffff0000017c6 <+178>:    auipc   ra,0x245
0xffffffff0000017ca <+182>:    jalr    -610(ra) # 0xffffffff000246564 <printk>
0xffffffff0000017ce <+186>:    j       0xffffffff0000017d4 <start_kernel+192>
0xffffffff0000017d0 <+188>:    sd      s1,0(s4)
0xffffffff0000017d4 <+192>:    auipc   a0,0x23
0xffffffff0000017d8 <+196>:    addi    a0,a0,-1996 # 0xffffffff00024008 <boot_command_line>
0xffffffff0000017dc <+200>:    auipc   ra,0x47f
0xffffffff0000017e0 <+204>:    jalr    1282(ra) # 0xffffffff000480cde <strlen>
0xffffffff0000017e4 <+208>:    addi    s1,a0,1
0xffffffff0000017e8 <+212>:    mv      a0,s1
0xffffffff0000017ea <+214>:    ld      s6,-72(s0)
0xffffffff0000017ee <+218>:    auipc   s4,0x106f
0xffffffff0000017f2 <+222>:    addi    s4,s4,-1966 # 0xffffffff001070040 <saved_command_line>
0xffffffff0000017f6 <+226>:    jal     ra,0xffffffff00000151e <memblock_alloc>
0xffffffff0000017fa <+230>:    sd      a0,0(s4)
0xffffffff0000017fe <+234>:    beqz    a0,0xffffffff000001816 <start_kernel+258>
0xffffffff000001800 <+236>:    mv      a0,s1
0xffffffff000001802 <+238>:    jal     ra,0xffffffff00000151e <memblock_alloc>
0xffffffff000001806 <+242>:    auipc   s5,0x106f
0xffffffff00000180a <+246>:    addi    s5,s5,-1998 # 0xffffffff001070038 <static_command_line>
0xffffffff00000180e <+250>:    sd      a0,0(s5)
0xffffffff000001812 <+254>:    mv      s3,a0
0xffffffff000001814 <+256>:    bnez    a0,0xffffffff000001830 <start_kernel+284>
0xffffffff000001816 <+258>:    mv      a2,s1
--Type <RET> for more, q to quit, c to continue without paging--n
0xffffffff000001818 <+260>:    auipc   a1,0x9ff

```



```

--Type <RET> for more, q to quit, c to continue without paging--stepi
0xffffffff0000018a8 <+404>: ld      a3,1884(a3) # 0xffffffff000e0000
0xffffffff0000018ac <+408>: mulw   a3,a5,a3
0xffffffff0000018b0 <+412>: ld      a1,0(s5)
0xffffffff0000018b4 <+416>: auipc   a7,0x0
0xffffffff0000018b8 <+420>: addi    a7,a7,-1418 # 0xffffffff00000132a <unknown_bootoption>
0xffffffff0000018bc <+424>: li      a6,0
0xffffffff0000018be <+426>: li      a5,-1
0xffffffff0000018c0 <+428>: li      a4,-1
0xffffffff0000018c2 <+430>: auipc   a0,0xb3d
0xffffffff0000018c6 <+434>: addi    a0,a0,-1098 # 0xffffffff000b3e478
0xffffffff0000018ca <+438>: auipc   ra,0x21e
0xffffffff0000018ce <+442>: jalr    84(ra) # 0xffffffff00021f91e <parse_args>
0xffffffff0000018d2 <+446>: mv      a1,a0
0xffffffff0000018d4 <+448>: beqz    a0,0xffffffff0000018fe <start_kernel+490>
0xffffffff0000018d6 <+450>: lui     a5,0xfffff
0xffffffff0000018d8 <+452>: bltu    a5,a0,0xffffffff0000018fe <start_kernel+490>
0xffffffff0000018dc <+456>: auipc   a7,0x0
0xffffffff0000018e0 <+460>: addi    a7,a7,-1568 # 0xffffffff0000012bc <set_init_arg>
0xffffffff0000018e4 <+464>: li      a6,0
0xffffffff0000018e6 <+466>: li      a5,-1
0xffffffff0000018e8 <+468>: li      a4,-1
0xffffffff0000018ea <+470>: li      a3,0
0xffffffff0000018ec <+472>: li      a2,0
0xffffffff0000018ee <+474>: auipc   a0,0xb3d
0xffffffff0000018f2 <+478>: addi    a0,a0,-1126 # 0xffffffff000b3e488
0xffffffff0000018f6 <+482>: auipc   ra,0x21e
0xffffffff0000018fa <+486>: jalr    40(ra) # 0xffffffff00021f91e <parse_args>
0xffffffff0000018fe <+490>: li      a0,0
0xffffffff000001900 <+492>: jal     ra,0xffffffff00000066f8 <setup_log_buf>
0xffffffff000001904 <+496>: jal     ra,0xffffffff000000e006 <vfs_caches_init_early>
0xffffffff000001908 <+500>: jal     ra,0xffffffff000000528c <sort_main_extable>
0xffffffff00000190c <+504>: auipc   ra,0x201
0xffffffff000001910 <+508>: jalr    -526(ra) # 0xffffffff0002026fe <trap_init>
0xffffffff000001914 <+512>: auipc   ra,0x2e3
0xffffffff000001918 <+516>: jalr    -1026(ra) # 0xffffffff0002e4512 <init_debug_pagealloc>
0xffffffff00000191c <+520>: auipc   a5,0x106f
0xffffffff000001920 <+524>: lw      a5,156(a5) # 0xffffffff0010709b8 <init_on_alloc>
0xffffffff000001924 <+528>: auipc   a2,0xb7f
0xffffffff000001928 <+532>: addi    a2,a2,1412 # 0xffffffff000b80ea8
--Type <RET> for more, q to quit, c to continue without paging--s

```

- 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/stepi 均表示汇编级别的断点定位，执行下一条汇编指令，其中 nexti 不进入函数内部直接跳过，stepi 进入函数内部；

next/step 均表示C语言级别的断点定位，执行下一条C语言指令，其中 next 不进入函数内部，step 进入函数内部；

(gdb) continue

# 这个地方会卡住，可以用 ctrl+c 强行中断

(gdb) quit

- 含义：

- quit: 退出GDB调试

- 执行结果：

```

0xffffffff000001b52 <+1086>: ret
End of assembler dump.
(gdb) continue
Continuing.
^C
Program received signal SIGINT, Interrupt.
do_strncpy_from_user (max=4056, count=<optimized out>, src=<optimized out>, dst=<optimized out>) at /home/oslab/lab0/linux/lib/strncpy_from_user.c:43
43      if (has_zero(c, &data, &constants)) {
(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@7ae1fd98f0e7:/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：

- 关系和区别

关系：

`vmlinux` 和 `Image` 都与 Linux 内核有关，都是用户对 Linux 内核代码进行编译后生成的；一般来说，它们都包含内核的二进制代码，而且它们都还没有被压缩；

区别：

`vmlinux` 实质上是 ELF 文件，它不仅包含完整的二进制内核代码，而且包含这些可执行文件的一些调试信息和符号表等；

`Image` 只含有内核的二进制代码，不包含符号信息和调试信息；

因此，`vmlinux` 常用于开发和调试，而 `Image` 用于引导和运行 Linux 系统；

- QEMU 使用 `Image` 而不是 `vmlinux`
  1. QEMU 通常在虚拟化环境中用于运行虚拟机，而不是直接在物理硬件上引导操作系统。因此，它需要一个包含已编译内核的二进制映像，而不是包含完整内核源代码和调试信息的 `vmlinux`；
  2. `vmlinux` 文件通常非常大，因为它包含了完整的内核源代码和调试信息。启动一个虚拟机时，加载这个巨大的文件会花费很多时间，而 `Image` 文件通常只包含已编译的内核二进制代码，因此启动速度更快；
  3. 虚拟机通常运行在受限资源的环境中，例如云服务器或嵌入式设备。使用 `Image` 文件可以节省磁盘空间和内存，因为它更小；
  4. 在生产环境中，通常不希望将内核的完整源代码和调试信息暴露给运行在虚拟机中的应用程序或用户。使用 `Image` 文件可以减少潜在的安全风险；

## 4 讨论和心得

---

实验过程中遇到的问题大部分集中在安装 docker 环境并创建容器那里，由于 docker 对计算机硬件的一些要求，我在安装 docker 时选择了只需要 Linux 环境的 docker engine，然而，缺乏图形化界面、缺乏硬件支持的问题难以解决；

经过多方查阅资料后我才发现，docker desktop in Windows 只需要电脑上安装好 WSL2 作为后端，并做好相应的配置即可正常使用，并不需要关心所谓的 Hyper-V、KVM 等硬件的虚拟化的支持，因此最后也就顺利解决了问题；

总体使用上而言，除了 docker 鲸吞了我大量 C 盘空间外，使用体验还行，命令的风格类似 Linux，后面作好容器目录的挂载后实现了文件夹的文件同步，十分畅快。

## 附录

---

### A. Linux 使用基础

#### Linux 简介

Linux 是一套免费使用和自由传播的类 Unix 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。



在 Linux 环境下，人们通常使用命令行接口来完成与计算机的交互。终端（Terminal）是用于处理该过程的一个应用程序，通过终端你可以运行各种程序以及在自己的计算机上处理文件。在类 Unix 的操作系统上，终端可以为你完成一切你所需要的操作。我们仅对实验中涉及的一些概念进行介绍，你可以通过下面的链接来对命令行的使用进行学习：

1. [The Missing Semester of Your CS Education>>Video<<](#)
2. [GNU/Linux Command-Line Tools Summary](#)
3. [Basics of UNIX](#)
4. [TLCL\(billie66.github.io\)](#) (Shell 命令基础介绍)
5. [实验楼 Linux 学习课程](#) 免费的 Linux 模拟环境，仅供练习。

## Linux中的环境变量

环境变量一般是指在操作系统中用来指定操作系统运行环境的一些参数。常见的例如 PATH 变量，它用来指明系统中可执行文件的默认搜索路径。我们可以通过 `which` 命令来做一些小的实验：

```
$ which gcc
/usr/bin/gcc

$ ls -l /usr/bin/gcc
lrwxrwxrwx 1 root root 5 May 20 2019 /usr/bin/gcc -> gcc-7
```

`which` 命令将会输出一个可执行程序的实际路径。可以看到，当我们在输入 `gcc` 命令时，终端实际执行的程序是 `/usr/bin/gcc`。实际上，终端在执行命令时，会从 `PATH` 环境变量所包含的地址中查找对应的程序来执行。我们可以将 `PATH` 变量打印出来来检查一下其是否包含 `/usr/bin`。

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/phantom/.local/bin
```

因此，如果你想直接使用 `riscv64-unknown-linux-gnu-gcc`、`qemu-system-riscv64` 等程序，你只需要把他们所在的目录添加到 `PATH` 环境变量中即可，镜像中已帮助你完成了这步操作，可在相应容器中使用 `echo $PATH` 命令检查是否包含 `/opt/riscv/bin` 目录。

```
$ export PATH=$PATH:/opt/riscv/bin
```

进一步的，你可以检查该目录下是否含有 `riscv64` 等相关编译文件。

## B. Docker 使用基础

### Docker 基本介绍

在生产开发环境中，常常会遇到应用程序和系统环境变量以及一些依赖的库文件不匹配，导致应用无法正常运行的问题，因此出现了 Docker。Docker 是一种利用容器（container）来进行创建、部署和运行应用的工具。Docker 把一个应用程序运行需要的二进制文件、运行需要的库以及其他依赖文件打包为一个包（package），然后通过该包创建容器并运行，由此被打包的应用便成功运行在了 Docker 容器中。

比如，你在本地用 Python 开发网站后台，开发测试完成后，就可以将 Python3 及其依赖包、Flask 及其各种插件、Mysql、Nginx 等打包到一个容器中，然后部署到任意你想部署到的环境。

Docker中有三个重要概念：

1. 镜像 (Image) : 类似于虚拟机中的镜像, 是一个包含有文件系统的面向 Docker 引擎的只读模板。任何应用程序运行都需要环境, 而镜像就是用来提供这种运行环境的。
2. 容器 (Container) : Docker 引擎利用容器来运行、隔离各个应用。**容器是镜像创建的应用实例**, 可以创建、启动、停止、删除容器, 各个容器之间是相互隔离的, 互不影响。镜像本身是只读的, 容器从镜像启动时, Docker 在镜像的上层创建一个可写层, 镜像本身不变。
3. 仓库 (Repository) : 镜像仓库是 Docker 用来集中存放镜像文件的地方, 一般每个仓库存放一类镜像, 每个镜像利用 tag 进行区分, 比如 Ubuntu 仓库存放有多个版本 (12.04、14.04等) 的 Ubuntu 镜像。

## Docker基本命令

实验中只需要了解 Docker 中容器操作的相关命令及选项即可, 可以通过 [Docker 命令大全 | 菜鸟教程 \(runoob.com\)](#) 进行初步了解。

```
# 列出所有镜像
$ docker image ls

# 从镜像 oslab:2023 中创建容器, 使用交互式终端, 并在容器内执行 /bin/bash 命令
$ docker run -it oslab:2023 /bin/bash

# 启动容器时, 使用 -v 参数指定挂载宿主主机目录, 启动一个 centos 容器将宿主机的 /test 目录挂载到容器的 /soft 目录
$ docker run -it -v /test:/soft centos /bin/bash

# 进入已经运行的容器
$ docker attach oslab

# 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

# 停止容器
$ docker stop ID

# 启动处于停止状态的容器
$ docker start ID

# 重命名容器
$ docker rename oldname newname

# 列出所有的容器
$ docker ps -a

# 删除容器
$ docker rm ID
```

## C. QEMU 使用基础

### 什么是QEMU

QEMU 最开始是由法国程序员 Fabrice Bellard 等人开发的可执行硬件虚拟化的开源托管虚拟机。

QEMU 主要有两种运行模式。**用户模式下, QEMU 能够将一个平台上编译的二进制文件在另一个不同的平台上运行。**如一个 ARM 指令集的二进制程序, 通过 QEMU 的 TCG (Tiny Code Generator) 引擎的处理之后, ARM 指令被转化为 TCG 中间代码, 然后再转化为目标平台 (如 Intel x86) 的代码。

系统模式下，QEMU 能够模拟一个完整的计算机系统，该虚拟机有自己的虚拟 CPU、芯片组、虚拟内存以及各种虚拟外部设备。它使得为跨平台编写的程序进行测试及除错工作变得容易。

## 如何使用 QEMU（常见参数介绍）

以该命令为例，我们简单介绍 QEMU 的参数所代表的含义

```
$ qemu-system-riscv64 \
    -nographic \
    -machine virt \
    -kernel build/linux/arch/riscv/boot/Image \
    -append "root=/dev/vda ro console=ttyS0" \
    -device virtio-blk-device,drive=hd0 \
    -bios default -drive file=rootfs.ext4,format=raw,id=hd0 \
    -netdev user,id=net0 -device virtio-net-device,netdev=net0
-S \
-s \
```

- **-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
- **-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 作为文件系统
- **-netdev user,id=str**: 指定 user mode 的虚拟网卡, 指定 ID 为 str
- **-S**: 启动时暂停 CPU 执行(使用 c 启动执行)
- **-s**: -gdb tcp::1234 的简写
- **-bios default**: 使用默认的 OpenSBI firmware 作为 bootloader

更多参数信息可以参考官方文档 [System Emulation — QEMU documentation \(qemu-project.gitlab.io\)](https://www.qemu.org/docs/latest/system/emulation/)

## D. GDB 使用基础

### 什么是 GDB

GNU调试器（英语：GNU Debugger，缩写：gdb）是一个由 GNU 开源组织发布的、UNIX/LINUX 操作系统下的、基于命令行的、功能强大的程序调试工具。借助调试器，我们能够查看另一个程序在执行时实际在做什么（比如访问哪些内存、寄存器），在其他程序崩溃的时候可以比较快速地了解导致程序崩溃的原因。被调试的程序可以是和 gdb 在同一台机器上，也可以是不同机器上。

总的来说，gdb 可以有以下4个功能：

- 启动程序，并指定可能影响其行为的所有内容
- 使程序在指定条件下停止
- 检查程序停止时发生了什么
- 更改程序中的内容，以便纠正一个 bug 的影响

### GDB 基本命令介绍

在 gdb 命令提示符“(gdb)”下输入“help”可以查看所有内部命令及使用说明。

- (gdb) start: 单步执行, 运行程序, 停在第一执行语句
- (gdb) next: 单步调试 (逐过程, 函数直接执行), 简写n
- (gdb) run: 重新开始运行文件 (run-text: 加载文本文件, run-bin: 加载二进制文件), 简写r
- (gdb) backtrace: 查看函数的调用的栈帧和层级关系, 简写bt
- (gdb) break 设置断点。比如断在具体的函数就break func; 断在某一行break filename:num
- (gdb) finish: 结束当前函数, 返回到函数调用点
- (gdb) frame: 切换函数的栈帧, 简写f
- (gdb) print: 打印值及地址, 简写p
- (gdb) info: 查看函数内部局部变量的数值, 简写i; 查看寄存器的值i register xxx
- (gdb) display: 追踪查看具体变量值
- (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
- Ctrl + x, 再按 a: 退出layout

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。

## E. LINUX 内核编译基础

### 交叉编译

交叉编译指的是在一个平台上编译可以在另一个平台运行的程序, 例如在 x86 机器上编译可以在 arm 平台运行的程序, 交叉编译需要交叉编译工具链的支持。

### 内核配置

内核配置是用于配置是否启用内核的各项特性, 内核会提供一个名为 `defconfig` (即 Default Configuration)的默认配置, 该配置文件位于各个架构目录的 `configs` 文件夹下, 例如对于 RISC-V 而言, 其默认配置文件为 `arch/riscv/configs/defconfig`。使用 `make ARCH=riscv defconfig` 命令可以在内核根目录下生成一个名为 `.config` 的文件, 包含了内核完整的配置, 内核在编译时会根据 `.config` 进行编译。配置之间存在相互的依赖关系, 直接修改 `defconfig` 文件或者 `.config` 有时候并不能达到想要的效果。因此如果需要修改配置一般采用 `make ARCH=riscv menuconfig` 的方式对内核进行配置。

### 常见参数

**ARCH** 指定架构, 可选的值包括 `arch` 目录下的文件夹名, 如 `x86`, `arm`, `arm64` 等, 不同于 `arm` 和 `arm64`, 32 位和 64 位的 RISC-V 共用 `arch/riscv` 目录, 通过使用不同的 `config` 可以编译 32 位或 64 位的内核。

**CROSS\_COMPILE** 指定使用的交叉编译工具链, 例如指定 `CROSS_COMPILE=aarch64-linux-gnu-`, 则编译时会采用 `aarch64-linux-gnu-gcc` 作为编译器, 编译可以在 `arm64` 平台上运行的 `kernel`。

**CC** 指定编译器, 通常指定该变量是为了使用 `clang` 编译而不是用 `gcc` 编译, Linux 内核在逐步提供对 `clang` 编译的支持, `arm64` 和 `x86` 已经能够很好的使用 `clang` 进行编译。

### 常用编译选项

```
$ make defconfig # 使用当前平台的默认配置，在 x86 机器上会使用 x86 的默认配置
$ make -j$(nproc) # 编译当前平台的内核，-j$(nproc)为以机器硬件线程数进行多线程编译

$ make ARCH=riscv defconfig # 使用RISC-V平台的默认配置
$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j$(nproc) # 编译RISC-V平台内核

$ make clean # 清除所有编译好的object文件
$ make mrproper # 清除编译的配置文件，中间文件和结果文件

$ make init/main.o # 编译当前平台的单个object文件init/main.o（会同时编译依赖的文件）
```