Lab 0: docker+GDB+QEMU调试64位RISC-V LINUX

1 实验目的

搭建虚拟机、docker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。

2 实验环境

在虚拟机环境运行docker，并且安装：

* VMware Workstation 15 Player
* Ubuntu 18.04.5 LTS
* docker
* qemu 5.0.0
* riscv-gnu-toolchain

可能还需要安装：

* buildroot 2020.08-rc1
* linux 5.4.0-77-generic

**2.1 虚拟机环境**

Linux受GNU通用公共许可证（GPL）保护，其内核源代码是完全开放的。现在很多Linux的网站都提供内核代码的下载。推荐你使用Linux的官方网站：http://www.kernel.org，在这里你可以找到所有的内核版本。

考虑到网络下载速度，你也可以从https://mirrors.aliyun.com/linux-kernel/就近下载。也可以从https://mirrors.zju.edu.cn/ubuntu-releases/或者https://mirrors.aliyun.com/ubuntu-releases/下载

ubuntu-18.04.5-desktop-amd64.iso。

VMware Workstation 15 Player环境安装ubuntu 18.04.5 LTS：

先行单独创建虚拟机ubuntu，不同时安装Linux系统；

虚拟机ubuntu的内存容量，从2GB调增至4GB，便于后续编译内核；硬盘容量50G

安装Linux前，光驱链接至文件ubuntu-18.04.5-desktop-amd64.iso；

安装完成Linux后，光驱链接至文件OSLAB.TAR.ISO（内含oslab.tar）;

更新软件源。/etc/apt/sources.list罗列的官方地址耗时间，选用<http://mirrors.aliyun.com/ubuntu>，修改配置文件的内容即可：

deb http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe multiverse

deb http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted universe multiverse

deb http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted universe multiverse

deb http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted universe multiverse

deb http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted universe multiverse

deb-src http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe multiverse

deb-src http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted universe multiverse

deb-src http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted universe multiverse

deb-src http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted universe multiverse

deb-src http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted universe multiverse

执行：

$ sudo apt-get update && sudo apt-get upgrade -y

安装VMWare Tools（高版本的环境，已经更名为open-vm-tools）：

$ sudo apt-get install open-vm-tools-desktop

$ sudo apt-get install open-vm-tools

安装gitee客户端（也就是ubuntu环境中git的客户端），方便做实验：

$sudo apt install git-all

保险起见，首先确认你的Linux运行环境（也就是ubuntu 18.04.5 LTS）究竟采用哪个版本的内核源代码，第一步首先想办法获取合适版本的Linux内核代码。通过命令

$ uname –r

5.4.0-77-generic

这就是说，ubuntu 18.04.5 LTS采用的内核版本是5.4.0-77-generic。检查/usr/src确认，本机环境尚未部署这个版本的Linux内核。

**2.2 实验工具**

配齐QEMU环境、gcc环境：

$ sudo apt-get install qemu

$ sudo apt-get install -y gcc make qemu build-essential module-assistant gcc-multilib g++-multilib

$ sudo dpkg --add-architecture i386

$ sudo apt-get install -y libc6:i386 libncurses5:i386 libstdc++6:i386

$ sudo apt-get install -y gdb-multiarch

3 实验基础知识

**3.1 Linux使用基础**

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。

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](https://missing-semester-cn.github.io/2020/shell-tools) [Video](https://www.bilibili.com/video/BV1x7411H7wa?p=2)

2. [GNU/Linux Command-Line Tools Summary](https://tldp.org/LDP/GNU-Linux-Tools-Summary/html/index.html)

3. [Basics of UNIX](https://github.com/berkeley-scf/tutorial-unix-basics)

**环境变量**

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:/riscv-glibc

可以在家目录的“.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 –

**3.2 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，以及其它依赖文件，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 image），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，动态地调整容器使用的计算资源（通过cgroups）。

**Docker安装**

如果你在 Ubuntu 发行版上安装 Docker，请参考[这里](https://docs.docker.com/engine/install/ubuntu/)，其余平台请自行查找。

然而，然而，然而，先别动手！

本实验已经准备好了RISC-V工具链，包括docker、QEMU等，使用方法请参见4.1。

**3.3 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。

**如何使用QEMU（常见参数介绍）**

以下述命令为例，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 \

-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

**-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”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: 是** -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](https://www.qemu.org/docs/master/system/index.html)

**3.4 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) start：**运行程序，停在第一执行语句

**(gdb) next：**单步调试（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

更多命令可以参考[100个gdb小技巧](https://wizardforcel.gitbooks.io/100-gdb-tips/content/)

**3.5 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，我们将在后续实验的Makefile中频繁遇到**

$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

4 实验步骤

通常情况下， $ 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。

但注意，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。

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。

**4.1 搭建 Docker 环境**

docker安装可使用下面命令，安装并运用命令行工具curl(client URL)，与docker的Web服务器交互：

$ sudo apt-get install curl

$ curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun

将用户加入docker组，免sudo

$ sudo usermod -aG docker $USER ### Linux关机，重新启动后，生效。

### 使用官方安装方法还需执行以下语句

### sudo chmod a+rw /home/$USER/.docker/config.json

### 将oslab.tar文件，复制到家目录

导入docker镜像

$ docker load --input oslab.tar

### 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

Alphavake/oslab latest b2b39a3bcd81 404 days ago 3.62GB

从镜像创建一个容器

$ docker run --name oslab -it alphavake/oslab 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

132a140bd724 alphavake/oslab "bash" About a minute ago Up 1 second oslab

从终端连入 docker 容器

$ docker exec -it oslab bash

**4.2 获取 Linux 源码和已经编译好的文件系统**

从<https://www.kernel.org>下载Linux 源码。

并且使用 git工具clone[本仓库](https://gitee.com/zjusec/os21fall)。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

**4.3 编译Linux内核（同时参考“实验一指导 Linux内核重建.doc”）**

不妨在家目录下设立lab0和lab0/linux

在此目录下部署Linux源码。

创建、进入docker容器：

$ cd ~/lab0

$ cp ~/os21fall/src/lab0/rootfs.img ~/lab0

$ docker run --name oslab -it -v `pwd`:/home/lab0 -w /home/lab0 alphavake/oslab bash

root@d143831ff6a9:/home/lab0#

# cd linux

生成配置文件

# make ARCH=riscv CROSS\_COMPILE=riscv64-unknown-linux-gnu- defconfig

编译内核

# make ARCH=riscv CROSS\_COMPILE=riscv64-unknown-linux-gnu- -j$(nproc)

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**

# cd ..

# qemu-system-riscv64 -nographic -machine virt -kernel 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

退出 QEMU 的方法为：使用 Ctrl+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

# qemu-system-riscv64 -nographic -machine virt -kernel 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

### Terminal 2

# riscv64-unknown-linux-gnu-gdb linux/vmlinux

(gdb) target remote:1234 # 连接 qemu

(gdb) b start\_kernel # 设置断点

(gdb) continue # 继续执行

(gdb) quit # 退出 gdb

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作业判为0分。
* 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（如backtrace、finish、frame、info、break、display、next、layout等）。
* 在“学在浙大”中提交 pdf 格式的实验报告，记录实验过程并截图（4.1-4.5），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？