

ARM 虚拟环境相关安装 及示例程序 实验指导



华为技术有限公司
大连理工大学 赖晓晨
(华科大 汇编语言课程组 摘录)

目录

前 言	4
简介	4
实验环境说明	4
1 基于 QEMU 模拟器的鲲鹏 920 处理器开发环境搭建	5
1.1 实验目的	5
1.2 实验设备	5
1.3 实验原理	5
1.3.1 QEMU 简介	5
1.3.2 QEMU 的优缺点	6
1.3.3 openEuler 操作系统	6
1.3.4 openEuler 社区	6
1.1 实验任务操作指导	7
1.3.5 QEMU 的安装配置	7
1.3.6 openEuler 操作系统安装	9
1.3.7 网络配置	12
1.4 编程工具的其他说明	17
1.4.1 汇编 as 与链接 ld 命令	17
1.4.2 调试工具 gdb	18
2 示例程序	20
2.1 C 与汇编的混合编程实验原理	20
2.1.1 C 语言调用汇编实现累加和求值	21
2.1.2 C 语言内嵌汇编	22
2.2 内存拷贝及优化实验原理	24
2.2.1 基础代码	24
2.2.2 循环展开优化	26
2.2.3 内存突发传输方式优化	27
3 附录 1: Linux 常用命令	29
3.1 基本命令	29
3.1.1 关机和重启	29
3.1.2 帮助命令	29
3.2 目录操作命令	29
3.2.1 目录切换命令	29
3.2.2 目录查看命令	30

3.2.3 目录操作命令	30
3.3 文件操作命令	31
3.3.1 新建文件	31
3.3.2 删除文件	31
3.3.3 修改文件	31
3.3.4 查看文件	32
4 附录 2: ARM 指令	33
4.1 LDR 字数据加载指令	33
4.2 LDRB 字节数据加载指令	33
4.3 LDRH 半字数据加载指令	34
4.4 STR 字数据存储指令	34
4.5 STRB 字节数据存储指令	34
4.6 STRH 半字数据存储指令	35
4.7 LDP/STP 指令	35

前言

简介

本实验指导手册为基于鲲鹏 920 处理器的实验指导，适用于希望了解 ARM 汇编基础知识、汇编代码优化、以及鲲鹏 920 处理器相关技术的读者。

实验环境说明

- QEMU 虚拟机；
- openEuler 20.03 操作系统；
- 配套的编辑器、编译器、调试器等。

1 基于 QEMU 模拟器的鲲鹏 920 处理器开发环境搭建

1.1 实验目的

鲲鹏处理器是基于 ARM 架构的企业级处理器产品，兼容了 ARM v8 指令集。本次实验旨在 x86 系统上搭建出能够兼容 ARM v8 指令集的模拟环境，为鲲鹏处理器的学习提供环境。目前 Windows 系统仍是主流，因此本节介绍一种在 x86+Windows 平台上运行与 ARM v8 指令集兼容的模拟环境的方法。

本实验将通过四个部分介绍模拟环境的搭建：

第一部分，介绍计算机模拟的开源软件——QEMU，其能够实现在一种体系结构上执行另一种体系结构程序的功能。

第二部分，介绍 openEuler 操作系统。

第三部分，为鲲鹏开发环境搭建的操作指南，从 QEMU 模拟器的安装到操作系统的安装，以及网络配置的相关操作。

第四部分，通过一个简单的程序，完成鲲鹏开发环境的测试。

1.2 实验设备

- 个人电脑，WINDOWS 操作系统。

1.3 实验原理

1.3.1 QEMU 简介

QEMU 是一款通用、开源的计算机仿真器，它通过动态翻译来模拟 CPU，将客户操作系统的指令翻译给真正的硬件执行，实现对另一种体系结构计算机的模拟。经过 QEMU 的翻译，客户操作系统可以间接地同真实主机中的 CPU、网卡、硬盘等硬件设备进行交互。由于程序执行过程需要 QEMU 的翻译，程序执行的性能与速度会比在真实主机上差。

1.3.2 QEMU 的优缺点

QEMU 的核心是能够支持多种架构，能够虚拟不同的硬件平台架构。表 1-1 为 QEMU 的优缺点列举。

表1-1 QEMU 的优缺点列举

优点	缺点
支持多种架构,可以虚拟不同的硬件平台架构	对不常用的架构支持不完善
可扩展,可自定义新的指令集	对某些操作系统支持的不完善
可以在其他平台上运行Linux的程序	安装与使用不是很方便,操作难度比其他管理系统要大
可以虚拟网卡	模拟速度稍慢
可以储存和还原运行状态	

选用 QEMU 的关键原因是它能够在 x86 上模拟出兼容 ARM v8 指令集的环境。

1.3.3 openEuler 操作系统

openEuler 是一款开源操作系统，当前 openEuler 内核源于 Linux，支持鲲鹏及其它多种处理器，能够充分释放计算芯片的潜能，是由全球开源贡献者构建的高效、稳定、安全的开源操作系统，适用于数据库、大数据、云计算、人工智能等应用场景。同时，openEuler 也拥有一个面向全球的操作系统开源社区，通过社区合作，打造创新平台，构建支持多处理器架构、统一和开放的操作系统，推动软硬件应用生态繁荣发展。

理论上所有可以支持 ARM v8 指令集的操作系统都可以兼容鲲鹏芯片。openEuler 作为华为多年研发投入的产品，针对鲲鹏芯片做了相当多的底层优化，可以更有效的发挥处理器的性能，所以我们选择 openEuler 作为模拟环境的操作系统。

1.3.4 openEuler 社区

openEuler 社区（<https://openeuler.org/zh/>）不仅仅是个操作系统社区，更是一个极具活力的开源社区。在最近一年时间内，openEuler 社区已经发展成了中国活跃度最高的开源社区。

openEuler 已经成为 Linux 内核的重要贡献者，尤其体现在对 ARM 体系架构的支持上。在 Linux Kernel 5.10 版本中，华为 patch 提交数量全球第一，代码修改行数全球第二。可以说，openEuler 是中国开源的变革者，中国开源的里程碑。

1.1 实验任务操作指导

1.3.5 QEMU 的安装配置

1.3.5.1 QEMU 下载安装

进入 QEMU 官方下载页（<https://qemu.weilnetz.de/w64/2019/>），找到 QEMU for Windows，下载最新版的 qemu-w64-setup-20190815.exe，以便在 Windows 上模拟鲲鹏处理器。（在华工汇编在线答疑群里也可下载）

下载完成后安装，自定义安装路径，在 D 盘中创建一个 D:/qemutest 文件夹，并把该文件夹作为安装路径，如图 1-1 所示：

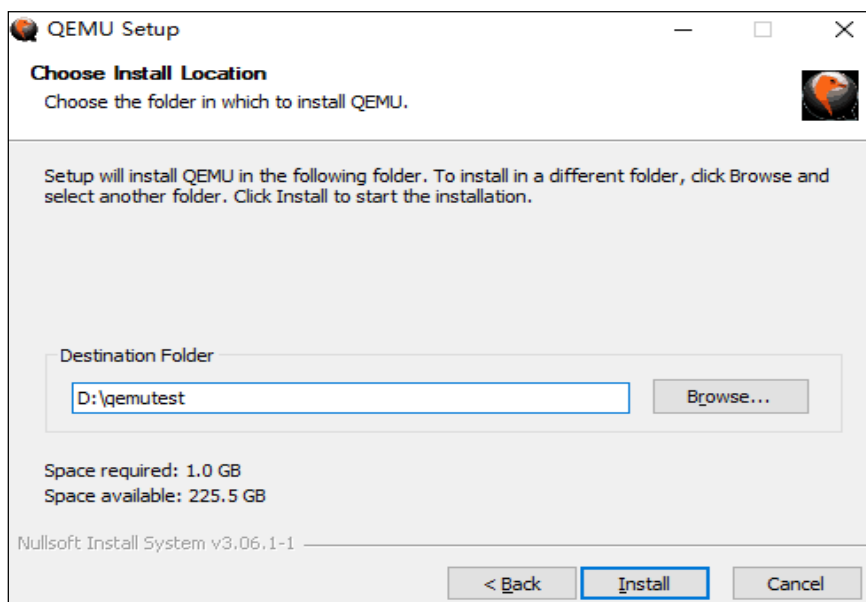


图1-1 安装路径选择

1.3.5.2 环境变量配置

软件安装完成后，还需要进行环境变量配置。

打开任意文件夹，选中左侧菜单栏中的此电脑，点击右键，选中属性进入计算机的系统界面。

在左侧控制面板主页中点击高级系统设置，如图 1-2 所示：



图1-2 控制面板主页

在系统属性中选择高级菜单中的环境变量，如图 1-3 所示：



图1-3 环境变量

在系统变量中找到 PATH，双击打开，进入编辑界面，如图 1-4 所示：

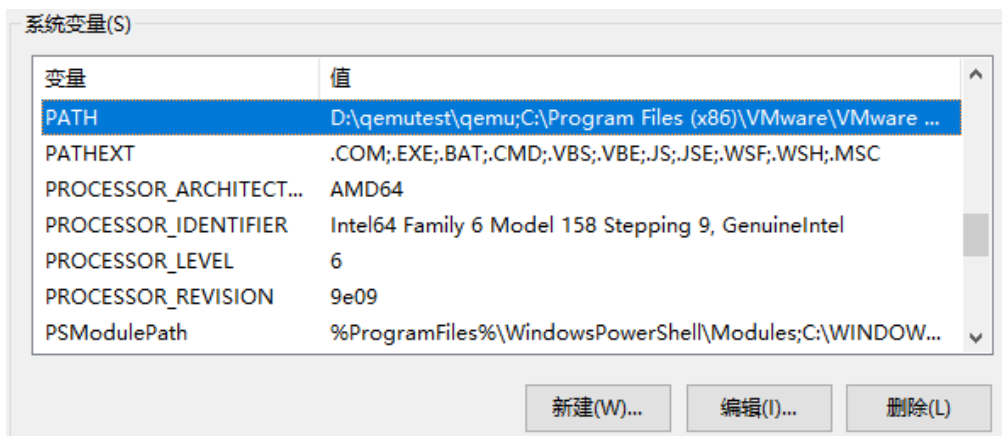


图1-4 找到系统变量中的 PATH

将之前 QEMU 的安装目录添加到 PATH 值中，如图 1-5 所示：

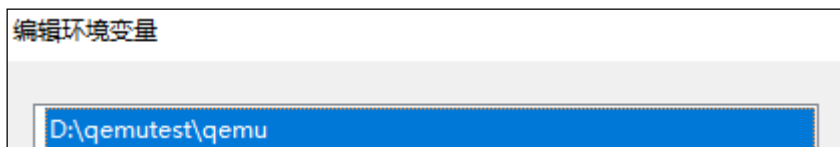


图1-5 添加 PATH 值

点击确定，保存并退出，重启计算机。

1.3.6 openEuler 操作系统安装

1.3.6.1 环境准备

在开始安装之前，我们要准备好 openEuler 镜像。进入 openEuler 开源社区下载 qcow2 镜像 (https://repo.openeuler.org/openEuler-20.03-LTS/virtual_machine_img/aarch64/) ， 如图 1-6 所示：

[openEuler-20.03-LTS.aarch64.qcow2.xz](#)

图1-6 下载 qcow2 镜像

下载完成后解压到 D 盘中的自定义文件夹中，进入到 QUMU 安装文件夹中，找到里面的 edk2-aarch64-code.fd。将这个文件拷贝至刚才 qcow2 镜像所在的同级目录中。

完成后的效果如图 1-7 所示：

名称	修改日期	类型	大小
edk2-aarch64-code.fd	2019/8/16 3:47	FD 文件	65,536 KB
openEuler-20.03-LTS.aarch64.qcow2	2021/1/28 12:46	QCOW2 文件	3,000,576...

图1-7 自定义文件夹的内容

1.3.6.2 openEuler 虚拟机创建

右击桌面左下角的“Windows”按钮，从其右键菜单中选择“搜索”项，搜索“cmd”，选择右侧的以管理员身份运行，如图 1-8 所示：

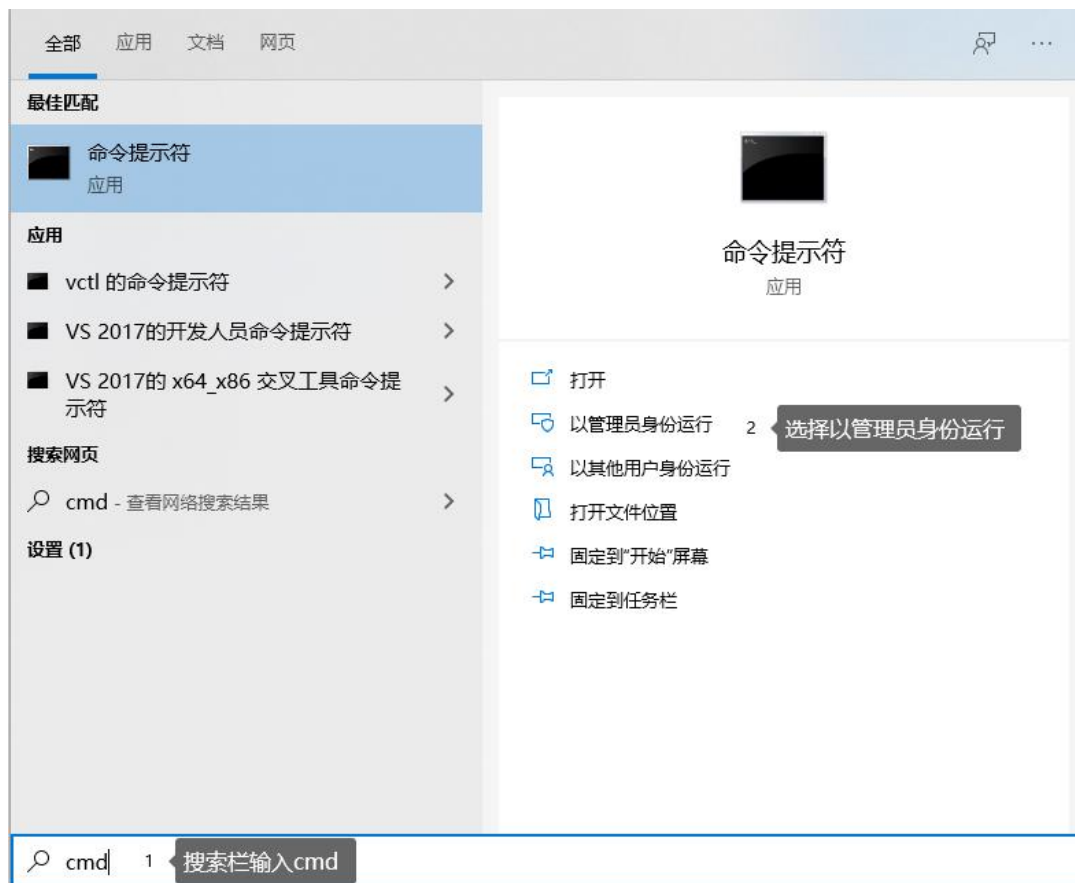


图1-8 以管理员身份运行 cmd

进入到刚才 qcow2 镜像所在的路径中。

输入以下命令：

```
qemu-system-aarch64 -m 4096 -cpu cortex-a57 -smp 4 -M virt -bios edk2-aarch64-code.fd -hda  
openEuler-20.03-  
LTS.aarch64.qcow2 -serial vc:800x600
```

(注意，上述串的中间不能有回车。可以先把上述串复制到记事本中，删掉中间的回车，再复制到命令行窗口中。可以用鼠标复制粘贴，或者 CTRL+INSERT 复制，SHIFT+INSERT 粘贴)

如图 1-9 所示：

```
D:\openEuler_test>qemu-system-aarch64 -m 4096 -cpu cortex-a57  
-smp 4 -M virt -bios edk2-aarch64-code.fd -hda openEuler-20.03  
-LTS.aarch64.qcow2 -serial vc:800x600_
```

图1-9 打开虚拟机

运行上述命令行后会出现一个窗口，鼠标移过去，选择第二个菜单“View”将串口改为 Serial0，如图 1-10 所示：

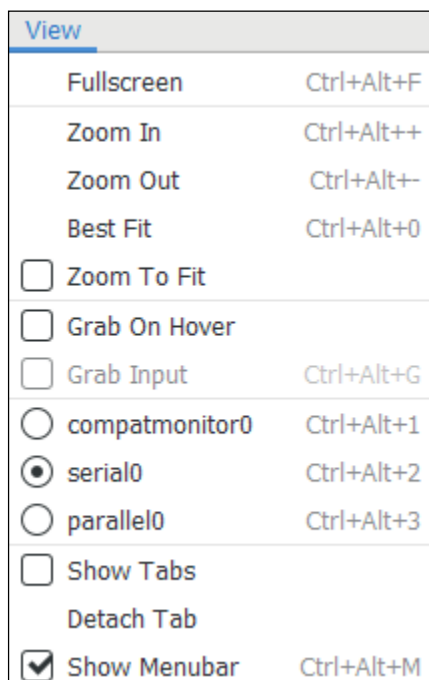


图1-10 修改串口为 serial0

等待片刻，出现登录提示，如图 1-11 所示：

```
Authorized users only. All activities may be monitored and reported.
localhost login: █
```

图1-11 用户登录

进行登录操作，其中用户名为 root，密码为 openEuler12#\$，如图 1-12 所示：

```
Authorized users only. All activities may be monitored and reported.
localhost login: root
Password:
Last login: Thu Jan 28 04:11:28 on ttyAMA0

Authorized users only. All activities may be monitored and reported.

Welcome to 4.19.90-2003.4.0.0036.oe1.aarch64

System information as of time: Wed Feb 3 16:12:17 UTC 2021

System load:    0.64
Processes:      86
Memory used:    4.6%
Swap used:      0.0%
Usage On:       6%
IP address:
Users online:   1
```

图1-12 登陆成功界面

至此，虚拟机安装完成，鲲鹏开发者环境也搭建成功。其中，操作系统可以换其他产品，支持 ARM 架构的都可以安装。

1.3.7 网络配置

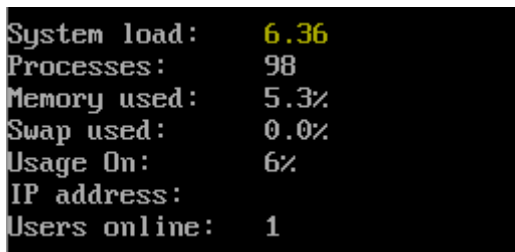
1.3.7.1 参数设置

我们采用最简单的方式配置网络。参数：-net nic, model=e1000 -net user-nic。这组参数可以同时配置网络前端和后端。（先执行 poweroff 退出系统，回到命令行窗口，再输入如下命令）

将这个网络参数加入到启动命令中：（以后每次启动虚拟机时，都使用该串命令）

```
qemu-system-aarch64 -m 4096 -cpu cortex-a57 -smp 4 -M virt -bios edk2-aarch64-code.fd -net  
nic,model=e1000 -net  
user -hda openEuler-20.03-LTS.aarch64.qcow2 -serial vc:800x600
```

登录到虚拟机中，如图 1-13 所示：



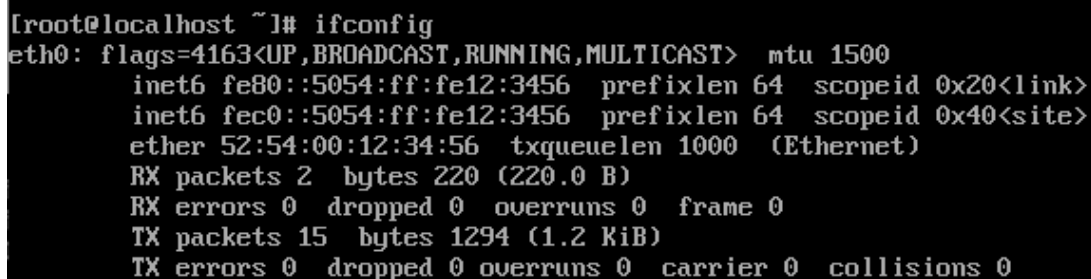
```
System load:    6.36  
Processes:     98  
Memory used:   5.3%  
Swap used:     0.0%  
Usage On:      6%  
IP address:  
Users online:  1
```

图1-13 登录成功后无 IP

可以发现此时没有 ip，我们还需要对网卡进行配置。

1.3.7.2 网卡及网络配置

查看你的网络信息：ifconfig，可以发现有个 eth0 网口，如图 1-14 所示：



```
[root@localhost ~]# ifconfig  
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
    inet6 fe80::5054:ff:fe12:3456 prefixlen 64 scopeid 0x20<link>  
    inet6 fec0::5054:ff:fe12:3456 prefixlen 64 scopeid 0x40<site>  
    ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)  
    RX packets 2  bytes 220 (220.0 B)  
    RX errors 0  dropped 0  overruns 0  frame 0  
    TX packets 15  bytes 1294 (1.2 KiB)  
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图1-14 查看网卡 MAC 地址

记录下来 eth0 中第四行 ether 后的一串字符：52:54:00:12:34:56。用 ethtool eth0 命令查看 eth0 网口信息，最后一行 Link detected: YES 说明网卡正常工作，如图 1-15 所示：

```
[root@localhost ~]# ethtool eth0
Settings for eth0:
    Supported ports: [ TP ]
    Supported link modes:   10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
                           1000baseT/Full

    Supported pause frame use: No
    Supports auto-negotiation: Yes
    Supported FEC modes: Not reported
    Advertised link modes:  10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
                           1000baseT/Full

    Advertised pause frame use: No
    Advertised auto-negotiation: Yes
    Advertised FEC modes: Not reported
    Speed: 1000Mb/s
    Duplex: Full
    Port: Twisted Pair
    PHYAD: 0
    Transceiver: internal
    Auto-negotiation: on
    MDI-X: off (auto)
    Supports Wake-on: umbg
    Wake-on: d
    Current message level: 0x00000007 (7)
                           drv probe link
    Link detected: yes
```

图1-15 查看网卡是否正常工作

确认无误后，输入：nmcli connection 来查看连接的设备信息，其中 eth0 口的 UUID 要记录下来，如图 1-16 所示：

```
[root@localhost ~]# nmcli connection
NAME        UUID                                     TYPE      DEVICE
System eth0 e449c48a-45a7-3db7-8cf3-349bd209b064 ethernet eth0
```

图1-16 记录网卡的 UUID

Linux 系统中，所有的网络接口配置文件都保持在/etc/sysconfig/network-scripts 目录中，进入到这个目录，如图 1-17 所示：

```
[root@localhost ~]# cd /etc/sysconfig/network-scripts/
[root@localhost network-scripts]#
```

图1-17 进入网络接口配置目录

如果目录下任何文件都没有，可以使用 vi 编辑器创建一个名为“ifcfg-eth0”的文件，如图 1-18 所示（文件创建之后的样子）：

```
[root@localhost network-scripts]# ls
ifcfg-eth0
```

图1-18 网卡配置文件

在“ifcfg-eth0”文件中写入以下内容：

```
TYPE=Ethernet #网卡类型
DEVICE=eth0 #网卡接口名称
ONBOOT=yes #系统启动时是否激活 yes | no
BOOTPROTO=dhcp #启用地址协议
HWADDR= 前面你的网卡 MAC 地址#网卡设备 MAC 地址
UUID=前面你自己的 UUID#网卡设备的 UUID
IPV6INIT=no
USERCTL=no
NM_CONTROLLED=yes
```

如图 1-19 所示：

```
[root@localhost network-scripts]# cat ifcfg-eth0
DEVICE=eth0
HWADDR=52:54:00:12:34:56
UUID=e449c48a-45a7-3db7-8cf3-349bd209b064
BOOTPROTO=dhcp
type=Ethernet
NM_CONTROLLED=yes
ONBOOT=yes
IPV6INIT=no
USERCTL=no
```

图1-19 网卡配置文件内容

保存并退出。

1.3.7.3 网络连接测试

打开网口，输入命令：ifup eth0，如图 1-20 所示：（重启时，若需使用网络，执行该命令）

```
[root@localhost network-scripts]# ifup eth0
Connection successfully activated (D-Bus active path:
```

图1-20 打开网口（如果报错，可以再次运行试试）

检测网络配置，输入命令：ifconfig，如图 1-21 所示：

```
[root@localhost ~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::5054:ff:fe12:3456 prefixlen 64 scopeid 0x20<link>
    inet6 fec0::5054:ff:fe12:3456 prefixlen 64 scopeid 0x40<site>
    ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)
    RX packets 76 bytes 16105 (15.7 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 123 bytes 11302 (11.0 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

图1-21 IP 配置成功

可以看到 eth0 网口已经有 IP 地址，证明网络配置成功。

1.3.7.4 yum 源配置

输入以下命令来查看 yum 源：

```
cd /etc/yum.repos.d/
cat openEuler_aarch64.repo
```

使用 vi 命令，在 openEuler_aarch64.repo 文件末尾处添加以下内容：

```
vi openEuler_aarch64.repo
```

```
[base]
name=openEuler20.03LTS
baseurl=https://repo.openeuler.org/openEuler-20.03-LTS/OS/aarch64/
enabled=1
gpgcheck=0
```

保存并退出，如图 1-22 所示：

```
[base]
name=openEuler20.03LTS
baseurl=https://repo.openeuler.org/openEuler-20.03-LTS/OS/aarch64/
enabled=1
gpgcheck=0
```

图1-22 设置yum 源

执行以下命令更新 yum 源：

```
yum makecache
```

如图 1-23 所示：

```
[root@localhost network-scripts]# yum makecache
openEuler20.03LTS 393 kB/s | 3.2 MB 00:08
Last metadata expiration check: 0:00:15 ago on Wed 27 Jan 2021 03:32:42 AM UTC.
Metadata cache created.
```

图1-23 更新yum 源

至此，yum 源配置已完成，可以从网络上下载工具了。

安装 C/C++语言编译器：

```
yum install gcc-c++ libstdc++-devel
```

如图 1-24 和 1-25 所示:

```
[root@localhost network-scripts]# yum install gcc gcc-c++ libstdc++-devel
Last metadata expiration check: 0:08:17 ago on Wed 27 Jan 2021 03:32:42 AM UTC.
Dependencies resolved.
=====
Package                Arch          Version                               Repo          Size
=====
Installing:
gcc                    aarch64      7.3.0-20190804.h31.oe1              base          9.7 M
gcc-c++                aarch64      7.3.0-20190804.h31.oe1              base          6.7 M
libstdc++-devel        aarch64      7.3.0-20190804.h31.oe1              base          1.1 M
Installing dependencies:
cpp                    aarch64      7.3.0-20190804.h31.oe1              base          6.0 M
glibc-devel            aarch64      2.28-36.oe1                         base          2.6 M
kernel-devel           aarch64      4.19.90-2003.4.0.0036.oe1           base          15 M
libmpc                 aarch64      1.1.0-3.oe1                         base          55 k
libxcrypt-devel        aarch64      4.4.8-4.oe1                         base          106 k
pkgconf                aarch64      1.6.3-6.oe1                         base          56 k

Transaction Summary
=====
Install 9 Packages

Total download size: 41 M
Installed size: 156 M
Is this ok [y/N]:
```

图1-24 编译器安装 (1) (输入 y)

```
Downloading Packages:
(1/9): cpp-7.3.0-20190804.h31.oe1.aarch64.rpm 788 kB/s | 6.0 MB 00:07
(2/9): gcc-c++-7.3.0-20190804.h31.oe1.aarch64.r 679 kB/s | 6.7 MB 00:10
(3/9): glibc-devel-2.28-36.oe1.aarch64.rpm 980 kB/s | 2.6 MB 00:02
(4/9): libmpc-1.1.0-3.oe1.aarch64.rpm 504 kB/s | 55 kB 00:00
(5/9): libstdc++-devel-7.3.0-20190804.h31.oe1.a 970 kB/s | 1.1 MB 00:01
(6/9): libxcrypt-devel-4.4.8-4.oe1.aarch64.rpm 687 kB/s | 106 kB 00:00
(7/9): pkgconf-1.6.3-6.oe1.aarch64.rpm 450 kB/s | 56 kB 00:00
(8/9): gcc-7.3.0-20190804.h31.oe1.aarch64.rpm 746 kB/s | 9.7 MB 00:13
(9/9): kernel-devel-4.19.90-2003.4.0.0036.oe1.a 1.0 MB/s | 15 MB 00:14
-----
Total 1.6 MB/s | 41 MB 00:25
Running transaction check
Transaction check succeeded.
Running transaction test
```

图1-25 编译器安装 (2)

安装完成后, 即可进行程序测试。

(注: 若下载安装过程中出现卡顿很久无反应的情况, 可以中止安装程序 “CTRL z”, 杀掉原来进程或重启后重新下载安装)

1.3.7.5 程序测试

用 C 语言编写 hello world 测试程序。

```
cat test.c
```

如图 1-26 所示:


```
[root@localhost ~]# cat test.c
#include<stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

图1-26 hello world 测试程序

编译运行。

```
gcc test.c -o hello
```

如图 1-27 所示：

```
[root@localhost ~]# gcc test.c -o hello
[root@localhost ~]# ./hello
Hello World!
[root@localhost ~]#
```

图1-27 hello world 运行

测试完成。至此，鲲鹏开发环境搭建完成，能够使用模拟器进行基于鲲鹏 920 处理器的程序开发与测试。

1.4 编程工具的其他说明

1.4.1 汇编 as 与链接 ld 命令

如果程序仅仅是由汇编语言源程序组成，不存在 C 语言的程序，则需要使用 as 命令进行汇编，再通过 ld 命令进行链接，例如有一个显示 Hello World! 的汇编语言程序：

```
vi hello.s
```

在 hello.s 中输入以下代码：

```
.text
.global tart1
tart1:
    mov x0,#0
    ldr x1,msg
    mov x2,len
    mov x8,64
    svc #0

    mov x0,123
    mov x8,93
    svc #0
```

```
.data
msg:
    .ascii "Hello World!\n"
len=.-msg
```

保存 hello.s 文件，然后通过运行以下命令将其编译为二进制目标文件

```
as hello.s -o hello.o
```

使用以下命令进行链接，输出可执行文件

```
ld hello.o -o hello
```

使用以下命令执行 hello 程序

```
./hello
```

1.4.2 调试工具 gdb

gdb 是 GNU 开源组织发布的 Linux 下的调试工具，它的功能十分强大，总的来说包含以下几个方面：

1. 按照用户自定义启动程序。
2. 可以使调试的程序在你设置的断点处停止。
3. 程序暂停时可以检查运行环境。
4. 程序暂停时可以动态改变运行环境。

gdb 调试的是可执行文件，要使用 gdb 调试程序，在使用 gcc 编译源文件时要加入 -g 选项。

步骤 1 安装 gdb

首先进行 gdb 的安装介绍，在进入命令行界面后，输入命令 yum install gdb 进行安装。

```
yum install gdb
```

如图 5-9 所示：

```
[root@ecs-001 ~]# yum install gdb
Last metadata expiration check: 0:34:34 ago on Wed 29 Dec 2021 10:22:16 AM CST.
Dependencies resolved.
=====
Package                                Architecture
=====
Installing:
gdb                                     aarch64
Installing dependencies:
babeltrace                             aarch64
gdb-headless                           aarch64
Transaction Summary
=====
Install 3 Packages
```

图1-28 安装 gdb

步骤 2 查看 gdb 版本

等待安装完成后输入命令 `gdb -v` 查看版本。

```
gdb -v
```

如图 5-10 所示：

```
[root@ecs-001 ~]# gdb -v
GNU gdb (GDB) EulerOS 8.3.1-11.oe1
Copyright (C) 2019 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.
```

图1-29 查看 gdb 版本

步骤 3 基本命令介绍

安装成功，接下来介绍一些有关 gdb 的基本命令：

<code>file</code>	//装入想要调试的可执行文件
<code>kill</code>	//终止正在调试的程序
<code>list</code> 或 <code>l</code>	//列出产生可执行文件的源代码的一部分
<code>next</code> 或 <code>n</code>	//执行一行源代码但不进入函数内部
<code>step</code> 或 <code>s</code>	//执行一行源代码而且进入函数内部
<code>run</code> 或 <code>r</code>	//执行当前被调试的程序
<code>quit</code> 或 <code>q</code>	//退出 gdb
<code>watch</code>	//监视一个变量的值而不管它何时被改变
<code>break</code> 或 <code>b</code>	//在代码里设置断点
<code>make</code>	//使不退出 gdb 就可以重新产生可执行文件
<code>shell</code>	//使不必离开 gdb 就能执行 shell 命令。

详细使用方法可以在网上查询，比如下列网址：

<http://www.4k8k.xyz/article/chexiuli0810/111747492>

2 示例程序

2.1 C 与汇编的混合编程实验原理

C 语言调用汇编有两个关键点——调用与传参。对于调用，我们需要在汇编程序中通过.global 定义一个全局函数，然后该函数就可以在 C 代码中通过 extend 关键字加以声明，使其能够在 C 代码中直接调用。

关于 C 与汇编的混合编程的参数传递，ARM64 提供了 31 个 64 位通用寄存器（X0-X30），对应低 32 位为 W0-W30，而 X31 为堆栈指针寄存器。各自的用途详见表 2-1。参数传递用到的是 x0~x7 这 8 个寄存器，若参数个数大于 8 个则需要使用堆栈来传递参数。

表2-1 ARM64 常用寄存器用途

寄存器	用途
x0~x7	传递参数和返回值，多余的参数用堆栈传递，64 位的返回结果保存在x0中。
X8	用于保存子程序的返回地址。
x9~x15	临时寄存器，也叫可变寄存器，无需保存。
x16~x17	子程序内部调用寄存器，使用时不需要保存，尽量不要使用。
x18	平台寄存器，它的使用与平台相关，尽量不要使用。
x19~x28	临时寄存器，子程序使用时必须保存。
x29	帧指针寄存器（FP），用于连接栈帧，使用时必须保存。
x30	链接寄存器（LR），用于保存子程序的返回地址。
x31	堆栈指针寄存器（SP），用于指向每个函数的栈顶。

2.1.1 C 语言调用汇编实现累加和求值

本示例实现的功能是：输入一个正整数，输出从 0 到该正整数的所有正整数的累加和，输入输出功能在 C 代码中实现，计算功能通过调用汇编函数实现。需要传入的参数是输入的正整数，汇编传出的参数为累加和，因此只用到一个 x0 寄存器即可实现参数传递功能。

步骤 1 创建 sum.c 文件

执行命令 vi sum.c 编写 C 程序，按“A”进入编辑模式后输入代码。

```
vim sum.c
```

编写内容如下：

```
#include <stdio.h>
extern int add(int num); //声明外部调用，函数名为 add。
int main()
{
    int i,sum;
    scanf("%d",&i); //输入初始正整数。
    sum=add(i); //调用汇编函数 add，返回值赋值给 sum。
    printf("sum=%d\n",sum); //将累加和输出。
    return 0;
}
```

编写完成后按“ESC”键进入命令行模式，输入“:wq”后回车保存并退出编辑。

步骤 2 创建 add.s 文件

执行 vi add.s 编写所调用的汇编代码，内容如下：

```
.global add //定义全局函数，函数名为 add。
add: //label:add
    ADD x1,x1,x0 //将 x0+x1 的值存入 x1 寄存器。
    SUB x0,x0,#1 //将 x0-1 的值存入 x0 寄存器。
    CMP x0,#0 //比较 x0 和 0 的大小。
    BNE add //若 x0 与 0 不相等，跳转到 add；x0=0 则继续执行。
    MOV x0,x1 //将 x1 的值存入 x0（需要 x0 来返回值）。
    RET
```

步骤 3 使用 gcc 编译生成可执行文件

GCC 是由 GNU 开发的编程语言译器。

GCC 最基本的语法是：gcc [filenames] [options]

其中[options]就是编译器所需要的参数，[filenames]给出相关的文件名称。

-c：只编译，不链接成为可执行文件，编译器只是由输入的.c 等源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output_filename：确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

-g: 产生符号调试工具（GNU 的 gdb）所必要的符号资讯，要想对源代码进行调试，我们就必须加入这个选项。

执行 `gcc sum.c add.s -o sum` 进行编译，然后执行 `./sum` 运行，输入 100，返回累加和 5050，如图 2-3 所示：

```
gcc sum.c add.s -o sum
./sum
```

```
[root@ecs-01 sum]# gcc sum.c add.s -o sum
[root@ecs-01 sum]# ./sum
100
sum=5050
[root@ecs-01 sum]#
```

图2-1 编译运行

编译成功，程序执行结果正确。

2.1.2 C 语言内嵌汇编

C 语言是无法完全代替汇编语言的，一方面是其效率比 C 要高，另一方面是某些特殊的指令在 C 语法中是没有等价的语法的。例如：操作某些特殊的 CPU 寄存器如状态寄存器、操作主板上的某些 I/O 端口或者对性能要求极其苛刻的场景等，我们都可以通过在 C 语言中内嵌汇编代码来满足要求。

在 C 语言代码中内嵌汇编语句的基本格式为：

```
__asm__ __volatile__ ("asm code"
: 输出操作数列表
: 输入操作数列表
: clobber 列表
);
```

说明：

1. `__asm__` 前后各两个下划线，并且两个下划线之间没有空格，用于声明这行代码是一个内嵌汇编表达式，是内嵌汇编代码时必不可少的关键字。
2. 关键字 `volatile` 前后各两个下划线，并且两个下划线之间没有空格。该关键字告诉编译器不要优化内嵌的汇编语句，如果想优化可以不加 `volatile`；在很多时候，如果不使用该关键字的话，汇编语句有可能被编译器修改而无法达到预期的执行效果。
3. 括号里面包含四个部分：汇编代码（asm code）、输出操作数列表（output）、输入操作数列表（input）和 clobber 列表（破坏描述符）。这四个部分之间用“:”隔开。其中，输入操作数列表部分和 clobber 列表部分是可选的，如果不使用 clobber 列表部分，则格式可以简化为：

```
__asm__ __volatile__ ("asm code":output: input);
```

如果不使用输入部分，则格式可以简化为：

```
__asm__ __volatile__ ("asm code":output::changed);
```

此时，即使输入部分为空，输出部分之后的“:”也是不能省略的。另外，输入部分和 clobber 列表部分是可选的，如果都为空，则格式可以简化为：

```
__asm__ __volatile__ ("asm code":output);
```

4. 括号之后要以“;”结尾。

以下示例程序实现的是计算累加和功能，与 2.4.1 中示例程序的功能相同，用到了 C 语言内嵌汇编的方法，汇编指令部分与 2.4.1 相同。

步骤 1 创建 builtin.c 文件

执行命令 vi builtin.c 编写 c 程序。

编写内容如下：

```
#include <stdio.h>
int main()
{
    int val;
    scanf("%d",&val);
    __asm__ __volatile__(
        "add:\n"
        "ADD x1,x1,x0\n"
        "SUB x0,x0,#1\n"
        "CMP x0,#0\n"
        "BNE add\n"
        "MOV x0,x1\n"
        : "=r"(val)
        //r 代表存放在某个通用寄存器中，即在汇编代码里用一个寄存器代替()部分
        //中定义的 c 变量即 val；=代表只写，即在汇编代码里只能改变 C 变量的
        //值，而不能取它的值。
        : "0"(val)
        //0 代表与第一个输出参数共用同一个寄存器。
        :
    );
    printf("sum is %d \n",val);
    return 0;
}
```

输入完成后保存并退出。

步骤 2 编译并运行可执行文件

输入命令 gcc builtin.c -o builtin 进行编译，编译成功后输入 ./builtin 执行程序，输入 100，返回累加和 5050。

```
gcc builtin.c -o builtin
./builtin
```

如图 2-8 所示：

```
[root@ecs-01 builtin]# gcc builtin.c -o builtin
[root@ecs-01 builtin]# ./builtin
100
sum is 5050
[root@ecs-01 builtin]#
```

图2-2 执行 builtin

编译成功，程序执行结果正确。

2.2 内存拷贝及优化实验原理

优化效果通过计算对应代码段的执行时间来判断。具体方案是通过 C 语言调用汇编，在 C 代码中计算时间，在汇编代码中设计不同的方案，对比每种方案的执行时间，判断优化效果。本示例的优化针对内存读写，示例程序功能是内存拷贝，拷贝功能在汇编函数中实现。

说明：在使用 ldrb/ldp 和 str/stp 等访存指令时，要注意区分这三种形式：

1. 前索引方式，形如：ldrb w2,[X1,#1] //将 x1+1 指向的地址处的一个字节放入 w2 中，x1 寄存器的值保持不变。
2. 自动索引方式，形如：ldrb w2,[X1,#1]! //将 x1+1 指向的地址处的一个字节放入 w2 中，然后 x1+1 → x1。
3. 后索引方式，形如 ldrb w2,[X1],#1 //将 x1 指向的地址处的一个字节放入 w2 中，然后 x1+1 → x1。

2.2.1 基础代码

本程序由两部分组成：

第一部分是主函数，采用 Linux C 语言编码，用来测试内存拷贝函数的执行时间；

第二部分是内存拷贝函数，采用 GNU ARM64 汇编语言编码。为了较为准确的测量内存拷贝函数 memorycopy() 的执行时间，调用了 clock_gettime() 来分别记录 memorycopy() 执行前和执行后的系统时间，以纳秒为计时单位。

步骤 1 创建 time.c 文件

执行 vi time.c 编写 c 语言计时程序。

通过 clock_gettime 函数来计算时间差，从而得出所求代码的执行时间，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define len 60000000 //内存拷贝长度为 60000000
char src[len],dst[len]; //源地址与目的地址
long int len1=len;
```



```
extern void memcpy(char *dst,char *src,long int len); //声明外部函数
int main()
{
    struct timespec t1,t2; //定义初始与结束时间
    int i,j;
    //为初始地址段赋值，以便后续从该地址段读取数据拷贝
    for(i=0;i<len-1;i++)
    {
        src[i]='a';
    }
    src[i]=0;
    clock_gettime(CLOCK_MONOTONIC,&t1); //计算开始时间。
    memcpy(dst,src,len); //汇编调用，执行相应代码段。
    clock_gettime(CLOCK_MONOTONIC,&t2); //计算结束时间。
    //得出目标代码段的执行时间。
    printf("memcpy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
    return 0;
}
```

内存拷贝函数 memcpy()的功能是实现将尺寸为 len（这里设置为 60000000）的 src 字符数组的内容拷贝到同样尺寸的 dst 字符数组中。memcpy()函数用 AArch64 汇编代码实现。

在本例中，需要传递的参数有三个：

第一个参数是目标字符串的首地址，用寄存器 x0 来传递；

第二个参数是源字符串的首地址，用寄存器 x1 来传递；

第三个参数是传输的字节数目，用寄存器 x2 来传递。

步骤 2 创建 copy.s 文件

执行 vi copy.s 编写优化前的原始汇编代码。

代码如下：

```
.global memcpy //声明全局函数
memcpy:
    ldrb w3,[x1],#1 //从源字符串地址中读取
    str w3,[x0],#1 //向目的字符串地址中写
    sub x2,x2,#1
    cmp x2,#0 //判断是否结束
    bne memcpy
    ret
```

步骤 3 编译并运行可执行文件

执行 gcc time.c copy.s -o m1 完成编译，并执行。

```
gcc time.c copy.s -o m1
./m1
```

结果如下，可以看到 memcpy 函数具体的执行时间为 47514738ns，如图 3-3 所示：

```
[root@ecs-01 memory]# gcc time.c copy.s -o m1
[root@ecs-01 memory]# ./m1
memcpy time is 47514738 ns
```

图2-3 原始程序执行时间

接下来我们基于原始代码进行修改，观察他们的执行时间进行对比。

2.2.2 循环展开优化

循环展开是最常见的代码优化思路，通过减少指令总数来实现代码优化。

步骤 1 创建 2 倍展开优化 copy121.s 文件

先将 copy.s 展开两倍,命名为 copy121.s，代码如下：

```
.global memcpy
memcpy:
sub x1,x1,#1 //传进去的地址首地址为 0，减 1 是移动到 0 前面的地址-1
sub x0,x0,#1
lp:
ldrb w3,[x1,#1]! //将地址+1 后就移动到了首地址 0
ldrb w4,[x1,#1]! //一次循环读两个字节
str w3,[x0,#1]!
str w4,[x0,#1]! //一次循环写两个字节
sub x2,x2,#2
cmp x2,#0
bne lp
ret
```

步骤 2 编译并运行执行文件

执行命令 gcc time.c copy121.s -o m121 编译并运行程序。

```
gcc time.c copy121.s -o m121
./m121
```

在进行了循环展开后，这次 memcpy 函数的执行时间变为了 36851856 ns，如图 3-5 所示：

```
[root@Malluma memory]# gcc time.c copy121.s -o m121
[root@Malluma memory]# ./m121
memcpy time is 36851856 ns
```

图2-4 编译执行程序 m121

步骤 3 创建 4 倍展开优化 copy122.s 文件

然后输入以下代码，并将其编译，得到 m122，具体代码如下：

```
.global memcpy
memcpy:
sub x1,x1,#1
```

```
        sub x0,x0,#1
lp:      ldrb w3,[x1,#1]!
        ldrb w4,[x1,#1]!
        ldrb w5,[x1,#1]!
        ldrb w6,[x1,#1]!
        str w3,[x0,#1]!
        str w4,[x0,#1]!
        str w5,[x0,#1]!
        str w6,[x0,#1]!
        sub x2,x2,#4
        cmp x2,#0
        bne lp
ret
```

步骤 4 编译并运行可执行文件

输入命令 `gcc time.c copy122.s -o m122` 执行程序。

```
gcc time.c copy122.s -o m122
./m122
```

结果为 33292783ns，如图 3-6 所示：

```
[root@Malluma memory]# gcc time.c copy122.s -o m122
[root@Malluma memory]# ./m122
memorycopy time is 33292783 ns
```

图2-5 编译执行程序 m122

函数执行时间基于二倍的基础上也得到了优化，那么目前根据实验现象可以看出，更多次数的循环展开可能会让程序的执行时间得到更好的优化。

2.2.3 内存突发传输方式优化

之前两次优化每次内存读写都是以一个字节为单位进行的，这样效率很低。由于内存在连续读/写多个数据时，其性能要优于非连续读/写数据的方式，此次优化思路是一次对多个字节进行读写。这就用到了 `ldp` 指令和 `stp` 指令，这两条指令可以一次访问 16 个字节的内存数据，大大提高了内存读写效率。

步骤 1 创建内存突发传输优化 copy21.s 文件

```
.global memorycopy
memorycopy:
ldp x3,x4,[x1],#16 //ldp 指令将 x1 向上加 16 个字节后存放到 x3 和 x4 中
stp x3,x4,[x0],#16
sub x2,x2,#16
cmp x2,#0
bne memorycopy
ret
```

步骤 2 编译并运行可执行文件

编译执行 `gcc time.c copy21.s -o m21` 和 `./m21` 命令。

```
gcc time.c copy21.s -o m21
./m21
```

如图 3-16 所示：

```
[root@ecs-01 memory]# gcc time.c copy21.s -o m21
[root@ecs-01 memory]# ./m21
memorycopy time is 12785612 ns
```

图2-6 执行 m21 程序

一次对 16 字节读写程序执行效率明显优于单字节读写。

3

附录 1: Linux 常用命令

3.1 基本命令

3.1.1 关机和重启

命令:

shutdown -h now	#立刻关机
shutdown -h 5	#5 分钟后关机
poweroff	#立刻关机

重启

shutdown -r now	#立刻重启
shutdown -r 5	#5 分钟后重启
reboot	#立刻重启

3.1.2 帮助命令

命令: --help

shutdown -help	#查看关机命令帮助信息
ifconfig --help	#查看网卡信息
man	# (命令说明书)
man shutdown	

注意: man shutdown 打开命令说明书之后, 使用按键 q 退出

3.2 2 目录操作命令

3.2.1 目录切换命令

命令: cd 目录

cd /	#切换到根目录
cd /usr	#切换到根目录下的 usr 目录
cd ../	#切换到上一级目录 或者 cd ..
cd ~	#切换到 home 目录
cd -	#切换到上次访问的目录

3.2.2 目录查看命令

命令：ls [-al]

ls	#查看当前目录下的所有目录和文件
ls -a	#查看当前目录下的所有目录和文件（包括隐藏的文件）
ls -l 或 ll	#列表查看当前目录下的所有目录和文件（显示更多信息）
ls /dir	#查看指定目录下的所有目录和文件 如：ls /usr

3.2.3 目录操作命令

3.2.3.1 创建目录

命令：mkdir 目录

mkdir	aaa	# 在当前目录下创建一个名为 aaa 的目录
mkdir	/usr/aaa	# 在指定目录下创建一个名为 aaa 的目录

3.2.3.2 删除目录或文件

命令：rm [-rf] 目录

删除文件：

rm 文件	#删除当前目录下的文件
rm -f 文件	#删除当前目录下的文件（不询问）
#删除目录：	
rm -r aaa	#递归删除当前目录下的 aaa 目录
rm -rf aaa	#递归删除当前目录下的 aaa 目录（不询问）
#全部删除：	
rm -rf *	#将当前目录下的所有目录和文件全部删除
rm -rf /*	#【慎用！】将根目录下的所有文件全部删除

注意：rm 不仅可以删除目录，也可以删除其他文件或压缩包，为了方便大家的记忆，无论删除任何目录或文件，都直接使用 rm -rf 目录/文件/压缩包

3.2.3.3 目录修改

重命名目录

命令：mv 当前目录 新目录

示例：mv aaa bbb #将目录 aaa 改为 bbb

注意：mv 的语法不仅可以对目录进行重命名而且也可以对各种文件，压缩包等进行重命名的操作。

剪切目录

命令：mv 目录名称 目录的新位置

示例：mv /usr/tmp/aaa /usr #将/usr/tmp 目录下的 aaa 目录剪切到 /usr 目录下面

注意：mv 语法不仅可以对目录进行剪切操作，对文件和压缩包等都可执行剪切操作。

拷贝目录

```
命令: cp -r 目录名称 目录拷贝的目标位置 -r 代表递归
```

示例: `cp /usr/tmp/aaa /usr` #将/usr/tmp 目录下的 aaa 目录复制到 /usr 目录下面

注意: cp 命令不仅可以拷贝目录还可以拷贝文件, 压缩包等, 拷贝文件和压缩包时不用写-r 递归。

3.2.3.4 目录搜索

```
命令: find 目录 参数 文件名称
```

示例: `find /usr/tmp -name 'a*'` #查找/usr/tmp 目录下的所有以 a 开头的目录或文件

3.3 文件操作命令

3.3.1 新建文件

```
命令: touch 文件名
```

示例: `touch aa.txt` #在当前目录创建一个名为 aa.txt 的文件

3.3.2 删除文件

```
命令: rm -rf 文件名
```

3.3.3 修改文件

打开文件

```
vi 文件名
```

示例: `vi aa.txt` 或者 `vim aa.txt` #打开当前目录下的 aa.txt 文件

若文件不存在则新建文件并打开

注意: 使用 vi 编辑器打开文件后, 并不能编辑, 因为此时处于命令模式, 点击键盘 i/a/o 进入编辑模式。

- 编辑文件

使用 vi 编辑器打开文件后点击按键: i , a 或者 o 即可进入编辑模式。

i: 在光标所在字符前开始插入

a: 在光标所在字符后开始插入

o: 在光标所在行的下面另起一新行插入

- 保存文件:

第一步: ESC 进入命令行模式

第二步: 进入底行模式

第三步: `wq` #保存并退出编辑

- 取消编辑：

第一步：ESC 进入命令行模式

第二步：: 进入底行模式

第三步：q! #撤销本次修改并退出编辑

3.3.4 查看文件

文件的查看命令：cat/more/less/tail

cat: 看最后一屏

示例：使用 cat 查看/etc/sudo.conf 文件，只能显示最后一屏内容。

cat sudo.conf

more: 百分比显示

示例：使用 more 查看/etc/sudo.conf 文件，可以显示百分比，回车可以向下一行，空格可以向下一页，q 可以退出查看

more sudo.conf

less: 翻页查看

示例：使用 less 查看/etc/sudo.conf 文件，可以使用键盘上的 PgUp 和 PgDn 向上和向下翻页，q 结束查看

less sudo.conf

tail: 指定行数或者动态查看

示例：使用 tail -10 查看/etc/sudo.conf 文件的后 10 行，Ctrl+C 结束

tail -10 sudo.conf

4

附录 2: ARM 指令

4.1 LDR 字数据加载指令

(注: ARMv8-A 处理器体系结构的执行状态有两种: AArch32 状态和 AArch64 状态, 用的寄存器名称不同。AArch32 状态下 32 位通用寄存器是 R0-R12, AArch64 状态下 64 位通用寄存器用 X0-X30(对应低 32 位 W0-W30))

LDR 指令的格式为:

LDR{条件} 目的寄存器, <存储器地址>

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取 32 位的字数据到通用寄存器, 然后对数据进行处理。当程序计数器 PC 作为目的寄存器时, 指令从存储器中读取的字数据被当作目的地址, 从而可以实现程序流程的跳转。

指令示例:

LDR R0, [R1] ; 将存储器地址为 R1 的字数据读入寄存器 R0。

LDR R0, [R1, R2] ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0。

LDR R0, [R1, # 8] ; 将存储器地址为 R1+8 的字数据读入寄存器 R0。

LDR R0, [R1, R2] ! ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0, 并将新地址 R1 + R2 写入 R1。

LDR R0, [R1, # 8] ! ; 将存储器地址为 R1+8 的字数据读入寄存器 R0, 并将新地址 R1 + 8 写入 R1。

LDR R0, [R1], R2 ; 将存储器地址为 R1 的字数据读入寄存器 R0, 并将新地址 R1 + R2 写入 R1。

LDR R0, [R1, R2, LSL # 2] ! ; 将存储器地址为 R1 + R2×4 的字数据读入寄存器 R0, 并将新地址 R1 + R2×4 写入 R1。

LDR R0, [R1], R2, LSL # 2 ; 将存储器地址为 R1 的字数据读入寄存器 R0, 并将新地址 R1 + R2×4 写入 R1。

4.2 LDRB 字节数据加载指令

LDRB 指令的格式为:

LDR{条件}B 目的寄存器, <存储器地址>

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中, 同时将寄存器的高 24 位清零。该指令通常用于从存储器中读取 8 位的字节数据到通用寄存器, 然后对数据进行处理。当

程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

LDRB R0, [R1] ; 将存储器地址为 R1 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零。

LDRB R0, [R1, #8]! ; 将存储器地址为 R1 + 8 的字节数据读入寄存器 R0，并将新地址 R1 + 8 写入 R1。

4.3 LDRH 半字数据加载指令

LDRH 指令的格式为：

LDR{条件}H 目的寄存器, <存储器地址>

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中，同时将寄存器的高 16 位清零。该指令通常用于从存储器中读取 16 位的半字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而实现程序流程的跳转。

指令示例：

LDRH R0, [R1] ; 将存储器地址为 R1 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

LDRH R0, [R1, #8] ; 将存储器地址为 R1 + 8 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

LDRH R0, [R1, R2] ; 将存储器地址为 R1 + R2 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。

4.4 STR 字数据存储指令

STR 指令的格式为：

STR{条件} 源寄存器, <存储器地址>

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。该指令在程序设计中比较常用，且寻址方式灵活多样，使用方式可参考指令 LDR。

指令示例：

STR R0, [R1], #8 ; 将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8 写入 R1。

STR R0, [R1, #8] ; 将 R0 中的字数据写入以 R1 + 8 为地址的存储器中。

STR R0, [R1, #8]! ; 将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1 + 8 写入 R1。

4.5 STRB 字节数据存储指令

STRB 指令的格式为：

STR{条件}B 源寄存器, <存储器地址>

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。该字节数据为源寄存器中的低 8 位。

指令示例:

STRB R0, [R1] ; 将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中。

STRB R0, [R1, #8] ; 将寄存器 R0 中的字节数据写入以 R1 + 8 为地址的存储器中。

4.6 STRH 半字数据存储指令

STRH 指令的格式为:

STR{条件}H 源寄存器, <存储器地址>

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。该半字数据为源寄存器中的低 16 位。

指令示例:

STRH R0, [R1] ; 将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中。

STRH R0, [R1, #8] ; 将寄存器 R0 中的半字数据写入以 R1 + 8 为地址的存储器中。

4.7 LDP/STP 指令

是 LDP/STP 的衍生, 可以同时读/写两个寄存器, 并访问 16 个字节的内存数据,

指令示例:

LDP x3,x4,[x1,#16] ; 读取 x1+16 地址后的 16 个字节的数据写入 x3、x4 寄存器中。

LDP x3,x4,[x1],#16 ; 读取 x1 地址后的 16 个字节的数据写入 x3、x4 寄存器中, 并更新 x1=x1+16。

LDP x9,x10,[x1,#64]! ; 读取 x1+64 地址后的 16 个字节的数据写入 x9、x10 寄存器中。并将新地址 x1 + 64 写入 x1。

STP x3,x4,[x0,#16] ; 将 x3、x4 中的数据写入以 x0+16 地址后的 16 个字节地址中

STP x3,x4,[x0],#16 ; 将 x3、x4 中的数据写入以 x0 地址后的 16 个字节地址中并更新 x0=x0+16。

STP x9,x10,[x0,#64]! ; 将 x9、x10 中的数据写入以 x0+64 地址后的 16 个字节地址中, 并将新地址 x0 + 64 写入 x0。