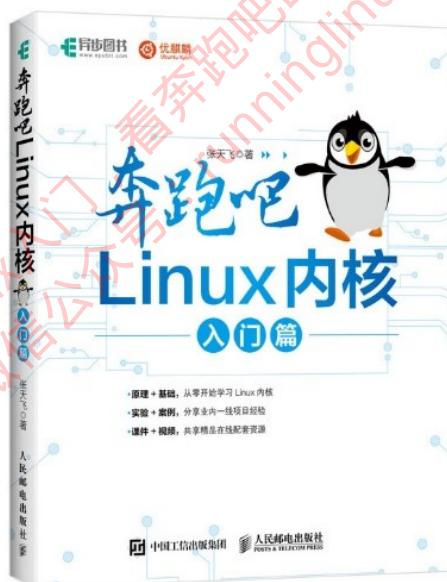


奔跑吧 Linux 内核 * 入门篇

实验指导手册



奔跑吧 Linux 社区出品

内部培训资料

2019-10

前 言

版本说明

日期	作者	版本	说明
2019-09-03	笨叔	1.0	第一版实验指导手册 vmware镜像版本: v4
2019-09-16	笨叔	1.1.1	增加3个小实验: 第一章 2个, 第三章1个 修订一些错别字
2019-09-19	笨叔	1.2	新增第4章实验4 新增附录7 – ARM64指令集 第7章实验6增加进阶思考题
2019-10-10	笨叔	1.3	新增第7章实验9 – 特工实验
2019-12-18	笨叔	1.3.1	更新第7章实验9的实验代码

To 高校老师:

《奔跑吧 Linux 内核*入门篇》配备了 600 多页的 ppt 课件, 可以免费提供给高校老师 ppt 源文件。

笨叔联系方式: runninglinuxkernel@126.com.

请用工作邮箱联系 (edu 域名的邮箱)。来信请告知学校名称, 院系, Linux 开课情况等。

版权声明:

本实验指导手册版权归作者所有。

本实验指导手册 pdf 版本可以免费和自由下载以及自由传阅。

前 言

2019 年 2 月白色《奔跑吧 Linux 内核》入门篇一书出版之后得到了广大 Linux 爱好者和高校老师同学们的喜爱。很多老师和同学在使用本书作实验时候，或多或少遇到一些小问题，于是给笨叔提了很多很棒的建议。特别是给笨叔指出了书中不少地方或者实验步骤语焉不详的。于是，笨叔觉得有必要写一本配套的实验指导手册。

这本配套的实验指导手册包含如下内容：

- 更详细的实验步骤。考虑到不少同学是第一次接触 Linux 系统，对书中很多省略的步骤会产生困扰，因此本实验指导手册希望能把步骤写的更加详细。
- 添加实验描述和实验代码分析。在《奔跑吧 Linux 内核》入门篇一书中，并没有对实验做详细的解释，只是在 [github](#) 中提供了参考代码。很多小伙伴希望笨叔能对每个实验做一些详细的分析和解答。
- 添加更多有趣的实验。比如笨叔最新研制的 QEMU+Debian 系统，读者可以在 QEMU 中运行 Debian 系统，通过 apt 命令在线安装软件。比如在每章中，尽可能添加更多有趣的实验。

另外，Linux 内核入门只需要《奔跑吧 Linux 内核*入门篇》一本书就够了。

入门篇的特色：

- ✧ 从 0 开始学习内核
- ✧ 学习最新开发工具，如 Vim 8 和 Git
- ✧ 内容循序渐进，深入浅出学习 Linux 内核
- ✧ 70 多个创新实验，突出动手能力
- ✧ 融入最新开源社区发展理念和资讯
- ✧ 秘制 O0 优化等级编译的内核
- ✧ 提供精美课件，约 600 页 ppt
- ✧ 提供实验参考代码
- ✧ 提供全套实验环境 - 基于优麒麟 Linux 18.04 系统的虚拟机镜像
- ✧ 提供全套实验环境 - docker 镜像
- ✧ 提供免费补充高清视频
- ✧ 提供免费补充音频节目（喜马拉雅）。
- ✧ 免费的知识星球。
- ✧ 进阶图书。
- ✧ 进阶视频课程。

需要说明一下，本实验指导手册采用不定期更新，重要一点是免费，大家可以免费下载和自由传播。最后，希望大家对《奔跑吧 Linux 内核》入门篇这本书多多提意见和建议。

前 言

最后，《奔跑吧 Linux 内核*入门篇》还配备了 33 集免费的补充配套视频，以及收费的进阶视频。免费视频可以到 B 站在线观看，收费进阶视频则在淘宝上。

笨叔的邮箱：runninglinuxkernel@126.com。

奔跑吧 linux 社区微信公众号：runninglinuxkernel

淘宝店：<https://shop115683645.taobao.com/>

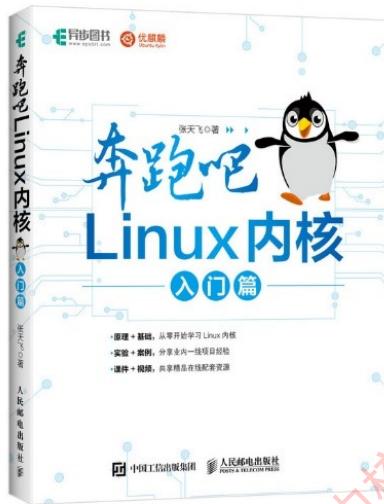


奔跑吧linux社区微信公众号

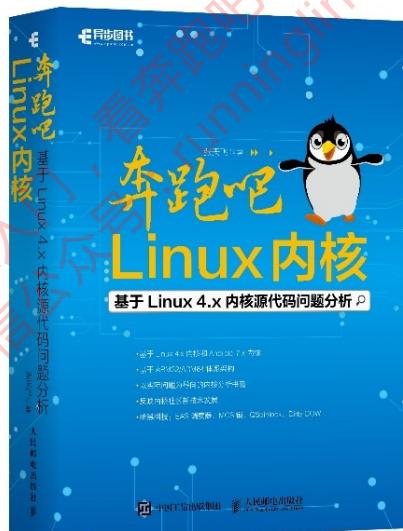
下载配套资料：

登录“奔跑吧 linux 社区”微信公众号，在微信公众号里输入：

- 输入“全套资料”，下载奔跑吧入门篇全套资料。
- 输入“实验指导手册”，下载本实验指导手册最新的 pdf 版本。
- 输入“免费视频”，下载奔跑吧入门篇免费配套视频。
- 输入“进阶视频”，进入进阶视频购买链接。
- 输入“vmware 镜像”，下载最新的实验指导手册配套实验平台 vmware 镜像。
- 输入“芯片手册”，下载 ARM64、x86、RISC-V 等芯片资料手册



奔跑吧Linux内核入门篇



奔跑吧Linux内核进阶篇

目 录

目 录

奔跑吧 LINUX 内核 * 入门篇	1
实验指导手册	1
版本说明	2
前 言	3
第 0 章	13
实验前准备	13
1. 关于教材	13
2. 关于实验平台	14
3. 实验代码	17
4. 体系结构支持	18
5. 系统默认配置	18
6. 学习建议	20
第 1 章	21
LINUX 系统入门	21
1.1 实验 1：在虚拟机中安装优麒麟 Linux 18.04 系统	21
1.2 实验 2：给优麒麟 Linux 系统更换心脏	24
1.3 实验 3：使用 O0 编译的内核 - runninglinuxkernel	34
1.4 实验 4：如何编译和运行一个 ARM Linux 内核	39
1.5 实验 5：运行 Debian+ARM32 系统（新增）	41
1.6 实验 6：运行 Debian+ARM64 系统（新增）	46

1.7 实验 7: 运行 Debian+X86_64 系统 (新增)	51
1.8 实验 8: 手把手制作一个 Debian rootfs 系统 (新增)	55
1.9 实验 9: 配置 QEMU 虚拟机的桥接网络 (新增)	61
1.10 实验 10: 动手 DIY 一个 RISC-V 的 Debian 系统 (新增)	68
第 2 章	70
LINUX 内核基础知识	71
2.1 实验 1: GCC 编译	71
2.2 实验 2: 内核链表	74
2.3 实验 3: 红黑树	74
2.4 实验 4: 使用 Vim 工具	74
2.5 实验 5: 把 Vim 打造成一个强大的 IDE 编辑工具	75
2.6 实验 6: 建立一个 git 本地仓库	82
2.7 实验 7: 解决合并分支冲突	84
2.8 实验 8: 利用 git 来管理 Linux 内核开发	86
2.9 实验 9: 利用 git 来管理项目代码	89
第 3 章	95
内核编译和调试	95
3.1 使用 O0 优化等级编译内核的好处	95
3.2 实验 1: 通过 QEMU 调试 ARM Linux 内核	96
3.3 实验 2: 通过 QEMU 调试 ARMv8 的 Linux 内核	98
3.4 实验 3: 通过 Eclipse+QEMU 单步调试内核	106
3.5 实验 4: 在 QEMU 中添加文件系统的支持	114
3.6 实验 5: 使用 DS-5 单步调试 arm64 内核	115
第 4 章	122
内核模块	122

目 录

4.1 实验 1: 编写一个简单的内核模块	122
4.2 实验 2: 向内核模块传递参数	125
4.3 实验 3: 在模块之间导出符号	126
4.4 实验 4: 在优麒麟系统中编译内核模块（新增）	127

第 5 章 129

简单的字符设备驱动 129

5.1 实验 1: 从一个简单的字符设备开始	130
5.2 实验 2: 使用 misc 机制来创建设备	136
5.3 实验 3: 为虚拟设备编写驱动	139
5.4 实验 4: 使用 KFIFO 改进设备驱动	143
5.5 实验 5: 把虚拟设备驱动改成非阻塞模式	148
5.6 实验 6: 把虚拟设备驱动改成阻塞模式	151
5.7 实验 7: 向虚拟设备中添加 I/O 多路复用支持	157
5.8 实验 8: 为什么不能唤醒读写进程	164
5.9 实验 9: 向虚拟设备中添加异步通知	165

第 6 章 176

系统调用 176

6.1 实验 1: 在 ARM32 机器上新增一个系统调用	176
6.2 实验 2: 在优麒麟 Linux 机器上新增一个系统调用	180

第 7 章 182

内存管理 182

7.1 实验 1: 查看系统内存信息	182
7.2 实验 2: 获取系统的物理内存信息	184
7.3 实验 3: 分配内存	189

7.4 实验 4: slab	196
7.5 实验 5: VMA	202
7.6 实验 6: mmap.....	207
7.7 实验 7: 映射用户内存.....	214
7.8 实验 8: OOM	225
7.9 实验 9: 特工队: 动态修改计算机的系统调用 (新增)	233
7.10 小结 (新增)	243
第 8 章	246
进程管理	247
8.1 实验 1: fork 和 clone	247
8.2 实验 2: 内核线程.....	248
8.3 实验 3: 后台守护进程.....	251
8.4 实验 4: 进程权限.....	254
8.5 实验 5: 设置优先级.....	257
8.6 实验 6: per-cpu 变量.....	259
第 9 章	262
同步管理	262
9.1 实验 1: 自旋锁.....	263
9.2 实验 2: 互斥锁.....	268
9.3 实验 3: RCU	272
第 10 章	276
中断管理	276
10.1 实验 1: tasklet	277
10.2 实验 2: 工作队列.....	280

目 录

10.3 实验 3: 定时器和内核线程	284
---------------------------	-----

第 11 章.....287**调试和性能优化.....287**

11.1 实验 1: printk.....	287
11.2 实验 2: 动态输出	287
11.3 实验 3: procfs	287
11.4 实验 4: sysfs	291
11.5 实验 5: debugfs.....	297
11.6 实验 6: 使用 frace	300
11.7 实验 7: 添加一个新的跟踪点	300
11.8 实验 8: 使用示踪标志	303
11.9 实验 9: 使用 kernelshark 来分析数据.....	307
11.10 实验 10: 分析 oops 错误.....	309
11.11 实验 11: 使用 perf 工具来进行性能分析	313
11.12 实验 12: 采集 perf 数据生成火焰图	314
11.13 实验 13: 使用 slub_debug 检查内存泄漏	316
11.14 实验 14: 使用 kmemleak 检查内存泄漏	323
11.15 实验 15: 使用 kasan 检查内存泄漏	328
11.16 实验 16: 使用 valgrind 检查内存泄漏	331
11.17 实验 17: 使用 lkp-tests 工具进行性能测试	333
11.18 实验 18: kdump 死机实战 1: 运行和配置 kdump （新增）	334
11.19 实验 19: kdump 死机实战 2: 访问已经删除的链表（新增）	340
11.20 实验 20: kdump 死机实战 3: 内存 bug（新增）	340
11.21 实验 21: kdump 死机实战 4: 死战驱动死机问题（新增）	340
11.22 实验 22: kdump 死机实战 5: 在 Centos 7.6 上安装和配置 kdump （新 增）	341
11.23 实验 23: kdump 死机实战 6: 实战 arm64 死机（新增）	343
11.24 死机问题进阶	347

第 12 章.....348

开源社区	348
12.1 实验 1: 使用 cppcheck 检查代码	348
12.2 实验 2: 提交第一个 Linux 内核补丁	349
12.3 实验 3: 管理和提交多个补丁组成的补丁集	351
12.4 实验 4: 在 github 中创建和管理一个开源项目	355
 第 13 章	356
 块设备和文件系统（新增）	356
13.1 实验 1: 块设备实验	356
13.2 实验 2: 动手写一个简单文件系统	356
 附录 1 – 免费视频	356
 附录 2 – 进阶视频课程	357
1. 第一季内存管理篇	358
2. 第二季内存管理篇	361
3. 死机黑屏专题	365
4. Git 专题	366
5. Vim 专题	366
 附录 3 – X86_64 体系结构基础	367
 附录 4 – ARM64 体系结构基础	370
1 ARMv8-A 架构介绍	370
2 常见的 ARMv8 处理器	370
3 ARM64 基本概念	371
4 ARMv8 处理器运行模式	372

目 录

附录 5 – ARM64 寄存器 373

1. 通用寄存器	373
2. 处理器状态 (Processor State)	374
3. 特殊寄存器	375
4. 系统寄存器	378

附录 6 – 常见 CRASH 命令 378

附录 7 – ARM64 指令集 386

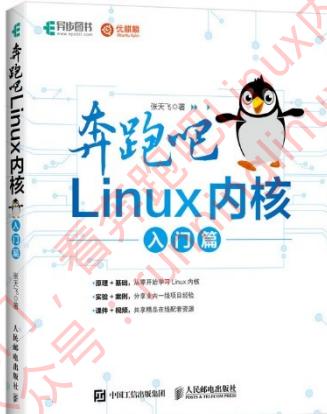
1 算术和逻辑操作指令	387
2 乘和除操作指令	389
3 移位操作	391
4 位操作指令	392
5 条件操作	393
6 内存加载指令	395
7 多字节内存加载和存储指令	398
8 非特权访问级别的加载和存储指令	398
9 内存屏障指令	398
10 独占访存指令	399
11 跳转指令	400
12 异常处理指令	401
13 系统寄存器访问指令	402

第 0 章

实验前准备

1. 关于教材

本实验指导手册是《奔跑吧 Linux 内核*入门篇》一书的配套实验书，请读者在使用本手册之前，先购买该教材的纸质书。各大网店有售。

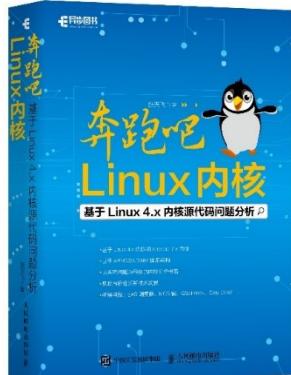


京东: <https://item.jd.com/12546036.html>

当当: <http://product.dangdang.com/26514293.html>

做完本手册所有实验并学有余力的同学可以继续阅读《奔跑吧 Linux 内核》进阶篇。

关于实验平台



2. 关于实验平台

《奔跑吧 Linux 内核 * 入门篇》的配套实验平台采用优麒麟 18.04 系统^①。建议初学者使用本书提供的 vmaware 镜像。等以后对 Linux 系统熟悉之后，再物理主机上安装优麒麟系统并配置开发环境。

本书采用的实验板子是基于软件模拟器 QEMU。QEMU 是免费软件，广泛应用于虚拟化和硬件仿真。QEMU 可以模拟很多体系结构以及硬件板子，包括 ARM Cortex A9 系列的 Vexpress 板子、支持 ARM64 的 QEMU 自带的 Virt 板子，以及 RISC-V 的板子等。读者使用本书实验平台不需要额外花钱购买硬件板子就可以完成本书所有实验。

笔者为各位读者创建了一个 vmware player 的镜像文件，并在该虚拟机中配置好本书所有实验需要的环境变量和实验素材。读者只需要下载此镜像文件，使用 VMware player 软件打开即可。

vmware 镜像文件下载地址：

https://pan.baidu.com/s/1XlJzdUBH_V7IwNRH1R4jrg

截止 2019 年 9 月，最新版本是 v4.0 版本。**本实验手册以此版本 vmware 镜像为例进行说明。**

^① 也可以使用 Ubuntu 18.04 系统。读者若使用其他 Linux 发行版或者其他版本的 Ubuntu/优麒麟系统，遇到问题，请自行解决。

全部文件 > 奔跑吧Linux内核入门篇-vmware镜像

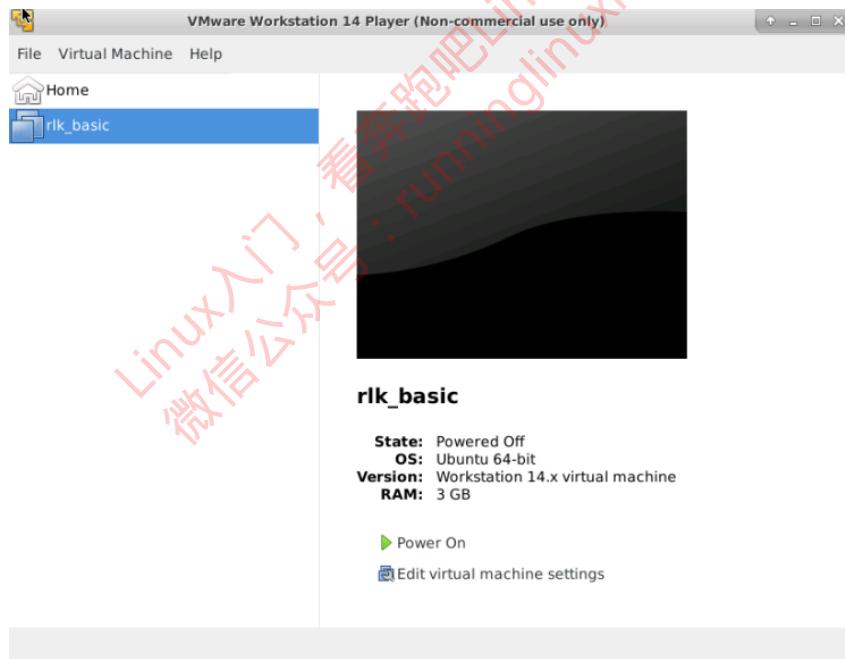


图 vmware镜像

关于实验平台

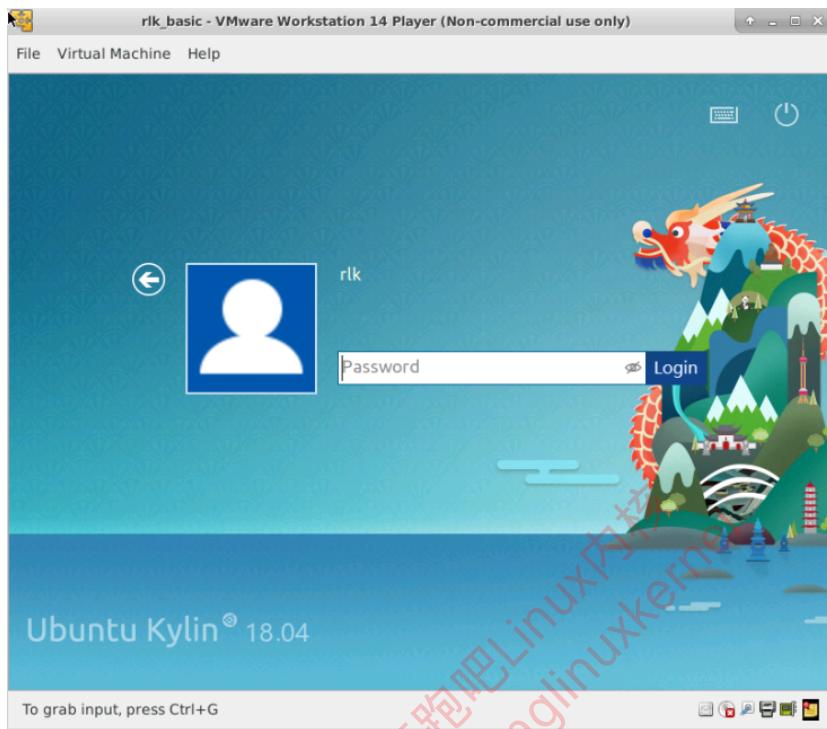


图 vmware虚拟机

vmware 镜像的登录用户名为 rlk，登录密码为“123”。

登录 Linux 界面之后第一件事情是更新最新的实验代码

```
$ cd /home/rlk/rlk_basic/runninglinuxkernel_4.0  
$ git pull
```

**若读者想自行搭建实验环境，可以到优麒麟官网下载 18.0.4 版本。
(优麒麟官网地址：<https://www.ubuntukylin.com>)**



优麒麟官网

国内读者可以通过国内镜像地址镜像下载，如：

<http://nudt.dl.360tpcdn.com/data/ubuntukylin-18.04.3-enhanced-amd64.iso>

若读者无法在电脑主机上安装优麒麟 18.04 系统，那么可以考虑采用 VMware Player 虚拟机来进行安装。VMware Player 软件对于个人用户是免费的。请读者到 VMware 官网上下载。

3. 实验代码

《奔跑吧 Linux 内核 * 入门篇》配套实验代码存放在 github 上，读者可以使用 git 命令进行克隆下载。

注意：实验代码已经从 Gitee 切换到 Github，Gitee 上代码树已经不在维护。

```
$ git clone https://github.com/figozhang/runninglinuxkernel\_4.0.git -b rlk_basic
```

欢迎大家提 git pull request。

在 vmware 虚拟机里，已经下载好了 runninglinuxkernel 代码是在：
/home/rnk/rnk basic/runninglinuxkernel 4.0 目录。

体系结构支持

```
rlk@ubuntu:~$ cd rlk_basic/runninglinuxkernel_4.0/
rlk@ubuntu:runninglinuxkernel_4.0$ ls
arch      firmware    ipc      Makefile   rootfs_debian_arm32.tar.xz  samples  '实验前必读 - 重要说明_v1.3.docx'
block     fs          Kbuild  mm        rootfs_debian_arm64.tar.xz  scripts  '实验前必读 - 重要说明_v1.3.pdf'
COPYING   include     Kconfig  net       rootfs_debian_x86_64.tar.xz security
CREDITS   init       kernel  README    run_debian_arm32.sh  sound
crypto    _install_arm32 kmodules README.md  run_debian_arm64.sh  tools
Documentation _install_arm64 lib      REPORTING-BUGS  run_debian_x86_64.sh  usr
drivers    install_x86 MAINTAINERS rtk_lab  run.sh    virt
rlk@ubuntu:runninglinuxkernel_4.0$ pwd
/home/rlk/rlk_basic/runninglinuxkernel_4.0
rlk@ubuntu:runninglinuxkernel_4.0$
```

本书配套的参考实验代码是在:

/home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic

4. 体系结构支持

奔跑吧 Linux 内核入门篇重点是介绍 Linux 内核入门和编程实践，因此大部分实验和体系结构不相关。但是本书配套实验平台 runninglinuxkernel_4.0 支持 ARM32、ARM64 以及 x86_64 体系结构。

本书大部分实验以 ARM32 为蓝本，小部分实验基于 ARM64 或者 x86_64 体系结构。

希望读者在使用本书之前对处理器体系结构有一定的了解，不管是 ARM 还是 x86。关于体系结构的资料，读者可以阅读 ARM 公司或者 Intel 公司提供的文档。

ARM32 相关资料：

- <ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition>
- <ARM Cortex-A Series Programmer's Guide, version 4.0>

ARM64 相关资料：

- <ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile>
- <ARM Cortex-A Series Programmer's Guide for ARMv8-A, version 1.0>

x86_64 相关资料：

- <Intel® 64 and IA-32 Architectures Software Developer's Manual>

5. 系统默认配置

《奔跑吧 Linux 内核 * 入门篇》实验平台有如下默认的配置：

- 主机 Host 系统：装载了优麒麟 18.04.2 的主机或者 vmware 虚拟机
- 研究内核版本：Linux 4.0

奔跑吧 linux 社区出品

- 工作目录: /home/rbk/rbk_base
- VIM: 8.0
- arm-linux-gnueabi-gcc 版本: 5.5.0

```
rlk@ubuntu:runninglinuxkernel_4.0$ arm-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/arm-linux-gnueabi/5/lto-wrapper
Target: arm-linux-gnueabi
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-locale=gnu --enable-libstdcxx-debug
--enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libitm --disable-libquadmath --enable-plugin --with-system-zlib --enable-multiarch --enable-sjlabel-exceptions --with-arch=armv5t --with-float=soft --disable-werror --enable-multi-lib --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=arm-linux-gnueabi --program-prefix=arm-linux-gnueabi-
--includedir=/usr/arm-linux-gnueabi/include
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu Linaro 5.5.0-12ubuntu1)
rlk@ubuntu:runninglinuxkernel_4.0$
```

- aarch64-linux-gnu-gcc 版本: 5.5.0

```
rlk@ubuntu:runninglinuxkernel_4.0$ aarch64-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/5/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-locale=gnu --enable-libstdcxx-debug
--enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libitm --enable-plugin --enable-default-pie
--with-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --enable-werror --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu- --includedir=/usr/aarch64-linux-gnu/include
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu Linaro 5.5.0-12ubuntu1)
rlk@ubuntu:runninglinuxkernel_4.0$
```

- gcc 版本: 5.5.0
- aarch64-linux-gnu-gdb 版本: 8.3

```
rlk@ubuntu:runninglinuxkernel_4.0$ aarch64-linux-gnu-gdb -v
GNU gdb (GDB) 8.3
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.
rlk@ubuntu:runninglinuxkernel_4.0$
```

- QEMU 版本: 4.1.0

```
rlk@ubuntu:runninglinuxkernel_4.0$ qemu-system-arm --version
QEMU emulator version 4.1.0
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
rlk@ubuntu:runninglinuxkernel_4.0$
```

- 系统环境变量。增加“BASEINCLUDE”系统环境变量，用于内核模块编译指定内核的路径。
- 修改/home/rbk/.bashrc 文件，增加一行。

```
export BASEINCLUDE=/home/rbk/rbk_base/runninglinuxkernel_4.0
```

```
120
121 export BASEINCLUDE=/home/rbk/rbk_base/runninglinuxkernel_4.0
NORMAL <kernel_4.0 .bashrc
1:.bashrc
```

注意：若读者没有配置这个环境变量，那么在编译内核模块时候，需要手工指定 BASEINCLUDE 目录。例如

学习建议

```
# make BASEINCLUDE=/home/xxx/xxx //指定runninglinuxkernel_4.0的绝对路径
```

➤ eclipse-cdt 版本: Oxygen.3 Release (4.7.3)



6. 学习建议

1. 建议读者使用笔者创建的 vmware 镜像来做实验，这样可以免去搭建环境的烦恼。
2. 建议读者一定要自行完成实验，可以参考 Linux 内核代码以及内核中的例子，**但切记不要直接抄作业(不加思考直接抄参考代码)**。本书提供的参考代码仅供学习之用，在参考本书代码之前一定要自己独立思考。若真做不出，才可以参考本书的代码，但也需要思考为什么没有做出来，把问题思考清楚，这样才能有提高。

本实验指导手册有的实验提出了一些思考题，这些思考题都是来自实际项目中经验总结，希望读者可以认真思考，融合贯通。

第 1 章

Linux 系统入门

1.1 实验 1：在虚拟机中安装优麒麟 Linux 18.04 系统

1. 实验目的

通过本实验熟悉 Linux 系统的安装过程。本实验会先在虚拟机上安装优麒麟 18.04 版本的 Linux，掌握了安装方法之后，读者可以在真实的物理机器上安装 Linux 了。

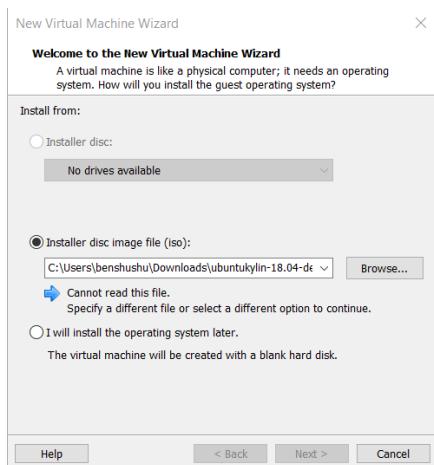
2. 实验步骤

- 首先需要到优麒麟官方网站上下载优麒麟 18.04 的安装介质。
- 到 VMware 官网下载 VMware Workstation Player 这个工具。这个工具对于个人用户是免费的，对于商业用户是收费的。

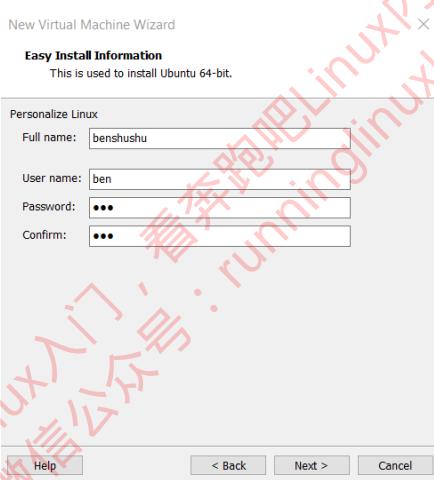


- 打开 VMware Player。在软件的主页面中选择“Create a New Virtual Machine”。
- 在 New Virtual Machine Wizard 界面中的“Installer disc image file (iso)”对话框中选择刚才下载的安装介质。然后点击下一步。

1.1 实验 1：在虚拟机中安装优麒麟 Linux 18.04 系统



5. 在接下来的窗口中输入即将要安装的 Linux 的用户名和密码。

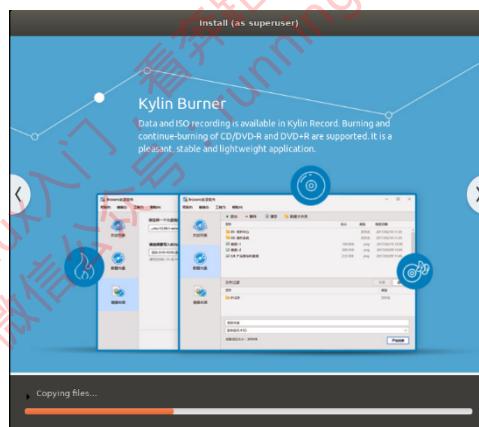


6. 接下来设置虚拟机的磁盘空间，这里尽可能设置大一点。虚拟机的磁盘空间是动态分配的，如这里设置了 200GB，并不会马上在 Host 主机上分配 200GB 的磁盘空间。

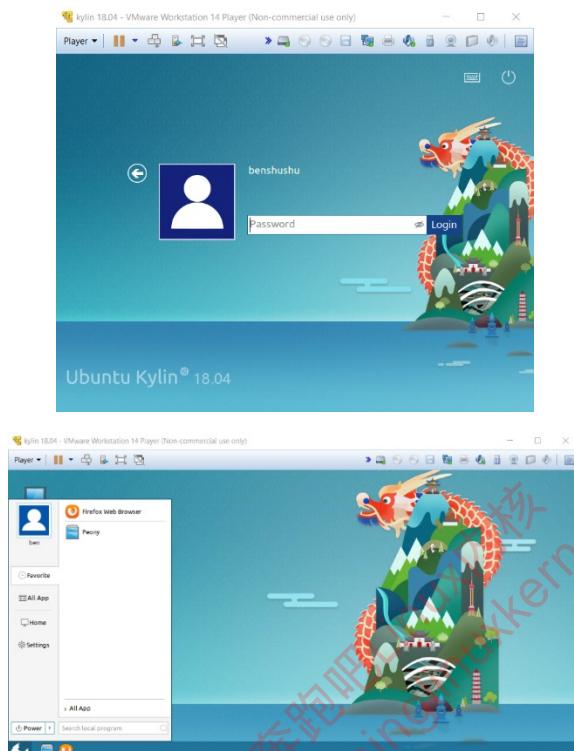
奔跑吧 linux 社区出品



7. 最后一步，可以在 Customize Hardware 选项里重新对一些硬件进行配置，如把内存设置大一点。完成 VMware Player 的设置之后，就马上进行虚拟机中。
8. 在虚拟机中会自动执行安装程序。安装完成之后就自动重启到新安装系统的登录界面中。



1.2 实验 2 : 给优麒麟 Linux 系统更换心脏



1.2 实验 2：给优麒麟 Linux 系统更换心脏

1. 实验目的

- 1) 通过本实验学会如何给 Linux 系统更换最新版本的 Linux 内核。
- 2) 如何编译和安装 Linux 内核。

2. 实验步骤

在编译 Linux 内核之前，我们需要安装如下软件包。

```
sudo apt-get install libncurses5-dev libssl-dev build-essential openssl bison  
bc flex
```

当然你可以使用如下命令来安装编译内核需要的所有依赖包。

```
sudo apt build-dep linux-image-generic
```

到 linux 内核的官方网站 (<https://www.kernel.org/>) 上下载最新的版本，比如现在最新的稳定的内核版本是 Linux 4.16.3。Linux 内核的版本号分成三部分，第一个数字表示主版本号，第二个数字表示次版本号，第三个数字表示修正版本号。



可以通过如下命令把下载的 xz 压缩包进行解压：

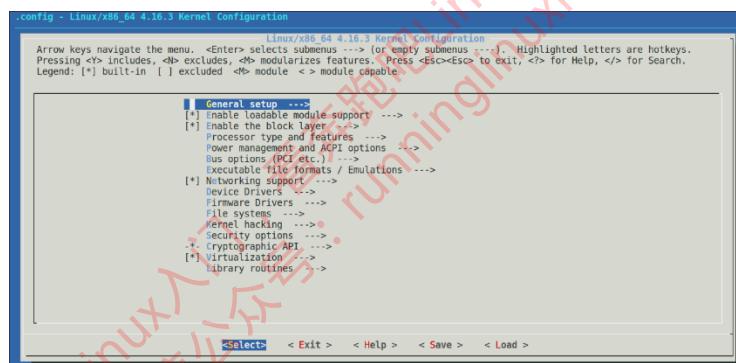
```
#xz -d linux-4.16.3.tar.xz
#tar -xf linux-4.16.3.tar
```

读者也可以使用如下命令来对 xz 的压缩包进行解压。

```
#tar -Jxf linux-4.16.3.tar.xz
```

解压缩完成之后，我们可以通过 make menuconfig 来进行内核配置。

```
#cd linux-4.16.3
#make menuconfig
```



若直接输入 make menuconfig 命令，则会默认使用 x86 目录默认的 config 文件，比如 x86_64_config 文件，它在 linux-4.16.3/arch/x86/config/x86_64_config。

```
rlik@ubuntu:~$ cd linux-4.16.3/
rlik@ubuntu:~/linux-4.16.3$ ls
arch certs CREDITS Documentation firmware include ipc Kconfig lib MAINTAINERS mm README scripts sound usr
block COPYING crypto drivers fs init Kbuild kernel LICENSES Makefile net samples security tools virt
rlik@ubuntu:~/linux-4.16.3$ ls arch/x86/configs/
i386_defconfig tiny.config x86_64_defconfig xen.config
rlik@ubuntu:~/linux-4.16.3$ ls arch/x86/configs/
```

Linux 内核的配置选项很多，大部分不需要读者去每一个单独配置，通常芯片厂商会给我们一个默认的配置文件，我们可以基于此配置来进行增减。

除了手工配置 Linux 内核的选项之外，还可以直接拷贝使用优麒麟 Linux 系统中自带的配置文件。

```
#cd linux-4.16.3
#cp /boot/config-4.15.0-29-generic .config
#make menuconfig
```

在 menuconfig 图形界面中，我们不打算继续配置内核，选择“Exit”保存退出。

1.2 实验 2：给优麒麟 Linux 系统更换心脏

<Select> < Exit > < Help > < Save > < Load >

开始编译内核，其中-jn 中的 n 表示使用多少个 CPU 核心来并行编译内核。

```
#make -jn
```

如何查看系统中有多少个 CPU 核心，可以通过如下命令来看。

```
#cat /proc/cpuinfo
```

...

```
processor : 3
vendor_id : GenuineIntel
cpu family    : 6
model       : 60
model name   : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping    : 3
```

processor 这一项等于 3，说明系统有 4 个 CPU 核心，因为是从 0 开始计数的。那么刚才那个 make -jn 的命令就可以变成 make -j4 了。

编译内核是一个漫长的过程，需要几十分钟时间，取决于电脑的运算能力和内核配置的选项。

make 编译完成之后，下一步需要编译和安装内核的模块。

```
#sudo make modules_install
```

最后一步就是把编译好的内核 image 安装到优麒麟 Linux 系统中。

```
#sudo make install
```

完成之后就可以重启电脑了，登录到最新的系统中了。

3 Linux 内核配置菜单补充说明

下面对 Linux 内核配置一些主要的配置选项做一些说明。make menuconfig 菜单下面有 Select, Exit, Help, Save, Load 等五个菜单。读者可以使用键盘的光标键可以左右选择。

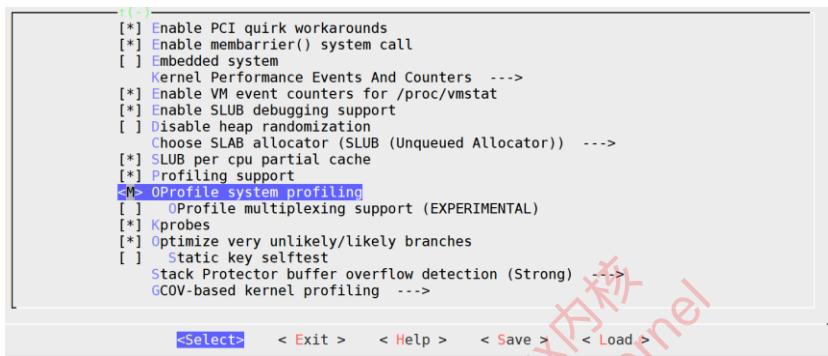
<Select> < Exit > < Help > < Save > < Load >

- Select: 表示选择光标高亮显示的子菜单。
- Exit: 退出当前子菜单
- Help: 进入帮助模式
- Save: 保存当前设置到 Linux 内核源代码根目录下的.config 文件中
- Load: 表示加载一个配置文件。

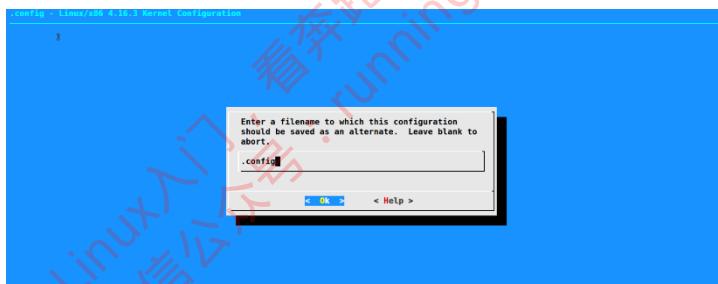
Linux 内核的模块有两种编译方法，一个是静态编译，另外一个是动态编译成模块。

- 在菜单中选择 “Y”，这时在菜单前面的中括号[]里显示 “*”，表示该模块会静态编译进 vmlinux。

- 在菜单中选择“N”，这时在菜单前面的中括号[]里显示空白，表示该模块不会编译进 vmlinux。
- 在菜单中选择“M”，这时在菜单前面的中括号[]里显示“M”，表示该模块编译成模块。



当选择和配置完内核模块，可以选择界面最下面的“<Save>”菜单，保存到 Linux 内核源代码根目录的.config 文件中。



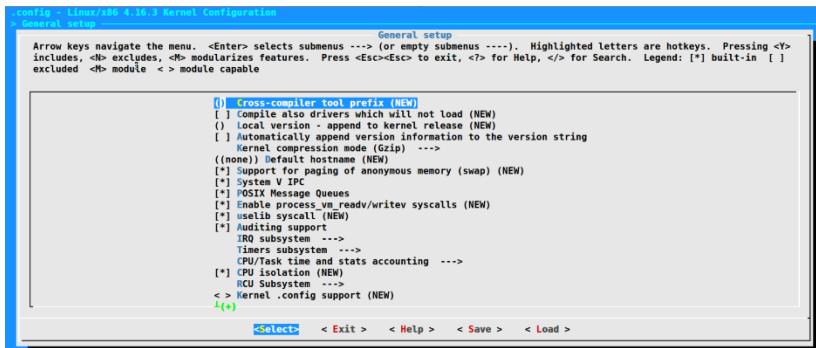
最后，选择“<Exit>”菜单退出内核配置。

下面对 Linux 内置内核主要菜单做一个简单介绍。读者不必对这些菜单死记硬背，实际工作中需要用到在认真阅读帮助文件。

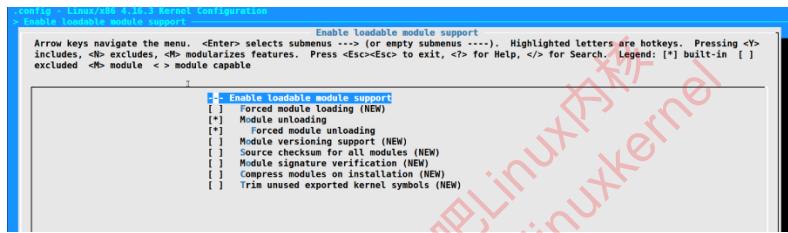
(1) General setup

General setup 选项为常规安装选项，包括版本信息、虚拟内存、进程间通信、系统调用、审计支持等基本内核配置选项。

1.2 实验 2：给优麒麟 Linux 系统更换心脏

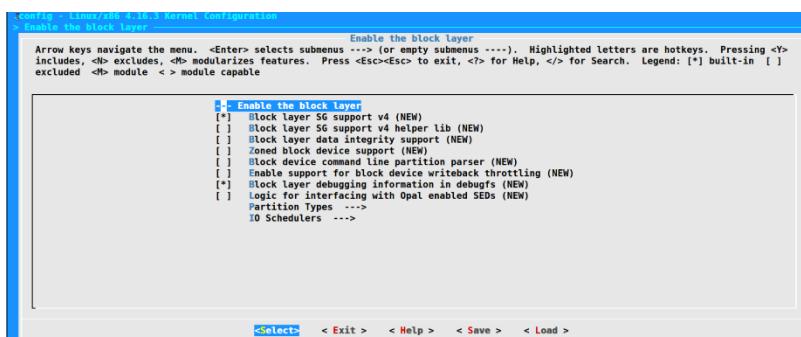


(2) Enable loadable module support



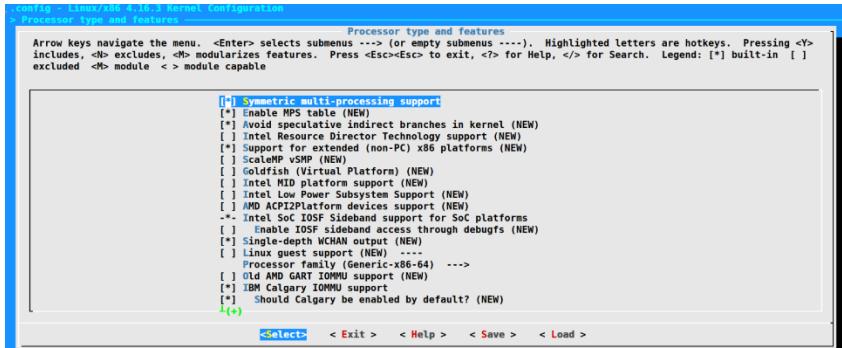
打开可加载模块支持，如果打开它则必须通过 `make modules_install` 把内核模块安装在`/lib/modules/`中。模块是一小段代码，编译后可在系统内核运行时动态地加入内核，从而为内核增加一些特性或是对某种硬件进行支持。一般一些不常用到的驱动或特性可以编译为模块以减少内核的体积。在运行时可以使用 `modprobe` 命令来加载它到内核中去（在不需要时还可以移除它）。一些特性是否编译为模块的原则有不常使用的，或是在系统启动时不需要的驱动可以将其编译为模块，如果是一些在系统启动时就要用到的驱动，比如说文件系统，系统总线的支持就不要编为模块，否则无法启动系统。在启动时不用到的功能编成模块是最有效的方式。可以查看 MAN 手册 来了解 `modprobe`、`lsmod`、`modinfo`、`insmod` 和 `rmmod`。

(3) Enable the block layer



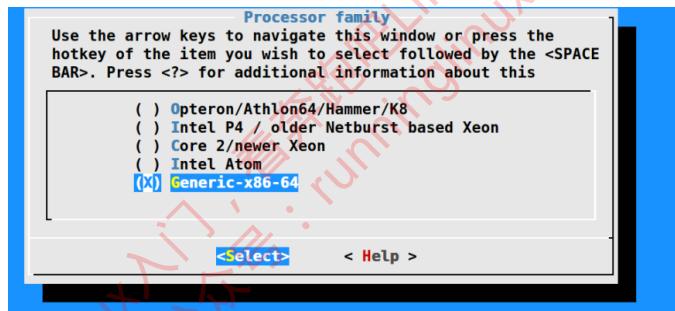
打开块设备的支持。

(4) Processor type and features

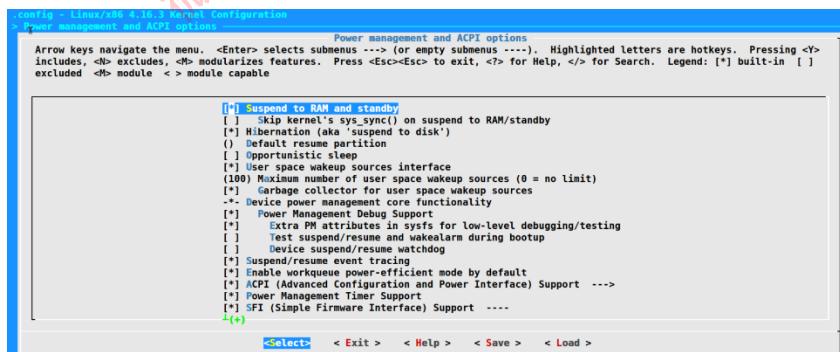


Processor type and features 即处理器类型及特性，该模块包括处理器系列、内核抢占模式、抢占式大内核锁、内存模式、使用寄存器参数等处理器配置相关信息。

选择支持的处理器型号以及处理器相关的特性。在本实验中，我们选择“Generic-x86-64”。



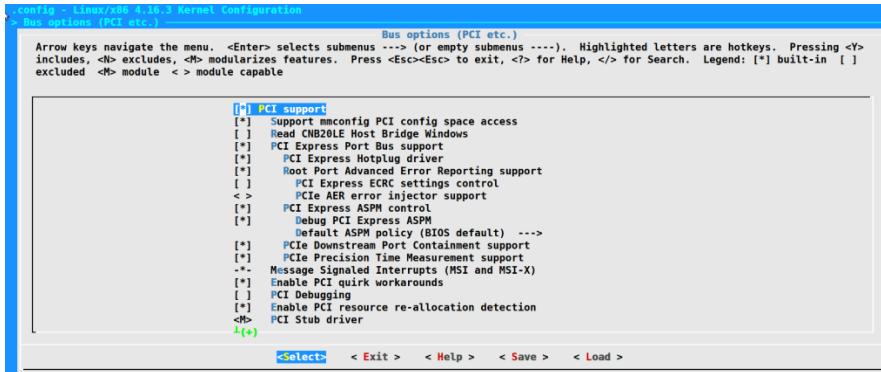
(5) Power management and ACPI options



选择和电源相关的属性，比如是否支持睡眠时候把内容存放在 RAM 中 (Suspend to RAM and standby)，是否支持深度睡眠，即睡眠时候把内存存放在磁盘中 (Hibernation (aka 'suspend to disk'))。

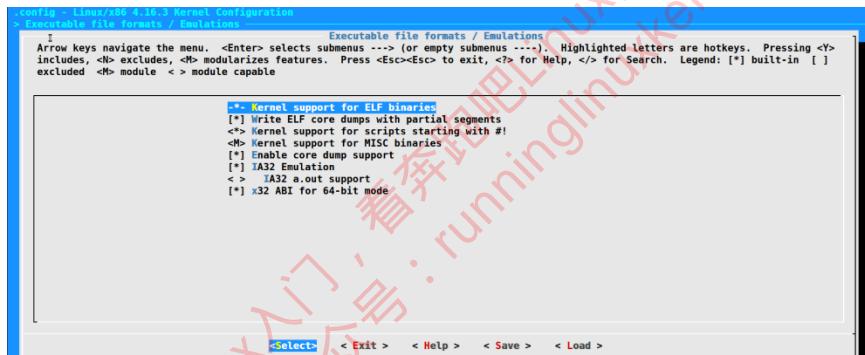
1.2 实验 2 : 给优麒麟 Linux 系统更换心脏

(6) Bus options (PCI etc.)



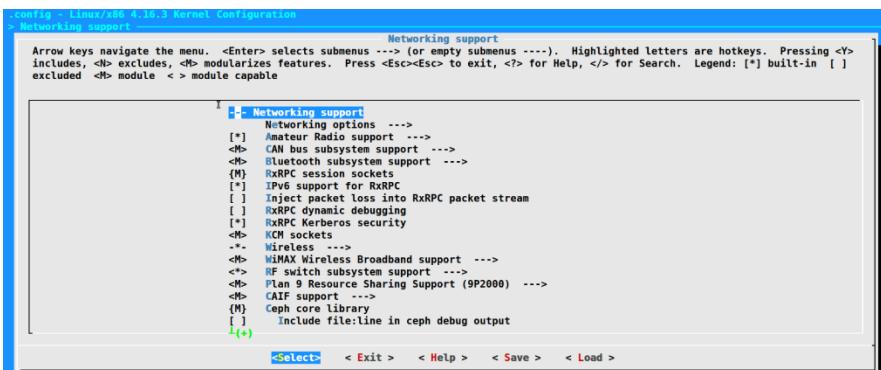
选择对 PCIe 等总线的支持。

(7) Executable file formats / Emulations



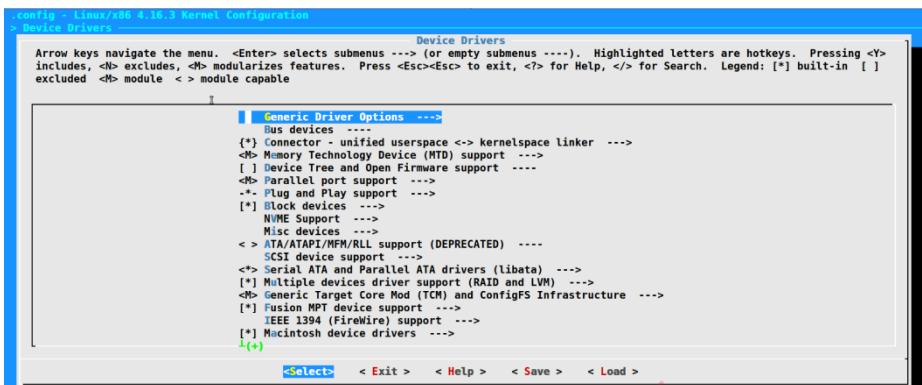
对可执行文件 ELF 相关支持选项。

(8) Networking support



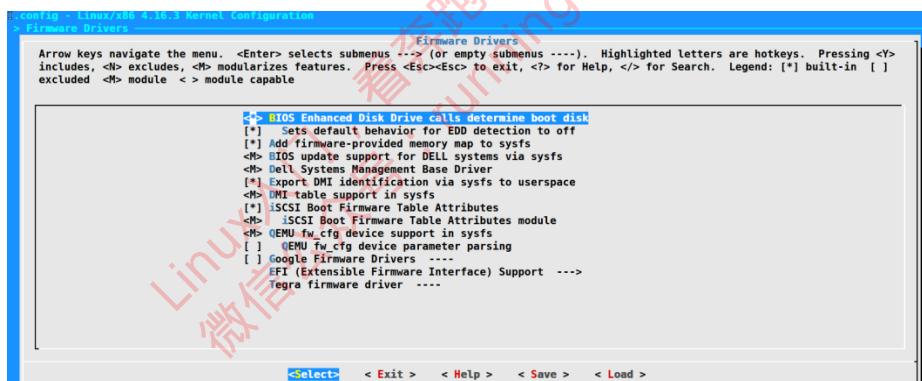
对网络协议栈比如 TCP/IP, WIFI 无线协议的支持等。

(9) Device Drivers



设备驱动支持。这里支持的驱动很多，包括 PCI 驱动，网卡驱动，声卡驱动，显卡驱动，I2C 驱动，SPI 驱动等等。

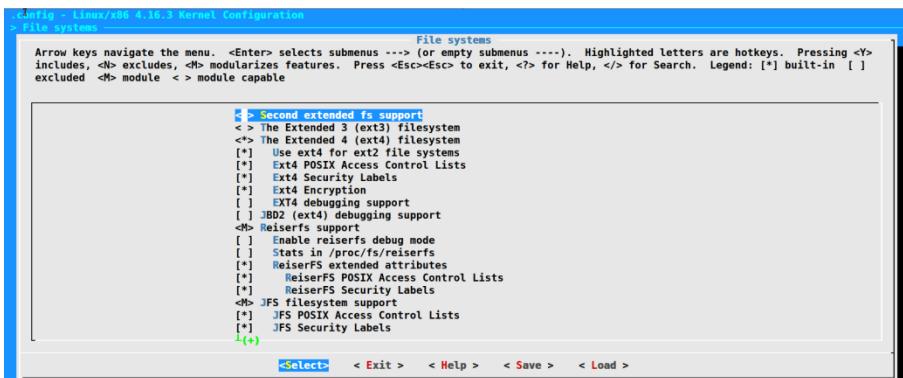
(10) Firmware Drivers



内核对固件的支持。

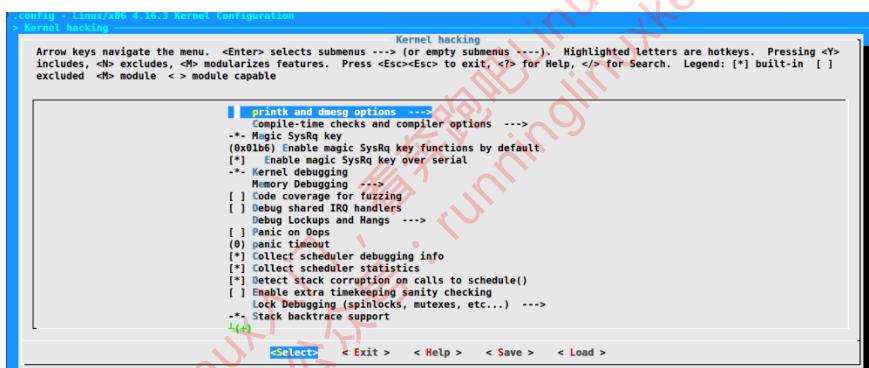
(11) File systems

1.2 实验 2 : 给优麒麟 Linux 系统更换心脏



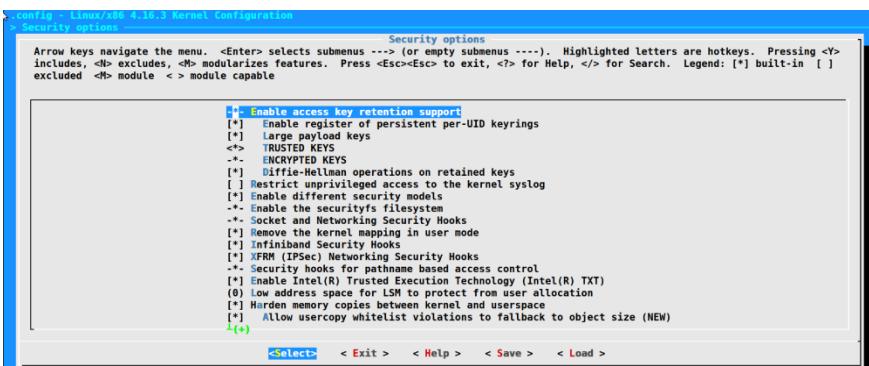
内核对文件系统的支持，包括 ext4, xfs 等常见的文件系统。还包括 Linux 内核支持的 sysfs, proc 等虚拟文件系统。

(12) Kernel hacking



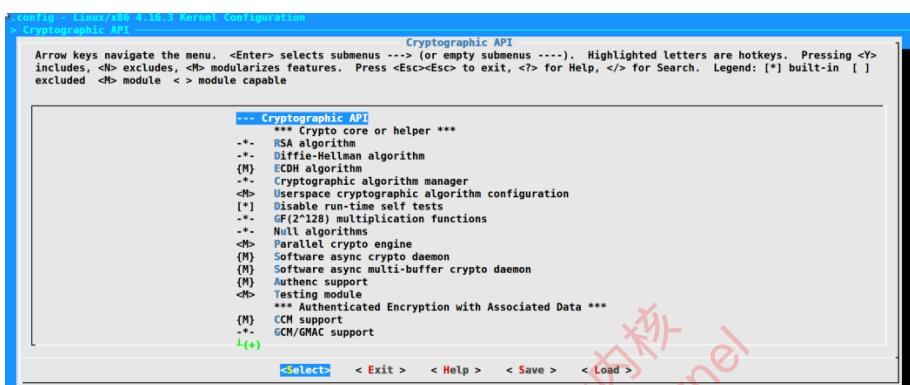
内核对调试相关的支持选项，比如设置 printk 的等级，是否编译带符号表的 image，打开 slab 调试功能等。

(13) Security options



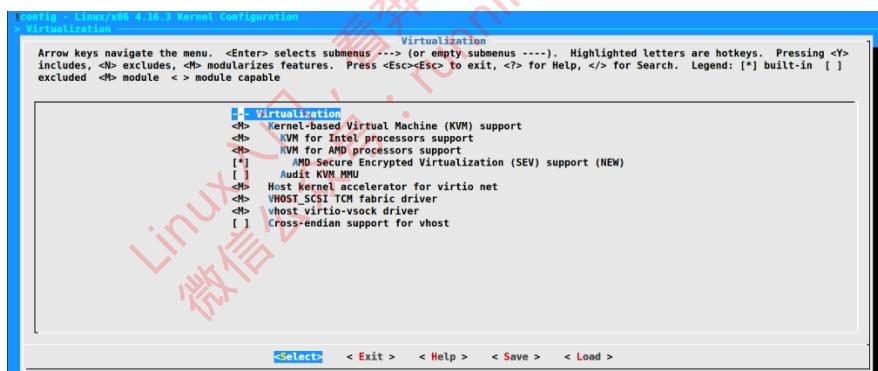
和系统安全相关的配置选项。

(14) Cryptographic API



内核加解密相关的配置选项。

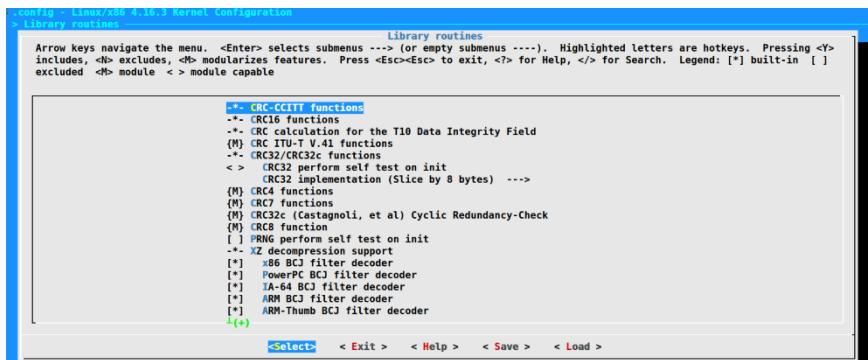
(15) Virtualization



虚拟化相关的配置选项，比如 KVM 的支持等。

(16) Library routines

1.3 实验 3：使用 O0 编译的内核 - runninglinuxkernel



内核支持的相关库，比如 CRC 算法，xz 解压缩算法。

1.3 实验 3：使用 O0 编译的内核 - runninglinuxkernel

1. 实验目的

通过本实验学习如何编译一个 ARM 版本的内核 image，并且在 QEMU 上运行起来。

2. 实验步骤

为了方便读者快速完成实验，我们制作定制了一个 Linux 内核的 git 仓库，读者可以很方便克隆这个 git 仓库下来完成本实验。这个 git 仓库包含如下内容。

- 定制了可以运行在 ARM、ARMA64 和 x86 架构的小文件系统
- 支持 GCC 的“O0”编译选项来编译内核
- Host 主机和 QEMU 虚拟机共享文件

1) 安装工具

首先在优麒麟 Linux 18.04 中安装如下工具。

```
$ sudo apt-get install qemu libncurses5-dev gcc-arm-linux-gnueabi build-essential gcc-5-arm-linux-gnueabi git
```

注意：使用本书提供的 VMware v4.0 镜像的小伙伴，不用重复安装上述软件包。

特别注意，vmware v4.0 镜像里的 qemu 版本已经升级到了 4.1.0 版本^①，而优麒麟 Linux 18.04 里内置的 qemu 版本是 2.11。

2) GCC 版本切换

在优麒麟 Linux 18.04 系统中默认安装的 ARM GCC 编译器的版本是 7.3 的。在

^① 若小伙伴们在自己配置的实验环境中安装 QEMU 4.1.0 版本，需要单独下载 QEMU 源码包进行编译和安装。

编译 Linux 4.0 内核时会编译出错，因此，我们需要切换成 5.5 版本的 ARM GCC 编译器。

```
CC      scripts/mod/empty.o
CC      scripts/mod/devicetable-offsets.s
In file included from include/linux/compiler.h:54:0,
                 from include/uapi/linux/stddef.h:1,
                 from include/linux/stddef.h:4,
                 from ./include/uapi/linux/posix_types.h:4,
                 from include/uapi/linux/types.h:13,
                 from include/linux/types.h:5,
                 from include/linux/mod_devicetable.h:11,
                 from scripts/mod/devicetable-offsets.c:2:
include/linux/compiler-gcc.h:107:1: fatal error: linux/compiler-gcc7.h: No such file or directory
#include gcc_header(__GNUC__)
~~~
compilation terminated.
scripts/Makefile.build:153: recipe for target 'scripts/mod/devicetable-offsets.s' failed
make[2]: *** [scripts/mod/devicetable-offsets.s] Error 1
make[2]: *** Waiting for unfinished jobs...
scripts/Makefile.build:403: recipe for target 'scripts/mod' failed
make[1]: *** [scripts/mod] Error 2
make[1]: *** Waiting for unfinished jobs...
CC      kernel/bounds.s
In file included from include/linux/compiler.h:54:0,
                 from include/uapi/linux/stddef.h:1,
                 from include/linux/stddef.h:4,
                 from ./include/uapi/linux/posix_types.h:4,
                 from include/uapi/linux/types.h:13,
                 from include/linux/types.h:5,
                 from include/linux/page-flags.h:8,
                 from kernel/bounds.c:9:
include/linux/compiler-gcc.h:107:1: fatal error: linux/compiler-gcc7.h: No such file or directory
#include gcc_header(__GNUC__)
~~~
```

如上图所示，出现这个问题，一定是 arm-linux-gnueabi-gcc 的版本过高了。我们可以通过如下命令查看 arm-linux-gnueabi-gcc 的版本。

```
rlk@figo-OptiPlex-9020:runninglinuxkernel_4.0$ arm-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/arm-linux-gnueabi/7/lto-wrapper
Target: arm-linux-gnueabi
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 7.4.0-1'
Thread model: posix
gcc version 7.4.0 (Ubuntu/Linaro 7.4.0-1ubuntu1~18.04.1)
rlk@figo-OptiPlex-9020:runninglinuxkernel_4.0$
```

接下来，使用 update-alternatives 命令来对 GCC 工具的多个版本进行管理，通过它可以很方便的设置系统默认使用的 GCC 工具的版本。下面使用这个命令来设置 arm-linux-gnueabi-gcc 的版本。刚才我们安装了两个版本的 ARM GCC 工具链，分别是 5.5 和 7.3 版本。

update-alternatives 命令的主要参数如下。

```
update-alternatives --install <link> <name> <path> <priority>
```

- **link:** 指向/etc/alternatives/<name>的符号引用
- **name:** 链接的名称
- **path:** 这个命令对应的可执行文件的实际路径
- **priority:** 优先级，在 auto 模式下，数字大的优先级比较高。

1.3 实验 3：使用 O0 编译的内核 - runninglinuxkernel

接下来往系统添加一个 arm-linux-gnueabi-gcc-5 的链接配置并设置优先级。

```
$ sudo update-alternatives --install /usr/bin/arm-linux-gnueabi-gcc arm-
linux-gnueabi-gcc /usr/bin/arm-linux-gnueabi-gcc-5 5
```

```
update-alternatives: using /usr/bin/arm-linux-gnueabi-gcc-5 to provide
/usr/bin/arm-linux-gnueabi-gcc (arm-linux-gnueabi-gcc) in auto mode
```

接下来往系统添加一个 arm-linux-gnueabi-gcc-7 的链接配置并设置优先级。

```
$ sudo update-alternatives --install /usr/bin/arm-linux-gnueabi-gcc arm-
linux-gnueabi-gcc /usr/bin/arm-linux-gnueabi-gcc-7 7
```

```
update-alternatives: using /usr/bin/arm-linux-gnueabi-gcc-7 to provide
/usr/bin/arm-linux-gnueabi-gcc (arm-linux-gnueabi-gcc) in auto mode
```

使用 update-alternatives 命令来选择一个配置。

```
$ sudo update-alternatives --config arm-linux-gnueabi-gcc
There are 2 choices for the alternative arm-linux-gnueabi-gcc (providing
/usr/bin/arm-linux-gnueabi-gcc).
```

Selection	Path	Priority	Status
*	/usr/bin/arm-linux-gnueabi-gcc-7	7	auto mode
1	/usr/bin/arm-linux-gnueabi-gcc-5	5	manual mode
2	/usr/bin/arm-linux-gnueabi-gcc-7	7	manual mode

```
Press <enter> to keep the current choice[*], or type selection number:
```

会看到有三个候选项，选择“1”就会设置系统默认使用 arm-linux-gnueabi-gcc-5。

查看系统的 arm-linux-gnueabi-gcc 的版本是否为 5.5 版本。

```
$ arm-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/arm-linux-gnueabi/5/lto-wrapper
Target: arm-linux-gnueabi
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,objc++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libitm --disable-libquadmath --enable-plugin --with-system-zlib --enable-multiarch --enable-multilib --disable-sjlj-exceptions --with-arch=armv5t --with-float=soft --disable-werror --enable-multilib --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=arm-linux-gnueabi --program-prefix=arm-linux-gnueabi- --
includedir=/usr/arm-linux-gnueabi/include
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu/Linaro 5.5.0-12ubuntu1)
```

3) 下载 runninglinuxkernel 的 git 仓库并切换到 rlk_basic 分支

```
$ git clone https://github.com/figozhang/runninglinuxkernel\_4.0.git
$ git checkout rlk_basic
```

注意：本书配套的 VMware 镜像里已经下载了 runninglinuxkernel，目录是在 /home/rbk/rbk_basic/runninglinuxkernel_4.0。

读者可以通过如下命令来更新在最新版本。

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ git pull //该命令需要连接互联网
```

4) 编译内核

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make menuconfig
```

在内核图形化配置菜单中，本实验不需要做额外配置，我们选择退出“Exit”即可。

在_install_arm32/dev 目录下创建如下设备节点，需要 root 权限。

```
$ cd _install_arm32
$ mkdir dev
$ cd dev
$ sudo mknod console c 5 1
```

开始编译 kernel。

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ make bzImage -j4
$ make dtbs
```

编译时间可能需要十几分钟，依赖主机处理器能力。

5) 运行 QEMU 虚拟机

运行 run.sh 脚本。runninglinuxkernel 在这个版本支持 Linux 主机和 Qemu 虚拟机的文件共享。

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ ./run.sh arm32
```

运行结果如下：

```
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version 4.0.0 (figo@figo-OptiPlex-9020) (gcc version 4.6.3
(Ubuntu/Linaro 4.6.3-1ubuntu5) ) #9 SMP Wed Jun 22 04:23:19 CST 2016
CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
```

1.3 实验 3：使用 O0 编译的内核 - runninglinuxkernel

```

Machine model: V2P-CA9
Memory policy: Data cache writealloc
On node 0 totalpages: 262144
free_area_init_node: node 0, pgdat c074c600, node_mem_map eefffa000
    Normal zone: 1520 pages used for memmap
    Normal zone: 0 pages reserved
    Normal zone: 194560 pages, LIFO batch:31
    HighMem zone: 67584 pages, LIFO batch:15
PERCPU: Embedded 10 pages/cpu @eefc1000 s11712 r8192 d21056 u40960
pcpu-alloc: s11712 r8192 d21056 u40960 alloc=10*4096
pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 260624
Kernel command line: rdinit=/linuxrc console=ttyAMA0 loglevel=8
log_buf_len individual max cpu contribution: 4096 bytes
log_buf_len total cpu_extra contributions: 12288 bytes
log_buf_len min size: 16384 bytes
log_buf_len: 32768 bytes
early log buf free: 14908(90%)
PID hash table entries: 4096 (order: 2, 16384 bytes)
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
Memory: 1031644K/1048576K available (4745K kernel code, 157K rwdata, 1364K
rodata, 1176K init, 166K bss, 16932K reserved, 0K cma-reserved, 270336K highmem)
Virtual kernel memory layout:
    vector : 0xfffff0000 - 0xfffff1000   ( 4 KB)
    fixmap : 0xfffc00000 - 0xfffc00000   (3072 KB)
    vmalloc : 0xf0000000 - 0xff000000   ( 240 MB)
    lowmem : 0xc0000000 - 0xef800000   ( 760 MB)
    pkmap : 0xbfe00000 - 0xc0000000   ( 2 MB)
    modules : 0xbf000000 - 0xbfe00000   ( 14 MB)
        .text : 0xc0008000 - 0xc05ff80c   (6111 KB)
        .init : 0xc0600000 - 0xc0726000   (1176 KB)
        .data : 0xc0726000 - 0xc074d540   ( 158 KB)
        .bss : 0xc074d540 - 0xc0776f38   ( 167 KB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
Hierarchical RCU implementation.
    Additional per-CPU info printed with stalls.
    RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=4
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
NR_IRQS:16 nr_irqs:16 16
smp_twd: clock not found -2
sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every 178956969942ns
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x604804f8 - 0x60480550
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
CPU2: thread -1, cpu 2, socket 0, mpidr 80000002
CPU3: thread -1, cpu 3, socket 0, mpidr 80000003
Brought up 4 CPUs
SMP: Total of 4 processors activated (1648.43 BogoMIPS).
Advanced Linux Sound Architecture Driver Initialized.
Switched to clocksource arm,sp804
Freeing unused kernel memory: 1176K (c0600000 - c0726000)

Please press Enter to activate this console.
/ # ls
bin      dev      etc      linuxrc  proc      sbin      sys      tmp      usr
/ #

```

在优麒麟 Linux 的另外一个超级终端中输入 killall qemu-system-arm，即可关闭 QEMU 平台，或者在 QEMU 虚拟机中输入“Ctrl+a+x”组合键来关闭 QEMU。

6) 测试 Host 主机和 QEMU 虚拟机共享文件

这个实验支持 Host 主机和 QEMU 虚拟机的共享文件，可以通过如下简单方法来测试。

拷贝一个文件到 runninglinuxkernel_4.0/kmodules 目录下面。下面以 test.c 文件为例。

```
$ cp test.c /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules
```

启动到 Qemu 虚拟机之后，首先检查一下/mnt 目录是否有 test.c 文件。

```
/ # cd /mnt/  
/mnt # ls  
README test.c  
/mnt #
```

我们在后续的实验中会经常利用这个特性，比如把编译好的内核模块放入到 QEMU 虚拟机中加载。

1.4 实验 4：如何编译和运行一个 ARM Linux 内核

1 实验目的

通过本实验学习如何编译一个 ARM 版本的内核 image，并且在 QEMU 上运行起来。

2 实验步骤

为了加速开发过程，ARM 公司提供了 Versatile Express 开发平台，客户可以基于 Versatile Express 平台进行产品原型开发。作为个人学习者，没有必要去购买 Versatile Express 开发平台或其他 ARM 开发板，完全可以通过 QEMU 来模拟开发平台，同样可以达到学习的效果。

1) 准备工具

下载如下代码包。

- linux-4.0 内核: <https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.0.tar.gz>。
- busybox 工具包: <https://busybox.net/downloads/busybox-1.28.0.tar.bz2>。

下载上述两个软件包，并解压。

2) 编译最小文件系统

首先利用 busybox 手工编译一个最小文件系统。

```
$ cd busybox  
$ export ARCH=arm
```

1.4 实验 4：如何编译和运行一个 ARM Linux 内核

```
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make menuconfig
```

进入 menuconfig 之后，配置成静态编译。

```
Busybox Settings --->
  Build Options --->
    [*] Build BusyBox as a static binary (no shared libs)
```

然后 make install 可以编译完成。编译完成后，在 busybox 根目录下会有一个 “_install” 的目录，该目录是编译好的文件系统需要的一些命令集合。

把 _install 目录复制到 linux-4.0 目录下。

进入 _install 目录，先创建 etc、dev 等目录。

```
#mkdir etc
#mkdir dev
#mkdir mnt
#mkdir -p etc/init.d/
```

在 _install /etc/init.d/ 目录下新创建一个 rcS 文件，并写入如下内容。

```
mkdir -p /proc
mkdir -p /tmp
mkdir -p /sys
mkdir -p /mnt
/bin/mount -a
mkdir -p /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

需要修改 _install/etc/init.d/rcS 文件需要可执行权限，可使用 chmod 命令来修改，比如 “chmod +x _install/etc/init.d/rcS”。

在 _install /etc 目录新创建一个 fstab 文件，并写入如下内容。

```
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
debugfs /sys/kernel/debug debugfs defaults 0 0
```

在 _install /etc 目录新创建一个 inittab 文件，并写入如下内容。

```
::sysinit:/etc/init.d/rcS
::respawn:-/bin/sh
::askfirst:-/bin/sh
::ctrlaltdel:/bin/umount -a -r
```

在 _install/dev 目录下创建如下设备节点，需要 root 权限。

```
$ cd _install/dev/
$ sudo mknod console c 5 1
$ sudo mknod null c 1 3
```

3) 编译内核

```
$ cd linux-4.0
$ export ARCH=arm
```

```
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make menuconfig
```

配置 initramfs，在 initramfs source file 中填入 _install，并把 Default kernel command string 清空。

```
General setup --->
[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
(_install) Initramfs source file(s)

Boot options -->
() Default kernel command string
```

配置 memory split 为“3G/1G user/kernel split”，并打开高端内存。

```
Kernel Features --->
Memory split (3G/1G user/kernel split) --->
[*] High Memory Support
```

开始编译 kernel。

```
$ cd linux-4.0
$ make bzImage -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
$ make dtbs
```

4) 运行 QEMU 虚拟机

运行 QEMU 来模拟 4 核 Cortex-A9 的 Versatile Express 开发平台。

```
$ qemu-system-arm -M vexpress-a9 -smp 4 -m 200M -kernel arch/arm/boot/zImage
-append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb
arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic
```

运行结果如实验 3 所示。

1.5 实验 5：运行 Debian+ARM32 系统（新增）

1 实验目的

通过本实验学习如何编译和运行一个 ARM32 版本的 Debian 系统，并且在 QEMU 上运行起来。

2 实验步骤

在 runninglinuxkernel_4.0 目录下面有一个 rootfs_debian_arm32.tar.xz 文件，这个是基于 ARM32 版本的 Debian 系统的根文件系统。但是这个根文件系统还只是一个半成品，我们还需要根据编译好的内核来安装内核镜像和内核模块。整个过程比较复杂：

- 编译内核
- 编译内核模块
- 安装内核模块

1.5 实验 5：运行 Debian+ARM32 系统（新增）

- 安装内核头文件
- 安装编译内核模块必须依赖文件
- 制作 ext4 根文件系统

这个过程比较繁琐，作者制作了一个脚本来简化上述过程。

注意，该脚本会使用 dd 命令来生成一个 2GB 大小的镜像文件，因此主机系统需要保证至少需要 4 个 GB 的空余磁盘空间。若读者需要生成一个更大的根文件系统镜像，可以修改 run_debian_arm32.sh 这个脚本文件。

(1) 编译内核

```
| $ cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
$ ./run_debian_arm32.sh build_kernel
```

执行上述脚本需要几十分钟，依赖于主机的计算能力。

```
rlk@ubuntu:runninglinuxkernel_4.0$ ./run_debian_arm32.sh build_kernel
start build kernel image...
arch/arm/Kconfig:1399:warning: 'HZ_FIXED': number is invalid
arch/arm/Kconfig:1400:warning: 'HZ_FIXED': number is invalid
#
# configuration written to .config
#
scripts/kconfig/conf --silentoldconfig Kconfig
arch/arm/Kconfig:1399:warning: 'HZ_FIXED': number is invalid
arch/arm/Kconfig:1400:warning: 'HZ_FIXED': number is invalid
CHK    include/config/kernel.release
CHK    include/generated/uapi/linux/version.h
CHK    include/generated/utsrelease.h
make[1]: 'include/generated/mach-types.h' is up to date.
CALL   scripts/checksyscalls.sh
```

(2) 编译 ARM32 版本的 Debian 系统 rootfs

```
| $ sudo ./run_debian_arm32.sh build_rootfs
```

注意：这里需要使用 root 权限。

```
rlk@ubuntu:runninglinuxkernel_4.0$ sudo ./run_debian_arm32.sh build_rootfs
[sudo] password for rlk:
```

编译完成后会生成一个 rootfs_debian_arm32.ext4 的文件系统。

(3) 运行 debian 系统。

```
| $ ./run_debian_arm32.sh run
```

注意：运行此命令不需要 root 权限。

奔跑吧 linux 社区出品

```
r1k@ubuntu:runninglinuxkernel_4.0$ ./run_debian_arm32.sh run
WARNING: Image format was not specified for 'rootfs_debian_arm32.ext4' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Initializing cgroup subsys cpuset
[ 0.000000] Linux version 4.0.0+ (r1k@ubuntu) (gcc version 5.5.0 20171010 (Ubuntu/linaro 5.5.0-12ubuntul) ) #1 SMP Su
[ 0.000000] CPU: ARMv7 Processor [412fc0f1] revision 1 (ARMv7), cr=1c5387d
[ 0.000000] CPU: PPI / VPI / nonaliasing data cache, PPI/P instruction cache
[ 0.000000] Machine model: linux,dummy-virt
[ 0.000000] Memory policy: Data cache writealloc
[ 0.000000] psci: probing for conduit method from DT.
[ 0.000000] psci: PSCIv0.2 detected in firmware.
[ 0.000000] psci: Using standard PSCI v0.2 function IDs
[ 0.000000] Reserving 128MB of memory at 0MB for crashkernel (System RAM: 760MB)
[ 0.000000] PERCPU: Embedded 11 pages/cpu @eefcf000-16268 r8192 d20596 u45056
[ 0.000000] Built 1 zonelists in zone order, mobility grouping on. Total pages: 260624
[ 0.000000] Kernel command line: crashkernel=128M root=/dev/vda rootfstype=ext4 rw
[ 0.000000] log_buf_len individual max cpu contribution: 4096 bytes
[ 0.000000] log_buf_len total cpu_extra contributions: 12288 bytes
[ 0.000000] log_buf_len min size: 16384 bytes
[ 0.000000] log_buf_len: 32768 bytes
[ 0.000000] early log buf free: 14680(89%)
[ OK ] Started System Logging Service.
[ OK ] Started Getty on tty4.
[ OK ] Started Getty on tty2.
[ OK ] Started Getty on tty1.
[ OK ] Started Serial Getty on ttyAMA0.
[ OK ] Started Getty on tty3.
[FAILED] Failed to start Remove Sta...ext4 Metadata Check Snapshots.
See 'systemctl status e2scrub_reap.service' for details.
[ OK ] Started Getty on tty6.
[ OK ] Started Getty on tty5.
[ OK ] Started getty on tty2-tty6... and logind are not available.
[ OK ] Reached target Login Prompts.
[ OK ] Reached target Multi-User System.
[ OK ] Reached target Graphical Interface.
      Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Update UTMP about System Runlevel Changes.

debian GNU/Linux 10 runninglinuxkernel ttyAMA0

runninglinuxkernel login:
runninglinuxkernel login: root
Password:
Linux runninglinuxkernel 4.0.0+ #1 SMP Sun Aug 18 12:29:01 PDT 2019 armv7l
Welcome Running Linux Kernel.
Created by Ben Shushu <runninglinuxkernel@126.com>

Buy linux kernel training course:
https://shop115683645.taobao.com/

Wechat: runninglinuxkernel
root@runninglinuxkernel:~#
```

Debian 系统的用户名为: root, 密码为: 123

(4) 在线安装软件包

QEMU 虚拟机可以通过 VirtIO-NET 技术来生成一个虚拟的网卡，并且通过 NAT 网络桥接技术和主机进行网络共享。首先使用 ifconfig 命令来检查网络配置。

1.5 实验 5：运行 Debian+ARM32 系统（新增）

```
root@runninglinuxkernel:~# 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 19 bytes 2560 (2.5 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 29 bytes 2660 (2.5 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
      loop txqueuelen 0 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@runninglinuxkernel:~#
```

可以看到生成了一个名为 eth0 的网卡设备，分配的 IP 地址为：10.0.2.15。

通过 apt update 命令来更新 Debian 系统的软件仓库。

```
$ apt update
```

```
root@runninglinuxkernel:~# apt update
Get:1 http://mirrors.ustc.edu.cn/debian unstable InRelease [149 kB]
Get:2 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.diff/Index [27.9 kB]
Ign:3 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.Index
Get:3 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.diff/Index [27.9 kB]
Ign:4 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.Index
Get:4 http://mirrors.ustc.edu.cn/debian unstable/non-free armel Packages.diff/Index [27.8 kB]
Get:5 http://mirrors.ustc.edu.cn/debian unstable/non-free Translation-en.diff/Index [27.8 kB]
Get:6 http://mirrors.ustc.edu.cn/debian unstable/contrib armel Packages.diff/Index [27.8 kB]
Get:7 http://mirrors.ustc.edu.cn/debian unstable/contrib Translation-en.diff/Index [27.8 kB]
```

如果更新失败，有可能是系统时间比较旧了，可以使用 date 命令来设置日期。

```
root@benshushu:~# date -s 2019-04-25 #假设最新日期是2019年4月25日
Thu Apr 25 00:00:00 UTC 2019
```

使用 apt install 命令来安装软件包。比如，可以在线安装 gcc。

```
root@runninglinuxkernel:~# apt install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-arm-linux-gnueabi binutils-common cpp cpp-9 gcc-9
  gcc-9-base libasan5 libatomic1 libbinutils libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgcc1 libgomp1 libisl19 libmpc3 libmpfr6 libubsan1
  linux-libc-dev manpages manpages-dev
Suggested packages:
  binutils-doc cpp-doc gcc-9-locales gcc-multilib make autoconf automake
  libtool flex bison gcc-doc gcc-9-doc libcc1-dbg libgomp1-dbg
  libitm1-dbg libatomic1-dbg libasan5-dbg liblsan0-dbg libtsan0-dbg
  libubsan1-dbg libquadmath0-dbg glibc-doc man-browser
The following NEW packages will be installed:
  cpp cpp-9 gcc gcc-9 gcc-9-base libasan5 libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgomp1 libisl19 libmpc3 libmpfr6 libubsan1 linux-libc-dev
  manpages manpages-dev
The following packages will be upgraded:
  binutils binutils-arm-linux-gnueabi binutils-common libatomic1 libbinutils
  libgcc1
6 upgraded, 18 newly installed, 0 to remove and 90 not upgraded.
Need to get 25.8 MB of archives.
After this operation, 72.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] ■
```

(5) 主机和 QEMU 虚拟机之间共享文件。

主机和 QEMU 虚拟机可以通过 NET_9P 技术进行文件共享，这个需要 QEMU 虚

拟机的 Linux 内核使能 NET_9P 的内核模块。本实验平台已经支持主机和 QEMU 虚拟机的共享文件，可以通过如下简单方法来测试。

我们在 runninglinuxkernel_4.0/kmodules 目录下面新建一个 test.c 文件。

```
$ cd runninglinuxkernel_4.0/kmodules
$ touch test.c
```

```
rlk@ubuntu:runninglinuxkernel_4.0$ cd kmodules/
rlk@ubuntu:kmodules$ ls
README
rlk@ubuntu:kmodules$ touch test.c
rlk@ubuntu:kmodules$
```

启动 QEMU 虚拟机之后，首先检查一下/mnt 目录是否有 test.c 文件。

```
/ # cd /mnt/
/mnt # ls
README      test.c
/mnt #
```

我们在后续的实验中会经常利用这个特性，比如把编译好的内核模块或者内核模块源代码放入 QEMU 虚拟机。

```
root@runninglinuxkernel:/mnt# ls
README  test.c
root@runninglinuxkernel:/mnt# pwd
/mnt
root@runninglinuxkernel:/mnt#
```

(6) 编译内核模块。

在本书中，常常需要编译内核模块然后放入到 QEMU 虚拟机中加载内核模块。我们可以在主机上通过交叉编译，然后共享到 QEMU 虚拟机。

进入实验代码目录，我们选择第 4 章的实验 1 的代码。

```
$ cd rlk_lab/rnk_basic/chapter_4/lab1_simple_module
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
$ $ make BASEINCLUDE=/home/rnk/rnk_base/runninglinuxkernel_4.0
```

把内核模块 ko 文件拷贝到 runninglinuxkernel_4.0/kmodules 目录下面。

```
$ cp mytest.ko runninglinuxkernel_4.0/kmodules
```

在 QEMU 虚拟机里的 mnt 目录可以看到这个 mytest.ko 模块。加载该内核模块。

```
$ insmod mytest.ko
```

1.6 实验 6：运行 Debian+ARM64 系统（新增）

1.6 实验 6：运行 Debian+ARM64 系统（新增）

1 实验目的

通过本实验学习如何编译和运行一个 ARM64 版本的 Debian 系统，并且在 QEMU 上运行起来。

2 实验步骤

在 runninglinuxkernel_4.0 目录下面有一个 rootfs_debian_arm64.tar.xz 文件，这个是基于 ARM64 版本的 Debian 系统的根文件系统。但是这个根文件系统还只是一个半成品，我们还需要根据编译好的内核来安装内核镜像和内核模块。整个过程比较复杂：

- 编译内核
- 编译内核模块
- 安装内核模块
- 安装内核头文件
- 安装编译内核模块必须依赖文件
- 制作 ext4 根文件系统

这个过程比较繁琐，作者制作了一个脚本来简化上述过程。

注意，该脚本会使用 dd 命令来生成一个 2GB 大小的镜像文件，因此主机系统需要保证至少需要 4 个 GB 的空余磁盘空间。若读者需要生成一个更大的根文件系统镜像，可以修该 run_debian_arm64.sh 这个脚本文件。

优麒麟 Linux 系统的 QEMU 工具包里包含了 qemu-system-aarch64 工具。安装如下工具包。

```
| $ sudo apt-get install gcc-aarch64-linux-gnu gcc-5-aarch64-linux-gnu
```

在优麒麟 Linux 系统中默认安装的 ARM64 版本 GCC 工具是 7.x 版本的，但是编译 Linux 4.0 内核需要使用 5.x 版本的 GCC 工具。因此，这里我们使用 update-alternatives 工具来切换 GCC 的版本，具体方法如下所示。

1) 设置gcc-5版本

```
$ sudo update-alternatives --install /usr/bin/aarch64-linux-gnu-gcc aarch64-linux-gnu-gcc /usr/bin/ aarch64-linux-gnu-gcc-5 5
```

2) 设置gcc-7版本

```
$ sudo update-alternatives --install /usr/bin/aarch64-linux-gnu-gcc aarch64-linux-gnu-gcc /usr/bin/ aarch64-linux-gnu-gcc-7 7
```

3) 选择使用gcc-5版本

```
$ sudo update-alternatives --config aarch64-linux-gnu-gcc
```

奔跑吧 linux 社区出品

```
rlk@ubuntu:runninglinuxkernel_4.0$ sudo update-alternatives --config aarch64-linux-gnu-gcc
There are 2 choices for the alternative aarch64-linux-gnu-gcc (providing /usr/bin/aarch64-linux-gnu-gcc).

Selection    Path                      Priority  Status
-----      -----
* 0          /usr/bin/aarch64-linux-gnu-gcc-7  7        auto mode
 1          /usr/bin/aarch64-linux-gnu-gcc-5      5        manual mode
 2          /usr/bin/aarch64-linux-gnu-gcc-7      7        manual mode

Press <enter> to keep the current choice[*], or type selection number: 1
```

如上图所示，选择 1，表示要选择 gcc-5 版本。

检查 aarch64-linux-gnu-gcc 版本是否为 5.5.0 版本。

```
rlk@ubuntu:runninglinuxkernel_4.0$ aarch64-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/5/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1'
--languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-
--without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-n
--enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique
--with-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --disabl
6_64-linux-gnu --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu-
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu/Linaro 5.5.0-12ubuntu1)
```

(1) 编译内核

```
$ cd /home/rilk/rilk_basic/runninglinuxkernel_4.0
$ ./run_debian_arm64.sh build_kernel
```

执行上述脚本需要几十分钟，依赖于主机的计算能力。

```
rlk@ubuntu:runninglinuxkernel_4.0$ ./run_debian_arm64.sh build_kernel
start build kernel image...
#
# configuration written to .config
#
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK  include/config/kernel.release
  WRAP arch/arm64/include/generated/asm/bugs.h
  WRAP arch/arm64/include/generated/asm/bug.h
  WRAP arch/arm64/include/generated/asm/clkdev.h
  WRAP arch/arm64/include/generated/asm/cputime.h
  WRAP arch/arm64/include/generated/asm/current.h
  WRAP arch/arm64/include/generated/asm/delay.h
  WRAP arch/arm64/include/generated/asm/div64.h
  WRAP arch/arm64/include/generated/asm/dma.h
  CHK  include/generated/uapi/linux/version.h
  WRAP arch/arm64/include/generated/asm/dma-contiguous.h
```

若出现如下错误，说明 aarch64-linux-gnu-gcc 的版本过高了。

1.6 实验 6：运行 Debian+ARM64 系统（新增）

```
scripts/Makefile.build:403: recipe for target 'scripts/mod' failed
make[1]: *** [scripts/mod] Error 2
make[1]: *** Waiting for unfinished jobs....
  SHIPPED scripts/genksyms/keywords.hash.c
  SHIPPED scripts/genksyms/parse.tab.h
HOSTCC  scripts/genksyms/parse.tab.o
HOSTCC  scripts/genksyms/lex.lex.o
CC      kernel/bounds.s
In file included from include/linux/compiler.h:54:0,
                 from include/uapi/linux/stddef.h:1,
                 from include/linux/stddef.h:4,
                 from ./include/uapi/linux posix_types.h:4,
                 from include/uapi/linux/types.h:13,
                 from include/linux/types.h:5,
                 from include/linux/page-flags.h:8,
                 from kernel/bounds.c:9:
include/linux/compiler-gcc.h:107:1: fatal error: linux/compiler-gcc7.h: No such file or directory
 #include <gcc_header(_GNUC_)>
 ^~~~~
compilation terminated.
```

需要使用 update-alternatives 命令来把它切换到 5.5 的版本。

(2) 编译 ARM64 版本的 Debian 系统 rootfs

```
| $ sudo ./run_debian_arm64.sh build_rootfs
```

注意：这里需要使用 root 权限。

编译完成后会生成一个 rootfs_debian_arm64.ext4 的文件系统。

(3) 运行 debian 系统。

```
| $ ./run_debian_arm64.sh run
```

注意：运行此命令不需要 root 权限。

```
[ OK ] Started System Logging Service.
[ OK ] Started Getty on tty4.
[ OK ] Started Getty on tty2.
[ OK ] Started Getty on tty1.
[ OK ] Started Serial Getty on ttyAMA0.
[ OK ] Started Getty on tty3.
[FAILED] Failed to start Remove Sta...ext4 Metadata Check Snapshots.
See 'systemctl status e2scrub_reap.service' for details.
[ OK ] Started Getty on tty6.
[ OK ] Started Getty on tty5.
[ OK ] Started getty on tty2-tty6... and logind are not available.
[ OK ] Reached target Login Prompts.
[ OK ] Reached target Multi-User System.
[ OK ] Reached target Graphical Interface.
          Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Update UTMP about System Runlevel Changes.

Debian GNU/Linux 10 runninglinuxkernel ttyAMA0

runninglinuxkernel login:
runninglinuxkernel login: root
Password:
Linux runninglinuxkernel 4.0.0+ #1 SMP Sun Aug 18 12:29:01 PDT 2019 armv7l
Welcome Running Linux Kernel.
Created by Ben Shushu <runninglinuxkernel@126.com>

Buy linux kernel training course:
https://shop115683645.taobao.com/

Wechat: runninglinuxkernel

root@runninglinuxkernel:~#
```

Debian 系统的用户名为: root, 密码为: 123

(4) 在线安装软件包

QEMU 虚拟机可以通过 VirtIO-NET 技术来生成一个虚拟的网卡，并且通过 NAT 网络桥接技术和主机进行网络共享。首先使用 ifconfig 命令来检查网络配置。

```
root@runninglinuxkernel:~# 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>
          ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)
            RX packets 19 bytes 2560 (2.5 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 29 bytes 2660 (2.5 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
          loop txqueuelen 0 (Local Loopback)
            RX packets 0 bytes 0 (0.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 0 bytes 0 (0.0 B)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@runninglinuxkernel:~#
```

可以看到生成了一个名为 eth0 的网卡设备，分配的 IP 地址为: 10.0.2.15。

通过 apt update 命令来更新 Debian 系统的软件仓库。

```
$ apt update
```

```
root@runninglinuxkernel:~# apt update
Get:1 http://mirrors.ustc.edu.cn/debian unstable InRelease [149 kB]
Get:2 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.diff/Index [27.9 kB]
Ign:3 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.diff/Index
Get:3 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.diff/Index [27.9 kB]
Ign:4 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.diff/Index
Get:5 http://mirrors.ustc.edu.cn/debian unstable/non-free armel Packages.diff/Index [27.8 kB]
Get:6 http://mirrors.ustc.edu.cn/debian unstable/non-free Translation-en.diff/Index [27.8 kB]
Get:7 http://mirrors.ustc.edu.cn/debian unstable/contrib armel Packages.diff/Index [27.8 kB]
```

如果更新失败，有可能是系统时间比较旧了，可以使用 date 命令来设置日期。

```
root@benshushu:~# date -s 2019-04-25 #假设最新日期是2019年4月25日
Thu Apr 25 00:00:00 UTC 2019
```

使用 apt install 命令来安装软件包。比如，可以在线安装 gcc。

1.6 实验 6：运行 Debian+ARM64 系统（新增）

```
root@runninglinuxkernel:~# apt install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-arm-linux-gnueabi binutils-common cpp cpp-9 gcc-9
  gcc-9-base libasan5 libatomic1 libbinutils libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgcc1 libgomp1 libisl19 libmpc3 libmpfr6 libubsan1
  linux-libc-dev manpages manpages-dev
Suggested packages:
  binutils-doc cpp-doc gcc-9-locales gcc-multilib make autoconf automake
  libtool flex bison gdb gcc-doc gcc-9-doc libgcc1-dbg libgomp1-dbg
  libitm1-dbg libatomic1-dbg libasan5-dbg libtsan0-dbg libtsan0-dbg
  libubsan1-dbg libquadmath0-dbg glibc-doc man-browser
The following NEW packages will be installed:
  cpp cpp-9 gcc gcc-9 gcc-9-base libasan5 libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgomp1 libisl19 libmpc3 libmpfr6 libubsan1 linux-libc-dev
  manpages manpages-dev
The following packages will be upgraded:
  binutils binutils-arm-linux-gnueabi binutils-common libatomic1 libbinutils
  libgcc1
6 upgraded, 18 newly installed, 0 to remove and 90 not upgraded.
Need to get 25.8 MB of archives.
After this operation, 72.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] ■
```

(5) 主机和 QEMU 虚拟机之间共享文件。

主机和 QEMU 虚拟机可以通过 NET_9P 技术进行文件共享，这个需要 QEMU 虚拟机的 Linux 内核使能 NET_9P 的内核模块。本实验平台已经支持主机和 QEMU 虚拟机的共享文件，可以通过如下简单方法来测试。

我们在 runninglinuxkernel_4.0/kmodules 目录下面新建一个 test.c 文件。

```
$ cd runninglinuxkernel_4.0/kmodules
$ touch test.c
```

```
rlk@ubuntu:runninglinuxkernel_4.0$ cd kmodules/
rlk@ubuntu:kmodules$ ls
README
rlk@ubuntu:kmodules$ touch test.c
rlk@ubuntu:kmodules$ ■
```

启动 QEMU 虚拟机之后，首先检查一下/mnt 目录是否有 test.c 文件。

```
/ # cd /mnt/
/mnt # ls
README      test.c
/mnt #
```

我们在后续的实验中会经常利用这个特性，比如把编译好的内核模块或者内核模块源代码放入 QEMU 虚拟机。

```
root@runninglinuxkernel:/mnt# ls
README  test.c
root@runninglinuxkernel:/mnt# pwd
/mnt
root@runninglinuxkernel:/mnt# ■
```

1.7 实验 7：运行 Debian+X86_64 系统（新增）

1 实验目的

通过本实验学习如何编译和运行一个 x86_64 版本的 Debian 系统，并且在 QEMU 上运行起来。

2 实验步骤

在 runninglinuxkernel_4.0 目录下面有一个 rootfs_debian_arm64.tar.xz 文件，这个是基于 x86_64 版本的 Debian 系统的根文件系统。但是这个根文件系统还只是一个半成品，我们还需要根据编译好的内核来安装内核镜像和内核模块。整个过程比较复杂：

- 编译内核
- 编译内核模块
- 安装内核模块
- 安装内核头文件
- 安装编译内核模块必须依赖文件
- 制作 ext4 根文件系统

这个过程比较繁琐，作者制作了一个脚本来简化上述过程。

注意，该脚本会使用 dd 命令来生成一个 2GB 大小的镜像文件，因此主机系统需要保证至少需要 4 个 GB 的空余磁盘空间。若读者需要生成一个更大的根文件系统镜像，可以修该 run_debian_x86_64.sh 这个脚本文件。

(1) 选择 GCC 版本。

首选安装一个 gcc 5 版本的编译器。

```
# sudo apt install gcc-5
```

使用 update-alternatives 命令来配置一个 gcc-5 的选项。

```
| sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 5
```

```
| rlk@ubuntu:runninglinuxkernel_4.0$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 5
rlk@ubuntu:runninglinuxkernel_4.0$
```

然后使用 update-alternatives 命令来选在 GCC 5.5 版本。

```
root@figo-OptiPlex-9020:~# update-alternatives --config gcc
There are 4 choices for the alternative gcc (providing /usr/bin/gcc).
      Selection    Path        Priority   Status
      -----      /-----      -----   -----
* 0            /usr/bin/gcc-8  8          auto mode
* 1            /usr/bin/gcc-5  5          manual mode
  2            /usr/bin/gcc-6  6          manual mode
  3            /usr/bin/gcc-7  7          manual mode
  4            /usr/bin/gcc-8  8          manual mode

Press <enter> to keep the current choice[*], or type selection number: |
```

1.7 实验 7：运行 Debian+X86_64 系统（新增）

如图选择 1，来选择 GCC-5。

(2) 编译内核

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ ./run_debian_x86_64.sh build_kernel
```

执行上述脚本需要几十分钟，依赖于主机的计算能力。

```
rbk@ubuntu:runninglinuxkernel_4.0$ ./run_debian_x86_64.sh build_kernel
start build kernel image...
#
# configuration written to .config
#
scripts/kconfig/conf --silentoldconfig Kconfig
SYSTBL arch/x86/syscalls/../include/generated/asm/syscalls_32.h
SYSHDR arch/x86/syscalls/../include/generated/asm/unistd_32_ia32.h
SYSHDR arch/x86/syscalls/../include/generated/asm/unistd_64_x32.h
CHK include/config/kernel.release
SYSTBL arch/x86/syscalls/../include/generated/asm/syscalls_64.h
SYSHDR arch/x86/syscalls/../include/generated/uapi/asm/unistd_32.h
SYSHDR arch/x86/syscalls/../include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/syscalls/../include/generated/uapi/asm/unistd_x32.h
WRAP arch/x86/include/generated/asm/clkdev.h
WRAP arch/x86/include/generated/asm/cputime.h
WRAP arch/x86/include/generated/asm/dma-contiguous.h
```

若编译内核出现如下错误，说明 gcc 版本过高。

```
WRAP arch/x86/include/generated/asm/mcs_spinlock.h
WRAP arch/x86/include/generated/asm/scatterlist.h
CHK include/generated/uapi/linux/version.h
> CHK include/generated/utsrelease.h
CC scripts/mod/empty.o
HOSTCC scripts/selinux/genheaders/genheaders
CC scripts/mod/devicetable-offsets.s
HOSTCC scripts/selinux/mdp/mdp
In file included from include/linux/compiler.h:54:0,
                 from include/uapi/linux/stddef.h:1,
                 from include/linux/stddef.h:4,
                 from ./include/uapi/linux/posix_types.h:4,
                 from include/uapi/linux/types.h:13,
                 from include/linux/types.h:5,
                 from include/linux/mod devicetable.h:11,
                 from scripts/mod/devicetable-offsets.c:2:
include/linux/compiler-gcc.h:107:1: fatal error: linux/compiler-gcc7.h: No such file or directory
 #include <gcc_header(<_GNUC_>)
 ^~~~~
compilation terminated.
scripts/Makefile.build:153: recipe for target 'scripts/mod/devicetable-offsets.s' failed
make[2]: *** [scripts/mod/devicetable-offsets.s] Error 1
scripts/Makefile.build:403: recipe for target 'scripts/mod' failed
make[1]: *** [scripts/mod] Error 2
make[1]: *** Waiting for unfinished jobs....
Makefile:557: recipe for target 'scripts' failed
make: *** [scripts] Error 2
make: *** Waiting for unfinished jobs....
rbk@ubuntu:runninglinuxkernel_4.0$
```

应该使用 update-alternatives 来选择 gcc-5 版本。

```
rbk@ubuntu:runninglinuxkernel_4.0$ sudo update-alternatives --config gcc
There are 2 choices for the alternative gcc (providing /usr/bin/gcc).

      Selection    Path        Priority   Status
      -----      -----
* 0            /usr/bin/gcc-7    7          auto mode
  1            /usr/bin/gcc-5    5          manual mode
  2            /usr/bin/gcc-7    7          manual mode

Press <enter> to keep the current choice[*], or type selection number:
```

(3) 编译 x86_64 版本的 Debian 系统 rootfs

```
| $sudo ./run_debian_x86_64.sh build_rootfs
```

注意：这里需要使用 root 权限。

编译完成后会生成一个 rootfs_debian_x86.ext4 的文件系统。

(4) 运行 debian 系统。

```
| $ ./run_debian_x86_64.sh run
```

注意：运行此命令不需要 root 权限。



```

Starting Rotate log files...
Starting Raise network interfaces...
[OK] Started LSB: Execute the kexec -e command to reboot system.
[OK] Found device Virtio network device.
[OK] Started ifup for enp0s4.
Starting LSB: Load kernel image with kexec...
[OK] Started getty on tty2-tty6, and login are not available.
[OK] Started LSB: Load kernel image with kexec.
[OK] Started Raise network interfaces.
[OK] Reached target Network.
Starting Permit User Sessions...
[OK] Reached target System is Online.
Starting Kernel crash dump capture service...
Starting Daily apt download activities...
[OK] Started Permit User Sessions.
[OK] Started Getty on tty3.
[OK] Started Getty on tty5.
[OK] Started Getty on tty1.
[OK] Started Getty on tty4.
[OK] Started Getty on tty2.
[OK] Started Serial Getty on ttys0.
[OK] Started Getty on ttys6.
[OK] Reached target Login Prompts.
[OK] Started Rotate log files.
[45.639710] kdump-tools[1213]: Starting kdump-tools: Creating symlink /var/lib/kdump/vmlinuz.
[47.592654] kdump-tools[1213]: kdump-tools: Generating /var/lib/kdump/initrd.img-4.0.0+
Debian GNU/Linux 10 runninglinuxkernel ttyS0
runninglinuxkernel login: [ 54.620511] kdump-tools[1213]: find: '/var/tmp/mkinitramfs_bTCROC/lib/modules'

```

Debian 系统的用户名为: root, 密码为: 123

(5) 在线安装软件包

QEMU 虚拟机可以通过 VirtIO-NET 技术来生成一个虚拟的网卡，并且通过 NAT 网络桥接技术和主机进行网络共享。首先使用 ifconfig 命令来检查网络配置。

1.7 实验 7：运行 Debian+X86_64 系统（新增）

```
root@runninglinuxkernel:~# 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 fe80::5054:ff:fe12:3456 prefixlen 64 scopeid 0x40<site>
        ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)
        RX packets 19 bytes 2560 (2.5 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 29 bytes 2660 (2.5 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
        loop txqueuelen 0 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@runninglinuxkernel:~#
```

可以看到生成了一个名为 eth0 的网卡设备，分配的 IP 地址为：10.0.2.15。

通过 apt update 命令来更新 Debian 系统的软件仓库。

```
$ apt update
```

```
root@runninglinuxkernel:~# apt update
Get:1 http://mirrors.ustc.edu.cn/debian unstable InRelease [149 kB]
Get:2 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.diff/Index [27.9 kB]
Ign:2 http://mirrors.ustc.edu.cn/debian unstable/main armel Packages.diff/Index
Get:3 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.diff/Index [27.9 kB]
Ign:3 http://mirrors.ustc.edu.cn/debian unstable/main Translation-en.diff/Index
Get:4 http://mirrors.ustc.edu.cn/debian unstable/non-free armel Packages.diff/Index [27.8 kB]
Get:5 http://mirrors.ustc.edu.cn/debian unstable/non-free Translation-en.diff/Index [27.8 kB]
Get:6 http://mirrors.ustc.edu.cn/debian unstable/contrib armel Packages.diff/Index [27.8 kB]
Get:7 http://mirrors.ustc.edu.cn/debian unstable/contrib Translation-en.diff/Index [27.8 kB]
```

如果更新失败，有可能是系统时间比较旧了，可以使用 date 命令来设置日期。

```
root@benshushu:~# date -s 2019-04-25 #假设最新日期是2019年4月25日
Thu Apr 25 00:00:00 UTC 2019
```

使用 apt install 命令来安装软件包。比如，可以在线安装 gcc。

```
root@runninglinuxkernel:~# apt install gcc
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  binutils binutils-arm-linux-gnueabi binutils-common cpp cpp-9 gcc-9
  gcc-9-base libasan5 libatomic1 libbinutils libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgcc1 libgomp1 libisl19 libmpc3 libmpfr6 libubsan1
  linux-libc-dev manpages manpages-dev
Suggested packages:
  binutils-doc cpp-doc gcc-9-locales gcc-multilib make autoconf automake
  libtool flex bison gdb gcc-doc gcc-9-doc libgcc1-dbg libgomp1-dbg
  libitm1-dbg libatomic1-dbg libasan5-dbg liblsan0-dbg libtsan0-dbg
  libubsan1-dbg libquadmath0-dbg glibc-doc man-browser
The following NEW packages will be installed:
  cpp cpp-9 gcc gcc-9 gcc-9-base libasan5 libc-dev-bin libc6-dev libcc1-0
  libgcc-9-dev libgomp1 libisl19 libmpc3 libmpfr6 libubsan1 linux-libc-dev
  manpages manpages-dev
The following packages will be upgraded:
  binutils binutils-arm-linux-gnueabi binutils-common libatomic1 libbinutils
  libgcc1
6 upgraded, 18 newly installed, 0 to remove and 90 not upgraded.
Need to get 25.8 MB of archives.
After this operation, 72.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] ■
```

(6) 主机和 QEMU 虚拟机之间共享文件。

主机和 QEMU 虚拟机可以通过 NET_9P 技术进行文件共享，这个需要 QEMU 虚拟机的 Linux 内核使能 NET_9P 的内核模块。本实验平台已经支持主机和 QEMU 虚拟机的共享文件，可以通过如下简单方法来测试。

我们在 runninglinuxkernel_4.0/kmodules 目录下面新建一个 test.c 文件。

```
$ cd runninglinuxkernel_4.0/kmodules
$ touch test.c
```

```
rlk@ubuntu:runninglinuxkernel_4.0$ cd kmodules/
rlk@ubuntu:kmodules$ ls
README
rlk@ubuntu:kmodules$ touch test.c
rlk@ubuntu:kmodules$
```

启动 QEMU 虚拟机之后，首先检查一下/mnt 目录是否有 test.c 文件。

```
/ # cd /mnt/
/mnt # ls
README      test.c
/mnt #
```

我们在后续的实验中会经常利用这个特性，比如把编译好的内核模块或者内核模块源代码放入 QEMU 虚拟机。

```
root@runninglinuxkernel:/mnt# ls
README test.c
root@runninglinuxkernel:/mnt# pwd
/mnt
root@runninglinuxkernel:/mnt#
```

1.8 实验 8：手把手制作一个 Debian rootfs 系统（新增）

1 实验目的

通过本实验学习如何制作一个基于 Debian 系统的 rootfs 文件系统，并且在 QEMU 中运行该 Debian 系统。本实验以 ARM32 为例，读者可以根据该实验方法来制作 ARM64 版本、x86_64 版本以及 RISC_V 版本的 Debian 根文件系统。

2 实验步骤

注意：本实验需要连接互联网。

本实验的主机是安装了 Ubuntu 18.04.2 系统主机或者 vmware 虚拟机。

在 Ubuntu 系统里有一个 debootstrap 的工具，可以帮助我们快速制作指定架构的根文件系统。debootstrap 命令最早源自 debian 系统，用来引导一个基础 Debian 系统（一种符合 Linux 文件系统标准(FHS)的根文件系统）。

(1) 安装 debootstrap 工具

1.8 实验 8 : 手把手制作一个 Debian rootfs 系统 (新增)

```
# sudo apt-get install binfmt-support qemu qemu-user-static debootstrap
```

(2) 用 debootstrap 工具

debootstrap 命令格式如下:

```
usage: [OPTION]... <suite> <target> [<mirror> [<script>]]
```

使用 debootstrap 制作根文件系统会分成两个阶段。

第一阶段是，使用 debootstrap 命令来下载软件包。

```
# sudo su
# mkdir -p myrootfs_arm32

#sudo      debootstrap      --arch=armel      --foreign      buster      myrootfs_arm32/
http://mirrors.ustc.edu.cn/debian/
```

- --arch: 指定要制作文件系统的处理器体系结构，比如 armel 或者 arm64
- buster: 指定 Debian 的版本。buster 是 Debian 10 系统。
- myrootfs_arm32: 本地目录，最后制作好的文件系统会在此目录。本实验使用“myrootfs_arm32”目录作为根文件系统目录。
- --foreign: 只执行引导的初始解包阶段，仅仅下载和解压。
- <http://mirrors.ustc.edu.cn/debian/>: 国内 debian 镜像源地址

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# debootstrap --arch=armel --foreign buster myrootfs_arm32/
http://mirrors.ustc.edu.cn/debian/
W: Cannot check Release signature; keyring file not available /usr/share/keyrings/debian-archive-keyring.gpg
I: Retrieving InRelease
I: Retrieving Packages
I: Validating Packages
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Found additional required dependencies: adduser debian-archive-keyring fdisk gcc-8-base gpgv libacl1 libapt-pkg5.0 libattr1 libaudit-common libaudit1 libblkid1 libbz2-1.0 libc6 libcap-ng0 libcom-err2 libdb5.3 libdebconfclient0 libext2fs2 libfdisk1 libff16 libgcc1 libgcrypt20 libgmp10 libgnutls30 libgpg-error0 libhogweed4 libidn2-0 liblzo2-1 liblzma5 libmount1 libncursesw6 libnettle6 libp11-kit0 libpam0g libpcre3 libseccomp2 libselinux1 libsemanage-common libsemanage1 libsep0 libsmartcols1 libss2 libstdc++6 libsystemd0 libtasn1-6 libtinfo6 libudev libunistring2 libuuid1 libzstd1 zlib1g
I: Found additional base dependencies: dmsetup libapparmor1 libapt-inst2.0 libargon2-1 libatomic1 libbsd0 libcap2 libcap2-bin libcryptsetup12 libdevmapper1.02.1 libdns-export1104 libestr0 libfastjson4 libidn11 libip4tc0 libiptc0 libiptc0 libisc-export1100 libjson-c3 libkmod2 liblocale-gettext-perl liblognorm5 libmnl0 libncurses6 libnetfilter-conntrack3 libnewt0.52 libnftnlink0 libnftn11 libpopt0 libprocps7 libslang2 libssl1.1 libtext-charwidth-perl libtext-iconv-perl libtext-wrap18n-perl libxtables12 lsb-base xx
I: Checking component main on http://mirrors.ustc.edu.cn/debian...
I: Retrieving libacl1 2.2.53-4
```

第二阶段，需要安装软件包。

因为主机跑在 x86 架构上，而我们要制作的文件系统是跑在 ARM32 上，因此可以使用 qemu-arm-static 来模拟成 arm32 环境的执行环境。

```
# cp /usr/bin/qemu-arm-static myrootfs_arm32/usr/bin/
```

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# cp /usr/bin/qemu-arm-static myrootfs_arm32/usr/bin/
root@figo-OptiPlex-9020:runninglinuxkernel_4.0#
```

下面使用 debootstrap 命令进行软件包的安装和配置。

```
# chroot myrootfs_arm32/ debootstrap/debootstrap --second-stage
```

其中命令参数--second-stage 表示执行第二阶段的安装。

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# chroot myrootfs_arm32/ debootstrap/debootstrap --second-stage
I: Installing core packages...
I: Unpacking required packages...
I: Unpacking libacl1:armel...
I: Unpacking adduser...
I: Unpacking apt...
I: Unpacking libapt-pkg5.0:armel...
I: Unpacking libattr1:armel...

I: Configuring libnewt0.52:armel...
I: Configuring apt-utils...
I: Configuring iproute2...
I: Configuring cron...
I: Configuring rsyslog...
I: Configuring isc-dhcp-client...
I: Configuring debconf-i18n...
I: Configuring libnftnl11:armel...
I: Configuring vim-tiny...
I: Configuring ifupdown...
I: Configuring bsdmainutils...
I: Configuring whiptail...
I: Configuring libnetfilter-conntrack3:armel...
I: Configuring iptables...
I: Configuring tasksel-data...
I: Configuring tasksel...
I: Configuring libc-bin...
I: Configuring systemd...
I: Base system installed successfully.
root@figo_OptiPlex-9020:runninglinuxkernel_4.0#
```

显示“**I: Base system installed successfully.**”这句话，说明第二阶段已经完成。

(3) 设置根文件系统的用户名和密码

使用 chroot 命令切换到刚才制作的根文件系统。

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0#
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# chroot myrootfs_arm32/
root@figo-OptiPlex-9020:#
root@figo-OptiPlex-9020:#
```

为 root 用户设置密码：

1.8 实验 8 : 手把手制作一个 Debian rootfs 系统 (新增)

```
root@figo-OptiPlex-9020:/# passwd root
New password:
Retype new password:
passwd: password updated successfully
root@figo-OptiPlex-9020:/#
```

配置网卡支持 DHCP 协议。修改/etc/network/interfaces 文件，增加如下内容。

```
auto lo
iface lo inet loopback

allow-hotplug eth0
iface eth0 inet dhcp
```

输入 exit 退出。基于 Debian 系统的根文件系统已经制作完成。

(4) 编译 ARM32 内核

runninglinuxkernel 里内置了一个 debian_defconfig 配置文件，我们使用该 config 文件来编译内核。

```
$ cd /home/rkk/rkk_basic/runninglinuxkernel_4.0
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make debian_defconfig
$ make -j8
```

(5) 制作 ext4 文件系统

下面要使用 dd 和 mout 命令来制作 ext4 文件系统了。

- 首先使用 dd 命令来创建一个 image 文件。bs=1M 表示 block 大小，count=2048，表示该 image 大小为 2GB，读者可以根据主机磁盘空间，适当调整该大小，建议不小于 2GB。

```
dd if=/dev/zero of=myrootfs_arm32.ext4 bs=1M count=2048
```

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# dd if=/dev/zero of=myrootfs_arm32.ext4 bs=1M count=2048
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB, 2.0 GiB) copied, 16.6955 s, 129 MB/s
```

- 使用 mkfs.ext4 命令来格式化

```
root@figo-OptiPlex-9020:runninglinuxkernel_4.0# mkfs.ext4 myrootfs_arm32.ext4
mke2fs 1.44.1 (24-Mar-2018)
Discarding device blocks: done
Creating filesystem with 524288 4k blocks and 131072 inodes
Filesystem UUID: 2d10a69e-78f8-4ecd-82da-73ed724d2ad9
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

root@figo-OptiPlex-9020:runninglinuxkernel_4.0#
```

3. 挂载 ext4 文件系统并拷贝内容

```
#mkdir -p tmpfs
# mount -t ext4 myrootfs_arm32.ext4 tmpfs/ -o loop

# cp -af myrootfs_arm32/* tmpfs/

# umount tmpfs

# chmod 777 myrootfs_arm32.ext4
```

这样，我们的 ext4 的文件系统就制作完成了。

(6) 运行 Debian 系统。

使用 qemu-system-arm 命令来跑我们刚才制作的 Debian 系统。

```
#qemu-system-arm -m 1024 -M virt -nographic -smp 4 -kernel
arch/arm/boot/zImage -append "root=/dev/vda rootfstype=ext4 rw" -drive
if=none,file=myrootfs_arm32.ext4,id=hd0 -device virtio-blk-device,drive=hd0 -
netdev user,id=mynet -device virtio-net-device,netdev=mynet --fsdev
local,id=kmod_dev,path=~/kmodules,security_model=none -device virtio-9p-
device,fsdev=kmod_dev,mount_tag=kmod_mount
```

其中：

- “-drive if=none,file=myrootfs_arm32.ext4,id=hd0 -device virtio-blk-device,drive=hd0” 添加根文件系统支持
- “-netdev user,id=mynet -device virtio-net-device,netdev=mynet” 添加对网络支持
- “--fsdev local,id=kmod_dev,path=~/kmodules,security_model=none -device virtio-9p-device,fsdev=kmod_dev,mount_tag=kmod_mount” 添加对 Host 主机和 QEMU 虚拟机之间文件共享

1.8 实验 8 : 手把手制作一个 Debian rootfs 系统 (新增)

```

root@figo-OptiPlex-9020:runninglinuxkernel_4.0# qemu-system-arm -m 1024 -M virt -nographic -smp 4 -kernel arch/arm/boot/zImage -append "root=/dev/vda rootfstype=ext4 rw" -drive if=none,file=myrootfs_arm32.ext4,id=hd0 -device virtio-blk-device,drive=hd0 -netdev user,id=mynet -device virtio-net-device,netdev=mynet -fsdev local,id=kmod_dev,path=../kmodules,security_model=none -device virtio-9p-device,fsdev=kmod_dev,mount_t_tag=kmod_mount
WARNING: Image format was not specified for 'myrootfs_arm32.ext4' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[ OK ] Booting Linux on physical CPU 0x0
[ OK ] Initializing cgroup subsys cpuset
[ OK ] Linux version 4.0.0+ (root@figo-OptiPlex-9020) (gcc version 5.5.0 20171010 (Ubuntu/Linaro 5.5.0-12ubuntu1) ) #7 SMP Mon Aug 19 19:07:21 CST 2019
[ OK ] CPU: ARMv7 Processor [412fc0f1] revision 1 (ARMv7), cr=10c5387d
[ OK ] CPU: PIPT / VIPT nonaliasing data cache, PIPT instruction cache
[ OK ] Machine model: linux,dummy-virt
[ OK ] Memory policy: Data cache writealloc
[ OK ] psci: probing for conduit method from DT.
[ OK ] psci: PSCIv0.2 detected in firmware.
[ OK ] psci: Using standard PSCI v0.2 function IDs
[ OK ] PERCPU: Embedded 11 pages/cpu @eefc1000 s16268 r8192 d20596 u45056
[ OK ] Built 1 zonelists in Zone order, mobility grouping on. Total pages: 260624
[ OK ] Kernel command line: root=/dev/vda rootfstype=ext4 rw
[ OK ] log_buf_len individual max cpu contribution: 4096 bytes
[ OK ] log_buf_len total cpu_extra contributions: 12288 bytes
[ OK ] log_buf_len min size: 16384 bytes
[ OK ] log_buf_len: 32768 bytes
[ OK ] early log buf free: 14768(90%)
[ OK ] PID hash table entries: 4096 (order: 2, 16384 bytes)
[ OK ] Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)

Starting Permit User Sessions...
[ OK ] Started Permit User Sessions.
[ OK ] Started System Logging Service.
[ OK ] Started Getty on tty6.
[ OK ] Started Getty on tty5.
[ OK ] Started Getty on tty4.
[ OK ] Started Getty on tty3.
[ OK ] Started Serial Getty on ttyAMA0.
[ OK ] Started Getty on tty2.
[ OK ] Started Getty on tty1.
[ OK ] Started Getty on tty6... and loginctl are not available.
[ OK ] Reached target Login Prompts.
[ OK ] Reached target Multi-User System.
[ OK ] Reached target Graphical Interface.
Starting Update UTMP about System Runlevel Changes...
[ OK ] Started Update UTMP about System Runlevel Changes.

Debian GNU/Linux 10 figo-OptiPlex-9020 ttyAMA0

figo-OptiPlex-9020 login:
figo-OptiPlex-9020 login: root
Password:
Linux figo-OptiPlex-9020 4.0.0+ #7 SMP Mon Aug 19 19:07:21 CST 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@figo-OptiPlex-9020:#

```

我们可以看到登录到刚才制作的 Debian 系统中了。

使用 apt 命令来更新源。

```

root@figo-OptiPlex-9020:~# apt update
Hit:1 http://cdn-fastly.deb.debian.org/debian buster InRelease
0% [Working] [ 433.021176] random: nonblocking pool is initialized
Get:2 http://cdn-fastly.deb.debian.org/debian buster/main Translation-en [5967 kB]
Fetched 5967 kB in 25s (240 kB/s)
Reading package lists... Done
Building dependency tree... Done
All packages are up to date.

```

ifconfig 命令没有包含在我们刚才制作的根文件系统里，可以使用如下命令来安装一个。

```
root@figo-OptiPlex-9020:~# apt install net-tools
Reading package lists... Done
Building dependency tree... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 230 kB of archives.
After this operation, 877 kB of additional disk space will be used.
Get:1 http://cdn-fastly.deb.debian.org/debian buster/main armel net-tools armel 1 [230 kB]
Fetched 230 kB in 21s (11.2 kB/s)
Selecting previously unselected package net-tools.
(Reading database ... 9205 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20180626.aebd88e-1_armel.deb ...
Unpacking net-tools (1.60+git20180626.aebd88e-1) ...
```

使用 ifconfig 命令来查看网卡的 IP 地址。

```
root@figo-OptiPlex-9020:~# 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 4431 bytes 6445464 (6.1 MiB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 1952 bytes 110216 (107.6 KiB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
  inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
      loop txqueuelen 0 (Local Loopback)
        RX packets 168 bytes 16576 (16.1 KiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 168 bytes 16576 (16.1 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@figo-OptiPlex-9020:~#
```

1.9 实验 9：配置 QEMU 虚拟机的桥接网络（新增）

1 实验目的

通过本实验学习如何配置 QEMU 虚拟的网络。

注意：本实验选做。学有余力的同学可以尝试做本实验。

2 实验要求

本实验的要求是，在 Windows 主机能 ping 通 QEMU+Debian 系统中的 IP 地址。

我们在实验 5~8 中，运行了一个基于 runninglinuxkernel 的 QEMU+Debian 的系统。虽然可以连接互联网并且通过 apt 命令下载软件包，但是我们却不能通过 ping 通 QEMU 虚拟机里的 Debian 系统。

1.9 实验 9：配置 QEMU 虚拟机的桥接网络（新增）

```
root@runninglinuxkernel:~# 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 19 bytes 2560 (2.5 KiB)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 29 bytes 2660 (2.5 KiB)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
                inet6 ::1 prefixlen 128 scopeid 0x10<host>
                    loop txqueuelen 0 (Local Loopback)
                    RX packets 0 bytes 0 (0.0 B)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 0 bytes 0 (0.0 B)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@runninglinuxkernel:~#
```

如上图所示，QEMU 虚拟机的 ip 地址为：10.0.2.15，我们配置的 QEMU 虚拟机采用 NAT 的桥接方式。

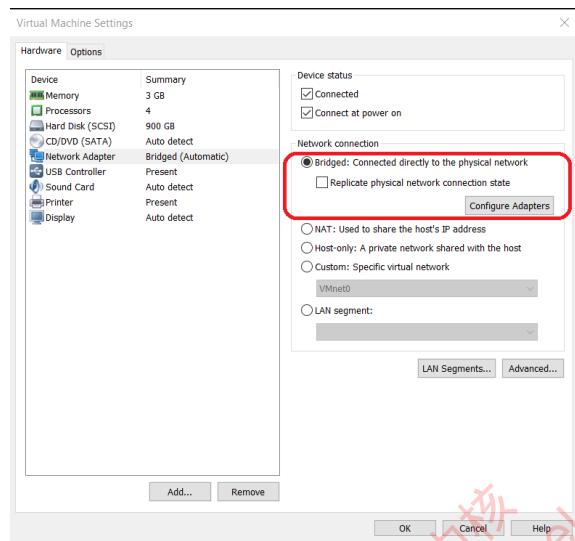
QEMU 里有两种方式网络桥接方式：

1. 用户模式。用户模式网络(Usermode Networking)，数据包由 NAT 方式通过主机的接口进行传送。
2. 桥接模式。使用桥接方式(Bridged Networking)，外部的机器可以直接联通到虚拟机，就像联通到你的主机一样。

目前 runninglinuxkernel 默认采用第一种方法，这种网络方式比较简单。若想我们的主机可以 ping 通 QEMU 虚拟机，我们需要实现第二种方式。

3 实验步骤

首先在 VMware Player 的 Virtual Machine Settings 中配置网络的桥接方式。



之前的配置是默认选择了“NAT: Used to share the host's IP address”，我们这里要重新选择“Bridged: Connected directly to the physical network”。

重新启动 Vmware player。

在 Vmware 虚拟机中的优麒麟 Linux 系统中。

(1) 在主机上创建一个网络桥

首先安装必要的软件包^①。

```
$ sudo apt install net-tools bridge-utils
```

查看 Vmware 里的优麒麟 Linux 中的网络设备情况。

```
rlk@ubuntu:rlk_basic$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.238.170.56  netmask 255.255.254.0  broadcast 10.238.171.255
          inet6 fe80::421c:e3d4:6ba3:5fed  prefixlen 64  scopeid 0x20<link>
            ether 00:0c:29:21:aa:8c  txqueuelen 1000  (Ethernet)
              RX packets 8  bytes 1136 (1.1 KB)
              RX errors 0  dropped 0  overruns 0  frame 0
              TX packets 58  bytes 6794 (6.7 KB)
              TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
          inet6 ::1  prefixlen 128  scopeid 0x10<host>
            loop  txqueuelen 1000  (Local Loopback)
              RX packets 136  bytes 9402 (9.4 KB)
              RX errors 0  dropped 0  overruns 0  frame 0
              TX packets 136  bytes 9402 (9.4 KB)
              TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

rlk@ubuntu:rlk_basic$
```

从上图我们可以得出两个信息：

网络设备名：ens33

^①本书配套的 vmware v4.0 镜像里没有包含此软件包，读者需要自行下载。

1.9 实验 9：配置 QEMU 虚拟机的桥接网络（新增）

ip 地址： 10.238.170.56

编辑一个脚本，文件名为： br.sh。

```
#!/bin/sh
netname="ens33"
ip=`ifconfig $netname|grep "inet "|awk '{print $2}'` 
netmask=`ifconfig $netname|grep "inet "|awk '{print $4}'` 
gateway=`route -n|grep UG|awk '{print $2}'` 
brctl addbr br0
ifconfig br0 $ip netmask $netmask
ifconfig $netname 0.0.0.0
brctl addif br0 $netname
route add default gw $gateway
echo "nameserver 114.114.114.114" >> /etc/resolv.conf
```

注意：上面第二行 **netname** 必须是网络设备名，在本例中，网络设备名为 ens33

执行上述脚本。

```
$ sudo sh br.sh
```

我们在执行 ifconfig 命令来查看网络信息。

```
rlk@ubuntu:rlk_basic$ ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 10.238.170.56 netmask 255.255.254.0 broadcast 10.238.171.255
        inet6 fe80::8c4:60ff:feff:752b prefixlen 64 scopeid 0x20<link>
          ether 00:0c:29:21:aa:8c txqueuelen 1000 (Ethernet)
            RX packets 1 bytes 46 (46.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 38 bytes 4536 (4.5 KB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet6 fe80::421c:e3d4:6ba3:5fed prefixlen 64 scopeid 0x20<link>
        ether 00:0c:29:21:aa:8c txqueuelen 1000 (Ethernet)
          RX packets 14 bytes 1582 (1.5 KB)
          RX errors 0 dropped 0 overruns 0 frame 0
          TX packets 120 bytes 13388 (13.3 KB)
          TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
          loop txqueuelen 1000 (Local Loopback)
            RX packets 203 bytes 13761 (13.7 KB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 203 bytes 13761 (13.7 KB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

rlk@ubuntu:rlk_basic$
```

可以看到新增了一个名为“br0”的网络设备。

(2) 编译 QEMU+Debian 系统

按照实验 5 的方法来编一个基于 ARM32 的 QEMU+Debian 系统。

(3) 运行 QEMU+Debian 系统

修改 run_debian_arm32.sh 脚本。

奔跑吧 linux 社区出品

```
rlk@ubuntu:runninglinuxkernel_4.0$ git diff
diff --git a/run_debian_arm32.sh b/run_debian_arm32.sh
index 4a311824..5da0e69b 100755
--- a/run_debian_arm32.sh
+++ b/run_debian_arm32.sh
@@ -99,8 +99,8 @@ run_qemu_debian(){
        -append "crashkernel=128M root=/dev/vda rootfstype=ext4 rw"\n        -drive if=none,file=rootsfs_debian_arm32.ext4,id=hd0 \
        -device virtio-blk-device,drive=hd0 \
        -device user,id=mynet \
        -device virtio-net-device,netdev=mynet \
        -netdev user,id=net0,type=tap,script=/etc/qemu-ifup,downscript=no \
        -device virtio-net-device,netdev=net0 \
        --fsdev local,id=kmod dev,path=/kmodules,security_model=none \
        -device virtio-9p-device,fsdev=kmod_dev,mount_tag=kmod_mount \
$DBG
rlk@ubuntu:runninglinuxkernel_4.0$
```

运行 `run_debian_arm32.sh` 脚本。注意这里需要 sudo 权限。

```
| sudo ./run_debian_arm32.sh run
```

登录到 QEMU 虚拟机后，查看 ip 地址。

```
root@runninglinuxkernel:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 10.238.171.53 netmask 255.255.254.0 broadcast 10.238.171.255
                inet6 fe80::5054:ff:fe12:3456 prefixlen 64 scopeid 0x20<link>
                  ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)
                    RX packets 25 bytes 3393 (3.3 KiB)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 8 bytes 926 (926.0 B)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
                inet6 ::1 prefixlen 128 scopeid 0x10<host>
                  loop txqueuelen 0 (Local Loopback)
                    RX packets 0 bytes 0 (0.0 B)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 0 bytes 0 (0.0 B)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@runninglinuxkernel:~#
```

我们发现 QEMU 虚拟机分配的 IP 地址不再是：10.0.2.15，而是局域网中分配的地址：10.238.171.53。

这时候我们可以在优麒麟 Linux 中 ping 这个 ip 地址。

```
rlk@ubuntu:runninglinuxkernel_4.0$ ping 10.238.171.53
PING 10.238.171.53 (10.238.171.53) 56(84) bytes of data.
64 bytes from 10.238.171.53: icmp_seq=1 ttl=64 time=14.4 ms
64 bytes from 10.238.171.53: icmp_seq=2 ttl=64 time=4.49 ms
64 bytes from 10.238.171.53: icmp_seq=3 ttl=64 time=9.80 ms
64 bytes from 10.238.171.53: icmp_seq=4 ttl=64 time=1.06 ms
64 bytes from 10.238.171.53: icmp_seq=5 ttl=64 time=1.41 ms
64 bytes from 10.238.171.53: icmp_seq=6 ttl=64 time=1.31 ms
64 bytes from 10.238.171.53: icmp_seq=7 ttl=64 time=2.48 ms
^C
--- 10.238.171.53 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6011ms
rtt min/avg/max/mdev = 1.060/5.003/14.451/4.795 ms
rlk@ubuntu:runninglinuxkernel_4.0$
```

我们也可以在 Windows 主机上 ping 这个 ip 地址。

1.9 实验 9：配置 QEMU 虚拟机的桥接网络（新增）

```
C:\Users>ping 10.238.171.53

Pinging 10.238.171.53 with 32 bytes of data:
Reply from 10.238.171.53: bytes=32 time=2ms TTL=64
Reply from 10.238.171.53: bytes=32 time=2ms TTL=64
Reply from 10.238.171.53: bytes=32 time=1ms TTL=64
Reply from 10.238.171.53: bytes=32 time=1ms TTL=64

Ping statistics for 10.238.171.53:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 2ms, Average = 1ms

C:\Users>
```

(4) ssh 登录 QEMU+Debian 系统

我们可以在 QEMU+Debian 系统中安装 ssh 服务，然后在 Windows 中通过 ssh 软件登录到该系统中。

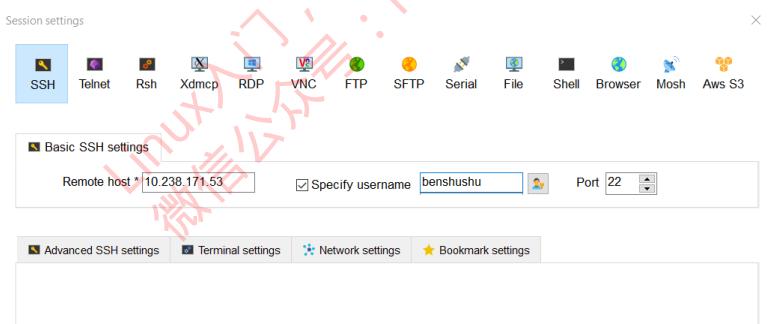
在 QEMU+Debian 系统中安装 ssh 服务。

```
#apt install openssh-server
```

关于 Windows 上的 ssh 软件，推荐使用 Mobaxterm。该软件有一个免费的版本。

<https://mobaxterm.mobatek.net/download.html>

在 Mobaxterm 软件界面中，点击“session”菜单，选择“New session”。如图所示，建立 ssh 连接。



```

? MobaXterm 10.5 ?
(SSH client, X-server and networking tools)

> SSH session to benshushu@10.238.171.53
? SSH compression : ✓
? SSH-browser : ✓
? X11-forwarding : ✓ (remote display is forwarded through SSH)
? DISPLAY : ✓ (automatically set on remote server)

> For more info, ctrl+click on help or visit our website

Linux runninglinuxkernel 4.0.0+ #1 SMP Wed Sep 4 21:00:22 PDT 2019 armv7l
Welcome Running Linux Kernel.
Created by Ben Shushu <runninglinuxkernel@126.com>

Buy linux kernel traning course:
https://shop115683645.taobao.com/

Wechat: runninglinuxkernel

Could not chdir to home directory /home/benshushu: No such file or directory
/usr/bin/xauth: error in locking authority file /home/benshushu/.Xauthority
$ 
$ 

```

如上图所示，通过 ssh 来登录 QEMU+Debian 系统。这样我们就可以得到了一个 ARM32 的 Debian 系统了，当然我们也可以得到一个 ARM64 的 Debian 系统。

4 补充材料

NAT 模式

NAT 模式，就是让虚拟系统借助 NAT(网络地址转换)功能，通过宿主机器所在的网络来访问因特网。使用 NAT 模式可以实现在虚拟系统里访问互联网。NAT 模式下的虚拟系统的 TCP/IP 配置信息是由 QEMU(NAT)虚拟网络的 DHCP 服务器提供的，无法进行手工修改，因此虚拟系统也就无法和本局域网中的其他真实主机进行通讯。采用 NAT 模式最大的优势是虚拟系统接入互联网非常简单，你不需要进行任何其他的配置，只需要宿主机器能访问互联网。

Bridge 模式

在这种模式下，VMWare 虚拟出来的操作系统就像是局域网中的一台独立的主机，它可以访问网内任何一台机器。

在桥接模式下，你需要手工为虚拟系统配置 IP 地址、子网掩码，而且还要和宿主机器处于同一网段，这样虚拟系统才能和宿主机器进行通信。同时，由于这个虚拟系统是局域网中的一个独立的主机系统，那么就可以手工配置它的 TCP/IP 配置信息，以实现通过局域网的网关或路由器访问互联网。

使用桥接模式的虚拟系统和宿主机器的关系，就像连接在同一个 Hub 上的两台电脑。想让它们相互通信，你就需要为虚拟系统配置 IP 地址和子网掩码，否则就无法通信。

1.10 实验 10 : 动手 DIY 一个 RISC-V 的 Debian 系统 (新增)

1.10 实验 10: 动手 DIY 一个 RISC-V 的 Debian 系统 (新增)

1 实验目的

通过本实验制作一个基于 RISC-V 的 Debian 系统。

注意: 本实验选做。学有余力的同学可以尝试做本实验。

2 实验要求

最近开源指令集 RISC-V 指令很火，国内外很多大公司都加入 RISC-V 阵营中。国内很多公司已经开始研制基于 RISC-V 的芯片了。但是基于 RISC-V 的开发板却很难买到，而且价格昂贵，给学习者带来巨大的困难。



不过，我们可以利用 QEMU 这个模拟器来运行 RISC-V 指令集甚至运行 Debian 系统。本实验动手制作一个 RISC-V 的 Debian 小系统。在 Linux 4.15 中，RISC-V 已经合并到 Linux 内核社区版本里。

由于本实验指导手册采用的内核版本是 4.0，不包含 RISC-V 的支持。若我们采用 backport 的方式，把 RISC-V 支持移植的 4.0 内核，工作量不小。因此，我们采用 Linux 5.0 的内核版本来制作一个 RISC-V 系统。

3 实验要求

读者可以参照实验 9 的步骤来制作一个 RISC-V 的 Debian 的根文件系统。
我们打算把这个实验留个读者独立完成。

Linux入门，看奔跑吧Linux内核
微信公众号：runninglinuxkernel

4 参考

我们为了这个实验做了一个 RSIC-V 的 git tree，供小伙伴们参考。这个 git tree 下载地址是：

| <https://github.com/figozhang/linux-5.0-kdump>

下载^①：

^① 本书配套的 vmware v4.0 镜像里不包含此 git tree，读者需要自行下载。

1.10 实验 10 : 动手 DIY 一个 RISC-V 的 Debian 系统 (新增)

```
| #git clone https://github.com/figozhang/linux-5.0-kdump.git  
|  
(1) 在 Ubuntu 中安装 riscv 的工具链①  
| #sudo apt install gcc-7-riscv64-linux-gnu gcc-riscv64-linux-gnu  
|  
(2) 编译内核  
| # ./run_debian_riscv.sh build_kernel  
|  
(3) 编译 Debian 根文件系统  
| #sudo ./run_debian_riscv.sh build_rootfs  
|  
注意这里需要 root 权限。  
|  
(4) 开始运行基于 RISCV 的 debian 系统了。  
| # ./run_debian_riscv.sh run
```

登录用户: root

密码: 123

第 2 章

^① 本书配套的 vmware v4.0 镜像里不包含 riscv 工具链, 读者需要自行下载。

Linux 内核基础知识

2.1 实验 1：GCC 编译

1. 实验目的

- 1) 熟悉 GCC 的编译过程，学会使用 ARM GCC 交叉工具链编译应用程序并放入 QEMU 上运行。
- 2) 学会写简单的 Makefile。

2. 实验详解

本实验通过一个简单的 C 语言程序代码演示 GCC 的编译过程。下面是一个简单的 test.c 的程序代码。

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

#define PAGE_SIZE 4096
#define MAX_SIZE 100*PAGE_SIZE

int main()
{
    char *buf = (char *)malloc(MAX_SIZE);
    memset(buf, 0, MAX_SIZE);
    printf("buffer address=0x%p\n", buf);
    free(buf);
    return 0;
}
```

1) 预处理。

GCC 的“-E”选项可以让编译器在预处理阶段就结束，选项“-o”可以指定输出的文件格式。

```
| arm-linux-gnueabi-gcc -E test.c -o test.i
```

预处理阶段会把 C 标准库的头文件中的代码包含到这段程序中。test.i 文件的内容如下所示。

```
extern void *malloc (size_t __size) __attribute__ ((__nothrow__, __leaf__))
__attribute__ ((__malloc__));
...
int main()
```

2.1 实验 1 : GCC 编译

```

char *buf = (char *)malloc(100*4096);

memset(buf, 0, 100*4096);

printf("buffer address=0x%p\n", buf);

free(buf);
return 0;
}

```

2) 编译。

编译阶段主要是对预处理好的.i 文件进行编译，并生成汇编代码。GCC 首先检查代码是否有语法错误等，然后把代码编译成汇编代码。我们这里使用“-S”选项来编译。

```
$ arm-linux-gnueabi-gcc -S test.i -o test.s
```

编译阶段生成的汇编代码如下。

```

.LC0:
.ascii  "buffer address=0x%p\012\000"
.text
.align  2
.global  main
.thumb
.thumb_func
.type   main, %function
main:
@ args = 0, pretend = 0, frame = 8
@ frame_needed = 1, uses_anonymous_args = 0
push   {r7, lr}
sub    sp, sp, #8
add    r7, sp, #0
mov    r0, #409600
bl    malloc
mov    r3, r0
str    r3, [r7, #4]
ldr    r3, [r7, #4]
mov    r2, r3
mov    r3, #409600
mov    r0, r2
mov    r1, #0
mov    r2, r3
bl    memset
movw  r3, #:lower16:.LC0
movt  r3, #:upper16:.LC0
mov    r0, r3
ldr    r1, [r7, #4]
bl    printf
ldr    r0, [r7, #4]
bl    free
mov    r3, #0
mov    r0, r3
add    r7, r7, #8
mov    sp, r7
pop   {r7, pc}
.size  main, .-main
.ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section .note.GNU-stack,"",%progbits

```

3) 汇编。

汇编阶段是将汇编文件转化成二进制文件，利用“-c”选项就可以生成二进制文

件。

```
| $ arm-linux-gnueabi-gcc -c test.s -o test.o
```

4) 链接。

链接阶段会对编译好的二进制文件进行链接，这里会默认链接 C 语言标准库 (libc)。我们的代码里调用的 malloc()、memset() 以及 printf() 等函数都由 C 语言标准库提供，链接过程会把程序的目标文件和所需的库文件链接起来，最终生成可执行文件。

Linux 的库文件分成两大类：一类是动态链接库（通常以.so 结尾），另一类是静态链接库（通常以.a 结尾）。在默认情况下，GCC 在链接时优先使用动态链接库，只有当动态链接库不存在时才使用静态链接库。下面使用 “--static” 来让 test 程序静态链接 C 语言标准库，原因是交叉工具链使用的 libc 的动态库和 QEMU 中使用的库可能不一样。如果使用动态链接，可能导致运行报错。

```
| $ arm-linux-gnueabi-gcc test.o -o test --static
```

以 ARM GCC 交叉工具链为例，C 函数库动态库的目录在 /usr/arm-linux-gnueabi/lib 里，最终的库文件是 libc-2.23.so 文件。

```
$ ls -l /usr/arm-linux-gnueabi/lib/libc.so.6
lrwxrwxrwx 1 root root 12 Apr 16 2016 /usr/arm-linux-gnueabi/lib/libc.so.6
-> libc-2.23.so
```

C 语言标准库的静态库地址如下。

```
| $ ls -l /usr/arm-linux-gnueabi/lib/libc.a
-rw-r----- 1 root root 3175586 Apr 16 2016 /usr/arm-linux-gnueabi/lib/libc.a
```

5) 放到 QEMU 上运行。

把 test 程序放入 runninglinuxkernel_4.0/kmodules 目录里，启动 QEMU 并运行 test 程序。

```
| / # ./mnt/test
buffer address=0x0xb6f73008
/ #
```

6) 编写一个简单的 Makefile 文件来编译。

```
cc = arm-linux-gnueabi-gcc
prom = test
obj = test.o
CFLAGS = -static

$(prom): $(obj)
    $(cc) -o $(prom) $(obj) $(CFLAGS)

%.o: %.c
    $(cc) -c $< -o $@

clean:
    rm -rf $(obj) $(prom)
```

2.2 实验 2：内核链表

2.2 实验 2：内核链表

1. 实验目的

- 1) 学会和研究 Linux 内核提供的链表机制。
- 2) 编写一个应用程序，利用内核提供的链表机制创建一个链表，把 100 个数字添加到链表中，循环该链表输出所有成员的数值。

2. 实验详解

Linux 内核链表提供的函数接口定义在 `include/linux/list.h` 文件中。本实验把这些接口函数移植到用户空间中，并使用它们完成链表的操作。

2.3 实验 3：红黑树

1. 实验目的

- 1) 学习和研究 Linux 内核提供的红黑树机制。
- 2) 编写一个应用程序，利用内核提供的红黑树机制创建一棵树，把 10 000 个随机数添加到红黑树中。
- 3) 实现一个查找函数，快速在这棵红黑树中查找到相应的数字。

2. 实验详解

Linux 内核提供的红黑树机制实现在 `lib/rbtree.c` 和 `include/linux/rbtree.h` 文件中。本实验要求把 Linux 内核实现的红黑树机制移植到用户空间，并且实现 10 000 个随机数的插入和查找功能。

2.4 实验 4：使用 Vim 工具

1. 实验目的

熟悉 Vim 工具的基本操作。

2. 实验详解

Vim 的操作需要一定的练习量才能达到熟练，读者可以使用优麒麟 Linux 18.04 系统中 Vim 程序进行编辑代码的练习。

2.5 实验 5：把 Vim 打造成一个强大的 IDE 编辑工具

1. 实验目的

通过配置把 Vim 打造成一个和 Source Insight 相媲美的 IDE 工具。

2. 实验步骤

Vim 工具可以支持很多个性化的特性，并使用插件来完成代码浏览和编辑的功能。使用过 Source Insight 的读者也许会对如下功能赞叹有加。

- 自动列出一个文件的函数和变量的列表。
- 查找函数和变量的定义。
- 查找哪些函数调用了该函数和变量。
- 高亮显示。
- 自动补全。

这些功能在 Vim 里都可以实现，而且比 Source Insight 高效和好用。本实验将带领读者着手打造一个属于自己的 IDE 编辑工具。

在打造之前先安装 git 工具。

```
$ sudo apt-get install git
```

(1) 插件管理工具 Vundle

Vim 支持很多插件，早期需要到每个插件网站上下载后复制到 home 主目录的.vim 目录中才能使用。现在 Vim 社区有很多个插件管理工具，其中 Vundle 就是很出色的一个，它可以在.vimrc 中跟踪、管理和自动更新插件等。

安装 Vundle 需要使用 git 工具，通过如下命令来下载 Vundle 工具。

```
$ git clone https://github.com/VundleVim/Vundle.vim.git  
~/vim/bundle/Vundle.vim
```

接下来需要在 home 主目录下面的.vimrc 配置文件中配置 Vundle。

<.vimrc文件中添加如下配置>

```
" Vundle manage
set nocompatible           " be iMproved, required
filetype off                " required

" set the runtime path to include Vundle and initialize
set rtp+=~/vim/bundle/Vundle.vim
call vundle#begin()

" let Vundle manage Vundle, required
Plugin 'VundleVim/Vundle.vim'

" All of your Plugins must be added before the following line
call vundle#end()           " required
filetype plugin indent on    " required
```

2.5 实验 5：把 Vim 打造成一个强大的 IDE 编辑工具

只需要在该配置文件中添加“Plugin xxx”，即安装名为“xxx”的插件。

接下来就是在线安装插件。启动 Vim，然后运行命令“:PluginInstall”，就会从网络上下载插件并安装。

(2) ctags 工具

ctags 工具全称 Generate tag files for source code。它扫描指定的源文件，找出其中包含的语法元素，并把找到的相关内容记录下来，这样在代码浏览和查找时就可以利用这些记录实现查找和跳转功能。ctags 工具已经被集成到各大 Linux 发行版中。在优麒麟 Linux 中使用如下命令安装 ctags 工具。

```
| $ sudo apt-get install ctags
```

在使用 ctags 之前需要手工生成索引文件。

```
| $ ctags -R . //递归扫描源代码根目录和所有子目录的文件并生成索引文件
```

上述命令会在当前目录下面生成一个 tags 文件。启动 Vim 之后需要加载这个 tags 文件，可以通过如下命令实现这个加载动作。

```
| :set tags=tags
```

ctags 常用的快捷键如表 2.7 所示。

表 2.7 常用的快捷键

快 捷 键	用 法
Ctrl +]	跳转到光标处的函数或者变量的定义所在的地方
Ctrl + T	返回到跳转之前的地方

(3) cscope 工具

刚才介绍的 ctags 工具可以跳转到标签定义的地方，但是如果想查找函数在哪里被调用过或者标签在哪些地方出现过，那么 ctags 就无能为力了。cscope 工具可以实现上述功能，这也是 Source Insight 强大的功能之一。

cscope 最早由贝尔实验室开发，后来由 SCO 公司以 BSD 协议公开发布。我们可以在优麒麟 Linux 发行版中安装它。

```
| $ sudo apt-get install cscope
```

在使用 cscope 之前需要对源代码生成索引库，可以使用如下命令来实现。

```
| $ cscope -Rbq
```

上述命令会生成 3 个文件：cscope.cout、cscope.in.out 和 cscope.po.out。其中 cscope.out 是基本符合的索引，后面两个文件是使用“-q”选项生成的，用于加快 cscope 索引的速度。

在 Vim 中使用 cscope 非常简单，首先调用“cscope add”命令添加一个 cscope 数据库，然后调用“cscope find”命令进行查找。Vim 支持 8 种 cscope 的查询功能。

- s: 查找 C 语言符号，即查找函数名、宏、枚举值等出现的地方。
- g: 查找函数、宏、枚举等定义的位置，类似 ctags 所提供的功能。
- d: 查找本函数调用的函数。
- c: 查找调用本函数的函数。
- t: 查找指定的字符串。
- e: 查找 egrep 模式，相当于 egrep 功能，但查找速度快多了。
- f: 查找并打开文件，类似 Vim 的 find 功能。
- i: 查找包含本文件的文件。

为了方便使用，我们可以在.vimrc 配置文件中添加如下快捷键。

```
"-----
" cscope:建立数据库: cscope -Rbq; F5 查找c符号; F6 查找字符串; F7 查找函数定义;
F8 查找函数谁调用了
"-----
if has("cscope")
  set csprg=/usr/bin/cscope
  set csto=1
  set cst
  set nocsverb
  " add any database in current directory
  if filereadable("cscope.out")
    cs add cscope.out
  endif
  set csverb
endif

:set cscopequickfix=s-,c-,d-,i-,t-,e-
:nmap <C-_>s :cs find s <C-R>=expand("<cword>")<CR><CR>
" F5 查找c符号; F6 查找字符串; F7 查找函数谁调用了
nmap <silent> <F5> :cs find s <C-R>=expand("<cword>")<CR><CR>
nmap <silent> <F6> :cs find t <C-R>=expand("<cword>")<CR><CR>
nmap <silent> <F7> :cs find c <C-R>=expand("<cword>")<CR><CR>
```

上述定义的快捷键如下。

- F5: 查找 C 语言符号，即查找函数名、宏、枚举值等出现的地方。
- F6: 查找指定的字符串。
- F7: 查找调用本函数的函数。

(4) Tagbar 插件

Tagbar 插件可以把源代码文件生成一个大纲，包括类、方法、变量以及函数名等，可以选中并快速跳转到目标位置。

安装 Tagbar 插件，在.vimrc 文件中添加：

```
Plugin 'majutsushi/tagbar' " Tag bar"
```

然后重启 Vim，输入并运行命令“:PluginInstall”完成安装。

配置 Tagbar 插件，可以在.vimrc 文件中添加如下配置。

2.5 实验 5：把 Vim 打造成一个强大的 IDE 编辑工具

```
" Tagbar
let g:tagbar_width=25
autocmd BufReadPost *.cpp,*.c,*.h,*.cc,*.cxx call tagbar#autoopen()
```

上述配置可让打开常见的源代码文件时会自动打开 Tagbar 插件。

(5) 文件浏览插件 NerdTree

NerdTree 插件可以显示树形目录。

安装 NerdTree 插件，在.vimrc 文件中添加：

```
Plugin 'scrooloose/nerdtree'
```

然后重启 Vim，输入并运行命令 “:PluginInstall” 完成安装。

配置 NerdTree 插件。

```
" NetRdTTree
autocmd StdinReadPre * let s:std_in=1
autocmd VimEnter * if argc() == 0 && !exists("s:std_in") | NERDTree | endif
let NERDTreeWinSize=15
let NERDTreeShowLineNumbers=1
let NERDTreeAutoCenter=1
let NERDTreeShowBookmarks=1
```

(6) 动态语法检测工具

动态语法检测工具可以在编写代码的过程中检测出语法错误，不用等到编译或者运行，这个工具对编写代码者非常有用。本实验安装的是被称为 ALE(Asynchronization Lint Engine) 的一款实时代码检测工具。ALE 工具在发现有错误的地方会实时提醒，在 Vim 的侧边会标注哪一行有错误，光标移动到这一行时下面会显示错误的原因。ALE 工具支持多种语言的代码分析器，比如 C 语言可以支持 gcc、clang 等。

安装 ALE 工具，在.vimrc 文件中添加：

```
Plugin 'w0rp/ale'
```

然后重启 Vim，输入并运行命令 “:PluginInstall” 完成安装。这个过程需要从网络上下载代码。

插件安装完成之后，做一些简单的配置，增加如下配置到.vimrc 文件中。

```
let g:ale_sign_column_always = 1
let g:ale_sign_error = 'X'
let g:ale_sign_warning = 'w'
let g:ale_statusline_format = ['X %d', '$ %d', '✓ OK']
let g:ale_echo_msg_format = '[%linter%] %code: %s'
let g:ale_lint_on_text_changed = 'normal'
let g:ale_lint_on_insert_leave = 1
let g:ale_c_gcc_options = '-Wall -O2 -std=c99'
let g:ale_cpp_gcc_options = '-Wall -O2 -std=c++14'
let g:ale_c_cppcheck_options = ''
let g:ale_cpp_cppcheck_options = ''
```

我们用 ALE 工具编写一个简单的 C 程序，如图 2.4 所示。

Vim 左边会显示错误或者警告的提示，其中 “w” 表示警告，“x” 表示错误。图

2.4 所示的第 3 行出现了一个警告，这是 gcc 编译器发现变量 i 定义了但并没有使用。

(7) 自动补全插件 YouCompleteMe

代码补全功能在 Vim 发展历史中是一个比较弱的功能，因此一直被使用 Source Insight 的人诟病。早些年出现的自动补全插件如 AutoComplPop、Omnicppcomplete、Neocomplcache 等在效率上低得惊人，特别是把整个 Linux 内核代码添加到工程时，要使用这些代码补全功能，每次都要等待 1~2 分钟的时间，简直让人抓狂。



图2.4 ALE工具

YouCompleteMe 插件是最近几年才出现的新插件，它利用 clang 为 C/C++ 代码提供代码提示和补全功能。借助 clang 的强大功能，YouCompleteMe 的补全效率和准确性极高，可以和 Source Insight 一比高下。因此，Linux 开发人员在 Vim 上配备了 YouCompleteMe 插件之后完全可以抛弃 Source Insight。

在安装 YouCompleteMe 插件之前，需要保证 Vim 的版本必须高于 7.4.1578，并且支持 Python 2 或者 Python 3。优麒麟 Linux 16.04 的版本中的 Vim 满足这个要求，使用其他发行版的读者可以用如下命令来检查。

```
| $ vim -version
```

安装 YouCompleteMe 插件，可在.vimrc 文件中添加：

```
| Plugin 'Valloric/YouCompleteMe'
```

然后重启 Vim，输入并运行命令 “:PluginInstall” 完成安装。这个过程从网络中下载代码，需要等待一段时间。

插件安装完成之后，需要重新编译它，所以在编译之前需要保证已经安装如下软件包。

```
| $ sudo apt-get install build-essential cmake python-dev python3-dev
```

接下来进入 YouCompleteMe 插件代码进行编译。

```
| $ cd ~/.vim/bundle/YouCompleteMe
```

2.5 实验 5：把 Vim 打造成一个强大的 IDE 编辑工具

```
$ ./install.py --clang-completer
```

--clang-completer 表示对 C/C++的支持。

编译完成后，还需要做一些配置工作，把
~/.vim/bundle/YouCompleteMe/third_party/ycmd/examples/.ycm_extra_conf.py 这个文件
复制到 ~/.vim 目录下面。

```
$ cp  
~/.vim/bundle/YouCompleteMe/third_party/ycmd/examples/.ycm_extra_conf.py  
~/.vim
```

在.vimrc 配置文件中还需要添加如下配置。

```
let g:ycm_server_python_interpreter='/usr/bin/python'  
let g:ycm_global_ycm_extra_conf='~/.vim/.ycm_extra_conf.py'
```

这样就完成了 YouCompleteMe 插件的安装和配置。

下面做一个简单测试。首先启动 Vim，输入“#include <stdio>”检查是否会出现
提示补全，如图 2.5 所示。



图2.5 代码补全测试

(8) 自动索引

在旧版本的 Vim 中是不支持异步模式的，因此每次写一部分代码需要手动运行
ctags 命令来生成索引，这是 Vim 的一大痛点。这个问题在 Vim 8 之后得到了改善。
下面推荐一个可以异步生成 tags 索引的插件，这个插件称为 vim-gutentags。

安装 vim-gutentags 插件。

```
Plugin 'ludovicchabant/vim-gutentags'
```

重启 Vim，输入命令“.PluginInstall”完成安装，这个过程需要从网络中下载代码。
对插件进行一些简单配置，将以下内容添加到.vimrc 文件中。

```
" 搜索工程目录的标志，碰到这些文件/目录名就停止向上一级目录递归  
let g:gutentags_project_root = ['.root', '.svn', '.git', '.hg', '.project']

" 配置 ctags 的参数  
let g:gutentags_ctags_extra_args = ['--fields=+nizazS', '--extra=+q']  
let g:gutentags_ctags_extra_args += ['--c++-kinds=+px']  
let g:gutentags_ctags_extra_args += ['--c-kinds=+px']
```

当我们修改了一个文件时，vim-gutentags 会在后台默默帮助我们更新 tags 数据索
引库。

(9) vimrc 的其他一些配置

vimrc 还有一些其他常用的配置，如显示行号等。

```
set nu!          " 显示行号

syntax enable
syntax on
colorscheme desert

:set autowrite " 自动保存
```

(10) 使用 Vim 来阅读 Linux 内核源代码

我们已经把 Vim 打造成一个媲美 Source Insight 的 IDE 工具了。下面介绍如何阅读 Linux 内核源代码。

下载 Linux 内核官方源代码或者 runninglinuxkernel 的源代码。

```
git clone https://github.com/figozhang/runninglinuxkernel_4.0.git
```

Linux 内核已经支持 ctags 和 cscope 来生成索引文件，而且会根据编译的 config 文件选择需要扫描的文件。我们使用 make 命令来生成 ctags 和 cscope，下面以 ARM vexpress 平台为例进行介绍。

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make tags cscope TAGS //生成tags,cscope, TAGS等索引文件
```

启动 Vim，通过“:e mm/memory.c”命令打开 memory.c 源文件，然后在 do_anonymous_page() 函数即在第 2563 行上输入“vma->”，会发现 Vim 自动出现了 struct vm_area_struct 数据结构的成员供你选择，而且速度快得惊人，如图 2.6 所示。

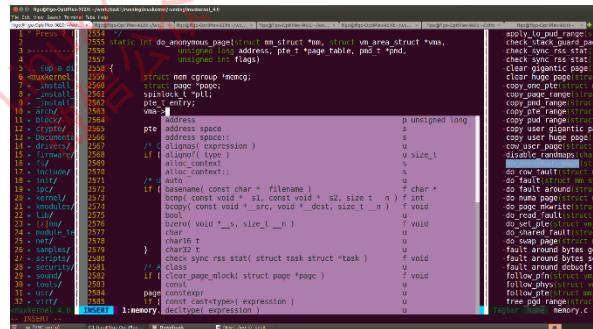


图2.6 在Linux内核代码中尝试代码补全

另外，我们在 do_anonymous_page() 函数的第 2605 行的 page_add_new_anon_rmap() 上按 F7 快捷键，会发现很快查找到了 Linux 内核中所有调用该函数的地方，如图 2.7 所示。

2.6 实验 6：建立一个 git 本地仓库

The screenshot shows a terminal session with multiple windows and tabs. The main window displays the search results for the function `page_add_new_anon_rmap()`. The results are shown in several tabs, each containing different parts of the kernel source code. The tabs include:

- `1:memory.c`
- `5:uprobes.c`
- `6:huge memory.c`
- `6:huge memory.c 10:` (highlighted tab)
- `1:mm.huge_memory.c 10:`
- `3:mm.huge_memory.c 1037:`
- `4:mm.huge_memory.c 1171:`
- `5:mm.huge_memory.c 2519:`
- `6:mm.memory.c 2181:`
- `7:mm/memory.c 2467:`
- `8:mm/memory.c 2605:`
- `9:mm/memory.c 2693:`
- `10:mm_swapsfile.c 1127:`

The search results are presented in a standard terminal format with line numbers and file paths. The highlighted tab shows the definition of the function in `huge memory.c`.

图2.7 查找哪些函数调用了`page_add_new_anon_rmap()`

2.6 实验 6：建立一个 git 本地仓库

1. 实验目的

学会如何快速创建一个 git 本地仓库，并将其运用到实际工作中。

2. 实验步骤

通常实际项目中会使用一台独立的机器作为 git 服务器，然后在 git 服务器中建立一个远程的仓库，这样项目中所有的人都可以通过局域网来访问这个 git 服务器。当然，我们在本实验中可以使用同一台机器来模拟这个 git 服务器。

(1) git 服务器端的操作

首先需要在服务器端建立一个目录，然后初始化这个 git 仓库。假设我们在 `"/opt/git/"` 目录下面来创建。

```
$ cd /opt/git/
$ mkdir test.git
$ cd test.git/
$ git --bare init
Initialized empty Git repository in /opt/git/test.git/
```

通过 `git --bare init` 命令创建了一个空的远程仓库。

(2) 客户端的操作

打开另外一个终端，然后在本地工作目录中编辑代码，比如在 `home` 目录下。

```
$ cd /home/rnk/
$ mkdir test
```

编辑一个 test.c 文件，添加简单的打印“hello world”的语句。

```
| $ vim test.c
```

初始化本地的 git 仓库。

```
| $ git init
| Initialized empty Git repository in /home/r1k/test/.git/
```

查看当前工作区的状态。

```
| $ git status
| On branch master
|
| Initial commit
|
| Untracked files:
|   (use "git add <file>..." to include in what will be committed)
|
|     test.c
|
| nothing added to commit but untracked files present (use "git add" to track)
```

可以看到工作区里有一个 test.c 文件，然后通过 git add 命令来添加 test.c 文件到缓存区中。

```
| $ git add test.c
```

用 git commit 生成一个新的提交。

```
| $ git commit -s
|
| test: add init code for xxx project
|
| Signed-off-by: Ben Shushu <runninglinuxkernel@126.com>
|
| # Please enter the commit message for your changes. Lines starting
| # with '#' will be ignored, and an empty message aborts the commit.
| # On branch master
| #
| # Initial commit
| #
| # Changes to be committed:
| #   new file: test.c
| #
```

在上述代码中添加对这个新提交的描述，保存之后自动生成一个新的提交。

```
| $ git commit -s
[master (root-commit) ea92c29] test: add init code for xxx project
1 file changed, 8 insertions(+)
create mode 100644 test.c
```

接下来需要把本地的 git 仓库推送到远程仓库中。

首先需要通过 git remote add 命令添加刚才远程仓库的地址。

```
| $ git remote add origin ssh://ben@192.168.0.1:/opt/git/test.git
```

其中“192.168.0.1”是服务器端的 IP 地址，“ben”是服务器端的登录名。最后用

2.7 实验 7：解决合并分支冲突

git push 命令来推送。

```
$ git push origin master
figo@192.168.0.1's password:
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://figo@10.239.76.39:/opt/git/test.git
 * [new branch]      master -> master
```

(3) 复制远端仓库

这时我们就可以在局域网内通过 git clone 复制这个远程仓库到本地了。

```
$ git clone ssh://ben@192.168.0.1:/opt/git/test.git
Cloning into 'test'...
ben@192.168.0.1's password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
$ cd test/
$ git log
commit ea92c29d88ba9e58960ec13911616f2c2068b3e6
Author: Ben Shushu <runninglinuxkernel@126.com>
Date:   Mon Apr 16 23:13:32 2018 +0800

    test: add init code for xxx project

Signed-off-by: Ben Shushu <runninglinuxkernel@126.com>
```

2.7 实验 7：解决合并分支冲突

1. 实验目的

了解和学会如何解决合并分支时遇到的冲突。

2. 实验详解

首先创建分支合并冲突的环境，可以通过如下步骤实现。

1) 创建一个本地分支。创建一个主分支和 dev 分支。

```
$ git init
```

2) 在主分支上新建一个 test.c 的文件。输入简单的“hello world”程序，然后创建一个提交。

```
#include <stdio.h>

int main()
{
    int i;
```

```

    printf("hello word\n");
}

}

```

3) 基于主分支创建一个新的 dev 分支。

```
$ git checkout -b dev
```

4) 在 dev 分支上做如下改动，并生成一个提交。

```

diff --git a/test.c b/test.c
index 39ee70f..ed431cc 100644
--- a/test.c
+++ b/test.c
@@ -2,7 +2,10 @@

int main()
{
-    int i;
+    int i = 10;
+    char *buf;
+
+    buf = malloc(100);

    printf("hello word\n");

```

5) 切换到主分支，然后继续修改 test.c 文件，并生成一个提交。

```

diff --git a/test.c b/test.c
index 39ee70f..e0ccfb9 100644
--- a/test.c
+++ b/test.c
@@ -3,6 +3,7 @@
int main()
{
    int i;
+    int j = 5;

    printf("hello word\n");

```

6) 这样我们的实验环境就搭建好了。在这个 git 仓库里有两个分支，一个是主分支，另一个是 dev 分支，它们同时修改了相同的文件，如图 2.8 所示。

Rev list	Author	Author Date
Graph	Short Log	
	Nothing to commit	
● ↗	master add int j	Ben Shushu<runninglinuxk... 7/8/18 5:20 PM
	dev malloc 100 bytes	Ben Shushu<runninglinuxk... 7/8/18 5:19 PM
	hello world	Ben Shushu<runninglinuxk... 7/8/18 5:17 PM

图2.8 主分支和git分支

7) 使用如下命令把 dev 分支上的提交合并到主分支上，如果遇到了冲突，请解决。

```

$ git branch //先确认当前分支是master分支
$ git merge dev //把dev分支合并到master分支

```

2.8 实验 8：利用 git 来管理 Linux 内核开发

下面简单介绍一下如何解决合并分支冲突。当合并分支遇到冲突时会显示如下提示，其中明确告诉我们是在合并哪个文件时发生了冲突。

```
$ git merge dev
Auto-merging test.c
CONFLICT (content): Merge conflict in test.c
Automatic merge failed; fix conflicts and then commit the result.
```

接下来要做的工作就是手工修改冲突了。打开 `test.c` 文件，会看到“<<<<<<”和“>>>>>”符号包括的区域就是发生冲突的地方。至于如何修改冲突，git 工具是没有办法做判断的，只能读者自己判断，前提条件是要对代码有深刻的理解。

```
#include <stdio.h>

int main()
{
<<<<< HEAD
    int i;
    int j = 5;
=====
    int i = 10;
    char *buf;

    buf = malloc(100);
>>>>> dev

    printf("hello word\n");
    return 0;
}
```

冲突修改完成之后，可以通过 `git add` 命令添加到 git 仓库中。

```
$ git add test.c
```

然后使用 `git merge --continue` 命令继续合并工作，直到合并完成为止。

```
$ git merge --continue
[master 9ad3b85] Merge branch 'dev'
```

读者可以重复该实验步骤，重建一个本地 git 仓库，使用变基命令合并 `dev` 分支到主分支上，遇到冲突并尝试解决。

2.8 实验 8：利用 git 来管理 Linux 内核开发

1. 实验目的

本实验通过模拟一个项目的实际操作来演示如何利用 git 进行 Linux 内核开发和管理。该项目的需求如下。

- 1) 该项目需要基于 Linux 4.0 内核进行二次开发。
- 2) 在本地建立一个名为“ben-linux-test”的项目，上传的内容要包含 Linux-4.0 中所有的 commit。

注意：本实验以及实验 9 需要访问网络来下载 Linux 内核的 git tree。

给教师的一点小提示：

- 可以在一台机器（作为 git 服务器）上提前下载好 Linux 内核的 git tree。
然后，做实验的同学可以通过局域网来访问这个机器。

git 服务器：

```
$ cd /opt/git/  
$ sudo chmod 777 -R /opt/git/  
$ git clone  
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

假设这个 git 服务器：

ip 地址为：192.168.0.10
用户名为：rlk
密码：123

那么做实验的同学们就可以通过如下命令下载 git tree。

```
$ git clone rlk@192.168.0.10:/opt/git/linux
```

- 由于下载整个 Linux 内核的 git tree 需要下载好几个 GB 的内容，本实验可以采用其他开源项目来替代 Linux 内核，比如 DPDK。

```
#git clone http://dpdk.org/git/dpdk
```

2. 实验步骤

- 参考实验 6，在本地建立一个名为“ben-linux-test”的空仓库。
- 下载 Linux 官方仓库代码。

接下来的工作就是在这个本地的 git 仓库里下载官方 Linux 4.0 的代码，那应该怎么做呢？首先我们需要添加 Linux 官方的 git 仓库。这里可以使用“git remote add”命令来添加一个远程仓库地址，如下所示。

```
$ git remote add linux  
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

git remote -v 命令把 Linux 官方的远程仓库添加到本地了，并且起了一个别名——linux。

```
$ git remote -v
```

2.8 实验 8 : 利用 git 来管理 Linux 内核开发

```
linux  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
(fetch)
linux  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
(push)
origin  https://github.com/figozhang/ben-linux-test.git (fetch)
origin  https://github.com/figozhang/ben-linux-test.git (push)
```

git fetch 命令可以把新添加的远程仓库代码下载到本地。

```
$ git fetch linux
remote: Counting objects: 6000860, done.
remote: Compressing objects: 100% (912432/912432), done.
Receiving objects: 1% (76970/6000860), 37.25 MiB | 694.00 KiB/s
```

下载完成后，用 git branch -a 命令查看分支情况。

```
$ git branch -a
* master
  remotes/linux/master
  remotes/origin/master
```

看到远程仓库有两个：一个是我们刚才在本地创建的仓库（remotes/origin/master），另一个是 Linux 内核官方的远程仓库（remotes/linux/master）。

3) 创建 Linux 4.0 分支。

接下来需要把官方仓库中 Linux 4.0 标签的所有提交添加到本地的主分支上。首先需要从 remotes/linux/master 分支上检查一个名为 linux-4.0 的本地分支。

```
$ git checkout -b linux-4.0 linux/master
Checking out files: 100% (61345/61345), done.
Branch linux-4.0 set up to track remote branch master from linux.
Switched to a new branch 'linux-4.0'.

$ git branch -a
* linux-4.0
  master
  remotes/linux/master
  remotes/origin/master
```

因为项目需要在 Linux 4.0 上工作，所以把该分支重新放到 Linux 4.0 的标签上，这时可以使用 git reset 命令。

```
$ git reset v4.0 --hard
Checking out files: 100% (61074/61074), done.
HEAD is now at 39a8804 Linux 4.0
```

这样本地 linux-4.0 分支就是真正基于 Linux 4.0 的内核，并且包含了 Linux 4.0 上所有的提交的信息。

4) 合并本地修改到 Linux 4.0 上。

接下来的工作就是把本地 linux-4.0 的分支上的提交都合并到本地的主分支上。

首先需要切换到本地的主分支上。

```
$ git checkout master
```

然后使用 git merge 命令把本地 linux-4.0 分支上所有的提交都合并到主分支上。

```
$ git merge linux-4.0 --allow-unrelated-histories
```

这个合并会生成一个叫作 merge branch 的提交，如下所示。

```
Merge branch 'linux-4.0'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

最后，这个本地的主分支的提交就变成这样：

```
$ git log --oneline
c67cf17 Merge branch 'linux-4.0'
f85279c first commit
39a8804 Linux 4.0
6a23b45 Merge branch 'for-linus' of
git://git.kernel.org/pub/scm/linux/kernel/git/viro/vfs
54d8ccc Merge branch 'fixes' of
git://git.kernel.org/pub/scm/linux/kernel/git/evalenti/linux-soc-thermal
56fd85b Merge tag 'asoc-fix-v4.0-rc7' of
git://git.kernel.org/pub/scm/linux/kernel/git/broonie/sound
14f0413c ASoC: pcm512x: Remove hardcoding of pll-lock to GPIO4
```

这样本地主分支上就包含了所有 Linux 4.0 内核的 git log 信息。最后一步只需要把这个主分支推送到远程仓库即可。

```
$ git push origin master
```

现在这个仓库的主分支已经包含了 Linux 4.0 内核的所有提交了，在这个基础上可以建立属于该项目的自己的分支，比如 dev-linux-4.0 分支、feature_a_v0 分支等。

```
$ git branch -a
  dev-linux-4.0
* feature_a_v0
  master
  remotes/linux/master
  remotes/origin/master
```

2.9 实验 9：利用 git 来管理项目代码

1. 实验目的

1) 在 Linux 4.0 上做开发。为了简化开发，我们假设只需要修改 Linux 4.0 根目录下面的 Makefile，如下所示。

```
VERSION = 4
PATCHLEVEL = 0
SUBLEVEL = 0
EXTRAVERSION =
NAME = Hurr durr I'ma sheep //修改这里，改成 benshushu
```

2) 把修改推送到本地仓库上。

2.9 实验 9：利用 git 来管理项目代码

3) 过了几个月，这个项目需要变基（rebase）到 Linux 4.15 的内核，并且把之前做的工作也变基到 Linux 4.15 内核，并且更新到本地仓库上。如果变基时遇到冲突，需要修复。

4) 在这个实验里，会学习到如何合并一个分支以及如何变基到最新的主分支上。

5) 在合并分支和变基分支的过程中，可能会遇到冲突，在本实验中可以学会如何修复冲突。

2. 实验步骤

在实际项目开发过程中，分支的管理是很重要的。以现在这个项目为例，项目开始时，我们会选择一个内核版本进行开发，比如选择 Linux 4.0 内核。等到项目开发到一定的阶段，比如 Beta 阶段，其需求发生变化。这时需要基于最新的内核进行开发，如基于 Linux 4.15。那么就要把开发工作变基到 Linux 4.15 上了。这种情形在实际开源项目中是很常见的。

因此，分支管理显得很重要。master 分支通常是用来与开源项目同步的，dev 分支是我们平常开发用的主分支。另外，每个开发人员在本地可以建立属于自己的分支，如 feature_a_v0 分支，表示开发者甲在本地创建的用来开发 feature a 的分支，版本是 v0。

```
$ git branch -a
* dev-linux-4.0
  feature_a_v0
  master
  remotes/linux/master
  remotes/origin/master
  remotes/origin/dev-linux-4.0
```

(1) 把开发工作推送到 dev-linux-4.0 分支

下面就是基于 dev-linux-4.0 分支进行工作了，比如这里实验中要求修改 Makefile，然后生成一个提交并且将其推送到 dev-linux-4.0 分支上。

首先修改 Makefile。

修改的内容如下：

```
diff --git a/Makefile b/Makefile
index fbd43bf..2c48222 100644
--- a/Makefile
+++ b/Makefile
@@ -2,7 +2,7 @@ VERSION = 4
 PATCHLEVEL = 0
 SUBLEVEL = 0
 EXTRAVERSION =
-NAME = Hurr durr I'ma sheep
+NAME = benshushu

 # *DOCUMENTATION*
 # To see a list of typical targets execute "make help"
@@ -1598,3 +1598,5 @@ FORCE:
 # Declare the contents of the .PHONY variable as phony. We keep that
 # information in a variable so we can use it in if_changed and friends.
 .PHONY: $(PHONY)
```

```
+  
+ #demo for rebase by benshush //在最后一行添加, 为了将来变基制造冲突
```

生成一个提交。

```
$ git add Makefile  
$ git commit -s  
  
demo: modify Makefile  
  
modify Makefile for demo  
  
v1: do it base on linux-4.0
```

把这个修改推送到远程仓库。

```
$ git push origin dev-linux-4.0  
Counting objects: 3, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 341 bytes | 0 bytes/s, done.  
Total 3 (delta 2), reused 0 (delta 0)  
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.  
remote: Checking connectivity: 3, done.  
c67cf17..f35ab68 dev-linux-4.0 -> dev-linux-4.0
```

(2) 新建 dev-linux-4.15 分支

首先从远程仓库（remotes/linux/master）分支上新建一个名为 linux-4.15-org 的分支。

```
$ git checkout -b linux-4.15-org linux/master
```

然后把这个 linux-4.15-org 分支重新放到 v4.15 的标签上。

```
$ git reset v4.15 --hard  
Checking out files: 100% (21363/21363), done.  
HEAD is now at d8a5b80 Linux 4.15
```

接着切换到主分支。

```
$ git checkout master  
Checking out files: 100% (57663/57663), done.  
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.
```

然后把 linux-4.15-org 分支上所有的提交都合并到主分支上。

```
figo@figo:~ben-linux-test$ git merge linux-4.15-org
```

合并完成之后，查看主分支的日志信息，如下所示。

```
figo@figo ~ben-linux-test$ git log --oneline  
749d619 Merge branch 'linux-4.15-org'  
c67cf17 Merge branch 'linux-4.0'  
f85279c first commit  
d8a5b80 Linux 4.15
```

最后，把主分支的更新推送到远程仓库，这样我们的远程仓库的主分支就是基于 Linux 4.15 内核了。

2.9 实验 9：利用 git 来管理项目代码

```
| figo@figo:~ben-linux-test$ git push origin master
```

(3) 变基到 Linux 4.15 上

首先基于 dev-linux-4.0 分支创建一个 dev-linux-4.15 分支。

```
| figo@figo:~ben-linux-test$ git checkout dev-linux-4.0
figo@figo:~ben-linux-test$ git checkout -b dev-linux-4.15
```

因为我们已经把远程仓库主分支更新到 Linux 4.15，所以接下来把主分支上所有的提交都变基到 dev-linux-4.15 分支上。这个过程可能有冲突。

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: demo: modify Makefile
Using index info to reconstruct a base tree...
M Makefile
Falling back to patching base and 3-way merge...
Auto-merging Makefile
CONFLICT (content): Merge conflict in Makefile
error: Failed to merge in the changes.
Patch failed at 0001 demo: modify Makefile
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

这里显示在合并“demo: modify Makefile”这个补丁时发生了冲突，并且告诉你冲突的文件是 Makefile。接下来就可以手工修改 Makefile 文件并处理冲突。

```
# SPDX-License-Identifier: GPL-2.0
VERSION = 4
PATCHLEVEL = 15
SUBLEVEL = 0
EXTRAVERSION =
<<<<< 749d619c8c85ab54387669ea206cddbaf01d0772
NAME = Fearless Coyote
=====
NAME = benshushu
>>>>> demo: modify Makefile
```

手工修改冲突之后，可以通过 git diff 命令看一下变化，通过 git add 命令添加修改的文件，然后通过 git rebase --continue 命令继续做变基。当后续遇到冲突时还会停下来，让你手工修改，继续通过 git add 来添加修改后的文件，直到所有冲突被修改完成。

```
$ git add Makefile
$ git rebase --continue
Applying: demo: modify Makefile
```

当变基完成之后，我们通过 git log --oneline 命令查看 dev-linux-4.15 分支的状况。

```
| figo@figo:~ben-linux-test$ git log --oneline
344e37a demo: modify Makefile
749d619 Merge branch 'linux-4.15-org'
c67cf17 Merge branch 'linux-4.0'
```

```
f85279c first commit
d8a5b80 Linux 4.15
```

最后我们把 dev-linux-4.15 分支推送到远程仓库来完成本次项目。

```
figo@figo:~ben-linux-test$ git push origin dev-linux-4.15
```

(4) 合并 (merge) 和变基 (rebase) 分支的区别

在本实验中使用了 merge 和 rebase 来合并分支，有些读者可能有些迷惑。

```
$ git merge master
$ git rebase master
```

上述两个命令都是将主分支合并到当前分支，结果有什么不同呢？

我们假设一个 git 仓库里有一个主分支，还有一个 dev 分支，如图 2.9 所示。

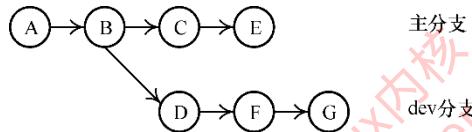


图2.9 执行合并分支之前

每个节点的提交时间如表 2.8 所示。

表 2.8 节点提交时间表

节 点	提 交 时 间
A	1号
B	2号
C	3号
D	4号
E	5号
F	6号
G	7号

在执行 git merge master 命令之后，dev 分支变成图 2.10 所示的结果。

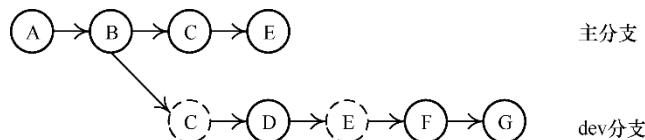


图2.10 执行git merge master合并之后的结果

我们可以看到执行 git merge master 命令之后，dev 分支上的提交都是基于时间轴来合并的。

执行 git rebase master 命令之后，dev 分支变成图 2.11 所示的结果。

2.9 实验 9：利用 git 来管理项目代码

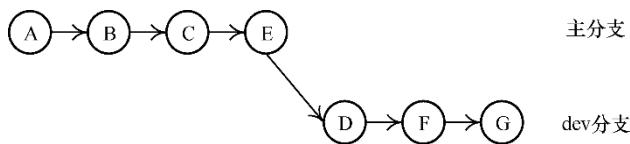


图2.11 执行git rebase master合并之后的结果

git rebase 命令用来改变一串提交基于哪个分支，如 git rebase master 就是把 dev 分支的 D、F 和 G 这 3 个提交基于最新的主分支上，也就是基于 E 这个提交之上。git rebase 的一个常见用途是保持你正在开发的分支(如 dev 分支)相对于另一个分支(如主分支)是最新的。

merge 和 rebase 命令都是用来合并分支，那分别在什么时候用呢？

- 当你需要合并别人的修改，可以考虑使用 merge 命令，如项目管理上需要合并其他开发者的分支。
- 当你的开发工作或者提交的补丁需要基于某个分支之上，就用 rebase 命令，如给 Linux 内核社区提交补丁。

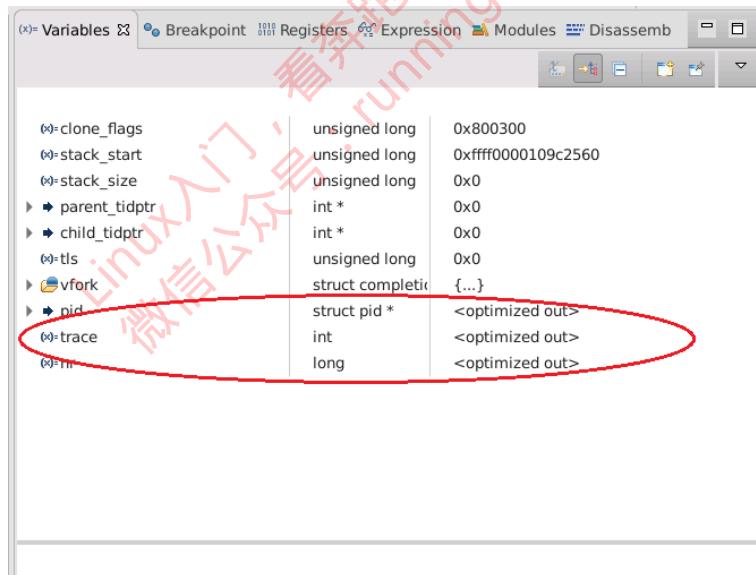
第 3 章

内核编译和调试

3.1 使用 O0 优化等级编译内核的好处

GCC 编译器有多个优化等级，比如 O0 表示关闭所有优化，O1 表示最基本的优化等级，O2 是比 O1 进阶的优化等级，也是很多软件默认使用的优化等级。Linux 内核默认是使用 O2 优化等级。

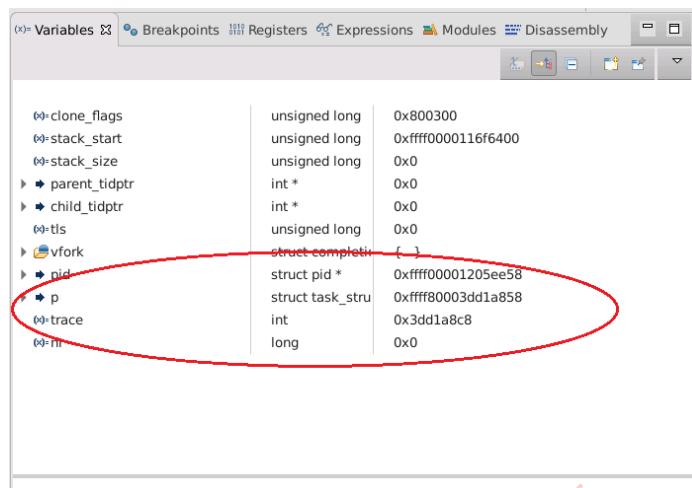
读者可能发现 `gdb` 在单步调试内核时会出现光标乱跳并且无法打印有些变量的值（例如出现`<optimized out>`）等问题，如图所示，使用 `eclipse` 进行图形化的单步调试 Linux 内核，在 `Variables` 标签页中查看变量的值，会出现大量的“`<optimized out>`”情况，影响调试效果。



使用O2优化等级编译的内核进行单步调试

其实这不是 `gdb` 或 `QEMU` 的问题，是因为内核编译的默认优化选项是 O2，因此如果不希望光标乱跳，可以尝试把 `linux-4.0` 根目录 `Makefile` 中的 O2 改成 O0，但是这样编译时有问题，本书配套的实验平台为此做了一些修改。

3.2 实验 1：通过 QEMU 调试 ARM Linux 内核



使用O0优化等级编译内核进行单步调试

最后需要特别说明一下，使用 GCC 的“O0”优化等级编译内核会导致内核运行性能下降，因此我们仅仅是为了方便单步调试内核。

本实验平台 (*runninglinuxkernel_4.0*) 默认支持使用GCC的“O0”优化等级编译内核。

3.2 实验 1：通过 QEMU 调试 ARM Linux 内核

1. 实验目的

熟悉如何使用 QEMU 调试 Linux 内核。

本实验调试 ARM32 的处理器。

2. 实验步骤

(1) 内核调试

在做本实验之前，请完成第 1 章中实验 4 的内容。

安装 `gdb-multiarch` 工具，顾名思义，它是支持多种硬件体系架构的 GDB 版本。

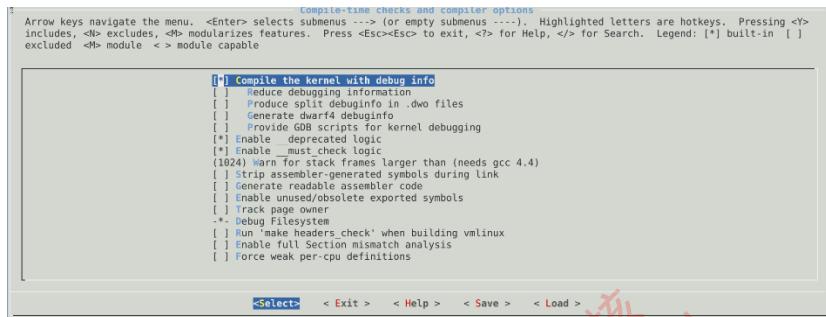
```
$ sudo apt-get install gdb-multiarch
```

进入内核配置菜单

```
$ cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make menuconfig
```

确保编译的内核包含调试信息。

```
Kernel hacking --->
Compile-time checks and compiler options --->
[*] Compile the kernel with debug info
```



重新编译内核，在超级终端中输入如下内容。

```
$ qemu-system-arm -nographic -M vexpress-a9 -m 100M -kernel arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -S -s
```

- -S: 表示 QEMU 虚拟机会冻结 CPU，直到远程的 GDB 输入相应的控制命令。
- -s: 表示在 1234 端口接受 GDB 的调试连接。

这里需要注意:-m 表示要分配给 QEMU 虚拟机多少内存，在书中指定了 1024MB，但是我们其实不需要给 QEMU 分配这么多内存，100MB 够了。

读者也可以使用如下脚本来代替上面命令。

```
#!/run.sh arm32 debug
```

```
rlk@ubuntu:runninglinuxkernel_4.0$ ./run.sh arm32 debug
Enable GDB debug mode
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
```

上面显示 pulseaudio 模块的一些错误信息，我们不用管它。

然后在另一个超级终端中启动 GDB。

```
$ cd /home/ralk/ralk_basic/runninglinuxkernel_4.0
$ gdb-multiarch --tui vmlinux
(gdb) set architecture arm          <= 设置GDB为ARM架构
(gdb) target remote localhost:1234    <= 通过1234端口远程连接到QEMU平台
(gdb) b start_kernel                <= 在内核的start_kernel处设置断点
(gdb) c
```

3.3 实验 2：通过 QEMU 调试 ARMv8 的 Linux 内核

如图 3.5 所示，GDB 开始接管 ARM-Linux 内核运行，并且到断点处暂停，这时即可使用 GDB 命令来调试内核。

```

remote Thread 1 In: start kernel
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x60000000 in ?? ()
(gdb) b start_kernel
Breakpoint 1 at 0xc065b8d8: file init/main.c, line 490.
(gdb) c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:490
(gdb)

```

图3.5 GDB调试内核

(2) 基于 runninglinuxkernel 的内核

读者可能发现，GDB 在单步调试内核时会出现光标乱跳，并且无法打印有些变量的值（例如出现<optimized out>）等问题，其实这不是 GDB 或 QEMU 的问题。这是因为内核编译的默认优化选项是 O2，因此如果不希望光标乱跳，可以尝试 runninglinuxkernel 的内核，该内核提供基于 O0 来编译内核，也就是关闭了 GCC 的所有优化选项。

下面的内容是基于第 1 章中实验 3 的，请先完成上述实验。使用 runninglinuxkernel 的内核可以通过 run.sh 脚本来启动 QEMU 的调试功能。

```
$ ./run.sh arm32 debug
```

接下来在另一个超级终端中启动 GDB 来进行内核调试。

3.3 实验 2：通过 QEMU 调试 ARMv8 的 Linux 内核

1. 实验目的

熟悉如何使用 QEMU 调试 ARMv8 的 Linux 内核。

2. 实验详解

(1) 运行 ARMv8 系统

优麒麟 Linux 系统的 QEMU 工具包里包含了 qemu-system-aarch64 工具。安装如下工具包。

```
$ sudo apt-get install gcc-aarch64-linux-gnu gcc-5-aarch64-linux-gnu
```

在优麒麟 Linux 系统中默认安装的 ARM64 版本 GCC 工具是 7.x 版本的，但是编译 Linux 4.0 内核需要使用 5.x 版本的 GCC 工具。因此，这里我们使用 update-alternatives 工具来切换 GCC 的版本，具体方法如下所示。

1) 设置gcc-5版本

```
$ sudo update-alternatives --install /usr/bin/aarch64-linux-gnu-gcc aarch64-linux-gnu-gcc /usr/bin/ aarch64-linux-gnu-gcc-5 5
```

2) 设置gcc-7版本

```
$ sudo update-alternatives --install /usr/bin/aarch64-linux-gnu-gcc aarch64-linux-gnu-gcc /usr/bin/ aarch64-linux-gnu-gcc-7 7
```

3) 选择使用gcc-5版本

```
$ sudo update-alternatives --config aarch64-linux-gnu-gcc
```

检查 aarch64-linux-gnu-gcc 版本是否为 gcc-5 版本。

```
$ aarch64-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/5/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libquadmath --enable-plugin --enable-default-pie --with-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --disable-werror --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu- --includedir=/usr/aarch64-linux-gnu/include
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu/Linaro 5.5.0-12ubuntu1)
```

同样需要编译和制作一个基于 aarch64 架构的最小文件系统，可以参照第 1 章中实验 3 的做法，只是编译环境变量不同。

```
$ cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
$ export ARCH=arm64
$ export CROSS_COMPILE=aarch64-linux-gnu-
```

下面开始编译内核，依然采用 linux-4.0 内核。

3.3 实验 2：通过 QEMU 调试 ARMv8 的 Linux 内核

```
$ cd runninglinuxkernel_4.0
$ export ARCH=arm64
$ export CROSS_COMPILE= aarch64-linux-gnu-
$ make defconfig
$ make menuconfig
```

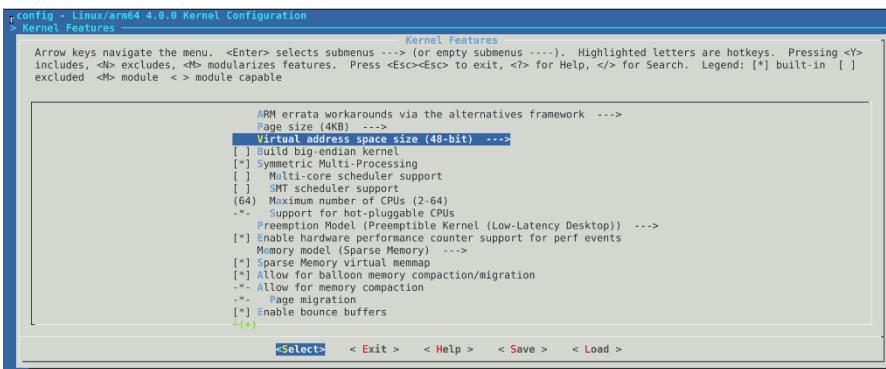
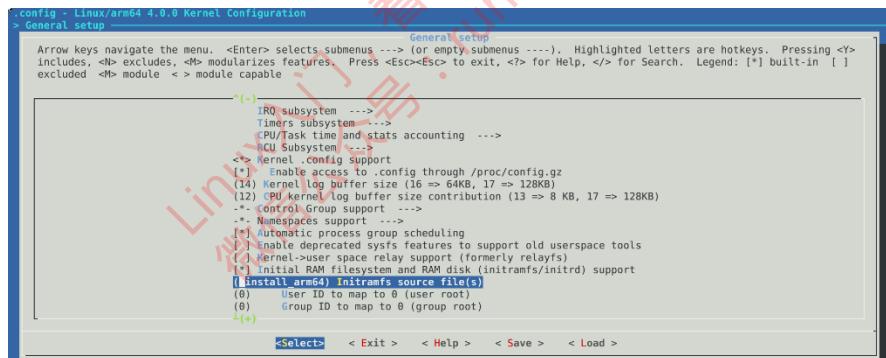
```
rlk@ubuntu:runninglinuxkernel_4.0$ export ARCH=arm64
rlk@ubuntu:runninglinuxkernel_4.0$ export CROSS_COMPILE=aarch64-linux-gnu-
rlk@ubuntu:runninglinuxkernel_4.0$ make defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
rlk@ubuntu:runninglinuxkernel_4.0$
```

依然采用 initramfs 方式来加载最小文件系统。假设编译的最小文件系统放在 Linux 4.0 根目录下，文件目录为 _install_arm64，以区别之前编译的 arm32 的最小文件系统。设置页的大小为 4KB，系统的总线位宽为 48bit。

```
General setup --->
  [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
    (_install_arm64) Initramfs source file(s)

Boot options -->
  () Default kernel command string

Kernel Features --->
  Page size (4KB) --->
    Virtual address space size (48-bit) --->
```



输入 make -j4 开始编译内核。

若出现如下错误，说明 aarch64-linux-gnu-gcc 的版本过高了。

```
scripts/Makefile.build:403: recipe for target 'scripts/mod' failed
make[1]: *** [scripts/mod] Error 2
make[1]: *** Waiting for unfinished jobs....
SHIPPED scripts/genksyms/keywords.hash.c
SHIPPED scripts/genksyms/parse.tab.h
HOSTCC scripts/genksyms/parse.tab.o
HOSTCC scripts/genksyms/lex.lex.o
CC      kernel/bounds.s
In file included from include/linux/compiler.h:54:0,
                 from include/uapi/linux/stddef.h:1,
                 from include/linux/stddef.h:4,
                 from ./include/uapi/linux/posix_types.h:4,
                 from include/uapi/linux/types.h:13,
                 from include/linux/types.h:5,
                 from include/linux/page-flags.h:8,
                 from kernel/bounds.c:9:
include/linux/compiler-gcc.h:107:1: fatal error: linux/compiler-gcc7.h: No such file or directory
 #include <gcc/header_GNUC_>
 ^~~~~
compilation terminated.
```

需要把它切换到 5.5 的版本。

运行 QEMU 来模拟 2 核 Cortex-A57 开发平台。

```
$ qemu-system-aarch64 -machine virt -cpu cortex-a57 -machine type=virt \
-nographic -m 2048 -smp 2 -kernel arch/arm64/boot/Image --append "rdinit=/ \
linuxrc console=ttyAMA0"
```

运行结果如下（删掉部分信息）。

```
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpu
Linux version 4.0.0 (figo@figo-OptiPlex-9020) (gcc version 4.9.1 20140529
(prerelease) (crosstool-NG linaro-1.13.1-4.9-2014.08 - Linaro GCC 4.9-
2014.08) ) #3 SMP PREEMPT Mon Jun 27 02:44:27 CST 2016
CPU: AArch64 Processor [411fd070] revision 0
Detected PIPT I-cache on CPU0
efi: Getting EFI parameters from FDT:
efi: UEFI not found.
cma: Reserved 16 MiB at 0x00000000bf000000
On node 0 totalpages: 524288
    DMA zone: 8192 pages used for memmap
    DMA zone: 0 pages reserved
    DMA zone: 524288 pages, LIFO batch:31
psci: probing for conduit method from DT.
psci: PSCIv0.2 detected in firmware.
psci: Using standard PSCI v0.2 function IDs
PERCPU: Embedded 14 pages/cpu @fffff80007efcb000 s19456 r8192 d29696 u57344
pcpu-alloc: s19456 r8192 d29696 u57344 alloc=14*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 516096
Kernel command line: rdinit=/linuxrc console=ttyAMA0 debug
PID hash table entries: 4096 (order: 3, 32768 bytes)
Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes)
Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes)
software IO TLB [mem 0xb8a00000-0xbca00000] (64MB) mapped at
[fffff800078a00000-fffff80007c9fffff]
Memory: 1969604K/2097152K available (5125K kernel code, 381K rwdata, 1984K
rodata, 1312K init, 205K bss, 111164K reserved, 16384K cma-reserved)
Virtual kernel memory layout:
```

3.3 实验 2：通过 QEMU 调试 ARMv8 的 Linux 内核

```

vmalloc : 0xfffff000000000000 - 0xfffff7bfff00000000 (126974 GB)
vmemmap : 0xfffff7bffc00000000 - 0xfffff7ffffc00000000 ( 4096 GB maximum)
          0xfffff7bffc10000000 - 0xfffff7bffc30000000 (   32 MB actual)
fixed : 0xfffff7fffffabfe000 - 0xfffff7fffffac000000 (     8 KB)
PCI I/O : 0xfffff7fffffae00000 - 0xfffff7ffffbe000000 (    16 MB)
modules : 0xfffff7fffffc000000 - 0xfffff8000000000000 (    64 MB)
memory : 0xfffff8000000000000 - 0xfffff8000080000000 ( 2048 MB)
  .init : 0xfffff800000774000 - 0xfffff8000008bc000 ( 1312 KB)
  .text : 0xfffff800000800000 - 0xfffff8000007734e4 ( 7118 KB)
  .data : 0xfffff8000008c0000 - 0xfffff80000091f400 (   381 KB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1
Preemptible hierarchical RCU implementation.
Additional per-CPU info printed with stalls.
RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS:64 nr_irqs:64 0
GICv2m: Node v2m: range[0x8020000:0x8020ffff], SPI[80:144]
Architected cp15 timer(s) running at 62.50MHz (virt).
sched_clock: 56 bits at 62MHz, resolution 16ns, wraps every 2199023255552ns
Console: colour dummy device 80x25
Calibrating delay loop (skipped), value calculated using timer frequency..
125.00 BogomIPS (lpj=625000)
pid_max: default: 32768 minimum: 301
Security Framework initialized
Mount-cache hash table entries: 4096 (order: 3, 32768 bytes)
Mountpoint-cache hash table entries: 4096 (order: 3, 32768 bytes)
Initializing cgroup subsys memory
Initializing cgroup subsys hugetlb
hw perfevents: no hardware support available
EFI services will not be available.
CPU1: Booted secondary processor
Detected PIPT I-cache on CPU1
Brought up 2 CPUs
SMP: Total of 2 processors activated.
devtmpfs: initialized
DMI not present or invalid.
NET: Registered protocol family 16
cpuidle: using governor ladder
cpuidle: using governor menu
vdso: 2 pages (1 code @ fffff8000008c5000, 1 data @ fffff8000008c4000)
hw-breakpoint: found 6 breakpoint and 4 watchpoint registers.
DMA: preallocated 256 KiB pool for atomic allocations
Freeing unused kernel memory: 1312K (fffff800000774000 - fffff8000008bc000)
Freeing alternatives memory: 8K (fffff8000008bc000 - fffff8000008be000)

Please press Enter to activate this console.
/ #

```

(2) 调试 ARMv8 处理器

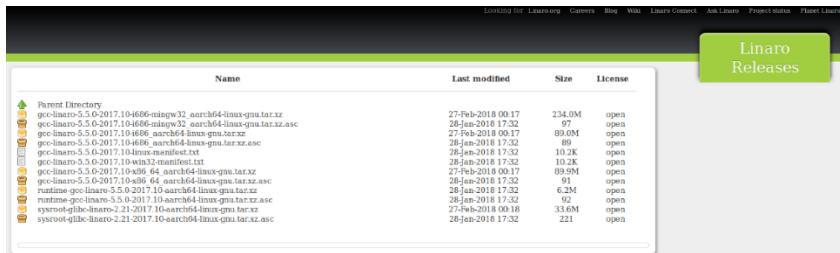
这部分内容在《奔跑吧 Linux 内核*入门篇》里没讲述，我们补充这部分内容。

由于 ubuntu 18.04 里面的 aarch64 工具链已经没有了 gdb，因为都使用 gdb-multiarch 这个工具，但是 gdb-multiarch 在单步调试 aarch64 的时候有问题，所以我们需要单独下载一个 aarch64-linux-gnu-gdb 的工具。

这里有两种办法安装 aarch64-linux-gnu-gdb 的工具。

方法一：

到 linaro 官网上下载。



使用 wget 命令来下载。

```
#wget http://releases.linaro.org/components/toolchain/binaries/5.5-2017.10/aarch64-linux-gnu/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz
```

然后解压到 home 目录下面，建议大家在 home 目录下面建立 bin 文件夹，解压这里。比如我的路径如下：/home/rbk/bin/目录。

```
rbk@ubuntu:~/bin$ gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu$ pwd
/home/rbk/bin/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu
```

然后设置系统 PATH 路径。打开/home/rbk/.bashrc 文件。在文件最后添加如下一句

```
export PATH=$PATH:/home/rbk/bin/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu/bin
```

```
112 if [ -f /usr/share/bash-completion/bash_completion ]; then
113     . /usr/share/bash-completion/bash_completion
114 elif [ -f /etc/bash_completion ]; then
115     . /etc/bash_completion
116 fi
117 fi
118
119 export PATH=$PATH://home/rbk/bin/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu/bin
120
121 export BASEINCLUDE=/home/rbk/rbk_basic/runninglinuxkernel_4.0
```

见上图中的第 119 行。

保存之后，运行如下命令。

```
# source ~/.bashrc
```

然后查看 aarch64-linux-gnu-gdb 命令是否安装正确。

3.3 实验 2：通过 QEMU 调试 ARMv8 的 Linux 内核

```
rlk@ubuntu:~/bin/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu$ aarch64-linux-gnu-gdb -v
GNU gdb (Linaro GDB-2017.10) 8.0.1.20171114-git
Copyright (C) 2017 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 for "--host=x86_64-unknown-linux-gnu --target=aarch64-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".
rlk@ubuntu:~/bin/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu$
```

方法二：

到 linaro git 官网下载 gdb 8.3 的源代码包。登录如下网站。

<https://git.linaro.org/toolchain/binutils-gdb.git/>

我们下载 binutils-gdb-gdb-8.3-release.tar.gz 这个压缩包。



```
# wget https://git.linaro.org/toolchain/binutils-gdb.git/snapshot/binutils-gdb-gdb-8.3-release.tar.gz
```

把 binutils-gdb-gdb-8.3-release.tar.gz 拷贝到 vmware 虚拟机里面。

解压并编译安装。

```
tar vxzf binutils-gdb-gdb-8.3-release.tar.gz
cd binutils-gdb-gdb-8.3-release
./configure --target=aarch64-linux-gnu
make
sudo make install
```

然后查看是否安装成功。

```
rlk@ubuntu:runninglinuxkernel_4.0$ aarch64-linux-gnu-gdb -v
GNU gdb (GDB) 8.3
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.
rlk@ubuntu:runninglinuxkernel_4.0$
```

注意：本书提供的 vmware 镜像 v4.0 版本采用方法二进行安装的 gdb。

下面来进行单步调试 aarch64 的内核。

首先保证 aarch64-linux-gnu-gcc 的版本是 5.x 的。

```
r@ubuntu:runninglinuxkernel_4.0$ aarch64-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/5/lto-wrapper
Target: aarch64-linux-gnu
Configured with: /src/configure -v --with-pkgversion='Ubuntu/Linaro 5.5.0-12ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,obj-obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot= --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-lhquadmath --enable-plugin --enable-default-pie --with-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --disable-werror --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu- --includedir=/usr/aarch64-linux-gnu/include
Thread model: posix
gcc version 5.5.0 20171010 (Ubuntu/Linaro 5.5.0-12ubuntu1)
r@ubuntu:runninglinuxkernel_4.0$
```

下面开始编译内核，首先设置 console 节点。

```
$ cd /home/rk/rk_basic/runninglinuxkernel_4.0
$ cd _install_arm64
$ mkdir dev
$ sudo mknod console c 5 1
```

然后开始编译内核。

```
$ cd /home/rk/rk_basic/runninglinuxkernel_4.0
$ export ARCH=arm64
$ export CROSS_COMPILE= aarch64-linux-gnu-
$ make defconfig
Make -j4
```

```
r@ubuntu:runninglinuxkernel_4.0$ export ARCH=arm64
r@ubuntu:runninglinuxkernel_4.0$ export CROSS_COMPILE=aarch64-linux-gnu-
r@ubuntu:runninglinuxkernel_4.0$ ls arch/arm64/configs/debian_defconfig_defconfig
r@ubuntu:runninglinuxkernel_4.0$ ls arch/arm64/configs/debian_defconfig_defconfig
r@ubuntu:runninglinuxkernel_4.0$ make defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
r@ubuntu:runninglinuxkernel_4.0$
```

下面开始调试内核。

```
$ cd /home/rk/rk_basic/runninglinuxkernel_4.0
$ qemu-system-aarch64 -machine virt -cpu cortex-a57 -machine type=virt -
nographic -m 100 -kernel arch/arm64/boot/Image --append "rdinit=/ linuxrc
console=ttyAMA0" -S -s
```

也可以使用如下脚本来替代上述命令。

```
$ cd /home/rk/rk_basic/runninglinuxkernel_4.0
$ ./run.sh arm64 debug
```

3.4 实验 3：通过 Eclipse+QEMU 单步调试内核

```
rlk@ubuntu:runninglinuxkernel_4.0$ ./run.sh arm64 debug
Enable GDB debug mode
```

在另一个终端：

```
$ cd /home/rlk/rlk_basic/runninglinuxkernel_4.0
# aarch64-linux-gnu-gdb -tui vmlinux
```

在 gdb 命令行中输入：

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000040000000 in ?? ()
(gdb) b do_fork
Breakpoint 1 at 0xffff800000d7150: file kernel/fork.c, line 1638.
(gdb) c
```

```
kernel/fork.c
1631     long do_fork(unsigned long clone_flags,
1632                  unsigned long stack_start,
1633                  unsigned long stack_size,
1634                  int __user *parent_tidptr,
1635                  int __user *child_tidptr)
1636     {
1637         struct task_struct *p;
1638         int trace = 0;
1639         long nr;
1640
1641         /*
1642          * Determine whether and which event to report to ptracer. When
1643          * called from kernel_thread or CLONE UNTRACED is explicitly
1644          * requested, no event is reported; otherwise, report if the event
1645          * for the type of forking is enabled.
1646         */
1647         if (!(clone_flags & CLONE UNTRACED)) {
```

Breakpoint 1 at 0xffff800000d7150: file kernel/fork.c, line 1638.
The "remote" target does not support "run". Try "help target" or "continue".
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, do_fork (clone_flags=8389376, stack_start=1844660336232402556, stack_size=0, parent_tidptr=0x0, child_tidptr=0x0)

我们使用“info reg”命令可以看到显示 ARMv8 中所有的系统寄存器。

```
remote Thread 1.1 In: do_fork
ID_AA64AFR0_EL1 0x0          0
ID_AA64AFR1_EL1 0x0          0
ID_AA64AFR2_EL1_RESERVED 0x0  0
ID_AA64AFR3_EL1_RESERVED 0x0  0
ID_AA64ISAR0_ELI 0x11120    69920
ID_AA64ISAR1_ELI 0x0        0
ID_AA64ISAR5_ELI_RESERVED 0x0 0
ID_AA64ISAR6_ELI_RESERVED 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--
```

3.4 实验 3：通过 Eclipse+QEMU 单步调试内核

1. 实验目的

熟悉如何使用 Eclipse+QEMU 以图形方式单步调试 Linux 内核。

2. 实验详解

本章实验 1 介绍了如何使用 GDB 和 QEMU 调试 Linux 内核源代码。由于 GDB 是命令行的方式，可能有些读者希望在 Linux 中能有类似 Virtual C++ 图形化的开发工具。这里介绍使用 Eclipse 工具来调试内核。Eclipse 是著名的跨平台的开源集成开发环境（IDE），最初主要用于 JAVA 语言开发，目前可以支持 C/C++、Python 等多种开发语言。Eclipse 最初由 IBM 公司开发，2001 年被贡献给开源社区，目前很多集成开发环境都是基于 Eclipse 完成的。

（1）安装 Eclipse-CDT 软件

Eclipse-CDT 是 Eclipse 的一个插件，可以提供强大的 C/C++ 编译和编辑功能。

```
$ sudo apt install eclipse-cdt①
```

由于优麒麟 ubuntu 18.04 默认安装的 eclipse-cdt 不能运行，所以我们到 eclipse 官网上下载一个。<http://www.eclipse.org/cdt/>

本书配套的 vmware 镜像里安装的版本是 2018 年 3 月的 Oxygen.3。读者可以到 eclipse 官网下载比较新的版本。



打开 Eclipse 菜单，选择“Help”→“About Eclipse”，可以看到当前软件的版本，如图所示。

^① 截至 2018 年 6 月，Ubuntu 18.04 系统默认安装的 Eclipse CDT 工具不能运行，读者可以到 Eclipse CDT 官网上直接下载 Oxygen.3 版本 x86_64 的 Linux 版本压缩包，解压并打开二进制文件即可。

3.4 实验 3：通过 Eclipse+QEMU 单步调试内核

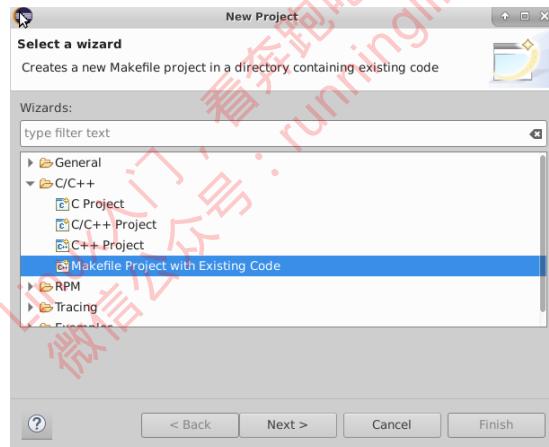


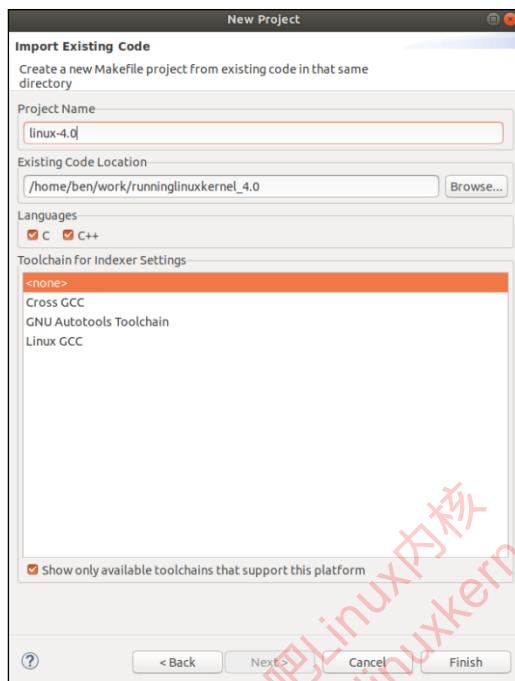
Eclipse CDT版本

(2) 创建工程

打开 Eclipse 菜单，选择“Window”→“Open Perspective”→“C/C++”。

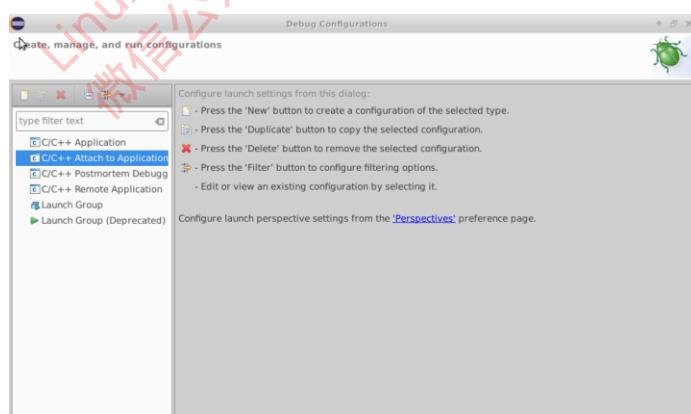
新建一个 C/C++ 的 Makefile 工程，在“File”→“New”→“Project”中选择“Makefile Project with Existing Code”，创建一个新的工程，如图 3.7 所示。





创建工程

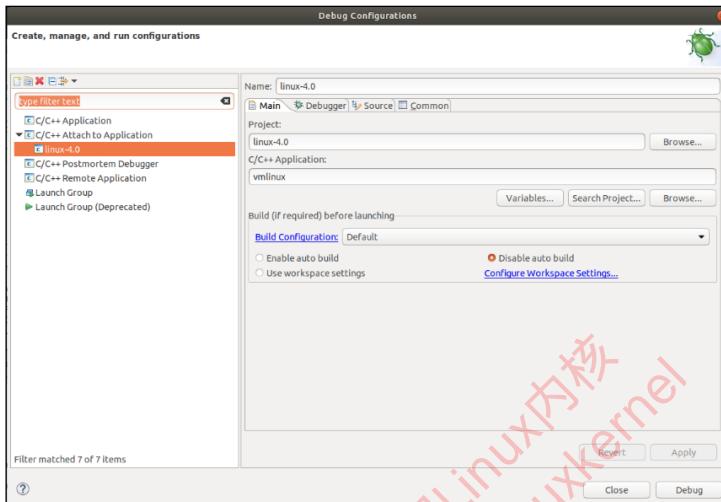
接下来配置调试选项。选择 Eclipse 菜单中的“Run” → “Debug Configurations”选项，创建一个“C/C++ Attach to Application”调试选项。双击“C/C++ Attach to Application”可以创建一个新的连接。



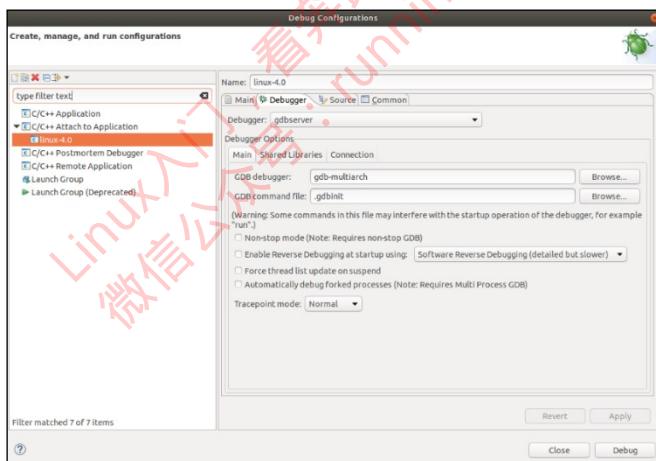
- Project: 选择刚才创建的工程。
- C/C++ Application: 选择编译 Linux 内核带符号表信息的 vmlinux。
- Build before launching: 选择“Disable auto build”，如图所示。
- Debugger: 选择 gdbserver。

3.4 实验 3：通过 Eclipse+QEMU 单步调试内核

- GDB debugger: 填入 gdb-multiarch, 如图所示。
- Host name or IP address: 填入 localhost。
- Port number: 填入 1234。



debug配置选项



debugger配置

单击“Apply”按钮来完成配置。

然后单击“Debug”按钮。

在优麒麟 Linux 的一个终端中先打开 QEMU。为了调试方便，这里没有指定多个 CPU，而是单个 CPU。

```
$ cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
$ qemu-system-arm -nographic -M vexpress-a9 -m 1024M -kernel
```

奔跑吧 linux 社区出品

```
arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -S -s
```

上述命令可以使用如下脚本来代替。

```
$ cd runninglinuxkernel_4.0
./run.sh arm32 debug
```



“小昆虫”图标

在 Eclipse 菜单的“Run”→“Debug History”中选择刚才创建的调试选项，或在快捷菜单中单击“小昆虫”图标，如图所示。

在 Eclipse 的 Debugger Console 控制台中输入“file vmlinux”命令，导入调试文件的符号表；输入“set architecture arm”选择 GDB 支持的 ARM 架构，如图所示。

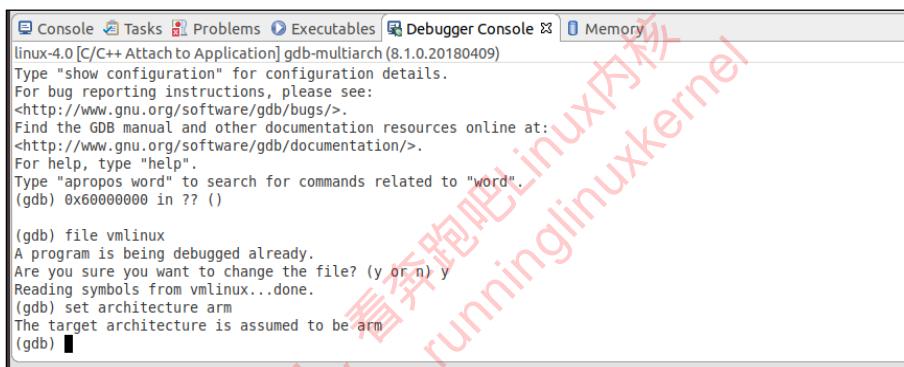
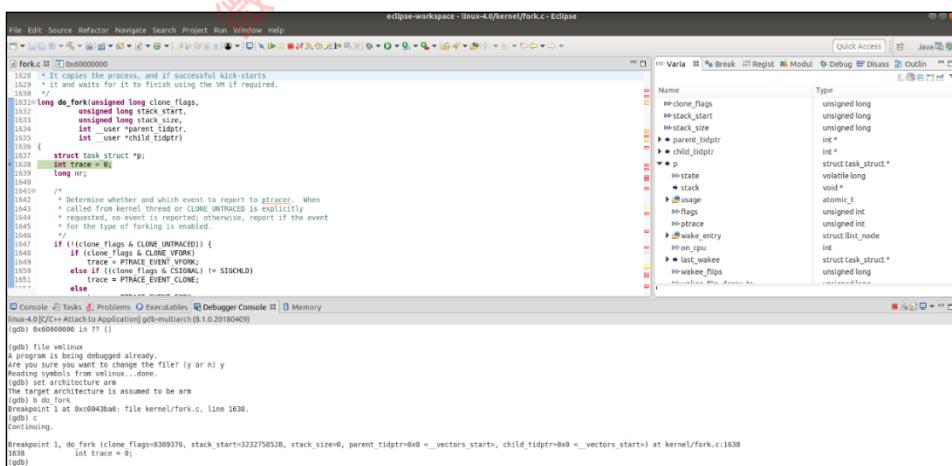


图3.11 Debugger Console控制台

在 Debugger Console 中输入“b do_fork”，在 do_fork 函数中设置一个断点。输入“c”命令，开始运行 QEMU 中的 Linux 内核，它会停在 do_fork 函数中，如图所示。

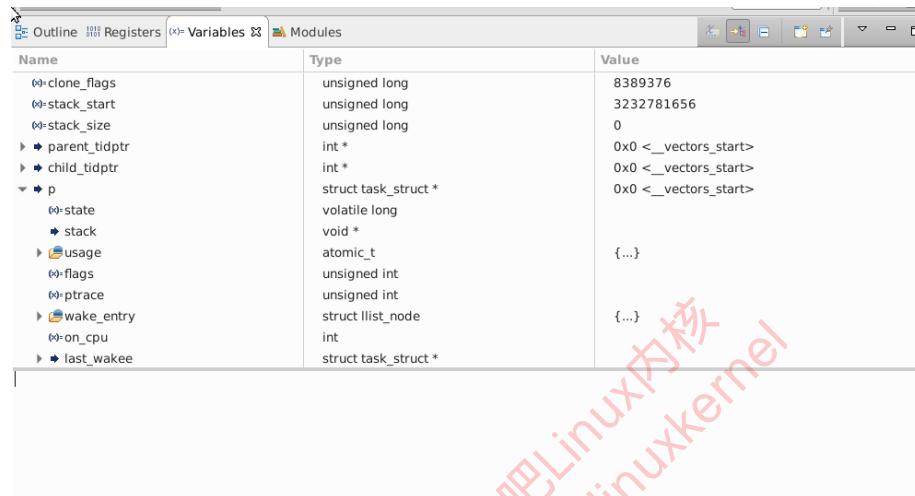


Eclipse调试内核

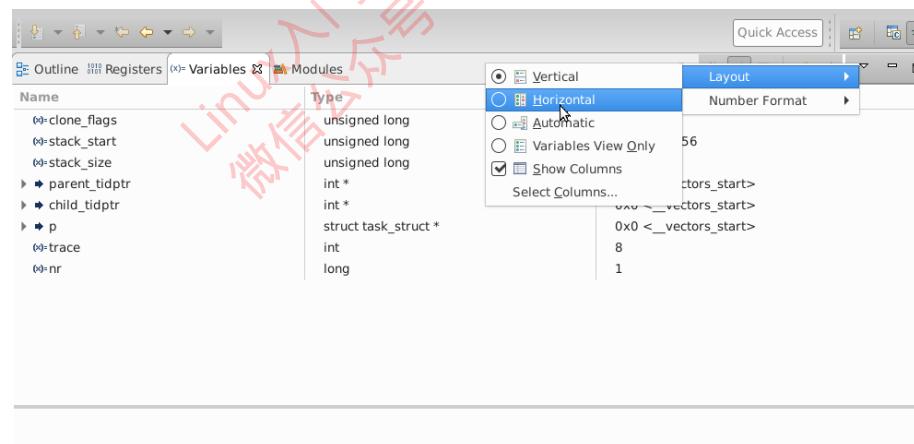
3.4 实验 3：通过 Eclipse+QEMU 单步调试内核

Eclipse 调试内核比使用 GDB 命令要直观很多，例如参数、局部变量和数据结构的值都会自动显示在“Variables”标签卡上，不需要每次都使用 GDB 的打印命令才能看到变量的值。读者可以单步并且直观地调试内核。

其中“Variables”标签卡上实时显示了当前系统的局部变量以及全局变量的值。



比如上图中的变量 p，表示的是局部变量 task_struct 数据结构，它可以显示这个 task_struct 数据结构每个成员的值。另外 clone_flags，stack_start 等是函数 do_fork 的函数，它都能实时显示当前的值。注意这里默认显示的十进制，我们可以把它设置为模式显示十六进制。



在“Register”标签卡上，显示当前系统的寄存器，包括通用寄存器和系统寄存器的值。

奔跑吧 linux 社区出品

Name	Value	Description
General Registers		
r0	8389376	General Purpose and FPU Register Group
r1	-1062185640	
r2	0	
r3	0	
r4	-1055967360	
r5	-1056193524	
r6	-1059542300	
r7	-1056178912	
r8	1610629226	
r9	1091551376	
r10	0	
r11	0	
r12	4	
sp	0xc10bbe48	
lr	-1073462772	
pc	0x0043ba0	
cpsr	1610613075	
d0	{u8 = {0, 0, 0}}	
d1	{u8 = {0, 0, 0}}	

还能显示系统寄存器的值。比如我们需要查看系统控制器寄存器 SCTRLR。

Name	Value	Description
IFSR	0x0	
AFSR1_EL3	0x0	
MIDR_S	0x410fc090	
ACTLR_EL2	0x0	
VBAR_S	0x0	
SDER	0x0	
CBAR	0x1e000000	
SCTRLR	0xc50078	
ACTLR_EL1	0x0	
PMEVCNTR3	0x0	
PMEVCNTR0	0x0	
PMEVCNTR1	0x0	
PMEVCNTR2	0x0	
VBAR	0x0	

Name : SCTRLR
Hex:0xc50078
Decimal:12910712
Octal:061200170
Binary:11000010100000000001110000
Default:12910712

系统控制器寄存器 SCTRLR 的描述在 ARMv7 芯片手册中。见<ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition>第 B4.1.130 章。

3.5 实验 4：在 QEMU 中添加文件系统的支持

B4.1.130 SCTLR, System Control Register, VMSA

The SCTLR characteristics are:

Purpose The SCTLR provides the top level control of the system, including its memory system.
This register is part of the Virtual memory control registers functional group.

Usage constraints Only accessible from PL1 or higher.
Control bits in the SCTLR that are not applicable to a VMSA implementation read as the value that most closely reflects that implementation, and ignore writes.

In ARMv7, some bits in the register are read-only. These bits relate to non-configurable features of an ARMv7 implementation, and are provided for compatibility with previous versions of the architecture.

Configurations In an implementation that includes the Security Extensions, the SCTLR:

- is Banked, with some bits common to the Secure and Non-secure copies of the register
- has write access to the Secure copy of the register disabled when the CP15SDISABLE signal is asserted HIGH.

For more information, see *Classification of system control registers* on page B3-1439.

Attributes A 32-bit RW register with an IMPLEMENTATION DEFINED reset value, see *Reset value of the SCTLR* on page B4-1693. See also *Reset behavior of CP14 and CP15 registers* on page B3-1438.

Note

In an implementation that includes the Virtualization Extensions, some reset requirements apply to the Non-secure copy of SCTLR.

Table B3-44 on page B3-1476 shows the encodings of all of the registers in the Virtual memory control registers functional group.

3.5 实验 4：在 QEMU 中添加文件系统的支持

1. 实验目的

熟悉如何在 QEMU 中添加文件系统的支持。

2. 实验详解

在优麒麟 Linux 中创建一个 64MB 的镜像（image）。

```
$ dd if=/dev/zero of=swap.img bs=512 count=131072 <=这里使用dd命令
```

然后通过 SD 卡的方式加载 swap.img 到 QEMU 中。

```
$ qemu-system-arm -nographic -M vexpress-a9 -m 64M -kernel arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -sd swap.img
[...]
# mkswap /dev/mmcblk0 <=第一次需要格式化swap分区
# swapon /dev/mmcblk0 <= 使能swap分区
# free
total        used         free       shared      buffers
Mem:    1026368     9844     1016524      1360          4
-/+ buffers:      9840     1016528
Swap:   65532          0      65532 <= 可以看到swap分区已经工作了
```

下面创建一个 ext4 文件系统分区。先在 Ubuntu 中创建一个 64MB 大小的镜像，方法 同上。

```
$ dd if=/dev/zero of=ext4.img bs=512 count=131072 <=创建一个镜像
$ mkfs.ext4 ext4.img <=格式化ext4.img成ext4格式
```

挂载 ext4 文件系统需要打开如下配置的选项。

```
[arch/arm/configs/vexpress_defconfig]
CONFIG_LBDAF=y
CONFIG_EXT4_FS=y
```

重新编译内核， make vexpress_defconfig && make。

```
$ qemu-system-arm -nographic -M vexpress-a9 -m 1024M -kernel
arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -
dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -sd ext4.img
[...]
# mount -t ext4 /dev/mmcblk0 /mnt/ <=挂载SD卡到/mnt目录
```

注意：读者想调试 ext4 的代码，也可以使用第一章中的实验 5~7 中的 QEMU+Debian 的方案，我们制作的 rootfs 采用 ext4 文件系统。

3.6 实验 5：使用 DS-5 单步调试 arm64 内核

1. 实验目的

熟悉使用 ARM 公司的 DS-5 社区版软件来单步调试内核。

注意：选做。本实验步骤繁琐而且有一定难度，留给学有余力的读者来独自完成。

2. 实验要求

使用 DS-5 社区版本来单步调试 ARM64 的内核。

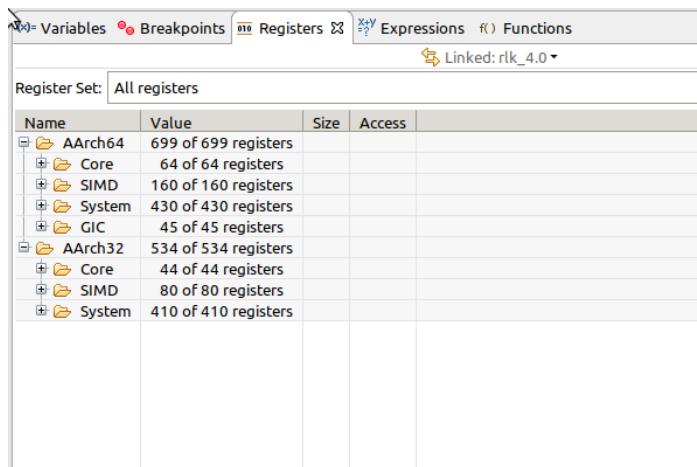
3. 使用 DS-5 调试内核的好处

我们在第 3 章的实验 3 里介绍了使用 Eclipse+QEMU 方式也可以图形化单步调试内核，那么使用 DS-5 社区版本调试内核的优势在哪呢？

第一个好处就是很方便查看 ARMv8 芯片内部的寄存器的值，包括通用寄存器，系统寄存器，特殊寄存器

打开“Register”窗口。寄存器分成 AArch64 和 AArch32 两大类。每一类下面又分成 Core、SIMD、System 和 GIC 几类寄存器。

3.6 实验 5：使用 DS-5 单步调试 arm64 内核



Core 寄存器主要是通用寄存器。

AArch64 699 of 699 registers			
-	X0	0x0000000000000000	64 R/W
-	X1	0xFFFF800000AFFE7C	64 R/W
-	X2	0x0000000000000000	64 R/W
-	X3	0x0000000000000000	64 R/W
-	X4	0x0000000000000000	64 R/W
-	X5	0x0000000000000000	64 R/W
-	X6	0xFFFF80007EF7AC20	64 R/W
-	X7	0x0000000000000000	64 R/W
-	X8	0xFFFF800078400D40	64 R/W
-	X9	0x0000000000000000	64 R/W
-	X10	0x000000000000000B	64 R/W
-	X11	0x000000000000002B	64 R/W
-	X12	0x0000000000000000	64 R/W
-	X13	0xFFFFFFFFFFFFFFFFFF	64 R/W
-	X14	0xFFFFF000000000000	64 R/W
-	X15	0xFFFFFFFFFFFFFFFFF	64 R/W
-	X16	0xFFFF800000E6FFF0	64 R/W
-	X17	0x0000000000000000	64 R/W
-	X18	0x0000000000000000	64 R/W
-	X19	0x0000000008000000	64 R/W
-	X20	0x0000000000000E12	64 R/W
-	X21	0x0000000008800000	64 R/W
-	X22	0x00000000410FD000	64 R/W
-	X23	0x00000000080E7A2B	64 R/W
-	X24	0x0000000008000000	64 R/W
-	X25	0x00000000080EF5000	64 R/W
-	X26	0x00000000080EF8000	64 R/W
-	X27	0xFFFF800000081260	64 R/W
-	X28	0x0000800008000000	64 R/W
-	X29	0xFFFF800000E6FEC0	64 R/W

打开“system”寄存器的标签页，可以看到系统寄存器按照功能进行分类。

奔跑吧 linux 社区出品

Name	Value	Size	Access
AArch64	699 of 699 registers		
Core	64 of 64 registers		
SIMD	160 of 160 registers		
System	430 of 430 registers		
Address	13 of 13 registers		
Cache	11 of 11 registers		
Debug	82 of 82 registers		
Exception	18 of 18 registers		
Float	6 of 6 registers		
GIC	46 of 46 registers		
ID	39 of 39 registers		
IMP_DEF	14 of 14 registers		
Memory	17 of 17 registers		
Other	7 of 7 registers		
PMU	32 of 32 registers		
PSTATE	5 of 5 registers		
Reset	2 of 2 registers		
Secure	11 of 11 registers		
Special	16 of 16 registers		
TLB	32 of 32 registers		
Thread	5 of 5 registers		
Timer	18 of 18 registers		
Virt	56 of 56 registers		
GIC	45 of 45 registers		
AArch32	534 of 534 registers		
Core	44 of 44 registers		
SIMD	80 of 80 registers		
System	410 of 410 registers		

这些寄存器除了按照功能进行分类，还把寄存器相关的字段进行了解析，非常方便工程师查看这些字段（域）的值。比如查看 PSTATE 寄存器每个域的值。

PSTATE 5 of 5 registers				
CurrentEL	0x00000004	32	RO	
DAIF	0x00000240	32	R/W	
NZCV	0x60000000	32	R/W	
SPSel	0x00000001	32	R/W	
Mode	0x00000005	32	R/W	

比如我们看系统控制寄存器 SCTRL_E1，它把每一个域的说明和当前的值都显示出来。

SCTRL_E1				
UCI	0x34D5D91D	32	R/W	
EE	This control does not cause any instructions ...	1	R/W	
E0E	Exp ✓ 0 : Explicit data accesses at EL1, and stage 1 tran...	1	R/W	
WXXN	Exp 1 : Explicit data accesses at EL1, and stage 1 tran...	1	R/W	
nTWE	This control does not cause any instructions ...	1	R/W	
nTWI	This control does not cause any instructions ...	1	R/W	
UCT	This control does not cause any instructions ...	1	R/W	
DZE	This control does not cause any instructions ...	1	R/W	
I	This control has no effect on the Cacheabilit...	1	R/W	
UMA	Any attempt at EL0 using AArch64 to execute a...	1	R/W	
SED	SETEND instructions are UNDEFINED at EL0 usin...	1	R/W	
ITD	All IT instruction functionality is enabled a...	1	R/W	
CP15BEN	EL0 using AArch32: EL0 execution of the CP15D...	1	R/W	
SAO	0x1	1	R/W	
SA	0x1	1	R/W	
C	This control has no effect on the Cacheabilit...	1	R/W	
A	Alignment fault checking disabled when execut...	1	R/W	
M	EL1 and EL0 stage 1 address translation enabl...	1	R/W	
SCTRL_E2	0x30C50830	32	R/W	
SCTRL_E3	0x30C50830	32	R/W	

另外 DS-5 还内置了寄存器的在线文档。

3.6 实验 5 : 使用 DS-5 单步调试 arm64 内核

The screenshot shows the DS-5 debugger's web interface for the SCLTR_EL1 register. The top navigation bar includes tabs for AArch32 Registers, AArch64 Registers, AArch32 Instructions, AArch64 Instructions, Index by Encoding, External Registers, External Registers by Offset, and Registers by Functional Group. The main content area is titled "SCLTR_EL1, System Control Register (EL1)". It details the register's characteristics, purpose (providing top-level control of the system), usage constraints (access levels across EL0-EL3), traps and enables (prioritization rules), and configuration (architectural mapping to AArch32 System register SCLTR).

4. 实验步骤

(1) 下载安装 ARM 公司的 DS-5 社区版本软件

为了帮助芯片公司和硬件厂商更好的调试 ARM 芯片，ARM 公司开发了一套仿真器和配套的软件，这个软件叫做 DS-5。DS-5 集成了很多很有用的调试特性，充分利用硬件仿真器的优势，可以深入到 ARM 芯片内部进行调试。由于硬件仿真器和 DS-5 软件都是需要购买授权的，因此大部分做 ARM 芯片和硬件的公司都会购买。

DS-5 有一个免费的社区版本 (DS-5 Community Edition)，这是 DS-5 软件的简化版，删掉了很多功能，但是我们依然可以使用 DS-5 来单步调试 Linux 内核。

DS-5 Editions Quick Reference

Feature	Community	Professional	Ultimate
Eclipse IDE	Yes	Yes	Yes
Arm Compilers	No	<div style="width: 50%;"><div style="background-color: #0070C0;"></div></div>	<div style="width: 100%;"><div style="background-color: #0070C0;"></div></div>
Linaro GCC	Yes	Yes	Yes
DS-5 Debugger	<div style="width: 10%;"><div style="background-color: #000000;"></div></div>	<div style="width: 50%;"><div style="background-color: #0070C0;"></div></div>	<div style="width: 100%;"><div style="background-color: #0070C0;"></div></div>
CoreSight Trace (ETM, PTM, ITM, STM)	No	Yes	Yes
Streamline Performance Analyzer	<div style="width: 10%;"><div style="background-color: #000000;"></div></div>	<div style="width: 50%;"><div style="background-color: #0070C0;"></div></div>	<div style="width: 100%;"><div style="background-color: #0070C0;"></div></div>
Simulation with Fixed Virtual Platform (FVP)	<div style="width: 10%;"><div style="background-color: #000000;"></div></div>	<div style="width: 50%;"><div style="background-color: #0070C0;"></div></div>	<div style="width: 100%;"><div style="background-color: #0070C0;"></div></div>
Processor Support	<div style="width: 10%;"><div style="background-color: #000000;"></div></div>	<div style="width: 50%;"><div style="background-color: #0070C0;"></div></div>	<div style="width: 100%;"><div style="background-color: #0070C0;"></div></div>

读者可以到 ARM 公司官网下载。

奔跑吧 linux 社区出品

Development Studio 5 Community Edition

The Arm Development Studio 5 (DS-5) Community Edition is a free professional quality tool chain developed by Arm to accelerate your first steps in Arm software development. Based on DS-5 Ultimate Edition, this toolkit offers essential debug and system analysis for you to create robust and highly optimized software for Arm processor-based devices.

Arm DS-5 Community Edition includes:

- DS-5 Debugger for Linux native language applications.
- DS-5 Debugger for bare-metal development with Armv8-A Foundation Model.
- Arm Streamline performance analyzer for Linux/Android™ applications - Basic features.
- Online help and Software examples
- Linux and Windows host support (64 bit hosts only)

Some third-party compilers are compatible with DS-5. For example, the GNU Compiler tools enable you to compile bare-metal, Linux kernel, and Linux applications for Arm targets.

[Download](#) [Get Started](#)

本实验下载的版本为 v5.29.1，下载的是 Linux 版本的安装包：ds5-ce-linux64-29rel1.tgz

解压后，执行./install.sh 安装文件，按照安装提示进行安装。

安装完成之后，选择“Eclipse for DS-5 CE v5.19.1”菜单进行打开。



下面是 DS-5 CE 版本启动界面。



3.6 实验 5 : 使用 DS-5 单步调试 arm64 内核

(2) 编译 FVP 模拟平台运行的 image

DS-5 软件内置了 ARM 公司开发的模拟器，叫做 FVP (Fixed Virtual Platform)，这个模拟平台和 QEMU 不一样，因此我们需要把编译好的内核 image 按照这个模拟平台的要求重新做一个引导程序。

笔者在 runninglinuxkernel_4.0 目录创建了一个自动化的脚本。

```
$ cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
$ ./build_ds5_arm64.sh
```

编译好的 image 文件是在 boot-wrapper-aarch64 目录下面。

```
r1k@ubuntu:boot-wrapper-aarch64$ ls
aclocal.m4      cache.o      fvp-base-gicv3-psci.dtb  Makefile.am
adpsci.pl      compile     gic.c                  Makefile.in
arch           config.log   gic-v3.c             missing
autom4te.cache config.status gic-v3.o            model.lds
bakery_lock.c  configure   include              model.lds.S
bakery_lock.o  configure.ac install-sh          platform.c
boot_common.c   fdt.dtb    lib.c                platform.o
boot_common.o   FDT.pm     lib.o                psci.c
build_boot_5.sh findbase.pl LICENSE.txt        psci.o
build_boot.sh   findcpuids.pl linux-system.axf README
cache.c         findmem.pl Makefile            tags
r1k@ubuntu:boot-wrapper-aarch64$
```

从上图可以看到，我们生成了一个 linux-system.axf 文件，这个文件就是把 Linux 内核 image 重新打包之后的镜像文件。

(3) 跑 FVP 模拟平台。

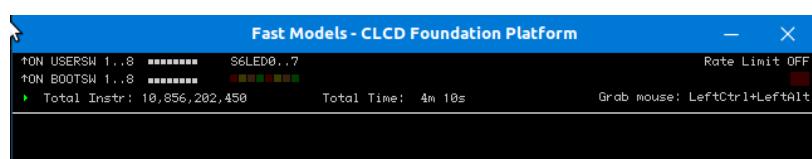
编译好了 axf 文件之后，我们可以尝试使用 FVP 模拟平台来跑一下 Linux 系统。

```
#cd boot-wrapper-aarch64
#/usr/local/DS-5_CE_v5.29.1/sw/models/bin/Foundation_Platform linux-
system.axf
```

```
r1k@ubuntu:boot-wrapper-aarch64$ /usr/local/DS-5_CE_v5.29.1/sw/models/bin/Foundation_Platform --image linux-system.axf
Fast Models [11.4.35 (Jun 18 2018)]
Copyright 2000-2018 ARM Limited.
All Rights Reserved.

terminal_0: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_3: Listening for serial connection on port 5003
```

它会新建两个窗口。一个是管理窗口：Fast Models，另外一个是 FVP 终端窗口。





The screenshot shows a terminal window titled "FVP terminal_0". The window contains a black background with white text representing the boot logs of a Linux kernel. The logs include messages about network drivers (mip6, sit, ip6_gret), protocol families (NET), certificates (X.509), and system time (RTC). It also shows errors related to the /etc/init.d/rcS script and the 9pnet_virtio device. At the end of the log, it prompts the user to press Enter to activate the console.

```
[ 0.000000] mip6: Mobile IPv6
[ 0.000000] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[ 0.000000] ip6_gret: GRE over IPv6 tunneling driver
[ 0.000000] NET: Registered protocol family 17
[ 0.000000] NET: Registered protocol family 15
[ 0.000000] 9pnet: Installing 9P2000 support
[ 0.000000] Loading compiled-in X.509 certificates
[ 0.000000] zswap: loaded using pool lzo/zbud
[ 0.000000] rtc-p1031 1c170000 rtc: setting system clock to 2019-08-21T10:45:
25 UTC (1566384325)
[ 0.000000] Freeing unused kernel memory: 6464K
[ 0.000000] Run /linuxrc as init process
/etc/init.d/rcS: line 9: can't create /proc/sys/kernel/hotplug: nonexistent directory
[ 0.000000] 9pnet_virtio: no channels available for device kmiod_mount
mount: mounting kmiod_mount on /mnt failed: No such file or directory
Memory for crashkernel is not reserved
Please reserve memory by passing "crashkernel=XY" parameter to kernel
Then try to loading kdump kernel

Please press Enter to activate this console.
/ #
/ #
/ # ]
```

在 FVP 终端窗口中，可以看到我们的 Linux 系统已经跑起来了。

关闭 Fast Models 窗口可以关闭 FVP 模拟器。我们暂时先关闭 FVP 模拟器。

(4) 单步调试内核

我们把剩下的步骤留给读者来独自完成。

笔者录制了一期关于如何使用 DS-5 进行单步调试内核的视频节目，有兴趣读者可以关注附录 2。

4.1 实验 1：编写一个简单的内核模块

第 4 章

内核模块

4.1 实验 1：编写一个简单的内核模块

1. 实验目的

了解和熟悉编译一个基本的内核模块需要包含的元素。

2. 实验步骤

- 1) 编写一个简单的内核模块程序。
- 2) 编写对应的 Makefile 文件。
- 3) 在优麒麟 Linux 机器上编译和加载运行该内核模块。
- 4) 在 QEMU 上运行 ARM32 的 Linux 系统，编译该内核模块并运行。

编译一个在 ARM32 的 Linux 系统中运行的内核模块和之前我们介绍的 Ubuntu 的方法略有不同。需要手工编写一个 Makefile 文件，示例如下如下：

```

0   BASEINCLUDE ?= /home/r1k/r1k_basic/runninglinuxkernel_4.0
1
2   mytest-objs := my_test.o
3   obj-m     := mytest.o
4
5   all :
6   $(MAKE) -C $(BASEINCLUDE) M=$(PWD) modules;
7
8   clean:
9   $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
10  rm -f *.ko;
```

最大的不同就是第 0 行的 BASEINCLUDE 需要指定到编译 ARM32 的内核目录，并且该内核目录需要提前编译完成。

本实验的参考代码是在：

/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_4/lab1_simple_module

我们以本实验代码为例来介绍如何编译和安装内核模块。

以 vexpress 板子为例：

```
$ cd /home/r1k/r1k_basic/runninglinuxkernel_4.0 //进入runninglinuxkernel内核目  
录
```

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make -j4
```

内核编译完成支持就可以编译内核模块了。进入本实验的参考代码目录。

```
$ cd
/home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make
```

```
rlk@ubuntu:lab1_simple_modules$ export ARCH=arm
rlk@ubuntu:lab1_simple_modules$ export CROSS_COMPILE=arm-linux-gnueabi-
rlk@ubuntu:lab1_simple_modules$ make
make -C /home/rlk/rlk_basic/runninglinuxkernel_4.0 M=/home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module modules
make[1]: Entering directory '/home/rlk/rlk_basic/runninglinuxkernel_4.0'
CC [M] /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/my_test.o
LD [M] /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/mytest.o
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/mytest.mod.o
LD [M] /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/mytest.ko
make[1]: Leaving directory '/home/rlk/rlk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab1_simple_modules $
```

注意：

1. 我们在编译内核模块的时候，常常会遇到这样的错误。如下图显示：

```
rlk@ubuntu:lab1_simple_modules$ make
make -C /home/rlk/rlk_basic/runninglinuxkernel_4.0 M=/home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module modules
make[1]: Entering directory '/home/rlk/rlk_basic/runninglinuxkernel_4.0'
CC [M] /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/my_test.o
In file included from ./arch/x86/include/asm/bittops.h:16:0,
                 from include/linux/bittops.h:36,
                 from include/linux/kernel.h:10,
                 from include/linux/list.h:8,
                 from include/linux/module.h:9,
                 from /home/rlk/rlk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_4/lab1_simple_module/my_test.c:1:
./arch/x86/include/asm/arch_hweight.h: In function '_arch_hweight64':
./arch/x86/include/asm/arch_hweight.h:53:42: error: expected ';' or ')' before 'POPCNT64'
    asm (ALTERNATIVE("call _sw_hweight64", POPCNT64, X86_FEATURE_POPCNT)
          ^
./arch/x86/include/asm/arch_hweight.h:53:31: note: in definition of macro 'ALTINSTR_REPLACEMENT'
    b_replacement(number)";\n\t" newinstr "\n" e_replacement(number) ";\n\t"
          ^
./arch/x86/include/asm/arch_hweight.h:53:7: note: in expansion of macro 'ALTINSTR'
    asm (ALTERNATIVE("call _sw_hweight64", POPCNT64, X86_FEATURE_POPCNT)
          ^
In file included from ./arch/x86/include/asm/thread_info.h:23:0,
                 from include/linux/thread_info.h:54,
                 from ./arch/x86/include/asm/preempt.h:6,
                 from include/linux/preempt.h:18,
```

上述错误说明在编译模块时候没有指定“**ARCH**”和编译器。设置如下即可。

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
```

2. 若在环境变量中没有设置**BASEINCLUDE**，那么需要在**make**命令时候手工指定。

使用**env**命令来检查环境变量。

```
rlk@figo-OptiPlex-9020:block_driver$ env | grep BASE
BASEINCLUDE=/home/rlk/rlk_basic/runninglinuxkernel_4.0/
rlk@figo-OptiPlex-9020:block_driver$
```

若没有设置**BASEINCLUDE**变量，需要手工添加。

4.1 实验 1：编写一个简单的内核模块

修改`/home/rbk/.bashrc`文件，增加一行。

```
export BASEINCLUDE=/home/rbk/rbk_base/runninglinuxkernel_4.0
```

```
120
121 export BASEINCLUDE=/home/rbk/rbk_basic/runninglinuxkernel_4.0
NORMAL <kernel_4.0 .bashrc
1:.bashrc
```

注意：若读者没有配置这个环境变量，那么在编译内核模块时候，需要手工指定 `BASEINCLUDE` 目录。例如

```
# make BASEINCLUDE=/home/xxx/xxx //指定runninglinuxkernel_4.0的绝对路径
```

编译完成之后就看到 `mytest.ko` 文件。用 `file` 命令检查编译的结果是否为 ARM 架构的 格式。

```
$ file mytest.ko
mytest.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV),
BuildID[sha1]=0c18be5a38637ba895b60487c805827233dceef, not stripped
```

```
rlk@ubuntu:lab1_simple_module$ file mytest.ko
mytest.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), BuildID[sha1]=081b3f5c6e75a77
3c0bc6b6bb6672de5ec22706e, with debug_info, not stripped
rlk@ubuntu:lab1_simple_module$
```

这时可以把该 `ko` 文件复制到 `runninglinuxkernel_4.0/kmodules` 目录下。

```
$ cp mytest.ko runninglinuxkernel_4.0/kmodules
```

```
rlk@ubuntu:lab1_simple_module$ cp mytest.ko /home/rbk/rbk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab1_simple_module$
```

运行 `run.sh` 脚本。`runninglinuxkernel` 在这个版本中支持 Linux 主机和 QEMU 虚拟机的文件共享。

```
$ cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
$ ./run.sh arm32
```

启动 QEMU 虚拟机之后，首先检查/`mnt` 目录是否有 `mytest.ko` 文件。

```
/ # cd /mnt/
/mnt # ls
README      mytest.ko
/mnt #
```

```
/ #
/ # cd /mnt/
/mnt # ls
README      mytest.ko
/mnt #
```

使用 `insmod` 命令加载内核模块。

```
/mnt # insmod mytest.ko
my first kernel module init
```

```
| /mnt #
```

```
'mnt '
/mnt # insmod mytest.ko
[ 733.326762] my first kernel module init
/mnt #
/mnt #
```

通过 “/proc/modules” 节点可以查看当前系统加载的内核模块。

```
/mnt # cat /proc/modules
mytest 584 0 - Live 0xbff00000 (0)
/mnt #
/mnt #
/mnt #
```

通过 rmmod 命令来卸载模块。

```
/mnt # rmmod mytest.ko
[ 939.959463] goodbye
/mnt #
```

3 实验代码解析

```
1 #include <linux/module.h>
2 #include <linux/init.h>
3
4 static int __init my_test_init(void)
5 {
6     printk(KERN_EMERG "my first kernel module init\n");
7     return 0;
8 }
9
10static void __exit my_test_exit(void)
11{
12    printk("goodbye\n");
13}
14
15module_init(my_test_init);
16module_exit(my_test_exit);
17
18MODULE_LICENSE("GPL");
19MODULE_AUTHOR("Ben Shushu");
20MODULE_DESCRIPTION("my test kernel module");
21MODULE_ALIAS("mytest");
```

代码分析见书上第 4.1 章内容。

4.2 实验 2：向内核模块传递参数

1. 实验目的

学会如何向内核模块传递参数。

4.3 实验 3：在模块之间导出符号

2. 实验步骤

编写一个内核模块，通过模块参数的方式向内核模块传递参数。

实验步骤和实验 1 类似，就不在详细说明。

实验代码：

```

1 #include <linux/module.h>
2 #include <linux/init.h>
3
4 static int debug = 1;
5 module_param(debug, int, 0644);
6 MODULE_PARM_DESC(debug, "enable debugging information");
7
8 #define printk(args...) \
9     if (debug) { \
10         printk(KERN_DEBUG args); \
11     }
12
13static int mytest = 100;
14module_param(mytest, int, 0644);
15MODULE_PARM_DESC(mytest, "test for module parameter");
16
17static int __init my_test_init(void)
18{
19    printk("my first kernel module init\n");
20    printk("module parameter=%d\n", mytest);
21    return 0;
22}
23
24static void __exit my_test_exit(void)
25{
26    printk("goodbye\n");
27}
28
29module_init(my_test_init);
30module_exit(my_test_exit);
31
32MODULE_LICENSE("GPL");
33MODULE_AUTHOR("Ben Shushu");
34MODULE_DESCRIPTION("my test kernel module");
35MODULE_ALIAS("mytest");

```

实验代码分析见书上第 4.2 章内容。

4.3 实验 3：在模块之间导出符号

1. 实验目的

- 1) 学会如何在模块之间导出符号。
- 2) 在设计模块时考虑其层次结构。

4.4 实验 4：在优麒麟系统中编译内核模块（新增）

1. 实验目的

了解和熟悉如何在 Host 主机上编译和运行一个内核模块。

2. 实验步骤

很多企业采用 Ubuntu 或者 Centos 系统来作为基础系统来开发成品，比如服务器厂商或者其他工业控制的厂商，因此我们有必要知道如何在 Ubuntu 和 Centos 系统中编译内核模块。

我们采用本章实验 1 的参考代码为例。

```
cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_4/lab1_
simple_module
```

因为我们这次要编译的内核是优麒麟 Linux 主机上的，因此我们需要使用优麒麟 Linux 主机上的内核头文件，而不能使用 runnlinuxkernel 中的头文件里。我们需要动手来修改 Makefile 文件。

我们把第 4 行代码注释掉，然后打开第 5 行代码，也就是 BASEINCLUDE 这个变量指向优麒麟 Linux 主机上的内核头文件，如图所示。

```
1 #ARCH = arm
2 #CROSS_COMPILE = arm-linux-gnueabi-
3
4 #BASEINCLUDE ?= /home/ben/work/runninglinuxkernel_4.0
5 #BASEINCLUDE ?= /lib/modules/`uname -r`/build
6
7 mytest-objs := my_test.o
8
9 obj-m := mytest.o
10 all :
11     $(MAKE) -C $(BASEINCLUDE) M=$(PWD) modules;
12
13 clean:
14     $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
15     rm -f *.ko;
16
```

其中 “uname -a” 命令是查看当前系统的内核版本。

```
root@ubuntu:lab1_simple_module# uname -a
Linux ubuntu 4.15.0-29-generic #31-Ubuntu SMP Tue Jul 17 15:39:52 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
root@ubuntu:lab1_simple_module#
```

表示当前优麒麟 Linux 系统的内核版本为：4.15.0-29-generic。

那么该版本内核对应的文件是在哪里呢？

它对应的目录是在：/lib/moudels/4.15.0-29-generic 目录下面，在这个目录下面有一个 build 链接文件，指向内核头文件目录为/usr/src/linux-headers-4.15.0-29-generic。

4.4 实验 4：在优麒麟系统中编译内核模块（新增）

```
r lk@ubuntu:4.15.0-29-generic$ ls -l
total 5280
lrwxrwxrwx 1 root root 40 Feb 17 2019 build -> /usr/src/linux-headers-4.15.0-29-generic
drwxr-xr-x 2 root root 4096 Jul 17 2018 initrd
drwxr-xr-x 15 root root 4096 Jul 24 2018 kernel
-rw-r--r-- 1 root root 1262760 Feb 17 2019 modules.alias
-rw-r--r-- 1 root root 1243311 Feb 17 2019 modules.alias.bin
-rw-r--r-- 1 root root 7594 Jul 17 2018 modules.builtin
-rw-r--r-- 1 root root 9600 Feb 17 2019 modules.builtin.bin
-rw-r--r-- 1 root root 552117 Feb 17 2019 modules.dep
-rw-r--r-- 1 root root 780401 Feb 17 2019 modules.dep.bin
-rw-r--r-- 1 root root 317 Feb 17 2019 modules.devname
-rw-r--r-- 1 root root 206162 Jul 17 2018 modules.order
-rw-r--r-- 1 root root 540 Feb 17 2019 modules.softdep
-rw-r--r-- 1 root root 589722 Feb 17 2019 modules.symbols
-rw-r--r-- 1 root root 719555 Feb 17 2019 modules.symbols.bin
drwxr-xr-x 3 root root 4096 Jul 24 2018 vmlinuz
r lk@ubuntu:4.15.0-29-generic$
```

接下来开始编译内核模块了。

注意，编译 host 主机内核模块我们需要 root 用户，而编译 runninglinuxkernel_4.0 的内核模块不需要 root 用户，建议使用 r lk 用户。

```
$ sudo su //进入root用户
```

```
#cd
/home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module

#make
```

```
root@ubuntu:lab1_simple_module# make
make -C /lib/modules/`uname -r`/build M=/home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module modules;
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-29-generic'
Makefile:976: "Cannot use CONFIG_STACK_VALIDATION=y, please install libelf-dev, libelf-devel or elfutils-libelf-devel"
CC [M] /home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module/my_test.o
LD [M] /home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module/mytest.o
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module/mytest.mod.o
LD [M] /home/r lk/r lk_basic/runninglinuxkernel_4.0/r lk_lab/r lk_basic/chapter_4/lab1_simple_module/mytest.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-29-generic'
root@ubuntu:lab1_simple_module#
```

编译完成，我们需要使用 file 命令来检查，确保编译成了 x86_64 的格式了。

```
root@ubuntu:lab1_simple_module# file mytest.ko
mytest.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), BuildID[sha1]=d59709a133af3a2b1c3c9b75df68b3459bbf5c7a, not stripped
root@ubuntu:lab1_simple_module#
```

然后使用 insmod 命令加载内核模块。

```
#insmod mytest.ko
```

使用 dmesg 命令来查看内核 log 信息。

```
[662434.779588] mytest: loading out-of-tree module taints kernel.
[662434.779889] mytest: module verification failed: signature and/or required key missing - tainting
kernel
[662434.782692] my first kernel module init
root@ubuntu:lab1_simple_module#
```

上面有 3 行 log：

第 1 行：表示这是内核代码树之外的一个内核模块。

第 2 行：我们内核模块没有签名，但是不影响使用。

若想去掉上述两个打印，可以在 Makefile 中增加一行“CONFIG_MODULE_SIG=n”。

```
#ARCH = arm
#CROSS_COMPILE = arm-linux-gnueabi-
BASEINCLUDE ?= /home/ben/work/runninglinuxkernel_4.0
BASEINCLUDE ?= /lib/modules/`uname -r`/build
CONFIG_MODULE_SIG=n
mytest-objs := my_test.o
obj-m := mytest.o
all :
    $(MAKE) -C $(BASEINCLUDE) M=$(PWD) modules;
clean:
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
    rm -f *.ko;
```

第 3 行：就是我们这个内核模块打印的。

第 5 章

简单的字符设备驱动

5.1 实验 1：从一个简单的字符设备开始

5.1 实验 1：从一个简单的字符设备开始

1. 实验目的

- 1) 编写一个简单的字符设备驱动，实现基本的 open、read 和 write 方法。
- 2) 编写相应的用户空间测试程序，要求测试程序调用 read() 函数，并能看到对应的驱动程序执行了相应的 read 方法。

2. 实验详解

在详细介绍字符设备驱动架构之前，我们先用一个简单的设备驱动来“热身”。

<一个简单的字符设备驱动例子 my_demo.c>

```

0   #include <linux/module.h>
1   #include <linux/fs.h>
2   #include <linux/uaccess.h>
3   #include <linux/init.h>
4   #include <linux/cdev.h>
5
6   #define DEMO_NAME "my_demo_dev"
7   static dev_t dev;
8   static struct cdev *demo_cdev;
9   static signed count = 1;
10
11  static int demodrv_open(struct inode *inode, struct file *file)
12  {
13      int major = MAJOR(inode->i_rdev);
14      int minor = MINOR(inode->i_rdev);
15
16      printk("%s: major=%d, minor=%d\n", __func__, major, minor);
17
18      return 0;
19  }
20
21  static int demodrv_release(struct inode *inode, struct file *file)
22  {
23      return 0;
24  }
25
26  static ssize_t
27  demodrv_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos)
28  {
29      printk("%s enter\n", __func__);
30      return 0;
31  }
32
33  static ssize_t
34  demodrv_write(struct file *file, const char __user *buf, size_t count,
35                 loff_t *f_pos)
36  {
37      printk("%s enter\n", __func__);
38      return 0;

```

奔跑吧 linux 社区出品

```

38
39 }
40
41 static const struct file_operations demodrv_fops = {
42     .owner = THIS_MODULE,
43     .open = demodrv_open,
44     .release = demodrv_release,
45     .read = demodrv_read,
46     .write = demodrv_write
47 };
48
49
50 static int __init simple_char_init(void)
51 {
52     int ret;
53
54     ret = alloc_chrdev_region(&dev, 0, count, DEMO_NAME);
55     if (ret) {
56         printk("failed to allocate char device region");
57         return ret;
58     }
59
60     demo_cdev = cdev_alloc();
61     if (!demo_cdev) {
62         printk("cdev_alloc failed\n");
63         goto unregister_chrdev;
64     }
65
66     cdev_init(demo_cdev, &demodrv_fops);
67
68     ret = cdev_add(demo_cdev, dev, count);
69     if (ret) {
70         printk("cdev_add failed\n");
71         goto cdev_fail;
72     }
73
74     printk("succeeded register char device: %s\n", DEMO_NAME);
75     printk("Major number = %d, minor number = %d\n",
76           MAJOR(dev), MINOR(dev));
77
78     return 0;
79
80 cdev_fail:
81     cdev_del(demo_cdev);
82 unregister_chrdev:
83     unregister_chrdev_region(dev, count);
84
85     return ret;
86 }
87
88 static void __exit simple_char_exit(void)
89 {
90     printk("removing device\n");
91
92     if (demo_cdev)
93         cdev_del(demo_cdev);
94
95     unregister_chrdev_region(dev, count);
96 }
```

5.1 实验 1：从一个简单的字符设备开始

```

97
98     module_init(simple_char_init);
99     module_exit(simple_char_exit);
100
101    MODULE_AUTHOR("Benshushu");
102    MODULE_LICENSE("GPL v2");
103    MODULE_DESCRIPTION("simple character device");

```

上述内容是一个简单的字符设备驱动的例子，它只有字符设备驱动的框架，并没有什么实际的意义。但是对于刚入门的读者来说，这确实是一个很好的学习例子，因为字符设备驱动中绝大多数的 API 接口都呈现在了这个例子中。

下面先看如何编译它。

Makefile文件：

```

BASEINCLUDE ?= /home/rnk/rnk_basic/runninglinuxkernel_4.0
mydemo-objs := my_demodrv.o

obj-m := mydemo.o
all :
    $(MAKE) -C $(BASEINCLUDE) M=$(PWD) modules;

clean:
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
    rm -f *.ko;

```

本实验的参考代码是在

/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver

本实验以 ARM Vexpress 平台为例。

首先编译内核。

```

$ cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make vexpress_defconfig
$ make -j4

```

接着编译内核模块。

```

$ cd rnk_lab/rnk_basic/chapter_5/lab1_simple_driver
$ make

```

```

[rnk@ubuntu:lab1_simple_driver]$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver/simple_char.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver/mydemo.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver/mydemo.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver/mydemo.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
[rnk@ubuntu:lab1_simple_driver]$ pwd
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab1_simple_driver
[rnk@ubuntu:lab1_simple_driver]$

```

内核模块 mydemo.ko 生成之后，可以将其复制到 runninglinuxkernel_4.0/kmodules 目录下面，然后启动 QEMU 虚拟机。

```
$ cp mydemo.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
$ cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
$ ./run.sh arm32
```

使用 insmod 命令来加载 mydemo.ko 内核模块。

```
/mnt # insmod mydemo.ko
succeeded register char device: my_demo_dev
Major number = 252, minor number = 0
```

```
/mnt # insmod mydemo.ko
[ 2095.376620] succeeded register char device: my_demo_dev
[ 2095.379026] Major number = 252, minor number = 0
mnt #
```

可以看到，内核模块在初始化时输出了两行结果语句，这正是上述字符设备驱动例子中第 74~76 行的代码语句所要输出的。系统为这个设备分配了主设备号为 252，以及次设备号为 0。查看/proc/devices 这个 proc 虚拟文件系统中的 devices 节点信息，看到生成了名称为“my_demo_dev”的设备，主设备号为 252。

```
/mnt # cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 ttys
 4 /dev/vc/0
 4 tty
 5 /dev/ttys
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
14 sound
29 fb
90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
204 ttymA
252 my_demo_dev
253 usbmmon
254 rtc
```

5.1 实验 1：从一个简单的字符设备开始

```
/mnt # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
10 misc
13 input
14 sound
29 fb
90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
204 my_demo_dev
253 usbmon
254 rtc
```

接下来，设计一个用户空间的测试程序，实现操控这个字符设备驱动。

```
<简单测试程序 test.c>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define DEMO_DEV_NAME "/dev/demo_drv"

int main()
{
    char buffer[64];
    int fd;

    fd = open(DEMO_DEV_NAME, O_RDONLY);
    if (fd < 0) {
        printf("open device %s failed\n", DEMO_DEV_NAME);
        return -1;
    }

    read(fd, buffer, 64);
    close(fd);

    return 0;
}
```

test 测试程序很简单，打开这个“/dev/demo_drv”设备，调用一次读函数 read()，然后就关闭了设备。

接下来使用 arm-linux-gnueabi-gcc 交叉编译工具把它编译成 ARM32 架构的应用程序。**注意，这里使用“-static”把程序进行静态编译和链接。**

```
# arm-linux-gnueabi-gcc test.c -o test --static
```

```
rlk@ubuntu:lab1_simple_driver$ arm-linux-gnueabi-gcc test.c -o test -static
rlk@ubuntu:lab1_simple_driver$
```

把编译好的 test 程序复制到 kmodules 目录下，然后回到 QEMU 虚拟机，在/mnt 目录里已经可以看到 test 程序，现在可以运行该程序了。

这时候我们该可以运行这个 test 程序了吧。我们来运行一下。

```
/mnt # ./test
open device /dev/demo_drv failed
/mnt #
/mnt #
/mnt #
```

这时候我们发现 test 程序运行发生了错误。打印“open device /dev/demo_drv failed”的错误，这句话是从 test.c 文件打印的。我们查看源代码发现，是调用 open 函数没成功，那是为什么呢？难道我们的驱动写的不对？

通过 cat /proc/devices 看到我们的设备的主设备号是 252 号，名称为 my_demo_dev，我们先到 /dev 目录看看有没有？

```
/mnt #
/mnt # ls -l /dev/
total 0
crw-rw---- 1 0 0 14, 4 Aug 24 11:37 audio
crw-rw---- 1 0 0 5, 1 Aug 24 11:37 console
crw-rw---- 1 0 0 10, 63 Aug 24 11:37 cpu_dma_latency
crw-rw---- 1 0 0 14, 3 Aug 24 11:37 dsp
crw-rw---- 1 0 0 29, 0 Aug 24 11:37 fb0
crw-rw---- 1 0 0 29, 1 Aug 24 11:37 fb1
crw-rw---- 1 0 0 1, 7 Aug 24 11:37 full
crw-rw---- 1 0 0 10, 183 Aug 24 11:37 hwrng
drwxr-xr-x 2 0 0 120 Aug 24 11:37 input
crw-rw---- 1 0 0 1, 2 Aug 24 11:37 kmem
crw-rw---- 1 0 0 1, 11 Aug 24 11:37 kmsq
crw-rw---- 1 0 0 1, 1 Aug 24 11:37 mem
crw-rw---- 1 0 0 10, 60 Aug 24 11:37 memory_bandwidth
crw-rw---- 1 0 0 14, 0 Aug 24 11:37 mixer
crw-rw---- 1 0 0 90, 0 Aug 24 11:37 mtd0
crw-rw---- 1 0 0 90, 1 Aug 24 11:37 mtd0ro
crw-rw---- 1 0 0 90, 2 Aug 24 11:37 mtd1
crw-rw---- 1 0 0 90, 3 Aug 24 11:37 mtd1ro
brw-rw---- 1 0 0 31, 0 Aug 24 11:37 mtdblock0
brw-rw---- 1 0 0 31, 1 Aug 24 11:37 mtdblock1
crw-rw---- 1 0 0 10, 62 Aug 24 11:37 network_latency
crw-rw---- 1 0 0 10, 61 Aug 24 11:37 network_throughput
```

我们发现的确没有。

生成的设备需要在 /dev/ 目录下面生成对应的节点，这只能手动生成了。

```
/ # mknod /dev/demo_drv c 252 0
```

生成之后可以通过 “ls -l” 命令查看 /dev/ 目录的情况。

```
/dev # ls -l
total 0
crw-rw---- 1 0 0 14, 4 May 18 11:34 audio
crw-rw---- 1 0 0 5, 1 May 18 11:34 console
crw-rw---- 1 0 0 10, 63 May 18 11:34 cpu_dma_latency
crw-rw---- 1 0 0 252, 0 May 18 14:14 demo_drv
crw-rw---- 1 0 0 14, 3 May 18 11:34 dsp
crw-rw---- 1 0 0 29, 0 May 18 11:34 fb0
crw-rw---- 1 0 0 29, 1 May 18 11:34 fb1
crw-rw---- 1 0 0 1, 7 May 18 11:34 full
```

5.2 实验 2：使用 misc 机制来创建设备

```
/mnt # ls -l /dev
total 0
crw-rw---- 1 0 0 14, 4 Aug 24 11:37 audio
crw-rw---- 1 0 0 5, 1 Aug 24 11:37 console
crw-rw---- 1 0 0 10, 63 Aug 24 11:37 cpu_dma_latency
crw-r--r-- 1 0 0 252, 0 Aug 24 12:21 demo_drv
crw-rw---- 1 0 0 14, 3 Aug 24 11:37 dsp
crw-rw---- 1 0 0 29, 0 Aug 24 11:37 fb0
crw-rw---- 1 0 0 29, 1 Aug 24 11:37 fb1
crw-rw---- 1 0 0 1, 7 Aug 24 11:37 full
crw-rw---- 1 0 0 10, 183 Aug 24 11:37 hwrng
drwxr-xr-x 2 0 0 120 Aug 24 11:37 input
crw-rw---- 1 0 0 1, 2 Aug 24 11:37 kmem
crw-rw---- 1 0 0 1, 11 Aug 24 11:37 kmsg
crw-rw---- 1 0 0 1, 1 Aug 24 11:37 mem
```

上述内容已经完成了和内核相关的事情。

```
/mnt # ./test
demodrv_open: major=252, minor=0
demodrv_read enter
```

```
/mnt #
/mnt # ./test
[ 2738.230905] demodrv_open: major=252, minor=0
[ 2738.233923] demodrv_read enter
/mnt #
/mnt #
```

可以看到，日志里有 `demodrv_open()` 和 `demodrv_read()` 函数的输出语句，和驱动源代码里预期的是一样的，说明 `test` 应用程序已经成功操控了 `mydemo` 驱动程序，并完成了一次成功的交互。

5.2 实验 2：使用 misc 机制来创建设备

1. 实验目的

学会使用 misc 机制创建设备驱动。

2. 实验详解

misc device 称为杂项设备，Linux 内核把一些不符合预先确定的字符设备划分为杂项设备，这类设备的主设备号是 10。Linux 内核使用 `struct miscdevice` 数据结构描述这类设备。

```
<include/linux/miscdevice.h>

struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    umode_t mode;
};
```

内核提供了注册杂项设备的两个接口函数，驱动程序采用 `misc_register()` 函数来注册。它会自动创建设备节点，不需要使用 `mknod` 命令手工创建设备节点，因此使用 `misc` 机制来创建字符设备驱动是比较方便、简捷的。

```
int misc_register(struct miscdevice *misc);
int misc_deregister(struct miscdevice *misc);
```

接下来把 5.1 节中实验 1 的代码修改成采用 `misc` 机制注册字符驱动。

```
#include <linux/miscdevice.h>

#define DEMO_NAME "my_demo_dev"
static struct device *mydemodrv_device;

static struct miscdevice mydemodrv_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEMO_NAME,
    .fops = &demodrv_fops,
};

static int __init simple_char_init(void)
{
    int ret;

    ret = misc_register(&mydemodrv_misc_device);
    if (ret) {
        printk("failed register misc device\n");
        return ret;
    }

    mydemodrv_device = mydemodrv_misc_device.this_device;
    printk("succeeded register char device: %s\n", DEMO_NAME);

    return 0;
}

static void __exit simple_char_exit(void)
{
    printk("removing device\n");

    misc_deregister(&mydemodrv_misc_device);
}
```

编译成内核模块。

```
rlk@ubuntu:lab2_misc_driver$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0 M=/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab2_misc_driver modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab2_misc_driver/mydemodrv_misc.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab2_misc_driver/mydemo_misc.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab2_misc_driver/mydemo_misc.mod.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab2_misc_driver/mydemo_misc.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab2_misc_driver$
```

并在 QEMU 虚拟机上装载模块。

5.2 实验 2：使用 misc 机制来创建设备

```
/mnt # insmod mydemo_misc.ko
succeeded register char device: my_demo_dev
```

```
/mnt # insmod mydemo_misc.ko
[ 2974.331316] succeeded register char device: my_demo_dev
/mnt #
/mnt #
```

查看/dev 目录，发现设备节点已经创建好了，其中主设备号是 10，次设备号是动态分配的。

```
/mnt # ls -l /dev/
total 0
crw-rw---- 1 0 0 14, 4 May 19 06:48 audio
crw-rw---- 1 0 0 10, 58 May 19 06:48 my_demo_dev
```

```
/mnt # ls -l /dev/
total 0
crw-rw---- 1 0 0 14, 4 Aug 24 11:37 audio
crw-rw---- 1 0 0 5, 1 Aug 24 11:37 console
crw-rw---- 1 0 0 10, 63 Aug 24 11:37 cpu_dma_latency
crw-r--r-- 1 0 0 252, 0 Aug 24 12:21 demo_drv
crw-rw---- 1 0 0 14, 3 Aug 24 11:37 dsp
crw-rw---- 1 0 0 29, 0 Aug 24 11:37 fb0
crw-rw---- 1 0 0 29, 1 Aug 24 11:37 fb1
crw-rw---- 1 0 0 1, 7 Aug 24 11:37 full
crw-rw---- 1 0 0 10, 183 Aug 24 11:37 hwrng
drwxr-xr-x 2 0 0 120 Aug 24 11:37 input
crw-rw---- 1 0 0 1, 2 Aug 24 11:37 kmem
crw-rw---- 1 0 0 1, 11 Aug 24 11:37 kms
crw-rw---- 1 0 0 1, 1 Aug 24 11:37 mem
crw-rw---- 1 0 0 10, 60 Aug 24 11:37 memory_bandwidth
crw-rw---- 1 0 0 14, 0 Aug 24 11:37 mixer
crw-rw---- 1 0 0 90, 0 Aug 24 11:37 mtd0
crw-rw---- 1 0 0 90, 1 Aug 24 11:37 mtd0ro
crw-rw---- 1 0 0 90, 2 Aug 24 11:37 mtd1
crw-rw---- 1 0 0 90, 3 Aug 24 11:37 mtd1ro
brw-rw---- 1 0 0 31, 0 Aug 24 11:37 mtdblock0
brw-rw---- 1 0 0 31, 1 Aug 24 11:37 mtdblock1
crw-rw---- 1 0 0 10, 58 Aug 24 12:26 my_demo_dev
crw-rw---- 1 0 0 10, 62 Aug 24 11:37 network_latency
crw-rw---- 1 0 0 10, 61 Aug 24 11:37 network_throughput
```

接下来编写一个简单 test 测试程序。

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4
5 #define DEMO_DEV_NAME "/dev/my_demo_dev"
6
7 int main()
8 {
9     char buffer[64];
10    int fd;
11
12    fd = open(DEMO_DEV_NAME, O_RDONLY);
13    if (fd < 0) {
14        printf("open device %s failed\n", DEMO_DEV_NAME);
15        return -1;
16    }
17
18    read(fd, buffer, 64);
19    close(fd);
20
21    return 0;
22}
```

上述 test 程序程序比较简单，首先使用 open 函数打开设备文件 “/dev/my_demo_dev”，然后调用 read 函数读 64 个字节到 buffer 中，最后关闭设备文件。

交叉编译这个 test 程序。

```
| # arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab2_misc_driver$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab2_misc_driver$
```

把编译好的 test 程序复制到 kmodules 目录下，然后回到 QEMU 虚拟机，在/mnt 目录里已经可以看到 test 程序，现在可以运行该程序了。

这时候我们该可以运行这个 test 程序了吧。我们来运行一下。

```
| /mnt # ./test
demodrv_open: major=10, minor=58
demodrv_read enter
```

```
/mnt # ./test
[ 3394.990348] demodrv_open: major=10, minor=58
[ 3394.994372] demodrv_read enter
/mnt #
/mnt #
/mnt #
```

3 进阶思考

- 大家可以思考一下，为什么 misc 机制会自动创建设备节点，而实验 1 使用的方法就不行？
- 请阅读内核代码，看看有哪些你熟悉的设备驱动是使用 misc 机制来创建的？

5.3 实验 3：为虚拟设备编写驱动

1. 实验目的

- 通过一个虚拟设备，学习如何实现一个字符设备驱动程序的读写函数。
- 在用户空间编写测试程序来检验读写函数是否成功。

2. 实验详解

根据这个虚拟设备的需求，给实验 2 的代码添加 read() 和 write() 函数的实现，代码片段如下。

```
| 虚拟FIFO设备的缓冲区 */
static char *device_buffer;
#define MAX_DEVICE_BUFFER_SIZE 64
```

5.3 实验 3：为虚拟设备编写驱动

```

static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    int actual_readed;
    int max_free;
    int need_read;
    int ret;

    max_free = MAX_DEVICE_BUFFER_SIZE - *ppos;
    need_read = max_free > count ? lbuf : max_free;
    if (need_read == 0)
        dev_warn(mydemodrv_device, "no space for read");

    ret = copy_to_user(buf, device_buffer + *ppos, need_read);
    if (ret == need_read)
        return -EFAULT;

    actual_readed = need_read - ret;
    *ppos += actual_readed;

    printk("%s, actual_readed=%d, pos=%d\n", __func__, actual_readed, *ppos);
    return actual_readed;
}

static ssize_t
demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
    int actual_write;
    int free;
    int need_write;
    int ret;

    free = MAX_DEVICE_BUFFER_SIZE - *ppos;
    need_write = free > count ? count : free;
    if (need_write == 0)
        dev_warn(mydemodrv_device, "no space for write");

    ret = copy_from_user(device_buffer + *ppos, buf, need_write);
    if (ret == need_write)
        return -EFAULT;

    actual_write = need_write - ret;
    *ppos += actual_write;
    printk("%s: actual_write=%d, ppos=%d\n", __func__, actual_write, *ppos);
    return actual_write;
}

```

`demodrv_read()`函数有 4 个参数。`file` 表示打开的设备文件；`buf` 表示用户空间的内存起始地址，注意这里使用`_user`来提醒驱动开发者这个地址空间是属于用户空间的；`count` 表示用户想读取多少字节的数据；`ppos` 表示文件的位置指针。

`max_free` 表示当前设备的 FIFO 还剩下多少空间，`need_read` 根据 `max_free` 和 `count` 两个值做判断，防止数据溢出。接下来，通过 `copy_to_user()` 函数把设备 FIFO 的内容复制到用户进程的缓冲区中，注意这里是从设备 FIFO（`device_buffer`）的 `ppos` 开始的地方复制数据的。`copy_to_user()` 函数返回 0 表示复制成功，返回 `need_read` 表示复制失败。最后，需要更新 `ppos` 指针，然后返回实际复制的字节数到用户空间。

`demodrv_write()` 函数实现写功能，原理和上述 `demodrv_read()` 函数类似，只不过

其中使用了 copy_from_user() 函数。

编译内核模块。

```
rlk@ubuntu:lab3_mydemo_dev$ make
make -C /home/rkk/rkk_basic/runninglinuxkernel_4.0 M=/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev modules;
make[1]: Entering directory '/home/rkk/rkk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemodrv_msc.o
/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemodrv_msc.c: In function 'demodrv_read':
/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemodrv_msc.c:53:9: warning: format '%d' expects argument of type 'int', but argument 4 has type 'loff_t {aka long long int}' [-Wformat=]
      printk("%s, actual_readed=%d, pos=%d\n", __func__, actual_readed, *ppos);
^
/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemodrv_msc.c: In function 'demodrv_write':
/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemodrv_msc.c:79:9: warning: format '%d' expects argument of type 'int', but argument 4 has type 'loff_t {aka long long int}' [-Wformat=]
      printk("%s: actual_write =%d, ppos=%d\n", __func__, actual_write, *ppos);
^
  LD [M]  /home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemo_msc.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/mydemo_msc.mod.o
  LD [M]  /home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_5/lab3_mydemo_dev/
```

我们看到，给出的参考代码里有编译警告，请小伙伴们自行修复这些编译警告。

把编译好的内核模块拷贝到 kmodules 目录中。

```
rlk@ubuntu:lab3_mydemo_dev$ cp mydemo_msc.ko /home/rkk/rkk_basic/runninglinuxkernel_4.0/kmodules/
mydemo.ko  mydemo_msc.ko  mytest.ko      README      test
rlk@ubuntu:lab3_mydemo_dev$ cp mydemo_msc.ko /home/rkk/rkk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab3_mydemo_dev$
```

接下来写一个测试程序来检验上述驱动程序是否工作正常。

```
0  #include <stdio.h>
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  #define DEMO_DEV_NAME "/dev/my_demo_dev"
5
6  int main()
7  {
8      char buffer[64];
9      int fd;
10     int ret;
11     size_t len;
12     char message[] = "Testing the virtual FIFO device";
13     char *read_buffer;
14
15     len = sizeof(message);
16
17     fd = open(DEMO_DEV_NAME, O_RDWR);
18     if (fd < 0) {
19         printf("open device %s failed\n", DEMO_DEV_NAME);
20         return -1;
21     }
22
23     /*1. write the message to device*/
```

5.3 实验 3：为虚拟设备编写驱动

```

24     ret = write(fd, message, len);
25     if (ret != len) {
26         printf("canot write on device %d, ret=%d", fd, ret);
27         return -1;
28     }
29
30     read_buffer = malloc(2*len);
31     memset(read_buffer, 0, 2*len);
32
33     /*close the fd, and reopen it*/
34     close(fd);
35
36     fd = open(DEMO_DEV_NAME, O_RDWR);
37     if (fd < 0) {
38         printf("open device %s failed\n", DEMO_DEV_NAME);
39         return -1;
40     }
41
42     ret = read(fd, read_buffer, 2*len);
43     printf("read %d bytes\n", ret);
44     printf("read buffer=%s\n", read_buffer);
45
46     close(fd);
47
48     return 0;
49 }
```

测试程序逻辑很简单。首先使用 `open` 方法打开这个设备驱动，向设备里写入 `message` 字符串，然后关闭这个设备并重新打开这个设备，最后通过 `read()` 函数把 `message` 字符串读出来。

编译 `test` 程序，并拷贝到 `kmodules` 目录中。

```

rlk@ubuntu:lab3_mydemo_devs$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab3_mydemo_devs$ rlk@ubuntu:lab3_mydemo_devs$ cp test /home/rnk/rnk_basic/runninglinuxkernel 4.0/kmodules/
rlk@ubuntu:lab3_mydemo_devs$
```

启动 QEMU 虚拟机，并进入到 `/mnt` 目录下面。加载内核模块。

```

/mnt # ./test
demodrv_open: major=10, minor=58
demodrv_write: actual_write =32, ppos=0

demodrv_open: major=10, minor=58
demodrv_read, actual_readed=64, pos=0
read 64 bytes
read buffer=Testing the virtual FIFO device
/mnt #
```

```

/mnt # 
/mnt # insmod mydemo_misc.ko
[86155.358588] succeeded register char device: my_demo_dev
/mnt #
/mnt #
/mnt # ./test &
/mnt # [86160.527018] demodrv_open: major=10, minor=58
[86160.533164] demodrv_write enter
[86160.533645] demodrv_write: actual_write =32, ppos=0
[86160.536913] demodrv_open: major=10, minor=58
[86160.537542] demodrv_read enter
[86160.538014] demodrv_read, actual_readed=64, pos=0
read 64 bytes
read buffer=Testing the virtual FIFO device

[1]+ Done                  ./test
/mnt #

```

读者可以思考一下，为什么这里需要关闭设备后重新打开一次设备？如果不进行这样的操作，是否可以呢？

3 进阶思考

- 读者可以思考一下，为什么这里需要关闭设备后重新打开一次设备？如果不进行这样的操作，是否可以呢？
- 这个 lab，是最经典的字符设备读写操作的示例了，读者可以查阅现有的内核代码 drivers 目录，看看有哪些现成的字符设备驱动，学习一下，他们是如何实现读写函数的？

5.4 实验 4：使用 KFIFO 改进设备驱动

1. 实验目的

学会使用内核的 KFIFO 的环形缓冲区实现虚拟字符设备的读写函数。

2. 实验详解

我们在 5.4.1 节中实验 3 中的驱动代码里只是简单地把用户数据复制到设备的缓冲区中，并没有考虑到读和写的并行管理问题。因此在对应的测试程序中，需要重启设备后才能正确地将数据读出来。

这实际上是一个典型的“生产者和消费者”的问题，我们可以设计和实现一个环形缓冲区来解决这个问题。环形缓冲区通常有一个读指针和一个写指针，读指针指向环形缓冲区可读的数据，写指针指向环形缓冲区可写的数据。通过移动读指针和写指针来实现缓冲区的数据读取和写入。

Linux 内核实现了一个称为 KFIFO 的环形缓冲区的机制，它可以在一个读者线程和一个写者线程并发执行的场景下，无须使用额外的加锁来保证环形缓冲区的数据安全。KFIFO 提供的接口函数定义在 include/linux/kfifo.h 文件中。

```
#define DEFINE_KFIFO(fifo, type, size)
```

5.4 实验 4：使用 KFIFO 改进设备驱动

```
#define kfifo_from_user(fifo, from, len, copied)
#define kfifo_to_user(fifo, to, len, copied)
```

DEFINE_KFIFO()宏用来初始化一个环形缓冲区，其中参数 fifo 表示环形缓冲区的名字；type 表示缓冲区中数据的类型；size 表示缓冲区有多少个元素，元素的个数必须是 2 的整数次幂。

kfifo_from_user()宏用来将用户空间的数据写入环形缓冲区中，其中参数 fifo 表示使用哪个环形缓冲区；from 表示用户空间缓冲区的起始地址；len 表示要复制多少个元素；copied 保存了成功复制元素的数量，通常用作返回值。

kfifo_to_user()宏用来读出环形缓冲区的数据并且复制到用户空间中，参数作用和 kfifo_from_user() 宏类似。

下面是使用 KFIFO 机制实现该字符设备驱动的 read 和 write 函数的代码片段。

```
#include <linux/kfifo.h>

DEFINE_KFIFO(mydemo_fifo, char, 64);

static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    int actual_readed;
    int ret;

    ret = kfifo_to_user(&mydemo_fifo, buf, count, &actual_readed);
    if (ret)
        return -EIO;

    printk("%s, actual_readed=%d, pos=%lld\n", __func__, actual_readed,
*ppos);
    return actual_readed;
}

static ssize_t
demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t
*ppos)
{
    unsigned int actual_write;
    int ret;

    ret = kfifo_from_user(&mydemo_fifo, buf, count, &actual_write);
    if (ret)
        return -EIO;

    printk("%s: actual_write =%d, ppos=%lld\n", __func__, actual_write,
*ppos);

    return actual_write;
}
```

编译内核模块。

```
rlk@ubuntu:lab4_mydemo_kfifo$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/r
lk_lab/rnk_basic/chapter_5/lab4_mydemo_kfifo modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab4_mydemo_kfif
o/mydemodrv_misc.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab4_mydemo_kfif
o/mydemo_misc.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab4_mydemo_kfif
o/mydemo_misc.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab4_mydemo_kfif
o/mydemo_misc.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab4_mydemo_kfifo$
```

把编译好的内核模块拷贝到 kmodules 目录下。

```
rlk@ubuntu:lab4_mydemo_kfifo$ cp mydemo_misc.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodul
es/
rlk@ubuntu:lab4_mydemo_kfifo$
```

测试示例和 5.4.1 节中的实验 3 类似，只不过这里不需要关闭和重新打开设备。

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define DEMO_DEV_NAME "/dev/my_demo_dev"

int main()
{
    char buffer[64];
    int fd;
    int ret;
    size_t len;
    char message[] = "Testing the virtual FIFO device";
    char *read_buffer;

    len = sizeof(message);

    fd = open(DEMO_DEV_NAME, O_RDWR);
    if (fd < 0) {
        printf("open device %s failed\n", DEMO_DEV_NAME);
        return -1;
    }

    /*1. write the message to device*/
    ret = write(fd, message, len);
    if (ret != len) {
        printf("cannot write on device %d, ret=%d", fd, ret);
        return -1;
    }

    read_buffer = malloc(2*len);
    memset(read_buffer, 0, 2*len);

    ret = read(fd, read_buffer, 2*len);
    printf("read %d bytes\n", ret);
    printf("read buffer=%s\n", read_buffer);

    close(fd);

    return 0;
}
```

5.4 实验 4：使用 KFIFO 改进设备驱动

编译 test 程序。

```
rlk@ubuntu:lab4_mydemo_kfifo$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab4_mydemo_kfifo$
```

编译好内核模块和测试程序，然后将它们放到 QEMU 虚拟机上运行。

```
rlk@ubuntu:lab4_mydemo_kfifo$ cp test /home/rlk/rlk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab4_mydemo_kfifo$
```

运行 QEMU 虚拟机器。进入到/mnt 目录，加载内核模块。

```
/mnt # ./test
demodrv_open: major=10, minor=58
demodrv_write: actual_write =32, ppos=0
demodrv_read, actual_readed=32, pos=0
read 32 bytes
read buffer=Testing the virtual FIFO device
/mnt #
```

```
/mnt # insmod mydemo_misc.ko
[86613.107006] succeeded register char device: my_demo_dev
/mnt #
/mnt #
/mnt #
/mnt #
/mnt # ./test &
/mnt # [86617.610397] demodrv_open: major=10, minor=58
[86617.614258] demodrv_write: actual_write =32, ppos=0
[86617.616177] demodrv_read, actual_readed=32, pos=0
read 32 bytes
read buffer=Testing the virtual FIFO device
[1]+ Done ./test
/mnt #
```

从上面可以看到，我们的 test 程序已经通过 read 函数从这个虚拟设备到读取到数据了，该数据就是“Testing the virtual FIFO device”这个字符串。

“demodrv_open: major=10, minor=58”这句日志是驱动打印的，说明 test 程序已经打开了这个设备。

“demodrv_write: actual_write =32, ppos=0”，这句日志也是驱动打印的，说明 test 程序已经成功的往虚拟设备 FIFO 中写入了数据。见驱动代码的 demodrv_write()函数。

“demodrv_read, actual_readed=32, pos=0”这句日志也是驱动打印的，说明 test 程序已经成功地从虚拟设备 FIFO 中读取了数据，见驱动代码中 demodrv_read()函数。

“read 32 bytes”这句日志是 test 程序打印的，说明 test 程序成功从虚拟设备 FIFO 中读取了 32 个字节。

“read buffer=Testing the virtual FIFO device”这句是 test 程序打印的，把读取到的 32 个字节的内容打印出来。

还有一种更简便的方法来测试，即使用 echo 和 cat 命令直接操作设备文件。首先我们可以使用 echo 命令来写一个字符串到设备到中。

```
/mnt #
/mnt # echo "i am at shanghai now" > /dev/my_demo_dev
[87147.183855] demodrv_open: major=10, minor=58
[87147.192520] demodrv_write: actual_write =21, ppos=0
/mnt #
```

接下来，使用 cat 命令来读取设备文件。

```
/mnt # cat /dev/my_demo_dev
[87155.588838] demodrv_open: major=10, minor=58
[87155.590473] demodrv_read, actual_readed=21, pos=0
i am at shanghai now
[87155.592306] demodrv_read, actual_readed=0, pos=0
/mnt #
```

我们发现我们测试也成功了，而且测试很方便。另外读者可能发现了，echo 和 cat 命令都会打开和关闭我们的设备文件/dev/my_demo_dev。

3 进阶思考

细心的读者可能会发现，这个设备驱动的 KFIFO 环形缓存区的大小为 64 字节。如果使用 echo 命令发送一个长度大于 64 字节的字符串到这个设备，我们会发现终端中一直输出如下语句。

```
demodrv_write: actual_write =0, ppos=0
```

```
/mnt # echo "i am at shanghai now, how are you! i am fine, thank very much!!!!!
" > /dev/my_demo_dev
[87402.327110] demodrv open: major=10, minor=58
[87402.327996] demodrv write: actual_write =64, ppos=0
[87402.328494] demodrv write: actual_write =0, ppos=0
[87402.328938] demodrv write: actual_write =0, ppos=0
[87402.329677] demodrv write: actual_write =0, ppos=0
[87402.330092] demodrv write: actual_write =0, ppos=0
[87402.330538] demodrv write: actual_write =0, ppos=0
[87402.331443] demodrv write: actual_write =0, ppos=0
[87402.331859] demodrv write: actual_write =0, ppos=0
[87402.332210] demodrv write: actual_write =0, ppos=0
[87402.332594] demodrv write: actual_write =0, ppos=0
[87402.333133] demodrv write: actual_write =0, ppos=0
[87402.333545] demodrv write: actual_write =0, ppos=0
[87402.333904] demodrv write: actual_write =0, ppos=0
[87402.334386] demodrv write: actual_write =0, ppos=0
[87402.334951] demodrv write: actual_write =0, ppos=0
[87402.335304] demodrv write: actual_write =0, ppos=0
[87402.335550] demodrv write: actual_write =0, ppos=0
[87402.335779] demodrv write: actual_write =0, ppos=0
[87402.336002] demodrv write: actual_write =0, ppos=0
```

请读者思考一下如何解决这个问题。

5.5 实验 5：把虚拟设备驱动改成非阻塞模式

5.5 实验 5：把虚拟设备驱动改成非阻塞模式

1. 实验目的

学习如何在字符设备驱动中添加非阻塞 I/O 操作。

2. 实验详解

`open()` 函数有一个 `flags` 参数，这些标志位通常用来表示文件打开的属性。

- ❑ `O_RDONLY`: 只读打开。
- ❑ `O_WRONLY`: 只写打开。
- ❑ `O_RDWR`: 读写打开。
- ❑ `O_CREAT`: 若文件不存在，则创建它。

除此之外，还有一个称为 `O_NONBLOCK` 的标志位，用来设置访问文件的方式为非阻塞模式。

下面把 5.4.2 节中的实验 4 修改为非阻塞模式。

```
static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    int actual_readed;
    int ret;

    if (kfifo_is_empty(&mydemo_fifo)) {
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;
    }

    ret = kfifo_to_user(&mydemo_fifo, buf, count, &actual_readed);
    if (ret)
        return -EIO;

    printk("%s, actual_readed=%d, pos=%lld\n", __func__, actual_readed,
*ppos);
    return actual_readed;
}

static ssize_t
demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t
*ppos)
{
    unsigned int actual_write;
    int ret;

    if (kfifo_is_full(&mydemo_fifo)){
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;
    }

    ret = kfifo_from_user(&mydemo_fifo, buf, count, &actual_write);
    if (ret)
        return -EIO;

    printk("%s: actual_write =%d, ppos=%lld, ret=%d\n", __func__,

```

奔跑吧 linux 社区出品

```

    actual_write, *ppos, ret);

    return actual_write;
}

```

编译内核模块。

```

rlk@ubuntu:lab5_mydemodrv_nonblock$ make
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab5_mydemodrv_nonblock/mydemodrv_misc.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab5_mydemodrv_nonblock/mydemodrv_misc.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab5_mydemodrv_nonblock/mydemodrv_misc.mod.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab5_mydemodrv_nonblock/mydemodrv_misc.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab5_mydemodrv_nonblock$ 

```

拷贝内核模块到 kmodues 目录。

```

rlk@ubuntu:lab5_mydemodrv_nonblock$ cp mydemo_misc.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodues/
rlk@ubuntu:lab5_mydemodrv_nonblock$ 

```

下面是对应的测试程序。

```

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define DEMO_DEV_NAME "/dev/my_demo_dev"

int main()
{
    int fd;
    int ret;
    size_t len;
    char message[80] = "Testing the virtual FIFO device";
    char *read_buffer;

    len = sizeof(message);

    read_buffer = malloc(2*len);
    memset(read_buffer, 0, 2*len);

    fd = open(DEMO_DEV_NAME, O_RDWR | O_NONBLOCK);
    if (fd < 0) {
        printf("open device %s failed\n", DEMO_DEV_NAME);
        return -1;
    }

/*1. 先读取数据*/
    ret = read(fd, read_buffer, 2*len);
    printf("read %d bytes\n", ret);
    printf("read buffer=%s\n", read_buffer);

/*2. 将信息写入设备*/
    ret = write(fd, message, len);
    if (ret != len)
        printf("have write %d bytes\n", ret);

/*3. 再写入*/
}

```

5.5 实验 5：把虚拟设备驱动改成非阻塞模式

```

ret = write(fd, message, len);
if (ret != len)
    printf("have write %d bytes\n", ret);

/*4. 最后读取*/
ret = read(fd, read_buffer, 2*len);
printf("read %d bytes\n", ret);
printf("read buffer=%s\n", read_buffer);

close(fd);
return 0;
}

```

编译 test 程序，并拷贝到 kmodules 目录中。

```

rlk@ubuntu:lab5_mydemodrv_nonblock$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab5_mydemodrv_nonblock$ cp test /home/rbk/rbk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab5_mydemodrv_nonblock$ 

```

这次测试程序有如下的不同之处。

- 在打开设备之后，马上进行读操作，请读者想想结果如何？
- message 的大小设置为 80，比设备驱动里的环形缓冲区的大小要大，写操作会发生什么事情？
- 再写一次会发生什么情况？

启动 QEMU 虚拟机，并加载内核模块。先检查上一次实验的内核模块是否已经卸载。

```

/mnt # lsmod
mydemo_misc 1827 0 - Live 0xbff014000 (0)
/mnt # rmmod mydemo_misc.ko
[87790.937896] removing device
/mnt # 

```

加载本次实验的内核模块。

```

/mnt # insmod mydemo_misc.ko
[87866.280964] succeeded register char device: my_demo_dev
/mnt # 

```

下面是测试程序的运行结果。

```

/mnt # ./test
demodrv_open: major=10, minor=58
read -1 bytes
read buffer=

demodrv_write: actual_write =64, ppos=0, ret=0
have write 64 bytes
have write -1 bytes

demodrv_read, actual_readed=64, pos=0
read 64 bytes
read buffer=Testing the virtual FIFO device

```

```
/mnt # ./test
[87922.401933] demodrv_open: major=10, minor=58
read -1 bytes
read buffer=
[87922.431426] demodrv_write: actual_write =64, ppos=0, ret=0
have write 64 bytes
have write -1 bytes
[87922.435638] demodrv_read, actual_readed=64, pos=0
read 64 bytes
read buffer=Testing the virtual FIFO device
/mnt #
```

从运行结果可以看出，打开设备后马上进行读操作，结果是什么也读不到，read()函数返回-1，说明读操作发生了错误。当第二次进行写操作时，write()函数返回-1，说明写操作发生了错误。

“demodrv_open: major=10, minor=58”，这句 log 是驱动打印的，说明 test 程序已经成功打开了我们的设备驱动。

“read -1 bytes” 这句是 test 程序打印的，说明 test 程序调用 read 函数，没有成功读到数据，返回值为-1

“read buffer=” 这句是 test 程序打印的，test 程序没有成功读到数据，所以打印 buffer 的字符串就是空的。

“demodrv_write: actual_write =64, ppos=0, ret=0” 这句是驱动打印的，说明 test 程序成功调用 write 函数往虚拟设备 FIFO 中写入了数据。

“have write 64 bytes” 这句是 test 程序打印的，说明刚才成功写入了 64 个字节。

“have write -1 bytes”，这句也是 test 程序打印的。我们在成功写入了 64 个字节之后，再尝试去写，这时候写不进去了。因为驱动中的 FIFO buffer 已经满了。

“demodrv_read, actual_readed=64, pos=0”，这句是驱动打印的，说明 test 程序执行了一个 read 操作。

“read 64 bytes” 这句是 test 程序打印的，test 程序成功地把驱动中的 FIFO 数据读取到用户空间。

“read buffer=Testing the virtual FIFO device” 这句是 test 程序打印的，把刚才读取到用户空间的数据打印出来。

5.6 实验 6：把虚拟设备驱动改成阻塞模式

1. 实验目的

学习如何在字符设备驱动中添加阻塞 I/O 操作。

2. 实验详解

当用户进程通过 read()或者 write()函数去读写设备时，如果驱动程序无法立刻满足请求的资源，那么应该怎么响应呢？在 5.5.1 节中的实验 5 中，驱动程序返回-EAGAIN，这是非阻塞模式的行为。

5.6 实验 6：把虚拟设备驱动改成阻塞模式

但是，非阻塞模式对于大部分应用场景来说不太合适，因此大部分用户进程通过 `read()` 或者 `write()` 函数进行 I/O 操作时希望能返回有效数据或者把数据写入设备中，而不是返回一个错误值。这该怎么办？

- 1) 在非阻塞模式下，采用轮询的方式来不断读写数据。
- 2) 采用阻塞模式，当请求数据无法立刻满足时，让该进程睡眠直到数据准备好为止。

上面提到的进程睡眠是什么意思呢？进程在运行生命周期里有不同的状态。

- `TASK_RUNNING`（可运行态或者就绪态）。
- `TASK_INTERRUPTIBLE`（可中断睡眠态）。
- `TASK_UNINTERRUPTIBLE`（不可中断睡眠态）。
- `_TASK_STOPPED`（终止态）。
- `EXIT_ZOMBIE`（“僵尸”态）。

把一个进程设置成睡眠状态，那么就是把这个进程从 `TASK_RUNNING` 状态设置为 `TASK_INTERRUPTIBLE` 或者 `TASK_UNINTERRUPTIBLE` 状态，并且从进程调度器的运行队列中移走，我们称这个点为“睡眠点”。当请求的资源或者数据到达时，进程会被唤醒，然后从睡眠点重新执行。

在 Linux 内核中，采用一个称为等待队列（wait queue）的机制来实现进程阻塞操作。

(1) 等待队列头

等待队列定义了一个被称为等待队列头 (`wait_queue_head_t`) 的数据结构，定义在 `<linux/wait.h>` 中。

```
struct __wait_queue_head {
    spinlock_t      lock;
    struct list_head task_list;
};

typedef struct __wait_queue_head wait_queue_head_t;
```

可以通过如下方法静态定义并初始化一个等待队列头。

```
DECLARE_WAIT_QUEUE_HEAD(name)
```

或者使用动态的方式来初始化。

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

(2) 等待队列元素 `wait_queue_t`

```
struct __wait_queue {
    unsigned int      flags;
    void             *private;
    wait_queue_func_t func;
    struct list_head task_list;
};

typedef struct __wait_queue wait_queue_t;
```

等待队列元素使用 `wait_queue_t` 数据结构来描述。

(3) 睡眠等待

Linux 内核提供了简单的睡眠方式，并封装成 `wait_event()` 的宏以及其他几个扩展宏，主要功能是在让进程睡眠时也检查进程的唤醒条件。

```
wait_event(wq, condition)
wait_event_interruptible(wq, condition)
wait_event_timeout(wq, condition, timeout)
wait_event_interruptible_timeout(wq, condition, timeout)
```

`wq` 表示等待队列头。`condition` 是一个布尔表达式，在 `condition` 变为真之前，进程会保持睡眠状态。`timeout` 表示当 `timeout` 时间到达之后，进程会被唤醒，因此它只会等待限定的时间。当给定的时间到了之后，`wait_event_timeout()` 和 `wait_event_interruptible_timeout()` 这两个宏无论 `condition` 是否为真，都会返回 0。

`wait_event_interruptible()` 会让进程进入可中断睡眠状态，而 `wait_event()` 会让进程进入不可中断睡眠态，也就是说不受干扰，对信号不做任何反应，不可能发送 `SIGKILL` 信号使它停止，因为它们不响应信号。因此，一般驱动程序不会采用这个睡眠模式。

(4) 唤醒

```
wake_up(x)
wake_up_interruptible(x)
```

`wake_up()` 会唤醒等待队列中所有的进程。`wake_up()` 应该和 `wait_event()` 或者 `wait_event_timeout()` 配对使用，而 `wake_up_interruptible()` 应该和 `wait_event_interruptible()` 和 `wait_event_interruptible_timeout()` 配对使用。

本实验运用等待队列来完善虚拟设备的读写函数。

```
struct mydemo_device {
    const char *name;
    struct device *dev;
    struct miscdevice *miscdev;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
};

static int __init simple_char_init(void)
{
    int ret;

    ...
    init_waitqueue_head(&device->read_queue);
    init_waitqueue_head(&device->write_queue);

    return 0;
}

static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;
    int actual_readed;
    int ret;

    if (kfifo_is_empty(&mydemo_fifo)) {
```

5.6 实验 6：把虚拟设备驱动改成阻塞模式

```

if (file->f_flags & O_NONBLOCK)
    return -EAGAIN;

printk("%s: pid=%d, going to sleep\n", __func__, current->pid);
ret = wait_event_interruptible(device->read_queue,
                               !kfifo_is_empty(&mydemo_fifo));
if (ret)
    return ret;
}

ret = kfifo_to_user(&mydemo_fifo, buf, count, &actual_readed);
if (ret)
    return -EIO;

if (!kfifo_is_full(&mydemo_fifo))
    wake_up_interruptible(&device->write_queue);

printk("%s, pid=%d, actual_readed=%d, pos=%lld\n", __func__,
       current->pid, actual_readed, *ppos);
return actual_readed;
}

static ssize_t
demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t
*ppos)
{
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;

    unsigned int actual_write;
    int ret;

    if (kfifo_is_full(&mydemo_fifo)){
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;

        printk("%s: pid=%d, going to sleep\n", __func__, current->pid);
        ret = wait_event_interruptible(device->write_queue,
                                       !kfifo_is_full(&mydemo_fifo));
        if (ret)
            return ret;
    }

    ret = kfifo_from_user(&mydemo_fifo, buf, count, &actual_write);
    if (ret)
        return -EIO;

    if (!kfifo_is_empty(&mydemo_fifo))
        wake_up_interruptible(&device->read_queue);

    printk("%s: pid=%d, actual_write=%d, ppos=%lld, ret=%d\n", __func__,
           current->pid, actual_write, *ppos, ret);

    return actual_write;
}

```

主要的改动见上面代码加粗字体部分。

1) 定义两个等待队列，其中 `read_queue` 为读操作的等待队列，`write_queue` 为写操作的等待队列。

2) 在 `demodrv_read()` 读函数中，当 KFIFO 环形缓冲区为空时，说明没有数据可

以读，调用 `wait_event_interruptible()` 函数让用户进程进入睡眠状态，因此这个位置就是所谓的“睡眠点”了。那什么时候进程会被唤醒呢？当 KFIFO 环形缓冲区有数据可读时就会被唤醒。

3) 在 `demodrv_read()` 读函数中，当把数据从设备驱动的 KFIFO 读到用户空间的缓冲区之后，KFIFO 有剩余的空间可以让写者进程写数据到 KFIFO，因此调用 `wake_up_interruptible()` 去唤醒 `write_queue` 中所有睡眠等待的写者进程。

4) 写函数和读函数很类似，只是判断进程是否进入睡眠的条件不一样。对于读操作，当 KFIFO 没有数据时，进入睡眠；对于写操作，当 KFIFO 满了，则进入睡眠。

编译内核模块。

```
rlk@ubuntu:lab6_mydemodrv_blocks$ make
make -C /home/r1k/r1k_basic/runninglinuxkernel_4.0 M=/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block modules;
make[1]: Entering directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'
  CC [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.o
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.c: In function `simple_char_exit':
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.c:169:2: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-statement]
  struct mydemo_device *dev = mydemo_device;
^
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.mod.o
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_5/lab6_mydemodrv_block/mydemodrv_misc.ko
make[1]: Leaving directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab6_mydemodrv_blocks$
```

拷贝内核模块到 `kmodes` 目录中。

```
rlk@ubuntu:lab6_mydemodrv_blocks$ cp mydemo_misc.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab6_mydemodrv_blocks$
```

启动 QEMU 虚拟机，并加载内核模块。先检查上一次实验的内核模块是否已经卸载。

```
/mnt # lsmod
mydemo_misc 1827 0 - Live 0xbff014000 (0)
/mnt # rmmod mydemo_misc.ko
[87790.937896] removing device
/mnt #
```

加载本次实验的内核模块。

```
/mnt # insmod mydemo_misc.ko
[88692.674381] succeeded register char device: my_demo_dev
/mnt #
```

下面使用 `echo` 和 `cat` 命令来验证驱动程序。

首先用 `cat` 命令打开这个设备，然后让其在后台运行，“`&`” 符号表示让其在后台运行。

5.6 实验 6：把虚拟设备驱动改成阻塞模式

```
/mnt # cat /dev/my_demo_dev &
/mnt # [88847.255418] demodrv_open: major=10, minor=58
[88847.259163] demodrv_read: pid=861, going to sleep
```

注意：“&”符号表示让其在后台运行

从日志中，“demodrv_read: pid=861, going to sleep”这句 log 可以看到出，cat 命令会先打开设备，然后进入 demodrv_read()函数，因为这时 KFIFO 里面没有可读数据，所以读者进程（pid 为 861）进入睡眠状态。

使用 top 命令来查看一下。

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
862	773	0	R	2308	2.6	0	0.9	top
7	2	0	SW	0	0.0	2	0.0	[rcu_sched]
4	2	0	SW	0	0.0	0	0.0	[kworker/0:0]
3	2	0	RW	0	0.0	0	0.0	[ksoftirqd/0]
1	0	0	S	2308	2.6	3	0.0	{linuxrc} init
773	1	0	S	2308	2.6	0	0.0	-/bin/sh
861	773	0	S	2308	2.6	2	0.0	cat /dev/my_demo_dev
754	2	0	SWN	0	0.0	3	0.0	[kmemleak]
686	2	0	SW	0	0.0	1	0.0	[kworker/u8:2]
411	2	0	SW	0	0.0	1	0.0	[kworker/l:1]
409	2	0	SW	0	0.0	2	0.0	[kworker/2:1]
11	2	0	SW	0	0.0	1	0.0	[ksoftirqd/1]
328	2	0	SW	0	0.0	3	0.0	[kworker/3:1]
19	2	0	SW	0	0.0	3	0.0	[ksoftirqd/3]
15	2	0	SW	0	0.0	2	0.0	[ksoftirqd/2]
23	2	0	SW	0	0.0	3	0.0	[kdevtmpfs]
279	2	0	SW	0	0.0	1	0.0	[khungtaskd]
2	0	0	SW	0	0.0	3	0.0	[kthreadd]
10	2	0	SW	0	0.0	1	0.0	[migration/1]
18	2	0	SW	0	0.0	3	0.0	[migration/3]

我们发现刚才的“cat /dev/my_demo_dev”这个命令显示在 top 中，而且进程 PID 为 861。大家注意看进程的状态为“S”，说明进程处于睡眠状态。

我们在这里列出 Linux 进程的几种状态。

(1) R 运行状态 (Running): 并不意味着进程一定在运行中，也可以在运行队列里；

(2) S 睡眠状态 (Sleeping): 进程在等待事件完成；(浅度睡眠，可以被唤醒)

(3) D 不可中断的睡眠状态 (Uninterrupt sleep) :不可中断睡眠(深度睡眠，不可以被唤醒，通常在磁盘写入时发生)

(4) T 停止状态 (Stopped): 可以通过发送 SIGSTOP 信号给进程来停止进程，可以发送 SIGCONT 信号让进程继续运行

(5) X 死亡状态 (Dead) :该状态是返回状态，在任务列表中看不到；

(6) Z 僵尸状态 (Zombie) :子进程退出，父进程还在运行，但是父进程没有读到子进程的退出状态，子进程进入僵尸状态；

使用 echo 命令进行写数据到设备文件中。

```
/mnt # echo "i am study runinglinuxkernel now" > /dev/my_demo_dev
[88990.794902] demodrv_open: major=10, minor=58
[88990.799619] demodrv_write: pid=773, actual_write =33, ppos=0, ret=0
[88990.802754] demodrv_read, pid=861, actual_readed=33, pos=0
i am study runinglinuxkernel now
[88990.806433] demodrv_read: pid=861, going to sleep
/mnt #
```

从日志中可以看出，当输出一个字符串到设备时，首先执行打开函数，然后执行写入操作，写入了 33 字节，写者进程的 pid 是 773。然后，写者进程马上唤醒了读者进程，读者进程（pid 号是 861）把刚才写入的数据读到用户空间，也就是把 KFIFO 的数据读空了，导致读者进程又进入了睡眠状态。

3. 进阶思考

这个字符设备最经典的操作了，怎么进入睡眠，怎么被唤醒，希望大家好好体会一下。

1. 当 `wake_up()` 函数唤醒一个等待队列，它是唤醒所有进程还是一个进程？
2. 阅读现有内核代码的设备驱动，找一个现成的驱动，研究他们是怎么进行睡眠等待和唤醒的？

5.7 实验 7：向虚拟设备中添加 I/O 多路复用支持

1. 实验目的

- 1) 对虚拟设备的字符驱动添加 I/O 多路复用的支持。
- 2) 编写应用程序对 I/O 多路复用进行测试。

2. 实验详解

我们对虚拟设备驱动做了修改，让这个驱动可以支持多个设备。

```
struct mydemo_device {
    char name[64];
    struct device *dev;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
    struct kfifo mydemo_fifo;
};

struct mydemo_private_data {
    struct mydemo_device *device;
    char name[64];
};
```

我们对这个虚拟设备采用 `mydemo_device` 数据结构进行抽象，这个结构体里包含了 KFIFO 的环形缓冲区，还包含读和写的等待队列头。

5.7 实验 7：向虚拟设备中添加 I/O 多路复用支持

另外，我们还抽象了一个 `mydemo_private_data` 的数据结构，这个数据结构主要包含一些驱动的私有数据。在这个简单的设备驱动程序里暂时只包含了 `name` 名字和指向 `struct mydemo_device` 的指针，等以后这个驱动程序实现功能变多之后，再添加很多其他的成员，如锁、设备打开计数器等。

接下来看驱动的初始化函数是如何支持多个设备的。

```
#define MYDEMO_MAX_DEVICES 8
static struct mydemo_device *mydemo_device[MYDEMO_MAX_DEVICES];

static int __init simple_char_init(void)
{
    int ret;
    int i;
    struct mydemo_device *device;

    ret = alloc_chrdev_region(&dev, 0, MYDEMO_MAX_DEVICES, DEMO_NAME);
    if (ret) {
        printk("failed to allocate char device region");
        return ret;
    }

    demo_cdev = cdev_alloc();
    if (!demo_cdev) {
        printk("cdev_alloc failed\n");
        goto unregister_chrdev;
    }

    cdev_init(demo_cdev, &demodrv_fops);

    ret = cdev_add(demo_cdev, dev, MYDEMO_MAX_DEVICES);
    if (ret) {
        printk("cdev_add failed\n");
        goto cdev_fail;
    }

    for (i = 0; i < MYDEMO_MAX_DEVICES; i++) {
        device = kmalloc(sizeof(struct mydemo_device), GFP_KERNEL);
        if (!device) {
            ret = -ENOMEM;
            goto free_device;
        }

        sprintf(device->name, "%s%d", DEMO_NAME, i);
        mydemo_device[i] = device;
        init_waitqueue_head(&device->read_queue);
        init_waitqueue_head(&device->write_queue);

        ret = kfifo_alloc(&device->mydemo_fifo,
                          MYDEMO_FIFO_SIZE,
                          GFP_KERNEL);
        if (ret) {
            ret = -ENOMEM;
            goto free_kfifo;
        }

        printk("mydemo_fifo=%p\n", &device->mydemo_fifo);
    }
}

printk("succeeded register char device: %s\n", DEMO_NAME);
```

```

    return 0;

free_kfifo:
    for (i = 0; i < MYDEMO_MAX_DEVICES; i++)
        if (&device->mydemo_fifo)
            kfifo_free(&device->mydemo_fifo);
free_device:
    for (i = 0; i < MYDEMO_MAX_DEVICES; i++)
        if (mydemo_device[i])
            kfree(mydemo_device[i]);
cdev_fail:
    cdev_del(demo_cdev);
unregister_chrdev:
    unregister_chrdev_region(dev, MYDEMO_MAX_DEVICES);
    return ret;
}

```

MYDEMO_MAX_DEVICES 表示设备驱动最多支持 8 个设备。在模块加载函数 simple_char_init() 里使用 alloc_chrdev_region() 函数去申请 8 个次设备号，然后通过 cdev_add() 函数把这 8 个次设备都注册到系统里。

然后为每一个设备都分配一个 mydemo_device 数据结构，并且初始化其等待队列头和 KFIFO 环形缓冲区。

接下来看 open 方法的实现和之前有何不同。

```

static int demodrv_open(struct inode *inode, struct file *file)
{
    unsigned int minor = iminor(inode);
    struct mydemo_private_data *data;
    struct mydemo_device *device = mydemo_device[minor];
    int ret;

    printk("%s: major=%d, minor=%d, device=%s\n",
           MAJOR(inode->i_rdev), MINOR(inode->i_rdev), device->name);

    data = kmalloc(sizeof(struct mydemo_private_data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;
    sprintf(data->name, "private_data_%d", minor);

    data->device = device;
    file->private_data = data;

    return 0;
}

```

加粗部分就是和之前 open 方法的不同之处。这里首先会通过次设备号找到对应的 mydemo_device 数据结构，然后分配一个私有的 mydemo_private_data 的数据结构，最后把这个私有数据的地址存放在 file->private_data 指针里。

接下来看 poll 方法的实现。

```

static const struct file_operations demodrv_fops = {
    .owner = THIS_MODULE,
    .open = demodrv_open,
    .release = demodrv_release,
    .read = demodrv_read,
    .write = demodrv_write,
    .poll = demodrv_poll,
}

```

5.7 实验 7：向虚拟设备中添加 I/O 多路复用支持

```

};

static unsigned int demodrv_poll(struct file *file, poll_table *wait)
{
    int mask = 0;
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;

    poll_wait(file, &device->read_queue, wait);
    poll_wait(file, &device->write_queue, wait);

    if (!kfifo_is_empty(&device->mydemo_fifo))
        mask |= POLLIN | POLLRDNORM;
    if (!kfifo_is_full(&device->mydemo_fifo))
        mask |= POLLOUT | POLLWRNORM;

    return mask;
}

```

编译内核模块。

```

rlk@ubuntu:lab7 mydemodrv_poll$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0 M=/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_ba
sic/chapter_5/lab7_mydemodrv_poll modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab7_mydemodrv_poll/mydemodrv_
_poll.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab7_mydemodrv_poll/mydemo_po
ll.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab7_mydemodrv_poll/mydemo_po
ll.mod.o
  LD      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_5/lab7_mydemodrv_poll/mydemo_po
ll.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab7 mydemodrv_poll$ 

```

把内核模块拷贝到 kmodules 目录中。

```

rlk@ubuntu:lab7 mydemodrv_poll$ cp mydemo_poll.ko /home/rwk/rwk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab7 mydemodrv_poll$ 

```

启动 QEMU 虚拟机，并加载内核模块。先检查上一次实验的内核模块是否已经卸载。

```

/mnt # lsmod
mydemo_misc 1827 0 - Live 0xbff014000 (0)
/mnt # rmmod mydemo_misc.ko
[87790.937896] removing device
/mnt # 

```

加载本次实验的内核模块。

```
/mnt # ls
README      mydemo_misc.ko mytest.ko
mydemo.ko   mydemo_poll.ko test
/mnt # insmod mydemo_poll.ko
[89960.946978] mydemo_fifo=c47a0cdc
[89960.950381] mydemo_fifo=c47a0e1c
[89960.951095] mydemo_fifo=c47a0f5c
[89960.951878] mydemo_fifo=c47a07dc
[89960.952955] mydemo_fifo=c47a109c
[89960.953703] mydemo_fifo=c47a091c
[89960.954850] mydemo_fifo=c47a11dc
[89960.955940] mydemo_fifo=c47a1bdc
[89960.956342] succeeded register char device: mydemo_dev
/mnt #
```

从上面 log 可以看到驱动在初始化时候创建了 8 个设备。读者可以认真阅读驱动代码中的 simple_char_init 函数。

注意：本次实验，我们的参考代码没有采用传统的注册字符设备的方法，而没有使用 misc 机制，所以需要手工来创建设备节点。

```
/mnt #
/mnt # mknod /dev/mydemo0 c 252 0
/mnt # mknod /dev/mydemo1 c 252 1
/mnt #
```

如上图所示，我们手工创建了两个设备节点，分别是 mydemo0 和 mydemo1。

本实验需要写一个应用程序来测试这个 poll 方法是否工作。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
#include <poll.h>
#include <linux/input.h>

int main(int argc, char *argv[])
{
    int ret;
    struct pollfd fds[2];
    char buffer0[64];
    char buffer1[64];

    fds[0].fd = open("/dev/mydemo0", O_RDWR);
    if (fds[0].fd == -1)
        goto fail;
    fds[0].events = POLLIN;
    fds[0].revents = 0;

    fds[1].fd = open("/dev/mydemo1", O_RDWR);
    if (fds[1].fd == -1)
        goto fail;
    fds[1].events = POLLIN;
    fds[1].revents = 0;

    while (1) {
        ret = poll(fds, 2, -1);
```

5.7 实验 7：向虚拟设备中添加 I/O 多路复用支持

```

if (ret == -1)
    goto fail;

if (fds[0].revents & POLLIN) {
    ret = read(fds[0].fd, buffer0, 64);
    if (ret < 0)
        goto fail;
    printf("%s\n", buffer0);
}

if (fds[1].revents & POLLIN) {
    ret = read(fds[1].fd, buffer1, 64);
    if (ret < 0)
        goto fail;

    printf("%s\n", buffer1);
}

fail:
    perror("poll test");
    exit(EXIT_FAILURE);
}

```

在这个测试程序中，我们打开两个设备，然后分别进行监听。如果其中一个设备的 KFIFO 有数据，就把它读出来，并且输出。

这个 test 程序我们使用了 Linux 编程中常用的接口 poll 函数对多个设备文件进行监听。

编译 test 程序，将其复制到 kmodules 目录，运行 QEMU 虚拟机。

```

rlk@ubuntu:lab7_mydemodrv_poll$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab7_mydemodrv_poll$

```

```

rlk@ubuntu:lab7_mydemodrv_poll$ cp test /home/rilk/rilk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab7_mydemodrv_poll$ 

```

运行 test 程序，注意我们想让 test 程序在后台运行，所以添加了“&”，目的是让 test 进程来监听。

```

/mnt #
/mnt # ./test &
/mnt # [90533.703785] demodrv_open: major=252, minor=0, device=mydemo_dev0
[90533.708233] demodrv_open: major=252, minor=1, device=mydemo_dev1
/mnt # 

```

我们可以看到 test 程序打开了两个设备文件，一个是 mydemo_dev0，另外一个是 mydemo_dev1。

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
876	773	0	R	2308	2.6	2	0.7	top
3	2	0	SW	0	0.0	0	0.0	[ksoftirqd/0]
15	2	0	SW	0	0.0	2	0.0	[ksoftirqd/2]
1	0	0	S	2308	2.6	3	0.0	{linuxrc} init
773	1	0	S	2308	2.6	0	0.0	-/bin/sh
875	773	0	S	700	0.8	0	0.0	./test
754	2	0	SWN	0	0.0	1	0.0	[kmemleak]
4	2	0	SW	0	0.0	0	0.0	[kworker/0:0]
686	2	0	SW	0	0.0	1	0.0	[kworker/u8:2]
411	2	0	SW	0	0.0	1	0.0	[kworker/1:1]
7	2	0	SW	0	0.0	3	0.0	[rcu_sched]
409	2	0	SW	0	0.0	2	0.0	[kworker/2:1]
11	2	0	SW	0	0.0	1	0.0	[ksoftirqd/1]
328	2	0	SW	0	0.0	3	0.0	[kworker/3:1]
19	2	0	SW	0	0.0	3	0.0	[ksoftirqd/3]
23	2	0	SW	0	0.0	3	0.0	[kdevtmpfs]
279	2	0	SW	0	0.0	1	0.0	[khungtaskd]
2	0	0	SW	0	0.0	3	0.0	[kthreadd]
10	2	0	SW	0	0.0	1	0.0	[migration/1]
18	2	0	SW	0	0.0	3	0.0	[migration/3]

从 top 命令中看到， test 进程处于睡眠等待状态(S)。

接下来使用 echo 来写，我们可以往设备 0 里写入字符串。

```
/mnt # echo "i am linuxer" > /dev/mydemo0
[90787.028794] demodrv_open: major=252, minor=0, device=mydemo dev0
[90787.032838] demodrv_write:mydemo_dev0 pid=773, actual_write =13, ppos=0, ret=0
[90787.037887] demodrv_read:mydemo_dev0, pid=875, actual_readed=13, pos=0
i am linuxer
```

从 log 中我们可以看到，首先打开了 mydemo_dev0 设备，然后调用 write 函数往里面写数据，然后调用 read 函数从设备到读取到数据，最后打印出字符串。

接下来我们继续使用 echo 命令来写设备 1.

```
/mnt # echo "i am study runninglinuxkernel now" > /dev/mydemo1
[90931.744278] demodrv_open: major=252, minor=1, device=mydemo dev1
[90931.746088] demodrv_write:mydemo_dev1 pid=773, actual_write =34, ppos=0, ret=0
[90931.747806] demodrv_read:mydemo_dev1, pid=875, actual_readed=34, pos=0
i am study runninglinuxkernel now
/mnt #
```

从 log 中，我们同样可以看到，我们打开了设备 1，然后往设备里写入数据，最后从设备中读取数据到用户空间，并且把数据打印出来。

说明我们的 test 程序，可以同时处理多个设备，这就是 I/O 多路复用最简单的模型了。

另外，可以在设备驱动程序的 poll 方法中添加输出信息，看看有什么变化。

3 进阶思考

5.8 实验 8：为什么不能唤醒读写进程

字符设备的 polling 的方法是字符设备里高级的技巧和技能。

1. 大家可以去内核代码里看看，有哪些字符设备驱动使用了 polling 的方法？
2. 这个 test 程序，如果修改成 select 方法，如何修改？

5.8 实验 8：为什么不能唤醒读写进程

1. 实验目的

本实验是在 5.6.2 节实验 7 中故意设置的一个错误。希望读者通过发现问题和深入调试来解决问题，找到问题的根本原因，对字符设备驱动有一个深刻的认识。

2. 实验详解

本实验在 5.6.2 节实验 7 设备驱动程序中修改了部分代码，并故意制造了一个错误。

主要的修改是把环形缓冲区 KFIFO 以及读写等待队列头 read_queue 和 write_queue 都放入 struct mydemo_private_data 数据结构中。

```
struct mydemo_private_data {
    struct mydemo_device *device;
    char name[64];
    struct kfifo mydemo_fifo;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
};
```

在 demodrv_open() 函数中分配 kfifo，并初始化等待队列头 read_queue 和 write_queue。

```
static int demodrv_open(struct inode *inode, struct file *file)
{
    unsigned int minor = iminor(inode);
    struct mydemo_private_data *data;
    struct mydemo_device *device = mydemo_device[minor];
    int ret;

    printk("%s: major=%d, minor=%d, device=%s\n", __func__,
        MAJOR(inode->i_rdev), MINOR(inode->i_rdev), device->name);

    data = kmalloc(sizeof(struct mydemo_private_data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    sprintf(data->name, "private_data_%d", minor);

    ret = kfifo_alloc(&data->mydemo_fifo,
        MYDEMO_FIFO_SIZE,
        GFP_KERNEL);
    if (ret) {
        kfree(data);
        return -ENOMEM;
```

```

    }

    init_waitqueue_head(&data->read_queue);
    init_waitqueue_head(&data->write_queue);

    data->device = device;

    file->private_data = data;

    return 0;
}

```

另外，还需要相应修改的 demodrv_read() 和 demodrv_write() 函数。

编译好设备驱动程序，并将其复制到 kmodules 目录。创建设备节点文件，然后加载内核模块。

首先，在后台使用 cat 命令打开 /dev/mydemo0 设备。

```

/mnt # cat /dev/mydemo0 &
/mnt # demodrv_open: major=252, minor=0, device=my_demo_dev0
demodrv_read:my_demo_dev0 pid=724, going to sleep, private_data_0

```

然后，使用 echo 命令向 /dev/mydemo0 设备中写入字符串。

```

/mnt #
/mnt # echo "i am study linux now" > /dev/mydemo0
demodrv_open: major=252, minor=0, device=my_demo_dev0
wait up read queue, private_data_0
demodrv_write:my_demo_dev0 pid=703, actual_write =21, ppos=0, ret=0
/mnt #

```

最后，我们发现字符串虽然被写入设备中，而且也调用了 wake_up_interruptible (&data->read_queue)，但为什么没有唤醒 pid 为 724 的读者进程呢？

注：读者可以认真思考一下这个问题。若不看答案能把这个问题弄明白，一定对 Linux 内核的理解上一个台阶。本题的提示见教材（《奔跑吧 Linux 内核入门篇》）的第 5.10 章。

5.9 实验 9：向虚拟设备中添加异步通知

异步通知有点类似中断，当请求的设备资源可以获取时，由驱动程序主动通知应用程序，应用程序调用 read() 或 write() 函数来发起 I/O 操作。异步通知不像我们之前介绍的阻塞操作，它不会造成阻塞，只有设备驱动满足条件之后才通过信号机制通知应用程序去发起 I/O 操作。

异步通知使用了系统调用的 signal 函数和 sigcuation 函数。signal 函数让一个信号和一个函数对应，每当接收到这个信号时会调用相应的函数来处理。

1. 实验目的

学会如何给一个字符设备驱动程序添加异步通知功能。

5.9 实验 9：向虚拟设备中添加异步通知

2. 实验详解

在字符设备中添加异步通知，需要完成如下几步。

- 1) 在 mydemo_device 数据结构中添加一个 struct fasync_struct 数据结构指针，该指针会构造一个 struct fasync_struct 的链表头。

```
struct mydemo_device {
    char name[64];
    struct device *dev;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
    struct kfifo mydemo_fifo;
    struct fasync_struct *fasync;
};
```

- 2) 异步通知在内核中使用 struct fasync_struct 数据结构来描述。

```
<include/linux/fs.h>

struct fasync_struct {
    spinlock_t      fa_lock;
    int            magic;
    int            fa_fd;
    struct fasync_struct *fa_next; /* 单链表 */
    struct file     *fa_file;
    struct rcu_head  fa_rcu;
};
```

- 3) 设备驱动的 file_operations 的操作方法集中有一个 fasync 的方法，我们需要实现它。

```
static const struct file_operations demodrv_fops = {
    .owner = THIS_MODULE,
    ...
    .fasync = demodrv_fasync,
};

static int demodrv_fasync(int fd, struct file *file, int on)
{
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;

    return fasync_helper(fd, file, on, &device->fasync);
}
```

这里直接使用 fasync_helper()函数来构造 struct fasync_struct 类型的节点，并添加到系统的链表中。

- 4) 修改 demodrv_read()函数和 demodrv_write()函数，当请求的资源可用时，调用 kill_fasync()接口函数来发送信号。

```
< demodrv_write() 函数代码片段 >

static ssize_t
```

奔跑吧 linux 社区出品

```

demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t
*ppos)
{
    if (kfifo_is_full(&device->mydemo_fifo)) {
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;

        ret = wait_event_interruptible(device->write_queue,
                                       !kfifo_is_full(&device->mydemo_fifo));
        if (ret)
            return ret;
    }

    ret = kfifo_from_user(&device->mydemo_fifo, buf, count, &actual_write);
    if (ret)
        return -EIO;

    if (!kfifo_is_empty(&device->mydemo_fifo)) {
        wake_up_interruptible(&device->read_queue);
        kill_fasync(&device->fasync, SIGIO, POLL_IN);
    }
    return actual_write;
}

```

在 demodrv_write()函数中，当从用户空间复制数据到 KFIFO 中时，KFIFO 不为空，并通过 kill_fasync()接口函数发送 SIGIO 信号给用户程序。

编译内核模块。

```

rlk@ubuntu:lab9_mydemodrv_fasync$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_ba
sic/chapter_5/lab9_mydemodrv_fasync modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab9_mydemodrv_fasync/mydemod
rv_fasync.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab9_mydemodrv_fasync/mydemo_
fasync.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab9_mydemodrv_fasync/mydemo_
fasync.mod.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_5/lab9_mydemodrv_fasync/mydemo_
fasync.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab9_mydemodrv_fasync$
```

拷贝内核模块到 kmodules 目录。

```

rlk@ubuntu:lab9_mydemodrv_fasync$ cp mydemo_fasync.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab9_mydemodrv_fasync$
```

启动 QEMU 虚拟机，并加载内核模块。先检查上一次实验的内核模块是否已经卸载。

```

/mnt # lsmod
mydemo_poll 3900 0 - Live 0xb020000 (0)
/mnt # rmmod mydemo_poll.ko
[91293.381156] removing device
/mnt #
```

加载本次实验的内核模块。

5.9 实验 9：向虚拟设备中添加异步通知

```
/mnt #
/mnt # insmod mydemo_fasync.ko
[91328.164943] mydemo_fifo=c47a11dc
[91328.168961] mydemo_fifo=c47a145c
[91328.171579] mydemo_fifo=c47a091c
[91328.173998] mydemo_fifo=c47a02dc
[91328.175719] mydemo_fifo=c47a109c
[91328.178597] mydemo_fifo=c47a0a5c
[91328.180592] mydemo_fifo=c47a07dc
[91328.183369] mydemo_fifo=c47a159c
[91328.185225] succeeded register char device: mydemo_dev
/mnt #
```

从上面 log 可以看到驱动在初始化时候创建了 8 个设备。读者可以认真阅读驱动代码中的 simple_char_init 函数。

注意：本次实验，我们的参考代码没有采用传统的注册字符设备的方法，而没有使用 misc 机制，所以需要手工来创建设备节点。

```
/mnt #
/mnt # mknod /dev/mydemo0 c 252 0
/mnt # mknod /dev/mydemo1 c 252 1
/mnt #
```

如上图所示，我们手工创建了两个设备节点，分别是 mydemo0 和 mydemo1。

下面来看如何编写测试程序。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <errno.h>
#include <poll.h>
#include <signal.h>

static int fd;

void my_signal_fun(int signum, siginfo_t *siginfo, void *act)
{
    int ret;
    char buf[64];

    if (signum == SIGIO) {
        if (siginfo->si_band & POLLIN) {
            printf("FIFO is not empty\n");
            if ((ret = read(fd, buf, sizeof(buf))) != -1) {
                buf[ret] = '\0';
                puts(buf);
            }
        }
        if (siginfo->si_band & POLLOUT)
            printf("FIFO is not full\n");
    }
}

int main(int argc, char *argv[])
{
```

```

{
    int ret;
    int flag;
    struct sigaction act, oldact;

    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGIO);
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = my_signal_fun;
    if (sigaction(SIGIO, &act, &oldact) == -1)
        goto fail;

    fd = open("/dev/mydemo0", O_RDWR);
    if (fd < 0)
        goto fail;

/*设置异步I/O所有权*/
    if (fcntl(fd, F_SETOWN, getpid()) == -1)
        goto fail;

/*设置SIGIO信号*/
    if (fcntl(fd, F_SETSIG, SIGIO) == -1)
        goto fail;

/*获取文件flags*/
    if ((flag = fcntl(fd, F_GETFL)) == -1)
        goto fail;

/*设置文件flags, 设置FASYNC, 支持异步通知*/
    if (fcntl(fd, F_SETFL, flag | FASYNC) == -1)
        goto fail;

    while (1)
        sleep(1);

fail:
    perror("fasync test");
    exit(EXIT_FAILURE);
}

```

首先，通过 `sigaction()` 函数设置进程接收指定的信号，以及接收信号之后的动作，这里指定接收 `SIGIO` 信号，信号处理函数是 `my_signal_fun()`。接下来，就是打开设备驱动文件，并使用 `fcntl()` 函数来设置打开设备文件支持 `FASYNC` 功能。当测试程序接收到 `SIGIO` 信号之后，会执行 `my_signal_fun()` 函数，然后判断事件类型是否为 `POLLIN`。如果事件类型是 `POLLIN`，那么可以主动调用 `read()` 函数并把数据读出来。

编译 `test` 程序。

```

rlk@ubuntu:lab9_mydemodrv_fasync$ arm-linux-gnueabi-gcc test.c -o test --static
rlk@ubuntu:lab9_mydemodrv_fasync$ cp test /home/rwk/rwk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab9_mydemodrv_fasync$ 

```

首先加载内核模块和生成设备节点，然后在后台运行 `test` 程序。

```

/mnt # ./test &
/mnt # demodrv_open: major=252, minor=0, device=mydemo_dev0

```

5.9 实验 9：向虚拟设备中添加异步通知

```
/mnt #
/mnt # ./test &
/mnt # [ 484.962105] demodrv_open: major=252, minor=0, device=mydemo_dev0
[ 484.975610] demodrv_fasync send SIGIO
/mnt #
```

接着使用 echo 命令往设备里写入字符串。

```
/mnt # echo "i am linuxer" > /dev/mydemo0
demodrv_open: major=252, minor=0, device=mydemo_dev0
demodrv_write kill fasync
demodrv_write:mydemo_dev0 pid=703, actual_write =13, ppos=0, ret=0

FIFO is not empty
demodrv_read:mydemo_dev0, pid=730, actual_readed=13, pos=0
i am linuxer
```

```
/mnt # echo "i am linuxer" > /dev/mydemo0
[ 537.931608] demodrv_open: major=252, minor=0, device=mydemo_dev0
[ 537.940513] demodrv_write kill fasync
[ 537.942838] demodrv_write:mydemo_dev0 pid=772, actual_write =13, ppos=0, ret=0
/mnt # FIFO is not empty
[ 537.959796] demodrv_read:mydemo_dev0, pid=779, actual_readed=13, pos=0
i am linuxer

FIFO is not full
/mnt #
```

从日志中可以看出，结果符合我们的预期，echo 命令向设备中写入字符串，通过 kill_fasync() 接口函数给测试程序发生 SIGIO 信号。测试程序接收到该信号之后，主动调用一次 read() 函数去读，最后把刚才写入的字符串读到了用户空间。

3 进阶思考

在这个实验里，小明和小李同时做这个实验，小李得到了正确的结果，而小明却没有，他运行 test 程序之后，发生了 Oops 错误。

奔跑吧 linux 社区出品

```

/mnt # ./test &
/mnt # my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
my_class mydemo:252:1: demodrv_fasync send SIGIO
Unable to handle kernel paging request at virtual address 5c558162
pgd = ee098000
[5c558162] *pgd=00000000
Internal error: Oops: 5 [#1] SMP ARM
Modules linked in: mydemo_fasync(0)
CPU: 0 PID: 716 Comm: test Tainted: G          0    4.0.0+ #1
Hardware name: ARM-Versatile Express
task: eeacc280 ti: ee0b2000 task.ti: ee0b2000
PC is at fasync_insert_entry+0x58/0x210
LR is at fasync_insert_entry+0x48/0x210
pc : [<c0269464>]   lr : [<c0269454>]   psr: 20000013
sp : ee0b3e38 ip : 00000010 fp : be83bd64
r10: 00000000 r9 : ee0b2000 r8 : c0014ec4
r7 : 000000dd r6 : 00088468 r5 : 00010158 r4 : 00086b98
r3 : 5c558152 r2 : 00000000 r1 : 00010000 r0 : 00000000
Flags: nzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
Control: 10c5387d Table: 8e09806a DAC: 00000015
Process test (pid: 716, stack limit = 0xee0b2210)
Stack: (0xee0b3e38 to 0xee0b4000)
3e20:
3e40: ee072f00 00000003 c02693d0 000000d0 ee045000 ee0b3fb0 00000001 00000030
3e60: c0259ab4 c0686e2c 00000000 c109152c c10a8c54 000000d0 00000020 00000020
3e80: ee0752a0 c02693d0 5c558152 ee072f18 00086b98 ee070af0 00086b98 c0269660
3ea0: ee1a6808 ee070af0 ee072f00 00000003 00000001 ee0752a0 ee07b800 c02696d4
3ec0: ee070af0 00000001 ee072f00 00000003 bf000e68 bf000a80 ee07b800 00000001
3ee0: ee072f00 00000003 ee07b824 c0267c2c ee070a80 ee966100 ee07b800 c02679e0

3ee0: ee072f00 00000003 ee07b824 c0267c2c ee070a80 ee966100 ee07b800 c02679e0
3f00: 0006147c 00002002 ee072f00 00000003 ee072f00 ee072f00 00000000 ee072f00
3f20: eefffe08 ee148fa0 00000000 00000000 eefffe00 c0268290 ee072f00 00002002
3f40: 00000004 00000003 00000001d ffffffea ee072f00 c0268724 ee072f00 00002002
3f60: 00000004 00000003 ee072f00 00000000 ee072f00 00000000 00002002 00000004
3f80: ee072f00 00000000 ee072f00 00000003 00002002 00000004 00000003 00000002
3fa0: 00000017 c0014d40 00086b98 00010158 00000003 00000004 00002002 00086b98
3fc0: 00086b98 00010158 00088468 000000dd 00000000 00000000 00000000 be83bd64
3fe0: 90231c00 be83bbe8 000296bc 000295a0 80000010 00000003 8f7fd821 8f7fdc21
[<c0269464>] (fasync_insert_entry) from [<c0269660>] (fasync_add_entry+0x44/0x70)
[<c0269660>] (fasync_add_entry) from [<c02696d4>] (fasync_helper+0x48/0x58)
[<c02696d4>] (fasync_helper) from [<bf000a80>] (demodrv_fasync+0x64/0x78 [mydemo_fasync])
[<bf000a80>] (demodrv_fasync [mydemo_fasync]) from [<c02679e0>] (setfl+0x1a8/0x270)
[<c02679e0>] (setfl) from [<c0268290>] (do_fcntl+0x1b8/0x33c)
[<c0268290>] (do_fcntl) from [<c0268724>] (sys_fcntl64+0x1a4/0x1ec)
[<c0268724>] (sys_fcntl64) from [<c0014d40>] (ret_fast_syscall+0x0/0x34)
Code: e59d3004 e58d305c ea000023 e59d3050 (e5932010)
---[ end trace 950a00a438f0262d ]---

[1]+ Segmentation fault      ./test
/mnt #

```

这是为什么呢？请您帮小明解决一下这个问题，分析这个问题产生的原因和给出解决办法。

下面是解决该问题的思路。

首先我们要观察和分析这个日志。

明确这是一个 OOPS 错误的 log。从这句日志“Unable to handle kernel paging request at virtual address 5c558162”我们可以看出内核发生了引用空指针的错误。

日志里显示了当前 PC 指针指向哪个函数。

```

PC is at fasync_insert_entry+0x58/0x210
LR is at fasync_insert_entry+0x48/0x210

```

我们可以看到 PC 指针指向 fasync_insert_entry 函数中的第 0x58 个字节的地方，

5.9 实验 9：向虚拟设备中添加异步通知

0x210 表示 fasync_insert_entry 函数编译成二进制文件一共有 0x210 个字节，而出错的地方就是在 0x58 个字节的地方。

接下来打印出错时候通用寄存器的值。

```
pc : [<c0269464>] lr : [<c0269454>] psr: 20000013
sp : ee0b3e38 ip : 00000010 fp : be83bd64
r10: 00000000 r9 : ee0b2000 r8 : c0014ec4
r7 : 000000dd r6 : 00088468 r5 : 00010158 r4 : 00086b98
r3 : 5c558152 r2 : 00000000 r1 : 00010000 r0 : 00000000
Flags: nzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
Control: 10c5387d Table: 8e09806a DAC: 00000015
```

接下来是打印出错时候栈的内容。

```
Stack: (0xee0b3e38 to 0xee0b4000)
3e20: ee0752a0 ee070af0
3e40: ee072f00 00000003 c02693d0 000000d0 ee045000 ee0b3fb0 00000001 00000030
3e60: c0259ab4 c0686e2c 00000000 c109152c c10a8c54 000000d0 00000020 00000020
3e80: ee0752a0 c02693d0 5c558152 ee072f18 00086b98 ee070af0 00086b98 c0269660
3ea0: ee1a6808 ee070af0 ee072f00 00000003 00000001 ee0752a0 ee07b800 c02696d4
3ec0: ee070af0 00000001 ee072f00 00000003 bf000e68 bf000a80 ee07b800 00000001
3ee0: ee072f00 00000003 ee07b824 c0267c2c ee070a80 ee968100 ee07b800 c02679e0
3f00: 0006147c 00002002 ee072f00 00000003 ee072f00 ee072f00 00000000 ee072f00
3f20: eeafffe08 ee148fa0 00000000 00000000 eeafffe00 c0268290 ee072f00 00002002
3f40: 00000004 00000003 0000001d ffffffea ee072f00 c0268724 ee072f00 00002002
3f60: 00000004 00000003 ee072f00 00000000 ee072f00 00000000 00002002 00000004
3f80: ee072f00 00000000 ee072f00 00000003 00002002 00000004 00000003 00000002
3fa0: 00000017 c0014d40 00086b98 00010158 00000003 00000004 00002002 00086b98
3fc0: 00086b98 00010158 00088468 000000dd 00000000 00000000 00000000 be83bd64
3fe0: 90231c00 be83bbe8 000296bc 000295a0 80000010 00000003 8f7fd821 8f7fdc21
```

接着是打印函数调用关系 calltrace。

```
[<c0269464>] (fasync_insert_entry) from [<c0269660>]
(fasync_add_entry+0x44/0x70)
[<c0269660>] (fasync_add_entry) from [<c02696d4>] (fasync_helper+0x48/0x58)
[<c02696d4>] (fasync_helper) from [<bf000a80>] (demodrv_fasync+0x64/0x78
[mydemo_fasync])
[<bf000a80>] (demodrv_fasync [mydemo_fasync]) from [<c02679e0>]
(setfl+0x1a8/0x270)
[<c02679e0>] (setfl) from [<c0268290>] (do_fcntl+0x1b8/0x33c)
[<c0268290>] (do_fcntl) from [<c0268724>] (SyS_fcntl64+0x1a4/0x1ec)
[<c0268724>] (SyS_fcntl64) from [<c0014d40>] (ret_fast_syscall+0x0/0x34)
Code: e59d3004 e58d305c ea000023 e59d3050 (e5932010)
---[ end trace 950a00a438f0262d ]---
```

从函数调用关系可以看到，函数的调用关系图是这样：

```
SyS_fcntl64-> do_fcntl-> setfl-> demodrv_fasync-> fasync_helper->
fasync_add_entry-> fasync_insert_entry
```

test 程序在用户空间调用了 fcntl()函数，然后在内核空间里，一直调用到我们驱动的 demodrv_fasync()函数里。在 demodrv_fasync()函数里，我们调用内核的 API 接口函数 fasync_helper()函数。最后在 fasync_helper()函数内部的 fasync_insert_entry()函数就结束了函数调用关系图，说明极有可能出错在 fasyncinsert_entry()函数里。

看明白了日志信息之后，我们可以开始动手解决问题了。

首先使用 `gdb` 来分析 PC 指针的指向的函数，它可以帮助我们查看出错是在哪一句代码里。这里使用 `gdb-multiarch` 程序。

```
| $ gdb-multiarch vmlinux
```

```
rlk@ubuntu:runninglinuxkernel_4.0$ gdb-multiarch vmlinux
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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 "x86_64-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 vmlinux...done.
(gdb) █
```

```
(gdb) l *fasync_insert_entry+0x58
0xc026eea4 is in fasync_insert_entry (fs/fcntl.c:630).
625         struct fasync_struct *fa, **fp;
626
627         spin_lock(&filp->f_lock);
628         spin_lock(&fasync_lock);
629         for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
630             if (fa->fa_file != filp)
631                 continue;
632             spin_lock_irq(&fa->fa_lock);
633             fa->fa_fd = fd;
(gdb) █
```

使用 “`l * fasync_insert_entry+0x58`” 命令，我们可以看到 `gdb` 显示出错的时候是在 `fs/fcntl.c` 的第 630 行代码里。

接下来就要去分析为什么会在第 630 行中出错。第 630 行，只是判断 `fa->fa_file` 是否等于 `filp`，为什么这里会出现空指针访问呢？即 log 中的：Unable to handle kernel paging request at virtual address。有一种可能性就是 `fa` 是一个空指针或者是一个非法指针。

5.9 实验 9：向虚拟设备中添加异步通知

```

623 struct fasync_struct *fasync_insert_entry(int fd, struct file *filp, struct fasync_struct **fapp, struct
624 {
625     struct fasync_struct *fa, **fp;
626
627     spin_lock(&filp->f_lock);
628     spin_lock(&fasync_lock);
629     for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
630         if (fa->fa_file != filp)
631             continue;
632
633         spin_lock_irq(&fa->fa_lock);
634         fa->fa_fd = fd;
635         spin_unlock_irq(&fa->fa_lock);
636         goto out;
637     }
638
639     spin_lock_init(&new->fa_lock);
640     new->magic = FASYNC_MAGIC;
641     new->fa_file = filp;
642     new->fa_fd = fd;
643     new->fa_next = *fapp;
644     rCU_assign_pointer(*fapp, new);
645     filp->f_flags |= FASYNC;
646 }
```

从第 629 行分析，fa 是从参数 fapp 过来的，fapp 是一个二级指针，那么 fa 就是 *fapp 的值。而这个 fapp 的参数是从我们的驱动程序 demodrv_fasync() 函数传递过来的。

```

153 static int demodrv_fasync(int fd, struct file *file, int on)
154 {
155     struct mydemo_private_data *data = file->private_data;
156     struct mydemo_device *device = data->device;
157
158     printk("%s send SIGIO\n", __func__);
159     return fasync_helper(fd, file, on, &device->fasync);
160 }
161 }
```

我们先看一下 device 数据结构的定义。

```

20 struct mydemo_device {
21     char name[64];
22     struct device *dev;
23     wait_queue_head_t read_queue;
24     wait_queue_head_t write_queue;
25     struct kfifo mydemo_fifo;
26     struct fasync_struct *fasync;
27 };
28 }
```

fasync 是 struct fasync_struct 数据结构的一个指针。在 demodrv_fasync() 函数把这个数据结构指针的地址作为 fasync_helper() 函数的参数。

而 struct fasync_struct 数据结构的初始化是在 simple_char_init() 函数中。

```

197
198     for (i = 0; i < MYDEMO_MAX_DEVICES; i++) {
199         device = kmalloc(sizeof(struct mydemo_device), GFP_KERNEL);
200         if (!device) {
201             ret = -ENOMEM;
202             goto free_device;
203         }
204
205         sprintf(device->name, "%s%d", DEMO_NAME, i);
206         mydemo_device[i] = device;
207         init_waitqueue_head(&device->read_queue);
208         init_waitqueue_head(&device->write_queue);
209
210         ret = kfifo_alloc(&device->mydemo_fifo,
211                           MYDEMO_FIFO_SIZE,
212                           GFP_KERNEL);
213         if (ret) {
214             ret = -ENOMEM;
215             goto free_kfifo;
216         }

```

在第 199 行，使用 kmalloc() 函数来为 struct fasync_struct 数据结构分配内存。按照我们的设想，这个 struct fasync_struct 数据结构里所有的成员都应该是全新的，也就是全部初始化为 0，也就是 device->fasync 也是指向空指针。

那么在 fs/fcntl.c 的第 629 行的这条判断语句 “(fa = *fp) != NULL”，理应判断为假 (false)，因此不会执行到第 630 行的语句中。

那什么原因让判断语句 “(fa = *fp) != NULL” 为真呢？

小明经过很多天思考，发现 kmalloc() 有可能导致这个问题。为什么呢？kmalloc 分配的内存不能保证内存都是初始化为 0，也就是说有可能是乱的数据。这样就可能导致 “(fa = *fp) != NULL” 语句判断为真，而执行第 630 行时候出现了空指针访问。

修改办法也很简单，打开 mydemodrv_fasync.c 文件，把第 199 行的 kmalloc() 函数改成 kzalloc() 函数即可。

kzalloc() 函数会在 kmalloc() 函数基础上，把分配得到的内存全部初始化为 0。

这个例子非常接近我们实际工作中遇到的驱动和内核 bug，了解其接近问题的思路非常重要，这可能对初学者来说比较难，但是我们可以先了解，然后在细细体会。

6.1 实验 1：在 ARM32 机器上新增一个系统调用

第 6 章

系统调用

6.1 实验 1：在 ARM32 机器上新增一个系统调用

1. 实验目的

通过新增一个系统调用，理解系统调用的实现过程。

2. 实验详解

1) 在 ARM Vexpress 平台上新增一个系统调用，该系统调用不用传递任何参数，在该系统调用里输出当前进程的 PID 和 UID 值。

2) 编写一个应用程序来调用这个新增的系统调用。

实验要求添加的系统调用不传递任何参数，一般将 pid 和 uid 直接通过 printk 输出到 dmesg 中，但是这样非常的不优雅。该参考代码添加了一个系统调用。

```
| long getpid(pid_t *pid, uid_t *uid);
```

pid 和 uid 通过参数返回到用户空间。

(1) 添加补丁

```
# cd runninglinuxkernel_4.0
# git am rlk_lab/r lk_basic/chapter_6/lab1/0001-arm32-add-a-new-syscall-which-
called-getpuid.patch
```

这里我们新添加的系统调用名称为 getpuid

getpuid 使用的系统调用号为 388

(2) 编译内核

```
# export ARCH=arm
# export CROSS_COMPILE=arm-linux-gnueabi-
# make vexpress_defconfig
# make menuconfig
# make -j4
```

(3) 编写测试程序

```
# cp rlk_lab/r lk_basic/chapter_6/lab1/test_getpuid_syscall.c
kmodules/test_getpuid_syscall.c
# cd kmodules/
# arm-linux-gnueabi-gcc --static -o test_getpuid_syscall
test_getpuid_syscall.c
```

(4) 启动 QEMU 虚拟机

```
# ./run.sh arm32
```

(5) 运行测试程序

```
/mnt # ./test_getpuid_syscall
call getpuid success, return pid = 774, uid = 0
/mnt #
/mnt # ./test_getpuid_syscall
call getpuid success, return pid = 775, uid = 0
/mnt #
```

3. 代码分析

对于 ARM32 架构的 Linux 内核，系统调用号的定义是实现在 arch/arm/include/uapi/asm/unistd.h 头文件中。

6.1 实验 1：在 ARM32 机器上新增一个系统调用

```

24 /*
25 * This file contains the system call numbers.
26 */
27
28 #define __NR_restart_syscall      ( __NR_SYSCALL_BASE+ 0)
29 #define __NR_exit                ( __NR_SYSCALL_BASE+ 1)
30 #define __NR_fork                ( __NR_SYSCALL_BASE+ 2)
31 #define __NR_read                ( __NR_SYSCALL_BASE+ 3)
32 #define __NR_write               ( __NR_SYSCALL_BASE+ 4)
33 #define __NR_open                ( __NR_SYSCALL_BASE+ 5)
34 #define __NR_close               ( __NR_SYSCALL_BASE+ 6)
35 /* 7 was sys waitpid */
36 #define __NR_creat               ( __NR_SYSCALL_BASE+ 8)
37 #define __NR_link                ( __NR_SYSCALL_BASE+ 9)
38 #define __NR_unlink              ( __NR_SYSCALL_BASE+ 10)
39 #define __NR_execve              ( __NR_SYSCALL_BASE+ 11)
40 #define __NR_chdir               ( __NR_SYSCALL_BASE+ 12)
41 #define __NR_time               ( __NR_SYSCALL_BASE+ 13)

```

那我们可以在系统调用最后的地方，增加一个我们新的系统调用，新的系统调用为 getpuid，号码为第 388 号，见代码中的第 417 行。

```

410 #define __NR_sched_getattr      ( __NR_SYSCALL_BASE+381)
411 #define __NR_renameat2          ( __NR_SYSCALL_BASE+382)
412 #define __NR_seccomp            ( __NR_SYSCALL_BASE+383)
413 #define __NR_getrandom          ( __NR_SYSCALL_BASE+384)
414 #define __NR_memfd_create       ( __NR_SYSCALL_BASE+385)
415 #define __NR_bpf                ( __NR_SYSCALL_BASE+386)
416 #define __NR_execveat           ( __NR_SYSCALL_BASE+387)
417 #define __NR_getpuid             ( __NR_SYSCALL_BASE+388)
418 */
419 */

```

另外在 arch/arm/include/asm/unistd.h 头文件中，有一个全局变量 __NR_syscalls 表示一共有多少的系统调用，该值需要设置比实际的系统调用总数要大，而且按 4 个字节对齐。我们把 __NR_syscalls 设置为 392。

```

diff --git a/arch/arm/include/asm/unistd.h b/arch/arm/include/asm/unistd.h
index 32640c43..7cba573c 100644
--- a/arch/arm/include/asm/unistd.h
+++ b/arch/arm/include/asm/unistd.h
@@ -19,7 +19,7 @@
 * This may need to be greater than __NR_last_syscall+1 in order to
 * account for the padding in the syscall table
 */
-#define __NR_syscalls (388)
+#define __NR_syscalls (392)

```

在 arch/arm/kernel/calls.S 汇编中，通过 CALL 宏来定义具体的系统调用要实现的接口函数，每一个系统调用在内核中的接口函数是以 sys_* 为开头的函数。因此，我们在这里增加 getpuid 的实现。

```

390             CALL(sys_kcmp)
391             CALL(sys_finit_module)
392 /* 380 */             CALL(sys_sched_setattr)
393             CALL(sys_sched_getattr)
394             CALL(sys_renameat2)
395             CALL(sys_seccomp)
396             CALL(sys_getrandom)
397 /* 385 */             CALL(sys_memfd_create)
398             CALL(sys_bpf)
399             CALL(sys_execveat)
400             CALL(sys_getpuid)
401 #ifndef syscalls_counted
402 .equ syscalls_padding, ((NR_syscalls + 3) & ~3) - NR_syscalls
403 #define syscalls_counted
404 #endif
405 .rept syscalls_padding
406             CALL(sys_ni_syscall)
407 .endr

```

这个 CALL 宏实现是在 kernel/entry-common.S 汇编文件中。

最后，我们在 arch/arm/kernel/sys_arm.c 文件中，实现 sys_getpuid 函数。

```

--- a/arch/arm/kernel/sys_arm.c
+++ b/arch/arm/kernel/sys_arm.c
@@ -37,3 +37,17 @@ asmlinkage long sys_arm_fadvise64_64(int fd, int advice,
{
    return sys_fadvise64_64(fd, offset, len, advice);
}
+
+SYSCALL_DEFINE2(getpuid, pid_t __user *, pid, uid_t __user *, uid)
+{
+    if (pid == NULL && uid == NULL)
+        return -EINVAL;
+
+    if (pid != NULL)
+        *pid = task_tgid_vnr(current);
+
+    if (uid != NULL)
+        *uid = from_kuid_munged(current_user_ns(), current_uid());
+
+    return 0;
+}

```

我们的 test 测试程序也很简单，直接使用 libc 中 syscall 函数来调用我们新加的系统调用函数。

6.2 实验 2：在优麒麟 Linux 机器上新增一个系统调用

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <errno.h>
5
6 int main(int argc, char **argv)
7 {
8     long pid, uid;
9     int ret;
10
11     ret = (int)syscall(388, &pid, &uid);
12     if (ret != 0) {
13         printf("call getpid failed\n");
14         return 1;
15     }
16
17     printf("call getpid success, return pid = %ld, uid = %ld\n", pid, uid);
18
19     return 0;
20 }
```

6.2 实验 2：在优麒麟 Linux 机器上新增一个系统调用

1. 实验目的

通过新增一个系统调用，理解系统调用的实现过程。

2. 实验详解

1) 在优麒麟 Linux 平台上新增一个系统调用，该系统调用不用传递任何参数，在该系统调用里输出当前进程的 PID 和 UID 值。该实验的目的是让读者学会如何在 x86_64 里添加一个系统调用，并比较和 ARM32 系统的区别。

2) 编写一个应用程序来调用这个新增的系统调用。

实验要求添加的系统调用不传递任何参数，一般将 pid 和 uid 直接通过 printk 输出到 dmesg 中，但是这样非常的不优雅。该参考代码添加了一个系统调用。

```
| long getpid(pid_t *pid, uid_t *uid);
```

pid 和 uid 通过参数返回到用户空间。

为了方便进行实验，以下实验基于《奔跑吧 Linux 入门版》中第一章的实验室 2，我们选定了最新的社区稳定版内核来修改添加系统调用，然后编译后，安装到优麒麟 Linux 机器上。

(1) 下载最新 Linux 内核

下载最新的社区稳定版内核

在完成该实验时，社区最新的稳定版内核是 linux-5.1.16，下载方法如下：

```
| $ wget -c https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.1.16.tar.xz
```

```
$ xz -d linux-5.1.16.tar.xz
$ tar -xf linux-5.1.16.tar
$ cd linux-5.1.16/
```

(2) 添加系统调用 getpuid

打上我们提供的参考补丁。

```
$ patch -p1 < /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_6/lab2/0001-x86-add-a-new-syscall-which-called-getpuid.patch
```

(3) 重新编译内核

为了方便，我们直接复制优麒麟 Linux 系统中自带的配置文件，我的系统上的配置文件为：/boot/config-4.15.0-29-generic，相关命令如下：

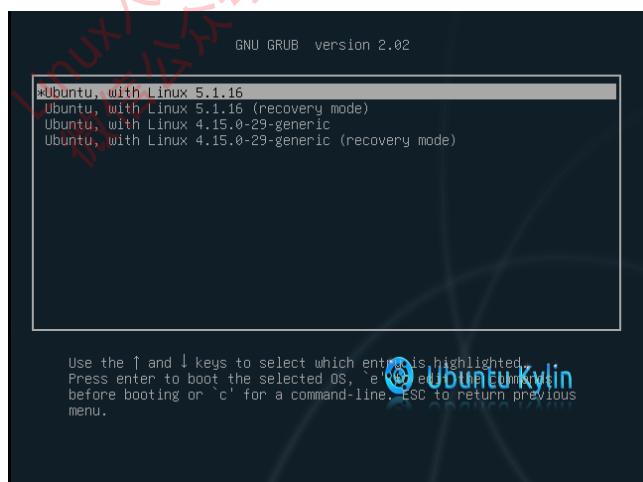
```
$ cd linux-5.1.16/
$ cp /boot/config-4.15.0-29-generic .config
$ make menuconfig
$ make -j4
```

编译时间取决于主机的处理能力。大概需要几十分钟。

编译完成之后，我们需要安装内核。

```
$ sudo make modules_install
$ sudo make install
```

安装完成后，重启电脑，用刚才编译的内核启动，登录系统。



(4) 编译测试程序。

进入到我们实验参考代码目录。

```
# cd /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_6/lab2
```

7.1 实验 1：查看系统内存信息

```
# gcc test_getpid_syscall.c -o test_getpid_syscall
#./test_getpid_syscall
```

```
rlk@ubuntu:lab2$ ./test_getpid_syscall
call getpid success, return pid = 2289, uid = 1000
rlk@ubuntu:lab2$
```

第 7 章

内存管理

7.1 实验 1：查看系统内存信息

1. 实验目的

- 1) 通过熟悉 Linux 系统中常用的内存监测工具来感性地认识和了解内存管理。
- 2) 在优麒麟 Linux 下查看系统内存信息。

2. 实验详解

(1) Top 工具

Top 命令是最常用的查看 Linux 系统信息的命令之一，它可以实时显示系统中各个进程的资源占用情况。

```
Tasks: 585 total, 1 running, 285 sleeping, 298 stopped, 1 zombie
%Cpu(s): 0.3 us, 0.1 sy, 0.0 ni, 99.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 7949596 total, 640464 free, 5042036 used, 2267096 buff/cache
KiB Swap: 16586748 total, 13447420 free, 3139328 used. 2226976 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  3958 figo      20   0  453988 117044  77620 S  3.0  1.5  20:27.13 Xvnc4
  4052 figo      20   0  668728  44504 11264 S  2.3  0.6  11:13.86 gnome-terminal-server
     8 root      20   0      0      0        0 S  0.3  0.0  15:07.14 [rcu_sched]
```

奔跑吧 linux 社区出品

```

2850 figo      20   0 1508396 288632  32944 S  0.3  3.6 404:23.96 compiz
6851 figo      20   0 44116  4156   3004 R  0.3  0.1  0:00.32 top
  2 root       20   0      0      0      0 S  0.0  0.0  0:00.89 [kthreadd]
  4 root       0 -20     0      0      0 S  0.0  0.0  0:00.00 [kworker/0:0H]
  6 root       0 -20     0      0      0 S  0.0  0.0  0:00.00 [mm_percpu_wq]
  7 root       20   0      0      0      0 S  0.0  0.0  0:07.72 [ksoftirqd/0]
  9 root       20   0      0      0      0 S  0.0  0.0  0:00.00 [rcu_bh]
 10 root      rt   0      0      0      0 S  0.0  0.0  0:00.31 [migration/0]
 11 root      rt   0      0      0      0 S  0.0  0.0  0:11.32 [watchdog/0]

```

第 3 行和第 4 行显示了主存 (Mem) 和交换分区 (Swap) 的总量、空闲量以及使用量。另外还显示了缓冲区以及页缓存大小 (buff/cache)。

第 5 行显示了进程信息区的统计数据，常用的如下所示。

- ❑ PID: 进程的 ID。
- ❑ USER: 进程所有者的用户名。
- ❑ PR: 进程优先级。
- ❑ NI: 进程的 nice 值。
- ❑ VIRT: 进程使用的虚拟内存总量，单位是 KB。
- ❑ RES: 进程使用的并且未被换出的物理内存大小，单位是 KB。
- ❑ SHR: 共享内存大小，单位是 KB。
- ❑ S: 进程的状态。(D=不可中断的睡眠状态, R=运行, S=睡眠, T=跟踪/停止, z=僵尸进程)。
- ❑ %CPU: 上一次更新到现在的 CPU 时间占用百分比。
- ❑ %MEM: 进程使用物理内存的百分比。
- ❑ TIME+: 进程使用的 CPU 时间总计，单位是 10ms。
- ❑ COMMAND: 命令名或命令行。

上面列出了常用的统计信息，还有一些隐藏的统计信息，比如 CODE (可执行代码大小)、SWAP (交换出去的内存大小)、nMaj/nMin (产生缺页异常的次数) 等，可以通过 f 键来选择要显示的内容。

除此之外，top 命令还可以在执行过程中使用一些交互命令，比如“M”可以根据进程使用内存的大小来排序。

(2) vmstat 命令

vmstat 命令也是常见的 Linux 系统的监控小工具，它可以显示系统的 CPU、内存以及 IO 的使用情况。

vmstat 命令通常带有两个参数，第一个参数采用时间间隔，单位是 s，第二个参数采用采样次数。比如“vmstat 2 5”表示每 2s 采样一次数据，并且连续采样 5 次。

```

figo@figo-OptiPlex-9020:~$ vmstat
procs -----memory----- --swap-- -----io---- -system-- -----cpu--
---
r b swpd free  buff cache si so bi bo in cs us sy id wa st
0 0 3139328 645744 1242708 1016716 0 0 4 2 0 1 0 0 99 0 0

```

vmstat 命令显示的单位是 KB。在大型的服务器中，可以使用-S 选项来按照 MB 或者 GB 来显示。

7.2 实验 2 : 获取系统的物理内存信息

```
figo@figo-OptiPlex-9020:~$ vmstat -S M
procs -----memory----- swap-- io---- system-- cpu--
--r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 3065 630 1213 992 0 0 4 2 0 1 0 0 99 0 0
```

下面简单介绍 `vmstat` 命令显示的各个参数的含义。

- r: 表示在运行队列中正在执行和等待的进程数。
- b: 表示阻塞的进程。
- swap: 表示交换到交换分区的内存大小。
- free: 空闲的物理内存大小。
- buff: 用作磁盘缓存的大小。
- cache: 用于页面缓存的内存大小。
- si: 每秒从交换分区读回到内存的大小。
- so: 每秒写入交换分区的大小。
- bi: 每秒读取磁盘 (块设备) 的块数量。
- bo: 每秒写入磁盘 (块设备) 的块数量。
- in: 每秒中断数, 包括时钟中断。
- cs: 每秒上下文切换数量。
- us: 用户进程执行时间百分比。
- sy: 内核系统进程执行时间百分比。
- wa: I/O 等待时间百分比。
- id: 空闲时间百分比。

7.2 实验 2：获取系统的物理内存信息

1. 实验目的

了解和熟悉 Linux 内核的物理内存管理的方法。比如 `struct page` 数据结构的使用, 特别是 `struct page` 的 `flags` 标志位的使用。

2. 实验要求

Linux 内核对每个物理页面都采用 `struct page` 数据结构来描述, 内核为每一个物理页面都分配了这样一个 `struct page` 数据结构, 并且存储到一个全局的数组 `mem_map[]` 中。它们之间的对应关系是 1:1 的线性映射, 即 `mem_map[]` 数组的第 0 个元素指向页帧号为 0 的物理页面的 `struct page` 数据结构。请写一个简单的内核模块程序, 通过遍历这个 `mem_map[]` 数组来统计当前系统有多少个空闲页面、保留页面、`swapcache` 页面、`slab` 页面、脏页面、活跃页面、正在回写的页面等。

3. 实验步骤

进入到该实验的参考代码中。

```
$ cd /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info
```

编译内核模块。

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ make
```

```
rlk@ubuntu:lab2_get_system_mem_info$ export ARCH=arm
rlk@ubuntu:lab2_get_system_mem_info$ export CROSS_COMPILE=arm-linux-gnueabi-
rlk@ubuntu:lab2_get_system_mem_info$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0 M=/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info/get_mm_info.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info/mm_info.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info/mm_info.mod.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab2_get_system_mem_info/mm_info.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab2_get_system_mem_info$
```

拷贝内核模块到 kmodules 目录。

```
rlk@ubuntu:lab2_get_system_mem_info$ cp mm_info.ko /home/rwk/rwk_basic/runninglinuxkernel_4.0/kmodules/
rlk@ubuntu:lab2_get_system_mem_info$
```

在另外一个终端中，运行 QEMU 虚拟机。

```
$ cd /home/rwk/rwk_basic/runninglinuxkernel_4.0
$ ./run.sh arm32
```

进入 mnt 目录，加载内核模块。

7.2 实验 2：获取系统的物理内存信息

```
/ # cd /mnt/
/mnt # ls
README          mm_info.ko      test_getpuid_syscall
/mnt # insmod mm_info.ko
[ 33.399851] mm_info.ko
[ 33.399851] Examining 25600 pages (num_phys_pages) = 100 MB
[ 33.401779] Pages with valid PFN's=25600, = 100 MB
[ 33.402402]
[ 33.402402]          Pages   KB   MB
[ 33.402402]
[ 33.403014] free     = 13427  53708  52
[ 33.403794] locked   =      0    0     0
[ 33.404164] reserved = 3759   15036  14
[ 33.404793] swapcache =      0    0     0
[ 33.405389] referenced = 1462   5848   5
[ 33.406224] slab     = 3389   13556  13
[ 33.406724] private  =      0    0     0
[ 33.407226] uptodate = 1648   6592   6
[ 33.407648] dirty    = 1596   6384   6
[ 33.408191] active   = 582    2328   2
[ 33.408511] writeback =      0    0     0
[ 33.408855] mappedtodisk =      0    0     0
/mnt #
```

从上面 log 可以看到，我们一共检查了 25600 个页面，对应内存大小是 100MB。每个页面对应一个页帧，同时我们也检查这些页面对应的页帧号是否有效的。因此有效的页帧号总数是 25600 个。

接下来，统计各种类型页面的数量：

- 空闲页面 free：一共 13427 个
- 加了 pagelock 的页面：0
- 保留的页面：3759
- swapcache 的页面：0
- referenced 页面：1462
- slab 页面：3389
- private 页面：0
- uptodate：1648
- dirty 页面：1596
- 活跃页面：582
- writeback 页面：0
- mappedtodisk 页面：0

读者可以对应/proc/meminfo 打印出来的数值进行比较，需要注意的是，因为内存是动态变化的，可能它们之间会有细微变化。

4. 参考代码分析

获取系统的物理内存信息的参考代码如下。

```
1 #include <linux/version.h>
2 #include <linux/module.h>
3 #include <linux/init.h>
```

奔跑吧 linux 社区出品

```

4 #include <linux/mm.h>
5
6 #define PRT(a, b) pr_info("%-15s=%ld %10ld %8ld\n", \
7                           a, b, (PAGE_SIZE*b)/1024, (PAGE_SIZE*b)/1024/1024)
8
9 static int __init my_init(void)
10{
11    struct page *p;
12    unsigned long i, pfn, valid = 0;
13    int free = 0, locked = 0, reserved = 0, swapcache = 0,
14        referenced = 0, slab = 0, private = 0, uptodate = 0,
15        dirty = 0, active = 0, writeback = 0, mappedtodisk = 0;
16
17    unsigned long num_physpages;
18
19    num_physpages = get_num_physpages();
20    for (i = 0; i < num_physpages; i++) {
21
22        /* Most of ARM systems have ARCH_PFN_OFFSET */
23        pfn = i + ARCH_PFN_OFFSET;
24        /* may be holes due to remapping */
25        if (!pfn_valid(pfn))
26            continue;
27
28        valid++;
29        p = pfn_to_page(pfn);
30        if (!p)
31            continue;
32        /* page_count(page) == 0 is a free page. */
33        if (!page_count(p)) {
34            free++;
35            continue;
36        }
37        if (PageLocked(p))
38            locked++;
39        if (PageReserved(p))
40            reserved++;
41        if (PageSwapCache(p))
42            swapcache++;
43        if (PageReferenced(p))
44            referenced++;
45        if (PageSlab(p))
46            slab++;
47        if (PagePrivate(p))
48            private++;
49        if (PageUptodate(p))
50            uptodate++;
51        if (PageDirty(p))
52            dirty++;
53        if (PageActive(p))
54            active++;
55        if (PageWriteback(p))
56            writeback++;
57        if (PageMappedToDisk(p))
58            mappedtodisk++;
59    }
60
61    pr_info("\nExamining %ld pages (num_phys_pages) = %ld MB\n",
62           num_physpages, num_physpages * PAGE_SIZE / 1024 / 1024);
63    pr_info("Pages with valid PFN's=%ld, = %ld MB\n", valid,
64           valid * PAGE_SIZE / 1024 / 1024);
65    pr_info("\n          Pages      KB      MB\n\n");
66
67    PRT("free", free);

```

7.2 实验 2：获取系统的物理内存信息

```

68      PRT("locked", locked);
69      PRT("reserved", reserved);
70      PRT("swapcache", swapcache);
71      PRT("referenced", referenced);
72      PRT("slab", slab);
73      PRT("private", private);
74      PRT("uptodate", uptodate);
75      PRT("dirty", dirty);
76      PRT("active", active);
77      PRT("writeback", writeback);
78      PRT("mappedtodisk", mappedtodisk);
79
80      return 0;
81}
82
83static void __exit my_exit(void)
84{
85      pr_info("Module exit\n");
86}
87
88module_init(my_init);
89module_exit(my_exit);
90
91MODULE_AUTHOR("Ben Shushu");
92MODULE_LICENSE("GPL v2");

```

第 19 行，`get_num_physpages()`函数获取系统所有内存的大小，返回物理内存的页面数量。

第 20~59 行，`for` 循环遍历系统中所有内存的物理页面，然后进行各种统计。

第 23 行，对于 ARM32 处理器来说，`ARCH_PFN_OFFSET` 宏表示物理内存在地址空间的起始地址对应的页帧号。

第 25 行，`pfn_valid()`函数检查页帧号 `pfn` 是否有效。

第 29 行，`pfn_to_page()`函数表示从页帧号 `pfn` 转换到 `struct page` 数据结构 `p`。

第 33 行，`page_count()`等于 0 的话，说明这个页面是空闲页面。

第 37 行，`PageLocked()`表示该页面已经上锁。

第 39 行，`PageReserved()`表示该页不可被换出。

第 41 行，`PageSwapCache()`表示这是交换页面。

第 43 行，`PageReferenced()`表示该页来实现 LRU 算法中的第二次机会法。

第 45 行，`PageSlab()`表示该页属于由 slab 分配器创建的 slab。

第 47 行，`PagePrivate()`表示该页是有效的，当 `page->private` 包含有效值时会设置该标志位。如果页面是 `pagecache`，那么包含一些文件系统相关的数据信息。

第 49 行，`PageUptodate()`表示页面内容是有效的，当该页面上的读操作完成后，设置该标志位。

第 51 行，`PageDirty()`表示页面内容被修改过，为脏页。

第 53 行，`PageActive()`表示该页在活跃 LRU 链表中。

第 55 行，`PageWriteback()`表示该页页面正在回写。

第 57 行，`PageMappedToDisk()`表示在磁盘中分配了 blocks。

7.3 实验 3：分配内存

1. 实验目的

理解 Linux 内核中分配内存常用的接口函数的使用方法和实现原理等。

2. 实验要求

(1) 分配页面

写一个内核模块，然后在 QEMU 上运行 ARM Cortex-A9 的机器上实验。使用 alloc_page() 函数分配一个物理页面，然后输出该物理页面的物理地址，并输出该物理页面在内核空间的虚拟地址，然后把这个物理页面全部填充为 0x55。

思考一下，如果使用 GFP_KERNEL 或者 GFP_HIGHUSER_MOVABLE 为分配掩码，会有什么不一样？

(2) 尝试分配最大的内存

写一个内核模块，然后在 QEMU 上运行 ARM Cortex-A9 的机器上实验。测试可以动态分配多大的物理内存块，使用 __get_free_pages() 函数去分配。可以从分配一个物理页面开始，一直加大分配页面的数量，然后看看当前系统最大可以分配多少个连续的物理页面。

注意，使用 GFP_ATOMIC 分配掩码，并思考如何使用该分配掩码。

同样使用 kmalloc() 函数去测试可以分配多大的内存。

3. 实验步骤

进入本实验参考代码。

```
#cd  
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab  
3_alloc_mm  
  
#export ARCH=arm  
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

7.3 实验 3：分配内存

```
rlk@ubuntu:lab3_alloc_mm$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/running
linuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab3_alloc_mm modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_
_mm/lab3_alloc_mm/alloc_mm.o
    LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_
_mm/lab3_alloc_mm/allocmm.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_
_mm/lab3_alloc_mm/allocmm.mod.o
    LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_
_mm/lab3_alloc_mm/allocmm.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab3_alloc_mm$
```

把编译好的内核模块拷贝到 kmodules 目录。

```
rlk@ubuntu:lab3_alloc_mm$ cp allocmm.ko /home/rnk/rnk_basic/runninglinuxkernel_4
.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
/mnt # insmod allocmm.ko
[ 54.913328] order= 0, pages= 1, size= 4096
[ 54.923242] ... __get_free_pages OK
[ 54.926839] order= 1, pages= 2, size= 8192
[ 54.928285] ... __get_free_pages OK
[ 54.930036] order= 2, pages= 4, size= 16384
[ 54.930579] ... __get_free_pages OK
[ 54.931095] order= 3, pages= 8, size= 32768
[ 54.931468] ... __get_free_pages OK
[ 54.931809] order= 4, pages= 16, size= 65536
[ 54.932342] ... __get_free_pages OK
[ 54.932602] order= 5, pages= 32, size= 131072
[ 54.932829] ... __get_free_pages OK
[ 54.933077] order= 6, pages= 64, size= 262144
[ 54.934381] ... __get_free_pages OK
[ 54.934900] order= 7, pages= 128, size= 524288
[ 54.935659] ... __get_free_pages OK
[ 54.936268] order= 8, pages= 256, size= 1048576
[ 54.938194] ... __get_free_pages OK
```

本实验的 log 比较长，我们可以分成三部分。

第一部分是测试 __get_free_pages() 函数的分配，下面是这部分的 log。我们可以看到，最大分配的 order 为 10。

```
/mnt # insmod allocmm.ko
[ 54.913328] order= 0, pages= 1, size= 4096
[ 54.923242] ... __get_free_pages OK
[ 54.926839] order= 1, pages= 2, size= 8192
[ 54.928285] ... __get_free_pages OK
[ 54.930036] order= 2, pages= 4, size= 16384
[ 54.930579] ... __get_free_pages OK
[ 54.931095] order= 3, pages= 8, size= 32768
```

奔跑吧 linux 社区出品

```
[ 54.931468] ... __get_free_pages OK
[ 54.931809] order= 4, pages= 16, size= 65536
[ 54.932342] ... __get_free_pages OK
[ 54.932602] order= 5, pages= 32, size= 131072
[ 54.932829] ... __get_free_pages OK
[ 54.933077] order= 6, pages= 64, size= 262144
[ 54.934381] ... __get_free_pages OK
[ 54.934900] order= 7, pages= 128, size= 524288
[ 54.935659] ... __get_free_pages OK
[ 54.936268] order= 8, pages= 256, size= 1048576
[ 54.938194] ... __get_free_pages OK
[ 54.938983] order= 9, pages= 512, size= 2097152
[ 54.939837] ... __get_free_pages OK
[ 54.941130] order=10, pages= 1024, size= 4194304
[ 54.942075] ... __get_free_pages OK
```

第二部分是测试 kmalloc，下面是这部分的 log。我们可以看到，最大分配的 order 为 10。

```
[ 54.942782] order= 0, pages= 1, size= 4096
[ 54.945105] ... kmalloc OK
[ 54.945596] order= 1, pages= 2, size= 8192
[ 54.946225] ... kmalloc OK
[ 54.947163] order= 2, pages= 4, size= 16384
[ 54.949112] ... kmalloc OK
[ 54.950433] order= 3, pages= 8, size= 32768
[ 54.952561] ... kmalloc OK
[ 54.953062] order= 4, pages= 16, size= 65536
[ 54.954482] ... kmalloc OK
[ 54.955007] order= 5, pages= 32, size= 131072
[ 54.956295] ... kmalloc OK
[ 54.957333] order= 6, pages= 64, size= 262144
[ 54.959559] ... kmalloc OK
[ 54.960676] order= 7, pages= 128, size= 524288
[ 54.962489] ... kmalloc OK
[ 54.962868] order= 8, pages= 256, size= 1048576
[ 54.963540] ... kmalloc OK
[ 54.964164] order= 9, pages= 512, size= 2097152
[ 54.965170] ... kmalloc OK
[ 54.965779] order=10, pages= 1024, size= 4194304
[ 54.967066] ... kmalloc OK
```

第三部分是测试 vmalloc，下面这部分的 log。

```
[ 54.968737] pages= 1024, size= 4
[ 55.009518] ... vmalloc OK
[ 55.023468] pages= 2048, size= 8
[ 55.046206] ... vmalloc OK
[ 55.053499] pages= 3072, size= 12
[ 55.076059] ... vmalloc OK
[ 55.086892] pages= 4096, size= 16
[ 55.142324] ... vmalloc OK
[ 55.164577] pages= 5120, size= 20
[ 55.200219] ... vmalloc OK
[ 55.217677] pages= 6144, size= 24
[ 55.250773] ... vmalloc OK
[ 55.271889] pages= 7168, size= 28
[ 55.310073] ... vmalloc OK
[ 56.925863] pages= 8192, size= 32
[ 56.959818] ... vmalloc OK
[ 56.987975] pages= 9216, size= 36
[ 57.032031] ... vmalloc OK
```

7.3 实验 3：分配内存

```

[ 57.058916] pages= 10240, size=      40
[ 57.111032] ... vmalloc OK
[ 58.876972] pages= 11264, size=      44
[ 58.949882] ... vmalloc OK
[ 58.984803] pages= 12288, size=      48
[ 59.037043] ... vmalloc OK
[ 59.075134] pages= 13312, size=      52
[ 59.136635] insmod invoked oom-killer: gfp_mask=0x2d2, order=0,
oom_score_adj=0
[ 59.137339] insmod cpuset=/ mems_allowed=0
[ 59.138925] CPU: 1 PID: 775 Comm: insmod Tainted: G          O  4.0.0+ #3
[ 59.139668] Hardware name: ARM-Versatile Express
[ 59.141321] [<c002489c>] (unwind_backtrace) from [<c001d76c>]
(show_stack+0x2c/0x38)
[ 59.142362] [<c001d76c>] (show_stack) from [<c059b264>]
(__dump_stack+0x1c/0x24)
[ 59.143386] [<c059b264>] (__dump_stack) from [<c059b33c>]
(dump_stack+0xd0/0xf8)
[ 59.144290] [<c059b33c>] (dump_stack) from [<c01a8414>]
(dump_header+0x104/0x158)
[ 59.145142] [<c01a8414>] (dump_header) from [<c01a8a04>]
(oom_kill_process+0x208/0xa48)
[ 59.146730] [<c01a8a04>] (oom_kill_process) from [<c01a9c08>]
(__out_of_memory+0x410/0x43c)
[ 59.147536] [<c01a9c08>] (__out_of_memory) from [<c01a9c98>]
(out_of_memory+0x64/0x88)
[ 59.148404] [<c01a9c98>] (out_of_memory) from [<c01b0b14>]
(__alloc_pages_nodemask+0xed8/0x1208)
[ 59.149703] [<c01b0b14>] (__alloc_pages_nodemask) from [<c021a374>]
(__vmalloc_area_node+0x218/0x3c8)
[ 59.151771] [<c021a374>] (__vmalloc_area_node) from [<c021a5e0>]
(__vmalloc_node_range+0xbc/0x124)
[ 59.153943] [<c021a5e0>] (__vmalloc_node_range) from [<c021a6b0>]
(__vmalloc_node+0x68/0x78)
[ 59.154781] [<c021a6b0>] (__vmalloc_node) from [<c021a764>]
(vmalloc+0x5c/0x70)
[ 59.156645] [<c021a764>] (vmalloc) from [<bf002184>] (my_init+0x184/0x1f4
[allocmm])
[ 59.157417] [<bf002184>] (my_init [allocmm]) from [<c0008dc8>]
(do_one_initcall+0x68/0x190)
[ 59.158217] [<c0008dc8>] (do_one_initcall) from [<c0119d5c>]
(do_init_module+0xb4/0x278)
[ 59.159474] [<c0119d5c>] (do_init_module) from [<c011a710>]
(load_module+0x3ec/0x570)
[ 59.161846] [<c011a710>] (load_module) from [<c011a93c>]
(SyS_init_module+0xa8/0xc0)
[ 59.162641] [<c011a93c>] (SyS_init_module) from [<c0014d40>]
(ret_fast_syscall+0x0/0x34)
[ 59.163511] Mem-info:
[ 59.164095] Normal per-cpu:
[ 59.164521] CPU 0: hi: 18, btch: 3 usd: 13
[ 59.166229] CPU 1: hi: 18, btch: 3 usd: 10
[ 59.167560] CPU 2: hi: 18, btch: 3 usd: 14
[ 59.168698] CPU 3: hi: 18, btch: 3 usd: 17
[ 59.170683] active_anon:565 inactive_anon:1066 isolated_anon:0
[ 59.170683] active_file:0 inactive_file:0 isolated_file:0
[ 59.170683] unevictable:0 dirty:0 writeback:0 unstable:0
[ 59.170683] free:287 slab_reclaimable:1014 slab_unreclaimable:4846
[ 59.170683] mapped:343 shmem:1596 pagetables:10 bounce:0
[ 59.170683] free_cma:0
[ 59.174133] Normal free:1148kB min:1156kB low:1444kB high:1732kB
active_anon:2260kB inactive_anon:4264kB active_file:0kB inactive_file:0kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:102400kB
managed:87364kB mlocked:0kB dirty:0kB writeback:0kB mapped:1372kB

```

奔跑吧 linux 社区出品

```

shmem:6384kB slab_reclaimable:4056kB slab_unreclaimable:19384kB
kernel_stack:768kB pagetables:40kB unstable:0kB bounce:0kB free_cma:0kB
writeback_tmpr:0kB pages_scanned:0 all_unreclaimable? yes
[ 59.178727] lowmem_reserve[]: 0 0
[ 59.179231] Normal: 1*4kB (R) 1*8kB (R) 1*16kB (R) 1*32kB (R) 1*64kB (R)
0*128kB 0*256kB 0*512kB 1*1024kB (R) 0*2048kB 0*4096kB = 1148kB
[ 59.183331] 1596 total pagecache pages
[ 59.183850] 0 pages in swap cache
[ 59.184289] Swap cache stats: add 0, delete 0, find 0/0
[ 59.184904] Free swap = 0kB
[ 59.185533] Total swap = 0kB
[ 59.191947] 25600 pages of RAM
[ 59.192366] 500 free pages
[ 59.192685] 3759 reserved pages
[ 59.193030] 3386 slab pages
[ 59.193312] 845 pages shared
[ 59.193641] 0 pages swap cached
[ 59.193978] [ pid ]   uid  tgid total_vm      rss nr_ptes nr_pmds swapents
oom score adj name
[ 59.194933] [ 772]     0    772      577      1      3      0      0
0 sh
[ 59.196272] [ 775]     0    775      577      1      3      0      0
0 insmod
[ 59.197572] Out of memory: Kill process 772 (sh) score 0 or sacrifice child
[ 59.199858] Killed process 775 (insmod) total-vm:2308kB, anon-rss:4kB,
file-rss:0kB
[ 59.205632] vmalloc: allocation failure, allocated 53760000 of 54530048
bytes
[ 59.206354] insmod: page allocation failure: order:0, mode:0xd2
[ 59.207036] CPU: 1 PID: 775 Comm: insmod Tainted: G          O  4.0.0+ #3
[ 59.207694] Hardware name: ARM-Versatile Express
[ 59.208247] [<c002489c>] (unwind_backtrace) from [<c001d76c>]
(show_stack+0x2c/0x38)
[ 59.209283] [<c001d76c>] (show_stack) from [<c059b264>]
(__dump_stack+0x1c/0x24)
[ 59.211742] [<c059b264>] (__dump_stack) from [<c059b33c>]
(dump_stack+0xd0/0xf8)
[ 59.213470] [<c059b33c>] (dump_stack) from [<c01af350>]
(warn_alloc_failed+0x1a8/0x1e4)
[ 59.214271] [<c01af350>] (warn_alloc_failed) from [<c021a500>]
(__vmalloc_area_node+0x3a4/0x3c8)
[ 59.215135] [<c021a500>] (__vmalloc_area_node) from [<c021a5e0>]
(__vmalloc_node_range+0xbc/0x124)
[ 59.217140] [<c021a5e0>] (__vmalloc_node_range) from [<c021a6b0>]
(__vmalloc_node+0x68/0x78)
[ 59.218792] [<c021a6b0>] (__vmalloc_node) from [<c021a764>]
(vmalloc+0x5c/0x70)
[ 59.219864] [<c021a764>] (vmalloc) from [<bf002184>] (my_init+0x184/0x1f4
[allocmm])
[ 59.220866] [<bf002184>] (my_init [allocmm]) from [<c0008dc8>]
(do_one_initcall+0x68/0x190)
[ 59.221636] [<c0008dc8>] (do_one_initcall) from [<c0119d5c>]
(do_init_module+0xb4/0x278)
[ 59.222530] [<c0119d5c>] (do_init_module) from [<c011a710>]
(load_module+0x3ec/0x570)
[ 59.223242] [<c011a710>] (load_module) from [<c011a93c>]
(SyS_init_module+0xa8/0xc0)
[ 59.223971] [<c011a93c>] (SyS_init_module) from [<c0014d40>]
(ret_fast_syscall+0x0/0x34)
[ 59.224820] Mem-info:
[ 59.225166] Normal per-cpu:
[ 59.226106] CPU 0: hi: 18, btch: 3 usd: 13
[ 59.226787] CPU 1: hi: 18, btch: 3 usd: 8

```

7.3 实验 3：分配内存

```
[ 59.226932] CPU    2: hi: 18, btch: 3 usd: 14
[ 59.227426] CPU    3: hi: 18, btch: 3 usd: 17
[ 59.227926] active_anon:565 inactive_anon:1066 isolated_anon:0
[ 59.227926] active_file:0 inactive_file:0 isolated_file:0
[ 59.227926] unevictable:0 dirty:0 writeback:0 unstable:0
[ 59.227926] free:0 slab_reclaimable:1014 slab_unreclaimable:4846
[ 59.227926] mapped:343 shmem:1596 pagetables:10 bounce:0
[ 59.227926] free_cma:0
[ 59.233203] Normal free:0kB min:1156kB low:1444kB high:1732kB
active_anon:2260kB inactive_anon:4264kB active_file:0kB inactive_file:0kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:102400kB
managed:87364kB mlocked:0kB dirty:0kB writeback:0kB mapped:1372kB
shmem:6384kB slab_reclaimable:4056kB slab_unreclaimable:19384kB
kernel_stack:768kB pagetables:40kB unstable:0kB bounce:0kB free_cma:0kB
writeback_tmp:0kB pages_scanned:0 all_unreclaimable? yes
[ 59.237655] lowmem_reserve[]: 0 0 0
[ 59.238120] Normal: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB
0*512kB 0*1024kB 0*2048kB 0*4096kB = 0kB
[ 59.238985] 1596 total pagecache pages
[ 59.239602] 0 pages in swap cache
[ 59.240415] Swap cache stats: add 0, delete 0, find 0/0
[ 59.241165] Free swap = 0kB
[ 59.241454] Total swap = 0kB
[ 59.248906] 25600 pages of RAM
[ 59.249572] 211 free pages
[ 59.250228] 3759 reserved pages
[ 59.250733] 3386 slab pages
[ 59.251049] 845 pages shared
[ 59.251373] 0 pages swap cached
[ 61.808358] ... vmalloc failed
Killed
/mnt #
```

在测试 `vmalloc` 时，使用 `vmalloc` 函数进行分配，每次分配内存的大小增加 4MB，直到分配失败为止。当我们发现当要分配 52MB 大小内存时候，发生了内存短缺情况，也就是 OOM，并且调用 `oom-killer` 内核模块。

4. 参考代码

尝试最大内存分配的参考代码如下。

```
1 #include <linux/module.h>
2 #include <linux/slab.h>
3 #include <linux/init.h>
4 #include <linux/vmalloc.h>
5
6 static int mem = 64;
7
8 #define MB (1024*1024)
9
10static int __init my_init(void)
11{
12    char *kbuf;
13    unsigned long order;
14    unsigned long size;
15    char *vm_buff;
16
17    /* try __get_free_pages__ */
18    for (size = PAGE_SIZE, order = 0; order < MAX_ORDER;
```

奔跑吧 linux 社区出品

```

19         order++, size *= 2) {
20             pr_info(" order=%2lu, pages=%5lu, size=%8lu ", order,
21                     size / PAGE_SIZE, size);
22             kbuf = (char *)__get_free_pages(GFP_ATOMIC, order);
23             if (!kbuf) {
24                 pr_err("... __get_free_pages failed\n");
25                 break;
26             }
27             pr_info("... __get_free_pages OK\n");
28             free_pages((unsigned long)kbuf, order);
29         }
30
31     /* try kmalloc */
32     for (size = PAGE_SIZE, order = 0; order < MAX_ORDER;
33          order++, size *= 2) {
34         pr_info(" order=%2lu, pages=%5lu, size=%8lu ", order,
35                 size / PAGE_SIZE, size);
36         kbuf = kmalloc((size_t) size, GFP_ATOMIC);
37         if (!kbuf) {
38             pr_err("... kmalloc failed\n");
39             break;
40         }
41         pr_info("... kmalloc OK\n");
42         kfree(kbuf);
43     }
44
45     /* try vmalloc */
46     for (size = 4 * MB; size <= mem * MB; size += 4 * MB) {
47         pr_info(" pages=%6lu, size=%8lu ", size / PAGE_SIZE, size /
MB);
48         vm_buff = vmalloc(size);
49         if (!vm_buff) {
50             pr_err("... vmalloc failed\n");
51             break;
52         }
53         pr_info("... vmalloc OK\n");
54         vfree(vm_buff);
55     }
56
57     return 0;
58}
59
60static void __exit my_exit(void)
61{
62    pr_info("Module exit\n");
63}
64
65module_init(my_init);
66module_exit(my_exit);
67
68MODULE_AUTHOR("Ben ShuShu");
69MODULE_LICENSE("GPL v2");

```

第 18~29 行，通过 for 循环让 order 从 0 一直到 MAX_ORDER 不断的加大，测试 __get_free_pages() 函数最大能分配的内存数。

第 32~43 行，通过 for 循环让 order 从 0 一直到 MAX_ORDER 不断的加大，测试 kmalloc() 函数最大能分配的内存数。

第 46~55 行，测试当前系统，vmlloc 能分配的最大内存，从 4MB 开始，每次增大 4MB，直到 64MB 为止。

7.4 实验 4 : slab

7.4 实验 4: slab

1. 实验目的

了解和熟悉使用 slab 机制分配内存，并理解 slab 机制的原理。

2. 实验要求

(1) 编写一个内核模块

创建名为“mycache”的 slab 描述符，大小为 20 字节，align 为 8 字节，flags 为 0。然后从这个 slab 描述符中分配一个空闲对象。

(2) 查看系统当前的所有的 slab

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab
4_slab

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rwk@figo-OptiPlex-9020:lab4_slab$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0/ M=/home/rwk/rwk_basic/runninglinuxkernel_4
.0/rwk_lab/rwk_basic/chapter_7_mm/lab4_slab modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab4_slab/
slab.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab4_slab/
slab_lab.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab4_slab/
slab_lab.mod.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab4_slab/
slab_lab.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rwk@figo-OptiPlex-9020:lab4_slab$
```

拷贝内核模块到 kmodues 目录。

```
#cp slab_lab.ko /home/rwk/rwk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rwk/rwk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
/ # cd /mnt/
/mnt # ls
README          slab_lab.ko
/mnt # insmod slab_lab.ko
[ 40.147473] create mycache correctly
[ 40.149759] successfully created a object, kbuf_addr=0xc4540000
/mnt #
```

查看/proc/slabinfo 信息。

```
/mnt # cat /proc/slabinfo
slabinfo - version: 2.1
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : sl
abdata <active_slabs> <num_slabs> <sharedavail>
mycache        1   21  [192] 21  1 : tunables    0   0   0 : slabdata    1   1   0
ubi_wl_entry_slab  0   0   192  21  1 : tunables    0   0   0 : slabdata    0   0   0
ubifs_inode_slab  0   0   592  27  4 : tunables    0   0   0 : slabdata    0   0   0
isp1760_gh      0   0   208  19  1 : tunables    0   0   0 : slabdata    0   0   0
isp1760_qtd     0   0   208  19  1 : tunables    0   0   0 : slabdata    0   0   0
isp1760_urb_listitem  0   0   184  22  1 : tunables    0   0   0 : slabdata    0   0   0
sd_ext_cdb      2   19   208  19  1 : tunables    0   0   0 : slabdata    1   1   0
virtio_scsi_cmd 64   84   384  21  2 : tunables    0   0   0 : slabdata    4   4   0
v9fs_inode_cache 1   30   536  30  4 : tunables    0   0   0 : slabdata    1   1   0
jffs2_inode_cache 0   0   200  20  1 : tunables    0   0   0 : slabdata    0   0   0
jffs2_node_frag  0   0   200  20  1 : tunables    0   0   0 : slabdata    0   0   0
jffs2_refblock   0   0   424  19  2 : tunables    0   0   0 : slabdata    0   0   0
jffs2_tmp_dnode  0   0   208  19  1 : tunables    0   0   0 : slabdata    0   0   0
```

其中：

- name: 表示该 slab 对象的名称
- active_objs: 活跃对象的个数
- num_objs: 对象的个数
- objsize: 对象的大小
- objperslab: 表示一个 slab 中有多少个对象
- pagesperslab: 表示一个 slab 占用有多少个物理页面。
- tunables: 表示可调参数。对于使用 slab 分配器，我们可以设置这些可调参数，对于 slab 分配器，这些参数不可调。

另外也使用 slabinfo 命令来查看。

```
/mnt # slabinfo
Name          Objects Objsize   Space Slabs/Part/Cpu 0/S 0 %Fr %Ef Flg
anon_vma         9   44   4.0K   1/1/0 18 0 100 9 PZFU
anon_vma_chain   9   32   4.0K   1/1/0 19 0 100 7 PZFU
bdev_cache       1   456  16.3K   1/1/0 25 2 100 2 APaZFU
bio-0            2   124  8.1K   1/1/0 25 1 100 3 APZFU
bio_integrity_payload 2   100  8.1K   1/1/0 25 1 100 2 APZFU
biovec-256       4   3072 32.7K   1/1/0 10 3 100 37 APZFU
blkdev_queue     2   1040 32.7K   1/1/0 26 3 100 6 PZFU
blkdev_requests   8   208  8.1K   1/1/0 21 1 100 20 PZFU
cred_jar          52   92   49.1K   6/5/0 25 1 83 9 APZFU
dentry           8506  136  2.6M   328/1/0 26 1 0 43 PaZFU
files_cache       6   256  8.1K   1/1/0 18 1 100 18 APZFU
filp              58   168  32.7K   4/2/0 21 1 50 29 APZFU
fs_cache           6   36   4.0K   1/1/0 16 0 100 5 APZFU
ftrace_event_field 1469   32   319.4K  78/1/0 19 0 1 14 PZFU
ftrace_event_file  498   48   114.6K  28/1/0 18 0 3 20 PZFU
idr_layer_cache   192   1068 294.9K   9/4/0 26 3 44 69 PZFU
inode_cache        7953   312  4.0M   498/1/0 16 1 0 60 PaZFU
kernfs_node_cache 13431   80   3.4M   840/1/0 16 0 0 31 PZFU
kmalloc-1024       78   1024 98.3K   3/0/0 26 3 0 81 PZFU
kmalloc-128         904   128  303.1K  37/1/0 25 1 2 38 PZFU
kmalloc-192         140   192  57.3K   7/1/0 21 1 14 46 PZFU
```

7.4 实验 4 : slab

读者可以使用 grep 命令来快速查找。

```
/mnt #  
/mnt # slabinfo | grep mycache  
mycache      1      20      4.0K      1/1/0    21 0 100  0 APZFU
```

slabinfo 命令显示了当前系统所有 slab 的信息。该 slabinfo 命令的源代码是在: /home/rllk/rllk_basic/runninglinuxkernel_4.0/tools/vm/slabinfo.c 文件。

第 1 列: name 显示 slab 的名称

第 2 列: Objects 表示对象的个数。

第 3 列: Objsize 表示对象的大小

第 4 列: Space 表示一个 slab 占用的内存大小。

第 5 列: Slabs/Part/Cpu, 其中 slabs 表示有多少个 slab, part 表示

第 6 列: O/S 表示一个 slab 中有多少个对象。

新创建 slab 缓存时会在 sysfs 虚拟文件系统 (/sys/kernel/slab) 中新建一个对应的目录。

```
/sys/kernel/slab # ls  
PING          ftrace_event_file  
RAW           idr_layer_cache  
TCP           inet_peer_cache  
UDP           inode_cache  
UDP-Lite      inotify_inode_mark  
UNIX          ip4 frags  
anon_vma      ip_dst_cache  
                           mycache  
                           names cache  
                           nfs_commit_data  
                           nfs_direct_cache  
                           nfs_inode_cache  
                           nfs_page  
                           nfs_read_data
```

进入 /sys/kernel/slab/mycache 目录, 我们可以查看该 slab 缓存中众多的参数。

```
/sys/kernel/slab/mycache # ls  
aliases      deactivate_bypass     objs_per_slab  
align        deactivate_empty     order  
alloc_calls  deactivate_full      order_fallback  
alloc_fastpath deactivate_remote_frees partial  
alloc_from_partial deactivate_to_head poison  
alloc_node_mismatch deactivate_to_tail reclaim_account  
alloc_refill   destroy_by_rcu    red_zone  
alloc_slab    free_add_partial reserved  
alloc_slowpath free_calls       sanity_checks  
cmpxchg_double_cpu_fail free_fastpath shrink  
cmpxchg_double_fail   free_frozen    slab_size  
cpu_partial   free_remove_partial slabs  
cpu_partial_alloc free_slab      slabs_cpu_partial  
cpu_partial_drain free_slowpath store_user  
cpu_partial_free hwcache_align total_objects  
cpu_partial_node min_partial    trace  
cpu_slabs     object_size      validate  
cpuslab_flush objects  
ctor         objects_partial
```

这里给读者留两个问题:

1. 本实验的驱动代码中分配的 slab 对象大小为 20 字节, 为啥 /proc/slabinfo 中显

示 objsize 为 192 字节，而 slabinfo 命令显示的 Objsize 为 20 字节？

2. 为什么有的同学在 slabinfo 中找不到新创建的 slab？使用 “slabinfo | grep mycache” 找不到，而且在 /proc/slabinfo 中也找不到。

4. 参考代码

创建 slab 的代码如下。

```

1 #include <linux/module.h>
2 #include <linux/mm.h>
3 #include <linux/slab.h>
4 #include <linux/init.h>
5
6 static char *kbuf;
7 static int size = 20;
8 static struct kmem_cache *my_cache;
9 module_param(size, int, 0644);
10
11static int __init my_init(void)
12{
13    /* create a memory cache */
14    if (size > KMALLOC_MAX_SIZE) {
15        pr_err
16            (" size=%d is too large; you can't have more than %lu!\n",
17             size, KMALLOC_MAX_SIZE);
18        return -1;
19    }
20
21    my_cache = kmem_cache_create("mycache", size, 0,
22                                SLAB_HWCACHE_ALIGN, NULL);
23    if (!my_cache) {
24        pr_err("kmem_cache_create failed\n");
25        return -ENOMEM;
26    }
27    pr_info("create mycache correctly\n");
28
29    /* allocate a memory cache object */
30    kbuf = kmem_cache_alloc(my_cache, GFP_ATOMIC);
31    if (!kbuf) {
32        pr_err(" failed to create a cache object\n");
33        (void)kmem_cache_destroy(my_cache);
34        return -1;
35    }
36    pr_info(" successfully created a object, kbuf_addr=0x%p\n", kbuf);
37
38    return 0;
39}
40
41static void __exit my_exit(void)
42{
43    /* destroy a memory cache object */
44    kmem_cache_free(my_cache, kbuf);
45    pr_info("destroyed a cache object\n");
46
47    /* destroy the memory cache */
48    kmem_cache_destroy(my_cache);
49    pr_info("destroyed mycache\n");
50}
51
52module_init(my_init);

```

7.4 实验 4 : slab

```
53module_exit(my_exit);
54
55MODULE_LICENSE("GPL v2");
56MODULE_AUTHOR("Ben ShuShu");
```

第 25 行, `kmem_cache_create()`用来创建一个 slab 缓存。读者需要注意该函数每个形参的作用。`kmem_cache_create()`函数的原型实现在 `mm/slab_common.c` 文件中

```
struct kmem_cache *
kmem_cache_create(const char *name, size_t size, size_t align,
                  unsigned long flags, void (*ctor)(void *))
```

其中 `flags` 参数表示创建 slab 缓存的一些行为, 这些标志位定义在 `include/linux/slab.h` 头文件中。其中 `SLAB_HWCACHE_ALIGN` 表示 slab 对象的大小要和硬件高速缓存大小对齐。

```
/*
 * Flags to pass to kmem_cache_create().
 * The ones marked DEBUG are only valid if CONFIG_SLAB_DEBUG is set.
 */
#define SLAB_DEBUG_FREE      0x00000100UL /* DEBUG: Perform (expensive) checks
on free */
#define SLAB_RED_ZONE        0x00000400UL /* DEBUG: Red zone objs in a cache
*/
#define SLAB_POISON          0x00000800UL /* DEBUG: Poison objects */
#define SLAB_HWCACHE_ALIGN   0x00002000UL /* Align objs on cache lines */
#define SLAB_CACHE_DMA        0x00004000UL /* Use GFP_DMA memory */
#define SLAB_STORE_USER       0x00010000UL /* DEBUG: Store the last owner for
bug hunting */
#define SLAB_PANIC           0x00040000UL /* Panic if kmem_cache_create() fails */
```

第 30 行, `kmem_cache_alloc()`函数从刚才创建的 slab 缓存中分配一个对象。`kmem_cache_alloc()`函数的实现定义在 `mm/slub.c` 文件中。

```
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
```

读者需要注意这里的 `gfpflags` 和 `kmem_cache_create()` 函数中的 `flags` 是不一样的。这个 `gfpflags` 是用来控制 slab 机制中分配物理页面的分配掩码。这些分配掩码定义在 `include/linux/gfp.h` 头文件中。

```
/* This equals 0, but use constants in case they ever change */
#define GFP_NOWAIT (GFP_ATOMIC & ~__GFP_HIGH)
/* GFP_ATOMIC means both !wait (__GFP_WAIT not set) and use emergency pool */
#define GFP_ATOMIC (__GFP_HIGH)
#define GFP_NOIO  (__GFP_WAIT)
#define GFP_NOFS  (__GFP_WAIT | __GFP_IO)
#define GFP_KERNEL (__GFP_WAIT | __GFP_IO | __GFP_FS)
#define GFP_TEMPORARY (__GFP_WAIT | __GFP_IO | __GFP_FS | \
                    __GFP_RECLAMABLE)
#define GFP_USER   (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_HIGHUSER (GFP_USER | __GFP_HIGHMEM)
#define GFP_HIGHUSER_MOVABLE (GFP_HIGHUSER | __GFP_MOVABLE)
#define GFP_IOFS   (__GFP_IO | __GFP_FS)
#define GFP_TRANSHUGE (GFP_HIGHUSER_MOVABLE | __GFP_COMP | \
                     __GFP_NOMEMALLOC | __GFP_NORETRY | __GFP_NOWARN | \
                     __GFP_NO_KSWAPD)
```

我们最后来研究一下在实验步骤中留下来的两个问题。

1. 本实验的驱动代码中分配的 slab 对象大小为 20 字节, 为啥/proc/slabinfo 中显示 objsize 为 192 字节, 而 slabinfo 命令显示的 Objsize 为 20 字节?

提示: 读者需要注意在 struct kmem_cache 数据结构中有两个成员, 一个是 object_size, 表示对象原本的大小, 另外一个是 size, 表示 slab 分配器最终为每一个对象实际分配的大小。

```
<mm/slab.h>

struct kmem_cache {
    unsigned int object_size; /* The original size of the object */
    unsigned int size;        /* The aligned/padded/added on size */
    unsigned int align;       /* Alignment as calculated */
    unsigned long flags;     /* Active flags on the slab */
    const char *name;         /* Slab name for sysfs */
    int refcount;            /* Use counter */
    void (*ctor)(void *);   /* Called on object slot creation */
    struct list_head list;   /* List of all slab caches on the system */
};
```

通常 size 要大于或者等于 object_size。为什么呢?

我们看 kmem_cache_create() 函数就会发现, 有很多情况下需要把对象的大小设置更大一点。比如, SLAB_HWCACHE_ALIGN 标志位, 需要和高速缓存对齐, 见 calculate_alignment() 函数。

```
unsigned long calculate_alignment(unsigned long flags,
                                  unsigned long align, unsigned long size)
{
    if (flags & SLAB_HWCACHE_ALIGN) {
        unsigned long ralign = cache_line_size();
        while (size <= ralign / 2)
            ralign /= 2;
        align = max(align, ralign);
    }

    if (align < ARCH_SLAB_MINALIGN)
        align = ARCH_SLAB_MINALIGN;

    return ALIGN(alignment, sizeof(void *));
}
```

另外, 在 __kmem_cache_create() 函数里, 当创建 slab 缓存的 flags 里设置了 SLAB_RED_ZONE 时候, size 会变大。

```
if (flags & SLAB_RED_ZONE) {
    ralign = REDZONE_ALIGN;
    /* If redzoning, ensure that the second redzone is suitably
     * aligned, by adjusting the object size accordingly. */
    size += REDZONE_ALIGN - 1;
    size &= ~(REDZONE_ALIGN - 1);
}
```

2. 为什么有的同学在 slabinfo 中找不到新创建的 slab? 使用 “slabinfo | grep mycache” 找不到, 而且在/proc/slabinfo 中也找不到。

7.5 实验 5 : VMA

提示: 使用 `kmem_cache_create()` 函数创建 slab 缓存时会去检查要创建的这个 slab 缓存是否可以复用现有的 slab 缓存。如果可以复用的话, 那么就不会新创建 slab 缓存, 见 `_kmem_cache_alias()` 函数。

7.5 实验 5: VMA

1. 实验目的

理解进程地址空间的管理, 特别是理解 VMA 的相关操作。

2. 实验要求

编写一个内核模块。遍历一个用户进程中所有的 VMA, 并且打印这些 VMA 的属性信息, 比如 VMA 的大小, 起始地址等。

然后通过比较`/proc/pid/maps` 中显示的信息看看编写的内核模块是否正确。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab
5_vma

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@figo-OptiPlex-9020:lab5_vma$ make
make -C /home/r1k/r1k_basic/runninglinuxkernel_4.0/ M=/home/r1k/r1k_basic/runninglinuxkernel_4
/chapter_7_mm/lab5_vma modules;
make[1]: Entering directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'
  CC [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab5_vma/v
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab5_vma/v
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab5_vma/v
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab5_vma/v
make[1]: Leaving directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'
rlk@figo-OptiPlex-9020:lab5_vma$
```

拷贝内核模块到 `kmodules` 目录。

```
#cp vma_test.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端, 进入 `runninglinuxkernel_4.0` 目录, 启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
#.run.sh arm32
```

进入 `mnt` 目录, 安装本实验的内核模块。这个 `vma_test.ko` 内核模块可以设置内

核参数, pid。使用 top 命令来查看系统中所有的进程。

Mem: 43928K used, 43436K free, 6384K shrd, 0K buff, 6384K cached								
CPU: 0.0% usr 0.4% sys 0.0% nic 99.3% idle 0.0% io 0.0% irq 0.0% sirq								
Load average: 0.00 0.01 0.05 1/44 808								
PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
808	773	0	R	2308	2.6	1	0.5	top
11	2	0	SW	0	0.0	1	0.0	[ksoftirqd/1]
1	0	0	S	2308	2.6	0	0.0	{linuxrc} init
773	1	0	S	2308	2.6	0	0.0	-/bin/sh
754	2	0	SWN	0	0.0	3	0.0	[kmemleak]
411	2	0	SW	0	0.0	2	0.0	[kworker/2:1]
688	2	0	SW	0	0.0	3	0.0	[kworker/u8:2]
409	2	0	SW	0	0.0	1	0.0	[kworker/1:1]
7	2	0	SW	0	0.0	0	0.0	[rcu_sched]
3	2	0	SW	0	0.0	0	0.0	[ksoftirqd/0]
19	2	0	SW	0	0.0	3	0.0	[ksoftirqd/3]
15	2	0	SW	0	0.0	2	0.0	[ksoftirqd/2]
328	2	0	SW	0	0.0	0	0.0	[kworker/0:1]
23	2	0	SW	0	0.0	1	0.0	[kdevtmpfs]
2	0	0	SW	0	0.0	0	0.0	[kthreadd]
279	2	0	SW	0	0.0	3	0.0	[khungtaskd]
10	2	0	SW	0	0.0	1	0.0	[migration/1]
410	2	0	SW	0	0.0	3	0.0	[kworker/3:1]
9	2	0	SW	0	0.0	0	0.0	[migration/0]

我们选择使用 “/bin/sh” 进程，它的 PID 号为 773。

```
/mnt # insmod vma_test.ko pid=773
[16761.974726] Examining vma's for pid=773, command=sh
[16761.977899] mm_struct addr = 0xc483c240
[16761.978198] vmas:
[16761.978755]   1: c4893ad0 10000 207000 2060288
[16761.979285]   2: c4892840 216000 219000 12288
[16761.979642]   3: c48935a8 219000 23d000 147456
[16761.980003]   4: c4892948 bedc1000 bede3000 139264
[16761.980350]   5: c4892738 bef56000 bef57000 4096
/mnt #
```

加载 vma_test.ko 内核模块之后，打印这个进程所有的 VMA 区域。从上图可以看到，这个进程包含了 5 个 VMA 区域。

- 第 1 个 vma 区域，VMA 区域的 struct vm_area_struct 数据结构 vma 的地址。VMA 区域的起始地址为 0x10000，结束地址 0x207000，长度为 2060288 字节。
- 第 2 个 vma 区域，VMA 区域的 struct vm_area_struct 数据结构 vma 的地址。VMA 区域的起始地址为 0x216000，结束地址 0x219000，长度为 12288 字节。
- 第 3 个 vma 区域，VMA 区域的 struct vm_area_struct 数据结构 vma 的地址。VMA 区域的起始地址为 0x219000，结束地址 0x23d000，长度为 147456 字节。
- 第 4 个 vma 区域，VMA 区域的 struct vm_area_struct 数据结构 vma 的地址。VMA 区域的起始地址为 0xbecd1000，结束地址 0xbede3000，长度为 139264 字节。
- 第 5 个 vma 区域，VMA 区域的 struct vm_area_struct 数据结构 vma 的地址。VMA 区域的起始地址为 0xbef56000，结束地址 0xbef57000，长度为 4096 字节。

7.5 实验 5 : VMA

节。

Linux 系统的“proc”虚拟文件系统有记录每一个进程的各种信息。其中“/proc/pid/smaps”记录了进程地址空间的 VMA 区域的情况。使用 cat 命令来查看，并且和我们编写的驱动模块打印的信息进行对比和相互印证。

```
/mnt # cat /proc/773/smaps
00010000-00207000 r-xp 00000000 00:02 1130      /bin/busybox
Size:          2012 kB
Rss:           1180 kB
Pss:            510 kB
Shared_Clean:    0 kB
Shared_Dirty:   1052 kB
Private_Clean:   0 kB
Private_Dirty:   128 kB
Referenced:    1180 kB
Anonymous:      0 kB
AnonHugePages:   0 kB
Swap:           0 kB
KernelPageSize: 4 kB
MMUPageSize:    4 kB
Locked:          0 kB
VmFlags: rd ex mr mw me dw
00216000-00219000 rw-p 001f6000 00:02 1130      /bin/busybox
Size:          12 kB
Rss:           12 kB
Pss:            12 kB
Shared_Clean:    0 kB
Shared_Dirty:    0 kB
Private_Clean:   0 kB
Private_Dirty:   12 kB
Referenced:    12 kB
Anonymous:      12 kB
AnonHugePages:   0 kB
Swap:           0 kB
KernelPageSize: 4 kB
MMUPageSize:    4 kB
Locked:          0 kB
VmFlags: rd wr mr mw me dw ac
00219000-0023d000 rw-p 00000000 00:00 0          [heap]
Size:          144 kB
Rss:            48 kB
Pss:            48 kB
Shared_Clean:    0 kB
Shared_Dirty:    0 kB
Private_Clean:   0 kB
Private_Dirty:   48 kB
Referenced:    48 kB
Anonymous:      48 kB
AnonHugePages:   0 kB
Swap:           0 kB
KernelPageSize: 4 kB
MMUPageSize:    4 kB
Locked:          0 kB
VmFlags: rd wr mr mw me ac
bedc2000-bede3000 rw-p 00000000 00:00 0          [stack]
Size:          136 kB
Rss:            8 kB
Pss:            8 kB
Shared_Clean:    0 kB
```

```

Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         8 kB
Referenced:           8 kB
Anonymous:            8 kB
AnonHugePages:        0 kB
Swap:                 0 kB
KernelPageSize:       4 kB
MMUPageSize:          4 kB
Locked:               0 kB
VmFlags: rd wr mr mw me gd ac
bef56000-bef57000 r-xp 00000000 00:00 0          [sigpage]
Size:                 4 kB
Rss:                  0 kB
Pss:                  0 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           0 kB
Anonymous:            0 kB
AnonHugePages:        0 kB
Swap:                 0 kB
KernelPageSize:       4 kB
MMUPageSize:          4 kB
Locked:               0 kB
VmFlags: rd ex mr mw me de
fffff0000-fffff1000 r-xp 00000000 00:00 0          [vectors]
Size:                 4 kB
Rss:                  0 kB
Pss:                  0 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           0 kB
Anonymous:            0 kB
AnonHugePages:        0 kB
Swap:                 0 kB
KernelPageSize:       4 kB
MMUPageSize:          4 kB
Locked:               0 kB
VmFlags: rd ex mr me

```

4. 参考代码

VMA 实验的参考代码如下。

```

1 #include <linux/module.h>
2 #include <linux/init.h>
3 #include <linux/mm.h>
4 #include <linux/sched.h>
5
6 static int pid;
7 module_param(pid, int, S_IRUGO);
8
9 static void printit(struct task_struct *tsk)
10{
11    struct mm_struct *mm;
12    struct vm_area_struct *vma;
13    int j = 0;
14    unsigned long start, end, length;

```

7.5 实验 5 : VMA

```

15
16     mm = tsk->mm;
17     pr_info("mm = %p\n", mm);
18     vma = mm->mmap;
19
20     /*使用mmap_sem读写信号量进行保护 */
21
22     down_read(&mm->mmap_sem);
23     pr_info
24         ("vmas:           vma          start          end          length\n");
25
26     while (vma) {
27         j++;
28         start = vma->vm_start;
29         end = vma->vm_end;
30         length = end - start;
31         pr_info("%d: %16p %12lx %12lx  %8lx=%8ld\n",
32                 j, vma, start, end, length, length);
33         vma = vma->vm_next;
34     }
35     up_read(&mm->mmap_sem);
36}
37
38static int __init my_init(void)
39{
40    struct task_struct *tsk;
41    if (pid == 0) {
42        tsk = current;
43        pid = current->pid;
44    } else {
45        tsk = pid_task(find_vpid(pid), PIDTYPE_PID);
46    }
47    if (!tsk)
48        return -1;
49    pr_info(" Examining vma's for pid=%d, command=%s\n", pid, tsk->comm);
50    printit(tsk);
51    return 0;
52}
53
54static void __exit my_exit(void)
55{
56    pr_info("Module Unloading\n");
57}
58
59module_init(my_init);
60module_exit(my_exit);

```

第 38~52 行是内核模块的初始化函数 `my_init`。

第 41~46 行, `pid` 是内核模块的参数, 用户在加载内核模块时可传递参数(某个进程的 `pid`)进来。如果没有传递 `pid` 参数, 那么会使用当前进程。在本实验中, 当前进程就是 `insmod` 这个命令执行时的进程。

第 45 行, 从 `pid` 可以获取到进程的描述符 `struct task_struct` 数据结构。

第 49 行, 打印接下来我们要查看哪个进程的 `vma`。

第 50 行, 调用 `printit` 函数。

第 9~36 行, 就是本实验的核心函数。

第 16 行, 获取待检查进程的内存描述符 `struct mm_struct` 数据结构 `mm`。

第 18 行, mm 数据结构中有一个 VMA 链表的头, mm->mmap。

第 22 行, 接下来我们需要遍历 VMA 链表了, 所以我们需要使用 down_read() 函数来申请一个读者信号量。注意, 我们这里只是读取 VMA 链表, 不会修改这个链表, 所以申请读者类型的信号量就够了。若需要对 VMA 链表进行修改的话, 我们就需要申请写者类型的信号量。

第 26~34 行, 遍历 VMA 链表, 并对每个 VMA 打印其起始地址, 结束地址和长度等信息。

第 35 行, 释放读者信号量。

7.6 实验 6: mmap

1. 实验目的

理解 mmap 系统调用的使用方法以及实现原理。

2. 实验要求

1) 编写一个简单的字符设备程序。分配一段物理内存, 然后使用 mmap 方法把这段物理内存映射到进程地址空间中, 用户进程打开这个驱动程序之后就可以读写这段物理内存了。需要实现 mmap、read 和 write 方法。

2) 写一个简单的用户空间的测试程序, 来测试这个字符设备驱动, 比如测试 open、mmap、read 和 write 方法。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab
6

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@figo-OptiPlex-9020:lab6_mmap$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0/ M=/home/rwk/rwk_basic/runninglinuxkernel_4
.0/rwk_lab/rwk_basic/chapter_7_mm/lab6_mmap modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M] /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab6_mmap/
mydev_mmap.o
  LD [M] /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab6_mmap/
mydevdemo_mmap.o
      Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab6_mmap/
mydevdemo_mmap.mod.o
  LD [M] /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_7_mm/lab6_mmap/
mydevdemo_mmap.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rlk@figo-OptiPlex-9020:lab6_mmap$
```

7.6 实验 6 : mmap

拷贝内核模块到 kmodues 目录。

```
#cp mydevdemo_mmap.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

编译测试程序，并拷贝到 kmodues 目录中。

```
# arm-linux-gnueabi-gcc test.c -o test -static
#cp test /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
/mnt # insmod mydevdemo_mmap.ko
[ 246.747271] succeeded register char device: my_demo_dev
/mnt #
```

运行 test 测试程序。

```
/mnt # ./test
[ 255.941431] demodrv_open: major=10, minor=58
driver max buffer size=40960
[ 256.021414] demodrv_mmap: mapping 40960 bytes of device buffer at offset 0
mmap driver buffer succeeded: 0xb6f80000
[ 256.050262] demodrv_read: read nbytes=40960 done at pos=40960
data modify and compare succussful
/mnt #
```

4. 参考代码

驱动程序的参考代码如下。

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/uaccess.h>
4 #include <linux/init.h>
5 #include <linux/miscdevice.h>
6 #include <linux/device.h>
7 #include <linux/slab.h>
8 #include <linux/kfifo.h>
9
10 #define DEMO_NAME "my_demo_dev"
11 static struct device *mydemodrv_device;
12
13 /*virtual FIFO device's buffer*/
14 static char *device_buffer;
15 #define MAX_DEVICE_BUFFER_SIZE (10 * PAGE_SIZE)
16
17 #define MYDEV_CMD_GET_BUFSIZE 1 /* defines our IOCTL cmd */
18
19 static int demodrv_open(struct inode *inode, struct file *file)
20 {
21     int major = MAJOR(inode->i_rdev);
22     int minor = MINOR(inode->i_rdev);
23
24     printk("%s: major=%d, minor=%d\n", __func__, major, minor);
25
26     return 0;
}
```

奔跑吧 linux 社区出品

```

27 }
28
29 static int demodrv_release(struct inode *inode, struct file *file)
30 {
31     return 0;
32 }
33
34 static ssize_t
35 demodrv_read(struct file *file, char __user *buf, size_t count, loff_t
*ppos)
36 {
37     int nbytes =
38         simple_read_from_buffer(buf, count, ppos, device_buffer,
MAX_DEVICE_BUFFER_SIZE
                                         E);
39
40     printk("%s: read nbytes=%d done at pos=%d\n",
41           __func__, nbytes, (int)*ppos);
42
43     return nbytes;
44 }
45
46 static ssize_t
47 demodrv_write(struct file *file, const char __user *buf, size_t count,
loff_t *ppos)
48 {
49     int nbytes =
50         simple_write_to_buffer(device_buffer,
MAX_DEVICE_BUFFER_SIZE, ppos, buf, count
                                         );
51
52     printk("%s: write nbytes=%d done at pos=%d\n",
53           __func__, nbytes, (int)*ppos);
54
55     return nbytes;
56 }
57
58 static int
59 demodrv_mmap(struct file *filp, struct vm_area_struct *vma)
60 {
61     unsigned long pfn;
62     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
63     unsigned long len = vma->vm_end - vma->vm_start;
64
65     if (offset >= MAX_DEVICE_BUFFER_SIZE)
66         return -EINVAL;
67     if (len > (MAX_DEVICE_BUFFER_SIZE - offset))
68         return -EINVAL;
69
70     printk("%s: mapping %ld bytes of device buffer at offset %ld\n",
71           __func__, len, offset);
72
73     /* pfn = page_to_pfn (virt_to_page (ramdisk + offset)); */
74     pfn = virt_to_phys(device_buffer + offset) >> PAGE_SHIFT;
75     vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
76     if (remap_pfn_range(vma, vma->vm_start, pfn, len,
vma->vm_page_prot))
77         return -EAGAIN;
78
79     return 0;
80 }
81
82 static long
83 demodrv_unlocked_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg)
84 {

```

7.6 实验 6 : mmap

```

85     unsigned long tbs = MAX_DEVICE_BUFFER_SIZE;
86     void __user *ioargp = (void __user *)arg;
87
88     switch (cmd) {
89     default:
90         return -EINVAL;
91
92     case MYDEV_CMD_GET_BUFSIZE:
93         if (copy_to_user(ioargp, &tbs, sizeof(tbs)))
94             return -EFAULT;
95         return 0;
96     }
97 }
98
99 static const struct file_operations demodrv_fops = {
100     .owner = THIS_MODULE,
101     .open = demodrv_open,
102     .release = demodrv_release,
103     .read = demodrv_read,
104     .write = demodrv_write,
105     .mmap = demodrv_mmap,
106     .unlocked_ioctl = demodrv_unlocked_ioctl,
107 };
108
109static struct miscdevice mydemodrv_misc_device = {
110     .minor = MISC_DYNAMIC_MINOR,
111     .name = DEMO_NAME,
112     .fops = &demodrv_fops,
113 };
114
115static int __init simple_char_init(void)
116{
117     int ret;
118
119     device_buffer = kmalloc(MAX_DEVICE_BUFFER_SIZE, GFP_KERNEL);
120     if (!device_buffer)
121         return -ENOMEM;
122
123     ret = misc_register(&mydemodrv_misc_device);
124     if (ret) {
125         printk("failed register misc device\n");
126         kfree(device_buffer);
127         return ret;
128     }
129
130     mydemodrv_device = mydemodrv_misc_device.this_device;
131
132     printk("succeeded register char device: %s\n", DEMO_NAME);
133
134     return 0;
135 }
136
137static void __exit simple_char_exit(void)
138{
139     printk("removing device\n");
140
141     kfree(device_buffer);
142     misc_deregister(&mydemodrv_misc_device);
143 }
144
145module_init(simple_char_init);
146module_exit(simple_char_exit);
147
148MODULE_AUTHOR("Benshushu");

```

```
149MODULE_LICENSE("GPL v2");
150MODULE_DESCRIPTION("simpe character device");
```

本实验是基于第 5 章的实验代码修改过来的，接下来我们只看和 mmap 相关的部分。

第 99 行，每个字符设备驱动都需要实现一个设备文件操作方法集 struct file_operations。我们这个实验也不例外。本实验需要实现 mmap 方法集，因此在第 105 行，添加了读 mmap 方法的实现。

```
.mmap = demodrv_mmap,
```

实现 mmap 方法的函数是 demodrv_mmap，实现在第 59 行。

第 59 行。demodrv_mmap()函数有两个参数，一个是 filp，设备文件操作符，另外一个是 vma，表示要映射的用户空间的区间。vma 这个概念在本章实验 5 中已经有说明了。

读者可能会问，第一个参数 filp 好理解，那第二个参数 vma 是从哪里来的呢？

要弄明白这个问题，需要读懂 Linux 内核的缺页异常和 mmap 机制的实现。有精力的读者可以阅读蓝色版本《奔跑吧 Linux 内核》一书，第 2 章相关内容。

第 62 行，vma->vm_pgoff 表示在 VMA 区域中的偏移，通常这个值为 0。

第 63 行，len 表示 VMA 区域的长度。

第 65~68 行，做一些必要的检查。

第 74 行，本实验中 FIFO 设备的内存是 device_buffer，它在 init 时候通过 kmalloc 来分配的。kmalloc 分配的内存是物理内存，并且是线性映射的内存，而且是物理上连续的内存。那么我们可以通过 virt_to_phys()来查找到该物理内存的起始的页帧号。

读者需要注意，若 device_buffer 的大小不是以页对齐的话，那么 mmap 做映射的时候需要考虑起始地址对齐的问题，以及 buffer 大小和页对齐的问题。本实验设置的 buffer 大小是 $10 * \text{PAGE_SIZE}$ ，而在实际开发过程中，需要考虑对齐的问题。

第 75 行，pgprot_noncached()函数用来关闭 cache。cache 的属性设置是在 PTE 页表中，该函数根据体系结构不同的，来设置硬件 PTE 页表中关于 cache 的相关属性。在 ARM32 和 ARM64 中，它们的 PTE 页表就不相同，该函数的实现也有差异。

第 76 行，调用 remap_pfn_range()函数来完成用户空间虚拟内存和物理内存的映射关系。

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
                     unsigned long pfn, unsigned long size, pgprot_t prot)
```

remap_pfn_range()函数实现在 mm/memory.c 文件中，它的主要功能是把内核态的物理内存映射到用户空间。它一共有 5 个参数。

- vma：描述用户空间的虚拟内存

7.6 实验 6 : mmap

- **addr:** 要映射的用户空间虚拟内存的起始地址, 这个地址必须在 vma 区域里。
- **pfn:** 物理内存的起始页帧号
- **size:** 映射的大小
- **prot:** 映射的属性

测试程序的参考代码如下。

```

1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <fcntl.h>
8 #include <sys/ioctl.h>
9 #include <malloc.h>
10
11#define DEMO_DEV_NAME "/dev/my_demo_dev"
12
13#define MYDEV_CMD_GET_BUFSIZE 1 /* defines our IOCTL cmd */
14
15int main()
16{
17    int fd;
18    int i;
19    size_t len;
20    char message[] = "Testing the virtual FIFO device";
21    char *read_buffer, *mmap_buffer;
22
23    len = sizeof(message);
24
25    fd = open(DEMO_DEV_NAME, O_RDWR);
26    if (fd < 0) {
27        printf("open device %s failed\n", DEMO_DEV_NAME);
28        return -1;
29    }
30
31    if (ioctl(fd, MYDEV_CMD_GET_BUFSIZE, &len) < 0) {
32        printf("ioctl fail\n");
33        goto open_fail;
34    }
35
36    printf("driver max buffer size=%d\n", len);
37
38    read_buffer = malloc(len);
39    if (!read_buffer)
40        goto open_fail;
41
42    mmap_buffer = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, 0);
43    if (mmap_buffer == (char *)MAP_FAILED) {
44        printf("mmap driver buffer fail\n");
45        goto map_fail;
46    }
47
48    printf("mmap driver buffer succeeded: %p\n", mmap_buffer);
49
50    /* modify the mapped buffer */
51    for (i = 0; i < len; i++)
52        *(mmap_buffer + i) = (char)random();
53

```

```

54     /* read the buffer back and compare with the mmap buffer*/
55     if (read(fd, read_buffer, len) != len) {
56         printf("read fail\n");
57         goto read_fail;
58     }
59
60     if (memcmp(read_buffer, mmap_buffer, len)) {
61         printf("buffer compare fail\n");
62         goto read_fail;
63     }
64
65     printf("data modify and compare succussful\n");
66
67     munmap(mmap_buffer, len);
68     free(read_buffer);
69     close(fd);
70
71     return 0;
72
73read_fail:
74     munmap(mmap_buffer, len);
75map_fail:
76     free(read_buffer);
77open_fail:
78     close(fd);
79     return -1;
80
81}

```

第 25 行，打开设备文件。

第 31 行，获取 buffer 的大小。

第 38 行，分配一个读 buffer。

第 42 行，通过 mmap 函数来把设备驱动的 buffer 映射到用户空间 mmap_buffer。

第 51~52 行，修改设备驱动 buffer 的内容。

第 55 行，通过 read 方法，把设备 buffer 读到用户空间的读 buffer 中。

第 60 行，比较读 buffer 和 mmap_buffer 的数据是否完全相同。

5. 进阶思考

本实验在内核空间申请一个 buffer，然后把这个 buffer 映射到用户空间中。那么用户空间就可以读写这个内核空间申请的 buffer 的数据了。

读者可以深入思考两个问题：

1. 先来考察内核空间申请的 buffer。在本实验中使用 kmalloc()函数来分配内存。
 - 站在 CPU 角度来看，CPU 访问这个 buffer，这个 buffer 中的页（page）对应的物理内存和对应的虚拟内存分别指向哪里？它对应的 PTE 页表又在哪里？
 - 它的 cache 是打开还是关闭的？
 - 若这时候有一个硬件设备也需要来访问这个 buffer，比如硬件设备想通过 DMA 来访问我们这个 buffer，我们是否考虑 cache 的问题？CPU 和

7.7 实验 7：映射用户内存

DMA 同时访问这个 buffer，怎么办？如何关闭这个 buffer 的 cache。

- 内核中有哪些接口函数可以分配关闭 cache 的 buffer？
2. 我们来考察通过 mmap 映射到用户空间的这个 user buffer。
 - 这个 user buffer 的物理内存是在哪里？对应的虚拟内存是在哪里？
 - 对应的 PTE 页表项是在哪里？
 - CPU 访问这个 user buffer，它对应的 cache 是关闭还是打开的？

7.7 实验 7：映射用户内存

1. 实验目的

映射用户内存用于把用户空间的虚拟内存空间传到内核空间。内核空间为其分配物理内存并建立相应的映射关系，并且锁住（pin）这些物理内存。这种方法在很多驱动程序中非常常见，比如在 camera 驱动的 V4L2 核心架构中可以使用用户空间内存类型（V4L2_MEMORY_USERPTR）来分配物理内存，其驱动的实现使用的是 get_user_pages() 函数。

本实验尝试使用 get_user_pages() 函数来分配和锁住物理内存。

2. 实验要求

- 1) 编写一个简单的字符设备程序。使用 get_user_pages() 函数为用户空间传递下来的虚拟地址空间分配和锁住物理内存。
- 2) 写一个简单的用户空间的测试程序，来测试这个字符设备驱动。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab
7

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@ubuntu:lab7_pin_page$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab7_pin_page modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab7_pin_page/mydev_pin_page.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab7_pin_page/mydevdemo-pin-page.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab7_pin_page/mydevdemo-pin-page.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_7_mm/lab7_pin_page/mydevdemo-pin-page.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab7_pin_page$
```

拷贝内核模块到 kmodues 目录。

```
#cp mydevdemo-pin-page.ko
/home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

编译测试程序，并拷贝到 kmodues 目录中。

```
# rm-linux-gnueabi-gcc test_ok.c -o test_ok --static
#cp test_issue /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
/mnt # insmod mydevdemo-pin-page.ko
[ 175.570091] succeeded register char device: my_demo_dev
/mnt #
```

运行 test_issue 程序。

```
/mnt # ./test_issue
[ 261.413465] demodrv_open: major=10, minor=58
driver max buffer size=4096
[ 261.438500] demodrv_read_write: len=4096, npage=1
[ 261.439002] pin 1 pages from user done
[ 261.441131] demodrv_read_write: write user buffer 4096 bytes done
[ 261.441374] demodrv_write: write nbytes=4096 done at pos=0
[ 261.441769] demodrv_read_write: len=4096, npage=1
[ 261.442569] pin 1 pages from user done
[ 261.442957] demodrv_read_write: read user buffer 4096 bytes done
[ 261.443298] demodrv_read: read nbytes=4096 done at pos=0
buffer compare fail
free(): invalid pointer
Aborted
/mnt #
```

发现跑出错误了。“buffer compare fail”说明 buffer 读回来的数据和原始值不一样。这是为什么呢？难道是我们驱动代码有问题吗？

我们再运行 test_ok 测试程序，发现跑通了，log 里显示：data modify and compare successful。

7.7 实验 7：映射用户内存

```
/mnt # ./test_ok
[ 235.529717] demodrv_open: major=10, minor=58
driver max buffer size=4096
[ 235.564851] demodrv_read_write: len=4096, npage=1
[ 235.568307] pin 1 pages from user done
[ 235.568828] demodrv_read_write: write user buffer 4096 bytes done
[ 235.569241] demodrv_write: write nbytes=4096 done at pos=0
[ 235.569882] demodrv_read_write: len=4096, npage=1
[ 235.570658] pin 1 pages from user done
[ 235.571163] demodrv_read_write: read user buffer 4096 bytes done
[ 235.571543] demodrv_read: read nbytes=4096 done at pos=0
data modify and compare succussful
/mnt #
```

请读者思考，为什么跑 test_issue 程序会出现错误？

4. 实验参考代码

驱动的参考代码如下。

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/uaccess.h>
4 #include <linux/init.h>
5 #include <linux/miscdevice.h>
6 #include <linux/device.h>
7 #include <linux/slab.h>
8 #include <linux/kfifo.h>
9 #include <linux/highmem.h>
10
11 #define DEMO_NAME "my_demo_dev"
12 static struct device *mydemodrv_device;
13
14 #define MYDEMO_READ 0
15 #define MYDEMO_WRITE 1
16
17 /*virtual FIFO device's buffer*/
18 static char *device_buffer;
19 #define MAX_DEVICE_BUFFER_SIZE (1 * PAGE_SIZE)
20
21 #define MYDEV_CMD_GET_BUFSIZE 1 /* defines our IOCTL cmd */
22
23 static size_t
24 demodrv_read_write(void *buf, size_t len,
25                     int rw)
26 {
27     int ret, npages, i;
28     struct page **pages;
29     struct mm_struct *mm = current->mm;
30     char *kmap_addr, *dev_buf;
31     size_t size = 0;
32     size_t count = 0;
33
34     dev_buf = device_buffer;
35
36     /* how mange pages? */
37     npages = DIV_ROUND_UP(len, PAGE_SIZE);
38
39     printk("%s: len=%d, npage=%d\n", __func__, len, npages);
40
41     pages = kmalloc(npages * sizeof(pages), GFP_KERNEL);
42     if (!pages) {
43         printk("alloc pages fail\n");
44         return -ENOMEM;
```

奔跑吧 linux 社区出品

```

45      }
46
47      down_read(&mm->mmap_sem);
48
49      ret = get_user_pages_fast((unsigned long)buf, npages, 1, pages);
50      if (ret < npages) {
51          printk("pin page fail\n");
52          goto fail_pin_pages;
53      }
54
55      up_read(&mm->mmap_sem);
56
57      printk("pin %d pages from user done\n", npages);
58
59      for (i = 0; i < npages; i++) {
60          kmap_addr = kmap(pages[i]);
61          //print_hex_dump_bytes("kmap:", DUMP_PREFIX_OFFSET,
kmap_addr, PAGE_SIZE);
62          size = min_t(size_t, PAGE_SIZE, len);
63          switch(rw) {
64              case MYDEMO_READ:
65                  memcpy(kmap_addr, dev_buf + PAGE_SIZE * i,
66                         size);
67                  //print_hex_dump_bytes("read:", DUMP_PREFIX_OFFSET,
kmap_addr, size);
68                  break;
69              case MYDEMO_WRITE:
70                  memcpy(dev_buf + PAGE_SIZE*i, kmap_addr,
71                         size);
72                  //print_hex_dump_bytes("write:", DUMP_PREFIX_OFFSET,
dev_buf + PAGE_SIZE*i, size);
73                  break;
74              default:
75                  break;
76          }
77          put_page(pages[i]);
78          kunmap(pages[i]);
79          len -= size;
80          count += size;
81      }
82
83      kfree(pages);
84
85      printk("%s: %s user buffer %d bytes done\n", __func__, rw ?
"write":"read", count);
86
87      return count;
88
89 fail_pin_pages:
90      up_read(&mm->mmap_sem);
91      for (i = 0; i < ret; i++)
92          put_page(pages[i]);
93      kfree(pages);
94
95      return -EFAULT;
96  }
97
98 static int demodrv_open(struct inode *inode, struct file *file)
99 {
100     int major = MAJOR(inode->i_rdev);
101     int minor = MINOR(inode->i_rdev);
102
103     printk("%s: major=%d, minor=%d\n", __func__, major, minor);
104

```

7.7 实验 7：映射用户内存

```

105     return 0;
106}
107
108static int demodrv_release(struct inode *inode, struct file *file)
109{
110     return 0;
111}
112
113static ssize_t
114demodrv_read(struct file *file, char __user *buf, size_t count, loff_t
*ppos)
115{
116     size_t nbytes =
117         demodrv_read_write(buf, count, MYDEMO_READ);
118
119     printk("%s: read nbytes=%d done at pos=%d\n",
120           __func__, nbytes, (int)*ppos);
121
122     return nbytes;
123}
124
125static ssize_t
126demodrv_write(struct file *file, const char __user *buf, size_t count,
loff_t *ppos)
127{
128
129     size_t nbytes =
130         demodrv_read_write((void *)buf, count, MYDEMO_WRITE);
131
132     printk("%s: write nbytes=%d done at pos=%d\n",
133           __func__, nbytes, (int)*ppos);
134
135     return nbytes;
136}
137
138static int
139demodrv_mmap(struct file *filp, struct vm_area_struct *vma)
140{
141     unsigned long pfn;
142     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
143     unsigned long len = vma->vm_end - vma->vm_start;
144
145     if (offset >= MAX_DEVICE_BUFFER_SIZE)
146         return -EINVAL;
147     if (len > (MAX_DEVICE_BUFFER_SIZE - offset))
148         return -EINVAL;
149
150     printk("%s: mapping %ld bytes of device buffer at offset %ld\n",
151           __func__, len, offset);
152
153     /* pfn = page_to_pfn (virt_to_page (ramdisk + offset)); */
154     pfn = virt_to_phys(device_buffer + offset) >> PAGE_SHIFT;
155
156     if (remap_pfn_range(vma, vma->vm_start, pfn, len,
vma->vm_page_prot))
157         return -EAGAIN;
158
159     return 0;
160}
161
162static long
163demodrv_unlocked_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg)
164{

```

奔跑吧 linux 社区出品

```

165     unsigned long tbs = MAX_DEVICE_BUFFER_SIZE;
166     void __user *ioargp = (void __user *)arg;
167
168     switch (cmd) {
169     default:
170         return -EINVAL;
171
172     case MYDEV_CMD_GET_BUFSIZE:
173         if (copy_to_user(ioargp, &tbs, sizeof(tbs)))
174             return -EFAULT;
175         return 0;
176     }
177 }
178
179static const struct file_operations demodrv_fops = {
180     .owner = THIS_MODULE,
181     .open = demodrv_open,
182     .release = demodrv_release,
183     .read = demodrv_read,
184     .write = demodrv_write,
185     .mmap = demodrv_mmap,
186     .unlocked_ioctl = demodrv_unlocked_ioctl,
187};
188
189static struct miscdevice mydemodrv_misc_device = {
190     .minor = MISC_DYNAMIC_MINOR,
191     .name = DEMO_NAME,
192     .fops = &demodrv_fops,
193};
194
195static int __init simple_char_init(void)
196{
197     int ret;
198
199     device_buffer = kmalloc(MAX_DEVICE_BUFFER_SIZE, GFP_KERNEL);
200     if (!device_buffer)
201         return -ENOMEM;
202
203     ret = misc_register(&mydemodrv_misc_device);
204     if (ret) {
205         printk("failed register misc device\n");
206         kfree(device_buffer);
207         return ret;
208     }
209
210     mydemodrv_device = mydemodrv_misc_device.this_device;
211
212     printk("succeeded register char device: %s\n", DEMO_NAME);
213
214     return 0;
215}
216
217static void __exit simple_char_exit(void)
218{
219     printk("removing device\n");
220
221     kfree(device_buffer);
222     misc_deregister(&mydemodrv_misc_device);
223}
224
225module_init(simple_char_init);
226module_exit(simple_char_exit);
227
228MODULE_AUTHOR("Benshushu");

```

7.7 实验 7：映射用户内存

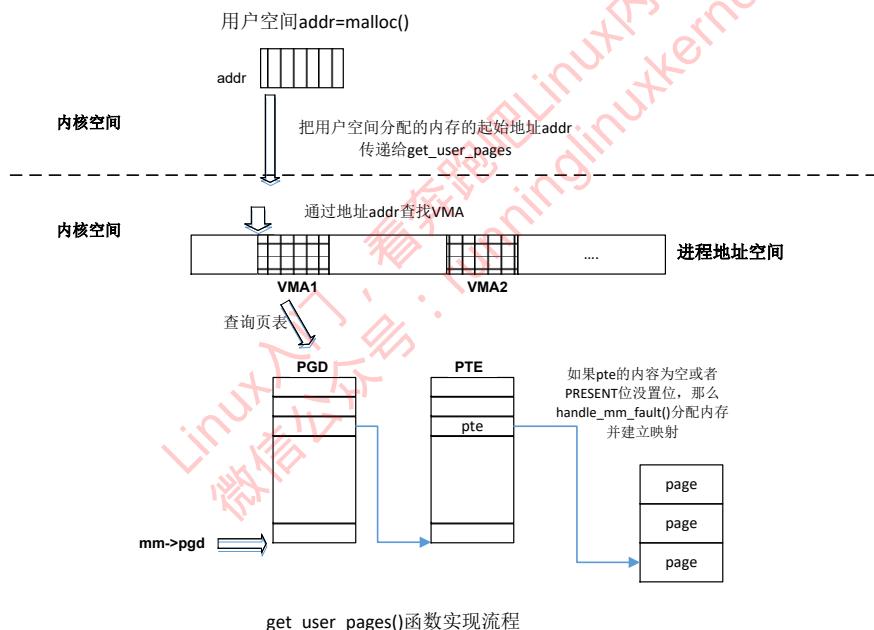
```
229MODULE_LICENSE("GPL v2");
230MODULE_DESCRIPTION("simple character device");
```

本实验是在上一个实验基础上实现锁定 (pin) 用户内存，也就是另外一个方式来映射用户内存。实现锁定用户内存的主要函数就是 `get_user_pages()` 函数。

`get_user_pages()` 函数

用于把用户空间的虚拟内存空间传到内核空间，内核空间为其分配物理内存并建立相应的映射关系，实现过程如图所示。例如，在 camera 驱动的 V4L2 核心架构中可以使用用户空间内存类型 (`V4L2_MEMORY_USERPTR`) 来分配物理内存，其驱动的实现使用的是 `get_user_pages()` 函数。

```
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
                     unsigned long start, unsigned long nr_pages, int write,
                     int force, struct page **pages, struct vm_area_struct **vmas)
```



get_user_pages() 函数实现流程

get_user_pages() 函数实现框图

在本实验中，我们把 `read` 和 `write` 方法中的用户空间的 buffer 进行锁定 (pin)，然后重新分配物理内存，并且映射到用户空间的 buffer 中，整个过程有点类似 `mmap` 方法。

我们来看本实验的 `read` 和 `write` 方法，分别调用了内部函数 `demodrv_read_write()`，该函数实现在第 24 行。

第 34 行，`dev_buf` 指向设备 buffer。

第 37 行，计算用户空间的 buffer 一共有多少页。

第 41 行，分配页数据结构 struct page，每一个页有一个页数据结构 struct page，这里 pages 可以看做是页数据结构 struct page 的数组，数组的成员就是页数据结构 struct page 指针。

第 47 行，申请一个 mm->mmap_sem 的读者类型的信号量。

第 49 行，调用 get_user_pages_fast() 函数来实现锁定任务，即 pin 住用户空间 buffer 对应的在内核态的物理内存，同时也完成新分配的物理内存和设备 buffer 之间的映射关系。

```
| int get_user_pages_fast(unsigned long start, int nr_pages, int write,
|                         struct page **pages)
```

get_user_pages_fast() 函数有 4 个参数：

- start：用户空间 buffer 的起始地址
- nr_pages：需要锁定的页面的数量
- write：这些页面是否需要可写
- pages：当锁定完成之后，pages 指向已经锁定完成的物理页面。

第 55 行，释放读者类型的信号量。

第 59 行，对已经锁定的页面进行进一步处理。

第 60 行，kmap 函数是把物理页面 pages[i] 进行一个临时的映射，从而得到一个内核态的虚拟地址 kmap_addr。

第 64 行，对于读操作，把设备 buffer 的内容拷贝到我们已经锁定好的物理页面。

第 69 行，对于写操作，把锁定好的物理页面的内容写入到设备 buffer 定义的页面。

第 77 行，这个步骤很重要，因为 get_user_pages_fast() 函数之所以锁定，是因为对物理页面进行了增加了页引用计数 (get_page())，这里要释放页索引 (put_page)。关于页引用计数，读者可以阅读蓝色《奔跑吧 Linux 内核》第 2 章相关内容。

第 78 行，取消刚才建立的临时映射。

test_issue 测试代码如下。

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <fcntl.h>
8 #include <sys/ioctl.h>
9 #include <malloc.h>
10
11#define DEMO_DEV_NAME "/dev/my_demo_dev"
```

7.7 实验 7：映射用户内存

```

12
13#define MYDEV_CMD_GET_BUFSIZE 1 /* defines our IOCTL cmd */
14
15int main()
16{
17    int fd;
18    int i;
19    size_t len;
20    char *read_buffer, *write_buffer;
21
22    fd = open(DEMO_DEV_NAME, O_RDWR);
23    if (fd < 0) {
24        printf("open device %s failed\n", DEMO_DEV_NAME);
25        return -1;
26    }
27
28    if (ioctl(fd, MYDEV_CMD_GET_BUFSIZE, &len) < 0) {
29        printf("ioctl fail\n");
30        goto open_fail;
31    }
32
33    printf("driver max buffer size=%d\n", len);
34
35    read_buffer = malloc(len);
36    if (!read_buffer)
37        goto open_fail;
38
39    write_buffer = malloc(len);
40    if (!write_buffer)
41        goto buffer_fail;
42
43    /* modify the write buffer */
44    for (i = 0; i < len; i++)
45        *(write_buffer + i) = 0x55;
46
47    if (write(fd, write_buffer, len) != len) {
48        printf("write fail\n");
49        goto rw_fail;
50    }
51
52    /* read the buffer back and compare with the mmap buffer*/
53    if (read(fd, read_buffer, len) != len) {
54        printf("read fail\n");
55        goto rw_fail;
56    }
57
58    if (memcmp(write_buffer, read_buffer, len)) {
59        printf("buffer compare fail\n");
60        goto rw_fail;
61    }
62
63    printf("data modify and compare successful\n");
64
65    free(write_buffer);
66    free(read_buffer);
67    close(fd);
68
69    return 0;
70
71rw_fail:
72    if (write_buffer)
73        free(write_buffer);
74buffer_fail:
75    if (read_buffer)

```

```

76         free(read_buffer);
77open_fail:
78     close(fd);
79     return 0;
80}

```

本实验的测试代码不复杂，就是通过 write 函数把设备 buffer 进行改写，然后通过 read 函数把设备 buffer 又读回来，看看内容是否一致。

5. 思考题提示

小明同学跑 test_issue 程序发现跑出错误了。“buffer compare fail”说明 buffer 读回来的数据和原始值不一样。而跑 test_ok 程序却是正确的。这是为什么呢？

小明同学对比了 test_issue 和 test_ok，最大的不同就是使用 malloc 来分配 user buffer，而不是通过 mmap 来分配的匿名页面，那究竟是什么原因导致的呢？

这问题其实是笨叔在实际项目开发中遇到的，因此抽象出来把它变成一个实验。这个思考题对初学者来说有不小的难度。若对该问题的来龙去脉想明白，那么对 Linux 的内存管理的理解会上一个台阶。

遇到这种问题，我们最简单最粗暴也是最有效的调试办法就是把 buffer 打印出来看看。在内核里，可以使用 print_hex_dump_bytes() 函数。比如在 demodrv_read_write() 函数的第 61,67,72 行添加打印语句。

```

59     for (i = 0; i < npages; i++) {
60         kmap_addr = kmap(pages[i]);
61         //print_hex_dump_bytes("kmap:", DUMP_PREFIX_OFFSET, kmap_addr, PAGE_SIZE);
62         size = min_t(size_t, PAGE_SIZE, len);
63         switch(rw) {
64             case MYDEMO_READ:
65                 memcpy(kmap_addr, dev_buf + PAGE_SIZE * i,
66                         size);
67                 //print_hex_dump_bytes("read:", DUMP_PREFIX_OFFSET, kmap_addr, size);
68                 break;
69             case MYDEMO_WRITE:
70                 memcpy(dev_buf + PAGE_SIZE*i, kmap_addr,
71                         size);
72                 //print_hex_dump_bytes("write:", DUMP_PREFIX_OFFSET, dev_buf + PAGE_SIZE*i, size);
73                 break;
74             default:
75                 break;
76         }

```

运行 test_issue 程序，打印结果如下：

7.7 实验 7：映射用户内存

发现 pin 的 page，开头的数据都是 0，而不是 0x55，这是为什么呢？理论上这个 page 应该全部都是 0x55 才对。

我们在运行 test_ok 程序，发现 pin 的 page 的数据，从开头到介绍都是 0x55，这就奇怪了？

我们需要进一步思考 malloc 和 mmap 来分配虚拟内存，究竟有什么不一样。

我们继续在 demodrv_read_write() 函数里添加打印，这次我们把用户空间 buffer 的起始地址打印出来。

运行 test_issue 程序，打印的用户空间 buffer 起始地址为 buf=0x0008c0d8。

```
/mnt # ./test_issue
[ 575.074668] demodrv_open: major=10, minor=58
driver max buffer size=4096
[ 575.099693] demodrv_read_write: len=4096, npage=1
[ 575.101724] demodrv read write: buf=0x0008c0d8
[ 575.103246] pin 1 pages from user done
[ 575.105738] kmap:00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.107920] kmap:00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.111371] kmap:00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.115558] kmap:00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.118507] kmap:00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.121631] kmap:00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.124182] kmap:00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.127227] kmap:00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.129757] kmap:00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.130594] kmap:00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.132381] kmap:000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.133943] kmap:000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.134659] kmap:000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.135637] kmap:000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 575.136735] kmap:000000e0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 575.137778] kmap:000000f0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 575.139063] kmap:00000100: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
```

运行 test_ok 程序，打印的用户空间 buffer 起始地址为 0xb6f99000。

```
/mnt # ./test_ok
[ 811.858460] demodrv_open: major=10, minor=58
driver max buffer size=4096
[ 811.892401] demodrv_read_write: len=4096, npage=1
[ 811.892969] demodrv read write: buf=0xb6f99000
[ 811.893653] pin 1 pages from user done
[ 811.894002] kmap:00000000: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.894606] kmap:00000010: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.895295] kmap:00000020: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.896593] kmap:00000030: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.899462] kmap:00000040: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.899961] kmap:00000050: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.901289] kmap:00000060: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.903883] kmap:00000070: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.905224] kmap:00000080: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.907513] kmap:00000090: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.909971] kmap:000000a0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
[ 811.911412] kmap:000000b0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
```

我们发现问题，使用 malloc 分配 4KB 的虚拟内存，起始地址为 0x0008c0d8，而使用 mmap 分配的 4KB 虚拟内存，起始地址是 0xb6f99000。前者的起始地址没有页面对齐（4KB），后者以 4KB 页面对齐。

这就是为什么使用 malloc 分配内存的 test_issue 程序，在内核空间中看到 pin 住的物理页面有一部分地址的内容是 0x0，而另外一部分地址的内容是 0x55。

7.8 实验 8：OOM

1. 实验目的

了解 OOM 机制实现的原理。

2. 实验要求

- 1) 编写一个简单的应用程序，这个应用程序只分配内存，不释放内存。然后不断地重复执行这个程序，直到系统的 OOM Killer 机制起作用。
- 2) 分析 OOM Killer 打印的日志信息。

7.8 实验 8 : OOM

3) 编写一个简单的内核模块，使用 alloc_pages()函数来不断分配内存直到触发系统的 OOM Killer 机制。

3. 实验步骤

本实验有两个方案，一个是使用 test 应用程序来触发 OOM Killer，另外一个是使用内核驱动的方式。

(1) test 应用程序方法

进入本实验的参考代码目录。

```
#cd  
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_7_mm/lab  
8_oom/oom_user_app
```

编译 oom 测试程序。

```
# export CC=arm-linux-gnueabi-gcc  
# make  
  
rlk@ubuntu:oom_user_app$ export CC=arm-linux-gnueabi-gcc  
rlk@ubuntu:oom_user_app$ make  
arm-linux-gnueabi-gcc -o oom oom.c -lpthread --static  
rlk@ubuntu:oom_user_app$
```

拷贝 oom 程序到 kmodules 目录。

```
#cp oom /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

启动 QEMU 虚拟机。进入 mnt 目录，运行 oom 测试程序。

```
/mnt # ./oom  
expected victim is 776.  
thread(b6fb2300), allocating 10485760 bytes.  
thread(b67b1300), allocating 10485760 bytes.  
thread(b5fb0300), allocating 10485760 bytes.  
thread(b55ff300), allocating 10485760 bytes.  
thread(b55ff300), allocating 10485760 bytes.  
thread(b67b1300), allocating 10485760 bytes.  
thread(b6fb2300), allocating 10485760 bytes.  
[ 74.174102] oom invoked oom-killer: gfp_mask=0x200da, order=0,  
oom_score_adj=0  
[ 74.178318] oom cpuset=/ mems_allowed=0  
[ 74.179912] CPU: 3 PID: 780 Comm: oom Not tainted 4.0.0+ #3  
[ 74.180434] Hardware name: ARM-Versatile Express  
[ 74.182073] [<c002489c>] (unwind_backtrace) from [<c001d76c>]  
(show_stack+0x2c/0x38)  
[ 74.182909] [<c001d76c>] (show_stack) from [<c059b264>]  
(__dump_stack+0x1c/0x24)  
[ 74.183813] [<c059b264>] (__dump_stack) from [<c059b33c>]  
(dump_stack+0xd0/0xf8)  
[ 74.186160] [<c059b33c>] (dump_stack) from [<c01a8414>]  
(dump_header+0x104/0x158)  
[ 74.187215] [<c01a8414>] (dump_header) from [<c01a8a04>]  
(oom_kill_process+0x208/0xa48)  
[ 74.188149] [<c01a8a04>] (oom_kill_process) from [<c01a9c08>]  
(__out_of_memory+0x410/0x43c)  
[ 74.189387] [<c01a9c08>] (__out_of_memory) from [<c01a9c98>]  
(out_of_memory+0x64/0x88)  
[ 74.190579] [<c01a9c98>] (out_of_memory) from [<c01b0b14>]  
(__alloc_pages_nodemask+0xed8/0x1208)
```

奔跑吧 linux 社区出品

```

[ 74.191903] [<c01b0b14>] (_alloc_pages_nodemask) from [<c0200634>]
(do_anonymous_page+0x2c8/0x6cc)
[ 74.193539] [<c0200634>] (do_anonymous_page) from [<c0202518>]
(handle_pte_fault+0xcc/0x364)
[ 74.196475] [<c0202518>] (handle_pte_fault) from [<c0202ab4>]
(__handle_mm_fault+0x304/0x314)
[ 74.197671] [<c0202ab4>] (__handle_mm_fault) from [<c0202bb4>]
(handle_mm_fault+0xf0/0x14c)
[ 74.198954] [<c0202bb4>] (handle_mm_fault) from [<c0b13984>]
(__do_page_fault+0x10c/0x168)
[ 74.200212] [<c0b13984>] (__do_page_fault) from [<c0b13bc4>]
(do_page_fault+0x1e4/0x6b0)
[ 74.201602] [<c0b13bc4>] (do_page_fault) from [<c000879c>]
(do_DataAbort+0x64/0x104)
[ 74.202913] [<c000879c>] (do_DataAbort) from [<c0b1311c>]
(__dabt_usr+0x3c/0x40)
[ 74.204091] Exception stack(0xc478bfb0 to 0xc478bff8)
[ 74.206055] bfa0: b1bff000 00a00000 00000007
b2127000
[ 74.206352] bfc0: b55ff300 ffffffff b55ff518 00000152 be930cd2 0009a4c0
be930cd4 b55fec0c
[ 74.206713] bfe0: 00000000 b55fecc8 0001035c 00010398 80000010 ffffffff
[ 74.207228] Mem-info:
[ 74.207631] Normal per-cpu:
[ 74.207900] CPU 0: hi: 18, btch: 3 usd: 0
[ 74.208143] CPU 1: hi: 18, btch: 3 usd: 0
[ 74.208445] CPU 2: hi: 18, btch: 3 usd: 0
[ 74.208745] CPU 3: hi: 18, btch: 3 usd: 0
[ 74.209637] active_anon:13491 inactive_anon:1067 isolated_anon:0
[ 74.209637] active_file:0 inactive_file:6 isolated_file:0
[ 74.209637] unevictable:0 dirty:0 writeback:0 unstable:0
[ 74.209637] free:288 slab_reclaimable:990 slab_unreclaimable:4853
[ 74.209637] mapped:295 shmem:1596 pagetables:37 bounce:0
[ 74.209637] free_cma:0
[ 74.212406] Normal free:1152kB min:1156kB low:1444kB high:1732kB
active_anon:53964kB inactive_anon:4268kB active_file:0kB inactive_file:24kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:102400kB
managed:87364kB mlocked:0kB dirty:0kB writeback:0kB mapped:1180kB
shmem:6384kB slab_reclaimable:3960kB slab_unreclaimable:19412kB
kernel_stack:752kB pagetables:148kB unstable:0kB bounce:0kB free_cma:0kB
writeback_tmp:0kB pages_scanned:12 all_unreclaimable? no
[ 74.219494] lowmem_reserve[]: 0 0
[ 74.220309] Normal: 3*4kB (MR) 6*8kB (EMR) 0*16kB 1*32kB (R) 1*64kB (R)
0*128kB 0*256kB 0*512kB 1*1024kB (R) 0*2048kB 0*4096kB = 1180kB
[ 74.222244] 1606 total pagecache pages
[ 74.223091] 0 pages in swap cache
[ 74.223688] Swap cache stats: add 0, delete 0, find 0/0
[ 74.225719] Free swap = 0kB
[ 74.226120] Total swap = 0kB
[ 74.231938] 25600 pages of RAM
[ 74.232225] 469 free pages
[ 74.232545] 3759 reserved pages
[ 74.232964] 3379 slab pages
[ 74.233228] 563 pages shared
[ 74.233597] 0 pages swap cached
[ 74.233927] [ pid ] uid tgid total_vm rss nr_ptes nr_pmds swapents
oom_score_adj name
[ 74.236576] [ 772] 0 772 577 1 3 0 0
0 sh
[ 74.236927] [ 775] 0 775 191 1 2 0 0
0 oom
[ 74.237151] [ 776] 0 776 24003 12800 36 0 0
0 oom
[ 74.237395] Out of memory: Kill process 776 (oom) score 570 or sacrifice

```

7.8 实验 8 : OOM

```
child
[ 74.238017] Killed process 776 (oom) total-vm:96012kB, anon-rss:51200kB,
file-rss:0kB
victim signalled: 9
/mnt #
```

(2) 内核驱动方式

进入本实验参考代码。

```
#cd
/home/rbk/rbk_basic/runninglinuxkernel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab
8_oom/oom_kernel_driver

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rbk@ubuntu:oom_kernel_driver$ make
make -C /home/rbk/rbk_basic/runninglinuxkernel_4.0 M=/home/rbk/rbk_basic/runninglinuxker
nel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab8_oom/oom_kernel_driver modules;
make[1]: Entering directory '/home/rbk/rbk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rbk/rbk_basic/runninglinuxkernel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab8_
_oom/oom_kernel_driver/alloc_oom.o
  LD [M]  /home/rbk/rbk_basic/runninglinuxkernel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab8_
_oom/oom_kernel_driver/alloc_oom.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rbk/rbk_basic/runninglinuxkernel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab8_
_oom/oom_kernel_driver/alloc_oom.mod.o
  LD [M]  /home/rbk/rbk_basic/runninglinuxkernel_4.0/rbk_lab/rbk_basic/chapter_7_mm/lab8_
_oom/oom_kernel_driver/alloc_oom.ko
make[1]: Leaving directory '/home/rbk/rbk_basic/runninglinuxkernel_4.0'
rbk@ubuntu:oom_kernel_driver$
```

拷贝内核模块到 kmodues 目录。

```
#cp alloc-oom.ko /home/rbk/rbk_basic/runninglinuxkernel_4.0/kmodues/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rbk/rbk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
/mnt # insmod alloc-oom.ko
[ 367.301063] insmod invoked oom-killer: gfp_mask=0xd0, order=0,
oom_score adj=0
[ 367.304444] insmod cpuset=/ mems_allowed=0
[ 367.306132] CPU: 0 PID: 782 Comm: insmod Tainted: G          O  4.0.0+ #3
[ 367.310378] Hardware name: ARM-Versatile Express
[ 367.311724] [<c002489c>] ( unwind_backtrace ) from [<c001d76c>]
(show_stack+0x2c/0x38)
[ 367.312932] [<c001d76c>] ( show_stack ) from [<c059b264>]
(_dump_stack+0x1c/0x24)
[ 367.313489] [<c059b264>] ( _dump_stack ) from [<c059b33c>]
(dump_stack+0xd0/0xf8)
[ 367.314565] [<c059b33c>] ( dump_stack ) from [<c01a8414>]
(dump_header+0x104/0x158)
[ 367.315717] [<c01a8414>] ( dump_header ) from [<c01a8a04>]
(oom_kill_process+0x208/0xa48)
```

奔跑吧 linux 社区出品

```

[ 367.317124] [<c01a8a04>] (oom_kill_process) from [<c01a9c08>]
(__out_of_memory+0x410/0x43c)
[ 367.319116] [<c01a9c08>] (__out_of_memory) from [<c01a9c98>]
(__out_of_memory+0x64/0x88)
[ 367.320226] [<c01a9c98>] (out_of_memory) from [<c01b0b14>]
(__alloc_pages_nodemask+0xed8/0x1208)
[ 367.321251] [<c01b0b14>] (__alloc_pages_nodemask) from [<bf00209c>]
(my_init+0x9c/0xf0 [alloc_oom])
[ 367.322914] [<bf00209c>] (my_init [alloc_oom]) from [<c0008dc8>]
(do_one_initcall+0x68/0x190)
[ 367.323941] [<c0008dc8>] (do_one_initcall) from [<c0119d5c>]
(do_init_module+0xb4/0x278)
[ 367.325126] [<c0119d5c>] (do_init_module) from [<c011a710>]
(load_module+0x3ec/0x570)
[ 367.326573] [<c011a710>] (load_module) from [<c011a93c>]
(SyS_init_module+0xa8/0xc0)
[ 367.330065] [<c011a93c>] (SyS_init_module) from [<c0014d40>]
(ret_fast_syscall+0x0/0x34)
[ 367.332531] Mem-info:
[ 367.333220] Normal per-cpu:
[ 367.333735] CPU 0: hi: 18, btch: 3 usd: 3
[ 367.334818] CPU 1: hi: 18, btch: 3 usd: 8
[ 367.335758] CPU 2: hi: 18, btch: 3 usd: 15
[ 367.336473] CPU 3: hi: 18, btch: 3 usd: 0
[ 367.337787] active_anon:563 inactive_anon:1066 isolated_anon:0
[ 367.337787] active_file:0 inactive_file:0 isolated_file:0
[ 367.337787] unevictable:0 dirty:0 writeback:0 unstable:0
[ 367.337787] free:284 slab_reclaimable:884 slab_unreclaimable:4857
[ 367.337787] mapped:345 shmem:1596 pagetables:8 bounce:0
[ 367.337787] free_cma:0
[ 367.347201] Normal free:1136kB min:1156kB low:1444kB high:1732kB
active_anon:2252kB inactive_anon:4264kB active_file:0kB inactive_file:0kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:102400kB
managed:87364kB mlocked:0kB dirty:0kB writeback:0kB mapped:1380kB
shmem:6384kB slab_reclaimable:3536kB slab_unreclaimable:19428kB
kernel_stack:736kB pagetables:32kB unstable:0kB bounce:0kB free_cma:0kB
writeback tmp:0kB pages_scanned:0 all_unreclaimable? yes
[ 367.353862] lowmem_reserve(): 0 0
[ 367.354861] Normal: 3*4kB (EMR) 1*8kB (M) 0*16kB 0*32kB 1*64kB (R) 0*128kB
0*256kB 0*512kB 1*1024kB (R) 0*2048kB 0*4096kB = 1108kB
[ 367.361644] 1596 total pagecache pages
[ 367.362379] 0 pages in swap cache
[ 367.362957] Swap cache stats: add 0, delete 0, find 0/0
[ 367.363517] Free swap = 0kB
[ 367.363887] Total swap = 0kB
[ 367.369119] 25600 pages of RAM
[ 367.369791] 468 free pages
[ 367.370576] 3759 reserved pages
[ 367.370883] 3328 slab pages
[ 367.371425] 855 pages shared
[ 367.371628] 0 pages swap cached
[ 367.372187] [ pid ] uid tgid total_vm rss nr_ptes nr_pmds swapents
oom_score_adj name
[ 367.373716] [ 772] 0 772 577 310 3 0 0
0 sh
[ 367.374903] [ 782] 0 782 577 1 3 0 0
0 insmod
[ 367.376131] Out of memory: Kill process 772 (sh) score 13 or sacrifice
child
[ 367.378175] Killed process 782 (insmod) total-vm:2308kB, anon-rss:4kB,
file-rss:0kB
[ 367.386959] insmod: page allocation failure: order:0, mode:0xd0
[ 367.389259] CPU: 1 PID: 782 Comm: insmod Tainted: G 0 4.0.0+ #3
[ 367.390786] Hardware name: ARM-Versatile Express

```

7.8 实验 8 : OOM

```

[ 367.391706] [<c002489c>] (unwind_backtrace) from [<c001d76c>]
(show_stack+0x2c/0x38)
[ 367.392610] [<c001d76c>] (show_stack) from [<c059b264>]
(__dump_stack+0x1c/0x24)
[ 367.394110] [<c059b264>] (__dump_stack) from [<c059b33c>]
(dump_stack+0xd0/0xf8)
[ 367.395435] [<c059b33c>] (dump_stack) from [<c01af350>]
(warn_alloc_failed+0x1a8/0x1e4)
[ 367.398584] [<c01af350>] (warn_alloc_failed) from [<c01b0c94>]
(__alloc_pages_nodemask+0x1058/0x1208)
[ 367.399569] [<c01b0c94>] (__alloc_pages_nodemask) from [<bf00209c>]
(my_init+0x9c/0xf0 [alloc_oom])
[ 367.401071] [<bf00209c>] (my_init [alloc_oom]) from [<c0008dc8>]
(do_one_initcall+0x68/0x190)
[ 367.402263] [<c0008dc8>] (do_one_initcall) from [<c0119d5c>]
(do_init_module+0xb4/0x278)
[ 367.403170] [<c0119d5c>] (do_init_module) from [<c011a710>]
(load_module+0x3ec/0x570)
[ 367.404017] [<c011a710>] (load_module) from [<c011a93c>]
(SyS_init_module+0xa8/0xc0)
[ 367.405031] [<c011a93c>] (SyS_init_module) from [<c0014d40>]
(ret_fast_syscall+0x0/0x34)
[ 367.408797] Mem-info:
[ 367.409221] Normal per-cpu:
[ 367.409650] CPU 0: hi: 18, btch: 3 usd: 3
[ 367.410562] CPU 1: hi: 18, btch: 3 usd: 8
[ 367.411906] CPU 2: hi: 18, btch: 3 usd: 15
[ 367.412997] CPU 3: hi: 18, btch: 3 usd: 0
[ 367.414351] active_anon:563 inactive_anon:1066 isolated_anon:0
[ 367.414351] active_file:0 inactive_file:0 isolated_file:0
[ 367.414351] unevictable:0 dirty:0 writeback:0 unstable:0
[ 367.414351] free:5 slab_reclaimable:884 slab_unreclaimable:4857
[ 367.414351] mapped:345 shmem:1596 pagetables:8 bounce:0
[ 367.414351] free_cma:0
[ 367.425642] Normal free:20kB min:1156kB low:1444kB high:1732kB
active_anon:2252kB inactive_anon:4264kB active_file:0kB inactive_file:0kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB present:1024000kB
managed:87364kB mlocked:0kB dirty:0kB writeback:0kB mapped:1380kB
shmem:6384kB slab_reclaimable:3536kB slab_unreclaimable:19428kB
kernel_stack:736kB pagetables:32kB unstable:0kB bounce:0kB free_cma:0kB
writeback_tmp:0kB pages_scanned:0 all_unreclaimable? yes
[ 367.443607] lowmem_reserve[]: 0 0 0
[ 367.444359] Normal: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB
0*512kB 0*1024kB 0*2048kB 0*4096kB = 0kB
[ 367.447602] 1596 total pagecache pages
[ 367.449416] 0 pages in swap cache
[ 367.450098] Swap cache stats: add 0, delete 0, find 0/0
[ 367.451051] Free swap = 0kB
[ 367.451685] Total swap = 0kB
[ 367.459276] 25600 pages of RAM
[ 367.459788] 191 free pages
[ 367.460168] 3759 reserved pages
[ 367.460816] 3328 slab pages
[ 367.461238] 855 pages shared
[ 367.461730] 0 pages swap cached
[ 367.462397] have alloc 0 pages, but continue alloc failed
Killed
/mnt #

```

4. 参考代码

(1) oom 测试程序代码

本程序创建了一个子进程，在子进程中根据 cpu 个数，创建了相应个数的线程，在每个线程中不停的申请内存，直到出错为止。父进程等待子进程的结束。最后会触发内核的 oom 机制，通过 dmesg 命令可以看到 oom 机制打印的信息。

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5 #include <errno.h>
6 #include <unistd.h>
7 #include <sys/mman.h>
8 #include <sys/wait.h>
9
10#define SIZE    (10*1024*1024)
11
12static int alloc_mem(long int size)
13{
14    char *s;
15    long i, pagesz = getpagesize();
16
17    printf("thread(%lx), allocating %ld bytes.\n", (unsigned
long)pthread_self(), size);
18
19    s = mmap(NULL, size, PROT_READ|PROT_WRITE,
MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
20    if (s == MAP_FAILED)
21        return errno;
22
23    /* touch the memory */
24    for (i = 0; i < size; i+= pagesz)
25        s[i] = '\a';
26
27    return 0;
28}
29
30static void *child_alloc_thread(void *args)
31{
32    int ret = 0;
33
34    /* keep allocating until there is an error */
35    while (!ret)
36        alloc_mem(SIZE);
37
38    exit(ret);
39}
40
41static void child_alloc(int threads)
42{
43    int i;
44    pthread_t *th;
45    int ret;
46
47    th = malloc(sizeof(pthread_t) * threads);
48    if (!th) {
49        printf("malloc failed\n");
50        goto out;
51    }
52
53    /* create threads */

```

7.8 实验 8 : OOM

```

54     for (i = 0; i < threads; i++) {
55         ret = pthread_create(&th[i], NULL, child_alloc_thread,
56         NULL);
56         if (ret) {
57             printf("pthread_create error: %s\n", strerror(errno));
58             /* keep going if thread other than first failed to
59             * spawn
60             */
61             if (i == 0 || ret != EAGAIN)
62                 goto out;
63         }
64     }
65
66     /* wait for one of threads to exit whole process */
67     while(1)
68         sleep(1);
69 out:
70     exit(1);
71 }
72
73 int main(int argc, char **argv)
74 {
75     int ncpus = -1;
76     pid_t pid;
77     int status, threads;
78     int retcode;
79
80     ncpus = sysconf(_SC_NPROCESSORS_ONLN);
81
82     switch(pid = fork()) {
83     case 0:
84         threads = ncpus > 1 ? ncpus : 1;
85         child_alloc(threads);
86     default:
87         break;
88     }
89
90     printf("expected victim is %d.\n", pid);
91     waitpid(-1, &status, 0);
92
93     if (WIFSIGNALED(status)) {
94         printf("victim signalled: %d\n", WTERMSIG(status));
95     } else if (WIFEXITED(status)) {
96         retcode = WEXITSTATUS(status);
97         printf("victim retcode: (%d) %s\n", retcode,
98         strerror(retcode));
99     } else{
100         printf("victim unexpected ended\n");
101     }
102
103     return 0;
104 }
```

(2) oom 测试驱动代码

```

1 #include <linux/module.h>
2 #include <linux/slab.h>
3 #include <linux/init.h>
4 #include <linux/vmalloc.h>
5
6 static int __init my_init(void)
7 {
8     char *kbuf;
```

```

9      unsigned long order = 0;
10     size_t count = 0;
11
12     /* try __get_free_pages__ */
13     for (;;) {
14         kbuf = (char *)alloc_pages(GFP_KERNEL, order);
15         if (!kbuf) {
16             pr_err("have alloc %d pages, but continue alloc
failed\n", count);
17             break;
18         }
19         count += (order << 1);
20     }
21
22     return 0;
23}
24
25static void __exit my_exit(void)
26{
27     pr_info("Module exit\n");
28}
29
30module_init(my_init);
31module_exit(my_exit);
32
33MODULE_AUTHOR("Ben ShuShu");
34MODULE_LICENSE("GPL v2");

```

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）

1. 实验目的

- 通过本实验可以深刻理解操作系统中的页表是怎么构建的。
- 如何去遍历页表。
- 如何修改页表。
- 深刻理解各级页表中页表项的各个字段是什么含义。
- 系统调用在内核是如何被调用的。
- 当编写的驱动发生 crash 和 panic 时，如何去 debug？

注意：本实验有相当难度，学有余力的同学可以尝试做本实验。

2. 实验要求

假设你是一名安全特工，正在执行一项秘密任务，这项秘密任务就是要深入到敌军的作战指挥中心的计算机里安装一个窃听的程序，简单来说就是把计算机的系统调用动态替换掉。假设你的同事已经帮你把敌军计算机的 root 密码给破解了，接下来就看你如何动态修改系统调用了。**注意：编写的内核模块不能让敌军的计算机重启、crash/panic，否则就暴露行踪，秘密行动失败。**

- 1) 编写一个内核模块。

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）

实验环境：ARM64 体系结构，Linux 5.0 内核^①。

要求替换系统调用表（sys_call_table）中某一项系统调用，替换成自己编写的系统调用处理函数（例如：my_new_syscall()），在新的系统调用函数中打印一句“hello, I have hacked this syscall”，然后再调用回原来的系统调用处理函数。

比如以 ioctl 系统调用为例，它在系统调用表中的编号就是__NR_ioctl。那么需要修改系统调用表 sys_call_table[__NR_ioctl]的指向，让其指向 my_new_syscall()函数，然后在 my_new_syscall()函数中打印一句话，调用原来的 sys_call_table[__NR_ioctl]指向的处理函数。

2) 卸载模块时候把系统系统表恢复原样。

3) 用 clone 系统调用来验证你的驱动，clone 系统调用号是__NR_clone。

3. 实验详解

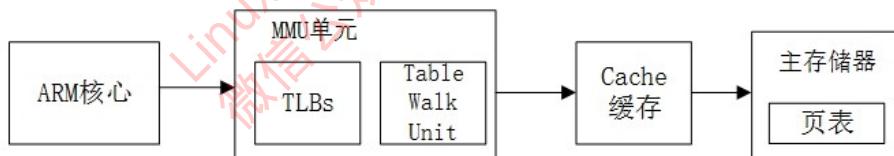
本实验源自一个实际项目中的需求。下面给出一些背景知识介绍。

根据实验要求，我们可以分解这个实验：

1. 如何修改只读属性的数据。
2. 如何替换系统调用。

（1）如何修改只读属性的数据

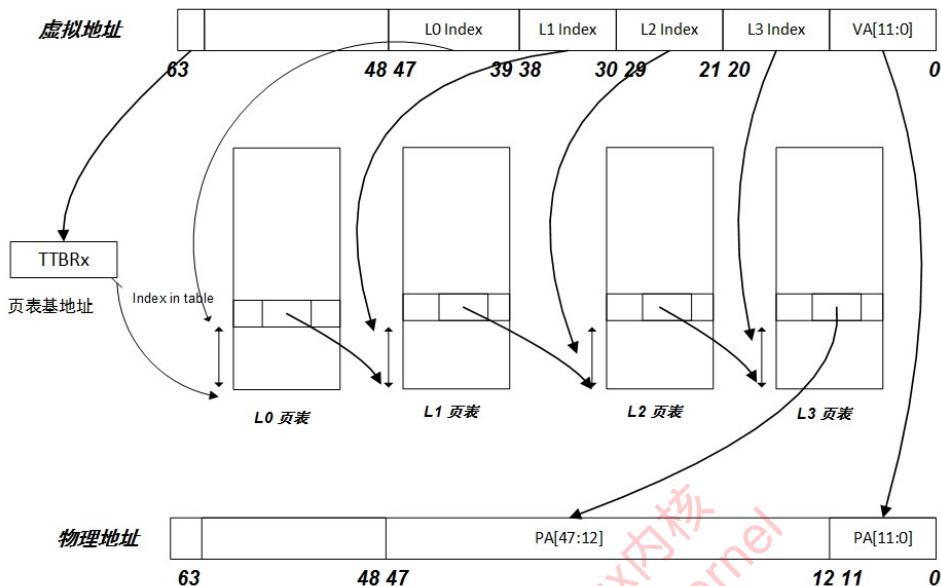
对于第一个任务，这个需要对内存管理有非常熟悉的了解。我们先回顾一下 ARM 处理器是如何访问内存的。



ARM 处理器的内存管理单元（Memory Management Unit, MMU）包括 TLB 和 Table Walk Unit 两个部件。TLB 是一块高速缓存，用于缓存页表转换的结果，从而减少内存访问的时间。一个完整的页表翻译和查找的过程叫作页表查询（Translation table walk），页表查询的过程由硬件自动完成，但是页表的维护需要软件来完成。页表查询是一个相对耗时的过程，理想的状态下是 TLB 里存有页表相关信息。当 TLB Miss 时，才会去查询页表，并且开始读入页表的内容。

所以，我们软件需要去查询页表，walk through 页表。我们来看一下页表长啥样。

^① <https://github.com/figozhang/linux-5.0-kdump.git>, 如何使用，请看项目 README。



ARM64 最多可以支持 4 级页表。

当 TLB 未命中时，处理器查询页表的过程如下。

- 处理器根据页表基地址控制寄存器 TTBRx 和虚拟地址来判断使用哪个页表基地址寄存器，是 TTBR0 还是 TTBR1。当虚拟地址第 63 比特位（简称 VA[63]）为 1 时选择 TTBR1 页表基地址寄存器；当 VA[63]为 0 时选择 TTBR0。页表基地址寄存器中存放着一级页表的基地址。
- 处理器根据虚拟地址的 VA[39:47]作为索引值 L0 Index，在一级页表（L0 页表）中找到页表项，一级页表一共有 512 个页表项。
- 第一级页表的表项中存放有二级页表（L1 页表）的物理基地址。处理器根据虚拟地址的 VA[38:30]作为索引值 L1 Index，在二级页表中找到相应的页表项，二级页表有 512 个页表项。
- 第二级页表的表项中存放有三级页表（L2 页表）的物理基地址。处理器根据虚拟地址的 VA[29:21]作为索引值 L2 Index，在三级页表（L2 页表）中找到相应的页表项，三级页表有 512 个页表项。
- 第三级页表的表项中存放有四级页表（L3 页表）的物理基地址。处理器根据虚拟地址的 VA[20:13]作为索引值 L3 Index，在四级页表（L3 页表）中找到相应的页表项，四级页表有 512 个页表项。
- 四级页表的页表项里存放有 4KB 页的物理基地址，然后加上虚拟地址的 VA[11:0]就构成了新的物理地址，因此处理器就完成了页表的查询和翻译工作。

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）

在 ARM64 版本的 Linux 内核中采用 4 级分页模型。

- 页全局目录 (Page Global Directory, 简称 PGD), 对应 L0 页表
- 页上级目录 (Page Upper Directory, 简称 PUD), 对应 L1 页表
- 页中间目录 (Page Middle Directory, 简称 PMD), 对应 L2 页表
- 页表 (Page Table, 简称 PT), 对应 L3 页表

上述 4 级页表分页模型分别对应 ARMv8 架构页表的 L0~L3 页表。关于 L0~L3 页表的页表项的格式，可以参考 ARMv8 芯片手册相关介绍。如下图是 L0~L3 页表项的描述。Bit 0 表示是一个有效的表项，Bit 1 为 0 表示是 block 类型的描述符，Bit 1 为 1 表示是 Table 类型的描述符。所谓的 table 的描述符，就是它页表中的中间表项，它的页表项内容包含了指向下一级页表的基地址。Block 或者 page 描述符。这个是页表的最后一级了，页表项中包含了最终物理地址 (output address) 的高位部分。

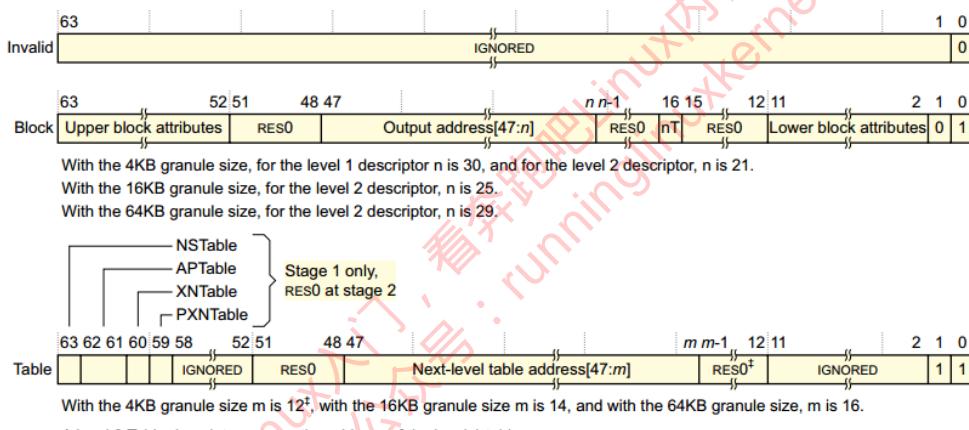


Figure D5-15 VMSAv8-64 level 0, level 1 and level 2 descriptor formats with 48-bit OAs

L3 的页表项格式如下：

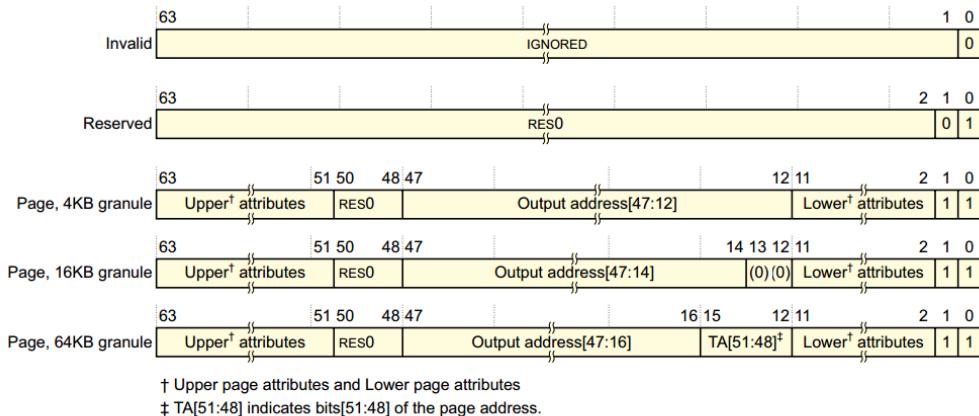


Figure D5-17 VMSAv8-64 level 3 descriptor format

我们在来看一下 ARM64 的内存布局。

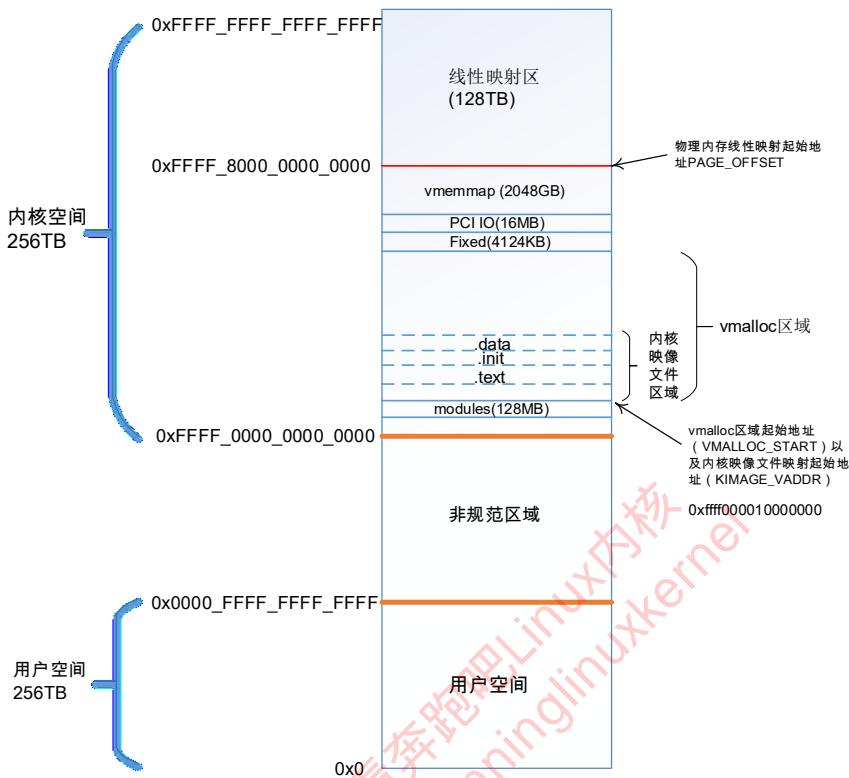
ARM64 架构处理器采用 48 位物理寻址机制，最大可以寻找 256TB 的物理地址空间。对于目前的应用来说已经足够了，不需要扩展到 64 位的物理寻址。虚拟地址也同样最大支持 48 位寻址，所以在处理器架构设计上，把虚拟地址空间划分为两个空间，每个空间最大支持 256TB。Linux 内核在大多数体系结构上都把两个地址空间划分为用户空间和内核空间。

- 用户空间: 0x0000_0000_0000_0000 到 0x0000_ffff_ffff_ffff。
- 内核空间: 0xffff_0000_0000_0000 到 0xffff_ffff_ffff_ffff。

在 QEMU 实验平台中，ARM64 架构的 Linux 5.0 内核的内存分布图，这个打印，在 Linux 4.16 内核里已经删除了，笨叔是在 5.0 内核里重新加上的，这样可以看到比较多有用的信息。

```
Virtual kernel memory layout:
modules : 0xffffffff8000000 - 0xffffffff00010000000 ( 128 MB)
vmalloc : 0xffffffff0001000000 - 0xffffffff7dfbf000000 (129022 GB)
.text : 0xffffffff00010080000 - 0xffffffff00011730000 ( 23232 KB)
.init : 0xffffffff00011a60000 - 0xffffffff00011ee0000 ( 4608 KB)
.rodata : 0xffffffff00011730000 - 0xffffffff00011a53000 ( 3212 KB)
.data : 0xffffffff00011ee0000 - 0xffffffff00011ff8a00 ( 1123 KB)
.bss : 0xffffffff00011ff8a00 - 0xffffffff00012076970 ( 504 KB)
fixed : 0xffffffff7dffe7f9000 - 0xffffffff7dffec00000 ( 4124 KB)
PCI I/O : 0xffffffff7dffffe00000 - 0xffffffff7dfffffe00000 ( 16 MB)
vmemmap : 0xffffffff7e0000000000 - 0xffffffff800000000000 ( 2048 GB maximum)
          0xffffffff7e0000000000 - 0xffffffff7e0001000000 ( 16 MB actual)
memory : 0xffffffff800000000000 - 0xffffffff800040000000 ( 1024 MB)
PAGE_OFFSET : 0xffffffff800000000000
kimage_voffset : 0xffffefffd0000000
PHYS_OFFSET : 0x40000000
start memory : 0x40000000
```

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）



我们先举一个 Linux 内核现成的例子来，告诉大家我们这个驱动代码应该如何去遍历页表。ARM64 内核里有一个 dump 页表的功能，你们把这个 CONFIG_ARM64_PTDUMP_CORE 和 CONFIG_ARM64_PTDUMP_DEBUGFS 这两个宏打开。然后再 cat /sys/kernel/debug/kernel_page_tables 这个节点，就可以看到系统中页表映射的情况。

其中 XN 表示的 PNX 字段，在特性模式下不能执行。

- RW: 表示这个页面是可读可写。
- SHD: 表示内存 cache 共享属性。
- AF: 访问比特位
- UXN: 用户模块不能执行。
- MEM/NORMAL: 表示内存属性。

这个 dump 功能的实现的代码是在 arch/arm64/mm/dump.c 文件里。我们先看 ptdump_debugfs.c 文件。这个文件很简单，注册一个 debugfs_create_file 的一个节点，然后在 show 动作里，直接调用 ptdump_walk_pgd()。

(2) 如何替换系统调用

在现代操作系统中，根据处理器的运行模式通常分成两个空间，一个是内核空间，

另外一个用户空间。大部分的应用程序是运行在用户空间的，而内核和设备驱动运行在内核空间。那应用程序需要访问硬件资源或者需要内核提供服务，怎么办呢？

现代操作系统架构中，在内核空间和用户空间之间增加了一个中间层，这个就是系统调用层（System Call），如图所示。系统调用层主要有如下作用：

Linux 系统为每一个系统调用赋予一个系统调用号。当应用程序执行一个系统调用时候，应用程序就可以知道执行和调用到哪个系统调用了，从而不会造成混乱。系统调用号，一旦分配之后就不会有任何变更，否则已经编译好的应用程序就不能运行了。

对于 ARM64 来说，它的系统调用表实现在：include/uapi/asm-generic/unistd.h 文件中。里面定义了每个系统调用的编号。系统还有一个系统调用表，sys_call_table，它是一个函数指针的数组，它的成员是 syscall_fn_t。这是一个函数指针。

```
| typedef long (*syscall_fn_t)(struct pt_regs *regs);
```

我们以 brk 系统调用为例。

brk 系统调用主要实现在 mm/mmap.c 函数中。

```
<mm/mmap.c>
SYSCALL_DEFINE1(brk, unsigned long, brk)
```

我们第一次看到系统调用的定义，它实现是通过 SYSCALL_DEFINE1 宏来实现的，这个宏实现在 include/linux/syscalls.h 头文件中。

```
<include/linux/syscalls.h>
#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, __##name,
__VA_ARGS__)
#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, __##name,
__VA_ARGS__)
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, __##name,
__VA_ARGS__)
#define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, __##name,
__VA_ARGS__)
#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, __##name,
__VA_ARGS__)
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, __##name,
__VA_ARGS__)

#define SYSCALL_DEFINE_MAXARGS 6
```

其中 SYSCALL_DEFINE1 表示有 1 个参数，SYSCALL_DEFINE2 表示有 2 个参数，以此类推。SYSCALL_DEFINEx 宏的定义如下。

```
#define SYSCALL_DEFINEx(x, sname, ...)
SYSCALL_METADATA(sname, x, __VA_ARGS__)
__SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

其中 SYSCALL_METADATA() 用于 ftrace 调试系统调用，__SYSCALL_DEFINEx()

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）

宏的定义和体系结构相关。对于 ARM64 来说，该宏定义在 arch/arm64/include/asm/syscall_wrapper.h 头文件中。

```
<arch/arm64/include/asm/syscall_wrapper.h>

#define __SYSCALL_DEFINEx(x, name, ...) \
    asmlinkage long __arm64_sys##name(const struct pt_regs *regs); \
    ALLOW_ERROR_INJECTION(__arm64_sys##name, ERRNO); \
    static long __se_sys##name(__MAP(x, __SC_LONG, __VA_ARGS__)); \
    static inline long __do_sys##name(__MAP(x, __SC_DECL, __VA_ARGS__)); \
    asmlinkage long __arm64_sys##name(const struct pt_regs *regs) \
    { \
        return __se_sys##name(SC_ARM64_REGS_TO_ARGS(x, __VA_ARGS__)); \
    } \
    static long __se_sys##name(__MAP(x, __SC_LONG, __VA_ARGS__)) \
    { \
        long ret = __do_sys##name(__MAP(x, __SC_CAST, __VA_ARGS__)); \
        __MAP(x, __SC_TEST, __VA_ARGS__); \
        __PROTECT(x, ret, __MAP(x, __SC_ARGS, __VA_ARGS__)); \
        return ret; \
    } \
    static inline long __do_sys##name(__MAP(x, __SC_DECL, __VA_ARGS__))
```

以本章的 brk 系统调用为例，`__SYSCALL_DEFINEx` 宏展开之后变成：

```
asmlinkage long __arm64_sys_brk(const struct pt_regs *regs); \
static long __se_sys_brk(unsigned long brk); \
static inline long __do_sys_brk(unsigned long brk); \
 \
asmlinkage long __arm64_sys_brk(const struct pt_regs *regs) \
{ \
    return __se_sys_brk(brk); \
} \
 \
static long __se_sys_brk(unsigned long brk) \
{ \
    long ret = __do_sys_brk(brk); \
    return ret; \
} \
 \
static inline long __do_sys_brk(unsigned long brk)
```

因此 `SYSCALL_DEFINE1(brk, unsigned long, brk)` 语句展开后会多出 2 个函数，分别是 `__arm64_sys_brk()` 以及 `__se_sys_brk()` 函数。其中 `__arm64_sys_brk()` 函数的地址会存放到系统调用表 `sys_call_table` 中。最后这个函数变成了 `__do_sys_brk` 函数。

在对比，arch/arm64/kernel/sys.c 文件中，关于 `__SYSCALL` 这个宏，

```
#define __SYSCALL(nr, sym) [nr] = (syscall_fn_t) __arm64_##sym,
```

我们就知道，每次初始化的是，都把 `__arm64_sys_xx()` 函数添加到这个 `sys_call_table` 数组里。因此 `sys_call_table` 的函数定义原型是：

```
typedef long (*syscall_fn_t)(struct pt_regs *regs);
```

参数是 `struct pt-reg *regs`。

4 动手做实验

下面代码是小明同学写的内核模块。

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/syscalls.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include <linux/version.h>
#include <linux/kallsyms.h>

MODULE_LICENSE("GPL");

void **syscall_table;

asmlinkage long (*orig_sys_ioctl)(unsigned int fd, unsigned int cmd,
                                  unsigned long arg);

struct mm_struct *init_mm_p;

static int syscall_nr = __NR_clone;
asmlinkage long my_new_ioctl(unsigned int fd, unsigned int cmd,
                            unsigned long arg)
{
    printk(KERN_INFO "hello, i have hacked this sysall\n");
    return orig_sys_ioctl(fd, cmd, arg);
}

pte_t *walk_pagetable(unsigned long address)
{
    pgd_t *pgd = NULL;
    pte_t *pte;
    pmd_t *pmd;

    /* pgd */
    pgd = pgd_offset(init_mm_p, address);
    if (pgd == NULL || pgd_none(*pgd))
        return NULL;

    //printk("ben: pgd=0x%llx\n", *pgd);

    pmd = pmd_offset(pud_offset(pgd, address), address);
    if (pmd == NULL || pmd_none(*pmd) || !pmd_present(*pmd))
        return NULL;
    //printk("ben: pmd=0x%llx\n", *pmd);
    if ((pmd_val(*pmd) & (PMD_TYPE_SECT | PMD_SECT_USER)) ==
        (PMD_TYPE_SECT | PMD_SECT_USER)) ||
        !pmd_present(*pmd)) {
        return NULL;
    } else if (pmd_val(*pmd) & PMD_TYPE_SECT) {
        return (pte_t *)pmd;
    }

    pte = pte_offset_kernel(pmd, address);
    if ((pte == NULL) || pte_none(*pte))
        return NULL;

    //printk("ben: pte=0x%llx\n", *pte);

    return pte;
}
```

7.9 实验 9：特工队：动态修改计算机的系统调用（新增）

```

}

static int __init hack_syscall_init(void)
{
    pte_t *p_pte;
    pte_t pte;           /* old pte */
    pte_t pte_new;

    syscall_table = (void **) kallsyms_lookup_name("sys_call_table");
    if (!syscall_table) {
        printk(KERN_ERR "ERROR: Cannot find the system call table
address.\n");
        return -1;
    }

    printk(KERN_INFO "Found the sys_call_table at %16lx.\n", (unsigned
long) syscall_table);

    init_mm_p = (struct mm_struct *)kallsyms_lookup_name("init_mm");
    if (!init_mm_p) {
        printk(KERN_ERR "ERROR: Cannot find init_mm\n");
        return -1;
    }

    printk(KERN_INFO "replace system call ...");

    p_pte = walk_pagetable((unsigned long)syscall_table + 8 * syscall_nr);
    if (!p_pte)
        return -1;

    pte = *p_pte;

    printk(KERN_INFO "walk_pagetable get pte=0x%llx", pte_val(pte));

    pte_new = pte_mkyoung(pte);
    pte_new = pte_mkwrite(pte_new);

    printk(KERN_INFO "mkwrite pte=0x%llx", pte_new);

    set_pte_at(init_mm_p, (unsigned long)syscall_table, p_pte, pte_new);

    p_pte = walk_pagetable((unsigned long)syscall_table);
    printk(KERN_INFO "walk_pagetable: pte=0x%llx", pte_val(*p_pte));

    printk(KERN_INFO "got sys_call_table[%d] at %16lx.\n", syscall_nr,
(unsigned long)orig_sys_ioctl);

    orig_sys_ioctl = syscall_table[syscall_nr];
    syscall_table[syscall_nr] = my_new_ioctl;

    printk(KERN_INFO "got sys_call_table[%d] at %16lx.\n", syscall_nr,
(unsigned long)syscall_table[syscall_nr]);

    set_pte_at(init_mm_p, (unsigned long)syscall_table, p_pte, pte);

    return 0;
}

static void __exit hack_syscall_exit(void)
{
    printk(KERN_INFO "syscall_release\n");
}

```

```
module_init(hack_syscall_init);
module_exit(hack_syscall_exit);
```

小明同学把这个内核模块编译之后，加载到 QEMU+Linux 5.0 系统^①中运行，马上就死机黑屏了，小明马上懵了！

```
root@benshu:~/mnt# insmod testsyscall_issue.ko
[ 191.649281] testsyscall_issue: loading out-of-tree module taints kernel.
[ 191.679779] testsyscall_issue: module verification failed: signature and/or required key missing - tainting kernel
[ 191.817883] Found the sys_call_table at ffff000011732b20.
[ 191.964776] replace system call ...
[ 191.965620] walk_pagetable get pte=0x7FFFC003
[ 191.965807] mknwrite pte=0x800007FFFC403
[ 191.967559] walk_pagetable: pte=0x800007fffc403
[ 191.968034] got sys_call_table[29] at 0.
[ 191.984000] Unable to handle kernel write to read-only memory at virtual address ffff000011732c08
[ 191.984900] Mem abort info:
[ 191.985070] ESR = 0x9600004f
[ 191.985330] Exception class = DABT (current EL), IL = 32 bits
[ 191.985566] SET = 0, FnV = 0
[ 191.985720] EA = 0, S1PTW = 0
[ 191.985911] Data abort info:
[ 191.986069] ISV = 0, ISS = 0x0000004f
[ 191.987128] CM = 0, WhR = 1
[ 191.988815] swapper_pgtable: 4k pages, 48-bit VAs, pgdp = (____ptrval____)
[ 191.989668] [ffff000011732c08] pgd=000000007ffffe003, pud=000800007fffc403, pte=00e000004173279
[ 191.994840] Internal error: Oops: 9600004f [#1] SMP
[ 191.996028] Modules linked in: testsyscall_issue(OE+)
[ 191.998209] CPU: 1 PID: 565 Comm: insmod Kdump: loaded Tainted: G OE 5.0 #1
[ 191.999715] Hardware name: linux_dummy-virt (DT)
[ 192.000771] pstate: 60000005 (nZCv daif -PAN -UAO)
[ 192.003707] pc : hack_syscall_init+0x49c/0x1000 [testsyscall_issue]
[ 192.004347] lr : hack_syscall_init+0x438/0x1000 [testsyscall_issue]
[ 192.004973] sp : fffff80002451f5b0
[ 192.005440] x29: fffff80002451f5b0 x28: fffff800024549c00
[ 192.006139] x27: 0000000000000000 x26: 0000000000000000
[ 192.007305] x25: 0000000056000000 x24: 0000000000000015
[ 192.008007] x23: 0000000040001000 x22: 0000ffff88ca4d44
```

你能帮小明把这个实验代码改好吗？

提示：要把小明的代码完全能满足实验的要求，期间会经历过 4 次的不同的宕机，你有时间来帮助小明同学吗^②？

7.10 小结（新增）

我们在实验 6 留了一些思考题。我们给读者一些提示。

1. 先来考察内核空间申请的 buffer。在本实验中使用 kmalloc()函数来分配内存。

➤ 站在 CPU 角度来看，CPU 访问这个 buffer，这个 buffer 中的页（page）对应的物理内存和对应的虚拟内存分别指向哪里？它对应的 PTE 页表又在哪里？

[笨叔] 这个 buffer 是通过 kmalloc()函数来分配的，因此它的映射关系在内核空间，属于线性映射。物理内存位于低端内存（仅对于 32 位处理器来说），而它对应的虚拟内存就是内核空间线性映射的区域（对于 32 为处理器来说，大概在 3G 往上的几百兆范围内）。它的 PTE 页表项反映了线性映射，也就是内核空间的线性映射区域通过页表寻址到低端物理内存。

^① <https://github.com/figozhang/linux-5.0-kdump>

^② 笨叔就本实验录制了 2 个多小时的详细讲解，有兴趣同学可以参见附录 2。

7.10 小结 (新增)

- 它的 cache 是打开还是关闭的？

[笨叔]在本实验中，这个线性映射对于的 cache 是打开的。PTE 页表项中，有描述 cache 相关的属性。对于 ARM32 处理器来说，PTE 页表中的 C 字段和 B 字段以及 TEX 字段组成了对 cache 访问的属性，我们成为内存属性。见 ARMv7 芯片手册^①的第 B3.5.1 章的图 B3-5。

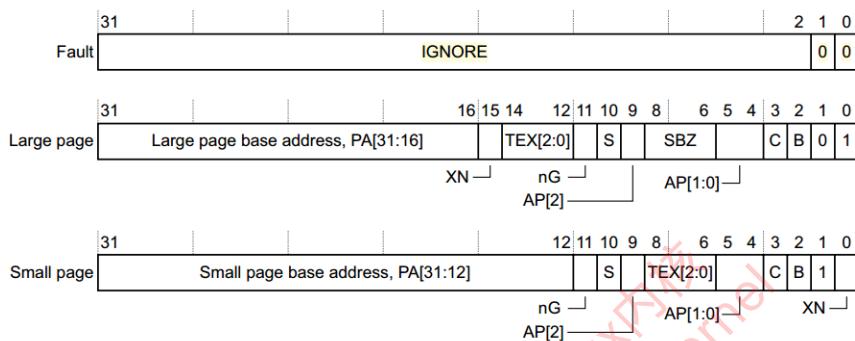


Figure B3-5 Short-descriptor second-level descriptor formats

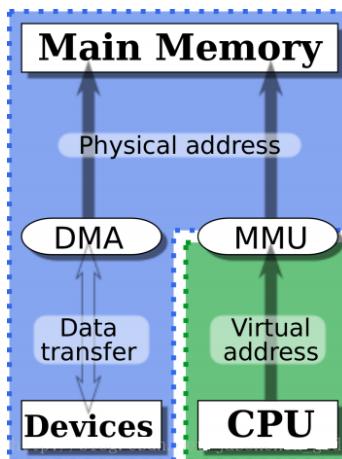
关于内存属性相关具体内容，读者可以查看 ARMv7 芯片手册的第 B3.8.2 章的表 B3-10。

- 若这时候有一个硬件设备也需要来访问这个 buffer，比如硬件设备想通过 DMA 来访问我们这个 buffer，我们是否考虑 cache 的问题？CPU 和 DMA 同时访问这个 buffer，怎么办？如何关闭这个 buffer 的 cache。
[笨叔] 当 CPU 和 DMA 都同时访问一个 buffer 时候，需要考虑 cache 的因素，这是所谓的 cache 一致性。

CPU 写内存的时候有两种方式：

1. write through: CPU 直接写内存，不经过 cache。
2. write back: CPU 只写到 cache 中。cache 的硬件控制器会在稍后把数据写入到内存中。

^① <ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition>



DMA 在内存到外设直接进行数据搬移，但 DMA 不能直接访问 CPU 的 cache。CPU 在读内存的时候，如果 cache 命中则直接从 cache 去读，而不会去从内存读。

CPU 写内存的时候，会先写入到 cache，稍后才由 cache 回写到内存。

以数据为观察点，我们可以看到两个维度：

数据从外设到内存。如果 DMA 把数据从外设写到内存，CPU 中 cache 中的数据（如果有的话）就是旧数据，这时 CPU 再读内存，若命中 cache 得到的就是旧数据。这种情况下，我们应该调用内核提供的函数接口把对应的 cache 置无效（Invalidate），这样 CPU 就会从内存中读到最新数据。

数据从内存到外设。CPU 写数据到内存时，DMA 把数据从内存读到外设的 FIFO 中。这种情况下，如果只是先写到了 cache，则内存里的数据就是旧数据了，这时候 DMA 读到的数据也是旧数据。这种情况下，我们应该先把对应的 cache 数据 flush 到内存中，然后在触发 DMA 去读数据。

这两种情况（两个方向）都存在 cache 一致性问题。例如，网卡发包的时候，CPU 将数据写到 cache，而网卡的 DMA 从内存里去读数据，就发送了错误的数据。

- 内核中有哪些接口函数可以分配关闭 cache 的 buffer？

[笨叔] 内核提供了多种分配 DMA buffer 的接口函数。

1. 一致性 DMA 缓存(Coherent DMA buffers)。

7.10 小结 (新增)

相关的接口函数有 `dma_alloc_coherent()` 和 `dma_free_coherent()`。

2. 流式 DMA 映射(DMA Streaming Mapping)

相关接口函数有 `dma_map_sg()`, `dma_unmap_sg()`, `dma_map_single()` 和 `dma_unmap_single()` 等。

2. 我们来考察通过 `mmap` 映射到用户空间的这个 user buffer。

- 这个 user buffer 的物理内存是在哪里？对应的虚拟内存是在哪里？
[笨叔] 使用 `kmalloc` 分配的 buffer 为例，物理内存就是 `kmalloc` 分配的物理页面。对应的虚拟内存就是用户空间的内存了，即进程地址空间。这是 `mmap` 函数在进程地址空间分配的 VMA 区域。对于 32 位处理器来说，这个区域是在 1GB~3GB 区域内。
- 对应的 PTE 页表项是在哪里？
[笨叔] `mmap` 映射的页面，在内核空间里需要做一次映射，也就是 VMA 对于的虚拟地址和这个 buffer 的物理地址做映射，即建立页表的关系，在本实验中通过 `remap_pfn_range()` 函数来建立页表。
- CPU 访问这个 user buffer，它对应的 cache 是关闭还是打开的？
[笨叔] 因为是 VMA 到物理内存之间建立映射关系，因此默认情况下，这个 cache 关系的打开的。刚才我们提到 PTE 页表项里有关 cache 的属性。我们可以通过 `pgprot_noncached()` 函数来改变 PTE 页表相关的内存属性。

进程管理

8.1 实验 1: fork 和 clone

1. 实验目的

了解和熟悉 Linux 中 fork 系统调用和 clone 系统调用的用法。

2. 实验要求

- 1) 使用 fork()函数创建一个子进程，然后在父进程和子进程中分别使用 printf 语句来判断谁是父进程和子进程。
- 2) 使用 clone()函数创建一个子进程。如果父进程和子进程共同访问一个全局变量，结果会如何？如果父进程比子进程先消亡，结果会如何？
- 3) 请思考，如下代码中会打印几个“_”？

```
int main(void)
{
    int i;
    for(i=0; i<2; i++){
        fork();
        printf("_\n");
    }
    wait(NULL);
    wait(NULL);
    return 0;
}
```

3. 实验步骤

- (1) fork 例子
 - (2) clone 例子
- 进入本实验例子。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_8/lab1
```

编译测试程序，拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc clone_test.c -o clone_test -static
# cp clone_test /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules
```

启动 QEMU 虚拟机。

```
# cd runninglinuxkernel_4.0
#./run.sh arm32
```

8.2 实验 2：内核线程

进入/mnt 目录，运行 close_test 程序。

```
/mnt # ./clone_test
starting parent process, pid=778
Setting a clone child thread with stacksize = 16384.... with tid=779
starting child thread fn, pid=779
parent running: j=0, param=1000 secs
child thread running: j=0, param=1
parent running: j=1, param=1001 secs
child thread running: j=1, param=2
parent running: j=2, param=1002 secs
child thread running: j=2, param=3
parent running: j=3, param=1003 secs
child thread running: j=3, param=4
parent running: j=4, param=1004 secs
child thread running: j=4, param=5
child thread running: j=5, param=1005
parent running: j=5, param=1006 secs
parent killitself
/mnt # child thread running: j=6, param=1006
child thread running: j=7, param=1007
child thread running: j=8, param=1008
child thread running: j=9, param=1009
child thread_fn exit
/mnt #
```

(3) 打印几个 “_”

8.2 实验 2：内核线程

1. 实验目的

了解和熟悉 Linux 内核中是如何创建内核线程的。

2. 实验要求

- 1) 写一个内核模块，创建一组内核线程，每个 CPU 一个内核线程。
- 2) 在每个内核线程中，打印当前 CPU 的状态，比如 ARM 通用寄存器的值。
- 3) 在每个内核线程中，打印当前进程的优先级等信息。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_8/lab2_k
thread

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@ubuntu:lab2_kthread$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/running
linuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab2_kthread modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8
/lab2_kthread/kthread.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8
/lab2_kthread/kthread_test.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8
/lab2_kthread/kthread_test.mod.o
  LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8
/lab2_kthread/kthread_test.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
```

拷贝内核模块到 kmodules 目录。

```
# cp kthread_test.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
[ 42.818597] running cpu=0.
[ 42.819605] cpsr:0x60000013, sp:0xc451ff18
[ 42.822239] msleep over in Thread Function cpu=1.
[ 42.822754] running cpu=1.
[ 42.823085] kdemo/0 pid:776, nice:0 prio:120 static_prio:120 normal_prio:120
[ 42.823377] SLEEP in Thread Function cpu=0.
[ 42.824566] cpsr:0x60000013, sp:0xc452bf18
[ 42.825094] kdemo/1 pid:777, nice:0 prio:120 static_prio:120 normal_prio:120
[ 42.825904] SLEEP in Thread Function cpu=1.
[ 42.830464] msleep over in Thread Function cpu=2.
[ 42.833082] running cpu=2.
[ 42.834322] cpsr:0x60000013, sp:0xc452ff18
[ 42.835943] kdemo/2 pid:778, nice:0 prio:120 static_prio:120 normal_prio:120
[ 42.838735] SLEEP in Thread Function cpu=2.
[ 42.841378] msleep over in Thread Function cpu=3.
[ 42.842001] running cpu=3.
[ 42.842296] cpsr:0x60000013, sp:0xc4533f18
[ 42.842782] kdemo/3 pid:779, nice:0 prio:120 static_prio:120 normal_prio:120
[ 42.843715] SLEEP in Thread Function cpu=3.
[ 44.826403] msleep over in Thread Function cpu=0.
```

4. 实验代码

```
1 #include <linux/module.h>
2 #include <linux/init.h>
3 #include <linux/module.h>
4 #include <linux/kthread.h>
5 #include <linux/delay.h>
6
7 static struct task_struct *tsk[NR_CPUS];
8
9 static void show_reg(void)
10{
11    unsigned int cpsr, sp;
12
13    asm("mrs %0, cpsr" : "=r" (cpsr) : : "cc");
14    asm("mov %0, sp" : "=r" (sp) : : "cc");
15
16    printk("cpsr:0x%x, sp:0x%x\n", cpsr, sp);
```

8.2 实验 2：内核线程

```

17}
18
19static void show_prio(void)
20{
21    struct task_struct *task = current;
22
23    printk("%s pid:%d, nice:%d prio:%d static_prio:%d normal_prio:%d\n",
24           task->comm, task->pid,
25           PRIO_TO_NICE(task->static_prio),
26           task->prio, task->static_prio,
27           task->normal_prio);
28}
29
30static void print_cpu(char *s)
31{
32    preempt_disable();
33    pr_info("%s cpu=%d.\n", s, smp_processor_id());
34    preempt_enable();
35}
36
37static int thread_fun(void *t)
38{
39    do {
40        print_cpu("SLEEP in Thread Function ");
41        msleep_interruptible(2000);
42        print_cpu("msleep over in Thread Function");
43        print_cpu("running");
44        show_reg();
45        show_prio();
46    } while (!kthread_should_stop());
47    return 0;
48}
49
50static int __init my_init(void)
51{
52    int i;
53    print_cpu("Loading module");
54    for_each_online_cpu(i) {
55        tsk[i] = kthread_create(thread_fun, NULL, "kdemo/%d", i);
56        if (!tsk[i]) {
57            pr_info("Failed to generate a kernel thread\n");
58            return -1;
59        }
60        kthread_bind(tsk[i], i);
61        pr_info("About to wake up and run the thread for cpu=%d\n", i);
62        wake_up_process(tsk[i]);
63        pr_info("Starting thread for cpu %d", i);
64        print_cpu("on");
65    }
66    return 0;
67}
68
69static void __exit my_exit(void)
70{
71    int i;
72    for_each_online_cpu(i) {
73        pr_info("Kill Thread %d", i);
74        kthread_stop(tsk[i]);
75        print_cpu("Kill was done on ");
76    }
77}
78
79module_init(my_init);
80module_exit(my_exit);

```

```

81
82MODULE_AUTHOR("Ben ShuShu");
83MODULE_LICENSE("GPL v2");

```

8.3 实验 3：后台守护进程

1. 实验目的

通过本实验了解和熟悉 Linux 是如何创建和使用后台守护进程的。

2. 实验要求

- 1) 写一个用户程序，创建一个守护进程。
- 2) 该守护进程每隔 5 秒去查看当前内核的日志中是否有 oops 错误。

3. 实验步骤

进入本实验例子。

```

#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_8/lab3_daemon

```

编译测试程序，拷贝到 kmodules 目录。

```

# arm-linux-gnueabi-gcc daemon_test1.c -o daemon_test1 --static
# cp daemon_test1 /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules

```

启动 QEMU 虚拟机。

```

# cd runninglinuxkernel_4.0
# ./run.sh arm32

```

进入/mnt 目录，运行 daemon_test1 程序。

```

/mnt # ./daemon_test1

```

```

/mnt # ls
2019.8.30.12.14.51.log  kthread_test.ko      mydevdemo_mmap.ko
2019.8.30.12.14.56.log  mydemo.ko          mytest.ko
README                   mydemo_fasync.ko    test
clone_test               mydemo_misc.ko     test_getpuid_syscall
daemon_test1             mydemo_poll.ko
/mnt #

```

在 mnt 目录可以看到 “2019.8.30*.log” 文件。打开这些 log 文件，可以看到的内容是内核的 dmesg 的日志。

另外我们通过 top 命令可以看到 daemon_test1 进程，进程 PID 为 775。

8.3 实验 3：后台守护进程

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
781	772	0	R	2308	2.6	1	13.0	top
11	2	0	SW	0	0.0	1	5.6	[ksoftirqd/1]
15	2	0	SW	0	0.0	2	0.8	[ksoftirqd/2]
7	2	0	SW	0	0.0	2	0.8	[rcu_sched]
410	2	0	SW	0	0.0	0	0.8	[kworker/0:1]
1	0	0	S	2308	2.6	1	0.0	{linuxrc} init
772	1	0	S	2308	2.6	0	0.0	-/bin/sh
775	1	0	S	768	0.8	0	0.0	./daemon test1
753	2	0	SWN	0	0.0	1	0.0	[kmemleak]
6	2	0	SW	0	0.0	1	0.0	[kworker/u8:0]
3	2	0	SW	0	0.0	0	0.0	[ksoftirqd/0]
19	2	0	SW	0	0.0	3	0.0	[ksoftirqd/3]
24	2	0	SW	0	0.0	3	0.0	[kworker/u8:1]
23	2	0	SW	0	0.0	1	0.0	[kdevtmpfs]
409	2	0	SW	0	0.0	1	0.0	[kworker/1:1]
328	2	0	SW	0	0.0	3	0.0	[kworker/3:1]
2	0	0	SW	0	0.0	1	0.0	[kthreadd]
411	2	0	SW	0	0.0	2	0.0	[kworker/2:1]
9	2	0	SW	0	0.0	0	0.0	[migration/0]
14	2	0	SW	0	0.0	2	0.0	[migration/2]

4. 实验代码

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <time.h>
5 #include <fcntl.h>
6 #include <string.h>
7 #include <sys/stat.h>
8 #include <sys/klog.h>
9
10 #define FALLBACK_KLOG_BUF_SHIFT 17 /* CONFIG_LOG_BUF_SHIFT in kernel */
11 #define FALLBACK_KLOG_BUF_LEN   (1 << FALLBACK_KLOG_BUF_SHIFT)
12
13 #define KLOG_CLOSE          0
14 #define KLOG_OPEN           1
15 #define KLOG_READ           2
16 #define KLOG_READ_ALL       3
17 #define KLOG_READ_CLEAR     4
18 #define KLOG_CLEAR          5
19 #define KLOG_CONSOLE_OFF    6
20 #define KLOG_CONSOLE_ON     7
21 #define KLOG_CONSOLE_LEVEL  8
22 #define KLOG_SIZE_UNREAD   9
23 #define KLOG_SIZE_BUFFER    10
24
25 /* we use 'Linux version' string instead of Oops in this lab */
26 // #define OOPS_LOG "Oops"
27 #define OOPS_LOG "Linux version"
28
29 int save_kernel_log(char *buffer)
30 {
31     char path[128];
32     time_t t;
33     struct tm *tm;
34     int fd;
35
36     t = time(0);
37     tm = localtime(&t);
38
39     snprintf(path, 128, "/mnt/%d.%d.%d.%d.log", tm->tm_year+1900,
40               tm->tm_mon+1, tm->tm_mday, tm->tm_hour,
```

奔跑吧 linux 社区出品

```

41         tm->tm_min, tm->tm_sec);
42     printf("%s\n", path);
43
44     fd = open(path, O_WRONLY|O_CREAT, 0644);
45     if(fd == -1) {
46         printf("open error\n");
47         return -1;
48     }
49     write(fd, buffer, strlen(buffer));
50     close(fd);
51
52     return 0;
53 }
54
55 int check_kernel_log()
56 {
57     char *buffer;
58     char *p;
59     ssize_t klog_size;
60     int ret = -1;
61     int size;
62
63     printf("start kernel log\n");
64
65     klog_size = klogctl(KLOG_SIZE_BUFFER, 0, 0);
66     if (klog_size <= 0) {
67         klog_size = FALLBACK_KLOG_BUF_LEN;
68     }
69
70     printf("kernel log size: %d\n", klog_size);
71
72     buffer = malloc(klog_size + 1);
73     if (!buffer)
74         return -1;
75
76     size = klogctl(KLOG_READ_ALL, buffer, klog_size);
77     if (size < 0) {
78         printf("klogctl read error\n");
79         goto done;
80     }
81
82     buffer[size] = '\0';
83
84     /* check if oops in klog */
85     p = strstr(buffer,OOPS_LOG);
86     if (p) {
87         printf("we found '%s' on kernel log\n", OOPS_LOG);
88         save_kernel_log(buffer);
89         ret = 0;
90     }
91 done:
92     free(buffer);
93     return ret;
94 }
95
96 int main(void)
97 {
98     if(daemon(0,0) == -1) {
99         printf("daemon error");
100        return 0;
101    }
102
103    while(1) {
104        check_kernel_log();

```

8.4 实验 4：进程权限

```

105
106     sleep(5);
107 }
108
109 return 0;
110}

```

8.4 实验 4：进程权限

1. 实验目的

了解和熟悉 Linux 是如何进行进程的权限管理的。

2. 实验要求

写一个用户程序，限制该程序的一些资源，比如进程的最大虚拟内存空间等。

3. 实验步骤

进入本实验例子。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_8/lab4
```

编译测试程序，拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc resource_limit.c -o resource_limit -static
# cp resource_limit /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules
```

启动 QEMU 虚拟机。

```
# cd runninglinuxkernel_4.0
#./run.sh arm32
```

进入/mnt 目录，运行 resource_limit 程序。

```
/mnt # ./resource_limit
Printing out all limits for pid=774:
RLIMIT_CPU=0: cur=        4294967295,    max=        4294967295
RLIMIT_FSIZE=1: cur=        4294967295,    max=        4294967295
RLIMIT_DATA=2: cur=        4294967295,    max=        4294967295
RLIMIT_STACK=3: cur=        8388608,      max=        4294967295
RLIMIT_CORE=4: cur=          0,      max=        4294967295
RLIMIT_RSS=5: cur=        4294967295,    max=        4294967295
RLIMIT_NPROC=6: cur=          326,      max=            326
RLIMIT_NOFILE=7: cur=          1024,      max=          4096
RLIMIT_MEMLOCK=8: cur=          65536,      max=          65536
RLIMIT_AS=9: cur=        4294967295,    max=        4294967295
RLIMIT_LOCKS=10: cur=        4294967295,   max=        4294967295

Before Modification, this is RLIMIT_CORE:
  RLIMIT_CORE=4: cur=          0,      max=        4294967295
I forked off a child with pid = 0

After Modification, this is RLIMIT_CORE:
  RLIMIT_CORE=4: cur=        8388608,    max=        4294967295

In child pid= 775 this is RLIMIT CORE:
```

4. 实验步骤

```

1 #include <sys/time.h>
2 #include <sys/resource.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <sys/wait.h>
7 #include <errno.h>
8
9 #define DEATH(mess) { perror(mess); exit(errno); }
10
11 void do_limit(int limit, const char *limit_string, struct rlimit *rlim)
12 {
13     if (getrlimit(limit, rlim))
14         fprintf(stderr, "Failed in getrlimit\n");
15     printf("%15s=%2d: cur=%20lu,      max=%20lu\n", limit_string,
16           limit, rlim->rlim_cur, rlim->rlim_max);
17 }
18
19 void print_limits(void)
20 {
21     struct rlimit rlim;
22     do_limit(RLIMIT_CPU, "RLIMIT_CPU", &rlim);
23     do_limit(RLIMIT_FSIZE, "RLIMITFSIZE", &rlim);
24     do_limit(RLIMIT_DATA, "RLIMIT_DATA", &rlim);
25     do_limit(RLIMIT_STACK, "RLIMIT_STACK", &rlim);
26     do_limit(RLIMIT_CORE, "RLIMIT_CORE", &rlim);
27     do_limit(RLIMIT_RSS, "RLIMIT_RSS", &rlim);
28     do_limit(RLIMIT_NPROC, "RLIMIT_NPROC", &rlim);
29     do_limit(RLIMIT_NOFILE, "RLIMIT_NOFILE", &rlim);
30     do_limit(RLIMIT_MEMLOCK, "RLIMIT_MEMLOCK", &rlim);
31     do_limit(RLIMIT_AS, "RLIMIT_AS", &rlim);
32     do_limit(RLIMIT_LOCKS, "RLIMIT_LOCKS", &rlim);
33 }
34
35 void print_rusage(int who)
36 {
37     struct rusage usage;
38     if (getrusage(who, &usage))
39         DEATH("getrusage failed");
40
41     if (who == RUSAGE_SELF)
42         printf("For RUSAGE_SELF\n");
43     if (who == RUSAGE_CHILDREN)
44         printf("\nFor RUSAGE_CHILDREN\n");
45
46     printf
47         ("ru_utime.tv_sec, ru_utime.tv_usec = %4d %4d (user time used)\n",
48          (int)usage.ru_utime.tv_sec, (int)usage.ru_utime.tv_usec);
49     printf
50         ("ru_stime.tv_sec, ru_stime.tv_usec = %4d %4d (system time
used)\n",
51          (int)usage.ru_stime.tv_sec, (int)usage.ru_stime.tv_usec);
52     printf("ru_maxrss = %4ld (max resident set size)\n", usage.ru_maxrss);
53     printf("ru_ixrss = %4ld (integral shared memory size)\n",
54           usage.ru_ixrss);
55     printf("ru_idrss = %4ld (integral unshared data size)\n",
56           usage.ru_idrss);
57     printf("ru_isrss = %4ld (integral unshared stack size)\n",
58           usage.ru_isrss);
59     printf("ru_minflt = %4ld (page reclaims)\n", usage.ru_minflt);
60     printf("ru_majflt = %4ld (page faults)\n", usage.ru_majflt);
61     printf("ru_nswap = %4ld (swaps)\n", usage.ru_nswap);

```

8.4 实验 4：进程权限

```

62     printf("ru_inblock = %4ld (block input operations)\n",
63         usage.ru_inblock);
64     printf("ru_oublock = %4ld (block output operations)\n",
65         usage.ru_oublock);
66     printf("ru_msgrcv = %4ld (messages received)\n", usage.ru_msgrcv);
67     printf("ru_nsignals= %4ld (signals received)\n", usage.ru_nsignals);
68     printf("ru_nvcsw= %4ld (voluntary context switches)\n",
69         usage.ru_nvcsw);
70     printf("ru_nivcsw= %4ld (involuntary context switches)\n",
71         usage.ru_nivcsw);
72 }
73
74
75 int main(int argc, char *argv[])
76 {
77     struct rlimit rlim;
78     pid_t pid = 0;
79     int status = 0, nchildren = 3, i;
80
81     /* Print out all limits */
82
83     printf("Printing out all limits for pid=%d:\n", getpid());
84     print_limits();
85
86     /* change and printout the limit for core file size */
87
88     printf("\nBefore Modification, this is RLIMIT_CORE:\n");
89     do_limit(RLIMIT_CORE, "RLIMIT_CORE", &rlim);
90     rlim.rlim_cur = 8 * 1024 * 1024;
91     printf("I forked off a child with pid = %d\n", (int)pid);
92
93     setrlimit(RLIMIT_CORE, &rlim);
94     printf("\nAfter Modification, this is RLIMIT_CORE:\n");
95     do_limit(RLIMIT_CORE, "RLIMIT_CORE", &rlim);
96
97     /* fork off the nchildren */
98
99     fflush(stdout);
100    for (i = 0; i < nchildren; i++) {
101        pid = fork();
102        if (pid < 0)
103            DEATH("Failed in fork");
104        if (pid == 0) { /* any child */
105            printf("\nIn child pid= %d this is RLIMIT_CORE:\n",
106                  (int)getpid());
107            do_limit(RLIMIT_CORE, "RLIMIT_CORE", &rlim);
108            fflush(stdout);
109            sleep(3);
110            exit(EXIT_SUCCESS);
111        }
112    }
113
114    while (pid > 0) { /* parent */
115        pid = wait(&status);
116        printf("Parent got return on pid=%dn\n", (int)pid);
117    }
118
119    printf(" ****\n");
120    print_rusage(RUSAGE_SELF);
121    print_rusage(RUSAGE_CHILDREN);
122
123    exit(EXIT_SUCCESS);
124}

```

8.5 实验 5：设置优先级

1. 实验目的

了解和熟悉 Linux 中 getpriority() 和 setpriority() 系统调用的用法。

2. 实验要求

- 1) 写一个用户进程，使用 setpriority() 来修改进程的优先级，然后使用 getpriority() 函数来验证。
- 2) 可以通过一个 for 循环来依次修改进程的优先级（-20~19）。

3. 实验步骤

进入本实验例子。

```
#cd  
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_8/lab5
```

编译测试程序，拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc process_priority.c -o process_priority --static  
# cp process_priority /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules
```

启动 QEMU 虚拟机。

```
# cd runninglinuxkernel_4.0  
#./run.sh arm32
```

进入 /mnt 目录，运行 process_priority 程序。

8.5 实验 5：设置优先级

```
/mnt # ./process_priority
Examining priorities forPID = 781
 Previous Requested Assigned
    0      -20      -20
   -20      -18      -18
   -18      -16      -16
   -16      -14      -14
   -14      -12      -12
   -12      -10      -10
   -10      -8       -8
   -8       -6       -6
   -6       -4       -4
   -4       -2       -2
   -2        0        0
    0        2        2
    2        4        4
    4        6        6
    6        8        8
    8       10       10
   10       12       12
   12       14       14
   14       16       16
   16       18       18
/mnt #
```

4. 实验代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/time.h>
5 #include <sys/resource.h>
6 #include <errno.h>
7
8 int main(int argc, char *argv[])
9 {
10     pid_t mypid;
11     int old_prio, new_prio, i, rc;
12
13     if (argc > 1) {
14         mypid = atoi(argv[1]);
15     } else {
16         mypid = getpid();
17     }
18
19     printf("\nExamining priorities forPID = %d \n", mypid);
20     printf("%10s%10s%10s\n", "Previous", "Requested", "Assigned");
21
22     for (i = -20; i < 20; i += 2) {
23
24         old_prio = getpriority(PRIO_PROCESS, (int)mypid);
25         rc = setpriority(PRIO_PROCESS, (int)mypid, i);
26         if (rc)
27             fprintf(stderr, "setpriority() failed ");
28
29         /* must clear errno before call to getpriority
30          because -1 is a valid return value */
31         errno = 0;
32
33         new_prio = getpriority(PRIO_PROCESS, (int)mypid);
34         printf("%10d%10d%10d\n", old_prio, i, new_prio);
35
36     }
37     exit(EXIT_SUCCESS);
38}
```

8.6 实验 6: per-cpu 变量

1. 实验目的

学会 Linux 内核中 per-cpu 变量的用法。

2. 实验要求

- 1) 写一个简单的内核模块，创建一个 per-cpu 变量，并且初始化该 per-cpu 变量，修改 per-cpu 变量的值，然后输出这些值。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@ubuntu:lab6_percpu$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6_percpu modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6_percpu/my_percpu.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6_percpu/my_percpu.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6_percpu/my_percpu.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_8/lab6_percpu/my_percpu.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rlk@ubuntu:lab6_percpu$
```

拷贝内核模块到 kmodues 目录。

```
# cp my_percpu.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodues/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

8.6 实验 6 : per-cpu 变量

```
/mnt # insmod mypercpu.ko
[ 2625.077780] module loaded at 0xbff022000
[ 2625.078341] init: cpubar on cpu0 = 15
[ 2625.078788] init: cpubar on cpu1 = 15
[ 2625.079564] init: cpubar on cpu2 = 15
[ 2625.080471] init: cpubar on cpu3 = 15
[ 2625.082389] init: cpu:0 cpualloc = 100
[ 2625.083327] init: cpu:1 cpualloc = 100
[ 2625.084275] init: cpu:2 cpualloc = 100
[ 2625.084912] init: cpu:3 cpualloc = 100
```

卸载内核模块。

```
/mnt # rmmod mypercpu.ko
[ 2629.591870] exit module...
[ 2629.592277] cpubar cpu0 = 20
[ 2629.592748] exit: cpualloc0 = 100
[ 2629.593137] cpubar cpu1 = 15
[ 2629.593403] exit: cpualloc1 = 100
[ 2629.593715] cpubar cpu2 = 15
[ 2629.594460] exit: cpualloc2 = 100
[ 2629.594858] cpubar cpu3 = 15
[ 2629.595791] exit: cpualloc3 = 100
[ 2629.596386] Bye: module unloaded from 0xbff020000
/mnt #
```

4. 实验代码

```
1 #include <linux/module.h>
2 #include <linux/init.h>
3 #include <linux/percpu.h>
4 #include <linux/cpumask.h>
5
6 static DEFINE_PER_CPU(long, cpubar) = 10;
7 static long __percpu *cpualloc;
8
9 static int __init my_init(void)
10{
11    int cpu;
12
13    pr_info("module loaded at 0x%p\n", my_init);
14
15    /* modify the cpubar value */
16    for_each_possible_cpu(cpu){
17        per_cpu(cpubar, cpu) = 15;
18        pr_info("init: cpubar on cpu%d = %ld\n",
19                cpu, get_cpu_var(cpubar));
20        put_cpu_var(cpubar);
21    }
22
23    __this_cpu_write(cpubar, 20);
24
25    /* alloc a percpu value */
26    cpualloc = alloc_percpu(long);
27
28    /* set all cpu for this value */
29    for_each_possible_cpu(cpu){
30        *per_cpu_ptr(cpualloc, cpu) = 100;
31        pr_info("init: cpu:%d cpualloc = %ld\n",
32                cpu, *per_cpu_ptr(cpualloc, cpu));
33    }
34}
```

```

36     return 0;
37}
38
39static void __exit my_exit(void)
40{
41     int cpu;
42     pr_info("exit module...\n");
43
44     for_each_possible_cpu(cpu) {
45         pr_info("cpuvar cpu%d = %ld\n", cpu, per_cpu(cpuvar, cpu));
46         pr_info("exit: cpualloc%d = %ld\n", cpu, *per_cpu_ptr(cpualloc,
47         cpu));
48     }
49     free_percpu(cpualloc);
50
51     pr_info("Bye: module unloaded from 0x%p\n", my_exit);
52}
53
54module_init(my_init);
55module_exit(my_exit);
56
57MODULE_AUTHOR("Ben ShuShu");
58MODULE_LICENSE("GPL v2");

```

per-cpu 变量是 Linux 内核中同步机制的一种。当系统中所有的 CPU 都访问共享的一个变量 v 时，CPU0 修改了变量 v 的值时，CPU1 也在同时修改变量 v 的值，那么就会导致变量 v 值不正确。一个可行的办法就是 CPU0 访问变量 v 时使用原子加锁指令，CPU1 访问变量 v 时只能等待了，可是这会有两个比较明显的缺点。

- 原子操作是比较耗时的。
- 现代处理器中，每个 CPU 都有 L1 缓存，那么多 CPU 同时访问同一个变量时会导致缓存一致性问题。当某个 CPU 对共享数据变量 v 修改后，其他 CPU 上对应的缓存行需要做无效操作，这对性能是有所损耗的。

per-cpu 变量为了解决上述问题出现一种有趣的特性，它为系统中每个处理器都分配该变量的副本。这样在多处理器系统中，当处理器只能访问属于它自己的那个变量副本，不需要考虑与其他处理器的竞争问题，还能充分利用处理器本地的硬件缓存来提升性能。

3) 声明 per-cpu 变量。per-cpu 变量的定义和声明有两种方式：一个是静态声明，另一个是动态分配。

静态 per-cpu 变量通过 DEFINE_PER_CPU 和 DECLARE_PER_CPU 宏定义和声明一个 per-cpu 变量。这些变量与普通变量的主要区别是放在一个特殊的段中。

```

#define DECLARE_PER_CPU(type, name) \
    DECLARE_PER_CPU_SECTION(type, name, "")

#define DEFINE_PER_CPU(type, name) \
    DEFINE_PER_CPU_SECTION(type, name, "")

```

动态分配和释放 per-cpu 变量的 API 函数如下。

```

#define alloc_percpu(type) \
    (typeof(type) __percpu *)__alloc_percpu(sizeof(type), \

```

8.6 实验 6 : per-cpu 变量

```
__alignof__(type))  
void free_percpu(void __percpu *ptr)
```

4) 使用 per-cpu 变量。对于静态定义的 per-cpu 变量，可以通过 `get_cpu_var()` 和 `put_cpu_var()` 函数来访问和修改 per-cpu 变量，这两个函数内置了关闭和打开内核抢占的功能。另外需要注意的是，这两个函数需要配对使用。

```
#define get_cpu_var(var) \
(*({ \
    preempt_disable(); \
    this_cpu_ptr(&var); \
})) \
#define put_cpu_var(var) \
do { \
    (void) &(var); \
    preempt_enable(); \
} while (0)
```

访问动态分配的 per-cpu 变量需要通过下面的接口函数来访问。

```
#define put_cpu_ptr(var) \
do { \
    (void) (var); \
    preempt_enable(); \
} while (0) \
#define get_cpu_ptr(var) \
({ \
    preempt_disable(); \
    this_cpu_ptr(var); \
})
```

第 9 章

同步管理

9.1 实验 1：自旋锁

1. 实验目的

了解和熟悉自旋锁的使用。

2. 实验要求

写一个简单的内核模块，然后测试如下功能。

- 在自旋锁里面，调用 alloc_page(GFP_KERNEL)函数来分配内存，观察会发生什么情况。
- 手工创造递归死锁，观察会发生什么情况。
- 手工创造 AB-BA 死锁，观察会发生什么情况。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rnk@ubuntu:lab1$ export ARCH=arm
rnk@ubuntu:lab1$ export CROSS_COMPILE=arm-linux-gnueabi-
rnk@ubuntu:lab1$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1 modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.o
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.c: In function 'lockdep_thread':
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.c:41:1: warning: no return statement in function returning non-void [-Wreturn-type]
}
^
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab1/spinlock_nest.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rnk@ubuntu:lab1$
```

拷贝内核模块到 kmodues 目录。

```
# cp spinlock-nest.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

9.1 实验 1：自旋锁

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod spinlock-nest.ko
```

查看内核日志信息。

```
/mnt # [ 55.761281] INFO: rCU_sched self-detected stall on CPU
[ 55.763414] 0: (2099 ticks this GP) idle=5c5/1400000000000001/0
softirq=2914/2917 fqs=2099
[ 55.764173] (t=2100 jiffies g=88 c=87 q=56)
[ 55.764599] Task dump for CPU 0:
[ 55.764996] lockdep_test R running 0 776 2 0x000000002
[ 55.766097] [<c002475c>] ( unwind_backtrace ) from [<c001d640>]
(show_stack+0x2c/0x38)
[ 55.766558] [<c001d640>] (show_stack) from [<c00960c0>]
(sched_show_task+0x1a8/0x1b4)
[ 55.766960] [<c00960c0>] (sched_show_task) from [<c009cd8c>]
(dump_cpu_task+0x58/0x64)
[ 55.767369] [<c009cd8c>] (dump_cpu_task) from [<c00db988>]
(rcu_dump_cpu_stacks+0x90/0x10c)
[ 55.767811] [<c00db988>] (rcu_dump_cpu_stacks) from [<c00dbf9c>]
(print_cpu_stall+0x160/0x210)
[ 55.768237] [<c00dbf9c>] (print_cpu_stall) from [<c00dc18c>]
(check_cpu_stall+0x140/0x194)
[ 55.768646] [<c00dc18c>] (check_cpu_stall) from [<c00e14dc>]
(__rcu_pending+0x3c/0x300)
[ 55.769077] [<c00e14dc>] (__rcu_pending) from [<c00e1800>]
(rcu_pending+0x60/0xb4)
[ 55.769461] [<c00e1800>] (rcu_pending) from [<c00df4b0>]
(rcu_check_callbacks+0x130/0x234)
[ 55.769886] [<c00df4b0>] (rcu_check_callbacks) from [<c00eb424>]
(update_process_times+0x38/0x70)
[ 55.770381] [<c00eb424>] (update_process_times) from [<c0105194>]
(tick_periodic+0x1a0/0x1b4)
[ 55.770841] [<c0105194>] (tick_periodic) from [<c01051dc>]
(tick_handle_periodic+0x34/0xb0)
[ 55.771270] [<c01051dc>] (tick_handle_periodic) from [<c0022a88>]
(twd_handler+0x38/0x50)
[ 55.771702] [<c0022a88>] (twd_handler) from [<c00d20d4>]
(handle_percpu_devid_irq+0x218/0x33c)
[ 55.772152] [<c00d20d4>] (handle_percpu_devid_irq) from [<c00ca11c>]
(generic_handle_irq+0x50/0x60)
[ 55.772622] [<c00ca11c>] (generic_handle_irq) from [<c00ca2b4>]
(__handle_domain_irq+0x188/0x240)
[ 55.773080] [<c00ca2b4>] (__handle_domain_irq) from [<c00089e4>]
(gic_handle_irq+0xc4/0x120)
[ 55.773516] [<c00089e4>] (gic_handle_irq) from [<c0b12d80>]
(__irq_svc+0x40/0x54)
[ 55.774045] Exception stack(0xc453be60 to 0xc453bea8)
[ 55.774442] be60: 00010000 00020000 00000001 00000000 c4524100 c007f7c4
00000000 00000000
[ 55.774875] be80: 00000000 00000000 00000000 00000000 c10bc184 c453bea8
bf0000fc c0b11a38
[ 55.775282] bea0: 20000013 ffffffff
[ 55.775544] [<c0b12d80>] (__irq_svc) from [<c0b11a38>]
(__raw_spin_lock+0x58/0x94)
[ 55.776080] [<c0b11a38>] (__raw_spin_lock) from [<bf0000fc>]
(nest_lock+0xfc/0x1c4 [spinlock_nest])
[ 55.776582] [<bf0000fc>] (nest_lock [spinlock_nest]) from [<bf000200>]
(lockdep_thread+0x3c/0x64 [spinlock_nest])
[ 55.777109] [<bf000200>] (lockdep_thread [spinlock_nest]) from
```

```
[<c007fa04>] (kthread+0x240/0x24c)
[ 55.777564] [<c007fa04>] (kthread) from [<c0014de0>]
(ret_from_fork+0x14/0x34)
```

4. 死锁检测

要在 Linux 内核中使用 Lockdep 功能，需要打开 CONFIG_DEBUG_LOCKDEP 选项。修改内核配置文件 arch/arm/configs/vexpress_defconfig。

```
CONFIG_PROVE_LOCKING=y
CONFIG_LOCKDEP=y
CONFIG_LOCK_STAT=y
CONFIG_DEBUG_LOCKDEP=y
```

```
diff --git a/arch/arm/configs/vexpress_defconfig b/arch/arm/configs/vexpress_defconfig
index 50eb58f6..d0e661b6 100644
--- a/arch/arm/configs/vexpress_defconfig
+++ b/arch/arm/configs/vexpress_defconfig
@@ -2468,8 +2468,9 @@ CONFIG_PANIC_TIMEOUT=0
 # CONFIG_DEBUG_MUTEXES is not set
 # CONFIG_DEBUG_WW_MUTEX_SLOWPATH is not set
 # CONFIG_DEBUG_LOCK_ALLLOC is not set
-# CONFIG_PROVE_LOCKING is not set
-# CONFIG_LOCK_STAT is not set
+CONFIG_PROVE_LOCKING=y
+CONFIG_LOCK_STAT=y
+CONFIG_DEBUG_LOCKDEP=y
 # CONFIG_DEBUG_ATOMIC_SLEEP is not set
```

重新编译内核以及本实验的内核模块。

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
make vexpress_defconfig
make -j4
```

另外一个重要地方是内核模块需要重新编译和拷贝到 kmodules 目录。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab1
#make
```

重新启动 QEMU 虚拟机。

```
# ./run.sh arm32
```

进入 mnt 目录，加载内核模块。

```
/mnt # insmod spinlock-nest.ko
/mnt # [ 128.165900]
[ 128.167414] =====
[ 128.167626] [ INFO: possible recursive locking detected ]
[ 128.168089] 4.0.0+ #4 Tainted: G          O
[ 128.168268] =====
[ 128.168462] lockdep_test/780 is trying to acquire lock:
[ 128.168815]  (hack_spinA){+.+.+}, at: [<bf0000fc>] nest_lock+0xfc/0x164
[spinlock_nest]
```

9.1 实验 1：自旋锁

```
[ 128.169968]
[ 128.169968] but task is already holding lock:
[ 128.170196]  (hack_spinA){+.+.+}, at: [<bf000028>] nest_lock+0x28/0x164
[spinlock_nest]
[ 128.170607]
[ 128.170607] other info that might help us debug this:
[ 128.170871] Possible unsafe locking scenario:
[ 128.170871]
[ 128.171077]     CPU0
[ 128.171181]     -----
[ 128.171285]         lock(hack_spinA);
[ 128.171464]         lock(hack_spinA);
[ 128.171640]
[ 128.171640] *** DEADLOCK ***
[ 128.171640]
[ 128.171854] May be due to missing lock nesting notation
[ 128.171854]
[ 128.172244] 1 lock held by lockdep_test/780:
[ 128.172483] #0: (hack_spinA){+.+.+}, at: [<bf000028>]
nest_lock+0x28/0x164 [spinlock_nest]
[ 128.173088]
[ 128.173088] stack backtrace:
[ 128.173935] CPU: 0 PID: 780 Comm: lockdep_test Tainted: G      O
4.0.0+ #4
[ 128.174561] Hardware name: ARM-Versatile Express
[ 128.175810] [<c002496c>] (unwind_backtrace) from [<c001d810>]
(show_stack+0x2c/0x38)
[ 128.176407] [<c001d810>] (show_stack) from [<c05bb318>]
(__dump_stack+0x1c/0x24)
[ 128.176830] [<c05bb318>] (__dump_stack) from [<c05bb3f0>]
(dump_stack+0xd0/0xf8)
[ 128.177200] [<c05bb3f0>] (dump_stack) from [<c00c2e5c>]
(print_deadlock_bug+0x11c/0x12c)
[ 128.177603] [<c00c2e5c>] (print_deadlock_bug) from [<c00c3110>]
(check_deadlock+0x2a4/0x2d0)
[ 128.178024] [<c00c3110>] (check_deadlock) from [<c00c51d4>]
(validate_chain+0xbc0/0xe58)
[ 128.178430] [<c00c51d4>] (validate_chain) from [<c00cb798>]
(__lock_acquire+0x17c4/0x1908)
[ 128.178831] [<c00cb798>] (__lock_acquire) from [<c00ce2fc>]
(lock_acquire+0x1fc/0x23c)
[ 128.179258] [<c00ce2fc>] (lock_acquire) from [<c0b3aba0>]
(__raw_spin_lock+0x48/0xa0)
[ 128.179671] [<c0b3aba0>] (__raw_spin_lock) from [<bf0000fc>]
(nest_lock+0xfc/0x164 [spinlock_nest])
[ 128.180129] [<bf0000fc>] (nest_lock [spinlock_nest]) from [<bf0001a0>]
(lockdep_thread+0x3c/0x64 [spinlock_nest])
[ 128.180651] [<bf0001a0>] (lockdep_thread [spinlock_nest]) from
[<c0080bc8>] (kthread+0x240/0x24c)
[ 128.181078] [<c0080bc8>] (kthread) from [<c0014e90>]
(ret_from_fork+0x14/0x24)
```

从内核日志可以看到，lockdep 已经很清晰地显示了死锁发生的路径和发生时的函数调用的栈信息，开发者根据这些信息可以很快速地定位问题和解决问题。

5. 实验代码

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/kthread.h>
```

```
5 #include <linux/freezer.h>
6 #include <linux/delay.h>
7
8 static DEFINE_SPINLOCK(hack_spinA);
9 static struct page *page;
10static struct task_struct *lock_thread;
11
12static int nest_lock(void)
13{
14    int order = 5;
15
16    spin_lock(&hack_spinA);
17    page = alloc_pages(GFP_KERNEL, order);
18    if (!page) {
19        printk("cannot alloc pages\n");
20        return -ENOMEM;
21    }
22
23    spin_lock(&hack_spinA);
24    msleep(10);
25    __free_pages(page, order);
26    spin_unlock(&hack_spinA);
27    spin_unlock(&hack_spinA);
28
29    return 0;
30}
31
32static int lockdep_thread(void *nothing)
33{
34    set_freezable();
35    set_user_nice(current, 0);
36
37    while (!kthread_should_stop()) {
38        msleep(10);
39        nest_lock();
40    }
41}
42
43static int __init my_init(void)
44{
45
46    lock_thread = kthread_run(lockdep_thread, NULL, "lockdep_test");
47    if (IS_ERR(lock_thread)) {
48        printk("create kthread fail\n");
49        return PTR_ERR(lock_thread);
50    }
51
52    return 0;
53}
54
55static void __exit my_exit(void)
56{
57    kthread_stop(lock_thread);
58}
59
60MODULE_LICENSE("GPL");
61module_init(my_init);
62module_exit(my_exit);
```

9.2 实验 2：互斥锁

9.2 实验 2：互斥锁

1. 实验目的

了解和熟悉互斥锁的使用。

2. 实验要求

在第 5 章的虚拟 FIFO 设备中，我们并没有考虑多个进程同时访问设备驱动的情况，请使用互斥锁对虚拟 FIFO 设备驱动程序进行并发保护。

我们首先要思考在这个虚拟 FIFO 设备驱动中有哪些资源是共享资源或者临界资源的。

3. 实验步骤

进入本实验参考代码。

```
#cd  
/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2  
  
#export ARCH=arm  
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rnk@figo-OptiPlex-9020:lab2$ make  
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0/ M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2 modules;  
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'  
CC [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2/mydemodrv_fasync.o  
LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2/mydemo_fasync.o  
Building modules, stage 2.  
MODPOST 1 modules  
CC [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2/mydemo_fasync.mod.o  
LD [M] /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_9/lab2/mydemo_fasync.ko  
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
```

拷贝内核模块到 kmodules 目录。

```
# cp mydemo_fasync.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

编译 test 测试程序并拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc test.c -o test -static  
  
# cp mydemo_fasync.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0  
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod mydemo_fasync.ko
```

```
/ # cd /mnt/
/mnt # insmod mydemo_fasync.ko
[ 82.754338] my_class mydemo:252:0: create device: 252:0
[ 82.756956] mydemo_fifo=c41daca4
[ 82.767580] my_class mydemo:252:1: create device: 252:1
[ 82.768403] mydemo_fifo=c41d93e4
[ 82.793580] my_class mydemo:252:2: create device: 252:2
[ 82.794247] mydemo_fifo=c41da9e4
[ 82.809173] my_class mydemo:252:3: create device: 252:3
[ 82.809805] mydemo_fifo=c41d96a4
[ 82.822762] my_class mydemo:252:4: create device: 252:4
[ 82.823351] mydemo_fifo=c41da724
[ 82.834984] my_class mydemo:252:5: create device: 252:5
[ 82.836102] mydemo_fifo=c41d9964
[ 82.847778] my_class mydemo:252:6: create device: 252:6
[ 82.848621] mydemo_fifo=c41da464
[ 82.912279] my_class mydemo:252:7: create device: 252:7
[ 82.918685] mydemo_fifo=c41d9c24
[ 82.919033] succeeded register char device: mydemo_dev
/mnt #
```

你会看到创建了 8 个设备。你可以到/sys/class/my_class/目录下面看到这些设备。

```
/mnt # cd /sys/class/my_class/
/sys/class/my_class # ls
mydemo:252:0  mydemo:252:2  mydemo:252:4  mydemo:252:6
mydemo:252:1  mydemo:252:3  mydemo:252:5  mydemo:252:7
/sys/class/my_class #
```

我们可以看到创建了主设备号为 252 的设备。我们再来看一下/dev/目录。

```
/sys/class/my_class # ls -l /dev
total 0
crw-rw--- 1 0 0 14, 4 Feb 1 09:46 audio
crw-rw--- 1 0 0 5, 1 Feb 1 09:46 console
crw-rw--- 1 0 0 10, 63 Feb 1 09:46 cpu_dma_latency
crw-rw--- 1 0 0 14, 3 Feb 1 09:46 dsp
crw-rw--- 1 0 0 29, 0 Feb 1 09:46 fb0
crw-rw--- 1 0 0 29, 1 Feb 1 09:46 fb1
crw-rw--- 1 0 0 1, 7 Feb 1 09:46 full
crw-rw--- 1 0 0 10, 183 Feb 1 09:46 hwrng
drwxr-xr-x 2 0 0 120 Feb 1 09:46 input
crw-rw--- 1 0 0 1, 2 Feb 1 09:46 kmem
crw-rw--- 1 0 0 1, 11 Feb 1 09:46 kmsg
crw-rw--- 1 0 0 1, 1 Feb 1 09:46 mem
crw-rw--- 1 0 0 10, 60 Feb 1 09:46 memory_bandwidth
crw-rw--- 1 0 0 14, 0 Feb 1 09:46 mixer
crw-rw--- 1 0 0 90, 0 Feb 1 09:46 mtd0
```

发现并没有主设备为 252 的设备。

所以我们需要手工创建一个设备用来 test app。

```
#mknod /dev/mydemo0 c 252 1
```

接下来跑我们的 test 程序：

9.2 实验 2：互斥锁

```
# ./test & #这里让 test 程序在后台跑
```

```
/mnt # ./test &
/mnt # my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
my_class mydemo:252:1: demodrv_fasync send SIGIO
```

然后使用 echo 命令来往/dev/mydemo0 这个设备写入字符串。

```
/mnt # echo "i am study linux now" > /dev/mydemo0
my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
demodrv_write kill fasync
my_class mydemo:252:1: demodrv_write:mydemo_dev1 pid=700, actual_write =21, ppos=0, ret=0
FIFO is not empty
my_class mydemo:252:1: demodrv_read:mydemo_dev1, pid=772, actual_readed=21, pos=0
i am study linux now
```

可以看到从 demodrv_read() 函数把刚才写入的字符串已经读到用户空间了。

4. 实验代码

首先我们需要在 struct mydemo_device 数据结构中添加一个 mutex 锁。

```
struct mydemo_device {
    char name[64];
    struct device *dev;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
    struct kfifo mydemo_fifo;
    struct fasync_struct *fasync;
    struct mutex lock;
};
```

struct mydemo_device 数据结构用来描述每一个设备，我们这个驱动中最多可以支持 8 个设备。因此这个 mutex 锁是每个设备一个锁。在使用 mutex 锁之前需要初始化。

```
static int __init simple_char_init(void)
{
    ...
    for (i = 0; i < MYDEMO_MAX_DEVICES; i++) {
        device = kzalloc(sizeof(struct mydemo_device), GFP_KERNEL);
        if (!device)
            ret = -ENOMEM;
        goto free_device;
    }

    sprintf(device->name, "%s%d", DEMO_NAME, i);
    mutex_init(&device->lock);

    device->dev = device_create(mydemo_class, NULL, MKDEV(dev, i),
        NULL, "mydemo:%d:%d", MAJOR(dev), i);
    dev_info(device->dev, "create device: %d:%d\n", MAJOR(dev),
        MINOR(i));
    mydemo_device[i] = device;
    init_waitqueue_head(&device->read_queue);
    init_waitqueue_head(&device->write_queue);
```

```

    ret = kfifo_alloc(&device->mydemo_fifo,
                      MYDEMO_FIFO_SIZE,
                      GFP_KERNEL);
    if (ret) {
        ret = -ENOMEM;
        goto free_kfifo;
    }
    printk("mydemo_fifo=%p\n", &device->mydemo_fifo);
}
...
}

```

在 simple_char_init() 函数中，为每一个设备都调用 mutex_init 函数对锁进行初始化。

读者需要思考，究竟在什么地方需要使用 mutex 锁进行保护。
我们以 demodrv_read 函数为例。

```

static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;
    int actual_readed;
    int ret;

    if (kfifo_is_empty(&device->mydemo_fifo)) {
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;

        dev_info(device->dev, "%s:%s pid=%d, going to sleep, %s\n",
        __func__, device->name, current->pid, data->name);
        ret = wait_event_interruptible(device->read_queue,
                                       !kfifo_is_empty(&device->mydemo_fifo));
        if (ret)
            return ret;
    }

    mutex_lock(&device->lock);
    ret = kfifo_to_user(&device->mydemo_fifo, buf, count, &actual_readed);
    if (ret)
        return -EIO;
    mutex_unlock(&device->lock);

    if (!kfifo_is_full(&device->mydemo_fifo)){
        wake_up_interruptible(&device->write_queue);
        kill_fasync(&device->fasync, SIGIO, POLL_OUT);
    }

    dev_info(device->dev, "%s:%s, pid=%d, actual_readed=%d,
pos=%lld\n", __func__,
           device->name, current->pid, actual_readed, *ppos);
    return actual_readed;
}

```

9.3 实验 3 : RCU

什么地方需要加锁保护？我们需要考虑什么地方有可能是临界区，即有可能有其他进程或者内核代码路径同时进入该区域，并对数据进行改写或者破坏。我们认为对设备的 FIFO 进行读写操作时需要进行保护。

9.3 实验 3: RCU

1. 实验目的

了解和熟悉 RCU 锁的使用。

2. 实验要求

编写一个简单的内核模块，创建一个读者内核线程和一个写者内核线程来模拟同步访问共享变量的情景。

3. 实验步骤

进入本实验参考代码。

```
#cd  
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab3  
  
#export ARCH=arm  
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@figo-OptiPlex-9020:lab3$ make  
make -C /home/r1k/r1k_basic/runninglinuxkernel_4.0/ SUBDIRS=/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_b  
asic/chapter_9/lab3 modules;  
make[1]: Entering directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'  
  CC [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab3/rcu_test.o  
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab3/rcu.o  
Building modules, stage 2.  
MODPOST 1 modules  
  CC      /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab3/rcu.mod.o  
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_9/lab3/rcu.ko  
make[1]: Leaving directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'  
rlk@figo-OptiPlex-9020:lab3$
```

拷贝内核模块到 kmodues 目录。

```
# cp rcu.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0  
#.run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod rcu.ko

/mnt # insmod rcu.ko
figo: my module init
/mnt # myrcu_reader_thread1: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread1: read a=0
myrcu_reader_thread2: write to new 5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_del: a=0
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_writer_thread: write to new 6
myrcu_reader_thread2: read a=6
myrcu_reader_thread1: read a=6
myrcu_del: a=5
```

可以看到：

该例子的目的是通过 RCU 机制保护 my_test_init() 分配的共享数据结构 g_ptr，另外创建了一个读者线程和一个写者线程来模拟同步场景。

对于读者线程 myrcu_reader_thread：

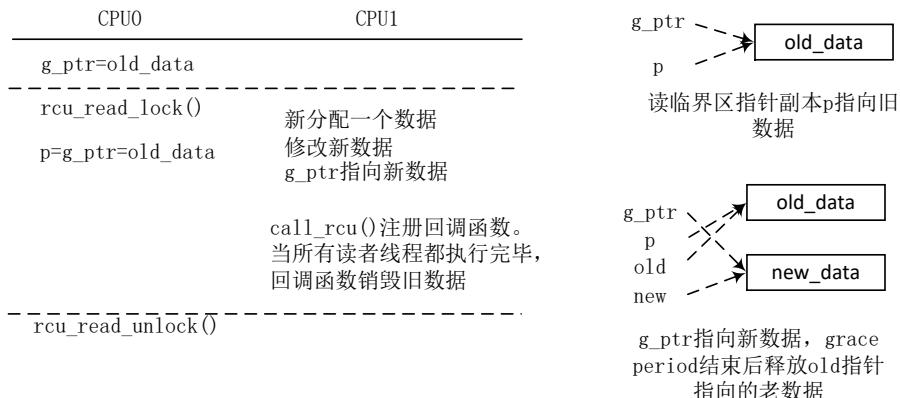
- 通过 rcu_read_lock() 和 rcu_read_unlock() 来构建一个读者临界区。
- 调用 rcu_dereference() 获取被保护数据 g_ptr 指针的一个副本，即指针 p，这时 p 和 g_ptr 都指向旧的被保护数据。
- 读者线程每隔一段时间读取一次被保护数据。

对于写者线程 myrcu_writer_thread：

- 分配一个新的保护数据 new_ptr，并修改相应数据。
- rcu_assign_pointer() 让 g_ptr 指向新数据。
- call_rcu() 注册一个回调函数，确保所有对旧数据的引用都执行完成之后，才调用回调函数来删除旧数据 old_data。
- 写者线程每隔一段时间修改被保护数据。

上述过程如图所示。

9.3 实验 3 : RCU



RCU时序图

在所有的读访问完成之后，内核可以释放旧数据，对于何时释放旧数据，内核提供了两个 API 函数：`synchronize_rcu()`和`call_rcu()`。

4. 实验代码

```

1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/init.h>
4 #include <linux/slab.h>
5 #include <linux/spinlock.h>
6 #include <linux/rcupdate.h>
7 #include <linux/kthread.h>
8 #include <linux/delay.h>
9
10 struct foo {
11     int a;
12     struct rcu_head rcu;
13 };
14
15 static struct foo *g_ptr;
16
17 static int myrcu_reader_thread1(void *data) //读者线程1
18 {
19     struct foo *p1 = NULL;
20
21     while (1) {
22         if(kthread_should_stop())
23             break;
24         msleep(20);
25         rCU_read_lock();
26         mDelay(200);
27         p1 = rcu_dereference(g_ptr);
28         if (p1)
29             printk("%s: read a=%d\n", __func__, p1->a);
30         rCU_read_unlock();
31     }
32
33     return 0;
34 }
35

```

奔跑吧 linux 社区出品

```

36 static int myrcu_reader_thread2(void *data) //读者线程2
37 {
38     struct foo *p2 = NULL;
39
40     while (1) {
41         if(kthread_should_stop())
42             break;
43         msleep(30);
44         rcu_read_lock();
45         mdelay(100);
46         p2 = rcu_dereference(g_ptr);
47         if (p2)
48             printk("%s: read a=%d\n", __func__, p2->a);
49
50         rcu_read_unlock();
51     }
52
53     return 0;
54 }
55
56 static void myrcu_del(struct rcu_head *rh)
57 {
58     struct foo *p = container_of(rh, struct foo, rCU);
59     printk("%s: a=%d\n", __func__, p->a);
60     kfree(p);
61 }
62
63 static int myrcu_writer_thread(void *p) //写者线程
64 {
65     struct foo *old;
66     struct foo *new_ptr;
67     int value = (unsigned long)p;
68
69     while (1) {
70         if(kthread_should_stop())
71             break;
72         msleep(250);
73         new_ptr = kmalloc(sizeof (struct foo), GFP_KERNEL);
74         old = g_ptr;
75         *new_ptr = *old;
76         new_ptr->a = value;
77         rCU_assign_pointer(g_ptr, new_ptr);
78         call_rcu(&old->rcu, myrcu_del);
79         printk("%s: write to new %d\n", __func__, value);
80         value++;
81     }
82
83     return 0;
84 }
85
86 static struct task_struct *reader_thread1;
87 static struct task_struct *reader_thread2;
88 static struct task_struct *writer_thread;
89
90 static int __init my_test_init(void)
91 {
92     int value = 5;
93
94     printk("figo: my module init\n");
95     g_ptr = kzalloc(sizeof (struct foo), GFP_KERNEL);
96
97     reader_thread1 = kthread_run(myrcu_reader_thread1, NULL,
"rcu_reader1");

```

9.3 实验 3 : RCU

```
98     reader_thread2 = kthread_run(myrcu_reader_thread2, NULL,
"rcu_reader2");
99     writer_thread = kthread_run(myrcu_writer_thread, (void *) (unsigned
long) value, "rcu_writer");
100    return 0;
101}
102}
103static void __exit my_test_exit(void)
104{
105    printk("goodbye\n");
106    kthread_stop(reader_thread1);
107    kthread_stop(reader_thread2);
108    kthread_stop(writer_thread);
109    if (g_ptr)
110        kfree(g_ptr);
111}
112MODULE_LICENSE("GPL");
113module_init(my_test_init);
114module_exit(my_test_exit);
```

Linux入门、看奔跑吧Linux内核
微信公众号：runninglinuxkernel

第 10 章

中断管理

10.1 实验 1: tasklet

1. 实验目的

了解和熟悉 Linux 内核的 tasklet 机制的使用。

2. 实验要求

- 1) 写一个简单的内核模块，初始化一个 tasklet，在 write()函数里调用该 tasklet 回调函数，在 tasklet 回调函数中输出用户程序写入的字符串。
- 2) 写一个应用程序，测试该功能。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_
tasklet

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlik@figo-OptiPlex-9020:~/lab1_tasklet$ make
make -C /home/rlik/rlik_basic/runninglinuxkernel_4.0/ M=/home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_tasklet modules;
make[1]: Entering directory '/home/rlik/rlik_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_tasklet/mydemodrv_tasklet.o
  LD [M]  /home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_tasklet/mydemo_tasklet.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_tasklet/mydemo_tasklet.mod.o
  LD [M]  /home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_10/lab1_tasklet/mydemo_tasklet.ko
make[1]: Leaving directory '/home/rlik/rlik_basic/runninglinuxkernel_4.0'
rlik@figo-OptiPlex-9020:~/lab1_tasklet$
```

拷贝内核模块到 kmodules 目录。

```
# cp mydemo_tasklet.ko /home/rlik/rlik_basic/runninglinuxkernel_4.0/kmodules/
```

编译 test 测试程序并拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc test.c -o test --static
```

```
# cp test /home/rlik/rlik_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rlik/rlik_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod mydemo_tasklet.ko
```

10.1 实验 1 : tasklet

```
/mnt # insmod mydemo_tasklet.ko
my_class mydemo:252:0: create device: 252:0
mydemo_fifo=ee97259c
my_class mydemo:252:1: create device: 252:1
mydemo_fifo=ee97205c
my_class mydemo:252:2: create device: 252:2
mydemo_fifo=ee97235c
my_class mydemo:252:3: create device: 252:3
mydemo_fifo=ee97265c
my_class mydemo:252:4: create device: 252:4
mydemo_fifo=ee97211c
my_class mydemo:252:5: create device: 252:5
mydemo_fifo=ee97271c
my_class mydemo:252:6: create device: 252:6
mydemo_fifo=ee97241c
my_class mydemo:252:7: create device: 252:7
mydemo_fifo=ee9724dc
succeeded register char device: mydemo_dev
/mnt #
```

你会看到创建了 8 个设备。你可以到 /sys/class/my_class/ 目录下面看到这些设备。

```
/mnt # cd /sys/class/my_class/
/sys/class/my_class # ls
mydemo:252:0 mydemo:252:2 mydemo:252:4 mydemo:252:6
mydemo:252:1 mydemo:252:3 mydemo:252:5 mydemo:252:7
/sys/class/my_class #
```

我们可以看到创建了主设备号为 252 的设备。我们再来看一下 /dev/ 目录。

```
/sys/class/my_class # ls -l /dev
total 0
crw-rw---- 1 0 0 14, 4 Feb 1 09:46 audio
crw-rw---- 1 0 0 5, 1 Feb 1 09:46 console
crw-rw---- 1 0 0 10, 63 Feb 1 09:46 cpu_dma_latency
crw-rw---- 1 0 0 14, 3 Feb 1 09:46 dsp
crw-rw---- 1 0 0 29, 0 Feb 1 09:46 fb0
crw-rw---- 1 0 0 29, 1 Feb 1 09:46 fb1
crw-rw---- 1 0 0 1, 7 Feb 1 09:46 full
crw-rw---- 1 0 0 10, 183 Feb 1 09:46 hwrng
drwxr-xr-x 2 0 0 120 Feb 1 09:46 input
crw-rw---- 1 0 0 1, 2 Feb 1 09:46 kmem
crw-rw---- 1 0 0 1, 11 Feb 1 09:46 kms
crw-rw---- 1 0 0 1, 1 Feb 1 09:46 mem
crw-rw---- 1 0 0 10, 60 Feb 1 09:46 memory_bandwidth
crw-rw---- 1 0 0 14, 0 Feb 1 09:46 mixer
crw-rw---- 1 0 0 90, 0 Feb 1 09:46 mtd0
```

发现并没有主设备为 252 的设备。

所以我们需要手工创建一个设备用来 test app。

```
#mknod /dev/mydemo0 c 252 0
```

接下来跑我们的 test 程序：

```
# ./test & #这里让test程序在后台跑
```

```
/mnt # ./test &
/mnt # my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
my_class mydemo:252:1: demodrv_fasync send SIGIO
```

然后使用 echo 命令来往 /dev/mydemo0 这个设备写入字符串。

```
/mnt # /mnt # echo "i am learning runninglinuxkernel" > /dev/mydemo0
my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
demodrv_write kill fasync
my_class mydemo:252:1: demodrv_write:mydemo_dev1 pid=702, actual_write =33, ppos=0, ret=0
FIFO is not empty
my_class mydemo:252:1: demodrv_read:mydemo_dev1, pid=768, actual_readed=33, pos=0
my_class mydemo:252:1: do_tasklet: trigger a tasklet
i am learning runninglinuxkernel
```

可以看到从 tasklet 的回调函数打印的一句话 “do_tasklet: trigger a tasklet”。

4. 实验代码分析

首先在每个设备的私有数据 struct mydemo_private_data 中添加一个 tasklet。

```
struct mydemo_private_data {
    struct mydemo_device *device;
    char name[64];
    struct tasklet_struct tasklet;
};
```

tasklet 使用 struct tasklet_struct 来表示。在使用 tasklet 之前需要初始化，使用 tasklet_init() 函数进行初始化。我们选择在 demodrv_open() 时初始化 tasklet。

```
static int demodrv_open(struct inode *inode, struct file *file)
{
    unsigned int minor = iminor(inode);
    struct mydemo_private_data *data;
    struct mydemo_device *device = mydemo_device[minor];

    dev_info(device->dev, "%s: major=%d, minor=%d, device=%s\n", __func__,
             MAJOR(inode->i_rdev), MINOR(inode->i_rdev),
             device->name);

    data = kmalloc(sizeof(struct mydemo_private_data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    sprintf(data->name, "private_data %d", minor);
    tasklet_init(&data->tasklet, do_tasklet, (unsigned long)device);

    data->device = device;
    file->private_data = data;

    return 0;
}
```

驱动关闭时需要调用 tasklet_kill() 来关闭 tasklet。

```
static int demodrv_release(struct inode *inode, struct file *file)
{
    struct mydemo_private_data *data = file->private_data;

    tasklet_kill(&data->tasklet);
    kfree(data);

    return 0;
}
```

那什么时候去触发 tasklet 呢？

10.2 实验 2 : 工作队列

我们在这个例子选择在 read 函数里触发。

```
static ssize_t
demodrv_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    ...
    mutex_lock(&device->lock);
    ret = kfifo_to_user(&device->mydemo_fifo, buf, count, &actual_readed);
    if (ret)
        return -EIO;
    tasklet_schedule(&data->tasklet);
    mutex_unlock(&device->lock);
    ...

    return actual_readed;
}
```

在 demodrv_read 函数中，当 FIFO 有数据可读时，我们调用 tasklet_schedule() 来触发一个 tasklet。

tasklet 的回调函数需要驱动开发人员来实现。我们在这个例子中，仅仅是添加一句打印。

```
static void do_tasklet(unsigned long data)
{
    struct mydemo_device *device = (struct mydemo_device *)data;
    dev_info(device->dev, "%s: trigger a tasklet\n", __func__);
}
```

10.2 实验 2：工作队列

1. 实验目的

通过本实验了解和熟悉 Linux 内核的工作队列机制的使用。

2. 实验要求

1) 写一个简单的内核模块，初始化一个工作队列，在 write() 函数里调用该工作队列回调函数，在回调函数中输出用户程序写入的字符串。

2) 写一个应用程序，测试该功能。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab2

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@ubuntu:lab2_workqueue$ make
make -C /home/ralk/ralk_basic/runninglinuxkernel_4.0 M=/home/ralk/ralk_basic/running
linuxkernel_4.0/rlk_lab/rlk_basic/chapter_10/lab2_workqueue modules;
make[1]: Entering directory '/home/ralk/ralk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/ralk/ralk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_1
0/Lab2_workqueue/mydemodrv_work.o
  LD [M]  /home/ralk/ralk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_1
0/Lab2_workqueue/mydemo_work.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/ralk/ralk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_1
0/Lab2_workqueue/mydemo_work.mod.o
  LD [M]  /home/ralk/ralk_basic/runninglinuxkernel_4.0/rlk_lab/rlk_basic/chapter_1
0/Lab2_workqueue/mydemo_work.ko
make[1]: Leaving directory '/home/ralk/ralk_basic/runninglinuxkernel_4.0'
```

拷贝内核模块到 kmodules 目录。

```
# cp mydemo_work.ko /home/ralk/ralk_basic/runninglinuxkernel_4.0/kmodules/
```

编译 test 测试程序并拷贝到 kmodules 目录。

```
#arm-linux-gnueabi-gcc test.c -o test -static
# cp test /home/ralk/ralk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/ralk/ralk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod mydemo_work.ko
/mnt # insmod mydemo_work.ko
[ 126.021608] my_class mydemo:252:0: create device: 252:0
[ 126.025414] mydemo_fifo=c4824adc
[ 126.038594] my_class mydemo:252:1: create device: 252:1
[ 126.041684] mydemo_fifo=c48253dc
[ 126.075785] my_class mydemo:252:2: create device: 252:2
[ 126.077765] mydemo_fifo=c4824c5c
[ 126.106613] my_class mydemo:252:3: create device: 252:3
[ 126.108985] mydemo_fifo=c482525c
[ 126.135330] my_class mydemo:252:4: create device: 252:4
[ 126.136216] mydemo_fifo=c4824ddc
[ 126.153225] my_class mydemo:252:5: create device: 252:5
[ 126.154066] mydemo_fifo=c48250dc
[ 126.172367] my_class mydemo:252:6: create device: 252:6
[ 126.177276] mydemo_fifo=c4824f5c
[ 126.198959] my_class mydemo:252:7: create device: 252:7
[ 126.199949] mydemo_fifo=c479605c
[ 126.200536] succeeded register char device: mydemo_dev
/mnt #
```

你会看到创建了 8 个设备。你可以到/sys/class/my_class/目录下面看到这些设备。

```
/mnt # cd /sys/class/my_class/
/sys/class/my_class # ls
mydemo:252:0  mydemo:252:2  mydemo:252:4  mydemo:252:6
mydemo:252:1  mydemo:252:3  mydemo:252:5  mydemo:252:7
/sys/class/my_class #
```

10.2 实验 2：工作队列

我们可以看到创建了主设备号为 252 的设备。我们再来看一下 /dev/ 目录。

```
/sys/class/my_class # ls -l /dev
total 0
crw-rw---- 1 0 0 14, 4 Feb 1 09:46 audio
crw-rw---- 1 0 0 5, 1 Feb 1 09:46 console
crw-rw---- 1 0 0 10, 63 Feb 1 09:46 cpu_dma_latency
crw-rw---- 1 0 0 14, 3 Feb 1 09:46 dsp
crw-rw---- 1 0 0 29, 0 Feb 1 09:46 fb0
crw-rw---- 1 0 0 29, 1 Feb 1 09:46 fb1
crw-rw---- 1 0 0 1, 7 Feb 1 09:46 full
crw-rw---- 1 0 0 10, 183 Feb 1 09:46 hwrng
drwxr-xr-x 2 0 0 120 Feb 1 09:46 input
crw-rw---- 1 0 0 1, 2 Feb 1 09:46 kmem
crw-rw---- 1 0 0 1, 11 Feb 1 09:46 kmsg
crw-rw---- 1 0 0 1, 1 Feb 1 09:46 mem
crw-rw---- 1 0 0 10, 60 Feb 1 09:46 memory_bandwidth
crw-rw---- 1 0 0 14, 0 Feb 1 09:46 mixer
crw-rw---- 1 0 0 90, 0 Feb 1 09:46 mtd0
```

发现并没有主设备为 252 的设备。

所以我们需要手工创建一个设备用来 test app。

```
#mknod /dev/mydemo0 c 252 1
```

接下来跑我们的 test 程序：

```
# ./test & #这里让 test 程序在后台跑
```

```
/mnt # ./test &
/mnt # echo "i am learning runninglinuxkernel" > /dev/mydemo0
my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
my_class mydemo:252:1: demodrv_fasync send SIGIO
```

然后使用 echo 命令来往 /dev/mydemo0 这个设备写入字符串。

```
/mnt #
/mnt # echo "i am learning runninglinuxkernel" > /dev/mydemo0
my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
demodrv_write kill fasync
my_class mydemo:252:1: demodrv_write:mydemo_dev1 pid=703, actual_write =33, ppos=0, ret=0
my_class mydemo:252:1: do_work: trigger a work
FIFO is not empty
my_class mydemo:252:1: demodrv_read:mydemo_dev1, pid=741, actual_readed=33, pos=0
i am learning runninglinuxkernel

FIFO is not full
```

可以看到从 workqueue 的回调函数打印的一句话 “do_work: trigger a work”。

4. 实验代码分析

本实验参考代码和实验 1 类似，只不过把 tasklet 机制换成了工作队列。

首先在每个设备的私有数据 struct mydemo_private_data 中添加一个工作队列。

```
struct mydemo_private_data {
    struct mydemo_device *device;
    char name[64];
    struct tasklet_struct tasklet;
```

```

    struct work_struct my_work;
};

工作队列使用 struct work_struct 来表示。在使用工作队列之前需要初始化，使用
INIT_WORK()函数进行初始化。我们选择在 demodrv_open()时初始化 tasklet。
static int demodrv_open(struct inode *inode, struct file *file)
{
    unsigned int minor = iminor(inode);
    struct mydemo_private_data *data;
    struct mydemo_device *device = mydemo_device[minor];

    dev_info(device->dev, "%s: major=%d, minor=%d, device=%s\n", __func__,
             MAJOR(inode->i_rdev), MINOR(inode->i_rdev),
             device->name);

    data = kzalloc(sizeof(struct mydemo_private_data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    sprintf(data->name, "private_data_%d", minor);
    tasklet_init(&data->tasklet, do_tasklet, (unsigned long)device);
INIT_WORK(&data->my_work, do_work);

    data->device = device;
    file->private_data = data;

    return 0;
}

```

那什么时候去触发工作队列呢？

我们在这个例子选择在 write 函数里触发。

```

static ssize_t
demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t
*ppos)
{
    struct mydemo_private_data *data = file->private_data;
    struct mydemo_device *device = data->device;

    ...

    mutex_lock(&device->lock);
    ret = kfifo_from_user(&device->mydemo_fifo, buf, count,
    &actual_write);
    if (ret)
        return -EIO;
schedule_work(&data->my_work);
    mutex_unlock(&device->lock);
    ...

    return actual_write;
}

```

在 demodrv_write 函数中，当 FIFO 可写时，我们调用 schedule_work()来触发一个工作。

worker 的回调函数需要驱动开发人员来实现。我们在这个例子中，仅仅是添加一句打印。

```

static void do_work(struct work_struct *work)
{
    struct mydemo_private_data *data;
    struct mydemo_device *device;

```

10.3 实验 3：定时器和内核线程

```

data = container_of(work, struct mydemo_private_data, my_work);
device = data->device;
dev_info(device->dev, "%s: trigger a work\n", __func__);
}

```

10.3 实验 3：定时器和内核线程

1. 实验目的

通过本实验了解和熟悉 Linux 内核的定时器和内核线程机制的使用。

2. 实验要求

写一个简单的内核模块，首先定义一个定时器来模拟中断，再新建一个内核线程。当定时器到来时，唤醒内核线程，然后在内核线程的主程序中输出该内核线程的相关信息，如 PID、当前 jiffies 等信息。

3. 实验步骤

进入本实验参考代码。

```

#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3
#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-

```

编译内核模块。

```

rlk@figo-OptiPlex-9020:lab3$ make
make -C /home/r1k/r1k_basic/runninglinuxkernel_4.0/ M=/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3 modules;
make[1]: Entering directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'
  CC [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3/lab3_test.o
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3/lab3-test.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3/lab3-test.mod.o
  LD [M]  /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_10/lab3/lab3-test.ko
make[1]: Leaving directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'

```

拷贝内核模块到 kmodues 目录。

```
# cp lab3-test.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod lab3-test.ko
```

```
/mnt # insmod lab3-test.ko
[ 81.320842] ben: my lockdep module init
[mnt # [ 81.822809] show_reg: ktest, pid:781
[ 81.823880] cpsr:0x20000013, sp:0xc43d7ec8
[ 83.817382] show_reg: ktest, pid:781
[ 83.817613] cpsr:0x20000013, sp:0xc43d7ec8
[ 85.816430] show_reg: ktest, pid:781
[ 85.816659] cpsr:0x20000013, sp:0xc43d7ec8
[ 87.815433] show_reg: ktest, pid:781
[ 87.815861] cpsr:0x20000013, sp:0xc43d7ec8
[ 89.814351] show_reg: ktest, pid:781
[ 89.814628] cpsr:0x20000013, sp:0xc43d7ec8
[ 91.813363] show_reg: ktest, pid:781
[ 91.813603] cpsr:0x20000013, sp:0xc43d7ec8
[ 93.812360] show_reg: ktest, pid:781
[ 93.812593] cpsr:0x20000013, sp:0xc43d7ec8
[ 95.811308] show_reg: ktest, pid:781
```

4. 实验代码

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/kthread.h>
5 #include <linux/freezer.h>
6 #include <linux/mutex.h>
7 #include <linux/delay.h>
8
9 static void my_timefunc(unsigned long);
10static DEFINE_TIMER(my_timer, my_timefunc, 0, 0);
11static atomic_t flags;
12wait_queue_head_t wait_head;
13
14static void my_timefunc(unsigned long dummy)
15{
16    atomic_set(&flags, 1);
17    //printk("%s: set flags %d\n", __func__, atomic_read(&flags));
18    wake_up_interruptible(&wait_head);
19    mod_timer(&my_timer, jiffies + msecs_to_jiffies(2000));
20}
21
22static void my_try_to_sleep(void)
23{
24    DEFINE_WAIT(wait);
25
26    if (freezing(current) || kthread_should_stop())
27        return;
28
29    prepare_to_wait(&wait_head, &wait, TASK_INTERRUPTIBLE);
30
31    if (!atomic_read(&flags))
32        schedule();
33
34    finish_wait(&wait_head, &wait);
35}
36
37static void show_reg(void)
38{
39    unsigned int cpsr, sp;
```

10.3 实验 3：定时器和内核线程

```

40         struct task_struct *task = current;
41
42         asm("mrs %0, cpsr" : "=r" (cpsr) : : "cc");
43         asm("mov %0, sp" : "=r" (sp) : : "cc");
44
45         printk("%s: %s, pid:%d\n", __func__, task->comm, task->pid);
46         printk("cpsr:0x%x, sp:0x%x\n", cpsr, sp);
47}
48
49static int my_thread(void *nothing)
50{
51     set_freezable();
52     set_user_nice(current, 0);
53
54     while (!kthread_should_stop()) {
55         my_try_to_sleep();
56         atomic_set(&flags, 0);
57         show_reg();
58     }
59     return 0;
60}
61
62
63static struct task_struct *thread;
64
65static int __init my_init(void)
66{
67     printk("ben: my lockdep module init\n");
68
69     /*创建一个线程来处理某些事情*/
70     thread = kthread_run(my_thread, NULL, "ktest");
71
72     /*创建一个定时器来模拟某些异步事件, 比如中断等*/
73     my_timer.expires = jiffies + msecs_to_jiffies(500);
74     add_timer(&my_timer);
75
76     init_waitqueue_head(&wait_head);
77
78     return 0;
79}
80
81static void __exit my_exit(void)
82{
83     printk("goodbye\n");
84
85     kthread_stop(thread);
86}
87MODULE_LICENSE("GPL");
88module_init(my_init);
89module_exit(my_exit);

```

第 11 章

调试和性能优化

11.1 实验 1: printk

1. 实验目的

了解如何使用内核的 printk 函数进行输出调试。

2. 实验步骤

- 1) 编写一个简单的内核模块，使用 printk 函数来进行输出。
- 2) 在内核中选择一个驱动程序或者内核代码，使用 printk 函数进行输出调试。

11.2 实验 2：动态输出

1. 实验目的

通过本实验学会使用动态输出的方式来辅助调试。

2. 实验要求

- 1) 选择一个你熟悉的内核模块或者驱动模块打开动态输出功能来观察日志信息。
- 2) 编写一个简单的内核模块，使用 pr_debug()/dev_dbg()函数来添加输出信息，并且在 QEMU 或者优麒麟上实验。

11.3 实验 3: procfs

1. 实验目的

- 1) 写一个内核模块，在/proc 中创建一个名为 “test” 的目录。
- 2) 在 test 目录下面创建两个节点，分别是 “read” 和 “write”。从 “read” 节点中可以读取内核模块的某个全局变量的值，往 “write” 节点写数据可以修改某个全局变量的值。

11.3 实验 3 : procfs

2. 实验要求

procfs 文件系统提供了一些常用的 API, 这些 API 函数定义在 fs/proc/internal.h 文件中。

`proc_mkdir()`可以在 `parent` 父目录中创建一个名字为 `name` 的目录, 如果 `parent` 指定为 `NULL`, 则在`/proc`的根目录下面创建一个目录。

```
struct proc_dir_entry *proc_mkdir(const char *name,
    struct proc_dir_entry *parent)
```

`proc_create()`函数会创建一个新的文件节点。

```
struct proc_dir_entry *proc_create(
    const char *name, umode_t mode, struct proc_dir_entry *parent,
    const struct file_operations *proc_fops)
```

其中, `name` 是该节点的名称, `mode` 是该节点的访问权限, 以 UGO 的模式来表示; `parent` 和 `proc_mkdir()`函数中的 `parent` 类型, 指向父进程的 `proc_dir_entry` 对象; `proc_fops` 指向该文件的操作函数。

比如 `misc` 驱动在初始化时就创建了一个名为“`misc`”的文件。

```
<driver/char/misc.c>

static int __init misc_init(void)
{
    int err;
#ifdef CONFIG_PROC_FS
    proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
...
}
```

`proc_fops` 会指向该文件的操作函数集, 比如 `misc` 驱动中会定义 `misc_proc_fops` 函数集, 里面有 `open`、`read`、`llseek`、`release` 等文件操作函数。

```
static const struct file_operations misc_proc_fops = {
    .owner      = THIS_MODULE,
    .open       = misc_seq_open,
    .read       = seq_read,
    .llseek     = seq_llseek,
    .release   = seq_release,
};
```

下面是读取`/proc/misc`这个文件的相关信息, 这里列出了系统中 `misc` 设备的信息。

```
/proc # cat misc
59 ubi_ctrl
60 memory_bandwidth
61 network_throughput
62 network_latency
63 cpu_dma_latency
1 psaux
183 hw_random
```

读者可以参照 Linux 内核中的例子来完成本实验。

3. 实验步骤

进入本实验参考代码。

```
#cd  
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3  
  
#export ARCH=arm  
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rlk@ubuntu:lab3_procfs$ make  
make -C /home/r1k/r1k_basic/runninglinuxkernel_4.0 M=/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3_procfs modules;  
make[1]: Entering directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'  
CC [M] /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3_procfs/proc_test.o  
LD [M] /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3_procfs/proc-test.o  
Building modules, stage 2.  
MODPOST 1 modules  
CC /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3_procfs/proc-test.mod.o  
LD [M] /home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab3_procfs/proc-test.ko  
make[1]: Leaving directory '/home/r1k/r1k_basic/runninglinuxkernel_4.0'  
rlk@ubuntu:lab3_procfs$
```

拷贝内核模块到 kmodues 目录。

```
# cp proc-test.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0  
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt  
#insmod proc-test.ko
```

```
/mnt # insmod proc-test.ko  
I created benshushu/my_proc
```

在 proc 目录下面创建了一个名为 benshushu 的新的目录，然后新建了一个 my_proc 的节点。

my_proc 节点有一个默认值。

```
/proc # cd benshushu/  
/proc/benshushu # ls  
my_proc  
/proc/benshushu # cat my_proc  
100
```

通过 echo 命令来往这个节点里写入新的值。

11.3 实验 3 : procfs

```
/proc/benshushu # echo 200 > my_proc
param has been set to 200
/proc/benshushu # cat my_proc
200
/proc/benshushu # █
```

4. 实验代码分析

```
1 #include <linux/module.h>
2 #include <linux/proc_fs.h>
3 #include <linux/uaccess.h>
4 #include <linux/init.h>
5
6 #define NODE "benshushu/my_proc"
7
8 static int param = 100;
9 static struct proc_dir_entry *my_proc;
10static struct proc_dir_entry *my_root;
11
12#define KS 32
13static char kstring[KS]; /* should be less sloppy about overflows : */
*/
14
15static ssize_t
16my_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos)
17{
18     int nbytes = sprintf(kstring, "%d\n", param);
19     return simple_read_from_buffer(buf, lbuf, ppos, kstring, nbytes);
20}
21
22static ssize_t my_write(struct file *file, const char __user *buf, size_t
lbuf,
23                         loff_t *ppos)
24{
25     ssize_t rc;
26     rc = simple_write_to_buffer(kstring, lbuf, ppos, buf, lbuf);
27     sscanf(kstring, "%d", &param);
28     pr_info("param has been set to %d\n", param);
29     return rc;
30}
31
32static const struct file_operations my_proc_fops = {
33     .owner = THIS_MODULE,
34     .read = my_read,
35     .write = my_write,
36};
37
38static int __init my_init(void)
39{
40     my_root = proc_mkdir("benshushu", NULL);
41     if (IS_ERR(my_root)) {
42         pr_err("I failed to make benshushu dir\n");
43         return -1;
44     }
45
46     my_proc = proc_create(NODE, 0, NULL, &my_proc_fops);
47     if (IS_ERR(my_proc)) {
48         pr_err("I failed to make %s\n", NODE);
49         return -1;
50     }
51     pr_info("I created %s\n", NODE);
52     return 0;
53}
```

```

54
55static void __exit my_exit(void)
56{
57    if (my_proc) {
58        proc_remove(my_proc);
59        proc_remove(my_root);
60        pr_info("Removed %s\n", NODE);
61    }
62}
63
64module_init(my_init);
65module_exit(my_exit);
66MODULE_LICENSE("GPL");

```

5. 进阶思考

创建 procfs 是内核调试或者说内核空间和用户空间进行交换的一个重要的手段。

内核里有不少全局的变量值，存放在 procfs 里面，有三个目录的节点，是值得我们去学习和研究的，特别是做运维和系统调优的朋友们。

1. /proc/sys/kernel 目录，里面存放了内核核心的调优参数
2. /proc/sys/vm 目录，里面存放了内核内存管理相关的调优参数
3. /proc/pid/目录，这里 pid 指的是具体的进程的 pid，这里面存放的是每个进程相关的调优参数。

```
/proc/sys/vm # ls
admin_reserve_kbytes      max_map_count
block_dump                 min_free_kbytes
compact_memory              mmap_min_addr
dirty_background_bytes     nr_pdflush_threads
dirty_background_ratio     oom_dump_tasks
dirty_bytes                 oom_kill Allocating_task
dirty_expire_centisecs    overcommit_kbytes
dirty_ratio                  overcommit_memory
dirty_writeback_centisecs overcommit_ratio
dirtytime_expire_seconds   page-cluster
drop_caches                  panic_on_oom
extfrag_threshold           percpu_pagelist_fraction
highmem_is_dirtyable       stat_interval
laptop_mode                  swappiness
legacy_va_layout             user_reserve_kbytes
lowmem_reserve_ratio        vfs_cache_pressure
```

11.4 实验 4: sysfs

1. 实验目的

- 1) 写一个内核模块，在/sys/目录下面创建一个名为“test”的目录。
- 2) 在 test 目录下面创建两个节点，分别是“read”和“write”。从“read”节点中可以读取内核模块的某个全局变量的值，往“write”节点写数据可以修改某个全局变量的值。

11.4 实验 4 : sysfs

2. 实验要求

下面介绍本实验会用到的一些 API 函数。

`kobject_create_and_add()` 函数会动态生成一个 `struct kobject` 数据结构，然后将其注册到 sysfs 文件系统中。其中，`name` 就是要创建的文件或者目录的名称，`parent` 指向父目录的 `kobject` 数据结构，若 `parent` 为 `NULL`，说明父目录就是 /sys 目录。

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
```

`sysfs_create_group()` 函数会在参数 1 的 `kobj` 目录下面创建一个属性集合，并且显示该集合的文件。

```
static inline int sysfs_create_group(struct kobject *kobj,
                                     const struct attribute_group *grp)
```

参数 2 中描述的是一组属性类型，其数据结构定义如下。

```
<include/linux/sysfs.h>

struct attribute_group {
    const char          *name;
    umode_t             (*is_visible)(struct kobject *,
                                         struct attribute *, int);
    struct attribute   **attrs;
    struct bin_attribute **bin_attrs;
};
```

其中，`struct attribute` 数据结构用于描述文件的属性。

下面以 /sys/kernel/ 目录下面的文件为例来说明它们是如何建立的。

```
/sys/kernel # ls -l
total 0
drwx----- 17 0      0          0 Jan  1 1970 debug
-rw-r--r--  1 0      0          4096 Apr 29 07:08 fscaps
-rw-r--r--  1 0      0          4096 Apr 29 07:08 kexec_crash_loaded
-rw-r--r--  1 0      0          4096 Apr 29 07:08 kexec_crash_size
-rw-r--r--  1 0      0          4096 Apr 29 07:08 kexec_loaded
drwxr-xr-x  2 0      0          0 Apr 29 07:08 mm
-rw-r--r--  1 0      0          36 Apr 29 07:08 notes
-rw-r--r--  1 0      0          4096 Apr 29 07:08 profiling
-rw-r--r--  1 0      0          4096 Apr 29 07:08 rcu_expedited
drwxr-xr-x  70 0     0          0 Apr 29 07:08 slab
-rw-r--r--  1 0      0          4096 Apr 29 07:08 uevent_helper
-rw-r--r--  1 0      0          4096 Apr 29 07:08 uevent_seqnum
-rw-r--r--  1 0      0          4096 Apr 29 07:08 vmcoreinfo
```

/sys/kernel 目录建立在内核源代码的 `kernel/ksysfs.c` 文件中。

```
static int __init ksysfs_init(void)
{
    kernel_kobj = kobject_create_and_add("kernel", NULL);
    ...
    error = sysfs_create_group(kernel_kobj, &kernel_attr_group);
    return 0;
}
```

这里 `kobject_create_and_add()` 在 /sys 目录下建立一个名为 “kernel” 的目录，然后

sysfs_create_group()函数在该目录下面创建一些属性集合。

```
static struct attribute * kernel_attrs[] = {
    &fscaps_attr.attr,
    &uevent_seqnum_attr.attr,
    &profiling_attr.attr,
    NULL
};

static struct attribute_group kernel_attr_group = {
    .attrs = kernel_attrs,
};
```

以 profiling 文件为例，这里实现 profiling_show()和 profiling_store()两个函数，分别对应读和写操作。

```
static ssize_t profiling_show(struct kobject *kobj,
                             struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, "%d\n", prof_on);
}
static ssize_t profiling_store(struct kobject *kobj,
                             struct kobj_attribute *attr,
                             const char *buf, size_t count)
{
    int ret;

    profile_setup((char *)buf);
    ret = profile_init();
    return count;
}
KERNEL_ATTR_RW(profiling);
```

其中 KERNEL_ATTR_RW 宏定义如下。

```
#define KERNEL_ATTR_RO(_name) \
static struct kobj_attribute _name##_attr = __ATTR_RO(_name)

#define KERNEL_ATTR_RW(_name) \
static struct kobj_attribute _name##_attr = __ATTR(_name, 0644, _name##_show, _name##_store)
```

上面是/sys/kernel 的一个例子，Linux 内核源代码里还有很多设备驱动的例子，读者可以参考这些例子来完成本实验。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/r1k/r1k_basic/runninglinuxkernel_4.0/r1k_lab/r1k_basic/chapter_11/lab4

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

11.4 实验 4 : sysfs

```
rlk@ubuntu:lab4_sysfs$ make
make -C /home/rnk/rnk_basic/runninglinuxkernel_4.0 M=/home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_11/lab4_sysfs modules;
make[1]: Entering directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_11/lab4_sysfs/sysfs-test.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_11/lab4_sysfs/sysfs-test.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_11/lab4_sysfs/sysfs-test.mod.o
  LD [M]  /home/rnk/rnk_basic/runninglinuxkernel_4.0/rnk_lab/rnk_basic/chapter_11/lab4_sysfs/sysfs-test.ko
make[1]: Leaving directory '/home/rnk/rnk_basic/runninglinuxkernel_4.0'
rnk@ubuntu:lab4_sysfs$
```

拷贝内核模块到 kmodules 目录。

```
# cp sysfs-test.ko /home/rnk/rnk_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/rnk/rnk_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod sysfs-test.ko
```

```
/mnt # insmod sysfs-test.ko
I created benshushu/my_proc on procfs
create sysfs node done
```

那这个 sysfs 目录创建到哪里呢？我们可以通过 find 命令来查找。

```
/mnt # find /sys -name "benshushu"
/sys/bus/platform/devices/benshushu
/sys/devices/platform/benshushu
/sys/devices/platform/benshushu/benshushu
/mnt #
```

那 我 们 看 到 在 /sys/bus/platform/devices/benshushu 和 /sys/devices/platform/benshushu 目录下面创建了“benshushu”的文件夹。

```
/sys/devices/platform/benshushu # cd benshushu/
/sys/devices/platform/benshushu/benshushu # ls
data
```

进入到 /sys/devices/platform/benshushu/benshushu 这个目录就可以看到 data 这个节点。

然后通过 cat 和 echo 命令来读写这个节点。

```
/sys/devices/platform/benshushu/benshushu # cat data
100
/sys/devices/platform/benshushu/benshushu # echo 200 > data
/sys/devices/platform/benshushu/benshushu #
/sys/devices/platform/benshushu/benshushu # cat data
200
```

4. 实验代码分析

```

1 #include <linux/module.h>
2 #include <linux/proc_fs.h>
3 #include <linux/uaccess.h>
4 #include <linux/init.h>
5 #include <linux/device.h>
6 #include <linux/platform_device.h>
7 #include <linux/sysfs.h>
8
9 #define NODE "benshushu/my_proc"
10
11 static int param = 100;
12 static struct proc_dir_entry *my_proc;
13 static struct proc_dir_entry *my_root;
14 static struct platform_device *my_device;
15
16 #define KS 32
17 static char kstring[KS]; /* should be less sloppy about overflows :)
*/
18
19 static ssize_t
20 my_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos)
21 {
22     int nbytes = sprintf(kstring, "%d\n", param);
23     return simple_read_from_buffer(buf, lbuf, ppos, kstring, nbytes);
24 }
25
26 static ssize_t my_write(struct file *file, const char __user *buf, size_t
lbuf,
27                         loff_t *ppos)
28 {
29     ssize_t rc;
30     rc = simple_write_to_buffer(kstring, lbuf, ppos, buf, lbuf);
31     sscanf(kstring, "%d", &param);
32     pr_info("param has been set to %d\n", param);
33     return rc;
34 }
35
36 static const struct file_operations my_proc_fops = {
37     .owner = THIS_MODULE,
38     .read = my_read,
39     .write = my_write,
40 };
41
42 static ssize_t data_show(struct device *d,
43                         struct device_attribute *attr, char *buf)
44 {
45     return sprintf(buf, "%d\n", param);
46 }
47
48 static ssize_t data_store(struct device *d,
49                         struct device_attribute *attr,
50                         const char *buf, size_t count)
51 {
52     sscanf(buf, "%d", &param);
53     dev_dbg(d, ": write %d into data\n", param);
54
55     return strnlen(buf, count);
56 }
57
58 static DEVICE_ATTR_RW(data);
59
60 static struct attribute *ben_sysfs_entries[] = {

```

11.4 实验 4 : sysfs

```

61      &dev_attr_data.attr,
62      NULL
63 };
64
65 static struct attribute_group mydevice_attr_group = {
66     .name = "benshushu",
67     .attrs = ben_sysfs_entries,
68 };
69
70 static int __init my_init(void)
71 {
72     int ret;
73
74     my_root = proc_mkdir("benshushu", NULL);
75     my_proc = proc_create(NODE, 0, NULL, &my_proc_fops);
76     if (IS_ERR(my_proc)) {
77         pr_err("I failed to make %s\n", NODE);
78         return PTR_ERR(my_proc);
79     }
80     pr_info("I created %s on procfs\n", NODE);
81
82     my_device = platform_device_register_simple("benshushu", -1, NULL,
83 0);
83     if (IS_ERR(my_device)) {
84         printk("platform device register fail\n");
85         ret = PTR_ERR(my_device);
86         goto proc_fail;
87     }
88
89     ret = sysfs_create_group(&my_device->dev.kobj,
90 &mydevice_attr_group);
91     if (ret) {
92         printk("create sysfs group fail\n");
93         goto register_fail;
94     }
95     pr_info("create sysfs node done\n");
96
97     return 0;
98
99 register_fail:
100     platform_device_unregister(my_device);
101proc_fail:
102     return ret;
103}
104
105static void __exit my_exit(void)
106{
107     if (my_proc) {
108         proc_remove(my_proc);
109         proc_remove(my_root);
110         pr_info("Removed %s\n", NODE);
111     }
112
113     sysfs_remove_group(&my_device->dev.kobj, &mydevice_attr_group);
114
115     platform_device_unregister(my_device);
116}
117
118module_init(my_init);
119module_exit(my_exit);
120MODULE_LICENSE("GPL");

```

11.5 实验 5: debugfs

1. 实验目的

- 1) 写一个内核模块，在 debugfs 文件系统中创建一个名为“test”的目录。
- 2) 在 test 目录下面创建两个节点，分别是“read”和“write”。从“read”节点中可以读取内核模块的某个全局变量的值，向“write”节点写数据可以修改某个全局变量的值。

2. 实验要求

debugfs 文件系统中有不少 API 函数可以使用，它们定义在 include/linux/debugfs.h 头文件中。

```
struct dentry *debugfs_create_dir(const char *name,
                                 struct dentry *parent)

void debugfs_remove(struct dentry *dentry)

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                  struct dentry *parent,
                                  struct debugfs_blob_wrapper *blob)

struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                  struct dentry *parent, void *data,
                                  const struct file_operations *fops)
```

读者可以参照 Linux 内核中的例子来完成本实验。

3. 实验步骤

进入本实验参考代码。

```
#cd
/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5

#export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
```

编译内核模块。

```
rwk@ubuntu:lab5 debugfs$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0 M=/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5
b5_debugfs modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5_debugfs/debugfs-test.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5_debugfs/debugfs-test.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5_debugfs/debugfs-test.mod.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab5_debugfs/debugfs-test.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rwk@ubuntu:lab5 debugfs$
```

11.5 实验 5 : debugfs

拷贝内核模块到 kmodues 目录。

```
# cp debugfs-test.ko /home/r1k/r1k_basic/runninglinuxkernel_4.0/kmodules/
```

打开另外一个终端，进入 runninglinuxkernel_4.0 目录，启动 QEMU 虚拟机。

```
#cd /home/r1k/r1k_basic/runninglinuxkernel_4.0
#./run.sh arm32
```

进入 mnt 目录，安装本实验的内核模块。

```
#cd /mnt
#insmod debugfs-test.ko
```

```
/mnt # insmod debugfs-test.ko
I created benshushu on debugfs
```

创建的目录会在哪里呢？debugfs 一般会 mount 到/sys/kernel/debug/这个目录。

如果大家发现这个目录不存在，或者里面没有东西，说明 debugfs 没有挂载，或者输入“df”命令来查看。那么可以手工挂载。

```
mount -t debugfs none /sys/kernel/debug/
```

进入到/sys/kernel/debug/目录后，你会发现有一个 benshushu 的目录。

```
/sys/kernel/debug # ls
bdi          kprobes      suspend_stats
benshushu    memblock     tracing
clk          mmc0        ubi
extfrag     pm_qos       ubifs
fault_around_bytes regmap     usb
gpio         regulator   virtio-ports
hid          sleep_time  wakeup_sources
```

进入 benshushu 目录之后，你发发现有一个 my_debug 的节点。

```
/sys/kernel/debug/benshushu # ls
my_debug
/sys/kernel/debug/benshushu # ls -l
total 0
-r--r--r--  1 0          0          0 Feb  3 13:56 my_debug
/sys/kernel/debug/benshushu #
```

接下来就可以对这个节点进行读写操作了。

```
/sys/kernel/debug/benshushu # cat my_debug
100
/sys/kernel/debug/benshushu # echo 200 > my_debug
param has been set to 200
/sys/kernel/debug/benshushu # cat my_debug
200
/sys/kernel/debug/benshushu #
```

4. 实验代码分析

```
1 #include <linux/module.h>
```

奔跑吧 linux 社区出品

```

2 #include <linux/proc_fs.h>
3 #include <linux/uaccess.h>
4 #include <linux/init.h>
5 #include <linux/debugfs.h>
6
7 #define NODE "benshushu"
8
9 static int param = 100;
10 struct dentry *debugfs_dir;
11
12#define KS 32
13 static char kstring[KS];           /* should be less sloppy about overflows :)
*/
14
15 static ssize_t
16 my_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos)
17{
18     int nbytes = sprintf(kstring, "%d\n", param);
19     return simple_read_from_buffer(buf, lbuf, ppos, kstring, nbytes);
20}
21
22 static ssize_t my_write(struct file *file, const char __user *buf, size_t
lbuf,
23                         loff_t *ppos)
24{
25     ssize_t rc;
26     rc = simple_write_to_buffer(kstring, lbuf, ppos, buf, lbuf);
27     sscanf(kstring, "%d", &param);
28     //pr_info("param has been set to %d\n", param);
29     return rc;
30}
31
32 static const struct file_operations mydebugfs_ops = {
33     .owner = THIS_MODULE,
34     .read = my_read,
35     .write = my_write,
36};
37
38 static int __init my_init(void)
39{
40     struct dentry *debug_file;
41
42     debugfs_dir = debugfs_create_dir(NODE, NULL);
43     if (IS_ERR(debugfs_dir)) {
44         printk("create debugfs dir fail\n");
45         return -EFAULT;
46     }
47
48     debug_file = debugfs_create_file("my_debug", 0444,
49                                     debugfs_dir, NULL, &mydebugfs_ops);
50     if (IS_ERR(debug_file)) {
51         printk("create debugfs file fail\n");
52         debugfs_remove_recursive(debugfs_dir);
53         return -EFAULT;
54     }
55
56     pr_info("I created %s on debugfs\n", NODE);
57     return 0;
58}
59
60 static void __exit my_exit(void)
61{
62     if (debugfs_dir) {
63         debugfs_remove_recursive(debugfs_dir);

```

11.6 实验 6：使用 ftrace

```

64             pr_info("Removed %s\n", NODE);
65     }
66}
67
68module_init(my_init);
69module_exit(my_exit);
70MODULE_LICENSE("GPL");

```

11.6 实验 6：使用 ftrace

1. 实验目的

学习如何使用 ftrace 的常用跟踪器。

2. 实验详解

读者可以使用本章介绍的常用的 ftrace 跟踪器来跟踪某个内核模块的运行状况，比如跟踪 CFS 调度器的运行机理。

11.7 实验 7：添加一个新的跟踪点

1. 实验目的

通过本实验来学习如何在内核代码中添加一个跟踪点。

在 CFS 调度器的核心函数 `update_curr()` 里，添加一个跟踪点来观察 `cfs_rq` 就绪队列中 `min_vruntime` 成员的变化情况。

2. 实验详解

内核各个子系统目前已经有大量的跟踪点，如果觉得这些跟踪点还不能满足需求，可以自己手动添加一个，这在实际工作中也是很常用的技巧。

同样以 CFS 调度器中核心函数 `update_curr()` 为例，例如现在增加一个跟踪点来观察 `cfs_rq` 就绪队列中 `min_vruntime` 成员的变化情况。首先，需要在 `include/trace/events/sched.h` 头文件中添加一个名为 `sched_stat_minvruntime` 的跟踪点。

```

[include/trace/events/sched.h]

0 TRACE_EVENT(sched_stat_minvruntime,
1
2     TP_PROTO(struct task_struct *tsk, u64 minvuntime),
3
4     TP_ARGS(tsk, minvuntime),
5
6     TP_STRUCT_entry(
7         __array( char,           comm,          TASK_COMM_LEN)
8         __field( pid_t,         pid            )

```

```

9      __field( u64,          vruntime)
10     ),
11
12     TP_fast_assign(
13         memcpy(_entry->comm, tsk->comm, TASK_COMM_LEN);
14         _entry->pid           = tsk->pid;
15         _entry->vruntime       = minvruntime;
16     ),
17
18     TP_printk("comm=%s pid=%d vruntime=%Lu [ns]",
19             _entry->comm, _entry->pid,
20             (unsigned long long)_entry->vruntime)
21);

```

为了方便添加跟踪点，内核定义了一个 TRACE_EVENT 宏，只需要按要求填写这个宏即可。TRACE_EVENT 宏的定义如下。

```
#define TRACE_EVENT(name, proto, args, struct, assign, print) \
    DECLARE_TRACE(name, PARAMS(proto), PARAMS(args))
```

- name: 表示该跟踪点的名字, 如上面第 0 行代码中的 sched_stat_minvruntime。
- proto: 该跟踪点调用的原型, 如第 2 行代码中, 该跟踪点的原型是 trace_sched_stat_minvruntime(tsk, minvruntime)。
- args: 参数。
- struct: 定义跟踪器内部使用的__entry 数据结构。
- assign: 把参数复制到__entry 数据结构中。
- print: 打印的格式。

把 trace_sched_stat_minvruntime()添加到 update_curr()函数里。

```

0 static void update_curr(struct cfs_rq *cfs_rq)
1 {
2     ...
3     curr->vruntime += calc_delta_fair(delta_exec, curr);
4     update_min_vruntime(cfs_rq);
5
6     if (entity_is_task(curr)) {
7         struct task_struct *curtask = task_of(curr);
8         trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
9         trace_sched_stat_minvruntime(curtask, cfs_rq->min_vruntime);
10    }
11    ...
12}

```

重新编译内核并在 QEMU 上运行，首先看 sys 节点中是否已经有刚才添加的跟踪点。

```
#cd /sys/kernel/debug/tracing/events/sched/sched_stat_minvruntime
# ls
enable filter format id trigger
# cat format
name: sched_stat_minvruntime
ID: 208
format:
```

11.7 实验 7 : 添加一个新的跟踪点

```

field:unsigned short common_type; offset:0; size:2; signed:0;
field:unsigned char common_flags; offset:2; size:1; signed:0;
field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
field:int common_pid; offset:4; size:4; signed:1;

field:char comm[16]; offset:8; size:16; signed:0;
field:pid_t pid; offset:24; size:4; signed:1;
field:u64 vruntime; offset:32; size:8; signed:0;

print fmt: "comm=%s pid=%d vruntime=%Lu [ns]", REC->comm, REC->pid, (unsigned
long long)REC->vruntime
/sys/kernel/debug/tracing/events/sched/sched_stat_minvruntime #

```

上述信息显示增加跟踪点成功，如下是抓取 sched_stat_minvruntime 的信息。

```

# cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 247/247    #P:1
#
#                                     _-----> irqs-off
#                                     / _----> need-resched
#                                     | / _---> hardirq/softirq
#                                     || / _---> preempt-depth
#                                     ||| / _---> delay
#      TASK-PID      CPU#      ||||  TIMESTAMP  FUNCTION
#      ||         | ||||   |       |
sh-629  [000] d..3  27.307974: sched_stat_minvruntime: comm= sh
pid=629 vruntime=2120013310 [ns]
    rCU preempt-7  [000] d..3  27.309178: sched_stat_minvruntime: comm=
rCU preempt pid=7 vruntime=2120013310 [ns]
    rCU preempt-7  [000] d..3  27.319042: sched_stat_minvruntime: comm=
rCU preempt pid=7 vruntime=2120013310 [ns]
    rCU preempt-7  [000] d..3  27.329015: sched_stat_minvruntime: comm=
rCU preempt pid=7 vruntime=2120013310 [ns]
    kworker/0:1-284  [000] d..3  27.359015: sched_stat_minvruntime: comm=
kworker/0:1 pid=284 vruntime=2120013310 [ns]
    kworker/0:1-284  [000] d..3  27.399005: sched_stat_minvruntime: comm=
kworker/0:1 pid=284 vruntime=2120013310 [ns]
    kworker/0:1-284  [000] d..3  27.599034: sched_stat_minvruntime: comm=
kworker/0:1 pid=284 vruntime=2120013310 [ns]

```

内核里还提供了一个跟踪点的例子，在 samples/trace_events/ 目录中，读者可以自行研究。其中除了使用 TRACE_EVENT() 宏来定义普通的跟踪点外，还可以使用 TRACE_EVENT_CONDITION() 宏来定义一个带条件的跟踪点。如果要定义多个格式相同的跟踪点，DECLARE_EVENT_CLASS() 宏可以帮助减少代码量。

```
[arch/arm/configs/vexpress_defconfig]

- # CONFIG_SAMPLES is not set
+ CONFIG_SAMPLES=y
+ CONFIG_SAMPLE_TRACE_EVENTS=m
```

增加 CONFIG_SAMPLES 和 CONFIG_SAMPLE_TRACE_EVENTS，然后重新编译内核。它会编译成一个内核模块 trace-events-sample.ko，复制到 QEMU 里最小文件系统中，运行 QEMU。下面是该例子抓取的数据。

```
/sys/kernel/debug/tracing # cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 45/45    #P:1
#
#                                     _-----> irqs-off
#                                     / _----> need-resched
#                                     | / _---> hardirq/softirq
#                                     || / _--> preempt-depth
#                                     ||| / _> delay
#      TASK-PID    CPU#    ||||  TIMESTAMP   FUNCTION
#      | |        | | | |  | | | |
event-sample-636 [000] ...1    53.029398: foo_bar: foo hello 41 {0x1}
Snoopy (000000ff)
event-sample-636 [000] ...1    53.030180: foo_with_template_simple: foo
HELLO 41
event-sample-636 [000] ...1    53.030284: foo_with_template_print: bar
I have to be different 41
event-sample-fn-640 [000] ...1    53.759157: foo_bar_with_fn: foo Look
at me 0
event-sample-fn-640 [000] ...1    53.759285: foo_with_template_fn: foo
Look at me too 0
event-sample-fn-641 [000] ...1    53.759365: foo_bar_with_fn: foo Look at me 0
event-sample-fn-641 [000] ...1    53.759373: foo_with_template_fn: foo Look at
me too 0
```

11.8 实验 8：使用示踪标志

1. 实验目的

学习如何使用示踪标志（trace marker）来跟踪应用程序。

2. 实验详解

有时需要跟踪用户程序和内核空间的运行情况，示踪标志可以很方便地跟踪用户程序。trace_marker 是一个文件节点，允许用户程序写入字符串。ftrace 会记录该写入动作的时间戳。

下面是一个简单实用的示踪标志例子。

```
[trace_marker_test.c]
0 #include <stdlib.h>
1 #include <stdio.h>
2 #include <string.h>
3 #include <time.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <sys/time.h>
8 #include <linux/unistd.h>
9 #include <stdarg.h>
10 #include <unistd.h>
11 #include <ctype.h>
12
```

11.8 实验 8 : 使用示踪标志

```

13static int mark_fd = -1;
14static __thread char buff[BUFSIZ+1];
15
16static void setup_ftrace_marker(void)
17{
18    struct stat st;
19    char *files[] = {
20        "/sys/kernel/debug/tracing/trace_marker",
21        "/debug/tracing/trace_marker",
22        "/debugfs/tracing/trace_marker",
23    };
24    int ret;
25    int i;
26
27    for (i = 0; i < (sizeof(files) / sizeof(char *)); i++) {
28        ret = stat(files[i], &st);
29        if (ret >= 0)
30            goto found;
31    }
32    /* todo, check mounts system */
33    printf("canot found the sys tracing\n");
34    return;
35found:
36    mark_fd = open(files[i], O_WRONLY);
37}
38
39static void ftrace_write(const char *fmt, ...)
40{
41    va_list ap;
42    int n;
43
44    if (mark_fd < 0)
45        return;
46
47    va_start(ap, fmt);
48    n = vsnprintf(buff, BUFSIZ, fmt, ap);
49    va_end(ap);
50
51    write(mark_fd, buff, n);
52}
53
54int main()
55{
56    int count = 0;
57    setup_ftrace_marker();
58    ftrace_write("figo start program\n");
59    while (1) {
60        usleep(100*1000);
61        count++;
62        ftrace_write("figo count=%d\n", count);
63    }
64}

```

在 Ubuntu Linux 下编译，然后运行 ftrace 来捕捉示踪标志信息。

```

# cd /sys/kernel/debug/tracing/
# echo nop > current_tracer //设置function跟踪器是不能捕捉到示踪标志的
# echo 1 > tracing_on //打开ftrace才能捕捉到示踪标志

```

```
# ./trace_marker_test //运行trace_marker_test测试程序
[...] //停顿一小会儿
# echo 0 > tracing_on
# cat trace
```

下面是捕捉到的 trace_marker_test 测试程序写入 ftrace 的信息。

```
root@figo-OptiPlex-9020:/sys/kernel/debug/tracing# cat trace
# tracer: nop
#
# nop latency trace v1.1.5 on 4.0.0
# -----
# latency: 0 us, #136/136, CPU#1 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:4)
# -----
#     | task: -0 (uid:0 nice:0 policy:0 rt_prio:0)
# -----
#
#           _-----> CPU#
#           / _-----> irqs-off
#           | / _-----> need-resched
#           || / _----> hardirq/softirq
#           ||| / _--> preempt-depth
#           |||| / _---> delay
# cmd      pid    ||||| time   |   caller
# \      /    ||||| \   |   /
<...>-15686 1....1 7322484us!: tracing_mark_write: figo start program
<...>-15686 1....1 7422324us!: tracing_mark_write: figo count=1
<...>-15686 1....1 7522186us!: tracing_mark_write: figo count=2
<...>-15686 1....1 7622052us!: tracing_mark_write: figo count=3
[...]
```

读者可以在捕捉示踪标志时打开其他一些示踪事件，例如调度方面的事件，这样可以观察用户程序在两个示踪标志之间的内核空间发生了什么事情。Android 系统利用示踪标志功能实现了一个 Trace 类，Java 应用程序编程者可以方便地捕捉程序信息到 ftrace 中，然后利用 Android 提供的 Systrace 工具进行数据采集和分析。

```
[Android/system/core/include/cutils/trace.h]

#define ATRACE_BEGIN(name) atrace_begin(ATRACE_TAG, name)
static inline void atrace_begin(uint64_t tag, const char* name)
{
    if (CC_UNLIKELY(atrace_is_tag_enabled(tag))) {
        char buf[ATRACE_MESSAGE_LENGTH];
        size_t len;

        len = snprintf(buf, ATRACE_MESSAGE_LENGTH, "B|%d|%s", getpid(),
name);
        write(atrace_marker_fd, buf, len);
    }
}

#define ATRACE_END() atrace_end(ATRACE_TAG)
static inline void atrace_end(uint64_t tag)
{
    if (CC_UNLIKELY(atrace_is_tag_enabled(tag))) {
        char c = 'E';
```

11.8 实验 8 : 使用示踪标志

```

        write(atrace_marker_fd, &c, 1);
    }

}

[Android/system/core/libcutils/trace.c]

static void atrace_init_once()
{
    atrace_marker_fd = open("/sys/kernel/debug/tracing/trace_marker", O_WRONLY);
    if (atrace_marker_fd == -1) {
        goto done;
    }
    atrace_enabled_tags = atrace_get_property();
done:
    android_atomic_release_store(1, &atrace_is_ready);
}

```

因此，利用 `atrace` 和 `Trace` 类提供的接口可以很方便地在 Java 和 C/C++ 程序中添加信息到 `trace` 中。

3. 实验步骤

本实验可以在优麒麟 Host 主机上做。

首先编译 `trace_marker_test`。直接使用 host 主机的 `gcc` 编译器。

```
rlk@ubuntu:rlk_basic$ gcc trace_marker_test.c -o trace_marker_test
rlk@ubuntu:rlk basic$
```

然后 enable trace 功能。

```
#sudo su
#cd /sys/kernel/debug/tracing
#echo nop > current_tracer
#echo 1 > tracing_on
```

然后运行 `trace_marker_test` 测试程序。需要在 root 权限下运行该程序。

```
root@ubuntu:/home/rlk/rlk_basic# ./trace_marker_test
```

运行一段时间，大概 10 秒差不了，可以 `ctrl+c` 终止该程序了。

关闭 `trace`。

```
echo 0 > tracing_on
```

查看 `trace` 的 log 信息。

```
root@ubuntu:/sys/kernel/debug/tracing# cat trace
# tracer: nop
#
#                                     ----=> irqs-off
#                                     /----=> need_resched
#                                     | /----=> hardirq/softirq
#                                     || /----=> preempt-depth
#                                     ||| /---=> delay
#
#      TASK-PID   CPU#  TIMESTAMP  FUNCTION
#      | | | | | | |
<...>-27631 [000] .... 4411.647111: tracing_mark_write: rlk start program
<...>-27631 [000] .... 4411.748655: tracing_mark_write: rlk count=1
<...>-27631 [000] .... 4411.850583: tracing_mark_write: rlk count=2
<...>-27631 [000] .... 4411.952045: tracing_mark_write: rlk count=3
<...>-27631 [000] .... 4412.053664: tracing_mark_write: rlk count=4
<...>-27631 [000] .... 4412.153830: tracing_mark_write: rlk count=5
<...>-27631 [000] .... 4412.254203: tracing_mark_write: rlk count=6
<...>-27631 [000] .... 4412.355694: tracing_mark_write: rlk count=7
<...>-27631 [000] .... 4412.456540: tracing_mark_write: rlk count=8
<...>-27631 [000] .... 4412.556889: tracing_mark_write: rlk count=9
<...>-27631 [000] .... 4412.657776: tracing_mark_write: rlk count=10
<...>-27631 [000] .... 4412.759564: tracing_mark_write: rlk count=11
<...>-27631 [000] .... 4412.860958: tracing_mark_write: rlk count=12
<...>-27631 [000] .... 4412.962778: tracing_mark_write: rlk count=13
```

11.9 实验 9：使用 kernelshark 来分析数据

1. 实验目的

学会使用 trace-cm 和 kernelshark 工具来抓取和分析 ftrace 数据。

2. 实验详解

前面介绍了 ftrace 的常用方法。有些人希望有一些图形化的工具，trace-cmd 和 kernelshark 工具就是为此而生。

在 Ubuntu 上安装 trace-cmd 和 kernelshark 工具。

```
#sudo apt-get install trace-cmd kernelshark
```

trace-cmd 的使用方式遵循 reset->record->stop->report 模式，要用 record 命令收集数据，按“Ctrl+c”组合键可以停止收集动作，在当前目录下生产 trace.dat 文件。使用 trace-cmd report 解析 trace.dat 文件，这是文字形式的，kernelshark 是图形化的，更方便开发者观察和分析 数据。

```
figo@figo-OptiPlex-9020:~/work/test1$ trace-cmd record -h
trace-cmd version 1.0.3
usage:
  trace-cmd record [-v] [-e event [-f filter]] [-p plugin] [-F] [-d] [-o file] \
    [-s usecs] [-O option] [-l func] [-g func] [-n func] \
    [-P pid] [-N host:port] [-t] [-r prio] [-b size] [command ...]
  -e run command with event enabled
  -f filter for previous -e event
  -p run command with plugin enabled
  -F filter only on the given process
  -P trace the given pid like -F for the command
  -l filter function name
  -g set graph function
  -n do not trace function
```

11.9 实验 9：使用 kernelshark 来分析数据

```
-v will negate all -e after it (disable those events)
-d disable function tracer when running
-o data output file [default trace.dat]
-O option to enable (or disable)
-r real time priority to run the capture threads
-s sleep interval between recording (in usecs) [default: 1000]
-N host:port to connect to (see listen)
-t used with -N, forces use of tcp in live trace
-b change kernel buffersize (in kilobytes per CPU)
```

常用的参数如下。

- **-p plugin:** 指定一个跟踪器，可以通过 trace-cmd list 来获取系统支持的跟踪器。常见的跟踪器有 function_graph、function、nop 等。
- **-e event:** 指定一个跟踪事件。
- **-f filter:** 指定一个过滤器，这个参数必须紧跟着“-e”参数。
- **-P pid:** 指定一个进程进行跟踪。
- **-l func:** 指定跟踪的函数，可以是一个或多个。
- **-n func:** 不跟踪某个函数。

以跟踪系统进程切换的情况为例。

```
#trace-cmd record -e 'sched_wakeup*' -e sched_switch -e 'sched_migrate*'
#kernelshark trace.dat
```

通过 kernelshark 可以图形化地查看需要的信息，直观、方便，如图 11.2 所示。

打开菜单中的“Plots”→“CPUs”选项，可以选择要观察的 CPU。选择“Plots”→“Tasks”，可以选择要观察的进程。如图 11.3 所示，选择要观察的进程是 PID 为“8228”的进程，该进程名称为“trace-cmd”。

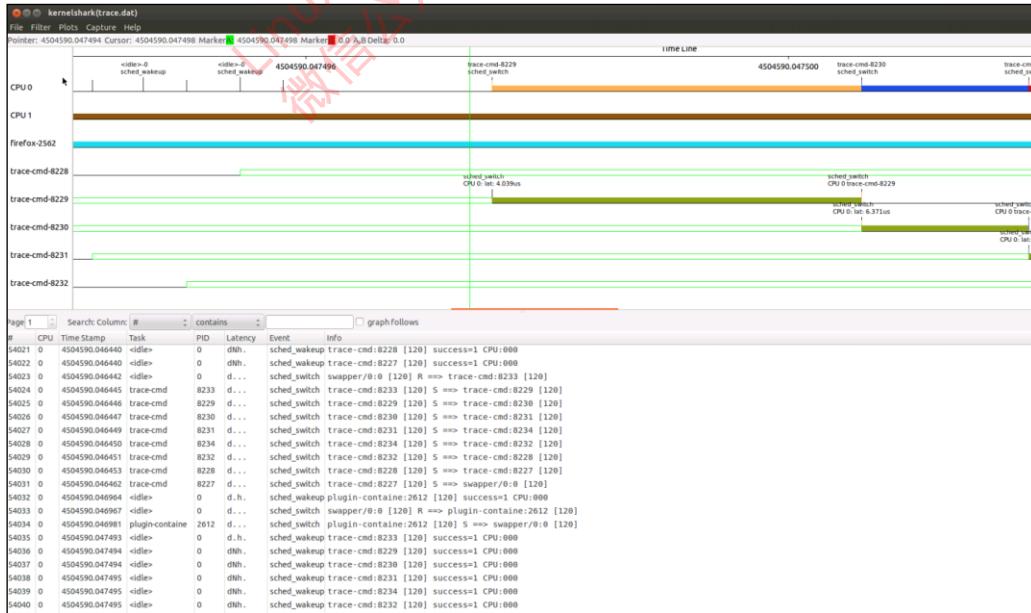


图11.2 kernelshark

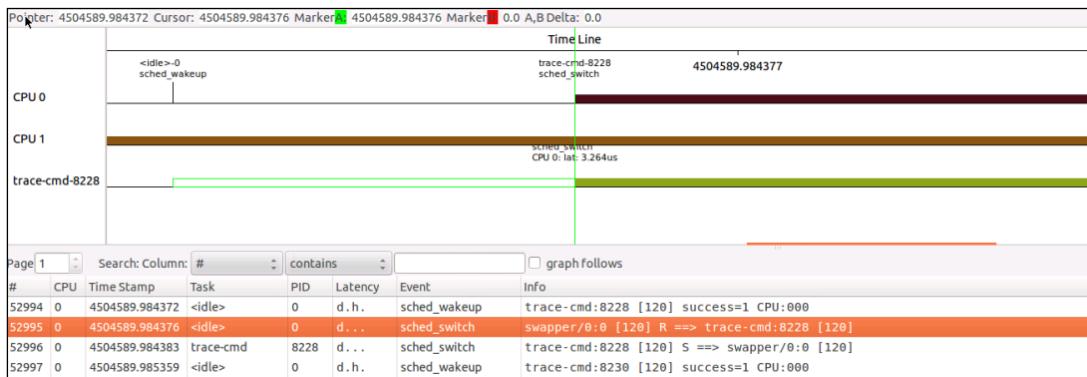


图11.3 用kernelshark查看进程切换

在时间戳 4504589.984372 中, trace-cmd-8228 进程在 CPU0 中被唤醒, 发生了 sched_wakeup 事件。在下一个时间戳中, 该进程被调度器调度执行, 在 sched_switch 事件中捕捉到该信息。

11.10 实验 10: 分析 oops 错误

在编写驱动程序或内核模块时, 常常会显式或隐式地对指针进行非法取值或使用不正确的指针, 导致内核发生一个 oops 错误。当处理器在内核空间访问一个非法指针时, 因为虚拟地址到物理地址的映射关系没有建立, 从而触发一个缺页中断。因为在缺页中断中该地址是非法的, 内核无法正确地为该地址建立映射关系, 因此内核触发了一个 oops 错误。

本章通过一个实验讲解如何分析一个 oops 错误。

1. 实验目的

编写一个简单的模块, 并且人为编造一个空指针访问错误来引发 oops 错误。

2. 实验详解

下面写一个简单的内核模块来验证如何分析一个内核 oops 错误。

```
[oops_test.c]
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

static void create_oops(void)
{
    *(int *)0 = 0; //人为编造一个空指针访问
```

11.10 实验 10：分析 oops 错误

```

}

static int __init my_oops_init(void)
{
    printk("oops module init\n");
    create_oops();
    return 0;
}

static void __exit my_oops_exit(void)
{
    printk("goodbye\n");
}

module_init(my_oops_init);
module_exit(my_oops_exit);
MODULE_LICENSE("GPL");

```

按照如下的 Makefile，把 oops_test.c 文件编译成内核模块。

```

BASEINCLUDE ?= /home/figo/work/test1/linux-4.0      #这里要用绝对路径
oops-objs := oops.o

obj-m     := oops.o
all :
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) modules;

clean:
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
    rm -f *.ko;

```

编译方法如下。

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

编译完成后，把 oops.ko 复制到 initramfs 文件系统的根目录，即 _install 目录下。重新编译内核并在 QEMU 上运行该内核，然后使用 insmod 命令加载该内核模块。oops 错误信息如下：

```

/ # insmod oops.ko
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = ee198000
[00000000] *pgd=8e135831, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1] PREEMPT SMP ARM
Modules linked in: oops(PO+)
CPU: 0 PID: 638 Comm: insmod Tainted: P          O  4.0.0 #25
Hardware name: ARM-Versatile Express
task: eeba6590 ti: ee150000 task.ti: ee150000
PC is at create_oops+0x18/0x20 [oops]
LR is at my_oops_init+0x18/0x24 [oops]
pc : [<bf000018>]    lr : [<bf002018>]    psr: 60000013
sp : ee151e48 ip : ee151e58 fp : ee151e54
r10: 00000000 r9 : ee150000 r8 : bf002000
r7 : bf0000cc r6 : 00000000 r5 : ee10a990 r4 : ee151f48
r3 : 00000000 r2 : 00000000 r1 : 00000000 r0 : 00000010
Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
Control: 10c5387d Table: 8e198059 DAC: 00000015
Process insmod (pid: 638, stack limit = 0xee150210)

Stack: (0xee151e48 to 0xee152000)

```

奔跑吧 linux 社区出品

```

1e40:          ee151e64 ee151e58 bf002018 bf00000c ee151ed4 ee151e68
1e60: c0008ae0 bf00200c c0e243bc 00000000 00000d0 ee800090 c0e243bc ee10a990
1e80: 000000d0 ee800090 ee151ed4 ee151e98 c0131ec4 c00c95c4 00000008 ee12aca0
1ea0: 00000000 0000000c c000f964 ee151f48 00000000 ee151f48 ee10a990 bf0000c0
1ec0: bf0000cc c000f964 ee151f04 ee151ed8 c009c3d8 c0008a88 c0126b9c c0126934
1ee0: ee151f48 ee151f48 00000000 bf0000c0 bf0000cc c000f964 ee151f3c ee151f08
1f00: c009d110 c009c378 ffff8000 00007fff c009b50c 00000080 ee151f3c 00160860
1f20: 00007a1a 0014f96d 00000080 c000f964 ee151fa4 ee151f40 c009d278 c009ceb8
1f40: ee151f5c ee151f50 f22c6000 00007a1a f22cb800 f22cb639 f22cd958 00000238
1f60: 000002a8 00000000 00000000 00000000 0000002b 0000002c 00000012 00000000
1f80: 00000016 00000000 00000000 00000000 beb07ea4 00000069 00000000 ee151fa8
1fa0: c000f7a0 c009d1ec 00000000 beb07ea4 00160860 00007a1a 0014f96d 7fffffff
1fc0: 00000000 beb07ea4 00000069 00000080 00000001 beb07ea8 0014f96d 00000000
1fe0: 00000000 beb07b38 0002b21b 0000af70 60000010 00160860 00000000 00000000
[<bf000018>] (create_oops [oops]) from [<bf002018>] (my_oops_init+0x18/0x24
[oops])
[<bf002018>] (my_oops_init [oops]) from [<c0008ae0>]
(do_one_initcall+0x64/0x110)
[<c0008ae0>] (do_one_initcall) from [<c009c3d8>] (do_init_module+0x6c/0x1c0)
[<c009c3d8>] (do_init_module) from [<c009d110>] (load_module+0x264/0x334)
[<c009d110>] (load_module) from [<c009d278>] (Sys_init_module+0x98/0xa8)
[<c009d278>] (Sys_init_module) from [<c000f7a0>] (ret_fast_syscall+0x0/0x4c)
Code: e24cb004 e92d4000 e8bd4000 e3a03000 (e5833000)
---[ end trace 2d2fed61250f46fa ]---
Segmentation fault
/ #

```

pgd=ee198000 表示出错时访问的地址对应的 PGD 页表地址，PC 指针指向出错指向的地址，另外 stack 也展示了出错时程序的调用关系。首先观察出错函数 create_oops+0x18/0x20，其中，0x18 表示指令指针在该函数第 0x18 字节处，该函数本身共 0x20 字节。

继续分析这个问题，假设有两种情况：一是有出错模块的源代码，二是没有源代码。在某些实际工作场景中，可能需要调试和分析没有源代码的 oops 情况。

先看有源代码的情况，通常在编译时添加在符号信息表中，下面用两种方法来分析。

(1) 使用 objdump 工具反汇编

```

figo$ arm-linux-gnueabi-objdump -SdCg oops.o //使用arm版本objdump工具

static void create_oops(void)
{
    0:   e1a0c00d    mov ip, sp
    4:   e92dd800    push {fp, ip, lr, pc}
    8:   e24cb004    sub fp, ip, #4
    c:   e92d4000    push {lr}
   10:   ebfffffe    bl 0 <__gnu_mcount_nc>
*(int *)0 = 0;
   14:   e3a03000    mov r3, #0
   18:   e5833000    str r3, [r3]
}
   1c:   e89da800    ldm sp, {fp, sp, pc}

```

通过反汇编工具可以看到出错函数 create_oops() 的汇编情况，这里把 C 语言和汇编语言一起显示出来了。0x14~0x18 字节的指令是把 0 赋值到 r3 寄存器，0x18~0x1c

11.10 实验 10：分析 oops 错误

字节的指令是把 r3 寄存器的值存放到 r3 寄存器指向的地址中，r3 寄存器的值为 0，所以这里是一个写空指针错误。

(2) 使用 gdb 工具

要方便快捷地定位到出错的具体地方，使用 gdb 中的“list”指令加上出错函数和偏移量即可。

```
$ arm-linux-gnueabi-gdb oops.o

(gdb) list *create_oops+0x18
0x18 is in create_oops
(/home/figo/work/test1/module_test_case/oops_test/oops_test.c:7).
2  #include <linux/module.h>
3  #include <linux/init.h>
4
5  static void create_oops(void)
6  {
7      *(int *)0 = 0;
8  }
9
10 static int __init my_oops_init(void)
11 {
(gdb)
```

如果出错地方是内核函数，那么可以使用 vmlinux 文件。

下面来看没有源代码的情况。对于没有编译符号表的二进制文件，可以使用 objdump 工具来 dump 出汇编代码，例如使用“arm-linux-gnueabi-objdump -d oops.o”命令来 dump 出 oops.o 文件。内核提供了一个非常好用的脚本，可以帮助快速定位问题，该脚本位于 Linux 内核源代码目录的 scripts/decodecode，首先把出错日志保存到一个 txt 文件中。

```
$ ./scripts/decodecode < oops.txt
Code: e24cb004 e92d4000 e8bd4000 e3a03000 (e5833000)
All code
=====
0:   e24cb004  sub fp, ip, #4
4:   e92d4000  push    {lr}
8:   e8bd4000  pop {lr}
c:   e3a03000  mov r3, #0
10:*  e5833000  str r3, [r3]          <- trapping instruction

Code starting with the faulting instruction
=====
0:   e5833000  str r3, [r3]
```

decodecode 脚本把出错的 oops 日志信息转换成直观、有用的汇编代码，并且告知出错具体是在哪个汇编语句中，这对于分析没有源代码的 oops 错误非常有用。

11.11 实验 11：使用 perf 工具来进行性能分析

1. 实验目的

通过一个例子来熟悉如何使用 perf 工具来进行性能分析。

2. 实验要求

- 1) 写一个 for 循环的测试程序例子。

```
//test.c
#include <stdio.h>
#include <stdlib.h>

void foo()
{
    int i,j;
    for(i=0; i< 10; i++)
        j+=2;
}
int main(void)
{
    int i;
    for(i = 0; i< 100000000; i++)
        foo();
    return 0;
}
```

使用以下命令进行编译：

```
$ gcc -o test -O0 test.c
```

- 2) 使用 perf stat 工具进行分析。
- 3) 使用 perf top 工具进行分析。
- 4) 使用 perf record 和 report 工具进行分析。

3. 实验步骤

本实验可以在优麒麟 Linux 上做。那首先要安装 perf 工具。

直接安装 perf 会提示找不到安装包。

```
rlk@ubuntu:lab12$ sudo apt install perf
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package perf
```

可以安装 linux-tools-generic 包。

11.12 实验 12：采集 perf 数据生成火焰图

```
rlk@ubuntu:lab12$ sudo apt install linux-tools-generic
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  linux-headers-4.15.0-29 linux-headers-4.15.0-29-generic linux-image-4.15.0-29-generic
  linux-modules-4.15.0-29-generic linux-modules-extra-4.15.0-29-generic
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  linux-tools-4.15.0-45 linux-tools-4.15.0-45-generic linux-tools-common
The following NEW packages will be installed:
  linux-tools-4.15.0-45 linux-tools-4.15.0-45-generic linux-tools-common linux-tools-generic
0 upgraded, 4 newly installed, 0 to remove and 74 not upgraded.
Need to get 4,768 kB of archives.
```

然后就可以运行 perf 工具了。

```
rlk@ubuntu:lab12$ perf
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
annotate      Read perf.data (created by perf record) and display annotated code
archive       Create archive with object files with build-ids found in perf.data file
bench         General framework for benchmark suites
buildid-cache Manage build-id cache.
buildid-list  List the buildids in a perf.data file
c2c          Shared Data C2C/HITM Analyzer.
config        Get and set variables in a configuration file.
data          Data file related processing
diff          Read perf.data files and display the differential profile
evlist        List the event names in a perf.data file
ftrace        simple wrapper for kernel's ftrace functionality
inject        Filter to augment the events stream with additional information
```

11.12 实验 12：采集 perf 数据生成火焰图

1. 实验目的

学会把 perf 采集的数据生成火焰图，并进行性能分析。

2. 实验要求

火焰图是性能大师 Brendang Gregg 的一个开源项目。

下面以上一个实验为例来介绍如何利用 perf 采集的数据生成一幅火焰图。

首先使用 perf record 命令来收集 test 程序的数据。

```
$sudo perf record -e cpu-clock -g ./test1
$sudo chmod 777 perf.data
```

然后使用 perf script 工具对 perf.data 进行解析。

```
$ perf script -i perf.data &> perf.unfold
```

接着将 perf.unfold 中的符号进行折叠。

```
$ ./stackcollapse-perf.pl perf.unfold &> perf.folded
```

最后生成火焰图，如图 11.6 所示。

```
| ./flamegraph.pl perf.folded > perf.svg
```

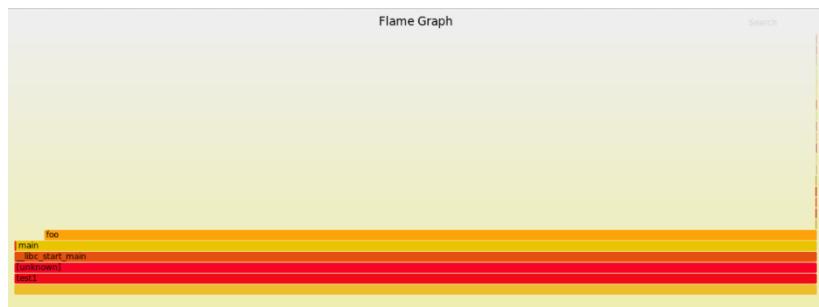


图11.6 火焰图

3. 实验步骤

由于出版社不让在书上放 网络地址，所以大家可以到下面地址下载火焰图项目。
火焰图（Flame Graph）是性能大师 Brendang Gregg 的一个开源项目，项目地址是在 <https://github.com/brendangregg/FlameGraph>

我们 runninglinuxkernel 代码里已经提前下载好了这个项目代码。

本实验在优麒麟 ubuntu 上完成。

```
root@ubuntu:lab12# ls
FlameGraph-master test.c
root@ubuntu:lab12#
```

编译 test 测试程序。

```
root@ubuntu:lab12# gcc test.c -o test
root@ubuntu:lab12#
```

运行 perf record 来收集数据 test 测试程序的数据。注意这里需要 root 权限。收集一小段时间即可，按“ctrl+c”键终止收集数据。

```
root@ubuntu:lab12# perf record -e cpu-clock -g ./test
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.057 MB perf.data (360 samples) ]
root@ubuntu:lab12#
```

使用 perf script 命令对 perf 数据进行解析。

```
# perf script > out.perf
```

将 perf.unfold 中的符号进行折叠：

```
# ./FlameGraph-master/stackcollapse-perf.pl out.perf > out.folded
```

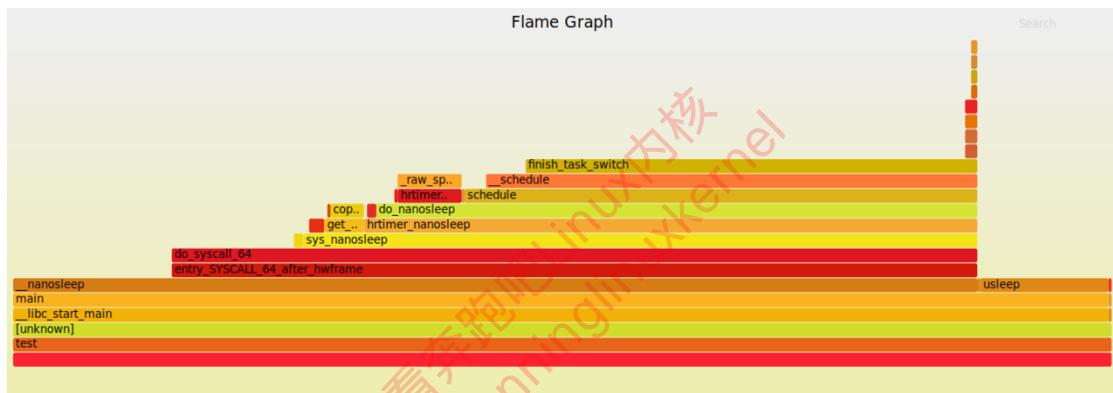
11.13 实验 13 : 使用 slab_debug 检查内存泄漏

最后生成 svg 图,

```
# ./FlameGraph-master/flamegraph.pl out.folded > kernel.svg
```

```
root@ubuntu:lab12#
root@ubuntu:lab12# ./FlameGraph-master/stackcollapse-perf.pl out.perf > out.folded
root@ubuntu:lab12#
root@ubuntu:lab12#
root@ubuntu:lab12# ./FlameGraph-master/flamegraph.pl out.folded > kernel.svg
```

生成的火焰图如下图所示:



注意:

- 书上列出的步骤，最好都在 root 权限下执行，比如 perf script 命令，不在 root 权限下执行的话，会报如下警告。

```
rlk@ubuntu:lab12$ perf script -i perf.data > perf.unfold
[kernel.kallsyms] with build id 67bf96b4286aa420b6c9218d00ad443e90a33187 not found, continuing without symbols
rlk@ubuntu:lab12$
```

- 项目的下载地址为: <https://github.com/brendangregg/FlameGraph>

11.13 实验 13: 使用 slab_debug 检查内存泄漏

1. 实验目的

学会使用 slab_debug 来检查内存泄漏。

2. 实验要求

在 Linux 内核中，小块内存分配大量使用 slab/slub 分配器。slab/slub 分配器提供了一个内存检测的小功能，很方便在产品开发阶段进行内存检查。内存访问中比较容易出

现错误的地方如下。

- 访问已经释放的内存。
- 越界访问。
- 释放已经释放过的内存。

本书以 slub_debug 为例，并在 QEMU 上实验。首先需要重新配置内核选项，打开 CONFIG_SLAB 和 CONFIG_SLUB_DEBUG_ON 这两个选项。

```
[arch/arm/configs/vexpress_defconfig]

# CONFIG_SLAB is not set
CONFIG_SLUB=y
CONFIG_SLUB_DEBUG_ON=y
CONFIG_SLUB_STATS=y
```

在 linux-4.0 内核 tools/vm 目录下编译一个 slabinfo 的工具。

```
# cd linux-4.0/tools/vm
# make slabinfo CFLAGS=-static ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

把 slabinfo 可执行文件复制到 QEMU 实验平台的 _install 目录中，然后重新 make vexpress_defconfig && make 来编译内核。slub_test.c 文件是模拟一次越界访问的场景，原本 buf 分配了 32 字节，但是 memset() 要越界写入 36 字节。

```
[slub_test.c]

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>

static char *buf;

static void create_slub_error(void)
{
    buf = kmalloc(32, GFP_KERNEL);
    if (buf) {
        memset(buf, 0x55, 36); <= 这里越界访问了
    }
}
static int __init my_test_init(void)
{
    printk("figo: my module init\n");
    create_slub_error();
    return 0;
}
static void __exit my_test_exit(void)
{
    printk("goodbye\n");
    kfree(buf);
}

MODULE_LICENSE("GPL");
module_init(my_test_init);
module_exit(my_test_exit);
```

11.13 实验 13：使用 slub_debug 检查内存泄漏

按照如下的 Makefile 把 slub_test.c 文件编译成内核模块。

```
BASEINCLUDE ?= /home/figo/work/test1/linux-4.0      #这里要用绝对路径
slub-objs := slub_test.o

obj-m      := slub.o
all :
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) modules;

clean:
    $(MAKE) -C $(BASEINCLUDE) SUBDIRS=$(PWD) clean;
    rm -f *.ko;
```

编译方法如下。

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

在内核 commandline 中添加“slub_debug”字符串来打开该功能。下面是在 QEMU 上加载 slub.ko 模块和运行 slabinfo 后的结果。

```
# insmod slub.ko
# ./slabinfo -v
=====
BUG kmalloc-32 (Tainted: G          O ) : Redzone overwritten
-----
INFO: 0xed6beab0-0xed6beab3. First byte 0x55 instead of 0xcc
INFO: Allocated in create_slub_error+0x28/0x50 [slub] age=1448 cpu=0 pid=775
      kmem_cache_alloc_trace+0xc4/0x270
      create_slub_error+0x28/0x50 [slub]
      0xbff002018
      do_one_initcall+0x64/0x110
      do_init_module+0x6c/0x1c0
      load_module+0x264/0x334
      SyS_init_module+0x98/0xa8
      ret_fast_syscall+0x0/0x4c
INFO: Freed in initcall_blacklisted+0xa8/0xc0 age=1448 cpu=0 pid=775
      kfree+0x268/0x270
      initcall_blacklisted+0xa8/0xc0
      do_one_initcall+0x30/0x110
      do_init_module+0x6c/0x1c0
      load_module+0x264/0x334
      SyS_init_module+0x98/0xa8
      ret_fast_syscall+0x0/0x4c
INFO: Slab 0xef5a77c0 objects=19 used=13 fp=0xed6be8f0 flags=0x0081
INFO: Object 0xed6bea90 @offset=2704 fp=0xed6be4e0

Bytes b4 ed6bea80: 09 03 00 00 30 97 ff ff 5a 5a 5a 5a 5a 5a 5a 5a
5a ....0...ZZZZZZZ
Object ed6bea90: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
UUUUUUUUUUUUUUUUU
Object ed6beaa0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
UUUUUUUUUUUUUUUUU
Redzone ed6beab0: 55 55 55 55                                     UUUU
Padding ed6beb58: 5a 5a 5a 5a 5a 5a 5a 5a                         ZZZZZZZZ
CPU: 0 PID: 777 Comm: slabinfo Tainted: G     B     O   4.0.0 #33
Hardware name: ARM-Versatile Express
```

```
[<c0018130>] (unwind_backtrace) from [<c0014158>] (show_stack+0x20/0x24)
[...]
[<c013da0c>] (SyS_write) from [<c000f7a0>] (ret_fast_syscall+0x0/0x4c)
FIX kmalloc-32: Restoring 0xed6beab0-0xed6beab3=0xcc
```

上述 slabinfo 信息显示这是一个 Redzone overwritten 错误，内存越界访问了。

下面来看另一种错误类型，对 slub_test.c 文件中的 create_slub_error() 函数进行如下修改。

```
static void create_slub_error(void)
{
    buf = kmalloc(32, GFP_KERNEL);
    if (buf) {
        memset(buf, 0x55, 32);
        kfree(buf);
        printk("figo:double free test\n");
        kfree(buf);    <= 这里重复释放了
    }
}
```

这是一个重复释放的例子，下面是运行该例子后的 slub 信息。该例子中的错误很明显，所以不需要运行 slabinfo 程序内核就能马上捕捉到错误。

```
/ # insmod slub.ko
figo: my module init
figo:double free test
=====
BUG kmalloc-32 (Tainted: G          O  ) : Object already free
-----

Disabling lock debugging due to kernel taint
INFO: Allocated in create_slub_error+0x28/0x74 [slub] age=0 cpu=0 pid=775
      kmem_cache_alloc_trace+0xc4/0x270
      create_slub_error+0x28/0x74 [slub]
      my_test_init+0x18/0x24 [slub]
      do_one_initcall+0x64/0x110
      do_init_module+0x6c/0x1c0
      load_module+0x264/0x334
      SyS_init_module+0x98/0xa8
      ret_fast_syscall+0x0/0x4c
INFO: Freed in create_slub_error+0x50/0x74 [slub] age=0 cpu=0 pid=775
      kfree+0x268/0x270
      create_slub_error+0x50/0x74 [slub]
      my_test_init+0x18/0x24 [slub]
      do_one_initcall+0x64/0x110
      do_init_module+0x6c/0x1c0
      load_module+0x264/0x334
      SyS_init_module+0x98/0xa8
      ret_fast_syscall+0x0/0x4c
INFO: Slab 0xef5a7640 objects=19 used=11 fp=0xed6b2a90 flags=0x0081
INFO: Object 0xed6b2a90 @offset=2704 fp=0xed6b29c0

Bytes b4 ed6b2a80: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a
5a .....ZZZZZZZ
Object ed6b2a90: 6b 6b
kkkkkkkkkkkkkkk
Object ed6b2aa0: 6b a5
```

11.13 实验 13 : 使用 slub_debug 检查内存泄漏

```

kkkkkkkkkkkkkkkk.
Redzone ed6b2ab0: bb bb bb bb
Padding ed6b2b58: 5a 5a 5a 5a 5a 5a 5a 5a      ....
CPU: 0 PID: 775 Comm: insmod Tainted: G     B      O    4.0.0 #34
Hardware name: ARM-Versatile Express
[<c0018130>] (unwind_backtrace) from [<c0014158>] (show_stack+0x20/0x24)
[...]
[<c009d270>] (SyS_init_module) from [<c000f7a0>] (ret_fast_syscall+0x0/0x4c)
FIX kmalloc-32: Object at 0xed6b2a90 not freed
/ # random: nonblocking pool is initialized

```

这是很典型的重复释放的例子，错误显而易见。可是在实际工程项目中没有这么简单，因为有些内存访问错误隐藏在层层的函数调用中或经过多层指针引用。

下面是另一个比较典型的内存访问错误，即访问了已经释放的内存。

```

static void create_slub_error(void)
{
    buf = kmalloc(32, GFP_KERNEL);
    if (buf) {
        kfree(buf);
        printk("figo:access free memory\n");
        memset(buf, 0x55, 32);  <=访问了已经被释放的内存
    }
}

```

下面是该内存访问错误的 slub 信息。

```

/ # insmod slub.ko
figo: my module init
figo:access free memory
/ #
/ #
/ #
/ # ./slabinfo -v
=====
BUG kmalloc-32 (Tainted: G      O  ) : Poison overwritten
-----
INFO: 0xed6d2a90-0xed6d2aae. First byte 0x55 instead of 0x6b
INFO: Allocated in create_slub_error+0x28/0x68 [slub] age=711 cpu=0 pid=775
    kmem_cache_alloc_trace+0xc4/0x270
    create_slub_error+0x28/0x68 [slub]
    0xbff002018
    do_one_initcall+0x64/0x110
    do_init_module+0x6c/0x1c0
    load_module+0x264/0x334
    SyS_init_module+0x98/0xa8
    ret_fast_syscall+0x0/0x4c
INFO: Freed in create_slub_error+0x3c/0x68 [slub] age=711 cpu=0 pid=775
    kfree+0x268/0x270
    create_slub_error+0x3c/0x68 [slub]
    0xbff002018
    do_one_initcall+0x64/0x110
    do_init_module+0x6c/0x1c0
    load_module+0x264/0x334
    SyS_init_module+0x98/0xa8
    ret_fast_syscall+0x0/0x4c
INFO: Slab 0xef5a7a40 objects=19 used=19 fp=0x (null) flags=0x0080

```

```

INFO: Object 0xed6d2a90 @offset=2704 fp=0xed6d29c0
Bytes b4 ed6d2a80: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a
5a .....ZZZZZZZ
Object ed6d2a90: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
UUUUUUUUUUUUUUUUU
Object ed6d2aa0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
UUUUUUUUUUUUUUUUU
Redzone ed6d2ab0: bb bb bb bb .....bb
Padding ed6d2b58: 5a 5a
CPU: 0 PID: 777 Comm: slabinfo Tainted: G B O 4.0.0 #35
Hardware name: ARM-Versatile Express
[<c0018130>] (unwind_backtrace) from [<c0014158>] (show_stack+0x20/0x24)
...
[<c013c3e0>] (SyS_open) from [<c000f7a0>] (ret_fast_syscall+0x0/0x4c)
FIX kmalloc-32: Restoring 0xed6d2a90-0xed6d2aae=0x6b
FIX kmalloc-32: Marking all objects used
SLUB: kmalloc-32 500 slabs counted but counter=501

```

该错误类型在 slub 中称为 Poison overwritten，即访问了已经释放的内存。如果产品中有内存访问错误，类似上述介绍的几种访问内存错误，那么也将存在隐患，就像是埋在产品中的一颗定时炸弹，也许用户在使用几天或几个月后就会出现莫名其妙的宕机，因此在产品开发阶段需要对内存做严格的检测。

3. 实验步骤

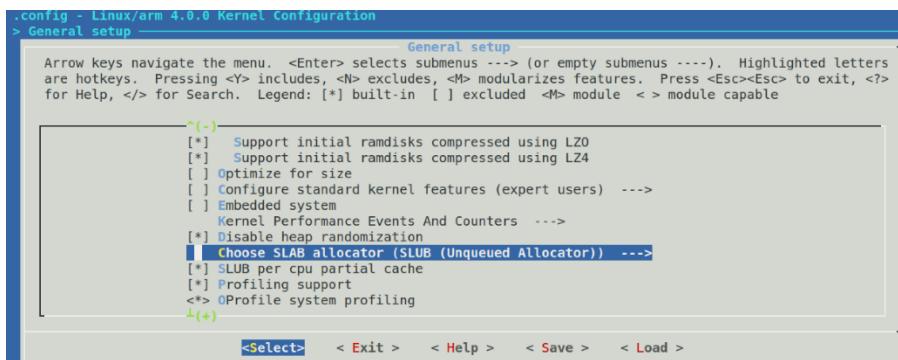
- 首先首先需要重新配置内核选项，打开 CONFIG_SLUB 和 CONFIG_SLUB_DEBUG_ON 这两个选项。

```

# export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
# make vexpress_defconfig
# make menuconfig

```

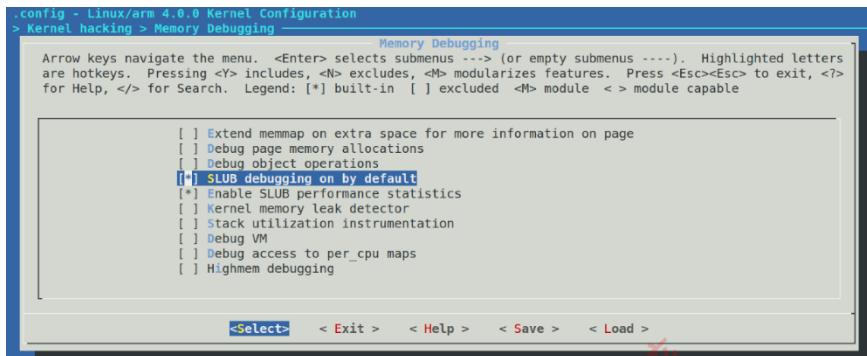
在“General setup”里选择 slub 做为默认的分配器。



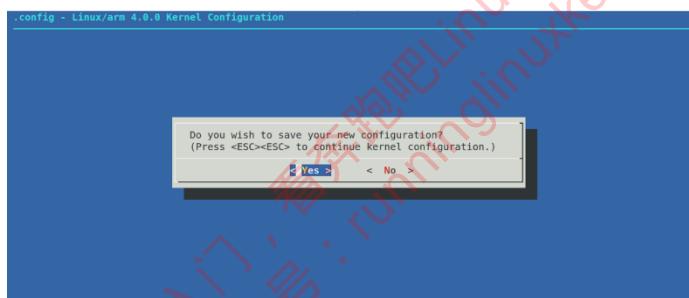
然后在 Kernel hacking-> Memory Debugging 菜单中选中如下两项：

11.13 实验 13 : 使用 slub_debug 检查内存泄漏

SLUB debugging on by default
Enable SLUB performance statistics



退出时候选择“Yes”保存。



当然你不用 menuconfig 这种图形化的菜单，也可以手工修改 config 文件。

```
[arch/arm/configs/vexpress_defconfig]

# CONFIG_SLAB is not set
CONFIG_SLUB=y
CONFIG_SLUB_DEBUG_ON=y
CONFIG_SLUB_STATS=y
```

2. 重新编译内核。
3. 在内核 commandline 中添加“slub_debug”字符串来打开该功能。
读者可以在 run.sh 这个脚本里添加了。

```
37      . . .
38      if [ ! -c $LR0OTFS ARM32/$CONSOLE_DEV_NODE ]; then
39          echo "please create console device node first, and recompile kernel"
40          exit 1
41      fi
42      qemu-system-arm -M vexpress-a9 -smp 4 -m 100M -kernel arch/arm/boot/zImage \
43          -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic \
44          -append "rdinit=/linuxrc console=ttyAMA0
45      loglevel=8 slub_debug" \
46      -9p-device,fsdev=kmod_dev,mount_tag=kmod_mount \
47      $DBG ;;
```

4. 编译 slaninfo 这个工具。

在 linux-4.0 内核 tools/vm 目录下编译一个 slabinfo 的工具。

```
# cd linux-4.0/tools/vm
# make slabinfo CFLAGS=-static ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

5. 编译本实验的内核模块 slub debug 的测试内核模块。

6. 把 slab info 和内核模块拷贝到 kmodules 目录。

7. 运行 Qemu 虚拟机。

```
# sh run.sh arm32
```

8. 进入 mnt 目录，运行 slub 测试内核模块。

```
/mnt # insmod slub1.ko
[ 17.791251] benshushu: my module init
[ 17.799584] =====
[ 17.803843] BUG kmalloc-64 (Tainted: G          O   ): Redzone overwritten
[ 17.810459]
[ 17.811356] -----
[ 17.811356] Disabling lock debugging due to kernel taint
[ 17.812053] INFO: 0xc5178640-0xc5178643. First byte 0x55 instead of 0xcc
[ 17.812938] INFO: Allocated at 0x55555555 age=0 cpu=0 pid=772
[ 17.813538] 0x55555555
[ 17.814265] create_slub_error+0x24/0x9c [slub1]
[ 17.814906] my_test_init+0x14/0x20 [slub1]
[ 17.817421] do_one_initcall+0x68/0x190
[ 17.817582] do_init_module+0xb4/0x278
[ 17.817682] load_module+0x3ec/0x570
[ 17.817795] Sys_init_module+0xa8/0xc0
[ 17.817899] ret_fast_syscall+0x0/0x34
[ 17.818258] INFO: Freed in initcall_blacklisted+0xc4/0xd4 age=1 cpu=0 pid=772
[ 17.818595] kfree+0x6d4/0x6ec
[ 17.818686] initcall_blacklisted+0xc4/0xd4
[ 17.818791] do_one_initcall+0x28/0x190
[ 17.818898] do_init_module+0xb4/0x278
[ 17.818994] load_module+0x3ec/0x570
[ 17.819086] Sys_init_module+0xa8/0xc0
[ 17.819183] ret_fast_syscall+0x0/0x34
[ 17.819345] TINFO: Slab 0xrc517864f00, objects=16 used=12 fn=0xc5178700 flags=0x0001
```

11.14 实验 14：使用 kmemleak 检查内存泄漏

1. 实验目的

学会使用 kmemleak 来检查内存泄漏。

2. 实验要求

kmemleak 是内核提供的一种检测内存泄漏工具，它会启动一个内核线程扫描内存，并打印发现新的未引用对象数量。kmemleak 有误报的可能性，但它给开发者提供了一个观察内存的路径和视角。要使用 kmemleak 功能，必须在内核配置时打开如下选项。

```
[arch/arm/configs/vexpress_defconfig]
```

11.14 实验 14：使用 kmemleak 检查内存泄漏

```
CONFIG_HAVE_DEBUG_KMEMLEAK=y
CONFIG_DEBUG_KMEMLEAK=y
CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF=y
CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE=4096
```

参照 slub_test.c 文件写一个内存泄漏的小例子。create_kmemleak()函数分别使用 kmalloc 和 vmalloc 分配内存，但一直不释放。

```
[kmemleak_test.c]

static void create_kmemleak(void)
{
    buf = kmalloc(120, GFP_KERNEL);
    buf = vmalloc(4096);
}
```

编译内核(make vexpress_defconfig && make)，并把 kmemleak.ko 复制到 initramfs 文件系统目录_install 中，然后重新编译内核。需要把“kmemleak=on”添加到内核启动 commandline 中。

```
$ qemu-system-arm -M vexpress-a9 -m 1024M -kernel arch/arm/boot/zImage -
append "rdinit=/linuxrc console=ttyAMA0 loglevel=8 kmemleak=on" -dtb
arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic

[...]
# echo scan > /sys/kernel/debug/kmemleak   <= 向kmemleak写入scan命令开始扫描
# insmod kmemleak_test.ko   <= 加载kmemleak_test.ko模块
[...]      <= 等待一会儿
# kmemleak: 2 new suspected memory leaks (see /sys/kernel/debug/kmemleak) <=
目标出现，发现两个可疑对象
# cat /sys/kernel/debug/kmemleak   <= 查看
```

下面是两个可疑对象的相关信息。

```
/ # cat /sys/kernel/debug/kmemleak
unreferenced object 0xec865690 (size 128):
comm "insmod", pid 781, jiffies 4294942049 (age 1147.540s)
hex dump (first 32 bytes):
  6b 6b
backtrace:
  [<c05b889c>] kmemleak_alloc+0x8c/0xcc
  [<c0135364>] kmem_cache_alloc_trace+0x1d8/0x29c
  [<bf000028>] create_kmemleak+0x28/0x54 [kmemleak]
  [<bf002018>] 0xbff002018
  [<c0008ae0>] do_one_initcall+0x64/0x110
  [<c009c454>] do_init_module+0x6c/0x1c0
  [<c009d1ac>] load_module+0x264/0x334
  [<c009d314>] SyS_init_module+0x98/0xa8
  [<c000f7a0>] ret_fast_syscall+0x0/0x4c
  [<ffffffff>] 0xffffffffffff
unreferenced object 0xf02b6000 (size 4096):
comm "insmod", pid 781, jiffies 4294942049 (age 1147.540s)
hex dump (first 32 bytes):
```

```

02 19 00 00 6a 28 00 00 02 19 00 00 76 28 00 00 ....j(.....v(..  

02 19 00 00 95 28 00 00 02 19 00 00 b1 28 00 00 ....(.....(..  

backtrace:  

[<c05b889c>] kmemleak_alloc+0x8c/0xcc  

[<c0126d88>] __vmalloc_node_range+0xb4/0xe0  

[<c0126e0c>] __vmalloc_node+0x58/0x60  

[<c0126e54>] vmalloc+0x40/0x48  

[<bf000038>] create_kmemleak+0x38/0x54 [kmemleak]  

[<bf002018>] 0xbff002018  

[<c0008ae0>] do_one_initcall+0x64/0x110  

[<c009c454>] do_init_module+0x6c/0x1c0  

[<c009d1ac>] load_module+0x264/0x334  

[<c009d314>] SyS_init_module+0x98/0xa8  

[<c000f7a0>] ret_fast_syscall+0x0/0x4c  

[<ffffffff>] 0xffffffff  

/ #

```

kmemleak 会提示内存泄漏可疑对象的具体栈调用信息，例如 `create_kmemleak+0x28/0x54`，表示在 `create_kmemleak()` 函数的第 0x28 字节处，以及可疑对象的大小、使用哪个分配函数等。

3. 实验步骤

首先需要重新配置内核选项，打开 `CONFIG_SLUB` 和 `CONFIG_SLUB_DEBUG_ON` 这两个选项。

```

# export ARCH=arm
#export CROSS_COMPILE=arm-linux-gnueabi-
# make vexpress_defconfig
# make menuconfig

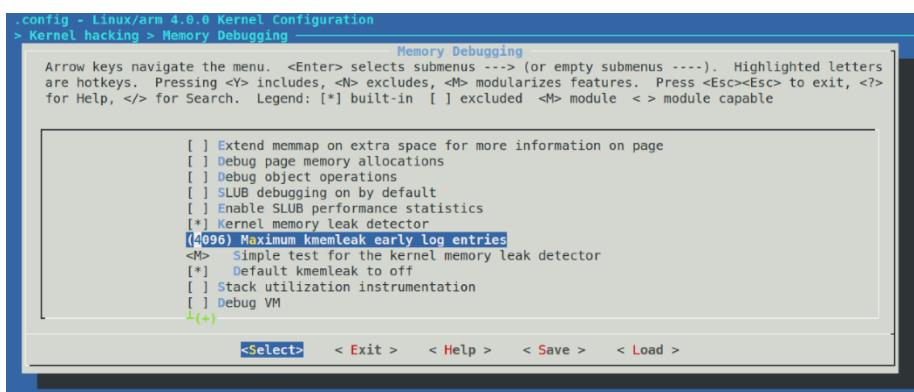
```

然后在 Kernel hacking-> Memory Debugging 菜单中选中如下两项：

```

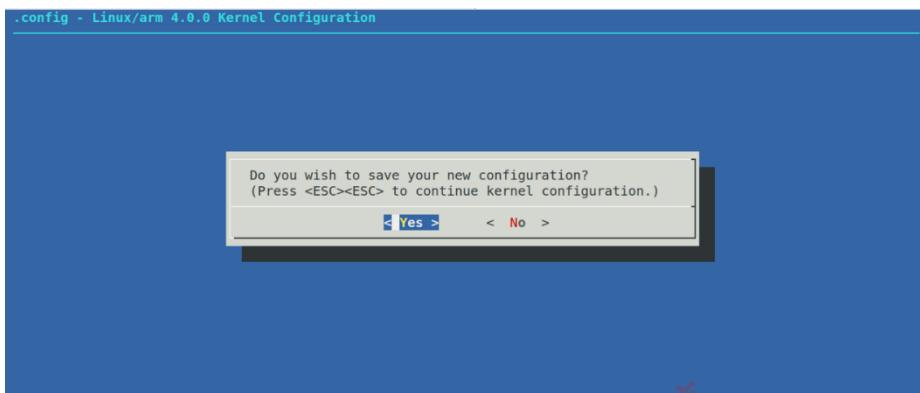
Kernel memory leak detector
Default kmemleak to off
Maximum kmemleak early log entries 设置4096, 这里不能设置太小, 否则kmemleak不
不成功。

```



11.14 实验 14：使用 kmemleak 检查内存泄漏

退出时候选择“Yes”保存。



当然你不用 menuconfig 这种图形化的菜单，也可以手工修改 config 文件。

```
[arch/arm/configs/vexpress_defconfig]
```

```
CONFIG_HAVE_DEBUG_KMEMLEAK=y
CONFIG_DEBUG_KMEMLEAK=y
CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF=y
CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE=4096
```

重新编译内核。

```
#make -j4
```

在内核 commandline 中添加“kmemleak=on”字符串来打开该功能。

读者可以在 run.sh 这个脚本里添加了。

```
37      arm32)    if [ ! -c $LROOT/$ROOTFS_ARM32/$CONSOLE_DEV_NODE ]; then
38          echo "please create console device node first, and recompile kernel"
39          exit 1
40      fi
41      qemu-system-arm -M vexpress-a9 -smp 4 -m 100M -kernel arch/arm/boot/zImage \
42          -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic \
43          -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8 kmemleak=on" \
44          --fsdev local,id=kmod_dev,path=$PWD/kmodules,security_model=none -device virtio
45          -9p-device,fsdev=kmod_dev,mount_tag=kmod_mount \
46          $DBG ;;
```

编译 kmemleak 的测试内核模块并把内核模块拷贝到 kmodules 目录。

运行 Qemu 虚拟机。

```
# sh run.sh arm32
```

检查 kmemleak 的 debug 节点有没有生成。Debug 目录是在/sys/kernel/debug/。

奔跑吧 linux 社区出品

```
/mnt # ls -l /sys/kernel/debug/
total 0
drwxr-xr-x 6 0 0 0 Jan 1 1970 bdi
drwxr-xr-x 15 0 0 0 Feb 17 03:58 clk
drwxr-xr-x 2 0 0 0 Jan 1 1970 extfrag
-rw-r--r-- 1 0 0 0 Jan 1 1970 fault_around_bytes
-r--r--r-- 1 0 0 0 Jan 1 1970 gpio
drwxr-xr-x 2 0 0 0 Jan 1 1970 hid
-r--r--r-- 1 0 0 0 Feb 17 04:00 kmemleak
drwxr-xr-x 2 0 0 0 Jan 1 1970 kprobes
drwxr-xr-x 2 0 0 0 Jan 1 1970 memblock
drwxr-xr-x 2 0 0 0 Jan 1 1970 mmc0
drwxr-xr-x 2 0 0 0 Jan 1 1970 pm_qos
drwxr-xr-x 26 0 0 0 Jan 1 1970 regmap
drwxr-xr-x 11 0 0 0 Jan 1 1970 regulator
-r--r--r-- 1 0 0 0 Jan 1 1970 sleep_time
-r--r--r-- 1 0 0 0 Jan 1 1970 suspend_stats
drwxr-xr-x 6 0 0 0 Jan 1 1970 tracing
drwxr-xr-x 2 0 0 0 Jan 1 1970 ubi
drwxr-xr-x 2 0 0 0 Jan 1 1970 ubifs
drwxr-xr-x 3 0 0 0 Jan 1 1970 usb
drwxr-xr-x 2 0 0 0 Jan 1 1970 virtio-ports
-r--r--r-- 1 0 0 0 Jan 1 1970 wakeup_sources
/mnt #
```

进入 mnt 目录，运行测试内核模块。

```
/sys/kernel/debug # cd /mnt/
/mnt # insmod kmemleak.ko
[ 49.155729] figo: my module init
/mnt #
```

向 kmemleak 写入 scan 命令开始扫描

```
/mnt # echo scan > /sys/kernel/debug/kmemleak
[ 130.847443] kmemleak: 1 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
/mnt #
```

如果没有发现可疑的内存泄漏，可以尝试在一次 scan.

```
/mnt # echo scan > /sys/kernel/debug/kmemleak
/mnt #
/mnt #
/mnt #
/mnt # cat /sys/kernel/debug/kmemleak
/mnt #
/mnt #
/mnt # cat /sys/kernel/debug/kmemleak
/mnt # echo scan > /sys/kernel/debug/kmemleak
[ 50.352306] kmemleak: 2 new suspected memory leaks (see /sys/kernel/debug/kmemleak)
/mnt #
```

发现了可疑的内存泄漏。

11.15 实验 15：使用 kasan 检查内存泄漏

```
/mnt # cat /sys/kernel/debug/kmemleak
unreferenced object 0xc6a9e000 (size 4096):
    comm "insmod", pid 707, jiffies 4294942214 (age 86.760s)
    hex dump (first 32 bytes):
        02 00 00 00 cd 00 00 bf 21 01 00 00 00 00 00 00 .....!.....
        00 00 00 00 00 00 00 01 00 00 00 cd 00 00 00 .....!
    backtrace:
        [<c0b04a64>] kmemleak_alloc+0x94/0xcc
        [<c021a4d4>] __vmalloc_node_range+0xf0/0x124
        [<c021a570>] __vmalloc_node+0x68/0x78
        [<c021a624>] vmalloc+0x5c/0x70
        [<bf00003c>] create_kmemleak+0x3c/0x70 [kmemleak]
        [<bf002014>] 0xbff002014
        [<<0008dc8>>] do_one_initcall+0x68/0x190
        [<c0119c1c>] do_init_module+0xb4/0x278
        [<c011a5d0>] load_module+0x3ec/0x570
        [<c011a7fc>] SyS_init_module+0xa8/0xc0
        [<c0014d40>] ret_fast_syscall+0x0/0x34
        [<ffffffff>] 0xffffffff
/mnt #
/mnt #
```

11.15 实验 15：使用 kasan 检查内存泄漏

1. 实验目的

学会使用 kasan 工具检查内存泄漏。

2. 实验要求

kasan (kernel address sanitizer) 在 Linux 4.0 中被合并到官方 Linux，它是一个动态检测内存错误的工具，可以检查内存越界访问和使用已经释放的内存等问题。Linux 内核早期有一个类似的工具 kmemcheck。kasan 比 kmemcheck 的速度快。虽然 kasan 在 Linux 4.0 时被合并到官方 Linux 中，但是直到 Linux 4.4 版本才开始支持 ARM64，因此我们采用 Linux 4.4 版本来做实验。要使用 kasan，必须打开 CONFIG_KASAN 等选项。

```
[linux-4.4/arch/arm64/configs/defconfig]

CONFIG_HAVE_ARCH_KASAN=y
CONFIG_KASAN=y
CONFIG_KASAN_OUTLINE=y
CONFIG_KASAN_INLINE=y
CONFIG_TEST_KASAN=m
```

kasan 模块提供了一个测试程序，在 lib/test_kasan.c 文件中定义了多种内存访问的错误类型。

- 访问已经释放的内存。
- 重复释放。
- 越界访问。

其中，越界访问是最常见的，而且情况比较复杂。test_kasan.c 文件抽象归纳了几

种常见的越界访问类型。

1) 右侧数组越界访问。

```
static noinline void __init kmalloc_oob_right(void)
{
    char *ptr;
    size_t size = 123;

    pr_info("out-of-bounds to right\n");
    ptr = kmalloc(size, GFP_KERNEL);

    ptr[size] = 'x';
    kfree(ptr);
}
```

2) 左侧数组越界访问。

```
static noinline void __init kmalloc_oob_left(void)
{
    char *ptr;
    size_t size = 15;

    pr_info("out-of-bounds to left\n");
    ptr = kmalloc(size, GFP_KERNEL);
    *ptr = *(ptr - 1);
    kfree(ptr);
}
```

3) Krealloc 扩大/缩小后越界访问。

```
static noinline void __init kmalloc_oob_krealloc_more(void)
{
    char *ptr1, *ptr2;
    size_t size1 = 17;
    size_t size2 = 19;

    pr_info("out-of-bounds after krealloc more\n");
    ptr1 = kmalloc(size1, GFP_KERNEL);
    ptr2 = krealloc(ptr1, size2, GFP_KERNEL);
    if (!ptr1 || !ptr2) {
        pr_err("Allocation failed\n");
        kfree(ptr1);
        return;
    }

    ptr2[size2] = 'x';
    kfree(ptr2);
}
```

4) 全局变量越界访问。

```
static char global_array[10];

static noinline void __init kasan_global_oob(void)
{
    volatile int i = 3;
    char *p = &global_array[ARRAY_SIZE(global_array) + i];
```

11.15 实验 15：使用 kasan 检查内存泄漏

```
    pr_info("out-of-bounds global variable\n");
    *(volatile char *)p;
}
```

5) 堆栈

```
static noinline void __init kasan_stack_oob(void)
{
    char stack_array[10];
    volatile int i = 0;
    char *p = &stack_array[ARRAY_SIZE(stack_array) + i];

    pr_info("out-of-bounds on stack\n");
    *(volatile char *)p;
}
```

以上几种越界访问都会导致严重的问题。

下面是一个越界访问的例子，KASAN 捕捉到的调试信息如下。

```
/ # insmod slab.ko
figo: my module init
=====
BUG: KASAN: slab-out-of-bounds in my_test_init+0x88/0xe8 [slub] at addr
fffffc067e48aff
Read of size 1 by task insmod/676
=====
BUG kmalloc-128 (Tainted: G          O  ): kasan: bad access detected
-----
Disabling lock debugging due to kernel taint
INFO: Slab 0xfffffffbd29f9200 objects=16 used=9 fp=0xfffffc067e48a00
flags=0x0080
INFO: Object 0xfffffc067e48a00 @offset=2560 fp=0xfffffc067e48900

Bytes b4 ffffffc067e489f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 .....
Object ffffffc067e48a00: 00 89 e4 67 c0 ff ff ff 00 00 00 00 00 00 00 00
00 ....g.....
Object ffffffc067e48a10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Padding ffffffc067e48af0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 .....
CPU: 0 PID: 676 Comm: insmod Tainted: G      B      O      4.4.0 #6
Hardware name: linux,dummy-virt (DT)
Call trace:
[<fffffc00008dc70>] dump_backtrace+0x0/0x270
[<fffffc00008def4>] show_stack+0x14/0x20
[<fffffc000604e30>] dump_stack+0x100/0x188
[<fffffc0002b0568>] print_trailer+0xf8/0x160
[<fffffc0002b547c>] object_err+0x3c/0x50
[<fffffc0002b72d8>] kasan_report_error+0x240/0x558
[<fffffc0002b7638>] __asan_report_load1_noabort+0x48/0x50
[<fffffbfffc0008088>] my_test_init+0x88/0xe8 [slub]
[<fffffc00008289c>] do_one_initcall+0x11c/0x310
[<fffffc00020ad8c>] do_init_module+0x1cc/0x588
[<fffffc0001c0838>] load_module+0x4070/0x5c40
[<fffffc0001c25b0>] SyS_init_module+0x1a8/0x1e0
[<fffffc0000864b0>] e10_svc_naked+0x24/0x28
Memory state around the buggy address:
fffffc067e48980: fc fc
```

```
=====
/ #
```

kasan 提示这是一个越界访问的错误类型 (slab-out-of-bounds)，并显示出错的函数名称和出错位置，为开发者修复问题提供便捷。

kasan 比 slub_debug 要高效得多，并且支持的内存错误访问类型更多。缺点是 kasan 需要比较新的内核 (Linux 4.0 以上, Linux 4.4 才支持 ARM64^①) 和比较新的 GCC 编译器 (GCC-4.9.2 以上)。

3. 实验步骤

虽然 kasan 在 Linux 4.0 时被合并到官方 Linux 中，但是直到 Linux 4.4 版本才开始支持 ARM64，因此我们采用 Linux 4.4 版本来做实验。

请读者按照第一章 lab4 的方法，在 Linux 4.4 上编一个 arm64 的系统，然后按照书上的步骤来完成本次实验。

11.16 实验 16：使用 valgrind 检查内存泄漏

1. 实验目的

通过本实验学会使用 valgrind 工具来检测应用程序的内存泄漏情况。

2. 实验要求

valgrind 是 Linux 上一套基于仿真技术的程序调试和分析工具，可以用来检测内存泄漏和内存越界等功能，valgrind 内置了很多功能。

- memcheck: 检查程序中的内存问题，如泄漏、越界、非法指针等。
- callgrind: 检测程序代码覆盖，以及分析程序性能。
- cachegrind: 分析 CPU 的缓存命中率、丢失率，用于进行代码优化。
- helgrind: 用于检查多线程程序的竞态条件。
- massif: 堆栈分析器，指示程序中使用了多少堆内存等信息。

本实验采用 memcheck 来检查应用程序的内存泄漏的情况，下面人为制造一个内存泄漏的测试程序。

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

^① 直到 Linux 4.9，kasan 仍然没有支持 ARM32 架构。

11.16 实验 16：使用 valgrind 检查内存泄漏

```
void test(void)
{
    int *buf = (int *)malloc(10 * sizeof(int));
    buf[10] = 0x55;
}

int main()
{
    test();
    return 0;
}
```

编译这个测试程序。

```
$ gcc -g -O0 valgrind_test.c -o valgrind_test
```

使用 valgrind 进行检查。

```
benshushu@ubuntu:~/work$ valgrind --leak-check=yes ./valgrind_test
==4160== Memcheck, a memory error detector
==4160== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4160== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4160== Command: ./valgrind_test
==4160==
==4160== Invalid write of size 4
==4160==   at 0x400544: test (valgrind_test.c:6)
==4160==     by 0x400555: main (valgrind_test.c:10)
==4160==   Address 0x5204068 is 0 bytes after a block of size 40 alloc'd
==4160==   at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4160==     by 0x400537: test (valgrind_test.c:5)
==4160==     by 0x400555: main (valgrind_test.c:10)
==4160==
==4160== HEAP SUMMARY:
==4160==   in use at exit: 40 bytes in 1 blocks
==4160==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==4160==
==4160== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4160==   at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4160==     by 0x400537: test (valgrind_test.c:5)
==4160==     by 0x400555: main (valgrind_test.c:10)
==4160==
==4160== LEAK SUMMARY:
==4160==   definitely lost: 40 bytes in 1 blocks
==4160==   indirectly lost: 0 bytes in 0 blocks
==4160==   possibly lost: 0 bytes in 0 blocks
==4160==   still reachable: 0 bytes in 0 blocks
==4160==   suppressed: 0 bytes in 0 blocks
==4160==
==4160== For counts of detected and suppressed errors, rerun with: -v
==4160== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

可以看到 valgrind 找到两个错误。

- 第 6 行代码存在无效的写入数据即越界访问。
- 内存泄漏，分配了 40 字节没有释放。

11.17 实验 17：使用 lkp-tests 工具进行性能测试

1. 实验目的

通过实验学习使用 lkp-tests 工具对 Linux 系统进行性能评估和分析，以便以后运用到实际工作中。

本实验需要连接网络，并且需要下载大量文件。不建议课堂做此实验。建议有精力的同学课后做该实验。

2. 实验详解

lkp-tests 项目是由 Intel 公司工程师创建的一个项目，该项目目前只能在 x86 平台上运行。lkp-tests 项目集成了很多性能测试工具和实用的测试脚本，很多测试用例一直运用在 Linux 内核社区的开发和测试中，比如内存性能测试 vm-scalability、综合性能测试 AIM7 等。

1) 下载安装 lkp-tests。

```
git clone https://github.com/01org/lkp-tests.git
cd lkp-tests
sudo make install
```

安装需要的依赖包。

```
$ sudo lkp install
```

2) 选择和安装某一项测试。

lkp-tests 项目支持的测试用例很多，读者可以查看 jobs 目录。

```
$ ls jobs/
dd-write.yaml          phoronix-test-suite.yaml
debug.yaml              piglit-all.yaml
```

3) 如果对 dd-write 测试感兴趣，需要通过如下命令安装和测试。

```
$ lkp split-job jobs/dd-write.yaml
jobs/dd-write.yaml => ./dd-write-10m-1HDD-ext4-2dd-4k.yaml
```

运行测试：

```
$ sudo lkp run ./dd-write-10m-1HDD-ext4-2dd-4k.yaml
```

4) 运行完之后，结果在/lkp/result 目录下面。lkp-tests 会默认收集很多信息，比如内核日志信息、ftrace 信息等。

11.18 实验 18 : kdump 死机实战 1 : 运行和配置 kdump (新增)

11.18 实验 18: kdump 死机实战 1: 运行和配置 kdump (新增)

kdump 是系统崩溃时用来把内存的内容进行转移存储的一个工具。kdump 会配置两个内核: 一个是正常使用的内核(或者称为 production kernel), 另一个是 crash dump 内核(或者称为 capture kernel)。它在 2006 年被合并到 Linux 内核社区里, kdump 会在内存中保留一块区域, 这个区域用来存放 capture kernel。当正常内核在运行过程中遇到崩溃等情况时, kdump 会通过 kexec 机制自动启动到 crash dump 内核, 这时会绕过 BIOS, 以免破坏了第一个内核的内存, 然后把 production kernel 的完整信息包括 CPU 寄存器、堆栈数据等转存储到指定文件中。

1. 实验目的

- 1) 学会使用 kdump 来定位崩溃等问题。
- 2) 学习在 runninglinuxkernel_4.0 上使用 kdump^①。

2. 实验步骤

注意: 在 Linux 4.0 上, 由于 ARM32 和 ARM64 对 kdump 的支持还不完善, 所以本实验没有办法在这两个架构上实现, 但是 x86_64 对 kdump 的支持就已经很完善了, 因此 runninglinuxkernel_4.0 可以实现对 x86_64 架构的 kdump 实验。

在做本实验之前, 请先安装第一章实验 7 来编译和运行一个 x86-64 的 QEMU+Debian 系统。

```
[ OK ] Started Serial Getty on ttyS0.
[ OK ] Started Getty on tty1.
[ OK ] Started getty on tty2-tty6.. and logind are not available.
[ OK ] Reached target Login Prompts.
[ OK ] Started LSB: Load kernel image with kexec.
[ 23.629756] kdump-tools[1200]: Starting kdump-tools: Creating symlink /var/lib/kdump/vmlinuz.
[ 23.691565] kdump-tools[1200]: ln: failed to create symbolic link '/var/lib/kdump/vmlinuz': No such file or directory
[ 25.433931] kdump-tools[1200]: Generating /var/lib/kdump/initrd.img-4.0.0+
Debian GNU/Linux 10 runninglinuxkernel tty50
runninglinuxkernel login: root
Password:
Linux runninglinuxkernel 4.0.0+ #3 SMP Sun Sep 1 22:15:09 PDT 2019 x86_64
Welcome Running Linux Kernel.
Created by BenshuShu <runninglinuxkernel@126.com>
Buy linux kernel training course:
https://shop115683645.taobao.com/
Wechat: runninglinuxkernel
```

首先我们检查一下 kdump 服务是否配置成功。运行 kdump-config show 命令来看。

^① 截至 2018 年 6 月, 在优麒麟 Linux 18.04 (包括 Ubuntu 18.04) 上使用 kdump 功能依然有问题, 如 crash 工具加载备份文件不成功, 因此本实验建议在 Ubuntu 16.04 上进行。另外, 本实验需要在实际物理机器上操作, 在 VMWare 等虚拟机上会有问题。

```
root@runninglinuxkernel:~# kdump-config show
DUMP MODE:          kdump
USE_KDUMP:          1
KDUMP_SYSCTL:      kernel.panic_on_oops=1
KDUMP_COREDIR:     /var/crash
crashkernel addr: 0x28000000
  /var/lib/kdump/vmlinuz: symbolic link to /boot/vmlinuz-4.0.0+
kdump initrd:
  /var/lib/kdump/initrd.img: symbolic link to /var/lib/kdump/initrd.img-4.0.0+
current state:    ready to kdump

kexec command:
  /sbin/kexec -p --command-line="noinitrd console=ttyS0 root=/dev/vda rootfstype=ext4 rw loglevel=8
nr_cpus=1 systemd.unit=kdump"
root@runninglinuxkernel:~# [ 251.717197] random: nonblocking pool is initialized
root@runninglinuxkernel:~#
```

从上图可以看到“current state: ready to kdump”，说明 kdump 服务已经配置成功。

接下来要编译内核模块。

打开另外一个终端，进入本实验代码目录。

```
#cd
/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17
/01_oops
```

```
rwk@figo-OptiPlex-9020:01_oops$ make
make -C /home/rwk/rwk_basic/runninglinuxkernel_4.0/ SUBDIRS=/home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17/01_oops modules;
make[1]: Entering directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
  CC [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17/01_oops/oops_test.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17/01_oops/oops.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17/01_oops/oops.mod.o
  LD [M]  /home/rwk/rwk_basic/runninglinuxkernel_4.0/rwk_lab/rwk_basic/chapter_11/lab17/01_oops/oops.ko
make[1]: Leaving directory '/home/rwk/rwk_basic/runninglinuxkernel_4.0'
rwk@figo-OptiPlex-9020:01_oops$
```

拷贝内核模块到 kmodues 目录。

```
# cp oops.ko /home/rwk/rwk_basic/runninglinuxkernel_4.0/kmodules/
```

进入到 QEMU 虚拟机中。进入 mnt 目录，加载 oops.ko 内核模块。

```
# cd /mnt
# insmod oops.ko
```

加载完内核模块之后，内核会触发 kdump 内核，并启动 kdump 服务进行内存转存功能。内存转存后的文件存放在/var/crash 目录。

```
root@runninglinuxkernel:~# cd /var/crash/
root@runninglinuxkernel:/var/crash# ls
201909021003  kexec_cmd
root@runninglinuxkernel:/var/crash#
```

在/var/crash 目录里，会生成以日期为文件夹。进入该目录之后，看到两个文件，一个是 dmesg 文件，这是内核日志，另外一个 dump 文件，这个就是内存转存的文件。

```
root@runninglinuxkernel:/var/crash/201909021003# ls
dmesg.201909021003  dump.201909021003
```

11.18 实验 18 : kdump 死机实战 1 : 运行和配置 kdump (新增)

使用 crash 命令来打开 dump 文件。

```
#crash dump.201909021003 /mnt/vmlinux
```

```
crash: cannot determine thread return address
  KERNEL: /mnt/vmlinux
  DUMPFILE: dump.201909021033 [PARTIAL DUMP]
    CPUs: 4
    DATE: Mon Sep  2 10:32:56 2019
    UPTIME: 00:00:59
  LOAD AVERAGE: 0.34, 0.13, 0.05
    TASKS: 79
  NODENAME: runninglinuxkernel
  RELEASE: 4.0.0+
  VERSION: #5 SMP Mon Sep 2 17:16:27 CST 2019
  MACHINE: x86_64 (3392 Mhz)
  MEMORY: 1 GB
  PANIC: "BUG: unable to handle kernel NULL pointer dereference at 0000000000000050"
    PID: 1343
  COMMAND: "insmod"
    TASK: ffff88003cd78900 [THREAD_INFO: ffff88003ce40000]
      CPU: 0
    STATE: TASK_RUNNING (PANIC)
```

- kernel: 发生内核崩溃的内核版本信息
- dumpfile: 我们要 dump 的文件
- cpus: 当前系统有多少 CPU
- date: 发生崩溃的时间
- load average: 发生奔溃时候系统的负载
- task: 发生奔溃时候系统有多少个进程
- release/version: 当前内核的版本信息
- machine: 这里显示的 x86 64 位机器
- memory: 显示内存大小
- panic: 发生奔溃的原因
- PID: 发生奔溃是进程的 PID
- command: 发生奔溃时候是那个进程导致的
- TASK: 发生奔溃时候进程的 task-struct 数据结构的地址, 这里还显示了 thread-info 数据结构的地址
- CPU: 发生奔溃的是在那个 CPU
- state: 导致发生奔溃的那个进程的进程状态是 task-running

从 PANIC 中可以看到, “"BUG: unable to handle kernel NULL pointer dereference at 0000000000000050"”, 这表明内核发生了空指针引用导致的内核错误。

下面使用 bt 命令来查看。

```

crash> bt
PID: 143 TASK: fffff88003cd78900 CPU: 0 COMMAND: "insmod"
#0 [ffff88003ce43920] machine_kexec at ffffffff81066ff1
#1 [ffff88003ce43980] crash_kexec at ffffffff8115c02a
#2 [ffff88003ce43a50] oops_end at ffffffff8100bc1a
#3 [ffff88003ce43ab0] no_context at ffffffff81076175
#4 [ffff88003ce43b10] __bad_area_nosemaphore at ffffffff81076379
#5 [ffff88003ce43b60] bad_area at ffffffff810763ea
#6 [ffff88003ce43b90] __do_page_fault at ffffffff810767bc
#7 [ffff88003ce43c00] do_page_fault at ffffffff810769c1
#8 [ffff88003ce43c10] page_fault at ffffffff819fb012
exception RIP: create oops+201
RIP: fffffffa0000014 RSP: fffff88003ce43cc8 RFLAGS: 00000282
RAX: 0000000000000000 RBX: 00007f05ce79f770 RCX: 000000006f676966
RDX: fffff88003ce43cf8 RSI: fffff88003ce43cf8 RDT: 0000000000000000
RBP: fffff88003ce43ce8 R8: 00000000006f6769 R9: fffff88003da64c00
R10: fffff88003d956e40 R11: ee6962732f3d4854 R12: 0000000000000000
R13: 00007f05ce79e3a0 R14: 0000000000000000 R15: 0000000000000000
ORIG_RAX: ffffffffffffc00000000000000 CS: 0010 SS: 0018
#9 [ffff88003ce43cf0] init_module at fffffffa000208b [oops]
#10 [ffff88003ce43d60] do_one_initcall at ffffffff81000e90
#11 [ffff88003ce43e00] do_init_module at ffffffff81155700
#12 [ffff88003ce43e50] load_module at ffffffff81155f2b
#13 [ffff88003ce43eb0] SYSC_finit_module at ffffffff81156204
#14 [ffff88003ce43f40] sys_finit_module at ffffffff81156158
#15 [ffff88003ce43f80] system_call_fastpath at ffffffff819fb95b2

```

造成内核崩溃的进程

造成内核崩溃的指令

rdi: 参数1

rsi: 参数2

使用 mod 命令加载内核符号表。

```

crash> mod -s oops /mnt/oops.ko
MODULE      NAME   SIZE  OBJECT FILE
fffffffffa00000a0  oops  841  /mnt/oops.ko
crash>

```

使用 dis 命令来反汇编出错的地方，RIP 寄存器指向出错的地方。

```

crash> dis -l fffffffa0000014
/home/rlik/rlik_basic/runninglinuxkernel_4.0/rlik_lab/rlik_basic/chapter_11/lab17/01_oops/oops_test.c: 16
0xfffffffffa0000014 <create_oops+0x14>: mov    0x50(%rax),%rax

```

可以看到 dis 命令已经可以显示出 RIP 寄存器指向出错的地方具体是在哪一行代码中。出错的代码是在 oops_test.c 的第 16 行。

```

12 int create_oops(struct vm_area_struct *vma, struct mydev_priv *priv)
13 {
14     unsigned long flags;
15
16     flags = vma->vm_flags;
17     printk("flags=0x%lx, name=%s\n", flags, priv->name);
18
19     return 0;
20 }
21

```

从源代码中可以看到出错的地方是在“flags=vma->vm_flags”这句话。从 dis 命令中看到的“mov 0x50(%rax),%rax”这句汇编中的 0x50 是指的什么意思呢？

使用 struct 命令来查看一个数据结构中成员和其偏移量。

11.18 实验 18 : kdump 死机实战 1 : 运行和配置 kdump (新增)

```
crash> struct -o vm_area_struct
struct vm_area_struct {
    [0x0] unsigned long vm_start;
    [0x8] unsigned long vm_end;
    [0x10] struct vm_area_struct *vm_next;
    [0x18] struct vm_area_struct *vm_prev;
    [0x20] struct rb_node vm_rb;
    [0x38] unsigned long rb_subtree_gap;
    [0x40] struct mm_struct *vm_mm;
    [0x48] pgprot_t vm_page_prot;
    [0x50] unsigned long VM_flags;
    struct {
        struct rb_node rb;
        unsigned long rb_subtree_last;
    } shared;
[0x58] }
```

因此我们可以知道 0x50 就是 vm_flags 成员在 vma 数据结构中的偏移量。

另外我们也可以使用 dis 命令来查看 create_oops 函数的反汇编。

```
crash> dis create_oops
0xfffffffffa000000 <create_oops>:      push   %rbp
0xfffffffffa000001 <create_oops+0x1>:    mov    %rsp,%rbp
0xfffffffffa000004 <create_oops+0x4>:    sub    $0x20,%rsp
0xfffffffffa000008 <create_oops+0x8>:    mov    %rdi,-0x18(%rbp)
0xfffffffffa00000c <create_oops+0xc>:    mov    %rsi,-0x20(%rbp)
0xfffffffffa000010 <create_oops+0x10>:   mov    -0x18(%rbp),%rax
0xfffffffffa000014 <create_oops+0x14>:   mov    0x50(%rax),%rax
0xfffffffffa000018 <create_oops+0x18>:   mov    %rax,-0x8(%rbp)
0xfffffffffa00001c <create_oops+0x1c>:   mov    -0x20(%rbp),%rdx
0xfffffffffa000020 <create_oops+0x20>:   mov    -0x8(%rbp),%rax
0xfffffffffa000024 <create_oops+0x24>:   mov    %rax,%rsi
0xfffffffffa000027 <create_oops+0x27>:   mov    $0xfffffffffa00007c,%rdi
0xfffffffffa00002e <create_oops+0x2e>:   mov    $0x0,%eax
0xfffffffffa000033 <create_oops+0x33>:   callq 0xfffffffff8110dbdb <printk>
0xfffffffffa000038 <create_oops+0x38>:   mov    $0x0,%eax
0xfffffffffa00003d <create_oops+0x3d>:   leaveq
0xfffffffffa00003e <create_oops+0x3e>:   retq
```

从 create_oops 函数的反汇编中，我们也可以找到出错的那条汇编指令，以及出错的地址。

接下来我们看出错的原因。

我们从 x86_64 架构的函数参数调用关系中可以知道，rdi 寄存器存放了函数的第一个参数，rsi 寄存器存放了函数的第二个参数。

先看第一个参数。creat_oops() 函数的第一个参数是 struct vm_area_struct *vma。使用 struct 命令来查看。

```
crash> struct vm_area_struct 0000000000000000
struct: invalid kernel virtual address: 0000000000000000
crash>
```

我们看到第一个参数是一个无效的内核虚拟地址，我们对这个无效的虚拟地址进行访问，当然会出现内核错误了，这也是本实验导致内核出错的罪魁祸首。

我们在来看第一个参数。creat_oops() 函数的第一个参数是 struct mydev_priv *priv。

```
crash> struct mydev_priv ffff88003ce43cf8
struct mydev_priv {
    name = "figo\000\352\377\377`|\326<\000\210\377\377\320\000\0
    i = 0xa
}
crash>
```

我们看到第二参数的值，符合我们的预期。

3. 实验代码

```

1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/init.h>
4 #include <linux/mm_types.h>
5 #include <linux/slab.h>
6
7 struct mydev_priv {
8     char name[64];
9     int i;
10};
11
12int create_oops(struct vm_area_struct *vma, struct mydev_priv *priv)
13{
14    unsigned long flags;
15
16    flags = vma->vm_flags;
17    printk("flags=0x%lx, name=%s\n", flags, priv->name);
18
19    return 0;
20}
21
22int __init my_oops_init(void)
23{
24    int ret;
25    struct vm_area_struct *vma = NULL;
26    struct mydev_priv priv;
27
28    vma = kmalloc(sizeof(*vma), GFP_KERNEL);
29    if (!vma)
30        return -ENOMEM;
31
32    kfree(vma);
33    vma = NULL;
34
35    smp_mb();
36
37    memcpy(priv.name, "figo", sizeof("figo"));
38    priv.i = 10;
39
40    ret = create_oops(vma, &priv);
41
42    return 0;
43}
44
45void __exit my_oops_exit(void)
46{
47    printk("goodbye\n");
48}
49
50module_init(my_oops_init);
51module_exit(my_oops_exit);
52MODULE_LICENSE("GPL");

```

11.19 实验 19 : kdump 死机实战 2 : 访问已经删除的链表 (新增)

11.19 实验 19: kdump 死机实战 2: 访问已经删除的链表 (新增)

1. 实验目的

- 1) 学会使用 kdump 来定位崩溃等问题。

2. 实验步骤

本实验参考代码是在:

/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_11/lab17/02_list

11.20 实验 20: kdump 死机实战 3: 内存 bug (新增)

1. 实验目的

- 1) 学会使用 kdump 来定位崩溃等问题。

2. 实验步骤

本实验参考代码是在:

/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_11/lab17/03_memory

11.21 实验 21: kdump 死机实战 4: 死战驱动死机问题 (新增)

1. 实验目的

- 1) 学会使用 kdump 来定位崩溃等问题。

2. 实验步骤

本实验参考代码是在:

/home/rkk/rkk_basic/runninglinuxkernel_4.0/rkk_lab/rkk_basic/chapter_11/lab17/04_re_gmap

11.22 实验 22: kdump 死机实战 5: 在 Centos 7.6 上安装和配置 kdump (新增)

1. 实验目的

- 1) 学会使用 kdump 来定位崩溃等问题。
- 2) 学习在 Centos 7.6 上使用 kdump^①。

2. 实验步骤

1. 优麒麟 18.04 (包括 ubuntu 18.04) 在 kdump+crash 方面还有 bug, 建议使用 Centos 7.6 或者 ubuntu 16.04 来做本实验。
2. 如果想在 vmware 虚拟机里做本实验, 建议使用 centos 7.6
3. 若想在 runninglinuxkernel_4.0 上做 kdump 实验。

在 vmware 虚拟机做 kdump 实验, 大概步骤如下:

1. 下载 windows 版本的 vmware player 安装软件。
2. 下载 Centos 7.6 安装 iso。
3. 安装软件包

```
#sudo yum install -y epel-release
#sudo yum update -y
```

4. 安装必备的软件包

```
# sudo yum install -y kernel kernel-devel kernel-headers gcc g++ gdb make ncurses-
devel gcc gcc-c++ autoconf automake boost ncurses ncurses-devel ncurses-libs boost-
devel openssl-devel ctags cscope openssh-server
```

5. 安装 kdump+crash

```
sudo yum install -y kexec-tools crash
```

6. 设置 crashkernel 预留内存大小, 修改/etc/default/grub 文件

这里设置 crashkernel 大小为 512M, 根据虚拟机物理内存大小来设置, 设置太小,

^① 截至 2018 年 6 月, 在优麒麟 Linux 18.04 (包括 Ubuntu 18.04) 上使用 kdump 功能依然有问题, 如 crash 工具加载备份文件不成功, 因此本实验建议在 Ubuntu 16.04 上进行。另外, 本实验需要在实际物理机器上操作, 在 VMware 等虚拟机上会有问题。

11.22 实验 22 : kdump 死机实战 5 : 在 Centos 7.6 上安装和配置 kdump (新增)

kdump 有可能会失败。2GB 内存，可以设置 384MB。

```

1 GRUB_TIMEOUT=5
2 GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
3 GRUB_DEFAULT=saved
4 GRUB_DISABLE_SUBMENU=true
5 GRUB_TERMINAL_OUTPUT="console"
6 GRUB_CMDLINE_LINUX="rd.lvm.lv=cl/root rd.lvm.lv=cl/swap crashkernel=512M nmi_watchdog=panic,1"
7 GRUB_DISABLE_RECOVERY="true"

```

7. 需重新生成 grub 配置文件，重启系统才能生效

```
#grub2-mkconfig -o /boot/grub2/grub.cfg
```

然后重启机器。

8. 开启 kdump 服务

```

# systemctl start kdump.service //启动 kdum
# systemctl enable kdump.service //设置开机启动

```

9. 检查 kdump 服务是否开启？

```
[root@localhost figo]# service kdump status
Redirecting to /bin/systemctl status kdump.service
● kdump.service - Crash recovery kernel arming
  Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled; vendor preset: enabled)
  Active: active (exited) since Fri 2019-01-11 12:34:09 EST; 1 day 14h ago
    Process: 1545 ExecStart=/usr/bin/kdumpctl start (code=exited, status=0/SUCCESS)
   Main PID: 1545 (code=exited, status=0/SUCCESS)
     CGroup: /system.slice/kdump.service
```

10. 手工触发一下 crash dump

```
# echo 1 > /proc/sys/kernel/sysrq ; echo c > /proc/sysrq-trigger
```

这时候，会重启，然后启动 kdump 内核进行数据抓取。

重启完成之后，系统会自动重启，重启后可以看到在/var/crash/目录下生成了 coredump 文件。

```
[root@localhost crash]# ls
127.0.0.1-2019-01-06-02:27:51
```

11. 安装对应的 kernel debuginfo。

编辑这个文件/etc/yum.repos.d/ CentOS-Debuginfo.repo
把里面 enable 设置为 1.

```

[base-debuginfo]
name=CentOS-7 - Debuginfo
baseurl=http://debuginfo.centos.org/7/$basearch/
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-Debug-7
enabled=1
#
```

然后

```

#yum update
#yum install -y kernel-debuginfo

```

12. 接下来就可以使用 crash 命令来分析 vmcore 文件。

```
[root@localhost 127.0.0.1-2019-01-06-02:27:51]# crash vmcore /usr/lib/debug/lib/modules/3.10.0-957.1.3.el7.x86_64/vmlinux
crash 7.2.3-8.el7
Copyright (C) 2002-2017 Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010 IBM Corporation
Copyright (C) 1999-2006 Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012 Fujitsu Limited
Copyright (C) 2006, 2007 VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011 NEC Corporation
Copyright (C) 1999, 2002, 2007 Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002 Mission Critical Linux, Inc.
This program is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions. Enter "help copying" to see the conditions.
This program has absolutely no warranty. Enter "help warranty" for details.
```

```
GNU gdb (GDB) 7.6
Copyright (C) 2013 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 "x86_64-unknown-linux-gnu"...
WARNING: kernel relocated [666MB]: patching 85619 gdb minimal_symbol values
      KERNEL: /usr/lib/debug/lib/modules/3.10.0-957.1.3.el7.x86_64/vmlinux
      DUMPFIELD: vmcore [PARTIAL DUMP]
      CPUS: 12
      DATE: Thu Jan 3 04:14:05 2019
      UPTIME: 00:02:09
LOAD AVERAGE: 0.93, 0.60, 0.24
      TASKS: 509
      NODENAME: localhost.localdomain
      RELEASE: 3.10.0-957.1.3.el7.x86_64
      VERSION: #1 SMP Thu Nov 29 14:49:43 UTC 2018
      MACHINE: x86_64 (3491 MHz)
      MEMORY: 31.9 GB
      PANIC: "SysRq : Trigger a crash"
      PID: 2466
COMMAND: "bash"
      TASK: fffffa0bcef234100 [THREAD_INFO: fffffa0bcd8a9c000]
      CPU: 0
      STATE: TASK_RUNNING (SYSRQ)

crash> █
```

13. 接下来就可以编译本次 lab 的内核模块。加载模块就会 触发 kdump。

14. 重启后，进行 vmcore 分析。Enjoy！

11.23 实验 23：kdump 死机实战 6：实战 arm64 死机（新增）

1. 实验目的

1) 学会使用 kdump 来定位在 ARM64 架构中的死机崩溃等问题。

2. 实验步骤

由于 Linux 4.0 内核对 ARM32 和 ARM64 架构的支持还不够完善，特别是 kdump 方面的支持。为此，我们基于 Linux 5.0 内核，特别制作了 arm64 的 kdump+crash 的

11.23 实验 23 : kdump 死机实战 6: 实战 arm64 死机 (新增)

开发套件。

这个开发套件已经上传到 github 上。

<https://github.com/figozhang/linux-5.0-kdump>

注意这个开发套件并没有包含在 vmware 镜像里，有需要的读者，可以在 vmware 镜像中自行下载。

```
| git clone https://github.com/figozhang/linux-5.0-kdump.git
```

1. 编译内核。

```
$ cd runninglinuxkernel-5.0
$ ./run_debian_arm64.sh build_kernel
```

2. 编译 rootfs 文件系统

```
$ sudo ./run_debian_arm64.sh build_rootfs
```

注意，编译 rootfs 文件系统，需要 root 权限。

3. 运行 QEMU+Debian 系统。

```
$ ./run_debian_arm64.sh run
```

登录 Debian 系统：

*用户名：root 或者 benshushu

*密码：123

4. 安装编译工具

QEMU 虚拟机可以通过 VirtIO-NET 技术来生成一个虚拟的网卡，并且通过 NAT 网络桥接技术和主机进行网络共享。首先使用 ifconfig 命令来检查网络配置。

```
root@benshushu:~# ifconfig
enp0s1: 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::c86e:28c4:625b:2767 prefixlen 64 scopeid 0x20<link>
                inet6 fe00::0:0:0:0/128 scopeid 0x40<site>
                  ether 52:54:00:12:34:56 txqueuelen 1000 (Ethernet)
                    RX packets 22 bytes 4219 (4.1 KiB)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 63 bytes 7432 (7.2 KiB)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

可以看到生成了一个名为 enp0s1 的网卡设备，分配的 IP 地址为：10.0.2.15。通过 apt update 命令来更新 Debian 系统的软件仓库。

在 QEMU 虚拟机中安装必要的软件包。

```
root@benshushu: # apt install build-essential
```

5. 编译内核模块

在 kmodules 目录中已经内置了 2 个小实验的代码。登录到 QEMU 系统中，这些实验代码就能在 mnt 目录中看到。

```
#cd /mnt/01_oops
# make

root@benshushu:/mnt/01_oops# make
make -C /lib/modules/5.0.0-rlk+/build M=/mnt/01_oops modules;
make[1]: Entering directory '/usr/src/linux'
  CC [M]  /mnt/01_oops/oops_test.o
  LD [M]  /mnt/01_oops/oops.o
Building modules, stage 2.
MODPOST 1 modules
make[2]: Warning: File '/mnt/01_oops/oops.mod.c' has modification time 0.12 s in the future
  CC      /mnt/01_oops/oops.mod.o
  LD [M]  /mnt/01_oops/oops.ko
make[2]: warning: Clock skew detected. Your build may be incomplete.
make[1]: Leaving directory '/usr/src/linux'
root@benshushu:/mnt/01 oops#
```

6. 检查 kdump 服务

```
root@benshushu:/mnt/01_oops# kdump-config show
DUMP_MODE:          kdump
USE_KDUMP:          1
KDUMP_SYSCTL:       kernel.panic_on_oops=1
KDUMP_COREDIR:      /var/crash
crashKernel addr: 0xufe00000
/var/lib/kdump/vmlinuz: symbolic link to /boot/vmlinuz-5.0.0-rlk+
kdump initrd:
/var/lib/kdump/initrd.img: symbolic link to /var/lib/kdump/initrd.img-5.0.0-rlk+
current state:      ready to kdump

kexec command:
/sbin/kexec -p --command-line="noinitrd root=/dev/vda rootfstype=ext4 rw loglevel=8 nr_cpus=1 systemd.unit=kdump-tools.service" --initrd=/var/lib/kdump/initrd.img /var/lib/kdump/vmlinuz
root@benshushu:/mnt/01_oops#
```

7. 加载内核模块

```
root@benshushu:/mnt/01_oops# insmod oops.ko
[ 567.273506] oops: loading out-of-tree module taints kernel.
[ 567.290785] Unable to handle kernel NULL pointer dereference at virtual address 0000000000000050
[ 567.291693] Mem abort info:
[ 567.291941]   ESR = 0x96000004
[ 567.292114]   Exception class = DABT (current EL), IL = 32 bits
[ 567.292335]   SET = 0, FnV = 0
[ 567.292447]   EA = 0, S1PTW = 0
[ 567.292566] Data abort info:
[ 567.292674]   ISV = 0, ISS = 0x00000004
[ 567.292804]   CM = 0, WnR = 0
[ 567.293153] user pgtable: 4k pages, 48-bit VAs, pgdp = 000000005768d5a5
[ 567.293409] [0000000000000050] pgd=0000000000000000
[ 567.293923] Internal error: Ooops: 96000004 [#1] SMP
[ 567.294188] Modules linked in: oops(OE+) aes_ce_blk(E) crypto_simd(E) cryptd(E) aes_ce_cipher(E) ghash_ce(E) gf128mul(E) aes_arm64(E) sha2_ce(E) sha256_arm64(E) evdev(E) shal_ce(E) cfg80211(E) rfkill(E) 8021q(E) garp(E) mrp(E) stp(E) llc(E) gpio_keys(E) virtio_net(E) net_failover(E) failover(E) 9p(E) fscache(E) ip_tables(E) x_tables(E) autofs4(E)
[ 567.295518] CPU: 2 PID: 2344 Comm: insmod Kdump: loaded Tainted: G          OE      5.0.0-rlk+ #1
[ 567.296016] Hardware name: linux,dummy-virt (DT)
[ 567.296298] pstate: 80000005 (Nzcv daif -PAN -UAO)
[ 567.296928] pc : create_oops+0x20/0x4c [oops]
[ 567.297105] lr : my_oops_init+0xa0/0x1000 [oops]
[ 567.297260] sp : ffff000012aa3b20
[ 567.297392] x29: ffff000012aa3b20 x28: ffff0000113fb090
[ 567.297599] x27: ffff000008c2e1d0 x26: ffff000008c2e180
[ 567.297787] x25: ffff000010b66988 x24: ffff0000113fb000
[ 567.297960] x23: ffff000008c2e018 x22: ffff8000235ff700
[ 567.298132] x21: 0000000000000000 x20: ffff000008c15000
```

11.23 实验 23 : kdump 死机实战 6: 实战 arm64 死机 (新增)

```
[ 567.299785] x3 : 0000000000000000 x2 : ffff000008c150a0
[ 567.299958] x1 : ffff000012aa3b84 x0 : 0000000000000000
[ 567.300179] Process insmod (pid: 2344, stack limit = 0x00000000abce95a7)
[ 567.300462] Call trace:
[ 567.300648]   create_oops+0x20/0x4c [oops]
[ 567.300807]   my_oops_init+0xa0/0x1000 [oops]
[ 567.301218]   do_one_initcall+0x54/0x1f0
[ 567.301359]   do_init_module+0x64/0x1d8
[ 567.301500]   load_module+0x1db8/0x1f00
[ 567.301640]   __se_sys_finit_module+0xf0/0x100
[ 567.301785]   __arm64_sys_finit_module+0x24/0x30
[ 567.301935]   e10_svc_common+0x78/0x120
[ 567.302085]   e10_svc_handler+0x38/0x78
[ 567.302214]   e10_svc+0x8/0xc
[ 567.302474] Code: f9000be1 aa0203e0 d503201f f9400fe0 (f9402800)
[ 567.303244] SMP: stopping secondary CPUs
[ 567.305496] Starting crashdump kernel...
[ 567.306074] Bye!
```

可以看到系统启动 crashdump 内核。

```
[ OK ] Reached target Network is Online.
Starting Kernel crash dump capture service...
[ 24.273401] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[ 25.233082] kdump-tools[235]: Starting kdump-tools: running makedumpfile -c -d 31 /proc/vmcore /var/crash/201909021130/dump-incomplete.
[ OK ] Stopped Kernel crash dump capture service.
Starting Kernel crash dump capture service...
[ 26.340266] virtio_net_virtio1 enp0s1: renamed from eth0
[ OK ] Found device Virtio network device.
[ OK ] Started ifup for enp0s1.
[ 28.326962] kdump-tools[261]: Starting kdump-tools: running makedumpfile -c -d 31 /proc/vmcore /var/crash/201909021130/dump-incomplete.
```

可以看到 makedumpfile 文件进行内存转存。

转存存放的地方在: /var/crash/ 目录。每一个转存目录会以日期为目录名字。

```
root@benshushu:/var/crash/201909021130# pwd
/var/crash/201909021130
```

8. 打开 crash 命令进行分析

进入 /var/crash 目录。拷贝 vmlinux 到 kmodules 目录。

```
# crash dump.201909021130 /mnt/vmlinux

KERNEL: /mnt/vmlinux
DUMPFILE: dump.201909021130 [PARTIAL DUMP]
CPUS: 4
DATE: Mon Sep  2 11:29:46 2019
UPTIME: 00:16:08
LOAD AVERAGE: 0.09, 0.28, 0.19
TASKS: 86
NODENAME: benshushu
RELEASE: 5.0.0-rlk+
VERSION: #1 SMP Thu Jun 20 05:53:19 CST 2019
MACHINE: aarch64 (unknown Mhz)
MEMORY: 1 GB
PANIC: "Unable to handle kernel NULL pointer dereference at virtual address 0000000000000050"
PID: 2344
COMMAND: "insmod"
TASK: ffff800023464880 [THREAD_INFO: ffff800023464880]
CPU: 2
STATE: TASK_RUNNING (PANIC)

crash>
```

11.24 死机问题进阶

研究 Linux 内核的死机问题非常有助于深刻理解 Linux 内核的运行机制，一方面有助于提高我们实际动手能力，另外一方面，在实际工作中会常常遇到 Linux 内核死机的难题。若我们在大学期间就很深入地去研究，那么到了企业之后就能很快的胜任实际的工作了。

为此，为了方便大家进一步深入去学习和研究 Linux 死机问题，笔者录制了一档《死机黑屏专题》节目，有兴趣同学可以到附录 2 了解详细情况。

Linux入门、看奔跑吧Linux内核
微信公众号：runninglinuxkernel

12.1 实验 1：使用 cppcheck 检查代码

第 12 章

开源社区

12.1 实验 1：使用 cppcheck 检查代码

1. 实验目的

学会使用代码缺陷静态检测工具完善代码质量。

2. 实验详解

cppcheck 是一个 C/C++ 的代码缺陷静态检测工具。它不仅可以检测代码中的语法错误，还可以检测出编译器检查不出来的缺陷类型，从而帮助程序员提升代码质量。

cppcheck 支持的检测功能如下。

- 野指针。
- 整型变量溢出。
- 无效的移位操作数。
- 无效的转换。
- 无效使用 STL 库。
- 内存泄漏检测。
- 代码格式错误以及性能原因检查。

cppcheck 的安装方法如下。

```
$ sudo apt install cppcheck
```

cppcheck 的使用方法如下。

```
cppcheck 选项 文件或者目录
```

默认情况下只显示错误信息，可以通过 “--enable” 命令来启动更多检查。

```
--enable=warning #打开警告消息  
--enable=performance #打开性能消息  
--enable=information #打开信息消息  
--enable=all #打开所有消息
```

12.2 实验 2：提交第一个 Linux 内核补丁

1. 实验目的

熟悉在 Linux 内核社区提交补丁的基本流程。

2. 实验详解

给 Linux 内核社区提交第一个补丁会涉及几个方面的问题：一是如何发现内核的缺陷，二是如何制作补丁，三是如何发送补丁。

(1) 如何发现内核的缺陷

作为一名新手，订阅 LKML 和下载 Linus 的 git 仓库是必备功课。

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

接下来就开始为 Linux 内核代码寻找错误或者缺陷，新手可以从如下几个方面入手。

- 查找编译警告。Linux 内核支持众多的 CPU 体系结构，以及多个版本的 GCC 编译器，通常会在某些情况下出现一些编译警告等信息。这些编译警告是新手制作补丁的好地方。
- 编码规范。读者可以仔细阅读内核源代码，包括注释、文档等，经常会有单词拼写错误、对齐不规范、代码格式不符合社区要求、代码不够简练等问题。这种问题也是新手入门的好地方。
- 其他人新提交的补丁集或者 staging 源代码。Linux 内核每次合并窗口时会合并大量的新特性，这些新进入的代码还没有经过社区的重复验证，常常有一些简单的错误，读者可以仔细阅读并发现里面可以制作成补丁的地方。另外，staging 源代码是一些没有经过充分测试的新增驱动模块，这些模块也是新手发掘补丁的好地方。

(2) 如何制作补丁

当我们发现了内核的错误或者缺陷时，下一步就是着手制作补丁了。制作补丁需要用到的工具是 git。

- 基于 Linux 内核主仓库最新的主分支创建一个新的分支。

```
$ git checkout -b "my-fix"
```

- 修改文件。这一步很重要的是进行测试，包括编译测试、单元测试和功能测试等。
- 生成新的提交。

```
$ git add .
$ git commit -s
```

“-s”命令会在提交信息末尾按照提交者名字加上一行“Signed-off-by”。下面以一

12.2 实验 2 : 提交第一个 Linux 内核补丁

个例子说明如何写一个合格提交的信息。

```

1  commit ffeb13aab68e2d0082cbb147dc765beb092f83f4
2  Author: Felipe Balbi <balbi@ti.com>
3  Date:   Wed Apr 8 11:45:42 2015 -0500
4
5      dmaengine: cppi41: add missing bitfields
6
7      Add missing directions, residue_granularity,
8      srd_addr_widths and dst_addr_widths bitfields.
9
10     Without those we will see a kernel WARN()
11     when loading musb on am335x devices.
12
13     Signed-off-by: Felipe Balbi <balbi@ti.com>
14     Signed-off-by: Vinod Koul <vinod.koul@intel.com>
```

第 1 行：该提交的 ID，这是 git 工具自动生成的。

第 2 行：该提交的作者。

第 3 行：该提交生成的日期。

第 5 行：对所做修改的简短描述。这一行以子系统、驱动或者架构的名字为前缀，然后是一句简短的描述。

第 7~11 行：对本次提交详尽的描述。

生成补丁。

使用 git format-patch 命令生成补丁。

```
$git format-patch -1 #生成一个补丁
```

对补丁进行代码格式检查。

Linux 内核有一套代码规范，所有提交到社区的补丁都必须遵守它。Linux 内核源代码中集成了一个脚本工具可以帮助我们检查补丁是否符合代码规范。

```
./scripts/checkpatch.pl your_fix.patch
```

(3) 如何发送补丁

推荐使用 git send-email 工具发送补丁到内核社区。安装 git send-email 工具。

```
$ sudo apt-get install git-email
```

配置 send-email。修改 ~/.gitconfig 文件，增加如下配置。

```
<~/gitconfig>

[sendemail]
    smtpencryption = tls
    smtpserver = smtp.126.com
    smtpuser = figo1802@126.com
    smtpserverport = 25
```

在发送补丁之前，我们需要知道这个补丁应该发给哪些审阅人。虽然你可以直接补丁发送到 LKML 邮件列表中，但是有可能审阅这个补丁的关键人物会错失了你的补丁。因此，最好将补丁发送给你修改的所属子系统的维护者。可以使用

get_maintainer.pl 来获取这些维护者的名字和邮箱地址。

```
| $ ./scripts/get_maintainer.pl your_fix.patch
```

最后可以使用如下命令来发送你的补丁。

```
| $ git send-email --to "tglx@linutronix.de" --to "xxx@redhat.com" --cc "linux-kernel@vger.kernel.org" 0001-your-fix.patch
```

这样补丁就被发送到社区里了，现在你需要做的就是耐心等待社区开发者的反馈。如果有社区开发者给你提了意见或者反馈，你应该积极回应，并根据意见进行修改，然后发送第二版的补丁，直到社区维护者接收你的补丁为止。

12.3 实验 3：管理和提交多个补丁组成的补丁集

1. 实验目的

学会如何管理和提交多个补丁组成的补丁集。

2. 实验详解

如果读者订阅了 Linux 社区的邮件列表，就会发现有一些补丁集有几个甚至几十个补丁，而且会不断地发送新的版本。如图 12.4 所示，这是一个关于 Speculative page faults 的补丁集，作者是 Laurent Dufour，该补丁集由 24 个补丁组成，目前已经是第 9 个版本。

主题	收件者	日期
[PATCH v9 00/24] Speculative page faults	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 01/24] mm: Introduce CONFIG_SPECULATIVE_PAGE_FAULT	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 02/24] x86/mm: Define CONFIG_SPECULATIVE_PAGE_FAULT	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 03/24] powerpc/mm: Define CONFIG_SPECULATIVE_PAGE_FAULT	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 04/24] mm: Prepare for FAULT_FLAG_SPECULATIVE	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 05/24] mm: Introduce pte_spinlock for FAULT_FLAG_SPECULATIVE	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 06/24] mm: make pte_unmap_same compatible with SPF	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 07/24] mm: VMA sequence count	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 08/24] mm: Protect VMA modifications using VMA sequence count	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 10/24] mm: Protect SPF handler against anon_vma changes	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 11/24] mm: Cache some VMA fields in the vm_fault structure	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 12/24] mm/migrate: Pass vma fault pointer to migrate_misplaced_page()	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 13/24] mm: Introduce _lru_cache_add_active_or_unevictable	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 14/24] mm: Introduce _maybe_mkwrite()	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 15/24] mm: Introduce _vm_normal_page()	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 16/24] mm: Introduce _page_add_new_anon_rmap()	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 17/24] mm: Protect mm_rb tree with a rwwlock	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 18/24] mm: Provide speculative fault infrastructure	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 19/24] mm: Adding speculative page fault failure trace events	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 20/24] perf: Add a speculative page fault sw event	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 21/24] perf tools: Add support for the SPF perf event	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 22/24] mm: Speculative page fault handler return VMA	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 23/24] x86/mm: Add speculative pagefault handling	Laurent Dufour	3/14/2018 1:59 AM
[PATCH v9 24/24] powerpc/mm: Add speculative page fault	Laurent Dufour	3/14/2018 1:59 AM

图12.4 Speculative page faults补丁集

读者通常会有如下的疑问。

- 这些补丁集是如何生成的呢？
- 当制作新版本的补丁集时，如何基于最新的 Linux 分支上进行？

12.3 实验 3：管理和提交多个补丁组成的补丁集

- 面对庞大的补丁集，如果社区针对某几个补丁有修改意见，那该如何制作新版的补丁集？

当开发某个功能或者新特性时，修改的文件和代码很多，这时需要把这些内容分割成功能单一的一个小补丁，然后这些小补丁组成一个大的补丁集。

(1) 从 v0 分支开始修改

首先基于最新的 Linux 内核主分支建立一个名为 my_feature_v0 的分支（下面简称 v0 分支）。

```
| $ git checkout -b "my_feature_v0"
```

在这个 v0 分支上进行开发和验证新功能。在这一步里，可以关注功能的实现和完善，最后不要忘记进行单元测试。

(2) 合理分割 v0 形成 v1 分支

当 v0 分支上的代码已经完成功能开发和验证之后，可以创建一个 v1 分支。

```
| $ git checkout -b "my_feature_v1"
```

在 v1 分支里，我们需要对代码进行合理的分割，也就是一个补丁只描述一个单一的修改。切记不要把多个不相关的修改放在一个补丁里，否则社区的维护者很难进行代码审阅。

(3) 发送补丁集到社区

使用 git format-patch 命令来生成补丁集。

```
| $ git format-patch -<patch number> --subject-prefix="PATCH v1" --cover-letter  
-o <patch-folder>
```

- patch-number: 根据补丁数目来生成补丁集。
- subject-prefix: 补丁集以“PATCH v1”为前缀。
- cover-letter: 生成一个总体描述的补丁，里面包含这个补丁集修改的文件以及修改的行数等信息。另外需要作者添加这个补丁的总体概述。
- patch-folder: 可以把补丁集生成到一个目录里，方便管理。

使用 checkpatch.pl 脚本对补丁集进行代码规范的检查。

最后使用 git send-email 发送 v1 版本的补丁集。接下来要做的就是静待社区给的反馈意见，并积极参与讨论。

(4) 创建 v2 分支并变基到最新的主分支上

假设社区里有不少开发者给了修改意见，这时你可以开始着手修改和发送 v2 版本的补丁集了。这时距离 v1 补丁集已经有一段时间了，可能过去了几周或者 1~2 个月时间了。Linux 内核 git 仓库的主分支已经发生了变化，这时你的 v2 补丁集必须基于最新的 Linux 内核主 分支。

基于 v1 的分支创建一个 v2 的分支。

```
| $ git checkout -b "my_feature_v2"
```

更新 Linux 内核的主分支到最新的状态。

```
$git checkout master
$git pull
```

然后把 v2 分支变基到最新的主分支上。

```
$git checkout my_feature_v2
$git rebase master
```

我们在第 2 章里已经学习过 git rebase 命令，它会让 v2 分支上的最新修改基于主分支上。当变基时发生了冲突，我们需要手动修改冲突。变基冲突的修改有如下几个步骤。

- 1) 修改冲突文件，例如 xx.c 文件。
- 2) 通过 git add 添加冲突文件，例如 git add xx.c。
- 3) 运行命令 git rebase --continue。
- 4) 在 v2 分支上修改社区的反馈。

接下来的工作就是基于 v2 分支进行社区反馈意见的修改了。这时我们可以使用 git rebase 命令对补丁进行逐一修改。假设补丁集只有 3 个小补丁，可以通过如下命令来进行修改。

```
$ git rebase -i HEAD~3 #对3个补丁进行修改
```

当运行 git rebase -i HEAD~3 命令之后，会对最新的 3 个补丁进行修改，如图 12.5 所示。我们可以选择如下命令对补丁进一步修订。

- p: 使用这个提交，不进行任何修改。
- r: 仅仅修改提交的信息，如补丁的说明等。如果社区里有人同意这个补丁，通常会给“Acked-by”或者“Reviewed-by”，这时可以把这些重要信息添加到补丁的提交信息里。
- e: 修改这个提交的代码。
- s: 合并到前一个的提交。
- f: 合并到前一个的提交，但是会丢弃这个提交信息。
- d: 删除这个提交。

12.3 实验 3：管理和提交多个补丁组成的补丁集

```
e 75c54072d40 add some experiment lab
r 0267d091363 lab: add some example drivers
pick 7148f3922fe rlk_basic: add lab9 for chapter_5

# Rebase 4b1a54721cf..7148f3922fe onto 4b1a54721cf (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

图12.5 git rebase命令修改补丁

如图 12.5 所示，我们对第三个补丁进行代码修改（选择 e 命令），对第二个补丁进行补丁信息修改（选择 r 命令），保存该文件之后，会自动停在第三个补丁里，如图 12.6 所示。

接下来就是动手修改社区给的反馈意见。当修改完成之后，可以通过如下命令完成变基工作。

```
$ git add xxx #添加修改过的文件
$ git commit -amend
$ git rebase --continue
```

```
ben@ubuntu:~/work/runninglinuxkernel_4.0$ git rebase --continue
Successfully rebased and updated refs/heads/r lk _ basic.
ben@ubuntu:~/work/runninglinuxkernel_4.0$ ben@ubuntu:~/work/runninglinuxkernel_4.0$ ben@ubuntu:~/work/runninglinuxkernel_4.0$ ben@ubuntu:~/work/runninglinuxkernel_4.0$ git rebase -i HEAD~3
Stopped at 9blcc00dcff... add some experiment lab
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
ben@ubuntu:~/work/runninglinuxkernel_4.0$ █
```

图12.6 git rebase

这样就完成了一个补丁的变基工作。当同时需要修改多个补丁的修改时，需要自动重复上述命令，直到变基结束，如图 12.7 所示。

```
ben@ubuntu:~/work/runninglinuxkernel_4.0$ git rebase --continue
Successfully rebased and updated refs/heads/rbk_basic.
ben@ubuntu:~/work/runninglinuxkernel_4.0$ █
```

图12.7 git rebase结束

当对补丁集的修改完成之后，我们就可以生成和发送 v2 版本的补丁集到社区了。给 Linux 内核社区发补丁是一件很需要耐心和毅力的事情，一个大的补丁集可能发送了好几版的补丁集也没有被接受，但是不要放弃，一定要坚持到底。最近有一个例子，Red Hat 工程师 Glisse 从 2014 年开始就往社区里推异构内存管理（Heterogeneous Memory Management, HMM）的补丁集，一直发到 v25 版本才被社区接受，前后花费了 3 年多时间。

12.4 实验 4：在 github 中创建和管理一个开源项目

1. 实验目的

熟悉如何创建和管理一个开源项目。

2. 实验详解

本实验以 Github 为例来学习如何创建和管理一个开源项目。

13.1 实验 1 : 块设备实验

第 13 章

块设备和文件系统 (新增)

13.1 实验 1：块设备实验

1. 实验目的

通过实验学习块设备机制。

2. 实验要求

- 写一个简单的 ramdisk 设备驱动。可以使用 ext4 或者 ext2 的格式化工具来格式化这个 ramdisk。
- 在这个设备驱动中，实现 HDIO_GETGEO 的 ioctl 命令，可以读出 ramdisk 的 hd_geometry 参数，比如有多少磁头，多少个柱面，多少个扇区等信息。写一个简单的用户空间的测试程序来读取 hd_geometry 参数。

13.2 实验 2：动手写一个简单文件系统

1. 实验目的

通过实验学习文件系统。

2. 实验要求

- 简单写一个简单文件系统，这个文件系统存储介质基于内存。

附录 1 – 免费视频

为了广大小伙伴能快速入门，我们精心制作了高清视频，购买了入门酱香篇的小

伙伴可以在线观看。

下载 33 集高清免费视频, 请登录 “奔跑吧 Linux 社区” 微信公众号, 在微信公众号中输入 “免费视频”, 然后就会弹出下载地址。



免费视频已经从 3 月 25 号登陆 B 站, 10 小时高清配套补充视频, 33 集连播, 每天一集。登陆 B 站 (<https://www.bilibili.com/>), 搜索: 奔跑吧 Linux 内核

Linux入门、看奔跑吧 Linux kernel
微信公众号: runninglinuxkernel

附录 2 – 进阶视频课程

除了提供免费的视频外, 我们还精心制作了更深入的 Linux 内核配套视频, 此视

第一季内存管理篇

频可以从淘宝或者微店上订阅。目前已经录制了：

- 第一季内存管理篇
- 第二季进程、中断、锁机制三合一
- 死机黑屏专题
- git 实战
- vim 实战

淘宝店地址： shop115683645.taobao.com



1. 第一季内存管理篇



第一季旗舰篇课程目录	
课程名称	时长
序言一： Linux内核学习方法论	0:09:13
序言二： 学习前准备	
序言2.1 Linux发行版和开发板的选择	0:13:56
序言2.2 搭建Qemu+gdb单步调试内核	0:13:51
序言2.3 搭建Eclipse图形化调试内核	0:10:59
实战运维1：查看系统内存信息的工具（一）	0:20:19
实战运维2：查看系统内存信息的工具（二）	0:16:32
实战运维3：读懂内核log中的内存管理信息	0:25:35
实战运维4：读懂 proc meminfo	0:27:59
实战运维5：Linux运维能力进阶线路图	0:09:40
实战运维6：Linux内存管理参数调优（一）	0:19:46
实战运维7：Linux内存管理参数调优（二）	0:31:20
实战运维8：Linux内存管理参数调优（三）	0:22:58
运维高级如何单步调试RHEL— CENTOS7的内核	0:15:45
运维高级如何单步调试RHEL— CENTOS7的内核二	0:41:28
vim:打造比source insight更强更好用的IDE（一）	0:24:58
vim:打造比source insight更强更好用的IDE（二）	0:20:28
vim:打造比source insight更强更好用的IDE（三）	0:23:25
实战git项目和社区patch管理	
2.0 Linux内存管理背景知识介绍	
奔跑2.0.0 内存管理硬件知识	0:15:25
奔跑2.0.1 内存管理总览一	0:23:27
奔跑2.0.2 内存管理总览二	0:07:35
奔跑2.0.3 内存管理常用术语	0:09:49
奔跑2.0.4 内存管理究竟管些什么东西	0:28:02
奔跑2.0.5 内存管理代码框架导读	0:38:09
2.1 Linux内存初始化	
奔跑2.1.0 DDR简介	0:06:47
奔跑2.1.1 物理内存三大数据结构	0:19:39
奔跑2.1.2 物理内存初始化	0:11:13
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
奔跑2.1 代码导读C语言部分（一）	0:27:34
奔跑2.1 代码导读C语言部分（二）	0:21:28

第一季基础和原理讲解课程

第一季内存管理篇

2.10 缺页中断处理	
奔跑2. 10. 1 缺页中断一	0:31:07
奔跑2. 10. 2 缺页中断二	0:16:58
2.11 page数据结构	
奔跑2. 11 page数据结构	0:29:41
2.12 反向映射机制	
奔跑2. 12. 1 反向映射机制的背景介绍	0:19:01
奔跑2. 12. 2 RMAP四部曲	0:07:31
奔跑2. 12. 3 手撕Linux2. 6. 11上的反向映射机制	0:07:35
奔跑2. 12. 4 手撕Linux4. x上的反向映射机制	0:10:08
2.13 回收页面	
奔跑2. 13 页面回收一	0:16:07
奔跑2. 13 页面回收二	0:11:41
2.14 匿名页面的生命周期	0:26:16
2.15 页面迁移	0:19:07
2.16 内存规整	0:24:03
2.17 KSM	0:28:17
2.18 Dirty COW内存漏洞	制作中会更新
2.19 内存数据结构和API总结	制作中会更新
2.20 Meltdown漏洞分析	
奔跑2. 20. 1 Meltdown背景知识	0:10:13
奔跑2. 20. 2 CPU体系结构之指令执行	0:11:25
奔跑2. 20. 3 CPU体系结构之乱序执行	0:11:03
奔跑2. 20. 4 CPU体系结构之异常处理	0:03:48
奔跑2. 20. 5 CPU体系结构之cache	0:10:56
奔跑2. 20. 6 进程地址空间和页表及TLB	0:17:39
奔跑2. 20. 7 Meltdown漏洞分析	0:06:04
奔跑2. 20. 8 Meltdown漏洞分析之x86篇	0:12:07
奔跑2. 20. 9 ARM64上的KPTI解决方案	0:25:39

后期课程不定期更新中，等您来提交topic

第一季基础和原理讲解课程

除了有基础课程之外，我们还录制手把手分析内核代码的代码导读视频。代码导读视频会不断更新。

< > ▶ ⌂ | 我的网盘 > 《奔跑吧视频教程》 > 3.0奔跑 > 第1季 > 代码导读 >

文件名	修改时间
奔跑2.1 代码导读C语言部分（二）.vep	2019-05-09 10:00
奔跑2.1 内存初始化之代码导读二.vep	2019-05-09 10:00
奔跑2.1 代码导读C语言部分(一).vep	2019-05-09 10:00
奔跑2.0.4 内存管理究竟管些什么东西.vep	2019-05-09 10:00
奔跑2.0.5 内存管理代码框架导读.vep	2019-05-09 10:00
使用DS-5调试arm64内核.vep	2019-08-26 10:51
奔跑2.1 内存初始化之代码导读一.vep	2019-05-09 10:00
代码导读4分配物理页面.vep	2019-05-03 17:17
代码导读3页表映射.vep	2019-05-03 17:17
代码导读5slab机制.vep	2019-05-03 17:17

第一季代码导读视频 (截止2019年8月)

2. 第二季内存管理篇



第二季内存管理篇

第二季旗舰篇课程目录	
课程名称	时长
进程管理初级篇	
课程一：进程管理基本概念 1. 进程的来由 2. 进程控制块 3. Linux 0.11内核task_struct结构剖析 4. Linux 4.x内核的task_struct结构剖析 5. 经典进程状态图和Linux进程状态图 6. init进程 7. 如何获取当前进程的task_struct结构? 8. 实验1：利用top命令来观察进程 9. 实验2：控制CPU使用率	约1小时
课程二：进程创建和终止 1. Linux 0.11内核的fork剖析 2. fork、vfork和clone以及内核线程 3. do_fork函数实现 4. 写时复制技术 5. 僵尸进程和托孤进程 6. 睡眠等待队列 7. 实验1：fork和clone实验 8. 实验2：fork面试题目 9. 实验3：内核线程实验 10. 实验4：睡眠等待队列实验	约1小时
课程三：进程调度和切换 1. 进程调度的本质 2. Linux 0.11内核的调度算法剖析 3. 多级反馈队列算法 4. CFS调度器 5. 进程切换的本质 6. 进程调度时机点 7. 实验1：优先级实验	约1小时
课程四：SMP调度和大小核调度 1. 调度域和调度组 2. Linux调度拓扑图 3. SMP负载均衡 4. 大小核简介 5. EAS调度器 6. 实验1：per-cpu变量实验 7. 实验2：taskset实验 8. 实验3：cgroup实验	约1小时

第二季视频课程目录

进程管理旗舰篇	
课程1: 进程创建代码导读	制作中
课程2: 进程终止代码导读	制作中
课程3: CFS调度器代码导读	制作中
课程4: 组调度代码导读	制作中
课程5: SMP负载均衡代码导读	制作中
课程6: EAS绿色节能调度器代码导读	制作中
课程7: 实时调度代码导读	制作中
课程8: 实时性补丁代码导读	制作中
课程9: 面试宝典 - 那些年我们被虐过的面试题目 (进程)	制作中
课程10: 综合创新实验: 在树莓派上实现一个小OS (进程篇) 1. 汇编启动 2. 进程创建 3. 实现一个简单的进程调度器 4. 完成进程切换功能 5. 支持ARM v7模式 6. 支持ARM v8模式	制作中
中断管理旗舰篇	
课程1: ARM v7 中断底层汇编代码导读	制作中
课程2: ARM v8中断底层汇编代码导读	制作中
课程3: 中断高层/中断线程化处理代码导读	制作中
课程4: 软中断代码导读	制作中
课程5: tasklet代码导读	制作中
课程6: workqueue代码导读	制作中
课程7: ARM v7和ARM v8异常处理底层汇编代码导读	制作中
课程8: 面试宝典 - 那些年我们被虐过的面试题目 (中断篇)	制作中
课程9: 综合创新实验: 在树莓派上实现一个小OS (中断篇) 1. 支持IRQ模式 2. 支持在内核态触发的中断 3. 支持在用户态触发的中断 4. 实现一个简单的中断注册函数	制作中
锁机制旗舰篇	
课程1: spinlock机制代码导读	制作中
课程2: 信号量机制代码导读	制作中
课程3: mutex机制代码导读	制作中
课程4: 读写信号量和读写锁代码导读	制作中
课程5: 锁实际运用例子代码导读	制作中
课程6: 面试宝典 - 那些年我们被虐过的面试题目 (锁机制篇)	制作中
课程7: 综合创新实验: 在树莓派上实现一个小OS (锁机制) 1. 实现简单的spinlock机制 2. 实现简单的信号量机制	制作中
后期课程不定期更新中，等您来提交topic	

第二季内存管理篇

第二季视频课程目录

除了有基础课程之外，我们还录制手把手分析内核代码的代码导读视频。代码导读视频会不断更新。

<input type="checkbox"/> 文件名	修改时间
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part3.vep	2019-07-25 09:46
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part5.vep	2019-08-01 00:48
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part1.vep	2019-07-06 18:04
<input type="checkbox"/> 第二季代码导读3-cfs调度器part2.vep	2019-06-02 10:25
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part2.vep	2019-07-11 14:11
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part6.vep	2019-08-01 00:50
<input type="checkbox"/> 第二季代码导读3-cfs调度器part5.vep	2019-06-08 11:15
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part4.vep	2019-07-25 09:50
<input type="checkbox"/> 第二季代码导读3-cfs调度器part3.vep	2019-06-02 14:00
<input type="checkbox"/> 第二季代码导读3-cfs调度器part4.vep	2019-06-08 11:18
<input type="checkbox"/> 第二季代码导读4-smp负载均衡part7.vep	2019-08-01 00:56
<input type="checkbox"/> 第二季代码导读3-cfs调度器part6.vep	2019-06-08 11:19
<input type="checkbox"/> 第二季代码导读3-cfs调度器part1.vep	2019-05-30 15:03
<input type="checkbox"/> 第二季代码导读1-系统上电后第一个进程怎么来的.vep	2019-05-30 15:03
<input type="checkbox"/> 使用DS-5调试arm64内核.vep	2019-08-26 10:52
<input type="checkbox"/> 第二季代码导读2-fork代码导读.vep	2019-05-30 15:03

第二季代码导读视频 (截止2019年8月)

3. 死机黑屏专题



全程约 5 小时高清，140 多页 ppt，9 大实验案例，基于 x86_64 的 Centos 7.6 和 arm64，提供全套实验素材和环境。全面介绍 kdump+crash 在死机黑屏方面的实战应用，全部案例源自线上云服务器和嵌入式产品开发实际案例！

九大实验案例：

- lab1：简单的空指针引发的 panic
- lab2：访问已经被删除的 list head 链表
- lab3：复杂一点的空指针引发的 panic
- lab4：一个真实的驱动引发的死机
- lab5：一个真实的驱动引发的死锁，导致系统假死
- lab6：如何找到函数调用参数的在栈中地址然后获取具体的值
- lab7：分析一个复杂的线上死锁导致的死机黑屏例子
- lab8：手工恢复函数调用栈 backtrack (arm64)
- 实战案例 9：企业虚拟化计算节点服务器性能问题

Git 专题

4. Git 专题



5. Vim 专题



附录 3 – x86_64 体系结构基础

背景知识: x86_64体系结构的通用寄存器

- 32位x86的通用寄存器: 8个通用寄存器
 - **eax** 一般作为累加器(add)
 - **ebx** 一般作为基地址寄存器(base)
 - **ecx** 一般作为计数寄存器(count)
 - **edx** 一般用来存放数据(data)
 - **esp** 一般作为栈指针寄存器(stack pointer)
 - **ebp** 一般作为基指针寄存器(base pointer)
 - **esi** 一般作为源变址寄存器(source index)
 - **edi** 一般作为目标变址寄存器(destination index)

- x86_64通用寄存器:
- 扩展到16个: %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rsp, %rbp, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15
- 原来的e开头变成r开头,原来通用的e开头寄存器依然可用,表示低32位
- 16个寄存器:
- %rax 作为函数返回值使用。
- %rsp 栈指针寄存器,指向栈顶
- %rdi, %rsi, %rdx, %rcx, %r8, %r9 用作函数参数,依次对应第1参数, 第2参数...
- %rbx, %rbp, %r12, %r13, %r14, %r15 用作通用寄存器, 数据存储
- %r10, %r11 用作通用寄存器

奔跑吧Linux社区出品

Register	Usage	function calls
%eax	temporary register, with variable arguments, passing information about the number of SSE registers used; 1 st return register	No
%ebx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4 th integer argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2 nd argument to functions	No
%rdi	used to pass 1 st argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r15	callee-saved registers	Yes
%xmm0-%xmm15	used to pass and return floating point arguments	No
%xmm2-%xmm7	used to pass floating point arguments	No
%xmm8-%xmm15	temporary registers	No
%mmx0-%mmx7	temporary registers	No
%st0	temporary register; used to return long double arguments	No
%st1-%st7	temporary registers	No
%fs	Reserved for system use (as thread specific data register)	No

1. x86 处理器寄存器组成

Vim 专题

x86 体系结构支持 32 位和 64 位处理器，它们在通用寄存器方面略有不同。32 位的 x86 处理器支持 8 个通用寄存器。

表 32 位 x86 处理器通用寄存器

寄存器	描述
EAX	操作数的运算、结果
EBX	指向DS段中的数据的指针
ECX	字符串操作或者循环计数器
EDX	输入输出指针
ESI	指向DS寄存器所指示的段中某个数据的指针，字符串操作中的复制源
EDI	指向ES寄存器所指示的段中某个数据的指针，字符串操作中的目的地
ESP	栈指针寄存器（stack pointer）
EBP	指向栈上数据的指针（base pointer）

除了 8 个通用寄存器外，还有 16 位段寄存器 6 个，32 位 EFLAGS 寄存器 1 个以及 32 位 EIP 寄存器一个，因此 32 位的 x86 处理器的寄存器有如下组成：

- 8 个 32 位通用寄存器
- 6 个 16 位段寄存器
- 1 个 EFLAGS 寄存器
- 1 个 EIP 寄存器

64 位的 x86 处理器的通用寄存器扩展到 16 个。原来以“E”字母命名的寄存器变成了“R”字母来命名。原来“E”字母的寄存器依然可以使用，表示低 32 位的寄存器。“R”字母的寄存器表示 64 位。另外 R8 ~ R15 是新增的通用寄存器。

64 位 x86 处理器的寄存器由如下组成：

- 16 个 64 位通用寄存器
- RIP 指令指针寄存器
- 64 位 RFLAGS 寄存器
- 6 个 128 位 XMM 寄存器

2. x86_64 体系结构函数参数调用关系

x86 和 x86_64 体系结构在函数参数调用关系上是有很多不同的地方。本章重点介绍 x86_64 体系结构，函数参数是如何传递和调用的。

当函数参数小于等于 6 个的时候，使用通用寄存器来传递形参。当函数参数大于 6 个时候，采用栈空间来传递形参^①。

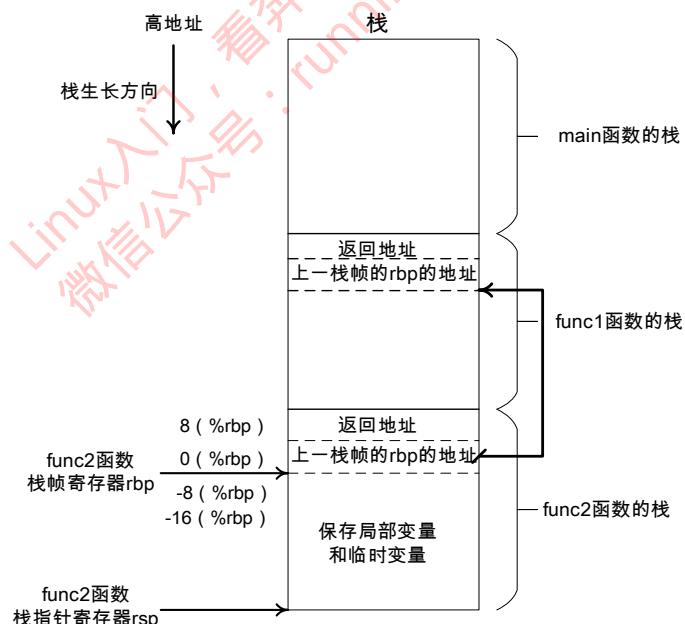
^① 参见< System V Application Binary Interface - AMD64 Architecture Processor Supplement v0.99.6> 图 3-4.
http://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

表 x86_64 体系结构寄存器

寄存器	用途
rdi	传递第一个函数参数
rsi	传递第二个函数参数
rdx	传递第三个函数参数或者第二个函数返回值
rcx	传递第四个函数参数
r8	传递第五个函数参数
r9	传递第6个函数参数
Rax	临时寄存器或者第一个函数返回值
rsp	栈指针寄存器
rbp	栈基址寄存器

3. 栈结构的布局图

函数的调用与栈有着密切的联系。程序的执行通常是一个函数嵌套着下一个函数，无论嵌套有多深，程序总能正确的返回到原点，这个就要依赖于栈的结构、rsp 栈指针寄存器以及 rbp 栈基址寄存器。假设函数调用关系如下：main() -> func1() -> func2()，那么栈的布局示意图如下图所示。



x86_64 栈布局示意图

附录 4 – arm64 体系结构基础

1 ARMv8-A 架构介绍

ARMv8-A 是 ARM 公司发布的第一代支持 64 位处理器的指令集和体系结构。它可以同时提供运行 32 位和 64 位应用程序的执行环境，因此它扩充了 64 位寄存器的同时提供了对上一代架构指令集的兼容。

ARMv8-A 架构引入了很多吸引人的新特性，从而提供了处理器的处理能力。

- 超大物理地址空间 (Large physical address)。可以提供超过 4GB 的物理内存的访问。
- 64 位宽的虚拟地址空间 (64-bit virtual addressing)。在 32 位处理器中只能提供 4GB 大小的虚拟地址空间访问，这极大限制了桌面系统和服务器等应用的发挥。64 位宽的虚拟地址空间可以提供超大的访问空间。
- 提供 31 个 64 位的通用寄存器，可以减少对栈的访问，从而提高性能。
- 提供 16KB 和 64KB 的页面转换，有助于减低 TLB 的未命中率 (miss rate)。
- 全新的异常处理模式，有助于降低操作系统和虚拟化的实现复杂度。
- 全新的加载-获取，存储-释放指令 (Load-Acquire, Store-Release instructions)。专门为 C++11, C11 以及 Java 内存模型设计。

2 常见的 ARMv8 处理器

下面介绍市面上常见的 ARMv8 架构的处理器。

(1) Cortex-A53 处理器

Cortex-A53 是 ARM 公司第一款采用 ARMv8-A 架构的设计的处理器，专门为低功耗设计的处理器。通常可以使用 1~4 个 Cortex-A53 处理器组成一个处理器族 (cluster) 或者和 Cortex-A57 或者 Cortex-A72 等高性能处理器组成大小核架构的 SoC 芯片。

(2) Cortex-A57 处理器

(3) Cortex-A72 处理器

2015 年年初正式发布的基于 ARMv8-A 架构、并在 Cortex-A57 处理器上做了很大优化和改进的一款处理器。在相同的移动设备电池寿命限制下，Cortex-A72 能相较基于 Cortex-A15 的设备提供 3.5 倍的性能表现，展现优异的整体功耗效率。

3 ARM64 基本概念

在 ARM 公司的官方技术手册中提到一个概念，把处理器处理事务的过程抽象为处理机（processing element），简称 PE。

ARM 处理器实现的是精简指令集架构（RISC）。

(1) 执行模式 (Execution state)

执行模式是处理器运行时的环境，包括寄存器的位宽，支持的指令集，异常模型，内存管理以及编程模型等。ARMv8 架构定义了两个执行模式。

- AArch64: 64 位的执行环境。
 - ✧ 提供 31 个 64 位的通用寄存器。
 - ✧ 提供 64 位的程序计数寄存器 PC、栈指针寄存器 SP 以及异常链接寄存器 ELR。
 - ✧ 提供 64 位的指令集。
 - ✧ 定义 ARMv8 异常模型，支持 4 个异常等级，EL0 ~ EL3。
 - ✧ 提供 64 位的内存模型。
 - ✧ 定义一组处理器状态 (PSTATE) 用来保存 PE 的状态。
- AArch32: 32 位的执行环境。
 - ✧ 提供 13 个 32 位的通用寄存器，程序计数寄存器 PC，栈指针寄存器，链接寄存器 LR。
 - ✧ 支持两套指令集，分别是 A32 和 T32 指令集。
 - ✧ 支持 ARMv7-A 异常模型，基于 PE 模式并映射到 ARMv8 的异常模型中。
 - ✧ 提供 32 位的虚拟内存访问机制。
 - ✧ 定义一组处理器状态 (PSTATE) 用来保存 PE 的状态。

(2) ARMv8 指令集

ARMv8 架构根据不同的执行模式提供不同指令集的支持。

- A64 指令集：运行在 AArch64 模式，提供 64 位指令集支持。
- A32 指令集：运行在 AArch32 模式，提供 32 位指令集支持。
- T32 指令集：运行在 AArch32 模式，提供 16 和 32 位指令集支持。

(3) 系统寄存器命名

在 AArch64 模式下，很多系统寄存器会根据不同的异常等级提供不同的变种

4 ARMv8 处理器运行模式

寄存器。

| <register_name>_ELx, where x is 0, 1, 2, or 3

比如 SP_EL0 表示在 EL0 下的栈指针寄存器, SP_EL1 表示在 EL1 下的栈指针寄存器。

4 ARMv8 处理器运行模式

ARMv8 架构处理器支持两种运行模式 (Execution States): AArch64 模式和 AArch32 模式。AArch64 模式是 ARMv8 新增的 64 位运行模式, 而 AArch32 是为了兼容 ARMv7 架构的 32 位运行模式。当处理器运行在 AArch64 模式时执行 A64 指令集; 而当运行在 AArch32 模式时, 可以运行 A32 指令集或者 T32 (Thumb 指令集) 指令集。

AArch64 架构的异常等级 (Exception Levels) 确定其运行特权级别, 类似 ARMv7 架构中特权等级, 如图所示。

- EL0: 用户特权, 用于运行普通用户程序。
- EL1: 系统特权, 通常用于运行操作系统。
- EL2: 运行虚拟化扩展的 Hypervisor。
- EL3: 运行安全世界中的 Secure Monitor。



在 ARMv8 架构里允许切换应用程序的运行模式。比如, 在一个运行 64 位操作系统的 ARMv8 处理器中, 我们可以同时运行 A64 指令集的应用程序和 A32 指令集的应用程序。但是在一个运行在 32 位的操作系统的 ARMv8 处理器中就不能执行 A64 指令集的应用程序了。当需要执行一个 A32 指令集的应用程序时, 需要通过一个管理

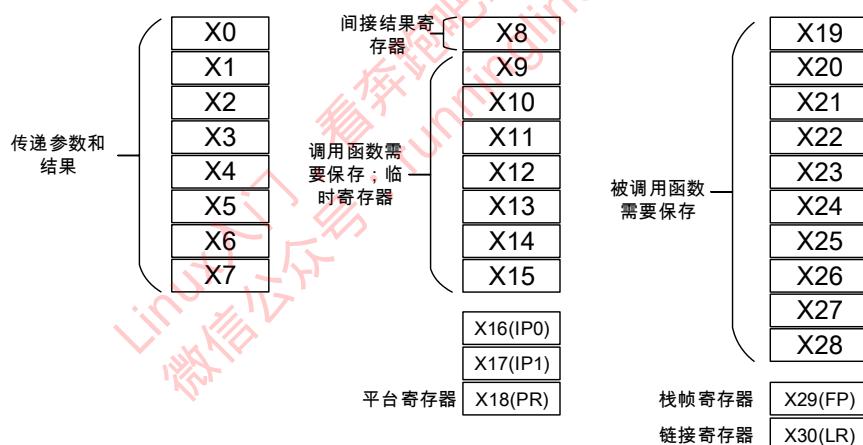
员调用指令 (Supervisor Call, SVC 指令) 切换到 EL1 等级, 操作系统会做任务的切换并且返回到 AArch64 的 EL0 等级中, 这时候系统就为这个应用程序准备好了 AArch32 的运行环境。

附录 5 – arm64 寄存器

1. 通用寄存器

AArch64 运行模式支持 31 个 64 位的通用寄存器, 分别是 x0~x30 寄存器, 而 AArch32 运行模式支持 16 个 32 位的通用寄存器。

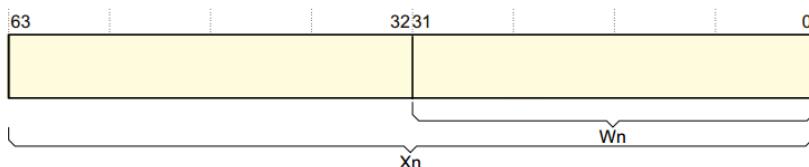
通用寄存器除了做普通寄存器用于数据的运算和存储之外, 还可以在函数调用过程中起到特殊的作用, 这是 ARM64 架构函数调用的标准和规范约定的, 如图所示。



AArch64的31个通用寄存器

在 AArch64 运行模式下, 使用 X 来表示 64 位通用寄存器, 比如 X0, X30 等。另外还可以使用 W 来表示低 32 位的寄存器, 比如 W0 表示 X0 寄存器的低 32 位数据, W1 表示 X1 寄存器的低 32 位数据, 如图所示。

处理器状态 (Processor State)



64位通用寄存器和32位通用寄存器

2. 处理器状态 (Processor State)

在 ARMv7 架构中使用 CPSR 寄存器来表示当前处理器的状态，而在 AArch64 里使用处理器状态寄存器来表示，简称 PSTATE。

表 PSTATE 处理器状态

分类	字段	描述
条件标志位	N	结果为负数
	Z	结果为0
	C	完成
	V	溢出
执行状态控制	SS	软件单步。该比特位为1，说明在异常处理时使能了软件单步功能。
	IL	不合法的异常状态。
	nRW	当前执行模式。 0: 处于AArch64执行模式 1: 处于AArch32执行模式
	EL	当前异常等级。 0: 表示EL0 1: 表示EL1 2: 表示EL2 3: 表示EL3
	SP	选择栈指针寄存器。当运行在EL0时，处理器选择EL0的栈指针，SP_EL0; 当处理器运行在其他异常等级时，处理器可以选择使用SP_EL0或者对应的SP_ELn寄存器。
异常掩码标志位	D	调试比特位。使能该比特位可以在异常处理过程中打开调试断点和软件单步等功能。
	A	用来屏蔽系统错误 (SError)。
	I	用来屏蔽IRQ中断。
	F	用来屏蔽FIQ中断。

访问权限	PAN	特性不访问 (Privileged Access Never) 比特位。ARMv8.1扩展特性。 1: 在EL1或者EL2访问属于EL0的虚拟地址时会触发一个访问权限错误。 0: 不支持该功能, 需要软件来模拟。
	UAO	用户特权访问覆盖标志位。ARMv8.2扩展特性。 1: 当在EL1或者EL2时, 没有特权的加载存储指令可以和有特权的加载存储指令一样访问内存, 比如LDTR指令。 0: 不支持该功能。

3. 特殊寄存器

ARMv8 架构除了支持 31 个通用寄存器之外, 还提供多个特殊的寄存器。

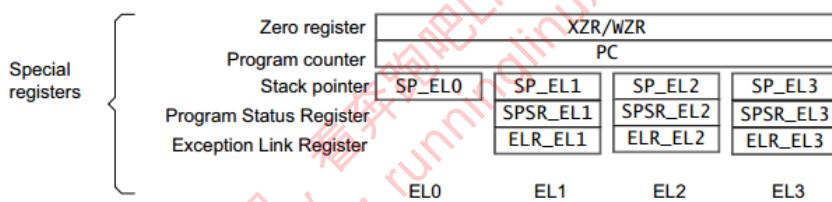


Figure 4-3 AArch64 special registers

1. 零寄存器 (Zero Register)

ARMv8 架构提供两个零寄存器, 这些寄存器的内容全是 0, 可以用作源寄存器, 也可以当作目标寄存器。WZR 寄存器是 32 位的零寄存器, XZR 是 64 位的零寄存器。

2. 栈指针 (Stack Pointer)

ARMv8 架构支持 4 个异常等级, 每一个异常等级有一个专门的栈指针 SP_ELn, 比如处理器运行在 EL1 异常等级时会选择 SP_EL1 寄存器作为栈指针。

当处理器运行在比 EL0 高级别的异常等级时, 处理器可以访问:

- 异常等级对应的栈指针 SP_ELn。
- EL0 对应的栈指针 SP_EL0 寄存器可以当做一个临时寄存器, 比如 Linux 内核里使用该寄存器里存放进程的内核栈地址。

特殊寄存器

当处理器运行在 EL0 异常等级时，它只能访问 SP_EL0，而不能访问其他高等级的 SP 寄存器。

- SP_EL0: EL0 下的栈指针寄存器。
- SP_EL1: EL1 下的栈指针寄存器。
- SP_EL2: EL2 下的栈指针寄存器。
- SP_EL3: EL3 下的栈指针寄存器。

3. PC 指针 (Program Counter)

通常用来指向当前运行指令的下一条指令的指针。

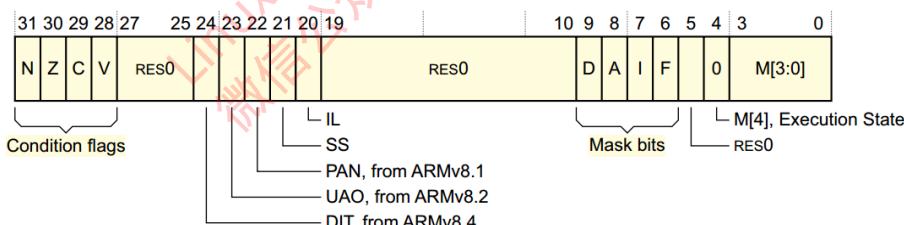
4. 异常链接寄存器 (Exception Link Register, ELR)

异常链接寄存器存放了异常返回地址。

- ELR_EL1 表示保存了从异常返回 EL1 的地址。
- ELR_EL2 表示保存了从异常返回 EL2 的地址。
- ELR_EL3 表示保存了从异常返回 EL3 的地址。

5. 保存处理状态寄存器 (Saved Process Status Register)

当我们执行一个异常处理时，处理器的处理状态会保存到 SPSR 寄存器里，这个寄存器非常类似 ARMv7 架构中的 CPSR 寄存器。当异常将要发生时，处理器会把处理状态寄存器的值暂时保存到 SPSR 寄存器里；当异常处理完成并返回时，再把 SPSR 寄存器的值恢复到处理器状态寄存器。



SPSR 寄存器

表 SPSR 寄存器

字段	描述
N	结果为负数
Z	结果为0
C	完成
V	溢出
SS	软件单步。该比特位为1，说明在异常处理时使能了软件单步功能。

IL	不合法的异常状态。
D	调试比特位。使能该比特位可以在异常处理过程中打开调试断点和软件单步等功能。
A	用来屏蔽系统错误 (SError)。
I	用来屏蔽IRQ中断。
F	用来屏蔽FIQ中断。
M[4]	用来表示异常处理过程中处于哪个运行模式，为0表示AArch64。
M[3:0]	异常模式。

6. CurrentEL 寄存器^①

处理器状态 PSTATE 中的 EL 字段保存了当前异常等级。使用 MRS 指令可以读取当前异常等级。

0: 表示 EL0

1: 表示 EL1

2: 表示 EL2

3: 表示 EL3

7. DAIF 寄存器

表示处理器状态 PSTATE 中的 {D, A, I, F} 字段。

8. SPSel 寄存器

表示处理器状态 PSTATE 中的 SP 字段，用来在 SP_EL0 和 SP_ELn 中选择栈指针寄存器。

9. PAN 寄存器

用来表示处理器状态 PSTATE 中的 PAN 字段。可以通过 MSR 和 MRS 指令来设置 PAN 寄存器。

10. UAO 寄存器

用来表示处理器状态 PSTATE 中的 UAO 字段。可以通过 MSR 和 MRS 指令来设置 UAO 寄存器。

^①详见<ARM Architecture Reference Manual, for ARMv8-A architecture profile, v8.4>第 C5.2 章

系统寄存器

4. 系统寄存器

除了上面介绍的通用寄存器和特殊寄存器之外，ARMv8 架构还定义了很多的系统寄存器，通过这些系统寄存器来完成对处理器不同的功能配置。在 ARMv7 架构里，我们需要通过访问 CP15 协处理器来间接访问这些系统寄存器，而在 ARMv8 架构中不需要协处理器，可直接访问到系统寄存器。

系统寄存器支持不同的异常等级的访问，通常系统寄存器会使用“Reg_ELn”的方式来表示。比如，

- Reg_EL1：处理器处于 EL1，EL2 以及 EL3 时可以访问该寄存器。
- Reg_EL2：处理器处于 EL2 和 EL3 时可以访问该寄存器。
- 大部分系统寄存器不支持处理器处于 EL0 时访问，但也有一些例外，比如 CTR_EL0 寄存器。

程序可以通过 MSR 和 MRS 指令访问系统寄存器，比如：

```
mrs x0, TTBR0_EL1 //把TTBR0_EL1寄存器的值拷贝到x0寄存器  
msr TTBR0_EL1, x0 //把x0寄存器的值拷贝到TTBR0_EL1寄存器
```

附录 6 – 常见 crash 命令

crash 工具支持大约 50 个子命令。

读者在使用 crash 工具进行内核分析之前需要熟悉常用的几个子命令，下面对常用的命令的使用做简单地介绍。

1. help 命令。在线查看 crash 命令的帮助。比如在 crash 命令行中输入 help 命令可以查看支持的子命令。

```
crash> help
*          extend      log        rd        task
alias      files       mach      repeat    timer
ascii      foreach     mod       runq      tree
bpf       fuser       mount     search    union
bt        gdb         net       set       vm
bttop     help        p         sig      vtop
dev       ipcs       ps        struct   waitq
dis       irq        pte      swap     whatis
eval      kmem      ptob     sym      wr
exit      list       ptov     sys      q
```

还有一个比较有用的技巧是使用 help 命令在查看具体某个子命令的参数选项，比如查看 bt 命令。

奔跑吧 linux 社区出品

```
crash> help bt

NAME
  bt - backtrace

SYNOPSIS
  bt [-a|-c cpu(s)|-g|-r|-t|-T|-l|-e|-E|-f|-F|-o|-O|-v] [-R ref] [-s [-x|d]]
      [-I ip] [-S sp] [pid | task]

DESCRIPTION
  Display a kernel stack backtrace. If no arguments are given, the stack
  trace of the current context will be displayed.

  -a displays the stack traces of the active task on each CPU.
  (only applicable to crash dumps)
  -A same as -a, but also displays vector registers (S390X only).
```

2. bt 命令。输出一个进程内核栈的函数调用关系 (backtrace)，包括所有异常栈的信息。

```
crash> bt
PID: 2653  TASK: fffff9107158e8000  CPU: 0  COMMAND: "bash"
#0 [fffff910701907ae0] machine_kexec at ffffffff8a663674
#1 [fffff910701907b40] __crash_kexec at ffffffff8a71cef2
#2 [fffff910701907c10] crash_kexec at ffffffff8a71cfe0
#3 [fffff910701907c28] oops_end at ffffffff8ad6c758
#4 [fffff910701907c50] no_context at ffffffff8ad5aafe
#5 [fffff910701907ca0] __bad_area_nosemaphore at ffffffff8ad5ab95
#6 [fffff910701907cf0] bad_area_nosemaphore at ffffffff8ad5ad06
#7 [fffff910701907d00] __do_page_fault at ffffffff8ad6f6b0
#8 [fffff910701907d70] do_page_fault at ffffffff8ad6f915
#9 [fffff910701907da0] page_fault at ffffffff8ad6b758
[exception RIP: sysrq_handle_crash+22]
RIP: ffffffff8aa61e66 RSP: fffff910701907e58 RFLAGS: 00010246
RAX: ffffffff8aa61e50 RBX: ffffffff8b2e4c60 RCX: 0000000000000000
RDX: 0000000000000000 RSI: fffff91077b613898 RDI: 0000000000000063
RBP: fffff910701907e58 R8: ffffffff8b5e38bc R9: 6873617263206120
R10: 000000000000072d R11: 0000000000000072c R12: 0000000000000063
R13: 0000000000000000 R14: 0000000000000007 R15: 0000000000000000
ORIG_RAX: ffffffff8aa6268d CS: 0010 SS: 0018
#10 [fffff910701907e60] __handle_sysrq at ffffffff8aa6268d
#11 [fffff910701907e90] write_sysrq_trigger at ffffffff8aa62af8
#12 [fffff910701907ea8] proc_req_write at ffffffff8a8b81a0
#13 [fffff910701907ec8] vfs_write at ffffffff8a841310
#14 [fffff910701907f08] sys_write at ffffffff8a84212f
#15 [fffff910701907f50] system_call_fastpath at ffffffff8ad74ddb
```

上面的例子是把系统在崩溃瞬间正在运行的进行的内核栈信息全部显示出来，当前进程的 PID 是 2653，进程的进程控制块 task_struct 数据结构的地址是 fffff9107158e8000，当前运行在 CPU 0 上，当前进程的运行命令是 bash。接着，列出了该进程在内核态的函数调用关系图，执行顺序是从下往上，也就是从 system_call_fastpath 函数一路执行到 machine_kexec 函数。在第 9 个执行函数里，显示了发生崩溃的函数地址：sysrq_handle_crash+22，并且打印出发生崩溃瞬间，CPU 的通用寄存器的值，这些信息对于后续分析是很有帮助的。

其他常用的参数选项：

系统寄存器

- -t 参数选项显示栈中所有的文本符号 (text symbol)。
- -f 参数选项显示每一栈帧里的数据。
- -l 参数选项显示文件名和行号。
- pid 参数选项可以用来显示指定 pid 号的进程的内核栈函数调用信息。

3. dis 命令, 用来输出反汇编。比如输出 sysrq_handle_crash 函数的反汇编。

```
crash> dis sysrq_handle_crash
0xffffffff8aa61e50 <sysrq_handle_crash>:      nopl    0x0(%rax,%rax,1)
[FTRACE NOE]
0xffffffff8aa61e55 <sysrq_handle_crash+5>:      push    %rbp
0xffffffff8aa61e56 <sysrq_handle_crash+6>:      mov     %rsp,%rbp
0xffffffff8aa61e59 <sysrq_handle_crash+9>:      movl    $0x1,0x7e54b1(%rip)
# 0xffffffff8b247314
0xffffffff8aa61e63 <sysrq_handle_crash+19>:      sfence
0xffffffff8aa61e66 <sysrq_handle_crash+22>:      movb    $0x1,0x0
0xffffffff8aa61e6e <sysrq_handle_crash+30>:      pop    %rbp
0xffffffff8aa61e6f <sysrq_handle_crash+31>:      retq
crash>
```

其他常用的参数选项:

- -l 选项: 显示反汇编以及对应源代码的行号。
- -s 选项: 显示对应的源代码。

4. mod 命令, 用来显示当前系统加载的内核模块信息, 也可以用来加载某个内核模块的符号信息和调试信息等。

```
crash> mod
      MODULE      NAME      SIZE   OBJECT FILE
e080d000  jbd       57016  (not loaded)  [CONFIG_KALLSYMS]
e081e000  ext3      92360  (not loaded)  [CONFIG_KALLSYMS]
e0838000  usbcore    83168  (not loaded)  [CONFIG_KALLSYMS]
e0850000  usb-uhci   27532  (not loaded)  [CONFIG_KALLSYMS]
e085a000  ehci-hcd   20904  (not loaded)  [CONFIG_KALLSYMS]
e0865000  input      6208   (not loaded)  [CONFIG_KALLSYMS]
e086a000  hid        22404  (not loaded)  [CONFIG_KALLSYMS]
e0873000  mousedev   5688   (not loaded)  [CONFIG_KALLSYMS]
```

常用参数选项:

- -s 选项: 加载某个内核模块的符号信息。
- -S 选项: 从某个特定目录加载所有内核模块的符号信息。默认会从 /lib/modules/<release> 目录加载。
- -d 选项: 删除某个内核模块的符号信息。

例如加载名为 oops 的内核模块的符号信息。

```
crash> mod -s oops /home/benshushu/crash/crash_lab_centos/01_oops/oops.ko
      MODULE      NAME      SIZE   OBJECT FILE
ffffffffffc0732000  oops      12741
/home/benshushu/crash/crash_lab_centos/01_oops/oops.ko
```

5. **sym** 命令，用来解析符号（symbol）。

常用的参数选项：

-l 选项：显示所有符号信息，等同于查看 System.map 文件。

-m 选项：显示某个内核模块的所有符号信息。

-q 选项：查询符号信息。

比如查看名为 oops 的内核模块的所有符号信息。

```
crash> sym -m oops
fffffffffc0730000 MODULE START: oops
fffffffffc0730000 (T) create_oops
fffffffffc0730044 (T) cleanup_module
fffffffffc0730044 (T) my_oops_exit
fffffffffc0731028 (r) vvaraddr_jiffies
fffffffffc0731030 (r) vvaraddr_vgetcpu_mode
fffffffffc0731038 (r) vvaraddr_vsyscall_gtod_data
fffffffffc0731068 (r) vvaraddr_jiffies
fffffffffc0731070 (r) vvaraddr_vgetcpu_mode
fffffffffc0731078 (r) vvaraddr_vsyscall_gtod_data
fffffffffc0732000 (D) __this_module
fffffffffc07331c5 MODULE END: oops
fffffffffc0735000 MODULE INIT START: oops
fffffffffc0735000 (t) my_oops_init
fffffffffc0735000 (t) init_module
fffffffffc07365f8 MODULE INIT END: oops
crash>
```

查询 `create_oops` 符号：

```
crash> sym -q create_oops
fffffffffc0730000 (T) create_oops [oops]
crash>
```

6. **rd** 命令，用来读取内存地址。

常用参数选项：

- **-p**: 读取物理地址
- **-u**: 读取用户空间虚拟地址
- **-d**: 显示十进制
- **-s**: 显示符号
- **-32**: 显示 32 位宽的值
- **-64**: 显示 64 位宽的值
- **-a**: 显示 ASCII 码

比如读地址为 0xffffffffc0730000 内存的值，并且连续读取和打印 20 个内存地址的值。

```
crash> rd ffffffc0730000 20
fffffffffc0730000: 8948550000441f0f 7d894818ec8348e5 ..D..UH..H..H..}
fffffffffc0730010: 458b48e8758948f0 45894850408b48f0 ..H..U..H..E..H..@PH..E
fffffffffc0730020: 458b48e8558b48f8 40c7c748c68948f8 ..H..U..H..E..H..@...
fffffffffc0730030: 00000000b8c07310 0000b8dde2b64ae8 ..S.....J.....
fffffffffc0730040: e5894855c3c90000 b8c073105bc7c748 ....UH..H..[.S..
```

系统寄存器

```
ffffffffffc0730050: e2b62ee800000000 00000000000c35ddd .....].....
ffffffffffc0730060: 0000000000000000 0000000000000000 .....].
ffffffffffc0730070: 0000000000000000 0000000000000000 .....].
ffffffffffc0730080: 0000000000000000 0000000000000000 .....].
ffffffffffc0730090: 0000000000000000 0000000000000000 .....].
```

7. struct 命令，用来显示内核里数据结构的定义或者具体的值。

常用的参数选项：

- struct_name：内核数据结构名称。
- .member：数据结构的某个成员。
- -o：显示成员在数据结构里的偏移值。

比如显示 vm_area_struct 数据结构的定义。

```
crash> struct vm_area_struct
struct vm_area_struct {
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    struct vm_area_struct *vm_prev;
    struct rb_node vm_rb;
    unsigned long rb_subtree_gap;
    struct mm_struct *vm_mm;
    ...
}
```

比如显示 vm_area_struct 数据结构中每个成员的偏移值。

```
crash> struct vm_area_struct -o
struct vm_area_struct {
[0] unsigned long vm_start;
[8] unsigned long vm_end;
[16] struct vm_area_struct *vm_next;
[24] struct vm_area_struct *vm_prev;
[32] struct rb_node vm_rb;
[56] unsigned long rb_subtree_gap;
[64] struct mm_struct *vm_mm;
[72] pgprot_t vm_page_prot;
}
```

另外 struct 命令后面还可以指定一个地址，用来按照数据结构的格式来显示每个成员的值，这个技巧在实际应用中非常有用。例如，已知 mydev_priv 数据结构存放在地址 0xffff946bd63bbce4，那么可以通过 struct 命令来查看该数据结构每个成员的值。

```
crash> struct mydev_priv ffff946bd63bbce4
struct mydev_priv {
    name = "figo\000\",
    i = 10
}
```

8. p 命令，用来打印内核变量、表达式或者符号的值。

比如打印 jiffies 的值。

```
crash> p jiffies
jiffies = $1 = 4295209831
```

```
| crash>
```

打印进程 0 的 mm 数据结构的值 init_mm。

```
crash> p init_mm
init_mm = $3 = {
    mmap = 0x0,
    mm_rb = {
        rb_node = 0x0
    },
    mmap_cache = 0x0,
    get_unmapped_area = 0x0,
    unmap_area = 0x0,
    mmap_base = 0,
    mmap_legacy_base = 0,
    ...
}
```

如果一个变量是 pre-cpu 类型的变量，那么会打印所有 pre-cpu 的地址。

```
crash> p irq_stat
PER-CPU DATA TYPE:
irq_cpustat_t irq_stat;
PER-CPU ADDRESSES:
[0]: fffff946c3b61a1c0
[1]: fffff946c3b65a1c0
[2]: fffff946c3b69a1c0
[3]: fffff946c3b6da1c0
crash>
```

比如要打印某个 CPU 上该变量的值，可以在变量后面指定 CPU 号。下面要打印 CPU 0 上的 irq_stat 数据结构的值。

```
crash> p irq_stat:0
per_cpu(irq_stat, 0) = $4 = {
    __softirq_pending = 0,
    __nmi_count = 1,
    apic_timer_irqs = 66867,
    irq_spurious_count = 0,
    icr_read_retry_count = 0,
    kvm_posted_intr_ipis = 0,
    x86_platform_ipis = 0,
    apic_perf_irqs = 0,
    apic_irq_work_irqs = 2001,
    irq_resched_count = 22511,
    irq_call_count = 3405,
    irq_tlb_count = 880,
    irq_thermal_count = 0,
    irq_threshold_count = 0
}
crash>
```

9. irq 命令，显示中断相关信息。

常用的参数选项：

- **index:** 显示某个指定 IRQ 中断的信息。
- **-b:** 显示中断下半部信息
- **-a:** 显示中断亲和性。
- **-s:** 显示系统中断信息。

系统寄存器

```
crash> irq
IRQ  IRQ_DESC/_DATA      IRQACTION      NAME
0   fffff946c3f978000  ffffffff9ea2d440  "timer"
1   fffff946c3f978100  fffff946c39cd9200  "i8042"
2   fffff946c3f978200  (unused)
3   fffff946c3f978300  (unused)
4   fffff946c3f978400  (unused)
5   fffff946c3f978500  (unused)
6   fffff946c3f978600  (unused)
7   fffff946c3f978700  (unused)
8   fffff946c3f978800  fffff946c39d6c580  "rtc0"
9   fffff946c3f978900  fffff946c3a3fbb00  "acpi"
```

10. task 命令，显示进程的 task_struct 数据结构以及 thread_info 数据结构的内容。

```
crash> task -x
PID: 4404  TASK: fffff946c35f51040  CPU: 3  COMMAND: "insmod"
struct task_struct {
    state = 0x0,
    stack = 0xfffff946bd63b8000,
    usage = {
        counter = 0x2
    },
    flags = 0x402100,
    ptrace = 0x0,
    wake_entry = {
        next = 0x0
    },
    on_cpu = 0x1,
    last_wakee = 0xfffff946bf5bbc100,
    wakee_flips = 0x3,
    wakee_flip_decay_ts = 0x10003b19a,
    wake_cpu = 0x3,
    ...
}
```

11. vm 命令，用来显示进程地址空间的相关信息。

常用参数选项：

- -p: 显示虚拟地址和物理地址
- -m: 显示 mm_struct 数据结构
- -R: 搜索特定字符串或者数值
- -v: 显示该进程所有 vm_area_struct 数据结构的值
- -f num: 数字 num 在 vm_flags 中对应的比特位

显示当前进程虚拟地址空间信息。

```
crash> vm
PID: 4404  TASK: fffff946c35f51040  CPU: 3  COMMAND: "insmod"
          MM           PGD      RSS      TOTAL VM
fffff946c0bd81900  fffff946bfd97e000  808k  13244k
          VMA          START      END      FLAGS FILE
fffff946bd630e798  400000    423000 8000875 /usr/bin/kmod
fffff946bd630f5f0  622000    623000 8100871 /usr/bin/kmod
fffff946bd630f950  623000    624000 8100873 /usr/bin/kmod
fffff946bd630e000  1259000   127a000 8100073
fffff946bd630f440  7fa800b3b000  7fa800b52000 8000075 /usr/lib64/libpthread-2.17.so
```

...

显示某个进程的所有的虚拟地址信息，包括了虚拟地址到物理地址的转换信息。
比如显示 pid 号 4159 的虚拟地址空间信息。

```
crash> vm -p 4159
PID: 4159  TASK: fffff946c362630c0  CPU: 1  COMMAND: "gdbus"
      MM          PGD          RSS   TOTAL VM
fffff946c0bd812c0  fffff946bd6378000  6740k  345872k
      VMA          START         END    FLAGS FILE
fffff946c3693a438  5583c095d000  5583c0964000  8000875 /usr/sbin/abrt-dbus
VIRTUAL      PHYSICAL
5583c095d000  5eca7000
5583c095e000  705ce000
5583c095f000  5eca5000
5583c0960000  5eca4000
5583c0961000  59044000
5583c0962000  5eca2000
5583c0963000  6c627000
...
...
```

显示 pid 号为 4159 的进程 mm_struct 的内容。

```
crash> vm -m 4159
PID: 4159  TASK: fffff946c362630c0  CPU: 1  COMMAND: "gdbus"
struct mm_struct {
    mmap = 0xfffff946c3693a438,
    mm_rb = {
        rb_node = 0xfffff946bfd9c2890
    },
    mmap_cache = 0x0,
    get_unmapped_area = 0xfffffffff9de30e90,
...
...
```

显示 pid 号为 4159 的进程所有 vm_area_struct 的内容。

```
crash> vm -v 4159
PID: 4159  TASK: fffff946c362630c0  CPU: 1  COMMAND: "gdbus"
struct vm_area_struct {
    vm_start = 94024360120320,
    vm_end = 94024360148992,
    vm_next = 0xfffff946c3693b290,
    vm_prev = 0x0,
...
...
```

12. kmem 命令，显示系统内存信息。

常用参数选项：

- -i: 显示系统内存使用情况。
- -v: 显示系统 vmalloc 使用情况。
- -V: 显示系统 vm_stat 情况。
- -z: 显示每个 zone 的情况。
- -s: 显示 slab 使用情况。
- -p: 显示每个页面的情况。
- -g: 显示 struct page 结构里 flags 的标志位。

系统寄存器

显示系统内存使用情况。

```
crash> kmem -i
      PAGES      TOTAL      PERCENTAGE
TOTAL MEM  404629    1.5 GB      -----
   FREE    105735    413 MB    26% of TOTAL MEM
   USED   298894    1.1 GB    73% of TOTAL MEM
  SHARED   18610    72.7 MB    4% of TOTAL MEM
BUFFERS      0        0        0% of TOTAL MEM
 CACHED  85565    334.2 MB   21% of TOTAL MEM
   SLAB   19500    76.2 MB    4% of TOTAL MEM

TOTAL HUGE      0        0        -----
HUGE FREE      0        0        0% of TOTAL HUGE

TOTAL SWAP  524287    2 GB      -----
 SWAP USED   194     776 KB    0% of TOTAL SWAP
 SWAP FREE  524093    2 GB    99% of TOTAL SWAP

COMMIT LIMIT  726601    2.8 GB      -----
COMMITTED    726661    2.8 GB   100% of TOTAL LIMIT
crash>
```

显示系统 slab 使用情况

```
crash> kmem -s
CACHE          NAME           OBJSIZE ALLOCATED      TOTAL  SLABS  SSIZE
ffff946c39840b00 fuse_inode      728       1        42      1    32k
ffff946c39931c00 hgfsInodeCache  640       1        46      1    32k
...
...
```

13. list 命令，用来遍历链表，并且可以打印链表成员的值。

附录 7 – ARM64 指令集

指令集的设计是处理器架构设计的重点之一。不同体系结构的设计会有不同的指令集架构 (ISA)。ARM 公司定义和实现的指令集架构一直在变化和发展中。ARMv8 架构最大的改变是增加了一个新的 64 位的指令集，这是早前 ARM 指令集的一个有益补充和增强。它可以处理 64 位宽的寄存器和数据处理并且使用 64 位的指针来访问

内存。这个新的指令集称为 A64 指令集，运行在 AArch64 运行环境。ARMv8 兼容老的 32 位指令集，我们称为 A32 指令集，运行在 AArch32 运行环境。

A64 指令集和 A32 指令集是不兼容的，它们是两套完全不一样的指令集架构，它们的指令编码是不一样。需要注意的是，A64 指令集的指令是 32 位长度，而不是 64 位宽。

指令格式说明：

- Xd : 目标寄存器，64 位宽。
- Xn : 第一个源寄存器，64 位宽。
- Xn : 第二个源寄存器，64 位宽。
- Xa : 第三个源寄存器，64 位宽。
- SP : 栈指针寄存器，64 位宽。
- imm : 立即数。
- $shift$: 移位操作的数量。
- WSP : 栈指针寄存器，32 位宽。
- Wd : 目标寄存器，64 位宽。
- Wn : 第一个源寄存器，32 位宽。
- Wn : 第二个源寄存器，32 位宽。
- Wa : 第三个源寄存器，32 位宽。

另外不同供应商的汇编工具如 ARM 汇编器 (armasm)、GNU 编译器等具有不同的语法。通常助记符和汇编指令是相同的，但汇编伪指令、定义、标号和注释语法则可能有差异。本章以 GNU 编译器为例，部分例子来自 Linux 内核的汇编代码片段。

1 算术和逻辑操作指令

算术和逻辑操作指令

指令分类	指令	描述
算术操作	ADD	加法指令。 1. 使用寄存器的加法 $ADD Wd WSP, Wn WSP, #imm\{, shift\} ; 32\text{-bit}$ $ADD Xd SP, Xn SP, #imm\{, shift\} ; 64\text{-bit}$ 3. 使用移位操作的加法 $ADD Wd, Wn, Wm\{, shift \#amount\} ; 32\text{-bit}$ $ADD Xd, Xn, Xm\{, shift \#amount\} ; 64\text{-bit}$
	SUB	减法指令 1. 使用寄存器的减法

1 算术和逻辑操作指令

		<pre>SUB Wd WSP, Wn WSP, Wm{, extend {#amount}} ; 32-bit SUB Xd SP, Xn SP, Rm{, extend {#amount}} ; 64-bit 2. 使用立即数的减法 SUB Wd WSP, Wn WSP, #imm{, shift} ; 32-bit SUB Xd SP, Xn SP, #imm{, shift} ; 64-bit 3. 使用移位操作的减法 SUB Wd, Wn, Wm{, shift #amount} ; 32-bit SUB Xd, Xn, Xm{, shift #amount} ; 64-bit</pre>
	ADC	<p>带进位的加法指令:</p> <pre>ADC Wd, Wn, Wm ; 32-bit ADC Xd, Xn, Xm ; 64-bit</pre>
	SBC	<p>带进位的减法</p> <pre>SBC Wd, Wn, Wm ; 32-bit SBC Xd, Xn, Xm ; 64-bit</pre>
	NGC	<p>负数减法</p> <pre>NGC Wd, Wm ; 32-bit NGC Xd, Xm ; 64-bit</pre>
逻辑操作	AND	<p>逻辑与操作</p> <ol style="list-style-type: none"> 使用立即数的与操作 <pre>AND Wd WSP, Wn, #imm ; 32-bit AND Xd SP, Xn, #imm ; 64-bit</pre> <ol style="list-style-type: none"> 使用移位操作的与操作 <pre>AND Wd, Wn, Wm{, shift #amount} ; 32-bit AND Xd, Xn, Xm{, shift #amount} ; 64-bit</pre>
	BIC	<p>清比特位</p> <pre>BIC Wd, Wn, Wm{, shift #amount} ; 32-bit BIC Xd, Xn, Xm{, shift #amount} ; 64-bit</pre>
	ORR	<p>按位执行或运算</p> <ol style="list-style-type: none"> 使用立即数 <pre>ORR Wd WSP, Wn, #imm ; 32-bit ORR Xd SP, Xn, #imm ; 64-bit</pre> <ol style="list-style-type: none"> 使用移位操作 <pre>ORR Wd, Wn, Wm{, shift #amount} ; 32-bit ORR Xd, Xn, Xm{, shift #amount} ; 64-bit</pre>
	ORN	<p>把源操作数按位取反后，在执行按位或运算</p> <pre>ORN Wd, Wn, Wm{, shift #amount} ; 32-bit ORN Xd, Xn, Xm{, shift #amount} ; 64-bit</pre>

	EOR	<p>按位异或运算</p> <ol style="list-style-type: none"> 使用立即数 EOR $Wd WSP, Wn, \#imm ; 32\text{-bit}$ EOR $Xd SP, Xn, \#imm ; 64\text{-bit}$ 使用移位操作 EOR $Wd, Wn, Wm\{, shift \#amount\} ; 32\text{-bit}$ EOR $Xd, Xn, Xm\{, shift \#amount\} ; 64\text{-bit}$
比较操作	CMP	<p>比较指令（比较两个数并且更新标志位）</p> <p>CMP $Wn WSP, \#imm\{, shift\} ; 32\text{-bit}$ CMP $Xn SP, \#imm\{, shift\} ; 64\text{-bit}$</p>
	CMN	负向比较（把一个数跟另外一个数的二进制补码相比较）
	TST	测试（执行按位与操作，并根据结构更新Z位）
搬移操作	MOV	<p>数据搬移指令</p> <ol style="list-style-type: none"> 加载立即数 MOV $Wd, \#imm ; 32\text{-bit}$ MOV $Xd, \#imm ; 64\text{-bit}$ 加载寄存器的值 MOV $Wd, Wm ; 32\text{-bit}$ MOV $Xd, Xm ; 64\text{-bit}$
	MVN	加载一个数的NOT值（取到逻辑反的值）

2 乘和除操作指令

乘法和除法指令

指令分类	指令	描述
乘法操作	MADD	<p>超级乘加指令</p> <p>MADD $Wd, Wn, Wm, Wa ; 32\text{-bit}$ MADD $Xd, Xn, Xm, Xa ; 64\text{-bit}$</p> <p>指令结果</p> <p>$Rd = Ra + Rn * Rm$</p>
	MNEG	<p>先乘然后取负数</p> <p>MNEG $Wd, Wn, Wm ; 32\text{-bit}$ MNEG $Xd, Xn, Xm ; 64\text{-bit}$</p> <p>指令结果：</p> <p>$Rd = -(Rn * Rm)$</p>
	MSUB	乘减运算

2 乘和除操作指令

	MSUB Wd, Wn, Wm, Wa ; 32-bit MSUB Xd, Xn, Xm, Xa ; 64-bit 指令结果: $Rd = Ra - Rn * Rm$, where R is either W or X
MUL	乘法运算 MUL Wd, Wn, Wm ; 32-bit MUL Xd, Xn, Xm ; 64-bit 指令结果 $Rd = Rn * Rm$
SMADDL	有符号的乘加运算 SMADDL Xd, Wn, Wm, Xa 指令结果: $Xd = Xa + Wn * Wm$
SMNEGL	有符号的乘负运算, 先乘后取负数 指令格式: SMNEGL Xd, Wn, Wm 指令结果: $Xd = -(Wn * Wm)$
SMSUBL	有符号的乘减运算 SMSUBL Xd, Wn, Wm, Xa 指令结果 $Xd = Xa - Wn * Wm$
SMULH	有符号的乘法运算, 但是只取高64比特位 SMULH Xd, Xn, Xm 指令结果 $Xd = \text{bits<}127:64\text{> of } Xn * Xm$
SMULL	有符号的乘法运算 SMULL Xd, Wn, Wm 指令结果 $Xd = Wn * Wm$
UMADDL	无符号的乘加运算 UMADDL Xd, Wn, Wm, Xa 指令结果 $Xd = Xa + Wn * Wm$
UMNEGL	无符号的乘负运算 UMNEGL Xd, Wn, Wm 指令结果

		$Xd = -(Wn * Wm)$
	UMULH	无符号的乘法运算，但是只取高64比特位 UMULH Xd, Xn, Xm 指令结果 $Xd = \text{bits<}127:64\text{> of } Xn * Xm$
	UMULL	无符号的乘法运算 UMULL Xd, Wn, Wm 指令结果 $Xd = Wn * Wm$
除法操作	SDIV	有符号的除法运算 SDIV $Xd, Wn, Wm ; 32\text{-bit}$ SDIV $Xd, Xn, Xm ; 64\text{-bit}$ 指令结果 $Rd = Rn / Rm, \text{ where } R \text{ is either } W \text{ or } X$
	UDIV	无符号的除法运算 UDIV $Xd, Wn, Wm ; 32\text{-bit}$ UDIV $Xd, Xn, Xm ; 64\text{-bit}$ 指令结果 $Rd = Rn / Rm, \text{ where } R \text{ is either } W \text{ or } X$

3 移位操作

指令	描述
LSL	逻辑左移指令。 LSL $Xd, Wn, Wm ; 32\text{-bit}$ LSL $Xd, Xn, Xm ; 64\text{-bit}$ 指令结果： $Rd = LSL(Rn, Rm), \text{ where } R \text{ is either } W \text{ or } X$
LSR	逻辑右移指令。 LSR $Xd, Wn, Wm ; 32\text{-bit}$ LSR $Xd, Xn, Xm ; 64\text{-bit}$ 指令结果： $Rd = LSR(Rn, Rm), \text{ where } R \text{ is either } W \text{ or } X$
ASR	算术右移指令。 ASR $Xd, Wn, Wm ; 32\text{-bit}$ ASR $Xd, Xn, Xm ; 64\text{-bit}$

4 位操作指令

	<p>指令结果:</p> <p>$Rd = ASR(Rn, Rm)$, where R is either W or X</p>
ROR	<p>循环右移指令。</p> <p>$ROR\ Wd, Ws, \#shift ; 32\text{-bit}$</p> <p>$ROR\ Xd, Xs, \#shift ; 64\text{-bit}$</p> <p>指令结果:</p> <p>$Rd = ROR(Rs, shift)$, where R is either W or X.</p>

4 位操作指令

指令	描述
BFI	<p>位段 (bitfield) 插入指令。</p> <p>$BFI\ Wd, Wn, \#lsb, \#width ; 32\text{-bit}$</p> <p>$BFI\ Xd, Xn, \#lsb, \#width ; 64\text{-bit}$</p> <p>指令结果:</p> <p>用 Xn 中的 $Bit[0: width]$ 替换 Xd 中的从 lsb 开始的 $width$ 位, Xd 其他位不变。</p>
BFC	<p>位段清零指令。</p> <p>$BFC\ Wd, \#lsb, \#width ; 32\text{-bit}$</p> <p>$BFC\ Xd, \#lsb, \#width ; 64\text{-bit}$</p> <p>指令结果:</p> <p>从 lsb 位开始清 Xd 中 $width$ 个比特位。</p>
BIC	<p>位清零指令。</p> <p>$BIC\ Wd, Wn, Wm\{, shift\#amount\} ; 32\text{-bit}$</p> <p>$BIC\ Xd, Xn, Xm\{, shift\#amount\} ; 64\text{-bit}$</p>
SBFX	<p>有符号的位段提取指令。</p> <p>$SBFX\ Xd, Xn, \#lsb, \#width ; 64\text{-bit}$</p> <p>指令结果:</p> <p>从 Xn 寄存器提取位段, 位段从第 lsb 位开始, 位宽为 $width$, 然后结果写入到 Xd 寄存器最低比特位中。</p>
UBFX	无符号的位段提取指令。
AND	<p>按位与操作</p> <p>$AND\ Xd, Xn$</p> <p>指令结果</p> <p>$Xd = Xd \& Xn$</p>
ORR	按位或操作

	ORR Xd, Xn 指令结果 $Xd = Xd Xn$
EOR	按位异或操作 EOR Xd, Xn 指令结果 $Xd = Xd ^ Xn$
CLZ	前导零计数指令

指令例子：

BFI X0, X1, #8, #4。把x1寄存器的bit[0:4]的位段，插入到x0寄存器的第8位中。

BFC X0, #8, #4。把x0寄存器中第8比特位开始清零，宽度为4

BIC W0, W0, #0xF0000000。将w0寄存器高4位清零。

BIC W1, W1, #0x0F。将w1寄存器低4位清零。

UBFX X8, X4, #8, #4。该指令等同于 $X8 = (X4 \& 0xF00) >> 8$

BFC 指令用来清除寄存器任意相邻的比特位。例如：

LDR, X0, =0x1234FFFF

BFC, X0, #4, #8

从第 4 位开始清 X0 中 8 个比特位，因此上述指令得到的结果为 X0=0x1234F00F

UBFX 和 SBFX 为无符号和有符号位域提取指令。指令格式为：

UBFX $Xd, Xn, \#lsb, \#width$

SBFX $Xd, Xn, \#lsb, \#width$

UBFX 从寄存器 (Xn) 中任意位置 (由 lsb 指定) 开始提取任意宽度 (由 width 指定) 的位域，将其零展开后放入到目的寄存器 (Xd)。例如：

LDR, X0, =0x5678ABCD

UBFX, X1, X0, #4, #8

上述指令执行完的结果为 X1 = 0x00000000_000000BC

和 UBFX 类似，SBFX 提取出位域后对目的寄存器进行有符号的展开。例如：

LDR, X0, =0x5678ABCD

SBFX, X1, X0, #4, #8

上述指令执行完的结果为 X1 = 0xFFFFFFFF_FFFFFFFBC

5 条件操作

A64 指令集沿用了 A32 指令集中的条件操作，在处理器状态寄存器中条件标志域 (NZCV) 描述了 4 种状态。

条件标志位	描述
-------	----

5 条件操作

N	负数标志（上一次运算结果为负值）
Z	零结果标志（上一次运算结果为零值）
C	进位或者借位标志（上一次执行的运算有进位或者有借位操作）
V	溢出标志（上一次运算结果溢出）

条件后缀	含义	标志	条件码
EQ	相等	Z=1	0b0000
NE	不相等	Z=0	0b0001
CS/HS	无符号数大于或者等于	C=1	0b0010
CC/LO	无符号数小于	C=0	0b0011
MI	负数	N=1	0b0100
PL	正数或零	N=0	0b0101
VS	溢出	V=1	0b0110
VC	未溢出	V=0	0b0111
HI	无符号数大于	(C=1) && (Z=0)	0b1000
LS	无符号数小于或等于	(C=0) (Z=1)	0b1001
GE	带符号数大于或等于	N == V	0b1010
LT	带符号数小于	N!=V	0b1011
GT	带符号数大于	(Z==0) && (N==V)	0b1100
LE	带符号数小于或等于	(Z==1) (n!=V)	0b1101
AL	无条件执行	-	0b1110
NV	无条件执行	-	0b1111

几乎大部分的 ARM 数据处理指令都可以根据执行结果来选择是否更新条件标志位。

指令	说明
CSEL	条件选择指令。 $CSEL \quad Wd, \quad Wn, \quad Wm, \quad cond ; \quad 32\text{-bit}$ $CSEL \quad Xd, \quad Xn, \quad Xm, \quad cond ; \quad 64\text{-bit}$ 指令结果： $Rd = \text{if } cond \text{ then } Rn \text{ else } Rm, \text{ where } R \text{ is either } W \text{ or } X$
CSET	条件置位指令 $CSET \quad Wd, \quad cond ; \quad 32\text{-bit}$ $CSET \quad Xd, \quad cond ; \quad 64\text{-bit}$

	指令结果: $Rd = \text{if } cond \text{ then } 1 \text{ else } 0, \text{ where } R \text{ is either } W \text{ or } X.$
CSINC	条件选择并增加指令。 CSINC $Wd, Wn, Wm, cond ; 32\text{-bit}$ CSINC $Xd, Xn, Xm, cond ; 64\text{-bit}$ 指令结果: $Rd = \text{if } cond \text{ then } Rn \text{ else } (Rm + 1), \text{ where } R \text{ is either } W \text{ or } X$

举例：

下面这段 C 语言代码：

```
if (i == 0)
    r = r + 2;
else
    r = r - 1;
```

可以使用如下汇编代码来表示：

```
CMP w0, #0 // if (i == 0)
SUB w2, w1, #1 // r = r - 1
ADD w1, w1, #2 // r = r + 2
CSEL w1, w1, w2, EQ //根据结果来选择
```

6 内存加载指令

和早期的 ARM 架构一样，ARMv8 架构也是基于指令加载和存储的架构 (Load/Store Architecture)。在这种架构下，所有的数据处理都需要在寄存器中完成，而不能直接在内存中完成。因此，首先把待处理数据从内存加载到通用寄存器，然后进行数据处理，最后把结果写回到内存中。

最常见的内存加载指令是 LDR 指令，存储指令是 STR 指令。

加载/存储指令格式：

```
LDR 目标寄存器, <存储器地址> //把存储器地址的数据加载到目标寄存器中
STR 源寄存器, <存储器地址> //把源寄存器的值存储到存储器中
```

LDR 和 STR 指令根据不同的数据大小有多种变种。

指令	说明
LDR	数据加载指令
LDRSW	有符号的数据加载指令，大小为字 (word)
LDRB	数据加载指令，大小为字节
LDRSB	有符号的加载指令，大小为字节

6 内存加载指令

LDRH	数据加载指令, 大小为半字 (Halfword)
LDRSH	有符号的数据加载指令, 大小为半字 (Halfword)
STRB	数据存储指令, 大小为字节
STRH	数据存储指令, 大小为半字

LDR 和 STR 指令有如下几个常用方式:

(1) 地址偏移模式

地址偏移模式常常使用寄存器的值来表示一个地址, 或者基于寄存器的值做一些偏移来计算出内存地址, 并且把这个内存地址的值加载到通用寄存器中。偏移量可以是正数, 也可以是负数。常见的指令格式如下:

```
| LDR Xd, [Xn, $offset]
```

首先在 Xn 寄存器的内容上加一个 offset 偏移量后作为内存地址, 加载此地址的内容到 Xd 寄存器。

举例:

```
LDR X0, [X1] //内存地址为X1寄存器的值, 加载此地址的值到X0寄存器
```

```
LDR X0, [X1, #8] //内存地址为X1+8, 加载此地址的值到X0
```

```
LDR X0, [X1, X2] //内存地址为X1+X2, 加载此地址的值到X0
```

```
LDR X0, [X1, X2, LSL, #3] //内存地址为X1+(X2<<3), 加载此地址的值到X0
```

```
LDR X0, [X1, W2, SXTW] //先把W2的值做有符号的扩展, 再和X1相加后作为地址, 加载此地址的值到X0
```

```
LDR X0, [X1, W2, SXTW, #3] //先把W2的值做有符号的扩展, 然后左移3位, 再和X1相加后作为地址, 加载此地址的值到X0
```

(2) 变基模式

变基模式主要有两种, 一种是前变基模式 (pre-index 模式), 先更新偏移地址然后再访问内存; 另外一种是后变基模式 (post-index 模式), 先访问内存地址然后再更新偏移地址。

举例:

```
LDR X0, [X1, #8]! //前变基模式。先更新X1的值为X1+8, 然后以新的X1值为地址, 加载内存的值到X0
```

```
LDR X0, [X1], #8 //后变基模式。以X1的值为地址, 加载该内存地址的值到X0, 然后再更新X1寄存器为X1+8
```

```
SDP X0, X1, [SP, #-16]! //把X0和X1的值压回栈中
```

```
LDP X0, X1, [SP], #16 //把X0和X1弹出栈
```

(3) PC 相对地址模式

汇编代码里常常会使用便签 (Label) 来标记代码逻辑片段。我们可以使用 PC 相对地址模式来访问这些便签。在 ARM 架构中，我们不能直接访问 PC 地址，但是通过 PC 相对地址模式来访问一个和 PC 相关的地址。

举例：

```
LDR X0, <label> //从label标签地址处加载8个字节到x0寄存器
```

利用这个特性可以实现地址重定位，比如在 Linux 内核启动汇编代码 head.S 文件中，启动 MMU 之后，使用该特性来实现从运行地址定位到链接地址。

```
<arch/arm64/kernel/head.S>
```

```
1    __primary_switch:
2        adrp x1, init_pg_dir
3        bl __enable_mm
4
5        ldr x8, =__primary_sw
6        adrp x0, __PHYS_OFFSET
7        br x8
8    ENDPROC(__primary_sw)
```

第 3 行 `__enable_mm` 函数打开 MMU，第 5 行和第 7 行实现跳转到 `__primary_sw` 函数中，其奥秘是 `__primary_sw` 函数的地址是链接地址，即内核空间的虚拟地址，而启动 MMU 前后处理器运行在实际的物理地址上，即运行地址。上述指令实现了地址重定位功能。

(4) 总结

读者容易对下面三条指令产生困扰。

1. LDR X0, [X1, #8] //内存地址为X1+8，加载此地址的值到X0
2. LDR X0, [X1, #8]! //前变基模式。先更新X1的值为X1+8，然后以新的X1值为地址，加载内存的值到X0
3. LDR X0, [X1], #8 //后变基模式。以X1的值为地址，加载该内存地址的值到X0，然后再更新X1寄存器为X1+8

中括号 ([]) 表示从这个地址中读取或者存储数据，而指令中的感叹号 (!) 表示是否更新存放地址的寄存器，即写回更新寄存器。

7 多字节内存加载和存储指令

7 多字节内存加载和存储指令

在 A32 指令集中提供 LDM 和 STM 指令的实现多字节内存加载和存储，到了 A64 指令集，不在提供 LDM 和 STM 指令，而是采用 LDP 和 STP 指令。

举例：

```
LDP X3, X7, [X0] //以X0的值为地址，加载此地址的值到X3寄存器，以X0+8为地址，加载此地址的值到X7寄存器
```

```
LDP X1, X2, [X0, #0x10]! //前变基。先把X0 = X0 + 0x10；然后以X0为地址，加载此地址的值到X1；然后以X0+8位地址，加载此地址的值到X2
```

```
STP X1, X2, [X4] //存储X1的值到地址为X4的内存中，然后存储X2的值到地址为X4+8的内存中。
```

8 非特权访问级别的加载和存储指令

ARMv8 架构中实现了一组非特权访问级别的加载和存储指令，它适用于在 EL0 进行的访问权限。

指令	描述
LDTR	非特权加载指令
LDTRB	非特权加载指令，加载一个字节
LDTRSB	非特权加载指令，加载一个有符号的字节
LDTRH	非特权加载指令，加载 2 个字节
LDTRSH	非特权加载指令，加载 2 个有符号的字节
LDTRSW	非特权加载指令，加载 4 个有符号的字节
STTR	非特权存储指令，存储 8 个字节
STTRB	非特权存储指令，存储 1 个字节
STTRH	非特权存储指令，存储 2 个字节

当处理器状态 PSTATE 中的 UAO 字段为 1 时，在 EL1 和 EL2 执行这些非特权指令的效果和特权指令是一样的，这个特性在 ARMv8.2 扩展特性中加入的。

9 内存屏障指令

ARMv8 架构实现了一个弱一致性内存模型，内存访问的次序有可能和程序的预期的次序不一样。A64 和 A32 提供了内存屏障的指令。

指令	描述
DMB	数据存储屏障 (Data Memory Barrier, DMB) 确保在执行新的存储器访问前所有的存储器访问都已经完成。
DSB	数据同步屏障 (Data synchronization Barrier, DSB) 确保在下一个指令执行前所有存储器访问都已经完成。
ISB	指令同步屏障 (Instruction synchronization Barrier, ISB) 清空流水线，确保在执行新的指令前，之前所有的指令都已完成。

除此之外，ARMv8 架构还提供一组新的加载和存储指令，显式包含了内存屏障功能。

指令	描述
LDAR	加载获取指令 (Load-Acquire) 所有的加载和存储指令必须执行完成才能执行LDAR后面的指令
STLR	存储释放指令 (Store-Release) 所有的加载和存储指令必须在STLR指令之前完成。

10 独占访存指令

ARMv7 和 ARMv8 架构都提供独占访问内存(Exclusive Memory Access)的指令。在 A64 指令，LDXR 指令尝试在内存总线中申请一个独占访问的锁，然后去访问一个内存地址。STXR 指令会往刚才 LDXR 指令已经申请独占访问的内存地址里写入新内容。LDXR 和 STXR 指令通常组合使用来完成一些同步操作，比如 Linux 内核的 spinlock 锁。

另外还提供多字节独占访问的指令，LDRP 和 STXP 指令。

指令	描述
LDXR	独占内存访问指令 LDXR Wt , [Xn SP{,#0}] ; 32-bit LDXR Xt , [Xn SP{,#0}] ; 64-bit
STXR	独占内存访问指令 STXR Ws , Wt , [Xn SP{,#0}] ; 32-bit STXR Ws , Xt , [Xn SP{,#0}] ; 64-bit
LDPX	多字节独占访问内存指令 LDPX $Wt1$, $Wt2$, [Xn SP{,#0}] ; 32-bit LDPX $Xt1$, $Xt2$, [Xn SP{,#0}] ; 64-bit
STXP	多字节独占访问内存指令

11 跳转指令

	<code>STXP Ws, Wt1, Wt2, [Xn SP{,#0}] ; 32-bit</code> <code>STXP Ws, Xt1, Xt2, [Xn SP{,#0}] ; 64-bit</code>
--	--

11 跳转指令

编写汇编代码常常会使用到跳转指令，A64 指令集提供了多种不同功能的跳转指令。

指令	描述
B	跳转指令。 <code>B label</code> 该跳转指令可以在当前PC地址相对±128MB的范围无条件地跳转到标签处。
B.cond	有条件的跳转指令。 <code>B.cond label</code> 比如 <code>B.EQ</code> , 该跳转指令可以在当前PC地址相对±1MB的范围有条件地跳转到标签处。
BL	带返回地址的跳转指令。 <code>BL label</code> 和B指令类似, 不同的地方是BL指令会返回地址设置到X30寄存器中。
BR	跳转到寄存器指定的地址处。 <code>BR Xn</code>
BLR	跳转到寄存器指定的地址处。 <code>BLR Xn</code> 和BR指令类似, 不同的地方是BLR指令会返回地址设置到X30寄存器中。
RET	从子函数返回。 <code>RET {Xn}</code> 从子函数返回到Xn寄存器指定的地址。若没有指定Xn, 那么默认会跳转到X30寄存器指定的地址。
CBZ	比较并跳转指令。 <code>CBZ Wt, label ; 32-bit</code> <code>CBZ Xt, label ; 64-bit</code> 比较Xi寄存器是否为0, 若为0则跳转到label标签处。跳转范围是±1MB。
CNBZ	比较并跳转指令。 <code>CBNZ Wt, label ; 32-bit</code> <code>CBNZ Xt, label ; 64-bit</code> 比较Xi寄存器是否为0, 若不为0则跳转到label标签处。跳转范围是±1MB。

TBZ	测试比特位并跳转指令。 TBZ R<t>, #imm, label 比较Rt寄存器中第imm比特位是否为0，若为0，则跳转到label标签处，跳转范围是±32KB。
TBNZ	测试比特位并跳转指令。 TBNZ R<t>, #imm, label 比较Rt寄存器中第imm比特位是否为0，若不为0，则跳转到label标签处，跳转范围是±32KB。

举例，在 Linux 内核代码中的 `ret_from_fork` 函数里多次使用了跳转指令。

```
< arch/arm64/kernel/entry.S >

ENTRY(ret_from_fork)
    bl    schedule_tail          //跳转到schedule_tail函数
    cbz   x19, 1f                //比较x19是否为0，为0说明不是一个内核线程。
    mov   x0, x20
    blr   x19                  //若是内核线程，则跳转到x19指定的地址处
1:    get_thread_info tsk
    b     ret_to_user           //跳转到ret_to_user函数
ENDPROC(ret_from_fork)
```

12 异常处理指令

指令	描述
SVC	系统调用指令。 SVC #imm 允许应用程序通过SVC指令来自陷到操作系统里，通常会进入到EL1异常等级。
HVC	虚拟化系统调用指令。 HVC #imm 允许主机操作系统通过HVC指令自陷到虚拟机管理程序(Hypervisor)中，通常会进入到EL2异常等级。
SMC	安全监控系统调用指令。 SMC #imm 允许主机操作系统或者虚拟机管理程序通过SMC指令自陷到安全监管 (Secure Monitor) 中，通常会进入到EL3异常等级。

13 系统寄存器访问指令

13 系统寄存器访问指令

在 ARMv7 架构，通过访问 CP15 协处理器来访问系统寄存器，而在 ARMv8 架构里进行了大幅改进和优化。通过 MRS 和 MSR 两条指令可以直接访问系统寄存器。

系统寄存器访问指令

指令	描述
MRS	访问系统寄存器
MSR	更新系统寄存器。

1. 访问系统特性寄存器

```
MRS X4, ELR_EL1 //访问ELR_EL1寄存器到X4
MSR SPSR_EL1, X0 //把X0的值更新到SPSR_EL1寄存器
```

2. 访问系统寄存器

```
mrs x20, sctlr_el1 //访问系统控制寄存器SCTLR_EL1①
msr sctlr_el1, x20 //设置系统控制寄存器SCTLR_EL1
```

系统控制寄存器 SCTLR_EL1 可以用来设置系统很多属性，比如系统大小端等。

我们可以使用 MRS 和 MSR 指令来访问系统寄存器。

3. 访问 PSTATE 字段

除了访问系统寄存器之外，还能通过 MSR 和 MRS 指令来访问处理器状态 PSTATE 相关的字段，这些字段可以看作是系统特殊用途的寄存器^②。

特殊用途的寄存器

寄存器	说明
CurrentEL	获取当前系统的异常等级
DAIF	获取和设置PSTATE中的DAIF掩码
NZCV	获取和设置PSTATE中的条件掩码
PAN	获取和设置PSTATE中的PAN字段
SPSel	设置和设置当前的栈指针寄存器
UAO	获取和设置PSTATE中的UAO字段

^①详见<ARM Architecture Reference Manual, for ARMv8-A architecture profile, v8.4>第 D12.2.100 章

^②详见<ARM Architecture Reference Manual, for ARMv8-A architecture profile, v8.4>第 C5.2 章

在 Linux 内核代码中使用该指令来关闭本地处理器的中断。

```
<arch/arm64/include/asm/assembler.h>

.macro disable_daif
    msr    daifset, #0xf
.endm

.macro enable_daif
    msr    daifclr, #0xf
.endm
```

disable_daif 宏用来关闭本地处理器中处理器状态寄存器(PSR)中的 DAIF 功能，也就是关闭处理器调试，系统错误，IRQ 和 FIQ 中断。而 enable_daif 宏用来打开上述功能。

下面是一个设置 SP 栈指针和获取当前异常等级的例子，代码实现在 arch/arm64/kernel/head.S 汇编文件中。

```
<arch/arm64/kernel/head.S>

ENTRY(el2_setup)
    msr SPsel, #1           // 设置栈指针，使用SP_EL1
    mrs x0, CurrentEL      // 获取当前异常等级
    cmp x0, #CurrentEL_EL2
    b.eq 1f
```