

# lab0 GDB + QEMU 调试 64 位 RISC-V LINUX 实验报告

---

Name: 李明伟 ID:3190106234

## lab0 GDB + QEMU 调试 64 位 RISC-V LINUX 实验报告

1. 搭建基础实验环境
  - 1.1 下载并安装 VMWare Workstation 16.1.2
  - 1.2 下载Ubuntu镜像
  - 1.3 安装Ubuntu虚拟机
  - 1.4 搭建docker环境
  - 1.5 下载并导入实验所需docker环境oslab.tar
  - 1.6 在docker中创建容器
    - 1.6.1 下载git, 并将实验需要用到的容器环境下载到虚拟机
    - 1.6.2 下载Linux源码
    - 1.6.3 从镜像中创建一个新容器
    - 1.6.4 将Linux源码拷贝到docker容器中
  - 1.7 编译Linux内核
2. 利用QEMU+GDB对Linux内核进行调试
  - 2.1 使用QEMU运行内核
  - 2.2 打开GDB调试
  - 2.3 调试细节
    - 2.3.1 backtrace
    - 2.3.2 layout
    - 2.3.3 finish
    - 2.3.4 info
    - 2.3.5 frame
    - 2.3.6 break
    - 2.3.7 display
    - 2.3.8 next & step
3. 总结与感想
  - 3.1 思考题
    - 3.1.1 使用 `riscv64-unknown-elf-gcc` 编译单个 `.c` 文件
    - 3.1.2使用 `riscv64-unknown-elf-objdump` 反汇编 1 中得到的编译产物
    - 3.1.3 调试 Linux 时:
    - 3.1.4 使用 `make` 工具清除 Linux 的构建产物
    - 3.1.5 `vmlinux` 和 `Image` 的关系和区别是什么?
  - 3.2 一些心得体会

## 1. 搭建基础实验环境

---

由于本实验最好在Linux系统下进行, 因而选择利用VMware Workstation 16.1.2来进行虚拟机的搭建。

- 实验环境: Windows 10 20H2
- 处理器: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz (**x86架构**)
- 所用软件: VMware Workstation 16.1.2 for Windows 64-bit
- 虚拟机版本: Ubuntu-18.04.5-desktop-amd64.iso

## 1.1 下载并安装 VMWare Workstation 16.1.2

主页 / VMware Workstation 16.1.2 Player

### 下载产品

选择版本 16.1.2

文档 [发行说明](#)

发布日期 2021-05-18

类型 产品的二进制文件

产品资源

- [查看我的下载历史记录](#)
- [产品信息](#)
- [文档](#)
- [Knowledge Base](#)
- [社区](#)
- [Workstation Player 升级](#)

[产品下载](#) [驱动程序和工具](#) [开源](#) [自定义 ISO](#) [OEM 附加模块](#)

文件	信息
<b>VMware Workstation 16.1.2 Player for Windows 64-bit Operating Systems</b>	<a href="#">立即下载</a>
文件大小: 215.30 MB 文件类型: exe <a href="#">了解更多信息</a>	
<b>VMware Workstation 16.1.2 Player for Linux 64-bit</b>	<a href="#">立即下载</a>
文件大小: 167.19 MB 文件类型: bundle <a href="#">了解更多信息</a>	



## 1.2 下载Ubuntu镜像

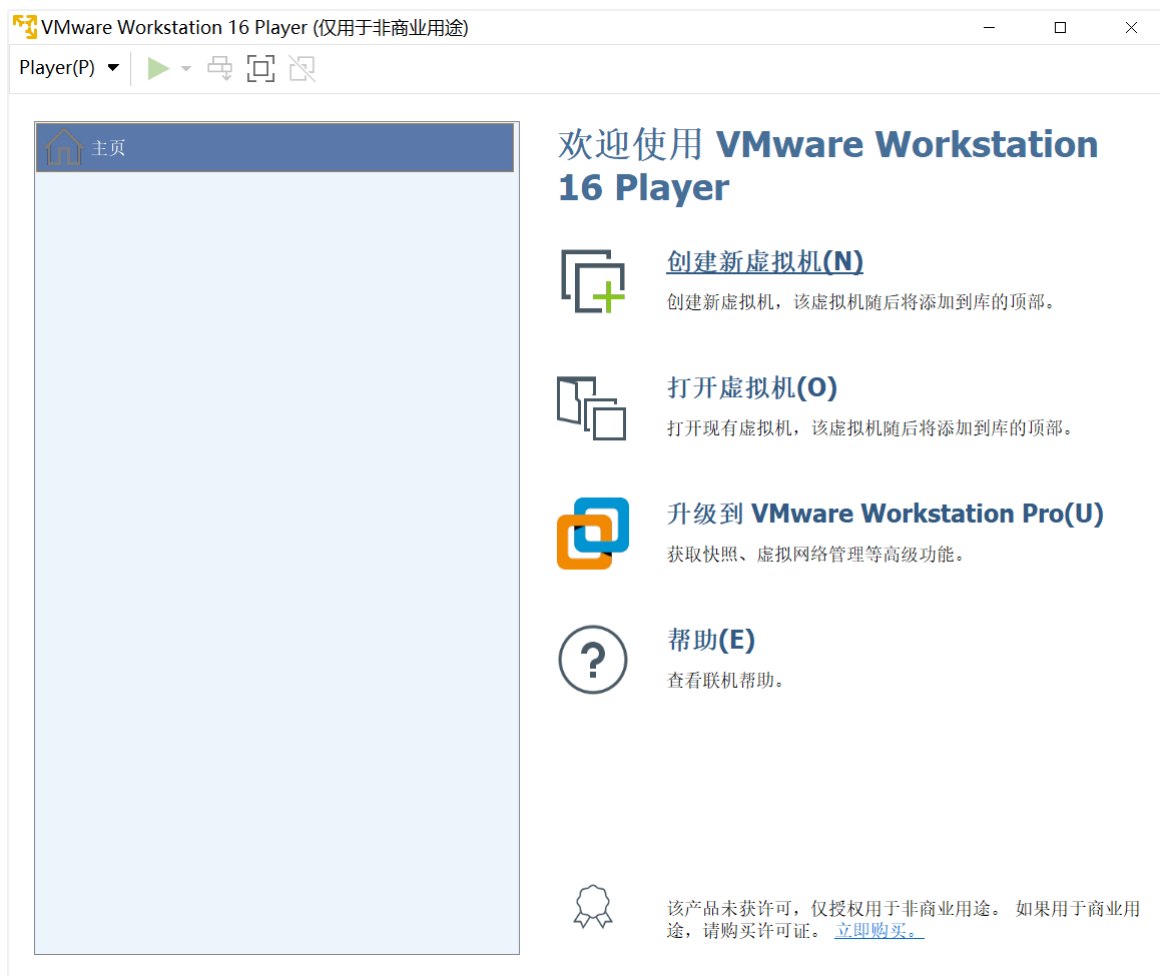
这里选择官网的 `ubuntu-18.04.5-desktop-amd64.iso` 版本进行下载

# Index of /ubuntu-releases/18.04/

<a href="#">../</a>		
<a href="#">FOOTER.html</a>	13-Aug-2020 23:02	810
<a href="#">HEADER.html</a>	13-Aug-2020 23:02	4006
<a href="#">MD5SUMS-metalink</a>	12-Feb-2020 21:42	296
<a href="#">MD5SUMS-metalink.gpg</a>	12-Feb-2020 21:42	916
<a href="#">SHA256SUMS</a>	13-Aug-2020 23:39	202
<a href="#">SHA256SUMS.gpg</a>	17-Aug-2020 20:28	833
<a href="#">ubuntu-18.04.5-desktop-amd64.iso</a>	07-Aug-2020 06:59	2G
<a href="#">ubuntu-18.04.5-desktop-amd64.iso.torrent</a>	13-Aug-2020 23:02	164K
<a href="#">ubuntu-18.04.5-desktop-amd64.iso.zsync</a>	13-Aug-2020 23:02	4M
<a href="#">ubuntu-18.04.5-desktop-amd64.list</a>	07-Aug-2020 06:59	8054
<a href="#">ubuntu-18.04.5-desktop-amd64.manifest</a>	07-Aug-2020 06:56	59K
<a href="#">ubuntu-18.04.5-live-server-amd64.iso</a>	07-Aug-2020 07:05	945M
<a href="#">ubuntu-18.04.5-live-server-amd64.iso.torrent</a>	13-Aug-2020 23:00	74K
<a href="#">ubuntu-18.04.5-live-server-amd64.iso.zsync</a>	13-Aug-2020 23:00	2M
<a href="#">ubuntu-18.04.5-live-server-amd64.list</a>	07-Aug-2020 07:05	10K
<a href="#">ubuntu-18.04.5-live-server-amd64.manifest</a>	07-Aug-2020 07:02	14K

## 1.3 安装Ubuntu虚拟机

- 打开VMware Workstation 16 Player，创建新虚拟机。



- 将先前下载好的Ubuntu-18.04.5-desktop-amd64.iso镜像搭载在虚拟机中

新建虚拟机向导

×

欢迎使用新建虚拟机向导

虚拟机如同物理机，需要操作系统。您将如何安装客户机操作系统？

安装来源:

☐ 安装程序光盘(D):

无可用驱动器

☒ 安装程序光盘映像文件(ISO)(M):

D:\下载\Documents\ubuntu-18.04.5-desktop-amd64.iso

浏览(R)...

 已检测到 Ubuntu 64 位 18.04.5。

该操作系统将使用简易安装。[\(这是什么?\)](#)

☐ 稍后安装操作系统(S)。

创建的虚拟机将包含一个空白硬盘。

帮助

< 上一步(B)

下一步(N) >

取消

- 为虚拟机分配合适的硬盘容量，为配合课程要求，这里分配35.0GB的磁盘空间以及4.0GB的内存

新建虚拟机向导

×

指定磁盘容量

磁盘大小为多少？

虚拟机的硬盘作为一个或多个文件存储在主机的物理磁盘中。这些文件最初很小，随着您向虚拟机中添加应用程序、文件和数据而逐渐变大。

最大磁盘大小 (GB)(S): 35.0

针对 Ubuntu 64 位 的建议大小: 20 GB

☐ 将虚拟磁盘存储为单个文件(Q)

☒ 将虚拟磁盘拆分成多个文件(M)

拆分磁盘后，可以更轻松地在计算机之间移动虚拟机，但可能会降低大容量磁盘的性能。

帮助

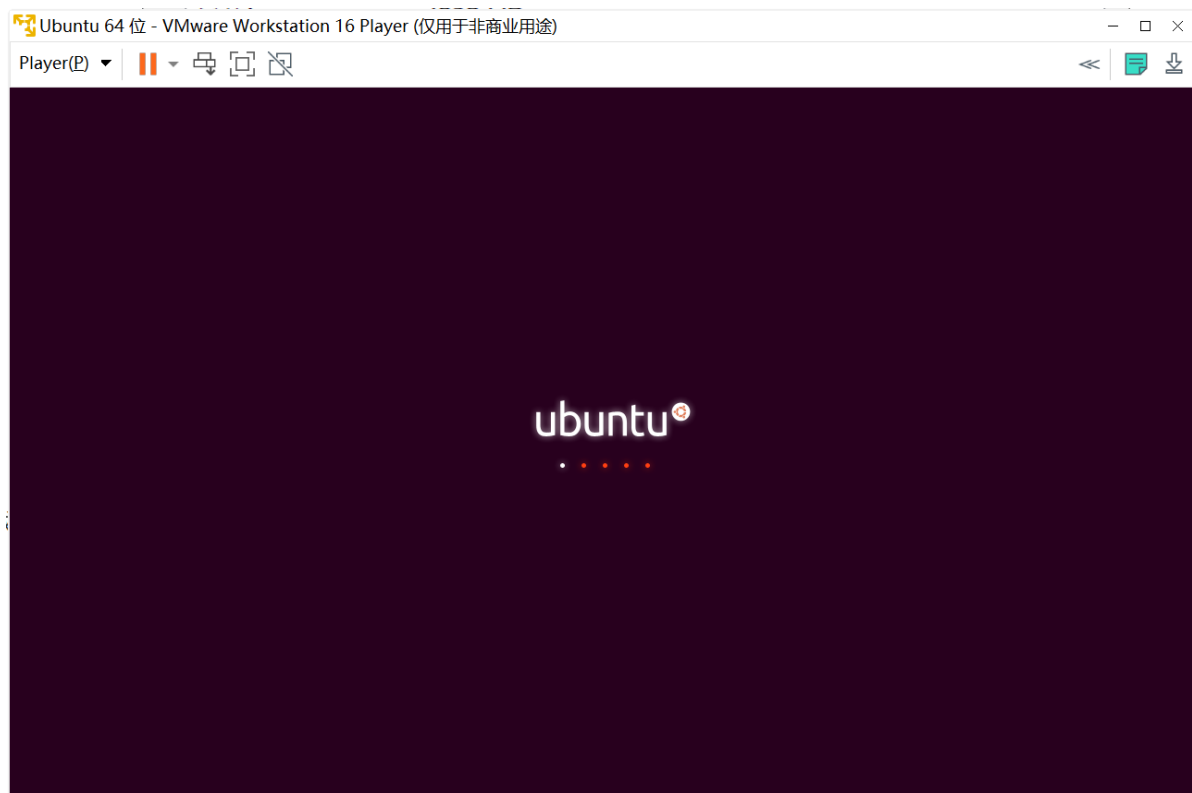
< 上一步(B)

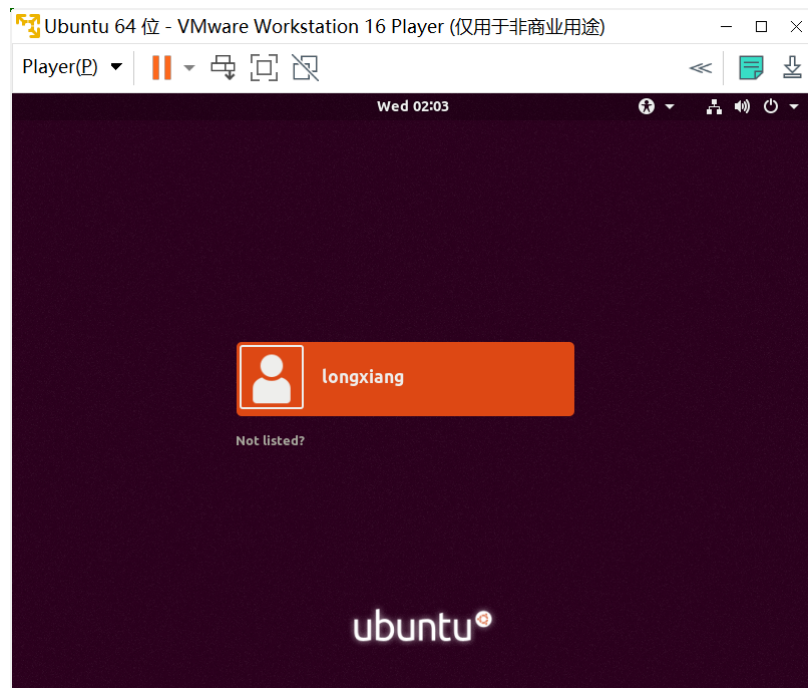
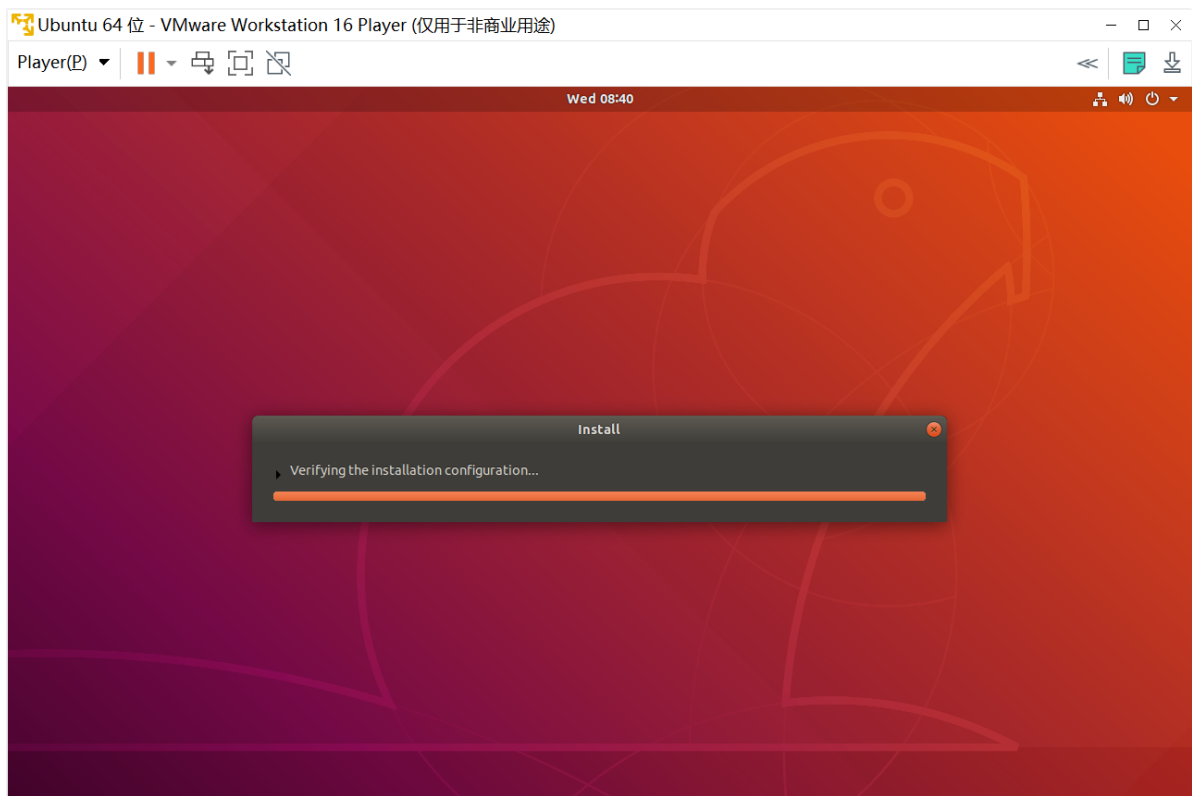
下一步(N) >

取消



- Ubuntu 成功运行





## 1.4 搭建docker环境

按照课程实验指导以及网络上的一些教程来对docker环境进行搭建。

首先对 `apt-get` 进行更新，在终端利用管理员权限运行来防止运行失败，接着安装必要的文件

```
$ sudo apt-get update
$ sudo apt-get install \
apt-transport-https \
curl \
gnupg \
lsb-release
```

```
limingwei@ubuntu:~$ sudo apt-get update
[sudo] password for limingwei:
Hit:1 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:2 http://us.archive.ubuntu.com/ubuntu bionic InRelease
Hit:3 http://us.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://us.archive.ubuntu.com/ubuntu bionic-backports InRelease
Reading package lists... Done
limingwei@ubuntu:~$ sudo apt-get install \
> apt-transport-https \
> ca-certificates \
> curl \
> gnupg \
> lsb-release
E: Could not get lock /var/lib/dpkg/lock-frontent - open (11: Resource temporarily unavailable)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontent), is another process using it?
```

这里发现提示 `could not get lock xxx open, unable to acquire the dpkg frontend lock`, 是系统提示权限不足, 因而用下面的两条语句来更改权限设置 (实际上是删除了 `lock`), 删除后则安装成功。

```
$ sudo rm /var/lib/dpkg/lock-frontent
$ sudo rm /var/lib/dpkg/lock
```

```
limingwei@ubuntu:~$ sudo rm /var/lib/dpkg/lock-frontent
limingwei@ubuntu:~$
limingwei@ubuntu:~$ sudo rm /var/lib/dpkg/lock
```

```
Setting up gpg-wks-client (2.2.4-1ubuntu1.4) ...
Setting up gnupg (2.2.4-1ubuntu1.4) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for install-info (6.5.0.dfsg.1-2) ...
Processing triggers for libc-bin (2.27-3ubuntu1.2) ...
Processing triggers for ca-certificates (20210119~18.04.1) .
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
```

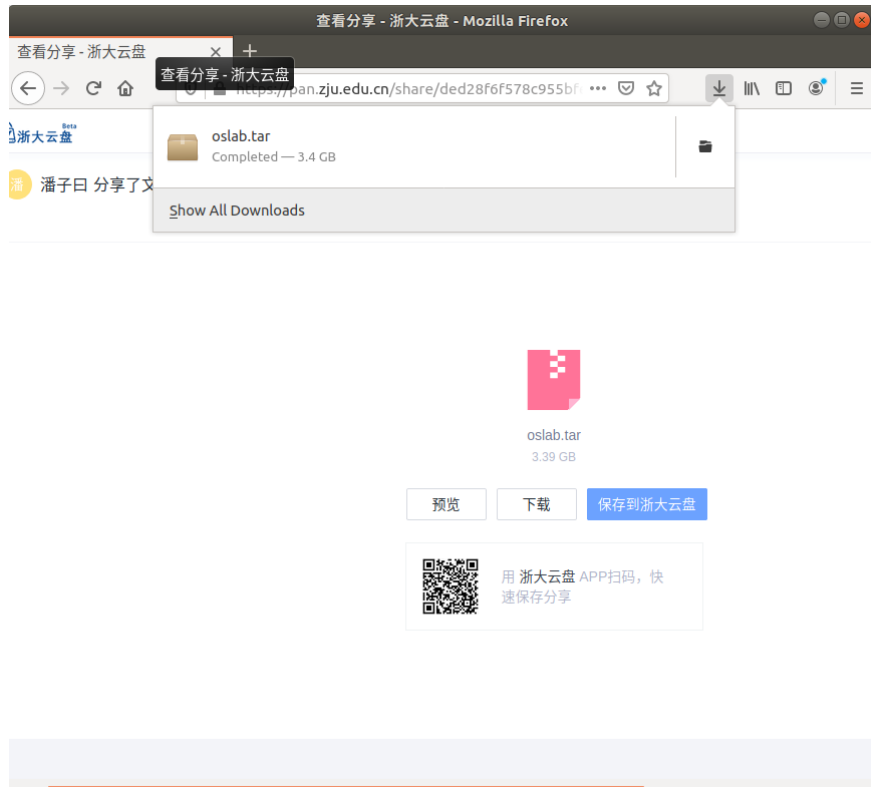
最后输入下面的命令来安装docker:

```
$ sudo snap install docker
```

```
limingwei@ubuntu:~$ sudo snap install docker
[sudo] password for limingwei:
docker 20.10.8 from Canonical✓ installed
limingwei@ubuntu:~$
```

## 1.5 下载并导入实验所需docker环境oslab.tar

- 从浙大云盘下载oslab.tar文件到虚拟机。



- 试图将oslab.tar导入docker

导入过程中即使使用了 `sudo` 仍存在 `Got permission denied` 的情况，证明所需权限不足，进行授权操作。其中进行了以下几种尝试。

```
$ sudo gpasswd -a $USER docker ### 试图将本用户（limingwei）加入到group docker之中，但似乎效果仍不佳
$ sudo chmod 777 oslab.tar ### 试图用chmod来对oslab.tar进行权限授予，但仍然失败
$ sudo chmod a+rw /var/run/docker.sock ### 修改docker.sock的权限为a+rw后成功
### 将oslab添加到docker镜像
$ sudo cat oslab.tar | docker import - oslab:2021
```

```
limingwei@ubuntu:~$ sudo cat oslab.tar | docker import - oslab:2021
cat: oslab.tar: No such file or directory
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/images/create?fromSrc=-&message=&repo=oslab%3A2021&tag=": dial unix /var/run/docker.sock: connect: permission denied
limingwei@ubuntu:~$ sudo groupadd docker
limingwei@ubuntu:~$ sudo gpasswd -a $limingwei docker
gpasswd: user 'docker' does not exist
limingwei@ubuntu:~$ sudo groupadd docker
groupadd: group 'docker' already exists
limingwei@ubuntu:~$ sudo gpasswd -a $limingwei docker
gpasswd: user 'docker' does not exist
limingwei@ubuntu:~$ sudo gpasswd -a $USER docker
Adding user limingwei to group docker
limingwei@ubuntu:~$
```



```
limingwei@ubuntu:~/Downloads$ sudo chmod 777 oslab.tar
limingwei@ubuntu:~/Downloads$ sudo cat oslab.tar | docker import - oslab:2021
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/images/create?fromSrc=-&message=&repo=oslab%3A2021&tag=": dial unix /var/run/docker.sock: connect: permission denied
limingwei@ubuntu:~/Downloads$
```

```
limingwei@ubuntu:~/Downloads$ sudo chmod a+rw /var/run/docker.sock
limingwei@ubuntu:~/Downloads$ sudo cat oslab.tar | docker import - oslab:2021
sha256:533150e2699e9a899cf8fcb71d1b55dbb96bb256966c769c83d4f198d6cfddf5
```

\$ docker images ### 导入docker镜像后查看是否导入成功

```
limingwei@ubuntu:~/Downloads$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
oslab          2021      533150e2699e   40 seconds ago 3.62GB
```

## 1.6 在docker中创建容器

### 1.6.1 下载git，并将实验需要用到的容器环境下载到虚拟机

```
$ sudo apt install git
$ git clone https://gitee.com/zjusec/os21fall
```

```
limingwei@ubuntu:~/Downloads$ sudo apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man liberror-perl
Suggested packages:
```

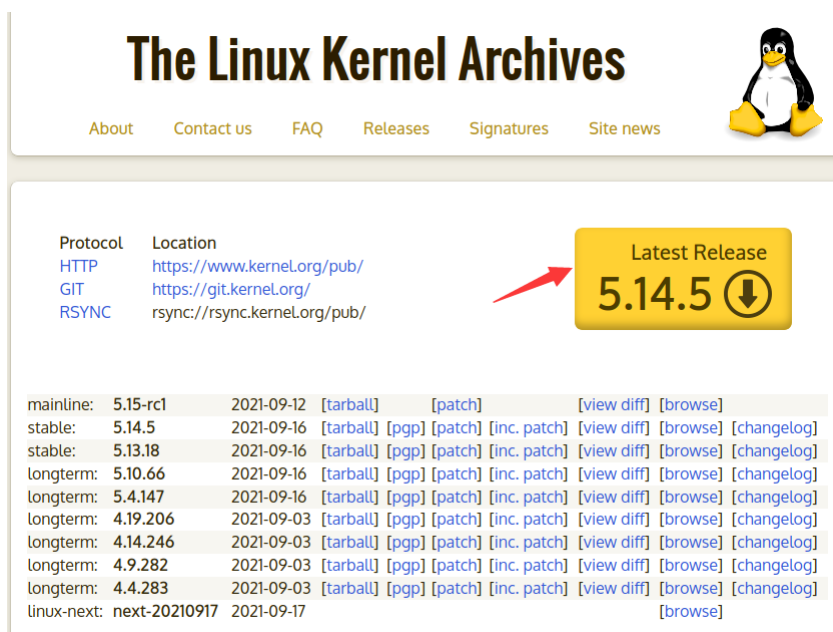
```
limingwei@ubuntu:~$ git clone https://gitee.com/zjusec/os21fall
Cloning into 'os21fall'...
remote: Enumerating objects: 99, done.
remote: Counting objects: 100% (99/99), done.
remote: Compressing objects: 100% (80/80), done.
remote: Total 172 (delta 43), reused 0 (delta 0), pack-reused 73
Receiving objects: 100% (172/172), 1.56 MiB | 371.00 KiB/s, done.
Resolving deltas: 100% (50/50), done.
```

可以使用ls指令看到后面需要使用的rootfs.img文件在/os21fall/src/lab0目录下

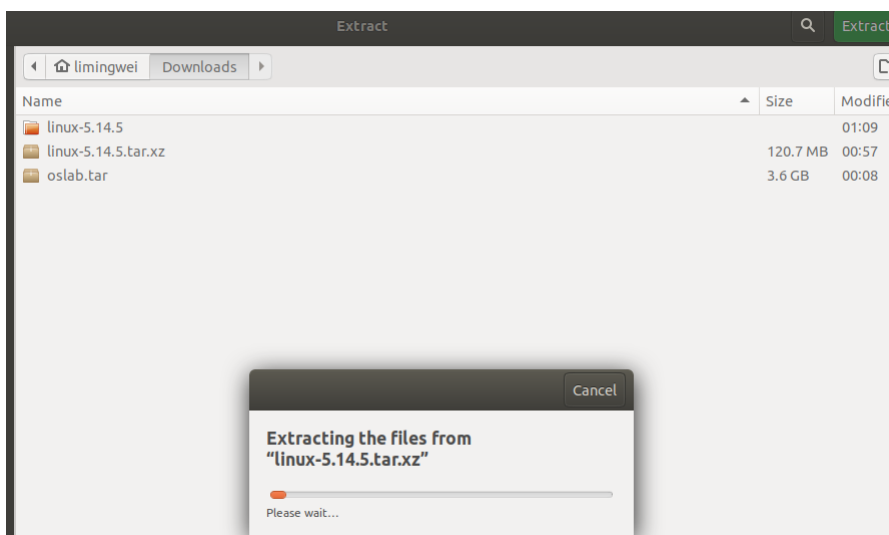
```
limingwei@ubuntu:~$ cd os21fall
limingwei@ubuntu:~/os21fall$ ls os21fall
ls: cannot access 'os21fall': No such file or directory
limingwei@ubuntu:~/os21fall$ ls
docs  mkdocs.yml  README.md  src
limingwei@ubuntu:~/os21fall$ cd src
limingwei@ubuntu:~/os21fall/src$ ls
lab0
limingwei@ubuntu:~/os21fall/src$ cd lab0
limingwei@ubuntu:~/os21fall/src/lab0$ ls
rootfs.img
```

## 1.6.2 下载Linux源码

由于我们接下来需要在容器中对Linux进行编译，所以这里先在网站<https://www.kernel.org/pub/>上下载最新版本的Linux源码，这里选择的是5.14.5版本的Linux



解压下载好的linux源码，这里解压到的目录是 /Downloads/linux-5.14.5



## 1.6.3 从镜像中创建一个新容器

```
$ docker run --name oslab -it oslab:2021 bash ### -it 指用交互式操作，在终端进行，最后在bash中进行交互
```

```
limingwei@ubuntu:~$ docker run --name oslab -it oslab:2021 bash
```

- 开启容器，以及一些docker的基本操作

```
$ docker start oslab    ### 开启oslab容器
$ docker ps            ### 查看docker目前各容器的状态，可以看到oslab容器成功开启了
$ docker exec -it oslab bash ### 以终端的方式执行容器
### 容器中进行的语句用 '#' 开头以作区分，注释采用仍 '###' 作为标识
# exit                ### 在容器中输入exit来退出
```

```

limingwei@ubuntu:~$ docker start oslab
oslab
limingwei@ubuntu:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
943b3fad0a4c   oslab:2021 "bash"    About a minute ago   Up 13 seconds   oslab

limingwei@ubuntu:~$ docker exec -it oslab bash
root@943b3fad0a4c:/# exit
exit

```

### 1.6.4 将Linux源码拷贝到docker容器中

**注意：**这里本人犯了一个错误，即先前忘记将Linux源码copy到容器中再进行编译，因而在编译的过程中会提示缺乏许多环境，无法顺利进行。而本实验提供的 `os21fa11` 文件夹中实际上已经包含了编译Linux源码需要的所有环境，因而，要将Linux源码先挪动到docker容器中才能顺利进行编译。

```

$ docker cp ~/Downloads/linux-5.14.5 oslab:/root/linux ### 这里的cp命令将前面目录中下载好的Linux源码copy到了名为oslab的容器中/root/linux文件夹内
### 注：上面的步骤其实也可以不必在docker外部重复进行，可以直接在先前创好的容器中下载Linux的源代码

```

```

limingwei@ubuntu:~$ docker cp ~/Downloads/linux-5.14.5 oslab:/root/linux

```

## 1.7 编译Linux内核

在进行了1.6.1~1.6.4的步骤之后，我们已经准备好了docker容器: `oslab`，以及编译Linux内核需要的环境(`os21fa11`)，并将Linux源码拷贝到容器中，接下来我们只需要使用make命令对Linux内核进行编译即可。

```

# cd /root/linux      ### 进入容器中先前拷贝好的linux内核的文件夹所在位置
# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig ### 这里选择生成riscv架构的配置文件
# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc) ### 使用-j选项进行编译

```

```

root@943b3fad0a4c:/# cd /root/linux
root@943b3fad0a4c:~/linux# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
root@943b3fad0a4c:~/linux# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc)
WRAP      arch/riscv/include/generated/uapi/asm/errno.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

```

在经过较长的一段时间等待后，编译成功完成：

```

CC [M]    fs/efivarfs/efivarfs.mod.o
GZIP      arch/riscv/boot/Image.gz
LD [M]    fs/efivarfs/efivarfs.ko
Kernel: arch/riscv/boot/Image.gz is ready
root@943b3fad0a4c:~/linux#

```

## 2. 利用QEMU+GDB对Linux内核进行调试



```
root@943b3fad0a4c: /
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
limingwei@ubuntu:~$ docker start oslab
oslab
limingwei@ubuntu:~$ docker exec -it oslab bash
root@943b3fad0a4c:/# riscv64-unknown-linux-gnu-gdb root/linux/vmlinux
GNU gdb (GDB) 10.1
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:
<https://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 root/linux/vmlinux...
(no debugging symbols found in root/linux/vmlinux)
(gdb)
```

注意到，由于先前使用QEMU时缺少参数，其不会在运行开始时自动暂停，因而这里需要在先前终端中对qemu的运行语句末端加入 `-s` 以确保启动时CPU自动暂停。同时，为GDB调试提供方便，需添加 `-s` 来将默认的1234端口作为调试端口(`-s` 为 `-gdb tcp::1234` 的缩写)。

```
# qemu-system-riscv64 -nographic -machine virt -kernel
/root/linux/arch/riscv/boot/Image -device virtio-blk-device,drive=hd0 -append
"root=/dev/vda ro console=ttyS0" -bios default -drive
file=linux/os21fall/src/lab0/rootfs.img,format=raw,id=hd0 -s -s
```

此时在GDB运行的终端中即可进行调试

## 2.3 调试细节

首先需要在GDB中连接先前设置的端口1234

```
(gdb) target remote localhost:1234 ### 此处的localhost也可以省略不写，如下
(gdb) target remote :1234
```

### 2.3.1 backtrace

`backtrace` 能够显示当前的函数运行栈，这里贴出几个使用的例子。

```
(gdb) backtrace
#0  0x0000000080000000 in ?? ()
(gdb) b *0x80200000
Breakpoint 2 at 0x80200000
(gdb) info b
Num      Type           Disp Enb Address            What
1        breakpoint     keep y   0x0000000080000000
          breakpoint already hit 1 time
2        breakpoint     keep y   0x0000000080200000
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000080200000 in ?? ()
(gdb) backtrace
#0  0x0000000080200000 in ?? ()
(gdb) █
```



注意到，这里使用backtrace的时候，由于断点设置在0x80000000以及0x80200000处，这里GDB并不能很好的定位出运行到哪条函数了，所以只现实了一层堆栈，且应表达的函数处显示了??字符

```
(gdb) b start_kernel
Breakpoint 3 at 0xffffffff8080066a
(gdb) continue
Continuing.

Breakpoint 3, 0xffffffff8080066a in start_kernel ()
(gdb) backtrace
#0  0xffffffff8080066a in start_kernel ()
#1  0xffffffff8000116a in _start_kernel ()
Backtrace stopped: frame did not save the PC
(gdb)
```

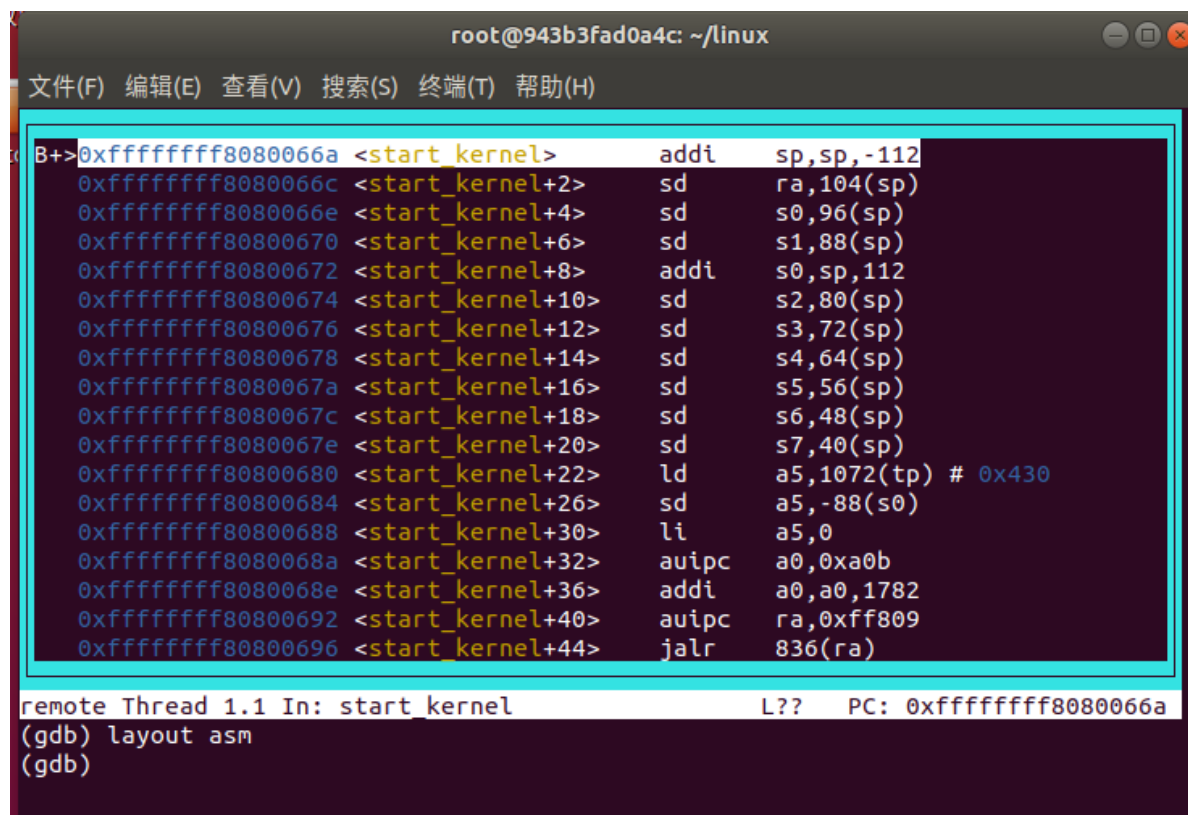
当我们程序运行断点位置设在start\_kernel时，便可以显示出如上图的调用关系

## 2.3.2 layout

layout 功能可以分割窗口，将显示区分为两个部分，可以加不同的参数来得到不同的效果：

```
(gdb) layout src          ### 显示源代码
(gdb) layout asm          ### 显示反汇编码
(gdb) layout regs         ### 显示CPU寄存器
(gdb) layout split        ### 显示源代码+反汇编
### ctrl+x 然后按a可以退出layout调试模式
```

下图是 layout asm 条件下的显示结果。



```
root@943b3fad0a4c: ~/linux
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

B+>0xffffffff8080066a <start_kernel>      addi    sp,sp,-112
0xffffffff8080066c <start_kernel+2>      sd      ra,104(sp)
0xffffffff8080066e <start_kernel+4>      sd      s0,96(sp)
0xffffffff80800670 <start_kernel+6>      sd      s1,88(sp)
0xffffffff80800672 <start_kernel+8>      addi    s0,sp,112
0xffffffff80800674 <start_kernel+10>     sd      s2,80(sp)
0xffffffff80800676 <start_kernel+12>     sd      s3,72(sp)
0xffffffff80800678 <start_kernel+14>     sd      s4,64(sp)
0xffffffff8080067a <start_kernel+16>     sd      s5,56(sp)
0xffffffff8080067c <start_kernel+18>     sd      s6,48(sp)
0xffffffff8080067e <start_kernel+20>     sd      s7,40(sp)
0xffffffff80800680 <start_kernel+22>     ld      a5,1072(tp) # 0x430
0xffffffff80800684 <start_kernel+26>     sd      a5,-88(s0)
0xffffffff80800688 <start_kernel+30>     li      a5,0
0xffffffff8080068a <start_kernel+32>     auipc   a0,0xa0b
0xffffffff8080068e <start_kernel+36>     addi    a0,a0,1782
0xffffffff80800692 <start_kernel+40>     auipc   ra,0xff809
0xffffffff80800696 <start_kernel+44>     jalr    836(ra)

remote Thread 1.1 In: start_kernel L?? PC: 0xffffffff8080066a
(gdb) layout asm
(gdb)
```

### 2.3.3 finish

`finish` 命令的功能是运行程序直到当前函数完成返回，并在返回后打印函数返回时的堆栈地址和返回值及参数值等信息。如运行下面该行命令，则会使得`start_kernel`函数返回，即整个程序结束(exit)。

```
(gdb) finish
```

```
(gdb) finish
Run till exit from #0  0xfffffffff8080066a in start_kernel ()
```

### 2.3.4 info

`info` 功能能够查看程序运行中的各种信息，`info`在调试的各种时刻均可使用，是非常有用的查询函数，通常与其它功能配合使用，这里便不再单独为其贴图，下面列出集中常用的 `info` 语句用法。

```
(gdb) info program      ### 输出当前程序运行的状态（正在运行/暂停/停止）
(gdb) info breakpoints  ### 查看所有断点信息
(gdb) info regs         ### 查看运行当前时寄存器信息
(gdb) info locals       ### 显示当前堆栈页的所有变量
(gdb) info function     ### 显示当前运行程序的所有函数及其地址
(gdb) info frame        ### 展示当前栈帧中储存的信息
```

### 2.3.5 frame

`frame` 命令能够切换函数的栈帧，简称为 `f`，通常与 `up` , `down` 以及 `backtrace` , `info frame` 配合使用。在任何一个函数被调用执行时，都会生成一个存储必要信息的栈帧，`frame`命令则能允许我们在递归函数的不同递归层级之间寻找栈帧的信息及变化。在调试具有循环调用或递归调用时的程序时，通过这个命令可以让我们更容易定位异常发生的位置。

```
(gdb) frame spec ### 将spec参数指定为当前的栈帧
(gdb) up n       ### 将x+n栈帧作为新的当前栈帧。
(gdb) down n     ### 将x-n栈帧作为新的当前栈帧。
(gdb) info frame ### 打印当前栈帧的编号、地址、对应函数的储存地址、该函数被调用时代码的储存地址等等
```

```
(gdb) i frame
Stack level 1, frame at 0xfffffffff81204000:
 pc = 0xfffffffff8000116a in _start_kernel; saved pc = <not saved>
Outermost frame: frame did not save the PC
 caller of frame at 0xfffffffff81204000
 Arglist at 0xfffffffff81204000, args:
 Locals at 0xfffffffff81204000, Previous frame's sp is 0xfffffffff81204000
```

### 2.3.6 break

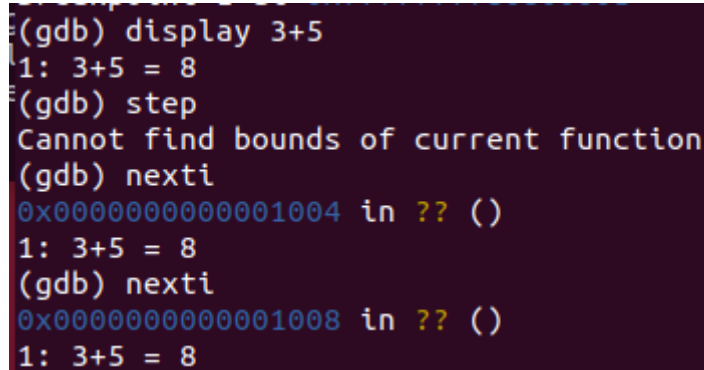
`break` 命令（简称为 `b`）能帮助我们设置断点，程序运行在断点处即自动暂停。该功能最为常用且使用较为简单，这里不做过多解释。

```
(gdb) b function_name ### 在function_name处设置断点
(gdb) b *0x80000000    ### 在程序运行至内存0x80000000处设置断点
(gdb) break test.c:10  ### 在test.c的第十行设置断点
(gdb) info b           ### 查看当前所有断点的信息
```

## 2.3.7 display

`display` 命令（简称为 `d`），它能够在设置一个表达式的值后，每次单步运行输出该表达式的值，起到实时监控的效果。在使用时可以采用如下的方式：

```
(gdb) display variable_a ### 设置显示变量a的内容
(gdb) next                ### 在执行next后变量a的内容将被打印
```



```
(gdb) display 3+5
1: 3+5 = 8
(gdb) step
Cannot find bounds of current function
(gdb) nexti
0x00000000000001004 in ?? ()
1: 3+5 = 8
(gdb) nexti
0x00000000000001008 in ?? ()
1: 3+5 = 8
```

如上图中的 `3+5` 可替换为任意其它在该语言下能够运行的表达式，如在C语言中可以写入一个C表达式，而在Python调试中可以写入一个Python表达式。

## 2.3.8 next & step

`next` 和 `step` 指令可以用于单步调试，主要使用的有以下四种形式，可根据情况要求的不同来灵活使用。

```
(gdb) nexti    ### 单步运行下一条指令，不进入函数
(gdb) stepi    ### 单步运行一条指令
(gdb) next     ### 单步进行到源代码的下一行，不进入函数
(gdb) step     ### 单步到下一个不同的源代码行（包括进入函数）。
```

# 3. 总结与感想

## 3.1 思考题

### 3.1.1 使用 `riscv64-unknown-elf-gcc` 编译单个 `.c` 文件

在terminal输入 `vim test.c`，输入`i`进入插入模式，然后编写一个简单的c语言程序输出"hello world"

```
$ vim test.c
```

```
#include<stdio.h>
#include<stdlib.h>
int main(){
printf("hello,world");
exit(0);
}
```

按下`esc`后输入 `:wq`，保存并退出vim



然后在控制台中输入 `riscv64-unknown-elf-gcc test.c -o test` 进行编译，输入 `ls` 可以看到编译成功的 `test` 可执行文件

```
$ riscv64-unknown-elf-gcc test.c -o test
$ ls
```

```
root@943b3fad0a4c:~/linux/os21fall/src/lab0# vim test.c
root@943b3fad0a4c:~/linux/os21fall/src/lab0# ls
rootfs.img  test  test.c
```

### 3.1.2 使用 `riscv64-unknown-elf-objdump` 反汇编 1 中得到的编译产物

完成1后，输入 `riscv64-unknown-elf-objdump -d test` 对 `test` 进行反汇编，可以得到反汇编的结果：

```
1c20c: 0055e593      ori      a1,a1,5
1c210: b569          j        1c09a <__trunctfdf2+0xa8>

0000000000001c212 <__clzdi2>:
1c212: 03800793      li       a5,56
1c216: 00f55733      srl     a4,a0,a5
1c21a: 0ff77713      zext.b  a4,a4
1c21e: e319         bnez     a4,1c224 <__clzdi2+0x12>
1c220: 17e1         addi     a5,a5,-8
1c222: fbf5         bnez     a5,1c216 <__clzdi2+0x4>
1c224: 6775         lui      a4,0x1d
1c226: 04000693      li       a3,64
1c22a: 8e9d         sub      a3,a3,a5
1c22c: 00f55533      srl     a0,a0,a5
1c230: 06870793      addi     a5,a4,104 # 1d068 <__clz_tab>
1c234: 97aa         add      a5,a5,a0
1c236: 0007c503      lbu      a0,0(a5)
1c23a: 40a6853b      subw     a0,a3,a0
1c23e: 8082         ret

root@943b3fad0a4c:~/linux/os21fall/src/lab0#
```

### 3.1.3 调试 Linux 时:

#### 1. 在 GDB 中查看汇编代码

首先使用 `info func` 来查看编译过程中所有的函数，然后随便选取其中的一个函数进行反汇编，这里选择 `_start`

```
(gdb) info func
All defined functions:

Non-debugging symbols:
0xffffffff80000000  _start
0xffffffff80000040  pe_head_start
0xffffffff80000044  coff_header
0xffffffff80000058  optional_header
0xffffffff80000070  extra_header_fields
0xffffffff800000f8  section_table
0xffffffff80001000  efi_header_end
0xffffffff80001000  relocate
0xffffffff80001062  secondary_start_sbi
0xffffffff800010a4  secondary_start_common
0xffffffff800010bc  setup_trap_vector
0xffffffff800010d4  __efistub__start_kernel
0xffffffff800010d4  _start_kernel
0xffffffff80001114  clear_bss
0xffffffff8000111e  clear_bss_done
0xffffffff80002000  _stext
0xffffffff80002000  _text
0xffffffff80002000  initcall_blacklisted
0xffffffff800020a4  do_one_initcall
```

```
(gdb) disass _start
```

结果发现系统提示 Cannot access memory

```
(gdb) disass _start
Dump of assembler code for function _start:
   0xffffffff80000000 <+0>:      Cannot access memory at address 0xffffffff800000
00
(gdb) █
```

这里证明在进行gdb调试的过程中，内存的访问并不是完全自由的，`_start`所处的内存此时无法访问，不过我们可以去访问指定内存范围的汇编码，如下面这行代码就可以访问0x80000000~0x800000ff处的汇编码：

```
(gdb) disass 0x80000000,0x800000ff
```

```

Breakpoint 1, 0x00000000 in ?? ()
(gdb) disass 0x80000000,0x800000ff
Dump of assembler code from 0x80000000 to 0x800000ff:
=> 0x0000000008000000: add    s0,a0,zero
    0x0000000008000004: add    s1,a1,zero
    0x0000000008000008: add    s2,a2,zero
    0x000000000800000c: jal    ra,0x800006a0
    0x0000000008000010: add    a6,a0,zero
    0x0000000008000014: add    a0,s0,zero
    0x0000000008000018: add    a1,s1,zero
    0x000000000800001c: add    a2,s2,zero
    0x0000000008000020: li     a7,-1
    0x0000000008000022: beq    a6,a7,0x8000002a
    0x0000000008000026: bne    a0,a6,0x80000160
    0x000000000800002a: auipc  a6,0x0
    0x000000000800002e: addi   a6,a6,1166 # 0x800004b8
    0x0000000008000032: li     a7,1
    0x0000000008000034: amoadw a6,a7,(a6)
    0x0000000008000038: bnez   a6,0x80000160
    0x000000000800003c: auipc  t0,0x0
    0x0000000008000040: addi   t0,t0,1164 # 0x800004c8
    0x0000000008000044: auipc  t1,0x0
    0x0000000008000048: addi   t1,t1,-68 # 0x80000000
    0x000000000800004c: sd     t1,0(t0)
    0x0000000008000050: auipc  t0,0x0
    0x0000000008000054: addi   t0,t0,1152 # 0x800004d0
    0x0000000008000058: ld     t0,0(t0)
--Type <RET> for more, q to quit, c to continue without paging--

```

2. 在 0x80000000 处下断点

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

3. 查看所有已下的断点

```
(gdb) info breakpoints
```

4. 在 0x80200000 处下断点

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

5. 清除 0x80000000 处的断点

```
(gdb) del 1 ### 此时0x80000000处的断点为第一个断点，所以清除时直接输入del 1即可
```

6. 继续运行直到触发 0x80200000 处的断点

```
(gdb) continue
```

## 7. 单步调试一次

```
(gdb) nexti    ### 单步运行下一条指令，不进入函数
(gdb) stepi    ### 单步运行一条指令
(gdb) next     ### 单步进行到源代码的下一行，不进入函数
(gdb) step     ### 单步到下一个不同的源代码行（包括进入函数）。
```

## 8. 退出 QEMU

```
(gdb) target remote :1234
Remote debugging using :1234
0x00000000000001000 in ?? ()
(gdb) break *0x80000000
Breakpoint 1 at 0x80000000
(gdb) b *0x80200000
Breakpoint 2 at 0x80200000
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint      keep y   0x0000000008000000
2        breakpoint      keep y   0x0000000008020000

(gdb) del 1
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
2        breakpoint      keep y   0x0000000008020000
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000008020000 in ?? ()
(gdb) step
Cannot find bounds of current function
(gdb) stepi
0x0000000008020002 in ?? ()
```

在QEMU中按 `ctrl+A` 然后按 `x` 即可退出QEMU

### 3.1.4 使用 `make` 工具清除 Linux 的构建产物

可以用以下的两条指令来清除，注意到，`make mrproper` 的程度更深，其首先会调用 `make clean`，这里选择 `make clean` 来进行执行即可。

```
$ make clean    ### 使用make clean可以清除所有编译好的obj文件
$ make mrproper ### 删除所有的编译生成文件、内核配置文件(.config文件)和各种备份文件
```

```
(gdb) make clean
CLEAN    drivers/firmware/efi/libstub
CLEAN    drivers/gpu/drm/radeon
CLEAN    drivers/scsi
CLEAN    drivers/tty/vt
CLEAN    kernel
CLEAN    lib
CLEAN    usr
CLEAN    vmlinux.symvers modules-only.symvers modules.builtin modules.b
odinfo
```

### 3.1.5 vmlinux 和 Image 的关系和区别是什么？

objcopy用于将object的部分或全部内容拷贝到另一个object，从而可以实现格式的变换。Image是经过objcopy后处理的，只包含内核代码和数据的文件，Image是一个二进制格式的文件而不是.elf格式文件，没有经过压缩。而vmlinux是由经过压缩的Image和加入解压头文件组成的ELF格式文件。因而，在使用QEMU运行的是Image而用GDB运行的是vmlinux。

## 3.2 一些心得体会

Linux作为一个之前未曾接触过的操作系统，我在学习Linux的终端、命令行操作的过程中花费了一些时间和精力，而对于这次的实验来说，重要的前提是理解虚拟机、Ubuntu系统、docker、GDB、QEMU以及所需编译的Linux内核之间的关系和结构。在了解清楚这些工具之间的关系之后，利用GDB强大的功能进行调试、测试的过程，其实是较为轻松的一个过程。除了完成实验本身的探索之外，我还学习了Linux系统一些其它软件、工具的使用方法，我相信这能给我今后更好的了解Linux操作系统，更方便的完成实验提供一个较好的基础。我在这次实验中的收获颇深。