实验0: Rinux环境搭建和内核编译

0 实验简介

搭建实验虚拟机、docker运行环境。通过在QEMU上运行Linux来熟悉如何从源代码开始将内核运行在QEMU模拟器上，并且掌握使用gdb协同QEMU进行联合调试，为后续实验打下基础。

---

1 实验目的

- 了解容器的使用
- 使用交叉编译工具, 完成Linux内核代码编译
- 使用QEMU运行内核
- 熟悉GDB和QEMU联合调试

---

2 实验环境

- Docker
- 实验环境镜像: oslab.tar 下载地址, 备用下载地址

Docker Image：是一个不可变（不可更改）的文件，其中包含应用程序运行所需的源代码、库、依赖项、工具和其他文件。他是一个模板，我们并不能直接使用它，而是通过`docker run`命令将Image生成Container使用。

Docker Container：是一个虚拟化的运行时环境，用户可以在其中将应用程序与底层系统隔离开来。我们可以在其中快速轻松地启动应用程序，就像使用虚拟机一样。我们可以通过`docker start`命令启动Container。

作为类比，可将Image认为是一张装机光盘，生成Container就是装机过程，装机之后我们直接使用电脑(Container)而不是光盘(Image)。

在本次实验中，我们所下载使用的`oslab.tar`是`Docker` Image，包含了本实验所需的Linux系统的命令程序、RISCV QEMU环境、GCC环境、GDB和部分源代码等。

---

3 实验基础知识介绍

3.1 Linux 使用基础

在Linux环境下，人们通常使用命令行接口（*command-line interface*,缩写:CLI）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，通过终端我们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，终端可以为我们完成一切所需要的操作。下面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

3.2 环境变量介绍

当我们在终端输入命令时，终端会找到对应的程序来运行。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 5月  21  2019 /usr/bin/gcc -> gcc-7
```

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
```

在后面的实验中，如果你想直接访问`riscv64-unknown-linux-gnu-gcc`、`qemu-system-riscv64`等程序，那么你需要把他们所在的目录添加到目录中。

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

也可以在主目录的.bashrc文件的最后，添加上述命令行。然后，用如下命令验证其效果:

```
$ env | grep PATH -
```

3.3 Docker 使用基础

Docker 基本介绍

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。之所以要把应用程序打包，并以容器的方式运行，主要是因为在生产开发环境中，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。Docker带来的好处是只要我们将应用程序打包完成（组装成为Docker imgae），在任意安装了Docker的机器上，都可以通过运行容器的方式来运行该应用程序，因而将依赖、环境变量等带来的应用部署问题解决了。

Docker和虚拟机功能上有共同点，但是和虚拟机不同，Docker不需要创建整个操作系统，只需要将应用程序的二进制和有关的依赖文件打包，因而容器内的应用程序实际上使用的是容器外Host的操作系统内核。这种共享内核的方式使得Docker的移植和启动非常的迅速，同时由于不需要创建新的OS，Docker对于容器物理资源的管理也更加的灵活，Docker用户可以根据需要动态的调整容器使用的计算资源（通过cgroups）。

如果想了解更多 Docker 的详情，请参考官网。

Docker 安装

如果你在 Ubuntu 发行版上安装 Docker，请参考这里。

其余平台请根据 https://docs.docker.com/get-docker 自行在本机安装 Docker 环境。

你可以从 2 实验环境 中获得实验所需的环境，我们已经在Docker镜像中为你准备好了 RISC-V 工具链，以及 QEMU 模拟器，使用方法请参见 4 实验步骤。

3.4 QEMU 使用基础

什么是QEMU

QEMU最开始是由法国程序员Fabrice Bellard开发的模拟器。QEMU能够完成用户程序模拟和系统虚拟化模拟。用户程序模拟指的是QEMU能够将为一个平台编译的二进制文件运行在另一个不同的平台，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（比如Intel x86）的代码。系统虚拟化模拟指的是QEMU能够模拟一个完整的系统虚拟机，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，能够为虚拟机中运行的操作系统和应用软件呈现出与物理计算机完全一致的硬件视图。

我们实验中采用 QEMU 来完成 RISC-V 架构的程序的模拟。

如何使用 QEMU（常见参数介绍）

以以下命令为例，我们简单介绍 QEMU 的参数所代表的含义

```
$ qemu-system-riscv64 \
    -nographic \
    -machine virt \
    -kernel path/to/linux/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 \
    -append "root=/dev/vda ro console=ttyS0" \
    -bios default \
    -drive file=rootfs.img,format=raw,id=hd0 \
    -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作为内核的命令行
- `-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_name` 作为文件系统
- `-S`: 启动时暂停CPU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这里

3.5 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（本地调试，or native debug），也可以是不同机器上（远程调试， or remote debug）。

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

- (gdb) layout asm: 显示汇编代码
- (gdb) start: 单步执行，运行程序，停在第一执行语句
- (gdb) continue: 从断点后继续执行，简写 `c`
- (gdb) next: 单步调试（逐过程，函数直接执行），简写 `n`
- (gdb) step instruction: 执行单条指令，简写 `si`
- (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，简写 `b`
  - 断在某地址: `b * 0x80200000`
- (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 ra 的值：`i r ra`
- (gdb) display：追踪查看具体变量值
- (gdb) x/4x : 以 16 进制打印 处开始的 16 Bytes 内容

更多命令可以参考100个gdb小技巧

3.6 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，在我们的实验中所用的交叉编译工具链就是 `riscv-gnu-toolchain`。

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。

常用的 Linux 下的编译选项

```
$ make help          # 查看make命令的各种参数解释

$ 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) 为以全部机器硬件线程数进行多线程编译
$ make -j4           # 编译当前平台的内核，-j4 为使用 4 线程进行多线程编译

$ 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
```

4 实验步骤

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。

> 注：通常在Linux环境下，`$` 提示符表示当前运行的用户为普通用户，`#` 代表当前运行的用户为特权用户。

> 注意，在下文的示例中：
>
> - 以 `###` 开头的行代表注释，
> - 以 `$` 开头的行代表在你的宿主机/虚拟机上运行的命令，
> - 以 `#` 开头的行代表在 `docker` 中运行的命令，
> - 以 `(gdb)` 开头的行代表在 `gdb` 中运行的命令。

4.1 搭建 Docker 环境

请根据 **3.3 Docker 使用基础** 安装 Docker 环境。然后**参考并理解**以下步骤，导入我们已经准备好的 Docker 镜像：

```
### 导入docker镜像
$ cat oslab.tar | docker import - oslab:2022

### 执行命令后若出现以下错误提示
### ERROR: Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock
### 可以使用下面命令为该文件添加权限来解决
### $ sudo chmod a+rw /var/run/docker.sock

### 查看docker镜像
$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED         SIZE
oslab            2022      8c15472cd111   5 months ago    3.63GB

### 从镜像创建一个容器
$ docker run --name oslab -it oslab:2022 /bin/bash   # --name:容器名称 -i:交互式操作 -t:终端
root@132a140bd724:/#                                 # 提示符变为 '#' 表明成功进入容器 后面的字符串根据容器而生成，为容器
id
root@132a140bd724:/# exit (or CTRL+D)                # 从容器中退出 此时运行docker ps，运行容器的列表为空

### 启动处于停止状态的容器
$ docker start oslab        # oslab为容器名称
$ docker ps                 # 可看到容器已经启动
CONTAINER ID   IMAGE         COMMAND         CREATED           STATUS          PORTS     NAMES
384d8e7f6bd4   oslab:2022    "/bin/bash"     23 minutes ago    Up 3 seconds              oslab


### 从终端连入 docker 容器
$ docker exec -it oslab /bin/bash

### 进入docker后，按4.2-4.5指导进行下一步实验
```

```
### 其它可能会用到的Docker命令：

### 1. 挂载本地目录
### 把用户的 home 目录映射到 docker 镜像内的 /home/lab0 目录
$ docker run --name oslab -it -v ${HOME}:/home/oslab oslab:2022 /bin/bash
### -v 本地目录:容器内目录


### 2. docker与本地文件复制命令，如：
$ docker cp /home/lab 368c4cc44221:/home/lab0
$ docker COMMAND --help 可以帮助你获得更多的使用docker的帮助信息
```

4.2 获取 Linux 源码和已经编译好的文件系统

1. 进入 `/home` 目录。

```
# cd /home
```

2. 使用 git 工具 clone 本仓库。其中已经准备好了根文件系统的镜像。

> 根文件系统为 Linux Kenrel 提供了基础的文件服务，在启动 Linux Kernel 时是必要的。

```
# git clone https://gitee.com/zjusec/os21fall
# cd os21fall/src/lab0
# ls
rootfs.img    ### 已经构建完成的根文件系统的镜像
```

3. 在当前目录下，从 https://www.kernel.org 下载最新稳定版本的 Linux 源码。

```
# pwd          ### 查看当前目录的全路径名称
/home/os21fall/src/lab0

# apt install wget    ### 安装下载工具
# wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.19.8.tar.xz

### 国内备用下载地址：
# wget http://ftp.sjtu.edu.cn/sites/ftp.kernel.org/pub/linux/kernel/v5.x/linux-5.19.8.tar.gz

# ls
linux-5.19.8.tar.gz  rootfs.img
```

4. 使用解压缩Linux 源码包至 /home/os21fall/src/lab0 目录下

```
# tar -zxvf linux-5.19.8.tar.gz
```

4.3 编译 linux 内核

```
# pwd
/home/os21fall/src/lab0/linux-5.19.8
# export RISCV=/opt/riscv                                                  ### 设置环境变量
# export PATH=$PATH:$RISCV/bin
# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig    ### 生成配置
# make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc)   ### 编译
```

> 使用多线程编译一般会耗费大量内存，如果 `-j` 选项导致内存耗尽 (out of memory)，请尝试调低线程数，比如 `-j4`, `-j8` 等。

4.4 使用QEMU运行内核

```
# pwd
/home/os21fall/src/lab0/
# qemu-system-riscv64 -nographic -machine virt -kernel path/to/linux-5.19.8/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
    -bios default -drive file=rootfs.img,format=raw,id=hd0
```

退出QEMU模拟器的方法为：使用`ctrl`+`a`(macOS下为`control+a`)，松开后再按下`x`键即可退出qemu

4.5 使用 gdb 对内核进行调试

这一步需要开启两个 Terminal Session，一个 Terminal 使用 QEMU 启动 Linux，另一个 Terminal 使用 GDB 与 QEMU 远程通信（使用 tcp::1234 端口）进行调试。

```
### Terminal 1
# pwd
/home/os21fall/src/lab0/
# qemu-system-riscv64 -nographic -machine virt -kernel path/to/linux-5.19.8/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
    -bios default -drive file=rootfs.img,format=raw,id=hd0 -S -s

### Terminal 2
# export RISCV=/opt/riscv           ### 设置环境变量
# export PATH=$PATH:$RISCV/bin
# riscv64-unknown-linux-gnu-gdb path/to/vmlinux
(gdb) target remote localhost:1234  ### 连接 qemu
(gdb) b start_kernel                ### 设置断点
(gdb) continue                      ### 继续执行
(gdb) quit                          ### 退出 gdb
```

若gdb提示如下信息：

```
Reading symbols from vmlinux…
(No debugging symbols found in vmlinux)
```

需要在内核`Makefile`的`KBUILD_CFLAGS`上添加 **-g** 选项，然后重新编译内核，继续运行上述命令行启动gdb开始调试。

5 实验任务与要求

- 请各位同学独立完成作业，任何抄袭行为都将使本次实验判为0分。
- 编译内核并用 gdb + QEMU 调试，在内核初始化过程中（用户登录之前）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（如backtrace、finish、frame、info、break、display、next等），通过实验验收。
- 在学在浙大中提交实验报告，记录实验过程并截图（4.1-4.4），对每一步的命令以及结果进行必要的解释，记录遇到的问题和心得体会。

思考题

1. 使用 `riscv64-unknown-elf-gcc` 编译单个 `.c` 文件
2. 使用 `riscv64-unknown-elf-objdump` 反汇编 1 中得到的编译产物
3. 调试 Linux 时:
    1. 在 GDB 中查看汇编代码
    2. 在 0x80000000 处下断点
    3. 查看所有已下的断点
    4. 在 0x80200000 处下断点
    5. 清除 0x80000000 处的断点
    6. 继续运行直到触发 0x80200000 处的断点
    7. 单步调试一次
    8. 退出 QEMU
4. 使用 `make` 工具清除 Linux 的构建产物
5. `vmlinux` 和 `Image` 的关系和区别是什么?

浙江大学实验报告

课程名称：　操作系统

实验项目名称：

学生姓名：　　学号：

电子邮件地址：

实验日期：　年　　月　　日

一、实验内容

记录实验过程并截图，对每一步的命令以及结果进行必要的解释

三、讨论、心得（20分）

　# 在这里写：实验过程中遇到的问题及解决的方法，你做本实验体会