

计算机组成与结构课程

实验指导手册

版本:2.0

大连理工大学 赖晓晨



华为技术有限公司

目录

前 言	6
简介	6
内容描述	6
读者知识背景	7
实验环境说明	7
1 基于 QEMU 模拟器的鲲鹏 920 处理器开发环境搭建	8
1.1 实验目的	8
1.2 实验设备	8
1.3 实验原理	8
1.3.1 QEMU 简介	8
1.3.2 QEMU 的优缺点	9
1.3.3 openEuler 操作系统	9
1.3.4 openEuler 社区	9
1.4 实验任务操作指导	10
1.4.1 QEMU 的安装配置	10
1.4.2 openEuler 操作系统安装	12
1.4.3 网络配置	15
2 C 语言与鲲鹏 920 处理器汇编语言混合编程	21
2.1 实验目的	21
2.2 实验设备	21
2.3 实验原理	21
2.4 实验任务操作指导	22
2.4.1 C 语言调用汇编实现累加和求值	22
2.4.2 C 语言调用汇编实现选择排序	25
2.4.3 C 语言内嵌汇编	28
3 鲲鹏 920 处理器的汇编代码优化	31
3.1 实验目的	31
3.2 实验设备	31
3.3 实验原理	31

3.4 实验任务操作指导	32
3.4.1 基础代码	32
3.4.2 循环展开优化	34
3.4.3 鲲鹏处理器流水线优化	36
3.4.4 内存突发传输方式优化	38
3.5 总结	39
4 鲲鹏 920 处理器增强型 SIMD 运算	40
4.1 实验目的	40
4.2 实验设备	40
4.3 实验原理	40
4.4 实验任务操作指导	41
4.4.1 基础运算	41
4.4.2 增强型 SIMD	43
5 鲲鹏 920 处理器 ARM 异常中断实验	47
5.1 实验目的	47
5.2 实验设备	47
5.3 实验原理	47
5.3.1 ARM 的异常中断响应过程	47
5.3.2 异常中断的返回	47
5.3.3 IRQ/FIQ 中断处理机制	48
5.3.4 IRQ 和 FIQ 异常中断处理程序的返回	49
5.4 实验任务操作指导	50
5.4.1 svc 的简单使用	50
5.4.2 core dump 案例介绍	52
6 鲲鹏 920 处理器利用存储器层次结构的程序优化实验	59
6.1 实验目的	59
6.2 实验设备	59
6.3 实验原理	60
6.4 实验任务操作指导	61
6.4.1 标准矩阵乘法	61
6.4.2 矩阵乘法优化	70

7 鲲鹏 920 处理器利用 CPU 任务并行的程序优化实验	78
7.1 实验目的	78
7.2 实验设备	78
7.3 实验原理	78
7.4 实验任务操作指导	78
7.4.1 单线程执行矩阵乘法运算	78
7.4.2 shell 中实现并行任务	81
7.4.3 多线程实现排序	84
8 大作业：鲲鹏代码迁移实验	97
8.1 实验目的	97
8.2 实验设备	97
8.3 实验原理	98
8.3.1 主流架构对比	98
8.3.2 程序的生命周期	98
8.3.3 程序运行	99
8.4 实验任务操作指导	101
8.4.1 x86 冒泡排序代码迁移	101
8.4.2 ARM 汇编实现冒泡排序	111
8.4.3 迁移分析	113
9 附录 1：Linux 常用命令	115
9.1 基本命令	115
9.1.1 关机和重启	115
9.1.2 帮助命令	116
9.2 目录操作命令	116
9.2.1 目录切换命令	116
9.2.2 目录查看命令	116
9.2.3 目录操作命令	116
9.3 文件操作命令	118
9.3.1 新建文件	118
9.3.2 删除文件	118
9.3.3 修改文件	118

9.3.4 查看文件	118
10 附录 2: ARM 指令.....	119
10.1 LDR 字数据加载指令	119
10.2 LDRB 字节数据加载指令	120
10.3 LDRH 半字数据加载指令	120
10.4 STR 字数据存储指令	121
10.5 STRB 字节数据存储指令	121
10.6 STRH 半字数据存储指令	122
10.7 LDP/STP 指令	122

前言

简介

本实验指导手册为基于鲲鹏 920 处理器的《计算机组成与结构》课程的实验指导，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、中断技术、代码迁移等相关技术的读者。

内容描述

本实验指导手册共包含 8 个实验，从本地虚拟机环境搭建开始，逐一介绍了 C 与汇编混合编程、汇编代码优化、SIMD 增强型运算、异常中断、程序优化、代码迁移等。

- 实验一为基于 QEMU 模拟器的鲲鹏 920 处理器开发环境搭建，通过基本的操作与配置，帮助读者在 x86 系统上搭建出兼容 ARM v8 指令集的模拟环境，为鲲鹏处理器的学习提供实验环境。
- 实验二为 C 语言与鲲鹏 920 处理器汇编语言混合编程，重点讲解了 C 语言调用汇编语言和 C 语言内嵌汇编语言操作，通过本实验，读者可以掌握 C 语言和汇编语言混合编程的方法。
- 实验三为鲲鹏 920 处理器的汇编代码优化，通过实验提供的递进的优化思路，对比优化效果，读者可以了解基于 ARM 架构硬件特性的优化方案。
- 实验四为鲲鹏 920 处理器增强型 SIMD 运算，本实验可以使读者掌握 SIMD 优化的原理和方法，了解 SIMD 在数据密集型计算领域的优势。
- 实验五为鲲鹏 920 处理器 ARM 异常中断实验，通过介绍中断机制以及调用中断指令，帮助读者掌握 ARM 软中断指令 SVC 的使用。
- 实验六为鲲鹏 920 处理器利用存储器层次结构的程序优化实验，通过一个 500*500 的矩阵乘法优化案例，并使用鲲鹏性能分析工具 Hyper-Tuner 进行分析，帮助读者掌握利用存储器层次结构进行程序优化的原理和方法。
- 实验七为鲲鹏 920 处理器利用 CPU 任务并行的程序优化实验，通过多线程案例程序，帮助读者了解 CPU 多线程技术以及利用 CPU 的任务并行方式进行程序优化的方法。
- 实验八为鲲鹏代码迁移实验，通过代码迁移实验案例，帮助读者掌握使用鲲鹏代码迁移工具将 x86 平台上的源代码迁移到基于鲲鹏 916/920 处理器平台的方法。

读者知识背景

本课程为计算机组成与结构基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具备基本的 Linux 命令能力；
- 具备基本的汇编语言编码能力；
- 具备基本的 C 语言编码能力。

实验环境说明

- 华为鲲鹏云主机、openEuler 20.03 操作系统；
- 安装鲲鹏性能分析工具；
- 安装鲲鹏代码迁移工具；
- 每套实验环境可供 1 名学员上机操作；
- 实验环境搭建请参考《实验环境搭建手册》，本手册中实验 2 至实验 8，都是在华为云鲲鹏 ECS 服务器上。

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

- 具备网络连接的个人电脑。

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。

1.4 实验任务操作指导

1.4.1 QEMU 的安装配置

1.4.1.1 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。

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：

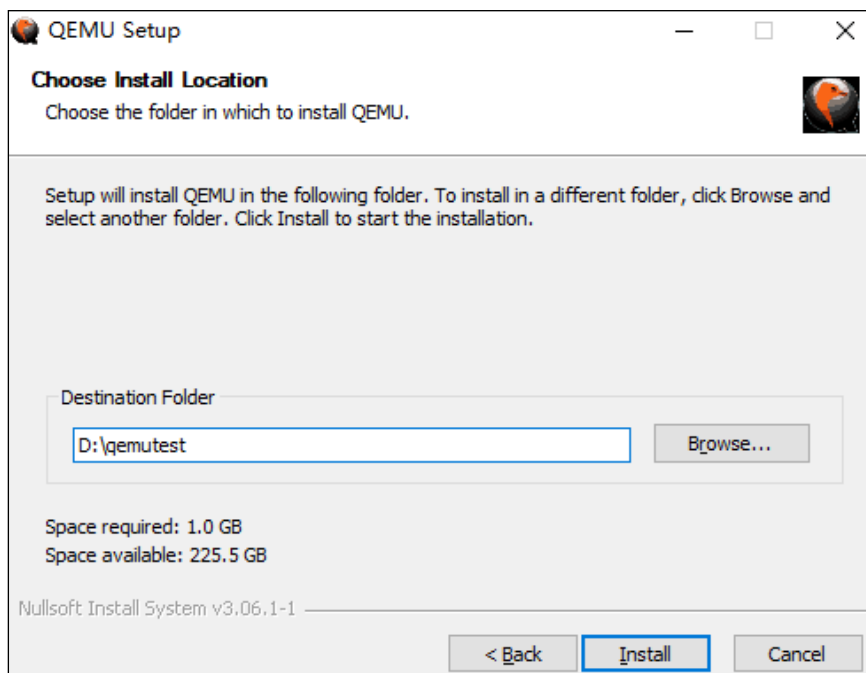


图1-1 安装路径选择

1.4.1.2 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：

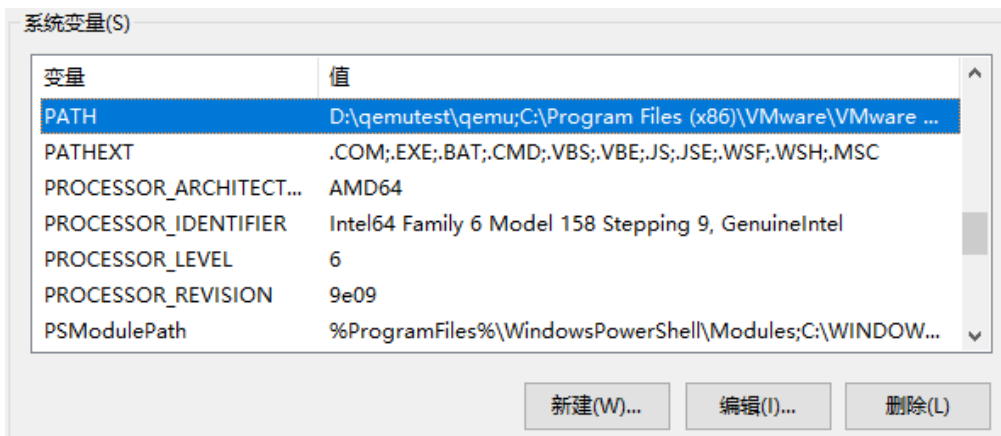


图1-4 找到系统变量中的 PATH

将之前 QEMU 的安装目录添加到 PATH 值中，如图 1-5 所示：

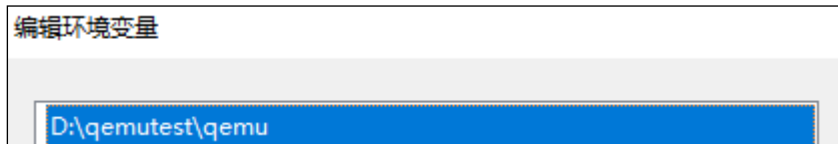


图1-5 添加 PATH 值

点击确定，保存并退出，重启计算机。

1.4.2 openEuler 操作系统安装

1.4.2.1 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](https://repo.openeuler.org/openEuler-20.03-LTS/virtual_machine_img/aarch64/)

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 QEMU 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

1.4.2.2 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”，选择左侧的以管理员身份运行，如图 1-8 所示：

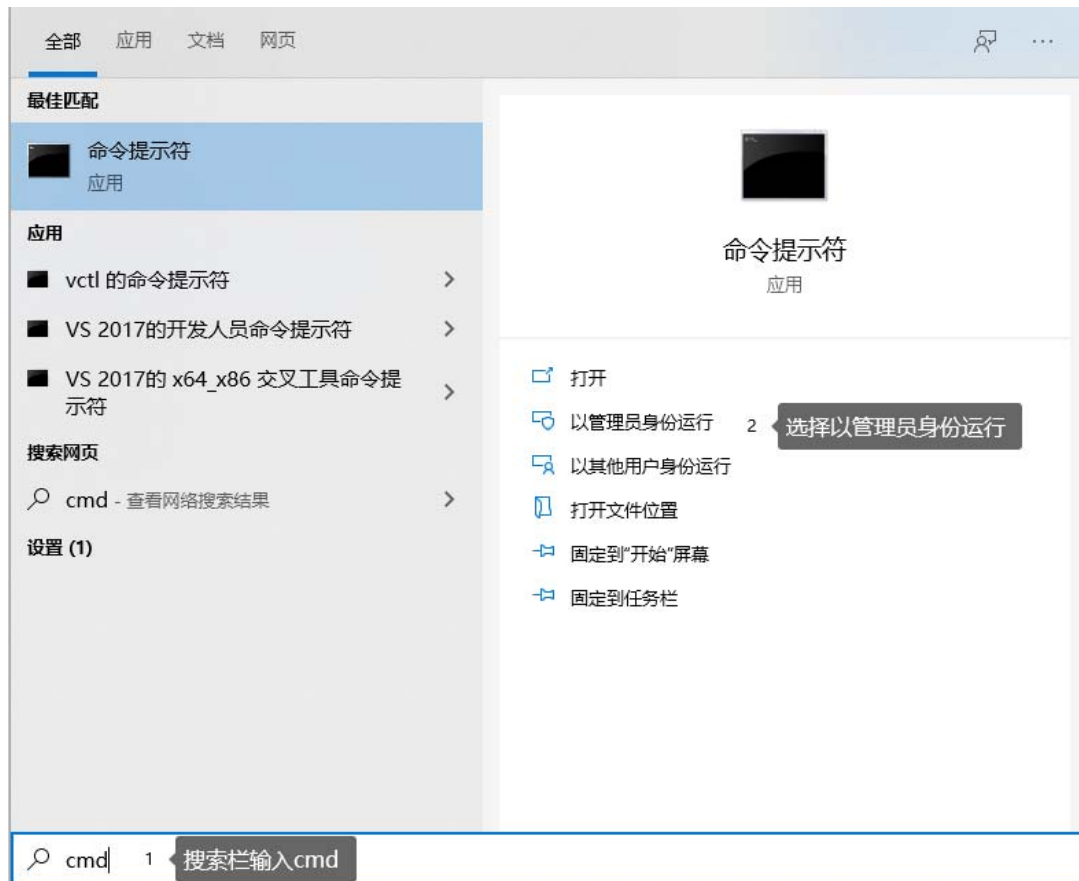


图1-8 以管理员身份运行 cmd

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
```

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

选择 View 将串口改为 Serial0，如图 1-10 所示：

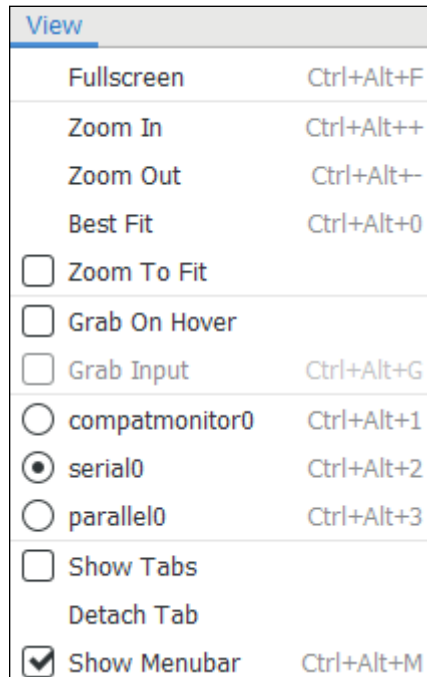


图1-10 修改串口为 serial0

等待片刻，出现登录提示，如图 1-11 所示：

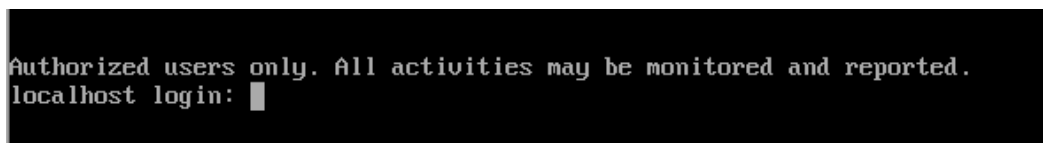


图1-11 用户登录

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

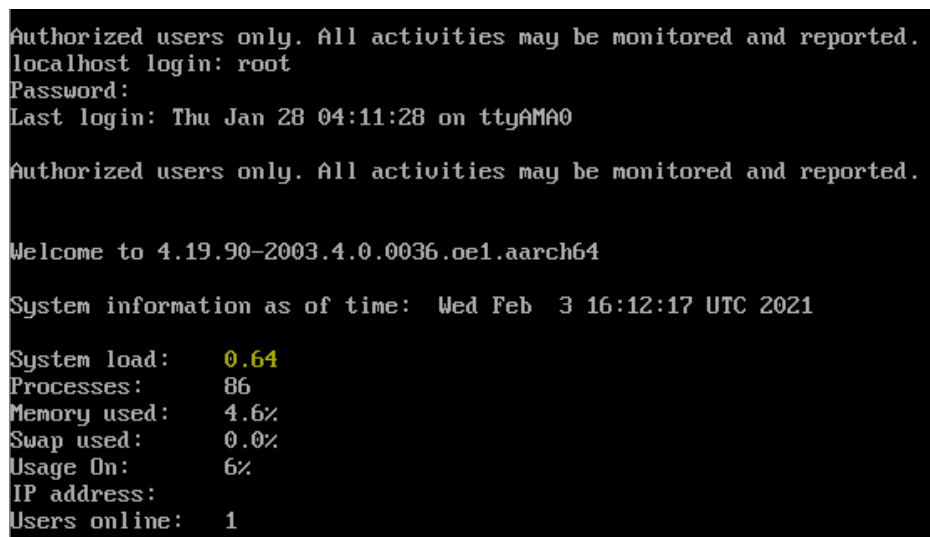


图1-12 登陆成功界面

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

1.4.3 网络配置

1.4.3.1 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。这组参数可以同时配置网络前端和后端。

将这个网络参数加入到启动命令中：

```
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。

1.4.3.2 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：

```
[root@localhost network-scripts]# ls
[root@localhost network-scripts]# vi 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：

```
TYPE=Ethernet
DEVICE=eth0
ONBOOT=yes
BOOTPROTO=dhcp
HWADDR=52:54:00:12:34:56
UUID=e449c48a-45a7-3db7-8cf3-349bd209b064
IPV6INIT=no
USERCTL=no
NM_CONTROLLED=yes
~
~
```

图1-19 网卡配置文件内容

保存并退出。

1.4.3.3 网络连接测试

打开网口，输入命令：ifup eth0，如图 1-20 所示：

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

图1-20 打开网口

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。

1.4.3.4 yum 源配置

为了从网络上下载实验需要的工具，例如 C/C++语言编译器，我们需要配置 yum 源。yum，全称“Yellow dog Updater, Modified”，是一个专门为了解决包的依赖关系而存在的软件包管理器。就好像 Windows 系统上可以通过软件商店实现软件的一键安装、升级和卸载，Linux 系统也提供有这样的工具，就是 yum。yum 源指的就是软件安装包的来源。

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 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)

```
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。

1.4.3.5 程序测试

用 C 语言编写 hello world 测试程序。

```
vim test.c
```

如图 1-26 所示:

```
#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。

2 C 语言与鲲鹏 920 处理器汇编语言混合编程

2.1 实验目的

本实验将通过三个部分介绍 C 调用汇编和 C 内嵌汇编两种混合编程方式以及 ARM 汇编的一些基础指令，ARM 部分指令的详细介绍以及 Linux 常用命令请参考附录中的 ARM 指令以及 Linux 常用命令。

第一部分，介绍 C 语言调用汇编实现累加和求值的方法。

第二部分，介绍 C 语言调用汇编实现更复杂的数组选择排序的方法。

第三部分，介绍 C 语言内嵌汇编的使用方法。

2.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

2.3 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 `extern` 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 个通用寄存器，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 通用寄存器用途

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。

2.4 实验任务操作指导

2.4.1 C 语言调用汇编实现累加和求值

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，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。

按照本实验的《实验环境搭建手册》中的步骤购买和登录鲲鹏云服务器后。（后续几个实验都是在华为鲲鹏云服务器上）

步骤 1 创建文件目录

输入命令 `cd /home` 进入 home 目录下，注意为了规范文件路径，后续实验文件夹都应在 home 目录下创建。

依次执行命令 `mkdir sum`、`cd sum` 创建并进入到 sum 目录。

```
cd /home
```

```
mkdir sum
cd sum
```

步骤 2 创建 sum.c 文件

执行命令 vim 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;
    printf("请输入一个正整数: ");
    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;
}
```

如图 2-1 所示：

```
#include <stdio.h>
extern int add(int num);
int main()
{
    int i,sum;
    printf("请输入一个正整数: ");
    scanf("%d",&i);
    sum=add(i);
    printf("sum=%d\n",sum);
    return 0;
}
```

图2-1 sum.c 代码

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 3 创建 add.s 文件

执行 vim 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。
        MOV X1,#0
add:                                //lable: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

```

如图 2-2 所示：

```

.global add
        MOV x1,#0
add:
        ADD x1,x1,x0
        SUB x0,x0,#1
        CMP x0,#0
        BNE add
        MOV x0,x1
        RET

```

图2-2 add.s 代码

步骤 4 使用 gcc 编译生成可执行文件

GCC 是由 GNU 开发的编程语言译码器。

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。

-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）所必要的信息，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2-3 编译运行

编译成功，程序执行结果正确，说明 C 程序成功调用了汇编程序。

2.4.2 C 语言调用汇编实现选择排序

本示例程序实现选择排序功能。C 代码中定义初始数组（排序前），调用汇编函数 sort 对初始数组进行排序，最后输出排序之后的数组。

步骤 1 创建文件目录

输入命令 `cd /home` 进入 home 目录。

依次执行命令 `mkdir sort`、`cd sort` 创建并进入 sort 文件夹。

```
cd /home
mkdir sort
cd sort
```

步骤 2 创建 sort.c 文件

执行命令 `vim sort.c` 编写 c 程序，

```
vim sort.c
```

编写内容如下：

```
#include<stdio.h>
extern void sort(int *a); //声明外部函数 sort。
int main()
{
    int a[6]={66,11,44,33,55,22}; //初始数组（排序前）。
    printf("before: ");
    for(int i=0;i<6;i++)
    {
        printf("%d ",a[i]);
    } //输出排序前数组。

    sort(a); //调用 sort 进行选择排序。
    printf("\nsort:  ");
    for(int i=0;i<6;i++)
    {
        printf("%d ",a[i]);
    } //输出排序后的数组。
    printf("\n");
    return 0;
}
```

如图 2-4 所示：

```
#include<stdio.h>
extern void sort(int *a);
int main()
{
    int a[6]={66,11,44,33,55,22};
    printf("before: ");
    for(int i=0;i<6;i++)
    {
        printf("%d ",a[i]);
    }
    sort(a);
    printf("\nsort: ");
    for(int i=0;i<6;i++)
    {
        printf("%d ",a[i]);
    }
    printf("\n");
    return 0;
}
```

图2-4 sort.c

输入完成后保存并退出。

步骤 3 创建 call.s 文件

执行命令 vim call.s 编写汇编代码。

```
vim call.s
```

在汇编代码中传入的参数为数组的初始地址，存放在 x0 中。x1、x2 分别存放比较两数的地址，w3 和 w4 分别存放比较的两个数的值，x6 和 x7 为内外两层循环的计数器。

对于 ARM64 寄存器来说，x 开头代表其是一个 64 位寄存器，w 开头代表它是一个 32 位寄存器。本示例中数组定义为 int 数组，int 类型数组占四字节，32 位，因此改用 w 寄存器来存放比较值。

编写代码如下：

```
.global sort //声明全局函数 sort
sort:
    mov x6,#6 //初始化外层循环数
loop1:
    mov x7,x6 //初始化内层循环数
    mov x2,x0
    mov x1,x0 //将初始地址存入 x1 和 x2 中
loop2:
    sub x7,x7,#1 //内层循环自减计数
    add x2,x2,#4 //x2 中的地址增加四位，即指向下一个数
    ldr w3,[x1]
    ldr w4,[x2] //将 x1 和 x2 存放的地址指向的值分别存入 w3 和 w4 中
    cmp w3,w4 //比较 w3 和 w4 的值，判断是否需要交换
```

```

        bls next  //前者小于后者无需交换，跳转到 next
        str w3,[x2]
        str w4,[x1]  //前者大于后者数值互换
next:    cmp x7,#1  //内层循环结束判断
        bne loop2
        add x0,x0,#4  //选择排序判断下一个数应该为多少
        sub x6,x6,#1
        cmp x6,#1
        bne loop1  //外层循环跳转
ret

```

如图 2-5 所示：

```

.global sort
sort:
    mov x6,#6
loop1:
    mov x7,x6
    mov x2,x0
    mov x1,x0
loop2:
    sub x7,x7,#1
    add x2,x2,#4
    ldr w3,[x1]
    ldr w4,[x2]
    cmp w3,w4
    bls next
    str w3,[x2]
    str w4,[x1]
next:  cmp x7,#1
    bne loop2

    add x0,x0,#4
    sub x6,x6,#1
    cmp x6,#1
    bne loop1

ret
~

```

图2-5 call.s

编写完成后，保存退出。

步骤 4 编译并运行可执行文件

执行以下命令进行编译：

```
gcc sort.c call.s -o sort
```

输入命令 ./sort 运行程序。

```
./sort
```

如图 2-6 所示：

```
[root@ecs-01 sort]# gcc sort.c call.s -o sort
[root@ecs-01 sort]# ./sort
before: 66 11 44 33 55 22
sort:   11 22 33 44 55 66
[root@ecs-01 sort]#
```

图2-6 运行排序程序

编译成功，程序执行结果正确。

2.4.3 C 语言内嵌汇编

C 语言是无法完全代替汇编语言的，一方面是汇编语言效率比 C 语言要高，另一方面是某些特殊的指令在 C 语言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::clobber);
```

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 创建文件目录

输入命令 `cd /home` 进入 home 目录。

执行命令 `mkdir builtin` 创建文件夹，输入 `cd builtin` 进入文件夹。

```
cd /home
mkdir builtin
cd builtin
```

步骤 2 创建 builtin.c 文件

执行命令 `vim builtin.c` 编写 c 程序。

```
vim builtin.c
```

编写内容如下：

```
#include <stdio.h>
int main()
{
    int val;
    printf("请输入一个正整数: ");
    scanf("%d",&val);
    __asm__ __volatile__(
        "MOV x1,#0 \n"
        "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)
        //代表只写，即在汇编代码里只能改变 C 语言变量的值，而不能取它的值。
        // 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 语言变量即 val;
        : "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;
}
```

如图 2-7 所示：

```
#include <stdio.h>
int main()
{
    int val;
    printf("请输入一个正整数: ");
    scanf("%d",&val);
    __asm__ __volatile__(
        "MOV X1,#0\n"
        "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)
        : "0"(val)
        :
    );
    printf("sum is %d \n",val);
    return 0;
}
```

图2-7 builtin.c

输入完成后保存并退出。

步骤3 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2-8 执行 builtin

编译成功，程序执行结果正确。

3 鲲鹏 920 处理器的汇编代码优化

3.1 实验目的

本实验提供了递进的优化思路，通过对比不同示例程序观察优化效果，学习了解基于 ARM 架构硬件特性的优化方案。

第一部分，介绍测试函数执行时间的方法和未经优化的内存拷贝函数。

第二部分，介绍针对内存拷贝的循环展开优化方案。

第三部分，介绍鲲鹏处理器流水线优化方案。

第四部分，介绍内存突发传输方式优化方案。

3.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

3.3 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`。

3.4 实验任务操作指导

3.4.1 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 `memcpy()` 的执行时间，调用了 `clock_gettime()` 来分别记录 `memcpy()` 执行前和执行后的系统时间，以纳秒为计时单位。

步骤 1 创建文件目录

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，依次执行 `mkdir memory`、`cd memory` 创建并进入 `memory` 文件夹。

```
mkdir memory
cd memory
```

步骤 2 创建 `time.c` 文件

执行 `vim time.c` 编写 c 语言计时程序。

```
vim time.c
```

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 len1); //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,len1); //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;

}

```

如图 3-1 所示:

```

#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 len1); //声明外部函数
int main()
{
    struct timespec t1,t2; //定义初始与结束时间
    int i,j;
    for(i=0;i<LEN-1;i++) //为初始地址段赋值，以便后续从该地址段读取数据拷贝
    {
        src[i]='a';
    }
    src[i]=0;
    clock_gettime(CLOCK_MONOTONIC,&t1); //计算开始时间。
    memcpy(dst,src,len1); //汇编调用，执行相应代码段。
    clock_gettime(CLOCK_MONOTONIC,&t2); //计算结束时间。
    printf("memcpy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
    return 0;
}

```

图3-1 time.c 计时

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 3 创建 copy.s 文件

执行 vim copy.s 编写优化前的原始汇编代码。

vim copy.s

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 memorycopy
    ret

```

如图 3-2 所示：

```

.global memorycopy

memorycopy:
    ldrb w3,[x1],#1
    str w3,[x0],#1
    sub x2,x2,#1
    cmp x2,#0
    bne memorycopy

ret

```

图3-2 copy.s 内存拷贝

步骤 4 编译并运行可执行文件

执行 `gcc time.c copy.s -o copy` 完成编译，并执行。

```

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

```

结果如下，可以看到 `memorycopy` 函数具体的执行时间为 48176221ns，如图 3-3 所示：

```

[root@ecs-003 3]# gcc time.c copy.s -o copy
[root@ecs-003 3]# ./copy
memorycopy time is 48176221 ns

```

图3-3 原始程序执行时间

接下来我们基于原始代码进行修改，通过三种方式对代码进行优化，观察他们的执行时间进行对比。

3.4.2 循环展开优化

循环展开是最常见的代码优化思路，通过减少循环开销的指令数目来实现代码优化。

步骤 1 创建 2 倍展开优化 `copy_2ops_loop.s` 文件

```

vim copy_2ops_loop.s

```

先将 `copy.s` 展开两倍,命名为 `copy_2ops_loop.s`，代码如下：

```

.global memorycopy
memorycopy:
    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
```

如图 3-4 所示:

```
.global memorycopy
memorycopy:
sub x1,x1,#1
sub x0,x0,#1
lp:
ldrb w3,[x1,#1]!
ldrb w4,[x1,#1]!
str w3,[x0,#1]!
str w4,[x0,#1]!

sub x2,x2,#2
cmp x2,#0
bne lp
ret
```

图3-4 copy_2ops_loop.s

步骤 2 编译并运行执行文件

执行命令 `gcc time.c copy_2ops_loop.s -o copy_2ops_loop` 编译并运行程序。

```
gcc time.c copy_2ops_loop.s -o copy_2ops_loop
./copy_2ops_loop
```

在进行了循环展开后, 这次 `memorycopy` 函数的执行时间变为了 41521352 ns, 如图 3-5 所示:

```
[root@ecs-003 3]# gcc time.c copy_2ops_loop.s -o copy_2ops_loop
[root@ecs-003 3]# ./copy_2ops_loop
memorycopy time is 41521352 ns
```

图3-5 编译执行程序 copy_2ops_loop

步骤 3 创建 4 倍展开优化 copy_4ops_loop.s 文件

同样的道理, 循环展开四倍, 输入命令 `vim copy_4ops_loop.s`。

```
vim copy_4ops_loop.s
```

然后输入以下代码, 并将其编译, 得到 `copy_4ops_loop`, 具体代码如下:

```
.global memorycopy
```

```
memorycopy:
    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 copy_4ops_loop.s -o copy_4ops_loop` 执行程序。

```
gcc time.c copy_4ops_loop.s -o copy_4ops_loop
./copy_4ops_loop
```

结果为 38978322ns，如图 3-6 所示：

```
[root@ecs-003 3]# gcc time.c copy_4ops_loop.s -o copy_4ops_loop
[root@ecs-003 3]# ./copy_4ops_loop
memorycopy time is 38978322 ns
```

图3-6 编译执行程序 copy_4ops_loop

函数执行时间基于二倍的基础上也得到了优化，但是优化效果并没有上次的突出。随着循环展开的程度越高，循环执行开销的比例就会越小，优化的空间也就会越来越小。

3.4.3 鲲鹏处理器流水线优化

鲲鹏 920 有两个 load/store 流水线，其访存单元支持每拍 2 条读或写访存操作。原始代码下由于源字符串的地址和目标的字符串的地址并不连续，而且这种不连续地址的一读一写交替进行，导致内存访问的连续性很差，并且一个循环中，指令之间的数据相关性比较强，无法充分利用 CPU 流水线机制。所以我们可以通过在一个循环中减少指令之间的数据相关性来改善这个缺陷，从而优化代码。

步骤 1 创建流水线优化二倍展开 copy_2ops_flow.s 文件

执行命令 `vim copy_2ops_flow.s`。

```
vim copy_2ops_flow.s
```

编写内存拷贝函数，代码如下：

```
.global memorycopy
memorycopy:
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,#2]! //一次循环读两个字节
str w3,[x0,#1]
str w4,[x0,#2]! //一次循环写两个字节
sub x2,x2,#2
cmp x2,#0
bne lp
ret
```

如图 3-7 所示：

```
.global memorycopy
memorycopy:
sub x1,x1,#1
sub x0,x0,#1
lp:
ldrb w3,[x1,#1]
ldrb w4,[x1,#2]!
str w3,[x0,#1]
str w4,[x0,#2]!

sub x2,x2,#2
cmp x2,#0
bne lp
ret
```

图3-7 copy_2ops_flow.s

保存退出。

步骤 2 编译并运行可执行文件

编译执行 gcc time.c copy_2ops_flow.s -o copy_2ops_flow。

```
gcc time.c copy_2ops_flow.s -o copy_2ops_flow
./copy_2ops_flow
```

执行时间为 38509839ns，如图 3-8 所示：

```
[root@ecs-003 3]# gcc time.c copy_2ops_flow.s -o copy_2ops_flow
[root@ecs-003 3]# ./copy_2ops_flow
memorycopy time is 38509839 ns
```

图3-8 执行 copy_2ops_flow 程序

可以看出利用访存流水线的特性进行优化，程序执行时间更少，性能优化效果更佳。

步骤 3 分析性能差异

对于流水线优化思路，同一类型的访存指令之间要求没有依赖关系，这样才能在循环展开后充分利用两条 load/store 流水线来并行地执行这些访存指令，从而有效提高性能。

对比 3.4.2 节中的访存方式：

```
ldrb w3,[x1,#1]!           ldrb w3,[x1,#1]
ldrb w4,[x1,#1]!           ldrb w4,[x1,#2]!
```

两种方式 ldrb 指令用到的寄存器都是 x1，但是左侧的第二条 ldrb 指令中的 x1 要依赖于第一条指令的 x1，即必须等待第一条 ldrb 执行完毕后 x1 更新后才能继续执行第二条指令，因此流水线优化效果较差。右边的两条指令中，x1 的值是不变的，指令间不存在依赖关系。能够充分利用流水线优化。

3.4.4 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 copy_2data_loop.s 文件

执行命令 vim copy_2data_loop.s。

```
vim copy_2data_loop.s
```

编写以下代码：

```
.global memorycopy
memorycopy:
ldp x3,x4,[x1],#16 //ldp 指令加载 x1 地址后 16 个字节的内存数据存放到 x3 和 x4 中
stp x3,x4,[x0],#16
sub x2,x2,#16
cmp x2,#0
bne memorycopy
ret
```

代码如图 3-9 所示：

```
.global memorycopy
memorycopy:
ldp x3,x4,[x1],#16
stp x3,x4,[x0],#16
sub x2,x2,#16
cmp x2,#0
bne memorycopy
ret
```

图3-9 copy_2data_loop.s**步骤 2 编译并运行可执行文件**

编译执行 `gcc time.c copy_2data_loop.s -o copy_2data_loop` 和 `./copy_2data_loop` 命令。

```
gcc time.c copy_2data_loop.s -o copy_2data_loop
./copy_2data_loop
```

如图 3-10 所示：

```
[root@ecs-003 3]# gcc time.c copy_2data_loop.s -o copy_2data_loop
[root@ecs-003 3]# ./copy_2data_loop
memorycopy time is 15081955 ns
```

图3-10 执行 copy_2data_loop 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。

3.5 总结

我们在最开始代码的基础上进行了三种不同方式的优化实验，根据实验结果可以发现，循环展开优化、处理器流水线优化、内存突发传输方式优化这三种方式都可以减少程序运行所需的时间。

4 鲲鹏 920 处理器增强型 SIMD 运算

4.1 实验目的

在支持 SIMD 操作的 CPU 中，指令译码后几个执行部件同时访问内存，一次性获得所有操作数进行运算。这种并行运算的特性使得 SIMD 在数据密集型计算领域有着显著优势。

本示例通过比较是否使用 SIMD 优化这两种方法进行矩阵点乘运算，观察 SIMD 优化对矩阵点乘运算的优化效果。

4.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

4.3 实验原理

鲲鹏 920 处理器基于 ARM v8，实现了专门的 32 个 128bit 的向量寄存器，即 NEON 寄存器。这些寄存器用于支持浮点运算和 SIMD 运算，记名为 v0-v31。表 4-1 提供了 NEON 寄存器的使用说明。

表4-1 ARM64 下 NEON 寄存器使用说明

寄存器	用途
v0~v7	用于参数传递和返回值，子程序不需要保存。
v8~v15	在传递的参数数目多于8个时，需要将其低64位压栈。
v16~v31	子程序使用时不需要保存。

如果仅使用 v0-v31 这些寄存器的低 64 位、低 32 位、低 16 位和低 8 位，那么也可以记名为 D0-D31（低 64 位）、S0-S31（低 32 位）、H0-H31（低 16 位）和 B0-B31（低 8 位）。

如果需要使用这些寄存器的全部 128bit，但同时也需要使用这 128bit 中的一部分，那么需要在汇编指令操作具体寄存器时在 v0-v31 之后添加后缀的方式来达到既使用全部又使用局部的目的，后缀格式见表 4-2。

表4-2 NEON 寄存器数据位宽控制标识表

标识	位宽	类型	示例
b	8bit	char	v1.1b: v1寄存器的低8位, v.16b: v1寄存器的后8位
h	16bit	short	v1.1h: v1寄存器的低16位 v1.8h: v1寄存器的后16位
s	32bit	int	v1.1s: v1寄存器的低32位 v1.4s: v1寄存器的后32位
d	64bit	long	v1.1d: v1寄存器的低64位, v1.2d: v1寄存器的后64位

4.4 实验任务操作指导

4.4.1 基础运算

步骤 1 创建文件目录

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，在 home 目录下执行命令 `mkdir simd` 创建 simd 文件夹，并进入 simd 目录中。

```
cd /home
mkdir simd
cd simd
```

步骤 2 创建 nosimd.c 文件

执行命令 `vim nosimd.c`。

```
vim nosimd.c
```

编写原始代码，即不使用 SIMD 进行矩阵乘法运算，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <arm_neon.h>

static void matrix_mul_asm(uint16_t **matrix_A,uint16_t **matrix_B,uint16_t **matrix_C) //矩阵相乘函数
{
```

```

uint16_t *a=(uint16_t *)matrix_A;
uint16_t *b=(uint16_t *)matrix_B;
uint16_t *c=(uint16_t *)matrix_C;
//内嵌汇编代码
__asm__ __volatile__ (
    "ldrh w0,[%0]\n"
    //将 matrix_A 矩阵的第一行第一列分别加载到 w0 中。
    "ldrh w1,[%1]\n"
    //将 matrix_B 矩阵的第一行第一列分别加载到 w1 中。
    "mul w0,w0,w1\n"
    "strh  w0,[%2]\n"
    //将计算结果储存到矩阵 matrix_C 中。
    //省略重复 14 次的过程
    //...
    "ldrh w0,[%0,#30]\n"
    "ldrh w1,[%1,#30]\n"
    "mul w0,w0,w1\n"
    "strh  w0,[%2,#30]\n"
    :"+r"(a),"+r"(b),"+r"(c)
    //+r 表示存放在寄存器中，可读可写。
    : "cc","memory","x0","x1","v2","v3","v4","v5","v6","v7"
    //
);
}
int main()
{
    struct timespec t1,t2;
    uint16_t matrix_A[4][4]={
        {1,2,3,4},
        {5,6,7,8},
        {1,2,3,4},
        {5,6,7,8}
    }; //初始化矩阵 matrix_A

    uint16_t matrix_B[4][4]={
        {1,5,8,4},
        {2,6,7,3},
        {3,7,6,2},
        {4,8,5,1}
    }; //初始化矩阵 matrix_B

    uint16_t matrix_C[4][4]={0}; //初始化矩阵 matrix_C
    int i,j;
    clock_gettime(CLOCK_MONOTONIC,&t1);
    matrix_mul_asm((uint16_t **)matrix_A,(uint16_t **)matrix_B,(uint16_t **)matrix_C);

```

```
//函数调用，实现矩阵相乘
clock_gettime(CLOCK_MONOTONIC,&t2);
printf("memorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
for(i=0;i<4;i++)
    for(j=0;j<4;j++)
    {
        printf("%11u \n",matrix_C[i][j]); //输出矩阵 matrix_C
    }
return 0;
}
```

保存并退出。

步骤 3 编译并运行可执行文件

执行以下命令编译运行：

```
gcc nosimd.c -o nosimd
./nosimd
```

程序运行结果如图 4-1 所示：

```
[root@ecs-004 simd]# gcc nosimd.c -o nosimd
[root@ecs-004 simd]# ./nosimd
memorycopy time is          90 ns
all element of matrixC is :
      1      10      24      16
     10      36      49      24
      3      14      18       8
     20      48      35       8
```

图4-1 nosimd 运行结果

4.4.2 增强型 SIMD

矩阵存储的数据类型为 short，位宽 16bit，每行需要 64 位，因此需要使用寄存器的低 64 位，表示为 vn.4h。示例程序中用到了 ld4 和 st4 指令，指令说明如下：

ld4：加载矢量元素，同时操作 4 个矢量寄存器

st4：存储矢量元素，同时操作 4 个矢量寄存器

步骤 1 创建 simd.c 文件

执行命令 vim simd.c。

```
vim simd.c
```

编写原始代码，使用 SIMD 进行矩阵相乘运算，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
#include <arm_neon.h>
static void matrix_mul_asm(uint16_t **matrix_A, uint16_t **matrix_B, uint16_t **matrix_C) //矩阵相乘函数
{
    uint16_t *a=(uint16_t *)matrix_A;
    uint16_t *b=(uint16_t *)matrix_B;
    uint16_t *c=(uint16_t *)matrix_C;
    //内嵌汇编代码
    __asm__ __volatile__ (
        "ld4 {v0.4h-v3.4h},[%0]\n"
        //将 matrix_A 矩阵的四行分别加载到 v0 到 v3 寄存器的低 64 位中。
        "ld4 {v4.4h,v5.4h,v6.4h,v7.4h},[%1]\n"
        //将 matrix_B 矩阵的四行分别加载到 v4 到 v7 寄存器的低 64 位中。
        "mul v3.4h,v3.4h,v7.4h\n"
        "mul v2.4h,v2.4h,v6.4h\n"
        "mul v1.4h,v1.4h,v5.4h\n"
        "mul v0.4h,v0.4h,v4.4h\n"
        //每行依次进行乘法计算
        //最后的矩阵相乘结果存放在 v0 到 v3 的低 64 位中。
        "st4 {v0.4h,v1.4h,v2.4h,v3.4h},[%2]\n"
        //将计算结果储存到矩阵 matrix_C 中。
        :"+r"(a),"+r"(b),"+r"(c)
        //+r 表示存放在寄存器中，可读可写。
        :
        : "cc", "memory", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7"
    )
};

int main()
{
    struct timespec t1,t2;
    uint16_t matrix_A[4][4]={
        {1,2,3,4},
        {5,6,7,8},
        {1,2,3,4},
        {5,6,7,8}
    }; //初始化矩阵 matrix_A

    uint16_t matrix_B[4][4]={
        {1,5,8,4},
        {2,6,7,3},
        {3,7,6,2},
        {4,8,5,1}
    }; //初始化矩阵 matrix_B
```

```

uint16_t matrix_C[4][4]={0}; //初始化矩阵 matrix_C

int i,j;
clock_gettime(CLOCK_MONOTONIC,&t1);
matrix_mul_asm((uint16_t **)matrix_A,(uint16_t **)matrix_B,(uint16_t **)matrix_C);
//函数调用，实现矩阵相乘
clock_gettime(CLOCK_MONOTONIC,&t2);
printf("memorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
for(i=0;i<4;i++)
    for(j=0;j<4;j++)
        printf("out is %11u \n",matrix_C[i][j]); //输出矩阵 matrix_C

return 0;
}

```

如图 4-2 所示:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <arm_neon.h>
static void matrix_mul_asm(uint16_t **aa,uint16_t **bb,uint16_t **cc)
{
    uint16_t *a=(uint16_t *)aa;
    uint16_t *b=(uint16_t *)bb;
    uint16_t *c=(uint16_t *)cc;

    __asm__ volatile (
        "ld4 {v0.4h-v3.4h}, [%0]\n"
        "ld4 {v4.4h,v5.4h,v6.4h,v7.4h}, [%1]\n"

        "mul v3.4h,v3.4h,v7.4h\n"
        "mul v2.4h,v2.4h,v6.4h\n"
        "mul v1.4h,v1.4h,v5.4h\n"
        "mul v0.4h,v0.4h,v4.4h\n"

        "st4 {v0.4h,v1.4h,v2.4h,v3.4h}, [%2]\n"

        :"+r"(a),"+r"(b),"+r"(c)
        :
        : "cc","memory","v0","v1","v2","v3","v4","v5","v6","v7"
    );
}

int main()
{
    struct timespec t1,t2;
    uint16_t aa[4][4]={
        {1,2,3,4},
        {5,6,7,8},
        {1,2,3,4},
        {5,6,7,8}
    };

    uint16_t bb[4][4]={
        {1,5,8,4},
        {2,6,7,3},
        {3,7,6,2},
        {4,8,5,1}
    };
    uint16_t cc[4][4]={0};
    int i,j;
    clock_gettime(CLOCK_MONOTONIC,&t1);
    matrix_mul_asm((uint16_t **)aa,(uint16_t **)bb,(uint16_t **)cc);
    clock_gettime(CLOCK_MONOTONIC,&t2);
    printf("memorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
    for(i=0;i<4;i++)
        for(j=0;j<4;j++)
            printf("out is %11u \n",cc[i][j]);

    return 0;
}

```

图4-2 simd.c

步骤 2 编译并运行可执行文件

执行以下命令编译运行：

```
gcc simd.c -o simd
./simd
```

程序运行结果如图 4-3 所示：

```
[root@ecs-004 simd]# gcc simd.c -o simd
[root@ecs-004 simd]# ./simd
memorycopy time is          40 ns
      1          10          24          16
     10          36          49          24
      3          14          18           8
     20          48          35           8
```

图4-3 simd 优化结果

对比二者执行时间：

nosimd：90ns

simd：40ns

增强型 SIMD 优化效果显著。

5 鲲鹏 920 处理器 ARM 异常中断实验

5.1 实验目的

本示例主要介绍基于 ARM 内核的异常和中断的处理过程。首先介绍 ARM 异常和中断的七种情况，说明 ARM 内核对各种异常中断的处理过程，然后介绍 FIQ/IRQ 中断处理机制，最后通过案例分别讨论鲲鹏 920 处理器软中断指令 svc 的使用以及常见的 coredump 异常的处理情况。

5.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

5.3 实验原理

5.3.1 ARM 的异常中断响应过程

当一个异常出现以后，ARM 微处理器会执行以下几步操作：

1. 保存处理器当前状态，中断屏蔽位以及各条件标志位。这是通过将当前程序状态寄存器 CPSR 的内容复制到要执行的异常中断对应的 SPSR 寄存器中实现的，各个异常中断都有自己的物理 SPSR 寄存器。
2. 设置当前程序状态寄存器 CPSR 中相应的位。首先改变处理器状态进入 ARM 状态，接着改变处理器模式进入相应的异常模式，设置中断禁止位禁止相应中断。
3. 保存返回地址。将下一条指令地址存入相应的连接寄存器 LR，以便程序在处理异常返回时能从正确的位置开始执行。
4. 执行中断处理程序。强制 PC 从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

5.3.2 异常中断的返回

从异常中断的程序中返回包括下面 3 个基本操作：

1. 恢复通用寄存器中的值。通用寄存器的恢复采用一般的堆栈操作指令。

2. 恢复状态寄存器的值。恢复被中断程序的处理器状态，将 SPSR_mode 寄存器内容复制到当前程序状态寄存器 CPSR 中。
3. 修改 PC 的值。使得程序能返回到发生异常中断的下一条指令处执行，将 LR_mode 寄存器的内容复制到程序计数器 PC 中。

5.3.3 IRQ/FIQ 中断处理机制

ARM 提供的 FIQ 和 IRQ 异常中断用于外部设备向 CPU 请求中断服务，这两个异常中断的引脚都是低电平有效。当前程序状态寄存器 CPSR 的 I 控制位可以屏蔽这个中断请求：即当程序状态寄存器 CPSR 中的 I 控制位为 1 时，FIQ 和 IRQ 异常中断被屏蔽；当程序状态寄存器 CPSR 中的 I 控制位为 0 时，CPU 正常响应 FIQ 和 IRQ 异常中断请求。

FIQ 异常中断意为快速异常中断，它比 IRQ 异常中断优先级高，主要表现在以下两方面：

1. 当 FIQ 和 IRQ 中断同时产生时，CPU 先处理 FIQ 异常中断。
2. 在 FIQ 异常中断处理程序中 IRQ 异常中断被禁止。

IRQ 和 FIQ 中断处理机制如图 5-1 所示：

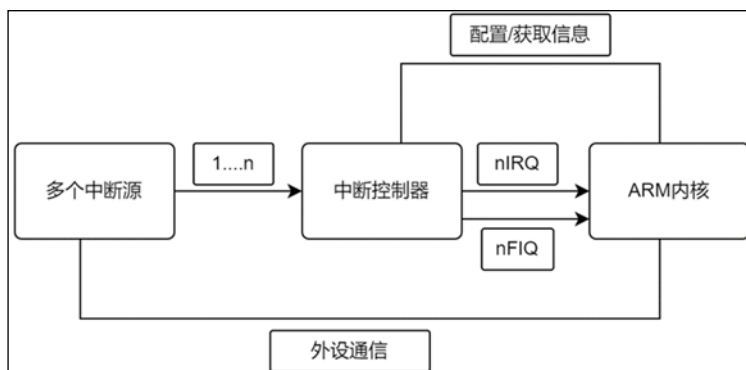


图5-1 IRQ/FIQ 中断处理机制

有的系统在 ARM 的异常向量表之外，又增加了一张由中断控制器控制的特殊向量表。当由外设触发一个中断以后，PC 能够自动跳转到这张特殊的向量表中，特殊向量表中的每个向量空间对应一个具体的中断源。例如，假设某系统一共有 20 个外设中断源，特殊向量表被直接放置在普通向量表后面，如图 5-2 所示：

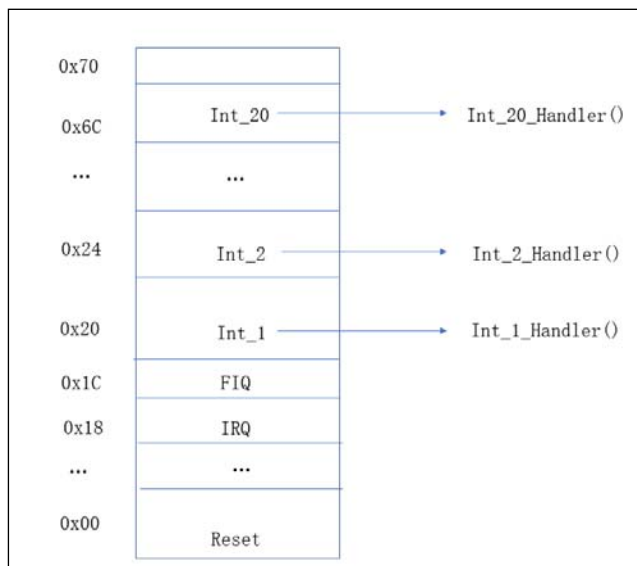


图5-2 额外的硬件异常向量表

当某个外部中断出发之后，首先触发 ARM 的内核异常，中断控制器检测到 ARM 的这种状态变化，再通过识别具体的中断源，使 PC 自动跳转到特殊向量表中的对应地址，从而开始一次异常响应。

多数情况下使用软件来处理异常分支，这是因为软件可以通过读取中断控制器来获得中断源的详细信息。如图 5-3 所示：

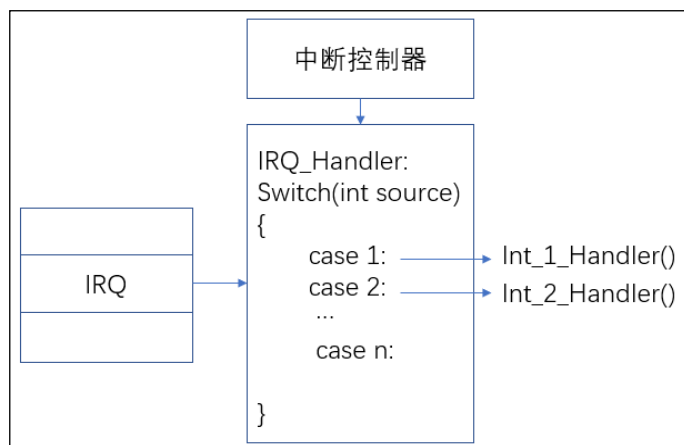


图5-3 软件控制中断分支

由于软件设计的灵活性，用户可以设计出比更好的流程控制方法

5.3.4 IRQ 和 FIQ 异常中断处理程序的返回

通常，处理器执行当前指令后查询 IRQ 中断引脚及 FIQ 中断引脚，并且查看系统是否允许 IRQ 中断及 FIQ 中断。如果中断引脚有效，并且系统允许该中断产生，则处理器将产生 IRQ 异常中断或 FIQ 异常中断。当 IRQ 和 FIQ 异常中断产生时，程序计数器 PC 的值已经更新，它指向当

前指令后面第三条指令（对于 ARM 指令来说，它指向当前地址指令加 12 个字节的位置）。在 ARM 的三级流水线架构中，程序流水线包括取址、译码和执行三个阶段，PC 指向的是当前取址的程序地址，所以 32 位 ARM 中，译码地址（正在解析还未执行的程序）为 PC-4，执行地址（当前正在执行的程序地址）为 PC-8。当 IRQ 和 FIQ 异常中断发生时，处理器将值 (PC-4) 保存到异常模式下的寄存器 LR_mode 中。这时 PC-4 指向当前指令后的第二条指令。

当异常中断处理程序使用了数据栈时，可以通过下面的指令在进入异常中断处理程序时保存被中断的执行现场，在退出异常中断处理程序时恢复被中断程序的执行现场。

```
SUBS LR,LR,#4
STMFD SP!,{Reglist, LR}      ; 数据入栈，保护现场
...
LDMFD SP!,{Reglist, PC}^     ; 数据出栈，恢复现场
```

在上述指令中，Reglist 是异常中断处理程序中使用的寄存器列表。

标识符^指示将 SPSR_mode 寄存器内容复制到当前程序状态寄存器 CPSR 中，该指令只能在特权模式即非用户模式下使用。

5.4 实验任务操作指导

5.4.1 svc 的简单使用

中断处理主要是和中断模式对应，用来切换安全级别，主要的指令包括 SVC\HVC\SMC，其中 SVC 用于应用调用内核，允许应用程序通过 SVC 指令自陷到操作系统中。HVC 用于 OS 调用 hypervisor，允许主机操作系统通过 HVC 指令自陷入到虚拟机管理程序中。SMC 用于 OS 和 hypervisor 调用 Monitor，允许主机操作系统或者管理程序通过 SMC 指令自陷入到安全监督程序。SVC 用于产生系统函数的调用请求。例如，操作系统通常不允许用户程序直接访问硬件，而是通过提供一些系统服务函数，让用户程序使用 SVC 发出对系统服务函数的调用请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就要产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务程序得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。我们这里主要介绍 SVC 指令，通过一个小程序介绍它的使用。

步骤 1 创建目录和 svc.s 文件

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，输入命令 mkdir svc 创建一个新的文件夹，然后 cd svc 进入文件夹，接着输入命令 vim svc.s 创建本次实验的示例程序代码文件。

```
cd svc
vim svc.s
```

程序代码如图 5-4 所示：

```
.text
```

```
.global _start
_start:
    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 "Test demo\n"
    .ascii "YES\n"

len=.-msg
```

```
.text
.global _start
_start:
    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 "Test demo\n"
    .ascii "YES\n"

len=.-msg
~
```

图5-4 svc.s 程序

在本示例中，两次使用软中断指令 `svc` 进行系统调用，通过 `x8` 寄存器传递系统调用号，最开始使用 `svc` 指令在屏幕上打印一个字符串。`x1` 寄存器存放的是待输出字符串的首地址，`x2` 存放待输出字符串的长度 `len`。`x8` 寄存器用于存放系统功能调用号 64，即 64 号系统功能 `sys_write()`，写的目标在 `x0` 中定义。`svc #0` 表示是一个系统功能调用。而第二次使用 `svc` 指令则是为了退出当前程序，`x0` 寄存器存放退出操作码 123，不同的退出操作码对应不同的退出操

作；x8 寄存器用于存放系统功能调用号 93，93 号系统功能为退出系统功能 `sys_exit()`；`svc #0` 表示是一个系统功能调用。

`svc` 指令是 AArch64 体系结构的指令，寄存器 `Xn` 都是 AArch64 体系结构中的寄存器，需要在 ARM v8 处理器上运行。

步骤 2 编译并运行可执行文件

接下来运行程序，首先输入命令 `as svc.s -o svc.o`，接着输入 `ld` 指令 `ld svc.o -o svc` 对程序进行链接。

```
as svc.s -o svc.o
ld svc.o -o svc
```

如图 5-5 所示：

```
[root@Mallumasvc ~]# as svc.s -o svc.o
[root@Mallumasvc ~]# ld svc.o -o svc
[root@Mallumasvc ~]#
[root@Mallumasvc ~]# ls
svc  svc.o  svc.s
[root@Mallumasvc ~]#
```

图5-5 链接程序

可以看到这里已经生成了可执行文件 `svc`，接着我们输入命令 `./svc` 执行程序。

```
./svc
```

如图 5-6 所示：

```
svc  svc.o  svc.s
[root@Mallumasvc ~]# ./svc
Test demo
YES
[root@Mallumasvc ~]#
```

图5-6 程序执行结果

通过本示例，我们介绍了 ARM 中断的几种类型，现代计算机系统都配有完善的异常和中断处理系统，CPU 的数据通路中有相应的异常和中断检测与响应逻辑，在外设接口中有相应的中断请求和控制逻辑，操作系统中有相应的中断服务程序。这些中断硬件线路和中断服务程序有机结合，共同完成异常和中断的处理过程。

5.4.2 core dump 案例介绍

5.4.2.1 Core dump 介绍

当程序运行的过程中异常终止或崩溃，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做 Core Dump（核心转储）。我们可以认为 core dump 是“内存快照”，

但实际上，除了内存信息之外，还有些关键的程序运行状态也会同时 dump 下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息。core dump 对于编程人员诊断和调试程序是非常有帮助的，因为对于有些程序错误是很难重现的，例如指针异常，而 core dump 文件可以再现程序出错时的情景。

在本实验中就模拟指针异常来触发 core dump 机制，随后利用 gdb 工具分析机制的运行效果。

步骤 1 查看 core dump 状态

要开启 core dump 功能，可以使用命令 ulimit，在终端中输入命令 ulimit -c，查看结果。

```
ulimit -c
```

如果返回的是 0，说明是默认关闭的 core dump 的，即当程序异常中止时，不会产生 core dump 文件。

步骤 2 开启 core dump 功能

如果在步骤 1 中使用 ulimit -c 得到的结果是 0，那么可以通过命令 ulimit -c unlimited 开启 core dump 功能。

```
ulimit -c unlimited
```

并且并不限制 core dump 文件大小，如果要修改文件大小，将 unlimited 改成想生成 core 文件最大的大小，单位为 KB。

步骤 3 验证 ecs 服务器中的 core dump 状态

登录到弹性云服务器后，输入命令 ulimit -c，返回 unlimited，说明已经开启，如图 5-7 结果：

```
[root@ecs-001 ~]# ulimit -c
unlimited
```

图5-7 查看 core dump 状态

5.4.2.2 实验介绍

要在程序中开启 core dump，通过如下的 API 可以查看和设置 RLIMIT_CORE。

```
#include<sys/resource.h>
int getrlimit(int resource, struct rlimit*rlim)
int setrlimit(int resource, const struct rlimit*rlim)
```

步骤 1 创建目录以及案例代码

在登陆到服务器后，首先创建本次实验的程序文件夹，输入命令 mkdir dump，然后输入命令 cd dump/进入文件夹，接着使用 vim 命令创建程序文件：vim Dumpdemo.c。

```
mkdir dump
cd dump/
vim Dumpdemo.c
```

如图 5-18 所示：

```
[root@ecs-001 ~]# mkdir dump
[root@ecs-001 ~]# cd dump/
[root@ecs-001 dump]# vim Dumpdemo.c
```

图5-8 创建程序文件

示例程序代码如下：

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#define CORE_SIZE    1024 * 1024 * 500
int main()
{
    struct rlimit rlimt;
    if (getrlimit(RLIMIT_CORE, &rlimt) == -1) {
        return -1;
    }
    printf("Before set rlimit CORE dump current is:%d, max is:%d\n", (int)rlimt.rlim_cur, (int)rlimt.rlim_max);

    rlimt.rlim_cur = (rlim_t)CORE_SIZE;
    rlimt.rlim_max = (rlim_t)CORE_SIZE;

    if (setrlimit(RLIMIT_CORE, &rlimt) == -1) {
        return -1;
    }

    if (getrlimit(RLIMIT_CORE, &rlimt) == -1) {
        return -1;
    }
    printf("After set rlimit CORE dump current is:%d, max is:%d\n", (int)rlimt.rlim_cur, (int)rlimt.rlim_max);

    char *ptr ;
    *ptr = NULL;
    *ptr = 'a';

    return 0;
}
```

步骤 2 编译生成可执行文件

接着输入命令 `gcc -g Dumpdemo.c -o Dumpdemo`，生成可执行文件。

```
gcc -g Dumpdemo.c -o Dumpdemo
```

步骤 3 执行可执行文件

接着我们就要查看生成的 core dump 文件了，通过修改 /proc/sys/kernel/core-pattern，控制 core 文件保存的位置以及文件格式，在命令行输入命令：

```
echo "/tmp/corefile-%e-%p-%t" > /proc/sys/kernel/core_pattern
```

接着执行刚才的程序：./Dumpdemo。

```
./Dumpdemo
```

结果如图 5-9 所示：

```
[root@ecs-001 dump]# ./Dumpdemo
Before set rlimit CORE dump current is:-1, max is:-1
After set rlimit CORE dump current is:524288000, max is:524288000
Segmentation fault (core dumped)
```

图5-9 程序执行

步骤 4 查看 core 文件

然后我们进入 tmp 目录下，输入命令 cd /tmp。

```
cd /tmp
```

发现果然产生了 core 文件，如图 5-10 所示：

```
[root@ecs-001 dump]# cd /tmp
[root@ecs-001 tmp]# ls
corefile-Dumpdemo-3141-1642580263
hspcrfdata_root
```

图5-10 查看 core 文件

接着我们使用 gdb 命令查看 core 文件，输入命令 yum install gdb 进行安装。

```
yum install gdb
```

一般命令格式为 gdb program core，其中 program 为可执行程序名，core 为生成的 core 文件名。我们先将他移动到程序文件夹下，然后进行进入到程序文件夹下（文件的数字后缀每台服务器不一样，需要更换）。输入命令：

```
mv corefile-Dumpdemo-3141-1642580263 /root/dump
cd /root/dump
```

步骤 5 定位 warning

输入命令查看 warning：

```
gdb ./Dumpdemo corefile-Dumpdemo-3141-1642580263
```

进入 gdb 界面后可以看到，此时直接定位到了 warning 位置，如图 5-11 所示：

```
[root@ecs-001 dump]# gdb ./Dumpdemo corefile-Dumpdemo-3141-1642580263
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.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-openEuler-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 ./Dumpdemo...
[New LWP 3141]
Core was generated by './Dumpdemo'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000000400768 in main () at Dumpdemo.c:27
27      *ptr = NULL;
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.28-36.oe1.aarch64
(gdb) █
```

图5-11 Gdb 界面

步骤 6 查看寄存器的值

然后再输入 info registers，查看寄存器的值。

```
info registers
```

如图 5-12 所示：


```
(gdb) info registers
Undefined info command: "registers". Try "help info".
(gdb) info registers
x0          0x0          0
x1          0x0          0
x2          0x4fe33aaf8d290600 5756509274371589632
x3          0x0          0
x4          0x0          0
x5          0x111b02a2    286982818
x6          0xa          10
x7          0xa          10
x8          0x40         64
x9          0xffffffff7  4294967287
x10         0x0          0
x11         0xffffffff2a95728 281474752927528
x12         0x20         32
x13         0x0          0
x14         0x0          0
x15         0xa          10
x16         0x420028     4325416
x17         0xfffd7ee02638 281464220427832
x18         0xffffffff2a9571f 281474752927519
x19         0x400788     4196232
x20         0x0          0
x21         0x4005c0     4195776
x22         0x0          0
x23         0x0          0
x24         0x0          0
x25         0x0          0
x26         0x0          0
x27         0x0          0
x28         0x0          0
x29         0xffffffff2a95860 281474752927840
x30         0x400764     4196196
sp          0xffffffff2a95860 0xffffffff2a95860
pc          0x400768     0x400768 <main+180>
cpsr       0x60000000    [ EL=0 C Z ]
fpsr       0x0          0
fpcr       0x0          0
(gdb)
```

图5-12 查看寄存器值

步骤 7 修改案例代码

修改后的源码名为 Dumpdemo_ok.c,具体如下

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#define CORE_SIZE 1024 * 1024 * 500
int main()
{
    struct rlimit rlimt;
    if (getrlimit(RLIMIT_CORE, &rlimt) == -1) {
        return -1;
    }
    printf("Before set rlimit CORE dump current is:%d, max is:%d\n", (int)rlimt.rlim_cur, (int)rlimt.rlim_max);

    rlimt.rlim_cur = (rlim_t)CORE_SIZE;
    rlimt.rlim_max = (rlim_t)CORE_SIZE;
}
```

```
    if (setrlimit(RLIMIT_CORE, &rlmt) == -1) {  
        return -1;  
    }  
  
    if (getrlimit(RLIMIT_CORE, &rlmt) == -1) {  
        return -1;  
    }  
    printf("After set rlimit CORE dump current is:%d, max is:%d\n", (int)rlmt.rlim_cur, (int)rlmt.rlim_max);  
  
    char *ptr ;  
    char A[10];  
    ptr = A;  
    *ptr = 'a' ;  
    return 0;  
}
```

步骤 8 案例代码编译执行

接着输入命令 `gcc -g Dumpdemo_ok.c -o Dumpdemo_ok`, 生成可执行文件。

```
gcc -g Dumpdemo_ok.c -o Dumpdemo_ok
```

接着执行刚才的程序: `./Dumpdemo_ok`。

```
./Dumpdemo_ok
```

执行结果, 如图 5-13 所示:

```
[root@ecs-003 svc]# ./Dumpdemo_ok  
Before set rlimit CORE dump current is:-1, max is:-1  
After set rlimit CORE dump current is:524288000, max is:524288000
```

图5-13 重新执行修正后的程序

6 鲲鹏 920 处理器利用存储器层次结构的程序优化实验

6.1 实验目的

当我们谈到程序优化时，往往想到的是改进算法的设计，从而提高程序的性能。事实上当我们的程序运行时，不仅仅是在 CPU 里运算，还包括加载数据等一系列操作，所以我们可以换一种思路，从充分利用硬件的角度去设计程序。

本示例主要利用存储器的层次结构优化程序，同时，我们还需要一种分析工具来帮我们对比优化前后的程序性能。本示例使用的是鲲鹏开发套件中的性能分析工具 Hyper-Tuner 和 Linux 自带的分析工具 perf。华为鲲鹏云服务器中并没有内置 Hyper-Tuner 工具，需自行安装，Hyper-Tuner 安装指导手册地址如下：

<https://support.huawei.com/enterprise/zh/doc/EDOC1100172781?section=j005>

Linux 性能计数器是一个新的基于内核的子系统，它提供一个性能分析框架，比如硬件，像 CPU、PMU (Performance Monitoring Unit) 功能和软件（软件计数器、trace point）功能。通过 perf，应用程序可以利用 PMU、trace point 和内核中的计数器来进行性能统计。它不但可以分析指定应用程序的性能问题（per thread），也可以用来分析内核的性能问题，当然也可以同时分析应用程序和内核，从而全面理解应用程序中的性能瓶颈。使用 perf，可以分析程序运行期间发生的硬件事件，比如 instructions retired, processor clock cycles 等，也可以分析软件时间，比如 page fault 和进程切换等参数。

Hyper-Tuner 可以进行系统性能分析，基于鲲鹏微架构，硬件，操作系统，应用线程/进程，函数等各层次的性能数据，针对多场景分析系统性能状况，定位出系统性能瓶颈点及热点函数等问题，并给出针对性的多维优化建议。同时还提供了诊断调试，支持在鲲鹏平台上诊断应用的内存异常使用情况，提供便捷的可视化界面，助力用户快速定位异常使用点，主要功能包括对内存泄漏，内存异常释放，内存的异常访问和进程内存消耗的诊断。

6.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

6.3 实验原理

由于计算机的存储结构是分层的，越靠近 CPU 的存储器，取数据时就越快。并且一个编写良好的程序往往具有良好的局部性，也就是说它们倾向于使用最近应用的数据地址附近的数据，或者最近引用过的数据本身，这种倾向性被称为局部性原理 (principle of locality)，利用这个原理，我们把需要的数据尽量放在上层来编写程序。这里举一个 500*500 的矩阵乘法优化案例，并使用鲲鹏性能分析工具 Hyper-Tuner 进行分析。

我们以搭载鲲鹏 920 芯片的 TaishanV110 服务器的结构作为例子来介绍存储器的层次结构，图 6-1 是 Taishan 服务器一个内核集群 (CCL) 的结构，共包括 4 个 Taishan 内核 V110，其中每个 Taishan 核包括：CPU core 部分，64KB L1 DCache，64KB L1 ICache，512KB L2，一个 L3 Tag Partition，(L3 Data Partition 在 CCL 之外)，具体如图 6-1 所示：

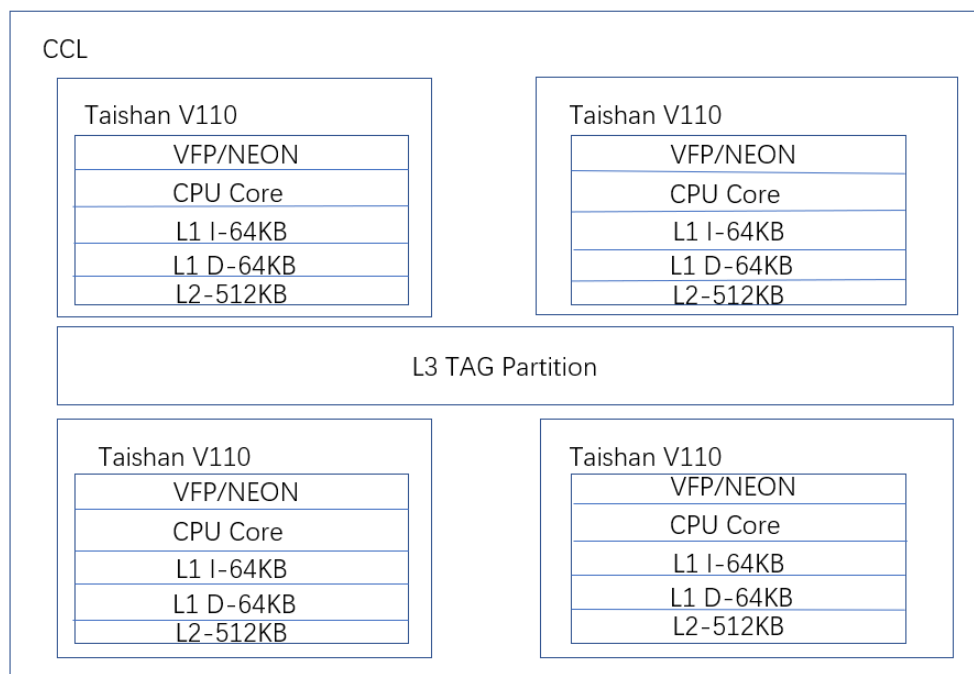


图6-1 Taishan core 结构

可以看出，每个 Taishan V110 都有自己的 L1 和 L2 Cache。L3 的 TAG Partition 是 4 个核共享的。我们从上图中可以看到 L1 Cache 距离 CPU Core 是最近的。对于一个 CPU 而言，一开始数据总是在内存，或者硬盘，甚至在更远的云端。在取数据的时候，存储器结构把数据一层一层向上 load。对 CPU 而言，需要把数据从内存按需读入 Cache 中再做运算。从内存中读入数据，是以 Cache Line 为单位。在一个程序的运行过程中，每次遇到需要去内存中取某些数据时，就会去取一个 Cache Line，当你取得了这次需要的数据，很有可能还会取到相邻的一段数据，如果在这次取到的 Cache Line 中有你需要的数据，那么 CPU 就不用再跑到内存中再取下一个 Cache Line 了，这样就节约了时间。

6.4 实验任务操作指导

6.4.1 标准矩阵乘法

标准矩阵乘法按照最基本的行列相乘，设有两个 500×500 的矩阵分别为 A 和 B，它们的乘积结果记录为矩阵 C。

步骤 1 创建文件目录

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，输入命令：cd /home 进入到 home 目录下，因为性能分析工具分析的应用路径只能是在 /opt 或 /home 下，进入到 home 目录后输入命令 mkdir Arraydemo，创建实验程序文件夹，然后输入 cd Arraydemo/ 进入到文件夹中。

```
cd /home
mkdir Arraydemo
cd Arraydemo/
```

步骤如图 6-2 所示：

```
[root@Malluma ~]# cd /home
[root@Malluma home]# mkdir Arraydemo
[root@Malluma home]# cd Arraydemo/
[root@Malluma Arraydemo]# vim arraydemo1.c
```

图6-2 创建文件夹及示例程序

步骤 2 创建 arraydemo.c 文件

输入命令 vim arraydemo.c 编写示例程序代码。

```
vim arraydemo.c
```

接下编写矩阵乘法函数，我们在这里设置的矩阵元素的数据类型是 double 类型，矩阵乘法函数代码如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define N 500
void MatrixMultiply(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            for(k=0;k<N;k++)
            {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
```

```

    }
    }
}
int main(){

    double a[N][N]={0};
    double b[N][N]={0};
    double c[N][N]={0};
    int i, j;

    srand(1);
    for( i=0;i<N;i++)
    {
        for(j=0 ;j<N;j++)
        {
            a[i][j]=rand()%100;
            b[i][j]=rand()%100;
            c[i][j]=0;
        }
    }

    struct timeval start={0,0},end={0,0};
    gettimeofday( &start, NULL );
    MatrixMultiply(a,b,c);
    gettimeofday( &end, NULL );
    long timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("Time use:%ld *10^-6 s\n",timeuse);
    printf("Total size: A:%d,B:%d\n",N*N,N*N);
    printf("Total operators: %d\n",2*N*N*N);
    return 0;
}

```

main 函数分别如图 6-3 和 6-4 所示：

```

#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define N 500
void MatrixMultiply(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            for(k=0;k<N;k++)
            {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}

```

图6-3 矩阵乘法函数

```
int main(){
    double a[N][N]={0};
    double b[N][N]={0};
    double c[N][N]={0};
    int i, j;

    srand(1);
    for(i=0;i<N;i++)
    {
        for(j=0 ;j<N;j++)
        {
            a[i][j]=rand()%100;
            b[i][j]=rand()%100;
            c[i][j]=0;
        }
    }

    struct timeval start={0,0},end={0,0};
    gettimeofday( &start, NULL );
    MatrixMultiply(a,b,c);
    gettimeofday( &end, NULL );
    long timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("Time use:%ld *10^-6 s\n",timeuse);
    printf("Total size: A:%d,B:%d\n",N*N,N*N);
    printf("Total operators: %d\n",2*N*N*N);
    printf("Element type:double\n");
    printf("Total Cycles: %ld\n",timeuse*2600);
    printf("Total Cycles/Total operators: %lf\n",((double)timeuse*2600)/((double)2*N*N*N));
    return 0;
}
```

图6-4 矩阵乘法主函数

程序分为两个部分：

第一部分是矩阵乘法函数，采用三层循环，最内层循环控制行列相乘并相加得到结果。主函数中首先对 A,B 两个矩阵随机赋值，然后通过 timeval 结构体定义 start, end 变量，分别记录程序运行的开始时间秒数和结束时间的秒数，通过头文件<sys/time.h>中的 gettimeofday()函数分别调用这两个参数，记录运行函数前的系统时间和运行后的时间，通过他们的差值就可以得出函数运行的时间 timeuse。这里介绍一下 timeval 结构体，如图 6-5 所示：

```
/*
 * Structure used in select() call, taken from the BSD file sys/time.h.
 */
struct timeval {
    long    tv_sec;        /* seconds */
    long    tv_usec;       /* and microseconds */
};
```

图6-5 timeval 结构体介绍

从上图中我们可以看出，timeval 结构体共有两个变量，其中 tv_sec 变量表示的是时间有多少秒，tv_usec 代表有多少微秒。

计算时间完成后，我们先引入度量标准“每元素的周期数(Cycles Per Element, CPE)”，作为衡量程序性能的第一种方法。由于处理器活动的顺序是由时钟控制的，时钟提供了某个频率的规

律信号，通常用千兆赫兹 (GHz)，即十亿时钟周期每秒表示。例如，当一个系统有 4GHz 处理器，这表示处理器时钟运行频率为每秒 4×10^9 个周期。每个时钟周期的时间是时钟频率的倒数。

步骤 3 编译运行可执行文件

使用 gcc 命令对程序进行编译然后执行：

```
gcc -o arraydemo arraydemo.c
./arraydemo
```

执行的结果如图 6-6 所示：

```
[root@ecs-003 Arraydemo]# ./arraydemo
Time use:970818 *10^-6 s
Total size: A:250000,B:250000
Total operators: 250000000
Element type:double
Total Cycles: 2524126800
Total Cycles/Total operators: 10.096507
```

图6-6 程序执行结果

观察程序执行的结果：

首先第一项 Time use 表示矩阵乘法函数的执行时间；

Total size 是我们设置的 A、B 两矩阵的大小；

Total operators 表示总共的运算次数；

代码中给出了计算方法为 $2 \times N \times N \times N$ ，我们的矩阵大小为 500×500 ，得到一行结果需要进行 500×500 次操作，总共 500 行一共需要 $500 \times 500 \times 500$ 次操作；

Element type 为我们矩阵元素的数据类型；

步骤 4 使用 perf 分析性能

使用 Linux 自带的分析工具查看，输入命令：

```
perf stat ./arraydemo
```

执行结果如图 6-7 所示：


```
Performance counter stats for './arraydemo1':

    944.70 msec task-clock                #    1.000 CPUs utilized
          0      context-switches        #    0.000 K/sec
          0      cpu-migrations          #    0.000 K/sec
        125      page-faults             #    0.132 K/sec
  2,456,165,609      cycles               #    2.600 GHz
  7,060,758,901      instructions         #    2.87  insn per cycle
<not supported>      branches
    280,773      branch-misses

    0.944901319 seconds time elapsed

    0.944183000 seconds user
    0.000000000 seconds sys
```

图6-7 perf 分析结果

接下来我们对 perf 的输出进行解读：

首先，task-clock 表示的是任务真正占用处理器的时间，单位为 ms，可以看到我们程序的运行时间为 944.70ms。其他的几项参数比如 cycles 和之前程序计算的也很接近，因为刚才程序计算的只是矩阵乘法函数的 cycles，perf 输出的是整个程序的 cycles（CPU 时钟周期）。instructions 为这段时间完成的 CPU 指令，branch-misses 是分支预测失败次数，这项参数也是之后要优化的主要目标。

步骤 5 登录 Hyper-tuner

我们再使用鲲鹏性能分析工具 Hyper-tuner 来对程序进行分析。打开浏览器，在地址栏输入 https://部署服务器的 IP: 端口号

示例截图使用的服务器 IP 为 210.30.97.64，端口号为 8086，按回车进入到鲲鹏性能分析工具网页版，如果出现了安全警告，则点击高级，然后继续浏览即可如图 6-8 所示：



图6-8 安全警告

登陆后的界面如图 6-9 所示：

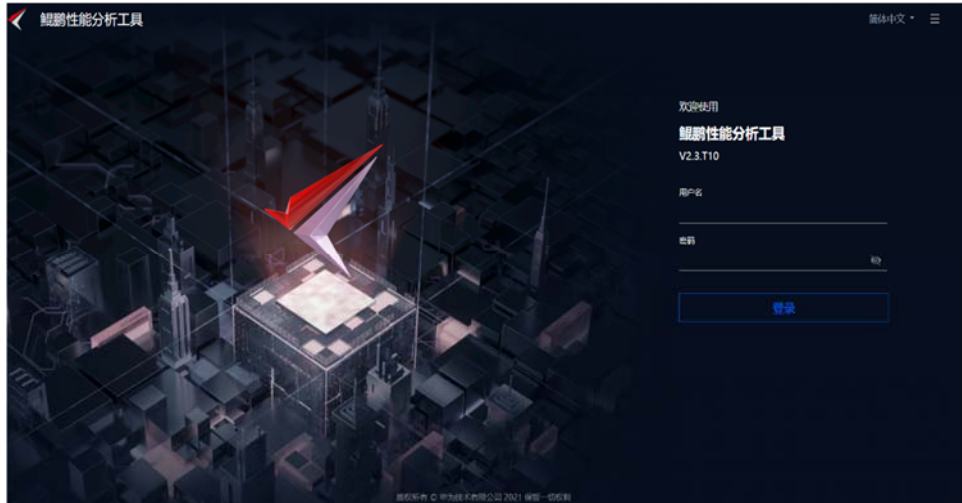


图6-9 登录界面

接着输入用户名和密码，管理员用户名默认为 tunadmin，密码在初次登陆时可以自行设置，设置好密码后点击登录，进入如图 6-10 所示界面。分析工具支持 10 个节点同时进行采样分析，支持普通用户之间互相浏览分析结果的权限，考虑到复杂的性能问题可能需要多人合作定位，用户之间可以相互查看已有的分析结果，可以方便问题定位，加快问题定位速度。



图6-10 功能选择界面

步骤 6 进入分析界面

点击系统性能分析，进入分析界面，如图 6-11 所示：

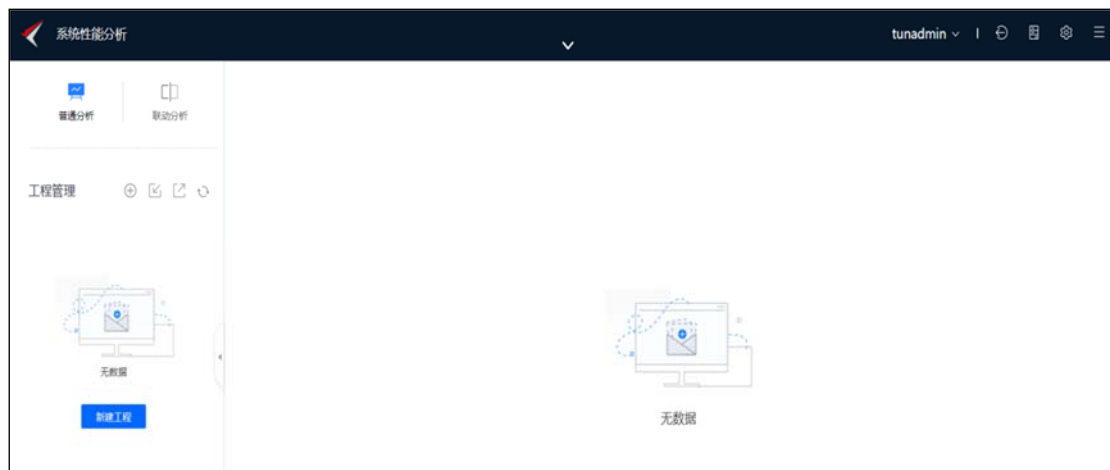


图6-11 工程界面

步骤 7 新建工程

下面创建一个新的工程，对刚刚完成的代码进行分析，在分析工具界面左侧的工程管理一栏，点击新建工程，具体配置如图 6-12 所示：

创建工程

×

★ 工程名称

Matrix_Multiply_demo

★ 场景选择

通用场景

大数据

分布式存储

HPC

数据库

采集分析服务器上的CPU、内存、存储IO、网络IO等资源，以及Top数据

★ 选择节点

支持10个节点同时进行采样分析 [管理节点](#)

节点名称	节点状态	节点IP
210.30.97.64	● 在线	210.30.97.64

确认

取消

图6-12 创建新工程

按图配置完成后，点击确认。

步骤 8 新建任务

然后点击页面中的新建分析任务，进入到新建任务界面，将任务名称设置为“Matrix_Multiply1”，分析类型选择全景分析，整体任务配置如图 6-13 所示：



新建分析任务 X

* 任务名称

分析对象

☒ 系统 ☐ 应用

采集整个系统的数据分析，无需关注系统中有哪些类型的应用在运行，采样时长由配置参数控制，适用于多业务混合运行和有子进程的场景

分析类型

通用分析 系统部件分析 专项分析

☒ 全景分析 ☐ 进程/线程性能分析 ☐ 热点函数分析 ☐ 微架构分析 ☐ 访存分析 ☐ I/O分析 ☐ 资源调度分析 ☐ 锁与等待分析

通过采集系统软硬件配置信息，以及系统CPU、内存、存储IO、网络IO资源的运行情况，识别系统瓶颈并提供优化建议

* 采样时长 (s) (2~300)

* 采样间隔 (s) 采样间隔应当小于或等于采样时长的1/2且最大为10s

图6-13 全景分析

配置完成后回到 ecs 服务器的终端。

步骤 9 编写 shell 脚本自动执行

现在要编写一个自动执行程序 shell 脚本，在刚刚完成的代码系统的路径下，我们使用 vim 命令创建一个 shell 脚本，输入命令 vim arraydemo.sh。

```
vim arraydemo.sh
```

在编辑界面输入以下代码：

```
#!/bin/bash
for i in {1..200};
do
    taskset -c 1 ./arraydemo;
done
```

这段代码的主要目的是自动执行 arraydemo 这个可执行文件，按 ESC，输入：wq 保存退出，如图 6-14 所示：

```
#!/bin/bash
for i in {1..200};
do
    taskset -c 1 ./arraydemo1;
done
```

图6-14 arraydemo.sh 脚本文件

在这段代码中，taskset 命令是 Linux 提供的一个命令，它可以让某个程序运行在某个 CPU 上，-c 就是 CPU 编号，后面的数字 1 指绑定在 CPU 核 1 上，这样在我们使用分析工具时，就可以执行这个脚本，我们的程序就可以一直循环运行，进行分析。

步骤 10 修改 shell 脚本的执行权限

接下来我们要修改一下文件的权限，进入到我们刚刚编写的程序的目录，输入命令 `chmod 777 /home`，`chmod 777 /home/Arraydemo`，`chmod 777 /home/Arraydemo/arraydemo` 将文件和路径都修改成所有用户都可进行读写执行的权限，我们的脚本文件也要修改权限，方法和之前一样，在 Arraydemo 目录下输入命令 `chmod 777 arraydemo.sh`。

```
chmod 777 /home
chmod 777 /home/Arraydemo
chmod 777 /home/Arraydemo/arraydemo
chmod 777 arraydemo.sh
```

权限都修改好后，可以输入指令：`ls -lh` 查看目录下文件的权限。

```
ls -lh
```

确认是否都修改成功，如图 6-15 所示：

```
-rwxrwxrwx 1 root root 70K 8月 29 09:36 arraydemo1
-rwxrwxrwx 1 root root 77 8月 29 09:35 array1.sh
```

图6-15 修改文件权限

步骤 11 执行 shell 脚本

然后我们输入命令 `./arraydemo.sh` 执行脚本。

```
./arraydemo.sh
```

步骤 12 使用 Hyper-tuner 分析工具

接着立即回到分析工具界面，点击开始分析进行分析，部分分析结果如图 6-16 所示：



CPU	%user	%nice	%sys	%iowait	%irq	%soft	%idle	%cpu
0	0.27	0	0.73	0	0	0.36	98.64	1.36
1	99.73	0	0.09	0	0.09	0	0.09	99.91

图6-16 全景分析部分结果

分析工具给出了如下的优化建议：建议尝试打开或关闭 CPU 预取开关。CPU 将内存中的数据读到 CPU 的高速缓冲 Cache 时，会根据局部性原理，除了读取本次要访问的数据，还会预取本次数据的周边数据到 Cache 里面，如果预取的数据是下次要访问的数据，那么性能会提升，如果预取的数据不是下次要取的数据，那么会浪费内存带宽。对于数据比较集中的场景，预取的命中率高，适合打开 CPU 预取，反之需要关闭。

我们可以看出，性能分析工具给出的优化建议与之前本节刚开始讨论的有关 Cache Line 的分析一样：由于标准矩阵乘法是行列相乘，当数据按列读取时，会一次跳过一行的数据，这样的话就没有充分利用空间局部性，而且我们的脚本文件因为绑定到 1 号核上运行，CPU 利用率（图中 %cpu 一项），核 1 达到了 99.91%。且大部分时间处于用户态（图中 %user 一项）由此说明，该程序绝大部分消耗在用户态计算，没有其他 I/O 或中断操作。

接下来我们根据优化建议对代码进行修改，以充分利用好空间局部性。

6.4.2 矩阵乘法优化

6.4.2.1 优化方案一：按行读取

由于矩阵是按照二维数组存储的，数组中的数据在内存里面是按行存储的，但是标准矩阵乘法的算法是行列相乘，所以其中的一个矩阵读取方式为按列读取，这样的话访问顺序与存放顺序不一致，每次访问需要跳过一行的元素，所以这次我们将相乘的两个矩阵都按行读取数据，使得访问顺序与存储顺序一致。

步骤 1 创建 arraydemo_row.c 文件

在之前创建的 Arraydemo 文件夹下，输入命令 `vim arraydemo_row.c` 创建一个新的 C 文件，在这里编写以下新的代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define N 500
void MatrixMultiply1(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(k=0;k<N;k++)
    {
        for(i=0;i<N;i++)
```

```

        {
            double temp=a[i][k];
            for(j=0;j<N;j++)
            {
                c[i][j]+=temp*b[k][j];
            }
        }
    }
}

int main(){

    double a[N][N]={0};
    double b[N][N]={0};
    double c[N][N]={0};
    int i, j;

    srand(1);
    for( i=0;i<N;i++)
    {
        for(j=0 ;j<N;j++)
        {
            a[i][j]=rand()%100;
            b[i][j]=rand()%100;
            c[i][j]=0;
        }
    }

    struct timeval start={0,0},end={0,0};
    gettimeofday( &start, NULL );
    //struct timespec t1,t2;
    //clock_gettime(CLOCK_MONOTONIC,&t1);
    MatrixMultiply1(a,b,c);
    //clock_gettime(CLOCK_MONOTONIC,&t2);
    gettimeofday( &end, NULL );
    long timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("Time use:%ld *10^-6 s\n",timeuse);
    printf("Total size: A:%d,B:%d\n",N*N,N*N);
    printf("Total operators: %d\n",2*N*N*N);
    printf("Element type:double\n");
    return 0;
}

```

主函数不变，修改的矩阵乘法代码如图 6-17 所示：

```
void MatrixMultiply1(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(k=0;k<N;k++)
    {
        for(i=0;i<N;i++)
        {
            double temp=a[i][k];
            for(j=0;j<N;j++)
            {
                c[i][j]+=temp*b[k][j];
            }
        }
    }
}
```

图6-17 按行读取的矩阵乘法函数

步骤 2 编译并运行可执行文件

输入命令 gcc 进行编译:

```
gcc -o arraydemo_row arraydemo_row.c
```

生成可执行文件 arraydemo_row, 输入命令 ./arraydemo_row 执行程序查看结果。

```
./arraydemo_row
```

可以看到 Time use 为 0.710s, 相较于第一次没有进行按行读取的实验时间优化了很多, 执行的时钟周期较上次减少了一半之多。

执行结果如图 6-18 所示:

```
[root@ecs-003 Arraydemo]# gcc -o arraydemo_row arraydemo_row.c
[root@ecs-003 Arraydemo]# ./arraydemo_row
Time use:683039 *10^-6 s
Total size: A:250000,B:250000
Total operators: 250000000
Element type:double
Total Cycles: 1092862400
Total Cycles/Total operators: 7.103606
```

图6-18 优化程序结果

步骤 3 使用 perf 进行分析

使用 Linux 自带分析工具 perf 执行 perf stat ./arraydem_row 命令。

```
perf stat ./arraydemo_row
```

执行结果如图 6-19 所示:


```
Performance counter stats for './arraydemo_row':

      694.50 msec task-clock           #    1.000 CPUs utilized
           9      context-switches    #    0.013 K/sec
          0      cpu-migrations        #    0.000 K/sec
        125      page-faults          #    0.180 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    0.694757446 seconds time elapsed

    0.693838000 seconds user
    0.000000000 seconds sys
```

图6-19 perf 工具执行程序结果

对比两次的 task-clock，我们可以看出，之前的程序执行总时间为 944.70msec，经过优化后时间为 685.75msec，时间优化效果非常显著；

cycles 一项也明显降低了很多，分支预测失败这项参数有所下降，但下降程度并不明显。比较两种方法，两次实验用到的数据类型都为 double 类型，占 8bytes。假设一个 cache line 的 block size 是 64bytes。对于第一种方法的矩阵 A，由于是按行读取，每次取上来 cache line 会用到 8 次，missrate 为 12.5%。这里的 miss rate 指所需数据不在 cache line 而需要去内存中取新的 cache line 的概率。而按列读取由于相邻数据不会被立即用到，miss rate 为 1。而在第二种方法中，B、C 两个矩阵都是按行读取，miss rate 都为 0.125。在充分利用 cache 的情况下相比第一次会更快。

步骤 4 使用 Hyper-tuner 分析

然后我们登录鲲鹏性能分析工具进行进一步的分析。

在进行分析之前，我们还是要修改文件的权限，输入命令 `chmod 777 arraydemo_row` 回车，然后输入 `ls -lh` 确认是否修改成功。

```
chmod 777 arraydemo_row
ls -lh
```

如图 6-20 所示：

```
-rwxrwxrwx 1 root root 70K 9月 22 20:12 arraydemo2
```

图6-20 查看文件权限

然后我们输入命令 `vim arraydemo_row.sh` 新建一个 shell 脚本文件用来执行 arraydemo_row。

```
vim arraydemo_row.sh
```

内容和上一个实验的脚本文件唯一不同的地方是需要修改执行的文件名，这次我们将绑定的核也进行了修改，绑定到 0，1，2 这三个核上，图 6-21 展示了 arraydemo_row.sh 的内容。

```
#!/bin/bash
```

```
for i in {1..200};  
do  
    taskset -c 0,1,2 ./arraydemo_row;  
done
```

```
#!/bin/bash  
for i in {1..200};  
do  
    taskset -c 0,1,2 ./arraydemo2;  
done
```

图6-21 arraydemo.sh

输入完后按 ESC 输入：wq 保存退出，输入命令 `chmod 777 arraydemo_row.sh` 修改文件权限。

```
chmod 777 arraydemo_row.sh
```

然后我们再回到鲲鹏性能分析工具进行分析。

我们还是新建一个任务，命名为 `Matrix_Multiply2`，先对系统进行一个全景分析，任务配置图如图 6-22 所示：



图6-22 全景分析配置

回到 ecs 服务器终端，输入命令 `./arraydemo_row.sh`。

```
./arraydemo_row.sh
```

回到分析工具点击确认开始分析，分析结果如图 6-23 和 6-24 所示：

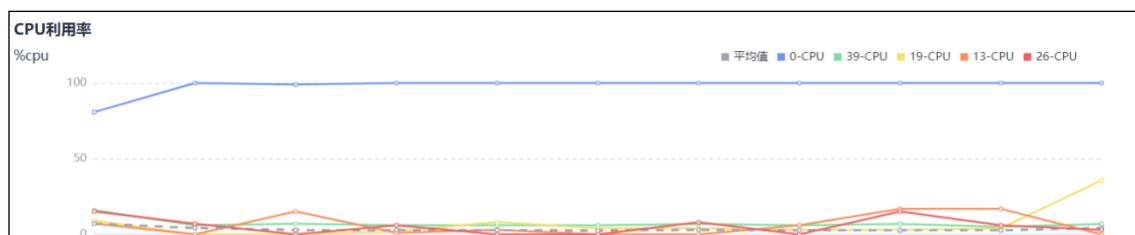


图6-23 全景分析

CPU	%user	%nice	%sys	%iowait	%irq	%soft	%idle	%cpu
0	97.89	0	0.09	0	0.09	0.09	1.83	98.17
1	0.55	0	0.55	0	0	0.45	98.45	1.55
2	0.64	0	1.27	0	0	0.09	98.00	2.00

图6-24 全景分析

由图 6-24 可以看出，和上次对比，我们这次将核绑定到 0，1，2 这三个核上，主要还是 0 核在工作，但是其他两个核的 CPU 使用率（图中%cpu）相较第一次实验的其他核也有提高，而这次分析优化建议库也未识别到更需优化的指标。

通过之前执行程序以及 perf 命令给出的程序信息，可以观察到本次实验的优化效果从执行时间的角度非常好，而且按行读取也一定程度上解决了 CPU 预取数据命中率低的问题。

6.4.2.2 优化方案二：并行计算

这一次我们同时采用两种优化思路，在采用按行读取的同时，通过并行计算，也就是将循环展开多次并行运算进行加速，这里给出的是展开 5 次的矩阵乘法代码。

步骤 1 创建 arraydemo_parallel.c 文件

主函数依旧不变，还是在刚才的/home/Arraydemo 的目录下，我们输入命令 vim arraydemo_parallel.c 新建一个文件。

```
vim arraydemo_parallel.c
```

编写以下程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define N 500
void MatrixMultiply(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(k=0;k<N;k++)
    {
        for(i=0;i<N;i++)
        {
            double temp=a[i][k];
            for(j=0;j<N;j+=5)
            {
                c[i][j]+=temp*b[k][j];
                c[i][j+1]+=temp*b[k][j+1];
                c[i][j+2]+=temp*b[k][j+2];
                c[i][j+3]+=temp*b[k][j+3];
            }
        }
    }
}
```

```

        c[i][j+4]+=temp*b[k][j+4];
    }
}
}
}
int main(){

    double a[N][N]={0};
    double b[N][N]={0};
    double c[N][N]={0};

    srand(1);
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
        {
            a[i][j]=rand()%100;
            b[i][j]=rand()%100;
            c[i][j]=0;
        }

    struct timeval start={0,0},end={0,0};
    gettimeofday( &start, NULL );
    MatrixMultiply(a,b,c);
    gettimeofday( &end, NULL );
    long timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec;
    printf("Time use:%ld *10^-6 s\n",timeuse);
    printf("Total size: A:%d,B:%d\n",N*N,N*N);
    printf("Total operators: %d\n",2*N*N*N);
    printf("Element type:double\n");
}

```

矩阵乘法代码如图 6-25 所示：

```

void MatrixMultiply(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(k=0;k<N;k++)
    {
        for(i=0;i<N;i++)
        {
            double temp=a[i][k];
            for(j=0;j<N;j+=5)
            {
                c[i][j]+=temp*b[k][j];
                c[i][j+1]+=temp*b[k][j+1];
                c[i][j+2]+=temp*b[k][j+2];
                c[i][j+3]+=temp*b[k][j+3];
                c[i][j+4]+=temp*b[k][j+4];
            }
        }
    }
}

```

图6-25 添加并行计算的矩阵乘法

步骤 2 编译并运行可执行文件

输入以下命令：

```
gcc -o arraydemo_parallel arraydemo_parallel.c
```

进行编译生成可执行文件，然后输入命令./arraydemo_parallel 执行程序，执行结果如图 6-26 所示：

```
[root@ecs-003 Arraydemo]# gcc -o arraydemo_parallel arraydemo_parallel.c
[root@ecs-003 Arraydemo]# ./arraydemo_parallel
Time use:666493 *10^-6 s
Total size: A:250000,B:250000
Total operators: 250000000
Element type:double
Total Cycles: 1066388800
Total Cycles/Total operators: 6.931527
```

图6-26 程序执行结果

步骤 3 使用 perf 进行分析

使用 Linux 分析工具 perf 执行命令：

```
perf stat ./arraydemo_parallel
```

分析结果如图 6-27 所示：

```
Performance counter stats for './arraydemo_parallel':

      692.36 msec task-clock           #    0.998 CPUs utilized
         25      context-switches      #    0.036 K/sec
          1      cpu-migrations         #    0.001 K/sec
        124      page-faults           #    0.179 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    0.693572997 seconds time elapsed

    0.690189000 seconds user
    0.000000000 seconds sys
```

图6-27 perf 分析结果

可以看出，五次循环展开对于时间的优化效果就很小了，但是其中，branch-misses（分支预测失败）的数目大大减少，之前的实验 branch-misses 达到 280756，进行展开后 branch-misses 降到了 30131，将矩阵乘法展开后，每条语句互不影响并行执行，这样就提高了效率。关于分支预测我们接下来的实验也会进一步进行分析。

7 鲲鹏 920 处理器利用 CPU 任务并行的程序优化实验

7.1 实验目的

本示例通过案例介绍 CPU 多线程技术以及利用 CPU 的任务并行方式进行程序优化的方法。

7.2 实验设备

- 华为鲲鹏云服务器；
- 具备网络连接的个人电脑。

7.3 实验原理

随着 CPU 的发展速度逐渐变快，渐渐偏离摩尔定律的轨迹，并即将到达一定的极限，越来越多的应用程序需要直面性能问题。效率，伸缩性，吞吐量等一系列指标不容忽视，不能让所有问题都依靠 CPU 来解决。面对这些问题，一个好的解决办法就是采用并发编程技术。

在计算机设计早期，为了响应更多计算性能的需要，单处理器系统发展成为多处理器系统。更现代的，类似的系统设计趋势是将多个计算核放到单个芯片中。无论多个计算核是在多个 CPU 芯片上还是在单个 CPU 芯片上，都可以称之为多核或多处理器系统。考虑一个应用，它有 4 个线程。对于单核系统，并发意味着线程随着时间推移交错执行，因为处理核只能同一时间执行单个线程。对于多核系统，并发表示线程可以并行运行，系统可以为每一个核分配一个单独线程。

7.4 实验任务操作指导

7.4.1 单线程执行矩阵乘法运算

步骤 1 创建文件目录

首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，首先在根目录下输入命令 `mkdir thread` 创建一个新的文件夹，输入命令 `cd thread/` 进入文件夹，之后我们就在当前目录下编写实验程序。

```
mkdir thread
cd thread/
```

步骤 2 创建 task_single.sh 文件

输入命令 vim arraydemo.c 编写示例程序代码。

```
vim arraydemo.c
```

我们使用第六章的矩阵乘法的程序来占用 CPU 的计算资源，输入以下代码。

```
#include<stdio.h>
#include<stdlib.h>
#define N 500
void MatrixMultiply(double a[][N],double b[][N],double c[][N])
{
    int i,j,k;
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {
            for(k=0;k<N;k++)
            {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
int main(){

    double a[N][N]={0};
    double b[N][N]={0};
    double c[N][N]={0};
    int i, j;

    srand(1);
    for( i=0;i<N;i++)
    {
        for(j=0 ;j<N;j++)
        {
            a[i][j]=rand()%100;
            b[i][j]=rand()%100;
            c[i][j]=0;
        }
    }

    MatrixMultiply(a,b,c);
    return 0;
}
```

输入命令 `gcc arraydemo.c -o arraydemo` 对其进行编译，生成可执行文件 `arraydemo`。

```
gcc arraydemo.c -o arraydemo
```

输入命令 `vim task_single.sh` 创建脚本文件。

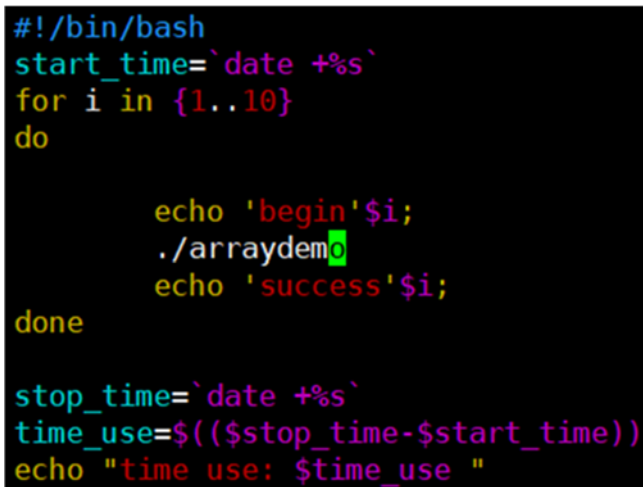
```
vim task_single.sh
```

编写以下内容：

```
#!/bin/bash
start_time=`date +%s`
for i in {1..10}
do
    echo 'begin'$i;
    ./arraydemo
    echo 'success'$i;
done

stop_time=`date +%s`
time_use=$((stop_time-start_time))
echo "time use: $time_use "
```

如图 7-1 所示：



```
#!/bin/bash
start_time=`date +%s`
for i in {1..10}
do
    echo 'begin'$i;
    ./arraydemo
    echo 'success'$i;
done

stop_time=`date +%s`
time_use=$((stop_time-start_time))
echo "time use: $time_use "
```

图7-1 task_single.sh

task1 脚本文件流程是这样的：首先定义该脚本文件开始运行的时间，然后开始循环，输出 `begin` 表示开始执行一次任务，`./arraydemo` 表示执行矩阵乘法的代码，接着会输出 `success` 表示成功执行一次任务，然后记录结束时间，最后做差得出完成本次任务总共消耗的时间。修改 `task.sh` 文件权限

在运行脚本文件前，首先要修改他的权限，输入命令 `chmod 777 task_single.sh` 修改文件的权限。


```
chmod 777 task_single.sh
```

步骤 3 执行 task_single.sh 脚本

然后输入命令 `./task_single.sh` 运行脚本，可以看到如图 7-2 所示的结果：

```
./task_single.sh
```

```
[root@ecs-004 thread]# ./task_single.sh
begin1
success1
begin2
success2
begin3
success3
begin4
success4
begin5
success5
begin6
success6
begin7
success7
begin8
success8
begin9
success9
begin10
success10
time use: 10
```

图7-2 task_single.sh 执行结果

运行脚本后，命令行就会开始输出 `begin` 和 `success` 及序号，但是我们在观察时发现，脚本运行的很慢，脚本的 `for` 循环次数设置为 10 次，观察程序执行过程我们可以发现这 10 次循环都是顺序执行的，当前一条 `begin` 和 `success` 输出完后隔一段时间才会输出下一条 `begin` 和 `success`，这样的效率相当低。所以关键的修改点就是，如何让这些任务能够并发执行。

7.4.2 shell 中实现并行任务

步骤 1 创建 task_multi.sh 文件

回到刚才的目录下，输入命令 `vim task_multi.sh` 创建一个新的脚本文件。

```
vim task_multi.sh
```

编写以下代码：

```
#!/bin/bash
start_time=`date +%s`
for i in {1..10}
do
{
    echo 'begin'$i;
```

```
./arraydemo
echo 'success'$i;
}&
done
wait
stop_time=`date +%s`
echo "time use:expr $stop_time-$start_time "
time_use=$((($stop_time-$start_time))
echo "time use: $time_use "
```

代码内容如图 7-3 所示:

```
#!/bin/bash
start_time=`date +%s`
for i in {1..10}
do
{
    echo 'begin'$i;
    ./arraydemo
    echo 'success'$i;
}&
done
wait
stop_time=`date +%s`
time_use=$((($stop_time-$start_time))
echo "time use: $time_use "
```

图7-3 task_multi.sh

为了起到并发执行的效果，首先将循环体括起来并在最后添加一个&符号，表示每次循环把指令放在后台运行，放到后台之后对这些任务并行处理。

可以看到新的脚本文件相较之前多了一条 wait 命令，wait 的意思就是等待上面已经放入后台的命令都执行完了再往下执行。也就是一旦有一条命令被放入后台，这个任务就交给了操作系统去执行，shell 脚本会继续执行下一个任务，wait 这里的作用就是等待所有放入后台命令都执行完。

步骤 2 修改 task_multi.sh 文件权限

编写好后我们依旧要修改文件的权限才能执行，输入命令 chmod 777 task_multi.sh。

```
chmod 777 task_multi.sh
```

步骤 3 执行 task_multi.sh 脚本

然后输入命令 ./task_multi.sh 执行。

```
./task_multi.sh
```

执行结果如图 7-4 所示:

```
[root@ecs-004 thread]# ./task_multi.sh
begin1
begin2
begin3
begin6
begin8
begin10
begin4
begin5
begin7
begin9
success2
success1
success3
success8
success4
success6
success10
success7
success5
success9
time use: 5
```

图7-4 task_multi.sh 执行结果

可以看到我们的修改起到了比较好的效果，运行时间只用了 5s，shell 中实现并发，就是在循环体的命令后加上&符号，放入后台运行，比起第一种方法使用的时间减少了一半。

步骤 4 使用 perf 分析实验结果

输入命令 perf stat ./task_single.sh 以及 perf stat ./task_multi.sh 执行。

```
perf stat ./task_single.sh
```

```
perf stat ./task_single.sh
```

执行结果如图 7-5 和 7-6 所示：

```
Performance counter stats for './task_single.sh':

      9,751.42 msec task-clock                #    1.000 CPUs utilized
         128      context-switches          #    0.013 K/sec
          14      cpu-migrations            #    0.001 K/sec
       1,670      page-faults               #    0.171 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

      9.754320092 seconds time elapsed

      9.729620000 seconds user
      0.011965000 seconds sys
```

图 7-5 perf stat ./task_single.sh 执行结果

```
Performance counter stats for './task_multi.sh':

      10,336.63 msec task-clock           #    1.994 CPUs utilized
           888      context-switches    #    0.086 K/sec
             9      cpu-migrations      #    0.001 K/sec
          2,021      page-faults        #    0.196 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

      5.183985833 seconds time elapsed

      10.312778000 seconds user
       0.003972000 seconds sys
```

图 7-6 perf stat ./task_multi.sh 执行结果

CPU's utilized 这一项代表 CPU 的使用率，可以看到运行 task_single.sh 脚本时，CPU 使用率为 1，而运行 task_multi.sh 脚本时，CPU 使用率为 1.994。代表我们对其进行的并行更改成功让任务运行在 CPU 的两个核上，减少了完成任务所需要的时间。

上述介绍的方案中，我们将任务放到后台执行，放入后台就意味着交由操作系统的一个线程处理了，但这里我们并不能看到具体的线程创建过程，接下来我们将自己创建线程来划分任务进行优化。

7.4.3 多线程实现排序

对于并行设计来说，可以将应用程序看成是众多相互依赖的任务的集合。将应用程序划分成多个独立的任务，并确定这些任务之间的相互依赖关系，这个过程被称为分解（Decomposition）。分解问题的方式主要有三种：任务分解、数据分解和数据流分解。

下面介绍一个排序应用案例，通过对比观察多线程相较于单线程的优化效果。

步骤 1 创建文件目录及 merge_single.c 文件

首先在根目录输入命令 mkdir thread 创建本次实验的程序文件夹，输入命令 cd thread/进入到文件夹中，然后输入命令 vim merge_single.c 创建本示例的第一个程序文件。

```
mkdir thread
cd thread/
vim merge_single.c
```

作为对比实验，首先在 merge_single.c 中实现一个单线程排序程序，编写以下程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define MAXSIZE 400000
int array[MAXSIZE];
```

```
int sort(const void*p1,const void*p2)
{
    return (*(int*)p1-*(int*)p2);
}
int main()
{
    srand((unsigned)time(NULL));
    int i;
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%100;
    }
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    qsort(array,MAXSIZE,sizeof(int),sort);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
    FILE *fp = fopen("data.txt","w+");
    unsigned int k;
    for(k=0;k<MAXSIZE;k++)
    {
        fprintf(fp,"%ld ",array[k]);
    }
    return 0;
}
```

编写的程序代码如图 7-7 所示：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<time.h>
#define MAXSIZE 400000
int array[MAXSIZE];
int sort(const void*p1,const void*p2)
{
    return (*(int*)p1-*(int*)p2);
}
int main()
{
    srand((unsigned)time(NULL));
    int i;
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%100;
    }
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    qsort(array,MAXSIZE,sizeof(int),sort);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
    FILE *fp = fopen("data.txt","w+");
    unsigned int k;
    for(k=0;k<MAXSIZE;k++)
    {
        fprintf(fp,"%ld ",array[k]);
    }
    printf("\n");
    return 0;
}
```

图7-7 merge_single.c

程序使用的编程语言为 C 语言，主要功能是通过 stdlib 库中的 qsort 函数实现对数组的排序，并通过 gettimeofday 函数获取程序执行前后的时间，做差获得排序的执行时间，最后通过将排好序的数组写入文件中观察实验对数组的排序结果。

步骤 2 编译并运行可执行文件

输入命令 gcc -o merge_single merge_single.c，然后输入命令 ./merge_single 执行程序，如图 7-8 所示。

```
gcc -o merge_single merge_single.c
./merge_single
```

接着输入 ls 命令观察是否生成了 data.txt 文件，如图 7-9 所示：

```
ls
[root@ecs-004 thread]# gcc merge_single.c -o merge_single
[root@ecs-004 thread]# ./merge_single
time use is 0.049732 s
```

图7-8 编译执行程序

```
[root@ecs-004 thread]# ls
data.txt  merge_cpu.c  merge_multi.c
```



```

CPU_ZERO(&mask);
CPU_SET(1,&mask);
qsort(arg,MAXSIZE,sizeof(int),sort);
if(pthread_setaffinity_np(pthread_self(),sizeof(mask),&mask)<0)
{
    fprintf(stderr, "set thread affinity failed\n");
}
CPU_ZERO(&get);
if(pthread_getaffinity_np(pthread_self(),sizeof(get),&get)<0)
{
    fprintf(stderr, "get thread affinity failed\n");
}
if(CPU_ISSET(1,&get))
{
    printf("thread %lu is running in processor 1\n", (long unsigned)pthread_self());
}
pthread_exit(NULL);
}
int main()
{
    srand((unsigned)time(NULL));
    int i;
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%100;
    }
    pthread_t tid;
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    pthread_create(&tid,NULL,(void*)func,array);
    pthread_join(tid,NULL);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
    FILE *fp = fopen("data2.txt", "w+");
    unsigned int k;
    for(k=0;k<MAXSIZE;k++)
    {
        fprintf(fp,"%ld ",array[k]);
    }
    return 0;
}

```

编写的代码如图 7-11 和 7-12 所示:


```

#define _GNU_SOURCE
#include<sched.h>
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<stdlib.h>
#include<sys/time.h>
#define MAXSIZE 400000
int array[MAXSIZE];
int sort(const void*p1,const void*p2)
{
    return (*(int*)p1-*(int*)p2);
}
void*func(int*arg)
{
    cpu_set_t mask;
    cpu_set_t get;
    CPU_ZERO(&mask);
    CPU_SET(1,&mask);
    qsort(arg,MAXSIZE,sizeof(int),sort);
    if(pthread_setaffinity_np(pthread_self(),sizeof(mask),&mask)<0)
    {
        fprintf(stderr, "set thread affinity failed\n");
    }
    CPU_ZERO(&get);
    if(pthread_getaffinity_np(pthread_self(),sizeof(get),&get)<0)
    {
        fprintf(stderr, "get therad affinity failed\n");
    }
    if(CPU_ISSET(1,&get))
    {
        printf("thread %lu is running in processor 1\n", (long unsigned)pthread_self());
    }
    pthread_exit(NULL);
}

```

图7-11 merge_cpu.c

```

int main()
{
    srand((unsigned)time(NULL));
    int i;
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%100;
    }
    pthread_t tid;
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    pthread_create(&tid,NULL,(void*)func,array);
    pthread_join(tid,NULL);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
    FILE *fp = fopen("data2.txt","w+");
    unsigned int k;
    for(k=0;k<MAXSIZE;k++)
    {
        fprintf(fp,"%ld ",array[k]);
    }
    return 0;
}

```

图7-12 merge_cpu.c 主函数

首先总结一下本示例中用到的函数，void CPU_ZERO (cpu_set_t*set) 是初始化函数，初始化某个 CPU 集，并将其设为空。然后用到了 CPU_SET (int cpu, cpu_set_t*set) ，这个函数是将某个 CPU 加入到 CPU 集中，本示例加入的是 1 号 CPU，CPU_ISSET (int cpu, const cpu_set_t *set) 函数是判断某个 CPU 是否已经在之前创建的 CPU 集中。pthread_setaffinity_np

(pthread_t thread, size_t cpusetsize, const cpu_set_t*cpuset) 函数用来设置某一线程运行在某个 CPU 上, pthread_getaffinity_np (pthread_t thread, size_t cpusetsize, const cpu_set_t*cpuset) 用来查看某一个 CPU 中有哪些线程。

本示例的代码流程首先创建了 func 函数, 新创建的线程从 func 的函数的地址开始运行, func 函数里包含了将线程绑定到 CPU 核 1 上, 然后进入主函数利用 pthread_create()函数创建新线程, pthread_join()等待线程完成后计算使用时间, 最后将排序好的数组写入到 data2.txt 中。

步骤 4 编译并运行可执行文件

代码编写完成后, 输入命令 gcc merge_cpu.c -o merge_cpu -lpthread, 因为本示例使用的头文件 pthread.h 不在 linux 默认库中, 所以在编译时需要链接 libpthread.a 这个库, 编译完成后输入命令 ./merge_cpu, 可以看到执行结果。

```
gcc merge_cpu.c -o merge_cpu -lpthread
./merge_cpu
```

如图 7-13 所示:

```
[root@ecs-004 thread]# gcc merge_cpu.c -o merge_cpu -lpthread
[root@ecs-004 thread]# ./merge_cpu
thread 281458919076320 is running in processor 1
time use is 0.049964 s
```

图7-13 merge_cpu.c 执行结果

可以看到输出分为两部分, 首先输出的第一句话表示我们的绑核操作成功, 将线程绑定到 CPU1 上, 接着是执行时间, 第一次单独对数组进行排序, 执行的时间为 0.049964。

然后输入命令 cat data2.txt 查看排序好的数据是否已经写入文件中。

```
cat data2.txt
```

这里仅截取部分排序结果, 如图 7-14 所示:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3
4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8
0 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 12 12 12 12 12 12
13 13 13 13 13 13 13 13 13 13 14 14 14 1
5 15 15 15 15 15 15 15 15 15 15 15 15 15
17 17 17 17 17 17 17 17 17 17 17 17 17
18 18 18 18 18 19 19 19 19 19 19 19 19 1
0 20 20 20 20 20 21 21 21 21 21 21 21 21
22 22 22 22 22 22 22 22 22 22 22 22 23
```

图7-14 data2.txt 执行结果

步骤 5 使用 perf 分析性能

使用 perf 指令查看两次程序的具体信息，分别输入命令 perf stat ./merge_single, perf stat ./merge_cpu。

```
perf stat ./merge_single
perf stat ./merge_cpu
```

如图 7-15 和 7-16 所示：

```
Performance counter stats for './merge_single':

    96.79 msec task-clock           #    0.997 CPUs utilized
          0      context-switches   #    0.000 K/sec
          0      cpu-migrations     #    0.000 K/sec
        82      page-faults        #    0.847 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    0.097110525 seconds time elapsed

    0.092795000 seconds user
    0.004039000 seconds sys
```

图7-15 merge_single

```
Performance counter stats for './merge_cpu':

    99.35 msec task-clock           #    0.966 CPUs utilized
          6      context-switches   #    0.060 K/sec
          2      cpu-migrations     #    0.020 K/sec
         97      page-faults        #    0.976 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    0.102806691 seconds time elapsed

    0.099269000 seconds user
    0.000000000 seconds sys
```

图7-16 merge_cpu

可以看到，cpu-migrations（处理器迁移次数），merge_cpu 程序的次数为 2，merge_single 为 0，也证明了我们绑核操作成功。merge_cpu 程序中，context-switches（上下文切换次数）为

6, 进行了上下文切换说明这过程中操作系统决定要把控制权从当前进程转移到某个转换到某个新进程。

步骤 6 创建 merge_multi.c 文件

本示例中, 我们要编写一个多线程排序程序, 它的工作原理如下: 将一个整数线性表分为两个长度相等的线性表。两个单独线程去对两个子线性表进行排序。接着由主线程去合并成一个排序好的列表。

输入指令 vim merge_multi.c。

```
vim merge_multi.c
```

编写以下程序代码:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<pthread.h>
#define MAXSIZE 400000
#define THREAD_NUM 2
#define midindex (MAXSIZE/THREAD_NUM)
int array[MAXSIZE];
int result_array[MAXSIZE] = { 0 };
int flag = 0;
int sort(const void*p1,const void*p2)
{
    return (*(int*)p1-*(int*)p2);
}
void Merge(int*list1,int list1_size,int*list2,int list2_size)
{
    int i=0,j=0,k=0;
    while(i<list1_size && j<list2_size)
    {
        if(list1[i]<list2[j])
        {
            result_array[k] = list1[i];
            k++;
            i++;
        }
        else
        {
            result_array[k++] = list2[j++];
        }
    }
    while(i<list1_size)
    {
        result_array[k++] = list1[i++];
    }
}
```

```
        while(j<list2_size)
        {
            result_array[k++] = list2[j++];
        }
    }
void dividing(int*array,int len)
{
    int*list1 = array;
    int list1_size = len/2;
    int*list2 = array+list1_size;
    int list2_size = len-list1_size;
    Merge(list1,list1_size,list2,list2_size);
}
void*threadfunc1(void*arg)
{
    qsort(arg,midindex,sizeof(int),sort);
    flag++;
    pthread_exit(NULL);
}

void threadfunc2(int*arg1)
{
    dividing(arg1,MAXSIZE);
}
int main()
{
    int i,j;
    srand((unsigned)time(NULL));
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%1000;
    }
    pthread_t tid1,tid2;
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    pthread_create(&tid1,NULL,threadfunc1,(void*)array);
    pthread_create(&tid2,NULL,threadfunc1,(void*)(array+midindex));
    while(flag < 2)
        ;//wait both pthread finish
    threadfunc2(array);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
}
```

```
FILE *fp = fopen("data3.txt", "w+");
int k;
for(k=0; k<MAXSIZE; k++)
{
    fprintf(fp, "%d ", result_array[k]);
}
return 0;
}
```

程序代码如图 7-17、7-18 和 7-19 所示：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<pthread.h>
#define MAXSIZE 400000
#define THREAD_NUM 2
#define midindex (MAXSIZE/THREAD_NUM)
int array[MAXSIZE];
int result_array[MAXSIZE] = { 0 };
int flag = 0;
int sort(const void*p1, const void*p2)
{
    return (*(int*)p1 - *(int*)p2);
}
void Merge(int*list1, int list1_size, int*list2, int list2_size)
{
    int i=0, j=0, k=0;
    while(i<list1_size && j<list2_size)
    {
        if(list1[i]<list2[j])
        {
            result_array[k] = list1[i];
            k++;
            i++;
        }
        else
        {
            result_array[k++] = list2[j++];
        }
    }
    while(i<list1_size)
    {
        result_array[k++] = list1[i++];
    }
    while(j<list2_size)
    {
        result_array[k++] = list2[j++];
    }
}
```

图7-17 merge_multi.c

```
void dividing(int*array, int len)
{
    int*list1 = array;
    int list1_size = len/2;
    int*list2 = array+list1_size;
    int list2_size = len-list1_size;
    Merge(list1, list1_size, list2, list2_size);
}
void*threadfunc1(void*arg)
{
    qsort(arg, midindex, sizeof(int), sort);
    flag++;
    pthread_exit(NULL);
}
void threadfunc2(int*arg1)
{
    dividing(arg1, MAXSIZE);
}
```

图7-18 merge_multi.c

```
int main()
{
    srand((unsigned)time(NULL));
    int i;
    for(i=0;i<MAXSIZE;i++)
    {
        array[i] = rand()%100;
    }
    pthread_t tid;
    struct timeval start={0,0},end={0,0};
    gettimeofday(&start,NULL);
    pthread_create(&tid,NULL,(void*)func,array);
    pthread_join(tid,NULL);
    gettimeofday(&end,NULL);
    long starttime_use = start.tv_sec*1000000+start.tv_usec;
    long endtime_use = end.tv_sec*1000000+end.tv_usec;
    double timeuse = (double)(endtime_use-starttime_use)/1000000.0;
    printf("time use is %.6f s\n",timeuse);
    FILE *fp = fopen("data2.txt","w+");
    unsigned int k;
    for(k=0;k<MAXSIZE;k++)
    {
        fprintf(fp,"%ld ",array[k]);
    }
    return 0;
}
```

图7-19 merge_multi.c-

程序的主要流程首先对创建的数组随机赋值，然后创建两个线程分别对前半段数组和后半段数组进行排序，接着等待两个线程都完成后（也就是主函数中 while 的判断），继续向下执行合并函数，然后计算时间，最后将结果输出到文件中。Merge 函数的主要功能就是对两段有序数组进行合并，并将结果写入 result_array 数组。

步骤 7 编译并运行可执行文件

输入命令 gcc merge_multi.c -o merge_multi -lpthread 进行编译，然后输入命令 ./merge_multi，查看执行结果。

```
gcc merge_multi.c -o merge_multi -lpthread
./merge_multi
```

如图 7-20 所示：

```
[root@ecs-004 thread]# gcc merge_multi.c -o merge_multi -lpthread
[root@ecs-004 thread]# ./merge_multi
time use is 0.040512 s
```

图7-20 merge_multi.c 执行结果

然后输入命令 cat data3.txt，查看排序结果是否已成功输入。

```
cat data3.txt
```

这里仅截取部分排序结果，如图 7-21 所示：

```

0 0 0 0 0 0 1 1 1 2 2 2 2 2 2 2 3 3 3 3
18 18 18 19 19 19 19 19 20 21 21 21 21 2
5 35 36 36 36 36 36 37 37 37 37 37 37 38
51 52 52 52 52 53 53 53 53 53 53 53 54
69 69 69 69 69 69 69 69 69 69 70 70 71 7
6 86 86 86 87 87 87 87 87 88 88 88 88 88
101 101 101 102 103 104 104 104 105 105
115 116 116 116 117 117 117 117 117 117
127 127 128 128 128 128 128 129 129 129
139 139 139 139 139 139 140 140 140 140
151 151 151 151 151 152 152 152 153 153
161 161 161 161 162 162 162 163 163 163

```

图7-21 data3.txt

对比之前单线程的 0.049964s，将数组分为多个相同部分通过多线程排序确实在时间上得到了优化，多线程排序确实是要优于单线程排序。而且也对分割的数组成功进行了合并，符合预期结果。

8 大作业：鲲鹏代码迁移实验

8.1 实验目的

CPU（Central Processing Unit，中央处理器）是每一台计算机大脑一样的存在，它由运算器，控制器组成。CPU 的指令系统发展至今，朝着两个截然不同的方向发展：

一是增强原有指令的功能，设置更为复杂的新指令实现软件的功能，这类机器称为复杂指令系统计算机(CISC)，典型的有采用 x86 架构的计算机；

二是减少指令种类和简化指令功能，提高指令的执行速度，这类机器称为精简指令系统计算机(RISC)，典型的有 ARM、MIPS 架构的计算机。

上世纪 70 年代，依赖于英特尔 x86 芯片的 PC 机出现了，其最大的贡献就是让计算机从企业走向个人，计算机发展至今，几乎所有软件均建立在 x86 架构之上。而如今新的拐点出现了，计算已经变成手持的了，每个人手上都有智能终端。以移动应用运行为例，传统的 x86 平台依赖指令翻译运行安卓应用，性能损耗大，兼容性也无法保证。此外，服务器端很多应用需要测试，过去的做法是真机测试，或者手机开发仿真环境，这种做法的资源灵活度低，故障率高，可靠性和易用性较差。而真正的革新可能还是需要自底向上，甚至直达芯片级。

企业和开发者都需要更多的选择。单核芯片面积算力更强，众核架构设计的 ARM 开始被关注。一个 ARM 核的面积仅为 x86 核的七分之一，同样的芯片尺寸下，ARM 的核数是 x86 的 4 倍以上，由于芯片的物理尺寸有限制，无法无限地增加，ARM 的众核横向扩展更符合分布式业务需求。

而软件迁移是指将某个可运行的程序，由原来的环境迁移到另一个环境，并重新运行。改变的环境可能是处理器架构，操作系统，软件运行环境等。

本示例使用的鲲鹏代码迁移工具是一款可以简化客户应用程序迁移到基于鲲鹏 916/920 的服务器过程的工具，工具目前仅支持 x86 Linux 到 Kunpeng Linux 的扫描与分析，当用户有 x86 平台上的软件源代码要迁移到基于鲲鹏 916/920 的服务器上时，既可以使用该工具分析可迁移性和迁移投入，也可以使用该工具自动分析出需修改的代码，并指导用户如何做出修改。华为鲲鹏云服务器中并没有内置鲲鹏代码迁移工具，需自行安装，代码迁移工具安装包下载地址为：

<https://support.huawei.com/enterprise/zh/doc/EDOC1100218623/325abd9d>。

8.2 实验设备

- 华为鲲鹏云服务器；

- 具备网络连接的个人电脑。

8.3 实验原理

8.3.1 主流架构对比

从 CPU 被发明出来到现在，有非常多种类的架构，从最基本的逻辑角度可以分为两大类，即“复杂指令集 (CISC)”和“精简指令集 (RISC)”。这也是 Intel 和 ARM 处理器最直观的一个区别，前者使用复杂指令集 (CISC)，后者使用精简指令集 (RISC)。

CISC 和 RISC 的详细对比可以查看表 8-1。

表8-1 CISC 和 RISC 对比

对比项目/类别	CISC	RISC
指令系统	复杂	简单
指令数目	一般大于200条	一般小于100条
指令字长	不固定	定长
可访存指令	不加限制	只有Load/Store指令
各种指令执行时间	相差较大	绝大多数在一个周期内完成
各种指令使用频度	相差很大	都比较常用
通用寄存器数量	较少	多
目标代码	难以用优化编译生成高效的目标代码程序	采用优化的编译程序，生成代码较为高效
控制方式	绝大多数为微程序控制	绝大多数为组合逻辑控制
指令流水线	可以通过一定方式实现	必须实现

8.3.2 程序的生命周期

本示例以一个 C 程序举例，介绍源程序到目标文件的转化过程。

使用 gcc 命令，从源文件到目标文件的转化是由编译器驱动程序完成的，如图 8-1 所示：

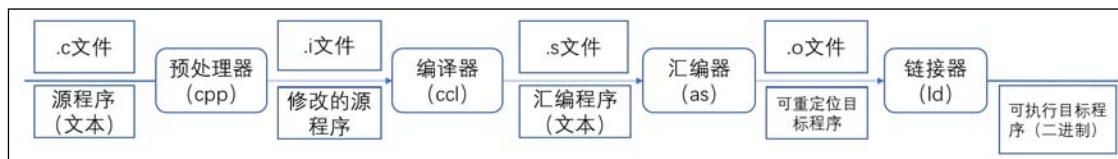


图8-1 编译系统

预处理阶段：预处理器 (cpp) 根据以字符#开头的命令，修改原始 C 程序，比如第一行命令为 `#include<stdio.h>` 的程序，该命令告诉预处理器读取头文件 `stdio.h` 的内容，并把它插入到程序文本中。得到另一个程序，以 `.i` 作为文件扩展名。

编译阶段：编译器将 `.i` 文件翻译成 `.s` 文件，变为一个汇编语言程序。

汇编阶段：下一步汇编器将 `.s` 文件翻译成机器语言指令，把这些指令打包成叫做可重定位目标程序的格式，并将结果保存在目标文件 `.o` 文件中。这是一个二进制文件。

链接阶段：当程序中调用了如 `printf` 函数，这是每个 C 编译器都提供的标准 C 库的一个函数。`printf` 函数存在于名为 `printf.o` 的单独的预编译好的目标文件中，而将这个文件以某种形式合并到 `.o` 程序中，这就是链接器的作用。结果就得到了一个可执行文件，可以被加载到内存中由系统执行。

8.3.3 程序运行

前面介绍了程序的生命周期，讲述了程序从开始到转变成可执行文件的过程，接下来我们来介绍当程序运行时发生了什么。初始时，shell 程序执行它的指令，等待我们输入一条命令。当我们输入执行程序的指令后，shell 程序将字符逐一读入寄存器，再将他放入内存中，如图 8-2 中的线段所示：

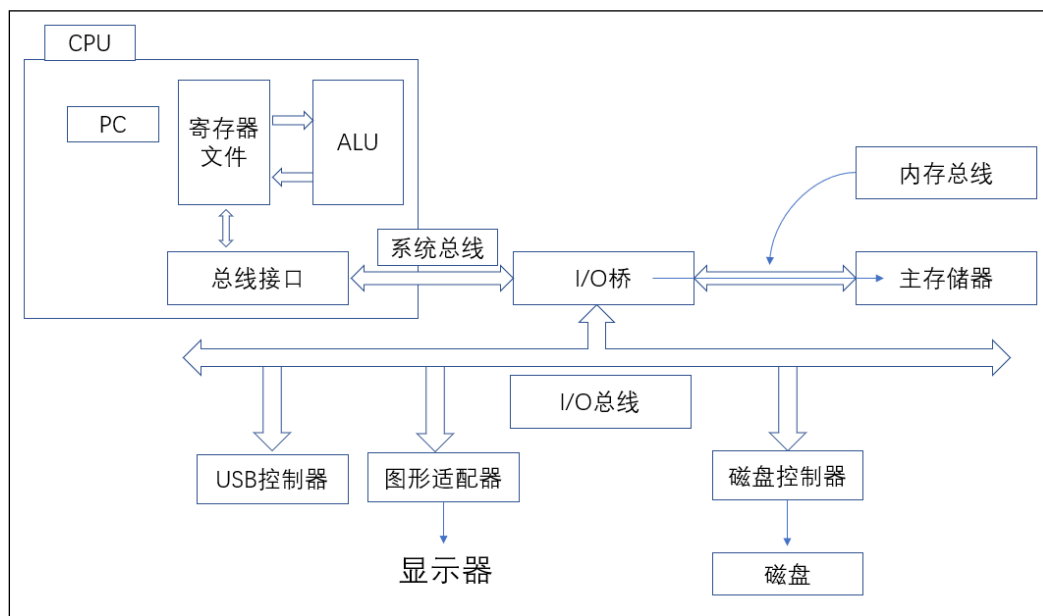


图8-2 读取命令

当回车键按下时，shell 程序就知道我们已经结束了命令的输入，然后 shell 执行一系列指令来加载可执行文件，这些指令将目标文件中的代码和数据复制到主存。图 8-2 展示了一个示例计算机系统的配置，主要部件是 CPU 芯片，我们将其称为 I/O 桥接器的芯片组，以及组成主存的 DRAM 主存模块。这些部件由一对总线连接起来，其中一条总线是系统总线，连接着 CPU 和

I/O 桥接器，另一条是内存总线，连接 I/O 桥和主存。I/O 桥将系统总线的电子信号翻译成内存总线的电子信号。

而利用 DMA（直接存储器存取），数据可以不通过处理器而直接从磁盘到达主存。这个过程如图 8-3 中的线段所示。在磁盘控制器收到来自 CPU 的读命令之后，它将逻辑块号翻译成一个扇区地址，读该扇区的内容，然后将这些内容直接传送到主存，不需要 CPU 的干涉，也就是我们所说的 DMA。

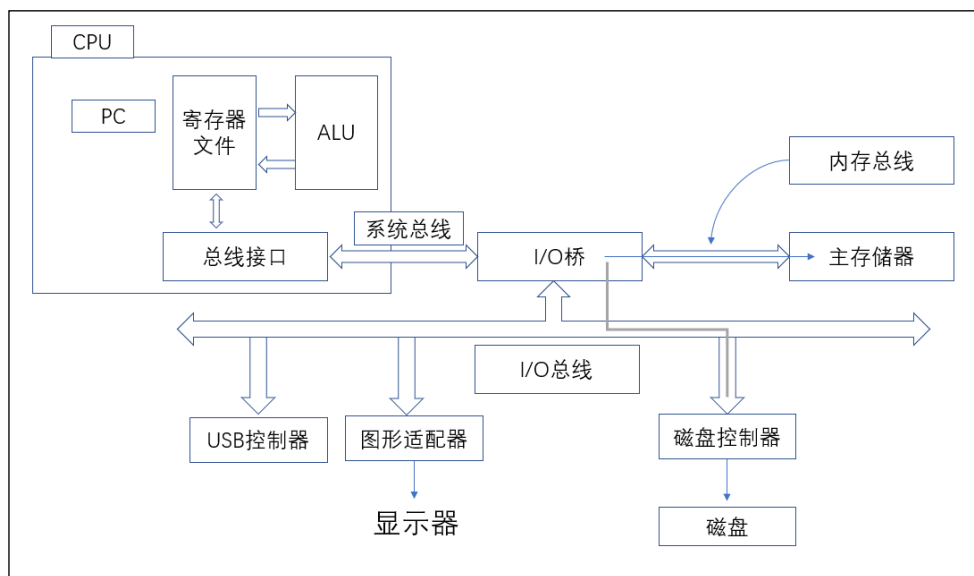


图8-3 从磁盘加载可执行文件到主存

当目标文件中的代码和数据被加载到主存，处理器就开始执行程序中的 main 程序中的机器语言指令，这些指令将代码中的输出内容从主存复制到寄存器文件，如图 8-4 所示。CPU 芯片上称为总线接口的电路在总线上发起读事务，读事务是由三个步骤组成的：

首先，CPU 将从具体地址取到的内容放到系统总线上，I/O 桥将信号传递到内存总线；

接下来，主存看到内存总线上的地址信号，从内存总线读取地址，从 DRAM 取出数据字，并将数据写到内存总线。I/O 桥将内存总线信号翻译成系统总线信号，然后沿着系统总线传递（如图 8-4 中箭头所示）；

最后，CPU 感觉到系统总线上的数据，从总线上读数据，并将数据复制到寄存器。再从寄存器文件中复制到显示设备，显示在屏幕上。

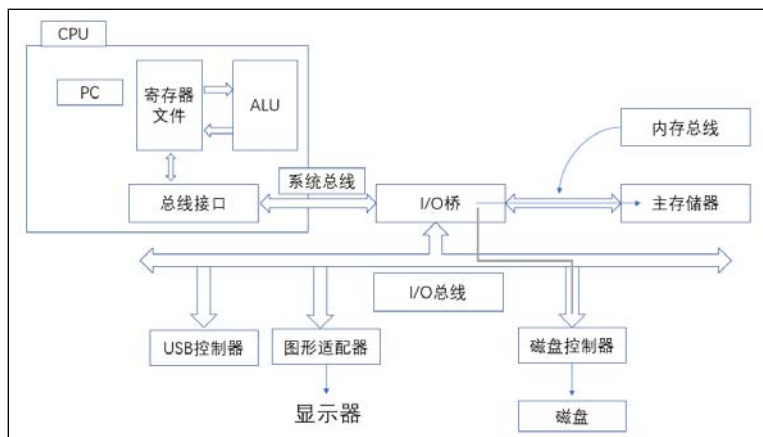


图8-4 将输出从存储器写到显示器（图形适配器）

8.4 实验任务操作指导

8.4.1 x86 冒泡排序代码迁移

8.4.1.1 环境搭建

8.4.1.1.1 安装 Visual Studio 2017

步骤 1 下载 Visual Studio 2017

在编写代码前首先确定编译器，这里使用的是 Visual Studio 2017（以下简称 VS 2017）。在修改相关属性后 VS 也可以运行汇编程序，下面介绍 VS 2017 的安装过程。

首先进行 VS 2017 的安装，打开浏览器，在网址栏输入 VS 2017 下载网址：

<https://my.visualstudio.com/Downloads?q=Visual%20Studio%202017>，进入到 VS 2017 下载界面，如图 8-5 所示。

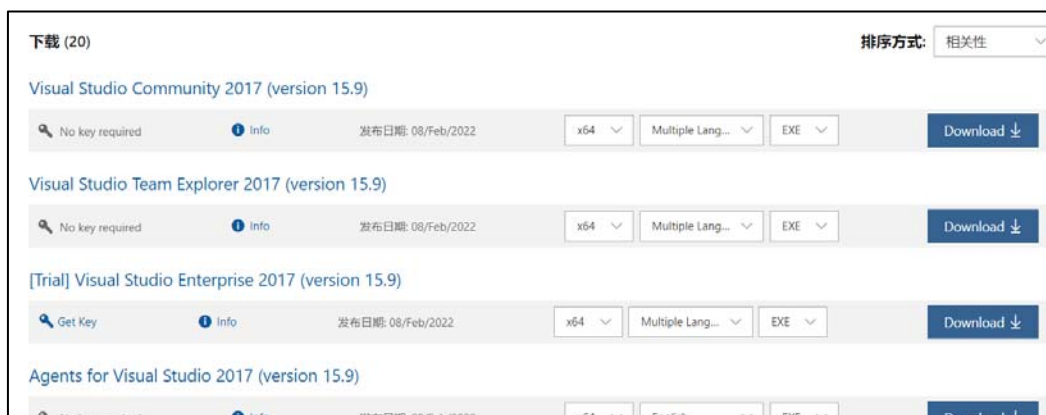


图8-5 VS 2017 下载界面

根据自己的电脑操作系统选择下载，这里下载的是 Visual Studio Community 2017（图片中第一项），点击页面中的 Download 按钮进行下载。

步骤2 配置文件属性

下载完成后按照步骤安装完成后，进入到编译器中，首先创建好本次实验需要的程序文件，本次实验首先要使用 x86 汇编实现冒泡排序功能，通过 C 调用汇编的方式实现，所以需要 .c 文件和一个 .asm 文件，首先点击左上角“文件-新建-项目”创建一个新的项目，选择 Visual C++，点击空项目，项目名称设为 BubbleSort，项目位置自定义即可，如图 8-6 所示。

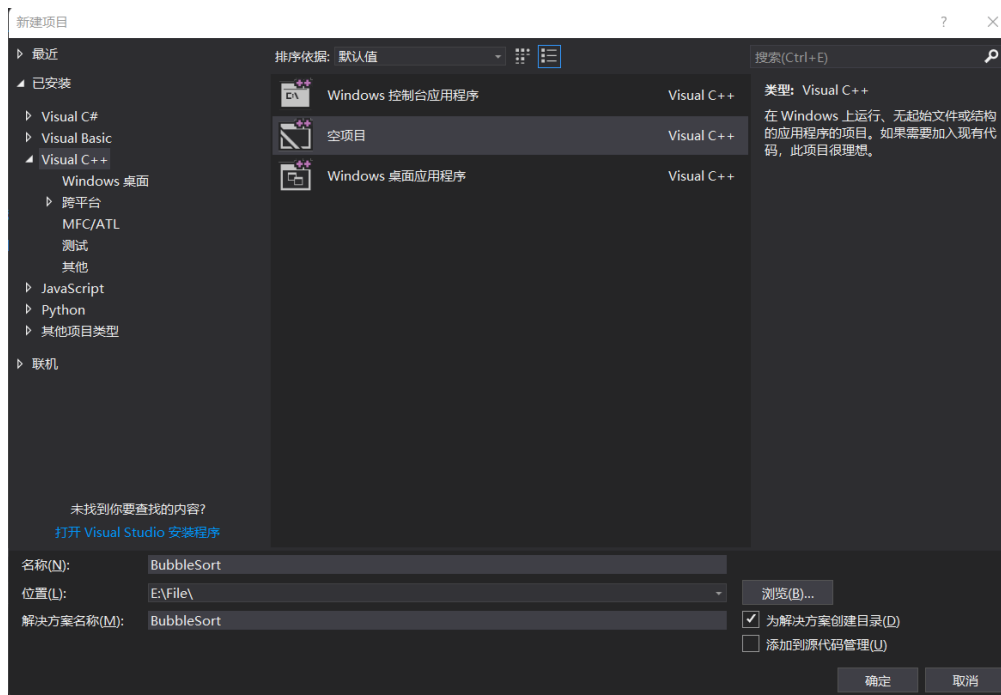


图8-6 创建项目文件

点击确定创建好项目文件后，可以看到右侧的解决方案资源管理器已经有了刚刚创建的项目，右击里面的“源文件-添加-新建项”，选择.cpp 文件添加本次的程序文件 Bubble.c 和 Bubble.asm，添加时将名称后缀改为.c 和.asm 即可，如图 8-7 所示。

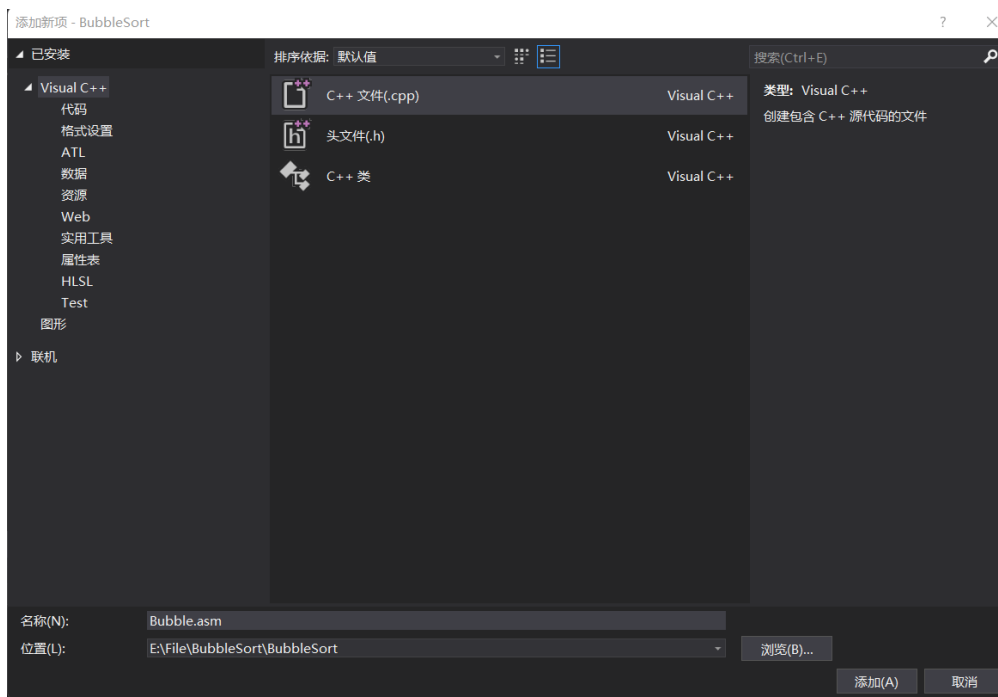


图8-7 添加程序文件

添加完成效果如图 8-8 所示。



图8-8 添加程序文件完成

步骤 3 修改文件属性

此时还不可以运行.asm 文件，右击 Bubble.asm，进入属性界面，如图 8-9 所示

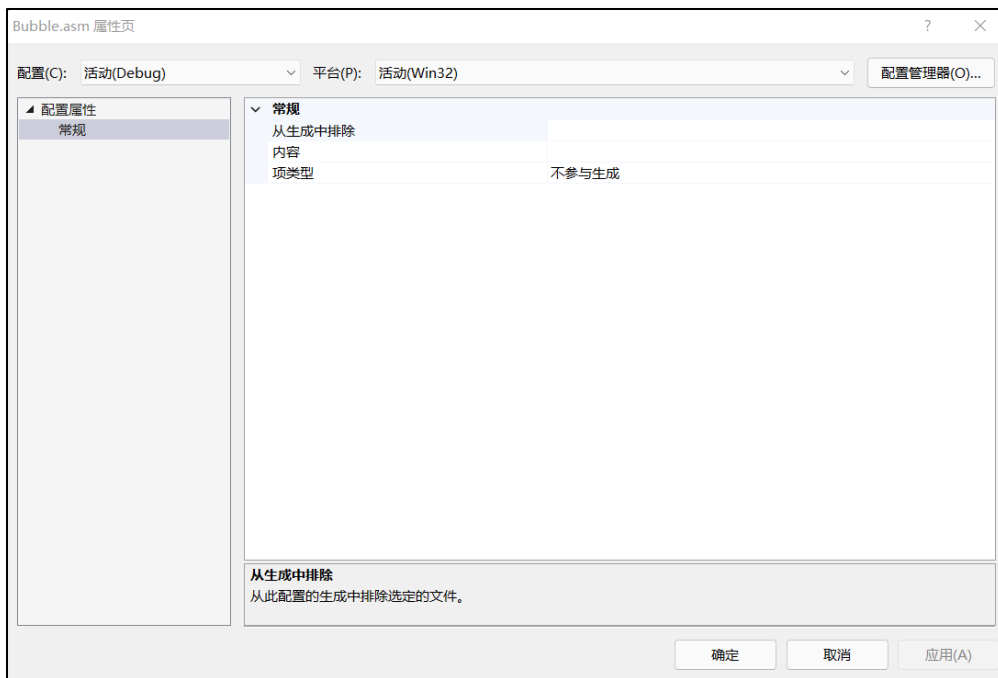


图8-9 修改配置界面

可以看到此时的项类型是不参与生成，此时如果直接生成解决方案，VS 直接忽略掉.asm 文件，链接时会发生失败。所以我们将其项类型改为自定义生成工具，从生成中排除改为“否”，点击确定。如图 8-10 所示。

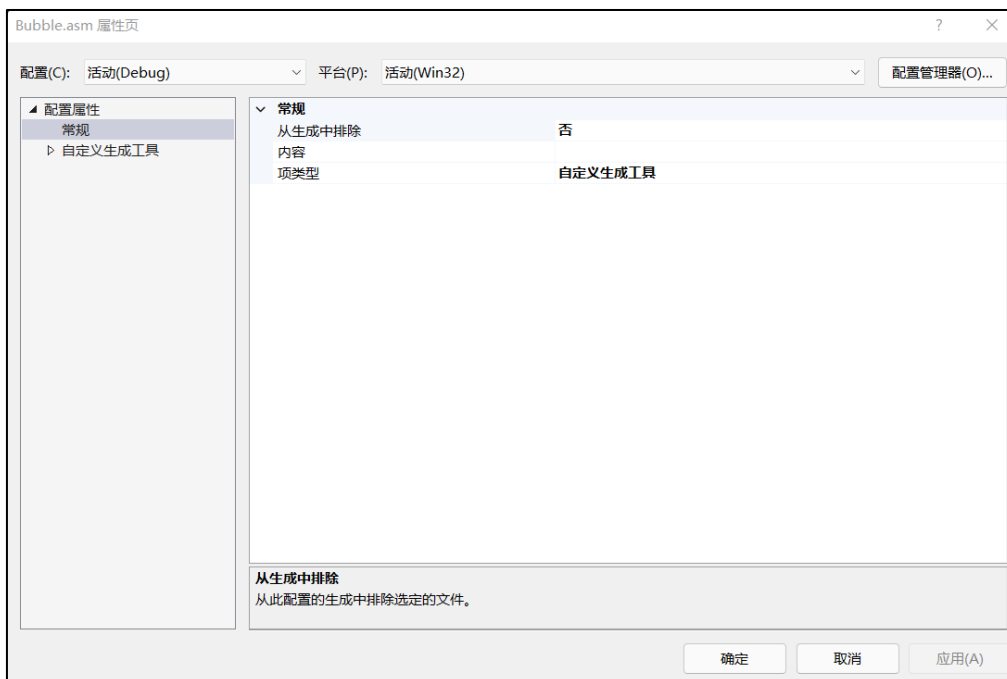


图8-10 修改程序属性

点击确定后再次进入 Bubble.asm 属性界面，点击自定义生成工具中的常规选项，如图 8-11 所示。

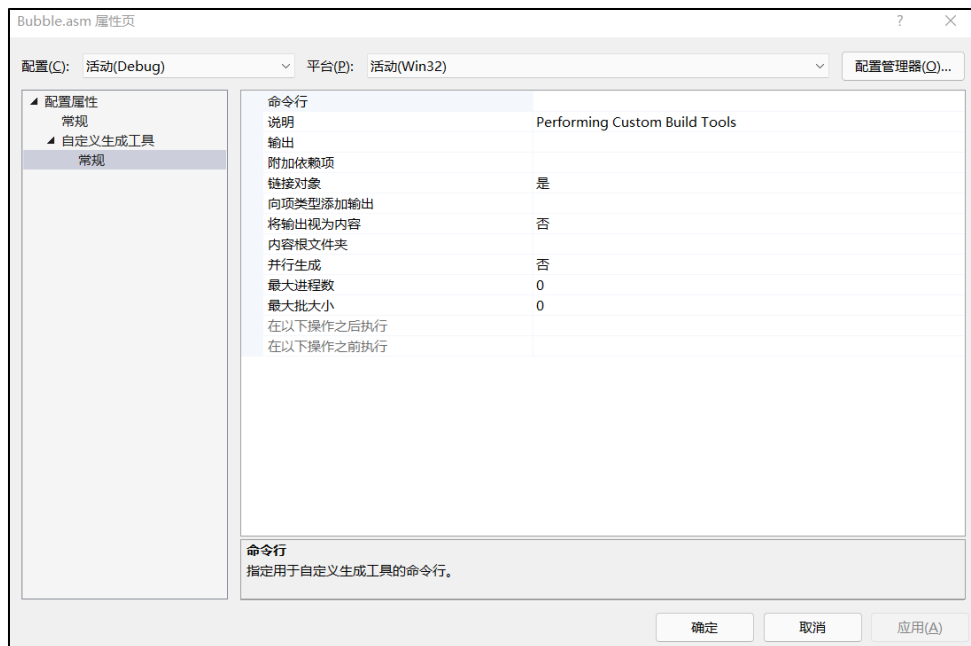


图8-11 修改程序属性

接下来对命令行和输出两个选项进行修改，在命令行一行输入以下内容：“ml /c /coff %(fileName).asm”，在输出一行输入：“%(fileName).obj;%(OutPuts)”。如图 8-12 所示。

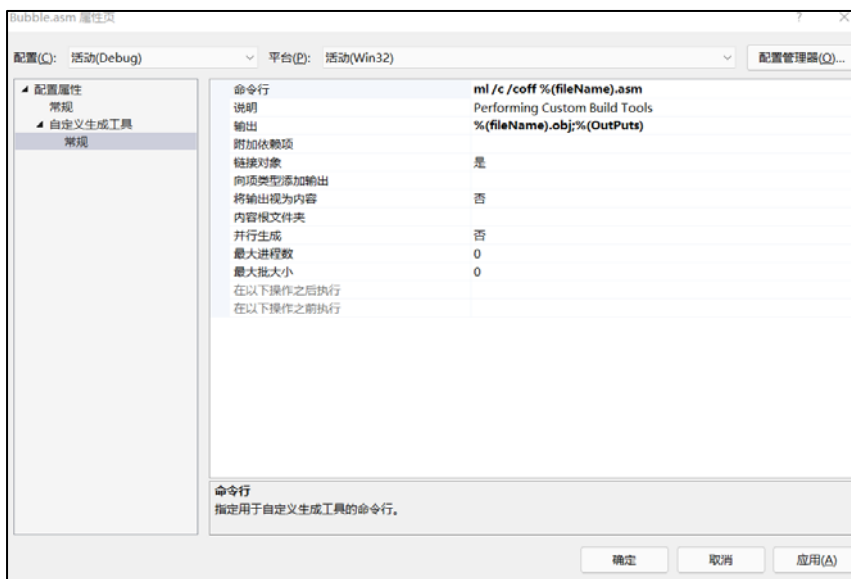


图8-12 修改程序属性

-ml 是 VS 中携带的宏汇编和链接程序，用于将 asm 转换为*.obj 文件并链接，/c 参数代表只汇编不链接，/coff 表示使用的文件格式，确认之后就可以运行了，接下来进行程序编写。

8.4.1.2 x86 汇编实现冒泡排序

步骤 1 代码编写

首先编写 Bubble.c 的内容，在这段程序中，我们定义了一个长度为 10 的数组，使用冒泡排序的方法对其进行从小到大的排序，其中冒泡排序的部分通过 Bubble 函数实现。编写的代码如下：

```
#include<stdio.h>
#include<stdlib.h>
extern int Bubble(int*arr);
int main()
{
    int Array[10] = { 1,9,7,6,4,3,0,2,8,5 };
    int i, j;
    printf("Before sort:\n");
    for (i = 0; i < 10; i++)
    {
        printf("%d ", Array[i]);
    }
    printf("\n");
    Bubble(Array);
    printf("after sort:\n");
    for (j = 0; j < 10; j++)
    {
        printf("%d ", Array[j]);
    }
    printf("\n");
    system("pause");
    return 0;
}
```

接下来编写 Bubble.asm 的内容，这段代码主要实现了冒泡排序的功能，编写的代码如下：

```
.model flat, c
.code
Bubble proc
push ebp
mov ebp,esp

sort:
    mov ebx,[ebp+8]
    mov eax,10
loop1:
    mov ecx,eax
    dec ecx
    mov edx,ecx
    mov esi,0
loop2:
    mov ecx,[ebx+esi]
    cmp ecx,[ebx+esi+4]
```

```
    jl loop3
    xchg ecx,[ebx+esi+4]
    mov [ebx+esi],ecx
loop3:
    add esi,4
    dec edx
    jnz loop2

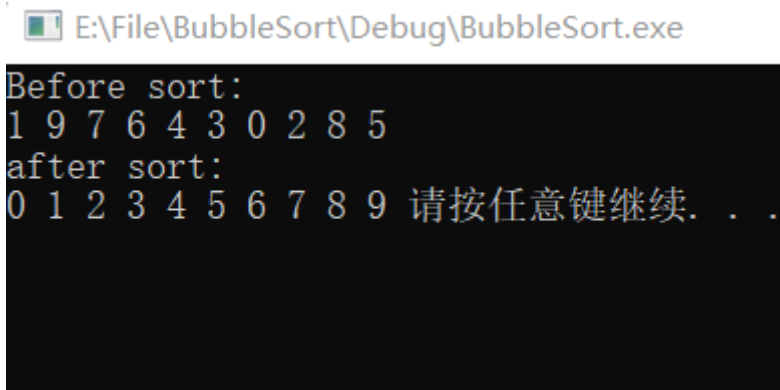
    dec eax
    cmp eax,1
    jnz loop1

    pop ebp
    ret

Bubble endp
End
```

步骤 2 运行代码

完成代码的编写后，首先 Ctrl+F7 编译，编译成功后 Ctrl+F5 运行，运行结果如图 8-13 所示。



```
E:\File\BubbleSort\Debug\BubbleSort.exe
Before sort:
1 9 7 6 4 3 0 2 8 5
after sort:
0 1 2 3 4 5 6 7 8 9 请按任意键继续. . .
```

图8-13 程序运行

成功完成排序功能，我们登录鲲鹏代码迁移工具，进行源码迁移。

步骤 3 登录鲲鹏代码迁移工具

首先打开浏览器，在地址栏输入 [https://部署服务器的 IP: 端口号](https://部署服务器的IP:端口号)，本示例使用的服务器 IP 为 114.116.194.44，端口号为 8084，登入界面如图 8-14 所示：



图8-14 代码迁移工具界面

输入用户名和密码，用户名默认为 portadmin，设置好自定义密码后，点击登录，进入代码迁移工具，如图 8-15 所示：



图8-15 迁移工具界面

步骤 4 点击源码迁移

我们点击源码迁移，进行分析，如图 8-16 所示：

分析源码

检查分析C/C++/Fortran/Go/解释型语言/汇编等源码文件，定位出需迁移代码并给出迁移指导，支持迁移编辑及一键代码替换功能。

源码文件存放路径

/opt/portadv/portadmin/sourcecode/

需要填写相对路径，可以通过以下两种方式实现：
1. 单击“上传”按钮上传压缩包（上传过程中自动解压）或文件夹；
2. 先将源码文件手动上传到服务器上本工具的指定路径下（/opt/portadv/portadmin/sourcecode/），给porting用户开读写和执行权限，再单击填写框选择下拉框中的源码路径即可，也可以手动填写源码路径，多个路径请通过“/”分隔。

上传

源码类型

☒ C/C++/ASM ☐ Fortran ☐ Go ☐ 解释型语言
汇编不支持迁移修改后再次扫描；如果扫描，会导致分析结果不准确。

目标操作系统

CentOS 7.6

目标系统内核版本

4.14.0

编译器版本

GCC 4.8.5

构建工具

make

编译命令

make

编译命令需根据构建工具配置文件确定，具体请参考[联机帮助](#)。

开始分析

图8-16 源码迁移界面

首先将代码上传到服务器上，这里采用的是第二种方式，将写好的汇编文件通过 SFTP 工具上传到服务器上，这里使用的是 WinSCP，下载地址为：<https://winscp.net/eng/docs/lang:chs>，下载安装完成后打开软件，将汇编代码上传到网页上提示的源码文件存放路径 /opt/portadv/portadmin/sourcecode/ 下，可以通过点击迁移工具页面的对话框查看是否成功上传文件。如果成功上传，对话框会弹出已上传的文件。由于我们的服务器配置的是 openEuler 操作系统，所以将操作系统修改为 openEuler 20.03，其余默认即可，如图 8-17 所示：

★ 源码类型

☒ C/C++/ASM ☐ Fortran ☐ Go ☐ 解释型语言
汇编不支持迁移修改后再次扫描；如果扫描，会导致分析结果不准确。

目标操作系统

openEuler 20.03

图8-17 分析配置

步骤 5 上传文件到鲲鹏服务器上

使用 WinSCP 登录，界面如图 8-18 所示：

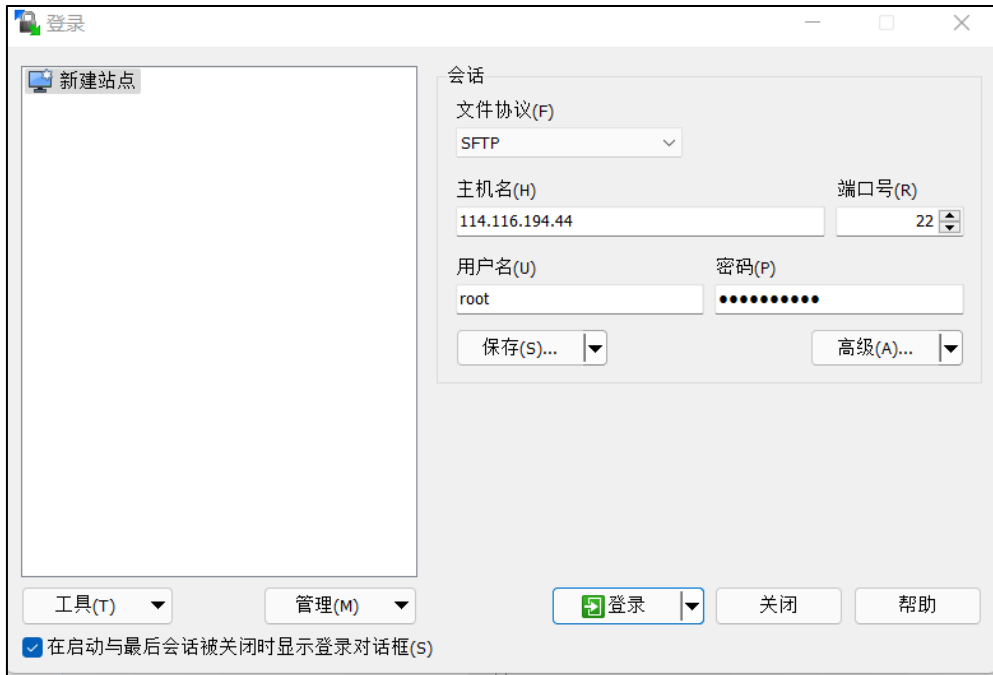


图8-18 WinSCP 登录

使用 WinSCP 将文件上传到指定目录后，分析之前仍需修改文件权限，首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，输入命令 `cd /opt/portadv/portadmin/sourcecode/` 进入目录。

```
cd /opt/portadv/portadmin/sourcecode/
```

然后看到已经上传的代码文件，输入命令 `chmod 777 Bubble.asm` 修改权限。

```
chmod 777 Bubble.asm
```

如图 8-19 所示：

```
[root@ecs-001 sourcecode]# ls
Bubble.asm
[root@ecs-001 sourcecode]# chmod 777 Bubble.asm
[root@ecs-001 sourcecode]# ls
Bubble.asm
[root@ecs-001 sourcecode]#
```

图8-19 修改权限

步骤 6 开始分析

点击“开始分析”进行分析，并产生分析报告。生成分析报告后，点击右侧产生的历史报告，将页面滑倒最下方下载报告的 html 版进行查看，如图 8-20 所示：

需要迁移的代码行数		需修改的代码行: 21行; ASM: 21 行;	
文件名	行号 (起始行, 结束行)	关键字	建议
/opt/portadv/portadmin/sourcecode/Bubble.asm	(4,4)	PUSH	This instruction has no suggestion for replacement on arm64 temporarily, which needs to be ported.
/opt/portadv/portadmin/sourcecode/Bubble.asm	(5,5)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register EBP suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(8,8)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register EBX suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(9,9)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register EAX suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(11,11)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register ECX suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(12,12)	DEC	Opcode suggested to be replaced with["SUB"] Register ECX suggested to be replaced with["W0", "...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(13,13)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register EDX suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(14,14)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register ESI suggested to be replaced ...
/opt/portadv/portadmin/sourcecode/Bubble.asm	(16,16)	MOV	Opcode suggested to be replaced with["LDR", "MOV", "STR"] Register ECX suggested to be replaced ...

图8-20 迁移建议

对于鲲鹏代码迁移工具来说，目前支持部分修改点提供移植建议修复，部分仅提示用户需要移植。所以我们可以从中看到部分代码给出了修改建议，比如部分指令的修改及寄存器的修改，但是有的指令还未提供替换的相应功能。

接下来再回到服务器上，根据迁移报告编写一段冒泡排序程序与之进行对比分析。

8.4.2 ARM 汇编实现冒泡排序

接下来，我们再根据之前的迁移报告的修改建议，使用 ARM64 汇编来实现一个冒泡排序。

步骤 1 新建目录和文件

本示例同样通过 C 语言调用汇编的方式来实现冒泡排序程序。首先使用远程登录工具，登录到鲲鹏 ECS 服务器上，进入到命令行后，我们先输入命令 `mkdir Bubblesort` 创建一个新的文件夹，然后输入命令 `vim Bubble.c`, `vim Bubble.s` 创建两个程序文件。

```
mkdir Bubblesort
vim Bubble.c
vim Bubble.s
```

并编写以下程序代码：

```
#include<stdio.h>
extern void Bubble(int*array);
int main()
{
    int array[10]={1,9,7,6,4,3,0,2,8,5};
    int i,j;
    printf("Before sort:\n");
    for(i=0;i<10;i++)
    {
        printf("%d ",array[i]);
    }
    printf("\n");
    Bubble(array);
    printf("after sort:\n");
    for(j=0;j<10;j++)
```

```
{  
    printf("%d ",array[j]);  
}  
printf("\n");  
return 0  
}
```

首先是 Bubble.c 的内容，这部分内容与之前的完全一致。

接下来是汇编代码文件 Bubble.s 的内容：

```
.text  
.global Bubble  
Bubble:  
    mov x7,#10  
loop1:  
    mov x2,x0  
    mov x6,x7  
    sub x6,x6,#1  
    mov x8,x6  
  
loop2:  
    ldr w3,[x2]  
    add x4,x2,#4  
    ldr w5,[x4]  
    cmp w3,w5  
    bls loop3  
    str w3,[x4]  
    str w5,[x2]  
loop3:  
    adds x2,x2,#4  
    sub x8,x8,#1  
    cmp x8,#0  
    bne loop2  
    sub x7,x7,#1  
    cmp x7,#1  
    bne loop1  
  
ret
```

汇编代码的编写思路如下：我们想将数组从小到大进行排序，首先初始化外层循环次数，接着将初始地址放入 x2 寄存器中，x7 寄存器记录总共要进行冒泡的趟数，x6 寄存器记录每趟冒泡需要比较的次数。loop2 开始内层循环，通过 adds 指令将 x4 的地址从基址增加移动到下一个数。然后通过 cmp 指令将两数相减进行比较，如果小于的话，那么就不需要进行交换，bls 指令就会跳转到 loop3 循环进行下一次比较，否则就进行交换。

步骤 2 编译并运行可执行文件

输入命令 `gcc Bubble.s Bubble.c -o Bubble`, 对程序进行编译, 生成可执行文件 `Bubble`, 输入指令 `./Bubble` 执行文件。

```
gcc Bubble.s Bubble.c -o Bubble
./Bubble
```

得到的结果如图 8-22 所示:

```
Before sort:
1 9 7 6 4 3 0 2 8 5
after sort:
0 1 2 3 4 5 6 7 8 9
```

图8-21 执行结果

根据执行结果, 可以看出排序成功, 接下来我们对比一下 ARM 冒泡排序和 x86 冒泡排序用到的指令。

8.4.3 迁移分析

通过查看鲲鹏代码迁移工具给出的迁移报告, 对于可以修改的代码, 迁移报告会在其下面生成一条红色的波浪, 首先观察对比的是两段汇编中的第一段代码段, x86 中的 `sort` 部分和 ARM 代码中的 `Bubble` 部分, 这两段代码的功能都是传入数组的首地址, 并定义本次排序比较的趟数, 本例介绍指令迁移报告如图 8-22 所示。可以看到对于 `mov` 指令, 迁移报告给出的建议是数据传输指令 `mov` 和加载数据指令 `ldr`, 存储数据指令 `str`, 这里我们主要是做数据传输的功能, 所以同样使用 `mov` 指令进行替换, 对于寄存器的修改建议, ARM v8 架构提供了 32bit 的通用寄存器 `w0-w30`, 64bit 的通用寄存器 `x0-x30`, 如果有需要的话可以当作 32bit 使用。

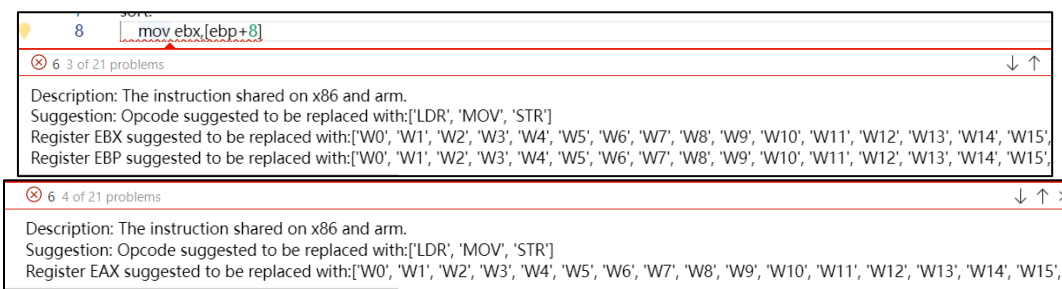


图8-22 分析报告

接着是 `loop1` 段对比, 这一段主要是定义了内层循环的次数, 也就是每趟冒泡要比较的次数, 在 x86 的代码中使用 `edx` 寄存器, ARM 代码中使用 `x8` 寄存器存储。对于 `dec` 指令, 迁移报告给出的建议是 `sub`, `sub` 是 ARM 中的减法指令, `dec` 指令的功能是自减一, 这里我们使用 `sub` 指令将 `x6` 寄存器中的值减 1 并传入 `x8` 寄存器中, 对应 x86 代码中的 `ecx` 寄存器中的值自减一后传入 `edx` 寄存器中。本例介绍指令迁移报告如图 8-23 所示。



图8-23 分析报告

本例介绍指令迁移报告如图 8-24 所示，对于 loop2 段，这段是冒泡的主要部分，由于我们要实现从小到大的排序，在对比相邻的两个数组元素后，若前一位元素小于后一位则直接跳转到下一对比较，如果前一位元素小于后一位元素，则需要对其进行交换。在本段中，“mov ecx,[ebx+esi]”是将数组元素放入到 ecx 寄存器中，对于迁移的 ARM 代码使用 ldr 指令将数组中的元素加载出来，可以看到此处修改为了“ldr w3,[x2]; add x4,x2,#4 ; ldr w5,[x4]”，将前后的两个元素分别放入 w3, w5 寄存器中。接着比较两个元素的代码迁移报告给出的结果同样是 cmp 指令，在比较完后，如果前一位元素小于后一位元素，则跳转到下一对比较，此处迁移报告并没有给出迁移建议，说明对于一些指令还无法做到完全迁移。本示例中用到的跳转指令如：JNZ, JMP, JE, JL 指令，分别表示：JNZ 表示当 z 标识位为 0，操作数不相等时跳转；JE 当 z 标识位为 1 时，两操作数相等时跳转；JL 指令是对有符号数进行判断，当源操作数小于目的操作数时跳转。JMP 指令是 x86 中的无条件跳转指令。

ARM 中的跳转指令 B 可以跳转到指定的地址。本示例中用到的 BNE 指令即当源操作数和目的操作数不相等的时候跳转；BLS 指令表示当源操作数大于目的操作数时，则跳转。如程序中语句 CMP w3, w5, w3 中的值与 w5 中的值相减，当 w3 中的值小于 w5 中的值时，则不会跳转。

在比较完成后如果前一位元素大于后一位元素，则须进行交换（示例为从小到大排序）。x86 代码中使用的是 xchg 指令，迁移报告给出的是使用 swp 指令，swp 指令也是将存储器中的值和指定存储器进行交换，但在 ARM v6 及以后版本中就没有采用了，所以这里我们使用 str 指令，将 w3 寄存器中的值存到 x4 中，将 w5 中的值存放到 x2 中，这样就完成了一次交换。

6 10 of 21 problems	↓ ↑
Description: The instruction shared on x86 and arm. Suggestion: Opcode suggested to be replaced with:['CMP'] Register ECX suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15', Register EBX suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15', Register ESI suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15',	
6 11 of 21 problems	↓ ↑ ×
Description: This is x86 or arm32 instructions. Suggestion: This instruction has no suggestion for replacement on arm64 temporarily, which needs to be ported. (10)	
6 12 of 21 problems	↓ ↑
Description: This is x86 or arm32 instructions. Suggestion: Opcode suggested to be replaced with:['SWP'] Register ECX suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15', Register EBX suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15', Register ESI suggested to be replaced with:['W0', 'W1', 'W2', 'W3', 'W4', 'W5', 'W6', 'W7', 'W8', 'W9', 'W10', 'W11', 'W12', 'W13', 'W14', 'W15',	

图8-24 迁移建议

最后是 loop3 代码段，loop3 主要也是控制循环的部分，进入 loop3 后，说明一对数组元素的比较已完成，要进行下一对的比较，首先要将当前数组元素的指针加 4，因为我们定义的是 int 型数组每个元素为 4 字节。可以看到对于 add 指令迁移报告给出的是修改为 adds，adds 指令同时可以将进位结果写入 CPSR(程序状态寄存器)，这里使用普通的 add 指令即可，但 adds 也可成功运行。接着还是使用 cmp 指令判断是否完成一趟比较，未完成则继续跳转到 loop2

继续比较，完成的话就将总比较趟数减一在跳回到 loop1 进行下一次比较。本例介绍指令迁移报告如图 8-25 所示。

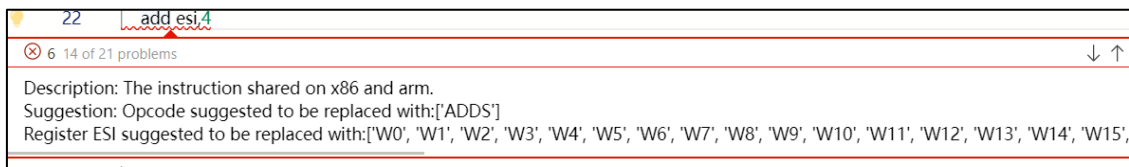


图8-25 迁移建议

最终完成所有排序后 ret 返回，根据 8.4.2 结果可以看到所有排序都成功实现，实验完成。

9 附录 1：Linux 常用命令

9.1 基本命令

9.1.1 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

9.1.2 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

9.2 2 目录操作命令

9.2.1 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

9.2.2 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

9.2.3 目录操作命令

9.2.3.1 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

9.2.3.2 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

9.2.3.3 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。

9.2.3.4 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

9.3 文件操作命令

9.3.1 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

9.3.2 删除文件

命令：rm -rf 文件名

9.3.3 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

9.3.4 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
```

10 附录 2: ARM 指令

10.1 LDR 字数据加载指令

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。

10.2 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。

10.3 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, 从而可以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。

10.4 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。

10.5 STRB 字节数据存储指令

STRB 指令的格式为:

STRB{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。

10.6 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。

10.7 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。

说明：《计算机组成与结构》课程配套实验手册中的实验内容由大连理工的赖晓晨老师提供, 华为公司负责实验手册文档的编写。