实验一：

编译运行Linux内核,制作initramfs，并通过qemu+gdb调试

实验目的

- 熟悉Linux系统运行环境
- 制作initramfs
- 掌握Linux内核编译方法
- 学习如何使用gdb调试内核
- 熟悉Linux下常用的文件操作指令

实验环境

- OS：Ubuntu 18.04 LTS (64 bit) (后续实验以本次实验为基础)
- 编译调试Linux内核版本：Kernel 4.9.263
- **注意**：本次实验文档基于Ubuntu系统上实现，可直接在机房完成，或在个人PC上安装虚拟机(vmware workstation/virtual box)完成，请注意需要下载安装64位的Ubuntu镜像。

先导知识

什么是initramfs(基于ramfs的临时文件系统)

- initramfs 是一种以 cpio 格式压缩后的 rootfs 文件系统，它通常和 Linux 内核文件一起被打包成 boot.img 作为启动镜像
- BootLoader 加载 boot.img，并启动内核之后，内核接着就对 cpio 格式的 initramfs 进行解压，并将解压后得到的 rootfs 加载进内存，最后内核会检查 rootfs 中是否存在 init 可执行文件（该 init 文件本质上是一个执行的 shell 脚本），如果存在，就开始执行 init 程序并创建 Linux 系统用户空间 PID 为 1 的进程，然后将磁盘中存放根目录内容的分区真正地挂载到 / 根目录上，最后通过 exec chroot . /sbin/init 命令来将 rootfs 中的根目录切换到挂载了实际磁盘分区文件系统中，并执行 /sbin/init 程序来启动系统中的其他进程和服务。
  基于ramfs开发initramfs，取代了initrd。

什么是initrd

- initrd代指内核启动过程中的一个阶段，临时挂载文件系统，加载硬盘的基础驱动，进而过渡到最终的根文件系统
- 是早期基于ramdisk生成的临时根文件系统的名称
- 现阶段虽然基于initramfs，但是临时根文件系统也依然存在某些发行版称其为initrd
- 例：CentOS 临时根文件系统命名为 initramfs-`uname -r`.img
  Ubuntu 临时根文件系统命名为 initrd-`uname -r`.img

为什么需要initrd/initramfs

- Linux kernel在自身初始化完成之后，需要能够找到并运行第一个用户程序（这个程序通常叫做“init”程序）。用户程序存在于文件系统之中，因此，内核必须找到并挂载一个文件系统才可以成功完成系统的引导过程。

  在grub中提供了一个选项“root=”用来指定第一个文件系统，但随着硬件的发展，很多情况下这个文件系统也许是存放在USB设备，SCSI设备等等多种多样的设备之上，如果需要正确引导，USB或者SCSI驱动模块首先需要运行起来，可是不巧的是，这些驱动程序也是存放在文件系统里，因此会形成一个悖论。

- 为解决此问题，Linux kernel提出了一个RAM disk的解决方案，把一些启动所必须的用户程序和驱动模块放在RAM disk中，这个RAM disk看上去和普通的disk一样，有文件系统，有cache，内核启动时，首先把RAM disk挂载起来，等到init程序和一些必要模块运行起来之后，再切到真正的文件系统之中。
- 上面提到的RAM disk的方案实际上就是initrd。 如果仔细考虑一下，initrd虽然解决了问题但并不完美。 比如，disk有cache机制，对于RAM disk来说，这个cache机制就显得很多余且浪费空间；disk需要文件系统，那文件系统（如ext2等）必须被编译进kernel而不能作为模块来使用。
- Linux 2.6 kernel提出了一种新的实现机制，即initramfs。顾名思义，initramfs只是一种RAM filesystem而不是disk。initramfs实际是一个cpio归档，启动所需的用户程序和驱动模块被归档成一个文件。因此，不需要cache，也不需要文件系统。

实验内容

一、编译内核，制作根文件系统，并使用qemu启动

1、熟悉linux简单指令

- 目标：掌握ls、touch、cat、echo、mkdir、mv、cd、cp等基本指令

```
ls          # 查看当前目录下的所有文件/文件夹
touch 1.txt # 创建1.txt
ls
echo i am 1.txt > 1.txt  # 向1.txt写入内容
cat 1.txt   # 查看1.txt内容
ls -l       # 查看当前目录下的所有文件/文件夹的详细信息
mkdir 1     # 创建目录1
mv 1.txt 1  # 将1.txt移动到目录1
cd 1        # 打开目录1
ls
rm 1.txt    # 删除文件
cd ..       # 回退到上级目录
rm -r 1     # 删除目录
# 当提示权限不允许执行操作时，在命令前加sudo能够以root用户运行此命令,如
sudo rm 1
```

2、下载并编译Linux内核

- 下载linux-4.9.263.tar.gz，解压缩得到目录linux-4.9.263，不妨称之为Linux源代码根目录(以下简称源码根目录)

```
mkdir ~/oslab
cd ~/oslab
wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.9.263.tar.xz #下载备用链接: https://od.srpr.cc/acgg0/linux-4.9.263.tar.xz
tar -Jvxf linux-4.9.263.tar.xz #解压
```

- 准备安装所需依赖库
  - 可能ubuntu官方镜像源安装下载镜像速度较慢，将镜像源切换成ustc源即可，具体方法见下
  - 更换apt-get源为ustc镜像源

```
sudo apt-get install git build-essential libelf-dev xz-utils libssl-dev bc
libncurses5-dev libncursesw5-dev
```

- 进入源代码根目录，并编译配置选择，两种方案可供选择：
  1. 精简配置，减少不必要的驱动编译(速度快，存储小)

```
cd ~/oslab/linux-4.9.263
wget https://raw.githubusercontent.com/ZacharyLiu-
CS/USTC_OS/master/term2021/lab1/.config # raw访问可能不畅，备用链接:
http://222.186.10.65:8080/directlink/3/.config
```

  2. 内核配置(make menuconfig)详述(编译时间较长，占空间)

```
cd ~/oslab/linux-4.9.263
make menuconfig   #本次实验直接选择Save,然后exit
```

- 执行编译指令

```
make -j $((`nproc`-1)) # 此处为使用(你的CPU核心数-1)个线程进行编译，如果虚拟机分配的
cpu数只有1(如Hyper-V默认只分配1核)则需先调整虚拟机分配的核心数.
```

- make过程中若遇到问题，可参考以下方案解决：

```
  # 问题1
  编译内核时遇到 make[1]: *** No rule to make target 'debian/canonical-
certs.pem', needed by 'certs/x509_certificate_list'.  Stop.

  # 问题1解决方案:
  用文本编辑器(vim 或 gedit)打开 PATH-TO-linux-4.9.263/.config文件，找到并注释掉包
含 CONFIG_SYSTEM_TRUSTED_KEY 和 CONFIG_MODULE_SIG_KEY 的两行即可.
  解决方案原链接: https://unix.stackexchange.com/questions/293642/attempting-
to-compile-kernel-yields-a-certification-error
```

3、准备模拟器qemu

- 直接安装qemu包即可

```
sudo apt install qemu

# Ubuntu 20.04/20.10 环境下执行以下指令
sudo apt install qemu-system-x86
```

4、制作根文件系统：利用busybox生成根文件系统,并使用qemu启动运行

- （1）下载busybox

```
  cd ~/oslab
  wget https://busybox.net/downloads/busybox-1.32.1.tar.bz2  #下载 备用链接:
https://od.srpr.cc/acgg0/busybox-1.32.1.tar.bz2
  tar -jxvf busybox-1.32.1.tar.bz2    #解压
  cd ~/oslab/busybox-1.32.1
```

- （2）编译busybox

```
make menuconfig  #修改配置如下:
```

修改配置如下：(空格键勾选)

```
Settings ->
  Build Options
   [*] Build static binary（no share libs）
```

编译并安装

```
make -j $((`nproc`-1))
sudo make install
```

- （3）准备根文件系统

```
cd ~/oslab/busybox-1.32.1/_install
sudo mkdir dev
sudo mknod dev/console c 5 1
sudo mknod dev/ram b 1 0
sudo touch init
```

在init中写入以下内容,你可以使用vim或gedit编辑器写入(vim tutorial)，执行sudo vim init 或 sudo gedit init.

```
#!/bin/sh
echo "INIT SCRIPT"
mkdir /proc
mkdir /sys
mount -t proc none /proc
mount -t sysfs none /sys
mkdir /tmp
mount -t tmpfs none /tmp
echo -e "\nThis boot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
exec /bin/sh
```

赋予init脚本执行，制作initramfs文件，将x86-busybox下面的内容打包归档成cpio文件，以供Linux内核做initramfs启动执行

```
sudo chmod +x init
cd ~/oslab/busybox-1.32.1/_install
find . -print0 | cpio --null -ov --format=newc | gzip -9 >
~/oslab/initramfs-busybox-x64.cpio.gz  # 注意：该命令一定要在busybox的 _install
目录下执行
# 注意：每次修改_install,都要重新执行该命令
```

- （4）运行

```
cd ~/oslab
## qemu 以图形界面，弹出窗口形式运行内核
qemu-system-x86_64 -s -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage
-initrd ~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr
root=/dev/ram init=/init"

## Ubuntu 20.04/20.10 环境下如果出现问题，可执行以下指令 (部分Ubuntu20.04/20.10系统
编译linux后bzImage也在x86_64中，根据实际情况选择bzImage路径)
qemu-system-x86_64 -s -kernel ~/oslab/linux-4.9.263/arch/x86/boot/bzImage -
initrd ~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr root=/dev/ram
init=/init"

## 如不希望qemu以图形界面启动，而希望以无界面形式启动，输出重定向到当前shell，使用以下命
令
qemu-system-x86_64 -s -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage
-initrd  ~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr
root=/dev/ram init=/init console=ttyS0 " -nographic
```

  - 在qemu窗口可以看到成功运行，且进入shell环境
  - 以无图形界面形式启动时, 可以输入Ctrl+A X结束模拟进程并关闭qemu.(图形界面无效)

二、gdb+qemu调试内核

1、gdb简介

- gdb是一款终端环境下常用的调试工具
- 使用gdb调试程序
  - ubuntu下安装gdb：sudo apt install gdb
  - 编译程序时加入-g选项，如：gcc -g -o test test.c
  - 运行gdb调试程序：gdb test
- 常用命令

```
r/run                 # 开始执行程序
b/break <location>    # 在location处添加断点，location可以是代码行数或函数名
b/break <location> if <condition> # 在location处添加断点，仅当caondition条件满足
才中断运行
c/continue            # 继续执行到下一个断点或程序结束
n/next                # 运行下一行代码，如果遇到函数调用直接跳到调用结束
s/step                # 运行下一行代码，如果遇到函数调用则进入函数内部逐行执行
ni/nexti              # 类似next，运行下一行汇编代码（一行c代码可能对应多行汇编代码）
si/stepi              # 类似step，运行下一行汇编代码
list                  # 显示当前行代码
p/print <expression>  # 查看表达式expression的值
```

2、在qemu中启动gdb server

- 在终端中执行以下指令启动qemu运行内核：

```
## qemu 以图形界面，弹出窗口形式运行内核
qemu-system-x86_64 -s -S -kernel ~/oslab/linux-
4.9.263/arch/x86_64/boot/bzImage -initrd ~/oslab/initramfs-busybox-
x64.cpio.gz --append "nokaslr root=/dev/ram init=/init"

## Ubuntu 20.04/20.10 环境下如果出现问题，可执行以下指令 (部分Ubuntu20.04/20.10系统编译linux后bzImage也在x86_64中，根据实际情况选择bzImage路径)
qemu-system-x86_64 -s -S -kernel ~/oslab/linux-4.9.263/arch/x86/boot/bzImage
-initrd ~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr
root=/dev/ram init=/init"

## 如不希望qemu以图形界面启动，而希望以无界面形式启动，输出重定向到当前shell，使用以下命令
qemu-system-x86_64 -s -S -kernel ~/oslab/linux-
4.9.263/arch/x86_64/boot/bzImage -initrd  ~/oslab/initramfs-busybox-
x64.cpio.gz --append "nokaslr root=/dev/ram init=/init console=ttyS0 " -
nographic
```

- 关于-s和-S选项的说明
  - -S freeze CPU at startup (use 'c' to start execution)
  - -s shorthand for -gdb tcp::1234 若不想使用1234端口，则可以使用-gdb tcp:xxxx来取代-s选项

3、建立gdb与gdb server之间的链接

- 在另外一个终端运行gdb，然后在gdb界面中运行如下命令：

```
gdb                 # 这里一定是在另外一个终端运行，不能在qemu的窗口上输入
target remote:1234  # 则可以建立gdb和gdbserver之间的连接
c                   # 让qemu上的Linux继续运行
```

可以看到gdb与qemu已经建立了连接。但是由于没有加载符号表，无法根据符号设置断点。下面说明如何加入断点。

4、重新配置Linux，使之携带调试信息

- 使用Ctrl+A X组合键退出之前打开的qemu终端
- 在原来配置的基础上，重新配置Linux，使之携带调试信息

```
cd ~/oslab/linux-4.9.263/
./scripts/config -e DEBUG_INFO -e GDB_SCRIPTS -d DEBUG_INFO_REDUCED -d
DEBUG_INFO_SPLIT -d DEBUG_INFO_DWARF4
```

- 重新编译

```
make -j $((`nproc`-1))
```

5、加载vmlinux中的符号表并设置断点

- 重新执行第2步 "在qemu中启动gdb server "
- 在另外一个终端输入如下指令运行gdb，加载符号表

```
gdb                                     # 这里一定是在另外一个终端运行，不能在qemu的
窗口上输入
file ~/oslab/linux-4.9.263/vmlinux      # 加载符号表
target remote:1234                      # 建立gdb和gdbserver之间的连接
```

- 在gdb界面中设置断点

```
break start_kernel          # 设置断点
c                           # 继续运行到断点
l                           # 查看断点处代码
p init_task                 # 查看断点处变量值
```

实验检查

1.shell命令检查

1. 以学号为名创建目录
2. 向该目录写入文件，文件名为time.txt,内容为当前时间
3. 打印出该文件内容
4. 删除该文件以及目录

2.gdb调试内核

1. 使用qemu启动内核
2. 使用gdb设置断点，查看断点处代码，变量值，执行下一步代码

参考资料

- 内核编译与制作根文件系统
- 引导过程简介
- Initramfs/指南
- initramfs原理探讨
- 内核调试技巧