**国产自研操作系统设计与实现**

**----DIM-SUM操作系统详解**

**谢宝友 著**

**2019/02**



****

第 1章 XIX

准备工作 XIX

1.1. DIM-SUM简介 XIX

1.1.1. 为什么是HOT-POT？ XIX

1.1.2. DIM-SUM是什么 XIX

1.1.3. DIM-SUM欢迎什么？ XX

1.1.4. DIM-SUM不欢迎什么？ XXI

1.2. 获得源码 XXI

1.2.1. 通过网页下载源码 XXI

1.2.2. 通过git获取源代码 XXII

1.3. 搭建调试开发环境 XXII

1.3.1. 安装ubuntu 16.04 XXII

1.3.2. ubuntu 16.04环境配置 XXIII

1.3.3. 搭建编译环境 XXIV

1.3.4. 编译HOT-POT XXVI

1.3.5. 运行HOT-POT XXVII

1.3.6. 开始调试 XXIX

1.4. 向DIM-SUM操作系统提交补丁 XXX

1.4.1. 心态 XXX

1.4.2. 准备工作 XXX

1.4.3. 制作补丁 XXXII

1.4.4. 制作正确的补丁 XXXIV

1.4.5. 发送补丁 XXXV

1.5. 获得帮助 XXXVI

1.6. 提醒 XXXVI

第 2章 XV

2. 算法基础 XV

2.1. 链表 XV

2.2. 哈希表 XV

2.3. 红黑树 XV

2.4. 基树 XV

第 3章 XV

3. 计数与同步互斥 XV

3.1. 计数 XV

3.1.1. 计数的难题 XVI

3.1.2. 精确计数器 XVI

3.1.2.1. 精确计数器的数据结构 XVI

3.1.2.2. 精确计数器API XVI

3.1.2.3. 精确计数器的实现 XVII

3.1.3. 近似计数器 XIX

3.1.3.1. 近似计数器的设计原理 XX

3.1.3.2. 近似计数器的数据结构 XX

3.1.3.3. 近似计数器API XX

3.1.3.4. 近似计数器的实现 XX

3.1.4. 引用计数 XX

3.2. 内核互斥原语 XX

3.2.1. 每CPU变量 XX

3.2.1.1. 每CPU变量为何重要 XX

3.2.1.2. 每CPU变量的设计与实现 XX

3.2.2. 自旋锁 XX

3.2.2.1. 自旋锁的用途 XX

3.2.2.2. 自旋锁的数据结构 XX

3.2.2.3. 自旋锁API XX

3.2.2.4. 自旋锁的实现 XX

3.2.3. 自旋位锁 XX

3.2.3.1. 自旋位锁的设计原理 XXI

3.2.3.2. 自旋位锁的数据结构 XXI

3.2.3.3. 自旋位锁API XXI

3.2.3.4. 自旋位锁的实现 XXI

3.2.4. 自旋顺序锁 XXI

3.2.4.1. 自旋顺序锁的设计原理 XXI

3.2.4.2. 自旋顺序锁的数据结构 XXI

3.2.4.3. 自旋顺序锁API XXI

3.2.4.4. 自旋顺序锁的实现 XXI

3.2.5. 自旋读写锁 XXI

3.2.5.1. 自旋读写锁的设计原理 XXI

3.2.5.2. 自旋读写锁的数据结构 XXI

3.2.5.3. 自旋读写锁API XXI

3.2.5.4. 自旋读写锁的实现 XXI

3.2.6. 读写信号量 XXI

3.2.6.1. 读写信号量的设计原理 XXI

3.2.6.2. 读写信号量的数据结构 XXI

3.2.6.3. 读写信号量API XXI

3.2.6.4. 读写信号量的实现 XXI

3.2.7. 互斥锁 XXI

3.2.7.1. 互斥锁的设计原理 XXII

3.2.7.2. 互斥锁的数据结构 XXII

3.2.7.3. 互斥锁API XXII

3.2.7.4. 互斥锁的实现 XXII

3.3. 内核同步原语 XXII

3.3.1. 信号量 XXII

3.3.1.1. 信号量的设计原理 XXII

3.3.1.2. 信号量的数据结构 XXII

3.3.1.3. 信号量API XXII

3.3.1.4. 信号量的实现 XXII

第 4章 XV

4. 进程 XV

4.1. SMP CPU初始化 XV

4.2. 调度子系统初始化 XV

4.3. 数据结构 XV

4.3.1. 进程 XV

4.3.2. 调度队列 XV

4.3.3. 杂项 XV

4.4. 进程调度 XVI

4.4.1. 上下文切换 XVI

4.4.2. 唤醒进程 XVI

4.5. 进程等待 XVI

4.6. Sleep XVI

4.7. 消息队列 XVI

第 5章 XV

5. 中断及定时器 XV

5.1. 中断控制器初始化 XV

5.2. 中断控制器维护 XV

5.3. 中断处理 XV

5.3.1. 序言 XV

5.3.2. 中断处理 XV

5.3.3. 尾声 XV

5.4. 工作队列 XVI

5.5. 定时器 XVI

第 6章 XV

6. 内存管理 XV

6.1. 内存初始化 XV

6.1.1. 早期内存映射 XV

6.2. Boot内存分配器 XV

6.3. 内存页面映射 XV

6.4. 页面分配器 XV

6.4.1. 页帧管理 XV

6.4.2. 页面分配器的设计原理 XV

6.4.3. 页面分配器的数据结构 XVI

6.4.4. 页面分配器的实现 XVI

6.5. Beehive内存分配器 XVI

6.6. 与文件缓存的关系 XVI

第 7章 XV

7. 块设备 XV

7.1. 块设备的识别及初始化 XV

7.2. 磁盘分区识别 XV

7.3. IO请求 XV

7.4. IO调度 XV

第 8章 XV

8. 文件系统 XV

8.1. 磁盘分区识别 XV

8.1.1. 磁盘初始化 XV

8.1.2. 分区识别 XV

8.2. 虚拟文件系统 XV

8.2.1. 文件系统调用 XV

8.2.2. 文件系统缓存 XV

8.3. 内存文件系统实现 XV

8.4. 设备文件系统 XV

8.5. 文件系统日志 XV

8.6. LEXT3文件系统 XV

8.7. 根文件系统 XVI

第 9章 XV

9. 杂项 XV

9.1. Klibc XV

9.2. 网络子系统 XV

9.3. SIMPLE-KSHELL XV

# **第** **1章**

# 准备工作

本书详细解剖了DIM-SUM操作系统的源代码。DIM-SUM操作系统是本书作者谢宝友在过去几年中，陆续实现的一个操作系统。它的版权遵循GPL V2开源软件协议。

由于DIM-SUM是一个不断发展中的操作系统，因此本书以其第一个发布版本为基础进行剖析。这个版本的名称是：HOT-POT。以后的版本可能会被称为“dumpling”、“noodle”，总之是我们能够想到的中国小吃名称:-)

## DIM-SUM简介

### 为什么是HOT-POT？

HOT-POT意为火锅，是非常有川渝特色的地方美食。之所以取这个名称，主要有如下原因：

1. 四川是作者的故乡，并且作者来到成都已经有20多年了。对这个城市的火锅印象深刻！
2. 红通通的火锅，有鲜明的民族烙印，代表了喜庆、热闹、欢快的含义。
3. 如果您真的吃过火锅，那么可能忘不了它的辣，一定不能小瞧它，并且久久不能忘怀以至于想要多尝几次。希望这款操作系统也能带给您这样的感觉。

### DIM-SUM是什么

如果说目前的DIM-SUM是一个完美的操作系统，那无疑是一个谎言。但是，如果说这个操作系统就仅仅是一个茶余饭后的谈资，那无疑也是另一个谎言。

最基本的，希望它可供操作系统爱好者用于学习目的，并且作者相信这完全没有问题。但是，作者的目的不仅仅是如此。其远大的目标，是实现一款工业级服务器操作系统。简而言之，就是一款可以在生产环境下面使用的，可以在服务器和个人电脑上面正常运行的操作系统。当然了，这样的一款操作系统必然也能够运行在嵌入式设备中。例如，运行在电视、电表、摄像头、手表，以及其他一些我们能够想象得到的嵌入式设备中。

任何心智正常的人，都知道实现这样的操作系统是一件很难的事情。读者可能忍不住想问：为什么作者您想去做这么一件很难，并且可能没有什么收益，也许还会让您招致耻笑的笨事呢？难道您真的是一个笨蛋？

作者也忍不住这样回答质疑：

1. 是的，作者确实是一个笨蛋。但是古语说的好：聪明人下笨功夫！
2. 正如作者在其译作《深入理解并行编程》一书的译者序中所说：

*20年前，当我正式成为一名软件工程师的时候，就有一个梦想：开发一款操作系统。那时候，虽然知道Linux的存在，但是实在找不到一台可以正常安装使用Linux的PC。因此只能阅读相关的源码分析书籍而不能动手实践。*

在浮躁的年代，谈论梦想可能一件不合时宜的行为。然而有什么办法呢？这毕竟真的是20年前的梦想，难道您想让我撒谎？

1. 开源软件运动，已经为我们实现操作系统提供了现实可能性。首先，象Linux这样的开源操作系统为我们提供了很好的基础，这样我们可以从开源软件中学习到不少的技能、方法、设计思路。其次，开源软件允许分散在各地的开发者协同工作，集思广益的开发操作系统。作者在Linux社区中，作为ARM/ZTE ARCHITECTURE的Maintainer，对此深有体会。
2. 操作系统是IT行业的核武器。到目前为止，我们还处于“缺芯少魂”的状态，其中的“魂”就是操作系统。真正核心的软件，需要一代人，甚至几代人耐心的雕琢，而不能寄希望于短时间内产生立竿见影的效果。换句话说，要有“板凳一坐十年冷”的心态来做这件事。有了这样的心态，就不会觉得难。
3. 从另一个角度来说：万古长空，一朝风月。任何难事，一旦想要去做，就需要把握当下。空谈误国，实干方能兴邦。即使这件事情很难，但是不动手永远不会有任何结果。况且，作者喜欢有挑战性的事情，例如：写一个工业级服务器操作系统:-)

目前，DIM-SUM已经实现了如下功能：

1、arm 64 qemu小系统，含内存、时钟初始化。

2、全局优先级调度模块，调度算法类似于Linux实时调度。

3、内存管理模块，包含页面管理、beehive分配器。

4、兼容linux ext3的LEXT3文件系统。

5、块设备层实现。

6、集成了LWIP网络协议栈。

7、移植了常用的内核态C库API。

8、实现了一个粗糙的命令控制台。

总之，作者认为HOT-POT是DIM-SUM操作系统的Good Start。在后续的开发过程中，作者怀着热切的心情，期待您参与到它的开发中来。

### DIM-SUM欢迎什么？

任何建设性对抗性质的建议、稳定优雅的代码、BUG报告、测试、社区建设等等，都是DIM-SUM所欢迎的！

### DIM-SUM不欢迎什么？

我们不欢迎空谈和只会抱怨的人。虽然我们知道，DIM-SUM并不完善，您有很多喷它的理由。

## 获得源码

Paul E.Mckenney曾经说过：

*If you want to do cool things, it is necessary to invest large amounts of time learning and (especially!) practicing.*

诚哉斯言！

请读者相信我这个20年传奇工匠程序员的经验：要深刻的理解象Linux操作系统这样的复杂代码，必须要动手实践，对着源代码看书！

本书也不会大量的粘贴DIM-SUM源码，这是故意为之的。目的是逼迫读者下载源代码并对照着源码阅读本书。

要获得本书配套源码，您有两种方式：

1. 直接通过网页下载
2. 通过git获取源代码

### 通过网页下载源码

通过网页下载源码是最简单快捷的方式。您可以通过如下两个网址下载DIM-SUM的源代码：

1. <https://download.csdn.net/user/xiebaoyou/>
2. <http://blog.chinaunix.net/uid-25845340-id-5785292.html>

将下载的源码包命名为dim-sum.20180519.tar.bz2。为了防止下载过程中，由于网络原因导致的文件损坏，您可以验证源码包的MD5值。在ubuntu 16.04系统中，可以在命令行控制台输入如下命令来得到源码包的MD5值：

md5sum dim-sum.20180519.tar.bz2

正确的MD5值应当是：

e888c22b47adcb266382e1791bb729f8

关于DIM-SUM操作系统的新消息，我也会通过博客发布，地址是：

<http://blog.chinaunix.net/uid/25845340.html>

通过网页下载源码，可以满足阅读本书的要求。但是，它满足不了您如下的要求：

1. 获得DIM-SUM操作系统最新的源代码
2. 查阅DIM-SUM操作系统的补丁记录

下一步将讨论如何通过git获得源代码。这也是作者推荐的方式。

### 通过git获取源代码

无论怎样强调git在开源项目中的重要性都不过分。作者甚至推荐您找一本git简明手册仔细读一读。

假设您的系统中已经安装好git工具。您可以通过如下命令获取DIM-SUM操作系统的源代码。

git clone <https://code.csdn.net/xiebaoyou/dim-sum.git>

这个命令会在您的当前目录中创建一个名为dim-sum的子目录，并将DIM-SUM操作系统的代码下载到本地。

小问题1.1：嘿！看起来您是想让读者在Linux环境下阅读并调试代码，但是Linux环境阅读代码是否真的方便。为什么不在书中直接贴出所有代码，您用意何在？

当然，对于大多数读者来说，不仅仅想对照着源码阅读本书。他们还希望：

1. 跟踪DIM-SUM的最新版本。
2. 查阅DIM-SUM的补丁记录，明白每一个补丁的作用，与补丁的作者联系。
3. 给DIM-SUM提交补丁。
4. 在DIM-SUM中添加自己的代码。
5. 在PC中调试DIM-SUM的代码。

这样的读者需要仔细阅读下一节的内容。

## 搭建调试开发环境

虽然通过上一节的方法获得源码以后，您将能够顺利的阅读本书。但是有一句话说得很正确：纸上得来终觉浅，绝知此事要躬行。因此强烈建议您按照本节的方法搭建调试开发环境。

### 安装ubuntu 16.04

作者的笔记本中，安装的操作系统是ubuntu 16.04，因此可以确保本节中的调试开发环境能顺利的运行在ubuntu 16.04系统中。其他的Linux版本应该也是可以的，但是作者并不完全保证其他版本完全正常工作。

首先，您需要在如下链接中下载ubuntu 16.04的安装镜像：

<http://old-releases.ubuntu.com/releases/16.04.3/>

视您的开发环境PC 机CPU配置，您可以下载不同的安装镜像：

1. 64位PC，通过如下链接下载

<http://old-releases.ubuntu.com/releases/16.04.3/ubuntu-16.04-desktop-amd64.iso>

1. 32位PC，通过如下链接下载

<http://old-releases.ubuntu.com/releases/16.04.3/ubuntu-16.04-desktop-i386.iso>

在http://old-releases.ubuntu.com/releases/16.04.3/MD5SUMS中，有这两个镜像文件的MD5值，分别为：

c94d54942a2954cf852884d656224186

610c4a399df39a78866f9236b8c658da

请检查下载文件的MD5值，确保与上面两个值一致。

小问题1.2：呃，您已经两次提到MD5值了，它有那么重要吗？

有两种方法安装您的ubuntu 16.04。

第一种方法，是直接在物理机上安装ubuntu 16.04。建议您找一台Linux机器，并使用dd命令将镜像烧写到USB中，并通过USB来安装系统。

第二种方法，是在虚拟机中安装ubuntu 16.04。这种方法可以不用将安装镜像烧写到USB中。在虚拟机中实际搭建开发环境，其效果和物理机中是一致的。

ubuntu 16.04的安装过程比较简单，这里不再详述。

小问题1.3：可是我没有Linux环境，也不知道怎么用dd命令来烧写镜像到USB？

为了方便Linux新手，后续假设您是在虚拟机中安装ubuntu 16.04。并以此为基础搭建调试开发环境。

### ubuntu 16.04环境配置

首先，请配置您的虚拟机，为它创建两个网卡。

作者在Oracle VM VirtualBox管理器中，是通过“设置”->“网络”进入配置界面。

在“网卡1”标签界面中，选择“启用网络连接”，并在“连接方式”中选择“仅主机(Host-Only)适配器”，在“界面名称”中选择“vboxnet0”。该网卡用于虚拟机与物理机的文件共享连接。

在“网卡2”标签界面中，选择“启用网络连接”，并在“连接方式”中选择“网络地址转换(NAT)”。该网卡用于虚拟机与互联网的连接。后面我们将会看到，无论是安装软件包，还是通过git下载代码，都需要连接到互联网。

第二步，启动ubuntu 16.04虚拟机，进行如下基本环境配置：

1、打开命令行控制台，在作者的环境中，是通过按“ctrl+alt+t”实现的。

2、在控制台中，输入如下命令切换到root用户。当然，为了防止误操作损坏系统，您也可以不用切换到root用户，但是请记得为后续的某些操作添加sudo前缀。

sudo -s

3、在控制台中，输入如下命令更新apt仓库：

apt-get update

视您的网络状态而定，这一步可能需要花费数分钟的时间。

由于您还没有正确的配置双网卡，因此您的虚拟机还不能正确的连接到互联网。这时可以简单的禁止第一个网卡。

4、在控制台中，输入如下命令安装vim：

apt-get install vim

5、编辑/etc/network/interfaces，输入如下内容：

# interfaces(5) file used by ifup(8) and ifdown(8)

auto lo

iface lo inet loopback

auto enp0s3

iface enp0s3 inet static

address 192.168.0.98

netmask 255.255.255.0

auto enp0s8

iface enp0s8 inet dhcp

其中enp0s3是作者创建的虚拟机中的内网网卡，用于虚拟机与主机之间的通信，在您的机器上可能是其他名称，请注意调整。192.168.0.98是该网卡的地址，请根据您的实际配置进行调整。

enp0s8则是作者创建的虚拟机中的外网网卡，用于虚拟机与互联网之间的通信，在您的机器上可能是其他名称，请注意调整。

编辑并保存/etc/network/interfaces文件后，请运行如下命令重启网络服务，使配置生效：

/etc/init.d/networking restart

第三步，运行如下命令创建工具目录：

mkdir /hot-pot

小问题1.4：一定要在根目录下创建hot-pot目录吗？其他目录名称可以吗？

### 搭建编译环境

首先，您需要在ubuntu 16.04中安装git工具。在控制台中输入如下命令开始安装：

apt-get install git

执行完毕后，在控制台中输入如下命令验证git工具是否正确安装：

git --version

如果看到如下控制台输出，则表示git工具安装成功：

git version 2.7.4

第二步，通过git工具下载辅助工具。可以通过如下命令获取这些工具：

cd /hot-pot/

git clone https://code.csdn.net/xiebaoyou/assistant.git

由于assistant这个git仓库中包含了数百M的工具链，可能会花费数分钟甚至一个小时的时间，请耐心等待。完成后，通过如下命令查看是否下载成功：

ls assistant -al

在控制台上，应当看到如下输出：

总用量 44

drwxrwxr-x 5 xiebaoyou xiebaoyou 4096 5月 20 15:12 .

drwxrwxr-x 31 xiebaoyou xiebaoyou 4096 5月 20 20:04 ..

drwxrwxr-x 2 xiebaoyou xiebaoyou 4096 5月 20 15:12 gcc-linaro-5.3

drwxrwxr-x 8 xiebaoyou xiebaoyou 4096 5月 20 15:15 .git

drwxrwxr-x 2 xiebaoyou xiebaoyou 20480 5月 19 17:16 hot-pot.toolchains.4.9.3

-rw-rw-r-- 1 xiebaoyou xiebaoyou 15 5月 19 16:45 README.md

第三步，您需要安装编译工具链。要编译HOT-POT，需要gcc 4.x版本的交叉编译工具链。作者已经将相应的工具链放到assistant/hot-pot.toolchains.4.9.3中。

在控制台中运行如下命令，获得hot-pot.toolchains.4.9.3.tar.bz2工具链压缩包：

cd /hot-pot/assistant/hot-pot.toolchains.4.9.3

cat x\* > hot-pot.toolchains.4.9.3.tar.bz2

其中第二条命令将hot-pot.toolchains.4.9.3目录中所有文件合并，生成压缩包文件。

然后使用如下命令检查生成的压缩包是否正确：

md5sum hot-pot.toolchains.4.9.3.tar.bz2

您将会看到如下输出：

3cdfaf72c2fe82535c596414fd4b026c hot-pot.toolchains.4.9.3.tar.bz2

如果输出的MD5值不是“3cdfaf72c2fe82535c596414fd4b026c”，则说明下载过程有误，生成的工具链压缩包不正确。

在控制台中输入如下命令，将压缩包解压，获得可以运行的工具链：

tar xvjf hot-pot.toolchains.4.9.3.tar.bz2 -C ../

在控制台中输入如下命令，检查工具链是否能够正常运行：

/hot-pot/assisant/aarch64-linux-gnu/bin/aarch64-linux-gnu-gcc --version

如果在控制台中看到如下输出，那么恭喜您：

aarch64-linux-gnu-gcc (GCC) 4.9.3

Copyright (C) 2015 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

第四步，您需要安装gdb工具。这样我们就能够象调试应用程序那样，对DIM-SUM操作系统进行单步跟踪调试了。这对于深入理解DIM-SUM操作系统有极大的帮助。因此，强烈建议您安装这个工具。

在Linaro gcc 5.3工具链中，已经包含了gdb工具。

同样的，作者在已经将Linaro gcc 5.3工具链放到assistant/gcc-linaro-5.3中。

您可以在控制台中输入如下命令，来生成Linaro gcc 5.3工具链压缩包：

cd /hot-pot/assistant/gcc-linaro-5.3

cat x\* > gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf.tar.bz2

同样的，请使用md5sum命令检查压缩包是否正确：

md5sum gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf.tar.bz2

您应当在控制台中看到如下输出。

e1b129d60b2b8e6f398014057f514b34 gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf.tar.bz2

同样的，请注意MD5值应当为“e1b129d60b2b8e6f398014057f514b34”。

在控制台中输入如下命令，将工具链解压：

tar xvjf gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf.tar.bz2 -C ../

在控制台中输入如下命令，确认gdb工具是否正常：

/hot-pot/assistant/gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf/bin/aarch64-elf-gdb -v

如果一切正常，您将在控制台中看到如下输出：

GNU gdb (Linaro GDB 2016.02) 7.10.1.20160210-cvs

第五步，通过git下载源码。在控制台中输入如下命令：

cd /hot-pot/

git clone <https://code.csdn.net/xiebaoyou/dim-sum.git>

这一步很快就能够完成，因为DIM-SUM目前的代码规模还不算大:-)

第六步，检出HOT-POT代码，在控制台中输入如下命令：

git checkout -b hot-pot v0.1

该命令会在您的git仓库中创建一个hot-pot分支，并将v0.1版本的代码检出。其中v0.1版本是DIM-SUM的HOT-POT版本。

是不是迫不及待的想为HOT-POT编译出一个可以运行的版本？请接着看下一节。

### 编译HOT-POT

编译HOT-POT的方法很简单。如果您编译过Linux源码，应该对下面的命令非常熟悉：

cd /hot-pot/dim-sum/

make ARCH=arm64 qemu\_defconfig

make ARCH=arm64 EXTRA\_CFLAGS="-g -D\_\_LINUX\_ARM\_ARCH\_\_=8 -DCONFIG\_QEMU=1" CROSS\_COMPILE=/hot-pot/assistant/aarch64-linux-gnu/bin/aarch64-linux-gnu- Image dtbs

如果在控制台中看到如下输出信息，则表示编译成功：

CC usr/shell/sh\_symbol.o

CC usr/shell/sh\_task.o

CC usr/shell/sh\_memory.o

CC usr/shell/sh\_register.o

LD usr/shell/built-in.o

LD usr/built-in.o

LD vmlinux.o

MODPOST vmlinux.o

WARNING: modpost: Found 20 section mismatch(es).

To see full details build your kernel with:

'make CONFIG\_DEBUG\_SECTION\_MISMATCH=y'

GEN .version

CHK include/generated/compile.h

UPD include/generated/compile.h

CC init/version.o

LD init/built-in.o

LD .tmp\_vmlinux1

KSYM .tmp\_kallsyms1.S

AS .tmp\_kallsyms1.o

LD .tmp\_vmlinux2

KSYM .tmp\_kallsyms2.S

AS .tmp\_kallsyms2.o

LD vmlinux

SYSMAP System.map

SYSMAP .tmp\_System.map

OBJCOPY arch/arm64/boot/Image

Kernel: arch/arm64/boot/Image is ready

目前，HOT-POT借用了Linux的编译框架，因此输出的符号表文件仍然是“vmlinux”，正如上图中“LD vmlinux”一行所示。当然，如果您能够提交一个补丁来修正这些问题，我们会非常感激的。

小问题1.5：您为什么不详细解释一下编译命令，就象大多数书籍中那样？

在控制台中输入如下命令，看看HOT-POT镜像是否生成成功：

ls arch/arm64/boot/Image -al

预期的结果大概是这样的：

-rwxrwxr-x 1 xiebaoyou xiebaoyou 1066496 5月 21 10:07 arch/arm64/boot/Image

小问题1.6：我的天，生成的镜像竟然超过1M？

### 运行HOT-POT

要运行HOT-POT，您需要安装qemu。这为您免除了购买单板的需要。在控制台中，输入如下命令：

apt-get install qemu

安装完成后，请使用如下命令确认成功安装qemu模拟器：

qemu-system-aarch64 --version

您应当在控制台中看到如下输出：

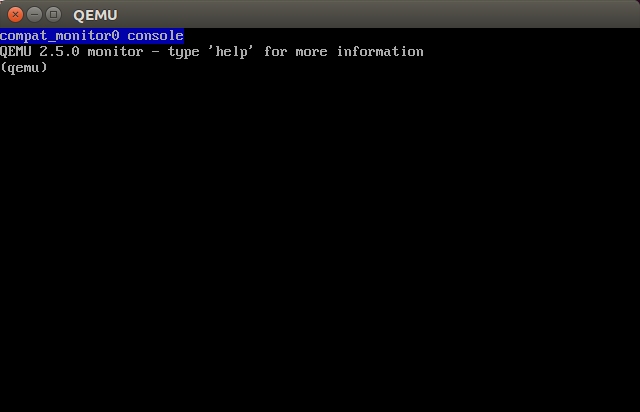
QEMU emulator version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.28), Copyright (c) 2003-2008 Fabrice Bellard

运行如下命令，在qemu中启动HOT-POT：

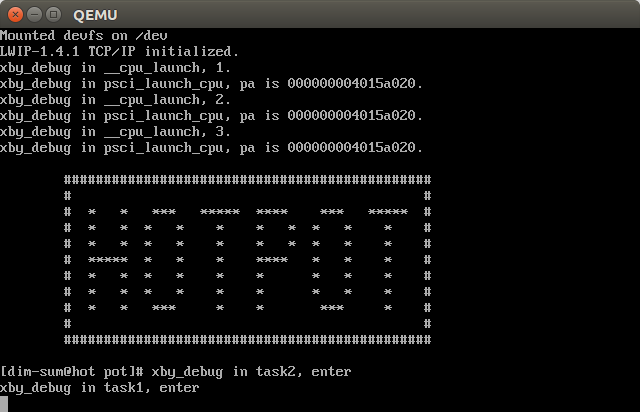
cd /hot-pot/dim-sum/

sudo qemu-system-aarch64 -machine virt -cpu cortex-a53 -m 512 -kernel arch/arm64/boot/Image -drive file=./disk.img,if=none,id=blk -device virtio-blk-device,drive=blk -device virtio-net-device,netdev=network0,mac=52:54:00:4a:1e:d4 -netdev tap,id=network0,ifname=tap0 --append "earlyprintk console=ttyAMA0 root=/dev/vda rootfstype=ext3 init=/bin/ash rw ip=10.0.0.10::10.0.0.1:255.255.255.0:::off"

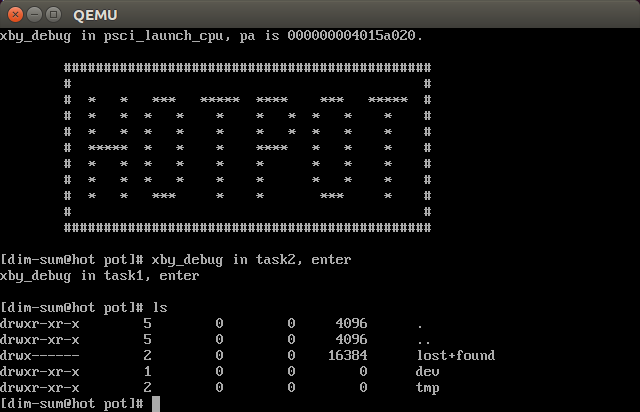
现在，您看到的应该是如下空白窗口：



别急，按“ctrl+alt+2”看看。激动人心的界面应当出现了，如下：



在这个界面中回车，并输入ls命令，您将会看到如下界面：



小问题1.7：但是请等一等，我想看看前面的输出，该怎么办？

看起来大功告成，但是似乎还缺少一点什么东西？

### 开始调试

在前面的步骤中，我们已经将gdb调试工具解压到：

/hot-pot/assistant/gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf/bin/aarch64-elf-gdb

现在是时候用用它了。

首先应当换一种方式启动HOT-POT，使用如下命令：

sudo qemu-system-aarch64 -machine virt -cpu cortex-a53 -m 512 -s -S -kernel arch/arm64/boot/Image -drive file=./disk.img,if=none,id=blk -device virtio-blk-device,drive=blk -device virtio-net-device,netdev=network0,mac=52:54:00:4a:1e:d4 -netdev tap,id=network0,ifname=tap0 --append "earlyprintk console=ttyAMA0 root=/dev/vda rootfstype=ext3 init=/bin/ash rw ip=10.0.0.10::10.0.0.1:255.255.255.0:::off"

请注意该命令中的“-s -S”参数，它会暂停HOT-POT的运行，并等待gdb调试。

在控制台中，按“ctrl+shift+t”启动一个新的控制台，我们称之为“调试控制台”。在调试控制台中，输入如下命令启动gdb，准备开始调试HOT-POT：

cd /hot-pot/dim-sum/

/hot-pot/assistant/gcc-linaro-5.3-2016.02-x86\_64\_aarch64-elf/bin/aarch64-elf-gdb vmlinux

在(gdb)提示符下，输入如下命令，连接到qemu：

target remote localhost:1234

您的调试控制台看起来是这样的：

(gdb) target remote localhost:1234

Remote debugging using localhost:1234

0x0000000040000000 in ?? ()

在(gdb)提示符下，输入“c”命令，启动HOT-POT。然后切换到qemu窗口，看看HOT-POT是不是已经正常启动了？

接下来，在调试控制台中，按“ctrl+c”，暂停HOT-POT的运行，并在(gdb)提示符下，输入“bt”命令，查看HOT-POT当前停留在什么地方？看起来应当是这样：

(gdb) bt

#0 cpu\_do\_idle () at arch/arm64/kernel/processor.S:12

#1 0xffffffc0000a8538 in default\_powersave () at kernel/sched/idle.c:15

#2 0xffffffc0000a85e8 in default\_idle () at kernel/sched/idle.c:27

#3 0xffffffc0000a863c in cpu\_idle () at kernel/sched/idle.c:48

#4 0xffffffc0001780c4 in start\_master () at init/main.c:96

#5 0x0000000040090240 in ?? ()

Backtrace stopped: previous frame identical to this frame (corrupt stack?)

当然了，在调试控制台中，您可以使用所有gdb调试命令。进行诸如单步跟踪、查看变量、查看寄存器、查看堆栈、切换CPU、汇编单步等等操作。

小问题1.8：在调试Linux内核时，我无论是用kgdb，还是qemu，发现单步跟踪时会杂乱无章的跳，有些变量值也看不到。但是HOT-POT中不会这样，宝友您有什么窍门？

## 向DIM-SUM操作系统提交补丁

想修改DIM-SUM的代码，并把它合入到DIM-SUM的git仓库吗？试着给DIM-SUM操作系统提交补丁吧。维护DIM-SUM操作系统的人，使用的都是汉语，沟通起来完全没有问题。并且他们都不是外星人，您不用觉得他们凶巴巴的:-)

### 心态

Paul在《深入理解并行编程》一书第11.1.2节中说，验证和测试工作都需要良好的心态。应当以一种破坏性的、甚至带一点仇恨的心理来验证代码，有时也应当考虑到：不少人的生命依赖于我们代码正确性的几率。总之，心态对事情的成败有重要的影响。

您在向DIM-SUM提交补丁之前，请保持如下正确的心态：

1、撇开DIM-SUM不谈。我们的代码，可能会影响不少人的生命，所以为任何项目编写代码，都一定要细心。

2、悲观的说，如果补丁做得不好，会影响自己的声誉，并且得不到足够的关注，最终会导致补丁没有被采纳。

3、乐观的说，DIM-SUM的维护者、开发者一般都比较Nice，如果你的水平真的比较牛的话。

4、更进一步乐观的说，您提交的高质量的补丁，可能会为您带来良好的声誉、满意的工作。

如果您和我一样，有着近乎自大的自信，想要在操作系统方面做出一些成绩，请仔细阅读后面的章节。

### 准备工作

跃跃欲试想要提交补丁的读者，一定已经熟悉Linux开发环境了。在此，作者假设您已经安装好Linux和git。这里推荐两份比较好的git参考资料：

http://git.oschina.net/progit/

https://item.jd.com/11615420.html

1、配置git用户名和邮箱

在配置用户名的时候，建议是“名在前，姓在后”，并且第一个拼音字母大写。例如我的用户名是：Baoyou Xie。

在配置邮箱时，请使用有意义的邮箱名，而不要用纯数字邮箱名。

以下是我的配置：

xiebaoyou@ThinkPad-T440$ git config -l | grep "user"

user.email=baoyou.xie@aliyun.com

user.name=Baoyou Xie

2、配置sendemail

您可以手工修改~/.gitconfig，或者git仓库下的.git/config文件，添加[sendemail]节。该配置用于指定发送补丁时用到的邮件服务器参数。

以下是我的配置，供参考：

[sendemail]

smtpencryption = tls

smtpserver = smtp.aliyun.com

smtpuser = baoyou.xie@aliyun.com

smtpserverport = 25

配置完成后，请用如下命令，向自己发送一个测试补丁：

git send-email your.patch --to your.mail --cc your.mail

3、下载源码

如果仅仅是为了阅读本书，而不想向DIM-SUM提交补丁，那么使用git直接从DIM-SUM主分支拉取代码就行了。

使用如下命令可以从主分支拉取DIM-SUM的代码：

git clone <https://code.csdn.net/xiebaoyou/dim-sum.git>

随时可以使用如下命令更新主分支代码：

git fetch origin master

但是，如果想参与到DIM-SUM的开发，那么就需要从多个git分支拉取代码。这是因为：主分支代码并不一定是最新的，如果基于这个代码制作补丁，很有可能不会顺利的合入到Maintainer那里。换句话说，如果您的代码分支没有与Maintainer保持一致，那么Maintainer有时会将补丁发回给您，要求您重新制作。所以，一般情况下，您需要再用以下命令，添加其他分支:

git remote add tag-name git-url

其中tag-name是您为分支添加的别名。git-url是DIM-SUM分支的URL路径。

随时可以使用如下命令更新分支代码：

git fetch tag-name

随着DIM-SUM的发展，不同的模块会有不同的Maintainer来维护。这些Maintainer会有自己的代码分支。可以在DIM-SUM源码目录\MAINTAINERS文件中，找到相应文件的维护者，及其git地址。

MAINTAINERS文件的格式与Linux类似。例如，Linux内核中，watchdog模块的信息如下：

WATCHDOG DEVICE DRIVERS

M: Wim Van Sebroeck <wim@iguana.be>

R: Guenter Roeck <linux@roeck-us.net>

L: linux-watchdog@vger.kernel.org

W: http://www.linux-watchdog.org/

T: git git://www.linux-watchdog.org/linux-watchdog.git

S: Maintained

F: Documentation/devicetree/bindings/watchdog/

F: Documentation/watchdog/

F: drivers/watchdog/

F: include/linux/watchdog.h

F: include/uapi/linux/watchdog.h

其中，git://www.linux-watchdog.org/linux-watchdog.git是其git地址。

1. 阅读Documentation/SubmittingPatches

认真阅读这个文件，对正确制作补丁来说很重要。

5、使用如下命令检出源码

git branch mybranch remote-branch

这个命令表示将remote-branch远程分支作为本地mybranch分支，作为我们工作的基础。在这个分支上制作补丁，更容易被Maintainer合入。

使用如下命令切换为本地mybranch分支：

git checkout mybranch

接下来，就可以修改本地代码，开始制作补丁了。

### 制作补丁

参与DIM-SUM的开发，可以从简单的事情入手。例如：

1. 消除编译警告。
2. 整理编码格式，例如注释里面的单词拼写错误、对齐不规范、代码格式不符合社区要求。

Linux社区里面的很多大牛，就是从消除Linux内核警告开始的。下面我们举一个简单的格式整理例子。

在kernel/sched/core.c的第192～193行，其代码看起来如下：

192 next = list\_first\_container(&sched\_runqueue\_list[idx],

193 struct task\_desc, run\_list);

其中第193行有两个问题：

1. 该行包含了84个字符，其中每个TAB键占用8个字符空间，超过了80个字符的限制。
2. 与上一行对齐有问题，排版太难看了。

我们删除该行前面几个TAB键，使其看起来如下：

192 next = list\_first\_container(&sched\_runqueue\_list[idx],

193 struct task\_desc, run\_list);

修改完成后，在控制台输入如下命令将补丁提交到本地git仓库：

git commit -a

然后使用如下命令生成补丁文件：

git format-patch -s -v 1 -1

生成的补丁内容如下：

cat v1-0001-.patch

From d75a0cea3945d79176645ce17748aebd5701a07e Mon Sep 17 00:00:00 2001

From: Baoyou Xie <baoyou.xie@aliyun.com>

Date: Mon, 21 May 2018 14:29:45 +0800

Subject: [PATCH v1] =?utf-8?q?=E8=B0=83=E5=BA=A6=EF=BC=9A=E6=95=B4?=

=?utf-8?q?=E7=90=86=E4=BB=A3=E7=A0=81=E6=A0=BC=E5=BC=8F?=

MIME-Version: 1.0

Content-Type: text/plain; charset=utf-8

Content-Transfer-Encoding: 8bit

在core.c第193行，代码行超过了80个字符，并且与上一行没有正确的对齐，因此应该删除其多余的前导TAB键。本补丁删除这些多余的TAB键以满足代码格式规范。

Signed-off-by: Baoyou Xie <baoyou.xie@aliyun.com>

---

kernel/sched/core.c | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/kernel/sched/core.c b/kernel/sched/core.c

index 3faa53c..c93df0d 100755

--- a/kernel/sched/core.c

+++ b/kernel/sched/core.c

@@ -190,7 +190,7 @@ need\_resched:

next = idle\_task\_desc[smp\_processor\_id()];

else

next = list\_first\_container(&sched\_runqueue\_list[idx],

- struct task\_desc, run\_list);

+ struct task\_desc, run\_list);

/\*\*

\* 什么情况下，二者会相等??

--

2.7.4

难道制作一个补丁就这么简单？我们可以将它发送给Maintainer了吗？

答案当然是否定的。这个补丁有如下问题：

1. 没有彻底解决模块中同类问题。
2. 补丁格式不正确。

首先我们应当将补丁回退。使用如下命令进行回退：

git reset HEAD~1

### 制作正确的补丁

实际上，在向Maintainer发送补丁之前，应当用如下对补丁进行检查：

./scripts/checkpatch.pl your.patch

对于刚才生成的补丁，我们会得到如下错误：

WARNING: Possible unwrapped commit description (prefer a maximum 75 chars per line)

#10:

在core.c第193行，代码行超过了80个字符，并且与上一行没有正确的对齐，因此应该删除其多余的前导TAB键。本补丁删除这些多余的TAB键以满足代码格式规范。

total: 0 errors, 1 warnings, 8 lines checked

错误在于：补丁描述行太长了，应当折行，使每一行低于75个字符。我们再次使用如下命令提交补丁：

git commit -a

在提交的时候，注意修改补丁描述，使其满足格式规范。反复制作补丁并使用checkpatch.pl检查其正确性，直到消除了所有警告为止。当然，极个别的警告是允许存在的。

使用checkpatch.pl仅仅能检查格式规范方面的错误。但是一个正确的补丁远不止格式正确这么简单。它还应该满足如下要求：

1. 一般情况下，同一个补丁只修改同一个模块的代码。

如果必须要同时修改多个模块中的代码，那么应该由所有模块的Maintainer同意，并确定由其中某一个Maintainer合入补丁。这种情况仅仅是特例。

但是怎么确定某个文件属于哪一个模块？您应当查看MAINTAINERS文件，里面有一个例子：

内核同步与互斥

M: Baoyou Xie <baoyou.xie@aliyun.com>

L: kernel@dim-sum.cn

T: git https://code.csdn.net/xiebaoyou/hot-pot.git core/locking

S: Maintained

F: kernel/locking/

可以看出：kernel/locking目录下的所有代码均属于“内核同步与互斥”模块。

1. 同一个补丁仅仅解决一个独立的问题。

不要试图在同一个补丁中解决多个问题。例如，既消除一个编译警告，又整理一行代码。

小问题1.9：但是，消除编译警告，以及整理代码，都仅仅修改了一行代码，并且位于同一个文件之中，也不能将它制作到同一个补丁中？如果不能，请告诉我正确的做法。

1. 同一个补丁必须完整的解决一个问题。

换句话说，不能将一个问题分拆到多个补丁中去。正如前一个例子所述，需要在一个补丁中将整个模块的格式全部整理完毕。如果补丁太大，可以考虑每个补丁整理其中一个文件。

1. 补丁不要太大，但这不是一个强制要求遵循的规则。

一般来说，一个补丁修改的代码行数，不要超过200行。不过此规则比较灵活。如果一个单独的问题确实需要修改超过200行代码，那么就突破这个规则吧。

要制作一个正确的补丁，还有一个问题比较重要：补丁的标题和描述。

补丁第一行是标题，它首先应当是模块名称。

但是我们怎么找到kernel/sched/core.c文件属于哪个模块？

可以试试下面这个命令，看看kernel/sched/core.c文件的历史补丁：

root@ThinkPad-T440:# git log kernel/sched/core.c

commit 0521afdc65cec3265827f68d637ed7d8b07061db

Author: Baoyou Xie <baoyou.xie@aliyun.com>

Date: Mon May 21 14:53:49 2018 +0800

调度：整理代码格式

在core.c第193行，代码行超过了80个字符，并且与上一行

没有正确的对齐，因此应该删除其多余的前导TAB键。

本补丁删除这些多余的TAB键以满足代码格式规范。

可以看到，kernel/sched/core.c文件所在的模块名称是“调度”。

其中第一行是标题，在模块名称后面是补丁标题，应当简洁，清楚的说明补丁的内容。当然，标题可以超过80个字符。

随后的内容是补丁描述符，要清楚的描述：

1. 为什么要制作这个补丁
2. 这个补丁是如何实现的

当然了，模块的Maintainer可能对补丁描述要额外的要求，您可能也有需要特殊说明的地方，都可以补充在描述中。这有点象写作文，既要求条理清楚，又没有成规。

### 发送补丁

在发送补丁前，我们需要用脚本再次检查一下补丁，确保其正确：

./scripts/checkpatch.pl your.patch

如果想要一次性生成并发送多个补丁，可以使用如下命令生成补丁：

git format-patch -s -v1 HEAD~2

在作者的环境中，上述命令生成了两个补丁：

v1-0001-DIM-SUM-Good-Start.patch v1-0002-.patch

然后用checkpatch.pl检查这两个补丁：

./scripts/checkpatch.pl v1-\*.patch

一切无误，可以使用准备将补丁发送给Maintainer了:-)

但是应该将补丁发给谁？这可以用get\_maintainer.pl来查看：

root@ThinkPad-T440:# ./scripts/get\_maintainer.pl v1-\*.patch

"GitAuthor: Baoyou Xie" <baoyou.xie@aliyun.com> (authored:2/1=100%,added\_lines:522/522=100%,removed\_lines:1/1=100%)

接下来，可以用git send-email命令发送补丁了：

git send-email v1-\*.patch --to baoyou.xie@aliyun.com --cc baoyou.xie@aliyun.com

需要注意分辨，哪些人应当作为邮件接收者，哪些人应当作为抄送者。在本例中，补丁是属于实验性质的，可以不抄送给邮件列表帐户。

提醒：您应当将补丁先发给自己，检查无误后再发出去。如果你有朋友在Linux社区有较高的威望，有补丁走查的经验，或者深度参与DIM-SUM的开发，那么也可以抄送给他，必要的时候，也许他能给您一些帮助。这有助于将补丁顺利的合入DIM-SUM。

重要提醒：本章讲述的，主要是实验性质的补丁，用于熟悉提交补丁的流程。真正重要的补丁，可能需要经过反复修改，才能合入DIM-SUM。并且，这需要您反复阅读本书后面章节中，对DIM-SUM内核的代码分析。最重要的一点，是需要您对DIM-SUM代码的熟练掌握。

## 获得帮助

如果您在下载源码、搭建环境、调试、提交补丁的过程中，遇到问题，可以通过如下途径获得帮助：

1. 通过git更新源码，并关注根目录下hot-pot.readme文件了解最新的帮助方式
2. 关注“操作系统黑客”公众号并留言
3. 关注作者的博客：<http://blog.chinaunix.net/uid/25845340.html>
4. 您也可以向baoyou.xie@aliyun.com发邮件来获得帮助

## 提醒

作者尽量将本书写得通俗易懂，以方便初学者入门。但是，要真正深入的理解任何一门学问，都需要花费大量的时间，做大量身体力行的练习，并且深入思考。因此，本书会提出一些让读者深入思考的小问题。这些小问题值得您在多次阅读本书后，认真回答。如果您真的想急切知道答案，请阅读附录中的答案。但是作者仍然提醒您，不要试图直接翻阅答案。我阅读了不少技术书籍，真正优秀的书都注重激发读者思考的习惯。

# **第** **2章**

# 算法基础

## 链表

## 哈希表

## 红黑树

## 基树

# **第** **3章**

# 计数与同步互斥

在多核系统中，甚至于在单核系统中，计数都不是一个可以忽略的简单问题。实际上，它是一个真正的大问题。需要读者对计算机硬件，特别是CPU缓存有一定的理解。

同步和互斥是操作系统正常运行的基础。在多任务操作系统，同时有大量运行任务存在。同步与互斥主要用于保护这些任务对共享资源的访问，以及运行时序的同步。这是多任务编程中的基本问题。除了任意务的同步与互斥之外，在操作系统设计中，还面临另一个重要问题，即中断与任务上下文之间的同步。

互斥主要用于任务/CPU核之间对共享资源的保护。当某个任务/CPU核在临界区中运行时，其它任务/CPU核就不能进入到临界区中运行，只能等到该任务/CPU核离开临界区后才可以进入临界区。一个最简单的例子是防止多个任务同时对某个全局变量进行递增操作。

同步则是指不同任务之间的若干程序片断，它们的运行必须严格按照规定的先后次序来运行。最简单的例子是生产者/消费者模型，只有生产者准备好数据以后，通过同步机制通知消费者继续进行下一步操作。

一般来说，可以使用同步机制来实现互斥原语。在Linux和DIM-SUM操作系统中，您都可以使用信号量来实现互斥/同步操作。但是，却不能使用同步原语来实现互斥操作。

很显然，可以通过互斥原语来实现对计数的访问。但是在操作系统中，还需要更复杂的手段，以保证系统的扩展性、性能。下一节将详细解释DIM-SUM操作系统中的计数。

## 计数

在《深入理解并行编程》一书，作者Paul E. McKenney故意给出一个简单的，但是却是错误的计数示例：

1 long counter = 0;

2

3 void inc\_count(void)

4 {

5 counter++;

6 }

7

8 long read\_count(void)

9 {

10 return counter;

11 }

在多核笔记本中，在多个CPU中并发调用上亿次inc\_count，然而最终counter计数值的最终值竟然只有5000多万，丢失了一半的计数！！！

小问题2.1：不可能，您的笔记本（哦，错了，是Paul 的笔记本）应该是x86架构吧，在x86系统中，递增操作不是“一条”汇编指令吗？

### 计数的难题

上述示例严重违反直觉，以至于您会发出质疑，以为作者在此胡言乱语。不仅仅是对于操作系统新手，甚至对于从事操作系统工作多年的老手来说，这个质疑或多或少都可能存在。

想要真正理解操作系统的底层构造，您还必须接受另一个离奇的事实。我们看一个例子：

假设系统中存在4个CPU，其中CPU 0反复的对全局变量进行递增操作，从0递增到10000。CPU 1～3在同一时刻读取变量的值。更进一步假设，CPU 1读取到值1234,那么CPU 2/3读取到的值是什么？

按照直觉来说，CPU 2/3也应当同时读取到1234这个值。然而事实并非如此，这两个CPU可能读取到1000，也可能读取到1001。更反直觉的是，可能是CPU2读取到1001而CPU3读取到1000。

对这两个例子的更进一步论述已经属于另外一门学问了，对此有兴趣的读者，可以阅读《深入理解并行编程》一书。

一个好消息是：这两个例子对于操作系统设计来说，虽然很重要，但是仅仅是对构造最底层的操作系统原语很重要，而这些原语已经由DIM-SUM实现。读者可以在参与DIM-SUM操作系统的开发时，只需要使用，而并不需要重新实现这些底层原语。

### 精确计数器

精确计数器类似于Linux中的原子变量。它可以对全局计数器进行原子的增加或者减小操作。其实现源代码位于include/dim-sum/accurate\_counter.h、arch/arm64/include/asm/accurate\_counter.h和arch/arm64/lib/accurate\_counter.c中。

#### 精确计数器的数据结构

精确计数器的数据结构定义如下：

1 struct accurate\_counter {

2 long counter;

3 };

该结构是如此简单，以至于您会怀疑是否值得出现在本书中。

小问题2.1：counter字段为什么不是int类型？

小问题2.2：为什么没有在counter字段声明中加上对齐指示？

#### 精确计数器API

下表是精确计数器提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| accurate\_read | 读取精确计数器的值 |
| accurate\_set | 设置精确计数器的值 |
| accurate\_inc | 递增精确计数器的值，并返回递增后的结果 |
| accurate\_dec | 递减精确计数器的值，并返回递减后的结果 |
| accurate\_add | 增加精确计数器的值，并返回增加后的结果 |
| accurate\_sub | 减少精确计数器的值，并返回减少后的结果 |
| accurate\_sub\_and\_test\_zero | 减少精确计数器的值。  如果结果为0,则返回true，否则返回 false。 |
| accurate\_dec\_and\_test\_zero | 递减精确计数器的值。  如果结果为0,则返回true，否则返回 false。 |
| accurate\_inc\_and\_test\_zero | 递增精确计数器的值。  如果结果为0,则返回true，否则返回 false。 |
| accurate\_add\_test\_negative | 增加精确计数器的值。  如果结果为负数，则返回true，否则返回 false。 |
| accurate\_add\_ifneq | 如果精确计数器的值不等于参数u，则增加精确计数器的值，并返回 true。否则返回false。 |
| accurate\_inc\_not\_zero | 如果精确计数器的值不等于0，则增加精确计数器的值，并返回 true。否则返回false。 |
| accurate\_cmpxchg | 将精确计数器的值与参数old进行比较，如果与old相等，则设置其值为new。 |
| accurate\_xchg | 将精确计数器的值设置为new，并返回其旧值 |

表2.1 精确计数器API

#### 精确计数器的实现

accurate\_read和accurate\_set的实现如下：

1 static inline long accurate\_read(struct accurate\_counter \*l)

2 {

3 return ACCESS\_ONCE(l->counter);

4 }

5

6 static inline void accurate\_set(struct accurate\_counter \*l, long i)

7 {

8 l->counter = i;

9 }

第3行的ACCESS\_ONCE能够确保：编译器在生成汇编代码的时候，会强制生成从内存中读取变量值的代码。而不会过度优化，使用汇编过程中临时保存在寄存器中的值。对编译器有兴趣的读者可以深入研究一下该关键字的原理。

accurate\_set看起来更简单，它就是简单的对精确计数器进行赋值。

小问题2.3：accurate\_set的实现一定有问题，这里为什么没有内存屏障？甚至没有ACCESS\_ONCE？

小问题2.4：accurate\_read的ACCESS\_ONCE也无法保证“原子”读取的语义？

小问题2.5：假设accurate\_read返回的值是a，此时系统中精确计数器中counter字段的值到底是a还其他什么值？

accurate\_inc、accurate\_dec、accurate\_add、accurate\_sub、accurate\_sub\_and\_test\_zero、accurate\_dec\_and\_test\_zero、accurate\_inc\_and\_test\_zero、accurate\_add\_test\_negative的实现是如果简单明了，请读者自行阅读相关代码。

下图是accurate\_add\_ifneq的实现代码:

1 static inline long accurate\_add\_ifneq(struct accurate\_counter \*l, long a, long u)

2 {

3 long c, old;

4

5 c = accurate\_read(l);

6 while (c != u && (old = arch\_accurate\_cmpxchg(l, c, c + a)) != c)

7 c = old;

8

9 return c != u;

10 }

在第5行，函数首先获得精确计数器的当前值到变量c中。

在第6行，进行如下判断:

* 精确计数器的值与参数u不相等，这应该是普遍的情况 。
* 将精确计数器与新值进行比较并交换，获取精确计数器的旧值并与临时变量c进行比较。这是为了确保在比较并交换的过程中，精确计数器的值没有被其他过程所修改。

一般情况下，第6行的判断都会由于第二个条件而退出。但是如果在比较并交换的过程中，真的有其他过程将精确计数器的值修改了，那么就会执行第7行并继续循环。

小问题2.6：STOP，有一个问题需要提出来：第一个条件存在的意义到底是什么？

第7行将精确计数器的当前值赋值给变量c。

在第9行，如果本函数成功的增加了精确计数器的值，则返回true，否则返回false。

所有的API都依赖于体系结构相关的实现函数：arch\_accurate\_add、arch\_accurate\_sub、arch\_accurate\_xchg、arch\_accurate\_cmpxchg。对于arm64架构来说，这些函数的实现位于arch/arm64/lib/accurate\_counter.c。

读者需要仔细分析这些函数的实现。我们以arch\_accurate\_add为例，来看看这些函数的实现到底有什么地方需要注意。arch\_accurate\_add函数的代码如下：

1 long arch\_accurate\_add(long i, struct accurate\_counter \*v)

2 {

3 unsigned long tmp;

4 long result;

5

6 asm volatile("// arch\_accurate\_add\n"

7 "1: ldxr %0, %2\n"

8 " add %0, %0, %3\n"

9 " stlxr %w1, %0, %2\n"

10 " cbnz %w1, 1b"

11 : "=&r" (result), "=&r" (tmp), "+Q" (v->counter)

12 : "Ir" (i)

13 : "memory");

14

15 smp\_mb();

16 return result;

17 }

第1行是函数定义，该函数的功能是将参数i原子的增加到精确计数器v中，并返回精确计数器的新值。

第6行声明了汇编代码块，其中volatile关键字指示编译器：汇编代码块不能与相邻代码进行优化，需要原封不动的保留汇编代码块的原貌。“//”后面的arch\_accurate\_add是汇编注释，这样可以在汇编调试的时候看到函数名称，易于跟踪调试。

第7行的汇编代码将精确计数器的counter字段值加载到result变量中。读者需要注意，此处的ldxr是arm64架构的排它装载指令。该指令与stlxr配对使用，类似于mips架构中的链接加载/条件存储指令对。

小问题2.7：到底该怎么样来理解ldxr/stlxr指令对？

第8行将参数i的值添加到临时变量result中。

第9行将result的值保存到精确计数器的counter字段中。排它存储的结果存放在tmp变量中。

第10行查看排它存储的结果。如果在第9行的排它存储过程中，发生了如下行为，将跳转到第7行进行重试：

其他CPU同时在更新精确计数器的值，产生了冲突。

在第8～9行之间，产生中断。

第11行声明了输出部的寄存器变量。要理解这一句话，需要读者对gcc汇编器的原理有所了解。相关背景知识请参阅其他书籍。简单的说，在输出部中声明的变量，其寄存器会被汇编块所修改，以指示汇编器的编译行为。

第12行声明了汇编块的输入部。由于在整个汇编块中，没有修改变量i的值，因此可以放心的将其声明在输入部。

第13行的声明告诉汇编器，相应的汇编块对内存进行了更新，破坏了内存中的内容。因此不能将代码块后面的语句与前面的语句进行编译乱序。

小问题2.8：第13行的语句真的有用吗？有谁在工程实践中遇到过相关问题？

第15行的内存屏障，确保在SMP系统中，其他CPU核上面看到本CPU中正确的内存顺序。

小问题2.9：作者您在说什么，某些版本的Linux就没有smp\_mb()这一行。难道您比Linus还聪明？

第16行简单的返回精确计数器的新值。

小问题2.9：显然，精确计数器是通过一个循环来实现的。这里会不会存在饥饿现象？

小问题2.10：为什么不将arch\_accurate\_sub实现为如下形式？现有的实现不太符合《代码整洁之道》所讲的编程原则。

1 long arch\_accurate\_sub(long i, struct accurate\_counter \*v)

2 {

3 return arch\_accurate\_add(-i, v);

4 }

### 近似计数器

#### 近似计数器的设计原理

精确计数器虽然能够确保计数器的值准确而无疑义，但这是以牺牲性能为前提的。主要有如下引起性能损失的方面：

1. 精确计数器在进行计数的时候，使用了一个循环语句块。这样的循环会生成条件判断和分支跳转的代码。这样的代码不能充分发挥CPU指令流水线的最大性能。
2. 排它装载/条件存储指令需要在多CPU之间传递电信号。与普通的寄存器数据访问指令相比，这样的指令所消耗的指令周期至少多一个数量级。
3. 内存屏障指令需要与CPU内的写缓冲进行交互，并影响CPU之间的内存一致性。它既难用，也影响性能。

从另一方面来说，某些场景并不需要严格准确的计数。例如，一段时间内的接收/发送的报文数量，系统已经使用的内存页数量。这样的统计值，误差保持在数十之内，并不影响系统正常使用。

为此，近似计数器应运而生。

近似计数器的原理，就是避免使用重量级的排它装载/条件存储指令对。而是使用每CPU变量来保存每个CPU中的计数值。调用者在修改计数值的时候，仅仅会修改本CPU的计数值。

当然，这会带来一问题：近似计数器在所有核中的全局计数值难于计算。这包含两个方面的问题：

1. 全局计数值计算起来比较复杂
2. 全局计数值难于计算精确

#### 近似计数器的数据结构

近似计数器的数据结构定义如下：

1 struct approximate\_counter {

2 struct smp\_lock lock;

3 long count;

4 long \*counters;

5 };

这些字段的含义如下：

1. lock字段用于保护近似计数器本身。
2. count字段表示近似计数器的全局值。
3. counters是每CPU变量指针。

小问题2.11：为什么counters字段是每CPU变量指针，而不是每CPU变量？

#### 近似计数器API

下表是近似计数器提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| approximate\_counter\_init | 对近似计数器进行初始化 |
| approximate\_counter\_destroy | 对近似计数器进行销毁 |
| approximate\_counter\_mod | 修改近似计数器的值 |
| approximate\_counter\_read | 读取近似计数器的值 |
| approximate\_counter\_read\_positive | 读取近似计数器的值，并确保结果为正数 |
| approximate\_counter\_inc | 将近似计数器的值加1 |
| approximate\_counter\_dec | 将近似计数器的值减1 |

#### 近似计数器的实现

近似计数器的实现代码位于include/dim-sum/approximate\_counter.h和kernel/count/approximate\_counter.c中。

approximate\_counter\_init和approximate\_counter\_destroy函数的实现如下：

1 inline void approximate\_counter\_init(struct approximate\_counter \*fbc)

2 {

3 smp\_lock\_init(&fbc->lock);

4 fbc->count = 0;

5 fbc->counters = alloc\_percpu(long);

6 }

7

8 static inline void approximate\_counter\_destroy(struct approximate\_counter \*fbc)

9 {

10 free\_percpu(fbc->counters);

11 }

第3行初始化近似计数器的自旋锁。

第4行将近似计数器的全局计数值设置为0。

第5行分配一个类型为long的每CPU变量，并将每CPU变量指针赋值到counters字段。

在approximate\_counter\_destroy函数的第10行，释放近似计数器所使用的每CPU变量。

小问题2.12：实际上，近似计数器的当前版本没有考虑分配每CPU变量失败的情况，应该怎么来完善它？

近似计数器的读取函数实现如下：

1 static inline long approximate\_counter\_read(struct approximate\_counter \*fbc)

2 {

3 return fbc->count;

4 }

5

6 static inline long approximate\_counter\_read\_positive(struct approximate\_counter \*fbc)

7 {

8 long ret = fbc->count;

9

10 barrier(); /\* Prevent reloads of fbc->count \*/

11 if (ret > 0)

12 return ret;

13 return 1;

14 }

approximate\_counter\_read函数的第3行简单的返回近似计数器的全局计数值。

approximate\_counter\_read\_positive函数的第8行首先获取近似计数值的当前值。

第10行的编译屏障确保：第11～12行并不会重新加载近似计数器的全局计数值并返回全局计数值。实际上，approximate\_counter\_read\_positive函数确保返回值是正数，如果意外的返回近似计数器的全局计数值，结果可能为0或者负数。

小问题2.13：调用者既然想要调用approximate\_counter\_read\_positive函数，那么该计数器一定代表了某个正数的统计值，例如处于某种状态的人员数量。为什么计数器的count字段会成为负数，因而需要在这里小心的处理？

如果第11行判断全局计数值为正数，就在第12行将其返回，否则在第13行返回1,函数始终返回一个正数。

近似计数器的计数相关函数实现如下：

1 static inline void approximate\_counter\_inc(struct approximate\_counter \*fbc)

2 {

3 approximate\_counter\_mod(fbc, 1);

4 }

5

6 static inline void approximate\_counter\_dec(struct approximate\_counter \*fbc)

7 {

8 approximate\_counter\_mod(fbc, -1);

9 }

10

11 void approximate\_counter\_mod(struct approximate\_counter \*fbc, long amount)

12 {

13 long count;

14 long \*pcount;

15

16 pcount = hold\_percpu\_ptr(fbc->counters);

17 count = \*pcount + amount;

18 if (count >= FBC\_BATCH || count <= -FBC\_BATCH) {

19 smp\_lock(&fbc->lock);

20 fbc->count += count;

21 smp\_unlock(&fbc->lock);

22 count = 0;

23 }

24 \*pcount = count;

25 loosen\_percpu\_ptr(fbc->counters);

26 }

其中approximate\_counter\_inc和approximate\_counter\_dec是对approximate\_counter\_mod的简单封装。

在approximate\_counter\_mod的第16行，获得每CPU变量的引用。

小问题2.14：如果不获取每CPU变量的引用，会发生什么错误？

第17行计算当前CPU计数值。

第18行判断当前CPU计数值，如果计数值过大或者过小，将会对全局计数器的准确性带来较大的影响，就执行第19～22行的代码块。

第19行获取近似计数器的自旋锁。

第20行更新近似计数器的全局计数值。

第21行释放近似计数器的自旋锁。

由于当前CPU计数值已经计入到全局计数值中，因此第22行将当前CPU计数值置为0。

第24行将当前CPU计数值设置为临时变量count的值。

最后，第25行释放每CPU变量的引用。

小问题2.15：获取/释放每CPU变量的引用，是否也需要一个全局精确计数器？这难道不会导致精确计数器也变得过于重量级，以至于失去它的存在意义？

小问题2.16：中断处理函数中可以调用approximate\_counter\_mod吗？如果可以，应该注意什么？

### 引用计数

#### 引用计数的设计原理

引用计数主要用于存在性保证。一个典型的应用场景是：

线程初次打开文件时，会在文件系统的哈希表中创建文件节点描述符，并将其引用计数设置为1。随后的调用者在打开同一个文件时，不会再次创建这样的文件节点描述符，而是简单的将文件节点描述符引用计数加1。

在每个线程关闭文件对象时，会将文件节点描述符的引用计数减1。显然，最后一个引用文件节点描述符的调用者，有责任释放相应的文件节点描述符。

由于不同的调用者可以多次打开同一个文件，并且不能确定文件被打开的次数。引用计数的目的，就是跟踪描述符的引用计数，确保既不会提前释放描述符，也不会造成内存泄漏。

#### 引用计数的实现

引用计数的实现基础是精确计数器。它的数据结构是如此简单：

1 struct ref\_count {

2 struct accurate\_counter count;

3 };

引用计数有两个简单的API：

1 void ref\_count\_init(struct ref\_count \*ref)

2 {

3 accurate\_set(&ref->count, 1);

4 }

5

6 static inline unsigned long get\_ref\_count(struct ref\_count \*ref)

7 {

8 return accurate\_read(&ref->count);

9 }

小问题2.17：如果在调用ref\_count\_init的同时，另一个线程修改引用计数值会怎么样？

小问题2.18：get\_ref\_count看起来是“原子的”读取值，它读到的到底是什么样的值？这个API设计出来的目的是什么？

1 void ref\_count\_hold(struct ref\_count \*ref)

2 {

3 accurate\_inc(&ref->count);

4 }

5

6 void ref\_count\_loosen(struct ref\_count \*ref,

7 void (\*loosen) (struct ref\_count \*ref))

8 {

9 if (accurate\_dec\_and\_test\_zero(&ref->count))

10 loosen(ref);

11 }

上图是获取引用计数以及释放引用计数的实现。

ref\_count\_hold函数获取引用计数，它在第3行简单将精确计数器加1。

ref\_count\_loosen函数释放引用计数。它在第9行将精确计数器减1。如果引用计数值被递减到0，则在第10行调用回调函数loosen将描述符释放。

小问题2.19：ref\_count\_loosen的实现有点丑陋，为什么每次释放引用计数的时候，都要传递loosen参数？将loosen参数直接传递给ref\_count\_init，这看起来是一个好主意？

## 内核互斥原语

### 每CPU变量

#### 每CPU变量为何重要

顾名思义，每CPU变量就是为每个CPU构造变量的副本。这样，多个CPU相互操作各自的副本，互不影响。

每CPU变量并不是为了在CPU或者线程之间进行互斥。相反的，它在是为了减少内核互斥而设计出来的一种原语。

精确计数器为了在所有CPU之间维护一个全局的精确计数值，它利用了CPU提供的排它加载/条件存储指令对，以及内存屏障这样的重量级CPU指令来实现。这严重的影响了CPU性能。

即使在x86这样的体系架构中，仅仅使用了lock前缀的内存操作指令来实现精确计数器，这看起来很简单。然而，这样的实现，可能消耗数百个CPU周期。与不带lock前缀的内存操作指令相比，足足下降了两个数量级的性能。

为此，每CPU变量避免使用这类消耗巨大的指令或者指令对。它仅仅操作本CPU对应的变量，因此避免了在CPU之间进行同步电信号的传输，极大的提高了性能。

在并行编程的世界中，这被称为“数据所有权”技术。正如Paul E.Mckenney在《深入理解并行编程》第8.6节所说：数据所有权可能是最不起眼的同步机制。当使用得当时，它能提供无与伦比的简单性、性能和可扩展性。也许它的简单性使它没有得到应有的尊重。

是的，我们应当更加尊重每CPU变量这样的数据所有权机制。

#### 静态每CPU变量的实现

静态每CPU变量，是指那些全局静态分配的每CPU变量。由于在编译期间即能够确定这些变量的大小和位置，因此我们可以将这些变量放到统一的数据段中。与动态每CPU变量相比，它能减少动态分配失败的风险，以及更快的性能。

实现静态每CPU变量的最简单方法，是利用数组。每个要访问每CPU变量的CPU，可以使用CPU编号为索引，访问这个数组。

然而，这种方法会形成CPU缓存的伪共享，这样的伪共享使得每CPU变量的性能优势大打折扣。除非我们将数组的每个元素都进行缓存行对齐。但是这样的对齐操作会带来内存的巨大浪费。

因此，现代操作系统在设计每CPU变量这样的机制时，使用了一些小技巧。DIM-SUM显然也不例外。

在编译阶段，DIM-SUM会为每CPU变量指定一个特殊的data段”data..percpu“，段的名称由PER\_CPU\_BASE\_SECTION宏确定。在系统加载阶段，DIM-SUM会将”data..percpu“段复制数份，确保每个CPU均有一份副本。如下图所示：

**ELF文件**

代码段

......

数据段

data..percpu段

per\_cpu\_var\_start

per\_cpu\_var\_start

**运行空间**

代码段

......

数据段

data..percpu段

data..percpu段副本1

data..percpu段副本2

data..percpu段副本...

data..percpu段副本MAX\_CPU - 1

系统加载时，初始化每CPU变量的代码如下：

1 void \_\_init init\_per\_cpu\_offsets(void)

2 {

3 unsigned long size, i;

4 char \*ptr;

5

6 size = ALIGN(per\_cpu\_var\_end - per\_cpu\_var\_start, SMP\_CACHE\_BYTES);

7

8 ptr = alloc\_boot\_mem\_permanent(size \* (MAX\_CPUS - 1), /\* -1!! \*/

9 SMP\_CACHE\_BYTES);

10

11 per\_cpu\_var\_offsets[0] = 0;

12 for (i = 1; i < MAX\_CPUS; i++, ptr += size) {

13 per\_cpu\_var\_offsets[i] = ptr - per\_cpu\_var\_start;

14 memcpy(ptr, per\_cpu\_var\_start, per\_cpu\_var\_end - per\_cpu\_var\_start);

15 }

16 }

在函数第6行，系统计算data..percpu段的长度，并保存到临时变量size中。其中，per\_cpu\_var\_start是data..percpu段的起始位置，per\_cpu\_var\_end是data..percpu段的结束位置。

ALIGN宏将段长度按缓存行长度对齐。这样可以避免data..percpu段末端的每CPU变量与其他变量共享缓存行，引起缓存行伪共享。

第8行，为第1~MAX\_CPUS这些CPU核分配每CPU变量副本空间。由于分配的空间比较大，因此使用alloc\_boot\_mem\_permanent函数，在启动阶段分配大内存空间。当然，这段内存空间也是以缓存行对齐。

per\_cpu\_var\_offsets数组保存了每CPU变量副本与per\_cpu\_var\_start之间的偏移。对于CPU 0来说，这个偏移值当然为0。因此第11行将元素0设置为0。

在第12行，函数进入循环，遍历CPU 1 ~ MAX\_CPUS - 1，为每个CPU设置其每CPU变量副本的偏移值。

第13行记录当前CPU的每CPU变量副本的偏移值。

第14行，将data..percpu段的内容复制到每CPU变量副本中。

在init\_per\_cpu\_offsets函数分配完成每CPU变量副本以后，系统还会在start\_arch和start\_slave初始化函数中调用set\_this\_cpu\_offset函数。其中start\_arch函数会在CPU 0初始化过程中被调用，start\_slave会在其他CPU初始化过程中被调用：

1 static inline void set\_this\_cpu\_offset(unsigned long off)

2 {

3 asm volatile("msr tpidr\_el1, %0" :: "r" (off) : "memory");

4 }

set\_this\_cpu\_offset仅仅将每CPU变量副本地址设置到tpidr\_el1寄存器中。

要访问每CPU变量，可以调用per\_cpu\_var/this\_cpu\_var宏。

其中per\_cpu\_var宏可以根据CPU编号来访问每CPU变量副本，相关的代码如下所示：

1 #ifndef per\_cpu\_var\_offsets

2 extern unsigned long per\_cpu\_var\_offsets[MAX\_CPUS];

3 #define per\_cpu\_offset(x) (per\_cpu\_var\_offsets[x])

4 #endif

5

6 #define per\_cpu\_var(var, cpu) \

7 (\*SHIFT\_PERCPU\_PTR(&(var), per\_cpu\_offset(cpu)))

在第7行，该宏首先取得每CPU变量的地址，并加上每CPU变量副本的偏移值，获得副本的地址。然后使用\*操作符，最终获得每CPU变量副本的左值或者右值。

小问题2.20：per\_cpu\_var\_offsets是一个数组，将它用于每CPU变量？作者您是不是在骗人，刚刚强调了不能在每CPU机制中使用数组，否则会产生缓存行伪共享的问题？

this\_cpu\_var宏可以访问线程所在CPU的每CPU变量副本，相关的代码如下所示：

1 static inline unsigned long \_\_this\_cpu\_offset(void)

2 {

3 unsigned long off;

4

5 asm("mrs %0, tpidr\_el1" : "=r" (off) :

6 "Q" (\*(const unsigned long \*)current\_stack\_pointer));

7

8 return off;

9 }

10

11 #define this\_cpu\_offset \_\_this\_cpu\_offset()

12

13 #ifndef arch\_raw\_cpu\_ptr

14 #define arch\_raw\_cpu\_ptr(ptr) SHIFT\_PERCPU\_PTR(ptr, this\_cpu\_offset)

15 #endif

16

17 #define this\_cpu\_var(ptr) \

18 ({ \

19 \_\_verify\_pcpu\_ptr(ptr); \

20 arch\_raw\_cpu\_ptr(ptr); \

21 })

在第19行，this\_cpu\_var宏首先利用编译器审查传入的指针是否真的是每CPU变量。

第20行，调用体系架构相关的arch\_raw\_cpu\_ptr宏获得当前CPU变量副本地址。

对于 ARM64架构中的静态每CPU变量来说，当前CPU上的所有每CPU变量副本偏移地址都是固定的，并且保存tpidr\_el1寄存器中。因此\_\_this\_cpu\_offset函数在第5行简单的返回该寄存器的值即可。

第6行的声明实际上没有真实的意义，然而该行是有必要的。这一行编译指示语句可以指示编译器，这里有一个伪装的对current\_stack\_pointer变量的访问，因此具有编译屏障的作用。这可以避免在第5行的asm语句中使用volatile关键字，这样允许编译器使用前次调用缓存的tpidr\_el1寄存器值。

小问题2.21：this\_cpu\_var和per\_cpu\_var的实现有什么奇怪的地方？是否妥当？

最后，作者希望简单的描述一下静态每CPU变量的定义及声明方法：

1 #define DECLARE\_PER\_CPU(type, name) \

2 DECLARE\_PER\_CPU\_SECTION(type, name, "")

3

4 #define DEFINE\_PER\_CPU(type, name) \

5 DEFINE\_PER\_CPU\_SECTION(type, name, "")

在访问每CPU变量副本前，您需要调用hold\_percpu\_var。相应的，使用完毕以后，您也需要调用loosen\_percpu\_var：

1 #define hold\_percpu\_var(var) (\*({ \

2 preempt\_disable(); \

3 &\_\_get\_cpu\_var(var); }))

4

5 #define loosen\_percpu\_var(var) do { \

6 (void)&(var); \

7 preempt\_enable(); \

8 } while (0)

关键的代码位于第2/7行，分别表示关闭/打开抢占。

小问题2.22：为什么搞的这么复杂，还需要考虑抢占？

#### 动态每CPU变量的实现

有时候，我们需要动态分配每CPU变量。例如，在动态加载的驱动代码中。

动态每CPU变量使用了私有数据结构，调用者不能直接访问该结构：

1 struct dynamic\_percpu {

2 void \*ptrs[MAX\_CPUS];

3 void \*data;

4 };

其中ptrs字段保存了每CPU变量副本的地址，data字段暂未用。

下表是动态每CPU变量提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| alloc\_percpu | 分配动态每CPU变量 |
| free\_percpu | 销毁动态每CPU变量 |
| hold\_percpu\_ptr | 获得动态每CPU变量的副本引用 |
| loosen\_percpu\_ptr | 释放动态每CPU变量的副本引用 |

分配动态每CPU变量的实现代码如下：

1 void \*\_\_alloc\_percpu(size\_t size, size\_t align)

2 {

3 struct dynamic\_percpu \*pdata;

4 int i;

5

6 pdata = kmalloc(sizeof (\*pdata), PAF\_KERNEL | \_\_PAF\_ZERO);

7 if (!pdata)

8 return NULL;

9

10 for (i = 0; i < MAX\_CPUS; i++) {

11 if (!cpu\_possible(i))

12 continue;

13 pdata->ptrs[i] = kmalloc(size, PAF\_KERNEL | \_\_PAF\_ZERO);

14

15 if (!pdata->ptrs[i])

16 goto oom;

17 }

18

19 return (void \*)(~(unsigned long)pdata);

20

21 oom:

22 while (--i >= 0) {

23 if (!cpu\_possible(i))

24 continue;

25 kfree(pdata->ptrs[i]);

26 }

27 kfree(pdata);

28

29 return NULL;

30 }

31

32 #define alloc\_percpu(type) \

33 (typeof(type) \_\_percpu \*)\_\_alloc\_percpu(sizeof(type), \_\_alignof\_\_(type))

第32行是alloc\_percpu的声明。

第33行简单的调用\_\_alloc\_percpu分配动态每CPU变量。

第5行调用kmalloc分配私有动态每CPU变量描述符，如果在第7行判断分配失败，则在第8行返回NULL。

如果分配成功，则在第10行开始循环，为每一个CPU分配每CPU变量副本。

第11行判断CPU是否存在，如果不存在则在第12行跳转到下一个CPU进行处理。

小问题2.23：竟然还有不存在的CPU？

第13行为当前CPU分配每CPU变量副本。

第14行判断是否分配成功。如果不成功，则跳转到第21行释放已经分配的副本。

第19行将私有数据结构取反码以后返回给调用者。

小问题2.24：为什么要在第19行取反码再返回给调用者？

free\_percpu是销毁动态每CPU变量的接口，其代码实现如下：

1 void free\_percpu(const void \*ptr)

2 {

3 struct dynamic\_percpu \*pdata;

4 int i;

5

6 pdata = (struct dynamic\_percpu \*)(~(unsigned long)ptr);

7 for (i = 0; i < MAX\_CPUS; i++) {

8 if (!cpu\_possible(i))

9 continue;

10 kfree(pdata->ptrs[i]);

11 }

12 kfree(pdata);

13 }

第6行首先将传入的每CPU变量参数取反，获得其私有数据结构。

第7行遍历所有CPU，释放其每CPU变量副本。

第8行判断当前CPU是否存在，如果不存在，则在第8行跳转到下一个CPU进行处理。

第10行释放当前CPU的每CPU变量副本。

第12行释放私有数据结构。

与静态每CPU变量类似，调用者应当在访问动态每CPU变量之前，调用hold\_percpu\_ptr以获得动态每CPU变量的副本引用。在访问完成以后调用loosen\_percpu\_ptr释放对动态每CPU变量的副本引用。

1 #define \_\_percpu\_ptr(ptr, cpu) \

2 ({ \

3 struct dynamic\_percpu \*\_\_p = \

4 (struct dynamic\_percpu \*)~(unsigned long)(ptr); \

5 (\_\_typeof\_\_(ptr))\_\_p->ptrs[(cpu)]; \

6 })

7

8 #define hold\_percpu\_ptr(var) ({ \

9 int cpu = get\_cpu(); \

10 \_\_percpu\_ptr(var, cpu); })

11

12 #define loosen\_percpu\_ptr(var) do { \

13 (void)(var); \

14 put\_cpu(); \

15 } while (0)

其中第9行调用get\_cpu获得当前CPU编号，并禁止抢占。

第14行调用put\_cpu打开抢占。

### 自旋锁

#### 自旋锁的设计

自旋锁是为了防止多处理器并发而引入的一种锁，它在内核中大量应用于中断处理等部分。简单的说，它允许CPU执行一个死循环等待锁，直到锁被其他CPU释放。

在glibc库中，也有类似的自旋锁实现：pthread\_spin\_lock。

小问题2.25：细心的读者是否认真阅读过pthread\_spin\_lock的实现？它有没有什么BUG？

DIM-SUM目前的自旋锁实现是参考Linux排队自旋锁实现的。详情可以参见http://lwn.net/Articles/267968/。

简单的自旋锁实现，可用一个整数来表示自旋锁。当为0时表示这个锁是可被获得的。要获取锁的CPU可以使用原子比较并交换指令将其值置为1,以表示锁被占用。如果不幸有多个CPU同时竞争自旋锁，那么仅仅只有其中一个CPU能成功执行原子比较并交换指令。这是由硬件来保证的。竞争失败的CPU将不得不继续循环重试，直到锁就得可用的时候，重新开始下一轮争抢。

当临界区的代码执行完毕后，锁的持有者会将锁的值设为0来释放这个锁。

这种实现效率很高，在没有锁争抢的情况下尤其如此。然而这种方法有一个弊端：它缺少公平性。当锁被释放后，如果有多个等待者在争抢自旋锁，有可能第一个等待者并不能立即获得自旋锁。相反的，最后一个等待者可能立即获得锁。并且，这样的不公平性可能会持续发生于某一个CPU上，最终导致饥饿现象产生。

小问题2.26：所谓的不公平性，只不过是理论推算而已。不客气的说，就是拍脑袋出来的结论，硬件设计者不会做一些简单的公平性保证？

这样的不公平性会导致不可预测的延迟产生。这在实时系统中不可被接受。

解决这个问题的办法是实现一种“排队自旋锁”。在排队自旋锁中，包含两个u16类型的字段，分别是“next”字段和“owner”字段。

如果将排队自旋锁与银行叫号机进行类比，那么next字段表示叫号机分发给排队者的号，owner字段则表示银行工作人员正在服务的号。

“next”字段和“owner”字均被初始化为0。获取锁的函数首先记录下锁的当前值，然后使用原子操作指令将next字段加1。如果next字段在加1之前等于owner字段，锁就会被当前CPU获得。否则CPU就会自旋，直到owner字段增长到合适的值。

释放锁操作很简单，只需要将owner字段加1即可。

与简单自旋锁实现相比，排队自旋锁总是需要修改next字段，因此有一些轻微的开销。但是它带来了公平性和确定性，这样的代价是值得的。

#### 自旋锁的数据结构

自旋锁的数据结构，如下所示：

1 struct arch\_smp\_lock {

2 u16 next;

3 u16 owner;

4 } \_\_attribute\_\_((aligned(4)));

5

6 struct smp\_lock {

7 struct arch\_smp\_lock lock;

8 };

自旋struct smp\_lock来表示。目前，它的实现完全依赖于体系架构，因此只包含一个名为lock的struct arch\_smp\_lock数据结构。当然，您可以试着在里面添加一些新的字段，以实现更多的功能。例如，自旋锁死锁检测功能。

struct arch\_smp\_lock包含两个字段：next和owner。其含义已经在上一节中描述过。

第4行的编译指示要求编译器将数据结构进行4字节对齐。这是因为自旋锁相关代码需要对该结构进行原子操作。一般的，硬件设计工程师要求将这样的数据结构进行对齐。该结构大小是4字节，因此需要4字节对齐。

#### 自旋锁API

下表是自旋锁提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| SMP\_LOCK\_UNLOCKED | 初始化自旋锁，未锁 |
| smp\_lock\_init | 初始化自旋锁，未锁 |
| smp\_lock\_is\_locked | 自旋锁是否处于锁定状态 |
| assert\_smp\_lock\_is\_locked | 自旋锁处于锁定状态的断言 |
| smp\_lock | 获取自旋锁 |
| smp\_lock\_irqsave | 获取自旋锁，并关中断，保存当前中断状态 |
| smp\_lock\_irq | 获取自旋锁，并关中断 |
| smp\_trylock | 试图获得自旋锁，如果成功则返回1,否则返回0 |
| smp\_trylock\_irqsave | 试图获得自旋锁，如果成功则关中断并保存当前中断状态 |
| smp\_unlock | 释放自旋锁 |
| smp\_unlock\_irqrestore | 释放自旋锁并恢复中断状态 |
| smp\_unlock\_irq | 释放自旋锁，并打开中断 |

小问题2.27：smp\_lock\_is\_locked判断自旋锁是否处于锁定状态，这里有一个疑问：在判断的过程中，自旋锁可能会被其他CPU不停的获取/释放，并且由于缓存一致性的原因，这里的判断结果并不一定可靠。这样的API还有何用处？

#### 自旋锁的实现

SMP\_LOCK\_UNLOCKED、smp\_lock\_init、smp\_lock\_is\_locked、assert\_smp\_lock\_is\_locked的实现代码如下：

1 #define \_\_SMP\_LOCK\_INITIALIZER(lockname) \

2 { \

3 .lock = \_\_ARCH\_SMP\_LOCK\_UNLOCKED, \

4 }

5

6 #define SMP\_LOCK\_UNLOCKED(lockname) \

7 (struct smp\_lock) \_\_SMP\_LOCK\_INITIALIZER(lockname)

8

9 static inline void smp\_lock\_init(struct smp\_lock \*lock)

10 {

11 \*(lock) = SMP\_LOCK\_UNLOCKED(lock);

12 }

13

14 static inline int smp\_lock\_is\_locked(struct smp\_lock \*lock)

15 {

16 return arch\_smp\_lock\_is\_locked(&lock->lock);

17 }

18

19 static inline void assert\_smp\_lock\_is\_locked(struct smp\_lock \*lock)

20 {

21 BUG\_ON(!smp\_lock\_is\_locked(lock));

22 }

SMP\_LOCK\_UNLOCKED的实现很简单，仅仅是定义一个next、owner字段均为0的数据结构。smp\_lock\_init则是将这样的数据结构赋值给自旋锁变量。

smp\_lock\_is\_locked函数判断自旋锁是否被锁定。对于排队自旋锁来说，如果next、owner字段相等，则说明自旋锁已经被锁定。反之则未锁定。

assert\_smp\_lock\_is\_locked则是对smp\_lock\_is\_locked的简单封装。

smp\_lock函数的实现代码如下：

1 static inline void arch\_smp\_lock(struct arch\_smp\_lock \*lock)

2 {

3 unsigned int tmp;

4 struct arch\_smp\_lock lockval, newval;

5

6 asm volatile(

7 " prfm pstl1strm, %3\n"

8 "1: ldaxr %w0, %3\n"

9 " add %w1, %w0, %w5\n"

10 " stxr %w2, %w1, %3\n"

11 " cbnz %w2, 1b\n"

12 " eor %w1, %w0, %w0, ror #16\n"

13 " cbz %w1, 3f\n"

14 " sevl\n"

15 "2: wfe\n"

16 " ldaxrh %w2, %4\n"

17 " eor %w1, %w2, %w0, lsr #16\n"

18 " cbnz %w1, 2b\n"

19 "3:"

20 : "=&r" (lockval), "=&r" (newval), "=&r" (tmp), "+Q" (\*lock)

21 : "Q" (lock->owner), "I" (1 << TICKET\_SHIFT)

22 : "memory");

23 }

24

25 void smp\_lock(struct smp\_lock \*lock)

26 {

27 preempt\_disable();

28 arch\_smp\_lock(&lock->lock);

29 }

第27行，smp\_lock首先调用preempt\_disable以关闭抢占。

小问题2.28：假设在smp\_lock中，不关闭抢占，会有什么后果？

第28行，调用体系架构相关的自旋锁获取函数arch\_smp\_lock。

arch\_smp\_lock是smp\_lock的主要实现函数。

第6行的volatile关键字与第22行的memory声明一起构成了编译屏障的语义。

第7行预取自旋锁的值到缓存行中，以加快后续内存访问指令的执行。

第8行将自旋锁的值读取到临时变量lockval中。

第9行将lockval的next字段加1,并保存到newval中。

第10行将newval的值存储到lock中。请注意这里的条件存储指令，与第8行的排它装载指令形成了原子操作指令对。条件存储的结果保存到tmp变量中。

第11行判断tmp的结果是否不为0,如果不为0,表示第10行的条件存储不成功。也就是说，有其他CPU在并发的修改锁的值，或者在存储过中发生了中断，这两种情况都需要跳转到第8行，重新加载自旋锁的值。

如果运行到第12行，那么说明已经成功修改自旋锁的next字段。该行的eor异或指令实际上是比较lockval的next和owner字段是否相等。请注意 ，lockval中目前的值是自旋锁的旧值。

如果locakval的next和owner字段相等，如果二者相等，则说明自旋锁处于未锁定状态。于是第13行跳转到第19行，也就是退出整个函数。

如果自旋锁已经被锁定，则需要忙等。第14~15行的sev/wfe指令对向其他CPU发送事件并等待事件。这样，当前CPU会进入低功耗状态，并等待其他CPU在释放自旋锁时将当前CPU唤醒到正常状态。

第16行，重新加载自旋锁的owner字段到tmp变量的owner字段中。

第17行，比较tmp的owner字段与next字段是否相等。

如果不相等，说明当前CPU还不能获得自旋锁，则在第18行跳转到第16行，继续等待自旋锁。

否则，函数运行到第19行，成功获得自旋锁。

释放自旋锁的代码如下：

1 static inline void arch\_smp\_unlock(struct arch\_smp\_lock \*lock)

2 {

3 asm volatile(

4 " stlrh %w1, %0\n"

5 : "=Q" (lock->owner)

6 : "r" (lock->owner + 1)

7 : "memory");

8 }

9

10 void smp\_unlock(struct smp\_lock \*lock)

11 {

12 arch\_smp\_unlock(&lock->lock);

13 preempt\_enable();

14 }

首先，smp\_unlock在第12行调用体系架构相关的释放自旋锁操作函数arch\_smp\_unlock.

在第13行，调用preempt\_enable打开抢占。这是与smp\_lock第27行对应的。

arch\_smp\_unlock的实现看似很简单，核心是将lock的owner字段加1。这样下一个正在等待自旋锁的CPU就能顺便的通过smp\_lock函数了。

小问题2.29：实际上，arch\_smp\_unlock的实现有好几处值得仔细推敲的地方，聪明的读者看出其中的微妙之处了吗？

### 自旋位锁

#### 自旋位锁的设计原理

自旋位锁与自旋锁有类似的地方：均是多个CPU竞争同一把锁，如果不成功，就通过自旋的方式，轮询锁状态直到成功获得锁为止。但是二者也有明显的区别：

1. 在一个自旋锁中，使用32位的字段表示锁的状态。而在自旋位锁中，每一个unsigned long类型的字段中，表示32/64个位锁。其中每一位表示一个锁的状态。
2. 适用的范围不同。自旋锁是通用的锁，在操作系统中大量使用。而自旋位锁则主要用于文件页缓存状态的锁定操作。
3. 实现方式不同。自旋锁可以有多种灵活的实现方式。例如排队自旋锁，因为象排队自旋锁这样的实现需要多个字段记录排队信息和当前叫号信息，这就需要多个二进制位来表示。而自旋位锁仅仅只有一位来表示锁状态，因而其实现必然更简单。

具体的说，自旋位锁是为了节省内存空间而实现的一种简单自旋锁。其中每一位表示一个锁。这是因为文件页缓存数量很大，理论上讲，每一个内存页面都可能需要一把这样的锁。在1T内存的系统中，如果使用传统的自旋锁来跟踪内存页，需要接近10G的内存用于锁。如果使用自旋位锁，则可以将内存需求缩小到原来的1/32。这显然是很有价值的事情。

虽然自旋位锁可以节省锁占用的内存空间，但是它仅仅使用一位来表示锁状态，这没法实现排队锁，当然也就存在饥饿现象。而现实世界中，很少会有多个CPU同时竞争访问同一页缓存的现象。因此这个限制可以被忽略。

可以武断一点的说，自旋位锁的实现更接近于Linux早期版本的经典自旋锁实现。在Linux早期版本中，仅仅使用锁数据结构中的一位来表示锁的状态，并且不考虑公平性。

自旋锁位的数据结构很简单，可以简单的用unsigned long类型的指针及要等待的位编号来表示。

同样的，由于实现自旋位锁需要进行原子操作，这要求锁进行位对齐。

#### 自旋位锁API

下表是自旋位锁提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| smp\_bit\_lock | 获得自旋位锁 |
| smp\_bit\_trylock | 试图获得自旋位锁。如果成功获得锁，则返回1,否则返回0 |
| smp\_bit\_unlock | 释放自旋位锁 |
| smp\_bit\_is\_locked | 自旋锁是否处于锁定状态 |

小问题2.30：怎么没有类似于smp\_lock\_init这样的初始化锁的接口？

#### 自旋位锁的实现

自旋位锁的定义和实现位于include/dim-sum/smp\_bit\_lock.h和kernel/locking/smp\_bit\_lock.c中。

自旋位锁的实现主要是基于位操作函数来实现的。先看看最简单的smp\_bit\_is\_locked实现：

1 static inline int smp\_bit\_is\_locked(int bitnum, unsigned long \*addr)

2 {

3 return test\_bit(bitnum, addr);

4 }

smp\_bit\_is\_locked函数简单的调用test\_bit测试相应的位是否为1。如果为1表示锁已经被获取。

然而这里仍然有两个重要的问题值得深入思考：

小问题2.31：test\_bit并没有任何编译屏障/内存屏障的东西，作者您是不是搞错了？

小问题2.32：另一个重要的问题是：这个API到底拿来做什么？

下面我们看看最重要的获取/释放接口函数的实现：

1 void smp\_bit\_lock(int bitnum, unsigned long \*addr)

2 {

3 preempt\_disable();

4 while (unlikely(atomic\_test\_and\_set\_bit(bitnum, addr))) {

5 while (test\_bit(bitnum, addr)) {

6 preempt\_enable();

7 cpu\_relax();

8 preempt\_disable();

9 }

10 }

11 }

12

13 void smp\_bit\_unlock(int bitnum, unsigned long \*addr)

14 {

15 smp\_mb();

16 atomic\_clear\_bit(bitnum, addr);

17 preempt\_enable();

18 }

smp\_bit\_lock函数在第3行调用preempt\_disable以关闭抢占。细心的读者已经注意到，所有类似于自旋锁这样的锁的都应当关闭抢占。preempt\_disable相关的实现代码，其微妙之处将在调度相关章节中阐述。

在第4行，原子的比较并测试第bitnum位。简单的说，atomic\_test\_and\_set\_bit如果发现该位为1,就表明锁已经被其他CPU获取，因此进入第5行开始内层循环。如果atomic\_test\_and\_set\_bit返回为0,表示锁空闲，并成功的进行原子位设置，将相应的位标记为1。

请注意：atomic\_test\_and\_set\_bit已经包含了编译屏障/内存屏障相关语义。

在没有产生锁竞争的情况下，第4行的循环会直接退出，并退出整个函数。在退出本函数，务必记得：当前CPU已经关闭了抢占。因此在实时系统中，应该尽快的释放锁。

如果不幸产生了锁竞争，则进入第5行的内层循环。在第5行调用test\_bit进行普通的读操作，轮询相应的位是否仍然为1。在自旋位锁被其他CPU持有期间，会进入第6行。

第6行首先打开抢占。

第7行会调用yield指令，以告诉硬件：可以将当前CPU释放给其他硬件线程，以提高整体系统。因为当前硬件线程在忙等，不如将CPU资源释放给其他硬件线程。

小问题2.33：不熟悉CPU架构的读者，需要补充一下硬件线程相关背景知识？这里的线程和posix线程是两个概念。

第8行继续关闭抢占后继续轮询位锁状态。

小问题2.34：第6/8行的关闭/打开抢占比较繁琐，有这个必要吗？自旋锁的实现就没有这些实现。

smp\_bit\_unlock的实现比较简单。

首先，它在第15行调用smp\_mb来实现一个全内存屏障。这是锁的语义所要求的。简单的说：这个屏障允许其他CPU在看到锁状态变化的时候，能够看到当前CPU在锁的保护下对变量的修改。

第16行简单的清除自旋位锁的当前位。这样其他CPU就能够获得锁了。

第17行打开抢占。这是与第3行对应的操作。

smp\_bit\_trylock的实现与smp\_bit\_lock类似。相关的代码分析就留读者作为练习了。

### 自旋顺序锁

无论是自旋锁还是自旋位锁，所有的锁申请者都处于同等优先级。换句话说，如果有多个CPU在并发的申请锁，那么其中会有一个CPU需要等待其他所有CPU释放锁。这可能引起该CPU长时间在进行忙等。

然而这种忙等会造成问题。例如：CPU在中断上下文中更新当前时间的时候，如果其他CPU在并发的读取当前时间，这会导致当前CPU被延迟。对于更新操作频繁的情况，这样的延迟是代价巨大的。它可能会引起长时间的关中断，并且导致线程不能得到及时调度。

在这种情况下，应该使用自旋顺序锁。

#### 自旋顺序锁的设计原理

自旋顺序锁的设计目的，是通过牺牲读端的性能，从而优先保证更新端能够获得锁。

自旋顺序锁基本的设计原理，是使用一个名为sequence的字段，来表示更新端的顺序号。读端必须两次判断sequence字段，只有当这个字段没有发生任何改变的时候，才表示读端读取到一致的数据。

自旋顺序锁还内嵌自旋锁以防止多个更新端并发的进行更新操作。

#### 自旋顺序锁的数据结构

自旋顺序锁的数据结构如下：

1 struct smp\_seq\_lock {

2 unsigned sequence;

3 struct smp\_lock lock;

4 };

其中sequence字段表示顺序计数器。每个读端需要在读数据前后两次读顺序计数器。只有在这个值没有变化时，才说明读取到的数据是有效的。

lock字段是保护自旋顺序锁数据结构的自旋锁。

#### 自旋顺序锁API

下表是自旋顺序锁提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| SMP\_SEQ\_LOCK\_UNLOCKED | 定义一个自旋顺序锁 |
| smp\_seq\_init | 初始化自旋顺序锁 |
| smp\_seq\_write\_lock | 获得自旋顺序锁写锁 |
| smp\_seq\_write\_trylock | 试图获得自旋顺序锁写锁 |
| smp\_seq\_read\_begin | 准备开始读数据，实际上是获得当前顺序计数器。 |
| smp\_seq\_read\_retry | 判断当前顺序计数器是否发生变化 |

一个简单的读端例子，可以参考：

1 u64 get\_jiffies\_64(void)

2 {

3 unsigned long seq;

4 u64 ret;

5

6 do {

7 seq = smp\_seq\_read\_begin(&time\_lock);

8 ret = jiffies\_64;

9 } while (smp\_seq\_read\_retry(&time\_lock, seq));

10

11 return ret;

12 }

#### 自旋顺序锁的实现

自旋顺序锁的初始化API很简单，如下：

1 #define SMP\_SEQ\_LOCK\_UNLOCKED(lockname) \

2 { \

3 .sequence = 0, \

4 .lock = SMP\_LOCK\_UNLOCKED(lockname) \

5 }

6

7 static inline void smp\_seq\_init(struct smp\_seq\_lock \*lock)

8 {

9 lock->sequence = 0;

10 smp\_lock\_init(&lock->lock);

11 }

它仅仅简单的将sequence设置为0,并且初始化lock字段为未锁状态。

smp\_seq\_write\_lock、smp\_seq\_write\_unlock和smp\_seq\_write\_trylock是更新端的API，其实现代码如下：

1 void smp\_seq\_write\_lock(struct smp\_seq\_lock \*lock)

2 {

3 smp\_lock(&lock->lock);

4 lock->sequence++;

5 smp\_wmb();

6 }

7

8 void smp\_seq\_write\_unlock(struct smp\_seq\_lock \*lock)

9 {

10 smp\_wmb();

11 lock->sequence++;

12 smp\_unlock(&lock->lock);

13 }

14

15 int smp\_seq\_write\_trylock(struct smp\_seq\_lock \*lock)

16 {

17 int ret;

18

19 ret = smp\_trylock(&lock->lock);

20 if (ret) {

21 lock->sequence++;

22 smp\_wmb();

23 }

24

25 return ret;

26 }

smp\_seq\_write\_lock函数在第1行获得保护数据结构的自旋锁，以防止多个更新端并发的进入自旋顺序锁的写锁。

第2行将更新计数器加1,此时更新计数器的值应当为奇数。

第3行的写屏障防止读端在看到顺序计数器的新值之前，看到对自旋顺序锁所保护的数据结构的更新。以上一节的例子来说，读端在看到jiffies\_64的新值时，将会看到顺序锁计数器的新值。

smp\_seq\_write\_unlock函数在第10行仍然调用写屏障，以确保再次看到第11行的新值之前，能看到锁保护的数据结构的新值。

第11行再次将更新计数器加1,此时更新计数器的值应当为偶数。

第12行释放自旋锁，以允许下一个更新者获得自旋顺序锁的写锁。

smp\_seq\_write\_trylock的实现类似于smp\_seq\_write\_lock，此处不再详述。

smp\_seq\_read\_begin和smp\_seq\_read\_retry是用于读端的API，其实现代码如下：

1 static inline unsigned smp\_seq\_read\_begin(const struct smp\_seq\_lock \*lock)

2 {

3 unsigned ret;

4

5 ret = lock->sequence;

6 smp\_rmb();

7

8 return ret;

9 }

10

11 static inline int smp\_seq\_read\_retry(const struct smp\_seq\_lock \*lock, unsigned iv)

12 {

13 smp\_rmb();

14 return (iv & 1) | (lock->sequence ^ iv);

15 }

smp\_seq\_read\_begin首先在第5行获得更新计数器的当前值。

在第6行的读屏障可以确保：在看到更新端对更新计数器的新值之前，随后对更新端的数据访问将能看到更新端对所保护的数据的更新。

第8行简单的返回更新计数器的当前值。

smp\_seq\_read\_retry在第13行仍然是一个读屏障。

第14行，它检查更新计数器的值，如果更新计数器是奇数，我们知道，这表示更新端在执行过程中，因此读取到的数据必要是无效的，需要重试。

如果更新计数器的当前值与smp\_seq\_read\_begin的返回值不相等，说明在读端执行过程中，至少经历一次更新端的执行过程。这种情况下也需要重试。

### 自旋读写锁

自旋顺序锁适用于更新端执行频繁而读端不频繁的情况。如果更新端执行不频繁而读端执行频繁时，显然不应该使用自旋顺序锁，也不应该使用自旋锁。

此时，自旋读写锁可以派上用场了。

#### 自旋读写锁的设计原理

自旋读写锁允许有一个更新端执行。在更新端执行的时候，其他读端或者更新端都必须等待更新端释放写锁。

在没有CPU持有自旋读写锁写锁的情况下，允许多个读端并发的运行。

自旋读写锁使用一个32位的int字段表示锁的状态。其中第31位表示是否某个CPU持有了写锁。第0~30位表示读者的个数。

#### 自旋读写锁的数据结构

1 struct arch\_smp\_rwlock {

2 volatile unsigned int lock;

3 } \_\_attribute\_\_((aligned(4)));

4

5 struct smp\_rwlock {

6 struct arch\_smp\_rwlock raw\_lock;

7 };

自旋读写锁由struct smp\_rwlock来表示。该数据结构仅仅只包括体系架构相关的定义。当然，如果您希望为自旋读写锁添加一些有意思的功能，例如调测功能，可以考虑在这个结构中添加附加字段。

struct arch\_smp\_rwlock是体系架构相关的数据结构。对于arm 64来说，只包含一个名为lock的int字段。

需要注意的是：为了保证原子操作指令的正确性，这个字段需要4字节对齐，因为lock字段正好是4字节长。对齐要求是由第3行的编译指示语句来保证的。

#### 自旋读写锁API

下表是自旋读写锁提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| SMP\_RWLOCK\_UNLOCKED | 定义一个自旋读写锁 |
| smp\_rwlock\_init | 初始化自旋读写锁 |
| smp\_rwlock\_can\_read | 判断是否可以获得自旋读写锁的读锁 |
| smp\_rwlock\_can\_write | 判断是否可以获得自旋读写锁的写锁 |
| smp\_tryread | 试图获得得自旋读写锁的读锁 |
| smp\_trywrite | 试图获得得自旋读写锁的写锁 |
| smp\_write\_lock | 获得自旋读写锁的写锁 |
| smp\_write\_lock\_irq | 获得自旋读写锁的写锁并关中断 |
| smp\_read\_lock | 获得自旋读写锁的读锁 |
| smp\_read\_lock\_irqsave | 获得自旋读写锁的读锁，并关中断保存当前中断标志 |
| smp\_write\_lock\_irqsave | 获得自旋读写锁的写锁，并关中断保存当前中断标志 |
| smp\_read\_unlock | 释放自旋读写锁的读锁 |
| smp\_write\_unlock | 释放自旋读写锁的写锁 |
| smp\_write\_unlock\_irq | 释放自旋读写锁的写锁，并打开中断 |
| smp\_read\_unlock\_irqrestore | 释放自旋读写锁的读锁，并恢复中断标志 |
| smp\_write\_unlock\_irqrestore | 释放自旋读写锁的写锁，并恢复中断标志 |
| smp\_tryread\_irqsave | 获得自旋读写锁的读锁，如果成功就关闭中断并保存中断标志 |
| smp\_trywrite\_irqsave | 获得自旋读写锁的写锁，如果成功就关闭中断并保存中断标志 |

#### 自旋读写锁的实现

自旋读写锁的初始化API很简单，如下所示：

1 #define \_\_ARCH\_SMP\_RWLOCK\_UNLOCKED { 0 }

2

3 #define SMP\_RWLOCK\_UNLOCKED(lockname) \

4 (struct smp\_rwlock) { .raw\_lock = \_\_ARCH\_SMP\_RWLOCK\_UNLOCKED, } \

5

6 #define smp\_rwlock\_init(lock) \

7 do { \*(lock) = SMP\_RWLOCK\_UNLOCKED(lock); } while (0)

它仅仅是将lock字段设置为0,表示锁状态为未锁状态。

1 #define arch\_smp\_rwlock\_can\_read(x) ((x)->lock < 0x80000000)

2 #define arch\_smp\_rwlock\_can\_write(x) ((x)->lock == 0)

3

4 #define smp\_rwlock\_can\_read(rwlock) arch\_smp\_rwlock\_can\_read(&(rwlock)->raw\_lock)

5 #define smp\_rwlock\_can\_write(rwlock) arch\_smp\_rwlock\_can\_write(&(rwlock)->raw\_lock)

判断自旋读写锁是否可以读写的代码也很简单。如果lock字段的值小于0x80000000，表示自旋读写锁没有写锁被申请，此时要么只有读者在运行，要么没有读者也没有更新者在运行。这两种情况都可以被读者申请。

如果lock字段为0,表示既没有读者也没有更新者在运行，此时允许更新者申请写锁。

smp\_read\_lock和smp\_write\_lock是申请自旋读写锁的API，具体实现如下：

1 static inline void arch\_smp\_read\_lock(struct arch\_smp\_rwlock \*rw)

2 {

3 unsigned int tmp, tmp2;

4

5 asm volatile(

6 " sevl\n"

7 "1: wfe\n"

8 "2: ldaxr %w0, %2\n"

9 " add %w0, %w0, #1\n"

10 " tbnz %w0, #31, 1b\n"

11 " stxr %w1, %w0, %2\n"

12 " cbnz %w1, 2b\n"

13 : "=&r" (tmp), "=&r" (tmp2), "+Q" (rw->lock)

14 :

15 : "memory");

16 }

17

18 void smp\_read\_lock(struct smp\_rwlock \*lock)

19 {

20 preempt\_disable();

21 arch\_smp\_read\_lock(&lock->raw\_lock);;

22 }

23

24 static inline void arch\_smp\_write\_lock(struct arch\_smp\_rwlock \*rw)

25 {

26 unsigned int tmp;

27

28 asm volatile(

29 " sevl\n"

30 "1: wfe\n"

31 "2: ldaxr %w0, %1\n"

32 " cbnz %w0, 1b\n"

33 " stxr %w0, %w2, %1\n"

34 " cbnz %w0, 2b\n"

35 : "=&r" (tmp), "+Q" (rw->lock)

36 : "r" (0x80000000)

37 : "memory");

38 }

39

40 void smp\_write\_lock(struct smp\_rwlock \*lock)

41 {

42 preempt\_disable();

43 arch\_smp\_write\_lock(&lock->raw\_lock);

44 }

smp\_read\_lock在第20行调用preempt\_disable以关闭抢占。然后在第21调用体系架构相关的获取读锁函数arch\_smp\_read\_lock。

arch\_smp\_read\_lock函数第6、7行的指令对会使CPU进入低功耗等待模式，直到其他核释放锁的时候，通过硬件事件寄存器唤醒CPU。

第8行通过排它加载的方式从lock字段中加载锁的状态。

第9行将当前lock字段加1后存入到tmp变量中。

第10行测试锁的第31位（即写锁标志位）是否为1。如果为1,表示在此期间，某个CPU获得了写锁，于是跳转到第7行，进入低功耗状态继续等待。否则运行第11行。

第11行的条件存储指令将tmp变量保存到lock字段中。即将读者数量加1。

第12行判断第11行的条件存储指令是否执行成功。如果成功，表示对读者的原子计数成功，调用者成功的获得读锁，退出本函数。否则跳转到第8行重试。

小问题2.35：为什么在失败的情况下，第12行是跳转到第8行而不是第7行？

smp\_write\_lock函数与smp\_read\_lock类似。它在第42行禁止抢占后，调用系统架构相关的arch\_smp\_write\_lock函数获取写锁。

arch\_smp\_write\_lock函数在第32行则会判断锁的当前值是否为0。如果为0,表示锁完全空闲，可以授予给申请者。其他逻辑与arch\_smp\_read\_lock类似，不再详述。

### 读写信号量

前几节中描述的自旋锁、自旋位锁、自旋顺序锁、自旋读写锁，均是通过CPU忙等锁状态变化来实现的。这要求锁保护的数据量小、相应的执行代码能很快的执行完毕，例如在数十微秒内执行完毕。否则就会导致锁惊群问题，导致大量的CPU陷入忙等状态。

自旋类的锁，还有一个使用限制，就是不能在持有锁期间睡眠。

小问题2.36：为什么在持有自旋锁时不能睡眠？

但是确实也大的数据结构需要保护。例如文件页缓存。也有一些必须要睡眠的情况，例如需要等待外部事件。在这种情况下，自旋类锁均不适用。

读写信号量和互斥锁可以用于这种情况。信号量与互斥锁均可以在持有锁期间睡眠，这是它们与自旋锁最重要的区别。

#### 读写信号量的设计原理

读写信号量将访问者分为更新者和读者。读者在持有读写信号量期间，只能对所保护的共享资源进行只读访问，否则就被视为更新者。可以有多个读者并发的进入临界区，但是不允许不多个更新者并发的进入临界区，也不允许更新者和读者并发的进入临界区。

更新者能够获得读写信号量的条件是：读写信号量当前既没有被更新者持有，也没有被任何读者持有。

读者能够获得读写信号量的条件是：没有更新者持有信号量，并且也没有更新者在等待读者释放信号量。

读写信号量使用一个名为count的整型字段维护信号量的读写状态，同时使用一个名为wait\_list的双向链表来保存所有等待获取信号的线程。

#### 读写信号量的数据结构

1 struct rw\_semaphore {

2 int count;

3 struct smp\_lock wait\_lock;

4 struct double\_list wait\_list;

5 };

读写信号量使用struct rw\_semaphore数据结构来表示。

count字段表示读写信号量持有者的信息。当该字段大于0时，表示持有该信号量的读者数量。当该字段等于0时，表示既无更新者，也无读者持有该信号量。当该字段为-1时，表示有一个更新者持有该信号量。

wait\_lock字段是一个自旋锁，用于保护整个信号量数据结构。

wait\_list字段是一个双向链表，该链表中保存了所有等待该信号量的线程。

#### 读写信号量API

下表是读写信号量提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| RWSEM\_INITIALIZER | 定义一个读写信号量 |
| init\_rwsem | 初始化读写信号量 |
| down\_read | 以读者的身份获得读写信号量 |
| down\_read\_trylock | 试图以读者的身份获得读写信号量 |
| down\_write | 以更新者的身份获得读写信号量 |
| down\_write\_trylock | 试图以更新者的身份获得读写信号量 |
| up\_read | 读者释放读写信号量 |
| up\_write | 更新者释放读写信号量 |
| downgrade\_write | 更新者将自己降级为读者 |

#### 读写信号量的实现

在详细描述信号量API之前，我们有必要先看看信号量等待描述符：

1 struct rwsem\_waiter {

2 struct double\_list list;

3 struct task\_desc \*task;

4 unsigned int flags;

5 };

该描述符表示位于信号量等待队列中的某个线程。

list字段被链接进入信号量的wait\_list字段。这样所有的等待者将形成一个双向链表。

task字段表示等待信号量的线程描述符。

flags字段表示等待标志，例如是读者还是更新者。

我们再来看看读者获取读写信号量的代码：

1 void fastcall \_\_sched \_\_down\_read(struct rw\_semaphore \*sem)

2 {

3 struct rwsem\_waiter waiter;

4 struct task\_desc \*tsk;

5

6 smp\_lock(&sem->wait\_lock);

7

8 if (sem->count >= 0 && list\_is\_empty(&sem->wait\_list)) {

9 sem->count++;

10 smp\_unlock(&sem->wait\_lock);

11

12 return;

13 }

14

15 tsk = current;

16 set\_task\_state(tsk, TASK\_UNINTERRUPTIBLE);

17

18 waiter.task = tsk;

19 waiter.flags = WAITING\_FOR\_READ;

20 hold\_task\_desc(tsk);

21 list\_insert\_behind(&waiter.list, &sem->wait\_list);

22

23 smp\_unlock(&sem->wait\_lock);

24

25 for (;;) {

26 if (!waiter.task)

27 break;

28 schedule();

29 set\_task\_state(tsk, TASK\_UNINTERRUPTIBLE);

30 }

31

32 tsk->state = TASK\_RUNNING;

33 }

34

35 static inline void down\_read(struct rw\_semaphore \*sem)

36 {

37 \_\_down\_read(sem);

38 }

down\_read是读者获取信号的API接口，它在第37行直接调用\_\_down\_read来获得读写信号量。

\_\_down\_read在第6行获得自旋锁，以保护对信号量数据结构的访问。

在第8行，判断是否可以直接获取信号量而不必睡眠。需要满足两个条件才能直接获得信号量：

1. 信号是的count值大于等于0,这表明信号量要么是只被读者所持有，要么没有任何持有者。
2. 等待列表为空，这表示没有任何等待者。

小问题2.37：如果有等待者，等待队列会有什么特征？

如第8行的条件满足，那么表示读者可以放心的获得信号量。因此在第9行将count字段加1,表示一个新的读者到达。

第10行简单的释放信号量数据结构的自旋锁，并在第12行返回。

如果运行到第15行，说明不能直接获得信号量，因而必须等待。

第16行首先将线程的状态设置为UNINTERRUPTIBLE状态。这样在睡眠过程中，线程将不会被信号所唤醒。虽然目前DIM-SUM还不支持信号处理，但是在不远的将来，必然是会支持信号处理的。

第18~19行初始化等待描述符。表示当前任务是在作为读者在等待信号量。

第20将当前线程的描述符引用计数加1,表示等待队列对该线程持有了一次引用。以防止线程描述符被意外的释放。

第21行添加等待描述符添加到信号量的等待队列末尾。

运行到第23行，我们已经完成对信号量数据结构的修改，因而可以安全的释放信号量的自旋锁了。

第25行开始的死循环，是处理线程睡眠。

第26行判断等待描述符的task字段是否为空。如果为空，表示信号量的持有者释放了信号量，并且将信号量授予给当前线程，因而当前线程可以获得信号量了。

如果当前线程能够获得信号量，则在第26行退出循环结束睡眠等待。

小问题2.38：为什么要由释放信号量的线程直接将信号量授予当前线程？它直接唤醒等待线程并由线程来竞争锁也是可以的。

第28行调用schedule函数将当前线程切换出去。

小问题2.39：如果在第26行的判断条件执行完成，还没有调用schedule函数前，其他线程将信号量授予给当前线程，这里会出现BUG吗？

运行到第29行，说明有其他线程将当前线程唤醒。此时线程的状态是TASK\_RUNNING。但是我们还不能确定是否真的获得了信号量，因此需要将线程状态设置为TASK\_UNINTERRUPTIBLE，并继续下一次循环。由下一次循环来判断是否真的获得了信号量。

运行到第32行，说明当前线程真的获得了信号量，因而可以放心的将当前状态设置为TASK\_RUNNING并退出函数。

down\_write处理逻辑与down\_read类似，但如下流程略有不同：

1 if (sem->count == 0 && list\_is\_empty(&sem->wait\_list)) {

2 sem->count = -1;

3 smp\_unlock(&sem->wait\_lock);

4

5 return;

6 }

第1行判断更新者是否可以直接获得信号量的条件，是count字段等于0。这表示当前信号量没有被任何更新者/读者所持有，因而可以被更新者所获得。

在第2行，将count字段设置为-1,表示信号量当前被更新者所持有。

接下来我们看看up\_read函数的实现：

1 void fastcall \_\_up\_read(struct rw\_semaphore \*sem)

2 {

3 smp\_lock(&sem->wait\_lock);

4

5 sem->count--;

6 if (sem->count == 0 && !list\_is\_empty(&sem->wait\_list)) {

7 struct rwsem\_waiter \*waiter;

8 struct task\_desc \*tsk;

9

10 sem->count = -1;

11 waiter = list\_first\_container(&sem->wait\_list,

12 struct rwsem\_waiter, list);

13 list\_del(&waiter->list);

14

15 tsk = waiter->task;

16 waiter->task = NULL;

17 wake\_up\_process(tsk);

18 loosen\_task\_desc(tsk);

19 }

20

21 smp\_unlock(&sem->wait\_lock);

22 }

23

24 static inline void up\_read(struct rw\_semaphore \*sem)

25 {

26 \_\_up\_read(sem);

27 }

up\_read函数在第26行调用\_\_up\_read以释放信号量。

\_\_up\_read函数是读者释放信号量的主函数。在第3行，该函数获得保护信号量数据结构的自旋锁。

在第5行，将信号量的count字段减1,这表现了信号量读者数量减1的事实。

第6行的判断条件有两个，满足这两个条件，说明需要唤醒等待信号量的等待者：

1. count字段为0,表示所有读者都已经释放完信号量。
2. 等待队列不为空，说明有等待者需要唤醒。

小问题2.40：为什么说，这个等待队列上的第一个等待者一定是一个更新者？

第10行直接将count字段设置为-1,这表明当前的信号量被更新者所持有。

第11行获得等待队列上的第一个元素，并取得该元素对应的等待描述符。

第13行将第一个元素从等待队列上摘除。

第15行获得等待线程描述符。

第16行将等待描述符的task指针设置为空，这表明已经将信号量授予给该线程的事实。

第17行唤醒等待线程，该线程一般情况下应该处于TASK\_UNINTERRUPTIBLE状态。

小问题2.41：什么情况下，被唤醒的等待线程并不处于TASK\_UNINTERRUPTIBLE状态？

第18行释放线程描述符的引用计数。相应的计数在down\_read中曾经被引用。

第21行释放信号量的自旋锁。

更新者释放信号量的逻辑有所不同，有必要详细阐述。相应的代码如下：

1 static inline void \_\_do\_wake(struct rw\_semaphore \*sem, int wake\_write)

2 {

3 struct rwsem\_waiter \*waiter;

4 struct task\_desc \*tsk;

5 int woken;

6

7 waiter = list\_first\_container(&sem->wait\_list, struct rwsem\_waiter, list);

8 if (wake\_write && (waiter->flags & WAITING\_FOR\_WRITE)) {

9 sem->count = -1;

10 list\_del(&waiter->list);

11 tsk = waiter->task;

12 smp\_mb();

13 waiter->task = NULL;

14 wake\_up\_process(tsk);

15 loosen\_task\_desc(tsk);

16

17 return;

18 }

19

20 if (!wake\_write) {

21 if (waiter->flags & WAITING\_FOR\_WRITE)

22 return;

23 }

24

25 woken = 0;

26 while (waiter->flags & WAITING\_FOR\_READ) {

27 list\_del(&waiter->list);

28 tsk = waiter->task;

29 waiter->task = NULL;

30 wake\_up\_process(tsk);

31 loosen\_task\_desc(tsk);

32 woken++;

33

34 if (list\_is\_empty(&sem->wait\_list))

35 break;

36

37 waiter = list\_next\_entry(waiter, list);

38 }

39

40 sem->count += woken;

41 }

42

43 void fastcall \_\_up\_write(struct rw\_semaphore \*sem)

44 {

45 smp\_lock(&sem->wait\_lock);

46

47 sem->count = 0;

48 if (!list\_is\_empty(&sem->wait\_list))

49 \_\_do\_wake(sem, 1);

50

51 smp\_unlock(&sem->wait\_lock);

52 }

53

54 static inline void up\_write(struct rw\_semaphore \*sem)

55 {

56 \_\_up\_write(sem);

57 }

up\_write函数在第56行调用\_\_up\_write函数来释放信号量。

在\_\_up\_write函数的第45行，首先获得信号量的自旋锁。

第47行将信号量的count字段设置为0,这表示当前信号量既没有更新者，也没有读者。

在第48行判断信号量是否有等待者。如果有，则在第49行调用\_\_do\_wake唤醒等待者。

第51行释放自旋锁后退出函数。

\_\_do\_wake函数比较复杂，有必要仔细分析。

在\_\_do\_wake函数的第7行，首先取得第一个等待者的描述符。

第8行的判断，是处理唤醒更新者的情况，有两个判断条件：

1. 如果传入的标志允许唤醒更新者，表明当前信号量的count为0,完全可用。
2. 第一个等待者是一个更新者。

如果这两个条件都满足，说明只需要唤醒第一个更新者。

在第9行将count字段设置为-1,表示当前信号量被更新者所持有。

第10行将等待者描述符从等待队列中摘除。

第12行的内存屏障是为了防止第13行的赋值先于第9~10的赋值被其他CPU所看到。

第13行将等待描述符的task字段设置为空，表示信号量已经被授予这个等待者线程。

第14行唤醒等待者线程。

第15行释放等待者线程描述符的引用。

第17行直接返回。因此在此情况下，只需要唤醒第一个更新者即可，没有其他事情需要处理了。

第20~23行的代码处理主要用于更新者将自己降为读者的情况。在这种情况下，更新者可以唤醒等待队列中的读者，但是不能唤醒其他更新者。

第21行判断第一个等待者是否为更新者，如果是，则不可继续处理，直接返回。

运行到第25行，说明可以连续唤醒等待队列中的多个读者。这里首先将唤醒线程的数量设置为0。

第26行的循环开始遍历等待队列中所有的读者。

第27行将等待者从队列中摘除。

第29行将等待描述符的task字段设置为空，表示已经将信号量授予给该线程。

第30行唤醒等待者线程。

第31行释放对等待者线程的引用计数。

第32行将唤醒线程的数量加1。

第34行判断是否已经唤醒所有等待者，如果是则在第35行退出循环。

第37行取得下一个等待者描述符。

小问题2.42：第37行的list\_next\_entry可以被替换为list\_first\_container吗？

第40行更新count字段，以正确的反映当前的读者数量。

读写信号量的其他函数都比较简单，相应的代码分析就留给读者作为练习了。

### 互斥锁

互斥锁的概念是如此简单，读者应该都或多或少的使用过它。我们甚至可以将上一节中的读写信号量退化为互斥锁。只需要所有调用者都以更新者的身份获取信号量即可。

但是操作系统内核中的互斥锁也有它的特殊地方，本节将对它进行详细描述。

#### 互斥锁的设计原理

互斥锁是为了防止多个线程同时进入锁临界区而设计出来的原语。举个形象的例子：当某个人打开锁进入一个房间后，他关上了门。另一个人想要进入这个房间，必须要等待前一个人打开门从房间里面出来。这里的锁相当于是互斥锁，房间则是临界区，只能有一个人进入临界区。

DIM-SUM中的互斥锁设计原理与读写信号量是类似的。但是它的运行速度更快。

互斥锁是在精确计数API 之上实现的。

对互斥锁的访问必须遵循一些规则：

1. 同一时间只能有一个任务持有互斥锁，而且只有这个任务可以对互斥锁进行解锁。
2. 也不支持同一个任务对互斥锁进行递归锁定或解锁。
3. 一个任务在持有互斥锁的时候是不能结束运行的。
4. 不能用于中断上下文。

#### 互斥锁的数据结构

1 struct mutex {

2 struct accurate\_counter count;

3 struct smp\_lock wait\_lock;

4 struct double\_list wait\_list;

5 };

互斥锁的数据结构与读写信号量是如此相似，以至于读者可能会怀疑看花了眼。

count字段是一个精确计数器而不是普通的整型变量。这个字段可能有如下取值：

1、当值为1时，表示锁处于未锁状态。

2、当值为0时，表示锁处于锁定状态。但是没有等待者。

3、当值为-1时，表示锁处于锁定状态。并且有等待者。

wait\_lock是保护等待队列的自旋锁，但是该自旋锁并不保护count字段。

wait\_list字段是保存所有等待者的等待队列。

#### 互斥锁API

下表是互斥锁提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| MUTEX\_INITIALIZER | 定义一个互斥锁 |
| mutex\_init | 初始化互斥锁 |
| mutex\_is\_locked | 判断互斥锁是否已经被锁定 |
| mutex\_lock | 获得互斥锁 |
| mutex\_lock\_interruptible | 获得互斥锁，但是可能被信号中断 |
| mutex\_trylock | 试图获得互斥锁 |
| mutex\_unlock | 释放互斥锁 |

#### 互斥锁的实现

互斥锁的初始化API很简单：

1 #define MUTEX\_INITIALIZER(lockname) \

2 { \

3 .count = ACCURATE\_COUNTER\_INIT(1), \

4 .wait\_lock = SMP\_LOCK\_UNLOCKED(lockname.wait\_lock), \

5 .wait\_list = LIST\_HEAD\_INITIALIZER(lockname.wait\_list) \

6 }

7

8 void mutex\_init(struct mutex \*lock)

9 {

10 accurate\_set(&lock->count, 1);

11 smp\_lock\_init(&lock->wait\_lock);

12 list\_init(&lock->wait\_list);

13 }

仅仅需要注意的是：这里将count字段初始化为1,表示未锁状态。读者在分析完代码后，可以想想为什么要这样设计。

mutex\_is\_locked函数简单的判断count字段是否为1。

static inline int mutex\_is\_locked(struct mutex \*lock)

{

return accurate\_read(&lock->count) != 1;

}

获取互斥锁的代码实现如下：

1 static int \_\_mutex\_lock\_slow(struct mutex \*lock, long state)

2 {

3 struct task\_desc \*task = current;

4 struct mutex\_waiter waiter;

5 unsigned long count;

6

7 smp\_lock(&lock->wait\_lock);

8

9 waiter.task = task;

10 list\_init(&waiter.list);

11 list\_insert\_behind(&waiter.list, &lock->wait\_list);

12

13 while (1) {

14 count = accurate\_xchg(&lock->count, -1);

15 if (count == 1)

16 break;

17

18 if (unlikely(state == TASK\_INTERRUPTIBLE &&

19 signal\_pending(task))) {

20 list\_del(&waiter.list);

21 smp\_unlock(&lock->wait\_lock);

22

23 return -EINTR;

24 }

25

26 \_\_set\_task\_state(task, state);

27 smp\_unlock(&lock->wait\_lock);

28 schedule();

29 smp\_lock(&lock->wait\_lock);

30 }

31

32 list\_del(&waiter.list);

33 if (likely(list\_is\_empty(&lock->wait\_list)))

34 accurate\_set(&lock->count, 0);

35

36 smp\_unlock(&lock->wait\_lock);

37

38 return 0;

39 }

40

41 void mutex\_lock(struct mutex \*lock)

42 {

43 if (unlikely(accurate\_dec(&lock->count) < 0))

44 \_\_mutex\_lock\_slow(lock, TASK\_UNINTERRUPTIBLE);

45 }

mutex\_lock函数的第43行是其快速路径。

视count字段的当前值而言，可能有两种情况：

1. count值为1,表示锁处于未锁定状态。减1后将其设置为锁定状态。这是没有锁冲突的情况，可以直接退出函数。
2. count值为0或者负数，表示锁处于锁定状态。减1后仍然为锁定状态。此时需要调用\_\_mutex\_lock\_slow进入慢速路径。

小问题2.43：与读写信号量相比，互斥锁的数据结构和代码都不是那么直观，为什么要这样设计？

\_\_mutex\_lock\_slow慢速路径的处理比较复杂。在第7行，获取互斥锁数据结构的自旋锁。

第9~11行初始化等待描述符并将其插入到等待队列的末尾。

第13~30行的死循环是在等待锁可用。

第14行的accurate\_xchg函数原子的将count字段设置为-1并获取它原来的值。

第15行判断count字段的原值是否为1，如果是，表示锁处于未锁定状态，可以被当前线程所获取。退是在第16行退出死循环。

第18~24行的代码块主要用于mutex\_lock\_interruptible函数。简单的说，mutex\_lock\_interruptible函数允许被信号打断。因此在19行判断是否有信号等待处理，如果有就中断互斥锁的获取流程并返回-EINTR。

第26行将线程的运行状态设置为TASK\_INTERRUPTIBLE或者TASK\_INTERRUPTIBLE。

第27行释放保护互斥锁数据结构的自旋锁。

第28行调用schedule函数将当前线程切换出来，等待其他线程将其唤醒。

运行到第29行，说明其他线程将当前线程唤醒，因此重新获得自旋锁后，重新开始下一轮的循环，开始判断互斥锁的状态。

运行到32行时，说明互斥锁的状态为可用状态，并且当前流程持有自旋锁。第32行首先将当前线程从等待队列中摘除。

第33行判断等待队列是否为空，如果是，表明没有等待者，则需要在第34行将count字段设置为0,以反映没有等待者这个事实。

第36行释放互斥锁的自旋锁，并在第38行返回0,以表示成功获得互斥锁。

mutex\_lock\_interruptible和mutex\_trylock的实现与mutex\_lock类似。相应的代码分析工作留给读者作为练习。

mutex\_unlock函数的实现则简单得多，代码如下：

1 void mutex\_unlock(struct mutex \*lock)

2 {

3 if (unlikely(accurate\_inc(&lock->count) <= 0)) {

4 smp\_lock(&lock->wait\_lock);

5

6 if (!list\_is\_empty(&lock->wait\_list)) {

7 struct mutex\_waiter \*waiter;

8

9 waiter = list\_first\_container(&lock->wait\_list,

10 struct mutex\_waiter, list);

11 wake\_up\_process(waiter->task);

12 }

13

14 smp\_unlock(&lock->wait\_lock);

15 }

16 }

第3行递增count字段的值。存在两种情况：

1. 如果递增前count字段值等于0,那么表示没有等待者。递增后的结果必然等于1,此时可以直接退出。这是没有锁竞争的理想情况。
2. 如果递增前count字段的值为负数，那么表示有等待者。递增后的结果小于等于0。此时进入第4~14行的代码块，唤醒等待者。

第4行首先获得保护等待队列的自旋锁。

第6行在锁的保护下再次判断等待队列是否为空。

小问题2.44：第3行已经判断了count字段，表明存在等待者了，为什么需要第6行的判断？

如果等待队列确实不为空，则在第9行取得等待队列中第一个等待者。

第10将第一个等待者唤醒。

第14行释放保护互斥锁的自旋锁。

## 内核同步原语

前面的章节主要描述互斥原语。用于CPU/线程之间的临界区保护。

然而，这些原语不能用于同步。所谓同步，是指在中断或者线程中，由于所需要的条件已经满足，因此可以唤醒线程开始后续处理。

一个典型例子：线程在等待用户键盘输入的时候，如果键盘输入缓冲区中没有字符，那么就可以在信号量上面等待。当用户在键盘上输入一个字符的时候，就可以将输入字符放到缓冲区，并通过信号量将等待的线程唤醒。被唤醒的线程可以从缓冲区中读取字符并继续处理。

当然，信号量也可以退化为互斥锁。也就是说，信号量既可以作为内核同步原语，也可以作为内核互斥原语。

### 信号量

#### 信号量的设计原理

作为互斥锁的信号量主要解决的问题是：对线程之间需要共享的资源进行保护，仅仅允许一个线程对共享资源进行访问。

作为同步机制的信号量则解决另一个经典问题：生产者/消费者问题。

一个经典的生产者/消费者模型的例子是：线程A 负责生产产品，例如创建并写入文件，线程B则负责消费产品，例如复制文件。正常过程是当线程A 创建并准备好文件数据后，线程B才开始复制文件。但是如果两个线程不按照规定的时序运行，就会产生预期外的结果。例如，线程A的生产工作还未完成，只创建了文件但是没写入数据，线程B 就开始复制文件，这必然导致线程B得到的是一个错误的文件。

这个问题的出现，就是两个线程间没有一种同步机制。

同步信号量解决了这个问题。信号量使用一个count字段来对生产者/消费者进行同步。当生产者准备好数据后，将count字段加1，并通知消费者。消费者将count字段减1,以完成同步过程。消费者如果发现count字段不为1,表示生产者没有准备好数据，因此会等待。

作为互斥锁的信号量，会将count字段设置为1。而作为同步机制的信号量，则需要将count字段设置为0。

#### 信号量的数据结构

信号量的数据结构用struct semaphore来表示：

1 struct semaphore{

2 struct smp\_lock lock;

3 unsigned int count;

4 struct double\_list wait\_list;

5 };

名为count的整型字段表示资源的计数值。存在如下可能的值：

1、count值大于0，表示信号量是空闲的。

2、count值等于0，表示信号量是忙的，但是没有线程在等待这个信号量。

3、count值小于0，表示信号量是忙的，并且至少有一个进程在等待。但是需要注意的是：负值并不代表等待信号量的线程数量。

名为lock的自旋锁保护整个信号量数据结构。

名为wait\_list的双向链表保存了所有等待信号量的等待者。

#### 信号量API

下表是信号量提供的API接口：

|  |  |
| --- | --- |
| API名称 | 功能描述 |
| SEMAPHORE\_INITIALIZER | 定义一个信号量 |
| sema\_init | 初始化信号量 |
| down | 消费者等待信号量可用 |
| down\_interruptible | 消费者等待信号量可用，但是等待过程可以被信号打断 |
| down\_trylock | 消费者试图等待信号量可用 |
| down\_timeout | 消费者等待信号量可用，如果一段时间内信号量不可用，则超时退出 |
| up | 生产者通知消费者，数据已经就绪 |

#### 信号量的实现

最重要、也最复杂的接口是down函数，其代码实现如下：

1 static inline int \_\_sched

2 \_\_down\_common(struct semaphore \*sem, long state, long timeout)

3 {

4 struct semaphore\_waiter waiter;

5 struct task\_desc \*task;

6 int ret = 0;

7

8 task = current;

9 waiter.task = task;

10 list\_init(&waiter.list);

11 list\_insert\_behind(&waiter.list, &sem->wait\_list);

12

13 while (1) {

14 if (signal\_pending(task)) {

15 ret = -EINTR;

16 goto out;

17 }

18

19 if (unlikely(timeout <= 0)) {

20 ret = -ETIME;

21 goto out;

22 }

23

24 \_\_set\_task\_state(task, state);

25 smp\_unlock\_irq(&sem->lock);

26 timeout = schedule\_timeout(timeout);

27 smp\_lock\_irq(&sem->lock);

28

29 if (waiter.task == NULL)

30 return ret;

31 }

32

33 out:

34 list\_del(&waiter.list);

35 return ret;

36 }

37

38 void down(struct semaphore \*sem)

39 {

40 unsigned long flags;

41

42 smp\_lock\_irqsave(&sem->lock, flags);

43

44 if (likely(sem->count > 0))

45 sem->count--;

46 else

47 \_\_down\_common(sem, TASK\_UNINTERRUPTIBLE, MAX\_SCHEDULE\_TIMEOUT);

48

49 smp\_unlock\_irqrestore(&sem->lock, flags);

50 }

down函数在第42行获得自旋锁，并关闭中断。

小问题2.45：前面章节的读写信号量，在保护自身数据结构的时候，仅仅使用了自旋锁，但是并没有关闭中断，为什么第42需要关闭中断？

第44行检查信号量的count字段，如果大于0,表示信号量可用，则跳转到第45行。将信号量计数值减1后退出。此时信号量将不再可用。

如果第44行检查到信号量的count字段小于等于0,说明信号量不可用，必须要等待其变为可用。则跳转到第47行调用\_\_down\_common函数进行慢速处理流程。

第49行释放信号量的自旋锁并恢复中断标志位。

\_\_down\_common函数是其慢速处理流程，它首先在第8~10行准备等待者描述符。

然后在第11行将等待者描述符插入到信号量的等待队列末尾。

第13~31行的死循环是处理信号量等待过程的关键流程。

第14~17行判断任务是否有待处理的信号，如果有则退出等待过程。

小问题2.46：第14~17行有什么BUG吗？

第19行判断超时时间是否已经到达，如果到过，则设置错误返回码为-ETIME并退出。

第24行设置线程的状态为TASK\_UNINTERRUPTIBLE或者TASK\_INTERRUPTIBLE。

第25行释放自锁并打开中断。请注意这里是直接打开中断，并不是恢复中断标志位。

第26行调用schedule\_timeout将当前线程切换出去。

运行到第27行，说明线程要么由于信号量可用的原因而被唤醒，要么由于超时时间到达而被唤醒。由于需要访问信号量的数据结构，因此需要重新获得自旋锁并关闭中断。

小问题2.47：这里关闭了中断，如果在第30行直接返回了，那么中断打开关闭过程就没有配对使用，不太合乎编程规范。作者您是不是搞错了？

第29行判断是否真的是由于信号量被授予而被唤醒，如果是则返回表示成功的返回值。

如果是由于信号或者超时的原因被唤醒，则跳转到下一轮循环继续处理。

运行到第33行，是由于线程被信号或者超时所唤醒，而不是被信号量所唤醒。此时需要手动将等待描述符从等待队列中摘除。并返回错误值。

up函数很重要，但是很简单。代码如下：

1 void up(struct semaphore \*sem)

2 {

3 unsigned long flags;

4

5 smp\_lock\_irqsave(&sem->lock, flags);

6

7 if (likely(list\_is\_empty(&sem->wait\_list)))

8 sem->count++;

9 else {

10 struct semaphore\_waiter \*waiter;

11

12 waiter = list\_first\_container(&sem->wait\_list,

13 struct semaphore\_waiter, list);

14 list\_del(&waiter->list);

15 waiter->task = NULL;

16 wake\_up\_process(waiter->task);

17 }

18

19 smp\_unlock\_irqrestore(&sem->lock, flags);

20 }

在第5行，up函数首先获得信号量的自旋锁并关闭中断。

第7行，判断信号量等待队列是否为空。如果为空，则仅仅将信号量计数器加1即可。

否则，跳转到第10,开始唤醒等待队列中的线程。

第12行获得等待队列中第一个等待者。

第14行将等待者从等待队列中摘除。

第15行将等待描述符的task字段设置为空，表示这是由于正常的信号量授予过程将等待者唤醒的。

第16行唤醒等待者线程。

不管是何种情况，均在第19行释放自旋锁并恢复中断标志。

# **第** **4章**

# 进程

## SMP CPU初始化

## 调度子系统初始化

## 数据结构

### 进程

### 调度队列

### 杂项

## 进程调度

### 上下文切换

### 唤醒进程

## 进程等待

## Sleep

## 消息队列

# **第** **5章**

# 中断及定时器

## 中断控制器初始化

## 中断控制器维护

## 中断处理

### 序言

### 中断处理

### 尾声

## 工作队列

## 定时器

# **第** **6章**

# 内存管理

## 内存初始化

### 早期内存映射

## Boot内存分配器

## 内存页面映射

## 页面分配器

### 页帧管理

### 页面分配器的设计原理

### 页面分配器的数据结构

### 页面分配器的实现

## Beehive内存分配器

## 与文件缓存的关系

# **第** **7章**

# 块设备

## 块设备的识别及初始化

## 磁盘分区识别

## IO请求

## IO调度

# **第** **8章**

# 文件系统

## 磁盘分区识别

### 磁盘初始化

### 分区识别

## 虚拟文件系统

### 文件系统调用

### 文件系统缓存

## 内存文件系统实现

## 设备文件系统

## 文件系统日志

## LEXT3文件系统

## 根文件系统

# **第** **9章**

# 杂项

## Klibc

## 网络子系统

## SIMPLE-KSHELL