Linux Translations Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

中文翻译

这些手册包含有关如何开发内核的整体信息。内核社区非常庞大，一年下来有数千名开发人员做出贡献。与任何大型社区一样，知道如何完成任务将使得更改合并的过程变得更加容易。

翻译计划: 内核中文文档欢迎任何翻译投稿，特别是关于内核用户和管理员指南部分。

> **Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
../../../admin-guide/index

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核用户和管理员指南

下面是一组随时间添加到内核中的面向用户的文档的集合。到目前为止，还没有一个整体的顺序或组织 - 这些材料不是一个单一的，连贯的文件！幸运的话，情况会随着时间的推移而迅速改善。

这个初始部分包含总体信息，包括描述内核的 README，关于内核参数的文档等。

Todolist:

README kernel-parameters devices sysctl/index

本节介绍 CPU 漏洞及其缓解措施。

Todolist:

hw-vuln/index

下面的一组文档，针对的是试图跟踪问题和 bug 的用户。

Todolist:

reporting-bugs security-bugs bug-hunting bug-bisect tainted-kernels ramoops dynamic-debug-howto init kdump/index perf/index

这是应用程序开发人员感兴趣的章节的开始。可以在这里找到涵盖内核 ABI 各个方面的文档。

Todolist:

sysfs-rules

本手册的其余部分包括各种指南，介绍如何根据您的喜好配置内核的特定行为。

* 清除 WARN_ONCE

WARN_ONCE / WARN_ON_ONCE / printk_once 仅仅打印一次消息.

echo 1 > /sys/kernel/debug/clear_warn_once

可以清除这种状态并且再次允许打印一次告警信息，这对于运行测试集后重现问题很有用。

* CPU 负载

Linux 通过``/proc/stat``和``/proc/uptime``导出各种信息，用户空间工具如 top(1) 使用这些信息计算系统花费在某个特定状态的平均时间。例如:

$ iostat Linux 2.6.18.3-exp (linmac) 02/20/2007

avg-cpu: %user %nice %system %iowait %steal %idle
10.01 0.00 2.92 5.44 0.00 81.63

…

这里系统认为在默认采样周期内有 10.01% 的时间工作在用户空间，2.92% 的时间用在系统空间，总体上有 81.63% 的时间是空闲的。

大多数情况下``/proc/stat``的信息几乎真实反映了系统信息，然而，由于内核采集这些数据的方式/时间的特点，有时这些信息根本不可靠。

那么这些信息是如何被搜集的呢？每当时间中断触发时，内核查看此刻运行的进程类型，并增加与此类型/状态进程对应的计数器的值。这种方法的问题是在两次时间中断之间系统（进程）能够在多种状态之间切换多次，而计数器只增加最后一种状态下的计数。

举例—

假设系统有一个进程以如下方式周期性地占用 cpu:

```
 两个时钟中断之间的时间线
|-----------------------|
 ^                       ^
 |_ 开始运行             |
                         |_ 开始睡眠
                          (很快会被唤醒)
```

在上面的情况下，根据``/proc/stat``的信息（由于当系统处于空闲状态时，时间中断经常会发生）系统的负载将会是 0

大家能够想象内核的这种行为会发生在许多情况下，这将导致``/proc/stat`` 中存在相当古怪的信息:

```
/* gcc -o hog smallhog.c */
#include <time.h>
#include <limits.h>
#include <signal.h>
#include <sys/time.h>
#define HIST 10

static volatile sig_atomic_t stop;

static void sighandler (int signr)
{
(void) signr;
stop = 1;
}
static unsigned long hog (unsigned long niters)
{
stop = 0;
while (!stop && --niters);
return niters;
}
int main (void)
{
int i;
struct itimerval it = { .it_interval = { .tv_sec = 0, .tv_usec = 1 }
 ↪,
                        .it_value = { .tv_sec = 0, .tv_usec = 1 } };
sigset_t set;
unsigned long v[HIST];
double tmp = 0.0;
unsigned long n;
signal (SIGALRM, &sighandler);
setitimer (ITIMER_REAL, &it, NULL);

hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) v[i] = ULONG_MAX - hog (ULONG_MAX);
for (i = 0; i < HIST; ++i) tmp += v[i];
tmp /= HIST;
n = tmp - (tmp / 3.0);

sigemptyset (&set);
sigaddset (&set, SIGALRM);

for (;;) {
        hog (n);
        sigwait (&set, &i);
}
return 0;
}
```

参考—

- http://lkml.org/lkml/2007/2/12/6
- Documentation/filesystems/proc.rst (1.8)

谢谢—

Con Kolivas, Pavel Machek

Todolist:

> acpi/index aoe/index auxdisplay/index bcache binderfs binfmt-misc blockdev/index bootconfig braille-console btmrvl cgroup-v1/index cgroup-v2 cifs/index cputopology dell_rbu device-mapper/index edid efi-stub ext4 nfs/index gpio/index highuid hw_random initrd iostats java jfs kernel-per-CPU-kthreads laptops/index lcd-panel-cgram ldm lockup-watchdogs LSM/index md media/index mm/index module-signing mono namespaces/index numastat parport perf-security pm/index pnp rapidio ras rtc serial-console svga sysrq thunderbolt ufs unicode vga-softcursor video-output wimax/index xfs

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/index.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 与 Linux 内核社区一起工作

那么你想成为 Linux 内核开发人员？欢迎！不但从技术意义上讲有很多关于内核的知识需要学，而且了解我们社区的工作方式也很重要。阅读这些文章可以让您以更轻松地，麻烦最少的方式将更改合并到内核。

以下是每位开发人员应阅读的基本指南。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/howto.rst

译者:

英文版维护者： Greg Kroah-Hartman <greg@kroah.com>
中文版维护者： 李阳 Li Yang <leoyang.li@nxp.com>
中文版翻译者： 李阳 Li Yang <leoyang.li@nxp.com>
时奎亮 Alex Shi <alex.shi@linux.alibaba.com>
中文版校译者：
钟宇 TripleX Chung <xxx.phy@gmail.com>
陈琦 Maggie Chen <chenqi@beyondsoft.com>
王聪 Wang Cong <xiyou.wangcong@gmail.com>

* 如何参与 Linux 内核开发

这是一篇将如何参与 Linux 内核开发的相关问题一网打尽的终极秘笈。它将指导你成为一名 Linux 内核开发者，并且学会如何同 Linux 内核开发社区合作。它尽可能不包括任何关于内核编程的技术细节，但会给你指引一条获得这些知识的正确途径。

如果这篇文章中的任何内容不再适用，请给文末列出的文件维护者发送补丁。

入门

你想了解如何成为一名 Linux 内核开发者？或者老板吩咐你“给这个设备写个 Linux 驱动程序”？这篇文章的目的就是教会你达成这些目标的全部诀窍，它将描述你需要经过的流程以及给出如何同内核社区合作的一些提示。它还将试图解释内核社区为何这样运作。

Linux 内核大部分是由 C 语言写成的，一些体系结构相关的代码用到了汇编语言。要参与内核开发，你必须精通 C 语言。除非你想为某个架构开发底层代码，否则你并不需要了解（任何体系结构的）汇编语言。下面列举的书籍虽然不能替代扎实的 C 语言教育和多年的开发经验，但如果需要的话，做为参考还是不错的：

- “The C Programming Language”by Kernighan and Ritchie [Prentice Hall]《C 程序设计语言（第 2 版・新版）》（徐宝文李志译）[机械工业出版社]
- “Practical C Programming”by Steve Oualline [O’Reilly]《实用 C 语言编程（第三版）》（郭大海译）[中国电力出版社]
- “C: A Reference Manual”by Harbison and Steele [Prentice Hall]《C 语言参考手册（原书第 5 版）》（邱仲潘等译）[机械工业出版社]

Linux 内核使用 GNU C 和 GNU 工具链开发。虽然它遵循 ISO C89 标准，但也用到了一些标准中没有定义的扩展。内核是自给自足的 C 环境，不依赖于标准 C 库的支持，所以并不支持 C 标准中的部分定义。比如 long long 类型的大数除法和浮点运算就不允许使用。有时候确实很难弄清楚内核对工具链的要求和它所使用的扩展，不幸的是目前还没有明确的参考资料可以解释它们。请查阅 gcc 信息页（使用“info gcc”命令显示）获得一些这方面信息。

请记住你是在学习怎么和已经存在的开发社区打交道。它由一群形形色色的人组成，他们对代码、风格和过程有着很高的标准。这些标准是在长期实践中总结出来的，适应于地理上分散的大型开发团队。它们已经被很好得整理成档，建议你在开发之前尽可能多的学习这些标准，而不要期望别人来适应你或者你公司的行为方式。

法律问题

Linux 内核源代码都是在 GPL（通用公共许可证）的保护下发布的。要了解这种许可的细节请查看源代码主目录下的 COPYING 文件。Linux 内核许可准则和如何使用 *SPDX <https://spdx.org/>* 标志符说明在这个文件中Linux 内核许可规则 如果你对它还有更深入问题请联系律师，而不要在 Linux 内核邮件组上提问。因为邮件组里的人并不是律师，不要期望他们的话有法律效力。

对于 GPL 的常见问题和解答，请访问以下链接：
https://www.gnu.org/licenses/gpl-faq.html

文档

Linux 内核代码中包含有大量的文档。这些文档对于学习如何与内核社区互动有着不可估量的价值。当一个新的功能被加入内核，最好把解释如何使用这个功能的文档也放进内核。当内核的改动导致面向用户空间的接口发生变化时，最好将相关信息或手册页 (manpages) 的补丁发到 mtk.manpages@gmail.com，以向手册页 (manpages) 的维护者解释这些变化。

以下是内核代码中需要阅读的文档：

Documentation/admin-guide/README.rst
文件简要介绍了 Linux 内核的背景，并且描述了如何配置和编译内核。内核的新用户应该从这里开始。

Documentation/process/changes.rst
文件给出了用来编译和使用内核所需要的最小软件包列表。

Documentation/translations/zh_CN/process/coding-style.rst
描述 Linux 内核的代码风格和理由。所有新代码需要遵守这篇文档中定义的规范。大多数维护者只会接收符合规定的补丁，很多人也只会帮忙检查符合风格的代码。

如何让你的改动进入内核 Documentation/process/submitting-drivers.rst

这两份文档明确描述如何创建和发送补丁，其中包括（但不仅限于）：

- 邮件内容
- 邮件格式
- 选择收件人

遵守这些规定并不能保证提交成功（因为所有补丁需要通过严格的内容和风格审查），但是忽视他们几乎就意味着失败。

其他关于如何正确地生成补丁的优秀文档包括：“The Perfect Patch”

https://www.ozlabs.org/~akpm/stuff/tpp.txt

“Linux kernel patch submission format”

https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html

Documentation/translations/zh_CN/process/stable-api-nonsense.rst
论证内核为什么特意不包括稳定的内核内部 API，也就是说不包括像这样的特性：

- 子系统中间层（为了兼容性?）
- 在不同操作系统间易于移植的驱动程序

- 减缓（甚至阻止）内核代码的快速变化

这篇文档对于理解 Linux 的开发哲学至关重要。对于将开发平台从其他操作系统转移到 Linux 的人来说也很重要。

Documentation/admin-guide/security-bugs.rst
如果你认为自己发现了 Linux 内核的安全性问题，请根据这篇文档中的步骤来提醒其他内核开发者并帮助解决这个问题。

Linux 内核管理风格

描述内核维护者的工作方法及其共有特点。这对于刚刚接触内核开发（或者对它感到好奇）的人来说很重要，因为它解释了很多对于内核维护者独特行为的普遍误解与迷惑。

Documentation/process/stable-kernel-rules.rst
解释了稳定版内核发布的规则，以及如何将改动放入这些版本的步骤。

Documentation/process/kernel-docs.rst
有助于内核开发的外部文档列表。如果你在内核自带的文档中没有找到你想找的内容，可以查看这些文档。

Documentation/process/applying-patches.rst
关于补丁是什么以及如何将它打在不同内核开发分支上的好介绍

内核还拥有大量从代码自动生成或者从 ReStructuredText(ReST) 标记生成的文档，比如这个文档，它包含内核内部 API 的全面介绍以及如何妥善处理加锁的规则。所有这些文档都可以通过运行以下命令从内核代码中生成为 PDF 或 HTML 文档:

```
make pdfdocs
make htmldocs
```

ReST 格式的文档会生成在 Documentation/output. 目录中。它们也可以用下列命令生成 LaTeX 和 ePub 格式文档:

```
make latexdocs
make epubdocs
```

如何成为内核开发者

如果你对 Linux 内核开发一无所知，你应该访问“Linux 内核新手”计划:

https://kernelnewbies.org

它拥有一个可以问各种最基本的内核开发问题的邮件列表（在提问之前一定要记得查找已往的邮件，确认是否有人已经回答过相同的问题）。它还拥有一个可以获得实时反馈的 IRC 聊天频道，以及大量对于学习 Linux 内核开发相当有帮助的文档。

网站简要介绍了源代码组织结构、子系统划分以及目前正在进行的项目（包括内核中的和单独维护的）。它还提供了一些基本的帮助信息，比如如何编译内核和打补丁。

如果你想加入内核开发社区并协助完成一些任务，却找不到从哪里开始，可以访问“Linux 内核房管员”计划:

https://kernelnewbies.org/KernelJanitors

这是极佳的起点。它提供一个相对简单的任务列表，列出内核代码中需要被重新整理或者改正的地方。通过和负责这个计划的开发者们一同工作，你会学到将补丁集成进内核的基本原理。如果还没有决定下一步要做什么的话，你还可能会得到方向性的指点。

在真正动手修改内核代码之前，理解要修改的代码如何运作是必需的。要达到这个目的，没什么办法比直接读代码更有效了（大多数花招都会有相应的注释），而且一些特制的工具还可以提供帮助。例如，“Linux 代码交叉引用”项目就是一个值得特别推荐的帮助工具，它将源代码显示在有编目和索引的网页上。其中一个更新及时的内核源码库，可以通过以下地址访问：

> https://elixir.bootlin.com/

开发流程

目前 Linux 内核开发流程包括几个“主内核分支”和很多子系统相关的内核分支。这些分支包括：

- Linus 的内核源码树
- 多个主要版本的稳定版内核树
- 子系统相关的内核树
- linux-next 集成测试树

主线树

主线树是由 Linus Torvalds 维护的。你可以在 https://kernel.org 网站或者代码库中下找到它。它的开发遵循以下步骤：

- 每当一个新版本的内核被发布，为期两周的集成窗口将被打开。在这段时间里维护者可以向 Linus 提交大段的修改，通常这些修改已经被放到-mm 内核中几个星期了。提交大量修改的首选方式是使用 git 工具（内核的代码版本管理工具，更多的信息可以在 https://git-scm.com/ 获取），不过使用普通补丁也是可以的。
- 两个星期以后-rc1 版本内核发布。之后只有不包含可能影响整个内核稳定性的新功能的补丁才可能被接受。请注意一个全新的驱动程序（或者文件系统）有可能在-rc1 后被接受是因为这样的修改完全独立，不会影响其他的代码，所以没有造成内核退步的风险。在-rc1 以后也可以用 git 向 Linus 提交补丁，不过所有的补丁需要同时被发送到相应的公众邮件列表以征询意见。
- 当 Linus 认为当前的 git 源码树已经达到一个合理健全的状态足以发布供人测试时，一个新的-rc 版本就会被发布。计划是每周都发布新的-rc 版本。
- 这个过程一直持续下去直到内核被认为达到足够稳定的状态，持续时间大概是 6 个星期。

关于内核发布，值得一提的是 Andrew Morton 在 linux-kernel 邮件列表中如是说：
“没有人知道新内核何时会被发布，因为发布是根据已知 bug 的情况来决定的，而不是根据一个事先制定好的时间表。”

子系统特定树

各种内核子系统的维护者——以及许多内核子系统开发人员——在源代码库中公开了他们当前的开发状态。这样，其他人就可以看到内核的不同区域发生了什么。在开发速度很快的领域，可能会要求开发人员将提交的内容建立在这样的子系统内核树上，这样就避免了提交与其他已经进行的工作之间的冲突。

这些存储库中的大多数都是 Git 树，但是也有其他的 scm 在使用，或者补丁队列被发布为 Quilt 系列。这些子系统存储库的地址列在 MAINTAINERS 文件中。其中许多可以在 https://git.kernel.org/上浏览。

在将一个建议的补丁提交到这样的子系统树之前，需要对它进行审查，审查主要发生在邮件列表上（请参见下面相应的部分）。对于几个内核子系统，这个审查过程是通过工具补丁跟踪的。Patchwork 提供了一个 Web 界面，显示补丁发布、对补丁的任何评论或修订，维护人员可以将补丁标记为正在审查、接受或拒绝。大多数补丁网站都列在 https://patchwork.kernel.org/

Linux-next 集成测试树

在将子系统树的更新合并到主线树之前，需要对它们进行集成测试。为此，存在一个特殊的测试存储库，其中几乎每天都会提取所有子系统树：

> https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git

通过这种方式，Linux-next 对下一个合并阶段将进入主线内核的内容给出了一个概要展望。非常欢冒险的测试者运行测试 Linux-next。

多个主要版本的稳定版内核树

由 3 个数字组成的内核版本号说明此内核是-stable 版本。它们包含内核的相对较小且至关重要的修补，这些修补针对安全性问题或者严重的内核退步。

这种版本的内核适用于那些期望获得最新的稳定版内核并且不想参与测试开发版或者实验版的用户。

稳定版内核树版本由“稳定版”小组（邮件地址 <stable@vger.kernel.org>）维护，一般隔周发布新版本。

内核源码中的 Documentation/process/stable-kernel-rules.rst 文件具体描述了可被稳定版内核接受的修改类型以及发布的流程。

报告 bug

bugzilla.kernel.org 是 Linux 内核开发者们用来跟踪内核 Bug 的网站。我们鼓励用户在这个工具中报告找到的所有 bug。如何使用内核 bugzilla 的细节请访问：

> http://test.kernel.org/bugzilla/faq.html

内核源码主目录中的:ref:*admin-guide/reporting-bugs.rst <reportingbugs>* 文件里有一个很好的模板。它指导用户如何报告可能的内核 bug 以及需要提供哪些信息来帮助内核开发者们找到问题的根源。

利用 bug 报告

练习内核开发技能的最好办法就是修改其他人报告的 bug。你不光可以帮助内核变得更加稳定，还可以学会如何解决实际问题从而提高自己的技能，并且让其他开发者感受到你的存在。修改 bug 是赢得其他开发者赞誉的最好办法，因为并不是很多人都喜欢浪费时间去修改别人报告的 bug。

要尝试修改已知的 bug，请访问 http://bugzilla.kernel.org 网址。

邮件列表

正如上面的文档所描述，大多数的骨干内核开发者都加入了 Linux Kernel 邮件列表。如何订阅和退订列表的细节可以在这里找到：

> http://vger.kernel.org/vger-lists.html#linux-kernel

网上很多地方都有这个邮件列表的存档 (archive)。可以使用搜索引擎来找到这些存档。比如：

> http://dir.gmane.org/gmane.linux.kernel

在发信之前，我们强烈建议你先在存档中搜索你想要讨论的问题。很多已经被详细讨论过的问题只在邮件列表的存档中可以找到。

大多数内核子系统也有自己独立的邮件列表来协调各自的开发工作。从 MAINTAINERS 文件中可以找到不同话题对应的邮件列表。

很多邮件列表架设在 kernel.org 服务器上。这些列表的信息可以在这里找到：

> http://vger.kernel.org/vger-lists.html

在使用这些邮件列表时，请记住保持良好的行为习惯。下面的链接提供了与这些列表（或任何其它邮件列表）交流的一些简单规则，虽然内容有点滥竽充数。

> http://www.albion.com/netiquette/

当有很多人回复你的邮件时，邮件的抄送列表会变得很长。请不要将任何人从抄送列表中删除，除非你有足够的理由这么做。也不要只回复到邮件列表。请习惯于同一封邮件接收两次（一封来自发送者一封来自邮件列表），而不要试图通过添加一些奇特的邮件头来解决这个问题，人们不会喜欢的。

记住保留你所回复内容的上下文和源头。在你回复邮件的顶部保留“某某某说到……”这几行。将你的评论加在被引用的段落之间而不要放在邮件的顶部。

如果你在邮件中附带补丁，请确认它们是可以直接阅读的纯文本（如如何让你的改动进入内核文档中所述）。内核开发者们不希望遇到附件或者被压缩了的补丁。只有这样才能保证他们可以直接评论你的每行代码。请确保你使用的邮件发送程序不会修改空格和制表符。一个防范性的测试方法是先将邮件发送给自己，然后自己尝试是否可以顺利地打上收到的补丁。如果测试不成功，请调整或者更换你的邮件发送程序直到它正确工作为止。

总而言之，请尊重其他的邮件列表订阅者。

同内核社区合作

内核社区的目标就是提供尽善尽美的内核。所以当你提交补丁期望被接受进内核的时候，它的技术价值以及其他方面都将被评审。那么你可能会得到什么呢?

- 批评
- 评论
- 要求修改
- 要求证明修改的必要性
- 沉默

要记住，这些是把补丁放进内核的正常情况。你必须学会听取对补丁的批评和评论，从技术层面评估它们，然后要么重写你的补丁要么简明扼要地论证修改是不必要的。如果你发的邮件没有得到任何回应，请过几天后再试一次，因为有时信件会湮没在茫茫信海中。

你不应该做的事情：

- 期望自己的补丁不受任何质疑就直接被接受
- 翻脸
- 忽略别人的评论
- 没有按照别人的要求做任何修改就重新提交

在一个努力追寻最好技术方案的社区里，对于一个补丁有多少好处总会有不同的见解。你必须要抱着合作的态度，愿意改变自己的观点来适应内核的风格。或者至少愿意去证明你的想法是有价值的。记住，犯错误是允许的，只要你愿意朝着正确的方案去努力。

如果你的第一个补丁换来的是一堆修改建议，这是很正常的。这并不代表你的补丁不会被接受，也不意味着有人和你作对。你只需要改正所有提出的问题然后重新发送你的补丁。

内核社区和公司文化的差异

内核社区的工作模式同大多数传统公司开发队伍的工作模式并不相同。下面这些例子，可以帮助你避免某些可能发生问题：用这些话介绍你的修改提案会有好处：

- 它同时解决了多个问题
- 它删除了 2000 行代码
- 这是补丁，它已经解释了我想要说明的
- 我在 5 种不同的体系结构上测试过它……
- 这是一系列小补丁用来……
- 这个修改提高了普通机器的性能……

应该避免如下的说法：

- 我们在 AIX/ptx/Solaris 就是这么做的，所以这么做肯定是好的……
- 我做这行已经 20 年了，所以……
- 为了我们公司赚钱考虑必须这么做
- 这是我们的企业产品线所需要的

- 这里是描述我观点的 1000 页设计文档
- 这是一个 5000 行的补丁用来……
- 我重写了现在乱七八糟的代码，这就是……
- 我被规定了最后期限，所以这个补丁需要立刻被接受

另外一个内核社区与大部分传统公司的软件开发队伍不同的地方是无法面对面地交流。使用电子邮件和 IRC 聊天工具做为主要沟通工具的一个好处是性别和种族歧视将会更少。Linux 内核的工作环境更能接受妇女和少数族群，因为每个人在别人眼里只是一个邮件地址。国际化也帮助了公平的实现，因为你无法通过姓名来判断人的性别。男人有可能叫李丽，女人也有可能叫王刚。大多数在 Linux 内核上工作过并表达过看法的女性对在 linux 上工作的经历都给出了正面的评价。

对于一些不习惯使用英语的人来说，语言可能是一个引起问题的障碍。在邮件列表中要正确地表达想法必需良好地掌握语言，所以建议你在发送邮件之前最好检查一下英文写得是否正确。

拆分修改

Linux 内核社区并不喜欢一下接收大段的代码。修改需要被恰当地介绍、讨论并且拆分成独立的小段。这几乎完全和公司中的习惯背道而驰。你的想法应该在开发最开始的阶段就让大家知道，这样你就可以及时获得对你正在进行的开发的反馈。这样也会让社区觉得你是在和他们协作，而不是仅仅把他们当作倾销新功能的对象。无论如何，你不要一次性地向邮件列表发送 50 封信，你的补丁序列应该永远用不到这么多。

将补丁拆开的原因如下：

1) 小的补丁更有可能被接受，因为它们不需要太多的时间和精力去验证其正确性。一个 5 行的补丁，可能在维护者看了一眼以后就会被接受。而 500 行的补丁则需要数个小时来审查其正确性（所需时间随补丁大小增加大约呈指数级增长）。

   当出了问题的时候，小的补丁也会让调试变得非常容易。一个一个补丁地回溯将会比仔细剖析一个被打上的大补丁（这个补丁破坏了其他东西）容易得多。

2）不光发送小的补丁很重要，在提交之前重新编排、化简（或者仅仅重新排列）
   补丁也是很重要的。

这里有内核开发者 Al Viro 打的一个比方：
   “想象一个老师正在给学生批改数学作业。老师并不希望看到学生为了得到正确解法所进行的尝试和产生的错误。他希望看到的是最干净最优雅的解答。好学生了解这点，绝不会把最终解决之前的中间方案提交上去。”

   内核开发也是这样。维护者和评审者不希望看到一个人在解决问题时的思考过程。他们只希望看到简单和优雅的解决方案。

直接给出一流的解决方案，和社区一起协作讨论尚未完成的工作，这两者之间似乎很难找到一个平衡点。所以最好尽早开始收集有利于你进行改进的反馈；同时也要保证修改分成很多小块，这样在整个项目都准备好被包含进内核之前，其中的一部分可能会先被接收。

必须了解这样做是不可接受的：试图将未完成的工作提交进内核，然后再找时间修复。

证明修改的必要性

除了将补丁拆成小块，很重要的一点是让 Linux 社区了解他们为什么需要这样修改。你必须证明新功能是有人需要的并且是有用的。

记录修改

当你发送补丁的时候，需要特别留意邮件正文的内容。因为这里的信息将会做为补丁的修改记录 (ChangeLog)，会被一直保留以备大家查阅。它需要完全地描述补丁，包括：

- 为什么需要这个修改
- 补丁的总体设计
- 实现细节
- 测试结果

想了解它具体应该看起来像什么，请查阅以下文档中的“ChangeLog”章节：

> **“The Perfect Patch”**
> https://www.ozlabs.org/~akpm/stuff/tpp.txt

这些事情有时候做起来很难。要在任何方面都做到完美可能需要好几年时间。这是一个持续提高的过程，它需要大量的耐心和决心。只要不放弃，你一定可以做到。很多人已经做到了，而他们都曾经和现在的你站在同样的起点上。

感谢

感谢 Paolo Ciarrocchi 允许 “开发流程” 部分基于他所写的文章 (http://www.kerneltravel.net/newbie/2.6-development_process)，感谢 Randy Dunlap 和 Gerrit Huizenga 完善了应该说和不该说的列表。感谢 Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk 和 Alex Shepard 的评审、建议和贡献。没有他们的帮助，这篇文档是不可能完成的。

英文版维护者：Greg Kroah-Hartman <greg@kroah.com>

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/code-of-conduct.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 贡献者契约行为准则

我们的誓言

为了营造一个开放、友好的环境，我们作为贡献者和维护人承诺，让我们的社区和参与者，拥有一个无骚扰的体验，无论年龄、体型、残疾、种族、性别特征、性别认同和表达、经验水平、教育程度、社会状况，经济地位、国籍、个人外貌、种族、宗教或性身份和取向。

我们的标准

有助于创造积极环境的行为包括：

- 使用欢迎和包容的语言
- 尊重不同的观点和经验
- 优雅地接受建设性的批评
- 关注什么对社区最有利
- 对其他社区成员表示同情

参与者的不可接受行为包括：

- 使用性意味的语言或意象以及不受欢迎的性注意或者更过分的行为
- 煽动、侮辱/贬损评论以及个人或政治攻击
- 公开或私下骚扰
- 未经明确许可，发布他人的私人信息，如物理或电子地址。
- 在专业场合被合理认为不适当的其他行为

我们的责任

维护人员负责澄清可接受行为的标准，并应针对任何不可接受行为采取适当和公平的纠正措施。

维护人员有权和责任删除、编辑或拒绝与本行为准则不一致的评论、承诺、代码、wiki 编辑、问题和其他贡献，或暂时或永久禁止任何贡献者从事他们认为不适当、威胁、冒犯或有害的其他行为。

范围

当个人代表项目或其社区时，本行为准则既适用于项目空间，也适用于公共空间。代表一个项目或社区的例子包括使用一个正式的项目电子邮件地址，通过一个正式的社交媒体帐户发布，或者在在线或离线事件中担任指定的代表。项目维护人员可以进一步定义和澄清项目的表示。

执行

如有滥用、骚扰或其他不可接受的行为，可联系行为准则委员会 <conduct@kernel.org>。所有投诉都将接受审查和调查，并将得到必要和适当的答复。行为准则委员会有义务对事件报告人保密。具体执行政策的进一步细节可单独公布。

归属

本行为准则改编自《贡献者契约》，版本 1.4，可从 https://www.contributor-covenant.org/version/1/4/code-of-conduct.html 获取。

解释

有关 Linux 内核社区如何解释此文档，请参阅*Linux 内核贡献者契约行为准则解释*

> **Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/code-of-conduct-interpretation.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核贡献者契约行为准则解释

贡献者契约行为准则 准则是一个通用文档，旨在为几乎所有开源社区提供一套规则。每个开源社区都是独一无二的，Linux 内核也不例外。因此，本文描述了 Linux 内核社区中如何解释它。我们也不希望这种解释随着时间的推移是静态的，并将根据需要进行调整。

与开发软件的“传统”方法相比，Linux 内核开发工作是一个非常个人化的过程。你的贡献和背后的想法将被仔细审查，往往导致批判和批评。审查将几乎总是需要改进，材料才能包括在内核中。要知道这是因为所有相关人员都希望看到 Linux 整体成功的最佳解决方案。这个开发过程已经被证明可以创建有史以来最健壮的操作系统内核，我们不想做任何事情来导致提交质量和最终结果的下降。

维护者

行为准则多次使用“维护者”一词。在内核社区中，“维护者”是负责子系统、驱动程序或文件的任何人，并在内核源代码树的维护者文件中列出。

责任

《行为准则》提到了维护人员的权利和责任，这需要进一步澄清。

首先，最重要的是，有一个合理的期望是由维护人员通过实例来领导。

也就是说，我们的社区是广阔的，对维护者没有新的要求，他们单方面处理其他人在他们活跃的社区的行为。这一责任由我们所有人承担，最终《行为准则》记录了最终的上诉路径，以防有关行为问题的问题悬而未决。

维护人员应该愿意在出现问题时提供帮助，并在需要时与社区中的其他人合作。如果您不确定如何处理出现的情况，请不要害怕联系技术咨询委员会（TAB）或其他维护人员。除非您愿意，否则不会将其视为违规报告。如果您不确定是否该联系 TAB 或任何其他维护人员，请联系我们的冲突调解人 Mishi Choudhary <mishi@linux.com>。

最后，“善待对方”才是每个人的最终目标。我们知道每个人都是人，有时我们都会失败，但我们所有人的首要目标应该是努力友好地解决问题。执行行为准则将是最后的选择。

我们的目标是创建一个强大的、技术先进的操作系统，以及所涉及的技术复杂性，这自然需要专业知识和决策。

所需的专业知识因贡献领域而异。它主要由上下文和技术复杂性决定，其次由贡献者和维护者的期望决定。

专家的期望和决策都要经过讨论，但在最后，为了取得进展，必须能够做出决策。这一特权掌握在维护人员和项目领导的手中，预计将善意使用。

因此，设定专业知识期望、作出决定和拒绝不适当的贡献不被视为违反行为准则。

虽然维护人员一般都欢迎新来者，但他们帮助（新）贡献者克服障碍的能力有限，因此他们必须确定优先事项。这也不应被视为违反了行为准则。内核社区意识到这一点，并以各种形式提供入门级节目，如 kernelnewbies.org 。

范围

Linux 内核社区主要在一组公共电子邮件列表上进行交互，这些列表分布在由多个不同公司或个人控制的多个不同服务器上。所有这些列表都在内核源代码树中的 MAINTAINERS 文件中定义。发送到这些邮件列表的任何电子邮件都被视为包含在行为准则中。

使用 kernel.org bugzilla 和其他子系统 bugzilla 或 bug 跟踪工具的开发人员应该遵循行为准则的指导原则。Linux 内核社区没有“官方”项目电子邮件地址或“官方”社交媒体地址。使用 kernel.org 电子邮件帐户执行的任何活动必须遵循为 kernel.org 发布的行为准则，就像任何使用公司电子邮件帐户的个人必须遵循该公司的特定规则一样。

行为准则并不禁止在邮件列表消息、内核更改日志消息或代码注释中继续包含名称、电子邮件地址和相关注释。

其他论坛中的互动包括在适用于上述论坛的任何规则中，通常不包括在行为准则中。除了在极端情况下可考虑的例外情况。

提交给内核的贡献应该使用适当的语言。在行为准则之前已经存在的内容现在不会被视为违反。然而，不适当的语言可以被视为一个 bug；如果任何相关方提交补丁，这样的 bug 将被更快地修复。当前属于用户/内核 API 的一部分的表达式，或者反映已发布标准或规范中使用的术语的表达式，不被视为 bug。

执行

行为准则中列出的地址属于行为准则委员会。https://kernel.org/code-of-conduct.html 列出了在任何给定时间接收这些电子邮件的确切成员。成员不能访问在加入委员会之前或离开委员会之后所做的报告。

最初的行为准则委员会由 TAB 的志愿者以及作为中立第三方的专业调解人组成。委员会的首要任务是建立文件化的流程，并将其公开。

如果报告人不希望将整个委员会纳入投诉或关切，可直接联系委员会的任何成员，包括调解人。

行为准则委员会根据流程审查案例（见上文），并根据需要和适当与 TAB 协商，例如请求和接收有关内核社区的信息。

委员会做出的任何决定都将提交到表中，以便在必要时与相关维护人员一起执行。行为准则委员会的决定可以通过三分之二的投票推翻。

每季度，行为准则委员会和标签将提供一份报告，概述行为准则委员会收到的匿名报告及其状态，以及任何否决决定的细节，包括完整和可识别的投票细节。

我们希望在启动期之后为行为准则委员会人员配备建立一个不同的流程。发生此情况时，将使用该信息更新此文档。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/submitting-patches.rst

译者:

```
中文版维护者:  钟宇 TripleX Chung <xxx.phy@gmail.com>
中文版翻译者:  钟宇 TripleX Chung <xxx.phy@gmail.com>
               时奎亮 Alex Shi <alex.shi@linux.alibaba.com>
中文版校译者:  李阳 Li Yang <leoyang.li@nxp.com>
               王聪 Wang Cong <xiyou.wangcong@gmail.com>
```

* 如何让你的改动进入内核

对于想要将改动提交到 Linux 内核的个人或者公司来说，如果不熟悉“规矩”，提交的流程会让人畏惧。本文档收集了一系列建议，这些建议可以大大的提高你的改动被接受的机会.

以下文档含有大量简洁的建议，具体请见：Documentation/process 同样，Linux 内核补丁提交清单 给出在提交代码前需要检查的项目的列表。如果你在提交一个驱动程序，那么同时阅读一下：Documentation/process/submitting-drivers.rst

其中许多步骤描述了 Git 版本控制系统的默认行为；如果您使用 Git 来准备补丁，您将发现它为您完成的大部分机械工作，尽管您仍然需要准备和记录一组合理的补丁。一般来说，使用 git 将使您作为内核开发人员的生活更轻松。

0) 获取当前源码树

如果您没有一个可以使用当前内核源代码的存储库，请使用 git 获取一个。您将要从主线存储库开始，它可以通过以下方式获取:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/
↪linux.git
```

但是，请注意，您可能不希望直接针对主线树进行开发。大多数子系统维护人员运行自己的树，并希望看到针对这些树准备的补丁。请参见 MAINTAINERS 文件中子系统的 **T:** 项以查找该树，或者简单地询问维护者该树是否未在其中列出。

仍然可以通过 tarballs 下载内核版本（如下一节所述），但这是进行内核开发的一种困难的方式。

1) “diff -up”

使用“diff -up”或者“diff -uprN”来创建补丁。

所有内核的改动，都是以补丁的形式呈现的，补丁由 diff(1) 生成。创建补丁的时候，要确认它是以“unified diff”格式创建的，这种格式由 diff(1) 的‘-u’参数生成。而且，请使用‘-p’参数，那样会显示每个改动所在的 C 函数，使得产生的补丁容易读得多。补丁应该基于内核源代码树的根目录，而不是里边的任何子目录。

为一个单独的文件创建补丁，一般来说这样做就够了:

```
SRCTREE=linux
MYFILE=drivers/net/mydriver.c

cd $SRCTREE
cp $MYFILE $MYFILE.orig
vi $MYFILE      # make your change
cd ..
diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

为多个文件创建补丁，你可以解开一个没有修改过的内核源代码树，然后和你自己的代码树之间做 diff 。例如:

```
MYSRC=/devel/linux

tar xvfz linux-3.19.tar.gz
mv linux-3.19 linux-3.19-vanilla
diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
        linux-3.19-vanilla $MYSRC > /tmp/patch
```

“dontdiff”是内核在编译的时候产生的文件的列表，列表中的文件在 diff(1) 产生的补丁里会被跳过。

确定你的补丁里没有包含任何不属于这次补丁提交的额外文件。记得在用 diff(1) 生成补丁之后，审阅一次补丁，以确保准确。

如果你的改动很散乱，你应该研究一下如何将补丁分割成独立的部分，将改动分割成一系列合乎逻辑的步骤。这样更容易让其他内核开发者审核，如果你想你的补丁被接受，这是很重要的。请参阅：*3) 拆分你的改动*

如果你用 `git`，`git rebase -i` 可以帮助你这一点。如果你不用 `git`, `quilt` <https://savannah.nongnu.org/projects/quilt> 另外一个流行的选择。

2) 描述你的改动

描述你的问题。无论您的补丁是一行错误修复还是 5000 行新功能，都必须有一个潜在的问题激励您完成这项工作。让审稿人相信有一个问题值得解决，让他们读完第一段是有意义的。

描述用户可见的影响。直接崩溃和锁定是相当有说服力的，但并不是所有的错误都那么明目张胆。即使在代码审查期间发现了这个问题，也要描述一下您认为它可能对用户产生的影响。请记住，大多数 Linux 安装运行的内核来自二级稳定树或特定于供应商/产品的树，只从上游精选特定的补丁，因此请包含任何可以帮助您将更改定位到下游的内容：触发的场景、DMESG 的摘录、崩溃描述、性能回归、延迟尖峰、锁定等。

量化优化和权衡。如果您声称在性能、内存消耗、堆栈占用空间或二进制大小方面有所改进，请包括支持它们的数字。但也要描述不明显的成本。优化通常不是免费的，而是在 CPU、内存和可读性之间进行权衡；或者，探索性的工作，在不同的工作负载之间进行权衡。请描述优化的预期缺点，以便审阅者可以权衡成本和收益。

一旦问题建立起来，就要详细地描述一下您实际在做什么。对于审阅者来说，用简单的英语描述代码的变化是很重要的，以验证代码的行为是否符合您的意愿。

如果您将补丁描述写在一个表单中，这个表单可以很容易地作为“提交日志”放入 Linux 的源代码管理系统 git 中，那么维护人员将非常感谢您。见*15) 明确回复邮件头 (In-Reply-To)*.

每个补丁只解决一个问题。如果你的描述开始变长，这就表明你可能需要拆分你的补丁。请见*3) 拆分你的改动*

提交或重新提交修补程序或修补程序系列时，请包括完整的修补程序说明和理由。不要只说这是补丁（系列）的第几版。不要期望子系统维护人员引用更早的补丁版本或引用 URL 来查找补丁描述并将其放入补丁中。也就是说，补丁（系列）及其描述应该是独立的。这对维护人员和审查人员都有好处。一些评审者可能甚至没有收到补丁的早期版本。

描述你在命令语气中的变化，例如“make xyzzy do frotz”而不是“[这个补丁]make xyzzy do frotz”或“[我]changed xyzzy to do frotz”，就好像你在命令代码库改变它的行为一样。

如果修补程序修复了一个记录的 bug 条目，请按编号和 URL 引用该 bug 条目。如果补丁来自邮件列表讨论，请给出邮件列表存档的 URL；使用带有 `Message-ID` 的 https://lkml.kernel.org/ 重定向，以确保链接不会过时。

但是，在没有外部资源的情况下，尽量让你的解释可理解。除了提供邮件列表存档或 bug 的 URL 之外，还要总结需要提交补丁的相关讨论要点。

如果您想要引用一个特定的提交，不要只引用提交的 SHA-1 ID。还请包括提交的一行摘要，以便于审阅者了解它是关于什么的。例如:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

您还应该确保至少使用前 12 位 SHA-1 ID. 内核存储库包含 * 许多 * 对象，使与较短的 ID 发生冲突的可能性很大。记住，即使现在不会与您的六个字符 ID 发生冲突，这种情况可能五年后改变。

如果修补程序修复了特定提交中的错误，例如，使用 `git bisct` ，请使用带有前 12 个字符 SHA-1 ID 的”Fixes:”标记和单行摘要。为了简化不要将标记拆分为多个，行、标记不受分析脚本“75 列换行”规则的限制。例如:

```
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the␣
↪number of pages it actually freed")
```

下列 `git config` 设置可以添加让 `git log`, `git show` 漂亮的显示格式:

```
[core]
        abbrev = 12
[pretty]
        fixes = Fixes: %h (\"%s\")
```

3) 拆分你的改动

将每个逻辑更改分隔成一个单独的补丁。

例如，如果你的改动里同时有 bug 修正和性能优化，那么把这些改动拆分到两个或者更多的补丁文件中。如果你的改动包含对 API 的修改，并且修改了驱动程序来适应这些新的 API，那么把这些修改分成两个补丁。

另一方面，如果你将一个单独的改动做成多个补丁文件，那么将它们合并成一个单独的补丁文件。这样一个逻辑上单独的改动只被包含在一个补丁文件里。

如果有一个补丁依赖另外一个补丁来完成它的改动，那没问题。简单的在你的补丁描述里指出“这个补丁依赖某补丁”就好了。

在将您的更改划分为一系列补丁时，要特别注意确保内核在系列中的每个补丁之后都能正常构建和运行。使用 `git bisect` 来追踪问题的开发者可能会在任何时候分割你的补丁系列；如果你在中间引入错误，他们不会感谢你。

如果你不能将补丁浓缩成更少的文件，那么每次大约发送出 15 个，然后等待审查和集成。

4) 检查你的更改风格

检查您的补丁是否存在基本样式冲突，详细信息可在Linux 内核代码风格 中找到。如果不这样做，只会浪费审稿人的时间，并且会导致你的补丁被拒绝，甚至可能没有被阅读。

一个重要的例外是在将代码从一个文件移动到另一个文件时——在这种情况下，您不应该在移动代码的同一个补丁中修改移动的代码。这清楚地描述了移动代码和您的更改的行为。这大大有助于审查实际差异，并允许工具更好地跟踪代码本身的历史。

在提交之前，使用补丁样式检查程序检查补丁（scripts/check patch.pl）。不过，请注意，样式检查程序应该被视为一个指南，而不是作为人类判断的替代品。如果您的代码看起来更好，但有违规行为，那么最好不要使用它。

检查者报告三个级别：

- ERROR：很可能出错的事情
- WARNING：需要仔细审查的事项
- CHECK：需要思考的事情

您应该能够判断您的补丁中存在的所有违规行为。

5) 选择补丁收件人

您应该总是在任何补丁上复制相应的子系统维护人员，以获得他们维护的代码；查看维护人员文件和源代码修订历史记录，以了解这些维护人员是谁。脚本 scripts/get_Maintainer.pl 在这个步骤中非常有用。如果您找不到正在工作的子系统的维护人员，那么 Andrew Morton（akpm@linux-foundation.org）将充当最后的维护人员。

您通常还应该选择至少一个邮件列表来接收补丁集的。linux-kernel@vger.kernel.org 作为最后一个解决办法的列表，但是这个列表上的体积已经引起了许多开发人员的拒绝。在 MAINTAINERS 文件中查找子系统特定的列表；您的补丁可能会在那里得到更多的关注。不过，请不要发送垃圾邮件到无关的列表。

许多与内核相关的列表托管在 vger.kernel.org 上；您可以在 http://vger.kernel.org/vger-lists.html 上找到它们的列表。不过，也有与内核相关的列表托管在其他地方。

不要一次发送超过 15 个补丁到 vger 邮件列表!!!!

Linus Torvalds 是决定改动能否进入 Linux 内核的最终裁决者。他的 e-mail 地址是 <torvalds@linux-foundation.org> 。他收到的 e-mail 很多，所以一般的说，最好别给他发 e-mail。

如果您有修复可利用安全漏洞的补丁，请将该补丁发送到 security@kernel.org。对于严重的 bug，可以考虑短期暂停以允许分销商向用户发布补丁；在这种情况下，显然不应将补丁发送到任何公共列表。

修复已发布内核中严重错误的补丁程序应该指向稳定版维护人员，方法是放这样的一行:

```
Cc: stable@vger.kernel.org
```

进入补丁的签准区（注意，不是电子邮件收件人）。除了这个文件之外，您还应该阅读 Documentation/process/stable-kernel-rules.rst

但是，请注意，一些子系统维护人员希望得出他们自己的结论，即哪些补丁应该被放到稳定的树上。尤其是网络维护人员，不希望看到单个开发人员在补丁中添加像上面这样的行。

如果更改影响到用户和内核接口，请向手册页维护人员（如维护人员文件中所列）发送手册页补丁，或至少发送更改通知，以便一些信息进入手册页。还应将用户空间 API 更改复制到 linux-api@vger.kernel.org。

对于小的补丁，你也许会 CC 到搜集琐碎补丁的邮件列表 (Trivial Patch Monkey) trivial@kernel.org，那里专门收集琐碎的补丁。下面这样的补丁会被看作“琐碎的”补丁：

- 文档的拼写修正。
- 修正会影响到 grep(1) 的拼写。
- 警告信息修正 (频繁的打印无用的警告是不好的。)
- 编译错误修正（代码逻辑的确是对的，只是编译有问题。）
- 运行时修正（只要真的修正了错误。）
- 移除使用了被废弃的函数/宏的代码 (例如 check_region。)
- 联系方式和文档修正。
- 用可移植的代码替换不可移植的代码（即使在体系结构相关的代码中，既然有
- 人拷贝，只要它是琐碎的）
- 任何文件的作者/维护者对该文件的改动（例如 patch monkey 在重传模式下）

(译注，关于“琐碎补丁”的一些说明：因为原文的这一部分写得比较简单，所以不得不违例写一下译注。”trivial”这个英文单词的本意是“琐碎的，不重要的。”但是在这里有稍微有一些变化，例如对一些明显的 NULL 指针的修正，属于运行时修正，会被归类到琐碎补丁里。虽然 NULL 指针的修正很重要，但是这样的修正往往很小而且很容易得到检验，所以也被归入琐碎补丁。琐碎补丁更精确的归类应该是“simple, localized & easy to verify”，也就是说简单的，局部的和易于检验的。trivial@kernel.org 邮件列表的目的是针对这样的补丁，为提交者提供一个中心，来降低提交的门槛。)

6) 没有 MIME 编码，没有链接，没有压缩，没有附件，只有纯文本

Linus 和其他的内核开发者需要阅读和评论你提交的改动。对于内核开发者来说，可以“引用”你的改动很重要，使用一般的 e-mail 工具，他们就可以在你的代码的任何位置添加评论。

因为这个原因，所有的提交的补丁都是 e-mail 中“内嵌”的。

> **Warning:** 如果你使用剪切-粘贴你的补丁，小心你的编辑器的自动换行功能破坏你的补丁

不要将补丁作为 MIME 编码的附件，不管是否压缩。很多流行的 e-mail 软件不是任何时候都将 MIME 编码的附件当作纯文本发送的，这会使得别人无法在你的代码中加评论。另外，MIME 编码的附件会让 Linus 多花一点时间来处理，这就降低了你的改动被接受的可能性。

例外：如果你的邮递员弄坏了补丁，那么有人可能会要求你使用 mime 重新发送补丁

请参阅Linux 邮件客户端配置信息 以获取有关配置电子邮件客户端以使其不受影响地发送修补程序的提示。

7) e-mail 的大小

大的改动对邮件列表不合适，对某些维护者也不合适。如果你的补丁，在不压缩的情况下，超过了 300kB，那么你最好将补丁放在一个能通过 internet 访问的服务器上，然后用指向你的补丁的 URL 替代。但是请注意，如果您的补丁超过了 300kb，那么它几乎肯定需要被破坏。

8）回复评审意见

你的补丁几乎肯定会得到评审者对补丁改进方法的评论。您必须对这些评论作出回应；让补丁被忽略的一个好办法就是忽略审阅者的意见。不会导致代码更改的意见或问题几乎肯定会带来注释或变更日志的改变，以便下一个评审者更好地了解正在发生的事情。

一定要告诉审稿人你在做什么改变，并感谢他们的时间。代码审查是一个累人且耗时的过程，审查人员有时会变得暴躁。即使在这种情况下，也要礼貌地回应并解决他们指出的问题。

9）不要泄气或不耐烦

提交更改后，请耐心等待。审阅者是忙碌的人，可能无法立即访问您的修补程序。

曾几何时，补丁曾在没有评论的情况下消失在空白中，但开发过程比现在更加顺利。您应该在一周左右的时间内收到评论；如果没有收到评论，请确保您已将补丁发送到正确的位置。在重新提交或联系审阅者之前至少等待一周-在诸如合并窗口之类的繁忙时间可能更长。

10）主题中包含 PATCH

由于到 linus 和 linux 内核的电子邮件流量很高，通常会在主题行前面加上 [PATCH] 前缀. 这使 Linus 和其他内核开发人员更容易将补丁与其他电子邮件讨论区分开。

11）签署你的作品-开发者原始认证

为了加强对谁做了何事的追踪，尤其是对那些透过好几层的维护者的补丁，我们建议在发送出去的补丁上加一个“sign-off”的过程。

“sign-off”是在补丁的注释的最后的简单的一行文字，认证你编写了它或者其他人有权力将它作为开放源代码的补丁传递。规则很简单：如果你能认证如下信息：

开发者来源证书 1.1

对于本项目的贡献，我认证如下信息：

（a）这些贡献是完全或者部分的由我创建，我有权利以文件中指出
的开放源代码许可证提交它；或者

（b）这些贡献基于以前的工作，据我所知，这些以前的工作受恰当的开放
源代码许可证保护，而且，根据许可证，我有权提交修改后的贡献，无论是完全还是部分由我创造，这些贡献都使用同一个开放源代码许可证（除非我被允许用其它的许可证），正如文件中指出的；或者

（c）这些贡献由认证（a），（b）或者（c）的人直接提供给我，而
且我没有修改它。

（d）我理解并同意这个项目和贡献是公开的，贡献的记录（包括我
一起提交的个人记录，包括 sign-off ）被永久维护并且可以和这个项目或者开放源代码的许可证同步地再发行。

那么加入这样一行:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

使用你的真名（抱歉，不能使用假名或者匿名。）

有人在最后加上标签。现在这些东西会被忽略，但是你可以这样做，来标记公司内部的过程，或者只是指出关于 sign-off 的一些特殊细节。

如果您是子系统或分支维护人员，有时需要稍微修改收到的补丁，以便合并它们，因为树和提交者中的代码不完全相同。如果你严格遵守规则（c），你应该要求提交者重新发布，但这完全是在浪费时间和精力。规则（b）允许您调整代码，但是更改一个提交者的代码并让他认可您的错误是非常不礼貌的。要解决此问题，建议在最后一个由签名行和您的行之间添加一行，指示更改的性质。虽然这并不是强制性的，但似乎在描述前加上您的邮件和/或姓名（全部用方括号括起来），这足以让人注意到您对最后一分钟的更改负有责任。例如:

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

如果您维护一个稳定的分支机构，同时希望对作者进行致谢、跟踪更改、合并修复并保护提交者不受投诉，那么这种做法尤其有用。请注意，在任何情况下都不能更改作者的 ID（From 头），因为它是出现在更改日志中的标识。

对回合（back-porters）的特别说明：在提交消息的顶部（主题行之后）插入一个补丁的起源指示似乎是一种常见且有用的实践，以便于跟踪。例如，下面是我们在 3.x 稳定版本中看到的内容:

```
Date:   Tue Oct 7 07:26:38 2014 -0400

  libata: Un-break ATA blacklist

  commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

还有，这里是一个旧版内核中的一个回合补丁:

```
Date:   Tue May 13 22:12:27 2008 +0200

    wireless, airo: waitbusy() won't delay

    [backport of 2.6 commit␣
↪b7acbdfbd1f277c1eb23f344f899cfa4cd0bf36a]
```

12）何时使用 Acked-by:，CC:，和 Co-Developed by:

Singed-off-by: 标记表示签名者参与了补丁的开发，或者他/她在补丁的传递路径中。

如果一个人没有直接参与补丁的准备或处理，但希望表示并记录他们对补丁的批准，那么他们可以要求在补丁的变更日志中添加一个 Acked-by:

Acked-by：通常由受影响代码的维护者使用，当该维护者既没有贡献也没有转发补丁时。

Acked-by: 不像签字人那样正式。这是一个记录，确认人至少审查了补丁，并表示接受。因此，补丁合并有时会手动将 Acker 的“Yep，looks good to me”转换为 Acked-By:（但请注意，通常最好要求一个明确的 Ack）。

Acked-by：不一定表示对整个补丁的确认。例如，如果一个补丁影响多个子系统，并且有一个：来自一个子系统维护者，那么这通常表示只确认影响维护者代码的部分。这里应该仔细判断。如有疑问，应参考邮件列表档案中的原始讨论。

如果某人有机会对补丁进行评论，但没有提供此类评论，您可以选择在补丁中添加 Cc：这是唯一一个标签，它可以在没有被它命名的人显式操作的情况下添加，但它应该表明这个人是在补丁上抄送的。讨论中包含了潜在利益相关方。

Co-developed-by: 声明补丁是由多个开发人员共同创建的；当几个人在一个补丁上工作时，它用于将属性赋予共同作者（除了 From: 所赋予的作者之外）。因为 Co-developed-by: 表示作者身份，所以每个共同开发人：必须紧跟在相关合作作者的签名之后。标准的签核程序要求：标记的签核顺序应尽可能反映补丁的时间历史，而不管作者是通过 From ： 还是由 Co-developed-by: 共同开发的。值得注意的是，最后一个签字人：必须始终是提交补丁的开发人员。

注意，当作者也是电子邮件标题“发件人:”行中列出的人时，“From: ”标记是可选的。

作者提交的补丁程序示例:

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

合作开发者提交的补丁示例:

```
From: From Author <from@author.example.org>

<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

13）使用报告人:、测试人:、审核人:、建议人:、修复人:

Reported-by: 给那些发现错误并报告错误的人致谢，它希望激励他们在将来再次帮助我们。请注意，如果 bug 是以私有方式报告的，那么在使用 Reported-by 标记之前，请先请求权限。

Tested-by: 标记表示补丁已由指定的人（在某些环境中）成功测试。这个标签通知维护人员已经执行了一些测试，为将来的补丁提供了一种定位测试人员的方法，并确保测试人员的信誉。

Reviewed-by：相反，根据审查人的声明，表明该补丁已被审查并被认为是可接受的：

审查人的监督声明

通过提供我的 Reviewed-by，我声明：

(a) 我已经对这个补丁进行了一次技术审查，以评估它是否适合被包含到主线内核中。

(b) 与补丁相关的任何问题、顾虑或问题都已反馈给提交者。我对提交者对我的评论的回应感到满意。

(c) 虽然这一提交可能会改进一些东西，但我相信，此时，（1）对内核进行了有价值的修改，（2）没有包含争论中涉及的已知问题。

(d) 虽然我已经审查了补丁并认为它是健全的，但我不会（除非另有明确说明）作出任何保证或保证它将在任何给定情况下实现其规定的目的或正常运行。

Reviewed-by 是一种观点声明，即补丁是对内核的适当修改，没有任何遗留的严重技术问题。任何感兴趣的审阅者（完成工作的人）都可以为一个补丁提供一个 Review-by 标签。此标签用于向审阅者提供致谢，并通知维护者已在修补程序上完成的审阅程度。Reviewed-by: 当由已知了解主题区域并执行彻底检查的审阅者提供时，通常会增加补丁进入内核的可能性。

Suggested-by: 表示补丁的想法是由指定的人提出的，并确保将此想法归功于指定的人。请注意，未经许可，不得添加此标签，特别是如果该想法未在公共论坛上发布。这就是说，如果我们勤快地致谢我们的创意者，他们很有希望在未来得到鼓舞，再次帮助我们。

Fixes: 指示补丁在以前的提交中修复了一个问题。它可以很容易地确定错误的来源，这有助于检查错误修复。这个标记还帮助稳定内核团队确定应该接收修复的稳定内核版本。这是指示补丁修复的错误的首选方法。请参阅*2) 描述你的改动* 描述您的更改以了解更多详细信息。

12）标准补丁格式

本节描述如何格式化补丁本身。请注意，如果您的补丁存储在 Git 存储库中，则可以使用 `git format-patch` 进行正确的补丁格式设置。但是，这些工具无法创建必要的文本，因此请务必阅读下面的说明。

标准的补丁，标题行是:

```
Subject: [PATCH 001/123] 子系统: 一句话概述
```

标准补丁的信体存在如下部分：

- 一个“from”行指出补丁作者。后跟空行（仅当发送修补程序的人不是作者时才需要）。
- 解释的正文，行以 75 列包装，这将被复制到永久变更日志来描述这个补丁。
- 一个空行
- 上面描述的“Signed-off-by”行，也将出现在更改日志中。

- 只包含 `---` 的标记线。
- 任何其他不适合放在变更日志的注释。
- 实际补丁（`diff` 输出）。

标题行的格式，使得对标题行按字母序排序非常的容易 - 很多 e-mail 客户端都可以支持 - 因为序列号是用零填充的，所以按数字排序和按字母排序是一样的。

e-mail 标题中的“子系统”标识哪个内核子系统将被打补丁。

e-mail 标题中的“一句话概述”扼要的描述 e-mail 中的补丁。“一句话概述”不应该是一个文件名。对于一个补丁系列（“补丁系列”指一系列的多个相关补丁），不要对每个补丁都使用同样的“一句话概述”。

记住 e-mail 的“一句话概述”会成为该补丁的全局唯一标识。它会蔓延到 git 的改动记录里。然后“一句话概述”会被用在开发者的讨论里，用来指代这个补丁。用户将希望通过 google 来搜索”一句话概述”来找到那些讨论这个补丁的文章。当人们在两三个月后使用诸如 `gitk` 或 `git log --oneline` 之类的工具查看数千个补丁时，也会很快看到它。

出于这些原因，概述必须不超过 70-75 个字符，并且必须描述补丁的更改以及为什么需要补丁。既要简洁又要描述性很有挑战性，但写得好的概述应该这样做。

概述的前缀可以用方括号括起来：“Subject: [PATCH <tag>…] < 概述 >”。标记不被视为概述的一部分，而是描述应该如何处理补丁。如果补丁的多个版本已发送出来以响应评审（即“v1，v2，v3”）或“rfc”，以指示评审请求，那么通用标记可能包括版本描述符。如果一个补丁系列中有四个补丁，那么各个补丁可以这样编号：1/4、2/4、3/4、4/4。这可以确保开发人员了解补丁应用的顺序，并且他们已经查看或应用了补丁系列中的所有补丁。

一些标题的例子:

```
Subject: [patch 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCHv2 001/207] x86: fix eflags tracking
```

“From”行是信体里的最上面一行，具有如下格式：

From: Patch Author <author@example.com>

“From”行指明在永久改动日志里，谁会被确认为作者。如果没有“From”行，那么邮件头里的“From: “行会被用来决定改动日志中的作者。

说明的主题将会被提交到永久的源代码改动日志里，因此对那些早已经不记得和这个补丁相关的讨论细节的有能力的读者来说，是有意义的。包括补丁程序定位错误的（内核日志消息、OOPS 消息等）症状，对于搜索提交日志以寻找适用补丁的人尤其有用。如果一个补丁修复了一个编译失败，那么可能不需要包含所有编译失败；只要足够让搜索补丁的人能够找到它就行了。与概述一样，既要简洁又要描述性。

“—”标记行对于补丁处理工具要找到哪里是改动日志信息的结束，是不可缺少的。

对于“—”标记之后的额外注解，一个好的用途就是用来写 diffstat，用来显示修改了什么文件和每个文件都增加和删除了多少行。diffstat 对于比较大的补丁特别有用。其余那些只是和时刻或者开发者相关的注解，不合适放到永久的改动日志里的，也应该放这里。使用 diffstat 的选项“-p 1 -w 70”这样文件名就会从内核源代码树的目录开始，不会占用太宽的空间（很容易适合 80 列的宽度，也许会有一些缩进。）

在后面的参考资料中能看到适当的补丁格式的更多细节。

15) 明确回复邮件头 (In-Reply-To)

手动添加回复补丁的的标题头 (In-Reply_To:) 是有帮助的（例如，使用 `git send-email`）将补丁与以前的相关讨论关联起来，例如，将 bug 修复程序链接到电子邮件和 bug 报告。但是，对于多补丁系列，最好避免在回复时使用链接到该系列的旧版本。这样，补丁的多个版本就不会成为电子邮件客户端中无法管理的引用序列。如果链接有用，可以使用 https://lkml.kernel.org/ 重定向器（例如，在封面电子邮件文本中）链接到补丁系列的早期版本。

16) 发送 git pull 请求

如果您有一系列补丁，那么让维护人员通过 git pull 操作将它们直接拉入子系统存储库可能是最方便的。但是，请注意，从开发人员那里获取补丁比从邮件列表中获取补丁需要更高的信任度。因此，许多子系统维护人员不愿意接受请求，特别是来自新的未知开发人员的请求。如果有疑问，您可以在封面邮件中使用 pull 请求作为补丁系列正常发布的一个选项，让维护人员可以选择使用其中之一。

pull 请求的主题行中应该有 [Git Pull]。请求本身应该在一行中包含存储库名称和感兴趣的分支；它应该看起来像:

```
Please pull from

    git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus

to get these changes:
```

pull 请求还应该包含一条整体消息，说明请求中将包含什么，一个补丁本身的 `Git shortlog` 以及一个显示补丁系列整体效果的 `diffstat` 。当然，将所有这些信息收集在一起的最简单方法是让 `git` 使用 `git request-pull` 命令为您完成这些工作。

一些维护人员（包括 Linus）希望看到来自已签名提交的请求；这增加了他们对你的请求信心。特别是，在没有签名标签的情况下，Linus 不会从像 Github 这样的公共托管站点拉请求。

创建此类签名的第一步是生成一个 GNRPG 密钥，并由一个或多个核心内核开发人员对其进行签名。这一步对新开发人员来说可能很困难，但没有办法绕过它。参加会议是找到可以签署您的密钥的开发人员的好方法。

一旦您在 Git 中准备了一个您希望有人拉的补丁系列，就用 `git tag -s` 创建一个签名标记。这将创建一个新标记，标识该系列中的最后一次提交，并包含用您的私钥创建的签名。您还可以将 changelog 样式的消息添加到标记中；这是一个描述拉请求整体效果的理想位置。

如果维护人员将要从中提取的树不是您正在使用的存储库，请不要忘记将已签名的标记显式推送到公共树。

生成拉请求时，请使用已签名的标记作为目标。这样的命令可以实现:

```
git request-pull master git://my.public.tree/linux.git my-signed-tag
```

参考文献

Andrew Morton, “The perfect patch”(tpp).
<https://www.ozlabs.org/~akpm/stuff/tpp.txt>

Jeff Garzik, “Linux kernel patch submission format”.
<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

Greg Kroah-Hartman, “How to piss off a kernel subsystem maintainer”.
<http://www.kroah.com/log/linux/maintainer.html>

<http://www.kroah.com/log/linux/maintainer-02.html>

<http://www.kroah.com/log/linux/maintainer-03.html>

<http://www.kroah.com/log/linux/maintainer-04.html>

<http://www.kroah.com/log/linux/maintainer-05.html>

<http://www.kroah.com/log/linux/maintainer-06.html>

NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people!
<https://lkml.org/lkml/2005/7/11/336>

Kernel Documentation/process/coding-style.rst:
Linux 内核代码风格

Linus Torvalds’s mail on the canonical patch format:
<http://lkml.org/lkml/2005/4/7/183>

Andi Kleen, “On submitting kernel patches”
Some strategies to get difficult or controversial changes in.

http://halobates.de/on-submitting-patches.pdf

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/programming-language.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 程序设计语言

内核是用 C 语言*c-language* 编写的。更准确地说，内核通常是用*gcc* 在 `-std=gnu89` *gcc-c-dialect-options* 下编译的：ISO C90 的 GNU 方言（包括一些 C99 特性）

这种方言包含对语言*gnu-extensions* 的许多扩展，当然，它们许多都在内核中使用。

对于一些体系结构，有一些使用*clang* 和*icc* 编译内核的支持，尽管在编写此文档时还没有完成，仍需要第三方补丁。

属性

在整个内核中使用的一个常见扩展是属性（attributes）*gcc-attribute-syntax* 属性允许将实现定义的语义引入语言实体（如变量、函数或类型），而无需对语言进行重大的语法更改（例如添加新关键字）*n2049*

在某些情况下，属性是可选的（即不支持这些属性的编译器仍然应该生成正确的代码，即使其速度较慢或执行的编译时检查/诊断次数不够）

内核定义了伪关键字（例如，`pure`），而不是直接使用 GNU 属性语法（例如，`__attribute__((__pure__))`），以检测可以使用哪些关键字和/或缩短代码，具体请参阅 `include/linux/compiler_attributes.h`

c-language
http://www.open-std.org/jtc1/sc22/wg14/www/standards

gcc
https://gcc.gnu.org

clang
https://clang.llvm.org

icc
https://software.intel.com/en-us/c-compilers

c-dialect-options
https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html

gnu-extensions
https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html

gcc-attribute-syntax
https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

n2049
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/coding-style.rst

译者:

```
中文版维护者:  张乐 Zhang Le <r0bertz@gentoo.org>
中文版翻译者:  张乐 Zhang Le <r0bertz@gentoo.org>
中文版校译者:  王聪 Wang Cong <xiyou.wangcong@gmail.com>
               wheelz <kernel.zeng@gmail.com>
               管旭东 Xudong Guan <xudong.guan@gmail.com>
               Li Zefan <lizf@cn.fujitsu.com>
               Wang Chen <wangchen@cn.fujitsu.com>
```

* Linux 内核代码风格

这是一个简短的文档，描述了 linux 内核的首选代码风格。代码风格是因人而异的，而且我不愿意把自己的观点强加给任何人，但这就像我去做任何事情都必须遵循的原则那样，我也希望在绝大多数事上保持这种的态度。请 (在写代码时) 至少考虑一下这里的代码风格。

首先，我建议你打印一份 GNU 代码规范，然后不要读。烧了它，这是一个具有重大象征性意义的动作。

不管怎样，现在我们开始:

1) 缩进

制表符是 8 个字符，所以缩进也是 8 个字符。有些异端运动试图将缩进变为 4 (甚至 2！) 字符深，这几乎相当于尝试将圆周率的值定义为 3。

理由：缩进的全部意义就在于清楚的定义一个控制块起止于何处。尤其是当你盯着你的屏幕连续看了 20 小时之后，你将会发现大一点的缩进会使你更容易分辨缩进。

现在，有些人会抱怨 8 个字符的缩进会使代码向右边移动的太远，在 80 个字符的终端屏幕上就很难读这样的代码。这个问题的答案是，如果你需要 3 级以上的缩进，不管用何种方式你的代码已经有问题了，应该修正你的程序。

简而言之，8 个字符的缩进可以让代码更容易阅读，还有一个好处是当你的函数嵌套太深的时候可以给你警告。留心这个警告。

在 switch 语句中消除多级缩进的首选的方式是让 switch 和从属于它的 case 标签对齐于同一列，而不要 两次缩进 case 标签。比如:

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
```

(continues on next page)

(continued from previous page)

```
        mem <<= 10;
        /* fall through */
default:
        break;
}
```

不要把多个语句放在一行里，除非你有什么东西要隐藏：

```
if (condition) do_this;
  do_something_everytime;
```

也不要在一行里放多个赋值语句。内核代码风格超级简单。就是避免可能导致别人误读的表达式。

除了注释、文档和 Kconfig 之外，不要使用空格来缩进，前面的例子是例外，是有意为之。

选用一个好的编辑器，不要在行尾留空格。

2) 把长的行和字符串打散

代码风格的意义就在于使用平常使用的工具来维持代码的可读性和可维护性。

每一行的长度的限制是 80 列，我们强烈建议您遵守这个惯例。

长于 80 列的语句要打散成有意义的片段。除非超过 80 列能显著增加可读性，并且不会隐藏信息。子片段要明显短于母片段，并明显靠右。这同样适用于有着很长参数列表的函数头。然而，绝对不要打散对用户可见的字符串，例如 printk 信息，因为这样就很难对它们 grep。

3) 大括号和空格的放置

C 语言风格中另外一个常见问题是大括号的放置。和缩进大小不同，选择或弃用某种放置策略并没有多少技术上的原因，不过首选的方式，就像 Kernighan 和 Ritchie 展示给我们的，是把起始大括号放在行尾，而把结束大括号放在行首，所以：

```
if (x is true) {
        we do y
}
```

这适用于所有的非函数语句块 (if, switch, for, while, do)。比如：

```
switch (action) {
case KOBJ_ADD:
        return "add";
case KOBJ_REMOVE:
        return "remove";
case KOBJ_CHANGE:
        return "change";
default:
        return NULL;
}
```

不过，有一个例外，那就是函数：函数的起始大括号放置于下一行的开头，所以：

```
int function(int x)
{
        body of function
}
```

全世界的异端可能会抱怨这个不一致性是…呃…不一致的，不过所有思维健全的人都知道 (a) K&R 是 **正确的**并且 (b) K&R 是正确的。此外，不管怎样函数都是特殊的 (C 函数是不能嵌套的)。

注意结束大括号独自占据一行，除非它后面跟着同一个语句的剩余部分，也就是 do 语句中的“while”或者 if 语句中的“else”，像这样：

```
do {
        body of do-loop
} while (condition);
```

和

```
if (x == y) {
        ..
} else if (x > y) {
        ...
} else {
        ....
}
```

理由：K&R。

也请注意这种大括号的放置方式也能使空 (或者差不多空的) 行的数量最小化，同时不失可读性。因此，由于你的屏幕上的新行是不可再生资源 (想想 25 行的终端屏幕)，你将会有更多的空行来放置注释。

当只有一个单独的语句的时候，不用加不必要的大括号。

```
if (condition)
        action();
```

和

```
if (condition)
        do_this();
else
        do_that();
```

这并不适用于只有一个条件分支是单语句的情况；这时所有分支都要使用大括号：

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

3.1) 空格

Linux 内核的空格使用方式 (主要) 取决于它是用于函数还是关键字。(大多数) 关键字后要加一个空格。值得注意的例外是 sizeof, typeof, alignof 和 __attribute__，这些关键字某些程度上看起来更像函数 (它们在 Linux 里也常常伴随小括号而使用，尽管在 C 里这样的小括号不是必需的，就像 `struct fileinfo info;` 声明过后的 `sizeof info`)。

所以在这些关键字之后放一个空格:

```
if, switch, case, for, do, while
```

但是不要在 sizeof, typeof, alignof 或者 __attribute__ 这些关键字之后放空格。例如，

```
s = sizeof(struct file);
```

不要在小括号里的表达式两侧加空格。这是一个 **反例**:

```
s = sizeof( struct file );
```

当声明指针类型或者返回指针类型的函数时，* 的首选使用方式是使之靠近变量名或者函数名，而不是靠近类型名。例子:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

在大多数二元和三元操作符两侧使用一个空格，例如下面所有这些操作符:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

但是一元操作符后不要加空格:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

后缀自加和自减一元操作符前不加空格:

```
++  --
```

前缀自加和自减一元操作符后不加空格:

```
++  --
```

. 和 -> 结构体成员操作符前后不加空格。

不要在行尾留空白。有些可以自动缩进的编辑器会在新行的行首加入适量的空白，然后你就可以直接在那一行输入代码。不过假如你最后没有在那一行输入代码，有些编辑器就不会移除已经加入的空白，就像你故意留下一个只有空白的行。包含行尾空白的行就这样产生了。

当 git 发现补丁包含了行尾空白的时候会警告你，并且可以应你的要求去掉行尾空白；不过如果你是正在打一系列补丁，这样做会导致后面的补丁失败，因为你改变了补丁的上下文。

4) 命名

C 是一个简朴的语言，你的命名也应该这样。和 Modula-2 和 Pascal 程序员不同，C 程序员不使用类似 ThisVariableIsATemporaryCounter 这样华丽的名字。C 程序员会称那个变量为 `tmp` ，这样写起来会更容易，而且至少不会令其难于理解。

不过，虽然混用大小写的名字是不提倡使用的，但是全局变量还是需要一个具描述性的名字。称一个全局函数为 `foo` 是一个难以饶恕的错误。

全局变量 (只有当你 **真正**需要它们的时候再用它) 需要有一个具描述性的名字，就像全局函数。如果你有一个可以计算活动用户数量的函数，你应该叫它 `count_active_users()` 或者类似的名字，你不应该叫它 `cntuser()` 。

在函数名中包含函数类型 (所谓的匈牙利命名法) 是脑子出了问题——编译器知道那些类型而且能够检查那些类型，这样做只能把程序员弄糊涂了。难怪微软总是制造出有问题的程序。

本地变量名应该简短，而且能够表达相关的含义。如果你有一些随机的整数型的循环计数器，它应该被称为 `i` 。叫它 `loop_counter` 并无益处，如果它没有被误解的可能的话。类似的，`tmp` 可以用来称呼任意类型的临时变量。

如果你怕混淆了你的本地变量名，你就遇到另一个问题了，叫做函数增长荷尔蒙失衡综合症。请看第六章 (函数)。

5) Typedef

不要使用类似 `vps_t` 之类的东西。

对结构体和指针使用 typedef 是一个 **错误**。当你在代码里看到：

```
vps_t a;
```

这代表什么意思呢?

相反，如果是这样

```
struct virtual_container *a;
```

你就知道 `a` 是什么了。

很多人认为 typedef 能提高可读性。实际不是这样的。它们只在下列情况下有用：

(a) 完全不透明的对象 (这种情况下要主动使用 typedef 来 **隐藏**这个对象实际上是什么)。

例如：`pte_t` 等不透明对象，你只能用合适的访问函数来访问它们。

Note: 不透明性和“访问函数”本身是不好的。我们使用 pte_t 等类型的原因在于真的是完全没有任何共用的可访问信息。

(b) 清楚的整数类型，如此，这层抽象就可以 **帮助**消除到底是 `int` 还是 `long` 的混淆。

u8/u16/u32 是完全没有问题的 typedef，不过它们更符合类别 (d) 而不是这里。

Note: 要这样做，必须事出有因。如果某个变量是 `unsigned long` ，那么没有必要

typedef unsigned long myflags_t;

不过如果有一个明确的原因，比如它在某种情况下可能会是一个 unsigned int 而在其他情况下可能为 unsigned long ，那么就不要犹豫，请务必使用 typedef。

(c) 当你使用 sparse 按字面的创建一个 **新**类型来做类型检查的时候。

(d) 和标准 C99 类型相同的类型，在某些例外的情况下。

虽然让眼睛和脑筋来适应新的标准类型比如 uint32_t 不需要花很多时间，可是有些人仍然拒绝使用它们。

因此，Linux 特有的等同于标准类型的 u8/u16/u32/u64 类型和它们的有符号类型是被允许的——尽管在你自己的新代码中，它们不是强制要求要使用的。

当编辑已经使用了某个类型集的已有代码时，你应该遵循那些代码中已经做出的选择。

(e) 可以在用户空间安全使用的类型。

在某些用户空间可见的结构体里，我们不能要求 C99 类型而且不能用上面提到的 u32 类型。因此，我们在与用户空间共享的所有结构体中使用 __u32 和类似的类型。

可能还有其他的情况，不过基本的规则是 **永远不要**使用 typedef，除非你可以明确的应用上述某个规则中的一个。

总的来说，如果一个指针或者一个结构体里的元素可以合理的被直接访问到，那么它们就不应该是一个 typedef。

6) 函数

函数应该简短而漂亮，并且只完成一件事情。函数应该可以一屏或者两屏显示完 (我们都知道 ISO/ANSI 屏幕大小是 80x24)，只做一件事情，而且把它做好。

一个函数的最大长度是和该函数的复杂度和缩进级数成反比的。所以，如果你有一个理论上很简单的只有一个很长 (但是简单) 的 case 语句的函数，而且你需要在每个 case 里做很多很小的事情，这样的函数尽管很长，但也是可以的。

不过，如果你有一个复杂的函数，而且你怀疑一个天分不是很高的高中一年级学生可能甚至搞不清楚这个函数的目的，你应该严格遵守前面提到的长度限制。使用辅助函数，并为之取个具描述性的名字 (如果你觉得它们的性能很重要的话，可以让编译器内联它们，这样的效果往往会比你写一个复杂函数的效果要好。)

函数的另外一个衡量标准是本地变量的数量。此数量不应超过 5 − 10 个，否则你的函数就有问题了。重新考虑一下你的函数，把它分拆成更小的函数。人的大脑一般可以轻松的同时跟踪 7 个不同的事物，如果再增多的话，就会糊涂了。即便你聪颖过人，你也可能会记不清你 2 个星期前做过的事情。

在源文件里，使用空行隔开不同的函数。如果该函数需要被导出，它的 **EXPORT** 宏应该紧贴在它的结束大括号之下。比如：

```
int system_is_up(void)
{
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

在函数原型中，包含函数名和它们的数据类型。虽然 C 语言里没有这样的要求，在 Linux 里这是提倡的做法，因为这样可以很简单的给读者提供更多的有价值的信息。

7) 集中的函数退出途径

虽然被某些人声称已经过时，但是 goto 语句的等价物还是经常被编译器所使用，具体形式是无条件跳转指令。

当一个函数从多个位置退出，并且需要做一些类似清理的常见操作时，goto 语句就很方便了。如果并不需要清理操作，那么直接 return 即可。

选择一个能够说明 goto 行为或它为何存在的标签名。如果 goto 要释放 buffer, 一个不错的名字可以是 out_free_buffer: 。别去使用像 err1: 和 err2: 这样的 GW_BASIC 名称，因为一旦你添加或删除了 (函数的) 退出路径，你就必须对它们重新编号，这样会难以去检验正确性。

使用 goto 的理由是:

- 无条件语句容易理解和跟踪
- 嵌套程度减小
- 可以避免由于修改时忘记更新个别的退出点而导致错误
- 让编译器省去删除冗余代码的工作;)

```
int fun(int a)
{
        int result = 0;
        char *buffer;

        buffer = kmalloc(SIZE, GFP_KERNEL);
        if (!buffer)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out_free_buffer;
        }
        ...
out_free_buffer:
        kfree(buffer);
        return result;
}
```

一个需要注意的常见错误是 一个 err 错误，就像这样:

```
err:
        kfree(foo->bar);
        kfree(foo);
        return ret;
```

这段代码的错误是，在某些退出路径上 foo 是 NULL。通常情况下，通过把它分离成两个错误标签 err_free_bar: 和 err_free_foo: 来修复这个错误:

```
err_free_bar:
        kfree(foo->bar);
err_free_foo:
        kfree(foo);
        return ret;
```

理想情况下，你应该模拟错误来测试所有退出路径。

8) 注释

注释是好的，不过有过度注释的危险。永远不要在注释里解释你的代码是如何运作的：更好的做法是让别人一看你的代码就可以明白，解释写的很差的代码是浪费时间。

一般的，你想要你的注释告诉别人你的代码做了什么，而不是怎么做的。也请你不要把注释放在一个函数体内部：如果函数复杂到你需要独立的注释其中的一部分，你很可能需要回到第六章看一看。你可以做一些小注释来注明或警告某些很聪明 (或者槽糕) 的做法，但不要加太多。你应该做的，是把注释放在函数的头部，告诉人们它做了什么，也可以加上它做这些事情的原因。

当注释内核 API 函数时，请使用 kernel-doc 格式。请看 Documentation/doc-guide/ 和 scripts/kernel-doc 以获得详细信息。

长 (多行) 注释的首选风格是：

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description:  A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */
```

对于在 net/ 和 drivers/net/ 的文件，首选的长 (多行) 注释风格有些不同。

```
/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */
```

注释数据也是很重要的，不管是基本类型还是衍生类型。为了方便实现这一点，每一行应只声明一个数据 (不要使用逗号来一次声明多个数据)。这样你就有空间来为每个数据写一段小注释来解释它们的用途了。

9) 你已经把事情弄糟了

这没什么，我们都是这样。可能你的使用了很长时间 Unix 的朋友已经告诉你 GNU emacs 能自动帮你格式化 C 源代码，而且你也注意到了，确实是这样，不过它所使用的默认值和我们想要的相去甚远 (实际上，甚至比随机打的还要差——无数个猴子在 GNU emacs 里打字永远不会创造出一个好程序) (译注：Infinite Monkey Theorem)

所以你要么放弃 GNU emacs，要么改变它让它使用更合理的设定。要采用后一个方案，你可以把下面这段粘贴到你的.emacs 文件里。

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
         (column (c-langelem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset)))

(dir-locals-set-class-variables
 'linux-kernel
 '((c-mode . (
        (c-basic-offset . 8)
        (c-label-minimum-indentation . 0)
        (c-offsets-alist . (
                (arglist-close         . c-lineup-arglist-tabs-only)
                (arglist-cont-nonempty .
                    (c-lineup-gcc-asm-reg c-lineup-arglist-tabs-
↪only))
                (arglist-intro         . +)
                (brace-list-intro      . +)
                (c                     . c-lineup-C-comments)
                (case-label            . 0)
                (comment-intro         . c-lineup-comment)
                (cpp-define-intro      . +)
                (cpp-macro             . -1000)
                (cpp-macro-cont        . +)
                (defun-block-intro     . +)
                (else-clause           . 0)
                (func-decl-cont        . +)
                (inclass               . +)
                (inher-cont            . c-lineup-multi-inher)
                (knr-argdecl-intro     . 0)
                (label                 . -1000)
                (statement             . 0)
                (statement-block-intro . +)
                (statement-case-intro  . +)
                (statement-cont        . +)
                (substatement          . +)
                ))
        (indent-tabs-mode . t)
```

(continues on next page)

(continued from previous page)

```
        (show-trailing-whitespace . t)
        ))))

(dir-locals-set-directory-class
 (expand-file-name "~/src/linux-trees")
 'linux-kernel)
```

这会让 emacs 在 ~/src/linux-trees 下的 C 源文件获得更好的内核代码风格。

不过就算你尝试让 emacs 正确的格式化代码失败了，也并不意味着你失去了一切：还可以用 indent 。

不过，GNU indent 也有和 GNU emacs 一样有问题的设定，所以你需要给它一些命令选项。不过，这还不算太糟糕，因为就算是 GNU indent 的作者也认同 K&R 的权威性 (GNU 的人并不是坏人，他们只是在这个问题上被严重的误导了)，所以你只要给 indent 指定选项 -kr -i8 (代表 K&R，8 字符缩进)，或使用 scripts/Lindent 这样就可以以最时髦的方式缩进源代码。

indent 有很多选项，特别是重新格式化注释的时候，你可能需要看一下它的手册。不过记住：indent 不能修正坏的编程习惯。

10) Kconfig 配置文件

对于遍布源码树的所有 Kconfig* 配置文件来说，它们缩进方式有所不同。紧挨着 config 定义的行，用一个制表符缩进，然而 help 信息的缩进则额外增加 2 个空格。举个例子:

```
config AUDIT
      bool "Auditing support"
      depends on NET
      help
        Enable auditing infrastructure that can be used with another
        kernel subsystem, such as SELinux (which requires this for
        logging of avc messages output).  Does not do system-call
        auditing without CONFIG_AUDITSYSCALL.
```

而那些危险的功能 (比如某些文件系统的写支持) 应该在它们的提示字符串里显著的声明这一点:

```
config ADFS_FS_RW
      bool "ADFS write support (DANGEROUS)"
      depends on ADFS_FS
      ...
```

要查看配置文件的完整文档，请看 Documentation/kbuild/kconfig-language.rst。

11) 数据结构

如果一个数据结构，在创建和销毁它的单线执行环境之外可见，那么它必须要有一个引用计数器。内核里没有垃圾收集 (并且内核之外的垃圾收集慢且效率低下)，这意味着你绝对需要记录你对这种数据结构的使用情况。

引用计数意味着你能够避免上锁，并且允许多个用户并行访问这个数据结构——而不需要担心这个数据结构仅仅因为暂时不被使用就消失了，那些用户可能不过是沉睡了一阵或者做了一些其他事情而已。

注意上锁 **不能**取代引用计数。上锁是为了保持数据结构的一致性，而引用计数是一个内存管理技巧。通常二者都需要，不要把两个搞混了。

很多数据结构实际上有 2 级引用计数，它们通常有不同 类的用户。子类计数器统计子类用户的数量，每当子类计数器减至零时，全局计数器减一。

这种 多级引用计数的例子可以在内存管理 (`struct mm_struct`: mm_users 和 mm_count)，和文件系统 (`struct super_block`: s_count 和 s_active) 中找到。

记住：如果另一个执行线索可以找到你的数据结构，但这个数据结构没有引用计数器，这里几乎肯定是一个 bug。

12) 宏，枚举和 RTL

用于定义常量的宏的名字及枚举里的标签需要大写。

```
#define CONSTANT 0x12345
```

在定义几个相关的常量时，最好用枚举。

宏的名字请用大写字母，不过形如函数的宏的名字可以用小写字母。

一般的，如果能写成内联函数就不要写成像函数的宏。

含有多个语句的宏应该被包含在一个 do-while 代码块里：

```
#define macrofun(a, b, c)                       \
        do {                                    \
                if (a == 5)                     \
                        do_this(b, c);          \
        } while (0)
```

使用宏的时候应避免的事情：

1) 影响控制流程的宏：

```
#define FOO(x)                                  \
        do {                                    \
                if (blah(x) < 0)                \
                        return -EBUGGERED;      \
        } while (0)
```

非常不好。它看起来像一个函数，不过却能导致 调用它的函数退出；不要打乱读者大脑里的语法分析器。

2) 依赖于一个固定名字的本地变量的宏：

```
#define FOO(val) bar(index, val)
```

可能看起来像是个不错的东西，不过它非常容易把读代码的人搞糊涂，而且容易导致看起来不相关的改动带来错误。

3) 作为左值的带参数的宏：FOO(x) = y；如果有人把 FOO 变成一个内联函数的话，这种用法就会出错了。

4) 忘记了优先级：使用表达式定义常量的宏必须将表达式置于一对小括号之内。带参数的宏也要注意此类问题。

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

5) 在宏里定义类似函数的本地变量时命名冲突：

```
#define FOO(x)                          \
({                                      \
        typeof(x) ret;                  \
        ret = calc_ret(x);              \
        (ret);                          \
})
```

ret 是本地变量的通用名字 - __foo_ret 更不容易与一个已存在的变量冲突。

cpp 手册对宏的讲解很详细。gcc internals 手册也详细讲解了 RTL，内核里的汇编语言经常用到它。

13) 打印内核消息

内核开发者应该是受过良好教育的。请一定注意内核信息的拼写，以给人以好的印象。不要用不规范的单词比如 dont，而要用 do not 或者 don't 。保证这些信息简单明了，无歧义。

内核信息不必以英文句号结束。

在小括号里打印数字 (%d) 没有任何价值，应该避免这样做。

<linux/device.h> 里有一些驱动模型诊断宏，你应该使用它们，以确保信息对应于正确的设备和驱动，并且被标记了正确的消息级别。这些宏有：dev_err(), dev_warn(), dev_info() 等等。对于那些不和某个特定设备相关连的信息，<linux/printk.h> 定义了 pr_notice(), pr_info(), pr_warn(), pr_err() 和其他。

写出好的调试信息可以是一个很大的挑战；一旦你写出后，这些信息在远程除错时能提供极大的帮助。然而打印调试信息的处理方式同打印非调试信息不同。其他 pr_XXX() 函数能无条件地打印，pr_debug() 却不；默认情况下它不会被编译，除非定义了 DEBUG 或设定了 CONFIG_DYNAMIC_DEBUG。实际这同样是为了 dev_dbg()，一个相关约定是在一个已经开启了 DEBUG 时，使用 VERBOSE_DEBUG 来添加 dev_vdbg()。

许多子系统拥有 Kconfig 调试选项来开启 -DDEBUG 在对应的 Makefile 里面；在其他情况下，特殊文件使用 #define DEBUG。当一条调试信息需要被无条件打印时，例如，如果已经包含一个调试相关的 #ifdef 条件，printk(KERN_DEBUG …) 就可被使用。

14) 分配内存

内核提供了下面的一般用途的内存分配函数：kmalloc(), kzalloc(), kmalloc_array(), kcalloc(), vmalloc() 和 vzalloc()。请参考 API 文档以获取有关它们的详细信息。

传递结构体大小的首选形式是这样的：

```
p = kmalloc(sizeof(*p), ...);
```

另外一种传递方式中，sizeof 的操作数是结构体的名字，这样会降低可读性，并且可能会引入 bug。有可能指针变量类型被改变时，而对应的传递给内存分配函数的 sizeof 的结果不变。

强制转换一个 void 指针返回值是多余的。C 语言本身保证了从 void 指针到其他任何指针类型的转换是没有问题的。

分配一个数组的首选形式是这样的：

```
p = kmalloc_array(n, sizeof(...), ...);
```

分配一个零长数组的首选形式是这样的：

```
p = kcalloc(n, sizeof(...), ...);
```

两种形式检查分配大小 n * sizeof(…) 的溢出，如果溢出返回 NULL。

15) 内联弊病

有一个常见的误解是 内联是 gcc 提供的可以让代码运行更快的一个选项。虽然使用内联函数有时候是恰当的 (比如作为一种替代宏的方式，请看第十二章)，不过很多情况下不是这样。inline 的过度使用会使内核变大，从而使整个系统运行速度变慢。因为体积大内核会占用更多的指令高速缓存，而且会导致 pagecache 的可用内存减少。想象一下，一次 pagecache 未命中就会导致一次磁盘寻址，将耗时 5 毫秒。5 毫秒的时间内 CPU 能执行很多很多指令。

一个基本的原则是如果一个函数有 3 行以上，就不要把它变成内联函数。这个原则的一个例外是，如果你知道某个参数是一个编译时常量，而且因为这个常量你确定编译器在编译时能优化掉你的函数的大部分代码，那仍然可以给它加上 inline 关键字。kmalloc() 内联函数就是一个很好的例子。

人们经常主张给 static 的而且只用了一次的函数加上 inline，如此不会有任何损失，因为没有什么好权衡的。虽然从技术上说这是正确的，但是实际上这种情况下即使不加 inline gcc 也可以自动使其内联。而且其他用户可能会要求移除 inline，由此而来的争论会抵消 inline 自身的潜在价值，得不偿失。

16) 函数返回值及命名

函数可以返回多种不同类型的值，最常见的一种是表明函数执行成功或者失败的值。这样的一个值可以表示为一个错误代码整数 (-Exxx = 失败，0 = 成功) 或者一个 成功布尔值 (0 = 失败，非 0 = 成功)。

混合使用这两种表达方式是难于发现的 bug 的来源。如果 C 语言本身严格区分整形和布尔型变量，那么编译器就能够帮我们发现这些错误…不过 C 语言不区分。为了避免产生这种 bug，请遵循下面的惯例：

```
如果函数的名字是一个动作或者强制性的命令，那么这个函数应该返回错误代
码整数。如果是一个判断，那么函数应该返回一个 " 成功" 布尔值。
```

比如，`add work` 是一个命令，所以 add_work() 在成功时返回 0，在失败时返回 -EBUSY。类似的，因为 `PCI device present` 是一个判断，所以 pci_dev_present() 在成功找到一个匹配的设备时应该返回 1，如果找不到时应该返回 0。

所有 EXPORTed 函数都必须遵守这个惯例，所有的公共函数也都应该如此。私有 (static) 函数不需要如此，但是我们也推荐这样做。

返回值是实际计算结果而不是计算是否成功的标志的函数不受此惯例的限制。一般的，他们通过返回一些正常值范围之外的结果来表示出错。典型的例子是返回指针的函数，他们使用 NULL 或者 ERR_PTR 机制来报告错误。

17) 不要重新发明内核宏

头文件 include/linux/kernel.h 包含了一些宏，你应该使用它们，而不要自己写一些它们的变种。比如，如果你需要计算一个数组的长度，使用这个宏

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

类似的，如果你要计算某结构体成员的大小，使用

```
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
```

还有可以做严格的类型检查的 min() 和 max() 宏，如果你需要可以使用它们。你可以自己看看那个头文件里还定义了什么你可以拿来用的东西，如果有定义的话，你就不应在你的代码里自己重新定义。

18) 编辑器模式行和其他需要罗嗦的事情

有一些编辑器可以解释嵌入在源文件里的由一些特殊标记标明的配置信息。比如，emacs 能够解释被标记成这样的行：

```
-*- mode: c -*-
```

或者这样的：

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim 能够解释这样的标记：

```
/* vim:set sw=8 noet */
```

不要在源代码中包含任何这样的内容。每个人都有他自己的编辑器配置，你的源文件不应该覆盖别人的配置。这包括有关缩进和模式配置的标记。人们可以使用他们自己定制的模式，或者使用其他可以产生正确的缩进的巧妙方法。

19) 内联汇编

在特定架构的代码中，你可能需要内联汇编与 CPU 和平台相关功能连接。需要这么做时就不要犹豫。然而，当 C 可以完成工作时，不要平白无故地使用内联汇编。在可能的情况下，你可以并且应该用 C 和硬件沟通。

请考虑去写捆绑通用位元 (wrap common bits) 的内联汇编的简单辅助函数，别去重复地写下只有细微差异内联汇编。记住内联汇编可以使用 C 参数。

大型，有一定复杂度的汇编函数应该放在.S 文件内，用相应的 C 原型定义在 C 头文件中。汇编函数的 C 原型应该使用 `asmlinkage` 。

你可能需要把汇编语句标记为 volatile，用来阻止 GCC 在没发现任何副作用后就把它移除了。你不必总是这样做，尽管，这不必要的举动会限制优化。

在写一个包含多条指令的单个内联汇编语句时，把每条指令用引号分割而且各占一行，除了最后一条指令外，在每个指令结尾加上 nt，让汇编输出时可以正确地缩进下一条指令：

```
asm ("magic %reg1, #42\n\t"
     "more_magic %reg2, %reg3"
     : /* outputs */ : /* inputs */ : /* clobbers */);
```

20) 条件编译

只要可能，就不要在.c 文件里面使用预处理条件 (#if, #ifdef)；这样做让代码更难阅读并且更难去跟踪逻辑。替代方案是，在头文件中用预处理条件提供给那些.c 文件使用，再给 #else 提供一个空桩 (no-op stub) 版本，然后在.c 文件内无条件地调用那些 (定义在头文件内的) 函数。这样做，编译器会避免为桩函数 (stub) 的调用生成任何代码，产生的结果是相同的，但逻辑将更加清晰。

最好倾向于编译整个函数，而不是函数的一部分或表达式的一部分。与其放一个 ifdef 在表达式内，不如分解出部分或全部表达式，放进一个单独的辅助函数，并应用预处理条件到这个辅助函数内。

如果你有一个在特定配置中，可能变成未使用的函数或变量，编译器会警告它定义了但未使用，把它标记为 __maybe_unused 而不是将它包含在一个预处理条件中。(然而，如果一个函数或变量总是未使用，就直接删除它。)

在代码中，尽可能地使用 IS_ENABLED 宏来转化某个 Kconfig 标记为 C 的布尔表达式，并在一般的 C 条件中使用它：

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
        ...
}
```

编译器会做常量折叠，然后就像使用 #ifdef 那样去包含或排除代码块，所以这不会带来任何运行时开销。然而，这种方法依旧允许 C 编译器查看块内的代码，并检查它的正确性 (语法，类型，符号引用，等等)。因此，如果条件不满足，代码块内的引用符号就不存在时，你还是必须去用 #ifdef。

在任何有意义的 #if 或 #ifdef 块的末尾 (超过几行的)，在 #endif 同一行的后面写下注解，注释这个条件表达式。例如：

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

附录 I) 参考

The C Programming Language, 第二版作者：Brian W. Kernighan 和 Denni M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (软皮), 0-13-110370-9 (硬皮).

The Practice of Programming 作者：Brian W. Kernighan 和 Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

GNU 手册 - 遵循 K&R 标准和此文本 - cpp, gcc, gcc internals and indent, 都可以从 https://www.gnu.org/manual/ 找到

WG14 是 C 语言的国际标准化工作组，URL: http://www.open-std.org/JTC1/SC22/WG14/

Kernel process/coding-style.rst，作者 greg@kroah.com 发表于 OLS 2002：http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/development-process.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 内核开发过程指南

内容:

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/1.Intro.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

介绍

执行摘要

本节的其余部分涵盖了内核开发过程的范围，以及开发人员及其雇主在这方面可能遇到的各种挫折。内核代码应该合并到正式的（“主线”）内核中有很多原因，包括对用户的自动可用性、多种形式的社区支持以及影响内核开发方向的能力。提供给 Linux 内核的代码必须在与 GPL 兼容的许可证下可用。

开发流程如何工作 介绍了开发过程、内核发布周期和合并窗口的机制。涵盖了补丁开发、审查和合并周期中的各个阶段。有一些关于工具和邮件列表的讨论。鼓励希望开始内核开发的开发人员作为初始练习跟踪并修复 bug。

早期规划 包括早期项目规划，重点是尽快让开发社区参与

使代码正确 是关于编码过程的；讨论了其他开发人员遇到的几个陷阱。对补丁的一些要求已经涵盖，并且介绍了一些工具，这些工具有助于确保内核补丁是正确的。

发布补丁 讨论发布补丁以供评审的过程。为了让开发社区认真对待，补丁必须正确格式化和描述，并且必须发送到正确的地方。遵循本节中的建议有助于确保为您的工作提供最好的接纳。

跟进 介绍了发布补丁之后发生的事情；该工作在这一点上还远远没有完成。与审阅者一起工作是开发过程中的一个重要部分；本节提供了一些关于如何在这个重要阶段避免问题的提示。当补丁被合并到主线中时，开发人员要注意不要假定任务已经完成。

高级主题 介绍了两个“高级”主题：使用 Git 管理补丁和查看其他人发布的补丁。

更多信息 总结了有关内核开发的更多信息，附带有带有指向资源的链接.

这个文件是关于什么的

Linux 内核有超过 800 万行代码，每个版本的贡献者超过 1000 人，是现存最大、最活跃的免费软件项目之一。从 1991 年开始，这个内核已经发展成为一个最好的操作系统组件，运行在袖珍数字音乐播放器、台式 PC、现存最大的超级计算机以及所有类型的系统上。它是一种适用于几乎任何情况的健壮、高效和可扩展的解决方案。

随着 Linux 的发展，希望参与其开发的开发人员（和公司）的数量也在增加。硬件供应商希望确保 Linux 能够很好地支持他们的产品，使这些产品对 Linux 用户具有吸引力。嵌入式系统供应商使用 Linux 作为集成产品的组件，希望 Linux 能够尽可能地胜任手头的任务。分销商和其他基于 Linux 的软件供应商对 Linux 内核的功能、性能和可靠性有着明确的兴趣。最终用户也常常希望修改 Linux，使之更好地满足他们的需求。

Linux 最引人注目的特性之一是这些开发人员可以访问它；任何具备必要技能的人都可以改进 Linux 并影响其开发方向。专有产品不能提供这种开放性，这是自由软件的一个特点。但是，如果有什么不同的话，内核比大多数其他自由软件项目更开放。一个典型的三个月内核开发周期可以涉及 1000 多个开发人员，他们为 100 多个不同的公司（或者根本没有公司）工作。

与内核开发社区合作并不是特别困难。但是，尽管如此，许多潜在的贡献者在尝试做内核工作时遇到了困难。内核社区已经发展了自己独特的操作方式，使其能够在每天都要更改数千行代码的环境中顺利运行（并生成高质量的产品）。因此，Linux 内核开发过程与专有的开发方法有很大的不同也就不足为奇了。

对于新开发人员来说，内核的开发过程可能会让人感到奇怪和恐惧，但这个背后有充分的理由和坚实的经验。一个不了解内核社区的方式的开发人员（或者更糟的是，他们试图抛弃或规避内核社区的方式）会有一个令人沮丧的体验。开发社区, 在帮助那些试图学习的人的同时，没有时间帮助那些不愿意倾听或不关心开发过程的人。

希望阅读本文的人能够避免这种令人沮丧的经历。这里有很多材料，但阅读时所做的努力会在短时间内得到回报。开发社区总是需要能让内核变更好的开发人员；下面的文本应该帮助您或为您工作的人员加入我们的社区。

致谢

本文件由 Jonathan Corbet 撰写，corbet@lwn.net。以下人员的建议使之更为完善：Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata, 和 Jochen Voß.

这项工作得到了 Linux 基金会的支持，特别感谢 Amanda McPherson，他看到了这项工作的价值并把它变成现实。

代码进入主线的重要性

有些公司和开发人员偶尔会想，为什么他们要费心学习如何与内核社区合作，并将代码放入主线内核（“主线”是由 Linus Torvalds 维护的内核，Linux 发行商将其用作基础）。在短期内，贡献代码看起来像是一种可以避免的开销；仅仅将代码分开并直接支持用户似乎更容易。事实上，保持代码独立（“树外”）是在经济上是错误的。

作为说明树外代码成本的一种方法，下面是内核开发过程的一些相关方面；本文稍后将更详细地讨论其中的大部分内容。考虑：

- 所有 Linux 用户都可以使用合并到主线内核中的代码。它将自动出现在所有启用它的发行版上。不需要驱动程序磁盘、下载，也不需要为多个发行版的多个版本提供支持；对于开发人员和用户来说，这一切都是可行的。并入主线解决了大量的分布和支持问题
- 当内核开发人员努力维护一个稳定的用户空间接口时，内部内核 API 处于不断变化之中. 缺乏一个稳定的内部接口是一个深思熟虑的设计决策；它允许在任何时候进行基本的改进，并产生更高质量的代码。但该策略的一个结果是，如果要使用新的内核，任何树外代码都需要持续的维护。维护树外代码需要大量的工作才能使代码保持工作状态。

  相反，位于主线中的代码不需要这样做，因为一个简单的规则要求进行 API 更改的任何开发人员也必须修复由于该更改而破坏的任何代码。因此，合并到主线中的代码大大降低了维护成本。
- 除此之外，内核中的代码通常会被其他开发人员改进。令人惊讶的结果可能来自授权您的用户社区和客户改进您的产品。
- 内核代码在合并到主线之前和之后都要经过审查。不管原始开发人员的技能有多强，这个审查过程总是能找到改进代码的方法。审查经常发现严重的错误和安全问题。这对于在封闭环境中开发的代码尤其如此；这种代码从外部开发人员的审查中获益匪浅。树外代码是低质量代码。
- 参与开发过程是您影响内核开发方向的方式。旁观者的抱怨会被听到，但是活跃的开发人员有更强的声音——并且能够实现使内核更好地满足其需求的更改。

- 当单独维护代码时，总是存在第三方为类似功能提供不同实现的可能性。如果发生这种情况，合并代码将变得更加困难——甚至到了不可能的地步。然后，您将面临以下令人不快的选择：（1）无限期地维护树外的非标准特性，或（2）放弃代码并将用户迁移到树内版本。
- 代码的贡献是使整个过程工作的根本。通过贡献代码，您可以向内核添加新功能，并提供其他内核开发人员使用的功能和示例。如果您已经为 Linux 开发了代码（或者正在考虑这样做），那么您显然对这个平台的持续成功感兴趣；贡献代码是确保成功的最好方法之一。

上述所有理由都适用于任何树外内核代码，包括以专有的、仅二进制形式分发的代码。然而，在考虑任何类型的纯二进制内核代码分布之前，还需要考虑其他因素。这些包括：

- 围绕专有内核模块分发的法律问题充其量是模糊的；相当多的内核版权所有者认为，大多数仅限二进制的模块是内核的派生产品，因此，它们的分发违反了 GNU 通用公共许可证（下面将详细介绍）。您的作者不是律师，本文档中的任何内容都不可能被视为法律建议。封闭源代码模块的真实法律地位只能由法院决定。但不管怎样，困扰这些模块的不确定性仍然存在。
- 二进制模块大大增加了调试内核问题的难度，以至于大多数内核开发人员甚至都不会尝试。因此，只分发二进制模块将使您的用户更难从社区获得支持。
- 对于只支持二进制的模块的发行者来说，支持也更加困难，他们必须为他们希望支持的每个发行版和每个内核版本提供一个版本的模块。为了提供相当全面的覆盖范围，可能需要一个模块的几十个构建，并且每次升级内核时，您的用户都必须单独升级您的模块。
- 上面提到的关于代码评审的所有问题都更加存在于封闭源代码。由于该代码根本不可用，因此社区无法对其进行审查，毫无疑问，它将存在严重问题。

尤其是嵌入式系统的制造商，可能会倾向于忽视本节中所说的大部分内容，因为他们相信自己正在商用一种使用冻结内核版本的独立产品，在发布后不需要再进行开发。这个论点忽略了广泛的代码审查的价值以及允许用户向产品添加功能的价值。但这些产品也有有限的商业寿命，之后必须发布新版本的产品。在这一点上，代码在主线上并得到良好维护的供应商将能够更好地占位，以使新产品快速上市。

许可

代码是根据一些许可证提供给 Linux 内核的，但是所有代码都必须与 GNU 通用公共许可证（GPLV2）的版本 2 兼容，该版本是覆盖整个内核分发的许可证。在实践中，这意味着所有代码贡献都由 GPLv2（可选地，语言允许在更高版本的 GPL 下分发）或 3 子句 BSD 许可（New BSD License, 译者注）覆盖。任何不包含在兼容许可证中的贡献都不会被接受到内核中。

贡献给内核的代码不需要（或请求）版权分配。合并到主线内核中的所有代码都保留其原始所有权；因此，内核现在拥有数千个所有者。

这种所有权结构的一个暗示是，任何改变内核许可的尝试都注定会失败。很少有实际的场景可以获得所有版权所有者的同意（或者从内核中删除他们的代码）。因此，特别是，在可预见的将来，不可能迁移到 GPL 的版本 3。

所有贡献给内核的代码都必须是合法的免费软件。因此，不接受匿名（或匿名）贡献者的代码。所有贡献者都需要在他们的代码上“sign off”，声明代码可以在 GPL 下与内核一起分发。无法提供未被其所有者许可为免费软件的代码，或可能为内核造成版权相关问题的代码（例如，由缺乏适当保护的反向工程工作派生的代码）不能被接受。

有关版权相关问题的问题在 Linux 开发邮件列表中很常见。这样的问题通常会得到不少答案，但要记住，回答这些问题的人不是律师，不能提供法律咨询。如果您有关于 Linux 源代码的法

律问题，那么与了解该领域的律师交流是无法替代的。依靠从技术邮件列表中获得的答案是一件冒险的事情。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/2.Process.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

开发流程如何工作

90 年代早期的 Linux 内核开发是一件相当松散的事情，涉及的用户和开发人员相对较少。由于拥有数以百万计的用户群，并且在一年的时间里有大约 2000 名开发人员参与进来，内核因此必须发展许多流程来保持开发的顺利进行。要成为流程的有效组成部分，需要对流程的工作方式有一个扎实的理解。

总览

内核开发人员使用一个松散的基于时间的发布过程，每两到三个月发布一次新的主要内核版本。最近的发布历史记录如下：

4.11	四月 30, 2017
4.12	七月 2, 2017
4.13	九月 3, 2017
4.14	十一月 12, 2017
4.15	一月 28, 2018
4.16	四月 1, 2018

每 4.x 版本都是一个主要的内核版本，具有新特性、内部 API 更改等等。一个典型的 4.x 版本包含大约 13000 个变更集，变更了几十万行代码。因此，4.x 是 Linux 内核开发的前沿；内核使用滚动开发模型，不断集成重大变化。

对于每个版本的补丁合并，遵循一个相对简单的规则。在每个开发周期的开始，“合并窗口”被打开。当时，被认为足够稳定（并且被开发社区接受）的代码被合并到主线内核中。在这段时间内，新开发周期的大部分变更（以及所有主要变更）将以接近每天 1000 次变更（“补丁”或“变更集”）的速度合并。

（顺便说一句，值得注意的是，合并窗口期间集成的更改并不是凭空产生的；它们是提前收集、测试和分级的。稍后将详细描述该过程的工作方式）。

合并窗口持续大约两周。在这段时间结束时，LinusTorvalds 将声明窗口已关闭，并释放第一个“rc”内核。例如，对于目标为 4.14 的内核，在合并窗口结束时发生的释放将被称为

4.14-rc1。RC1 版本是一个信号，表示合并新特性的时间已经过去，稳定下一个内核的时间已经开始。

在接下来的 6 到 10 周内，只有修复问题的补丁才应该提交给主线。有时会允许更大的更改，但这种情况很少发生；试图在合并窗口外合并新功能的开发人员往往会受到不友好的接待。一般来说，如果您错过了给定特性的合并窗口，最好的做法是等待下一个开发周期。（对于以前不支持的硬件，偶尔会对驱动程序进行例外；如果它们不改变已有代码，则不会导致回归，并且应该可以随时安全地添加）。

随着修复程序进入主线，补丁速度将随着时间的推移而变慢。Linus 大约每周发布一次新的-rc 内核；一个正常的系列将在-rc6 和-rc9 之间，内核被认为足够稳定并最终发布。然后，整个过程又重新开始了。

例如，这里是 4.16 的开发周期进行情况（2018 年的所有日期）：

一月 28	4.15 稳定版发布
二月 11	4.16-rc1, 合并窗口关闭
二月 18	4.16-rc2
二月 25	4.16-rc3
三月 4	4.16-rc4
三月 11	4.16-rc5
三月 18	4.16-rc6
三月 25	4.16-rc7
四月 1	4.16 稳定版发布

开发人员如何决定何时结束开发周期并创建稳定的版本？使用的最重要的指标是以前版本的回归列表。不欢迎出现任何错误，但是那些破坏了以前能工作的系统的错误被认为是特别严重的。因此，导致回归的补丁是不受欢迎的，很可能在稳定期内删除。

开发人员的目标是在稳定发布之前修复所有已知的回归。在现实世界中，这种完美是很难实现的；在这种规模的项目中，变量太多了。有一点，延迟最终版本只会使问题变得更糟；等待下一个合并窗口的一堆更改将变大，从而在下次创建更多的回归错误。因此，大多数 4.x 内核都有一些已知的回归错误，不过，希望没有一个是严重的。

一旦一个稳定的版本发布，它正在进行的维护工作就被移交给“稳定团队”，目前由 Greg Kroah-Hartman 组成。稳定团队将使用 4.x.y 编号方案不定期的发布稳定版本的更新。要加入更新版本，补丁程序必须（1）修复一个重要的 bug，（2）已经合并到下一个开发主线中。内核通常会在超过其初始版本的一个以上的开发周期内接收稳定的更新。例如，4.13 内核的历史如下

九月 3	4.13 稳定版发布
九月 13	4.13.1
九月 20	4.13.2
九月 27	4.13.3
十月 5	4.13.4
十月 12	4.13.5
…	…
十一月 24	4.13.16

4.13.16 是 4.13 版本的最终稳定更新。

有些内核被指定为“长期”内核；它们将得到更长时间的支持。在本文中，当前的长期内核及其维护者是：

3.16	Ben Hutchings	(长期稳定内核)
4.1	Sasha Levin	
4.4	Greg Kroah-Hartman	(长期稳定内核)
4.9	Greg Kroah-Hartman	
4.14	Greg Kroah-Hartman	

为长期支持选择内核纯粹是维护人员有必要和时间来维护该版本的问题。目前还没有为即将发布的任何特定版本提供长期支持的已知计划。

补丁的生命周期

补丁不会直接从开发人员的键盘进入主线内核。相反，有一个稍微复杂（如果有些非正式）的过程，旨在确保对每个补丁进行质量审查，并确保每个补丁实现了一个在主线中需要的更改。对于小的修复，这个过程可能会很快发生，或者，在大的和有争议的变更的情况下，会持续数年。许多开发人员的挫折来自于对这个过程缺乏理解或者试图绕过它。

为了减少这种挫折感，本文将描述补丁如何进入内核。下面是一个介绍，它以某种理想化的方式描述了这个过程。更详细的过程将在后面的章节中介绍。

补丁程序经历的阶段通常是：

- 设计。这就是补丁的真正需求——以及满足这些需求的方式——的所在。设计工作通常是在不涉及社区的情况下完成的，但是如果可能的话，最好是在公开的情况下完成这项工作；这样可以节省很多稍后再重新设计的时间。
- 早期评审。补丁被发布到相关的邮件列表中，列表中的开发人员会回复他们可能有的任何评论。如果一切顺利的话，这个过程应该会发现补丁的任何主要问题。
- 更广泛的评审。当补丁接近准备好纳入主线时，它应该被相关的子系统维护人员接受——尽管这种接受并不能保证补丁会一直延伸到主线。补丁将出现在维护人员的子系统树中，并进入 -next 树（如下所述）。当流程工作时，此步骤将导致对补丁进行更广泛的审查，并发现由于将此补丁与其他人所做的工作集成而导致的任何问题。
- 请注意，大多数维护人员也有日常工作，因此合并补丁可能不是他们的最高优先级。如果您的补丁程序得到了关于所需更改的反馈，那么您应该进行这些更改，或者为不应该进行这些更改的原因辩护。如果您的补丁没有评审意见，但没有被其相应的子系统或驱动程序维护者接受，那么您应该坚持不懈地将补丁更新到当前内核，使其干净地应用，并不断地将其发送以供审查和合并。
- 合并到主线。最终，一个成功的补丁将被合并到由 LinusTorvalds 管理的主线存储库中。此时可能会出现更多的评论和/或问题；开发人员应对这些问题并解决出现的任何问题很重要。
- 稳定版发布。可能受补丁影响的用户数量现在很大，因此可能再次出现新的问题。
- 长期维护。虽然开发人员在合并代码后可能会忘记代码，但这种行为往往会给开发社区留下不良印象。合并代码消除了一些维护负担，因为其他代码将修复由 API 更改引起的问题。但是，如果代码要长期保持有用，原始开发人员应该继续为代码负责。

内核开发人员（或他们的雇主）犯的最大错误之一是试图将流程简化为一个“合并到主线”步骤。这种方法总是会让所有相关人员感到沮丧。

补丁如何进入内核

只有一个人可以将补丁合并到主线内核存储库中：LinusTorvalds。但是，在进入 2.6.38 内核的 9500 多个补丁中，只有 112 个（大约 1.3%）是由 Linus 自己直接选择的。内核项目已经发展到一个规模，没有一个开发人员可以在没有支持的情况下检查和选择每个补丁。内核开发人员处理这种增长的方式是通过使用围绕信任链构建的助理系统。

内核代码库在逻辑上被分解为一组子系统：网络、特定的体系结构支持、内存管理、视频设备等。大多数子系统都有一个指定的维护人员，开发人员对该子系统中的代码负有全部责任。这些子系统维护者（松散地）是他们所管理的内核部分的守护者；他们（通常）会接受一个补丁以包含到主线内核中。

子系统维护人员每个人都使用 git 源代码管理工具管理自己版本的内核源代码树。Git 等工具（以及 Quilt 或 Mercurial 等相关工具）允许维护人员跟踪补丁列表，包括作者信息和其他元数据。在任何给定的时间，维护人员都可以确定他或她的存储库中的哪些补丁在主线中找不到。

当合并窗口打开时，顶级维护人员将要求 Linus 从其存储库中“拉出”他们为合并选择的补丁。如果 Linus 同意，补丁流将流向他的存储库，成为主线内核的一部分。Linus 对拉操作中接收到的特定补丁的关注程度各不相同。很明显，有时他看起来很关注。但是，作为一般规则，Linus 相信子系统维护人员不会向上游发送坏补丁。

子系统维护人员反过来也可以从其他维护人员那里获取补丁。例如，网络树是由首先在专用于网络设备驱动程序、无线网络等的树中积累的补丁构建的。此存储链可以任意长，但很少超过两个或三个链接。由于链中的每个维护者都信任那些管理较低级别树的维护者，所以这个过程称为“信任链”。

显然，在这样的系统中，获取内核补丁取决于找到正确的维护者。直接向 Linus 发送补丁通常不是正确的方法。

Next 树

子系统树链引导补丁流到内核，但它也提出了一个有趣的问题：如果有人想查看为下一个合并窗口准备的所有补丁怎么办？开发人员将感兴趣的是，还有什么其他的更改有待解决，以查看是否存在需要担心的冲突；例如，更改核心内核函数原型的修补程序将与使用该函数旧形式的任何其他修补程序冲突。审查人员和测试人员希望在所有这些变更到达主线内核之前，能够访问它们的集成形式中的变更。您可以从所有有趣的子系统树中提取更改，但这将是一项大型且容易出错的工作。

答案以-next 树的形式出现，在这里子系统树被收集以供测试和审查。Andrew Morton 维护的这些旧树被称为“-mm”（用于内存管理，这就是它的启动名字）。-mm 树集成了一长串子系统树中的补丁；它还包含一些旨在帮助调试的补丁。

除此之外，-mm 还包含大量由 Andrew 直接选择的补丁。这些补丁可能已经发布在邮件列表上，或者它们可能应用于内核中没有指定子系统树的部分。结果，-mm 作为一种最后手段的子系统树运行；如果没有其他明显的路径可以让补丁进入主线，那么它很可能以-mm 结束。累积在-mm 中的各种补丁最终将被转发到适当的子系统树，或者直接发送到 Linus。在典型的开发周期中，大约 5-10% 的补丁通过-mm 进入主线。

当前-mm 补丁可在“mmotm”（-mm of the moment）目录中找到，地址：

> https://www.ozlabs.org/~akpm/mmotm/

然而，使用 mmotm 树可能是一种令人沮丧的体验；它甚至可能无法编译。

下一个周期补丁合并的主要树是 linux-next，由 Stephen Rothwell 维护。根据设计 linux-next 是下一个合并窗口关闭后主线的快照。linux-next 树在 Linux-kernel 和 Linux-next 邮件列表中发布，可从以下位置下载：

https://www.kernel.org/pub/linux/kernel/next/

Linux-next 已经成为内核开发过程中不可或缺的一部分；在一个给定的合并窗口中合并的所有补丁都应该在合并窗口打开之前的一段时间内找到进入 Linux-next 的方法。

Staging 树

内核源代码树包含 drivers/staging/directory，其中有许多驱动程序或文件系统的子目录正在被添加到内核树中。它们然需要更多的工作的时候可以保留在 driver/staging 目录中；一旦完成，就可以将它们移到内核中。这是一种跟踪不符合 Linux 内核编码或质量标准的驱动程序的方法，但人们可能希望使用它们并跟踪开发。

Greg Kroah Hartman 目前负责维护 staging 树。仍需要工作的驱动程序将发送给他，每个驱动程序在 drivers/staging/中都有自己的子目录。除了驱动程序源文件之外，目录中还应该有一个 TODO 文件。todo 文件列出了驱动程序需要接受的挂起的工作，以及驱动程序的任何补丁都应该抄送的人员列表。当前的规则要求，staging 的驱动程序必须至少正确编译。

Staging 是一种相对容易的方法，可以让新的驱动程序进入主线，幸运的是，他们会引起其他开发人员的注意，并迅速改进。然而，进入 staging 并不是故事的结尾；staging 中没有看到常规进展的代码最终将被删除。经销商也倾向于相对不愿意使用 staging 驱动程序。因此，在成为一名合适的主线驱动的路上，staging 充其量只是一个停留。

工具

从上面的文本可以看出，内核开发过程在很大程度上依赖于在不同方向上聚集补丁的能力。如果没有适当强大的工具，整个系统将无法在任何地方正常工作。关于如何使用这些工具的教程远远超出了本文档的范围，但是还是有一些指南的空间。

到目前为止，内核社区使用的主要源代码管理系统是 git。Git 是在自由软件社区中开发的许多分布式版本控制系统之一。它非常适合内核开发，因为它在处理大型存储库和大量补丁时性能非常好。它还有一个难以学习和使用的名声，尽管随着时间的推移它变得更好了。对于内核开发人员来说，对 Git 的某种熟悉几乎是一种要求；即使他们不将它用于自己的工作，他们也需要 Git 来跟上其他开发人员（以及主线）正在做的事情。

现在几乎所有的 Linux 发行版都打包了 Git。主页位于：

https://git-scm.com/

那个页面有指向文档和教程的指针。

在不使用 git 的内核开发人员中，最流行的选择几乎肯定是 mercurial：

http://www.seleric.com/mercurial/

Mercurial 与 Git 共享许多特性，但它提供了一个界面，许多人觉得它更易于使用。

另一个值得了解的工具是 quilt:

https://savannah.nongnu.org/projects/quilt

Quilt 是一个补丁管理系统，而不是源代码管理系统。它不会随着时间的推移跟踪历史；相反，它面向根据不断发展的代码库跟踪一组特定的更改。一些主要的子系统维护人员使用 Quilt 来管理打算向上游移动的补丁。对于某些树的管理（例如-mm），quilt 是最好的工具。

邮件列表

大量的 Linux 内核开发工作是通过邮件列表完成的。如果不在某个地方加入至少一个列表，就很难成为社区中一个功能完备的成员。但是，Linux 邮件列表对开发人员来说也是一个潜在的危险，他们可能会被一堆电子邮件淹没，违反 Linux 列表上使用的约定，或者两者兼而有之。

大多数内核邮件列表都在 vger.kernel.org 上运行；主列表位于：

> http://vger.kernel.org/vger-lists.html

不过，也有一些列表托管在别处；其中一些列表位于 lists.redhat.com。

当然，内核开发的核心邮件列表是 linux-kernel。这个名单是一个令人生畏的地方；每天的信息量可以达到 500 条，噪音很高，谈话技术性很强，参与者并不总是表现出高度的礼貌。但是，没有其他地方可以让内核开发社区作为一个整体聚集在一起；避免使用此列表的开发人员将错过重要信息。

有一些提示可以帮助在 linux-kernel 生存：

- 将邮件转移到单独的文件夹，而不是主邮箱。我们必须能够持续地忽略洪流。
- 不要试图跟踪每一次谈话-其他人都不会。重要的是要对感兴趣的主题（尽管请注意，长时间的对话可以在不更改电子邮件主题行的情况下偏离原始主题）和参与的人进行筛选。
- 不要挑事。如果有人试图激起愤怒的反应，忽略他们。
- 当响应 Linux 内核电子邮件（或其他列表上的电子邮件）时，请为所有相关人员保留 cc:header。如果没有强有力的理由（如明确的请求），则不应删除收件人。一定要确保你要回复的人在 cc:list 中。这个惯例也使你不必在回复邮件时明确要求被抄送。
- 在提出问题之前，搜索列表档案（和整个网络）。有些开发人员可能会对那些显然没有完成家庭作业的人感到不耐烦。
- 避免贴顶帖（把你的答案放在你要回复的引文上面的做法）。这会让你的回答更难理解，印象也很差。
- 询问正确的邮件列表。linux-kernel 可能是通用的讨论点，但它不是从所有子系统中寻找开发人员的最佳场所。

最后一点——找到正确的邮件列表——是开发人员出错的常见地方。在 Linux 内核上提出与网络相关的问题的人几乎肯定会收到一个礼貌的建议，转而在 netdev 列表上提出，因为这是大多数网络开发人员经常出现的列表。还有其他列表可用于 scsi、video4linux、ide、filesystem 等子系统。查找邮件列表的最佳位置是与内核源代码一起打包的 MAINTAINERS 文件。

开始内核开发

关于如何开始内核开发过程的问题很常见——来自个人和公司。同样常见的是错误，这使得关系的开始比必须的更困难。

公司通常希望聘请知名的开发人员来启动开发团队。实际上，这是一种有效的技术。但它也往往是昂贵的，而且没有增长经验丰富的内核开发人员储备。考虑到时间的投入，可以让内部开发人员加快 Linux 内核的开发速度。花这个时间可以让雇主拥有一批了解内核和公司的开发人员，他们也可以帮助培训其他人。从中期来看，这往往是更有利可图的方法。

可以理解的是，单个开发人员往往对起步感到茫然。从一个大型项目开始可能会很吓人；人们往往想先用一些较小的东西来测试水域。这是一些开发人员开始创建修补拼写错误或轻微编码风格问题的补丁的地方。不幸的是，这样的补丁会产生一定程度的噪音，这会分散整个开发社区的注意力，因此，越来越多的人看不起它们。希望向社区介绍自己的新开发人员将无法通过这些方式获得他们想要的那种接待。

Andrew Morton 为有抱负的内核开发人员提供了这个建议

```
所有内核初学者的 No.1 项目肯定是“确保内核在所有的机器上，你可以触摸
到的，始终运行良好" 通常这样做的方法是与其他人一起解决问题（这
可能需要坚持!）但这很好——这是内核开发的一部分
```

(http://lwn.net/articles/283982/)

在没有明显问题需要解决的情况下，建议开发人员查看当前的回归和开放式错误列表. 解决需要修复的问题没有任何缺点；通过解决这些问题，开发人员将获得处理过程的经验，同时与开发社区的其他人建立尊重。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/3.Early-stage.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

早期规划

当考虑一个 Linux 内核开发项目时，很可能会直接跳进去开始编码。然而，与任何重要的项目一样，成功的许多基础最好是在第一行代码编写之前就做好了。在早期计划和沟通中花费一些时间可以节省更多的时间。

详述问题

与任何工程项目一样，成功的内核增强从要解决的问题的清晰描述开始。在某些情况下，这个步骤很容易：例如，当某个特定硬件需要驱动程序时。不过，在其他方面，将实际问题与建议的解决方案混淆是很有诱惑力的，这可能会导致困难。

举个例子：几年前，使用 Linux 音频的开发人员寻求一种方法来运行应用程序，而不因系统延迟过大而导致退出或其他工件。他们得到的解决方案是一个内核模块，旨在连接到 Linux 安全模块（LSM）框架中；这个模块可以配置为允许特定的应用程序访问实时调度程序。这个模块被实现并发送到 Linux 内核邮件列表，在那里它立即遇到问题。

对于音频开发人员来说，这个安全模块足以解决他们当前的问题。但是，对于更广泛的内核社区来说，这被视为对 LSM 框架的滥用（LSM 框架并不打算授予他们原本不具备的进程特权），并对系统稳定性造成风险。他们首选的解决方案包括短期的通过 rlimit 机制进行实时调度访问，以及长期的减少延迟的工作。

然而，音频社区看不到他们实施的特定解决方案的过去；他们不愿意接受替代方案。由此产生的分歧使这些开发人员对整个内核开发过程感到失望；其中一个开发人员返回到音频列表并发布了以下内容：

> 有很多非常好的 Linux 内核开发人员，但他们往往会被一群傲慢的傻瓜所压倒。试图向这些人传达用户需求是浪费时间。他们太“聪明”了，根本听不到少数人的话。

（http://lwn.net/articles/131776/）

实际情况不同；与特定模块相比，内核开发人员更关心系统稳定性、长期维护以及找到正确的问题解决方案。这个故事的寓意是把重点放在问题上——而不是具体的解决方案上——并在投入创建代码之前与开发社区讨论这个问题。

因此，在考虑一个内核开发项目时，我们应该得到一组简短问题的答案：

- 究竟需要解决的问题是什么？
- 受此问题影响的用户是谁？解决方案应该解决哪些用例？
- 内核现在为何没能解决这个问题？

只有这样，才能开始考虑可能的解决方案。

早期讨论

在计划内核开发项目时，在开始实施之前与社区进行讨论是很有意义的。早期沟通可以通过多种方式节省时间和麻烦：

- 很可能问题是由内核以您不理解的方式解决的。Linux 内核很大，具有许多不明显的特性和功能。并不是所有的内核功能都像人们所希望的那样有文档记录，而且很容易遗漏一些东西。你的作者发出了一个完整的驱动程序，复制了一个新作者不知道的现有驱动程序。重新设计现有轮子的代码不仅浪费，而且不会被接受到主线内核中。
- 建议的解决方案中可能有一些元素不适用于主线合并。在编写代码之前，最好先了解这样的问题。
- 其他开发人员完全有可能考虑过这个问题；他们可能有更好的解决方案的想法，并且可能愿意帮助创建这个解决方案。

在内核开发社区的多年经验给了我们一个明确的教训：闭门设计和开发的内核代码总是有一些问题，这些问题只有在代码发布到社区中时才会被发现。有时这些问题很严重，需要数月或数年的努力才能使代码达到内核社区的标准。一些例子包括：

- 设计并实现了单处理器系统的 DeviceScape 网络栈。只有使其适合于多处理器系统，才能将其合并到主线中。在代码中改装锁等等是一项困难的任务；因此，这段代码（现在称为 mac80211）的合并被推迟了一年多。
- Reiser4 文件系统包含许多功能，核心内核开发人员认为这些功能应该在虚拟文件系统层中实现。它还包括一些特性，这些特性在不将系统暴露于用户引起的死锁的情况下是不容易实现的。这些问题的最新发现——以及对其中一些问题的拒绝——已经导致 Reiser4 远离了主线内核。
- Apparmor 安全模块以被认为不安全和不可靠的方式使用内部虚拟文件系统数据结构。这种担心（包括其他）使 Apparmor 多年不在主线上。

在每一种情况下，通过与内核开发人员的早期讨论，可以避免大量的痛苦和额外的工作。

找谁交流

当开发人员决定公开他们的计划时，下一个问题是：我们从哪里开始？答案是找到正确的邮件列表和正确的维护者。对于邮件列表，最好的方法是在维护者 (MAINTAINERS) 文件中查找要发布的相关位置。如果有一个合适的子系统列表，那么发布它通常比在 Linux 内核上发布更可取；您更有可能接触到在相关子系统中具有专业知识的开发人员，并且环境可能具支持性。

找到维护人员可能会有点困难。同样，维护者文件是开始的地方。但是，该文件往往不总是最新的，并且并非所有子系统都在那里表示。实际上，维护者文件中列出的人员可能不是当前实际担任该角色的人员。因此，当对联系谁有疑问时，一个有用的技巧是使用 git（尤其是“git-log”）查看感兴趣的子系统中当前活动的用户。看看谁在写补丁，如果有人的话，谁会在这些补丁上加上用线签名的。这些人将是帮助新开发项目的最佳人选。

找到合适的维护者的任务有时是非常具有挑战性的，以至于内核开发人员添加了一个脚本来简化过程：

```
.../scripts/get_maintainer.pl
```

当给定“-f”选项时，此脚本将返回给定文件或目录的当前维护者。如果在命令行上传递了一个补丁，它将列出可能接收补丁副本的维护人员。有许多选项可以调节 get_maintainer.pl 搜索维护者的难易程度；请小心使用更具攻击性的选项，因为最终可能会包括对您正在修改的代码没有真正兴趣的开发人员。

如果所有其他方法都失败了，那么与 Andrew Morton 交谈可以成为一种有效的方法来跟踪特定代码段的维护人员。

何时邮寄？

如果可能的话，在早期阶段发布你的计划只会有帮助。描述正在解决的问题以及已经制定的关于如何实施的任何计划。您可以提供的任何信息都可以帮助开发社区为项目提供有用的输入。

在这个阶段可能发生的一件令人沮丧的事情不是敌对的反应，而是很少或根本没有反应。可悲的事实是：(1) 内核开发人员往往很忙；(2) 不缺少有宏伟计划和很少代码（甚至代码前景）支持他们的人；(3) 没有人有义务审查或评论别人发表的想法。除此之外，高级设计常常隐藏一些问题，这些问题只有在有人真正尝试实现这些设计时才会被发现；因此，内核开发人员宁愿看到代码。

如果发表评论的请求在评论的方式上没有什么效果，不要假设这意味着对项目没有兴趣。不幸的是，你也不能假设你的想法没有问题。在这种情况下，最好的做法是继续进行，把你的进展随时通知社区。

获得官方认可

如果您的工作是在公司环境中完成的，就像大多数 Linux 内核工作一样，显然，在您将公司的计划或代码发布到公共邮件列表之前，必须获得适当授权的经理的许可。发布不确定是否兼容 GPL 的代码可能是有特别问题的；公司的管理层和法律人员越早能够就发布内核开发项目达成一致，对参与的每个人都越好。

一些读者可能会认为他们的核心工作是为了支持还没有正式承认存在的产品。将雇主的计划公布在公共邮件列表上可能不是一个可行的选择。在这种情况下，有必要考虑保密是否真的是必要的；通常不需要把开发计划关在门内。

也就是说，有些情况下，一家公司在开发过程的早期就不能合法地披露其计划。拥有经验丰富的内核开发人员的公司可以选择以开环的方式进行，前提是他们以后能够避免严重的集成问题。对于没有这种内部专业知识的公司，最好的选择往往是聘请外部开发商根据保密协议审查计划。Linux 基金会运行了一个 NDA 程序，旨在帮助解决这种情况；

http://www.linuxfoundation.org/en/NDA_program

这种审查通常足以避免以后出现严重问题，而无需公开披露项目。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/4.Coding.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

使代码正确

虽然对于一个坚实的、面向社区的设计过程有很多话要说，但是任何内核开发项目的证明都在生成的代码中。它是将由其他开发人员检查并合并（或不合并）到主线树中的代码。所以这段代码的质量决定了项目的最终成功。

本节将检查编码过程。我们将从内核开发人员出错的几种方式开始。然后重点将转移到正确的事情和可以帮助这个任务的工具上。

陷阱

编码风格

内核长期以来都有一种标准的编码风格，如Linux 内核代码风格 中所述。在大部分时间里，该文件中描述的政策被认为至多是建议性的。因此，内核中存在大量不符合编码风格准则的代码。代码的存在会给内核开发人员带来两个独立的危害。

首先，要相信内核编码标准并不重要，也不强制执行。事实上，如果没有按照标准对代码进行编码，那么向内核添加新代码是非常困难的；许多开发人员甚至会在审查代码之前要求对代码进行重新格式化。一个与内核一样大的代码库需要一些统一的代码，以使开发人员能够快速理解其中的任何部分。所以已经没有空间来存放奇怪的格式化代码了。

偶尔，内核的编码风格会与雇主的强制风格发生冲突。在这种情况下，内核的风格必须在代码合并之前获胜。将代码放入内核意味着以多种方式放弃一定程度的控制权——包括控制代码的格式化方式。

另一个陷阱是假定已经在内核中的代码迫切需要编码样式的修复。开发人员可能会开始生成重新格式化补丁，作为熟悉过程的一种方式，或者作为将其名称写入内核变更日志的一种方式，或者两者兼而有之。但是纯编码风格的修复被开发社区视为噪音；它们往往受到冷遇。因此，最好避免使用这种类型的补丁。由于其他原因，在处理一段代码的同时修复它的样式是很自然的，但是编码样式的更改不应该仅为了更改而进行。

编码风格的文档也不应该被视为绝对的法律，这是永远不会被违反的。如果有一个很好的理由反对这种样式（例如，如果拆分为适合 80 列限制的行，那么它的可读性就会大大降低），那么就这样做。

请注意，您还可以使用 `clang-format` 工具来帮助您处理这些规则，自动重新格式化部分代码，并查看完整的文件，以发现编码样式错误、拼写错误和可能的改进。它还可以方便地进行排序，包括对齐变量/宏、回流文本和其他类似任务。有关详细信息，请参阅文件 Documentation/process/clang-format.rst

抽象层

计算机科学教授教学生以灵活性和信息隐藏的名义广泛使用抽象层。当然，内核广泛地使用了抽象；任何涉及数百万行代码的项目都不能做到这一点并存活下来。但经验表明，过度或过早的抽象可能和过早的优化一样有害。抽象应用于所需的级别，不要过度。

在一个简单的级别上，考虑一个函数的参数，该参数总是由所有调用方作为零传递。我们可以保留这个论点：以防有人最终需要使用它提供的额外灵活性。不过，到那时，实现这个额外参数的代码很有可能以某种从未被注意到的微妙方式被破坏——因为它从未被使用过。或者，当需要额外的灵活性时，它不会以符合程序员早期期望的方式来这样做。内核开发人员通常会提交补丁来删除未使用的参数；一般来说，首先不应该添加这些参数。

隐藏硬件访问的抽象层——通常允许大量的驱动程序在多个操作系统中使用——尤其不受欢迎。这样的层使代码变得模糊，可能会造成性能损失；它们不属于 Linux 内核。

另一方面，如果您发现自己从另一个内核子系统复制了大量的代码，那么现在是时候问一下，事实上，将这些代码中的一些提取到单独的库中，或者在更高的层次上实现这些功能是否有意义。在整个内核中复制相同的代码没有价值。

#ifdef 和预处理

C 预处理器似乎给一些 C 程序员带来了强大的诱惑，他们认为它是一种有效地将大量灵活性编码到源文件中的方法。但是预处理器不是 C，大量使用它会导致代码对其他人来说更难读取，对编译器来说更难检查正确性。大量的预处理器几乎总是代码需要一些清理工作的标志。

使用 ifdef 的条件编译实际上是一个强大的功能，它在内核中使用。但是很少有人希望看到代码被大量地撒上 ifdef 块。作为一般规则，ifdef 的使用应尽可能限制在头文件中。有条件编译的代码可以限制函数，如果代码不存在，这些函数就会变成空的。然后编译器将悄悄地优化对空函数的调用。结果是代码更加清晰，更容易理解。

C 预处理器宏存在许多危险，包括可能对具有副作用且没有类型安全性的表达式进行多重评估。如果您试图定义宏，请考虑创建一个内联函数。结果相同的代码，但是内联函数更容易读取，不会多次计算其参数，并且允许编译器对参数和返回值执行类型检查。

内联函数

不过，内联函数本身也存在风险。程序员可以倾心于避免函数调用和用内联函数填充源文件所固有的效率。然而，这些功能实际上会降低性能。因为它们的代码在每个调用站点都被复制，所以它们最终会增加编译内核的大小。反过来，这会对处理器的内存缓存造成压力，从而大大降低执行速度。通常，内联函数应该非常小，而且相对较少。毕竟，函数调用的成本并不高；大量内联函数的创建是过早优化的典型例子。

一般来说，内核程序员会忽略缓存效果，这会带来危险。在开始的数据结构课程中，经典的时间/空间权衡通常不适用于当代硬件。空间就是时间，因为一个大的程序比一个更紧凑的程序运行得慢。

最近的编译器在决定一个给定函数是否应该被内联方面扮演着越来越积极的角色。因此，“inline”关键字的自由放置可能不仅仅是过度的，它也可能是无关的。

锁

2006 年 5 月，“deviceescape”网络堆栈在 GPL 下发布，并被纳入主线内核。这是一个受欢迎的消息；对 Linux 中无线网络的支持充其量被认为是不合格的，而 deviceescape 堆栈提供了修复这种情况的承诺。然而，直到 2007 年 6 月（2.6.22），这段代码才真正进入主线。发生了什么?

这段代码显示了许多闭门造车的迹象。但一个特别大的问题是，它并不是设计用于多处理器系统。在合并这个网络堆栈（现在称为 mac80211）之前，需要对其进行一个锁方案的改造。

曾经，Linux 内核代码可以在不考虑多处理器系统所带来的并发性问题的情况下进行开发。然而，现在，这个文件是写在双核笔记本电脑上的。即使在单处理器系统上，为提高响应能力所做的工作也会提高内核内的并发性水平。编写内核代码而不考虑锁的日子已经过去很长了。

可以由多个线程并发访问的任何资源（数据结构、硬件寄存器等）必须由锁保护。新的代码应该记住这一要求；事后改装锁是一项相当困难的任务。内核开发人员应该花时间充分了解可用的锁原语，以便为作业选择正确的工具。显示对并发性缺乏关注的代码进入主线将很困难。

回归

最后一个值得一提的危险是：它可能会引起改变（这可能会带来很大的改进），从而导致现有用户的某些东西中断。这种变化被称为“回归”，回归已经成为主线内核最不受欢迎的。除少数例外情况外，如果回归不能及时修正，会导致回归的变化将被取消。最好首先避免回归。

人们常常争论，如果回归让更多人可以工作，远超过产生问题，那么回归是合理的。如果它破坏的一个系统却为十个系统带来新的功能，为什么不进行更改呢？2007 年 7 月，Linus 对这个问题给出了最佳答案:

::

> 所以我们不会通过引入新问题来修复错误。那样的谎言很疯狂，没有人知道你是否真的有进展。是前进两步，后退一步，还是向前一步，向后两步？

（http://lwn.net/articles/243460/）

一种特别不受欢迎的回归类型是用户空间 ABI 的任何变化。一旦接口被导出到用户空间，就必须无限期地支持它。这一事实使得用户空间接口的创建特别具有挑战性：因为它们不能以不兼容的方式进行更改，所以必须第一次正确地进行更改。因此，用户空间界面总是需要大量的思考、清晰的文档和广泛的审查。

代码检查工具

至少目前，编写无错误代码仍然是我们中很少人能达到的理想状态。不过，我们希望做的是，在代码进入主线内核之前，尽可能多地捕获并修复这些错误。为此，内核开发人员已经组装了一系列令人印象深刻的工具，可以自动捕获各种各样的模糊问题。计算机发现的任何问题都是一个以后不会困扰用户的问题，因此，只要有可能，就应该使用自动化工具。

第一步只是注意编译器产生的警告。当代版本的 GCC 可以检测（并警告）大量潜在错误。通常，这些警告都指向真正的问题。提交以供审阅的代码通常不会产生任何编译器警告。在消除警告时，注意了解真正的原因，并尽量避免“修复”，使警告消失而不解决其原因。

请注意，并非所有编译器警告都默认启用。使用“make EXTRA_CFLAGS=-W”构建内核以获得完整集合。

内核提供了几个配置选项，可以打开调试功能；大多数配置选项位于“kernel hacking”子菜单中。对于任何用于开发或测试目的的内核，都应该启用其中几个选项。特别是，您应该打开：

- 启用 ENABLE_MUST_CHECK and FRAME_WARN 以获得一组额外的警告，以解决使用不推荐使用的接口或忽略函数的重要返回值等问题。这些警告生成的输出可能是冗长的，但您不必担心来自内核其他部分的警告。
- DEBUG_OBJECTS 将添加代码，以跟踪内核创建的各种对象的生存期，并在出现问题时发出警告。如果要添加创建（和导出）自己的复杂对象的子系统，请考虑添加对对象调试基础结构的支持。
- DEBUG_SLAB 可以发现各种内存分配和使用错误；它应该用于大多数开发内核。
- DEBUG_SPINLOCK, DEBUG_ATOMIC_SLEEP and DEBUG_MUTEXES 会发现许多常见的锁定错误.

还有很多其他调试选项，其中一些将在下面讨论。其中一些具有显著的性能影响，不应一直使用。但是，在学习可用选项上花费的一些时间可能会在短期内得到多次回报。

其中一个较重的调试工具是锁定检查器或“lockdep”。该工具将跟踪系统中每个锁（spinlock 或 mutex）的获取和释放、获取锁的相对顺序、当前中断环境等等。然后，它可以确保总是以

相同的顺序获取锁，相同的中断假设适用于所有情况，等等。换句话说，lockdep 可以找到许多场景，在这些场景中，系统很少会死锁。在部署的系统中，这种问题可能会很痛苦（对于开发人员和用户而言）；LockDep 允许提前以自动方式发现问题。具有任何类型的非普通锁定的代码在提交包含前应在启用 lockdep 的情况下运行。

作为一个勤奋的内核程序员，毫无疑问，您将检查任何可能失败的操作（如内存分配）的返回状态。然而，事实上，最终的故障恢复路径可能完全没有经过测试。未测试的代码往往会被破坏；如果所有这些错误处理路径都被执行了几次，那么您可能对代码更有信心。

内核提供了一个可以做到这一点的错误注入框架，特别是在涉及内存分配的情况下。启用故障注入后，内存分配的可配置百分比将失败；这些失败可以限制在特定的代码范围内。在启用了故障注入的情况下运行，程序员可以看到当情况恶化时代码如何响应。有关如何使用此工具的详细信息，请参阅 Documentation/fault-injection/fault-injection.rst。

使用“sparse”静态分析工具可以发现其他类型的错误。对于 sparse，可以警告程序员用户空间和内核空间地址之间的混淆、big endian 和 small endian 数量的混合、在需要一组位标志的地方传递整数值等等。sparse 必须单独安装 (如果您的分发服务器没有将其打包，可以在 https://sparse.wiki.kernel.org/index.php/Main_page) 找到, 然后可以通过在 make 命令中添加“C=1”在代码上运行它。

“Coccinelle”工具 http://coccinelle.lip6.fr/ 能够发现各种潜在的编码问题；它还可以为这些问题提出修复方案。在 scripts/coccinelle 目录下已经打包了相当多的内核“语义补丁”；运行“make coccicheck”将运行这些语义补丁并报告发现的任何问题。有关详细信息，请参阅 Documentation/dev-tools/coccinelle.rst

其他类型的可移植性错误最好通过为其他体系结构编译代码来发现。如果没有 S/390 系统或 Blackfin 开发板，您仍然可以执行编译步骤。可以在以下位置找到一组用于 x86 系统的大型交叉编译器：

> https://www.kernel.org/pub/tools/crosstool/

花一些时间安装和使用这些编译器将有助于避免以后的尴尬。

文档

文档通常比内核开发规则更为例外。即便如此，足够的文档将有助于简化将新代码合并到内核中的过程，使其他开发人员的生活更轻松，并对您的用户有所帮助。在许多情况下，文件的添加已基本上成为强制性的。

任何补丁的第一个文档是其关联的变更日志。日志条目应该描述正在解决的问题、解决方案的形式、处理补丁的人员、对性能的任何相关影响，以及理解补丁可能需要的任何其他内容。确保 changelog 说明了为什么补丁值得应用；大量开发人员未能提供这些信息。

任何添加新用户空间界面的代码（包括新的 sysfs 或/proc 文件）都应该包含该界面的文档，该文档使用户空间开发人员能够知道他们在使用什么。请参阅 Documentation/ABI/README，了解如何格式化此文档以及需要提供哪些信息。

文件 Documentation/admin-guide/kernel-parameters.rst 描述了内核的所有引导时间参数。任何添加新参数的补丁都应该向该文件添加适当的条目。

任何新的配置选项都必须附有帮助文本，帮助文本清楚地解释了这些选项以及用户可能希望何时选择它们。

许多子系统的内部 API 信息通过专门格式化的注释进行记录；这些注释可以通过“kernel-doc”脚本以多种方式提取和格式化。如果您在具有 kerneldoc 注释的子系统中工作，则应该维护它们，并根据需要为外部可用的功能添加它们。即使在没有如此记录的领域中，为将来添加

kerneldoc 注释也没有坏处；实际上，这对于刚开始开发内核的人来说是一个有用的活动。这些注释的格式以及如何创建 kerneldoc 模板的一些信息可以在 Documentation/doc-guide/ 上找到。

任何阅读大量现有内核代码的人都会注意到，注释的缺失往往是最值得注意的。再一次，对新代码的期望比过去更高；合并未注释的代码将更加困难。这就是说，人们几乎不希望用语言注释代码。代码本身应该是可读的，注释解释了更微妙的方面。

某些事情应该总是被注释。使用内存屏障时，应附上一行文字，解释为什么需要设置内存屏障。数据结构的锁定规则通常需要在某个地方解释。一般来说，主要数据结构需要全面的文档。应该指出单独代码位之间不明显的依赖性。任何可能诱使代码看门人进行错误的“清理”的事情都需要一个注释来说明为什么要这样做。等等。

内部 API 更改

内核提供给用户空间的二进制接口不能被破坏，除非在最严重的情况下。相反，内核的内部编程接口是高度流动的，当需要时可以更改。如果你发现自己不得不处理一个内核 API，或者仅仅因为它不满足你的需求而不使用特定的功能，这可能是 API 需要改变的一个标志。作为内核开发人员，您有权进行此类更改。

当然, 可以进行 API 更改，但它们必须是合理的。因此，任何进行内部 API 更改的补丁都应该附带一个关于更改内容和必要原因的描述。这种变化也应该分解成一个单独的补丁，而不是埋在一个更大的补丁中。

另一个要点是，更改内部 API 的开发人员通常要负责修复内核树中被更改破坏的任何代码。对于一个广泛使用的函数，这个职责可以导致成百上千的变化，其中许多变化可能与其他开发人员正在做的工作相冲突。不用说，这可能是一项大工作，所以最好确保理由是可靠的。请注意，coccinelle 工具可以帮助进行广泛的 API 更改。

在进行不兼容的 API 更改时，应尽可能确保编译器捕获未更新的代码。这将帮助您确保找到该接口的树内用处。它还将警告开发人员树外代码存在他们需要响应的更改。支持树外代码不是内核开发人员需要担心的事情，但是我们也不必使树外开发人员的生活有不必要的困难。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/5.Posting.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

发布补丁

迟早，当您的工作准备好提交给社区进行审查，并最终包含到主线内核中时。不出所料，内核开发社区已经发展出一套用于发布补丁的约定和过程；遵循这些约定和过程将使参与其中的每个人的生活更加轻松。本文件将试图合理详细地涵盖这些期望；更多信息也可在以下文件中找到如何让你的改动进入内核, Documentation/process/submitting-drivers.rst 和Linux内核补丁提交清单.

何时邮寄

在补丁完全“准备好”之前，有一个不断的诱惑来避免发布补丁。对于简单的补丁，这不是问题。但是，如果正在完成的工作很复杂，那么在工作完成之前从社区获得反馈就可以获得很多好处。因此，您应该考虑发布正在进行的工作，甚至使 Git 树可用，以便感兴趣的开发人员可以随时赶上您的工作。

当发布还没有准备好包含的代码时，最好在发布本身中这样说。还应提及任何有待完成的主要工作和任何已知问题。很少有人会看到那些被认为是半生不熟的补丁，但是那些人会想到他们可以帮助你把工作推向正确的方向。

创建补丁之前

在考虑将补丁发送到开发社区之前，有许多事情应该做。这些包括：

- 尽可能地测试代码。利用内核的调试工具，确保内核使用所有合理的配置选项组合进行构建，使用跨编译器为不同的体系结构进行构建等。
- 确保您的代码符合内核编码风格指南。
- 您的更改是否具有性能影响？如果是这样，您应该运行基准测试来显示您的变更的影响（或好处）；结果的摘要应该包含在补丁中。
- 确保您有权发布代码。如果这项工作是为雇主完成的，雇主对这项工作具有所有权，并且必须同意根据 GPL 对其进行放行。

一般来说，在发布代码之前进行一些额外的思考，几乎总是能在短时间内得到回报。

补丁准备

准备发布补丁可能是一个惊人的工作量，但再次尝试节省时间在这里通常是不明智的，即使在短期内。

必须针对内核的特定版本准备补丁。作为一般规则，补丁程序应该基于 Linus 的 Git 树中的当前主线。当以主线为基础时，从一个众所周知的发布点开始——一个稳定的或 RC 的发布——而不是在一个主线分支任意点。

但是，可能需要针对-mm、linux-next 或子系统树生成版本，以便于更广泛的测试和审查。根据补丁的区域以及其他地方的情况，针对这些其他树建立补丁可能需要大量的工作来解决冲突和处理 API 更改。

只有最简单的更改才应格式化为单个补丁；其他所有更改都应作为一系列逻辑更改进行。分割补丁是一门艺术；一些开发人员花了很长时间来弄清楚如何按照社区期望的方式来做。然而，有一些经验法则可以大大帮助：

- 您发布的补丁程序系列几乎肯定不会是工作系统中的一系列更改。相反，您所做的更改需要在最终形式中加以考虑，然后以有意义的方式进行拆分。开发人员对离散的、自包含的更改感兴趣，而不是您获取这些更改的路径。
- 每个逻辑上独立的变更都应该格式化为单独的补丁。这些更改可以是小的（“向此结构添加字段”）或大的（例如，添加一个重要的新驱动程序），但它们在概念上应该是小的，并且可以接受一行描述。每个补丁都应该做一个特定的更改，可以单独检查并验证它所做的事情。
- 作为重申上述准则的一种方法：不要在同一补丁中混合不同类型的更改。如果一个补丁修复了一个关键的安全漏洞，重新排列了一些结构，并重新格式化了代码，那么很有可能它会被忽略，而重要的修复将丢失。
- 每个补丁都应该产生一个内核，它可以正确地构建和运行；如果补丁系列在中间被中断，那么结果应该仍然是一个工作的内核。补丁系列的部分应用是使用“git bisct”工具查找回归的一个常见场景；如果结果是一个损坏的内核，那么对于那些从事追踪问题的高尚工作的开发人员和用户来说，将使他们的生活更加艰难。
- 不过，不要过分。一位开发人员曾经将一组编辑内容作为 500 个单独的补丁发布到一个文件中，这并没有使他成为内核邮件列表中最受欢迎的人。一个补丁可以相当大，只要它仍然包含一个单一的逻辑变更。
- 用一系列补丁添加一个全新的基础设施是很有诱惑力的，但是在系列中的最后一个补丁启用整个补丁之前，该基础设施是不使用的。如果可能的话，应该避免这种诱惑；如果这个系列增加了回归，那么二分法将指出最后一个补丁是导致问题的补丁，即使真正的 bug 在其他地方。只要有可能，添加新代码的补丁程序应该立即激活该代码。

创建完美补丁系列的工作可能是一个令人沮丧的过程，在完成“真正的工作”之后需要花费大量的时间和思考。但是，如果做得好，这是一段很好的时间。

补丁格式和更改日志

所以现在你有了一系列完美的补丁可以发布，但是这项工作还没有完成。每个补丁都需要被格式化成一条消息，它可以快速而清晰地将其目的传达给世界其他地方。为此，每个补丁将由以下部分组成：

- 命名补丁作者的可选“from”行。只有当你通过电子邮件传递别人的补丁时，这一行才是必要的，但是如果有疑问，添加它不会有任何伤害。
- 一行描述补丁的作用。对于没有其他上下文的读者来说，此消息应该足够了解补丁的范围；这是将在“短格式”变更日志中显示的行。此消息通常首先用相关的子系统名称格式化，然后是补丁的目的。例如：

```
gpio: fix build on CONFIG_GPIO_SYSFS=n
```

- 一个空白行，后面是补丁内容的详细描述。这个描述可以是必需的；它应该说明补丁的作用以及为什么它应该应用于内核。
- 一个或多个标记行，至少有一个由补丁作者的：signed-off-by 签名。签名将在下面更详细地描述。

上面的项目一起构成补丁的变更日志。写一篇好的变更日志是一门至关重要但常常被忽视的艺术；值得花一点时间来讨论这个问题。当你写一个变更日志时，你应该记住有很多不同的人会读你的话。其中包括子系统维护人员和审查人员，他们需要决定是否应该包括补丁，分销商和其他维护人员试图决定是否应该将补丁反向移植到其他内核，bug 搜寻人员想知道补丁是否负

责他们正在追查的问题，想知道内核如何变化的用户。等等。一个好的变更日志以最直接和最简洁的方式向所有这些人传达所需的信息。

为此，总结行应该描述变更的影响和动机，以及在一行约束条件下可能发生的变化。然后，详细的描述可以详述这些主题，并提供任何需要的附加信息。如果补丁修复了一个 bug，请引用引入该 bug 的 commit（如果可能，请在引用 commits 时同时提供 commit id 和标题）。如果某个问题与特定的日志或编译器输出相关联，请包含该输出以帮助其他人搜索同一问题的解决方案。如果更改是为了支持以后补丁中的其他更改，那么就这么说。如果更改了内部 API，请详细说明这些更改以及其他开发人员应该如何响应。一般来说，你越能把自己放在每个阅读你的 changelog 的人的位置上，changelog（和内核作为一个整体）就越好。

不用说，变更日志应该是将变更提交到修订控制系统时使用的文本。接下来是：

- 补丁本身，采用统一的（“-u”）补丁格式。将“-p”选项用于 diff 将使函数名与更改相关联，从而使结果补丁更容易被其他人读取。

您应该避免在补丁中包括对不相关文件（例如，由构建过程生成的文件或编辑器备份文件）的更改。文档目录中的文件“dontdiff”在这方面有帮助；使用“-X”选项将其传递给 diff。

上面提到的标签用于描述各种开发人员如何与这个补丁的开发相关联。如何让你的改动进入内核 文档中对它们进行了详细描述；下面是一个简短的总结。每一行的格式如下：

```
tag: Full Name <email address>  optional-other-stuff
```

常用的标签有：

- Signed-off-by: 这是一个开发人员的证明，他或她有权提交补丁以包含到内核中。这是开发来源认证协议，其全文可在如何让你的改动进入内核 中找到，如果没有适当的签字，则不能合并到主线中。
- Co-developed-by: 声明补丁是由多个开发人员共同创建的；当几个人在一个补丁上工作时，它用于将属性赋予共同作者（除了 From: 所赋予的作者之外）。因为 Co-developed-by: 表示作者身份，所以每个共同开发人，必须紧跟在相关合作作者的签名之后。具体内容和示例可以在以下文件中找到如何让你的改动进入内核
- Acked-by: 表示另一个开发人员（通常是相关代码的维护人员）同意补丁适合包含在内核中。
- Tested-by: 声明指定的人已经测试了补丁并发现它可以工作。
- Reviewed-by: 指定的开发人员已经审查了补丁的正确性；有关详细信息，请参阅如何让你的改动进入内核
- Reported-by: 指定报告此补丁修复的问题的用户；此标记用于提供感谢。
- Cc：指定的人收到了补丁的副本，并有机会对此发表评论。

在补丁中添加标签时要小心：只有 cc: 才适合在没有指定人员明确许可的情况下添加。

发送补丁

在邮寄补丁之前，您还需要注意以下几点：

- 您确定您的邮件发送程序不会损坏补丁吗？有免费的空白更改或由邮件客户端执行的行包装的补丁不会在另一端复原，并且通常不会进行任何详细检查。如果有任何疑问，把补丁寄给你自己，让你自己相信它是完好无损的。

  Linux 邮件客户端配置信息 提供了一些有用的提示，可以让特定的邮件客户机工作以发送补丁。

- 你确定你的补丁没有愚蠢的错误吗？您应该始终通过 scripts/checkpatch.pl 运行补丁程序，并解决它提出的投诉。请记住，checkpatch.pl 虽然是大量思考内核补丁应该是什么样子的体现，但它并不比您聪明。如果修复 checkpatch.pl 投诉会使代码变得更糟，请不要这样做。

补丁应始终以纯文本形式发送。请不要将它们作为附件发送；这使得审阅者在答复中更难引用补丁的部分。相反，只需将补丁直接放到您的消息中。

邮寄补丁时，重要的是将副本发送给任何可能感兴趣的人。与其他一些项目不同，内核鼓励人们错误地发送过多的副本；不要假定相关人员会看到您在邮件列表中的发布。尤其是，副本应发送至：

- 受影响子系统的维护人员。如前所述，维护人员文件是查找这些人员的第一个地方。
- 其他在同一领域工作的开发人员，尤其是那些现在可能在那里工作的开发人员。使用 git 查看还有谁修改了您正在处理的文件，这很有帮助。
- 如果您对错误报告或功能请求做出响应，也可以抄送原始发送人。
- 将副本发送到相关邮件列表，或者，如果没有其他应用，则发送到 Linux 内核列表。
- 如果您正在修复一个 bug，请考虑该修复是否应进入下一个稳定更新。如果是这样，stable@vger.kernel.org 应该得到补丁的副本。另外，在补丁本身的标签中添加一个“cc:stable@vger.kernel.org”；这将使稳定团队在修复进入主线时收到通知。

当为一个补丁选择接收者时，最好知道你认为谁最终会接受这个补丁并将其合并。虽然可以将补丁直接发送给 LinusTorvalds 并让他合并，但通常情况下不会这样做。Linus 很忙，并且有子系统维护人员负责监视内核的特定部分。通常您会希望维护人员合并您的补丁。如果没有明显的维护人员，Andrew Morton 通常是最后的补丁目标。

补丁需要好的主题行。补丁程序行的规范格式如下：

```
[PATCH nn/mm] subsys: one-line description of the patch
```

其中“nn”是补丁的序号，“mm”是系列中补丁的总数，“subsys”是受影响子系统的名称。显然，一个单独的补丁可以省略 nn/mm。

如果您有一系列重要的补丁，那么通常将介绍性描述作为零部分发送。不过，这种约定并没有得到普遍遵循；如果您使用它，请记住简介中的信息不会使它进入内核变更日志。因此，请确保补丁本身具有完整的变更日志信息。

一般来说，多部分补丁的第二部分和后续部分应作为对第一部分的回复发送，以便它们在接收端都连接在一起。像 git 和 coilt 这样的工具有命令，可以通过适当的线程发送一组补丁。但是，如果您有一个长系列，并且正在使用 git，请远离–chain reply-to 选项，以避免创建异常深的嵌套。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/6.Followthrough.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

跟进

在这一点上，您已经遵循了到目前为止给出的指导方针，并且，随着您自己的工程技能的增加，已经发布了一系列完美的补丁。即使是经验丰富的内核开发人员也能犯的最大错误之一是，认为他们的工作现在已经完成了。事实上，发布补丁意味着进入流程的下一个阶段，可能还需要做很多工作。

一个补丁在第一次发布时就非常出色，没有改进的余地, 这是很罕见的。内核开发流程认识到这一事实，因此，它非常注重对已发布代码的改进。作为代码的作者，您应该与内核社区合作，以确保您的代码符合内核的质量标准。如果不参与这个过程，很可能会阻止将补丁包含到主线中。

与审阅者合作

任何意义上的补丁都会导致其他开发人员在审查代码时发表大量评论。对于许多开发人员来说，与审查人员合作可能是内核开发过程中最令人生畏的部分。但是，如果你记住一些事情，生活会变得容易得多：

- 如果你已经很好地解释了你的补丁，评论人员会理解它的价值，以及为什么你会费尽心思去写它。但是这个并不能阻止他们提出一个基本的问题：五年或十年后用这个代码维护一个内核会是什么感觉？你可能被要求做出的许多改变——从编码风格的调整到大量的重写——都来自于对 Linux 的理解，即从现在起十年后，Linux 仍将在开发中。
- 代码审查是一项艰苦的工作，这是一项相对吃力不讨好的工作；人们记得谁编写了内核代码，但对于那些审查它的人来说，几乎没有什么持久的名声。因此，评论人员可能会变得暴躁，尤其是当他们看到同样的错误被一遍又一遍地犯下时。如果你得到了一个看起来愤怒、侮辱或完全冒犯你的评论，抵制以同样方式回应的冲动。代码审查是关于代码的，而不是关于人的，代码审查人员不会亲自攻击您。
- 同样，代码审查人员也不想以牺牲你雇主的利益为代价来宣传他们雇主的议程。内核开发人员通常希望今后几年能在内核上工作，但他们明白他们的雇主可能会改变。他们真的，几乎毫无例外地，致力于创造他们所能做到的最好的内核；他们并没有试图给雇主的竞争对手造成不适。

所有这些归根结底都是，当审阅者向您发送评论时，您需要注意他们正在进行的技术观察。不要让他们的表达方式或你自己的骄傲阻止这种事情的发生。当你在一个补丁上得到评论时，花点时间去理解评论人想说什么。如果可能的话，请修复审阅者要求您修复的内容。然后回复审稿人：谢谢他们，并描述你将如何回答他们的问题。

请注意，您不必同意审阅者建议的每个更改。如果您认为审阅者误解了您的代码，请解释到底发生了什么。如果您对建议的更改有技术上的异议，请描述它并证明您对该问题的解决方案是正确的。如果你的解释有道理，审稿人会接受的。不过，如果你的解释不能证明是有说服力的，尤其是当其他人开始同意审稿人的观点时，请花些时间重新考虑一下。你很容易对自己解决问题的方法视而不见，以至于你没有意识到某个问题根本是错误的，或者你甚至没有解决正确的问题。

Andrew Morton 建议，每一条不会导致代码更改的评论都应该导致额外的代码注释；这可以帮助未来的评论人员避免出现第一次出现的问题。

一个致命的错误是忽视评论，希望它们会消失。他们不会走的。如果您在没有对之前收到的注释做出响应的情况下重新发布代码，那么很可能会发现补丁毫无用处。

说到重新发布代码：请记住，审阅者不会记住您上次发布的代码的所有细节。因此，提醒审查人员以前提出的问题以及您如何处理这些问题总是一个好主意；补丁变更日志是提供此类信息的好地方。审阅者不必搜索列表档案来熟悉上次所说的内容；如果您帮助他们开始运行，当他们重新访问您的代码时，他们的心情会更好。

如果你已经试着做正确的事情，但事情仍然没有进展呢？大多数技术上的分歧都可以通过讨论来解决，但有时人们只需要做出决定。如果你真的认为这个决定对你不利，你可以试着向更高的权力上诉。在这篇文章中，更高的权力倾向于 Andrew Morton。Andrew 在内核开发社区中受 i 很大的尊重；他经常为似乎被绝望地阻塞事情清障。尽管如此，对 Andrew 的呼吁不应轻而易举，也不应在所有其他替代方案都被探索之前使用。当然，记住，他也可能不同意你的意见。

接下来会发生什么

如果一个补丁被认为是添加到内核中的一件好事，并且一旦大多数审查问题得到解决，下一步通常是进入子系统维护人员的树中。工作方式因子系统而异；每个维护人员都有自己的工作方式。特别是，可能有不止一棵树——一棵树，也许，专门用于计划下一个合并窗口的补丁，另一棵树用于长期工作。

对于应用于没有明显子系统树（例如内存管理修补程序）的区域的修补程序，默认树通常以-mm结尾。影响多个子系统的补丁也可以最终通过-mm 树。

包含在子系统树中可以提高补丁的可见性。现在，使用该树的其他开发人员将默认获得补丁。子系统树通常也为 Linux 提供支持，使其内容对整个开发社区可见。在这一点上，您很可能会从一组新的审阅者那里得到更多的评论；这些评论需要像上一轮那样得到回答。

在这一点上也会发生什么，这取决于你的补丁的性质，是与其他人正在做的工作发生冲突。在最坏的情况下，严重的补丁冲突可能会导致一些工作被搁置，以便剩余的补丁可以成形并合并。另一些时候，冲突解决将涉及到与其他开发人员合作，可能还会在树之间移动一些补丁，以确保所有的应用都是干净的。这项工作可能是一件痛苦的事情，但要计算您的福祉：在 Linux 下一棵树出现之前，这些冲突通常只在合并窗口中出现，必须迅速解决。现在可以在合并窗口打开之前，在空闲时解决这些问题。

有朝一日，如果一切顺利，您将登录并看到您的补丁已经合并到主线内核中。祝贺你！然而，一旦庆祝活动完成（并且您已经将自己添加到维护人员文件中），就值得记住一个重要的小事实：工作仍然没有完成。并入主线带来了自身的挑战。

首先，补丁的可见性再次提高。可能会有新一轮的开发者评论，他们以前不知道这个补丁。忽略它们可能很有诱惑力，因为您的代码不再存在任何被合并的问题。但是，要抵制这种诱惑，您仍然需要对有问题或建议的开发人员作出响应。

不过，更重要的是：将代码包含在主线中会将代码交给更大的一组测试人员。即使您为尚未提供的硬件提供了驱动程序，您也会惊讶于有多少人会将您的代码构建到内核中。当然，如果有测试人员，也会有错误报告。

最糟糕的错误报告是回归。如果你的补丁导致回归，你会发现很多不舒服的眼睛盯着你；回归需要尽快修复。如果您不愿意或无法修复回归（其他人都不会为您修复），那么在稳定期内，您的补丁几乎肯定会被移除。除了否定您为使补丁进入主线所做的所有工作之外，如果由于未能修复回归而取消补丁，很可能会使将来的工作更难合并。

在处理完任何回归之后，可能还有其他普通的 bug 需要处理。稳定期是修复这些错误并确保代码在主线内核版本中的首次发布尽可能可靠的最好机会。所以，请回答错误报告，并尽可能解决问题。这就是稳定期的目的；一旦解决了旧补丁的任何问题，就可以开始创建酷的新补丁。

别忘了，还有其他里程碑也可能会创建 bug 报告：下一个主线稳定版本，当著名的发行商选择包含补丁的内核版本时，等等。继续响应这些报告是您工作的基本骄傲。但是，如果这不是足够的动机，那么也值得考虑的是，开发社区会记住那些在合并后对代码失去兴趣的开发人员。下一次你发布补丁时，他们会以你以后不会在身边维护它为假设来评估它。

其他可能发生的事情

有一天，你可以打开你的邮件客户端，看到有人给你寄了一个代码补丁。毕竟，这是让您的代码公开存在的好处之一。如果您同意这个补丁，您可以将它转发给子系统维护人员（确保包含一个正确的 From: 行，这样属性是正确的，并添加一个您自己的签准），或者回复一个 Acked-by，让原始发送者向上发送它。

如果您不同意补丁，请发送一个礼貌的回复，解释原因。如果可能的话，告诉作者需要做哪些更改才能让您接受补丁。对于代码的编写者和维护者所反对的合并补丁，存在着一定的阻力，但仅此而已。如果你被认为不必要的阻碍了好的工作，那么这些补丁最终会经过你身边并进入主线。在 Linux 内核中，没有人对任何代码拥有绝对的否决权。除了 Linus。

在非常罕见的情况下，您可能会看到完全不同的东西：另一个开发人员发布了针对您的问题的不同解决方案。在这一点上，两个补丁中的一个可能不会合并，“我的在这里是第一个”不被认为是一个令人信服的技术论据。如果有人的补丁取代了你的补丁而进入了主线，那么只有一种方法可以回应你：高兴你的问题得到解决，继续你的工作。以这种方式把一个人的工作推到一边可能会伤害和气馁，但是在他们忘记了谁的补丁真正被合并很久之后，社区会记住你的反应。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/7.AdvancedTopics.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

高级主题

现在，希望您能够掌握开发流程的工作方式。然而，还有更多的东西要学！本节将介绍一些主题，这些主题对希望成为 Linux 内核开发过程常规部分的开发人员有帮助。

使用 Git 管理补丁

内核使用分布式版本控制始于 2002 年初，当时 Linus 首次开始使用专有的 Bitkeeper 应用程序。虽然 bitkeeper 存在争议，但它所体现的软件版本管理方法却肯定不是。分布式版本控制可以立即加速内核开发项目。在当前的时代，有几种免费的比特保持器替代品。无论好坏，内核项目都将 Git 作为其选择的工具。

使用 Git 管理补丁可以使开发人员的生活更加轻松，尤其是随着补丁数量的增加。Git 也有其粗糙的边缘和一定的危险，它是一个年轻和强大的工具，仍然在其开发人员完善中。本文档不会试图教会读者如何使用 git；这会是个巨长的文档。相反，这里的重点将是 Git 如何特别适合内核开发过程。想要加快 Git 的开发人员可以在以下网站上找到更多信息：

> https://git-scm.com/
>
> https://www.kernel.org/pub/software/scm/git/docs/user-manual.html

在尝试使用它使补丁可供其他人使用之前，第一要务是阅读上述站点，对 Git 的工作方式有一个扎实的了解。使用 Git 的开发人员应该能够获得主线存储库的副本，探索修订历史，提交对树的更改，使用分支等。了解 Git 用于重写历史的工具（如 Rebase）也很有用。Git 有自己的术语和概念；Git 的新用户应该了解 refs、远程分支、索引、快进合并、推拉、分离头等。一开始可能有点吓人，但这些概念不难通过一点学习来理解。

使用 git 生成通过电子邮件提交的补丁是提高速度的一个很好的练习。

当您准备好开始安装 Git 树供其他人查看时，您当然需要一个可以从中提取的服务器。如果您有一个可以访问 Internet 的系统，那么使用 git 守护进程设置这样的服务器相对简单。否则，免费的公共托管网站（例如 github）开始出现在网络上。成熟的开发人员可以在 kernel.org 上获得一个帐户，但这些帐户并不容易找到；有关更多信息，请参阅 https://kernel.org/faq/

正常的 Git 工作流程涉及到许多分支的使用。每一条开发线都可以分为单独的“主题分支”，并独立维护。Git 的分支机构很便宜，没有理由不免费使用它们。而且，在任何情况下，您都不应该在任何您打算让其他人从中受益的分支中进行开发。应该小心地创建公开可用的分支；当它们处于完整的形式并准备好运行时 (而不是之前)，合并开发分支的补丁。

Git 提供了一些强大的工具，可以让您重写开发历史。一个不方便的补丁（比如说，一个打破二分法的补丁，或者有其他一些明显的缺陷）可以在适当的位置修复，或者完全从历史中消失。一个补丁系列可以被重写，就好像它是在今天的主线之上写的一样，即使你已经花了几个月的时间在写它。可以透明地将更改从一个分支转移到另一个分支。等等。明智地使用 git 修改历史的能力可以帮助创建问题更少的干净补丁集。

然而，过度使用这种能力可能会导致其他问题，而不仅仅是对创建完美项目历史的简单痴迷。重写历史将重写该历史中包含的更改，将经过测试（希望）的内核树变为未经测试的内核树。但是，除此之外，如果开发人员没有对项目历史的共享视图，他们就无法轻松地协作；如果您重写了其他开发人员拉入他们存储库的历史，您将使这些开发人员的生活更加困难。因此，这里有一个简单的经验法则：被导出到其他人的历史在此后通常被认为是不可变的。

因此，一旦将一组更改推送到公开可用的服务器上，就不应该重写这些更改。如果您尝试强制进行不会导致快进合并（即不共享同一历史记录的更改）的更改，Git 将尝试强制执行此规则。可以重写此检查，有时可能需要重写导出的树。在树之间移动变更集以避免 Linux-next 中的

冲突就是一个例子。但这种行为应该是罕见的。这就是为什么开发应该在私有分支中进行（必要时可以重写）并且只有在公共分支处于合理的高级状态时才转移到公共分支中的原因之一。

当主线（或其他一组变更所基于的树）前进时，很容易与该树合并以保持领先地位。对于一个私有的分支，rebasing 可能是一个很容易跟上另一棵树的方法，但是一旦一棵树被导出到全世界，rebasing 就不是一个选项。一旦发生这种情况，就必须进行完全合并（merge）。合并有时是很有意义的，但是过于频繁的合并会不必要地扰乱历史。在这种情况下，建议的技术是不经常合并，通常只在特定的发布点（如主线-rc 发布）合并。如果您对特定的更改感到紧张，则可以始终在私有分支中执行测试合并。在这种情况下，git rerere 工具很有用；它记住合并冲突是如何解决的，这样您就不必重复相同的工作。

关于 Git 这样的工具的一个最大的反复抱怨是：补丁从一个存储库到另一个存储库的大量移动使得很容易陷入错误建议的变更中，这些变更避开审查雷达进入主线。当内核开发人员看到这种情况发生时，他们往往会感到不高兴；在 Git 树上放置未查看或主题外的补丁可能会影响您将来获取树的能力。引用 Linus:

> 你可以给我发补丁，但要我从你哪里取一个 Git 补丁，我需要知道你知道
> 你在做什么，我需要能够相信事情而不去检查每个个人改变。

（http://lwn.net/articles/224135/）。

为了避免这种情况，请确保给定分支中的所有补丁都与相关主题紧密相关；“驱动程序修复”分支不应更改核心内存管理代码。而且，最重要的是，不要使用 Git 树来绕过审查过程。不时的将树的摘要发布到相关的列表中，当时间合适时，请求 Linux-next 中包含该树。

如果其他人开始发送补丁以包含到您的树中，不要忘记查看它们。还要确保您维护正确的作者信息；git am 工具在这方面做得最好，但是如果它通过第三方转发给您，您可能需要在补丁中添加“From:”行。

请求 pull 操作时，请务必提供所有相关信息：树的位置、要拉的分支以及拉操作将导致的更改。在这方面，git request pull 命令非常有用；它将按照其他开发人员的预期格式化请求，并检查以确保您记住了将这些更改推送到公共服务器。

审查补丁

一些读者当然会反对将本节与“高级主题”放在一起，因为即使是刚开始的内核开发人员也应该检查补丁。当然，学习如何在内核环境中编程没有比查看其他人发布的代码更好的方法了。此外，审阅者永远供不应求；通过查看代码，您可以对整个流程做出重大贡献。

审查代码可能是一个令人生畏的前景，特别是对于一个新的内核开发人员来说，他们可能会对公开询问代码感到紧张，而这些代码是由那些有更多经验的人发布的。不过，即使是最有经验的开发人员编写的代码也可以得到改进。也许对评审员（所有评审员）最好的建议是：把评审评论当成问题而不是批评。询问“在这条路径中如何释放锁?”总是比说“这里的锁是错误的”更好。

不同的开发人员将从不同的角度审查代码。一些主要关注的是编码样式以及代码行是否有尾随空格。其他人将主要关注补丁作为一个整体实现的变更是否对内核有好处。然而，其他人会检查是否存在锁定问题、堆栈使用过度、可能的安全问题、在其他地方发现的代码重复、足够的文档、对性能的不利影响、用户空间 ABI 更改等。所有类型的检查，如果它们导致更好的代码进入内核，都是受欢迎和值得的。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/8.Conclusion.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

更多信息

关于 Linux 内核开发和相关主题的信息来源很多。首先是在内核源代码分发中找到的文档目录。顶级如何参与 Linux 内核开发 文件是一个重要的起点如何让你的改动进入内核 和 process/submitting-drivers.rst 也是所有内核开发人员都应该阅读的内容。许多内部内核 API 都是使用 kerneldoc 机制记录的；“make htmldocs”或“make pdfdocs”可用于以 HTML 或 PDF 格式生成这些文档（尽管某些发行版提供的 tex 版本会遇到内部限制，无法正确处理文档）。

不同的网站在各个细节层次上讨论内核开发。您的作者想谦虚地建议用 https://lwn.net/ 作为来源；有关许多特定内核主题的信息可以通过以下网址的 lwn 内核索引找到：

http://lwn.net/kernel/index/

除此之外，内核开发人员的一个宝贵资源是：

https://kernelnewbies.org/

当然，我们不应该忘记 https://kernel.org/ 这是内核发布信息的最终位置。

关于内核开发有很多书：

Linux 设备驱动程序，第三版（Jonathan Corbet、Alessandro Rubini 和 Greg Kroah Hartman）。在线：http://lwn.net/kernel/ldd3/

Linux 内核开发（Robert Love）。

了解 Linux 内核（Daniel Bovet 和 Marco Cesati）。

然而，所有这些书都有一个共同的缺点：当它们上架时，它们往往有些过时，而且它们已经上架一段时间了。不过，在那里还可以找到相当多的好信息。

有关 git 的文档，请访问：

https://www.kernel.org/pub/software/scm/git/docs/

https://www.kernel.org/pub/software/scm/git/docs/user-manual.html

结论

祝贺所有通过这篇冗长的文件的人。希望它能够帮助您理解 Linux 内核是如何开发的，以及您如何参与这个过程。

最后，重要的是参与。任何开源软件项目都不超过其贡献者投入其中的总和。Linux 内核的发展速度和以前一样快，因为它得到了大量开发人员的帮助，他们都在努力使它变得更好。内核是一个主要的例子，说明当成千上万的人为了一个共同的目标一起工作时，可以做些什么。

不过，内核总是可以从更大的开发人员基础中获益。总有更多的工作要做。但是，同样重要的是，Linux 生态系统中的大多数其他参与者可以通过为内核做出贡献而受益。使代码进入主线是提高代码质量、降低维护和分发成本、提高对内核开发方向的影响程度等的关键。这是一种人人都赢的局面。踢开你的编辑，来加入我们吧，你会非常受欢迎的。

本文档的目的是帮助开发人员（及其经理）以最小的挫折感与开发社区合作。它试图记录这个社区如何以一种不熟悉 Linux 内核开发（或者实际上是自由软件开发）的人可以访问的方式工作。虽然这里有一些技术资料，但这是一个面向过程的讨论，不需要深入了解内核编程就可以理解。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/email-clients.rst

译者:

```
中文版维护者:  贾威威  Harry Wei <harryxiyou@gmail.com>
中文版翻译者:  贾威威  Harry Wei <harryxiyou@gmail.com>
                 时奎亮  Alex Shi <alex.shi@linux.alibaba.com>
中文版校译者:  Yinglin Luan <synmyth@gmail.com>
                 Xiaochen Wang <wangxiaochen0@gmail.com>
                 yaxinsn <yaxinsn@163.com>
```

* Linux 邮件客户端配置信息

Git

现在大多数开发人员使用 `git send-email` 而不是常规的电子邮件客户端。这方面的手册非常好。在接收端，维护人员使用 `git am` 加载补丁。

如果你是 `git` 新手，那么把你的第一个补丁发送给你自己。将其保存为包含所有标题的原始文本。运行 `git am raw_email.txt` ，然后使用 `git log` 查看更改日志。如果工作正常，再将补丁发送到相应的邮件列表。

普通配置

Linux 内核补丁是通过邮件被提交的，最好把补丁作为邮件体的内嵌文本。有些维护者接收附件，但是附件的内容格式应该是”text/plain”。然而，附件一般是不赞成的，因为这会使补丁的引用部分在评论过程中变的很困难。

用来发送 Linux 内核补丁的邮件客户端在发送补丁时应该处于文本的原始状态。例如，他们不能改变或者删除制表符或者空格，甚至是在每一行的开头或者结尾。

不要通过”format=flowed”模式发送补丁。这样会引起不可预期以及有害的断行。

不要让你的邮件客户端进行自动换行。这样也会破坏你的补丁。

邮件客户端不能改变文本的字符集编码方式。要发送的补丁只能是 ASCII 或者 UTF-8 编码方式，如果你使用 UTF-8 编码方式发送邮件，那么你将会避免一些可能发生的字符集问题。

邮件客户端应该形成并且保持 References: 或者 In-Reply-To: 标题，那么邮件话题就不会中断。

复制粘帖 (或者剪贴粘帖) 通常不能用于补丁，因为制表符会转换为空格。使用 xclipboard, xclip 或者 xcutsel 也许可以，但是最好测试一下或者避免使用复制粘帖。

不要在使用 PGP/GPG 署名的邮件中包含补丁。这样会使得很多脚本不能读取和适用于你的补丁。(这个问题应该是可以修复的)

在给内核邮件列表发送补丁之前，给自己发送一个补丁是个不错的主意，保存接收到的邮件，将补丁用’patch’命令打上，如果成功了，再给内核邮件列表发送。

一些邮件客户端提示

这里给出一些详细的 MUA 配置提示，可以用于给 Linux 内核发送补丁。这些并不意味是所有的软件包配置总结。

说明：TUI = 以文本为基础的用户接口 GUI = 图形界面用户接口

Alpine (TUI)

配置选项：在”Sending Preferences”部分：

- “Do Not Send Flowed Text”必须开启
- “Strip Whitespace Before Sending”必须关闭

当写邮件时，光标应该放在补丁会出现的地方，然后按下 CTRL-R 组合键，使指定的补丁文件嵌入到邮件中。

Evolution (GUI)

一些开发者成功的使用它发送补丁

当选择邮件选项：Preformat
从 Format->Heading->Preformatted (Ctrl-7) 或者工具栏

然后使用：
Insert->Text File…(Alt-n x) 插入补丁文件。

你还可以”diff -Nru old.c new.c | xclip”，选择 Preformat，然后使用中间键进行粘帖。

Kmail (GUI)

一些开发者成功的使用它发送补丁。

默认设置不为 HTML 格式是合适的；不要启用它。

当书写一封邮件的时候，在选项下面不要选择自动换行。唯一的缺点就是你在邮件中输入的任何文本都不会被自动换行，因此你必须在发送补丁之前手动换行。最简单的方法就是启用自动换行来书写邮件，然后把它保存为草稿。一旦你在草稿中再次打开它，它已经全部自动换行了，那么你的邮件虽然没有选择自动换行，但是还不会失去已有的自动换行。

在邮件的底部，插入补丁之前，放上常用的补丁定界符：三个连字号 (—)。

然后在”Message”菜单条目，选择插入文件，接着选取你的补丁文件。还有一个额外的选项，你可以通过它配置你的邮件建立工具栏菜单，还可以带上”insert file”图标。

你可以安全地通过 GPG 标记附件，但是内嵌补丁最好不要使用 GPG 标记它们。作为内嵌文本的签发补丁，当从 GPG 中提取 7 位编码时会使他们变的更加复杂。

如果你非要以附件的形式发送补丁，那么就右键点击附件，然后选中属性，突出”Suggest automatic display”，这样内嵌附件更容易让读者看到。

当你要保存将要发送的内嵌文本补丁，你可以从消息列表窗格选择包含补丁的邮件，然后右击选择“save as”。你可以使用一个没有更改的包含补丁的邮件，如果它是以正确的形式组成。当你正真在它自己的窗口之下察看，那时没有选项可以保存邮件–已经有一个这样的 bug 被汇报到了 kmail 的 bugzilla 并且希望这将会被处理。邮件是以只针对某个用户可读写的权限被保存的，所以如果你想把邮件复制到其他地方，你不得不把他们的权限改为组或者整体可读。

Lotus Notes (GUI)

不要使用它。

Mutt (TUI)

很多 Linux 开发人员使用 mutt 客户端，所以证明它肯定工作的非常漂亮。

Mutt 不自带编辑器，所以不管你使用什么编辑器都不应该带有自动断行。大多数编辑器都带有一个”insert file”选项，它可以通过不改变文件内容的方式插入文件。

‘vim’作为 mutt 的编辑器：
set editor=”vi”

如果使用 xclip，敲入以下命令:set paste 按中键之前或者 shift-insert 或者使用:r filename

如果想要把补丁作为内嵌文本。(a)ttach 工作的很好，不带有”set paste”。

你可以通过 `git format-patch` 生成补丁，然后用 Mutt 发送它们:

```
$ mutt -H 0001-some-bug-fix.patch
```

配置选项: 它应该以默认设置的形式工作。然而，把”send_charset”设置为”us-ascii::utf-8”也是一个不错的主意。

Mutt 是高度可配置的。这里是个使用 mutt 通过 Gmail 发送的补丁的最小配置:

```
# .muttrc
# ================  IMAP ====================
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ================  SMTP  ====================
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
set ssl_force_tls = yes # Require encrypted connection

# ================  Composition  ====================
set editor = `echo \$EDITOR`
set edit_headers = yes  # See the headers when editing
set charset = UTF-8     # value of $LANG; also fallback for send_
↪charset
# Sender, email address, and sign-off line must match
unset use_domain        # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes
```

Mutt 文档含有更多信息:

http://dev.mutt.org/trac/wiki/UseCases/Gmail

http://dev.mutt.org/doc/manual.html

Pine (TUI)

Pine 过去有一些空格删减问题，但是这些现在应该都被修复了。

如果可以，请使用 alpine(pine 的继承者)

配置选项：- 最近的版本需要消除流程文本 - “no-strip-whitespace-before-send”选项也是需要的。

Sylpheed (GUI)

- 内嵌文本可以很好的工作（或者使用附件）。
- 允许使用外部的编辑器。
- 对于目录较多时非常慢。
- 如果通过 non-SSL 连接，无法使用 TLS SMTP 授权。
- 在组成窗口中有一个很有用的 ruler bar。
- 给地址本中添加地址就不会正确的了解显示名。

Thunderbird (GUI)

默认情况下，thunderbird 很容易损坏文本，但是还有一些方法可以强制它变得更好。

- 在用户帐号设置里，组成和寻址，不要选择”Compose messages in HTML format”。
- 编辑你的 Thunderbird 配置设置来使它不要拆行使用：user_pref(“mail-news.wraplength”, 0);
- 编辑你的 Thunderbird 配置设置，使它不要使用”format=flowed”格式：user_pref(“mailnews. send_plaintext_flowed”, false);
- 你需要使 Thunderbird 变为预先格式方式：如果默认情况下你书写的是 HTML 格式，那不是很难。仅仅从标题栏的下拉框中选择”Preformat”格式。如果默认情况下你书写的是文本格式，你不得把它改为 HTML 格式（仅仅作为一次性的）来书写新的消息，然后强制使它回到文本格式，否则它就会拆行。要实现它，在写信的图标上使用 shift 键来使它变为 HTML 格式，然后标题栏的下拉框中选择”Preformat”格式。
- 允许使用外部的编辑器：针对 Thunderbird 打补丁最简单的方法就是使用一个”external editor”扩展，然后使用你最喜欢的 $EDITOR 来读取或者合并补丁到文本中。要实现它，可以下载并且安装这个扩展，然后添加一个使用它的按键 View->Toolbars->Customize…最后当你书写信息的时候仅仅点击它就可以了。

TkRat (GUI)

可以使用它。使用”Insert file…”或者外部的编辑器。

Gmail (Web GUI)

不要使用它发送补丁。

Gmail 网页客户端自动地把制表符转换为空格。

虽然制表符转换为空格问题可以被外部编辑器解决，同时它还会使用回车换行把每行拆分为 78 个字符。

另一个问题是 Gmail 还会把任何不是 ASCII 的字符的信息改为 base64 编码。它把东西变的像欧洲人的名字。

###

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/license-rules.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核许可规则

Linux 内核根据 LICENSES/preferred/GPL-2.0 中提供的 GNU 通用公共许可证版本 2（GPL-2.0）的条款提供，并在 LICENSES/exceptions/Linux-syscall-note 中显式描述了例外的系统调用，如 COPYING 文件中所述。

此文档文件提供了如何对每个源文件进行注释以使其许可证清晰明确的说明。它不会取代内核的许可证。

内核源代码作为一个整体适用于 COPYING 文件中描述的许可证，但是单个源文件可以具有不同的与 GPL-20 兼容的许可证:

```
GPL-1.0+ : GNU 通用公共许可证 v1.0 或更高版本
GPL-2.0+ : GNU 通用公共许可证 v2.0 或更高版本
LGPL-2.0 : 仅限 GNU 库通用公共许可证 v2
LGPL-2.0+: GNU 库通用公共许可证 v2 或更高版本
LGPL-2.1 : 仅限 GNU 宽通用公共许可证 v2.1
LGPL-2.1+: GNU 宽通用公共许可证 v2.1 或更高版本
```

除此之外，个人文件可以在双重许可下提供，例如一个兼容的 GPL 变体，或者 BSD，MIT 等许可。

用户空间 API（UAPI）头文件描述了用户空间程序与内核的接口，这是一种特殊情况。根据内核 COPYING 文件中的注释，syscall 接口是一个明确的边界，它不会将 GPL 要求扩展到任何使用它与内核通信的软件。由于 UAPI 头文件必须包含在创建在 Linux 内核上运行的可执行文件的任何源文件中，因此此例外必须记录在特别的许可证表述中。

表达源文件许可证的常用方法是将匹配的样板文本添加到文件的顶部注释中。由于格式，拼写错误等，这些“样板”很难通过那些在上下文中使用的验证许可证合规性的工具。

样板文本的替代方法是在每个源文件中使用软件包数据交换（SPDX）许可证标识符。SPDX 许可证标识符是机器可解析的，并且是用于提供文件内容的许可证的精确缩写。SPDX 许可证标识符由 Linux 基金会的 SPDX 工作组管理，并得到了整个行业，工具供应商和法律团队的合作伙伴的一致同意。有关详细信息，请参阅 https://spdx.org/

Linux 内核需要所有源文件中的精确 SPDX 标识符。内核中使用的有效标识符在许可标识符一节中进行了解释，并且已可以在 https://spdx.org/licenses/ 上的官方 SPDX 许可证列表中检索，并附带许可证文本。

许可标识符语法

1. 安置:

   内核文件中的 SPDX 许可证标识符应添加到可包含注释的文件中的第一行。对于大多
   数文件，这是第一行，除了那些在第一行中需要’#!PATH_TO_INTERPRETER’的脚本。对于这些脚本，SPDX 标识符进入第二行。

2. 风格:

   SPDX 许可证标识符以注释的形式添加。注释样式取决于文件类型:

   ```
   C source: // SPDX-License-Identifier: <SPDX License Expression>
   C header: /* SPDX-License-Identifier: <SPDX License Expression>␣
    ↪*/
   ASM:      /* SPDX-License-Identifier: <SPDX License Expression>␣
    ↪*/
   scripts:  # SPDX-License-Identifier: <SPDX License Expression>
   .rst:     .. SPDX-License-Identifier: <SPDX License Expression>
   .dts{i}:  // SPDX-License-Identifier: <SPDX License Expression>
   ```

   如果特定工具无法处理标准注释样式，则应使用工具接受的相应注释机制。这是在 C 头文件中使用“/**/”样式注释的原因。过去在使用生成的.lds 文件中观察到构建被破坏，其中’ld’无法解析 C++ 注释。现在已经解决了这个问题，但仍然有较旧的汇编程序工具无法处理 C++ 样式的注释。

3. 句法:

<SPDX 许可证表达式 > 是 SPDX 许可证列表中的 SPDX 短格式许可证标识符，或者在许可证例外适用时由“WITH”分隔的两个 SPDX 短格式许可证标识符的组合。当应用多个许可证时，表达式由分隔子表达式的关键字“AND”，“OR”组成，并由“(”，“)”包围。

带有“或更高”选项的 [L]GPL 等许可证的许可证标识符通过使用“+”来表示“或更高”选项来构建。：

```
// SPDX-License-Identifier: GPL-2.0+
// SPDX-License-Identifier: LGPL-2.1+
```

当需要修正的许可证时，应使用 WITH。 例如，linux 内核 UAPI 文件使用表达式:

```
// SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
// SPDX-License-Identifier: GPL-2.0+ WITH Linux-syscall-note
```

其它在内核中使用 WITH 例外的事例如下:

```
// SPDX-License-Identifier: GPL-2.0 WITH mif-exception
// SPDX-License-Identifier: GPL-2.0+ WITH GCC-exception-2.0
```

例外只能与特定的许可证标识符一起使用。有效的许可证标识符列在异常文本文件的标记中。有关详细信息，请参阅许可标识符 一章中的例外 。

如果文件是双重许可且只选择一个许可证，则应使用 OR。例如，一些 dtsi 文件在双许可下可用:

```
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
```

内核中双许可文件中许可表达式的示例:

```
// SPDX-License-Identifier: GPL-2.0 OR MIT
// SPDX-License-Identifier: GPL-2.0 OR BSD-2-Clause
// SPDX-License-Identifier: GPL-2.0 OR Apache-2.0
// SPDX-License-Identifier: GPL-2.0 OR MPL-1.1
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note)␣
↪OR MIT
// SPDX-License-Identifier: GPL-1.0+ OR BSD-3-Clause OR OpenSSL
```

如果文件具有多个许可证，其条款全部适用于使用该文件，则应使用 AND。例如，如果代码是从另一个项目继承的，并且已经授予了将其放入内核的权限，但原始许可条款需要保持有效:

```
// SPDX-License-Identifier: (GPL-2.0 WITH Linux-syscall-note)␣
↪AND MIT
```

另一个需要遵守两套许可条款的例子是:

```
// SPDX-License-Identifier: GPL-1.0+ AND LGPL-2.1+
```

许可标识符

当前使用的许可证以及添加到内核的代码许可证可以分解为：

1. 优先许可:

   应尽可能使用这些许可证，因为它们已知完全兼容并广泛使用。这些许可证在内核目录:

   ```
   LICENSES/preferred/
   ```

   此目录中的文件包含完整的许可证文本和元标记 。文件名与 SPDX 许可证标识符相同，后者应用于源文件中的许可证。

   例如:

   ```
   LICENSES/preferred/GPL-2.0
   ```

   包含 GPLv2 许可证文本和所需的元标签:

   ```
   LICENSES/preferred/MIT
   ```

   包含 MIT 许可证文本和所需的元标记

   元标记:

   许可证文件中必须包含以下元标记：

   - Valid-License-Identifier:

   一行或多行，声明那些许可标识符在项目内有效，以引用此特定许可的文本。通
   常这是一个有效的标识符，但是例如对于带有’或更高’选项的许可证，两个标识符都有效。

   - SPDX-URL:

     SPDX 页面的 URL，其中包含与许可证相关的其他信息.

   - Usage-Guidance:

     使用建议的自由格式文本。该文本必须包含 SPDX 许可证标识符的正确示例，因为它们应根据许可标识符语法 指南放入源文件中。

   - License-Text:

     此标记之后的所有文本都被视为原始许可文本

   文件格式示例:

   ```
   Valid-License-Identifier: GPL-2.0
   Valid-License-Identifier: GPL-2.0+
   SPDX-URL: https://spdx.org/licenses/GPL-2.0.html
   Usage-Guide:
     To use this license in source code, put one of the following␣
   ↪SPDX
     tag/value pairs into a comment according to the placement
     guidelines in the licensing rules documentation.
     For 'GNU General Public License (GPL) version 2 only' use:
       SPDX-License-Identifier: GPL-2.0
   ```

(continues on next page)

(continued from previous page)

```
  For 'GNU General Public License (GPL) version 2 or any later␣
↪version' use:
    SPDX-License-Identifier: GPL-2.0+
License-Text:
  Full license text
```

```
SPDX-License-Identifier: MIT
SPDX-URL: https://spdx.org/licenses/MIT.html
Usage-Guide:
  To use this license in source code, put the following SPDX
  tag/value pair into a comment according to the placement
  guidelines in the licensing rules documentation.
    SPDX-License-Identifier: MIT
License-Text:
  Full license text
```

2. 不推荐的许可证:

   这些许可证只应用于现有代码或从其他项目导入代码。这些许可证在内核目录:

   ```
   LICENSES/other/
   ```

   此目录中的文件包含完整的许可证文本和元标记 。文件名与 SPDX 许可证标识符相同，后者应用于源文件中的许可证。

   例如:

   ```
   LICENSES/other/ISC
   ```

   包含国际系统联合许可文本和所需的元标签:

   ```
   LICENSES/other/ZLib
   ```

   包含 ZLIB 许可文本和所需的元标签.

   元标签:

   “其他”许可证的元标签要求与优先许可 的要求相同。

   文件格式示例:

   ```
   Valid-License-Identifier: ISC
   SPDX-URL: https://spdx.org/licenses/ISC.html
   Usage-Guide:
     Usage of this license in the kernel for new code is␣
   ↪discouraged
     and it should solely be used for importing code from an␣
   ↪already
   ```

(continues on next page)

(continued from previous page)

```
  existing project.
  To use this license in source code, put the following SPDX
  tag/value pair into a comment according to the placement
  guidelines in the licensing rules documentation.
    SPDX-License-Identifier: ISC
License-Text:
  Full license text
```

3. 例外:

   某些许可证可以修改，并允许原始许可证不具有的某些例外权利。这些例外在内核目录:

   ```
   LICENSES/exceptions/
   ```

   此目录中的文件包含完整的例外文本和所需的例外元标记 。

   例如:

   ```
   LICENSES/exceptions/Linux-syscall-note
   ```

   包含 Linux 内核的 COPYING 文件中记录的 Linux 系统调用例外，该文件用于 UAPI 头文件。例如:

   ```
   LICENSES/exceptions/GCC-exception-2.0
   ```

   包含 GCC’链接例外’，它允许独立于其许可证的任何二进制文件与标记有此例外的文件的编译版本链接。这是从 GPL 不兼容源代码创建可运行的可执行文件所必需的。

   例外元标记:

   以下元标记必须在例外文件中可用：

   - SPDX-Exception-Identifier:

   一个可与 SPDX 许可证标识符一起使用的例外标识符。

   - SPDX-URL:

     SPDX 页面的 URL，其中包含与例外相关的其他信息。

   - SPDX-Licenses:

   以逗号分隔的例外可用的 SPDX 许可证标识符列表。

   - Usage-Guidance:

     使用建议的自由格式文本。必须在文本后面加上 SPDX 许可证标识符的正确示例，因为它们应根据许可标识符语法 指南放入源文件中。

   - Exception-Text:

     此标记之后的所有文本都被视为原始异常文本

   文件格式示例:

```
SPDX-Exception-Identifier: Linux-syscall-note
SPDX-URL: https://spdx.org/licenses/Linux-syscall-note.html
SPDX-Licenses: GPL-2.0, GPL-2.0+, GPL-1.0+, LGPL-2.0, LGPL-2.0+,
 ↪ LGPL-2.1, LGPL-2.1+
Usage-Guidance:
  This exception is used together with one of the above SPDX-
 ↪Licenses
  to mark user-space API (uapi) header files so they can be␣
 ↪included
  into non GPL compliant user-space application code.
  To use this exception add it with the keyword WITH to one of␣
 ↪the
  identifiers in the SPDX-Licenses tag:
    SPDX-License-Identifier: <SPDX-License> WITH Linux-syscall-
 ↪note
Exception-Text:
  Full exception text
```

```
SPDX-Exception-Identifier: GCC-exception-2.0
SPDX-URL: https://spdx.org/licenses/GCC-exception-2.0.html
SPDX-Licenses: GPL-2.0, GPL-2.0+
Usage-Guidance:
  The "GCC Runtime Library exception 2.0" is used together with␣
 ↪one
  of the above SPDX-Licenses for code imported from the GCC␣
 ↪runtime
  library.
  To use this exception add it with the keyword WITH to one of␣
 ↪the
  identifiers in the SPDX-Licenses tag:
    SPDX-License-Identifier: <SPDX-License> WITH GCC-exception-
 ↪2.0
Exception-Text:
  Full exception text
```

所有 SPDX 许可证标识符和例外都必须在 LICENSES 子目录中具有相应的文件。这是允许工具验证（例如 checkpatch.pl）以及准备好从源读取和提取许可证所必需的，这是各种 FOSS 组织推荐的，例如 FSFE REUSE initiative.

模块许可

可加载内核模块还需要 MODULE_LICENSE（）标记。此标记既不替代正确的源代码许可证信息（SPDX-License-Identifier），也不以任何方式表示或确定提供模块源代码的确切许可证。

此标记的唯一目的是提供足够的信息，该模块是否是自由软件或者是内核模块加载器和用户空间工具的专有模块。

MODULE_LICENSE（）的有效许可证字符串是:

“GPL”	模块是根据 GPL 版本 2 许可的。这并不表示仅限于 GPL-2.0 或 GPL-2.0 或更高版本之间的任何区别。最正确许可证信息只能通过相应源文件中的许可证信息来确定
“GPL v2”	和”GPL”相同，它的存在是因为历史原因。
“GPL and additional rights”	表示模块源在 GPL v2 变体和 MIT 许可下双重许可的历史变体。请不要在新代码中使用。
“Dual MIT/GP	表达该模块在 GPL v2 变体或 MIT 许可证选择下双重许可的正确方式。
“Dual BSD/GP	该模块根据 GPL v2 变体或 BSD 许可证选择进行双重许可。BSD 许可证的确切变体只能通过相应源文件中的许可证信息来确定。
“Dual MPL/GF	该模块根据 GPL v2 变体或 Mozilla Public License（MPL）选项进行双重许可。MPL 许可证的确切变体只能通过相应的源文件中的许可证信息来确定。
“Proprietary”	该模块属于专有许可。此字符串仅用于专有的第三方模块，不能用于在内核树中具有源代码的模块。以这种方式标记的模块在加载时会使用’P’标记污染内核，并且内核模块加载器拒绝将这些模块链接到使用 EXPORT_SYMBOL_GPL () 导出的符号。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/kernel-enforcement-statement.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核执行声明

作为 Linux 内核的开发人员，我们对如何使用我们的软件以及如何实施软件许可证有着浓厚的兴趣。遵守 GPL-2.0 的互惠共享义务对我们软件和社区的长期可持续性至关重要。

虽然有权强制执行对我们社区的贡献中的单独版权权益，但我们有共同的利益，即确保个人强制执行行动的方式有利于我们的社区，不会对我们软件生态系统的健康和增长产生意外的负面影响。为了阻止无益的执法行动，我们同意代表我们自己和我们版权利益的任何继承人对 Linux 内核用户作出以下符合我们开发社区最大利益的承诺:

尽管有 GPL-2.0 的终止条款，我们同意，采用以下 GPL-3.0 条款作为我们许可证

> 下的附加许可，作为任何对许可证下权利的非防御性主张，这符合我们开发社区的最佳利益。
>
> > 但是，如果您停止所有违反本许可证的行为，则您从特定版权持有人处获得的许可证将被恢复：(a) 暂时恢复，除非版权持有人明确并最终终止您的许可证；以及 (b) 永久恢复, 如果版权持有人未能在你终止违反后 60 天内以合理方式通知您违反本许可证的行为，则永久恢复您的许可证。
> >
> > 此外，如果版权所有者以某种合理的方式通知您违反了本许可，这是您第一次从该版权所有者处收到违反本许可的通知（对于任何作品），并且您在收到通知后的 30 天内纠正违规行为。则您从特定版权所有者处获得的许可将永久恢复.

我们提供这些保证的目的是鼓励更多地使用该软件。我们希望公司和个人使用、修改和分发此软件。我们希望以公开和透明的方式与用户合作，以消除我们对法规遵从性或强制执行的任何不确定性，这些不确定性可能会限制我们软件的采用。我们将法律行动视为最后手段，只有在其他社区努力未能解决这一问题时才采取行动。

最后，一旦一个不合规问题得到解决，我们希望用户会感到欢迎，加入我们为之努力的这个项目。共同努力，我们会更强大。

除了下面提到的以外，我们只为自己说话，而不是为今天、过去或将来可能为之工作的任何公司说话。

- Laura Abbott
- Bjorn Andersson (Linaro)
- Andrea Arcangeli
- Neil Armstrong
- Jens Axboe
- Pablo Neira Ayuso
- Khalid Aziz
- Ralf Baechle
- Felipe Balbi
- Arnd Bergmann
- Ard Biesheuvel
- Tim Bird
- Paolo Bonzini
- Christian Borntraeger
- Mark Brown (Linaro)
- Paul Burton
- Javier Martinez Canillas
- Rob Clark
- Kees Cook (Google)
- Jonathan Corbet

- Dennis Dalessandro
- Vivien Didelot (Savoir-faire Linux)
- Hans de Goede
- Mel Gorman (SUSE)
- Sven Eckelmann
- Alex Elder (Linaro)
- Fabio Estevam
- Larry Finger
- Bhumika Goyal
- Andy Gross
- Juergen Gross
- Shawn Guo
- Ulf Hansson
- Stephen Hemminger (Microsoft)
- Tejun Heo
- Rob Herring
- Masami Hiramatsu
- Michal Hocko
- Simon Horman
- Johan Hovold (Hovold Consulting AB)
- Christophe JAILLET
- Olof Johansson
- Lee Jones (Linaro)
- Heiner Kallweit
- Srinivas Kandagatla
- Jan Kara
- Shuah Khan (Samsung)
- David Kershner
- Jaegeuk Kim
- Namhyung Kim
- Colin Ian King
- Jeff Kirsher
- Greg Kroah-Hartman (Linux Foundation)
- Christian König

- Vinod Koul
- Krzysztof Kozlowski
- Viresh Kumar
- Aneesh Kumar K.V
- Julia Lawall
- Doug Ledford
- Chuck Lever (Oracle)
- Daniel Lezcano
- Shaohua Li
- Xin Long
- Tony Luck
- Catalin Marinas (Arm Ltd)
- Mike Marshall
- Chris Mason
- Paul E. McKenney
- Arnaldo Carvalho de Melo
- David S. Miller
- Ingo Molnar
- Kuninori Morimoto
- Trond Myklebust
- Martin K. Petersen (Oracle)
- Borislav Petkov
- Jiri Pirko
- Josh Poimboeuf
- Sebastian Reichel (Collabora)
- Guenter Roeck
- Joerg Roedel
- Leon Romanovsky
- Steven Rostedt (VMware)
- Frank Rowand
- Ivan Safonov
- Anna Schumaker
- Jes Sorensen
- K.Y. Srinivasan

- David Sterba (SUSE)
- Heiko Stuebner
- Jiri Kosina (SUSE)
- Willy Tarreau
- Dmitry Torokhov
- Linus Torvalds
- Thierry Reding
- Rik van Riel
- Luis R. Rodriguez
- Geert Uytterhoeven (Glider bvba)
- Eduardo Valentin (Amazon.com)
- Daniel Vetter
- Linus Walleij
- Richard Weinberger
- Dan Williams
- Rafael J. Wysocki
- Arvind Yadav
- Masahiro Yamada
- Wei Yongjun
- Lv Zheng
- Marc Zyngier (Arm Ltd)

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/kernel-driver-statement.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 内核驱动声明

关于 Linux 内核模块的立场声明

我们，以下署名的 Linux 内核开发人员，认为任何封闭源 Linux 内核模块或驱动程序都是有害的和不可取的。我们已经一再发现它们对 Linux 用户，企业和更大的 Linux 生态系统有害。这样的模块否定了 Linux 开发模型的开放性，稳定性，灵活性和可维护性，并使他们的用户无法使用 Linux 社区的专业知识。提供闭源内核模块的供应商迫使其客户放弃 Linux 的主要优势或选择新的供应商。因此，为了充分利用开源所提供的成本节省和共享支持优势，我们敦促供应商采取措施，以开源内核代码在 Linux 上为其客户提供支持。

我们只为自己说话，而不是我们今天可能会为之工作，过去或将来会为之工作的任何公司。

- Dave Airlie
- Nick Andrew
- Jens Axboe
- Ralf Baechle
- Felipe Balbi
- Ohad Ben-Cohen
- Muli Ben-Yehuda
- Jiri Benc
- Arnd Bergmann
- Thomas Bogendoerfer
- Vitaly Bordug
- James Bottomley
- Josh Boyer
- Neil Brown
- Mark Brown
- David Brownell
- Michael Buesch
- Franck Bui-Huu
- Adrian Bunk
- François Cami
- Ralph Campbell
- Luiz Fernando N. Capitulino
- Mauro Carvalho Chehab
- Denis Cheng
- Jonathan Corbet
- Glauber Costa

- Alan Cox
- Magnus Damm
- Ahmed S. Darwish
- Robert P. J. Day
- Hans de Goede
- Arnaldo Carvalho de Melo
- Helge Deller
- Jean Delvare
- Mathieu Desnoyers
- Sven-Thorsten Dietrich
- Alexey Dobriyan
- Daniel Drake
- Alex Dubov
- Randy Dunlap
- Michael Ellerman
- Pekka Enberg
- Jan Engelhardt
- Mark Fasheh
- J. Bruce Fields
- Larry Finger
- Jeremy Fitzhardinge
- Mike Frysinger
- Kumar Gala
- Robin Getz
- Liam Girdwood
- Jan-Benedict Glaw
- Thomas Gleixner
- Brice Goglin
- Cyrill Gorcunov
- Andy Gospodarek
- Thomas Graf
- Krzysztof Halasa
- Harvey Harrison
- Stephen Hemminger

- Michael Hennerich
- Tejun Heo
- Benjamin Herrenschmidt
- Kristian Høgsberg
- Henrique de Moraes Holschuh
- Marcel Holtmann
- Mike Isely
- Takashi Iwai
- Olof Johansson
- Dave Jones
- Jesper Juhl
- Matthias Kaehlcke
- Kenji Kaneshige
- Jan Kara
- Jeremy Kerr
- Russell King
- Olaf Kirch
- Roel Kluin
- Hans-Jürgen Koch
- Auke Kok
- Peter Korsgaard
- Jiri Kosina
- Aaro Koskinen
- Mariusz Kozlowski
- Greg Kroah-Hartman
- Michael Krufky
- Aneesh Kumar
- Clemens Ladisch
- Christoph Lameter
- Gunnar Larisch
- Anders Larsen
- Grant Likely
- John W. Linville
- Yinghai Lu

- Tony Luck
- Pavel Machek
- Matt Mackall
- Paul Mackerras
- Roland McGrath
- Patrick McHardy
- Kyle McMartin
- Paul Menage
- Thierry Merle
- Eric Miao
- Akinobu Mita
- Ingo Molnar
- James Morris
- Andrew Morton
- Paul Mundt
- Oleg Nesterov
- Luca Olivetti
- S.Çağlar Onur
- Pierre Ossman
- Keith Owens
- Venkatesh Pallipadi
- Nick Piggin
- Nicolas Pitre
- Evgeniy Polyakov
- Richard Purdie
- Mike Rapoport
- Sam Ravnborg
- Gerrit Renker
- Stefan Richter
- David Rientjes
- Luis R. Rodriguez
- Stefan Roese
- Francois Romieu
- Rami Rosen

- Stephen Rothwell
- Maciej W. Rozycki
- Mark Salyzyn
- Yoshinori Sato
- Deepak Saxena
- Holger Schurig
- Amit Shah
- Yoshihiro Shimoda
- Sergei Shtylyov
- Kay Sievers
- Sebastian Siewior
- Rik Snel
- Jes Sorensen
- Alexey Starikovskiy
- Alan Stern
- Timur Tabi
- Hirokazu Takata
- Eliezer Tamir
- Eugene Teo
- Doug Thompson
- FUJITA Tomonori
- Dmitry Torokhov
- Marcelo Tosatti
- Steven Toth
- Theodore Tso
- Matthias Urlichs
- Geert Uytterhoeven
- Arjan van de Ven
- Ivo van Doorn
- Rik van Riel
- Wim Van Sebroeck
- Hans Verkuil
- Horst H. von Brand
- Dmitri Vorobiev

- Anton Vorontsov
- Daniel Walker
- Johannes Weiner
- Harald Welte
- Matthew Wilcox
- Dan J. Williams
- Darrick J. Wong
- David Woodhouse
- Chris Wright
- Bryan Wu
- Rafael J. Wysocki
- Herbert Xu
- Vlad Yasevich
- Peter Zijlstra
- Bartlomiej Zolnierkiewicz

其它大多数开发人员感兴趣的社区指南：

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/submitting-drivers.rst

如果想评论或更新本文的内容，请直接联系原文档的维护者。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者:

```
中文版维护者: 李阳  Li Yang <leoyang.li@nxp.com>
中文版翻译者: 李阳  Li Yang <leoyang.li@nxp.com>
中文版校译者: 陈琦 Maggie Chen <chenqi@beyondsoft.com>
              王聪 Wang Cong <xiyou.wangcong@gmail.com>
              张巍 Zhang Wei <wezhang@outlook.com>
```

* 如何向 Linux 内核提交驱动程序

这篇文档将会解释如何向不同的内核源码树提交设备驱动程序。请注意，如果你感兴趣的是显卡驱动程序，你也许应该访问 XFree86 项目 (https://www.xfree86.org/) 和／或 X.org 项目 (https://x.org)。

另请参阅如何让你的改动进入内核 文档。

分配设备号

块设备和字符设备的主设备号与从设备号是由 Linux 命名编号分配权威 LANANA（现在是 Torben Mathiasen）负责分配。申请的网址是 https://www.lanana.org/。即使不准备提交到主流内核的设备驱动也需要在这里分配设备号。有关详细信息，请参阅 Documentation/admin-guide/devices.rst。

如果你使用的不是已经分配的设备号，那么当你提交设备驱动的时候，它将会被强制分配一个新的设备号，即便这个设备号和你之前发给客户的截然不同。

设备驱动的提交对象

Linux 2.0:
此内核源码树不接受新的驱动程序。

Linux 2.2:
此内核源码树不接受新的驱动程序。

Linux 2.4:
如果所属的代码领域在内核的 MAINTAINERS 文件中列有一个总维护者，那么请将驱动程序提交给他。如果此维护者没有回应或者你找不到恰当的维护者，那么请联系 Willy Tarreau <w@1wt.eu>。

Linux 2.6:
除了遵循和 2.4 版内核同样的规则外，你还需要在 linux-kernel 邮件列表上跟踪最新的 API 变化。向 Linux 2.6 内核提交驱动的顶级联系人是 Andrew Morton <akpm@linux-foundation.org>。

决定设备驱动能否被接受的条件

许可：代码必须使用 GNU 通用公开许可证 (GPL) 提交给 Linux，但是
我们并不要求 GPL 是唯一的许可。你或许会希望同时使用多种许可证发布，如果希望驱动程序可以被其他开源社区（比如 BSD）使用。请参考 include/linux/module.h 文件中所列出的可被接受共存的许可。

版权：版权所有者必须同意使用 GPL 许可。最好提交者和版权所有者
是相同个人或实体。否则，必需列出授权使用 GPL 的版权所有人或实体，以备验证之需。

接口：如果你的驱动程序使用现成的接口并且和其他同类的驱动程序行
为相似，而不是去发明无谓的新接口，那么它将会更容易被接受。如果你需要一个 Linux 和 NT 的通用驱动接口，那么请在用户空间实现它。

代码：请使用 Documentation/process/coding-style.rst 中所描述的 Linux 代码风
格。如果你的某些代码段（例如那些与 Windows 驱动程序包共享的代码段）需要使用其他格式，而你却只希望维护一份代码，那么请将它们很好地区分出来，并且注明原因。

可移植性：请注意，指针并不永远是 32 位的，不是所有的计算机都使用小
尾模式 (little endian) 存储数据，不是所有的人都拥有浮点单元，不要随便在你的驱动程序里嵌入 x86 汇编指令。只能在 x86 上运行的驱动程序一般是不受欢迎的。虽然你可能只有 x86 硬件，很难测试驱动程序在其他平台上是否可用，但是确保代码可以被轻松地移植却是很简单的。

清晰度：做到所有人都能修补这个驱动程序将会很有好处，因为这样你将
会直接收到修复的补丁而不是 bug 报告。如果你提交一个试图隐藏硬件工作机理的驱动程序，那么它将会被扔进废纸篓。

电源管理：因为 Linux 正在被很多移动设备和桌面系统使用，所以你的驱
动程序也很有可能被使用在这些设备上。它应该支持最基本的电源管理，即在需要的情况下实现系统级休眠和唤醒要用到的.suspend 和.resume 函数。你应该检查你的驱动程序是否能正确地处理休眠与唤醒，如果实在无法确认，请至少把.suspend 函数定义成返回 -ENOSYS（功能未实现）错误。你还应该尝试确保你的驱动在什么都不干的情况下将耗电降到最低。要获得驱动程序测试的指导，请参阅 Documentation/power/drivers-testing.rst。有关驱动程序电源管理问题相对全面的概述，请参阅 Documentation/driver-api/pm/devices.rst。

管理：如果一个驱动程序的作者还在进行有效的维护，那么通常除了那
些明显正确且不需要任何检查的补丁以外，其他所有的补丁都会被转发给作者。如果你希望成为驱动程序的联系人和更新者，最好在代码注释中写明并且在 MAINTAINERS 文件中加入这个驱动程序的条目。

不影响设备驱动能否被接受的条件

供应商：由硬件供应商来维护驱动程序通常是一件好事。不过，如果源码
树里已经有其他人提供了可稳定工作的驱动程序，那么请不要期望“我是供应商”会成为内核改用你的驱动程序的理由。理想的情况是：供应商与现有驱动程序的作者合作，构建一个统一完美的驱动程序。

作者：驱动程序是由大的 Linux 公司研发还是由你个人编写，并不影
响其是否能被内核接受。没有人对内核源码树享有特权。只要你充分了解内核社区，你就会发现这一点。

资源列表

Linux 内核主源码树：
ftp.??.kernel.org:/pub/linux/kernel/…?? == 你的国家代码，例如“cn”、”us”、”uk”、”fr”等等

Linux 内核邮件列表：
linux-kernel@vger.kernel.org [可通过向 majordomo@vger.kernel.org 发邮件来订阅]

Linux 设备驱动程序，第三版（探讨 2.6.10 版内核）：
https://lwn.net/Kernel/LDD3/ （免费版）

LWN.net:
每周内核开发活动摘要 - https://lwn.net/

2.6 版中 API 的变更：

https://lwn.net/Articles/2.6-kernel-api/

将旧版内核的驱动程序移植到 2.6 版：

https://lwn.net/Articles/driver-porting/

内核新手 (KernelNewbies):
为新的内核开发者提供文档和帮助 https://kernelnewbies.org/

Linux USB 项目：
http://www.linux-usb.org/

写内核驱动的“不要”（Arjan van de Ven 著）：
http://www.fenrus.org/how-to-not-write-a-device-driver-paper.pdf

内核清洁工 (Kernel Janitor):
https://kernelnewbies.org/KernelJanitors

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/submit-checklist.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核补丁提交清单

如果开发人员希望看到他们的内核补丁提交更快地被接受，那么他们应该做一些基本的事情。

这些都是在如何让你的改动进入内核 和其他有关提交 Linux 内核补丁的文档中提供的。

1) 如果使用工具，则包括定义/声明该工具的文件。不要依赖于其他头文件拉入您使用的头文件。
2) 干净的编译：
   a) 使用适用或修改的 `CONFIG` 选项 `=y`、`=m` 和 `=n` 。没有 GCC 警告/错误，没有链接器警告/错误。
   b) 通过 allnoconfig、allmodconfig
   c) 使用 `O=builddir` 时可以成功编译
3) 通过使用本地交叉编译工具或其他一些构建场在多个 CPU 体系结构上构建。
4) PPC64 是一种很好的交叉编译检查体系结构，因为它倾向于对 64 位的数使用无符号长整型。
5) 如下所述Linux 内核代码风格. 检查您的补丁是否为常规样式。在提交（`scripts/checkpatch.pl` ）之前，使用补丁样式检查器检查是否有轻微的冲突。您应该能够处理您的补丁中存在的所有违规行为。

6) 任何新的或修改过的 CONFIG 选项都不会弄脏配置菜单，并默认为关闭，除非它们符合 Documentation/kbuild/kconfig-language.rst 中记录的异常条件，菜单属性：默认值.

7) 所有新的 kconfig 选项都有帮助文本。

8) 已仔细审查了相关的 Kconfig 组合。这很难用测试来纠正——脑力在这里是有回报的。

9) 用 sparse 检查干净。

10) 使用 make checkstack 和 make namespacecheck 并修复他们发现的任何问题。

Note: checkstack 并没有明确指出问题，但是任何一个在堆栈上使用超过 512 字节的函数都可以进行更改。

---

11) 包括 kernel-doc 内核文档以记录全局内核 API。（静态函数不需要，但也可以。）使用 make htmldocs 或 make pdfdocs 检查 kernel-doc 并修复任何问题。

12) 通过以下选项同时启用的测试 CONFIG_PREEMPT, CONFIG_DEBUG_PREEMPT, CONFIG_DEBUG_SLAB, CONFIG_DEBUG_PAGEALLOC, CONFIG_DEBUG_MUTEXES, CONFIG_DEBUG_SPINLOCK, CONFIG_DEBUG_ATOMIC_SLEEP, CONFIG_PROVE_RCU and CONFIG_DEBUG_OBJECTS_RCU_HEAD

13) 已经过构建和运行时测试，包括有或没有 CONFIG_SMP, CONFIG_PREEMPT.

14) 如果补丁程序影响 IO/磁盘等：使用或不使用 CONFIG_LBDAF 进行测试。

15) 所有代码路径都已在启用所有 lockdep 功能的情况下运行。

16) 所有新的/proc 条目都记录在 Documentation/

17) 所有新的内核引导参数都记录在 Documentation/admin-guide/kernel-parameters.rst 中。

18) 所有新的模块参数都记录在 MODULE_PARM_DESC()

19) 所有新的用户空间接口都记录在 Documentation/ABI/ 中。有关详细信息，请参阅 Documentation/ABI/README 。更改用户空间接口的补丁应该抄送 linux-api@vger.kernel.org。

20) 检查是否全部通过 make headers_check 。

21) 已通过至少注入 slab 和 page 分配失败进行检查。请参阅 Documentation/fault-injection/ 如果新代码是实质性的，那么添加子系统特定的故障注入可能是合适的。

22) 新添加的代码已经用 gcc -W 编译（使用 make EXTRA-CFLAGS=-W ）。这将产生大量噪声，但对于查找诸如“警告：有符号和无符号之间的比较”之类的错误很有用。

23) 在它被合并到-mm 补丁集中之后进行测试，以确保它仍然与所有其他排队的补丁以及 VM、VFS 和其他子系统中的各种更改一起工作。

24) 所有内存屏障例如 barrier(), rmb(), wmb() 都需要源代码中的注释来解释它们正在执行的操作及其原因的逻辑。

25) 如果补丁添加了任何 ioctl，那么也要更新 Documentation/userspace-api/ioctl/ioctl-number.rst

26) 如果修改后的源代码依赖或使用与以下 Kconfig 符号相关的任何内核 API 或功能，则在禁用相关 Kconfig 符号和/或 =m （如果该选项可用）的情况下测试以下多个构建 [并非所有这些都同时存在，只是它们的各种/随机组合]：

    CONFIG_SMP, CONFIG_SYSFS, CONFIG_PROC_FS, CONFIG_INPUT, CONFIG_PCI, CONFIG_BLOCK, CONFIG_PM, CONFIG_MAGIC_SYSRQ, CONFIG_NET, CONFIG_INET=n (但是后者伴随 CONFIG_NET=y).

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/stable-api-nonsense.rst

译者:

```
中文版维护者： 钟宇  TripleX Chung <xxx.phy@gmail.com>
中文版翻译者： 钟宇  TripleX Chung <xxx.phy@gmail.com>
中文版校译者： 李阳  Li Yang <leoyang.li@nxp.com>
```

* Linux 内核驱动接口

写作本文档的目的，是为了解释为什么 Linux 既没有二进制内核接口，也没有稳定的内核接口。这里所说的内核接口，是指内核里的接口，而不是内核和用户空间的接口。内核到用户空间的接口，是提供给应用程序使用的系统调用，系统调用在历史上几乎没有过变化，将来也不会有变化。我有一些老应用程序是在 0.9 版本或者更早版本的内核上编译的，在使用 2.6 版本内核的 Linux 发布上依然用得很好。用户和应用程序作者可以将这个接口看成是稳定的。

执行纲要

你也许以为自己想要稳定的内核接口，但是你不清楚你要的实际上不是它。你需要的其实是稳定的驱动程序，而你只有将驱动程序放到公版内核的源代码树里，才有可能达到这个目的。而且这样做还有很多其它好处，正是因为这些好处使得 Linux 能成为强壮，稳定，成熟的操作系统，这也是你最开始选择 Linux 的原因。

入门

只有那些写驱动程序的“怪人”才会担心内核接口的改变，对广大用户来说，既看不到内核接口，也不需要去关心它。

首先，我不打算讨论关于任何非 GPL 许可的内核驱动的法律问题，这些非 GPL 许可的驱动程序包括不公开源代码，隐藏源代码，二进制或者是用源代码包装，或者是其它任何形式的不能以 GPL 许可公开源代码的驱动程序。如果有法律问题，请咨询律师，我只是一个程序员，所以我只打算探讨技术问题（不是小看法律问题，法律问题很实际，并且需要一直关注）。

既然只谈技术问题，我们就有了下面两个主题：二进制内核接口和稳定的内核源代码接口。这两个问题是互相关联的，让我们先解决掉二进制接口的问题。

二进制内核接口

假如我们有一个稳定的内核源代码接口，那么自然而然的，我们就拥有了稳定的二进制接口，是这样的吗？错。让我们看看关于 Linux 内核的几点事实：

- 取决于所用的 C 编译器的版本，不同的内核数据结构里的结构体的对齐方式会有差别，代码中不同函数的表现形式也不一样（函数是不是被 inline 编译取决于编译器行为）。不同的函数的表现形式并不重要，但是数据结构内部的对齐方式很关键。
- 取决于内核的配置选项，不同的选项会让内核的很多东西发生改变：
  - 同一个结构体可能包含不同的成员变量
  - 有的函数可能根本不会被实现（比如编译的时候没有选择 SMP 支持一些锁函数就会被定义成空函数）。
  - 内核使用的内存会以不同的方式对齐，这取决于不同的内核配置选项。
- Linux 可以在很多的不同体系结构的处理器上运行。在某个体系结构上编译好的二进制驱动程序，不可能在另外一个体系结构上正确的运行。

对于一个特定的内核，满足这些条件并不难，使用同一个 C 编译器和同样的内核配置选项来编译驱动程序模块就可以了。这对于给一个特定 Linux 发布的特定版本提供驱动程序，是完全可以满足需求的。但是如果你要给不同发布的不同版本都发布一个驱动程序，就需要在每个发布上用不同的内核设置参数都编译一次内核，这简直跟噩梦一样。而且还要注意到，每个 Linux 发布还提供不同的 Linux 内核，这些内核都针对不同的硬件类型进行了优化（有很多种不同的处理器，还有不同的内核设置选项）。所以每发布一次驱动程序，都需要提供很多不同版本的内核模块。

相信我，如果你真的要采取这种发布方式，一定会慢慢疯掉，我很久以前就有过深刻的教训…

稳定的内核源代码接口

如果有人不将他的内核驱动程序，放入公版内核的源代码树，而又想让驱动程序一直保持在最新的内核中可用，那么这个话题将会变得没完没了。内核开发是持续而且快节奏的，从来都不会慢下来。内核开发人员在当前接口中找到 bug，或者找到更好的实现方式。一旦发现这些，他们就很快会去修改当前的接口。修改接口意味着，函数名可能会改变，结构体可能被扩充或者删减，函数的参数也可能发生改变。一旦接口被修改，内核中使用这些接口的地方需要同时修正，这样才能保证所有的东西继续工作。

举一个例子，内核的 USB 驱动程序接口在 USB 子系统的整个生命周期中，至少经历了三次重写。这些重写解决以下问题：

- 把数据流从同步模式改成非同步模式，这个改动减少了一些驱动程序的复杂度，提高了所有 USB 驱动程序的吞吐率，这样几乎所有的 USB 设备都能以最大速率工作了。
- 修改了 USB 核心代码中为 USB 驱动分配数据包内存的方式，所有的驱动都需要提供更多的参数给 USB 核心，以修正了很多已经被记录在案的死锁。

这和一些封闭源代码的操作系统形成鲜明的对比，在那些操作系统上，不得不额外的维护旧的 USB 接口。这导致了一个可能性，新的开发者依然会不小心使用旧的接口，以不恰当的方式编写代码，进而影响到操作系统的稳定性。在上面的例子中，所有的开发者都同意这些重要的改动，在这样的情况下修改代价很低。如果 Linux 保持一个稳定的内核源代码接口，那么就得创建一个新的接口；旧的，有问题的接口必须一直维护，给 Linux USB 开发者带来额外的工作。既然所有的 Linux USB 驱动的作者都是利用自己的时间工作，那么要求他们去做毫无意义的免费额外工作，是不可能的。安全问题对 Linux 来说十分重要。一个安全问题被发现，就会在短时间内得到修正。在很多情况下，这将导致 Linux 内核中的一些接口被重写，以从根本上避免安全问题。一旦接口被重写，所有使用这些接口的驱动程序，必须同时得到修正，以确定安全问题已经得到修复并且不可能在未来还有同样的安全问题。如果内核内部接口不允许改变，那么就不可能修复这样的安全问题，也不可能确认这样的安全问题以后不会发生。开发者一直在清理内核接口。如果一个接口没有人在使用了，它就会被删除。这样可以确保内核尽可能的小，而且所有潜在的接口都会得到尽可能完整的测试（没有人使用的接口是不可能得到良好的测试的）。

要做什么

如果你写了一个 Linux 内核驱动，但是它还不在 Linux 源代码树里，作为一个开发者，你应该怎么做？为每个发布的每个版本提供一个二进制驱动，那简直是一个噩梦，要跟上永远处于变化之中的内核接口，也是一件辛苦活。很简单，让你的驱动进入内核源代码树（要记得我们在谈论的是以 GPL 许可发行的驱动，如果你的代码不符合 GPL，那么祝你好运，你只能自己解决这个问题了，你这个吸血鬼 < 把 Andrew 和 Linus 对吸血鬼的定义链接到这里 >）。当你的代码加入公版内核源代码树之后，如果一个内核接口改变，你的驱动会直接被修改接口的那个人修改。保证你的驱动永远都可以编译通过，并且一直工作，你几乎不需要做什么事情。

把驱动放到内核源代码树里会有很多的好处：

- 驱动的质量会提升，而维护成本（对原始作者来说）会下降。
- 其他人会给驱动添加新特性。
- 其他人会找到驱动中的 bug 并修复。
- 其他人会在驱动中找到性能优化的机会。
- 当外部的接口的改变需要修改驱动程序的时候，其他人会修改驱动程序
- 不需要联系任何发行商，这个驱动会自动的随着所有的 Linux 发布一起发布。

和别的操作系统相比，Linux 为更多不同的设备提供现成的驱动，而且能在更多不同体系结构的处理器上支持这些设备。这个经过考验的开发模式，必然是错不了的:)

感谢

感谢 Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, and Nishanth Aravamudan 对于本文档早期版本的评审和建议。

英文版维护者：Greg Kroah-Hartman <greg@kroah.com>

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/stable-kernel-rules.rst

如果想评论或更新本文的内容，请直接联系原文档的维护者。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者:

```
中文版维护者:  钟宇  TripleX Chung <xxx.phy@gmail.com>
中文版翻译者:  钟宇  TripleX Chung <xxx.phy@gmail.com>
中文版校译者:
    - 李阳  Li Yang <leoyang.li@nxp.com>
    - Kangkai Yin <e12051@motorola.com>
```

* 所有你想知道的事情 - 关于 linux 稳定版发布

关于 Linux 2.6 稳定版发布，所有你想知道的事情。

关于哪些类型的补丁可以被接收进入稳定版代码树，哪些不可以的规则：

- 必须是显而易见的正确，并且经过测试的。
- 连同上下文，不能大于 100 行。
- 必须只修正一件事情。
- 必须修正了一个给大家带来麻烦的真正的 bug（不是“这也许是一个问题…”那样的东西）。
- 必须修正带来如下后果的问题：编译错误（对被标记为 CONFIG_BROKEN 的例外），内核崩溃，挂起，数据损坏，真正的安全问题，或者一些类似“哦，这不好”的问题。简短的说，就是一些致命的问题。
- 没有“理论上的竞争条件”，除非能给出竞争条件如何被利用的解释。
- 不能存在任何的“琐碎的”修正（拼写修正，去掉多余空格之类的）。

- 必须被相关子系统的维护者接受。
- 必须遵循 Documentation/translations/zh_CN/process/submitting-patches.rst 里的规则。

向稳定版代码树提交补丁的过程：

- 在确认了补丁符合以上的规则后，将补丁发送到 stable@vger.kernel.org。
- 如果补丁被接受到队列里，发送者会收到一个 ACK 回复，如果没有被接受，收到的是 NAK 回复。回复需要几天的时间，这取决于开发者的时间安排。
- 被接受的补丁会被加到稳定版本队列里，等待其他开发者的审查。
- 安全方面的补丁不要发到这个列表，应该发送到 security@kernel.org。

审查周期：

- 当稳定版的维护者决定开始一个审查周期，补丁将被发送到审查委员会，以及被补丁影响的领域的维护者（除非提交者就是该领域的维护者）并且抄送到 linux-kernel 邮件列表。
- 审查委员会有 48 小时的时间，用来决定给该补丁回复 ACK 还是 NAK。
- 如果委员会中有成员拒绝这个补丁，或者 linux-kernel 列表上有人反对这个补丁，并提出维护者和审查委员会之前没有意识到的问题，补丁会从队列中丢弃。
- 在审查周期结束的时候，那些得到 ACK 回应的补丁将会被加入到最新的稳定版发布中，一个新的稳定版发布就此产生。
- 安全性补丁将从内核安全小组那里直接接收到稳定版代码树中，而不是通过通常的审查周期。请联系内核安全小组以获得关于这个过程的更多细节。

审查委员会：

- 由一些自愿承担这项任务的内核开发者，和几个非志愿的组成。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/management-style.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* Linux 内核管理风格

这是一个简短的文档，描述了 Linux 内核首选的（或胡编的，取决于您问谁）管理风格。它的目的是在某种程度上参照 process/coding-style.rst 主要是为了避免反复回答[1] 相同（或类似）的问题。

管理风格是非常个人化的，比简单的编码风格规则更难以量化，因此本文档可能与实际情况有关，也可能与实际情况无关。起初它是一个玩笑，但这并不意味着它可能不是真的。你得自己决定。

顺便说一句，在谈到“核心管理者”时，主要是技术负责人，而不是在公司内部进行传统管理的人。如果你签署了采购订单或者对你的团队的预算有任何了解，你几乎肯定不是一个核心管理者。这些建议可能适用于您，也可能不适用于您。

首先，我建议你购买“高效人的七个习惯”，而不是阅读它。烧了它，这是一个伟大的象征性姿态。

不管怎样，这里是：

1）决策

每个人都认为管理者做决定，而且决策很重要。决定越大越痛苦，管理者就必须越高级。这很明显，但事实并非如此。

游戏的名字是 **避免**做出决定。尤其是，如果有人告诉你“选择（a）或（b），我们真的需要你来做决定”，你就是陷入麻烦的管理者。你管理的人比你更了解细节，所以如果他们来找你做技术决策，你完蛋了。你显然没有能力为他们做这个决定。

（推论：如果你管理的人不比你更了解细节，你也会被搞砸，尽管原因完全不同。也就是说，你的工作是错的，他们应该管理你的才智）

所以游戏的名字是 **避免**做出决定，至少是那些大而痛苦的决定。做一些小的和非结果性的决定是很好的，并且使您看起来好像知道自己在做什么，所以内核管理者需要做的是将那些大的和痛苦的决定变成那些没有人真正关心的小事情。

这有助于认识到一个大的决定和一个小的决定之间的关键区别是你是否可以在事后修正你的决定。任何决定都可以通过始终确保如果你错了（而且你一定会错），你以后总是可以通过回溯来弥补损失。突然间，你就要做两个无关紧要的决定，一个是错误的，另一个是正确的。

人们甚至会认为这是真正的领导能力（咳，胡说，咳）。

因此，避免重大决策的关键在于避免做那些无法挽回的事情。不要被引导到一个你无法逃离的角落。走投无路的老鼠可能很危险——走投无路的管理者真可怜。

事实证明，由于没有人会愚蠢到让内核管理者承担巨大的财政责任，所以通常很容易回溯。既然你不可能浪费掉你无法偿还的巨额资金，你唯一可以回溯的就是技术决策，而回溯很容易：只要告诉大家你是个不称职的傻瓜，说对不起，然后撤销你去年让别人所做的毫无价值的工作。突然间，你一年前做的决定不在是一个重大的决定，因为它很容易被推翻。

事实证明，有些人对接受这种方法有困难，原因有两个：

- 承认你是个白痴比看起来更难。我们都喜欢保持形象，在公共场合说你错了有时确实很难。
- 如果有人告诉你，你去年所做的工作终究是不值得的，那么对那些可怜的低级工程师来说也是很困难的，虽然实际的 **工作**很容易删除，但你可能已经不可挽回地失去了工程师的

[1] 本文件并不是通过回答问题，而是通过让提问者痛苦地明白，我们不知道答案是什么。

信任。记住：“不可撤销”是我们一开始就试图避免的，而你的决定终究是一个重大的决定。

令人欣慰的是，这两个原因都可以通过预先承认你没有任何线索，提前告诉人们你的决定完全是初步的，而且可能是错误的事情来有效地缓解。你应该始终保留改变主意的权利，并让人们**意识**到这一点。当你 **还没有**做过真正愚蠢的事情的时候，承认自己是愚蠢的要容易得多。

然后，当它真的被证明是愚蠢的时候，人们就转动他们的眼珠说“哎呀，下次不要了”。

这种对不称职的先发制人的承认，也可能使真正做这项工作的人也会三思是否值得做。毕竟，如果他们不确定这是否是一个好主意，你肯定不应该通过向他们保证他们所做的工作将会进入（内核）鼓励他们。在他们开始一项巨大的努力之前，至少让他们三思而后行。

记住：他们最好比你更了解细节，而且他们通常认为他们对每件事都有答案。作为一个管理者，你能做的最好的事情不是灌输自信，而是对他们所做的事情进行健康的批判性思考。

顺便说一句，另一种避免做出决定的方法是看起来很可怜的抱怨“我们不能两者兼得吗?”相信我，它是有效的。如果不清楚哪种方法更好，他们最终会弄清楚的。最终的答案可能是两个团队都会因为这种情况而感到沮丧，以至于他们放弃了。

这听起来像是一个失败，但这通常是一个迹象，表明两个项目都有问题，而参与其中的人不能做决定的原因是他们都是错误的。你最终会闻到玫瑰的味道，你避免了另一个你本可以搞砸的决定。

2）人

大多数人都是白痴，做一名管理者意味着你必须处理好这件事，也许更重要的是，**他们**必须处理好你。

事实证明，虽然很容易纠正技术错误，但不容易纠正人格障碍。你只能和他们的和你的（人格障碍）共处。

但是，为了做好作为内核管理者的准备，最好记住不要烧掉任何桥梁，不要轰炸任何无辜的村民，也不要疏远太多的内核开发人员。事实证明，疏远人是相当容易的，而亲近一个疏远的人是很难的。因此，“疏远”立即属于“不可逆”的范畴，并根据*1）决策* 成为绝不可以做的事情。

这里只有几个简单的规则：

(1) 不要叫人笨蛋（至少不要在公共场合）

(2) 学习如何在忘记规则 (1) 时道歉

问题在于 #1 很容易去做，因为你可以用数百万种不同的方式说“你是一个笨蛋”[2] 有时甚至没有意识到，而且几乎总是带着一种白热化的信念，认为你是对的。

你越确信自己是对的（让我们面对现实吧，你可以把几乎所有人都称为坏人，而且你经常是对的），事后道歉就越难。

要解决此问题，您实际上只有两个选项：

- 非常擅长道歉
- 把“爱”均匀地散开，没有人会真正感觉到自己被不公平地瞄准了。让它有足够的创造性，他们甚至可能会觉得好笑。

选择永远保持礼貌是不存在的。没有人会相信一个如此明显地隐藏了他们真实性格的人。

[2] 保罗·西蒙演唱了“离开爱人的 50 种方法”，因为坦率地说，“告诉开发者他们是 D*CKHEAD”的 100 万种方法都无法确认。但我确信他已经这么想了。

3）人 2 - 好人

虽然大多数人都是白痴，但不幸的是，据此推论你也是白痴，尽管我们都自我感觉良好，我们比普通人更好（让我们面对现实吧，没有人相信他们是普通人或低于普通人），我们也应该承认我们不是最锋利的刀，而且会有其他人比你更不像白痴。

有些人对聪明人反应不好。其他人利用它们。

作为内核维护人员，确保您在第二组中。接受他们，因为他们会让你的工作更容易。特别是，他们能够为你做决定，这就是游戏的全部内容。

所以当你发现一个比你聪明的人时，就顺其自然吧。你的管理职责在很大程度上变成了“听起来像是个好主意——去尝试吧”，或者“听起来不错，但是 XXX 呢?”“。第二个版本尤其是一个很好的方法，要么学习一些关于“XXX”的新东西，要么通过指出一些聪明人没有想到的东西来显得更具管理性。无论哪种情况，你都会赢。

要注意的一件事是认识到一个领域的伟大不一定会转化为其他领域。所以你可能会向特定的方向刺激人们，但让我们面对现实吧，他们可能擅长他们所做的事情，而且对其他事情都很差劲。好消息是，人们往往会自然而然地重拾他们擅长的东西，所以当你向某个方向刺激他们时，你并不是在做不可逆转的事情，只是不要用力推。

4）责备

事情会出问题的，人们希望去责备人。贴标签，你就是受责备的人。

事实上，接受责备并不难，尤其是当人们意识到这不 **全是**你的过错时。这让我们找到了承担责任的最佳方式：为别人承担这件事。你会感觉很好，他们会感觉很好，没有受到指责. 那谁，失去了他们的全部 36GB 色情收藏的人，因为你的无能将勉强承认，你至少没有试图逃避责任。

然后让真正搞砸了的开发人员（如果你能找到他们）私下知道他们搞砸了。不仅是为了将来可以避免，而且为了让他们知道他们欠你一个人情。而且，也许更重要的是，他们也可能是能够解决问题的人。因为，让我们面对现实吧，肯定不是你。

承担责任也是你首先成为管理者的原因。这是让人们信任你，让你获得潜在的荣耀的一部分，因为你就是那个会说“我搞砸了”的人。如果你已经遵循了以前的规则，你现在已经很擅长说了。

5）应避免的事情

有一件事人们甚至比被称为“笨蛋”更讨厌，那就是在一个神圣的声音中被称为“笨蛋”。第一个你可以道歉，第二个你不会真正得到机会。即使你做得很好，他们也可能不再倾听。

我们都认为自己比别人强，这意味着当别人装腔作势时，这会让我们很恼火。你也许在道德和智力上比你周围的每个人都优越，但不要试图太明显，除非你真的打算激怒某人[3]

同样，不要对事情太客气或太微妙。礼貌很容易落得落花流水，把问题隐藏起来，正如他们所说，“在互联网上，没人能听到你的含蓄。”用一个钝器把这一点锤进去，因为你不能真的依靠别人来获得你的观点。

一些幽默可以帮助缓和直率和道德化。过度到荒谬的地步，可以灌输一个观点，而不会让接受者感到痛苦，他们只是认为你是愚蠢的。因此，它可以帮助我们摆脱对批评的个人心理障碍。

[3] 提示：与你的工作没有直接关系的网络新闻组是消除你对他人不满的好方法。偶尔写些侮辱性的帖子，打个喷嚏，让你的情绪得到净化。别把牢骚带回家

6）为什么是我?

既然你的主要责任似乎是为别人的错误承担责任，并且让别人痛苦地明白你是不称职的，那么显而易见的问题之一就变成了为什么首先要这样做。

首先，虽然你可能会或可能不会听到十几岁女孩（或男孩，让我们不要在这里评判或性别歧视）敲你的更衣室门，你会得到一个巨大的个人成就感为“负责”。别介意你真的在领导别人，你要跟上别人，尽可能快地追赶他们。每个人都会认为你是负责人。

如果你可以做到这个，这是个伟大的工作!

> **Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/embargoed-hardware-issues.rst

Translator
Alex Shi <alex.shi@linux.alibaba.com>

* 被限制的硬件问题

范围

导致安全问题的硬件问题与只影响 Linux 内核的纯软件错误是不同的安全错误类别。

必须区别对待诸如熔毁 (Meltdown)、Spectre、L1TF 等硬件问题，因为它们通常会影响所有操作系统（“OS”），因此需要在不同的 OS 供应商、发行版、硬件供应商和其他各方之间进行协调。对于某些问题，软件缓解可能依赖于微码或固件更新，这需要进一步的协调。

接触

Linux 内核硬件安全小组独立于普通的 Linux 内核安全小组。

该小组只负责协调被限制的硬件安全问题。Linux 内核中纯软件安全漏洞的报告不由该小组处理，报告者将被引导至常规 Linux 内核安全小组 (Documentation/admin-guide/) 联系。

可以通过电子邮件 <hardware-security@kernel.org> 与小组联系。这是一份私密的安全官名单，他们将帮助您根据我们的文档化流程协调问题。

邮件列表是加密的，发送到列表的电子邮件可以通过 PGP 或 S/MIME 加密，并且必须使用报告者的 PGP 密钥或 S/MIME 证书签名。该列表的 PGP 密钥和 S/MIME 证书可从 https://www.kernel.org/…. 获得。

虽然硬件安全问题通常由受影响的硬件供应商处理，但我们欢迎发现潜在硬件缺陷的研究人员或个人与我们联系。

硬件安全官

目前的硬件安全官小组:

- Linus Torvalds（Linux 基金会院士）
- Greg Kroah Hartman（Linux 基金会院士）
- Thomas Gleixner（Linux 基金会院士）

邮件列表的操作

处理流程中使用的加密邮件列表托管在 Linux Foundation 的 IT 基础设施上。通过提供这项服务，Linux 基金会的 IT 基础设施安全总监在技术上有能力访问被限制的信息，但根据他的雇佣合同，他必须保密。Linux 基金会的 IT 基础设施安全总监还负责 kernel.org 基础设施。

Linux 基金会目前的 IT 基础设施安全总监是 Konstantin Ryabitsev。

保密协议

Linux 内核硬件安全小组不是正式的机构，因此无法签订任何保密协议。核心社区意识到这些问题的敏感性，并提供了一份谅解备忘录。

谅解备忘录

Linux 内核社区深刻理解在不同操作系统供应商、发行商、硬件供应商和其他各方之间进行协调时，保持硬件安全问题处于限制状态的要求。

Linux 内核社区在过去已经成功地处理了硬件安全问题，并且有必要的机制允许在限制限制下进行符合社区的开发。

Linux 内核社区有一个专门的硬件安全小组负责初始联系，并监督在限制规则下处理此类问题的过程。

硬件安全小组确定开发人员（领域专家），他们将组成特定问题的初始响应小组。最初的响应小组可以引入更多的开发人员（领域专家）以最佳的技术方式解决这个问题。

所有相关开发商承诺遵守限制规定，并对收到的信息保密。违反承诺将导致立即从当前问题中排除，并从所有相关邮件列表中删除。此外，硬件安全小组还将把违反者排除在未来的问题之外。这一后果的影响在我们社区是一种非常有效的威慑。如果发生违规情况，硬件安全小组将立即通知相关方。如果您或任何人发现潜在的违规行为，请立即向硬件安全人员报告。

流程

由于 Linux 内核开发的全球分布式特性，面对面的会议几乎不可能解决硬件安全问题。由于时区和其他因素，电话会议很难协调，只能在绝对必要时使用。加密电子邮件已被证明是解决此类问题的最有效和最安全的通信方法。

开始披露

披露内容首先通过电子邮件联系 Linux 内核硬件安全小组。此初始联系人应包含问题的描述和任何已知受影响硬件的列表。如果您的组织制造或分发受影响的硬件，我们建议您也考虑哪些其他硬件可能会受到影响。

硬件安全小组将提供一个特定于事件的加密邮件列表，用于与报告者进行初步讨论、进一步披露和协调。

硬件安全小组将向披露方提供一份开发人员（领域专家）名单，在与开发人员确认他们将遵守本谅解备忘录和文件化流程后，应首先告知开发人员有关该问题的信息。这些开发人员组成初始响应小组，并在初始接触后负责处理问题。硬件安全小组支持响应小组，但不一定参与缓解开发过程。

虽然个别开发人员可能通过其雇主受到保密协议的保护，但他们不能以 Linux 内核开发人员的身份签订个别保密协议。但是，他们将同意遵守这一书面程序和谅解备忘录。

披露方应提供已经或应该被告知该问题的所有其他实体的联系人名单。这有几个目的:

- 披露的实体列表允许跨行业通信，例如其他操作系统供应商、硬件供应商等。
- 可联系已披露的实体，指定应参与缓解措施开发的专家。
- 如果需要处理某一问题的专家受雇于某一上市实体或某一上市实体的成员，则响应小组可要求该实体披露该专家。这确保专家也是实体反应小组的一部分。

披露

披露方通过特定的加密邮件列表向初始响应小组提供详细信息。

根据我们的经验，这些问题的技术文档通常是一个足够的起点，最好通过电子邮件进行进一步的技术澄清。

缓解开发

初始响应小组设置加密邮件列表，或在适当的情况下重新修改现有邮件列表。

使用邮件列表接近于正常的 Linux 开发过程，并且在过去已经成功地用于为各种硬件安全问题开发缓解措施。

邮件列表的操作方式与正常的 Linux 开发相同。发布、讨论和审查修补程序，如果同意，则应用于非公共 git 存储库，参与开发人员只能通过安全连接访问该存储库。存储库包含针对主线内核的主开发分支，并根据需要为稳定的内核版本提供向后移植分支。

最初的响应小组将根据需要从 Linux 内核开发人员社区中确定更多的专家。引进专家可以在开发过程中的任何时候发生，需要及时处理。

如果专家受雇于披露方提供的披露清单上的实体或其成员，则相关实体将要求其参与。

否则，披露方将被告知专家参与的情况。谅解备忘录涵盖了专家，要求披露方确认参与。如果披露方有令人信服的理由提出异议，则必须在五个工作日内提出异议，并立即与事件小组解决。如果披露方在五个工作日内未作出回应，则视为默许。

在确认或解决异议后，专家由事件小组披露，并进入开发过程。

协调发布

有关各方将协商限制结束的日期和时间。此时，准备好的缓解措施集成到相关的内核树中并发布。

虽然我们理解硬件安全问题需要协调限制时间，但限制时间应限制在所有有关各方制定、测试和准备缓解措施所需的最短时间内。人为地延长限制时间以满足会议讨论日期或其他非技术原因，会给相关的开发人员和响应小组带来了更多的工作和负担，因为补丁需要保持最新，以便跟踪正在进行的上游内核开发，这可能会造成冲突的更改。

CVE 分配

硬件安全小组和初始响应小组都不分配 CVE，开发过程也不需要 CVE。如果 CVE 是由披露方提供的，则可用于文档中。

流程专使

为了协助这一进程，我们在各组织设立了专使，他们可以回答有关报告流程和进一步处理的问题或提供指导。专使不参与特定问题的披露，除非响应小组或相关披露方提出要求。现任专使名单:

ARM	
AMD	Tom Lendacky <tom.lendacky@amd.com>
IBM	
Intel	Tony Luck <tony.luck@intel.com>
Qualcomm	Trilok Soni <tsoni@codeaurora.org>
Microsoft	Sasha Levin <sashal@kernel.org>
VMware	
Xen	Andrew Cooper <andrew.cooper3@citrix.com>
Canonical	John Johansen <john.johansen@canonical.com>
Debian	Ben Hutchings <ben@decadent.org.uk>
Oracle	Konrad Rzeszutek Wilk <konrad.wilk@oracle.com>
Red Hat	Josh Poimboeuf <jpoimboe@redhat.com>
SUSE	Jiri Kosina <jkosina@suse.cz>
Amazon	
Google	Kees Cook <keescook@chromium.org>

如果要将您的组织添加到专使名单中，请与硬件安全小组联系。被提名的专使必须完全理解和支持我们的过程，并且在 Linux 内核社区中很容易联系。

加密邮件列表

我们使用加密邮件列表进行通信。这些列表的工作原理是，发送到列表的电子邮件使用列表的 PGP 密钥或列表的/MIME 证书进行加密。邮件列表软件对电子邮件进行解密，并使用订阅者的 PGP 密钥或 S/MIME 证书为每个订阅者分别对其进行重新加密。有关邮件列表软件和用于确保列表安全和数据保护的设置的详细信息，请访问：https://www.kernel.org/….

关键点

初次接触见接触. 对于特定于事件的邮件列表，密钥和 S/MIME 证书通过特定列表发送的电子邮件传递给订阅者。

订阅事件特定列表

订阅由响应小组处理。希望参与通信的披露方将潜在订户的列表发送给响应组，以便响应组可以验证订阅请求。

每个订户都需要通过电子邮件向响应小组发送订阅请求。电子邮件必须使用订阅服务器的 PGP 密钥或 S/MIME 证书签名。如果使用 PGP 密钥，则必须从公钥服务器获得该密钥，并且理想情况下该密钥连接到 Linux 内核的 PGP 信任网。另请参见：https://www.kernel.org/signature.html.

响应小组验证订阅者，并将订阅者添加到列表中。订阅后，订阅者将收到来自邮件列表的电子邮件，该邮件列表使用列表的 PGP 密钥或列表的/MIME 证书签名。订阅者的电子邮件客户端可以从签名中提取 PGP 密钥或 S/MIME 证书，以便订阅者可以向列表发送加密电子邮件。

这些是一些总体技术指南，由于缺乏更好的地方，现在已经放在这里

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/magic-number.rst

如果想评论或更新本文的内容，请直接发信到 LKML。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者:

```
中文版维护者:  贾威威 Jia Wei Wei <harryxiyou@gmail.com>
中文版翻译者:  贾威威 Jia Wei Wei <harryxiyou@gmail.com>
中文版校译者:  贾威威 Jia Wei Wei <harryxiyou@gmail.com>
```

* Linux 魔术数

这个文件是有关当前使用的魔术值注册表。当你给一个结构添加了一个魔术值，你也应该把这个魔术值添加到这个文件，因为我们最好把用于各种结构的魔术值统一起来。

使用魔术值来保护内核数据结构是一个非常好的主意。这就允许你在运行期检查 (a) 一个结构是否已经被攻击，或者 (b) 你已经给一个例行程序通过了一个错误的结构。后一种情况特别地有用—特别是当你通过一个空指针指向结构体的时候。tty 源码，例如，经常通过特定驱动使用这种方法并且反复地排列特定方面的结构。

使用魔术值的方法是在结构的开始处声明的，如下:

```
struct tty_ldisc {
        int     magic;
        ...
};
```

当你以后给内核添加增强功能的时候，请遵守这条规则！这样就会节省数不清的调试时间，特别是一些古怪的情况，例如，数组超出范围并且重新写了超出部分。遵守这个规则， 这些情况可以被快速地，安全地避免。

Theodore Ts’o
31 Mar 94

给当前的 Linux 2.1.55 添加魔术表。

Michael Chastain <mailto:mec@shout.net> 22 Sep 1997

现在应该最新的 Linux 2.1.112. 因为在特性冻结期间，不能在 2.2.x 前改变任何东西。这些条目被数域所排序。

Krzysztof G.Baranowski <mailto: kgb@knm.org.pl> 29 Jul 1998

更新魔术表到 Linux 2.5.45。刚好越过特性冻结，但是有可能还会有一些新的魔术值在 2.6.x 之前融入到内核中。

Petr Baudis <pasky@ucw.cz> 03 Nov 2002

更新魔术表到 Linux 2.5.74。

Fabian Frederick <ffrederick@users.sourceforge.net> 09 Jul 2003

魔术数名	数字	结构	文件
PG_MAGIC	‘P’	pg_{read,write}_hdr	include/linu
CMAGIC	0x0111	user	include/linu
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/
HDLC_MAGIC	0x239e	n_hdlc	drivers/char
APM_BIOS_MAGIC	0x4101	apm_user	arch/x86/ker
CYCLADES_MAGIC	0x4359	cyclades_port	include/linu
DB_MAGIC	0x4442	fc_info	drivers/net/
DL_MAGIC	0x444d	fc_info	drivers/net/
FASYNC_MAGIC	0x4601	fasync_struct	include/linu
FF_MAGIC	0x4646	fc_info	drivers/net/
ISICOM_MAGIC	0x4d54	isi_port	include/linu
PTY_MAGIC	0x5001		drivers/char

Table 1 – continued from previous page

魔术数名	数字	结构	文件
PPP_MAGIC	0x5002	ppp	include/linu
SSTATE_MAGIC	0x5302	serial_state	include/linu
SLIP_MAGIC	0x5302	slip	drivers/net/
STRIP_MAGIC	0x5303	strip	drivers/net/
X25_ASY_MAGIC	0x5303	x25_asy	drivers/net/
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/
AX25_MAGIC	0x5316	ax_disp	drivers/net/
TTY_MAGIC	0x5401	tty_struct	include/linu
MGSL_MAGIC	0x5401	mgsl_info	drivers/char
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linu
MGSLPC_MAGIC	0x5402	mgslpc_info	drivers/char
TTY_LDISC_MAGIC	0x5403	tty_ldisc	include/linu
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/
FULL_DUPLEX_MAGIC	0x6969		drivers/net/
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/
RFCOMM_TTY_MAGIC	0x6d02		net/bluetoot
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linu
RPORT_MAGIC	0x00525001	r_port	drivers/char
LSEMAGIC	0x05091998	lse	drivers/fc4/
GDTIOCTL_MAGIC	0x06030f07	gdth_iowr_str	drivers/scsi
RIEBL_MAGIC	0x09051990		drivers/net/
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linu
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn
ECP_MAGIC	0x21504345	cdkecpsig	include/linu
LSOMAGIC	0x27091997	lso	drivers/fc4/
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linu
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linu
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/driv
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i8
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/tr
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga
GDA_MAGIC	0x58464552	gda	arch/mips/in
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linu
PCXX_MAGIC	0x5c6df104	channel	drivers/char

Table 1 – continued from previous page

魔术数名	数字	结构	文件
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/in
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i8
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/tr
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/ma
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/
SLOT_MAGIC	0x67267321	slot	drivers/hotp
SLOT_MAGIC	0x67267322	slot	drivers/hotp
LO_MAGIC	0x68797548	nbd_device	include/linu
OPROFILE_MAGIC	0x6f70726f	super_block	drivers/oprc
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/ma
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/m
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/m
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linu
ENI155_MAGIC	0xa54b872d	midway_eprom	drivers/atm/
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda
DPMEM_MAGIC	0xc0ffee11	gdt_pci_sram	drivers/scsi
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sc
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/in

请注意，在声音记忆管理中仍然有一些特殊的为每个驱动定义的魔术值。查看 include/sound/sndmagic.h 来获取他们完整的列表信息。很多 OSS 声音驱动拥有自己从声卡 PCI ID 构建的魔术值-他们也没有被列在这里。

IrDA 子系统也使用了大量的自己的魔术值，查看 include/net/irda/irda.h 来获取他们完整的信息。

HFS 是另外一个比较大的使用魔术值的文件系统-你可以在 fs/hfs/hfs.h 中找到他们。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/process/volatile-considered-harmful.rst

如果想评论或更新本文的内容，请直接联系原文档的维护者。如果你使用英文交流有困难的话，也可以向中文版维护者求助。如果本翻译更新不及时或者翻译存在问题，请联系中文版维护者:

英文版维护者： Jonathan Corbet <corbet@lwn.net>
中文版维护者： 伍鹏 Bryan Wu <bryan.wu@analog.com>
中文版翻译者： 伍鹏 Bryan Wu <bryan.wu@analog.com>
中文版校译者： 张汉辉 Eugene Teo <eugeneteo@kernel.sg>
杨瑞 Dave Young <hidave.darkstar@gmail.com>
时奎亮 Alex Shi <alex.shi@linux.alibaba.com>

* 为什么不应该使用“volatile”类型

C 程序员通常认为 volatile 表示某个变量可以在当前执行的线程之外被改变；因此，在内核中用到共享数据结构时，常常会有 C 程序员喜欢使用 volatile 这类变量。换句话说，他们经常会把 volatile 类型看成某种简易的原子变量，当然它们不是。在内核中使用 volatile 几乎总是错误的；本文档将解释为什么这样。

理解 volatile 的关键是知道它的目的是用来消除优化，实际上很少有人真正需要这样的应用。在内核中，程序员必须防止意外的并发访问破坏共享的数据结构，这其实是一个完全不同的任务。用来防止意外并发访问的保护措施，可以更加高效的避免大多数优化相关的问题。

像 volatile 一样，内核提供了很多原语来保证并发访问时的数据安全（自旋锁, 互斥量, 内存屏障等等），同样可以防止意外的优化。如果可以正确使用这些内核原语，那么就没有必要再使用 volatile。如果仍然必须使用 volatile，那么几乎可以肯定在代码的某处有一个 bug。在正确设计的内核代码中，volatile 能带来的仅仅是使事情变慢。

思考一下这段典型的内核代码:

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

如果所有的代码都遵循加锁规则，当持有 the_lock 的时候，不可能意外的改变 shared_data 的值。任何可能访问该数据的其他代码都会在这个锁上等待。自旋锁原语跟内存屏障一样——它们显式的用来书写成这样——意味着数据访问不会跨越它们而被优化。所以本来编译器认为它知道在 shared_data 里面将有什么，但是因为 spin_lock() 调用跟内存屏障一样，会强制编译器忘记它所知道的一切。那么在访问这些数据时不会有优化的问题。

如果 shared_data 被声名为 volatile，锁操作将仍然是必须的。就算我们知道没有其他人正在使用它，编译器也将被阻止优化对临界区内 shared_data 的访问。在锁有效的同时，shared_data 不是 volatile 的。在处理共享数据的时候，适当的锁操作可以不再需要 volatile——并且是有潜在危害的。

volatile 的存储类型最初是为那些内存映射的 I/O 寄存器而定义。在内核里，寄存器访问也应该被锁保护，但是人们也不希望编译器“优化”临界区内的寄存器访问。内核里 I/O 的内存访问是通过访问函数完成的；不赞成通过指针对 I/O 内存的直接访问，并且不是在所有体系架构上都能工作。那些访问函数正是为了防止意外优化而写的，因此，再说一次，volatile 类型不是必需的。

另一种引起用户可能使用 volatile 的情况是当处理器正忙着等待一个变量的值。正确执行一个忙等待的方法是:

```
while (my_variable != what_i_want)
    cpu_relax();
```

cpu_relax() 调用会降低 CPU 的能量消耗或者让位于超线程双处理器；它也作为内存屏障一样出现，所以，再一次，volatile 不是必需的。当然，忙等待一开始就是一种反常规的做法。

在内核中，一些稀少的情况下 volatile 仍然是有意义的：

- 在一些体系架构的系统上，允许直接的 I/0 内存访问，那么前面提到的访问函数可以使用 volatile。基本上，每一个访问函数调用它自己都是一个小的临界区域并且保证了按照程序员期望的那样发生访问操作。
- 某些会改变内存的内联汇编代码虽然没有什么其他明显的附作用，但是有被 GCC 删除的可能性。在汇编声明中加上 volatile 关键字可以防止这种删除操作。
- Jiffies 变量是一种特殊情况，虽然每次引用它的时候都可以有不同的值，但读 jiffies 变量时不需要任何特殊的加锁保护。所以 jiffies 变量可以使用 volatile，但是不赞成其他跟 jiffies 相同类型变量使用 volatile。Jiffies 被认为是一种“愚蠢的遗留物”（Linus 的话）因为解决这个问题比保持现状要麻烦的多。
- 由于某些 I/0 设备可能会修改连续一致的内存, 所以有时, 指向连续一致内存的数据结构的指针需要正确的使用 volatile。网络适配器使用的环状缓存区正是这类情形的一个例子，其中适配器用改变指针来表示哪些描述符已经处理过了。

对于大多代码，上述几种可以使用 volatile 的情况都不适用。所以，使用 volatile 是一种 bug 并且需要对这样的代码额外仔细检查。那些试图使用 volatile 的开发人员需要退一步想想他们真正想实现的是什么。

非常欢迎删除 volatile 变量的补丁－只要证明这些补丁完整的考虑了并发问题。

注释

[1] https://lwn.net/Articles/233481/ [2] https://lwn.net/Articles/233482/

致谢

最初由 Randy Dunlap 推动并作初步研究由 Jonathan Corbet 撰写参考 Satyam Sharma，Johannes Stezenbach，Jesper Juhl，Heikki Orsila，H. Peter Anvin，Philipp Hahn 和 Stefan Richter 的意见改善了本档。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/filesystems/index.rst

Translator
Wang Wenhu <wenhu.wang@vivo.com>

* Linux Kernel 中的文件系统

这份正在开发的手册或许在未来某个辉煌的日子里以易懂的形式将 Linux 虚拟文件系统（VFS）层以及基于其上的各种文件系统如何工作呈现给大家。当前可以看到下面的内容。

* 文件系统

文件系统实现文档。

> **Warning:** 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/filesystems/virtiofs.rst

译者

```
中文版维护者:  王文虎 Wang Wenhu <wenhu.wang@vivo.com>
中文版翻译者:  王文虎 Wang Wenhu <wenhu.wang@vivo.com>
中文版校译者:  王文虎 Wang Wenhu <wenhu.wang@vivo.com>
```

virtiofs: virtio-fs 主机 <-> 客机共享文件系统

- Copyright (C) 2020 Vivo Communication Technology Co. Ltd.

介绍

Linux 的 virtiofs 文件系统实现了一个半虚拟化 VIRTIO 类型“virtio-fs”设备的驱动，通过该类型设备实现客机 <-> 主机文件系统共享。它允许客机挂载一个已经导出到主机的目录。

客机通常需要访问主机或者远程系统上的文件。使用场景包括：在新客机安装时让文件对其可见；从主机上的根文件系统启动；对无状态或临时客机提供持久存储和在客机之间共享目录。

尽管在某些任务可能通过使用已有的网络文件系统完成，但是却需要非常难以自动化的配置步骤，且将存储网络暴露给客机。而 virtio-fs 设备通过提供不经过网络的文件系统访问文件的设计方式解决了这些问题。

另外，virto-fs 设备发挥了主客机共存的优点提高了性能，并且提供了网络文件系统所不具备的一些语义功能。

用法

以``myfs``标签将文件系统挂载到``/mnt``:

```
guest# mount -t virtiofs myfs /mnt
```

请查阅 https://virtio-fs.gitlab.io/ 了解配置 QEMU 和 virtiofsd 守护程序的详细信息。

内幕

由于 virtio-fs 设备将 FUSE 协议用于文件系统请求，因此 Linux 的 virtiofs 文件系统与 FUSE 文件系统客户端紧密集成在一起。客机充当 FUSE 客户端而主机充当 FUSE 服务器，内核与用户空间之间的/dev/fuse 接口由 virtio-fs 设备接口代替。

FUSE 请求被置于虚拟队列中由主机处理。主机填充缓冲区中的响应部分，而客机处理请求的完成部分。

将/dev/fuse 映射到虚拟队列需要解决/dev/fuse 和虚拟队列之间语义上的差异。每次读取/dev/fuse 设备时，FUSE 客户端都可以选择要传输的请求，从而可以使某些请求优先于其他请求。虚拟队列有其队列语义，无法更改已入队请求的顺序。在虚拟队列已满的情况下尤其关键，因为此时不可能加入高优先级的请求。为了解决此差异，virtio-fs 设备采用“hiprio”（高优先级）虚拟队列，专门用于有别于普通请求的高优先级请求。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
../../../filesystems/debugfs

Debugfs

译者

```
中文版维护者： 罗楚成 Chucheng Luo <luochucheng@vivo.com>
中文版翻译者： 罗楚成 Chucheng Luo <luochucheng@vivo.com>
中文版校译者： 罗楚成 Chucheng Luo <luochucheng@vivo.com>
```


Debugfs 是内核开发人员在用户空间获取信息的简单方法。与/proc 不同，proc 只提供进程信息。也不像 sysfs, 具有严格的“每个文件一个值“的规则。debugfs 根本没有规则, 开发人员可以在这里放置他们想要的任何信息。debugfs 文件系统也不能用作稳定的 ABI 接口。从理论上讲，debugfs 导出文件的时候没有任何约束。但是 [1] 实际情况并不总是那么简单。即使是 debugfs 接口，也最好根据需要进行设计, 并尽量保持接口不变。

Debugfs 通常使用以下命令安装:

```
mount -t debugfs none /sys/kernel/debug
```

（或等效的/etc/fstab 行）。debugfs 根目录默认仅可由 root 用户访问。要更改对文件树的访问，请使用“uid”，“gid”和“mode”挂载选项。请注意，debugfs API 仅按照 GPL 协议导出到模块。

使用 debugfs 的代码应包含 <linux/debugfs.h>。然后，首先是创建至少一个目录来保存一组 debugfs 文件:

```
struct dentry *debugfs_create_dir(const char *name, struct dentry␣
↪*parent);
```

如果成功，此调用将在指定的父目录下创建一个名为 name 的目录。如果 parent 参数为空，则会在 debugfs 根目录中创建。创建目录成功时，返回值是一个指向 dentry 结构体的指针。该 dentry 结构体的指针可用于在目录中创建文件（以及最后将其清理干净）。ERR_PTR （-ERROR）返回值表明出错。如果返回 ERR_PTR（-ENODEV），则表明内核是在没有 debugfs 支持的情况下构建的，并且下述函数都不会起作用。

在 debugfs 目录中创建文件的最通用方法是:

```
struct dentry *debugfs_create_file(const char *name, umode_t mode,
                                   struct dentry *parent, void␣
↪*data,
                                   const struct file_operations␣
↪*fops);
```

在这里，name 是要创建的文件的名称，mode 描述了访问文件应具有的权限，parent 指向应该保存文件的目录，data 将存储在产生的 inode 结构体的 i_private 字段中，而 fops 是一组文件操作函数，这些函数中实现文件操作的具体行为。至少，read () 和/或 write () 操作应提供；其他可以根据需要包括在内。同样的，返回值将是指向创建文件的 dentry 指针，错误时返回 ERR_PTR（-ERROR），系统不支持 debugfs 时返回值为 ERR_PTR （-ENODEV）。创建一个初始大小的文件，可以使用以下函数代替:

```
struct dentry *debugfs_create_file_size(const char *name, umode_t␣
↪mode,
                                struct dentry *parent, void *data,
                                const struct file_operations *fops,
                                loff_t file_size);
```

file_size 是初始文件大小。其他参数跟函数 debugfs_create_file 的相同。

在许多情况下，没必要自己去创建一组文件操作; 对于一些简单的情况,debugfs 代码提供了许多帮助函数。包含单个整数值的文件可以使用以下任何一项创建:

```
void debugfs_create_u8(const char *name, umode_t mode,
                       struct dentry *parent, u8 *value);
void debugfs_create_u16(const char *name, umode_t mode,
                        struct dentry *parent, u16 *value);
struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                  struct dentry *parent, u32␣
↪*value);
void debugfs_create_u64(const char *name, umode_t mode,
                        struct dentry *parent, u64 *value);
```

这些文件支持读取和写入给定值。如果某个文件不支持写入，只需根据需要设置 mode 参数位。这些文件中的值以十进制表示；如果需要使用十六进制，可以使用以下函数替代:

```
void debugfs_create_x8(const char *name, umode_t mode,
                       struct dentry *parent, u8 *value);
void debugfs_create_x16(const char *name, umode_t mode,
                        struct dentry *parent, u16 *value);
void debugfs_create_x32(const char *name, umode_t mode,
                        struct dentry *parent, u32 *value);
void debugfs_create_x64(const char *name, umode_t mode,
                        struct dentry *parent, u64 *value);
```

这些功能只有在开发人员知道导出值的大小的时候才有用。某些数据类型在不同的架构上有不同的宽度，这样会使情况变得有些复杂。在这种特殊情况下可以使用以下函数:

```
void debugfs_create_size_t(const char *name, umode_t mode,
                           struct dentry *parent, size_t *value);
```

不出所料，此函数将创建一个 debugfs 文件来表示类型为 size_t 的变量。

同样地，也有导出无符号长整型变量的函数，分别以十进制和十六进制表示如下:

```
struct dentry *debugfs_create_ulong(const char *name, umode_t mode,
                                    struct dentry *parent,
                                    unsigned long *value);
void debugfs_create_xul(const char *name, umode_t mode,
                        struct dentry *parent, unsigned long␣
↪*value);
```

布尔值可以通过以下方式放置在 debugfs 中:

```
struct dentry *debugfs_create_bool(const char *name, umode_t mode,
                                   struct dentry *parent, bool␣
↪*value);
```

读取结果文件将产生 Y（对于非零值）或 N，后跟换行符写入的时候，它只接受大写或小写值或 1 或 0。任何其他输入将被忽略。

同样，atomic_t 类型的值也可以放置在 debugfs 中:

```
void debugfs_create_atomic_t(const char *name, umode_t mode,
                             struct dentry *parent, atomic_t *value)
```

读取此文件将获得 atomic_t 值，写入此文件将设置 atomic_t 值。

另一个选择是通过以下结构体和函数导出一个任意二进制数据块:

```
struct debugfs_blob_wrapper {
    void *data;
    unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
```

(continues on next page)

(continued from previous page)

```
                                    struct dentry *parent,
                                    struct debugfs_blob_wrapper␣
↪*blob);
```

读取此文件将返回由指针指向 debugfs_blob_wrapper 结构体的数据。一些驱动使用“blobs”作为一种返回几行（静态）格式化文本的简单方法。这个函数可用于导出二进制信息，但似乎在主线中没有任何代码这样做。请注意，使用 debugfs_create_blob () 命令创建的所有文件是只读的。

如果您要转储一个寄存器块（在开发过程中经常会这么做，但是这样的调试代码很少上传到主线中。Debugfs 提供两个函数：一个用于创建仅寄存器文件，另一个把一个寄存器块插入一个顺序文件中:

```
struct debugfs_reg32 {
    char *name;
    unsigned long offset;
};

struct debugfs_regset32 {
    struct debugfs_reg32 *regs;
    int nregs;
    void __iomem *base;
};

struct dentry *debugfs_create_regset32(const char *name, umode_t␣
↪mode,
                                   struct dentry *parent,
                                   struct debugfs_regset32 *regset);

void debugfs_print_regs32(struct seq_file *s, struct debugfs_reg32␣
↪*regs,
                          int nregs, void __iomem *base, char *prefix);
```

“base”参数可能为 0，但您可能需要使用 __stringify 构建 reg32 数组，实际上有许多寄存器名称（宏）是寄存器块在基址上的字节偏移量。

如果要在 debugfs 中转储 u32 数组，可以使用以下函数创建文件:

```
void debugfs_create_u32_array(const char *name, umode_t mode,
                      struct dentry *parent,
                      u32 *array, u32 elements);
```

“array”参数提供数据，而“elements”参数为数组中元素的数量。注意：数组创建后，数组大小无法更改。

有一个函数来创建与设备相关的 seq_file:

```
struct dentry *debugfs_create_devm_seqfile(struct device *dev,
                               const char *name,
                               struct dentry *parent,
                               int (*read_fn)(struct seq_file *s,
                                       void *data));
```

“dev”参数是与此 debugfs 文件相关的设备，并且“read_fn”是一个函数指针，这个函数在打印 seq_file 内容的时候被回调。

还有一些其他的面向目录的函数:

```
struct dentry *debugfs_rename(struct dentry *old_dir,
                              struct dentry *old_dentry,
                              struct dentry *new_dir,
                              const char *new_name);

struct dentry *debugfs_create_symlink(const char *name,
                                      struct dentry *parent,
                                      const char *target);
```

调用 debugfs_rename() 将为现有的 debugfs 文件重命名，可能同时切换目录。new_name 函数调用之前不能存在；返回值为 old_dentry，其中包含更新的信息。可以使用 debugfs_create_symlink () 创建符号链接。

所有 debugfs 用户必须考虑的一件事是:

debugfs 不会自动清除在其中创建的任何目录。如果一个模块在不显式删除 debugfs 目录的情况下卸载模块，结果将会遗留很多野指针，从而导致系统不稳定。因此，所有 debugfs 用户-至少是那些可以作为模块构建的用户-必须做模块卸载的时候准备删除在此创建的所有文件和目录。一份文件可以通过以下方式删除:

```
void debugfs_remove(struct dentry *dentry);
```

dentry 值可以为 NULL 或错误值，在这种情况下，不会有任何文件被删除。

很久以前，内核开发者使用 debugfs 时需要记录他们创建的每个 dentry 指针，以便最后所有文件都可以被清理掉。但是，现在 debugfs 用户能调用以下函数递归清除之前创建的文件:

```
void debugfs_remove_recursive(struct dentry *dentry);
```

如果将对应顶层目录的 dentry 传递给以上函数，则该目录下的整个层次结构将会被删除。

注释：[1] http://lwn.net/Articles/309298/

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/arm64/index.rst

Translator
Bailu Lin <bailu.lin@vivo.com>

* ARM64 架构

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/arm64/amu.rst

Translator: Bailu Lin <bailu.lin@vivo.com>

* AArch64 Linux 中扩展的活动监控单元

作者: Ionela Voinescu <ionela.voinescu@arm.com>

日期: 2019-09-10

本文档简要描述了 AArch64 Linux 支持的活动监控单元的规范。

架构总述

活动监控是 ARMv8.4 CPU 架构引入的一个可选扩展特性。

活动监控单元 (在每个 CPU 中实现) 为系统管理提供了性能计数器。既可以通过系统寄存器的方式访问计数器，同时也支持外部内存映射的方式访问计数器。

AMUv1 架构实现了一个由 4 个固定的 64 位事件计数器组成的计数器组。

- CPU 周期计数器：同 CPU 的频率增长
- 常量计数器：同固定的系统时钟频率增长
- 淘汰指令计数器: 同每次架构指令执行增长
- 内存停顿周期计数器：计算由在时钟域内的最后一级缓存中未命中而引起的指令调度停顿周期数

当处于 WFI 或者 WFE 状态时，计数器不会增长。

AMU 架构提供了一个高达 16 位的事件计数器空间，未来新的 AMU 版本中可能用它来实现新增的事件计数器。

另外，AMUv1 实现了一个多达 16 个 64 位辅助事件计数器的计数器组。

冷复位时所有的计数器会清零。

基本支持

内核可以安全地运行在支持 AMU 和不支持 AMU 的 CPU 组合中。因此，当配置 CONFIG_ARM64_AMU_EXTN 后我们无条件使能后续 (secondary or hotplugged) CPU 检测和使用这个特性。

当在 CPU 上检测到该特性时，我们会标记为特性可用但是不能保证计数器的功能，仅表明有扩展属性。

固件 (代码运行在高异常级别，例如 arm-tf) 需支持以下功能：

- 提供低异常级别 (EL2 和 EL1) 访问 AMU 寄存器的能力。
- 使能计数器。如果未使能，它的值应为 0。
- 在从电源关闭状态启动 CPU 前或后保存或者恢复计数器。

当使用使能了该特性的内核启动但固件损坏时，访问计数器寄存器可能会遭遇 panic 或者死锁。即使未发现这些症状，计数器寄存器返回的数据结果并不一定能反映真实情况。通常，计数器会返回 0，表明他们未被使能。

如果固件没有提供适当的支持最好关闭 CONFIG_ARM64_AMU_EXTN。值得注意的是，出于安全原因，不要绕过 AMUSERRENR_EL0 设置而捕获从 EL0(用户空间) 访问 EL1(内核空间)。因此，固件应该确保访问 AMU 寄存器不会困在 EL2 或 EL3。

AMUv1 的固定计数器可以通过如下系统寄存器访问：

- SYS_AMEVCNTR0_CORE_EL0
- SYS_AMEVCNTR0_CONST_EL0
- SYS_AMEVCNTR0_INST_RET_EL0
- SYS_AMEVCNTR0_MEM_STALL_EL0

特定辅助计数器可以通过 SYS_AMEVCNTR1_EL0(n) 访问，其中 n 介于 0 到 15。

详细信息定义在目录：arch/arm64/include/asm/sysreg.h。

用户空间访问

由于以下原因，当前禁止从用户空间访问 AMU 的寄存器：

- 安全因数：可能会暴露处于安全模式执行的代码信息。
- 意愿：AMU 是用于系统管理的。

同样，该功能对用户空间不可见。

虚拟化

由于以下原因，当前禁止从 KVM 客户端的用户空间 (EL0) 和内核空间 (EL1) 访问 AMU 的寄存器：

- 安全因数：可能会暴露给其他客户端或主机端执行的代码信息。

任何试图访问 AMU 寄存器的行为都会触发一个注册在客户端的未定义异常。

Warning: 此文件的目的是为让中文读者更容易阅读和理解，而不是作为一个分支。因此，如果您对此文件有任何意见或更新，请先尝试更新原始英文文件。

Note: 如果您发现本文档与原始文件有任何不同或者有翻译问题，请联系该文件的译者，或者请求时奎亮的帮助：<alex.shi@linux.alibaba.com>。

Original
Documentation/arm64/hugetlbpage.rst

Translator: Bailu Lin <bailu.lin@vivo.com>

* ARM64 中的 HugeTLBpage

大页依靠有效利用 TLBs 来提高地址翻译的性能。这取决于以下两点 -

- 大页的大小
- TLBs 支持的条目大小

ARM64 接口支持 2 种大页方式。

1) pud/pmd 级别的块映射

这是常规大页，他们的 pmd 或 pud 页面表条目指向一个内存块。不管 TLB 中支持的条目大小如何，块映射可以减少翻译大页地址所需遍历的页表深度。

2) 使用连续位

架构中转换页表条目 (D4.5.3, ARM DDI 0487C.a) 中提供一个连续位告诉 MMU 这个条目是一个连续条目集的一员，它可以被缓存在单个 TLB 条目中。

在 Linux 中连续位用来增加 pmd 和 pte(最后一级) 级别映射的大小。受支持的连续页表条目数量因页面大小和页表级别而异。

支持以下大页尺寸配置 -

•	CONT PTE	PMD	CONT PMD	PUD
4K:	64K	2M	32M	1G
16K:	2M	32M	1G	
64K:	2M	512M	16G	

* 目录和表格

- genindex

CHAPTER

TWO

TRADUZIONE ITALIANA

manutentore
Federico Vaga <federico.vaga@vaga.pv.it>

* Avvertenze

L'obiettivo di questa traduzione è di rendere più facile la lettura e la comprensione per chi non capisce l'inglese o ha dubbi sulla sua interpretazione, oppure semplicemente per chi preferisce leggere in lingua italiana. Tuttavia, tenete ben presente che l'*unica* documentazione ufficiale è quella in lingua inglese: linux_doc

La propagazione simultanea a tutte le traduzioni di una modifica in linux_doc è altamente improbabile. I manutentori delle traduzioni - e i contributori - seguono l'evolversi della documentazione ufficiale e cercano di mantenere le rispettive traduzioni allineate nel limite del possibile. Per questo motivo non c'è garanzia che una traduzione sia aggiornata all'ultima modifica. Se quello che leggete in una traduzione non corrisponde a quello che leggete nel codice, informate il manutentore della traduzione e - se potete - verificate anche la documentazione in inglese.

Una traduzione non è un *fork* della documentazione ufficiale, perciò gli utenti non vi troveranno alcuna informazione diversa rispetto alla versione ufficiale. Ogni aggiunta, rimozione o modifica dei contenuti deve essere fatta prima nei documenti in inglese. In seguito, e quando è possibile, la stessa modifica dovrebbe essere applicata anche alle traduzioni. I manutentori delle traduzioni accettano contributi che interessano prettamente l'attività di traduzione (per esempio, nuove traduzioni, aggiornamenti, correzioni).

Le traduzioni cercano di essere il più possibile accurate ma non è possibile mappare direttamente una lingua in un'altra. Ogni lingua ha la sua grammatica e una sua cultura alle spalle, quindi la traduzione di una frase in inglese potrebbe essere modificata per adattarla all'italiano. Per questo motivo, quando leggete questa traduzione, potreste trovare alcune differenze di forma ma che trasmettono comunque il messaggio originale. Nonostante la grande diffusione di inglesismi nella lingua parlata, quando possibile, questi verranno sostituiti dalle corrispettive parole italiane

Se avete bisogno d'aiuto per comunicare con la comunità Linux ma non vi sentite a vostro agio nello scrivere in inglese, potete chiedere aiuto al manutentore della traduzione.

* La documentazione del kernel Linux

Questo è il livello principale della documentazione del kernel in lingua italiana. La traduzione è incompleta, noterete degli avvisi che vi segnaleranno la mancanza di una traduzione o di un gruppo di traduzioni.

Più in generale, la documentazione, come il kernel stesso, sono in costante sviluppo; particolarmente vero in quanto stiamo lavorando alla riorganizzazione della documentazione in modo più coerente. I miglioramenti alla documentazione sono sempre i benvenuti; per cui, se vuoi aiutare, iscriviti alla lista di discussione linux-doc presso vger.kernel.org.

* Documentazione sulla licenza dei sorgenti

I seguenti documenti descrivono la licenza usata nei sorgenti del kernel Linux (GPLv2), come licenziare i singoli file; inoltre troverete i riferimenti al testo integrale della licenza.

- it_kernel_licensing

* Documentazione per gli utenti

I seguenti manuali sono scritti per gli *utenti* del kernel - ovvero, coloro che cercano di farlo funzionare in modo ottimale su un dato sistema

Warning: TODO ancora da tradurre

* Documentazione per gli sviluppatori di applicazioni

Il manuale delle API verso lo spazio utente è una collezione di documenti che descrivono le interfacce del kernel viste dagli sviluppatori di applicazioni.

Warning: TODO ancora da tradurre

* Introduzione allo sviluppo del kernel

Questi manuali contengono informazioni su come contribuire allo sviluppo del kernel. Attorno al kernel Linux gira una comunità molto grande con migliaia di sviluppatori che contribuiscono ogni anno. Come in ogni grande comunità, sapere come le cose vengono fatte renderà il processo di integrazione delle vostre modifiche molto più semplice

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/index.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Lavorare con la comunità di sviluppo del kernel

Quindi volete diventare sviluppatori del kernel? Benvenuti! C'è molto da imparare sul lato tecnico del kernel, ma è anche importante capire come funziona la nostra comunità. Leggere questi documenti renderà più facile l'accettazione delle vostre modifiche con il minimo sforzo.

Di seguito le guide che ogni sviluppatore dovrebbe leggere.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/howto.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Come partecipare allo sviluppo del kernel Linux

Questo è il documento fulcro di quanto trattato sull'argomento. Esso contiene le istruzioni su come diventare uno sviluppatore del kernel Linux e spiega come lavorare con la comunità di sviluppo kernel Linux. Il documento non tratterà alcun aspetto tecnico relativo alla programmazione del kernel, ma vi aiuterà indirizzandovi sulla corretta strada.

Se qualsiasi cosa presente in questo documento diventasse obsoleta, vi preghiamo di inviare le correzioni agli amministratori di questo file, indicati in fondo al presente documento.

Introduzione

Dunque, volete imparare come diventare sviluppatori del kernel Linux? O vi è stato detto dal vostro capo, "Vai, scrivi un driver Linux per questo dispositivo". Bene, l'obbiettivo di questo documento è quello di insegnarvi tutto ciò che dovete sapere per raggiungere il vostro scopo descrivendo il procedimento da seguire e consigliandovi su come lavorare con la comunità. Il documento cercherà, inoltre, di spiegare alcune delle ragioni per le quali la comunità lavora in un modo suo particolare.

Il kernel è scritto prevalentemente nel linguaggio C con alcune parti specifiche dell'architettura scritte in linguaggio assembly. Per lo sviluppo kernel è richiesta una buona conoscenza del linguaggio C. L'assembly (di qualsiasi architettura)

non è richiesto, a meno che non pensiate di fare dello sviluppo di basso livello per un' architettura. Sebbene essi non siano un buon sostituto ad un solido studio del linguaggio C o ad anni di esperienza, i seguenti libri sono, se non altro, utili riferimenti:

- "The C Programming Language" di Kernighan e Ritchie [Prentice Hall]
- "Practical C Programming" di Steve Oualline [O' Reilly]
- "C: A Reference Manual" di Harbison and Steele [Prentice Hall]

Il kernel è stato scritto usando GNU C e la toolchain GNU. Sebbene si attenga allo standard ISO C89, esso utilizza una serie di estensioni che non sono previste in questo standard. Il kernel è un ambiente C indipendente, che non ha alcuna dipendenza dalle librerie C standard, così alcune parti del C standard non sono supportate. Le divisioni `long long` e numeri in virgola mobile non sono permessi. Qualche volta è difficile comprendere gli assunti che il kernel ha riguardo gli strumenti e le estensioni in uso, e sfortunatamente non esiste alcuna indicazione definitiva. Per maggiori informazioni, controllate, la pagina *info gcc*.

Tenete a mente che state cercando di apprendere come lavorare con la comunità di sviluppo già esistente. Questo è un gruppo eterogeneo di persone, con alti standard di codifica, di stile e di procedura. Questi standard sono stati creati nel corso del tempo basandosi su quanto hanno riscontrato funzionare al meglio per un squadra così grande e geograficamente sparsa. Cercate di imparare, in anticipo, il più possibile circa questi standard, poichè ben spiegati; non aspettatevi che gli altri si adattino al vostro modo di fare o a quello della vostra azienda.

Note legali

Il codice sorgente del kernel Linux è rilasciato sotto GPL. Siete pregati di visionare il file, COPYING, presente nella cartella principale dei sorgente, per eventuali dettagli sulla licenza. Se avete ulteriori domande sulla licenza, contattate un avvocato, non chiedete sulle liste di discussione del kernel Linux. Le persone presenti in queste liste non sono avvocati, e non dovreste basarvi sulle loro dichiarazioni in materia giuridica.

Per domande più frequenti e risposte sulla licenza GPL, guardare:

https://www.gnu.org/licenses/gpl-faq.html

Documentazione

I sorgenti del kernel Linux hanno una vasta base di documenti che vi insegneranno come interagire con la comunità del kernel. Quando nuove funzionalità vengono aggiunte al kernel, si raccomanda di aggiungere anche i relativi file di documentatione che spiegano come usarele. Quando un cambiamento del kernel genera anche un cambiamento nell' interfaccia con lo spazio utente, è raccomandabile che inviate una notifica o una correzione alle pagine *man* spiegando tale modifica agli amministratori di queste pagine all' indirizzo mtk.manpages@gmail.com, aggiungendo in CC la lista linux-api@vger.kernel.org.

Di seguito una lista di file che sono presenti nei sorgente del kernel e che è richiesto che voi leggiate:

Documentation/translations/it_IT/admin-guide/README.rst
Questo file da una piccola anteprima del kernel Linux e descrive il minimo necessario per configurare e generare il kernel. I novizi del kernel dovrebbero iniziare da qui.

Requisiti minimi per compilare il kernel

Questo file fornisce una lista dei pacchetti software necessari a compilare e far funzionare il kernel con successo.

Stile del codice per il kernel Linux

Questo file descrive lo stile della codifica per il kernel Linux, e parte delle motivazioni che ne sono alla base. Tutto il nuovo codice deve seguire le linee guida in questo documento. Molti amministratori accetteranno patch solo se queste osserveranno tali regole, e molte persone revisioneranno il codice solo se scritto nello stile appropriato.

Inviare patch: la guida essenziale per vedere il vostro codice nel kernel e Sottomettere driver per il kernel Linux

Questo file descrive dettagliatamente come creare ed inviare una patch con successo, includendo (ma non solo questo):

- Contenuto delle email
- Formato delle email
- I destinatari delle email

Seguire tali regole non garantirà il successo (tutte le patch sono soggette a controlli realitivi a contenuto e stile), ma non seguirle lo precluderà sempre.

Altre ottime descrizioni di come creare buone patch sono:

“The Perfect Patch”
https://www.ozlabs.org/~akpm/stuff/tpp.txt

“Linux kernel patch submission format”
https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html

L’ interfaccia dei driver per il kernel Linux

Questo file descrive la motivazioni sottostanti la conscia decisione di non avere un API stabile all’ interno del kernel, incluso cose come:

- Sottosistemi shim-layers (per compatibilità?)
- Portabilità fra Sistemi Operativi dei driver.
- Attenuare i rapidi cambiamenti all’ interno dei sorgenti del kernel (o prevenirli)

Questo documento è vitale per la comprensione della filosifia alla base dello sviluppo di Linux ed è molto importante per le persone che arrivano da esperienze con altri Sistemi Operativi.

Documentation/translations/it_IT/admin-guide/security-bugs.rst
Se ritenete di aver trovato un problema di sicurezza nel kernel Linux, seguite i passaggi scritti in questo documento per notificarlo agli sviluppatori del kernel, ed aiutare la risoluzione del problema.

Documentation/translations/it_IT/process/management-style.rst
Questo documento descrive come i manutentori del kernel Linux operano e la filosofia comune alla base del loro metodo. Questa è un' importante lettura per tutti coloro che sono nuovi allo sviluppo del kernel (o per chi è semplicemente curioso), poiché risolve molti dei più comuni fraintendimenti e confusioni dovuti al particolare comportamento dei manutentori del kernel.

Documentation/translations/it_IT/process/stable-kernel-rules.rst
Questo file descrive le regole sulle quali vengono basati i rilasci del kernel, e spiega cosa fare se si vuole che una modifica venga inserita in uno di questi rilasci.

Documentation/translations/it_IT/process/kernel-docs.rst
Una lista di documenti pertinenti allo sviluppo del kernel. Per favore consultate questa lista se non trovate ciò che cercate nella documentazione interna del kernel.

Documentation/translations/it_IT/process/applying-patches.rst
Una buona introduzione che descrivere esattamente cos'è una patch e come applicarla ai differenti rami di sviluppo del kernel.

Il kernel inoltre ha un vasto numero di documenti che possono essere automaticamente generati dal codice sorgente stesso o da file ReStructuredText (ReST), come questo. Esso include una completa descrizione dell' API interna del kernel, e le regole su come gestire la sincronizzazione (locking) correttamente

Tutte queste tipologie di documenti possono essere generati in PDF o in HTML utilizzando:

```
make pdfdocs
make htmldocs
```

rispettivamente dalla cartella principale dei sorgenti del kernel.

I documenti che impiegano ReST saranno generati nella cartella Documentation/output. Questi posso essere generati anche in formato LaTex e ePub con:

```
make latexdocs
make epubdocs
```

Diventare uno sviluppatore del kernel

Se non sapete nulla sullo sviluppo del kernel Linux, dovreste dare uno sguardo al progetto *Linux KernelNewbies*:

https://kernelnewbies.org

Esso prevede un' utile lista di discussione dove potete porre più o meno ogni tipo di quesito relativo ai concetti fondamentali sullo sviluppo del kernel (assicuratevi di cercare negli archivi, prima di chiedere qualcosa alla quale è già stata fornita risposta in passato). Esistono inoltre, un canale IRC che potete usare per formulare domande in tempo reale, e molti documenti utili che vi faciliteranno nell' apprendimento dello sviluppo del kernel Linux.

Il sito internet contiene informazioni di base circa l' organizzazione del codice, sottosistemi e progetti attuali (sia interni che esterni a Linux). Esso descrive, inoltre, informazioni logistiche di base, riguardanti ad esempio la compilazione del kernel e l' applicazione di una modifica.

Se non sapete dove cominciare, ma volete cercare delle attività dalle quali partire per partecipare alla comunità di sviluppo, andate al progetto Linux Kernel Janitor' s.

https://kernelnewbies.org/KernelJanitors

È un buon posto da cui iniziare. Esso presenta una lista di problematiche relativamente semplici da sistemare e pulire all' interno della sorgente del kernel Linux. Lavorando con gli sviluppatori incaricati di questo progetto, imparerete le basi per l' inserimento delle vostre modifiche all' interno dei sorgenti del kernel Linux, e possibilmente, sarete indirizzati al lavoro successivo da svolgere, se non ne avrete ancora idea.

Prima di apportare una qualsiasi modifica al codice del kernel Linux, è imperativo comprendere come tale codice funziona. A questo scopo, non c' è nulla di meglio che leggerlo direttamente (la maggior parte dei bit più complessi sono ben commentati), eventualmente anche con l' aiuto di strumenti specializzati. Uno degli strumenti che è particolarmente raccomandato è il progetto Linux Cross-Reference, che è in grado di presentare codice sorgente in un formato autoreferenziale ed indicizzato. Un eccellente ed aggiornata fonte di consultazione del codice del kernel la potete trovare qui:

https://elixir.bootlin.com/

Il processo di sviluppo

Il processo di sviluppo del kernel Linux si compone di pochi "rami" principali e di molti altri rami per specifici sottosistemi. Questi rami sono:

- I sorgenti kernel 4.x
- I sorgenti stabili del kernel 4.x.y -stable
- Sorgenti dei sottosistemi del kernel e le loro modifiche
- Il kernel 4.x -next per test d' integrazione

I sorgenti kernel 4.x

I kernel 4.x sono amministrati da Linus Torvald, e possono essere trovati su https://kernel.org nella cartella pub/linux/kernel/v4.x/. Il processo di sviluppo è il seguente:

- Non appena un nuovo kernel viene rilasciato si apre una finestra di due settimane. Durante questo periodo i manutentori possono proporre a Linus dei grossi cambiamenti; solitamente i cambiamenti che sono già stati inseriti nel ramo -next del kernel per alcune settimane. Il modo migliore per sottoporre dei cambiamenti è attraverso git (lo strumento usato per gestire i sorgenti del kernel, più informazioni sul sito https://git-scm.com/) ma anche delle patch vanno bene.
- Al termine delle due settimane un kernel -rc1 viene rilasciato e l'obbiettivo ora è quello di renderlo il più solido possibile. A questo punto la maggior parte delle patch dovrebbero correggere un'eventuale regressione. I bachi che sono sempre esistiti non sono considerabili come regressioni, quindi inviate questo tipo di cambiamenti solo se sono importanti. Notate che un intero driver (o filesystem) potrebbe essere accettato dopo la -rc1 poiché non esistono rischi di una possibile regressione con tale cambiamento, fintanto che quest'ultimo è auto-contenuto e non influisce su aree esterne al codice che è stato aggiunto. git può essere utilizzato per inviare le patch a Linus dopo che la -rc1 è stata rilasciata, ma è anche necessario inviare le patch ad una lista di discussione pubblica per un'ulteriore revisione.
- Una nuova -rc viene rilasciata ogni volta che Linus reputa che gli attuali sorgenti siano in uno stato di salute ragionevolmente adeguato ai test. L'obiettivo è quello di rilasciare una nuova -rc ogni settimana.
- Il processo continua fino a che il kernel è considerato "pronto"; tale processo dovrebbe durare circa in 6 settimane.

È utile menzionare quanto scritto da Andrew Morton sulla lista di discussione kernel-linux in merito ai rilasci del kernel:

> "*Nessuno sa quando un kernel verrà rilasciato, poichè questo è legato allo stato dei bachi e non ad una cronologia preventiva.*"

I sorgenti stabili del kernel 4.x.y -stable

I kernel con versioni in 3-parti sono "kernel stabili". Essi contengono correzioni critiche relativamente piccole nell'ambito della sicurezza oppure significative regressioni scoperte in un dato 4.x kernel.

Questo è il ramo raccomandato per gli utenti che vogliono un kernel recente e stabile e non sono interessati a dare il proprio contributo alla verifica delle versioni di sviluppo o sperimentali.

Se non è disponibile alcun kernel 4.x.y., quello più aggiornato e stabile sarà il kernel 4.x con la numerazione più alta.

4.x.y sono amministrati dal gruppo "stable" <stable@vger.kernel.org>, e sono rilasciati a seconda delle esigenze. Il normale periodo di rilascio è approssimati-

vamente di due settimane, ma può essere più lungo se non si verificano problematiche urgenti. Un problema relativo alla sicurezza, invece, può determinare un rilascio immediato.

Il file Documentation/process/stable-kernel-rules.rst (nei sorgenti) documenta quali tipologie di modifiche sono accettate per i sorgenti -stable, e come avviene il processo di rilascio.

Sorgenti dei sottosistemi del kernel e le loro patch

I manutentori dei diversi sottosistemi del kernel —ed anche molti sviluppatori di sottosistemi —mostrano il loro attuale stato di sviluppo nei loro repositori. In questo modo, altri possono vedere cosa succede nelle diverse parti del kernel. In aree dove lo sviluppo è rapido, potrebbe essere chiesto ad uno sviluppatore di basare le proprie modifiche su questi repositori in modo da evitare i conflitti fra le sottomissioni ed altri lavori in corso

La maggior parte di questi repositori sono git, ma esistono anche altri SCM in uso, o file di patch pubblicate come una serie quilt. Gli indirizzi dei repositori di sottosistema sono indicati nel file MAINTAINERS. Molti di questi posso essere trovati su https://git.kernel.org/.

Prima che una modifica venga inclusa in questi sottosistemi, sarà soggetta ad una revisione che inizialmente avviene tramite liste di discussione (vedere la sezione dedicata qui sotto). Per molti sottosistemi del kernel, tale processo di revisione è monitorato con lo strumento patchwork. Patchwork offre un' interfaccia web che mostra le patch pubblicate, inclusi i commenti o le revisioni fatte, e gli amministratori possono indicare le patch come "in revisione", "accettate", o "rifiutate". Diversi siti Patchwork sono elencati al sito https://patchwork.kernel.org/.

Il kernel 4.x -next per test d' integrazione

Prima che gli aggiornamenti dei sottosistemi siano accorpati nel ramo principale 4.x, sarà necessario un test d' integrazione. A tale scopo, esiste un repositorio speciale di test nel quale virtualmente tutti i rami dei sottosistemi vengono inclusi su base quotidiana:

https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git

In questo modo, i kernel -next offrono uno sguardo riassuntivo su quello che ci si aspetterà essere nel kernel principale nel successivo periodo d' incorporazione. Coloro che vorranno fare dei test d' esecuzione del kernel -next sono più che benvenuti.

Riportare Bug

https://bugzilla.kernel.org è dove gli sviluppatori del kernel Linux tracciano i bachi del kernel. Gli utenti sono incoraggiati nel riportare tutti i bachi che trovano utilizzando questo strumento. Per maggiori dettagli su come usare il bugzilla del kernel, guardare:

https://bugzilla.kernel.org/page.cgi?id=faq.html

Il file admin-guide/reporting-bugs.rst nella cartella principale del kernel fornisce un buon modello sul come segnalare un baco nel kernel, e spiega quali informazioni sono necessarie agli sviluppatori per poter aiutare il rintracciamento del problema.

Gestire i rapporti sui bug

Uno dei modi migliori per mettere in pratica le vostre capacità di hacking è quello di riparare bachi riportati da altre persone. Non solo aiuterete a far diventare il kernel più stabile, ma imparerete a riparare problemi veri dal mondo ed accrescerete le vostre competenze, e gli altri sviluppatori saranno al corrente della vostra presenza. Riparare bachi è una delle migliori vie per acquisire meriti tra gli altri sviluppatori, perchè non a molte persone piace perdere tempo a sistemare i bachi di altri.

Per lavorare sui rapporti di bachi già riportati, andate su https://bugzilla.kernel.org.

Liste di discussione

Come descritto in molti dei documenti qui sopra, la maggior parte degli sviluppatori del kernel partecipano alla lista di discussione Linux Kernel. I dettagli su come iscriversi e disiscriversi dalla lista possono essere trovati al sito:

http://vger.kernel.org/vger-lists.html#linux-kernel

Ci sono diversi archivi della lista di discussione. Usate un qualsiasi motore di ricerca per trovarli. Per esempio:

http://dir.gmane.org/gmane.linux.kernel

É caldamente consigliata una ricerca in questi archivi sul tema che volete sollevare, prima di pubblicarlo sulla lista. Molte cose sono già state discusse in dettaglio e registrate negli archivi della lista di discussione.

Molti dei sottosistemi del kernel hanno anche una loro lista di discussione dedicata. Guardate nel file MAINTAINERS per avere una lista delle liste di discussione e il loro uso.

Molte di queste liste sono gestite su kernel.org. Per informazioni consultate la seguente pagina:

http://vger.kernel.org/vger-lists.html

Per favore ricordatevi della buona educazione quando utilizzate queste liste. Sebbene sia un pò dozzinale, il seguente URL contiene alcune semplici linee guida per interagire con la lista (o con qualsiasi altra lista):

http://www.albion.com/netiquette/

Se diverse persone rispondo alla vostra mail, la lista dei riceventi (copia conoscenza) potrebbe diventare abbastanza lunga. Non cancellate nessuno dalla lista di CC: senza un buon motivo, e non rispondete solo all'indirizzo della lista di discussione. Fateci l'abitudine perché capita spesso di ricevere la stessa email due volte: una dal mittente ed una dalla lista; e non cercate di modificarla aggiungendo intestazioni stravaganti, agli altri non piacerà.

Ricordate di rimanere sempre in argomento e di mantenere le attribuzioni delle vostre risposte invariate; mantenete il "John Kernelhacker wrote …:" in cima alla vostra replica e aggiungete le vostre risposte fra i singoli blocchi citati, non scrivete all'inizio dell'email.

Se aggiungete patch alla vostra mail, assicuratevi che siano del tutto leggibili come indicato in Documentation/process/submitting-patches.rst. Gli sviluppatori kernel non vogliono avere a che fare con allegati o patch compresse; vogliono invece poter commentare le righe dei vostri cambiamenti, il che può funzionare solo in questo modo. Assicuratevi di utilizzare un gestore di mail che non alterì gli spazi ed i caratteri. Un ottimo primo test è quello di inviare a voi stessi una mail e cercare di sottoporre la vostra stessa patch. Se non funziona, sistemate il vostro programma di posta, o cambiatelo, finché non funziona.

Ed infine, per favore ricordatevi di mostrare rispetto per gli altri sottoscriventi.

Lavorare con la comunità

L'obiettivo di questa comunità è quello di fornire il miglior kernel possibile. Quando inviate una modifica che volete integrare, sarà valutata esclusivamente dal punto di vista tecnico. Quindi, cosa dovreste aspettarvi?

- critiche
- commenti
- richieste di cambiamento
- richieste di spiegazioni
- nulla

Ricordatevi che questo fa parte dell'integrazione della vostra modifica all'interno del kernel. Dovete essere in grado di accettare le critiche, valutarle a livello tecnico ed eventualmente rielaborare nuovamente le vostre modifiche o fornire delle chiare e concise motivazioni per le quali le modifiche suggerite non dovrebbero essere fatte. Se non riceverete risposte, aspettate qualche giorno e riprovate ancora, qualche volta le cose si perdono nell'enorme mucchio di email.

Cosa non dovreste fare?

- aspettarvi che la vostra modifica venga accettata senza problemi
- mettervi sulla difensiva
- ignorare i commenti
- sottomettere nuovamente la modifica senza fare nessuno dei cambiamenti richiesti

In una comunità che è alla ricerca delle migliori soluzioni tecniche possibili, ci saranno sempre opinioni differenti sull' utilità di una modifica. Siate cooperativi e vogliate adattare la vostra idea in modo che sia inserita nel kernel. O almeno vogliate dimostrare che la vostra idea vale. Ricordatevi, sbagliare è accettato fintanto che siate disposti a lavorare verso una soluzione che è corretta.

È normale che le risposte alla vostra prima modifica possa essere semplicemente una lista con dozzine di cose che dovreste correggere. Questo **non** implica che la vostra patch non sarà accettata, e questo **non** è contro di voi personalmente. Semplicemente correggete tutte le questioni sollevate contro la vostra modifica ed inviatela nuovamente.

Differenze tra la comunità del kernel e le strutture aziendali

La comunità del kernel funziona diversamente rispetto a molti ambienti di sviluppo aziendali. Qui di seguito una lista di cose che potete provare a fare per evitare problemi:

Cose da dire riguardanti le modifiche da voi proposte:

- "Questo risolve più problematiche."
- "Questo elimina 2000 stringhe di codice."
- "Qui una modifica che spiega cosa sto cercando di fare."
- "L' ho testato su 5 diverse architetture.."
- "Qui una serie di piccole modifiche che.."
- "Questo aumenta le prestazioni di macchine standard…"

Cose che dovreste evitare di dire:

- **"Lo abbiamo fatto in questo modo in AIX/ptx/Solaris, di conseguenza** deve per forza essere giusto…"
- "Ho fatto questo per 20 anni, quindi.."
- "Questo è richiesto dalla mia Azienda per far soldi"
- "Questo è per la linea di prodotti della nostra Azienda"
- **"Ecco il mio documento di design di 1000 pagine che descrive ciò che ho** in mente"
- "Ci ho lavorato per 6 mesi…"
- "Ecco una patch da 5000 righe che.."
- "Ho riscritto il pasticcio attuale, ed ecco qua.."
- "Ho una scadenza, e questa modifica ha bisogno di essere approvata ora"

Un' altra cosa nella quale la comunità del kernel si differenzia dai più classici ambienti di ingegneria del software è la natura "senza volto" delle interazioni umane. Uno dei benefici dell' uso delle email e di irc come forma primordiale

di comunicazione è l'assenza di discriminazione basata su genere e razza. L'ambienti di lavoro Linux accetta donne e minoranze perchè tutto quello che sei è un indirizzo email. Aiuta anche l'aspetto internazionale nel livellare il terreno di gioco perchè non è possibile indovinare il genere basandosi sul nome di una persona. Un uomo può chiamarsi Andrea ed una donna potrebbe chiamarsi Pat. Gran parte delle donne che hanno lavorato al kernel Linux e che hanno espresso una personale opinione hanno avuto esperienze positive.

La lingua potrebbe essere un ostacolo per quelle persone che non si trovano a loro agio con l'inglese. Una buona padronanza del linguaggio può essere necessaria per esporre le proprie idee in maniera appropiata all'interno delle liste di discussione, quindi è consigliabile che rileggiate le vostre email prima di inviarle in modo da essere certi che abbiano senso in inglese.

Spezzare le vostre modifiche

La comunità del kernel Linux non accetta con piacere grossi pezzi di codice buttati lì tutti in una volta. Le modifiche necessitano di essere adeguatamente presentate, discusse, e suddivise in parti più piccole ed indipendenti. Questo è praticamente l'esatto opposto di quello che le aziende fanno solitamente. La vostra proposta dovrebbe, inoltre, essere presentata prestissimo nel processo di sviluppo, così che possiate ricevere un riscontro su quello che state facendo. Lasciate che la comunità senta che state lavorando con loro, e che non li stiate sfruttando come discarica per le vostre aggiunte. In ogni caso, non inviate 50 email nello stesso momento in una lista di discussione, il più delle volte la vostra serie di modifiche dovrebbe essere più piccola.

I motivi per i quali dovreste frammentare le cose sono i seguenti:

1) Piccole modifiche aumentano le probabilità che vengano accettate, altrimenti richiederebbe troppo tempo o sforzo nel verificarne la correttezza. Una modifica di 5 righe può essere accettata da un manutentore con a mala pena una seconda occhiata. Invece, una modifica da 500 linee può richiedere ore di rilettura per verificarne la correttezza (il tempo necessario è esponenzialmente proporzionale alla dimensione della modifica, o giù di lì)

   Piccole modifiche sono inoltre molto facili da debuggare quando qualcosa non va. È molto più facile annullare le modifiche una per una che dissezionare una patch molto grande dopo la sua sottomissione (e rompere qualcosa).

2) È importante non solo inviare piccole modifiche, ma anche riscriverle e semplificarle (o più semplicemente ordinarle) prima di sottoporle.

Qui un'analogia dello sviluppatore kernel Al Viro:

> "*Pensate ad un insegnante di matematica che corregge il compito di uno studente (di matematica). L'insegnante non vuole vedere le prove e gli errori commessi dallo studente prima che arrivi alla soluzione. Vuole vedere la risposta più pulita ed elegante possibile. Un buono studente lo sa, e non presenterebbe mai le proprie bozze prima prima della soluzione finale*"
>
> "*Lo stesso vale per lo sviluppo del kernel. I manutentori ed i revisori non vogliono vedere il procedimento che sta dietro al problema che uno*

sta risolvendo. Vogliono vedere una soluzione semplice ed elegante.”

Può essere una vera sfida il saper mantenere l’ equilibrio fra una presentazione elegante della vostra soluzione, lavorare insieme ad una comunità e dibattere su un lavoro incompleto. Pertanto è bene entrare presto nel processo di revisione per migliorare il vostro lavoro, ma anche per riuscire a tenere le vostre modifiche in pezzettini che potrebbero essere già accettate, nonostante la vostra intera attività non lo sia ancora.

In fine, rendetevi conto che non è accettabile inviare delle modifiche incomplete con la promessa che saranno “sistemate dopo”.

Giustificare le vostre modifiche

Insieme alla frammentazione delle vostre modifiche, è altrettanto importante permettere alla comunità Linux di capire perché dovrebbero accettarle. Nuove funzionalità devono essere motivate come necessarie ed utili.

Documentare le vostre modifiche

Quando inviate le vostre modifiche, fate particolare attenzione a quello che scrivete nella vostra email. Questa diventerà il *ChangeLog* per la modifica, e sarà visibile a tutti per sempre. Dovrebbe descrivere la modifica nella sua interezza, contenendo:

- perchè la modifica è necessaria
- l’ approccio d’ insieme alla patch
- dettagli supplementari
- risultati dei test

Per maggiori dettagli su come tutto ciò dovrebbe apparire, riferitevi alla sezione ChangeLog del documento:

“The Perfect Patch”
http://www.ozlabs.org/~akpm/stuff/tpp.txt

A volte tutto questo è difficile da realizzare. Il perfezionamento di queste pratiche può richiedere anni (eventualmente). È un processo continuo di miglioramento che richiede molta pazienza e determinazione. Ma non mollate, si può fare. Molti lo hanno fatto prima, ed ognuno ha dovuto iniziare dove siete voi ora.

Grazie a Paolo Ciarrocchi che ha permesso che la sezione “Development Process” (https://lwn.net/Articles/94386/) fosse basata sui testi da lui scritti, ed a Randy Dunlap e Gerrit Huizenga per la lista di cose che dovreste e non dovreste dire. Grazie anche a Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, e Alex Shepard per le loro revisioni, commenti e contributi. Senza il loro aiuto, questo documento non sarebbe stato possibile.

Manutentore: Greg Kroah-Hartman <greg@kroah.com>

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/code-of-conduct.rst

Accordo dei contributori sul codice di condotta

Warning: TODO ancora da tradurre

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/development-process.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Una guida al processo di sviluppo del Kernel

Contenuti:

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/1.Intro.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Introduzione

Riepilogo generale

Il resto di questa sezione riguarda il processo di sviluppo del kernel e quella sorta di frustrazione che gli sviluppatori e i loro datori di lavoro potrebbero dover affrontare. Ci sono molte ragioni per le quali del codice per il kernel debba essere incorporato nel kernel ufficiale, fra le quali: disponibilità immediata agli utilizzatori, supporto della comunità in differenti modalità, e la capacità di influenzare la direzione dello sviluppo del kernel. Il codice che contribuisce al kernel Linux deve essere reso disponibile sotto una licenza GPL-compatibile.

La sezione *Come funziona il processo di sviluppo* introduce il processo di sviluppo, il ciclo di rilascio del kernel, ed i meccanismi della finestra d'incorporazione. Il capitolo copre le varie fasi di una modifica: sviluppo, revisione e ciclo d'incorporazione. Ci sono alcuni dibattiti su strumenti e liste di discussione. Gli sviluppatori che sono in attesa di poter sviluppare qualcosa per il kernel sono invitati ad individuare e sistemare bachi come esercizio iniziale.

La sezione *I primi passi della pianificazione* copre i primi stadi della pianificazione di un progetto di sviluppo, con particolare enfasi sul coinvolgimento della comunità, il prima possibile.

La sezione *Scrivere codice corretto* riguarda il processo di scrittura del codice. Qui, sono esposte le diverse insidie che sono state già affrontate da altri sviluppatori. Il capitolo copre anche alcuni dei requisiti per le modifiche, ed esiste un'introduzione ad alcuni strumenti che possono aiutarvi nell'assicurarvi che le modifiche per il kernel siano corrette.

La sezione *Pubblicare modifiche* parla del processo di pubblicazione delle modifiche per la revisione. Per essere prese in considerazione dalla comunità di sviluppo, le modifiche devono essere propriamente formattate ed esposte, e devono essere inviate nel posto giusto. Seguire i consigli presenti in questa sezione dovrebbe essere d'aiuto nell'assicurare la migliore accoglienza possibile del vostro lavoro.

La sezione *Completamento* copre ciò che accade dopo la pubblicazione delle modifiche; a questo punto il lavoro è lontano dall'essere concluso. Lavorare con i revisori è una parte cruciale del processo di sviluppo; questa sezione offre una serie di consigli su come evitare problemi in questa importante fase. Gli sviluppatori sono diffidenti nell'affermare che il lavoro è concluso quando una modifica è incorporata nei sorgenti principali.

La sezione *Argomenti avanzati* introduce un paio di argomenti "avanzati": gestire le modifiche con git e controllare le modifiche pubblicate da altri.

La sezione *Per maggiori informazioni* chiude il documento con dei riferimenti ad altre fonti che forniscono ulteriori informazioni sullo sviluppo del kernel.

Di cosa parla questo documento

Il kernel Linux, ha oltre 8 milioni di linee di codice e ben oltre 1000 contributori ad ogni rilascio; è uno dei più vasti e più attivi software liberi progettati mai esistiti. Sin dal sul modesto inizio nel 1991, questo kernel si è evoluto nel miglior componente per sistemi operativi che fanno funzionare piccoli riproduttori musicali, PC, grandi super computer e tutte le altre tipologie di sistemi fra questi estremi. È una soluzione robusta, efficiente ed adattabile a praticamente qualsiasi situazione.

Con la crescita di Linux è arrivato anche un aumento di sviluppatori (ed aziende) desiderosi di partecipare a questo sviluppo. I produttori di hardware vogliono assicurarsi che il loro prodotti siano supportati da Linux, rendendo questi prodotti attrattivi agli utenti Linux. I produttori di sistemi integrati, che usano Linux come componente di un prodotto integrato, vogliono che Linux sia capace ed adeguato agli obiettivi ed il più possibile alla mano. Fornitori ed altri produttori di software che basano i propri prodotti su Linux hanno un chiaro interesse verso capacità, prestazioni ed affidabilità del kernel Linux. E gli utenti finali, anche, spesso vorrebbero cambiare Linux per renderlo più aderente alle proprie necessità.

Una delle caratteristiche più coinvolgenti di Linux è quella dell'accessibilità per gli sviluppatori; chiunque con le capacità richieste può migliorare Linux ed influenzarne la direzione di sviluppo. Prodotti non open-source non possono offrire questo tipo di apertura, che è una caratteristica del software libero. Ma, anzi, il kernel è persino più aperto rispetto a molti altri progetti di software libero. Un classico ciclo di sviluppo trimestrale può coinvolgere 1000 sviluppatori che lavorano per più di 100 differenti aziende (o per nessuna azienda).

Lavorare con la comunità di sviluppo del kernel non è particolarmente difficile. Ma, ciononostante, diversi potenziali contributori hanno trovato delle difficoltà quando hanno cercato di lavorare sul kernel. La comunità del kernel utilizza un proprio modo di operare che gli permette di funzionare agevolmente (e genera un prodotto di alta qualità) in un ambiente dove migliaia di stringhe di codice sono modificate ogni giorni. Quindi non deve sorprendere che il processo di sviluppo del kernel differisca notevolmente dai metodi di sviluppo privati.

Il processo di sviluppo del Kernel può, dall'altro lato, risultare intimidatorio e strano ai nuovi sviluppatori, ma ha dietro di se buone ragioni e solide esperienze. Uno sviluppatore che non comprende i modi della comunità del kernel (o, peggio, che cerchi di aggirarli o violarli) avrà un'esperienza deludente nel proprio bagaglio. La comunità di sviluppo, sebbene sia utile a coloro che cercano di imparare, ha poco tempo da dedicare a coloro che non ascoltano o coloro che non sono interessati al processo di sviluppo.

Si spera che coloro che leggono questo documento saranno in grado di evitare queste esperienze spiacevoli. C'è molto materiale qui, ma lo sforzo della lettura sarà ripagato in breve tempo. La comunità di sviluppo ha sempre bisogno di sviluppatori che vogliano aiutare a rendere il kernel migliore; il testo seguente potrebbe esservi d'aiuto - o essere d'aiuto ai vostri collaboratori- per entrare a far parte della nostra comunità.

Crediti

Questo documento è stato scritto da Jonathan Corbet, corbet@lwn.net. È stato migliorato da Johannes Berg, James Berry, Alex Chiang, Roland Dreier, Randy Dunlap, Jake Edge, Jiri Kosina, Matt Mackall, Arthur Marsh, Amanda McPherson, Andrew Morton, Andrew Price, Tsugikazu Shibata e Jochen Voß.

Questo lavoro è stato supportato dalla Linux Foundation; un ringraziamento speciale ad Amanda McPherson, che ha visto il valore di questo lavoro e lo ha reso possibile.

L'importanza d'avere il codice nei sorgenti principali

Alcune aziende e sviluppatori ogni tanto si domandano perché dovrebbero preoccuparsi di apprendere come lavorare con la comunità del kernel e di inserire il loro codice nel ramo di sviluppo principale (per ramo principale s'intende quello mantenuto da Linus Torvalds e usato come base dai distributori Linux). Nel breve termine, contribuire al codice può sembrare un costo inutile; può sembra più facile tenere separato il proprio codice e supportare direttamente i suoi utilizzatori. La verità è che il tenere il codice separato ("fuori dai sorgenti", "*out-of-tree*") è un falso risparmio.

Per dimostrare i costi di un codice "fuori dai sorgenti", eccovi alcuni aspetti rilevanti del processo di sviluppo kernel; la maggior parte di essi saranno approfonditi dettagliatamente più avanti in questo documento. Considerate:

- Il codice che è stato inserito nel ramo principale del kernel è disponibile a tutti gli utilizzatori Linux. Sarà automaticamente presente in tutte le distribuzioni che lo consentono. Non c'è bisogno di: driver per dischi, scaricare file, o della scocciatura del dover supportare diverse versioni di diverse distribuzioni; funziona già tutto, per gli sviluppatori e per gli utilizzatori. L'inserimento nel ramo principale risolve un gran numero di problemi di distribuzione e di supporto.

- Nonostante gli sviluppatori kernel si sforzino di tenere stabile l'interfaccia dello spazio utente, quella interna al kernel è in continuo cambiamento. La mancanza di un'interfaccia interna è deliberatamente una decisione di progettazione; ciò permette che i miglioramenti fondamentali vengano fatti in un qualsiasi momento e che risultino fatti con un codice di alta qualità. Ma una delle conseguenze di questa politica è che qualsiasi codice "fuori dai sorgenti" richiede costante manutenzione per renderlo funzionante coi kernel più recenti. Tenere un codice "fuori dai sorgenti" richiede una mole di lavoro significativa solo per farlo funzionare.

  Invece, il codice che si trova nel ramo principale non necessita di questo tipo di lavoro poiché ad ogni sviluppatore che faccia una modifica alle interfacce viene richiesto di sistemare anche il codice che utilizza quell'interfaccia. Quindi, il codice che è stato inserito nel ramo principale ha dei costi di mantenimento significativamente più bassi.

- Oltre a ciò, spesso il codice che è all'interno del kernel sarà migliorato da altri sviluppatori. Dare pieni poteri alla vostra comunità di utenti e ai clienti può portare a sorprendenti risultati che migliorano i vostri prodotti.

- Il codice kernel è soggetto a revisioni, sia prima che dopo l'inserimento nel ramo principale. Non importa quanto forti fossero le abilità dello sviluppatore originale, il processo di revisione troverà il modo di migliore il codice. Spesso la revisione trova bachi importanti e problemi di sicurezza. Questo è particolarmente vero per il codice che è stato sviluppato in un ambiente chiuso; tale codice ottiene un forte beneficio dalle revisioni provenienti da sviluppatori esteri. Il codice "fuori dai sorgenti", invece, è un codice di bassa qualità.
- La partecipazione al processo di sviluppo costituisce la vostra via per influenzare la direzione di sviluppo del kernel. Gli utilizzatori che "reclamano da bordo campo" sono ascoltati, ma gli sviluppatori attivi hanno una voce più forte - e la capacità di implementare modifiche che renderanno il kernel più funzionale alle loro necessità.
- Quando il codice è gestito separatamente, esiste sempre la possibilità che terze parti contribuiscano con una differente implementazione che fornisce le stesse funzionalità. Se dovesse accadere, l'inserimento del codice diventerà molto più difficile - fino all'impossibilità. Poi, dovrete far fronte a delle alternative poco piacevoli, come: (1) mantenere un elemento non standard "fuori dai sorgenti" per un tempo indefinito, o (2) abbandonare il codice e far migrare i vostri utenti alla versione "nei sorgenti".
- Contribuire al codice è l'azione fondamentale che fa funzionare tutto il processo. Contribuendo attraverso il vostro codice potete aggiungere nuove funzioni al kernel e fornire competenze ed esempi che saranno utili ad altri sviluppatori. Se avete sviluppato del codice Linux (o state pensando di farlo), avete chiaramente interesse nel far proseguire il successo di questa piattaforma. Contribuire al codice è une delle migliori vie per aiutarne il successo.

Il ragionamento sopra citato si applica ad ogni codice "fuori dai sorgenti"dal kernel, incluso il codice proprietario distribuito solamente in formato binario. Ci sono, comunque, dei fattori aggiuntivi che dovrebbero essere tenuti in conto prima di prendere in considerazione qualsiasi tipo di distribuzione binaria di codice kernel. Questo include che:

- Le questioni legali legate alla distribuzione di moduli kernel proprietari sono molto nebbiose; parecchi detentori di copyright sul kernel credono che molti moduli binari siano prodotti derivati del kernel e che, come risultato, la loro diffusione sia una violazione della licenza generale di GNU (della quale si parlerà più avanti). L'autore qui non è un avvocato, e niente in questo documento può essere considerato come un consiglio legale. Il vero stato legale dei moduli proprietari può essere determinato esclusivamente da un giudice. Ma l'incertezza che perseguita quei moduli è lì comunque.
- I moduli binari aumentano di molto la difficoltà di fare debugging del kernel, al punto che la maggior parte degli sviluppatori del kernel non vorranno nemmeno tentare. Quindi la diffusione di moduli esclusivamente binari renderà difficile ai vostri utilizzatori trovare un supporto dalla comunità.
- Il supporto è anche difficile per i distributori di moduli binari che devono fornire una versione del modulo per ogni distribuzione e per ogni versione del kernel che vogliono supportate. Per fornire una copertura ragionevole e comprensiva, può essere richiesto di produrre dozzine di singoli moduli. E inoltre i vostri utilizzatori dovranno aggiornare il vostro modulo separatamente

ogni volta che aggiornano il loro kernel.

- Tutto ciò che è stato detto prima riguardo alla revisione del codice si applica doppiamente al codice proprietario. Dato che questo codice non è del tutto disponibile, non può essere revisionato dalla comunità e avrà, senza dubbio, seri problemi.

I produttori di sistemi integrati, in particolare, potrebbero esser tentati dall'evitare molto di ciò che è stato detto in questa sezione, credendo che stiano distribuendo un prodotto finito che utilizza una versione del kernel immutabile e che non richiede un ulteriore sviluppo dopo il rilascio. Questa idea non comprende il valore di una vasta revisione del codice e il valore del permettere ai propri utenti di aggiungere funzionalità al vostro prodotto. Ma anche questi prodotti, hanno una vita commerciale limitata, dopo la quale deve essere rilasciata una nuova versione. A quel punto, i produttori il cui codice è nel ramo principale di sviluppo avranno un codice ben mantenuto e saranno in una posizione migliore per ottenere velocemente un nuovo prodotto pronto per essere distribuito.

Licenza

IL codice Linux utilizza diverse licenze, ma il codice completo deve essere compatibile con la seconda versione della licenza GNU General Public License (GPLv2), che è la licenza che copre la distribuzione del kernel. Nella pratica, ciò significa che tutti i contributi al codice sono coperti anche' essi dalla GPLv2 (con, opzionalmente, una dicitura che permette la possibilità di distribuirlo con licenze più recenti di GPL) o dalla licenza three-clause BSD. Qualsiasi contributo che non è coperto da una licenza compatibile non verrà accettata nel kernel.

Per il codice sottomesso al kernel non è necessario (o richiesto) la concessione del Copyright. Tutto il codice inserito nel ramo principale del kernel conserva la sua proprietà originale; ne risulta che ora il kernel abbia migliaia di proprietari.

Una conseguenza di questa organizzazione della proprietà è che qualsiasi tentativo di modifica della licenza del kernel è destinata ad un quasi sicuro fallimento. Esistono alcuni scenari pratici nei quali il consenso di tutti i detentori di copyright può essere ottenuto (o il loro codice verrà rimosso dal kernel). Quindi, in sostanza, non esiste la possibilità che si giunga ad una versione 3 della licenza GPL nel prossimo futuro.

È imperativo che tutto il codice che contribuisce al kernel sia legittimamente software libero. Per questa ragione, un codice proveniente da un contributore anonimo (o sotto pseudonimo) non verrà accettato. È richiesto a tutti i contributori di firmare il proprio codice, attestando così che quest' ultimo può essere distribuito insieme al kernel sotto la licenza GPL. Il codice che non è stato licenziato come software libero dal proprio creatore, o che potrebbe creare problemi di copyright per il kernel (come il codice derivante da processi di ingegneria inversa senza le opportune tutele), non può essere diffuso.

Domande relative a questioni legate al copyright sono frequenti nelle liste di discussione dedicate allo sviluppo di Linux. Tali quesiti, normalmente, non riceveranno alcuna risposta, ma una cosa deve essere tenuta presente: le persone che risponderanno a quelle domande non sono avvocati e non possono fornire supporti legali. Se avete questioni legali relative ai sorgenti del codice Linux, non

esiste alternativa che quella di parlare con un avvocato esperto nel settore. Fare affidamento sulle risposte ottenute da una lista di discussione tecnica è rischioso.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/2.Process.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Come funziona il processo di sviluppo

Lo sviluppo del Kernel agli inizi degli anno '90 era abbastanza libero, con un numero di utenti e sviluppatori relativamente basso. Con una base di milioni di utenti e con 2000 sviluppatori coinvolti nel giro di un anno, il kernel da allora ha messo in atto un certo numero di procedure per rendere lo sviluppo più agevole. È richiesta una solida conoscenza di come tale processo si svolge per poter esserne parte attiva.

Il quadro d'insieme

Gli sviluppatori kernel utilizzano un calendario di rilascio generico, dove ogni due o tre mesi viene effettuata un rilascio importante del kernel. I rilasci più recenti sono stati:

5.0	3 marzo, 2019
5.1	5 maggio, 2019
5.2	7 luglio, 2019
5.3	15 settembre, 2019
5.4	24 novembre, 2019
5.5	6 gennaio, 2020

Ciascun rilascio 5.x è un importante rilascio del kernel con nuove funzionalità, modifiche interne dell'API, e molto altro. Un tipico rilascio contiene quasi 13,000 gruppi di modifiche con ulteriori modifiche a parecchie migliaia di linee di codice. La 5.x. è pertanto la linea di confine nello sviluppo del kernel Linux; il kernel utilizza un sistema di sviluppo continuo che integra costantemente nuove importanti modifiche.

Viene seguita una disciplina abbastanza lineare per l'inclusione delle patch di ogni rilascio. All'inizio di ogni ciclo di sviluppo, la "finestra di inclusione" viene dichiarata aperta. In quel momento il codice ritenuto sufficientemente stabile(e che è accettato dalla comunità di sviluppo) viene incluso nel ramo principale del kernel. La maggior parte delle patch per un nuovo ciclo di sviluppo (e tutte le più importanti modifiche) saranno inserite durante questo periodo, ad un ritmo che si attesta sulle 1000 modifiche ("patch" o "gruppo di modifiche") al giorno.

(per inciso, vale la pena notare che i cambiamenti integrati durante la "finestra di inclusione" non escono dal nulla; questi infatti, sono stati raccolti e, verificati in anticipo. Il funzionamento di tale procedimento verrà descritto dettagliatamente più avanti).

La finestra di inclusione resta attiva approssimativamente per due settimane. Al termine di questo periodo, Linus Torvald dichiarerà che la finestra è chiusa e rilascerà il primo degli "rc"del kernel. Per il kernel che è destinato ad essere 5.6, per esempio, il rilascio che emerge al termine della finestra d' inclusione si chiamerà 5.6-rc1. Questo rilascio indica che il momento di aggiungere nuovi componenti è passato, e che è iniziato il periodo di stabilizzazione del prossimo kernel.

Nelle successive sei/dieci settimane, potranno essere sottoposte solo modifiche che vanno a risolvere delle problematiche. Occasionalmente potrà essere consentita una modifica più consistente, ma tali occasioni sono rare. Gli sviluppatori che tenteranno di aggiungere nuovi elementi al di fuori della finestra di inclusione, tendenzialmente, riceveranno un accoglienza poco amichevole. Come regola generale: se vi perdete la finestra di inclusione per un dato componente, la cosa migliore da fare è aspettare il ciclo di sviluppo successivo (un' eccezione può essere fatta per i driver per hardware non supportati in precedenza; se toccano codice non facente parte di quello attuale, che non causino regressioni e che potrebbero essere aggiunti in sicurezza in un qualsiasi momento)

Mentre le correzioni si aprono la loro strada all' interno del ramo principale, il ritmo delle modifiche rallenta col tempo. Linus rilascia un nuovo kernel -rc circa una volta alla settimana; e ne usciranno circa 6 o 9 prima che il kernel venga considerato sufficientemente stabile e che il rilascio finale venga fatto. A quel punto tutto il processo ricomincerà.

Esempio: ecco com' è andato il ciclo di sviluppo della versione 5.4 (tutte le date si collocano nel 2018)

15 settembre	5.3 rilascio stabile
30 settembre	5.4-rc1, finestra di inclusione chiusa
6 ottobre	5.4-rc2
13 ottobre	5.4-rc3
20 ottobre	5.4-rc4
27 ottobre	5.4-rc5
3 novembre	5.4-rc6
10 novembre	5.4-rc7
17 novembre	5.4-rc8
24 novembre	5.4 rilascio stabile

In che modo gli sviluppatori decidono quando chiudere il ciclo di sviluppo e creare quindi una rilascio stabile? Un metro valido è il numero di regressioni rilevate nel precedente rilascio. Nessun baco è il benvenuto, ma quelli che procurano problemi su sistemi che hanno funzionato in passato sono considerati particolarmente seri. Per questa ragione, le modifiche che portano ad una regressione sono viste sfavorevolmente e verranno quasi sicuramente annullate durante il periodo di stabilizzazione.

L' obiettivo degli sviluppatori è quello di aggiustare tutte le regressioni conosciute prima che avvenga il rilascio stabile. Nel mondo reale, questo tipo di perfezione

difficilmente viene raggiunta; esistono troppe variabili in un progetto di questa portata. Arriva un punto dove ritardare il rilascio finale peggiora la situazione; la quantità di modifiche in attesa della prossima finestra di inclusione crescerà enormemente, creando ancor più regressioni al giro successivo. Quindi molti kernel 5.x escono con una manciata di regressioni delle quali, si spera, nessuna è grave.

Una volta che un rilascio stabile è fatto, il suo costante mantenimento è affidato al "squadra stabilità", attualmente composta da Greg Kroah-Hartman. Questa squadra rilascia occasionalmente degli aggiornamenti relativi al rilascio stabile usando la numerazione 5.x.y. Per essere presa in considerazione per un rilascio d'aggiornamento, una modifica deve: (1) correggere un baco importante (2) essere già inserita nel ramo principale per il prossimo sviluppo del kernel. Solitamente, passato il loro rilascio iniziale, i kernel ricevono aggiornamenti per più di un ciclo di sviluppo. Quindi, per esempio, la storia del kernel 5.2 appare così (anno 2019):

15 settembre	5.2 rilascio stabile FIXME settembre è sbagliato
14 luglio	5.2.1
21 luglio	5.2.2
26 luglio	5.2.3
28 luglio	5.2.4
31 luglio	5.2.5
…	…
11 ottobre	5.2.21

La 5.2.21 fu l'aggiornamento finale per la versione 5.2.

Alcuni kernel sono destinati ad essere kernel a "lungo termine"; questi riceveranno assistenza per un lungo periodo di tempo. Al momento in cui scriviamo, i manutentori dei kernel stabili a lungo termine sono:

3.16	Ben Hutchings	(kernel stabile molto più a lungo termine)
4.4	Greg Kroah-Hartman e Sasha Levin	(kernel stabile molto più a lungo termine)
4.9	Greg Kroah-Hartman e Sasha Levin	
4.14	Greg Kroah-Hartman e Sasha Levin	
4.19	Greg Kroah-Hartman e Sasha Levin	
5.4i	Greg Kroah-Hartman e Sasha Levin	

Questa selezione di kernel di lungo periodo sono puramente dovuti ai loro manutentori, alla loro necessità e al tempo per tenere aggiornate proprio quelle versioni. Non ci sono altri kernel a lungo termine in programma per alcun rilascio in arrivo.

Il ciclo di vita di una patch

Le patch non passano direttamente dalla tastiera dello sviluppatori al ramo principale del kernel. Esiste, invece, una procedura disegnata per assicurare che ogni patch sia di buona qualità e desiderata nel ramo principale. Questo processo avviene velocemente per le correzioni meno importanti, o, nel caso di patch ampie e controverse, va avanti per anni. Per uno sviluppatore la maggior frustrazione viene dalla mancanza di comprensione di questo processo o dai tentativi di aggirarlo.

Nella speranza di ridurre questa frustrazione, questo documento spiegherà come una patch viene inserita nel kernel. Ciò che segue è un' introduzione che descrive il processo ideale. Approfondimenti verranno invece trattati più avanti.

Una patch attraversa, generalmente, le seguenti fasi:

- Progetto. In questa fase sono stabilite quelli che sono i requisiti della modifica - e come verranno soddisfatti. Il lavoro di progettazione viene spesso svolto senza coinvolgere la comunità, ma è meglio renderlo il più aperto possibile; questo può far risparmiare molto tempo evitando eventuali riprogettazioni successive.
- Prima revisione. Le patch vengono pubblicate sulle liste di discussione interessate, e gli sviluppatori in quella lista risponderanno coi loro commenti. Se si svolge correttamente, questo procedimento potrebbe far emergere problemi rilevanti in una patch.
- Revisione più ampia. Quando la patch è quasi pronta per essere inserita nel ramo principale, un manutentore importante del sottosistema dovrebbe accettarla - anche se, questa accettazione non è una garanzia che la patch arriverà nel ramo principale. La patch sarà visibile nei sorgenti del sottosistema in questione e nei sorgenti -next (descritti sotto). Quando il processo va a buon fine, questo passo porta ad una revisione più estesa della patch e alla scoperta di problemi d' integrazione con il lavoro altrui.
- Per favore, tenete da conto che la maggior parte dei manutentori ha anche un lavoro quotidiano, quindi integrare le vostre patch potrebbe non essere la loro priorità più alta. Se una vostra patch riceve dei suggerimenti su dei cambiamenti necessari, dovreste applicare quei cambiamenti o giustificare perché non sono necessari. Se la vostra patch non riceve alcuna critica ma non è stata integrata dal manutentore del driver o sottosistema, allora dovreste continuare con i necessari aggiornamenti per mantenere la patch aggiornata al kernel più recente cosicché questa possa integrarsi senza problemi; continuate ad inviare gli aggiornamenti per essere revisionati e integrati.
- Inclusione nel ramo principale. Eventualmente, una buona patch verrà inserita all' interno nel repositorio principale, gestito da Linus Torvalds. In questa fase potrebbero emergere nuovi problemi e/o commenti; è importante che lo sviluppatore sia collaborativo e che sistemi ogni questione che possa emergere.
- Rilascio stabile. Ora, il numero di utilizzatori che sono potenzialmente toccati dalla patch è aumentato, quindi, ancora una volta, potrebbero emergere nuovi problemi.

- Manutenzione di lungo periodo. Nonostante sia possibile che uno sviluppatore si dimentichi del codice dopo la sua integrazione, questo comportamento lascia una brutta impressione nella comunità di sviluppo. Integrare il codice elimina alcuni degli oneri facenti parte della manutenzione, in particolare, sistemerà le problematiche causate dalle modifiche all' API. Ma lo sviluppatore originario dovrebbe continuare ad assumersi la responsabilità per il codice se quest' ultimo continua ad essere utile nel lungo periodo.

Uno dei più grandi errori fatti dagli sviluppatori kernel (o dai loro datori di lavoro) è quello di cercare di ridurre tutta la procedura ad una singola "integrazione nel remo principale". Questo approccio inevitabilmente conduce a una condizione di frustrazione per tutti coloro che sono coinvolti.

Come le modifiche finiscono nel Kernel

Esiste una sola persona che può inserire le patch nel repositorio principale del kernel: Linus Torvalds. Ma, per esempio, di tutte le 9500 patch che entrarono nella versione 2.6.38 del kernel, solo 112 (circa l' 1,3%) furono scelte direttamente da Linus in persona. Il progetto del kernel è cresciuto fino a raggiungere una dimensione tale per cui un singolo sviluppatore non può controllare e selezionare indipendentemente ogni modifica senza essere supportato. La via scelta dagli sviluppatori per indirizzare tale crescita è stata quella di utilizzare un sistema di "sottotenenti" basato sulla fiducia.

Il codice base del kernel è spezzato in una serie si sottosistemi: rete, supporto per specifiche architetture, gestione della memoria, video e strumenti, etc. Molti sottosistemi hanno un manutentore designato: ovvero uno sviluppatore che ha piena responsabilità di tutto il codice presente in quel sottosistema. Tali manutentori di sottosistema sono i guardiani (in un certo senso) della parte di kernel che gestiscono; sono coloro che (solitamente) accetteranno una patch per l' inclusione nel ramo principale del kernel.

I manutentori di sottosistema gestiscono ciascuno la propria parte dei sorgenti del kernel, utilizzando abitualmente (ma certamente non sempre) git. Strumenti come git (e affini come quilt o mercurial) permettono ai manutentori di stilare una lista delle patch, includendo informazioni sull' autore ed altri metadati. In ogni momento, il manutentore può individuare quale patch nel sua repositorio non si trova nel ramo principale.

Quando la "finestra di integrazione" si apre, i manutentori di alto livello chiederanno a Linus di "prendere" dai loro repositori le modifiche che hanno selezionato per l' inclusione. Se Linus acconsente, il flusso di patch si convoglierà nel repositorio di quest ultimo, divenendo così parte del ramo principale del kernel. La quantità d' attenzione che Linus presta alle singole patch ricevute durante l' operazione di integrazione varia. È chiaro che, qualche volta, guardi più attentamente. Ma, come regola generale, Linus confida nel fatto che i manutentori di sottosistema non selezionino pessime patch.

I manutentori di sottosistemi, a turno, possono "prendere" patch provenienti da altri manutentori. Per esempio, i sorgenti per la rete rete sono costruiti da modifiche che si sono accumulate inizialmente nei sorgenti dedicati ai driver per dispositivi di rete, rete senza fili, ecc. Tale catena di repositori può essere più o meno lunga, benché raramente ecceda i due o tre collegamenti. Questo processo è conosciuto

come “la catena della fiducia”, perché ogni manutentore all’interno della catena si fida di coloro che gestiscono i livelli più bassi.

Chiaramente, in un sistema come questo, l’inserimento delle patch all’interno del kernel si basa sul trovare il manutentore giusto. Di norma, inviare patch direttamente a Linus non è la via giusta.

Sorgenti -next

La catena di sottosistemi guida il flusso di patch all’interno del kernel, ma solleva anche un interessante quesito: se qualcuno volesse vedere tutte le patch pronte per la prossima finestra di integrazione? Gli sviluppatori si interesseranno alle patch in sospeso per verificare che non ci siano altri conflitti di cui preoccuparsi; una modifica che, per esempio, cambia il prototipo di una funzione fondamentale del kernel andrà in conflitto con qualsiasi altra modifica che utilizzi la vecchia versione di quella funzione. Revisori e tester vogliono invece avere accesso alle modifiche nella loro totalità prima che approdino nel ramo principale del kernel. Uno potrebbe prendere le patch provenienti da tutti i sottosistemi d’interesse, ma questo sarebbe un lavoro enorme e fallace.

La risposta ci viene sotto forma di sorgenti -next, dove i sottosistemi sono raccolti per essere testati e controllati. Il più vecchio di questi sorgenti, gestito da Andrew Morton, è chiamato “-mm” (memory management, che è l’inizio di tutto). L’-mm integra patch proveniente da una lunga lista di sottosistemi; e ha, inoltre, alcune patch destinate al supporto del debugging.

Oltre a questo, -mm contiene una raccolta significativa di patch che sono state selezionate da Andrew direttamente. Queste patch potrebbero essere state inviate in una lista di discussione, o possono essere applicate ad una parte del kernel per la quale non esiste un sottosistema dedicato. Di conseguenza, -mm opera come una specie di sottosistema “ultima spiaggia”; se per una patch non esiste una via chiara per entrare nel ramo principale, allora è probabile che finirà in -mm. Le patch passate per -mm eventualmente finiranno nel sottosistema più appropriato o saranno inviate direttamente a Linus. In un tipico ciclo di sviluppo, circa il 5-10% delle patch andrà nel ramo principale attraverso -mm.

La patch -mm correnti sono disponibili nella cartella “mmotm”(-mm of the moment) all’indirizzo:

> http://www.ozlabs.org/~akpm/mmotm/

È molto probabile che l’uso dei sorgenti MMOTM diventi un’esperienza frustrante; ci sono buone probabilità che non compili nemmeno.

I sorgenti principali per il prossimo ciclo d’integrazione delle patch è linux-next, gestito da Stephen Rothwell. I sorgenti linux-next sono, per definizione, un’istantanea di come dovrà apparire il ramo principale dopo che la prossima finestra di inclusione si chiuderà. I linux-next sono annunciati sulla lista di discussione linux-kernel e linux-next nel momento in cui vengono assemblati; e possono essere scaricate da:

> http://www.kernel.org/pub/linux/kernel/next/

Linux-next è divenuto parte integrante del processo di sviluppo del kernel; tutte le patch incorporate durante una finestra di integrazione dovrebbero aver trovato

la propria strada in linux-next, a volte anche prima dell'apertura della finestra di integrazione.

Sorgenti in preparazione

Nei sorgenti del kernel esiste la cartella drivers/staging/, dove risiedono molte sotto-cartelle per i driver o i filesystem che stanno per essere aggiunti al kernel. Questi restano nella cartella drivers/staging fintanto che avranno bisogno di maggior lavoro; una volta completato, possono essere spostate all'interno del kernel nel posto più appropriato. Questo è il modo di tener traccia dei driver che non sono ancora in linea con gli standard di codifica o qualità, ma che le persone potrebbero voler usare ugualmente e tracciarne lo sviluppo.

Greg Kroah-Hartman attualmente gestisce i sorgenti in preparazione. I driver che non sono completamente pronti vengono inviati a lui, e ciascun driver avrà la propria sotto-cartella in drivers/staging/. Assieme ai file sorgenti dei driver, dovrebbe essere presente nella stessa cartella anche un file TODO. Il file TODO elenca il lavoro ancora da fare su questi driver per poter essere accettati nel kernel, e indica anche la lista di persone da inserire in copia conoscenza per ogni modifica fatta. Le regole attuali richiedono che i driver debbano, come minimo, compilare adeguatamente.

La *preparazione* può essere una via relativamente facile per inserire nuovi driver all'interno del ramo principale, dove, con un po' di fortuna, saranno notati da altri sviluppatori e migliorati velocemente. Entrare nella fase di preparazione non è però la fine della storia, infatti, il codice che si trova nella cartella staging che non mostra regolari progressi potrebbe essere rimosso. Le distribuzioni, inoltre, tendono a dimostrarsi relativamente riluttanti nell'attivare driver in preparazione. Quindi lo preparazione è, nel migliore dei casi, una tappa sulla strada verso il divenire un driver del ramo principale.

Strumenti

Come è possibile notare dal testo sopra, il processo di sviluppo del kernel dipende pesantemente dalla capacità di guidare la raccolta di patch in diverse direzioni. L'intera cosa non funzionerebbe se non venisse svolta con l'uso di strumenti appropriati e potenti. Spiegare l'uso di tali strumenti non è lo scopo di questo documento, ma c'è spazio per alcuni consigli.

In assoluto, nella comunità del kernel, predomina l'uso di git come sistema di gestione dei sorgenti. Git è una delle diverse tipologie di sistemi distribuiti di controllo versione che sono stati sviluppati nella comunità del software libero. Esso è calibrato per lo sviluppo del kernel, e si comporta abbastanza bene quando ha a che fare con repositori grandi e con un vasto numero di patch. Git ha inoltre la reputazione di essere difficile da imparare e utilizzare, benché stia migliorando. Agli sviluppatori del kernel viene richiesta un po' di familiarità con git; anche se non lo utilizzano per il proprio lavoro, hanno bisogno di git per tenersi al passo con il lavoro degli altri sviluppatori (e con il ramo principale).

Git è ora compreso in quasi tutte le distribuzioni Linux. Esiste una sito che potete consultare:

http://git-scm.com/

Qui troverete i riferimenti alla documentazione e alle guide passo-passo.

Tra gli sviluppatori Kernel che non usano git, la scelta alternativa più popolare è quasi sicuramente Mercurial:

http://www.selenic.com/mercurial/

Mercurial condivide diverse caratteristiche con git, ma fornisce un'interfaccia che potrebbe risultare più semplice da utilizzare.

L'altro strumento che vale la pena conoscere è Quilt:

http://savannah.nongnu.org/projects/quilt/

Quilt è un sistema di gestione delle patch, piuttosto che un sistema di gestione dei sorgenti. Non mantiene uno storico degli eventi; ma piuttosto è orientato verso il tracciamento di uno specifico insieme di modifiche rispetto ad un codice in evoluzione. Molti dei più grandi manutentori di sottosistema utilizzano quilt per gestire le patch che dovrebbero essere integrate. Per la gestione di certe tipologie di sorgenti (-mm, per esempio), quilt è il miglior strumento per svolgere il lavoro.

Liste di discussione

Una grossa parte del lavoro di sviluppo del Kernel Linux viene svolto tramite le liste di discussione. È difficile essere un membro della comunità pienamente coinvolto se non si partecipa almeno ad una lista da qualche parte. Ma, le liste di discussione di Linux rappresentano un potenziale problema per gli sviluppatori, che rischiano di venir sepolti da un mare di email, restare incagliati nelle convenzioni in vigore nelle liste Linux, o entrambi.

Molte delle liste di discussione del Kernel girano su vger.kernel.org; l'elenco principale lo si trova sul sito:

http://vger.kernel.org/vger-lists.html

Esistono liste gestite altrove; un certo numero di queste sono in lists.redhat.com.

La lista di discussione principale per lo sviluppo del kernel è, ovviamente, linux-kernel. Questa lista è un luogo ostile dove trovarsi; i volumi possono raggiungere i 500 messaggi al giorno, la quantità di "rumore" è elevata, la conversazione può essere strettamente tecnica e i partecipanti non sono sempre preoccupati di mostrare un alto livello di educazione. Ma non esiste altro luogo dove la comunità di sviluppo del kernel si unisce per intero; gli sviluppatori che evitano tale lista si perderanno informazioni importanti.

Ci sono alcuni consigli che possono essere utili per sopravvivere a linux-kernel:

- Tenete la lista in una cartella separata, piuttosto che inserirla nella casella di posta principale. Così da essere in grado di ignorare il flusso di mail per un certo periodo di tempo.
- Non cercate di seguire ogni conversazione - nessuno lo fa. È importante filtrare solo gli argomenti d'interesse (sebbene va notato che le conversazioni di lungo periodo possono deviare dall'argomento originario senza cambiare il titolo della mail) e le persone che stanno partecipando.

- Non alimentate i troll. Se qualcuno cerca di creare nervosismo, ignoratelo.
- Quando rispondete ad una mail linux-kernel (o ad altre liste) mantenete tutti i Cc:. In assenza di importanti motivazioni (come una richiesta esplicita), non dovreste mai togliere destinatari. Assicuratevi sempre che la persona alla quale state rispondendo sia presente nella lista Cc. Questa usanza fa si che divenga inutile chiedere esplicitamente di essere inseriti in copia nel rispondere al vostro messaggio.
- Cercate nell' archivio della lista (e nella rete nella sua totalità) prima di far domande. Molti sviluppatori possono divenire impazienti con le persone che chiaramente non hanno svolto i propri compiti a casa.
- Evitate il *top-posting* (cioè la pratica di mettere la vostra risposta sopra alla frase alla quale state rispondendo). Ciò renderebbe la vostra risposta difficile da leggere e genera scarsa impressione.
- Chiedete nella lista di discussione corretta. Linux-kernel può essere un punto di incontro generale, ma non è il miglior posto dove trovare sviluppatori da tutti i sottosistemi.

Infine, la ricerca della corretta lista di discussione è uno degli errori più comuni per gli sviluppatori principianti. Qualcuno che pone una domanda relativa alla rete su linux-kernel riceverà quasi certamente il suggerimento di chiedere sulla lista netdev, che è la lista frequentata dagli sviluppatori di rete. Ci sono poi altre liste per i sottosistemi SCSI, video4linux, IDE, filesystem, etc. Il miglior posto dove cercare una lista di discussione è il file MAINTAINERS che si trova nei sorgenti del kernel.

Iniziare con lo sviluppo del Kernel

Sono comuni le domande sul come iniziare con lo sviluppo del kernel - sia da singole persone che da aziende. Altrettanto comuni sono i passi falsi che rendono l' inizio di tale relazione più difficile di quello che dovrebbe essere.

Le aziende spesso cercano di assumere sviluppatori noti per creare un gruppo di sviluppo iniziale. Questo, in effetti, può essere una tecnica efficace. Ma risulta anche essere dispendiosa e non va ad accrescere il bacino di sviluppatori kernel con esperienza. È possibile anche "portare a casa" sviluppatori per accelerare lo sviluppo del kernel, dando comunque all' investimento un po' di tempo. Prendersi questo tempo può fornire al datore di lavoro un gruppo di sviluppatori che comprendono sia il kernel che l' azienda stessa, e che possono supportare la formazione di altre persone. Nel medio periodo, questa è spesso uno delle soluzioni più proficue.

I singoli sviluppatori sono spesso, comprensibilmente, una perdita come punto di partenza. Iniziare con un grande progetto può rivelarsi intimidatorio; spesso all' inizio si vuole solo verificare il terreno con qualcosa di piccolo. Questa è una delle motivazioni per le quali molti sviluppatori saltano alla creazione di patch che vanno a sistemare errori di battitura o problematiche minori legate allo stile del codice. Sfortunatamente, tali patch creano un certo livello di rumore che distrae l' intera comunità di sviluppo, quindi, sempre di più, esse vengono degradate. I nuovi sviluppatori che desiderano presentarsi alla comunità non riceveranno l' accoglienza che vorrebbero con questi mezzi.

Andrew Morton da questo consiglio agli aspiranti sviluppatori kernel

```
Il primo progetto per un neofita del kernel dovrebbe essere
sicuramente quello di "assicurarsi che il kernel funzioni alla
perfezione sempre e su tutte le macchine sulle quali potete stendere
la vostra mano".  Solitamente il modo per fare ciò è quello di
collaborare con gli altri nel sistemare le cose (questo richiede
persistenza!) ma va bene - è parte dello sviluppo kernel.
```

(http://lwn.net/Articles/283982/).

In assenza di problemi ovvi da risolvere, si consiglia agli sviluppatori di consultare, in generale, la lista di regressioni e di bachi aperti. Non c'è mai carenza di problematiche bisognose di essere sistemate; accollandosi tali questioni gli sviluppatori accumuleranno esperienza con la procedura, ed allo stesso tempo, aumenteranno la loro rispettabilità all'interno della comunità di sviluppo.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/3.Early-stage.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

I primi passi della pianificazione

Osservando un progetto di sviluppo per il kernel Linux, si potrebbe essere tentati dal saltare tutto e iniziare a codificare. Tuttavia, come ogni progetto significativo, molta della preparazione per giungere al successo viene fatta prima che una sola linea di codice venga scritta. Il tempo speso nella pianificazione e la comunicazione può far risparmiare molto tempo in futuro.

Specificare il problema

Come qualsiasi progetto ingegneristico, un miglioramento del kernel di successo parte con una chiara descrizione del problema da risolvere. In alcuni casi, questo passaggio è facile: ad esempio quando un driver è richiesto per un particolare dispositivo. In altri casi invece, si tende a confondere il problema reale con le soluzioni proposte e questo può portare all'emergere di problemi.

Facciamo un esempio: qualche anno fa, gli sviluppatori che lavoravano con linux audio cercarono un modo per far girare le applicazioni senza dropouts o altri artefatti dovuti all'eccessivo ritardo nel sistema. La soluzione alla quale giunsero fu un modulo del kernel destinato ad agganciarsi al framework Linux Security Module (LSM); questo modulo poteva essere configurato per dare ad una specifica applicazione accesso allo schedulatore *realtime*. Tale modulo fu implementato e inviato nella lista di discussione linux-kernel, dove incontrò subito dei problemi.

Per gli sviluppatori audio, questo modulo di sicurezza era sufficiente a risolvere il loro problema nell'immediato. Per l'intera comunità kernel, invece, era un uso improprio del framework LSM (che non è progettato per conferire privilegi a processi che altrimenti non avrebbero potuto ottenerli) e un rischio per la stabilità del sistema. Le loro soluzioni di punta nel breve periodo, comportavano un accesso alla schedulazione realtime attraverso il meccanismo rlimit, e nel lungo periodo un costante lavoro nella riduzione dei ritardi.

La comunità audio, comunque, non poteva vedere al di là della singola soluzione che avevano implementato; erano riluttanti ad accettare alternative. Il conseguente dissenso lasciò in quegli sviluppatori un senso di disillusione nei confronti dell'intero processo di sviluppo; uno di loro scrisse questo messaggio:

> Ci sono numerosi sviluppatori del kernel Linux davvero bravi, ma rischiano di restare sovrastati da una vasta massa di stolti arroganti. Cercare di comunicare le richieste degli utenti a queste persone è una perdita di tempo. Loro sono troppo "intelligenti" per stare ad ascoltare dei poveri mortali.
>
> (http://lwn.net/Articles/131776/).

La realtà delle cose fu differente; gli sviluppatori del kernel erano molto più preoccupati per la stabilità del sistema, per la manutenzione di lungo periodo e cercavano la giusta soluzione alla problematica esistente con uno specifico modulo. La morale della storia è quella di concentrarsi sul problema - non su di una specifica soluzione- e di discuterne con la comunità di sviluppo prima di investire tempo nella scrittura del codice.

Quindi, osservando un progetto di sviluppo del kernel, si dovrebbe rispondere a questa lista di domande:

- Qual'è, precisamente, il problema che dev'essere risolto?
- Chi sono gli utenti coinvolti da tal problema? A quale caso dovrebbe essere indirizzata la soluzione?
- In che modo il kernel risulta manchevole nell'indirizzare il problema in questione?

Solo dopo ha senso iniziare a considerare le possibili soluzioni.

Prime discussioni

Quando si pianifica un progetto di sviluppo per il kernel, sarebbe quanto meno opportuno discuterne inizialmente con la comunità prima di lanciarsi nell'implementazione. Una discussione preliminare può far risparmiare sia tempo che problemi in svariati modi:

- Potrebbe essere che il problema sia già stato risolto nel kernel in una maniera che non avete ancora compreso. Il kernel Linux è grande e ha una serie di funzionalità e capacità che non sono scontate nell'immediato. Non tutte le capacità del kernel sono documentate così bene come ci piacerebbe, ed è facile perdersi qualcosa. Il vostro autore ha assistito alla pubblicazione di un driver intero che duplica un altro driver esistente di cui il nuovo autore era ignaro.

Il codice che rinnova ingranaggi già esistenti non è soltanto dispendioso; non verrà nemmeno accettato nel ramo principale del kernel.

- Potrebbero esserci proposte che non sono considerate accettabili per l'integrazione all'interno del ramo principale. È meglio affrontarle prima di scrivere il codice.
- È possibile che altri sviluppatori abbiano pensato al problema; potrebbero avere delle idee per soluzioni migliori, e potrebbero voler contribuire alla loro creazione.

Anni di esperienza con la comunità di sviluppo del kernel hanno impartito una chiara lezione: il codice per il kernel che è pensato e sviluppato a porte chiuse, inevitabilmente, ha problematiche che si rivelano solo quando il codice viene rilasciato pubblicamente. Qualche volta tali problemi sono importanti e richiedono mesi o anni di sforzi prima che il codice possa raggiungere gli standard richiesti della comunità. Alcuni esempi possono essere:

- La rete Devicescape è stata creata e implementata per sistemi monoprocessore. Non avrebbe potuto essere inserita nel ramo principale fino a che non avesse supportato anche i sistemi multi-processore. Riadattare i meccanismi di sincronizzazione e simili è un compito difficile; come risultato, l'inserimento di questo codice (ora chiamato mac80211) fu rimandato per più di un anno.
- Il filesystem Reiser4 include una seria di funzionalità che, secondo l'opinione degli sviluppatori principali del kernel, avrebbero dovuto essere implementate a livello di filesystem virtuale. Comprende anche funzionalità che non sono facilmente implementabili senza esporre il sistema al rischio di uno stallo. La scoperta tardiva di questi problemi - e il diniego a risolverne alcuni - ha avuto come conseguenza il fatto che Raiser4 resta fuori dal ramo principale del kernel.
- Il modulo di sicurezza AppArmor utilizzava strutture dati del filesystem virtuale interno in modi che sono stati considerati rischiosi e inattendibili. Questi problemi (tra le altre cose) hanno tenuto AppArmor fuori dal ramo principale per anni.

Ciascuno di questi casi è stato un travaglio e ha richiesto del lavoro straordinario, cose che avrebbero potuto essere evitate con alcune "chiacchierate" preliminari con gli sviluppatori kernel.

Con chi parlare?

Quando gli sviluppatori hanno deciso di rendere pubblici i propri progetti, la domanda successiva sarà: da dove partiamo? La risposta è quella di trovare la giusta lista di discussione e il giusto manutentore. Per le liste di discussione, il miglior approccio è quello di cercare la lista più adatta nel file MAINTAINERS. Se esiste una lista di discussione di sottosistema, è preferibile pubblicare lì piuttosto che sulla lista di discussione generale del kernel Linux; avrete maggiori probabilità di trovare sviluppatori con esperienza sul tema, e l'ambiente che troverete potrebbe essere più incoraggiante.

Trovare manutentori può rivelarsi un po'difficoltoso. Ancora, il file MAINTAINERS

è il posto giusto da dove iniziare. Il file potrebbe non essere sempre aggiornato, inoltre, non tutti i sottosistemi sono rappresentati qui. Coloro che sono elencati nel file MAINTAINERS potrebbero, in effetti, non essere le persone che attualmente svolgono quel determinato ruolo. Quindi, quando c'è un dubbio su chi contattare, un trucco utile è quello di usare git (git log in particolare) per vedere chi attualmente è attivo all'interno del sottosistema interessato. Controllate chi sta scrivendo le patch, e chi, se non ci fosse nessuno, sta aggiungendo la propria firma (Signed-off-by) a quelle patch. Quelle sono le persone maggiormente qualificate per aiutarvi con lo sviluppo di nuovo progetto.

Il compito di trovare il giusto manutentore, a volte, è una tale sfida che ha spinto gli sviluppatori del kernel a scrivere uno script che li aiutasse in questa ricerca:

```
.../scripts/get_maintainer.pl
```

Se questo script viene eseguito con l'opzione "-f"ritornerà il manutentore(i) attuale per un dato file o cartella. Se viene passata una patch sulla linea di comando, lo script elencherà i manutentori che dovrebbero riceverne una copia. Ci sono svariate opzioni che regolano quanto a fondo get_maintainer.pl debba cercare i manutentori; siate quindi prudenti nell'utilizzare le opzioni più aggressive poiché potreste finire per includere sviluppatori che non hanno un vero interesse per il codice che state modificando.

Se tutto ciò dovesse fallire, parlare con Andrew Morton potrebbe essere un modo efficace per capire chi è il manutentore di un dato pezzo di codice.

Quando pubblicare

Se potete, pubblicate i vostri intenti durante le fasi preliminari, sarà molto utile. Descrivete il problema da risolvere e ogni piano che è stato elaborato per l'implementazione. Ogni informazione fornita può aiutare la comunità di sviluppo a fornire spunti utili per il progetto.

Un evento che potrebbe risultare scoraggiate e che potrebbe accadere in questa fase non è il ricevere una risposta ostile, ma, invece, ottenere una misera o inesistente reazione. La triste verità è che: (1) gli sviluppatori del kernel tendono ad essere occupati, (2) ci sono tante persone con grandi progetti e poco codice (o anche solo la prospettiva di avere un codice) a cui riferirsi e (3) nessuno è obbligato a revisionare o a fare osservazioni in merito ad idee pubblicate da altri. Oltre a questo, progetti di alto livello spesso nascondono problematiche che si rivelano solo quando qualcuno cerca di implementarle; per questa ragione gli sviluppatori kernel preferirebbero vedere il codice.

Quindi, se una richiesta pubblica di commenti riscuote poco successo, non pensate che ciò significhi che non ci sia interesse nel progetto. Sfortunatamente, non potete nemmeno assumere che non ci siano problemi con la vostra idea. La cosa migliore da fare in questa situazione è quella di andare avanti e tenere la comunità informata mentre procedete.

Ottenere riscontri ufficiali

Se il vostro lavoro è stato svolto in un ambiente aziendale - come molto del lavoro fatto su Linux - dovete, ovviamente, avere il permesso dei dirigenti prima che possiate pubblicare i progetti, o il codice aziendale, su una lista di discussione pubblica. La pubblicazione di codice che non è stato rilascio espressamente con licenza GPL-compatibile può rivelarsi problematico; prima la dirigenza, e il personale legale, troverà una decisione sulla pubblicazione di un progetto, meglio sarà per tutte le persone coinvolte.

A questo punto, alcuni lettori potrebbero pensare che il loro lavoro sul kernel è preposto a supportare un prodotto che non è ancora ufficialmente riconosciuto. Rivelare le intenzioni dei propri datori di lavori in una lista di discussione pubblica potrebbe non essere una soluzione valida. In questi casi, vale la pena considerare se la segretezza sia necessaria o meno; spesso non c'è una reale necessità di mantenere chiusi i progetti di sviluppo.

Detto ciò, ci sono anche casi dove l'azienda legittimamente non può rivelare le proprie intenzioni in anticipo durante il processo di sviluppo. Le aziende che hanno sviluppatori kernel esperti possono scegliere di procedere a carte coperte partendo dall'assunto che saranno in grado di evitare, o gestire, in futuro, eventuali problemi d'integrazione. Per le aziende senza questo tipo di esperti, la migliore opzione è spesso quella di assumere uno sviluppatore esterno che revisioni i progetti con un accordo di segretezza. La Linux Foundation applica un programma di NDA creato appositamente per aiutare le aziende in questa particolare situazione; potrete trovare più informazioni sul sito:

http://www.linuxfoundation.org/en/NDA_program

Questa tipologia di revisione è spesso sufficiente per evitare gravi problemi senza che sia richiesta l'esposizione pubblica del progetto.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/4.Coding.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Scrivere codice corretto

Nonostante ci sia molto da dire sul processo di creazione, sulla sua solidità e sul suo orientamento alla comunità, la prova di ogni progetto di sviluppo del kernel si trova nel codice stesso. È il codice che sarà esaminato dagli altri sviluppatori ed inserito (o no) nel ramo principale. Quindi è la qualità di questo codice che determinerà il successo finale del progetto.

Questa sezione esaminerà il processo di codifica. Inizieremo con uno sguardo sulle diverse casistiche nelle quali gli sviluppatori kernel possono sbagliare. Poi,

l'attenzione si sposterà verso "il fare le cose correttamente" e sugli strumenti che possono essere utili in questa missione.

Trappole

Lo stile del codice

Il kernel ha da tempo delle norme sullo stile di codifica che sono descritte in *Stile del codice per il kernel Linux*. Per la maggior parte del tempo, la politica descritta in quel file è stata praticamente informativa. Ne risulta che ci sia una quantità sostanziale di codice nel kernel che non rispetta le linee guida relative allo stile. La presenza di quel codice conduce a due distinti pericoli per gli sviluppatori kernel.

Il primo di questi è credere che gli standard di codifica del kernel non sono importanti e possono non essere applicati. La verità è che aggiungere nuovo codice al kernel è davvero difficile se questo non rispetta le norme; molti sviluppatori richiederanno che il codice sia riformulato prima che anche solo lo revisionino. Una base di codice larga quanto il kernel richiede una certa uniformità, in modo da rendere possibile per gli sviluppatori una comprensione veloce di ogni sua parte. Non ci sono, quindi, più spazi per un codice formattato alla carlona.

Occasionalmente, lo stile di codifica del kernel andrà in conflitto con lo stile richiesto da un datore di lavoro. In alcuni casi, lo stile del kernel dovrà prevalere prima che il codice venga inserito. Mettere il codice all'interno del kernel significa rinunciare a un certo grado di controllo in differenti modi - incluso il controllo sul come formattare il codice.

L'altra trappola è quella di pensare che il codice già presente nel kernel abbia urgentemente bisogno di essere sistemato. Gli sviluppatori potrebbero iniziare a generare patch che correggono lo stile come modo per prendere famigliarità con il processo, o come modo per inserire i propri nomi nei changelog del kernel – o entrambe. La comunità di sviluppo vede un attività di codifica puramente correttiva come "rumore"; queste attività riceveranno una fredda accoglienza. Di conseguenza è meglio evitare questo tipo di patch. Mentre si lavora su un pezzo di codice è normale correggerne anche lo stile, ma le modifiche di stile non dovrebbero essere fatte fini a se stesse.

Il documento sullo stile del codice non dovrebbe essere letto come una legge assoluta che non può mai essere trasgredita. Se c'è un a buona ragione (per esempio, una linea che diviene poco leggibile se divisa per rientrare nel limite di 80 colonne), fatelo e basta.

Notate che potete utilizzare lo strumento "clang-format" per aiutarvi con le regole, per una riformattazione automatica e veloce del vostro codice e per revisionare interi file per individuare errori nello stile di codifica, refusi e possibili miglioramenti. Inoltre è utile anche per classificare gli `#includes`, per allineare variabili/macro, per testi derivati ed altri compiti del genere. Consultate il file *clang-format* per maggiori dettagli

Livelli di astrazione

I professori di Informatica insegnano ai propri studenti a fare ampio uso dei livelli di astrazione nel nome della flessibilità e del nascondere informazioni. Certo il kernel fa un grande uso dell'astrazione; nessun progetto con milioni di righe di codice potrebbe fare altrimenti e sopravvivere. Ma l'esperienza ha dimostrato che un'eccessiva o prematura astrazione può rivelarsi dannosa al pari di una prematura ottimizzazione. L'astrazione dovrebbe essere usata fino al livello necessario e non oltre.

Ad un livello base, considerate una funzione che ha un argomento che viene sempre impostato a zero da tutti i chiamanti. Uno potrebbe mantenere quell'argomento nell'eventualità qualcuno volesse sfruttare la flessibilità offerta. In ogni caso, tuttavia, ci sono buone possibilità che il codice che va ad implementare questo argomento aggiuntivo, sia stato rotto in maniera sottile, in un modo che non è mai stato notato - perché non è mai stato usato. Oppure, quando sorge la necessità di avere più flessibilità, questo argomento non la fornisce in maniera soddisfacente. Gli sviluppatori di Kernel, sottopongono costantemente patch che vanno a rimuovere gli argomenti inutilizzate; anche se, in generale, non avrebbero dovuto essere aggiunti.

I livelli di astrazione che nascondono l'accesso all'hardware - spesso per poter usare dei driver su diversi sistemi operativi - vengono particolarmente disapprovati. Tali livelli oscurano il codice e possono peggiorare le prestazioni; essi non appartengono al kernel Linux.

D'altro canto, se vi ritrovate a dover copiare una quantità significativa di codice proveniente da un altro sottosistema del kernel, è tempo di chiedersi se, in effetti, non avrebbe più senso togliere parte di quel codice e metterlo in una libreria separata o di implementare quella funzionalità ad un livello più elevato. Non c'è utilità nel replicare lo stesso codice per tutto il kernel.

#ifdef e l'uso del preprocessore in generale

Il preprocessore C sembra essere una fonte di attrazione per qualche programmatore C, che ci vede una via per ottenere una grande flessibilità all'interno di un file sorgente. Ma il preprocessore non è scritto in C, e un suo massiccio impiego conduce a un codice che è molto più difficile da leggere per gli altri e che rende più difficile il lavoro di verifica del compilatore. L'uso eccessivo del preprocessore è praticamente sempre il segno di un codice che necessita di un certo lavoro di pulizia.

La compilazione condizionata con #ifdef è, in effetti, un potente strumento, ed esso viene usato all'interno del kernel. Ma esiste un piccolo desiderio: quello di vedere il codice coperto solo da una leggera spolverata di blocchi #ifdef. Come regola generale, quando possibile, l'uso di #ifdef dovrebbe essere confinato nei file d'intestazione. Il codice compilato condizionatamente può essere confinato a funzioni tali che, nel caso in cui il codice non deve essere presente, diventano vuote. Il compilatore poi ottimizzerà la chiamata alla funzione vuota rimuovendola. Il risultato è un codice molto più pulito, più facile da seguire.

Le macro del preprocessore C presentano una serie di pericoli, inclusi valutazioni multiple di espressioni che hanno effetti collaterali e non garantiscono una si-

curezza rispetto ai tipi. Se siete tentati dal definire una macro, considerate l'idea di creare invece una funzione inline. Il codice che ne risulterà sarà lo stesso, ma le funzioni inline sono più leggibili, non considerano i propri argomenti più volte, e permettono al compilatore di effettuare controlli sul tipo degli argomenti e del valore di ritorno.

Funzioni inline

Comunque, anche le funzioni inline hanno i loro pericoli. I programmatori potrebbero innamorarsi dell'efficienza percepita derivata dalla rimozione di una chiamata a funzione. Queste funzioni, tuttavia, possono ridurre le prestazioni. Dato che il loro codice viene replicato ovunque vi sia una chiamata ad esse, si finisce per gonfiare le dimensioni del kernel compilato. Questi, a turno, creano pressione sulla memoria cache del processore, e questo può causare rallentamenti importanti. Le funzioni inline, di norma, dovrebbero essere piccole e usate raramente. Il costo di una chiamata a funzione, dopo tutto, non è così alto; la creazione di molte funzioni inline è il classico esempio di un' ottimizzazione prematura.

In generale, i programmatori del kernel ignorano gli effetti della cache a loro rischio e pericolo. Il classico compromesso tempo/spazio teorizzato all' inizio delle lezioni sulle strutture dati spesso non si applica all' hardware moderno. Lo spazio *è* tempo, in questo senso un programma più grande sarà più lento rispetto ad uno più compatto.

I compilatori più recenti hanno preso un ruolo attivo nel decidere se una data funzione deve essere resa inline oppure no. Quindi l' uso indiscriminato della parola chiave "inline"potrebbe non essere non solo eccessivo, ma anche irrilevante.

Sincronizzazione

Nel maggio 2006, il sistema di rete "Devicescape" fu rilasciato in pompa magna sotto la licenza GPL e reso disponibile per la sua inclusione nella ramo principale del kernel. Questa donazione fu una notizia bene accolta; il supporto per le reti senza fili era considerata, nel migliore dei casi, al di sotto degli standard; il sistema Deviscape offrì la promessa di una risoluzione a tale situazione. Tuttavia, questo codice non fu inserito nel ramo principale fino al giugno del 2007 (2.6.22). Cosa accadde?

Quel codice mostrava numerosi segnali di uno sviluppo in azienda avvenuto a porte chiuse. Ma in particolare, un grosso problema fu che non fu progettato per girare in un sistema multiprocessore. Prima che questo sistema di rete (ora chiamato mac80211) potesse essere inserito, fu necessario un lavoro sugli schemi di sincronizzazione.

Una volta, il codice del kernel Linux poteva essere sviluppato senza pensare ai problemi di concorrenza presenti nei sistemi multiprocessore. Ora, comunque, questo documento è stato scritto su di un portatile dual-core. Persino su sistemi a singolo processore, il lavoro svolto per incrementare la capacità di risposta aumenterà il livello di concorrenza interno al kernel. I giorni nei quali il codice poteva essere scritto senza pensare alla sincronizzazione sono da passati tempo.

Ogni risorsa (strutture dati, registri hardware, etc.) ai quali si potrebbe avere accesso simultaneo da più di un thread deve essere sincronizzato. Il nuovo codice dovrebbe essere scritto avendo tale accortezza in testa; riadattare la sincronizzazione a posteriori è un compito molto più difficile. Gli sviluppatori del kernel dovrebbero prendersi il tempo di comprendere bene le primitive di sincronizzazione, in modo da sceglier lo strumento corretto per eseguire un compito. Il codice che presenta una mancanza di attenzione alla concorrenza avrà un percorso difficile all'interno del ramo principale.

Regressioni

Vale la pena menzionare un ultimo pericolo: potrebbe rivelarsi accattivante l'idea di eseguire un cambiamento (che potrebbe portare a grandi miglioramenti) che porterà ad alcune rotture per gli utenti esistenti. Questa tipologia di cambiamento è chiamata "regressione", e le regressioni son diventate mal viste nel ramo principale del kernel. Con alcune eccezioni, i cambiamenti che causano regressioni saranno fermati se quest'ultime non potranno essere corrette in tempo utile. È molto meglio quindi evitare la regressione fin dall'inizio.

Spesso si è argomentato che una regressione può essere giustificata se essa porta risolve più problemi di quanti non ne crei. Perché, dunque, non fare un cambiamento se questo porta a nuove funzionalità a dieci sistemi per ognuno dei quali esso determina una rottura? La migliore risposta a questa domanda ci è stata fornita da Linus nel luglio 2007:

::

> Dunque, noi non sistemiamo bachi introducendo nuovi problemi. Quella via nasconde insidie, e nessuno può sapere del tutto se state facendo dei progressi reali. Sono due passi avanti e uno indietro, oppure un passo avanti e due indietro?

(http://lwn.net/Articles/243460/).

Una particolare tipologia di regressione mal vista consiste in una qualsiasi sorta di modifica all'ABI dello spazio utente. Una volta che un'interfaccia viene esportata verso lo spazio utente, dev'essere supportata all'infinito. Questo fatto rende la creazione di interfacce per lo spazio utente particolarmente complicato: dato che non possono venir cambiate introducendo incompatibilità, esse devono essere fatte bene al primo colpo. Per questa ragione sono sempre richieste: ampie riflessioni, documentazione chiara e ampie revisioni dell'interfaccia verso lo spazio utente.

Strumenti di verifica del codice

Almeno per ora la scrittura di codice priva di errori resta un ideale irraggiungibile ai più. Quello che speriamo di poter fare, tuttavia, è trovare e correggere molti di questi errori prima che il codice entri nel ramo principale del kernel. A tal scopo gli sviluppatori del kernel devono mettere insieme una schiera impressionante di strumenti che possano localizzare automaticamente un'ampia varietà di problemi. Qualsiasi problema trovato dal computer è un problema che non affliggerà l'utente

in seguito, ne consegue che gli strumenti automatici dovrebbero essere impiegati ovunque possibile.

Il primo passo consiste semplicemente nel fare attenzione agli avvertimenti proveniente dal compilatore. Versioni moderne di gcc possono individuare (e segnalare) un gran numero di potenziali errori. Molto spesso, questi avvertimenti indicano problemi reali. Di regola, il codice inviato per la revisione non dovrebbe produrre nessun avvertimento da parte del compilatore. Per mettere a tacere gli avvertimenti, cercate di comprenderne le cause reali e cercate di evitare le "riparazioni" che fan sparire l'avvertimento senza però averne trovato la causa.

Tenete a mente che non tutti gli avvertimenti sono disabilitati di default. Costruite il kernel con "make EXTRA_CFLAGS=-W" per ottenerli tutti.

Il kernel fornisce differenti opzioni che abilitano funzionalità di debugging; molti di queste sono trovano all'interno del sotto menu "kernel hacking". La maggior parte di queste opzioni possono essere attivate per qualsiasi kernel utilizzato per lo sviluppo o a scopo di test. In particolare dovreste attivare:

- ENABLE_MUST_CHECK e FRAME_WARN per ottenere degli avvertimenti dedicati a problemi come l'uso di interfacce deprecate o l'ignorare un importante valore di ritorno di una funzione. Il risultato generato da questi avvertimenti può risultare verboso, ma non bisogna preoccuparsi per gli avvertimenti provenienti da altre parti del kernel.
- DEBUG_OBJECTS aggiungerà un codice per tracciare il ciclo di vita di diversi oggetti creati dal kernel e avvisa quando qualcosa viene eseguito fuori controllo. Se state aggiungendo un sottosistema che crea (ed esporta) oggetti complessi propri, considerate l'aggiunta di un supporto al debugging dell'oggetto.
- DEBUG_SLAB può trovare svariati errori di uso e di allocazione di memoria; esso dovrebbe esser usato dalla maggior parte dei kernel di sviluppo.
- DEBUG_SPINLOCK, DEBUG_ATOMIC_SLEEP, e DEBUG_MUTEXES troveranno un certo numero di errori comuni di sincronizzazione.

Esistono ancora delle altre opzioni di debugging, di alcune di esse discuteremo qui sotto. Alcune di esse hanno un forte impatto e non dovrebbero essere usate tutte le volte. Ma qualche volta il tempo speso nell'capire le opzioni disponibili porterà ad un risparmio di tempo nel breve termine.

Uno degli strumenti di debugging più tosti è il *locking checker*, o "lockdep". Questo strumento traccerà qualsiasi acquisizione e rilascio di ogni *lock* (spinlock o mutex) nel sistema, l'ordine con il quale i *lock* sono acquisiti in relazione l'uno con l'altro, l'ambiente corrente di interruzione, eccetera. Inoltre esso può assicurare che i *lock* vengano acquisiti sempre nello stesso ordine, che le stesse assunzioni sulle interruzioni si applichino in tutte le occasioni, e così via. In altre parole, lockdep può scovare diversi scenari nei quali il sistema potrebbe, in rari casi, trovarsi in stallo. Questa tipologia di problema può essere grave (sia per gli sviluppatori che per gli utenti) in un sistema in uso; lockdep permette di trovare tali problemi automaticamente e in anticipo.

In qualità di programmatore kernel diligente, senza dubbio, dovrete controllare il valore di ritorno di ogni operazione (come l'allocazione della memoria) poiché esso potrebbe fallire. Il nocciolo della questione è che i percorsi di gestione degli

errori, con grande probabilità, non sono mai stati collaudati del tutto. Il codice collaudato tende ad essere codice bacato; potrete quindi essere più a vostro agio con il vostro codice se tutti questi percorsi fossero stati verificati un po' di volte.

Il kernel fornisce un framework per l'inserimento di fallimenti che fa esattamente al caso, specialmente dove sono coinvolte allocazioni di memoria. Con l'opzione per l'inserimento dei fallimenti abilitata, una certa percentuale di allocazione di memoria sarà destinata al fallimento; questi fallimenti possono essere ridotti ad uno specifico pezzo di codice. Procedere con l'inserimento dei fallimenti attivo permette al programmatore di verificare come il codice risponde quando le cose vanno male. Consultate: Documentation/fault-injection/fault-injection.rst per avere maggiori informazioni su come utilizzare questo strumento.

Altre tipologie di errori possono essere riscontrati con lo strumento di analisi statica "sparse". Con Sparse, il programmatore può essere avvisato circa la confusione tra gli indirizzi dello spazio utente e dello spazio kernel, un miscuglio fra quantità big-endian e little-endian, il passaggio di un valore intero dove ci sia aspetta un gruppo di flag, e così via. Sparse deve essere installato separatamente (se il vostra distribuzione non lo prevede, potete trovarlo su https://sparse.wiki.kernel.org/index.php/Main_Page); può essere attivato sul codice aggiungendo "C=1" al comando make.

Lo strumento "Coccinelle"(http://coccinelle.lip6.fr/) è in grado di trovare una vasta varietà di potenziali problemi di codifica; e può inoltre proporre soluzioni per risolverli. Un buon numero di "patch semantiche" per il kernel sono state preparate nella cartella scripts/coccinelle; utilizzando "make coccicheck" esso percorrerà tali patch semantiche e farà rapporto su qualsiasi problema trovato. Per maggiori informazioni, consultate Documentation/dev-tools/coccinelle.rst.

Altri errori di portabilità sono meglio scovati compilando il vostro codice per altre architetture. Se non vi accade di avere un sistema S/390 o una scheda di sviluppo Blackfin sotto mano, potete comunque continuare la fase di compilazione. Un vasto numero di cross-compilatori per x86 possono essere trovati al sito:

http://www.kernel.org/pub/tools/crosstool/

Il tempo impiegato nell'installare e usare questi compilatori sarà d'aiuto nell'evitare situazioni imbarazzanti nel futuro.

Documentazione

La documentazione è spesso stata più un'eccezione che una regola nello sviluppo del kernel. Nonostante questo, un'adeguata documentazione aiuterà a facilitare l'inserimento di nuovo codice nel kernel, rende la vita più facile per gli altri sviluppatori e sarà utile per i vostri utenti. In molti casi, la documentazione è divenuta sostanzialmente obbligatoria.

La prima parte di documentazione per qualsiasi patch è il suo changelog. Questi dovrebbero descrivere le problematiche risolte, la tipologia di soluzione, le persone che lavorano alla patch, ogni effetto rilevante sulle prestazioni e tutto ciò che può servire per la comprensione della patch. Assicuratevi che il changelog dica *perché*, vale la pena aggiungere la patch; un numero sorprendente di sviluppatori sbaglia nel fornire tale informazione.

Qualsiasi codice che aggiunge una nuova interfaccia in spazio utente - inclusi nuovi file in sysfs o /proc - dovrebbe includere la documentazione di tale interfaccia così da permette agli sviluppatori dello spazio utente di sapere con cosa stanno lavorando. Consultate: Documentation/ABI/README per avere una descrizione di come questi documenti devono essere impostati e quali informazioni devono essere fornite.

Il file Documentation/translations/it_IT/admin-guide/kernel-parameters.rst descrive tutti i parametri di avvio del kernel. Ogni patch che aggiunga nuovi parametri dovrebbe aggiungere nuove voci a questo file.

Ogni nuova configurazione deve essere accompagnata da un testo di supporto che spieghi chiaramente le opzioni e spieghi quando l'utente potrebbe volerle selezionare.

Per molti sottosistemi le informazioni sull'API interna sono documentate sotto forma di commenti formattati in maniera particolare; questi commenti possono essere estratti e formattati in differenti modi attraverso lo script "kernel-doc". Se state lavorando all'interno di un sottosistema che ha commenti kerneldoc dovreste mantenerli e aggiungerli, in maniera appropriata, per le funzioni disponibili esternamente. Anche in aree che non sono molto documentate, non c'è motivo per non aggiungere commenti kerneldoc per il futuro; infatti, questa può essere un'attività utile per sviluppatori novizi del kernel. Il formato di questi commenti, assieme alle informazione su come creare modelli per kerneldoc, possono essere trovati in Documentation/translations/it_IT/doc-guide/.

Chiunque legga un ammontare significativo di codice kernel noterà che, spesso, i commenti si fanno maggiormente notare per la loro assenza. Ancora una volta, le aspettative verso il nuovo codice sono più alte rispetto al passato; inserire codice privo di commenti sarà più difficile. Detto ciò, va aggiunto che non si desiderano commenti prolissi per il codice. Il codice dovrebbe essere, di per sé, leggibile, con dei commenti che spieghino gli aspetti più sottili.

Determinate cose dovrebbero essere sempre commentate. L'uso di barriere di memoria dovrebbero essere accompagnate da una riga che spieghi perché sia necessaria. Le regole di sincronizzazione per le strutture dati, generalmente, necessitano di una spiegazioni da qualche parte. Le strutture dati più importanti, in generale, hanno bisogno di una documentazione onnicomprensiva. Le dipendenze che non sono ovvie tra bit separati di codice dovrebbero essere indicate. Tutto ciò che potrebbe indurre un inserviente del codice a fare una "pulizia" incorretta, ha bisogno di un commento che dica perché è stato fatto in quel modo. E così via.

Cambiamenti interni dell'API

L'interfaccia binaria fornita dal kernel allo spazio utente non può essere rotta tranne che in circostanze eccezionali. L'interfaccia di programmazione interna al kernel, invece, è estremamente fluida e può essere modificata al bisogno. Se vi trovate a dover lavorare attorno ad un'API del kernel o semplicemente non state utilizzando una funzionalità offerta perché questa non rispecchia i vostri bisogni, allora questo potrebbe essere un segno che l'API ha bisogno di essere cambiata. In qualità di sviluppatore del kernel, hai il potere di fare questo tipo di modifica.

Ci sono ovviamente alcuni punti da cogliere. I cambiamenti API possono essere

fatti, ma devono essere giustificati. Quindi ogni patch che porta ad una modifica dell'API interna dovrebbe essere accompagnata da una descrizione della modifica in sé e del perché essa è necessaria. Questo tipo di cambiamenti dovrebbero, inoltre, essere fatti in una patch separata, invece di essere sepolti all'interno di una patch più grande.

L'altro punto da cogliere consiste nel fatto che uno sviluppatore che modifica l'API deve, in generale, essere responsabile della correzione di tutto il codice del kernel che viene rotto per via della sua modifica. Per una funzione ampiamente usata, questo compito può condurre letteralmente a centinaia o migliaia di modifiche, molte delle quali sono in conflitto con il lavoro svolto da altri sviluppatori. Non c'è bisogno di dire che questo può essere un lavoro molto grosso, quindi è meglio essere sicuri che la motivazione sia ben solida. Notate che lo strumento Coccinelle può fornire un aiuto con modifiche estese dell'API.

Quando viene fatta una modifica API incompatibile, una persona dovrebbe, quando possibile, assicurarsi che quel codice non aggiornato sia trovato dal compilatore. Questo vi aiuterà ad essere sicuri d'avere trovato, tutti gli usi di quell'interfaccia. Inoltre questo avviserà gli sviluppatori di codice fuori dal kernel che c'è un cambiamento per il quale è necessario del lavoro. Il supporto al codice fuori dal kernel non è qualcosa di cui gli sviluppatori del kernel devono preoccuparsi, ma non dobbiamo nemmeno rendere più difficile del necessario la vita agli sviluppatori di questo codice.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/5.Posting.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Pubblicare modifiche

Prima o poi arriva il momento in cui il vostro lavoro è pronto per essere presentato alla comunità per una revisione ed eventualmente per la sua inclusione nel ramo principale del kernel. Com'era prevedibile, la comunità di sviluppo del kernel ha elaborato un insieme di convenzioni e di procedure per la pubblicazione delle patch; seguirle renderà la vita più facile a tutti quanti. Questo documento cercherà di coprire questi argomenti con un ragionevole livello di dettaglio; più informazioni possono essere trovare nella cartella 'Documentation', nei file *translations/it_IT/process/submitting-patches.rst*, *translations/it_IT/process/submitting-drivers.rst*, e *translations/it_IT/process/submit-checklist.rst*.

Quando pubblicarle

C'è sempre una certa resistenza nel pubblicare patch finché non sono veramente "pronte". Per semplici patch questo non è un problema. Ma quando il lavoro è di una certa complessità, c'è molto da guadagnare dai riscontri che la comunità può darvi prima che completiate il lavoro. Dovreste considerare l'idea di pubblicare un lavoro incompleto, o anche preparare un ramo git disponibile agli sviluppatori interessati, cosicché possano stare al passo col vostro lavoro in qualunque momento.

Quando pubblicate del codice che non è considerato pronto per l'inclusione, è bene che lo diciate al momento della pubblicazione. Inoltre, aggiungete informazioni sulle cose ancora da sviluppare e sui problemi conosciuti. Poche persone guarderanno delle patch che si sa essere fatte a metà, ma quelli che lo faranno penseranno di potervi aiutare a condurre il vostro sviluppo nella giusta direzione.

Prima di creare patch

Ci sono un certo numero di cose che dovreste fare prima di considerare l'invio delle patch alla comunità di sviluppo. Queste cose includono:

- Verificare il codice fino al massimo che vi è consentito. Usate gli strumenti di debug del kernel, assicuratevi che il kernel compili con tutte le più ragionevoli combinazioni d'opzioni, usate cross-compilatori per compilare il codice per differenti architetture, eccetera.
- Assicuratevi che il vostro codice sia conforme alla linee guida del kernel sullo stile del codice.
- La vostra patch ha delle conseguenze in termini di prestazioni? Se è così, dovreste eseguire dei *benchmark* che mostrino il loro impatto (anche positivo); un riassunto dei risultati dovrebbe essere incluso nella patch.
- Siate certi d'avere i diritti per pubblicare il codice. Se questo lavoro è stato fatto per un datore di lavoro, egli avrà dei diritti su questo lavoro e dovrà quindi essere d'accordo alla sua pubblicazione con una licenza GPL

Come regola generale, pensarci un po' di più prima di inviare il codice ripaga quasi sempre lo sforzo.

Preparazione di una patch

La preparazione delle patch per la pubblicazione può richiedere una quantità di lavoro significativa, ma, ripetiamolo ancora, generalmente sconsigliamo di risparmiare tempo in questa fase, anche sul breve periodo.

Le patch devono essere preparate per una specifica versione del kernel. Come regola generale, una patch dovrebbe basarsi sul ramo principale attuale così come lo si trova nei sorgenti git di Linus. Quando vi basate sul ramo principale, cominciate da un punto di rilascio ben noto - uno stabile o un -rc - piuttosto che creare il vostro ramo da quello principale in un punto a caso.

Per facilitare una revisione e una verifica più estesa, potrebbe diventare necessaria la produzione di versioni per -mm, linux-next o i sorgenti di un sottosistema. Basare questa patch sui suddetti sorgenti potrebbe richiedere un lavoro significativo nella risoluzione dei conflitti e nella correzione dei cambiamenti di API; questo potrebbe variare a seconda dell'area d'interesse della vostra patch e da quello che succede altrove nel kernel.

Solo le modifiche più semplici dovrebbero essere preparate come una singola patch; tutto il resto dovrebbe essere preparato come una serie logica di modifiche. Spezzettare le patch è un po'un'arte; alcuni sviluppatori passano molto tempo nel capire come farlo in modo che piaccia alla comunità. Ci sono alcune regole spannometriche, che comunque possono aiutare considerevolmente:

- La serie di patch che pubblicherete, quasi sicuramente, non sarà come quella che trovate nel vostro sistema di controllo di versione. Invece, le vostre modifiche dovranno essere considerate nella loro forma finale, e quindi separate in parti che abbiano un senso. Gli sviluppatori sono interessati in modifiche che siano discrete e indipendenti, non alla strada che avete percorso per ottenerle.

- Ogni modifica logicamente indipendente dovrebbe essere preparata come una patch separata. Queste modifiche possono essere piccole ("aggiunto un campo in questa struttura") o grandi (l'aggiunta di un driver nuovo, per esempio), ma dovrebbero essere concettualmente piccole da permettere una descrizione in una sola riga. Ogni patch dovrebbe fare modifiche specifiche che si possano revisionare indipendentemente e di cui si possa verificare la veridicità.

- Giusto per riaffermare quando detto sopra: non mischiate diversi tipi di modifiche nella stessa patch. Se una modifica corregge un baco critico per la sicurezza, riorganizza alcune strutture, e riformatta il codice, ci sono buone probabilità che venga ignorata e che la correzione importante venga persa.

- Ogni modifica dovrebbe portare ad un kernel che compila e funziona correttamente; se la vostra serie di patch si interrompe a metà il risultato dovrebbe essere comunque un kernel funzionante. L'applicazione parziale di una serie di patch è uno scenario comune nel quale il comando "git bisect" viene usato per trovare delle regressioni; se il risultato è un kernel guasto, renderete la vita degli sviluppatori più difficile così come quella di chi s'impegna nel nobile lavoro di scovare i problemi.

- Però, non strafate. Una volta uno sviluppatore pubblicò una serie di 500 patch che modificavano un unico file - un atto che non lo rese la persona più popolare sulla lista di discussione del kernel. Una singola patch può essere ragionevolmente grande fintanto che contenga un singolo cambiamento *logico*.

- Potrebbe essere allettante l'idea di aggiungere una nuova infrastruttura come una serie di patch, ma di lasciare questa infrastruttura inutilizzata finché l'ultima patch della serie non abilita tutto quanto. Quando è possibile, questo dovrebbe essere evitato; se questa serie aggiunge delle regressioni, "bisect" indicherà quest'ultima patch come causa del problema anche se il baco si trova altrove. Possibilmente, quando una patch aggiunge del nuovo codice dovrebbe renderlo attivo immediatamente.

Lavorare per creare la serie di patch perfetta potrebbe essere frustrante perché richiede un certo tempo e soprattutto dopo che il "vero lavoro" è già stato fatto. Quando ben fatto, comunque, è tempo ben speso.

Formattazione delle patch e i changelog

Quindi adesso avete una serie perfetta di patch pronte per la pubblicazione, ma il lavoro non è davvero finito. Ogni patch deve essere preparata con un messaggio che spieghi al resto del mondo, in modo chiaro e veloce, il suo scopo. Per ottenerlo, ogni patch sarà composta dai seguenti elementi:

- Un campo opzionale "From" col nome dell'autore della patch. Questa riga è necessaria solo se state passando la patch di qualcun altro via email, ma nel dubbio non fa di certo male aggiungerlo.
- Una descrizione di una riga che spieghi cosa fa la patch. Questo messaggio dovrebbe essere sufficiente per far comprendere al lettore lo scopo della patch senza altre informazioni. Questo messaggio, solitamente, presenta in testa il nome del sottosistema a cui si riferisce, seguito dallo scopo della patch. Per esempio:

  ```
  gpio: fix build on CONFIG_GPIO_SYSFS=n
  ```

- Una riga bianca seguita da una descrizione dettagliata della patch. Questa descrizione può essere lunga tanto quanto serve; dovrebbe spiegare cosa fa e perché dovrebbe essere aggiunta al kernel.
- Una o più righe etichette, con, minimo, una riga *Signed-off-by:* col nome dall'autore della patch. Queste etichette verranno descritte meglio più avanti.

Gli elementi qui sopra, assieme, formano il changelog di una patch. Scrivere un buon changelog è cruciale ma è spesso un'arte trascurata; vale la pena spendere qualche parola in più al riguardo. Quando scrivete un changelog dovreste tenere ben presente che molte persone leggeranno le vostre parole. Queste includono i manutentori di un sotto-sistema, e i revisori che devono decidere se la patch debba essere inclusa o no, le distribuzioni e altri manutentori che cercano di valutare se la patch debba essere applicata su kernel più vecchi, i cacciatori di bachi che si chiederanno se la patch è la causa di un problema che stanno cercando, gli utenti che vogliono sapere com'è cambiato il kernel, e molti altri. Un buon changelog fornisce le informazioni necessarie a tutte queste persone nel modo più diretto e conciso possibile.

A questo scopo, la riga riassuntiva dovrebbe descrivere gli effetti della modifica e la motivazione della patch nel modo migliore possibile nonostante il limite di una sola riga. La descrizione dettagliata può spiegare meglio i temi e fornire maggiori informazioni. Se una patch corregge un baco, citate, se possibile, il commit che lo introdusse (e per favore, quando citate un commit aggiungete sia il suo identificativo che il titolo), Se il problema è associabile ad un file di log o all'output del compilatore, includeteli al fine d'aiutare gli altri a trovare soluzioni per lo stesso problema. Se la modifica ha lo scopo di essere di supporto a sviluppi successivi, ditelo. Se le API interne vengono cambiate, dettagliate queste modifiche e come gli altri dovrebbero agire per applicarle. In generale, più riuscirete ad entrare nei

panni di tutti quelli che leggeranno il vostro changelog, meglio sarà il changelog (e il kernel nel suo insieme).

Non serve dirlo, un changelog dovrebbe essere il testo usato nel messaggio di commit in un sistema di controllo di versione. Sarà seguito da:

- La patch stessa, nel formato unificato per patch (“-u”). Usare l’ opzione “-p” assocerà alla modifica il nome della funzione alla quale si riferisce, rendendo il risultato più facile da leggere per gli altri.

Dovreste evitare di includere nelle patch delle modifiche per file irrilevanti (quelli generati dal processo di generazione, per esempio, o i file di backup del vostro editor). Il file “dontdiff” nella cartella Documentation potrà esservi d’ aiuto su questo punto; passatelo a diff con l’ opzione “-X” .

Le etichette sopra menzionante sono usate per descrivere come i vari sviluppatori sono stati associati allo sviluppo di una patch. Sono descritte in dettaglio nel documento *translations/it_IT/process/submitting-patches.rst*; quello che segue è un breve riassunto. Ognuna di queste righe ha il seguente formato:

```
tag: Full Name <email address>  optional-other-stuff
```

Le etichette in uso più comuni sono:

- Signed-off-by: questa è la certificazione che lo sviluppatore ha il diritto di sottomettere la patch per l’ integrazione nel kernel. Questo rappresenta il consenso verso il certificato d’ origine degli sviluppatori, il testo completo potrà essere trovato in Inviare patch: la guida essenziale per vedere il vostro codice nel kernel. Codice che non presenta una firma appropriata non potrà essere integrato.
- Co-developed-by: indica che la patch è stata cosviluppata da diversi sviluppatori; viene usato per assegnare più autori (in aggiunta a quello associato all’ etichetta From:) quando più persone lavorano ad una patch. Ogni Co-developed-by: dev’ essere seguito immediatamente da un Signed-off-by: del corrispondente coautore. Maggiori dettagli ed esempi sono disponibili in Inviare patch: la guida essenziale per vedere il vostro codice nel kernel.
- Acked-by: indica il consenso di un altro sviluppatore (spesso il manutentore del codice in oggetto) all’ integrazione della patch nel kernel.
- Tested-by: menziona la persona che ha verificato la patch e l’ ha trovata funzionante.
- Reviwed-by: menziona lo sviluppatore che ha revisionato la patch; per maggiori dettagli leggete la dichiarazione dei revisori in Inviare patch: la guida essenziale per vedere il vostro codice nel kernel
- Reported-by: menziona l’ utente che ha riportato il problema corretto da questa patch; quest’ etichetta viene usata per dare credito alle persone che hanno verificato il codice e ci hanno fatto sapere quando le cose non funzionavano correttamente.
- Cc: la persona menzionata ha ricevuto una copia della patch ed ha avuto l’ opportunità di commentarla.

State attenti ad aggiungere queste etichette alla vostra patch: solo "Cc:" può essere aggiunta senza il permesso esplicito della persona menzionata.

Inviare la modifica

Prima di inviare la vostra patch, ci sarebbero ancora un paio di cose di cui dovreste aver cura:

- Siete sicuri che il vostro programma di posta non corromperà le patch? Le patch che hanno spazi bianchi in libertà o andate a capo aggiunti dai programmi di posta non funzioneranno per chi le riceve, e spesso non verranno nemmeno esaminate in dettaglio. Se avete un qualsiasi dubbio, inviate la patch a voi stessi e verificate che sia integra.

  Informazioni sui programmi di posta elettronica per Linux contiene alcuni suggerimenti utili sulla configurazione dei programmi di posta al fine di inviare patch.

- Siete sicuri che la vostra patch non contenga sciocchi errori? Dovreste sempre processare le patch con scripts/checkpatch.pl e correggere eventuali problemi riportati. Per favore tenete ben presente che checkpatch.pl non è più intelligente di voi, nonostante sia il risultato di un certa quantità di ragionamenti su come debba essere una patch per il kernel. Se seguire i suggerimenti di checkpatch.pl rende il codice peggiore, allora non fatelo.

Le patch dovrebbero essere sempre inviate come testo puro. Per favore non inviatele come allegati; questo rende molto più difficile, per i revisori, citare parti della patch che si vogliono commentare. Invece, mettete la vostra patch direttamente nel messaggio.

Quando inviate le patch, è importante inviarne una copia a tutte le persone che potrebbero esserne interessate. Al contrario di altri progetti, il kernel incoraggia le persone a peccare nell' invio di tante copie; non presumente che le persone interessate vedano i vostri messaggi sulla lista di discussione. In particolare le copie dovrebbero essere inviate a:

- I manutentori dei sottosistemi affetti della modifica. Come descritto in precedenza, il file MAINTAINERS è il primo luogo dove cercare i nomi di queste persone.
- Altri sviluppatori che hanno lavorato nello stesso ambiente - specialmente quelli che potrebbero lavorarci proprio ora. Usate git potrebbe essere utile per vedere chi altri ha modificato i file su cui state lavorando.
- Se state rispondendo a un rapporto su un baco, o a una richiesta di funzionalità, includete anche gli autori di quei rapporti/richieste.
- Inviate una copia alle liste di discussione interessate, o, se nient' altro è adatto, alla lista linux-kernel
- Se state correggendo un baco, pensate se la patch dovrebbe essere inclusa nel prossimo rilascio stabile. Se è così, la lista di discussione stable@vger.kernel.org dovrebbe riceverne una copia. Aggiungete anche l' etichetta "Cc: stable@vger.kernel.org" nella patch stessa; questo permetterà alla squadra *stable* di ricevere una notifica quando questa correzione

viene integrata nel ramo principale.

Quando scegliete i destinatari della patch, è bene avere un'idea di chi pensiate che sia colui che, eventualmente, accetterà la vostra patch e la integrerà. Nonostante sia possibile inviare patch direttamente a Linus Torvalds, e lasciare che sia lui ad integrarle,solitamente non è la strada migliore da seguire. Linus è occupato, e ci sono dei manutentori di sotto-sistema che controllano una parte specifica del kernel. Solitamente, vorreste che siano questi manutentori ad integrare le vostre patch. Se non c'è un chiaro manutentore, l'ultima spiaggia è spesso Andrew Morton.

Le patch devono avere anche un buon oggetto. Il tipico formato per l'oggetto di una patch assomiglia a questo:

```
[PATCH nn/mm] subsys: one-line description of the patch
```

dove "nn" è il numero ordinale della patch, "mm" è il numero totale delle patch nella serie, e "subsys" è il nome del sottosistema interessato. Chiaramente, nn/mm può essere omesso per una serie composta da una singola patch.

Se avete una significative serie di patch, è prassi inviare una descrizione introduttiva come parte zero. Tuttavia questa convenzione non è universalmente seguita; se la usate, ricordate che le informazioni nell'introduzione non faranno parte del changelog del kernel. Quindi per favore, assicuratevi che ogni patch abbia un changelog completo.

In generale, la seconda parte e quelle successive di una patch "composta" dovrebbero essere inviate come risposta alla prima, cosicché vengano viste come un unico *thread*. Strumenti come git e quilt hanno comandi per inviare gruppi di patch con la struttura appropriata. Se avete una serie lunga e state usando git, per favore state alla larga dall'opzione –chain-reply-to per evitare di creare un annidamento eccessivo.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/6.Followthrough.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Completamento

A questo punto, avete seguito le linee guida fino a questo punto e, con l'aggiunta delle vostre capacità ingegneristiche, avete pubblicato una serie perfetta di patch. Uno dei più grandi errori che possono essere commessi persino da sviluppatori kernel esperti è quello di concludere che il lavoro sia ormai finito. In verità, la pubblicazione delle patch simboleggia una transizione alla fase successiva del processo, con, probabilmente, ancora un po' di lavoro da fare.

È raro che una modifica sia così bella alla sua prima pubblicazione che non ci sia alcuno spazio di miglioramento. Il programma di sviluppo del kernel riconosce questo fatto e quindi, è fortemente orientato al miglioramento del codice pubblicato. Voi, in qualità di autori del codice, dovrete lavorare con la comunità del kernel per assicurare che il vostro codice mantenga gli standard qualitativi richiesti. Un fallimento in questo processo è quasi come impedire l'inclusione delle vostre patch nel ramo principale.

Lavorare con i revisori

Una patch che abbia una certa rilevanza avrà ricevuto numerosi commenti da parte di altri sviluppatori dato che avranno revisionato il codice. Lavorare con i revisori può rivelarsi, per molti sviluppatori, la parte più intimidatoria del processo di sviluppo del kernel. La vita può esservi resa molto più facile se tenete presente alcuni dettagli:

- Se avete descritto la vostra modifica correttamente, i revisori ne comprenderanno il valore e il perché vi siete presi il disturbo di scriverla. Ma tale valore non li tratterrà dal porvi una domanda fondamentale: come verrà mantenuto questo codice nel kernel nei prossimi cinque o dieci anni? Molti dei cambiamenti che potrebbero esservi richiesti - da piccoli problemi di stile a sostanziali ristesure - vengono dalla consapevolezza che Linux resterà in circolazione e in continuo sviluppo ancora per diverse decadi.
- La revisione del codice è un duro lavoro, ed è un mestiere poco riconosciuto; le persone ricordano chi ha scritto il codice, ma meno fama è attribuita a chi lo ha revisionato. Quindi i revisori potrebbero divenire burberi, specialmente quando vendono i medesimi errori venire fatti ancora e ancora. Se ricevete una revisione che vi sembra abbia un tono arrabbiato, insultante o addirittura offensivo, resistente alla tentazione di rispondere a tono. La revisione riguarda il codice e non la persona, e i revisori non vi stanno attaccando personalmente.
- Similarmente, i revisori del codice non stanno cercando di promuovere i loro interessi a vostre spese. Gli sviluppatori del kernel spesso si aspettano di lavorare sul kernel per anni, ma sanno che il loro datore di lavoro può cambiare. Davvero, senza praticamente eccezioni, loro stanno lavorando per la creazione del miglior kernel possibile; non stanno cercando di creare un disagio ad aziende concorrenti.

Quello che si sta cercando di dire è che, quando i revisori vi inviano degli appunti dovete fare attenzione alle osservazioni tecniche che vi stanno facendo. Non lasciate che il loro modo di esprimersi o il vostro orgoglio impediscano che ciò accada.

Quando avete dei suggerimenti sulla revisione, prendetevi il tempo per comprendere cosa il revisore stia cercando di comunicarvi. Se possibile, sistemate le cose che il revisore vi chiede di modificare. E rispondete al revisore ringraziandolo e spiegando come intendete fare.

Notate che non dovete per forza essere d'accordo con ogni singola modifica suggerita dai revisori. Se credete che il revisore non abbia compreso il vostro codice, spiegateglielo. Se avete un'obiezione tecnica da fargli su di una modifica suggerita, spiegatela inserendo anche la vostra soluzione al problema. Se la vostra spiegazione ha senso, il revisore la accetterà. Tuttavia, la vostra motivazione potrebbe non essere del tutto persuasiva, specialmente se altri iniziano ad essere d'accordo con il revisore. Prendetevi quindi un po' di tempo per pensare ancora alla cosa. Può risultare facile essere accecati dalla propria soluzione al punto che non realizzate che c'è qualcosa di fondamentalmente sbagliato o, magari, non state nemmeno risolvendo il problema giusto.

Andrew Morton suggerisce che ogni suggerimento di revisione che non è presente nella modifica del codice dovrebbe essere inserito in un commento aggiuntivo; ciò può essere d'aiuto ai futuri revisori nell'evitare domande che sorgono al primo sguardo.

Un errore fatale è quello di ignorare i commenti di revisione nella speranza che se ne andranno. Non andranno via. Se pubblicherete nuovamente il codice senza aver risposto ai commenti ricevuti, probabilmente le vostre modifiche non andranno da nessuna parte.

Parlando di ripubblicazione del codice: per favore tenete a mente che i revisori non ricorderanno tutti i dettagli del codice che avete pubblicato l'ultima volta. Quindi è sempre una buona idea quella di ricordare ai revisori le questioni sollevate precedetemene e come le avete risolte. I revisori non dovrebbero star lì a cercare all'interno degli archivi per famigliarizzare con ciò che è stato detto l'ultima volta; se li aiutate in questo senso, saranno di umore migliore quando riguarderanno il vostro codice.

Se invece avete cercato di far tutto correttamente ma le cose continuano a non andar bene? Molti disaccordi di natura tecnica possono essere risolti attraverso la discussione, ma ci sono volte dove qualcuno deve prendere una decisione. Se credete veramente che tale decisione andrà contro di voi ingiustamente, potete sempre tentare di rivolgervi a qualcuno più in alto di voi. Per cose di questo genere la persona con più potere è Andrew Morton. Andrew è una figura molto rispettata all'interno della comunità di sviluppo del kernel; lui può spesso sbrogliare situazioni che sembrano irrimediabilmente bloccate. Rivolgersi ad Andrew non deve essere fatto alla leggera, e non deve essere fatto prima di aver esplorato tutte le altre alternative. E tenete a mente, ovviamente, che nemmeno lui potrebbe non essere d'accordo con voi.

Cosa accade poi

Se la modifica è ritenuta un elemento valido da essere aggiunta al kernel, e una volta che la maggior parte degli appunti dei revisori sono stati sistemati, il passo successivo solitamente è quello di entrare in un sottosistema gestito da un manutentore. Come ciò avviene dipende dal sottosistema medesimo; ogni manutentore ha il proprio modo di fare le cose. In particolare, ci potrebbero essere diversi sorgenti - uno, magari, dedicato alle modifiche pianificate per la finestra di fusione successiva, e un altro per il lavoro di lungo periodo.

Per le modifiche proposte in aree per le quali non esiste un sottosistema preciso (modifiche di gestione della memoria, per esempio), i sorgenti di ripiego finiscono per essere -mm. Ed anche le modifiche che riguardano più sottosistemi possono finire in quest' ultimo.

L'inclusione nei sorgenti di un sottosistema può comportare per una patch, un alto livello di visibilità. Ora altri sviluppatori che stanno lavorando in quei medesimi sorgenti avranno le vostre modifiche. I sottosistemi solitamente riforniscono anche Linux-next, rendendo i propri contenuti visibili all'intera comunità di sviluppo. A questo punto, ci sono buone possibilità per voi di ricevere ulteriori commenti da un nuovo gruppo di revisori; anche a questi commenti dovrete rispondere come avete già fatto per gli altri.

Ciò che potrebbe accadere a questo punto, in base alla natura della vostra modifica, riguarda eventuali conflitti con il lavoro svolto da altri. Nella peggiore delle situazioni, i conflitti più pesanti tra modifiche possono concludersi con la messa a lato di alcuni dei lavori svolti cosicché le modifiche restanti possano funzionare ed essere integrate. Altre volte, la risoluzione dei conflitti richiederà del lavoro con altri sviluppatori e, possibilmente, lo spostamento di alcune patch da dei sorgenti a degli altri in modo da assicurare che tutto sia applicato in modo pulito. Questo lavoro può rivelarsi una spina nel fianco, ma consideratevi fortunati: prima dell'avvento dei sorgenti linux-next, questi conflitti spesso emergevano solo durante l'apertura della finestra di integrazione e dovevano essere smaltiti in fretta. Ora essi possono essere risolti comodamente, prima dell'apertura della finestra.

Un giorno, se tutto va bene, vi collegherete e vedrete che la vostra patch è stata inserita nel ramo principale de kernel. Congratulazioni! Terminati i festeggiamenti (nel frattempo avrete inserito il vostro nome nel file MAINTAINERS) vale la pena ricordare una piccola cosa, ma importante: il lavoro non è ancora finito. L'inserimento nel ramo principale porta con se nuove sfide.

Cominciamo con il dire che ora la visibilità della vostra modifica è ulteriormente cresciuta. Ci potrebbe portare ad una nuova fase di commenti dagli sviluppatori che non erano ancora a conoscenza della vostra patch. Ignorarli potrebbe essere allettante dato che non ci sono più dubbi sull'integrazione della modifica. Resistete a tale tentazione, dovete mantenervi disponibili agli sviluppatori che hanno domande o suggerimenti per voi.

Ancora più importante: l'inclusione nel ramo principale mette il vostro codice nelle mani di un gruppo di *tester* molto più esteso. Anche se avete contribuito ad un driver per un hardware che non è ancora disponibile, sarete sorpresi da quante persone inseriranno il vostro codice nei loro kernel. E, ovviamente, dove ci sono *tester*, ci saranno anche dei rapporti su eventuali bachi.

La peggior specie di rapporti sono quelli che indicano delle regressioni. Se la vostra modifica causa una regressione, avrete un gran numero di occhi puntati su di voi; la regressione deve essere sistemata il prima possibile. Se non vorrete o non sarete capaci di sistemarla (e nessuno lo farà per voi), la vostra modifica sarà quasi certamente rimossa durante la fase di stabilizzazione. Oltre alla perdita di tutto il lavoro svolto per far si che la vostra modifica fosse inserita nel ramo principale, l'avere una modifica rimossa a causa del fallimento nel sistemare una regressione, potrebbe rendere più difficile per voi far accettare il vostro lavoro in futuro.

Dopo che ogni regressione è stata affrontata, ci potrebbero essere altri bachi ordinari da "sconfiggere". Il periodo di stabilizzazione è la vostra migliore opportunità per sistemare questi bachi e assicurarvi che il debutto del vostro codice nel ramo principale del kernel sia il più solido possibile. Quindi, per favore, rispondete ai rapporti sui bachi e ponete rimedio, se possibile, a tutti i problemi. È a questo che serve il periodo di stabilizzazione; potete iniziare creando nuove fantastiche modifiche una volta che ogni problema con le vecchie sia stato risolto.

Non dimenticate che esistono altre pietre miliari che possono generare rapporti sui bachi: il successivo rilascio stabile, quando una distribuzione importante usa una versione del kernel nel quale è presente la vostra modifica, eccetera. Il continuare a rispondere a questi rapporti è fonte di orgoglio per il vostro lavoro. Se questa non è una sufficiente motivazione, allora, è anche consigliabile considera che la comunità di sviluppo ricorda gli sviluppatori che hanno perso interesse per il loro codice una volta integrato. La prossima volta che pubblicherete una patch, la comunità la valuterà anche sulla base del fatto che non sarete disponibili a prendervene cura anche nel futuro.

Altre cose che posso accadere

Un giorno, potreste aprire la vostra email e vedere che qualcuno vi ha inviato una patch per il vostro codice. Questo, dopo tutto, è uno dei vantaggi di avere il vostro codice "là fuori". Se siete d'accordo con la modifica, potrete anche inoltrarla ad un manutentore di sottosistema (assicuratevi di includere la riga "From:" cosicché l'attribuzione sia corretta, e aggiungete una vostra firma "Signed-off-by"), oppure inviate un "Acked-by:" e lasciate che l'autore originale la invii.

Se non siete d'accordo con la patch, inviate una risposta educata spiegando il perché. Se possibile, dite all'autore quali cambiamenti servirebbero per rendere la patch accettabile da voi. C'è una certa riluttanza nell'inserire modifiche con un conflitto fra autore e manutentore del codice, ma solo fino ad un certo punto. Se siete visti come qualcuno che blocca un buon lavoro senza motivo, quelle patch vi passeranno oltre e andranno nel ramo principale in ogni caso. Nel kernel Linux, nessuno ha potere di veto assoluto su alcun codice. Eccezione fatta per Linus, forse.

In rarissime occasioni, potreste vedere qualcosa di completamente diverso: un altro sviluppatore che pubblica una soluzione differente al vostro problema. A questo punto, c'è una buona probabilità che una delle due modifiche non verrà integrata, e il "c'ero prima io"non è considerato un argomento tecnico rilevante. Se la modifica di qualcun'altro rimpiazza la vostra ed entra nel ramo principale, esiste un unico modo di reagire: siate contenti che il vostro problema sia stato risolto e andate avanti con il vostro lavoro. L'avere un vostro lavoro spintonato da parte in

questo modo può essere avvilente e scoraggiante, ma la comunità ricorderà come avrete reagito anche dopo che avrà dimenticato quale fu la modifica accettata.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/7.AdvancedTopics.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Argomenti avanzati

A questo punto, si spera, dovreste avere un'idea su come funziona il processo di sviluppo. Ma rimane comunque molto da imparare! Questo capitolo copre alcuni argomenti che potrebbero essere utili per gli sviluppatori che stanno per diventare parte integrante del processo di sviluppo del kernel.

Gestire le modifiche con git

L'uso di un sistema distribuito per il controllo delle versioni del kernel ebbe inizio nel 2002 quando Linux iniziò a provare il programma proprietario BitKeeper. Nonostante l'uso di BitKeeper fosse opinabile, di certo il suo approccio alla gestione dei sorgenti non lo era. Un sistema distribuito per il controllo delle versioni accelerò immediatamente lo sviluppo del kernel. Oggigiorno, ci sono diverse alternative libere a BitKeeper. Per il meglio o il peggio, il progetto del kernel ha deciso di usare git per gestire i sorgenti.

Gestire le modifiche con git può rendere la vita dello sviluppatore molto più facile, specialmente quando il volume delle modifiche cresce. Git ha anche i suoi lati taglienti che possono essere pericolosi; è uno strumento giovane e potente che è ancora in fase di civilizzazione da parte dei suoi sviluppatori. Questo documento non ha lo scopo di insegnare l'uso di git ai suoi lettori; ci sarebbe materiale a sufficienza per un lungo documento al riguardo. Invece, qui ci concentriamo in particolare su come git è parte del processo di sviluppo del kernel. Gli sviluppatori che desiderassero diventare agili con git troveranno più informazioni ai seguenti indirizzi:

http://git-scm.com/

http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

e su varie guide che potrete trovare su internet.

La prima cosa da fare prima di usarlo per produrre patch che saranno disponibili ad altri, è quella di leggere i siti qui sopra e di acquisire una base solida su come funziona git. Uno sviluppatore che sappia usare git dovrebbe essere capace di ottenere una copia del repositorio principale, esplorare la storia della revisione,

registrare le modifiche, usare i rami, eccetera. Una certa comprensione degli strumenti git per riscrivere la storia (come `rebase`) è altrettanto utile. Git ha i propri concetti e la propria terminologia; un nuovo utente dovrebbe conoscere *refs*, *remote branch*, *index*, *fast-forward merge*, *push* e *pull*, *detached head*, eccetera. Il tutto potrebbe essere un po' intimidatorio visto da fuori, ma con un po' di studio i concetti non saranno così difficili da capire.

Utilizzare git per produrre patch da sottomettere via email può essere un buon esercizio da fare mentre si sta prendendo confidenza con lo strumento.

Quando sarete in grado di creare rami git che siano guardabili da altri, vi servirà, ovviamente, un server dal quale sia possibile attingere le vostre modifiche. Se avete un server accessibile da Internet, configurarlo per eseguire git-daemon è relativamente semplice . Altrimenti, iniziano a svilupparsi piattaforme che offrono spazi pubblici, e gratuiti (Github, per esempio). Gli sviluppatori permanenti possono ottenere un account su kernel.org, ma non è proprio facile da ottenere; per maggiori informazioni consultate la pagina web http://kernel.org/faq/.

In git è normale avere a che fare con tanti rami. Ogni linea di sviluppo può essere separata in "rami per argomenti"e gestiti indipendentemente. In git i rami sono facilissimi, per cui non c'è motivo per non usarli in libertà. In ogni caso, non dovreste sviluppare su alcun ramo dal quale altri potrebbero attingere. I rami disponibili pubblicamente dovrebbero essere creati con attenzione; integrate patch dai rami di sviluppo solo quando sono complete e pronte ad essere consegnate - non prima.

Git offre alcuni strumenti che vi permettono di riscrivere la storia del vostro sviluppo. Una modifica errata (diciamo, una che rompe la bisezione, oppure che ha un qualche tipo di baco evidente) può essere corretta sul posto o fatta sparire completamente dalla storia. Una serie di patch può essere riscritta come se fosse stata scritta in cima al ramo principale di oggi, anche se ci avete lavorato per mesi. Le modifiche possono essere spostate in modo trasparente da un ramo ad un altro. E così via. Un uso giudizioso di git per revisionare la storia può aiutare nella creazione di una serie di patch pulite e con meno problemi.

Un uso eccessivo può portare ad altri tipi di problemi, tuttavia, oltre alla semplice ossessione per la creazione di una storia del progetto che sia perfetta. Riscrivere la storia riscriverà le patch contenute in quella storia, trasformando un kernel verificato (si spera) in uno da verificare. Ma, oltre a questo, gli sviluppatori non possono collaborare se non condividono la stessa vista sulla storia del progetto; se riscrivete la storia dalla quale altri sviluppatori hanno attinto per i loro repositori, renderete la loro vita molto più difficile. Quindi tenete conto di questa semplice regola generale: la storia che avete esposto ad altri, generalmente, dovrebbe essere vista come immutabile.

Dunque, una volta che il vostro insieme di patch è stato reso disponibile pubblicamente non dovrebbe essere più sovrascritto. Git tenterà di imporre questa regola, e si rifiuterà di pubblicare nuove patch che non risultino essere dirette discendenti di quelle pubblicate in precedenza (in altre parole, patch che non condividono la stessa storia). È possibile ignorare questo controllo, e ci saranno momenti in cui sarà davvero necessario riscrivere un ramo già pubblicato. Un esempio è linux-next dove le patch vengono spostate da un ramo all'altro al fine di evitare conflitti. Ma questo tipo d'azione dovrebbe essere un'eccezione. Questo è uno dei motivi per cui lo sviluppo dovrebbe avvenire in rami privati (che possono essere sovrascritti quando lo si ritiene necessario) e reso pubblico solo quando è in uno

stato avanzato.

Man mano che il ramo principale (o altri rami su cui avete basato le modifiche) avanza, diventa allettante l'idea di integrare tutte le patch per rimanere sempre aggiornati. Per un ramo privato, il *rebase* può essere un modo semplice per rimanere aggiornati, ma questa non è un'opzione nel momento in cui il vostro ramo è stato esposto al mondo intero. *Merge* occasionali possono essere considerati di buon senso, ma quando diventano troppo frequenti confondono inutilmente la storia. La tecnica suggerita in questi casi è quella di fare *merge* raramente, e più in generale solo nei momenti di rilascio (per esempio gli -rc del ramo principale). Se siete nervosi circa alcune patch in particolare, potete sempre fare dei *merge* di test in un ramo privato. In queste situazioni git "rerere" può essere utile; questo strumento si ricorda come i conflitti di *merge* furono risolti in passato cosicché non dovrete fare lo stesso lavoro due volte.

Una delle lamentele più grosse e ricorrenti sull'uso di strumenti come git è il grande movimento di patch da un repositorio all'altro che rende facile l'integrazione nel ramo principale di modifiche mediocri, il tutto sotto il naso dei revisori. Gli sviluppatori del kernel tendono ad essere scontenti quando vedono succedere queste cose; preparare un ramo git con patch che non hanno ricevuto alcuna revisione o completamente avulse, potrebbe influire sulla vostra capacita di proporre, in futuro, l'integrazione dei vostri rami. Citando Linus

```
Potete inviarmi le vostre patch, ma per far si che io integri una
vostra modifica da git, devo sapere che voi sappiate cosa state
facendo, e ho bisogno di fidarmi *senza* dover passare tutte
le modifiche manualmente una per una.
```

(http://lwn.net/Articles/224135/).

Per evitare queste situazioni, assicuratevi che tutte le patch in un ramo siano strettamente correlate al tema delle modifiche; un ramo "driver fixes" non dovrebbe fare modifiche al codice principale per la gestione della memoria. E, più importante ancora, non usate un repositorio git per tentare di evitare il processo di revisione. Pubblicate un sommario di quello che il vostro ramo contiene sulle liste di discussione più opportune, e , quando sarà il momento, richiedete che il vostro ramo venga integrato in linux-next.

Se e quando altri inizieranno ad inviarvi patch per essere incluse nel vostro repositorio, non dovete dimenticare di revisionarle. Inoltre assicuratevi di mantenerne le informazioni di paternità; al riguardo git "am" fa del suo meglio, ma potreste dover aggiungere una riga "From:" alla patch nel caso in cui sia arrivata per vie traverse.

Quando richiedete l'integrazione, siate certi di fornire tutte le informazioni: dov'è il vostro repositorio, quale ramo integrare, e quali cambiamenti si otterranno dall'integrazione. Il comando git request-pull può essere d'aiuto; preparerà una richiesta nel modo in cui gli altri sviluppatori se l'aspettano, e verificherà che vi siate ricordati di pubblicare quelle patch su un server pubblico.

Revisionare le patch

Alcuni lettori potrebbero avere obiezioni sulla presenza di questa sezione negli "argomenti avanzati" sulla base che anche gli sviluppatori principianti dovrebbero revisionare le patch. É certamente vero che non c'è modo migliore di imparare come programmare per il kernel che guardare il codice pubblicato dagli altri. In aggiunta, i revisori sono sempre troppo pochi; guardando il codice potete apportare un significativo contributo all' intero processo.

Revisionare il codice potrebbe risultare intimidatorio, specialmente per i nuovi arrivati che potrebbero sentirsi un po' nervosi nel questionare il codice - in pubblico - pubblicato da sviluppatori più esperti. Perfino il codice scritto dagli sviluppatori più esperti può essere migliorato. Forse il suggerimento migliore per i revisori (tutti) è questo: formulate i commenti come domande e non come critiche. Chiedere "Come viene rilasciato il *lock* in questo percorso?"funziona sempre molto meglio che "qui la sincronizzazione è sbagliata" .

Diversi sviluppatori revisioneranno il codice con diversi punti di vista. Alcuni potrebbero concentrarsi principalmente sullo stile del codice e se alcune linee hanno degli spazio bianchi di troppo. Altri si chiederanno se accettare una modifica interamente è una cosa positiva per il kernel o no. E altri ancora si focalizzeranno sui problemi di sincronizzazione, l' uso eccessivo di *stack*, problemi di sicurezza, duplicazione del codice in altri contesti, documentazione, effetti negativi sulle prestazioni, cambi all' ABI dello spazio utente, eccetera. Qualunque tipo di revisione è ben accetta e di valore, se porta ad avere un codice migliore nel kernel.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l' unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/8.Conclusion.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Per maggiori informazioni

Esistono numerose fonti di informazioni sullo sviluppo del kernel Linux e argomenti correlati. Primo tra questi sarà sempre la cartella Documentation che si trova nei sorgenti kernel.

Il file *process/howto.rst* è un punto di partenza importante; *process/submitting-patches.rst* e *process/submitting-drivers.rst* sono anch' essi qualcosa che tutti gli sviluppatori del kernel dovrebbero leggere. Molte API interne al kernel sono documentate utilizzando il meccanismo kerneldoc; "make htmldocs" o "make pdfdocs" possono essere usati per generare quei documenti in HTML o PDF (sebbene le versioni di TeX di alcune distribuzioni hanno dei limiti interni e fallisce nel processare appropriatamente i documenti).

Diversi siti web approfondiscono lo sviluppo del kernel ad ogni livello di dettaglio. Il vostro autore vorrebbe umilmente suggerirvi http://lwn.net/ come fonte; usando l'indice 'kernel' su LWN troverete molti argomenti specifici sul kernel:

http://lwn.net/Kernel/Index/

Oltre a ciò, una risorsa valida per gli sviluppatori kernel è:

http://kernelnewbies.org/

E, ovviamente, una fonte da non dimenticare è http://kernel.org/, il luogo definitivo per le informazioni sui rilasci del kernel.

Ci sono numerosi libri sullo sviluppo del kernel:

Linux Device Drivers, 3rd Edition (Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman). In linea all'indirizzo http://lwn.net/Kernel/LDD3/.

Linux Kernel Development (Robert Love).

Understanding the Linux Kernel (Daniel Bovet and Marco Cesati).

Tutti questi libri soffrono di un errore comune: tendono a risultare in un certo senso obsoleti dal momento che si trovano in libreria da diverso tempo. Comunque contengono informazioni abbastanza buone.

La documentazione per git la troverete su:

http://www.kernel.org/pub/software/scm/git/docs/

http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

Conclusioni

Congratulazioni a chiunque ce l'abbia fatta a terminare questo documento di lungo-respiro. Si spera che abbia fornito un'utile comprensione d'insieme di come il kernel Linux viene sviluppato e di come potete partecipare a tale processo.

Infine, quello che conta è partecipare. Qualsiasi progetto software open-source non è altro che la somma di quello che i suoi contributori mettono al suo interno. Il kernel Linux è cresciuto velocemente e bene perché ha ricevuto il supporto di un impressionante gruppo di sviluppatori, ognuno dei quali sta lavorando per renderlo migliore. Il kernel è un esempio importante di cosa può essere fatto quando migliaia di persone lavorano insieme verso un obiettivo comune.

Il kernel può sempre beneficiare di una larga base di sviluppatori, tuttavia, c'è sempre molto lavoro da fare. Ma, cosa non meno importante, molti degli altri partecipanti all'ecosistema Linux possono trarre beneficio attraverso il contributo al kernel. Inserire codice nel ramo principale è la chiave per arrivare ad una qualità del codice più alta, bassa manutenzione e bassi prezzi di distribuzione, alti livelli d'influenza sulla direzione dello sviluppo del kernel, e molto altro. È una situazione nella quale tutti coloro che sono coinvolti vincono. Mollate il vostro editor e raggiungeteci; sarete più che benvenuti.

Lo scopo di questo documento è quello di aiutare gli sviluppatori (ed i loro supervisori) a lavorare con la comunità di sviluppo con il minimo sforzo. È un tentativo

di documentare il funzionamento di questa comunità in modo che sia accessibile anche a coloro che non hanno famigliarità con lo sviluppo del Kernel Linux (o, anzi, con lo sviluppo di software libero in generale). Benchè qui sia presente del materiale tecnico, questa è una discussione rivolta in particolare al procedimento, e quindi per essere compreso non richiede una conoscenza approfondità sullo sviluppo del kernel.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/submitting-patches.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Inviare patch: la guida essenziale per vedere il vostro codice nel kernel

Una persona o un'azienda che volesse inviare una patch al kernel potrebbe sentirsi scoraggiata dal processo di sottomissione, specialmente quando manca una certa familiarità col "sistema". Questo testo è una raccolta di suggerimenti che aumenteranno significativamente le probabilità di vedere le vostre patch accettate.

Questo documento contiene un vasto numero di suggerimenti concisi. Per maggiori dettagli su come funziona il processo di sviluppo del kernel leggete *Documentation/translations/it_IT/process*. Leggete anche Lista delle verifiche da fare prima di inviare una patch per il kernel Linux per una lista di punti da verificare prima di inviare del codice. Se state inviando un driver, allora leggete anche Sottomettere driver per il kernel Linux; per delle patch relative alle associazioni per Device Tree leggete Documentation/devicetree/bindings/submitting-patches.rst.

Molti di questi passi descrivono il comportamento di base del sistema di controllo di versione `git`; se utilizzate `git` per preparare le vostre patch molto del lavoro più ripetitivo lo troverete già fatto per voi, tuttavia dovete preparare e documentare un certo numero di patch. Generalmente, l'uso di `git` renderà la vostra vita di sviluppatore del kernel più facile.

0) Ottenere i sorgenti attuali

Se non avete un repositorio coi sorgenti del kernel più recenti, allora usate `git` per ottenerli. Vorrete iniziare col repositorio principale che può essere recuperato col comando:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/
↪linux.git
```

Notate, comunque, che potreste non voler sviluppare direttamente coi sorgenti principali del kernel. La maggior parte dei manutentori hanno i propri sorgenti

e desiderano che le patch siano preparate basandosi su di essi. Guardate l'elemento **T:** per un determinato sottosistema nel file MAINTANERS che troverete nei sorgenti, o semplicemente chiedete al manutentore nel caso in cui i sorgenti da usare non siano elencati il quel file.

Esiste ancora la possibilità di scaricare un rilascio del kernel come archivio tar (come descritto in una delle prossime sezioni), ma questa è la via più complicata per sviluppare per il kernel.

1) `diff -up`

Se dovete produrre le vostre patch a mano, usate `diff -up` o `diff -uprN` per crearle. Git produce di base le patch in questo formato; se state usando `git`, potete saltare interamente questa sezione.

Tutte le modifiche al kernel Linux avvengono mediate patch, come descritte in *`diff(1)`*. Quando create la vostra patch, assicuratevi di crearla nel formato "unified diff", come l'argomento `-u` di *`diff(1)`*. Inoltre, per favore usate l'argomento `-p` per mostrare la funzione C alla quale si riferiscono le diverse modifiche - questo rende il risultato di `diff` molto più facile da leggere. Le patch dovrebbero essere basate sulla radice dei sorgenti del kernel, e non sulle sue sottocartelle.

Per creare una patch per un singolo file, spesso è sufficiente fare:

```
SRCTREE=linux
MYFILE=drivers/net/mydriver.c

cd $SRCTREE
cp $MYFILE $MYFILE.orig
vi $MYFILE      # make your change
cd ..
diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

Per creare una patch per molteplici file, dovreste spacchettare i sorgenti "vergini", o comunque non modificati, e fare un `diff` coi vostri. Per esempio:

```
MYSRC=/devel/linux

tar xvfz linux-3.19.tar.gz
mv linux-3.19 linux-3.19-vanilla
diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
        linux-3.19-vanilla $MYSRC > /tmp/patch
```

`dontdiff` è una lista di file che sono generati durante il processo di compilazione del kernel; questi dovrebbero essere ignorati in qualsiasi patch generata con *`diff(1)`*.

Assicuratevi che la vostra patch non includa file che non ne fanno veramente parte. Al fine di verificarne la correttezza, assicuratevi anche di revisionare la vostra patch -dopo- averla generata con *`diff(1)`*.

Se le vostre modifiche producono molte differenze, allora dovrete dividerle in patch indipendenti che modificano le cose in passi logici; leggete split_changes.

Questo faciliterà la revisione da parte degli altri sviluppatori, il che è molto importante se volete che la patch venga accettata.

Se state utilizzando `git`, `git rebase -i` può aiutarvi nel procedimento. Se non usate `git`, un'alternativa popolare è `quilt` <http://savannah.nongnu.org/projects/quilt>.

2) Descrivete le vostre modifiche

Descrivete il vostro problema. Esiste sempre un problema che via ha spinto ha fare il vostro lavoro, che sia la correzione di un baco da una riga o una nuova funzionalità da 5000 righe di codice. Convincete i revisori che vale la pena risolvere il vostro problema e che ha senso continuare a leggere oltre al primo paragrafo.

Descrivete ciò che sarà visibile agli utenti. Chiari incidenti nel sistema e blocchi sono abbastanza convincenti, ma non tutti i bachi sono così evidenti. Anche se il problema è stato scoperto durante la revisione del codice, descrivete l'impatto che questo avrà sugli utenti. Tenete presente che la maggior parte delle installazioni Linux usa un kernel che arriva dai sorgenti stabili o dai sorgenti di una distribuzione particolare che prende singolarmente le patch dai sorgenti principali; quindi, includete tutte le informazioni che possono essere utili a capire le vostre modifiche: le circostanze che causano il problema, estratti da dmesg, descrizioni di un incidente di sistema, prestazioni di una regressione, picchi di latenza, blocchi, eccetera.

Quantificare le ottimizzazioni e i compromessi. Se affermate di aver migliorato le prestazioni, il consumo di memoria, l'impatto sollo stack, o la dimensione del file binario, includete dei numeri a supporto della vostra dichiarazione. Ma ricordatevi di descrivere anche eventuali costi che non sono ovvi. Solitamente le ottimizzazioni non sono gratuite, ma sono un compromesso fra l'uso di CPU, la memoria e la leggibilità; o, quando si parla di ipotesi euristiche, fra differenti carichi. Descrivete i lati negativi che vi aspettate dall'ottimizzazione cosicché i revisori possano valutare i costi e i benefici.

Una volta che il problema è chiaro, descrivete come lo risolvete andando nel dettaglio tecnico. È molto importante che descriviate la modifica in un inglese semplice cosicché i revisori possano verificare che il codice si comporti come descritto.

I manutentori vi saranno grati se scrivete la descrizione della patch in un formato che sia compatibile con il gestore dei sorgenti usato dal kernel, `git`, come un "commit log". Leggete *15) Usare esplicitamente In-Reply-To nell'intestazione*.

Risolvete solo un problema per patch. Se la vostra descrizione inizia ad essere lunga, potrebbe essere un segno che la vostra patch necessita d'essere divisa. Leggete split_changes.

Quando inviate o rinviate una patch o una serie, includete la descrizione completa delle modifiche e la loro giustificazione. Non limitatevi a dire che questa è la versione N della patch (o serie). Non aspettatevi che i manutentori di un sottosistema vadano a cercare le versioni precedenti per cercare la descrizione da aggiungere. In pratica, la patch (o serie) e la sua descrizione devono essere un'unica cosa. Questo aiuta i manutentori e i revisori. Probabilmente, alcuni revisori non hanno nemmeno ricevuto o visto le versioni precedenti della patch.

Descrivete le vostro modifiche usando l'imperativo, per esempio "make xyzzy do frotz" piuttosto che "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", come se steste dando ordini al codice di cambiare il suo comportamento.

Se la patch corregge un baco conosciuto, fare riferimento a quel baco inserendo il suo numero o il suo URL. Se la patch è la conseguenza di una discussione su una lista di discussione, allora fornite l'URL all'archivio di quella discussione; usate i collegamenti a https://lkml.kernel.org/ con il `Message-Id`, in questo modo vi assicurerete che il collegamento non diventi invalido nel tempo.

Tuttavia, cercate di rendere la vostra spiegazione comprensibile anche senza far riferimento a fonti esterne. In aggiunta ai collegamenti a bachi e liste di discussione, riassumente i punti più importanti della discussione che hanno portato alla creazione della patch.

Se volete far riferimento a uno specifico commit, non usate solo l'identificativo SHA-1. Per cortesia, aggiungete anche la breve riga riassuntiva del commit per rendere la chiaro ai revisori l'oggetto. Per esempio:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

Dovreste anche assicurarvi di usare almeno i primi 12 caratteri dell'identificativo SHA-1. Il repositorio del kernel ha *molti* oggetti e questo rende possibile la collisione fra due identificativi con pochi caratteri. Tenete ben presente che anche se oggi non ci sono collisioni con il vostro identificativo a 6 caratteri, potrebbero essercene fra 5 anni da oggi.

Se la vostra patch corregge un baco in un commit specifico, per esempio avete trovato un problema usando `git bisect`, per favore usate l'etichetta 'Fixes:' indicando i primi 12 caratteri dell'identificativo SHA-1 seguiti dalla riga riassuntiva. Per esempio:

```
Fixes: e21d2170f366 ("video: remove unnecessary platform_set_
↪drvdata()")
```

La seguente configurazione di `git config` può essere usata per formattare i risultati dei comandi `git log` o `git show` come nell'esempio precedente:

```
[core]
        abbrev = 12
[pretty]
        fixes = Fixes: %h (\"%s\")
```

3) Separate le vostre modifiche

Separate ogni **cambiamento logico** in patch distinte.

Per esempio, se i vostri cambiamenti per un singolo driver includono sia delle correzioni di bachi che miglioramenti alle prestazioni, allora separateli in due o più patch. Se i vostri cambiamenti includono un aggiornamento dell'API e un nuovo driver che lo sfrutta, allora separateli in due patch.

D'altro canto, se fate una singola modifica su più file, raggruppate tutte queste modifiche in una singola patch. Dunque, un singolo cambiamento logico è contenuto in una sola patch.

Il punto da ricordare è che ogni modifica dovrebbe fare delle modifiche che siano facilmente comprensibili e che possano essere verificate dai revisori. Ogni patch dovrebbe essere giustificabile di per sé.

Se al fine di ottenere un cambiamento completo una patch dipende da un'altra, va bene. Semplicemente scrivete una nota nella descrizione della patch per farlo presente: **"this patch depends on patch X"**.

Quando dividete i vostri cambiamenti in una serie di patch, prestate particolare attenzione alla verifica di ogni patch della serie; per ognuna il kernel deve compilare ed essere eseguito correttamente. Gli sviluppatori che usano `git bisect` per scovare i problemi potrebbero finire nel mezzo della vostra serie in un punto qualsiasi; non vi saranno grati se nel mezzo avete introdotto dei bachi.

Se non potete condensare la vostra serie di patch in una più piccola, allora pubblicatene una quindicina alla volta e aspettate che vengano revisionate ed integrate.

4) Verificate lo stile delle vostre modifiche

Controllate che la vostra patch non violi lo stile del codice, maggiori dettagli sono disponibili in Stile del codice per il kernel Linux. Non farlo porta semplicemente a una perdita di tempo da parte dei revisori e voi vedrete la vostra patch rifiutata, probabilmente senza nemmeno essere stata letta.

Un'eccezione importante si ha quando del codice viene spostato da un file ad un altro – in questo caso non dovreste modificare il codice spostato per nessun motivo, almeno non nella patch che lo sposta. Questo separa chiaramente l'azione di spostare il codice e il vostro cambiamento. Questo aiuta enormemente la revisione delle vere differenze e permette agli strumenti di tenere meglio la traccia della storia del codice.

Prima di inviare una patch, verificatene lo stile usando l'apposito verificatore (scripts/checkpatch.pl). Da notare, comunque, che il verificator di stile dovrebbe essere visto come una guida, non come un sostituto al giudizio umano. Se il vostro codice è migliore nonostante una violazione dello stile, probabilmente è meglio lasciarlo com'è.

Il verificatore ha tre diversi livelli di severità:

- ERROR: le cose sono molto probabilmente sbagliate
- WARNING: le cose necessitano d'essere revisionate con attenzione

- CHECK: le cose necessitano di un pensierino

Dovreste essere in grado di giustificare tutte le eventuali violazioni rimaste nella vostra patch.

5) Selezionate i destinatari della vostra patch

Dovreste sempre inviare una copia della patch ai manutentori dei sottosistemi interessati dalle modifiche; date un' occhiata al file MAINTAINERS e alla storia delle revisioni per scoprire chi si occupa del codice. Lo script scripts/get_maintainer.pl può esservi d' aiuto. Se non riuscite a trovare un manutentore per il sottosistema su cui state lavorando, allora Andrew Morton (akpm@linux-foundation.org) sarà la vostra ultima possibilità.

Normalmente, dovreste anche scegliere una lista di discussione a cui inviare la vostra serie di patch. La lista di discussione linux-kernel@vger.kernel.org è proprio l' ultima spiaggia, il volume di email su questa lista fa si che diversi sviluppatori non la seguano. Guardate nel file MAINTAINERS per trovare la lista di discussione dedicata ad un sottosistema; probabilmente lì la vostra patch riceverà molta più attenzione. Tuttavia, per favore, non spammate le liste di discussione che non sono interessate al vostro lavoro.

Molte delle liste di discussione relative al kernel vengono ospitate su vger.kernel.org; potete trovare un loro elenco alla pagina http://vger.kernel.org/vger-lists.html. Tuttavia, ci sono altre liste di discussione ospitate altrove.

Non inviate più di 15 patch alla volta sulle liste di discussione vger!!!

L' ultimo giudizio sull' integrazione delle modifiche accettate spetta a Linux Torvalds. Il suo indirizzo e-mail è <torvalds@linux-foundation.org>. Riceve moltissime e-mail, e, a questo punto, solo poche patch passano direttamente attraverso il suo giudizio; quindi, dovreste fare del vostro meglio per -evitare di-inviargli e-mail.

Se avete una patch che corregge un baco di sicurezza che potrebbe essere sfruttato, inviatela a security@kernel.org. Per bachi importanti, un breve embargo potrebbe essere preso in considerazione per dare il tempo alle distribuzioni di prendere la patch e renderla disponibile ai loro utenti; in questo caso, ovviamente, la patch non dovrebbe essere inviata su alcuna lista di discussione pubblica.

Patch che correggono bachi importanti su un kernel già rilasciato, dovrebbero essere inviate ai manutentori dei kernel stabili aggiungendo la seguente riga:

```
Cc: stable@vger.kernel.org
```

nella vostra patch, nell' area dedicata alle firme (notate, NON come destinatario delle e-mail). In aggiunta a questo file, dovreste leggere anche Tutto quello che volevate sapere sui rilasci -stable di Linux

Tuttavia, notate, che alcuni manutentori di sottosistema preferiscono avere l' ultima parola su quali patch dovrebbero essere aggiunte ai kernel stabili. La rete di manutentori, in particolare, non vorrebbe vedere i singoli sviluppatori aggiungere alle loro patch delle righe come quella sopracitata.

Se le modifiche hanno effetti sull'interfaccia con lo spazio utente, per favore inviate una patch per le pagine man ai manutentori di suddette pagine (elencati nel file MAINTAINERS), o almeno una notifica circa la vostra modifica, cosicché l'informazione possa trovare la sua strada nel manuale. Le modifiche all'API dello spazio utente dovrebbero essere inviate in copia anche a linux-api@vger.kernel.org.

Per le piccole patch potreste aggiungere in CC l'indirizzo *Trivial Patch Monkey trivial@kernel.org* che ha lo scopo di raccogliere le patch "banali". Date uno sguardo al file MAINTAINERS per vedere chi è l'attuale amministratore.

Le patch banali devono rientrare in una delle seguenti categorie:

- errori grammaticali nella documentazione
- errori grammaticali negli errori che potrebbero rompere *grep(1)*
- correzione di avvisi di compilazione (riempirsi di avvisi inutili è negativo)
- correzione di errori di compilazione (solo se correggono qualcosa sul serio)
- rimozione di funzioni/macro deprecate
- sostituzione di codice non potabile con uno portabile (anche in codice specifico per un'architettura, dato che le persone copiano, fintanto che la modifica sia banale)
- qualsiasi modifica dell'autore/manutentore di un file (in pratica "patch monkey" in modalità ritrasmissione)

6) Niente: MIME, links, compressione, allegati. Solo puro testo

Linus e gli altri sviluppatori del kernel devono poter commentare le modifiche che sottomettete. Per uno sviluppatore è importante essere in grado di "citare" le vostre modifiche, usando normali programmi di posta elettronica, cosicché sia possibile commentare una porzione specifica del vostro codice.

Per questa ragione tutte le patch devono essere inviate via e-mail come testo.

> **Warning:** Se decidete di copiare ed incollare la patch nel corpo dell'e-mail, state attenti che il vostro programma non corrompa il contenuto con andate a capo automatiche.

La patch non deve essere un allegato MIME, compresso o meno. Molti dei più popolari programmi di posta elettronica non trasmettono un allegato MIME come puro testo, e questo rende impossibile commentare il vostro codice. Inoltre, un allegato MIME rende l'attività di Linus più laboriosa, diminuendo così la possibilità che il vostro allegato-MIME venga accettato.

Eccezione: se il vostro servizio di posta storpia le patch, allora qualcuno potrebbe chiedervi di rinviarle come allegato MIME.

Leggete Informazioni sui programmi di posta elettronica per Linux per dei suggerimenti sulla configurazione del programmi di posta elettronica per l'invio di patch intatte.

7) Dimensione delle e-mail

Le grosse modifiche non sono adatte ad una lista di discussione, e nemmeno per alcuni manutentori. Se la vostra patch, non compressa, eccede i 300 kB di spazio, allora caricatela in una spazio accessibile su internet fornendo l'URL (collegamento) ad essa. Ma notate che se la vostra patch eccede i 300 kB è quasi certo che necessiti comunque di essere spezzettata.

8) Rispondere ai commenti di revisione

Quasi certamente i revisori vi invieranno dei commenti su come migliorare la vostra patch. Dovete rispondere a questi commenti; ignorare i revisori è un ottimo modo per essere ignorati. Riscontri o domande che non conducono ad una modifica del codice quasi certamente dovrebbero portare ad un commento nel changelog cosicché il prossimo revisore potrà meglio comprendere cosa stia accadendo.

Assicuratevi di dire ai revisori quali cambiamenti state facendo e di ringraziarli per il loro tempo. Revisionare codice è un lavoro faticoso e che richiede molto tempo, e a volte i revisori diventano burberi. Tuttavia, anche in questo caso, rispondete con educazione e concentratevi sul problema che hanno evidenziato.

9) Non scoraggiatevi - o impazientitevi

Dopo che avete inviato le vostre modifiche, siate pazienti e aspettate. I revisori sono persone occupate e potrebbero non ricevere la vostra patch immediatamente.

Un tempo, le patch erano solite scomparire nel vuoto senza alcun commento, ma ora il processo di sviluppo funziona meglio. Dovreste ricevere commenti in una settimana o poco più; se questo non dovesse accadere, assicuratevi di aver inviato le patch correttamente. Aspettate almeno una settimana prima di rinviare le modifiche o sollecitare i revisori - probabilmente anche di più durante la finestra d'integrazione.

10) Aggiungete PATCH nell'oggetto

Dato l'alto volume di e-mail per Linus, e la lista linux-kernel, è prassi prefiggere il vostro oggetto con [PATCH]. Questo permette a Linus e agli altri sviluppatori del kernel di distinguere facilmente le patch dalle altre discussioni.

11) Firmate il vostro lavoro - Il certificato d'origine dello sviluppatore

Per migliorare la tracciabilità su "chi ha fatto cosa", specialmente per quelle patch che per raggiungere lo stadio finale passano attraverso diversi livelli di manutentori, abbiamo introdotto la procedura di "firma" delle patch che vengono inviate per e-mail.

La firma è una semplice riga alla fine della descrizione della patch che certifica che l'avete scritta voi o che avete il diritto di pubblicarla come patch open-source. Le regole sono abbastanza semplici: se potete certificare quanto segue:

Il certificato d'origine dello sviluppatore 1.1

Contribuendo a questo progetto, io certifico che:

(a) Il contributo è stato creato interamente, o in parte, da me e che ho il diritto di inviarlo in accordo con la licenza open-source indicata nel file; oppure

(b) Il contributo è basato su un lavoro precedente che, nei limiti della mia conoscenza, è coperto da un' appropriata licenza open-source che mi da il diritto di modificarlo e inviarlo, le cui modifiche sono interamente o in parte mie, in accordo con la licenza open-source (a meno che non abbia il permesso di usare un' altra licenza) indicata nel file; oppure

(c) Il contributo mi è stato fornito direttamente da qualcuno che ha certificato (a), (b) o (c) e non l' ho modificata.

(d) Capisco e concordo col fatto che questo progetto e i suoi contributi sono pubblici e che un registro dei contributi (incluse tutte le informazioni personali che invio con essi, inclusa la mia firma) verrà mantenuto indefinitamente e che possa essere ridistribuito in accordo con questo progetto o le licenze open-source coinvolte.

poi dovete solo aggiungere una riga che dice:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

usando il vostro vero nome (spiacenti, non si accettano pseudonimi o contributi anonimi).

Alcune persone aggiungono delle etichette alla fine. Per ora queste verranno ignorate, ma potete farlo per meglio identificare procedure aziendali interne o per aggiungere dettagli circa la firma.

Se siete un manutentore di un sottosistema o di un ramo, qualche volta dovrete modificare leggermente le patch che avete ricevuto al fine di poterle integrare; questo perché il codice non è esattamente lo stesso nei vostri sorgenti e in quelli dei vostri contributori. Se rispettate rigidamente la regola (c), dovreste chiedere al mittente di rifare la patch, ma questo è controproducente e una totale perdita di tempo ed energia. La regola (b) vi permette di correggere il codice, ma poi diventa davvero maleducato cambiare la patch di qualcuno e addossargli la responsabilità per i vostri bachi. Per risolvere questo problema dovreste aggiungere una riga, fra l' ultimo Signed-off-by e il vostro, che spiega la vostra modifica. Nonostante non ci sia nulla di obbligatorio, un modo efficace è quello di indicare il vostro nome o indirizzo email fra parentesi quadre, seguito da una breve descrizione; questo renderà abbastanza visibile chi è responsabile per le modifiche dell' ultimo minuto. Per esempio:

```
Signed-off-by: Random J Developer <random@developer.example.org>
[lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>
```

Questa pratica è particolarmente utile se siete i manutentori di un ramo stabile ma al contempo volete dare credito agli autori, tracciare e integrare le modifiche, e proteggere i mittenti dalle lamentele. Notate che in nessuna circostanza è permessa la modifica dell' identità dell' autore (l' intestazione From), dato che è quella

che appare nei changelog.

Un appunto speciale per chi porta il codice su vecchie versioni. Sembra che sia comune l'utile pratica di inserire un'indicazione circa l'origine della patch all'inizio del messaggio di commit (appena dopo la riga dell' oggetto) al fine di migliorare la tracciabilità. Per esempio, questo è quello che si vede nel rilascio stabile 3.x-stable:

```
Date:   Tue Oct 7 07:26:38 2014 -0400

  libata: Un-break ATA blacklist

  commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.
```

E qui quello che potrebbe vedersi su un kernel più vecchio dove la patch è stata applicata:

```
Date:   Tue May 13 22:12:27 2008 +0200

    wireless, airo: waitbusy() won't delay

    [backport of 2.6 commit␣
↪b7acbdfbd1f277c1eb23f344f899cfa4cd0bf36a]
```

Qualunque sia il formato, questa informazione fornisce un importante aiuto alle persone che vogliono seguire i vostri sorgenti, e quelle che cercano dei bachi.

12) Quando utilizzare Acked-by:, Cc:, e Co-developed-by:

L' etichetta Signed-off-by: indica che il firmatario è stato coinvolto nello sviluppo della patch, o che era nel suo percorso di consegna.

Se una persona non è direttamente coinvolta con la preparazione o gestione della patch ma desidera firmare e mettere agli atti la loro approvazione, allora queste persone possono chiedere di aggiungere al changelog della patch una riga Acked-by:.

Acked-by: viene spesso utilizzato dai manutentori del sottosistema in oggetto quando quello stesso manutentore non ha contribuito né trasmesso la patch.

Acked-by: non è formale come Signed-off-by:. Questo indica che la persona ha revisionato la patch e l' ha trovata accettabile. Per cui, a volte, chi integra le patch convertirà un "sì, mi sembra che vada bene" in un Acked-by: (ma tenete presente che solitamente è meglio chiedere esplicitamente).

Acked-by: non indica l' accettazione di un' intera patch. Per esempio, quando una patch ha effetti su diversi sottosistemi e ha un Acked-by: da un manutentore di uno di questi, significa che il manutentore accetta quella parte di codice relativa al sottosistema che mantiene. Qui dovremmo essere giudiziosi. Quando si hanno dei dubbi si dovrebbe far riferimento alla discussione originale negli archivi della lista di discussione.

Se una persona ha avuto l' opportunità di commentare la patch, ma non lo ha fatto, potete aggiungere l' etichetta `Cc:` alla patch. Questa è l' unica etichetta che può

essere aggiunta senza che la persona in questione faccia alcunché - ma dovrebbe indicare che la persona ha ricevuto una copia della patch. Questa etichetta documenta che terzi potenzialmente interessati sono stati inclusi nella discussione.

Co-developed-by: indica che la patch è stata cosviluppata da diversi sviluppatori; viene usato per assegnare più autori (in aggiunta a quello associato all'etichetta From:) quando più persone lavorano ad una patch. Dato che Co-developed-by: implica la paternità della patch, ogni Co-developed-by: dev'essere seguito immediatamente dal Signed-off-by: del corrispondente coautore. Qui si applica la procedura di base per sign-off, in pratica l'ordine delle etichette Signed-off-by: dovrebbe riflettere il più possibile l'ordine cronologico della storia della patch, indipendentemente dal fatto che la paternità venga assegnata via From: o Co-developed-by:. Da notare che l'ultimo Signed-off-by: dev'essere quello di colui che ha sottomesso la patch.

Notate anche che l'etichetta From: è opzionale quando l'autore in From: è anche la persona (e indirizzo email) indicato nel From: dell'intestazione dell'email.

Esempio di una patch sottomessa dall'autore in From::

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

Esempio di una patch sottomessa dall'autore Co-developed-by::

```
From: From Author <from@author.example.org>

<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

13) Utilizzare Reported-by:, Tested-by:, Reviewed-by:, Suggested-by: e Fixes:

L'etichetta Reported-by da credito alle persone che trovano e riportano i bachi e si spera che questo possa ispirarli ad aiutarci nuovamente in futuro. Rammentate che se il baco è stato riportato in privato, dovrete chiedere il permesso prima di poter utilizzare l'etichetta Reported-by.

L'etichetta Tested-by: indica che la patch è stata verificata con successo (su un qualche sistema) dalla persona citata. Questa etichetta informa i manutentori che qualche verifica è stata fatta, fornisce un mezzo per trovare persone che possano

verificare il codice in futuro, e garantisce che queste stesse persone ricevano credito per il loro lavoro.

Reviewd-by:, invece, indica che la patch è stata revisionata ed è stata considerata accettabile in accordo con la dichiarazione dei revisori:

Dichiarazione di svista dei revisori

Offrendo la mia etichetta Reviewed-by, dichiaro quanto segue:

(a) Ho effettuato una revisione tecnica di questa patch per valutarne l'adeguatezza ai fini dell'inclusione nel ramo principale del kernel.

(b) Tutti i problemi e le domande riguardanti la patch sono stati comunicati al mittente. Sono soddisfatto dalle risposte del mittente.

(c) Nonostante ci potrebbero essere cose migliorabili in queste sottomissione, credo che sia, in questo momento, (1) una modifica di interesse per il kernel, e (2) libera da problemi che potrebbero metterne in discussione l'integrazione.

(d) Nonostante abbia revisionato la patch e creda che vada bene, non garantisco (se non specificato altrimenti) che questa otterrà quello che promette o funzionerà correttamente in tutte le possibili situazioni.

L'etichetta Reviewed-by è la dichiarazione di un parere sulla bontà di una modifica che si ritiene appropriata e senza alcun problema tecnico importante. Qualsiasi revisore interessato (quelli che lo hanno fatto) possono offrire il proprio Reviewed-by per la patch. Questa etichetta serve a dare credito ai revisori e a informare i manutentori sul livello di revisione che è stato fatto sulla patch. L'etichetta Reviewd-by, quando fornita da revisori conosciuti per la loro conoscenza sulla materia in oggetto e per la loro serietà nella revisione, accrescerà le probabilità che la vostra patch venga integrate nel kernel.

L'etichetta Suggested-by: indica che l'idea della patch è stata suggerita dalla persona nominata e le da credito. Tenete a mente che questa etichetta non dovrebbe essere aggiunta senza un permesso esplicito, specialmente se l'idea non è stata pubblicata in un forum pubblico. Detto ciò, dando credito a chi ci fornisce delle idee, si spera di poterli ispirare ad aiutarci nuovamente in futuro.

L'etichetta Fixes: indica che la patch corregge un problema in un commit precedente. Serve a chiarire l'origine di un baco, il che aiuta la revisione del baco stesso. Questa etichetta è di aiuto anche per i manutentori dei kernel stabili al fine di capire quale kernel deve ricevere la correzione. Questo è il modo suggerito per indicare che un baco è stato corretto nella patch. Per maggiori dettagli leggete *2) Descrivete le vostre modifiche*

14) Il formato canonico delle patch

Questa sezione descrive il formato che dovrebbe essere usato per le patch. Notate che se state usando un repositorio `git` per salvare le vostre patch potere usare il comando `git format-patch` per ottenere patch nel formato appropriato. Lo strumento non crea il testo necessario, per cui, leggete le seguenti istruzioni.

L'oggetto di una patch canonica è la riga:

```
Subject: [PATCH 001/123] subsystem: summary phrase
```

Il corpo di una patch canonica contiene i seguenti elementi:

- Una riga `from` che specifica l'autore della patch, seguita da una riga vuota (necessaria soltanto se la persona che invia la patch non ne è l'autore).
- Il corpo della spiegazione, con linee non più lunghe di 75 caratteri, che verrà copiato permanentemente nel changelog per descrivere la patch.
- Una riga vuota
- Le righe `Signed-off-by:`, descritte in precedenza, che finiranno anch'esse nel changelog.
- Una linea di demarcazione contenente semplicemente `---`.
- Qualsiasi altro commento che non deve finire nel changelog.
- Le effettive modifiche al codice (il prodotto di `diff`).

Il formato usato per l'oggetto permette ai programmi di posta di usarlo per ordinare le patch alfabeticamente - tutti i programmi di posta hanno questa funzionalità - dato che al numero sequenziale si antepongono degli zeri; in questo modo l'ordine numerico ed alfabetico coincidono.

Il `subsystem` nell'oggetto dell'email dovrebbe identificare l'area o il sottosistema modificato dalla patch.

La `summary phrase` nell'oggetto dell'email dovrebbe descrivere brevemente il contenuto della patch. La `summary phrase` non dovrebbe essere un nome di file. Non utilizzate la stessa `summary phrase` per tutte le patch in una serie (dove una `serie di patch` è una sequenza ordinata di diverse patch correlate).

Ricordatevi che la `summary phrase` della vostra email diventerà un identificatore globale ed unico per quella patch. Si propaga fino al changelog `git`. La `summary phrase` potrà essere usata in futuro dagli sviluppatori per riferirsi a quella patch. Le persone vorranno cercare la `summary phrase` su internet per leggere le discussioni che la riguardano. Potrebbe anche essere l'unica cosa che le persone vedranno quando, in due o tre mesi, riguarderanno centinaia di patch usando strumenti come `gitk` o `git log --oneline`.

Per queste ragioni, dovrebbe essere lunga fra i 70 e i 75 caratteri, e deve descrivere sia cosa viene modificato, sia il perché sia necessario. Essere brevi e descrittivi è una bella sfida, ma questo è quello che fa un riassunto ben scritto.

La `summary phrase` può avere un'etichetta (*tag*) di prefisso racchiusa fra le parentesi quadre "Subject: [PATCH <tag>…] <summary phrase>". Le etichette non verranno considerate come parte della frase riassuntiva, ma indicano come la

patch dovrebbe essere trattata. Fra le etichette più comuni ci sono quelle di versione che vengono usate quando una patch è stata inviata più volte (per esempio, “v1, v2, v3”); oppure “RFC” per indicare che si attendono dei commenti (*Request For Comments*). Se ci sono quattro patch nella serie, queste dovrebbero essere enumerate così: 1/4, 2/4, 3/4, 4/4. Questo assicura che gli sviluppatori capiranno l’ ordine in cui le patch dovrebbero essere applicate, e per tracciare quelle che hanno revisionate o che hanno applicato.

Un paio di esempi di oggetti:

```
Subject: [PATCH 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCH v2 01/27] x86: fix eflags tracking
```

La riga `from` dev’ essere la prima nel corpo del messaggio ed è nel formato:

From: Patch Author <author@example.com>

La riga `from` indica chi verrà accreditato nel changelog permanente come l’ autore della patch. Se la riga `from` è mancante, allora per determinare l’autore da inserire nel changelog verrà usata la riga `From` nell’ intestazione dell’ email.

Il corpo della spiegazione verrà incluso nel changelog permanente, per cui deve aver senso per un lettore esperto che è ha dimenticato i dettagli della discussione che hanno portato alla patch. L’ inclusione di informazioni sui problemi oggetto dalla patch (messaggi del kernel, messaggi di oops, eccetera) è particolarmente utile per le persone che potrebbero cercare fra i messaggi di log per la patch che li tratta. Se la patch corregge un errore di compilazione, non sarà necessario includere proprio _tutto_ quello che è uscito dal compilatore; aggiungete solo quello che è necessario per far si che la vostra patch venga trovata. Come nella `summary phrase`, è importante essere sia brevi che descrittivi.

La linea di demarcazione `---` serve essenzialmente a segnare dove finisce il messaggio di changelog.

Aggiungere il `diffstat` dopo `---` è un buon uso di questo spazio, per mostrare i file che sono cambiati, e il numero di file aggiunto o rimossi. Un `diffstat` è particolarmente utile per le patch grandi. Altri commenti che sono importanti solo per i manutentori, quindi inadatti al changelog permanente, dovrebbero essere messi qui. Un buon esempio per questo tipo di commenti potrebbe essere quello di descrivere le differenze fra le versioni della patch.

Se includete un `diffstat` dopo `---`, usate le opzioni `-p 1 -w70` cosicché i nomi dei file elencati non occupino troppo spazio (facilmente rientreranno negli 80 caratteri, magari con qualche indentazione). (`git` genera di base dei diffstat adatti).

Maggiori dettagli sul formato delle patch nei riferimenti qui di seguito.

15) Usare esplicitamente In-Reply-To nell'intestazione

Aggiungere manualmente In-Reply-To: nell'intestazione dell'e-mail potrebbe essere d'aiuto per associare una patch ad una discussione precedente, per esempio per collegare la correzione di un baco con l'e-mail che lo riportava. Tuttavia, per serie di patch multiple è generalmente sconsigliato l'uso di In-Reply-To: per collegare precedenti versioni. In questo modo versioni multiple di una patch non diventeranno un'ingestibile giungla di riferimenti all'interno dei programmi di posta. Se un collegamento è utile, potete usare https://lkml.kernel.org/ per ottenere i collegamenti ad una versione precedente di una serie di patch (per esempio, potete usarlo per l'email introduttiva alla serie).

16) Inviare richieste `git pull`

Se avete una serie di patch, potrebbe essere più conveniente per un manutentore tirarle dentro al repositorio del sottosistema attraverso l'operazione `git pull`. Comunque, tenete presente che prendere patch da uno sviluppatore in questo modo richiede un livello di fiducia più alto rispetto a prenderle da una lista di discussione. Di conseguenza, molti manutentori sono riluttanti ad accettare richieste di *pull*, specialmente dagli sviluppatori nuovi e quindi sconosciuti. Se siete in dubbio, potete fare una richiesta di *pull* come messaggio introduttivo ad una normale pubblicazione di patch, così il manutentore avrà la possibilità di scegliere come integrarle.

Una richiesta di *pull* dovrebbe avere nell'oggetto [GIT] o [PULL]. La richiesta stessa dovrebbe includere il nome del repositorio e quello del ramo su una singola riga; dovrebbe essere più o meno così:

```
Please pull from

    git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus

to get these changes:
```

Una richiesta di *pull* dovrebbe includere anche un messaggio generico che dica cos'è incluso, una lista delle patch usando `git shortlog`, e una panoramica sugli effetti della serie di patch con `diffstat`. Il modo più semplice per ottenere tutte queste informazioni è, ovviamente, quello di lasciar fare tutto a `git` con il comando `git request-pull`.

Alcuni manutentori (incluso Linus) vogliono vedere le richieste di *pull* da commit firmati con GPG; questo fornisce una maggiore garanzia sul fatto che siate stati proprio voi a fare la richiesta. In assenza di tale etichetta firmata Linus, in particolare, non prenderà alcuna patch da siti pubblici come GitHub.

Il primo passo verso la creazione di questa etichetta firmata è quello di creare una chiave GNUPG ed averla fatta firmare da uno o più sviluppatori principali del kernel. Questo potrebbe essere difficile per i nuovi sviluppatori, ma non ci sono altre vie. Andare alle conferenze potrebbe essere un buon modo per trovare sviluppatori che possano firmare la vostra chiave.

Una volta che avete preparato la vostra serie di patch in `git`, e volete che qualcuno

le prenda, create una etichetta firmata col comando `git tag -s`. Questo creerà una nuova etichetta che identifica l'ultimo commit della serie contenente una firma creata con la vostra chiave privata. Avrete anche l'opportunità di aggiungere un messaggio di changelog all'etichetta; questo è il posto ideale per descrivere gli effetti della richiesta di *pull*.

Se i sorgenti da cui il manutentore prenderà le patch non sono gli stessi del repositorio su cui state lavorando, allora non dimenticatevi di caricare l'etichetta firmata anche sui sorgenti pubblici.

Quando generate una richiesta di *pull*, usate l'etichetta firmata come obiettivo. Un comando come il seguente farà il suo dovere:

```
git request-pull master git://my.public.tree/linux.git my-signed-tag
```

Riferimenti

Andrew Morton, "La patch perfetta" (tpp).
<http://www.ozlabs.org/~akpm/stuff/tpp.txt>

Jeff Garzik, "Formato per la sottomissione di patch per il kernel Linux"
<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

Greg Kroah-Hartman, "Come scocciare un manutentore di un sottosistema"
<http://www.kroah.com/log/linux/maintainer.html>

<http://www.kroah.com/log/linux/maintainer-02.html>

<http://www.kroah.com/log/linux/maintainer-03.html>

<http://www.kroah.com/log/linux/maintainer-04.html>

<http://www.kroah.com/log/linux/maintainer-05.html>

<http://www.kroah.com/log/linux/maintainer-06.html>

No!!!! Basta gigantesche bombe patch alle persone sulla lista linux-kernel@vger.kernel.org!
<https://lkml.org/lkml/2005/7/11/336>

Kernel Documentation/translations/it_IT/process/coding-style.rst:
Stile del codice per il kernel Linux

E-mail di Linus Torvalds sul formato canonico di una patch:
<http://lkml.org/lkml/2005/4/7/183>

Andi Kleen, "Su come sottomettere patch del kernel"
Alcune strategie su come sottomettere modifiche toste o controverse.

http://halobates.de/on-submitting-patches.pdf

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/programming-language.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Linguaggio di programmazione

Il kernel è scritto nel linguaggio di programmazione C [it-c-language]. Più precisamente, il kernel viene compilato con `gcc` [it-gcc] usando l'opzione `-std=gnu89` [it-gcc-c-dialect-options]: il dialetto GNU dello standard ISO C90 (con l'aggiunta di alcune funzionalità da C99)

Questo dialetto contiene diverse estensioni al linguaggio [it-gnu-extensions], e molte di queste vengono usate sistematicamente dal kernel.

Il kernel offre un certo livello di supporto per la compilazione con `clang` [it-clang] e `icc` [it-icc] su diverse architetture, tuttavia in questo momento il supporto non è completo e richiede delle patch aggiuntive.

Attributi

Una delle estensioni più comuni e usate nel kernel sono gli attributi [it-gcc-attribute-syntax]. Gli attributi permettono di aggiungere una semantica, definita dell'implementazione, alle entità del linguaggio (come le variabili, le funzioni o i tipi) senza dover fare importanti modifiche sintattiche al linguaggio stesso (come l'aggiunta di nuove parole chiave) [it-n2049].

In alcuni casi, gli attributi sono opzionali (ovvero un compilatore che non dovesse supportarli dovrebbe produrre comunque codice corretto, anche se più lento o che non esegue controlli aggiuntivi durante la compilazione).

Il kernel definisce alcune pseudo parole chiave (per esempio `__pure`) in alternativa alla sintassi GNU per gli attributi (per esempio `__attribute__((__pure__))`) allo scopo di mostrare quali funzionalità si possono usare e/o per accorciare il codice.

Per maggiori informazioni consultate il file d'intestazione `include/linux/compiler_attributes.h`.

> **Warning:** In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/coding-style.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Stile del codice per il kernel Linux

Questo è un breve documento che descrive lo stile di codice preferito per il kernel Linux. Lo stile di codifica è molto personale e non voglio **forzare** nessuno ad accettare il mio, ma questo stile è quello che dev'essere usato per qualsiasi cosa che io sia in grado di mantenere, e l'ho preferito anche per molte altre cose. Per favore, almeno tenete in considerazione le osservazioni espresse qui.

La prima cosa che suggerisco è quella di stamparsi una copia degli standard di codifica GNU e di NON leggerla. Bruciatela, è un grande gesto simbolico.

Comunque, ecco i punti:

1) Indentazione

La tabulazione (tab) è di 8 caratteri e così anche le indentazioni. Ci sono alcuni movimenti di eretici che vorrebbero l'indentazione a 4 (o perfino 2!) caratteri di profondità, che è simile al tentativo di definire il valore del pi-greco a 3.

Motivazione: l'idea dell'indentazione è di definire chiaramente dove un blocco di controllo inizia e finisce. Specialmente quando siete rimasti a guardare lo schermo per 20 ore a file, troverete molto più facile capire i livelli di indentazione se questi sono larghi.

Ora, alcuni rivendicano che un'indentazione da 8 caratteri sposta il codice troppo a destra e che quindi rende difficile la lettura su schermi a 80 caratteri. La risposta a questa affermazione è che se vi servono più di 3 livelli di indentazione, siete comunque fregati e dovreste correggere il vostro programma.

In breve, l'indentazione ad 8 caratteri rende più facile la lettura, e in aggiunta vi avvisa quando state annidando troppo le vostre funzioni. Tenete ben a mente questo avviso.

Al fine di facilitare l'indentazione del costrutto switch, si preferisce allineare sulla stessa colonna la parola chiave `switch` e i suoi subordinati `case`. In questo modo si evita una doppia indentazione per i `case`. Un esempio.:

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
        mem <<= 10;
        /* fall through */
default:
        break;
}
```

A meno che non vogliate nascondere qualcosa, non mettete più istruzioni sulla stessa riga:

```
if (condition) do_this;
  do_something_everytime;
```

né mettete più assegnamenti sulla stessa riga. Lo stile del kernel è ultrasemplice. Evitate espressioni intricate.

Al di fuori dei commenti, della documentazione ed escludendo i Kconfig, gli spazi non vengono mai usati per l'indentazione, e l'esempio qui sopra è volutamente errato.

Procuratevi un buon editor di testo e non lasciate spazi bianchi alla fine delle righe.

2) Spezzare righe lunghe e stringhe

Lo stile del codice riguarda la leggibilità e la manutenibilità utilizzando strumenti comuni.

Il limite delle righe è di 80 colonne e questo e un limite fortemente desiderato.

Espressioni più lunghe di 80 colonne saranno spezzettate in pezzi più piccoli, a meno che eccedere le 80 colonne non aiuti ad aumentare la leggibilità senza nascondere informazioni. I pezzi derivati sono sostanzialmente più corti degli originali e vengono posizionati più a destra. Lo stesso si applica, nei file d'intestazione, alle funzioni con una lista di argomenti molto lunga. Tuttavia, non spezzettate mai le stringhe visibili agli utenti come i messaggi di printk, questo perché inibireste la possibilità d'utilizzare grep per cercarle.

3) Posizionamento di parentesi graffe e spazi

Un altro problema che s'affronta sempre quando si parla di stile in C è il posizionamento delle parentesi graffe. Al contrario della dimensione dell'indentazione, non ci sono motivi tecnici sulla base dei quali scegliere una strategia di posizionamento o un'altra; ma il modo qui preferito, come mostratoci dai profeti Kernighan e Ritchie, è quello di posizionare la parentesi graffa di apertura per ultima sulla riga, e quella di chiusura per prima su una nuova riga, così:

```
if (x is true) {
        we do y
}
```

Questo è valido per tutte le espressioni che non siano funzioni (if, switch, for, while, do). Per esempio:

```
switch (action) {
case KOBJ_ADD:
        return "add";
case KOBJ_REMOVE:
        return "remove";
```

(continues on next page)

(continued from previous page)

```
case KOBJ_CHANGE:
        return "change";
default:
        return NULL;
}
```

Tuttavia, c'è il caso speciale, le funzioni: queste hanno la parentesi graffa di apertura all'inizio della riga successiva, quindi:

```
int function(int x)
{
        body of function
}
```

Eretici da tutto il mondo affermano che questa incoerenza è ⋯insomma ⋯incoerente, ma tutte le persone ragionevoli sanno che (a) K&R hanno **ragione** e (b) K&R hanno ragione. A parte questo, le funzioni sono comunque speciali (non potete annidarle in C).

Notate che la graffa di chiusura è da sola su una riga propria, ad **eccezione** di quei casi dove è seguita dalla continuazione della stessa espressione, in pratica `while` nell'espressione do-while, oppure `else` nell'espressione if-else, come questo:

```
do {
        body of do-loop
} while (condition);
```

e

```
if (x == y) {
        ..
} else if (x > y) {
        ...
} else {
        ....
}
```

Motivazione: K&R.

Inoltre, notate che questo posizionamento delle graffe minimizza il numero di righe vuote senza perdere di leggibilità. In questo modo, dato che le righe sul vostro schermo non sono una risorsa illimitata (pensate ad uno terminale con 25 righe), avrete delle righe vuote da riempire con dei commenti.

Non usate inutilmente le graffe dove una singola espressione è sufficiente.

```
if (condition)
        action();
```

e

```
if (condition)
        do_this();
else
        do_that();
```

Questo non vale nel caso in cui solo un ramo dell'espressione if-else contiene una sola espressione; in quest'ultimo caso usate le graffe per entrambe i rami:

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

Inoltre, usate le graffe se un ciclo contiene più di una semplice istruzione:

```
while (condition) {
        if (test)
                do_something();
}
```

3.1) Spazi

Lo stile del kernel Linux per quanto riguarda gli spazi, dipende (principalmente) dalle funzioni e dalle parole chiave. Usate una spazio dopo (quasi tutte) le parole chiave. L'eccezioni più evidenti sono sizeof, typeof, alignof, e __attribute__, il cui aspetto è molto simile a quello delle funzioni (e in Linux, solitamente, sono usate con le parentesi, anche se il linguaggio non lo richiede; come `sizeof info` dopo aver dichiarato `struct fileinfo info`).

Quindi utilizzate uno spazio dopo le seguenti parole chiave:

```
if, switch, case, for, do, while
```

ma non con sizeof, typeof, alignof, o __attribute__. Ad esempio,

```
s = sizeof(struct file);
```

Non aggiungete spazi attorno (dentro) ad un'espressione fra parentesi. Questo esempio è **brutto**:

```
s = sizeof( struct file );
```

Quando dichiarate un puntatore ad una variabile o una funzione che ritorna un puntatore, il posto suggerito per l'asterisco * è adiacente al nome della variabile o della funzione, e non adiacente al nome del tipo. Esempi:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Usate uno spazio attorno (da ogni parte) alla maggior parte degli operatori binari o ternari, come i seguenti:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

ma non mettete spazi dopo gli operatori unari:

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

nessuno spazio dopo l'operatore unario suffisso di incremento o decremento:

```
++  --
```

nessuno spazio dopo l'operatore unario prefisso di incremento o decremento:

```
++  --
```

e nessuno spazio attorno agli operatori dei membri di una struttura `.` e `->`.

Non lasciate spazi bianchi alla fine delle righe. Alcuni editor con l'indentazione `furba` inseriranno gli spazi bianchi all'inizio di una nuova riga in modo appropriato, quindi potrete scrivere la riga di codice successiva immediatamente. Tuttavia, alcuni di questi stessi editor non rimuovono questi spazi bianchi quando non scrivete nulla sulla nuova riga, ad esempio perché volete lasciare una riga vuota. Il risultato è che finirete per avere delle righe che contengono spazi bianchi in coda.

Git vi avviserà delle modifiche che aggiungono questi spazi vuoti di fine riga, e può opzionalmente rimuoverli per conto vostro; tuttavia, se state applicando una serie di modifiche, questo potrebbe far fallire delle modifiche successive perché il contesto delle righe verrà cambiato.

4) Assegnare nomi

C è un linguaggio spartano, e così dovrebbero esserlo i vostri nomi. Al contrario dei programmatori Modula-2 o Pascal, i programmatori C non usano nomi graziosi come ThisVariableIsATemporaryCounter. Un programmatore C chiamerebbe questa variabile `tmp`, che è molto più facile da scrivere e non è una delle più difficili da capire.

TUTTAVIA, nonostante i nomi con notazione mista siano da condannare, i nomi descrittivi per variabili globali sono un dovere. Chiamare una funzione globale `pippo` è un insulto.

Le variabili GLOBALI (da usare solo se vi servono **davvero**) devono avere dei nomi descrittivi, così come le funzioni globali. Se avete una funzione che conta gli utenti attivi, dovreste chiamarla `count_active_users()` o qualcosa di simile, **non** dovreste chiamarla `cntusr()`.

Codificare il tipo di funzione nel suo nome (quella cosa chiamata notazione ungherese) è stupido - il compilatore conosce comunque il tipo e può verificarli, e inoltre confonde i programmatori. Non c'è da sorprendersi che MicroSoft faccia programmi bacati.

Le variabili LOCALI dovrebbero avere nomi corti, e significativi. Se avete un qualsiasi contatore di ciclo, probabilmente sarà chiamato `i`. Chiamarlo `loop_counter` non è produttivo, non ci sono possibilità che `i` possa non essere capito. Analogamente, `tmp` può essere una qualsiasi variabile che viene usata per salvare temporaneamente un valore.

Se avete paura di fare casino coi nomi delle vostre variabili locali, allora avete un altro problema che è chiamato sindrome dello squilibrio dell'ormone della crescita delle funzioni. Vedere il capitolo 6 (funzioni).

5) Definizione di tipi (typedef)

Per favore non usate cose come `vps_t`. Usare il typedef per strutture e puntatori è uno **sbaglio**. Quando vedete:

```
vps_t a;
```

nei sorgenti, cosa significa? Se, invece, dicesse:

```
struct virtual_container *a;
```

potreste dire cos' è effettivamente `a`.

Molte persone pensano che la definizione dei tipi `migliori la leggibilità`. Non molto. Sono utili per:

(a) gli oggetti completamente opachi (dove typedef viene proprio usato allo scopo di **nascondere** cosa sia davvero l' oggetto).

Esempio: `pte_t` eccetera sono oggetti opachi che potete usare solamente con le loro funzioni accessorie.

Note: Gli oggetti opachi e le `funzioni accessorie` non sono, di per se, una bella cosa. Il motivo per cui abbiamo cose come pte_t eccetera è che davvero non c' è alcuna informazione portabile.

(b) i tipi chiaramente interi, dove l' astrazione **aiuta** ad evitare confusione sul fatto che siano `int` oppure `long`.

u8/u16/u32 sono typedef perfettamente accettabili, anche se ricadono nella categoria (d) piuttosto che in questa.

Note:

Ancora - dev' esserci una **ragione** per farlo. Se qualcosa è `unsigned long`, non c' è alcun bisogno di avere:

typedef unsigned long myfalgs_t;

ma se ci sono chiare circostanze in cui potrebbe essere `unsigned int` e in altre configurazioni `unsigned long`, allora certamente typedef è una buona scelta.

(c) quando di rado create letteralmente dei **nuovi** tipi su cui effettuare verifiche.

(d) circostanze eccezionali, in cui si definiscono nuovi tipi identici a quelli definiti dallo standard C99.

Nonostante ci voglia poco tempo per abituare occhi e cervello all'uso dei tipi standard come `uint32_t`, alcune persone ne obiettano l'uso.

Perciò, i tipi specifici di Linux `u8/u16/u32/u64` e i loro equivalenti con segno, identici ai tipi standard, sono permessi- tuttavia, non sono obbligatori per il nuovo codice.

(e) i tipi sicuri nella spazio utente.

In alcune strutture dati visibili dallo spazio utente non possiamo richiedere l'uso dei tipi C99 e nemmeno i vari `u32` descritti prima. Perciò, utilizziamo `__u32` e tipi simili in tutte le strutture dati condivise con lo spazio utente.

Magari ci sono altri casi validi, ma la regola di base dovrebbe essere di non usare MAI MAI un typedef a meno che non rientri in una delle regole descritte qui.

In generale, un puntatore, o una struttura a cui si ha accesso diretto in modo ragionevole, non dovrebbero **mai** essere definite con un typedef.

6) Funzioni

Le funzioni dovrebbero essere brevi e carine, e fare una cosa sola. Dovrebbero occupare uno o due schermi di testo (come tutti sappiamo, la dimensione di uno schermo secondo ISO/ANSI è di 80x24), e fare una cosa sola e bene.

La massima lunghezza di una funziona è inversamente proporzionale alla sua complessità e al livello di indentazione di quella funzione. Quindi, se avete una funzione che è concettualmente semplice ma che è implementata come un lunga (ma semplice) sequenza di caso-istruzione, dove avete molte piccole cose per molti casi differenti, allora va bene avere funzioni più lunghe.

Comunque, se avete una funzione complessa e sospettate che uno studente non particolarmente dotato del primo anno delle scuole superiori potrebbe non capire cosa faccia la funzione, allora dovreste attenervi strettamente ai limiti. Usate funzioni di supporto con nomi descrittivi (potete chiedere al compilatore di renderle inline se credete che sia necessario per le prestazioni, e probabilmente farà un lavoro migliore di quanto avreste potuto fare voi).

Un'altra misura delle funzioni sono il numero di variabili locali. Non dovrebbero eccedere le 5-10, oppure state sbagliando qualcosa. Ripensate la funzione, e dividetela in pezzettini. Generalmente, un cervello umano può seguire facilmente circa 7 cose diverse, di più lo confonderebbe. Lo sai d'essere brillante, ma magari vorresti riuscire a capire cos'avevi fatto due settimane prima.

Nei file sorgenti, separate le funzioni con una riga vuota. Se la funzione è esportata, la macro **EXPORT** per questa funzione deve seguire immediatamente la riga della parentesi graffa di chiusura. Ad esempio:

```
int system_is_up(void)
{
```

(continues on next page)

(continued from previous page)

```
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

Nei prototipi di funzione, includete i nomi dei parametri e i loro tipi. Nonostante questo non sia richiesto dal linguaggio C, in Linux viene preferito perché è un modo semplice per aggiungere informazioni importanti per il lettore.

Non usate la parola chiave `extern` coi prototipi di funzione perché rende le righe più lunghe e non è strettamente necessario.

7) Centralizzare il ritorno delle funzioni

Sebbene sia deprecata da molte persone, l'istruzione goto è impiegata di frequente dai compilatori sotto forma di salto incondizionato.

L'istruzione goto diventa utile quando una funzione ha punti d'uscita multipli e vanno eseguite alcune procedure di pulizia in comune. Se non è necessario pulire alcunché, allora ritornate direttamente.

Assegnate un nome all'etichetta di modo che suggerisca cosa fa la goto o perché esiste. Un esempio di un buon nome potrebbe essere `out_free_buffer:` se la goto libera (free) un `buffer`. Evitate l'uso di nomi GW-BASIC come `err1:` ed `err2:`, potreste doverli riordinare se aggiungete o rimuovete punti d'uscita, e inoltre rende difficile verificarne la correttezza.

I motivo per usare le goto sono:

- i salti incondizionati sono più facili da capire e seguire
- l'annidamento si riduce
- si evita di dimenticare, per errore, di aggiornare un singolo punto d'uscita
- aiuta il compilatore ad ottimizzare il codice ridondante ;)

```
int fun(int a)
{
        int result = 0;
        char *buffer;

        buffer = kmalloc(SIZE, GFP_KERNEL);
        if (!buffer)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out_free_buffer;
        }
```

(continues on next page)

(continued from previous page)

```
        ...
out_free_buffer:
        kfree(buffer);
        return result;
}
```

Un baco abbastanza comune di cui bisogna prendere nota è il `one err bugs` che assomiglia a questo:

```
err:
        kfree(foo->bar);
        kfree(foo);
        return ret;
```

Il baco in questo codice è che in alcuni punti d'uscita la variabile `foo` è NULL. Normalmente si corregge questo baco dividendo la gestione dell'errore in due parti `err_free_bar:` e `err_free_foo:`:

```
err_free_bar:
       kfree(foo->bar);
err_free_foo:
       kfree(foo);
       return ret;
```

Idealmente, dovreste simulare condizioni d'errore per verificare i vostri percorsi d'uscita.

8) Commenti

I commenti sono una buona cosa, ma c'è anche il rischio di esagerare. MAI spiegare COME funziona il vostro codice in un commento: è molto meglio scrivere il codice di modo che il suo funzionamento sia ovvio, inoltre spiegare codice scritto male è una perdita di tempo.

Solitamente, i commenti devono dire COSA fa il codice, e non COME lo fa. Inoltre, cercate di evitare i commenti nel corpo della funzione: se la funzione è così complessa che dovete commentarla a pezzi, allora dovreste tornare al punto 6 per un momento. Potete mettere dei piccoli commenti per annotare o avvisare il lettore circa un qualcosa di particolarmente arguto (o brutto), ma cercate di non esagerare. Invece, mettete i commenti in testa alla funzione spiegando alle persone cosa fa, e possibilmente anche il PERCHÉ.

Per favore, quando commentate una funzione dell'API del kernel usate il formato kernel-doc. Per maggiori dettagli, leggete i file in :ref:*Documentation/translations/it_IT/doc-guide/* e in `script/kernel-doc`.

Lo stile preferito per i commenti più lunghi (multi-riga) è:

```
/*
 * This is the preferred style for multi-line
```

(continues on next page)

(continued from previous page)

```
 * comments in the Linux kernel source code.
 * Please use it consistently.
 *
 * Description:  A column of asterisks on the left side,
 * with beginning and ending almost-blank lines.
 */
```

Per i file in net/ e in drivers/net/ lo stile preferito per i commenti più lunghi (multi-riga) è leggermente diverso.

```
/* The preferred comment style for files in net/ and drivers/net
 * looks like this.
 *
 * It is nearly the same as the generally preferred comment style,
 * but there is no initial almost-blank line.
 */
```

È anche importante commentare i dati, sia per i tipi base che per tipi derivati. A questo scopo, dichiarate un dato per riga (niente virgole per una dichiarazione multipla). Questo vi lascerà spazio per un piccolo commento per spiegarne l'uso.

9) Avete fatto un pasticcio

Va bene, li facciamo tutti. Probabilmente vi è stato detto dal vostro aiutante Unix di fiducia che `GNU emacs` formatta automaticamente il codice C per conto vostro, e avete notato che sì, in effetti lo fa, ma che i modi predefiniti non sono proprio allettanti (infatti, sono peggio che premere tasti a caso - un numero infinito di scimmie che scrivono in GNU emacs non faranno mai un buon programma).

Quindi, potete sbarazzarvi di GNU emacs, o riconfigurarlo con valori più sensati. Per fare quest'ultima cosa, potete appiccicare il codice che segue nel vostro file .emacs:

```
(defun c-lineup-arglist-tabs-only (ignored)
  "Line up argument lists by tabs, not spaces"
  (let* ((anchor (c-langelem-pos c-syntactic-element))
         (column (c-langelem-2nd-pos c-syntactic-element))
         (offset (- (1+ column) anchor))
         (steps (floor offset c-basic-offset)))
    (* (max steps 1)
       c-basic-offset)))

(dir-locals-set-class-variables
 'linux-kernel
 '((c-mode . (
        (c-basic-offset . 8)
        (c-label-minimum-indentation . 0)
        (c-offsets-alist . (
                (arglist-close         . c-lineup-arglist-tabs-only)
```

(continues on next page)

(continued from previous page)

```
                (arglist-cont-nonempty .
                    (c-lineup-gcc-asm-reg c-lineup-arglist-tabs-
↪only))
                (arglist-intro         . +)
                (brace-list-intro      . +)
                (c                     . c-lineup-C-comments)
                (case-label            . 0)
                (comment-intro         . c-lineup-comment)
                (cpp-define-intro      . +)
                (cpp-macro             . -1000)
                (cpp-macro-cont        . +)
                (defun-block-intro     . +)
                (else-clause           . 0)
                (func-decl-cont        . +)
                (inclass               . +)
                (inher-cont            . c-lineup-multi-inher)
                (knr-argdecl-intro     . 0)
                (label                 . -1000)
                (statement             . 0)
                (statement-block-intro . +)
                (statement-case-intro  . +)
                (statement-cont        . +)
                (substatement          . +)
                ))
        (indent-tabs-mode . t)
        (show-trailing-whitespace . t)
        ))))

(dir-locals-set-directory-class
 (expand-file-name "~/src/linux-trees")
 'linux-kernel)
```

Questo farà funzionare meglio emacs con lo stile del kernel per i file che si trovano nella cartella `~/src/linux-trees`.

Ma anche se doveste fallire nell'ottenere una formattazione sensata in emacs non tutto è perduto: usate `indent`.

Ora, ancora, GNU indent ha la stessa configurazione decerebrata di GNU emacs, ed è per questo che dovete passargli alcune opzioni da riga di comando. Tuttavia, non è così terribile, perché perfino i creatori di GNU indent riconoscono l'autorità di K&R (le persone del progetto GNU non sono cattive, sono solo mal indirizzate sull'argomento), quindi date ad indent le opzioni `-kr -i8` (che significa `K&R, 8 caratteri di indentazione`), o utilizzate `scripts/Lindent` che indenterà usando l'ultimo stile.

`indent` ha un sacco di opzioni, e specialmente quando si tratta di riformattare i commenti dovreste dare un'occhiata alle pagine man. Ma ricordatevi: `indent` non è un correttore per una cattiva programmazione.

Da notare che potete utilizzare anche `clang-format` per aiutarvi con queste regole, per riformattare rapidamente ad automaticamente alcune parti del vostro

codice, e per revisionare interi file al fine di identificare errori di stile, refusi e possibilmente anche delle migliorie. È anche utile per ordinare gli `#include`, per allineare variabili/macro, per ridistribuire il testo e altre cose simili. Per maggiori dettagli, consultate il file clang-format.

10) File di configurazione Kconfig

Per tutti i file di configurazione Kconfig* che si possono trovare nei sorgenti, l'indentazione è un po' differente. Le linee dopo un `config` sono indentate con un tab, mentre il testo descrittivo è indentato di ulteriori due spazi. Esempio:

```
config AUDIT
      bool "Auditing support"
      depends on NET
      help
        Enable auditing infrastructure that can be used with another
        kernel subsystem, such as SELinux (which requires this for
        logging of avc messages output).  Does not do system-call
        auditing without CONFIG_AUDITSYSCALL.
```

Le funzionalità davvero pericolose (per esempio il supporto alla scrittura per certi filesystem) dovrebbero essere dichiarate chiaramente come tali nella stringa di titolo:

```
config ADFS_FS_RW
      bool "ADFS write support (DANGEROUS)"
      depends on ADFS_FS
      ...
```

Per la documentazione completa sui file di configurazione, consultate il documento Documentation/kbuild/kconfig-language.rst

11) Strutture dati

Le strutture dati che hanno una visibilità superiore al contesto del singolo thread in cui vengono create e distrutte, dovrebbero sempre avere un contatore di riferimenti. Nel kernel non esiste un *garbage collector* (e fuori dal kernel i *garbage collector* sono lenti e inefficienti), questo significa che **dovete** assolutamente avere un contatore di riferimenti per ogni cosa che usate.

Avere un contatore di riferimenti significa che potete evitare la sincronizzazione e permette a più utenti di accedere alla struttura dati in parallelo - e non doversi preoccupare di una struttura dati che improvvisamente sparisce dalla loro vista perché il loro processo dormiva o stava facendo altro per un attimo.

Da notare che la sincronizzazione **non** si sostituisce al conteggio dei riferimenti. La sincronizzazione ha lo scopo di mantenere le strutture dati coerenti, mentre il conteggio dei riferimenti è una tecnica di gestione della memoria. Solitamente servono entrambe le cose, e non vanno confuse fra di loro.

Quando si hanno diverse classi di utenti, le strutture dati possono avere due livelli di contatori di riferimenti. Il contatore di classe conta il numero dei suoi utenti, e il

contatore globale viene decrementato una sola volta quando il contatore di classe va a zero.

Un esempio di questo tipo di conteggio dei riferimenti multi-livello può essere trovato nella gestore della memoria (`struct mm_sturct`: mm_user e mm_count), e nel codice dei filesystem (`struct super_block`: s_count e s_active).

Ricordatevi: se un altro thread può trovare la vostra struttura dati, e non avete un contatore di riferimenti per essa, quasi certamente avete un baco.

12) Macro, enumerati e RTL

I nomi delle macro che definiscono delle costanti e le etichette degli enumerati sono scritte in maiuscolo.

```
#define CONSTANT 0x12345
```

Gli enumerati sono da preferire quando si definiscono molte costanti correlate.

I nomi delle macro in MAIUSCOLO sono preferibili ma le macro che assomigliano a delle funzioni possono essere scritte in minuscolo.

Generalmente, le funzioni inline sono preferibili rispetto alle macro che sembrano funzioni.

Le macro che contengono più istruzioni dovrebbero essere sempre chiuse in un blocco do - while:

```
#define macrofun(a, b, c)                       \
        do {                                    \
                if (a == 5)                     \
                        do_this(b, c);          \
        } while (0)
```

Cose da evitare quando si usano le macro:

1) le macro che hanno effetti sul flusso del codice:

```
#define FOO(x)                                  \
        do {                                    \
                if (blah(x) < 0)                \
                        return -EBUGGERED;      \
        } while (0)
```

sono **proprio** una pessima idea. Sembra una chiamata a funzione ma termina la funzione chiamante; non cercate di rompere il decodificatore interno di chi legge il codice.

2) le macro che dipendono dall' uso di una variabile locale con un nome magico:

```
#define FOO(val) bar(index, val)
```

potrebbe sembrare una bella cosa, ma è dannatamente confusionario quando uno legge il codice e potrebbe romperlo con una cambiamento che sembra innocente.

3) le macro con argomenti che sono utilizzati come l-values; questo potrebbe ritorcervisi contro se qualcuno, per esempio, trasforma FOO in una funzione inline.

4) dimenticatevi delle precedenze: le macro che definiscono espressioni devono essere racchiuse fra parentesi. State attenti a problemi simili con le macro parametrizzate.

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

5) collisione nello spazio dei nomi quando si definisce una variabile locale in una macro che sembra una funzione:

```
#define FOO(x)                          \
({                                      \
        typeof(x) ret;                  \
        ret = calc_ret(x);              \
        (ret);                          \
})
```

ret è un nome comune per una variabile locale - __foo_ret difficilmente andrà in conflitto con una variabile già esistente.

Il manuale di cpp si occupa esaustivamente delle macro. Il manuale di sviluppo di gcc copre anche l'RTL che viene usato frequentemente nel kernel per il linguaggio assembler.

13) Visualizzare i messaggi del kernel

Agli sviluppatori del kernel piace essere visti come dotti. Tenete un occhio di riguardo per l'ortografia e farete una belle figura. In inglese, evitate l'uso incorretto di abbreviazioni come `dont`: usate `do not` oppure `don't`. Scrivete messaggi concisi, chiari, e inequivocabili.

I messaggi del kernel non devono terminare con un punto fermo.

Scrivere i numeri fra parentesi (%d) non migliora alcunché e per questo dovrebbero essere evitati.

Ci sono alcune macro per la diagnostica in <linux/device.h> che dovreste usare per assicurarvi che i messaggi vengano associati correttamente ai dispositivi e ai driver, e che siano etichettati correttamente: dev_err(), dev_warn(), dev_info(), e così via. Per messaggi che non sono associati ad alcun dispositivo, <linux/printk.h> definisce pr_info(), pr_warn(), pr_err(), eccetera.

Tirar fuori un buon messaggio di debug può essere una vera sfida; e quando l'avete può essere d'enorme aiuto per risolvere problemi da remoto. Tuttavia, i messaggi di debug sono gestiti differentemente rispetto agli altri. Le funzioni pr_XXX() stampano incondizionatamente ma pr_debug() no; essa non viene compilata nella configurazione predefinita, a meno che DEBUG o CONFIG_DYNAMIC_DEBUG non vengono impostati. Questo vale anche per dev_dbg() e in aggiunta VERBOSE_DEBUG per aggiungere i messaggi dev_vdbg().

Molti sottosistemi hanno delle opzioni di debug in Kconfig che aggiungono -DDEBUG nei corrispettivi Makefile, e in altri casi aggiungono #define DEBUG in specifici file. Infine, quando un messaggio di debug dev' essere stampato incondizionatamente, per esempio perché siete già in una sezione di debug racchiusa in #ifdef, potete usare printk(KERN_DEBUG ⋯).

14) Assegnare memoria

Il kernel fornisce i seguenti assegnatori ad uso generico: kmalloc(), kzalloc(), kmalloc_array(), kcalloc(), vmalloc(), e vzalloc(). Per maggiori informazioni, consultate la documentazione dell' API: Documentation/translations/it_IT/core-api/memory-allocation.rst

Il modo preferito per passare la dimensione di una struttura è il seguente:

```
p = kmalloc(sizeof(*p), ...);
```

La forma alternativa, dove il nome della struttura viene scritto interamente, peggiora la leggibilità e introduce possibili bachi quando il tipo di puntatore cambia tipo ma il corrispondente sizeof non viene aggiornato.

Il valore di ritorno è un puntatore void, effettuare un cast su di esso è ridondante. La conversione fra un puntatore void e un qualsiasi altro tipo di puntatore è garantito dal linguaggio di programmazione C.

Il modo preferito per assegnare un vettore è il seguente:

```
p = kmalloc_array(n, sizeof(...), ...);
```

Il modo preferito per assegnare un vettore a zero è il seguente:

```
p = kcalloc(n, sizeof(...), ...);
```

Entrambe verificano la condizione di overflow per la dimensione d' assegnamento n * sizeof(⋯), se accade ritorneranno NULL.

Questi allocatori generici producono uno *stack dump* in caso di fallimento a meno che non venga esplicitamente specificato __GFP_NOWARN. Quindi, nella maggior parte dei casi, è inutile stampare messaggi aggiuntivi quando uno di questi allocatori ritornano un puntatore NULL.

15) Il morbo inline

Sembra che ci sia la percezione errata che gcc abbia una qualche magica opzione "rendimi più veloce" chiamata `inline`. In alcuni casi l' uso di inline è appropriato (per esempio in sostituzione delle macro, vedi capitolo 12), ma molto spesso non lo è. L' uso abbondante della parola chiave inline porta ad avere un kernel più grande, che si traduce in un sistema nel suo complesso più lento per via di una cache per le istruzioni della CPU più grande e poi semplicemente perché ci sarà meno spazio disponibile per una pagina di cache. Pensateci un attimo; una fallimento nella cache causa una ricerca su disco che può tranquillamente richiedere

5 millisecondi. Ci sono TANTI cicli di CPU che potrebbero essere usati in questi 5 millisecondi.

Spesso le persone dicono che aggiungere inline a delle funzioni dichiarate static e utilizzare una sola volta è sempre una scelta vincente perché non ci sono altri compromessi. Questo è tecnicamente vero ma gcc è in grado di trasformare automaticamente queste funzioni in inline; i problemi di manutenzione del codice per rimuovere gli inline quando compare un secondo utente surclassano il potenziale vantaggio nel suggerire a gcc di fare una cosa che avrebbe fatto comunque.

16) Nomi e valori di ritorno delle funzioni

Le funzioni possono ritornare diversi tipi di valori, e uno dei più comuni è quel valore che indica se una funzione ha completato con successo o meno. Questo valore può essere rappresentato come un codice di errore intero (-Exxx = fallimento, 0 = successo) oppure un booleano di successo (0 = fallimento, non-zero = successo).

Mischiare questi due tipi di rappresentazioni è un terreno fertile per i bachi più insidiosi. Se il linguaggio C includesse una forte distinzione fra gli interi e i booleani, allora il compilatore potrebbe trovare questi errori per conto nostro ⋯ma questo non c'è. Per evitare di imbattersi in questo tipo di baco, seguite sempre la seguente convenzione:

```
Se il nome di una funzione è un'azione o un comando imperativo,
essa dovrebbe ritornare un codice di errore intero.  Se il nome
è un predicato, la funzione dovrebbe ritornare un booleano di
"successo"
```

Per esempio, `add work` è un comando, e la funzione add_work() ritorna 0 in caso di successo o -EBUSY in caso di fallimento. Allo stesso modo, `PCI device present` è un predicato, e la funzione pci_dev_present() ritorna 1 se trova il dispositivo corrispondente con successo, altrimenti 0.

Tutte le funzioni esportate (EXPORT) devono rispettare questa convenzione, e così dovrebbero anche tutte le funzioni pubbliche. Le funzioni private (static) possono non seguire questa convenzione, ma è comunque raccomandato che lo facciano.

Le funzioni il cui valore di ritorno è il risultato di una computazione, piuttosto che l'indicazione sul successo di tale computazione, non sono soggette a questa regola. Solitamente si indicano gli errori ritornando un qualche valore fuori dai limiti. Un tipico esempio è quello delle funzioni che ritornano un puntatore; queste utilizzano NULL o ERR_PTR come meccanismo di notifica degli errori.

17) L’uso di bool

Nel kernel Linux il tipo bool deriva dal tipo _Bool dello standard C99. Un valore bool può assumere solo i valori 0 o 1, e implicitamente o esplicitamente la conversione a bool converte i valori in vero (*true*) o falso (*false*). Quando si usa un tipo bool il costrutto !! non sarà più necessario, e questo va ad eliminare una certa serie di bachi.

Quando si usano i valori booleani, dovreste utilizzare le definizioni di true e false al posto dei valori 1 e 0.

Per il valore di ritorno delle funzioni e per le variabili sullo stack, l’uso del tipo bool è sempre appropriato. L’uso di bool viene incoraggiato per migliorare la leggibilità e spesso è molto meglio di ‘int’ nella gestione di valori booleani.

Non usate bool se per voi sono importanti l’ordine delle righe di cache o la loro dimensione; la dimensione e l’allineamento cambia a seconda dell’architettura per la quale è stato compilato. Le strutture che sono state ottimizzate per l’allineamento o la dimensione non dovrebbero usare bool.

Se una struttura ha molti valori true/false, considerate l’idea di raggrupparli in un intero usando campi da 1 bit, oppure usate un tipo dalla larghezza fissa, come u8.

Come per gli argomenti delle funzioni, molti valori true/false possono essere raggruppati in un singolo argomento a bit denominato ‘flags’; spesso ‘flags’ è un’ alternativa molto più leggibile se si hanno valori costanti per true/false.

Detto ciò, un uso parsimonioso di bool nelle strutture dati e negli argomenti può migliorare la leggibilità.

18) Non reinventate le macro del kernel

Il file di intestazione include/linux/kernel.h contiene un certo numero di macro che dovreste usare piuttosto che implementarne una qualche variante. Per esempio, se dovete calcolare la lunghezza di un vettore, sfruttate la macro:

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Analogamente, se dovete calcolare la dimensione di un qualche campo di una struttura, usate

```
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
```

Ci sono anche le macro min() e max() che, se vi serve, effettuano un controllo rigido sui tipi. Sentitevi liberi di leggere attentamente questo file d’intestazione per scoprire cos’altro è stato definito che non dovreste reinventare nel vostro codice.

19) Linee di configurazione degli editor e altre schifezze

Alcuni editor possono interpretare dei parametri di configurazione integrati nei file sorgenti e indicati con dai marcatori speciali. Per esempio, emacs interpreta le linee marcate nel seguente modo:

```
-*- mode: c -*-
```

O come queste:

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim interpreta i marcatori come questi:

```
/* vim:set sw=8 noet */
```

Non includete nessuna di queste cose nei file sorgenti. Le persone hanno le proprie configurazioni personali per l'editor, e i vostri sorgenti non dovrebbero sovrascrivergliele. Questo vale anche per i marcatori d'indentazione e di modalità d'uso. Le persone potrebbero aver configurato una modalità su misura, oppure potrebbero avere qualche altra magia per far funzionare bene l'indentazione.

20) Inline assembly

Nel codice specifico per un'architettura, potreste aver bisogno di codice *inline assembly* per interfacciarvi col processore o con una funzionalità specifica della piattaforma. Non esitate a farlo quando è necessario. Comunque, non usatele gratuitamente quando il C può fare la stessa cosa. Potete e dovreste punzecchiare l'hardware in C quando è possibile.

Considerate la scrittura di una semplice funzione che racchiude pezzi comuni di codice assembler piuttosto che continuare a riscrivere delle piccole varianti. Ricordatevi che l'*inline assembly* può utilizzare i parametri C.

Il codice assembler più corposo e non banale dovrebbe andare nei file .S, coi rispettivi prototipi C definiti nei file d'intestazione. I prototipi C per le funzioni assembler dovrebbero usare `asmlinkage`.

Potreste aver bisogno di marcare il vostro codice asm come volatile al fine d'evitare che GCC lo rimuova quando pensa che non ci siano effetti collaterali. Non c'è sempre bisogno di farlo, e farlo quando non serve limita le ottimizzazioni.

Quando scrivete una singola espressione *inline assembly* contenente più istruzioni, mettete ognuna di queste istruzioni in una stringa e riga diversa; ad eccezione dell'ultima stringa/istruzione, ognuna deve terminare con `\n\t` al fine di allineare correttamente l'assembler che verrà generato:

```
asm ("magic %reg1, #42\n\t"
     "more_magic %reg2, %reg3"
     : /* outputs */ : /* inputs */ : /* clobbers */);
```

21) Compilazione sotto condizione

Ovunque sia possibile, non usate le direttive condizionali del preprocessore (#if, #ifdef) nei file .c; farlo rende il codice difficile da leggere e da seguire. Invece, usate queste direttive nei file d' intestazione per definire le funzioni usate nei file .c, fornendo i relativi stub nel caso #else, e quindi chiamate queste funzioni senza condizioni di preprocessore. Il compilatore non produrrà alcun codice per le funzioni stub, produrrà gli stessi risultati, e la logica rimarrà semplice da seguire.

È preferibile non compilare intere funzioni piuttosto che porzioni d'esse o porzioni d'espressioni. Piuttosto che mettere una ifdef in un'espressione, fattorizzate parte dell' espressione, o interamente, in funzioni e applicate la direttiva condizionale su di esse.

Se avete una variabile o funzione che potrebbe non essere usata in alcune configurazioni, e quindi il compilatore potrebbe avvisarvi circa la definizione inutilizzata, marcate questa definizione come __maybe_unused piuttosto che racchiuderla in una direttiva condizionale del preprocessore. (Comunque, se una variabile o funzione è *sempre* inutilizzata, rimuovetela).

Nel codice, dov' è possibile, usate la macro IS_ENABLED per convertire i simboli Kconfig in espressioni booleane C, e quindi usatela nelle classiche condizioni C:

```
if (IS_ENABLED(CONFIG_SOMETHING)) {
        ...
}
```

Il compilatore valuterà la condizione come costante (constant-fold), e quindi includerà o escluderà il blocco di codice come se fosse in un #ifdef, quindi non ne aumenterà il tempo di esecuzione. Tuttavia, questo permette al compilatore C di vedere il codice nel blocco condizionale e verificarne la correttezza (sintassi, tipi, riferimenti ai simboli, eccetera). Quindi dovete comunque utilizzare #ifdef se il codice nel blocco condizionale esiste solo quando la condizione è soddisfatta.

Alla fine di un blocco corposo di #if o #ifdef (più di alcune linee), mettete un commento sulla stessa riga di #endif, annotando la condizione che termina. Per esempio:

```
#ifdef CONFIG_SOMETHING
...
#endif /* CONFIG_SOMETHING */
```

Appendice I) riferimenti

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).

The Practice of Programming by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.

Manuali GNU - nei casi in cui sono compatibili con K&R e questo documento - per indent, cpp, gcc e i suoi dettagli interni, tutto disponibile qui http://www.gnu.org/manual/

WG14 è il gruppo internazionale di standardizzazione per il linguaggio C, URL: http://www.open-std.org/JTC1/SC22/WG14/

Kernel process/coding-style.rst, by greg@kroah.com at OLS 2002: http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/maintainer-pgp-guide.rst

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

La guida a PGP per manutentori del kernel

Author
Konstantin Ryabitsev <konstantin@linuxfoundation.org>

Questo documento è destinato agli sviluppatori del kernel Linux, in particolar modo ai manutentori. Contiene degli approfondimenti riguardo informazioni che sono state affrontate in maniera più generale nella sezione "Protecting Code Integrity" pubblicata dalla Linux Foundation. Per approfondire alcuni argomenti trattati in questo documento è consigliato leggere il documento sopraindicato

Il ruolo di PGP nello sviluppo del kernel Linux

PGP aiuta ad assicurare l'integrità del codice prodotto dalla comunità di sviluppo del kernel e, in secondo luogo, stabilisce canali di comunicazione affidabili tra sviluppatori attraverso lo scambio di email firmate con PGP.

Il codice sorgente del kernel Linux è disponibile principalmente in due formati:

- repositori distribuiti di sorgenti (git)
- rilasci periodici di istantanee (archivi tar)

Sia i repositori git che gli archivi tar portano le firme PGP degli sviluppatori che hanno creato i rilasci ufficiali del kernel. Queste firme offrono una garanzia crittografica che le versioni scaricabili rese disponibili via kernel.org, o altri portali, siano identiche a quelle che gli sviluppatori hanno sul loro posto di lavoro. A tal scopo:

- i repositori git forniscono firme PGP per ogni tag
- gli archivi tar hanno firme separate per ogni archivio

Fidatevi degli sviluppatori e non dell'infrastruttura

Fin dal 2011, quando i sistemi di kernel.org furono compromessi, il principio generale del progetto Kernel Archives è stato quello di assumere che qualsiasi parte dell'infrastruttura possa essere compromessa in ogni momento. Per questa ragione, gli amministratori hanno intrapreso deliberatemene dei passi per enfatizzare che la fiducia debba risiedere sempre negli sviluppatori e mai nel codice che gestisce l'infrastruttura, indipendentemente da quali che siano le pratiche di sicurezza messe in atto.

Il principio sopra indicato è la ragione per la quale è necessaria questa guida. Vogliamo essere sicuri che il riporre la fiducia negli sviluppatori non sia fatto semplicemente per incolpare qualcun'altro per future falle di sicurezza. L'obiettivo è quello di fornire una serie di linee guida che gli sviluppatori possano seguire per creare un ambiente di lavoro sicuro e salvaguardare le chiavi PGP usate nello stabilire l'integrità del kernel Linux stesso.

Strumenti PGP

Usare GnuPG v2

La vostra distribuzione potrebbe avere già installato GnuPG, dovete solo verificare che stia utilizzando la versione 2.x e non la serie 1.4 – molte distribuzioni forniscono entrambe, di base il comando "gpg" invoca GnuPG v.1. Per controllate usate:

```
$ gpg --version | head -n1
```

Se visualizzate `gpg (GnuPG) 1.4.x`, allora state usando GnuPG v.1. Provate il comando `gpg2` (se non lo avete, potreste aver bisogno di installare il pacchetto gnupg2):

```
$ gpg2 --version | head -n1
```

Se visualizzate `gpg (GnuPG) 2.x.x`, allora siete pronti a partire. Questa guida assume che abbiate la versione 2.2.(o successiva) di GnuPG. Se state usando la versione 2.0, alcuni dei comandi indicati qui non funzioneranno, in questo caso considerate un aggiornamento all'ultima versione, la 2.2. Versioni di gnupg-2.1.11 e successive dovrebbero essere compatibili per gli obiettivi di questa guida.

Se avete entrambi i comandi: `gpg` e `gpg2`, assicuratevi di utilizzare sempre la versione V2, e non quella vecchia. Per evitare errori potreste creare un alias:

```
$ alias gpg=gpg2
```

Potete mettere questa opzione nel vostro `.bashrc` in modo da essere sicuri.

Configurare le opzioni di gpg-agent

L'agente GnuPG è uno strumento di aiuto che partirà automaticamente ogni volta che userete il comando `gpg` e funzionerà in background con l'obiettivo di individuare la passphrase. Ci sono due opzioni che dovreste conoscere per personalizzare la scadenza della passphrase nella cache:

- `default-cache-ttl` (secondi): Se usate ancora la stessa chiave prima che il time-to-live termini, il conto alla rovescia si resetterà per un altro periodo. Di base è di 600 (10 minuti).
- `max-cache-ttl` (secondi): indipendentemente da quanto sia recente l'ultimo uso della chiave da quando avete inserito la passphrase, se il massimo time-to-live è scaduto, dovrete reinserire nuovamente la passphrase. Di base è di 30 minuti.

Se ritenete entrambe questi valori di base troppo corti (o troppo lunghi), potete creare il vostro file `~/.gnupg/gpg-agent.conf` ed impostare i vostri valori:

```
# set to 30 minutes for regular ttl, and 2 hours for max ttl
default-cache-ttl 1800
max-cache-ttl 7200
```

Note: Non è più necessario far partire l'agente gpg manualmente all'inizio della vostra sessione. Dovreste controllare i file rc per rimuovere tutto ciò che riguarda vecchie le versioni di GnuPG, poiché potrebbero non svolgere più bene il loro compito.

Impostare un *refresh* con cronjob

Potreste aver bisogno di rinfrescare regolarmente il vostro portachiavi in modo aggiornare le chiavi pubbliche di altre persone, lavoro che è svolto al meglio con un cronjob giornaliero:

```
@daily /usr/bin/gpg2 --refresh >/dev/null 2>&1
```

Controllate il percorso assoluto del vostro comando `gpg` o `gpg2` e usate il comando `gpg2` se per voi `gpg` corrisponde alla versione GnuPG v.1.

Proteggere la vostra chiave PGP primaria

Questa guida parte dal presupposto che abbiate già una chiave PGP che usate per lo sviluppo del kernel Linux. Se non ne avete ancora una, date uno sguardo al documento "Protecting Code Integrity" che abbiamo menzionato prima.

Dovreste inoltre creare una nuova chiave se quella attuale è inferiore a 2048 bit (RSA).

Chiave principale o sottochiavi

Le sottochiavi sono chiavi PGP totalmente indipendenti, e sono collegate alla chiave principale attraverso firme certificate. È quindi importante comprendere i seguenti punti:

1. Non ci sono differenze tecniche tra la chiave principale e la sottochiave.
2. In fesa di creazione, assegniamo limitazioni funzionali ad ogni chiave assegnando capacità specifiche.
3. Una chiave PGP può avere 4 capacità:
   - **[S]** può essere usata per firmare
   - **[E]** può essere usata per criptare
   - **[A]** può essere usata per autenticare
   - **[C]** può essere usata per certificare altre chiavi
4. Una singola chiave può avere più capacità
5. Una sottochiave è completamente indipendente dalla chiave principale. Un messaggio criptato con la sottochiave non può essere decrittato con quella principale. Se perdete la vostra sottochiave privata, non può essere rigenerata in nessun modo da quella principale.

La chiave con capacità **[C]** (certify) è identificata come la chiave principale perché è l'unica che può essere usata per indicare la relazione con altre chiavi. Solo la chiave **[C]** può essere usata per:

- Aggiungere o revocare altre chiavi (sottochiavi) che hanno capacità S/E/A
- Aggiungere, modificare o eliminare le identità (unids) associate alla chiave
- Aggiungere o modificare la data di termine di sé stessa o di ogni sottochiave
- Firmare le chiavi di altre persone a scopo di creare una rete di fiducia

Di base, alla creazione di nuove chiavi, GnuPG genera quanto segue:

- Una chiave madre che porta sia la capacità di certificazione che quella di firma (**[SC]**)
- Una sottochiave separata con capacità di criptaggio (**[E]**)

Se avete usato i parametri di base per generare la vostra chiave, quello sarà il risultato. Potete verificarlo utilizzando `gpg --list-secret-keys`, per esempio:

```
sec   rsa2048 2018-01-23 [SC] [expires: 2020-01-23]
      0000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb   rsa2048 2018-01-23 [E] [expires: 2020-01-23]
```

Qualsiasi chiave che abbia la capacità **[C]** è la vostra chiave madre, indipendentemente da quali altre capacità potreste averle assegnato.

La lunga riga sotto la voce `sec` è la vostra impronta digitale – negli esempi che seguono, quando vedere `[fpr]` ci si riferisce a questa stringa di 40 caratteri.

Assicuratevi che la vostra passphrase sia forte

GnuPG utilizza le passphrases per criptare la vostra chiave privata prima di salvarla sul disco. In questo modo, anche se il contenuto della vostra cartella `.gnupg` venisse letto o trafugato nella sia interezza, gli attaccanti non potrebbero comunque utilizzare le vostre chiavi private senza aver prima ottenuto la passphrase per decriptarle.

È assolutamente essenziale che le vostre chiavi private siano protette da una passphrase forte. Per impostarla o cambiarla, usate:

```
$ gpg --change-passphrase [fpr]
```

Create una sottochiave di firma separata

Il nostro obiettivo è di proteggere la chiave primaria spostandola su un dispositivo sconnesso dalla rete, dunque se avete solo una chiave combinata **[SC]** allora dovreste creare una sottochiave di firma separata:

```
$ gpg --quick-add-key [fpr] ed25519 sign
```

Ricordate di informare il keyserver del vostro cambiamento, cosicché altri possano ricevere la vostra nuova sottochiave:

```
$ gpg --send-key [fpr]
```

Note: Supporto ECC in GnuPG GnuPG 2.1 e successivi supportano pienamente *Elliptic Curve Cryptography*, con la possibilità di combinare sottochiavi ECC con le tradizionali chiavi primarie RSA. Il principale vantaggio della crittografia ECC è che è molto più veloce da calcolare e crea firme più piccole se confrontate byte per byte con le chiavi RSA a più di 2048 bit. A meno che non pensiate di utilizzare un dispositivo smartcard che non supporta le operazioni ECC, vi raccomandiamo ti creare sottochiavi di firma ECC per il vostro lavoro col kernel.

Se per qualche ragione preferite rimanere con sottochiavi RSA, nel comando precedente, sostituite “ed25519” con “rsa2048”. In aggiunta, se avete intenzione di usare un dispositivo hardware che non supporta le chiavi ED25519 ECC,

come la Nitrokey Pro o la Yubikey, allora dovreste usare “nistp256” al posto di “ed25519”.

Copia di riserva della chiave primaria per gestire il recupero da disastro

Maggiori sono le firme di altri sviluppatori che vengono applicate alla vostra, maggiori saranno i motivi per avere una copia di riserva che non sia digitale, al fine di effettuare un recupero da disastro.

Il modo migliore per creare una copia fisica della vostra chiave privata è l’uso del programma `paperkey`. Consultate `man paperkey` per maggiori dettagli sul formato dell’output ed i suoi punti di forza rispetto ad altre soluzioni. Paperkey dovrebbe essere già pacchettizzato per la maggior parte delle distribuzioni.

Eseguite il seguente comando per creare una copia fisica di riserva della vostra chiave privata:

```
$ gpg --export-secret-key [fpr] | paperkey -o /tmp/key-backup.txt
```

Stampate il file (o fate un pipe direttamente verso lpr), poi prendete una penna e scrivete la passphare sul margine del foglio. **Questo è caldamente consigliato** perché la copia cartacea è comunque criptata con la passphrase, e se mai doveste cambiarla non vi ricorderete qual’era al momento della creazione di quella copia – *garantito*.

Mettete la copia cartacea e la passphrase scritta a mano in una busta e mettetela in un posto sicuro e ben protetto, preferibilmente fuori casa, magari in una cassetta di sicurezza in banca.

Note: Probabilmente la vostra stampante non è più quello stupido dispositivo connesso alla porta parallela, ma dato che il suo output è comunque criptato con la passphrase, eseguire la stampa in un sistema “cloud” moderno dovrebbe essere comunque relativamente sicuro. Un’opzione potrebbe essere quella di cambiare la passphrase della vostra chiave primaria subito dopo aver finito con paperkey.

Copia di riserva di tutta la cartella GnuPG

Warning: !!!Non saltate questo passo!!!

Quando avete bisogno di recuperare le vostre chiavi PGP è importante avere una copia di riserva pronta all’uso. Questo sta su un diverso piano di prontezza rispetto al recupero da disastro che abbiamo risolto con `paperkey`. Vi affiderete a queste copie esterne quando dovreste usare la vostra chiave Certify – ovvero quando fate modifiche alle vostre chiavi o firmate le chiavi di altre persone ad una conferenza o ad un gruppo d’incontro.

Incominciate con una piccola chiavetta di memoria USB (preferibilmente due) che userete per le copie di riserva. Dovrete criptarle usando LUKS – fate riferimento alla documentazione della vostra distribuzione per capire come fare.

Per la passphrase di criptazione, potete usare la stessa della vostra chiave primaria.

Una volta che il processo di criptazione è finito, reinserite il disco USB ed assicurativi che venga montato correttamente. Copiate interamente la cartella `.gnugp` nel disco criptato:

```
$ cp -a ~/.gnupg /media/disk/foo/gnupg-backup
```

Ora dovreste verificare che tutto continui a funzionare:

```
$ gpg --homedir=/media/disk/foo/gnupg-backup --list-key [fpr]
```

Se non vedete errori, allora dovreste avere fatto tutto con successo. Smontate il disco USB, etichettatelo per bene di modo da evitare di distruggerne il contenuto non appena vi serve una chiavetta USB a caso, ed infine mettetelo in un posto sicuro – ma non troppo lontano, perché vi servirà di tanto in tanto per modificare le identità, aggiungere o revocare sottochiavi, o firmare le chiavi di altre persone.

Togliete la chiave primaria dalla vostra home

I file che si trovano nella vostra cartella home non sono poi così ben protetti come potreste pensare. Potrebbero essere letti o trafugati in diversi modi:

- accidentalmente quando fate una rapida copia della cartella home per configurare una nuova postazione
- da un amministratore di sistema negligente o malintenzionato
- attraverso copie di riserva insicure
- attraverso malware installato in alcune applicazioni (browser, lettori PDF, eccetera)
- attraverso coercizione quando attraversate confini internazionali

Proteggere la vostra chiave con una buona passphare aiuta notevolmente a ridurre i rischi elencati qui sopra, ma le passphrase possono essere scoperte attraverso i keylogger, il shoulder-surfing, o altri modi. Per questi motivi, nella configurazione si raccomanda di rimuove la chiave primaria dalla vostra cartella home e la si archivia su un dispositivo disconnesso.

Warning: Per favore, fate riferimento alla sezione precedente e assicuratevi di aver fatto una copia di riserva totale della cartella GnuPG. Quello che stiamo per fare renderà la vostra chiave inutile se non avete delle copie di riserva utilizzabili!

Per prima cosa, identificate il keygrip della vostra chiave primaria:

```
$ gpg --with-keygrip --list-key [fpr]
```

L'output assomiglierà a questo:

```
pub   rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
      Keygrip = 1111000000000000000000000000000000000000
uid           [ultimate] Alice Dev <adev@kernel.org>
sub   rsa2048 2018-01-24 [E] [expires: 2020-01-24]
      Keygrip = 2222000000000000000000000000000000000000
sub   ed25519 2018-01-24 [S]
      Keygrip = 3333000000000000000000000000000000000000
```

Trovate la voce keygrid che si trova sotto alla riga pub (appena sotto all'impronta digitale della chiave primaria). Questo corrisponderà direttamente ad un file nella cartella ~/.gnupg:

```
$ cd ~/.gnupg/private-keys-v1.d
$ ls
1111000000000000000000000000000000000000.key
2222000000000000000000000000000000000000.key
3333000000000000000000000000000000000000.key
```

Quello che dovrete fare è rimuovere il file .key che corrisponde al keygrip della chiave primaria:

```
$ cd ~/.gnupg/private-keys-v1.d
$ rm 1111000000000000000000000000000000000000.key
```

Ora, se eseguite il comando --list-secret-keys, vedrete che la chiave primaria non compare più (il simbolo # indica che non è disponibile):

```
$ gpg --list-secret-keys
sec#  rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb   rsa2048 2018-01-24 [E] [expires: 2020-01-24]
ssb   ed25519 2018-01-24 [S]
```

Dovreste rimuovere anche i file secring.gpg che si trovano nella cartella ~/.gnupg, in quanto rimasugli delle versioni precedenti di GnuPG.

Se non avete la cartella “private-keys-v1.d”

Se non avete la cartella `~/.gnupg/private-keys-v1.d`, allora le vostre chiavi segrete sono ancora salvate nel vecchio file `secring.gpg` usato da GnuPG v1. Effettuare una qualsiasi modifica alla vostra chiave, come cambiare la passphare o aggiungere una sottochiave, dovrebbe convertire automaticamente il vecchio formato `secring.gpg``nel nuovo ``private-keys-v1.d`.

Una volta che l’avete fatto, assicuratevi di rimuovere il file `secring.gpg`, che continua a contenere la vostra chiave privata.

Spostare le sottochiavi in un apposito dispositivo criptato

Nonostante la chiave primaria sia ora al riparo da occhi e mani indiscrete, le sottochiavi si trovano ancora nella vostra cartella home. Chiunque riesca a mettere le sue mani su quelle chiavi riuscirà a decriptare le vostre comunicazioni o a falsificare le vostre firme (se conoscono la passphrase). Inoltre, ogni volta che viene fatta un’operazione con GnuPG, le chiavi vengono caricate nella memoria di sistema e potrebbero essere rubate con l’uso di malware sofisticati (pensate a Meltdown e a Spectre).

Il miglior modo per proteggere le proprie chiave è di spostarle su un dispositivo specializzato in grado di effettuare operazioni smartcard.

I benefici di una smartcard

Una smartcard contiene un chip crittografico che è capace di immagazzinare le chiavi private ed effettuare operazioni crittografiche direttamente sulla carta stessa. Dato che la chiave non lascia mai la smartcard, il sistema operativo usato sul computer non sarà in grado di accedere alle chiavi. Questo è molto diverso dai dischi USB criptati che abbiamo usato allo scopo di avere una copia di riserva sicura – quando il dispositivo USB è connesso e montato, il sistema operativo potrà accedere al contenuto delle chiavi private.

L’uso di un disco USB criptato non può sostituire le funzioni di un dispositivo capace di operazioni di tipo smartcard.

Dispositivi smartcard disponibili

A meno che tutti i vostri computer dispongano di lettori smartcard, il modo più semplice è equipaggiarsi di un dispositivo USB specializzato che implementi le funzionalità delle smartcard. Sul mercato ci sono diverse soluzioni disponibili:

- Nitrokey Start: è Open hardware e Free Software, è basata sul progetto GnuK della FSIJ. Questo è uno dei pochi dispositivi a supportare le chiavi ECC ED25519, ma offre meno funzionalità di sicurezza (come la resistenza alla manomissione o alcuni attacchi ad un canale laterale).
- Nitrokey Pro 2: è simile alla Nitrokey Start, ma è più resistente alla manomissione e offre più funzionalità di sicurezza. La Pro 2 supporta la crittografia ECC (NISTP).

- Yubikey 5: l'hardware e il software sono proprietari, ma è più economica della Nitrokey Pro ed è venduta anche con porta USB-C il che è utile con i computer portatili più recenti. In aggiunta, offre altre funzionalità di sicurezza come FIDO, U2F, e ora supporta anche le chiavi ECC (NISTP)

Su LWN c'è una buona recensione dei modelli elencati qui sopra e altri. La scelta dipenderà dal costo, dalla disponibilità nella vostra area geografica e vostre considerazioni sull' hardware aperto/proprietario.

Se volete usare chiavi ECC, la vostra migliore scelta sul mercato è la Nitrokey Start.

Configurare il vostro dispositivo smartcard

Il vostro dispositivo smartcard dovrebbe iniziare a funzionare non appena lo collegate ad un qualsiasi computer Linux moderno. Potete verificarlo eseguendo:

```
$ gpg --card-status
```

Se vedete tutti i dettagli della smartcard, allora ci siamo. Sfortunatamente, affrontare tutti i possibili motivi per cui le cose potrebbero non funzionare non è lo scopo di questa guida. Se avete problemi nel far funzionare la carta con GnuPG, cercate aiuto attraverso i soliti canali di supporto.

Per configurare la vostra smartcard, dato che non c' è una via facile dalla riga di comando, dovrete usate il menu di GnuPG:

```
$ gpg --card-edit
[...omitted...]
gpg/card> admin
Admin commands are allowed
gpg/card> passwd
```

Dovreste impostare il PIN dell'utente (1), quello dell'amministratore (3) e il codice di reset (4). Assicuratevi di annotare e salvare questi codici in un posto sicuro – specialmente il PIN dell' amministratore e il codice di reset (che vi permetterà di azzerare completamente la smartcard). Il PIN dell' amministratore viene usato così raramente che è inevitabile dimenticarselo se non lo si annota.

Tornando al nostro menu, potete impostare anche altri valori (come il nome, il sesso, informazioni d' accesso, eccetera), ma non sono necessari e aggiunge altre informazioni sulla carta che potrebbero trapelare in caso di smarrimento.

Note: A dispetto del nome "PIN", né il PIN utente né quello dell'amministratore devono essere esclusivamente numerici.

Spostare le sottochiavi sulla smartcard

Uscite dal menu (usando “q”) e salverete tutte le modifiche. Poi, spostiamo tutte le sottochiavi sulla smartcard. Per la maggior parte delle operazioni vi serviranno sia la passphrase della chiave PGP che il PIN dell’ amministratore:

```
$ gpg --edit-key [fpr]

Secret subkeys are available.

pub  rsa2048/AAAABBBBCCCCDDDD
     created: 2018-01-23  expires: 2020-01-23  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa2048/1111222233334444
     created: 2018-01-23  expires: never       usage: E
ssb  ed25519/5555666677778888
     created: 2017-12-07  expires: never       usage: S
[ultimate] (1). Alice Dev <adev@kernel.org>

gpg>
```

Usando `--edit-key` si tornerà alla modalità menu e noterete che la lista delle chiavi è leggermente diversa. Da questo momento in poi, tutti i comandi saranno eseguiti nella modalità menu, come indicato da `gpg>`.

Per prima cosa, selezioniamo la chiave che verrà messa sulla carta – potete farlo digitando `key 1` (è la prima della lista, la sottochiave **[E]**):

```
gpg> key 1
```

Nel’ output dovreste vedere `ssb*` associato alla chiave **[E]**. Il simbolo * indica che la chiave è stata “selezionata” . Funziona come un interruttore, ovvero se scrivete nuovamente `key 1`, il simbolo * sparirà e la chiave non sarà più selezionata.

Ora, spostiamo la chiave sulla smartcard:

```
gpg> keytocard
Please select where to store the key:
   (2) Encryption key
Your selection? 2
```

Dato che è la nostra chiave **[E]**, ha senso metterla nella sezione criptata. Quando confermerete la selezione, vi verrà chiesta la passphrase della vostra chiave PGP, e poi il PIN dell’ amministratore. Se il comando ritorna senza errori, allora la vostra chiave è stata spostata con successo.

Importante: digitate nuovamente `key 1` per deselezionare la prima chiave e selezionate la seconda chiave **[S]** con `key 2`:

```
gpg> key 1
gpg> key 2
gpg> keytocard
Please select where to store the key:
```

(continues on next page)

(continued from previous page)

```
   (1) Signature key
   (3) Authentication key
Your selection? 1
```

Potete usare la chiave **[S]** sia per firmare che per autenticare, ma vogliamo che sia nella sezione di firma, quindi scegliete (1). Ancora una volta, se il comando ritorna senza errori, allora l'operazione è avvenuta con successo:

```
gpg> q
Save changes? (y/N) y
```

Salvando le modifiche cancellerete dalla vostra cartella home tutte le chiavi che avete spostato sulla carta (ma questo non è un problema, perché abbiamo fatto delle copie di sicurezza nel caso in cui dovessimo configurare una nuova smartcard).

Verificare che le chiavi siano state spostate

Ora, se doveste usare l'opzione `--list-secret-keys`, vedrete una sottile differenza nell'output:

```
$ gpg --list-secret-keys
sec#  rsa2048 2018-01-24 [SC] [expires: 2020-01-24]
      0000000000000000000000000AAAABBBBCCCCDDDD
uid           [ultimate] Alice Dev <adev@kernel.org>
ssb>  rsa2048 2018-01-24 [E] [expires: 2020-01-24]
ssb>  ed25519 2018-01-24 [S]
```

Il simbolo > in `ssb>` indica che la sottochiave è disponibile solo nella smartcard. Se tornate nella vostra cartella delle chiavi segrete e guardate al suo contenuto, noterete che i file `.key` sono stati sostituiti con degli stub:

```
$ cd ~/.gnupg/private-keys-v1.d
$ strings *.key | grep 'private-key'
```

Per indicare che i file sono solo degli stub e che in realtà il contenuto è sulla smartcard, l'output dovrebbe mostrarvi `shadowed-private-key`.

Verificare che la smartcard funzioni

Per verificare che la smartcard funzioni come dovuto, potete creare una firma:

```
$ echo "Hello world" | gpg --clearsign > /tmp/test.asc
$ gpg --verify /tmp/test.asc
```

Col primo comando dovrebbe chiedervi il PIN della smartcard, e poi dovrebbe mostrare "Good signature" dopo l'esecuzione di `gpg --verify`.

Complimenti, siete riusciti a rendere estremamente difficile il furto della vostra identità digitale di sviluppatore.

Altre operazioni possibili con GnuPG

Segue un breve accenno ad alcune delle operazioni più comuni che dovrete fare con le vostre chiavi PGP.

Montare il disco con la chiave primaria

Vi servirà la vostra chiave principale per tutte le operazioni che seguiranno, per cui per prima cosa dovrete accedere ai vostri backup e dire a GnuPG di usarli:

```
$ export GNUPGHOME=/media/disk/foo/gnupg-backup
$ gpg --list-secret-keys
```

Dovete assicurarvi di vedere `sec` e non `sec#` nell'output del programma (il simbolo # significa che la chiave non è disponibile e che state ancora utilizzando la vostra solita cartella di lavoro).

Estendere la data di scadenza di una chiave

La chiave principale ha una data di scadenza di 2 anni dal momento della sua creazione. Questo per motivi di sicurezza e per rendere obsolete le chiavi che, eventualmente, dovessero sparire dai keyserver.

Per estendere di un anno, dalla data odierna, la scadenza di una vostra chiave, eseguite:

```
$ gpg --quick-set-expire [fpr] 1y
```

Se per voi è più facile da memorizzare, potete anche utilizzare una data specifica (per esempio, il vostro compleanno o capodanno):

```
$ gpg --quick-set-expire [fpr] 2020-07-01
```

Ricordatevi di inviare l'aggiornamento ai keyserver:

```
$ gpg --send-key [fpr]
```

Aggiornare la vostra cartella di lavoro dopo ogni modifica

Dopo aver fatto delle modifiche alle vostre chiavi usando uno spazio a parte, dovreste importarle nella vostra cartella di lavoro abituale:

```
$ gpg --export | gpg --homedir ~/.gnupg --import
$ unset GNUPGHOME
```

Usare PGP con Git

Una delle caratteristiche fondanti di Git è la sua natura decentralizzata – una volta che il repositorio è stato clonato sul vostro sistema, avete la storia completa del progetto, inclusi i suoi tag, i commit ed i rami. Tuttavia, con i centinaia di repositori clonati che ci sono in giro, come si fa a verificare che la loro copia di linux.git non è stata manomessa da qualcuno?

Oppure, cosa succede se viene scoperta una backdoor nel codice e la riga "Autore" dice che sei stato tu, mentre tu sei abbastanza sicuro di non averci niente a che fare?

Per risolvere entrambi i problemi, Git ha introdotto l'integrazione con PGP. I tag firmati dimostrano che il repositorio è integro assicurando che il suo contenuto è lo stesso che si trova sulle macchine degli sviluppatori che hanno creato il tag; mentre i commit firmati rendono praticamente impossibile ad un malintenzionato di impersonarvi senza avere accesso alle vostre chiavi PGP.

Configurare git per usare la vostra chiave PGP

Se avete solo una chiave segreta nel vostro portachiavi, allora non avete nulla da fare in più dato che sarà la vostra chiave di base. Tuttavia, se doveste avere più chiavi segrete, potete dire a git quale dovrebbe usare (`[fpg]` è la vostra impronta digitale):

```
$ git config --global user.signingKey [fpr]
```

IMPORTANTE: se avete una comando dedicato per `gpg2`, allora dovreste dire a git di usare sempre quello piuttosto che il vecchio comando `gpg`:

```
$ git config --global gpg.program gpg2
```

Come firmare i tag

Per creare un tag firmato, passate l'opzione `-s` al comando tag:

```
$ git tag -s [tagname]
```

La nostra raccomandazione è quella di firmare sempre i tag git, perché questo permette agli altri sviluppatori di verificare che il repositorio git dal quale stanno prendendo il codice non è stato alterato intenzionalmente.

Come verificare i tag firmati

Per verificare un tag firmato, potete usare il comando `verify-tag`:

```
$ git verify-tag [tagname]
```

Se state prendendo un tag da un fork del repositorio del progetto, git dovrebbe verificare automaticamente la firma di quello che state prendendo e vi mostrerà il risultato durante l'operazione di merge:

```
$ git pull [url] tags/sometag
```

Il merge conterrà qualcosa di simile:

```
Merge tag 'sometag' of [url]

[Tag message]

# gpg: Signature made [...]
# gpg: Good signature from [...]
```

Se state verificando il tag di qualcun altro, allora dovrete importare la loro chiave PGP. Fate riferimento alla sezione "*Come verificare l'identità degli sviluppatori del kernel*" che troverete più avanti.

Configurare git per firmare sempre i tag con annotazione

Se state creando un tag con annotazione è molto probabile che vogliate firmarlo. Per imporre a git di firmare sempre un tag con annotazione, dovete impostare la seguente opzione globale:

```
$ git config --global tag.forceSignAnnotated true
```

Come usare commit firmati

Creare dei commit firmati è facile, ma è molto più difficile utilizzarli nello sviluppo del kernel linux per via del fatto che ci si affida alle liste di discussione e questo modo di procedere non mantiene le firme PGP nei commit. In aggiunta, quando si usa *rebase* nel proprio repositorio locale per allinearsi al kernel anche le proprie firme PGP verranno scartate. Per questo motivo, la maggior parte degli sviluppatori del kernel non si preoccupano troppo di firmare i propri commit ed ignoreranno quelli firmati che si trovano in altri repositori usati per il proprio lavoro.

Tuttavia, se avete il vostro repositorio di lavoro disponibile al pubblico su un qualche servizio di hosting git (kernel.org, infradead.org, ozlabs.org, o altri), allora la raccomandazione è di firmare tutti i vostri commit anche se gli sviluppatori non ne beneficeranno direttamente.

Vi raccomandiamo di farlo per i seguenti motivi:

1. Se dovesse mai esserci la necessità di fare delle analisi forensi o tracciare la provenienza di un codice, anche sorgenti mantenuti esternamente che hanno firme PGP sui commit avranno un certo valore a questo scopo.
2. Se dovesse mai capitarvi di clonare il vostro repositorio locale (per esempio dopo un danneggiamento del disco), la firma vi permetterà di verificare l'integrità del repositorio prima di riprendere il lavoro.
3. Se qualcuno volesse usare *cherry-pick* sui vostri commit, allora la firma permetterà di verificare l'integrità dei commit prima di applicarli.

Creare commit firmati

Per creare un commit firmato, dovete solamente aggiungere l'opzione `-S` al comando `git commit` (si usa la lettera maiuscola per evitare conflitti con un'altra opzione):

```
$ git commit -S
```

Configurare git per firmare sempre i commit

Potete dire a git di firmare sempre i commit:

```
git config --global commit.gpgSign true
```

Note: Assicuratevi di aver configurato `gpg-agent` prima di abilitare questa opzione.

Come verificare l'identità degli sviluppatori del kernel

Firmare i tag e i commit è facile, ma come si fa a verificare che la chiave usata per firmare qualcosa appartenga davvero allo sviluppatore e non ad un impostore?

Configurare l'auto-key-retrieval usando WKD e DANE

Se non siete ancora in possesso di una vasta collezione di chiavi pubbliche di altri sviluppatori, allora potreste iniziare il vostro portachiavi affidandovi ai servizi di auto-scoperta e auto-recupero. GnuPG può affidarsi ad altre tecnologie di delega della fiducia, come DNSSEC e TLS, per sostenervi nel caso in cui iniziare una propria rete di fiducia da zero sia troppo scoraggiante.

Aggiungete il seguente testo al vostro file `~/.gnupg/gpg.conf`:

```
auto-key-locate wkd,dane,local
auto-key-retrieve
```

La *DNS-Based Authentication of Named Entities* (“DANE”) è un metodo per la pubblicazione di chiavi pubbliche su DNS e per renderle sicure usando zone firmate con DNSSEC. Il *Web Key Directory* (“WKD”) è un metodo alternativo che usa https a scopo di ricerca. Quando si usano DANE o WKD per la ricerca di chiavi pubbliche, GnuPG validerà i certificati DNSSEC o TLS prima di aggiungere al vostro portachiavi locale le eventuali chiavi trovate.

Kernel.org pubblica la WKD per tutti gli sviluppatori che hanno un account kernel.org. Una volta che avete applicato le modifiche al file `gpg.conf`, potrete autorecuperare le chiavi di Linus Torvalds e Greg Kroah-Hartman (se non le avete già):

```
$ gpg --locate-keys torvalds@kernel.org gregkh@kernel.org
```

Se avete un account kernel.org, al fine di rendere più utile l’uso di WKD da parte di altri sviluppatori del kernel, dovreste aggiungere alla vostra chiave lo UID di kernel.org.

Web of Trust (WOT) o Trust on First Use (TOFU)

PGP incorpora un meccanismo di delega della fiducia conosciuto come “Web of Trust”. Di base, questo è un tentativo di sostituire la necessità di un’autorità certificativa centralizzata tipica del mondo HTTPS/TLS. Invece di avere svariati produttori software che decidono chi dovrebbero essere le entità di certificazione di cui dovreste fidarvi, PGP lascia la responsabilità ad ogni singolo utente.

Sfortunatamente, solo poche persone capiscono come funziona la rete di fiducia. Nonostante sia un importante aspetto della specifica OpenPGP, recentemente le versioni di GnuPG (2.2 e successive) hanno implementato un meccanisco alternativo chiamato “Trust on First Use” (TOFU). Potete pensare a TOFU come “ad un approccio all fidicia simile ad SSH”. In SSH, la prima volta che vi connettete ad un sistema remoto, l’impronta digitale della chiave viene registrata e ricordata. Se la chiave dovesse cambiare in futuro, il programma SSH vi avviserà e si rifiuterà di connettersi, obbligandovi a prendere una decisione circa la fiducia che riponete nella nuova chiave. In modo simile, la prima volta che importate la chiave PGP di qualcuno, si assume sia valida. Se ad un certo punto GnuPG trova un’altra chiave con la stessa identità, entrambe, la vecchia e la nuova, verranno segnate come invalide e dovrete verificare manualmente quale tenere.

Vi raccomandiamo di usare il meccanisco TOFU+PGP (che è la nuova configurazione di base di GnuPG v2). Per farlo, aggiungete (o modificate) l’impostazione `trust-model` in `~/.gnupg/gpg.conf`:

```
trust-model tofu+pgp
```

Come usare i keyserver in sicurezza

Se ottenete l'errore "No public key" quando cercate di validate il tag di qualcuno, allora dovreste cercare quella chiave usando un keyserver. È importante tenere bene a mente che non c'è alcuna garanzia che la chiave che avete recuperato da un keyserver PGP appartenga davvero alla persona reale – è progettato così. Dovreste usare il Web of Trust per assicurarvi che la chiave sia valida.

Come mantenere il Web of Trust va oltre gli scopi di questo documento, semplicemente perché farlo come si deve richiede sia sforzi che perseveranza che tendono ad andare oltre al livello di interesse della maggior parte degli esseri umani. Qui di seguito alcuni rapidi suggerimenti per aiutarvi a ridurre il rischio di importare chiavi maligne.

Primo, diciamo che avete provato ad eseguire `git verify-tag` ma restituisce un errore dicendo che la chiave non è stata trovata:

```
$ git verify-tag sunxi-fixes-for-4.15-2
gpg: Signature made Sun 07 Jan 2018 10:51:55 PM EST
gpg:                using RSA key␣
↪DA73759BF8619E484E5A3B47389A54219C0F2430
gpg:                issuer "wens@...org"
gpg: Can't check signature: No public key
```

Cerchiamo nel keyserver per maggiori informazioni sull'impronta digitale della chiave (l'impronta digitale, probabilmente, appartiene ad una sottochiave, dunque non possiamo usarla direttamente senza trovare prima l'ID della chiave primaria associata ad essa):

```
$ gpg --search DA73759BF8619E484E5A3B47389A54219C0F2430
gpg: data source: hkp://keys.gnupg.net
(1) Chen-Yu Tsai <wens@...org>
      4096 bit RSA key C94035C21B4F2AEB, created: 2017-03-14,␣
↪expires: 2019-03-15
Keys 1-1 of 1 for "DA73759BF8619E484E5A3B47389A54219C0F2430". ␣
↪Enter number(s), N)ext, or Q)uit > q
```

Localizzate l'ID della chiave primaria, nel nostro esempio `C94035C21B4F2AEB`. Ora visualizzate le chiavi di Linus Torvalds che avete nel vostro portachiavi:

```
$ gpg --list-key torvalds@kernel.org
pub   rsa2048 2011-09-20 [SC]
      ABAF11C65A2970B130ABE3C479BE3E4300411886
uid           [ unknown] Linus Torvalds <torvalds@kernel.org>
sub   rsa2048 2011-09-20 [E]
```

Poi, aprite il PGP pathfinder. Nel campo "From", incollate l'impronta digitale della chiave di Linus Torvalds che si vede nell'output qui sopra. Nel campo "to", incollate il key-id della chiave sconosciuta che avete trovato con `gpg --search`, e poi verificare il risultato:

- Finding paths to Linus

Se trovate un paio di percorsi affidabili è un buon segno circa la validità della chiave. Ora, potete aggiungerla al vostro portachiavi dal keyserver:

```
$ gpg --recv-key C94035C21B4F2AEB
```

Questa procedura non è perfetta, e ovviamente state riponendo la vostra fiducia nell'amministratore del servizio *PGP Pathfinder* sperando che non sia malintenzionato (infatti, questo va contro *Fidatevi degli sviluppatori e non dell'infrastruttura*). Tuttavia, se mantenete con cura la vostra rete di fiducia sarà un deciso miglioramento rispetto alla cieca fiducia nei keyserver.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
../../../process/email-clients

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Informazioni sui programmi di posta elettronica per Linux

Git

Oggigiorno, la maggior parte degli sviluppatori utilizza `git send-email` al posto dei classici programmi di posta elettronica. Le pagine man sono abbastanza buone. Dal lato del ricevente, i manutentori utilizzano `git am` per applicare le patch.

Se siete dei novelli utilizzatori di `git` allora inviate la patch a voi stessi. Salvatela come testo includendo tutte le intestazioni. Poi eseguite il comando `git am messaggio-formato-testo.txt` e revisionatene il risultato con `git log`. Quando tutto funziona correttamente, allora potete inviare la patch alla lista di discussione più appropriata.

Panoramica delle opzioni

Le patch per il kernel vengono inviate per posta elettronica, preferibilmente come testo integrante del messaggio. Alcuni manutentori accettano gli allegati, ma in questo caso gli allegati devono avere il *content-type* impostato come `text/plain`. Tuttavia, generalmente gli allegati non sono ben apprezzati perché rende più difficile citare porzioni di patch durante il processo di revisione.

I programmi di posta elettronica che vengono usati per inviare le patch per il kernel Linux dovrebbero inviarle senza alterazioni. Per esempio, non dovrebbero modificare o rimuovere tabulazioni o spazi, nemmeno all'inizio o alla fine delle righe.

Non inviate patch con `format=flowed`. Questo potrebbe introdurre interruzioni di riga inaspettate e indesiderate.

Non lasciate che il vostro programma di posta vada a capo automaticamente. Questo può corrompere le patch.

I programmi di posta non dovrebbero modificare la codifica dei caratteri nel testo. Le patch inviate per posta elettronica dovrebbero essere codificate in ASCII o UTF-8. Se configurate il vostro programma per inviare messaggi codificati con UTF-8 eviterete possibili problemi di codifica.

I programmi di posta dovrebbero generare e mantenere le intestazioni "References" o "In-Reply-To:" cosicché la discussione non venga interrotta.

Di solito, il copia-e-incolla (o taglia-e-incolla) non funziona con le patch perché le tabulazioni vengono convertite in spazi. Usando xclipboard, xclip e/o xcutsel potrebbe funzionare, ma è meglio che lo verifichiate o meglio ancora: non usate il copia-e-incolla.

Non usate firme PGP/GPG nei messaggi che contengono delle patch. Questo impedisce il corretto funzionamento di alcuni script per leggere o applicare patch (questo si dovrebbe poter correggere).

Prima di inviare le patch sulle liste di discussione Linux, può essere una buona idea quella di inviare la patch a voi stessi, salvare il messaggio ricevuto, e applicarlo ai sorgenti con successo.

Alcuni suggerimenti per i programmi di posta elettronica (MUA)

Qui troverete alcuni suggerimenti per configurare i vostri MUA allo scopo di modificare ed inviare patch per il kernel Linux. Tuttavia, questi suggerimenti non sono da considerarsi come un riassunto di una configurazione completa.

Legenda:

- TUI = interfaccia utente testuale (*text-based user interface*)
- GUI = interfaccia utente grafica (*graphical user interface*)

Alpine (TUI)

Opzioni per la configurazione:

Nella sezione *Sending Preferences*:

- *Do Not Send Flowed Text* deve essere `enabled`
- *Strip Whitespace Before Sending* deve essere `disabled`

Quando state scrivendo un messaggio, il cursore dev' essere posizionato dove volete che la patch inizi, poi premendo `CTRL-R` vi verrà chiesto di selezionare il file patch da inserire nel messaggio.

Claws Mail (GUI)

Funziona. Alcune persone riescono ad usarlo con successo per inviare le patch.

Per inserire una patch usate *Messaggio→Inserisci file* (`CTRL-I`) oppure un editor esterno.

Se la patch che avete inserito dev' essere modificata usato la finestra di scrittura di Claws, allora assicuratevi che l ' "auto-interruzione" sia disabilitata *Configurazione→Preferenze→Composizione→Interruzione riga*.

Evolution (GUI)

Alcune persone riescono ad usarlo con successo per inviare le patch.

Quando state scrivendo una lettera selezionate: Preformattato
: da *Formato→Stile del paragrafo→Preformattato* (`CTRL-7`) o dalla barra degli strumenti

Poi per inserire la patch usate: *Inserisci→ File di testo⋯* (`ALT-N x`)

Potete anche eseguire `diff -Nru old.c new.c | xclip`, selezionare *Preformattato*, e poi usare il tasto centrale del mouse.

Kmail (GUI)

Alcune persone riescono ad usarlo con successo per inviare le patch.

La configurazione base che disabilita la composizione di messaggi HTML è corretta; non abilitatela.

Quando state scrivendo un messaggio, nel menu opzioni, togliete la selezione a "A capo automatico". L' unico svantaggio sarà che qualsiasi altra cosa scriviate nel messaggio non verrà mandata a capo in automatico ma dovrete farlo voi. Il modo più semplice per ovviare a questo problema è quello di scrivere il messaggio con l' opzione abilitata e poi di salvarlo nelle bozze. Riaprendo ora il messaggio dalle bozze le andate a capo saranno parte integrante del messaggio, per cui togliendo l' opzione "A capo automatico" non perderete nulla.

Alla fine del vostro messaggio, appena prima di inserire la vostra patch, aggiungete il delimitatore di patch: tre trattini (`---`).

Ora, dal menu *Messaggio*, selezionate *Inserisci file di testo⋯* quindi scegliete la vostra patch. Come soluzione aggiuntiva potreste personalizzare la vostra barra degli strumenti aggiungendo un' icona per *Inserisci file di testo⋯*.

Allargate la finestra di scrittura abbastanza da evitare andate a capo. Questo perché in Kmail 1.13.5 (KDE 4.5.4), Kmail aggiunge andate a capo automaticamente al momento dell' invio per tutte quelle righe che graficamente, nella vostra finestra di composizione, si sono estete su una riga successiva. Disabilitare l' andata a capo automatica non è sufficiente. Dunque, se la vostra patch contiene delle righe molto lunghe, allora dovrete allargare la finestra di composizione per evitare che quelle righe vadano a capo. Vedere: https://bugs.kde.org/show_bug.cgi?id=174034

Potete firmare gli allegati con GPG, ma per le patch si preferisce aggiungerle al testo del messaggio per cui non usate la firma GPG. Firmare le patch inserite come testo del messaggio le rende più difficili da estrarre dalla loro codifica a 7-bit.

Se dovete assolutamente inviare delle patch come allegati invece di integrarle nel testo del messaggio, allora premete il tasto destro sull'allegato e selezionate *Proprietà*, e poi attivate *Suggerisci visualizzazione automatica* per far si che l'allegato sia più leggibile venendo visualizzato come parte del messaggio.

Per salvare le patch inviate come parte di un messaggio, selezionate il messaggio che la contiene, premete il tasto destro e selezionate *Salva come*. Se il messaggio fu ben preparato, allora potrete usarlo interamente senza alcuna modifica. I messaggi vengono salvati con permessi di lettura-scrittura solo per l'utente, nel caso in cui vogliate copiarli altrove per renderli disponibili ad altri gruppi o al mondo, ricordatevi di usare `chmod` per cambiare i permessi.

Lotus Notes (GUI)

Scappate finché potete.

IBM Verse (Web GUI)

Vedi il commento per Lotus Notes.

Mutt (TUI)

Un sacco di sviluppatori Linux usano `mutt`, per cui deve funzionare abbastanza bene.

Mutt non ha un proprio editor, quindi qualunque sia il vostro editor dovrete configurarlo per non aggiungere automaticamente le andate a capo. Molti editor hanno un'opzione *Inserisci file* che inserisce il contenuto di un file senza alterarlo.

Per usare `vim` come editor per mutt:

```
set editor="vi"
```

Se per inserire la patch nel messaggio usate xclip, scrivete il comando:

```
:set paste
```

prima di premere il tasto centrale o shift-insert. Oppure usate il comando:

```
:r filename
```

(a)llega funziona bene senza `set paste`

Potete generare le patch con `git format-patch` e usare Mutt per inviarle:

```
$ mutt -H 0001-some-bug-fix.patch
```

Opzioni per la configurazione:

Tutto dovrebbe funzionare già nella configurazione base. Tuttavia, è una buona idea quella di impostare `send_charset`:

```
set send_charset="us-ascii:utf-8"
```

Mutt è molto personalizzabile. Qui di seguito trovate la configurazione minima per iniziare ad usare Mutt per inviare patch usando Gmail:

```
# .muttrc
# ================  IMAP ====================
set imap_user = 'yourusername@gmail.com'
set imap_pass = 'yourpassword'
set spoolfile = imaps://imap.gmail.com/INBOX
set folder = imaps://imap.gmail.com/
set record="imaps://imap.gmail.com/[Gmail]/Sent Mail"
set postponed="imaps://imap.gmail.com/[Gmail]/Drafts"
set mbox="imaps://imap.gmail.com/[Gmail]/All Mail"

# ================  SMTP  ====================
set smtp_url = "smtp://username@smtp.gmail.com:587/"
set smtp_pass = $imap_pass
set ssl_force_tls = yes # Require encrypted connection

# ================  Composition  ====================
set editor = `echo \$EDITOR`
set edit_headers = yes  # See the headers when editing
set charset = UTF-8     # value of $LANG; also fallback for send_
↪charset
# Sender, email address, and sign-off line must match
unset use_domain        # because joe@localhost is just embarrassing
set realname = "YOUR NAME"
set from = "username@gmail.com"
set use_from = yes
```

La documentazione di Mutt contiene molte più informazioni:

> https://gitlab.com/muttmua/mutt/-/wikis/UseCases/Gmail
>
> http://www.mutt.org/doc/manual/

Pine (TUI)

Pine aveva alcuni problemi con gli spazi vuoti, ma questi dovrebbero essere stati risolti.

Se potete usate alpine (il successore di pine).

Opzioni di configurazione:

- Nelle versioni più recenti è necessario avere `quell-flowed-text`
- l'opzione `no-strip-whitespace-before-send` è necessaria

Sylpheed (GUI)

- funziona bene per aggiungere testo in linea (o usando allegati)
- permette di utilizzare editor esterni
- è lento su cartelle grandi
- non farà l'autenticazione TSL SMTP su una connessione non SSL
- ha un utile righello nella finestra di scrittura
- la rubrica non comprende correttamente il nome da visualizzare e l'indirizzo associato

Thunderbird (GUI)

Thunderbird è un clone di Outlook a cui piace maciullare il testo, ma esistono modi per impedirglielo.

- permettere l'uso di editor esterni: La cosa più semplice da fare con Thunderbird e le patch è quello di usare l'estensione "external editor" e di usare il vostro `$EDITOR` preferito per leggere/includere patch nel vostro messaggio. Per farlo, scaricate ed installate l'estensione e aggiungete un bottone per chiamarla rapidamente usando *Visualizza→Barra degli strumenti→Personalizza⋯*; una volta fatto potrete richiamarlo premendo sul bottone mentre siete nella finestra *Scrivi*

  Tenete presente che "external editor" richiede che il vostro editor non faccia alcun fork, in altre parole, l'editor non deve ritornare prima di essere stato chiuso. Potreste dover passare dei parametri aggiuntivi al vostro editor oppure cambiargli la configurazione. Per esempio, usando gvim dovrete aggiungere l'opzione -f `/usr/bin/gvim -f` (Se il binario si trova in `/usr/bin`) nell'apposito campo nell'interfaccia di configurazione di *external editor*. Se usate altri editor consultate il loro manuale per sapere come configurarli.

Per rendere l'editor interno un po' più sensato, fate così:

- Modificate le impostazioni di Thunderbird per far si che non usi `format=flowed`. Andate in *Modifica→Preferenze→Avanzate→Editor di configurazione* per invocare il registro delle impostazioni.
- impostate `mailnews.send_plaintext_flowed` a `false`
- impostate `mailnews.wraplength` da `72` a `0`
- *Visualizza→Corpo del messaggio come→Testo semplice*
- *Visualizza→Codifica del testo→Unicode*

TkRat (GUI)

Funziona. Usare “Inserisci file⋯” o un editor esterno.

Gmail (Web GUI)

Non funziona per inviare le patch.

Il programma web Gmail converte automaticamente i tab in spazi.

Allo stesso tempo aggiunge andata a capo ogni 78 caratteri. Comunque il problema della conversione fra spazi e tab può essere risolto usando un editor esterno.

Un altro problema è che Gmail usa la codifica base64 per tutti quei messaggi che contengono caratteri non ASCII. Questo include cose tipo i nomi europei.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l’unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/kernel-enforcement-statement.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Applicazione della licenza sul kernel Linux

Come sviluppatori del kernel Linux, abbiamo un certo interessa su come il nostro software viene usato e su come la sua licenza viene fatta rispettare. Il rispetto reciproco degli obblighi di condivisione della GPL-2.0 è fondamentale per la sostenibilità di lungo periodo del nostro software e della nostra comunità.

Benché ognuno abbia il diritto a far rispettare il diritto d’autore per i propri contributi alla nostra comunità, condividiamo l’interesse a far si che ogni azione individuale nel far rispettare i propri diritti sia condotta in modo da portare beneficio alla comunità e che non abbia, involontariamente, impatti negativi sulla salute e la crescita del nostro ecosistema software. Al fine di scoraggiare l’esecuzione di azioni inutili, concordiamo che è nel migliore interesse della nostra comunità di sviluppo di impegnarci nel rispettare i seguenti obblighi nei confronti degli utenti del kernel Linux per conto nostro e di qualsiasi successore ai nostri interessi sul diritto d’autore:

> Malgrado le clausole di risoluzione della licenza GPL-2.0, abbiamo concordato che è nel migliore interesse della nostra comunità di sviluppo adottare le seguenti disposizioni della GPL-3.0 come permessi aggiuntivi alla nostra licenza nei confronti di qualsiasi affermazione non difensiva di diritti sulla licenza.
>
> > In ogni caso, se cessano tutte le violazioni di questa Licenza, allora la tua licenza da parte di un dato detentore del copyright viene ripristinata (a) in via cautelativa, a meno che e fino

> a quando il detentore del copyright non cessa esplicitamente e definitivamente la tua licenza, e (b) in via permanente se il detentore del copyright non ti notifica in alcun modo la violazione entro 60 giorni dalla cessazione della licenza.
>
> Inoltre, la tua licenza da parte di un dato detentore del copyright viene ripristinata in maniera permanente se il detentore del copyright ti notifica la violazione in maniera adeguata, se questa è la prima volta che ricevi una notifica di violazione di questa Licenza (per qualunque Programma) dallo stesso detentore di copyright, e se rimedi alla violazione entro 30 giorni dalla data di ricezione della notifica di violazione.

Fornendo queste garanzie, abbiamo l'intenzione di incoraggiare l'uso del software. Vogliamo che le aziende e le persone usino, modifichino e distribuiscano a questo software. Vogliamo lavorare con gli utenti in modo aperto e trasparente per eliminare ogni incertezza circa le nostre aspettative sul rispetto o l'ottemperanza alla licenza che possa limitare l'uso del nostro software. Vediamo l'azione legale come ultima spiaggia, da avviare solo quando gli altri sforzi della comunità hanno fallito nel risolvere il problema.

Per finire, una volta che un problema di non rispetto della licenza viene risolto, speriamo che gli utenti si sentano i benvenuti ad aggregarsi a noi nello sviluppo di questo progetto. Lavorando assieme, saremo più forti.

Ad eccezione deve specificato, parliamo per noi stessi, e non per una qualsiasi azienda per la quale lavoriamo oggi, o per cui abbiamo lavorato in passato, o lavoreremo in futuro.

- Laura Abbott
- Bjorn Andersson (Linaro)
- Andrea Arcangeli
- Neil Armstrong
- Jens Axboe
- Pablo Neira Ayuso
- Khalid Aziz
- Ralf Baechle
- Felipe Balbi
- Arnd Bergmann
- Ard Biesheuvel
- Tim Bird
- Paolo Bonzini
- Christian Borntraeger
- Mark Brown (Linaro)
- Paul Burton
- Javier Martinez Canillas

- Rob Clark
- Kees Cook (Google)
- Jonathan Corbet
- Dennis Dalessandro
- Vivien Didelot (Savoir-faire Linux)
- Hans de Goede
- Mel Gorman (SUSE)
- Sven Eckelmann
- Alex Elder (Linaro)
- Fabio Estevam
- Larry Finger
- Bhumika Goyal
- Andy Gross
- Juergen Gross
- Shawn Guo
- Ulf Hansson
- Stephen Hemminger (Microsoft)
- Tejun Heo
- Rob Herring
- Masami Hiramatsu
- Michal Hocko
- Simon Horman
- Johan Hovold (Hovold Consulting AB)
- Christophe JAILLET
- Olof Johansson
- Lee Jones (Linaro)
- Heiner Kallweit
- Srinivas Kandagatla
- Jan Kara
- Shuah Khan (Samsung)
- David Kershner
- Jaegeuk Kim
- Namhyung Kim
- Colin Ian King

- Jeff Kirsher
- Greg Kroah-Hartman (Linux Foundation)
- Christian König
- Vinod Koul
- Krzysztof Kozlowski
- Viresh Kumar
- Aneesh Kumar K.V
- Julia Lawall
- Doug Ledford
- Chuck Lever (Oracle)
- Daniel Lezcano
- Shaohua Li
- Xin Long
- Tony Luck
- Catalin Marinas (Arm Ltd)
- Mike Marshall
- Chris Mason
- Paul E. McKenney
- Arnaldo Carvalho de Melo
- David S. Miller
- Ingo Molnar
- Kuninori Morimoto
- Trond Myklebust
- Martin K. Petersen (Oracle)
- Borislav Petkov
- Jiri Pirko
- Josh Poimboeuf
- Sebastian Reichel (Collabora)
- Guenter Roeck
- Joerg Roedel
- Leon Romanovsky
- Steven Rostedt (VMware)
- Frank Rowand
- Ivan Safonov

- Anna Schumaker
- Jes Sorensen
- K.Y. Srinivasan
- David Sterba (SUSE)
- Heiko Stuebner
- Jiri Kosina (SUSE)
- Willy Tarreau
- Dmitry Torokhov
- Linus Torvalds
- Thierry Reding
- Rik van Riel
- Luis R. Rodriguez
- Geert Uytterhoeven (Glider bvba)
- Eduardo Valentin (Amazon.com)
- Daniel Vetter
- Linus Walleij
- Richard Weinberger
- Dan Williams
- Rafael J. Wysocki
- Arvind Yadav
- Masahiro Yamada
- Wei Yongjun
- Lv Zheng
- Marc Zyngier (Arm Ltd)

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/kernel-driver-statement.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Dichiarazioni sui driver per il kernel

Presa di posizione sui moduli per il kernel Linux

Noi, i sottoscritti sviluppatori del kernel, consideriamo pericoloso o indesiderato qualsiasi modulo o driver per il kernel Linux di tipo *a sorgenti chiusi* (*closed-source*). Ripetutamente, li abbiamo trovati deleteri per gli utenti Linux, le aziende, ed in generale l'ecosistema Linux. Questi moduli impediscono l'apertura, la stabilità, la flessibilità, e la manutenibilità del modello di sviluppo di Linux e impediscono ai loro utenti di beneficiare dell'esperienza dalla comunità Linux. I fornitori che distribuiscono codice a sorgenti chiusi obbligano i propri utenti a rinunciare ai principali vantaggi di Linux o a cercarsi nuovi fornitori. Perciò, al fine di sfruttare i vantaggi che codice aperto ha da offrire, come l'abbattimento dei costi e un supporto condiviso, spingiamo i fornitori ad adottare una politica di supporto ai loro clienti Linux che preveda il rilascio dei sorgenti per il kernel.

Parliamo solo per noi stessi, e non per una qualsiasi azienda per la quale lavoriamo oggi, o abbiamo lavorato in passato, o lavoreremo in futuro.

- Dave Airlie
- Nick Andrew
- Jens Axboe
- Ralf Baechle
- Felipe Balbi
- Ohad Ben-Cohen
- Muli Ben-Yehuda
- Jiri Benc
- Arnd Bergmann
- Thomas Bogendoerfer
- Vitaly Bordug
- James Bottomley
- Josh Boyer
- Neil Brown
- Mark Brown
- David Brownell
- Michael Buesch
- Franck Bui-Huu
- Adrian Bunk
- François Cami
- Ralph Campbell
- Luiz Fernando N. Capitulino

- Mauro Carvalho Chehab
- Denis Cheng
- Jonathan Corbet
- Glauber Costa
- Alan Cox
- Magnus Damm
- Ahmed S. Darwish
- Robert P. J. Day
- Hans de Goede
- Arnaldo Carvalho de Melo
- Helge Deller
- Jean Delvare
- Mathieu Desnoyers
- Sven-Thorsten Dietrich
- Alexey Dobriyan
- Daniel Drake
- Alex Dubov
- Randy Dunlap
- Michael Ellerman
- Pekka Enberg
- Jan Engelhardt
- Mark Fasheh
- J. Bruce Fields
- Larry Finger
- Jeremy Fitzhardinge
- Mike Frysinger
- Kumar Gala
- Robin Getz
- Liam Girdwood
- Jan-Benedict Glaw
- Thomas Gleixner
- Brice Goglin
- Cyrill Gorcunov
- Andy Gospodarek

- Thomas Graf
- Krzysztof Halasa
- Harvey Harrison
- Stephen Hemminger
- Michael Hennerich
- Tejun Heo
- Benjamin Herrenschmidt
- Kristian Høgsberg
- Henrique de Moraes Holschuh
- Marcel Holtmann
- Mike Isely
- Takashi Iwai
- Olof Johansson
- Dave Jones
- Jesper Juhl
- Matthias Kaehlcke
- Kenji Kaneshige
- Jan Kara
- Jeremy Kerr
- Russell King
- Olaf Kirch
- Roel Kluin
- Hans-Jürgen Koch
- Auke Kok
- Peter Korsgaard
- Jiri Kosina
- Aaro Koskinen
- Mariusz Kozlowski
- Greg Kroah-Hartman
- Michael Krufky
- Aneesh Kumar
- Clemens Ladisch
- Christoph Lameter
- Gunnar Larisch

- Anders Larsen
- Grant Likely
- John W. Linville
- Yinghai Lu
- Tony Luck
- Pavel Machek
- Matt Mackall
- Paul Mackerras
- Roland McGrath
- Patrick McHardy
- Kyle McMartin
- Paul Menage
- Thierry Merle
- Eric Miao
- Akinobu Mita
- Ingo Molnar
- James Morris
- Andrew Morton
- Paul Mundt
- Oleg Nesterov
- Luca Olivetti
- S.Çağlar Onur
- Pierre Ossman
- Keith Owens
- Venkatesh Pallipadi
- Nick Piggin
- Nicolas Pitre
- Evgeniy Polyakov
- Richard Purdie
- Mike Rapoport
- Sam Ravnborg
- Gerrit Renker
- Stefan Richter
- David Rientjes

- Luis R. Rodriguez
- Stefan Roese
- Francois Romieu
- Rami Rosen
- Stephen Rothwell
- Maciej W. Rozycki
- Mark Salyzyn
- Yoshinori Sato
- Deepak Saxena
- Holger Schurig
- Amit Shah
- Yoshihiro Shimoda
- Sergei Shtylyov
- Kay Sievers
- Sebastian Siewior
- Rik Snel
- Jes Sorensen
- Alexey Starikovskiy
- Alan Stern
- Timur Tabi
- Hirokazu Takata
- Eliezer Tamir
- Eugene Teo
- Doug Thompson
- FUJITA Tomonori
- Dmitry Torokhov
- Marcelo Tosatti
- Steven Toth
- Theodore Tso
- Matthias Urlichs
- Geert Uytterhoeven
- Arjan van de Ven
- Ivo van Doorn
- Rik van Riel

- Wim Van Sebroeck
- Hans Verkuil
- Horst H. von Brand
- Dmitri Vorobiev
- Anton Vorontsov
- Daniel Walker
- Johannes Weiner
- Harald Welte
- Matthew Wilcox
- Dan J. Williams
- Darrick J. Wong
- David Woodhouse
- Chris Wright
- Bryan Wu
- Rafael J. Wysocki
- Herbert Xu
- Vlad Yasevich
- Peter Zijlstra
- Bartlomiej Zolnierkiewicz

Poi ci sono altre guide sulla comunità che sono di interesse per molti degli sviluppatori:

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/changes.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Requisiti minimi per compilare il kernel

Introduzione

Questo documento fornisce una lista dei software necessari per eseguire i kernel 4.x.

Questo documento è basato sul file "Changes" del kernel 2.0.x e quindi le persone che lo scrissero meritano credito (Jared Mauch, Axel Boldt, Alessandro Sigala, e tanti altri nella rete).

Requisiti minimi correnti

Prima di pensare d' avere trovato un baco, aggiornate i seguenti programmi **almeno** alla versione indicata! Se non siete certi della versione che state usando, il comando indicato dovrebbe dirvelo.

Questa lista presume che abbiate già un kernel Linux funzionante. In aggiunta, non tutti gli strumenti sono necessari ovunque; ovviamente, se non avete una PC Card, per esempio, probabilmente non dovreste preoccuparvi di pcmciautils.

Programma	Versione minima	Comando per verificare la versione
GNU C	4.6	gcc –version
GNU make	3.81	make –version
binutils	2.21	ld -v
flex	2.5.35	flex –version
bison	2.0	bison –version
util-linux	2.10o	fdformat –version
kmod	13	depmod -V
e2fsprogs	1.41.4	e2fsck -V
jfsutils	1.1.3	fsck.jfs -V
reiserfsprogs	3.6.3	reiserfsck -V
xfsprogs	2.6.0	xfs_db -V
squashfs-tools	4.0	mksquashfs -version
btrfs-progs	0.18	btrfsck
pcmciautils	004	pccardctl -V
quota-tools	3.09	quota -V
PPP	2.4.0	pppd –version
nfs-utils	1.0.5	showmount –version
procps	3.2.0	ps –version
oprofile	0.9	oprofiled –version
udev	081	udevd –version
grub	0.93	grub –version || grub-install –version
mcelog	0.6	mcelog –version
iptables	1.4.2	iptables -V
openssl & libcrypto	1.0.0	openssl version
bc	1.06.95	bc –version
Sphinx[1]	1.3	sphinx-build –version

[1] Sphinx è necessario solo per produrre la documentazione del Kernel

Compilazione del kernel

GCC

La versione necessaria di gcc potrebbe variare a seconda del tipo di CPU nel vostro calcolatore.

Make

Per compilare il kernel vi servirà GNU make 3.81 o successivo.

Binutils

Per generare il kernel è necessario avere Binutils 2.21 o superiore.

pkg-config

Il sistema di compilazione, dalla versione 4.18, richiede pkg-config per verificare l'esistenza degli strumenti kconfig e per determinare le impostazioni da usare in 'make {g,x}config'. Precedentemente pkg-config veniva usato ma non verificato o documentato.

Flex

Dalla versione 4.16, il sistema di compilazione, durante l'esecuzione, genera un analizzatore lessicale. Questo richiede flex 2.5.35 o successivo.

Bison

Dalla versione 4.16, il sistema di compilazione, durante l'esecuzione, genera un parsificatore. Questo richiede bison 2.0 o successivo.

Perl

Per compilare il kernel vi servirà perl 5 e i seguenti moduli `Getopt::Long`, `Getopt::Std`, `File::Basename`, e `File::Find`.

BC

Vi servirà bc per compilare i kernel dal 3.10 in poi.

OpenSSL

Il programma OpenSSL e la libreria crypto vengono usati per la firma dei moduli e la gestione dei certificati; sono usati per la creazione della chiave e la generazione della firma.

Se la firma dei moduli è abilitata, allora vi servirà openssl per compilare il kernel 3.7 e successivi. Vi serviranno anche i pacchetti di sviluppo di openssl per compilare il kernel 4.3 o successivi.

Strumenti di sistema

Modifiche architetturali

DevFS è stato reso obsoleto da udev (http://www.kernel.org/pub/linux/utils/kernel/hotplug/)

Il supporto per UID a 32-bit è ora disponibile. Divertitevi!

La documentazione delle funzioni in Linux è una fase di transizione verso una documentazione integrata nei sorgenti stessi usando dei commenti formattati in modo speciale e posizionati vicino alle funzioni che descrivono. Al fine di arricchire la documentazione, questi commenti possono essere combinati con i file ReST presenti in Documentation/; questi potranno poi essere convertiti in formato PostScript, HTML, LaTex, ePUB o PDF. Per convertire i documenti da ReST al formato che volete, avete bisogno di Sphinx.

Util-linux

Le versioni più recenti di util-linux: forniscono il supporto a `fdisk` per dischi di grandi dimensioni; supportano le nuove opzioni di mount; riconoscono più tipi di partizioni; hanno un fdformat che funziona con i kernel 2.4; e altre chicche. Probabilmente vorrete aggiornarlo.

Ksymoops

Se l'impensabile succede e il kernel va in oops, potrebbe servirvi lo strumento ksymoops per decodificarlo, ma nella maggior parte dei casi non vi servirà. Generalmente è preferibile compilare il kernel con l'opzione `CONFIG_KALLSYMS` cosicché venga prodotto un output più leggibile che può essere usato così com'è (produce anche un output migliore di ksymoops). Se per qualche motivo il vostro kernel non è stato compilato con `CONFIG_KALLSYMS` e non avete modo di ricompilarlo e riprodurre l'oops con quell'opzione abilitata, allora potete usare ksymoops per decodificare l'oops.

Mkinitrd

I cambiamenti della struttura in `/lib/modules` necessita l'aggiornamento di mkinitrd.

E2fsprogs

L'ultima versione di `e2fsprogs` corregge diversi bachi in fsck e debugfs. Ovviamente, aggiornarlo è una buona idea.

JFSutils

Il pacchetto `jfsutils` contiene programmi per il file-system JFS. Sono disponibili i seguenti strumenti:

- `fsck.jfs` - avvia la ripetizione del log delle transizioni, e verifica e ripara una partizione formattata secondo JFS
- `mkfs.jfs` - crea una partizione formattata secondo JFS
- sono disponibili altri strumenti per il file-system.

Reiserfsprogs

Il pacchetto reiserfsprogs dovrebbe essere usato con reiserfs-3.6.x (Linux kernel 2.4.x). Questo è un pacchetto combinato che contiene versioni funzionanti di `mkreiserfs`, `resize_reiserfs`, `debugreiserfs` e `reiserfsck`. Questi programmi funzionano sulle piattaforme i386 e alpha.

Xfsprogs

L'ultima versione di `xfsprogs` contiene, fra i tanti, i programmi `mkfs.xfs`, `xfs_db` e `xfs_repair` per il file-system XFS. Dipendono dell'architettura e qualsiasi versione dalla 2.0.0 in poi dovrebbe funzionare correttamente con la versione corrente del codice XFS nel kernel (sono raccomandate le versioni 2.6.0 o successive per via di importanti miglioramenti).

PCMCIAutils

PCMCIAutils sostituisce `pcmica-cs`. Serve ad impostare correttamente i connettori PCMCIA all'avvio del sistema e a caricare i moduli necessari per i dispositivi a 16-bit se il kernel è stato modularizzato e il sottosistema hotplug è in uso.

Quota-tools

Il supporto per uid e gid a 32 bit richiedono l'uso della versione 2 del formato quota. La versione 3.07 e successive di quota-tools supportano questo formato. Usate la versione raccomandata nella lista qui sopra o una successiva.

Micro codice per Intel IA32

Per poter aggiornare il micro codice per Intel IA32, è stato aggiunto un apposito driver; il driver è accessibile come un normale dispositivo a caratteri (misc). Se non state usando udev probabilmente sarà necessario eseguire i seguenti comandi come root prima di poterlo aggiornare:

```
mkdir /dev/cpu
mknod /dev/cpu/microcode c 10 184
chmod 0644 /dev/cpu/microcode
```

Probabilmente, vorrete anche il programma microcode_ctl da usare con questo dispositivo.

udev

`udev` è un programma in spazio utente il cui scopo è quello di popolare dinamicamente la cartella `/dev` coi dispositivi effettivamente presenti. `udev` sostituisce le funzionalità base di devfs, consentendo comunque nomi persistenti per i dispositivi.

FUSE

Serve libfuse 2.4.0 o successiva. Il requisito minimo assoluto è 2.3.0 ma le opzioni di mount `direct_io` e `kernel_cache` non funzioneranno.

Rete

Cambiamenti generali

Se per quanto riguarda la configurazione di rete avete esigenze di un certo livello dovreste prendere in considerazione l'uso degli strumenti in ip-route2.

Filtro dei pacchetti / NAT

Il codice per filtraggio dei pacchetti e il NAT fanno uso degli stessi strumenti come nelle versioni del kernel antecedenti la 2.4.x (iptables). Include ancora moduli di compatibilità per 2.2.x ipchains e 2.0.x ipdwadm.

PPP

Il driver per PPP è stato ristrutturato per supportare collegamenti multipli e per funzionare su diversi livelli. Se usate PPP, aggiornate pppd almeno alla versione 2.4.0.

Se non usate udev, dovete avere un file /dev/ppp che può essere creato da root col seguente comando:

```
mknod /dev/ppp c 108 0
```

NFS-utils

Nei kernel più antichi (2.4 e precedenti), il server NFS doveva essere informato sui clienti ai quali si voleva fornire accesso via NFS. Questa informazione veniva passata al kernel quando un cliente montava un file-system mediante `mountd`, oppure usando `exportfs` all' avvio del sistema. exportfs prende le informazioni circa i clienti attivi da `/var/lib/nfs/rmtab`.

Questo approccio è piuttosto delicato perché dipende dalla correttezza di rmtab, che non è facile da garantire, in particolare quando si cerca di implementare un *failover*. Anche quando il sistema funziona bene, `rmtab` ha il problema di accumulare vecchie voci inutilizzate.

Sui kernel più recenti il kernel ha la possibilità di informare mountd quando arriva una richiesta da una macchina sconosciuta, e mountd può dare al kernel le informazioni corrette per l' esportazione. Questo rimuove la dipendenza con `rmtab` e significa che il kernel deve essere al corrente solo dei clienti attivi.

Per attivare questa funzionalità, dovete eseguire il seguente comando prima di usare exportfs o mountd:

```
mount -t nfsd nfsd /proc/fs/nfsd
```

Dove possibile, raccomandiamo di proteggere tutti i servizi NFS dall' accesso via internet mediante un firewall.

mcelog

Quando `CONFIG_x86_MCE` è attivo, il programma mcelog processa e registra gli eventi *machine check*. Gli eventi *machine check* sono errori riportati dalla CPU. Incoraggiamo l'analisi di questi errori.

Documentazione del kernel

Sphinx

Per i dettaglio sui requisiti di Sphinx, fate riferimento a *Installazione Sphinx* in Introduzione

Ottenere software aggiornato

Compilazione del kernel

gcc

- <ftp://ftp.gnu.org/gnu/gcc/>

Make

- <ftp://ftp.gnu.org/gnu/make/>

Binutils

- <https://www.kernel.org/pub/linux/devel/binutils/>

Flex

- <https://github.com/westes/flex/releases>

Bison

- <ftp://ftp.gnu.org/gnu/bison/>

OpenSSL

- <https://www.openssl.org/>

Strumenti di sistema

Util-linux

- <https://www.kernel.org/pub/linux/utils/util-linux/>

Kmod

- <https://www.kernel.org/pub/linux/utils/kernel/kmod/>
- <https://git.kernel.org/pub/scm/utils/kernel/kmod/kmod.git>

Ksymoops

- <https://www.kernel.org/pub/linux/utils/kernel/ksymoops/v2.4/>

Mkinitrd

- <https://code.launchpad.net/initrd-tools/main>

E2fsprogs

- <https://www.kernel.org/pub/linux/kernel/people/tytso/e2fsprogs/>
- <https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git/>

JFSutils

- <http://jfs.sourceforge.net/>

Reiserfsprogs

- <https://git.kernel.org/pub/scm/linux/kernel/git/jeffm/reiserfsprogs.git/>

Xfsprogs

- <https://git.kernel.org/pub/scm/fs/xfs/xfsprogs-dev.git>
- <https://www.kernel.org/pub/linux/utils/fs/xfs/xfsprogs/>

Pcmciautils

- <https://www.kernel.org/pub/linux/utils/kernel/pcmcia/>

Quota-tools

- <http://sourceforge.net/projects/linuxquota/>

Microcodice Intel P6

- <https://downloadcenter.intel.com/>

udev

- <http://www.freedesktop.org/software/systemd/man/udev.html>

FUSE

- <https://github.com/libfuse/libfuse/releases>

mcelog

- <http://www.mcelog.org/>

Rete

PPP

- <https://download.samba.org/pub/ppp/>
- <https://git.ozlabs.org/?p=ppp.git>
- <https://github.com/paulusmack/ppp/>

NFS-utils

- <http://sourceforge.net/project/showfiles.php?group_id=14>

Iptables

- <https://netfilter.org/projects/iptables/index.html>

Ip-route2

- <https://www.kernel.org/pub/linux/utils/net/iproute2/>

OProfile

- <http://oprofile.sf.net/download/>

NFS-Utils

- <http://nfs.sourceforge.net/>

Documentazione del kernel

Sphinx

- <http://www.sphinx-doc.org/>

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/submitting-drivers.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Sottomettere driver per il kernel Linux

Note: Questo documento è vecchio e negli ultimi anni non è stato più aggiornato; dovrebbe essere aggiornato, o forse meglio, rimosso. La maggior parte di quello che viene detto qui può essere trovato anche negli altri documenti dedicati allo sviluppo. Per questo motivo il documento non verrà tradotto.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/stable-api-nonsense.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

L'interfaccia dei driver per il kernel Linux

(tutte le risposte alle vostre domande e altro)

Greg Kroah-Hartman <greg@kroah.com>

Questo è stato scritto per cercare di spiegare perché Linux **non ha un'interfaccia binaria, e non ha nemmeno un' interfaccia stabile**.

Note: Questo articolo parla di interfacce **interne al kernel**, non delle interfacce verso lo spazio utente.

L'interfaccia del kernel verso lo spazio utente è quella usata dai programmi, ovvero le chiamate di sistema. Queste interfacce sono **molto** stabili nel tempo e non verranno modificate. Ho vecchi programmi che sono stati compilati su un kernel 0.9 (circa) e tuttora funzionano sulle versioni 2.6 del kernel. Queste interfacce sono quelle che gli utenti e i programmatori possono considerare stabili.

Riepilogo generale

Pensate di volere un' interfaccia del kernel stabile, ma in realtà non la volete, e nemmeno sapete di non volerla. Quello che volete è un driver stabile che funzioni, e questo può essere ottenuto solo se il driver si trova nei sorgenti del kernel. Ci sono altri vantaggi nell' avere il proprio driver nei sorgenti del kernel, ognuno dei quali hanno reso Linux un sistema operativo robusto, stabile e maturo; questi sono anche i motivi per cui avete scelto Linux.

Introduzione

Solo le persone un po' strambe vorrebbero scrivere driver per il kernel con la costante preoccupazione per i cambiamenti alle interfacce interne. Per il resto del mondo, queste interfacce sono invisibili o non di particolare interesse.

Innanzitutto, non tratterò **alcun** problema legale riguardante codice chiuso, nascosto, avvolto, blocchi binari, o qualsia altra cosa che descrive driver che non hanno i propri sorgenti rilasciati con licenza GPL. Per favore fate riferimento ad un avvocato per qualsiasi questione legale, io sono un programmatore e perciò qui vi parlerò soltanto delle questioni tecniche (non per essere superficiali sui problemi legali, sono veri e dovete esserne a conoscenza in ogni circostanza).

Dunque, ci sono due tematiche principali: interfacce binarie del kernel e interfacce stabili nei sorgenti. Ognuna dipende dall'altra, ma discuteremo prima delle cose binarie per toglierle di mezzo.

Interfaccia binaria del kernel

Supponiamo d'avere un'interfaccia stabile nei sorgenti del kernel, di conseguenza un'interfaccia binaria dovrebbe essere anche' essa stabile, giusto? Sbagliato. Prendete in considerazione i seguenti fatti che riguardano il kernel Linux:

- A seconda della versione del compilatore C che state utilizzando, diverse strutture dati del kernel avranno un allineamento diverso, e possibilmente un modo diverso di includere le funzioni (renderle inline oppure no). L'organizzazione delle singole funzioni non è poi così importante, ma la spaziatura (*padding*) nelle strutture dati, invece, lo è.
- In base alle opzioni che sono state selezionate per generare il kernel, un certo numero di cose potrebbero succedere:
  - strutture dati differenti potrebbero contenere campi differenti
  - alcune funzioni potrebbero non essere implementate (per esempio, alcuni *lock* spariscono se compilati su sistemi mono-processore)
  - la memoria interna del kernel può essere allineata in differenti modi a seconda delle opzioni di compilazione.
- Linux funziona su una vasta gamma di architetture di processore. Non esiste alcuna possibilità che il binario di un driver per un'architettura funzioni correttamente su un'altra.

Alcuni di questi problemi possono essere risolti compilando il proprio modulo con la stessa identica configurazione del kernel, ed usando la stessa versione del compilatore usato per compilare il kernel. Questo è sufficiente se volete fornire un modulo per uno specifico rilascio su una specifica distribuzione Linux. Ma moltiplicate questa singola compilazione per il numero di distribuzioni Linux e il numero dei rilasci supportati da quest' ultime e vi troverete rapidamente in un incubo fatto di configurazioni e piattaforme hardware (differenti processori con differenti opzioni); dunque, anche per il singolo rilascio di un modulo, dovreste creare differenti versioni dello stesso.

Fidatevi, se tenterete questa via, col tempo, diventerete pazzi; l'ho imparato a mie spese molto tempo fa…

Interfaccia stabile nei sorgenti del kernel

Se parlate con le persone che cercano di mantenere aggiornato un driver per Linux ma che non si trova nei sorgenti, allora per queste persone l'argomento sarà "ostico".

Lo sviluppo del kernel Linux è continuo e viaggia ad un ritmo sostenuto, e non rallenta mai. Perciò, gli sviluppatori del kernel trovano bachi nelle interfacce attuali, o trovano modi migliori per fare le cose. Se le trovano, allora le correggeranno per migliorarle. In questo frangente, i nomi delle funzioni potrebbero cambiare, le strutture dati potrebbero diventare più grandi o più piccole, e gli argomenti delle funzioni potrebbero essere ripensati. Se questo dovesse succedere, nello stesso momento, tutte le istanze dove questa interfaccia viene utilizzata verranno corrette, garantendo che tutto continui a funzionare senza problemi.

Portiamo ad esempio l'interfaccia interna per il sottosistema USB che ha subito tre ristrutturazioni nel corso della sua vita. Queste ristrutturazioni furono fatte per risolvere diversi problemi:

- È stato fatto un cambiamento da un flusso di dati sincrono ad uno asincrono. Questo ha ridotto la complessità di molti driver e ha aumentato la capacità di trasmissione di tutti i driver fino a raggiungere quasi la velocità massima possibile.
- È stato fatto un cambiamento nell'allocazione dei pacchetti da parte del sottosistema USB per conto dei driver, cosicché ora i driver devono fornire più informazioni al sottosistema USB al fine di correggere un certo numero di stalli.

Questo è completamente l'opposto di quello che succede in alcuni sistemi operativi proprietari che hanno dovuto mantenere, nel tempo, il supporto alle vecchie interfacce USB. I nuovi sviluppatori potrebbero usare accidentalmente le vecchie interfacce e sviluppare codice nel modo sbagliato, portando, di conseguenza, all'instabilità del sistema.

In entrambe gli scenari, gli sviluppatori hanno ritenuto che queste importanti modifiche erano necessarie, e quindi le hanno fatte con qualche sofferenza. Se Linux avesse assicurato di mantenere stabile l'interfaccia interna, si sarebbe dovuto procedere alla creazione di una nuova, e quelle vecchie, e mal funzionanti, avrebbero dovuto ricevere manutenzione, creando lavoro aggiuntivo per gli sviluppatori del sottosistema USB. Dato che gli sviluppatori devono dedicare il proprio tempo a questo genere di lavoro, chiedergli di dedicarne dell'altro, senza benefici, magari gratuitamente, non è contemplabile.

Le problematiche relative alla sicurezza sono molto importanti per Linux. Quando viene trovato un problema di sicurezza viene corretto in breve tempo. A volte, per prevenire il problema di sicurezza, si sono dovute cambiare delle interfacce interne al kernel. Quando è successo, allo stesso tempo, tutti i driver che usavano quelle interfacce sono stati aggiornati, garantendo la correzione definitiva del problema senza doversi preoccupare di rivederlo per sbaglio in futuro. Se non si fossero

cambiate le interfacce interne, sarebbe stato impossibile correggere il problema e garantire che non si sarebbe più ripetuto.

Nel tempo le interfacce del kernel subiscono qualche ripulita. Se nessuno sta più usando un' interfaccia, allora questa verrà rimossa. Questo permette al kernel di rimanere il più piccolo possibile, e garantisce che tutte le potenziali interfacce sono state verificate nel limite del possibile (le interfacce inutilizzate sono impossibili da verificare).

Cosa fare

Dunque, se avete un driver per il kernel Linux che non si trova nei sorgenti principali del kernel, come sviluppatori, cosa dovreste fare? Rilasciare un file binario del driver per ogni versione del kernel e per ogni distribuzione, è un incubo; inoltre, tenere il passo con tutti i cambiamenti del kernel è un brutto lavoro.

Semplicemente, fate sì che il vostro driver per il kernel venga incluso nei sorgenti principali (ricordatevi, stiamo parlando di driver rilasciati secondo una licenza compatibile con la GPL; se il vostro codice non ricade in questa categoria: buona fortuna, arrangiatevi, siete delle sanguisughe)

Se il vostro driver è nei sorgenti del kernel e un' interfaccia cambia, il driver verrà corretto immediatamente dalla persona che l' ha modificata. Questo garantisce che sia sempre possibile compilare il driver, che funzioni, e tutto con un minimo sforzo da parte vostra.

Avere il proprio driver nei sorgenti principali del kernel ha i seguenti vantaggi:

- La qualità del driver aumenterà e i costi di manutenzione (per lo sviluppatore originale) diminuiranno.
- Altri sviluppatori aggiungeranno nuove funzionalità al vostro driver.
- Altri persone troveranno e correggeranno bachi nel vostro driver.
- Altri persone troveranno degli aggiustamenti da fare al vostro driver.
- Altri persone aggiorneranno il driver quando è richiesto da un cambiamento di un' interfaccia.
- Il driver sarà automaticamente reso disponibile in tutte le distribuzioni Linux senza dover chiedere a nessuna di queste di aggiungerlo.

Dato che Linux supporta più dispositivi di qualsiasi altro sistema operativo, e che girano su molti più tipi di processori di qualsiasi altro sistema operativo; ciò dimostra che questo modello di sviluppo qualcosa di giusto, dopo tutto, lo fa :)

Dei ringraziamenti vanno a Randy Dunlap, Andrew Morton, David Brownell, Hanna Linder, Robert Love, e Nishanth Aravamudan per la loro revisione e per i loro commenti sulle prime bozze di questo articolo.

Original
../../../process/management-style

Translator
Alessia Mantegazza <amantegazza@vaga.pv.it>

Il modello di gestione del kernel Linux

Questo breve documento descrive il modello di gestione del kernel Linux. Per certi versi, esso rispecchia il documento *translations/it_IT/process/coding-style.rst*, ed è principalmente scritto per evitare di rispondere[1] in continuazione alle stesse identiche (o quasi) domande.

Il modello di gestione è qualcosa di molto personale e molto più difficile da qualificare rispetto a delle semplici regole di codifica, quindi questo documento potrebbe avere più o meno a che fare con la realtà. È cominciato come un gioco, ma ciò non significa che non possa essere vero. Lo dovrete decidere voi stessi.

In ogni caso, quando si parla del "dirigente del kernel", ci si riferisce sempre alla persona che dirige tecnicamente, e non a coloro che tradizionalmente hanno un ruolo direttivo all'interno delle aziende. Se vi occupate di convalidare acquisti o avete una qualche idea sul budget del vostro gruppo, probabilmente non siete un dirigente del kernel. Quindi i suggerimenti qui indicati potrebbero fare al caso vostro, oppure no.

Prima di tutto, suggerirei di acquistare "Le sette regole per avere successo", e di non leggerlo. Bruciatelo, è un grande gesto simbolico.

Comunque, partiamo:

1) Le decisioni

Tutti pensano che i dirigenti decidano, e che questo prendere decisioni sia importante. Più grande e dolorosa è la decisione, più importante deve essere il dirigente che la prende. Questo è molto profondo ed ovvio, ma non è del tutto vero.

Il gioco consiste nell'"evitare" di dover prendere decisioni. In particolare se qualcuno vi chiede di "Decidere"tra (a) o (b), e vi dice che ha davvero bisogno di voi per questo, come dirigenti siete nei guai. Le persone che gestite devono conoscere i dettagli più di quanto li conosciate voi, quindi se vengono da voi per una decisione tecnica, siete fottuti. Non sarete chiaramente competente per prendere quella decisione per loro.

(Corollario: se le persone che gestite non conoscono i dettagli meglio di voi, anche in questo caso sarete fregati, tuttavia per altre ragioni. Ossia state facendo il lavoro sbagliato, e che invece dovrebbero essere "loro" a gestirvi)

Quindi il gioco si chiama "evitare" decisioni, almeno le più grandi e difficili. Prendere decisioni piccoli e senza conseguenze va bene, e vi fa sembrare competenti in quello che state facendo, quindi quello che un dirigente del kernel ha bisogno

[1] Questo documento non fa molto per risponde alla domanda, ma rende così dannatamente ovvio a chi la pone che non abbiamo la minima idea di come rispondere.

di fare è trasformare le decisioni grandi e difficili in minuzie delle quali nessuno importa.

Ciò aiuta a capire che la differenza chiave tra una grande decisione ed una piccola sta nella possibilità di modificare tale decisione in seguito. Qualsiasi decisione importante può essere ridotta in decisioni meno importanti, ma dovete assicurarvi che possano essere reversibili in caso di errori (presenti o futuri). Improvvisamente, dovrete essere doppiamente dirigenti per **due** decisioni non sequenziali - quella sbagliata **e** quella giusta.

E le persone vedranno tutto ciò come prova di vera capacità di comando (*cough* cavolata *cough*)

Così la chiave per evitare le decisioni difficili diviene l'evitare di fare cose che non possono essere disfatte. Non infilatevi in un angolo dal quale non potrete sfuggire. Un topo messo all'angolo può rivelarsi pericoloso - un dirigente messo all'angolo è solo pietoso.

In ogni caso dato che nessuno è stupido al punto da lasciare veramente ad un dirigente del kernel un enorme responsabilità, solitamente è facile fare marcia indietro. Annullare una decisione è molto facile: semplicemente dite a tutti che siete stati degli scemi incompetenti, dite che siete dispiaciuti, ed annullate tutto l'inutile lavoro sul quale gli altri hanno lavorato nell'ultimo anno. Improvvisamente la decisione che avevate preso un anno fa non era poi così grossa, dato che può essere facilmente annullata.

È emerso che alcune persone hanno dei problemi con questo tipo di approccio, questo per due ragioni:

- ammettere di essere degli idioti è più difficile di quanto sembri. A tutti noi piace mantenere le apparenze, ed uscire allo scoperto in pubblico per ammettere che ci si è sbagliati è qualcosa di davvero impegnativo.
- avere qualcuno che ti dice che ciò su cui hai lavorato nell'ultimo anno non era del tutto valido, può rivelarsi difficile anche per un povero ed umile ingegnere, e mentre il **lavoro** vero era abbastanza facile da cancellare, dall'altro canto potreste aver irrimediabilmente perso la fiducia di quell'ingegnere. E ricordate che l'"irrevocabile" era quello che avevamo cercato di evitare fin dall'inizio, e la vostra decisione ha finito per esserlo.

Fortunatamente, entrambe queste ragioni posso essere mitigate semplicemente ammettendo fin dal principio che non avete una cavolo di idea, dicendo agli altri in anticipo che la vostra decisione è puramente ipotetica, e che potrebbe essere sbagliata. Dovreste sempre riservarvi il diritto di cambiare la vostra opinione, e rendere gli altri ben **consapevoli** di ciò. Ed è molto più facile ammettere di essere stupidi quando non avete **ancora** fatto quella cosa stupida.

Poi, quando è realmente emersa la vostra stupidità, le persone semplicemente roteeranno gli occhi e diranno "Uffa, no, ancora".

Questa ammissione preventiva di incompetenza potrebbe anche portare le persone che stanno facendo il vero lavoro, a pensarci due volte. Dopo tutto, se **loro** non sono certi se sia una buona idea, voi, sicuro come la morte, non dovreste incoraggiarli promettendogli che ciò su cui stanno lavorando verrà incluso. Fate si che ci pensino due volte prima che si imbarchino in un grosso lavoro.

Ricordate: loro devono sapere più cose sui dettagli rispetto a voi, e solitamente

pensano di avere già la risposta a tutto. La miglior cosa che potete fare in qualità di dirigente è di non instillare troppa fiducia, ma invece fornire una salutare dose di pensiero critico su quanto stanno facendo.

Comunque, un altro modo di evitare una decisione è quello di lamentarsi malinconicamente dicendo : "non possiamo farli entrambi e basta?" e con uno sguardo pietoso. Fidatevi, funziona. Se non è chiaro quale sia il miglior approccio, lo scopriranno. La risposta potrebbe essere data dal fatto che entrambe i gruppi di lavoro diventano frustati al punto di rinunciarvi.

Questo può suonare come un fallimento, ma di solito questo è un segno che c' era qualcosa che non andava in entrambe i progetti, e il motivo per il quale le persone coinvolte non abbiano potuto decidere era che entrambe sbagliavano. Voi ne uscirete freschi come una rosa, e avrete evitato un' altra decisione con la quale avreste potuto fregarvi.

2) Le persone

Ci sono molte persone stupide, ed essere un dirigente significa che dovrete scendere a patti con questo, e molto più importate, che **loro** devono avere a che fare con **voi**.

Ne emerge che mentre è facile annullare degli errori tecnici, non è invece così facile rimuovere i disordini della personalità. Dovrete semplicemente convivere con i loro, ed i vostri, problemi.

Comunque, al fine di preparavi in qualità di dirigenti del kernel, è meglio ricordare di non abbattere alcun ponte, bombardare alcun paesano innocente, o escludere troppi sviluppatori kernel. Ne emerge che escludere le persone è piuttosto facile, mentre includerle nuovamente è difficile. Così "l' esclusione" immediatamente cade sotto il titolo di "non reversibile", e diviene un no-no secondo la sezione *1) Le decisioni*.

Esistono alcune semplici regole qui:

(1) non chiamate le persone teste di c*** (al meno, non in pubblico)

(2) imparate a scusarvi quando dimenticate la regola (1)

Il problema del punto numero 1 è che è molto facile da rispettare, dato che è possibile dire "sei una testa di c***" in milioni di modi differenti[2], a volte senza nemmeno pensarci, e praticamente sempre con la calda convinzione di essere nel giusto.

E più convinti sarete che avete ragione (e diciamolo, potete chiamare praticamente **tutti** testa di c**, e spesso **sarete** nel giusto), più difficile sarà scusarvi successivamente.

Per risolvere questo problema, avete due possibilità:

- diventare davvero bravi nello scusarsi

[2] Paul Simon cantava: "50 modi per lasciare il vostro amante", perché, molto francamente, "Un milione di modi per dire ad uno sviluppatore Testa di c***" non avrebbe funzionato. Ma sono sicuro che ci abbia pensato.

- essere amabili così che nessuno finirà col sentirsi preso di mira. Siate creativi abbastanza, e potrebbero esserne divertiti.

L'opzione dell'essere immancabilmente educati non esiste proprio. Nessuno si fiderà di qualcuno che chiaramente sta nascondendo il suo vero carattere.

3) Le persone II - quelle buone

Mentre emerge che la maggior parte delle persone sono stupide, il corollario a questo è il triste fatto che anche voi siete fra queste, e che mentre possiamo tutti crogiolarci nella sicurezza di essere migliori della media delle persone (diciamocelo, nessuno crede di essere nelle media o sotto di essa), dovremmo anche ammettere che non siamo il "coltello più affilato" del circondario, e che ci saranno altre persone che sono meno stupide di quanto lo siete voi.

Molti reagiscono male davanti alle persone intelligenti. Altri le usano a proprio vantaggio.

Assicuratevi che voi, in qualità di manutentori del kernel, siate nel secondo gruppo. Inchinatevi dinanzi a loro perché saranno le persone che vi renderanno il lavoro più facile. In particolare, prenderanno le decisioni per voi, che è l'oggetto di questo gioco.

Quindi quando trovate qualcuno più sveglio di voi, prendetevela comoda. Le vostre responsabilità dirigenziali si ridurranno in gran parte nel dire "Sembra una buona idea - Vai", oppure "Sembra buono, ma invece circa questo e quello?". La seconda versione in particolare è una gran modo per imparare qualcosa di nuovo circa "questo e quello"o di sembrare **extra** dirigenziali sottolineando qualcosa alla quale i più svegli non avevano pensato. In entrambe i casi, vincete.

Una cosa alla quale dovete fare attenzione è che l'essere grandi in qualcosa non si traduce automaticamente nell'essere grandi anche in altre cose. Quindi dovreste dare una spintarella alle persone in una specifica direzione, ma diciamocelo, potrebbero essere bravi in ciò che fanno e far schifo in tutto il resto. La buona notizia è che le persone tendono a gravitare attorno a ciò in cui sono bravi, quindi non state facendo nulla di irreversibile quando li spingete verso una certa direzione, solo non spingete troppo.

4) Addossare le colpe

Le cose andranno male, e le persone vogliono qualcuno da incolpare. Sarete voi.

Non è poi così difficile accettare la colpa, specialmente se le persone riescono a capire che non era **tutta** colpa vostra. Il che ci porta sulla miglior strada per assumersi la colpa: fatelo per qualcun'altro. Vi sentirete bene nel assumervi la responsabilità, e loro si sentiranno bene nel non essere incolpati, e coloro che hanno perso i loro 36GB di pornografia a causa della vostra incompetenza ammetteranno a malincuore che almeno non avete cercato di fare il furbetto.

Successivamente fate in modo che gli sviluppatori che in realtà hanno fallito (se riuscite a trovarli) sappiano **in privato** che sono "fottuti". Questo non per fargli sapere che la prossima volta possono evitarselo ma per fargli capire che sono in

debito. E, forse cosa più importante, sono loro che devono sistemare la cosa. Perché, ammettiamolo, è sicuro non sarete voi a farlo.

Assumersi la colpa è anche ciò che vi rendere dirigenti in prima battuta. È parte di ciò che spinge gli altri a fidarsi di voi, e vi garantisce la gloria potenziale, perché siete gli unici a dire "Ho fatto una cavolata". E se avete seguito le regole precedenti, sarete decisamente bravi nel dirlo.

5) Le cose da evitare

Esiste una cosa che le persone odiano più che essere chiamate "teste di c****", ed è essere chiamate "teste di c****" con fare da bigotto. Se per il primo caso potrete comunque scusarvi, per il secondo non ve ne verrà data nemmeno l'opportunità. Probabilmente smetteranno di ascoltarvi anche se tutto sommato state svolgendo un buon lavoro.

Tutti crediamo di essere migliori degli altri, il che significa che quando qualcuno inizia a darsi delle arie, ci da **davvero** fastidio. Potreste anche essere moralmente ed intellettualmente superiore a tutti quelli attorno a voi, ma non cercate di renderlo ovvio per gli altri a meno che non **vogliate** veramente far arrabbiare qualcuno[3].

Allo stesso modo evitate di essere troppo gentili e pacati. Le buone maniere facilmente finiscono per strabordare e nascondere i problemi, e come si usa dire, "su internet nessuno può sentire la vostra pacatezza". Usate argomenti diretti per farvi capire, non potete sperare che la gente capisca in altro modo.

Un po' di umorismo può aiutare a smorzare sia la franchezza che la moralità. Andare oltre i limiti al punto d'essere ridicolo può portare dei punti a casa senza renderlo spiacevole per i riceventi, i quali penseranno che stavate facendo gli scemi. Può anche aiutare a lasciare andare quei blocchi mentali che abbiamo nei confronti delle critiche.

6) Perché io?

Dato che la vostra responsabilità principale è quella di prendervi le colpe d'altri, e rendere dolorosamente ovvio a tutti che siete degli incompetenti, la domanda naturale che ne segue sarà : perché dovrei fare tutto ciò?

Innanzitutto, potreste diventare o no popolari al punto da avere la fila di ragazzine (o ragazzini, evitiamo pregiudizi o sessismo) che gridano e bussano alla porta del vostro camerino, ma comunque **proverete** un immenso senso di realizzazione personale dall'essere "in carica". Dimenticate il fatto che voi state discutendo con tutti e che cercate di inseguirli il più velocemente che potete. Tutti continueranno a pensare che voi siete la persona in carica.

È un bel lavoro se riuscite ad adattarlo a voi.

[3] Suggerimento: i forum di discussione su internet, che non sono collegati col vostro lavoro, sono ottimi modi per sfogare la frustrazione verso altre persone. Di tanto in tanto scrivete messaggi offensivi col ghigno in faccia per infiammare qualche discussione: vi sentirete purificati. Solo cercate di non cagare troppo vicino a casa.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/stable-kernel-rules.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Tutto quello che volevate sapere sui rilasci -stable di Linux

Regole sul tipo di patch che vengono o non vengono accettate nei sorgenti "-stable" :

- Ovviamente dev'essere corretta e verificata.
- Non dev'essere più grande di 100 righe, incluso il contesto.
- Deve correggere una cosa sola.
- Deve correggere un baco vero che sta disturbando gli utenti (non cose del tipo "Questo potrebbe essere un problema …").
- Deve correggere un problema di compilazione (ma non per cose già segnate con CONFIG_BROKEN), un kernel oops, un blocco, una corruzione di dati, un vero problema di sicurezza, o problemi del tipo "oh, questo non va bene" . In pratica, qualcosa di critico.
- Problemi importanti riportati dagli utenti di una distribuzione potrebbero essere considerati se correggono importanti problemi di prestazioni o di interattività. Dato che questi problemi non sono così ovvi e la loro correzione ha un'alta probabilità d'introdurre una regressione, dovrebbero essere sottomessi solo dal manutentore della distribuzione includendo un link, se esiste, ad un rapporto su bugzilla, e informazioni aggiuntive sull'impatto che ha sugli utenti.
- Non deve correggere problemi relativi a una "teorica sezione critica", a meno che non venga fornita anche una spiegazione su come questa si possa verificare.
- Non deve includere alcuna correzione "banale" (correzioni grammaticali, pulizia dagli spazi bianchi, eccetera).
- Deve rispettare le regole scritte in Inviare patch: la guida essenziale per vedere il vostro codice nel kernel
- Questa patch o una equivalente deve esistere già nei sorgenti principali di Linux

Procedura per sottomettere patch per i sorgenti -stable

- Se la patch contiene modifiche a dei file nelle cartelle net/ o drivers/net, allora seguite le linee guida descritte in Documentation/translations/it_IT/networking/netdev-FAQ.rst; ma solo dopo aver verificato al seguente indirizzo che la patch non sia già in coda: https://patchwork.kernel.org/bundle/netdev/stable/?state=*
- Una patch di sicurezza non dovrebbero essere gestite (solamente) dal processo di revisione -stable, ma dovrebbe seguire le procedure descritte in Documentation/translations/it_IT/admin-guide/security-bugs.rst.

Per tutte le altre sottomissioni, scegliere una delle seguenti procedure

Opzione 1

Per far sì che una patch venga automaticamente inclusa nei sorgenti stabili, aggiungete l'etichetta

```
Cc: stable@vger.kernel.org
```

nell'area dedicata alla firme. Una volta che la patch è stata inclusa, verrà applicata anche sui sorgenti stabili senza che l'autore o il manutentore del sottosistema debba fare qualcosa.

Opzione 2

Dopo che la patch è stata inclusa nei sorgenti Linux, inviate una mail a stable@vger.kernel.org includendo: il titolo della patch, l'identificativo del commit, il perché pensate che debba essere applicata, e in quale versione del kernel la vorreste vedere.

Opzione 3

Inviata la patch, dopo aver verificato che rispetta le regole descritte in precedenza, a stable@vger.kernel.org. Dovete annotare nel changelog l'identificativo del commit nei sorgenti principali, così come la versione del kernel nel quale vorreste vedere la patch.

L'*Opzione 1* è fortemente raccomandata; è il modo più facile e usato. L'*Opzione 2* e l'*Opzione 3* sono più utili quando, al momento dell'inclusione dei sorgenti principali, si ritiene che non debbano essere incluse anche in quelli stabili (per esempio, perché si crede che si dovrebbero fare più verifiche per eventuali regressioni). L'*Opzione 3* è particolarmente utile se la patch ha bisogno di qualche modifica per essere applicata ad un kernel più vecchio (per esempio, perché nel frattempo l'API è cambiata).

Notate che per l'*Opzione 3*, se la patch è diversa da quella nei sorgenti principali (per esempio perché è stato necessario un lavoro di adattamento) allora dev'essere ben documentata e giustificata nella descrizione della patch.

L'identificativo del commit nei sorgenti principali dev'essere indicato sopra al messaggio della patch, così:

```
commit <sha1> upstream.
```

In aggiunta, alcune patch inviate attraverso l'*Opzione 1* potrebbero dipendere da altre che devo essere incluse. Questa situazione può essere indicata nel seguente modo nell'area dedicata alle firme:

```
Cc: <stable@vger.kernel.org> # 3.3.x: a1f84a3: sched: Check for idle
Cc: <stable@vger.kernel.org> # 3.3.x: 1b9508f: sched: Rate-limit␣
↪newidle
Cc: <stable@vger.kernel.org> # 3.3.x: fd21073: sched: Fix affinity␣
↪logic
Cc: <stable@vger.kernel.org> # 3.3.x
Signed-off-by: Ingo Molnar <mingo@elte.hu>
```

La sequenza di etichette ha il seguente significato:

```
git cherry-pick a1f84a3
git cherry-pick 1b9508f
git cherry-pick fd21073
git cherry-pick <this commit>
```

Inoltre, alcune patch potrebbero avere dei requisiti circa la versione del kernel. Questo può essere indicato usando il seguente formato nell'area dedicata alle firme:

```
Cc: <stable@vger.kernel.org> # 3.3.x
```

L'etichetta ha il seguente significato:

```
git cherry-pick <this commit>
```

per ogni sorgente "-stable" che inizia con la versione indicata.

Dopo la sottomissione:

- Il mittente riceverà un ACK quando la patch è stata accettata e messa in coda, oppure un NAK se la patch è stata rigettata. A seconda degli impegni degli sviluppatori, questa risposta potrebbe richiedere alcuni giorni.
- Se accettata, la patch verrà aggiunta alla coda -stable per essere revisionata dal altri sviluppatori e dal principale manutentore del sottosistema.

Ciclo di una revisione

- Quando i manutentori -stable decidono di fare un ciclo di revisione, le patch vengono mandate al comitato per la revisione, ai manutentori soggetti alle modifiche delle patch (a meno che il mittente non sia anche il manutentore di quell' area del kernel) e in CC: alla lista di discussione linux-kernel.
- La commissione per la revisione ha 48 ore per dare il proprio ACK o NACK alle patch.
- Se una patch viene rigettata da un membro della commissione, o un membro della lista linux-kernel obietta la bontà della patch, sollevando problemi che i manutentori ed i membri non avevano compreso, allora la patch verrà rimossa dalla coda.
- Alla fine del ciclo di revisione tutte le patch che hanno ricevuto l'ACK verranno aggiunte per il prossimo rilascio -stable, e successivamente questo nuovo rilascio verrà fatto.
- Le patch di sicurezza verranno accettate nei sorgenti -stable direttamente dalla squadra per la sicurezza del kernel, e non passerà per il normale ciclo di revisione. Contattate la suddetta squadra per maggiori dettagli su questa procedura.

Sorgenti

- La coda delle patch, sia quelle già applicate che in fase di revisione, possono essere trovate al seguente indirizzo:

  https://git.kernel.org/pub/scm/linux/kernel/git/stable/stable-queue.git

- Il rilascio definitivo, e marchiato, di tutti i kernel stabili può essere trovato in rami distinti per versione al seguente indirizzo:

  https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

Comitato per la revisione

- Questo comitato è fatto di sviluppatori del kernel che si sono offerti volontari per questo lavoro, e pochi altri che non sono proprio volontari.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/submit-checklist.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Lista delle verifiche da fare prima di inviare una patch per il kernel Linux

Qui troverete una lista di cose che uno sviluppatore dovrebbe fare per vedere le proprie patch accettate più rapidamente.

Tutti questi punti integrano la documentazione fornita riguardo alla sottomissione delle patch, in particolare Inviare patch: la guida essenziale per vedere il vostro codice nel kernel.

1) Se state usando delle funzionalità del kernel allora includete (#include) i file che le dichiarano/definiscono. Non dipendente dal fatto che un file d' intestazione include anche quelli usati da voi.

2) Compilazione pulita:

a) con le opzioni `CONFIG` negli stati `=y`, `=m` e `=n`. Nessun avviso/errore di `gcc` e nessun avviso/errore dal linker.

b) con `allnoconfig`, `allmodconfig`

c) quando si usa `O=builddir`

3) Compilare per diverse architetture di processore usando strumenti per la cross-compilazione o altri.

4) Una buona architettura per la verifica della cross-compilazione è la ppc64 perché tende ad usare `unsigned long` per le quantità a 64-bit.

5) Controllate lo stile del codice della vostra patch secondo le direttive scritte in Stile del codice per il kernel Linux. Prima dell' invio della patch, usate il verificatore di stile (`script/checkpatch.pl`) per scovare le violazioni più semplici. Dovreste essere in grado di giustificare tutte le violazioni rimanenti nella vostra patch.

6) Le opzioni `CONFIG`, nuove o modificate, non scombussolano il menu di configurazione e sono preimpostate come disabilitate a meno che non soddisfino i criteri descritti in `Documentation/kbuild/kconfig-language.rst` alla punto "Voci di menu: valori predefiniti".

7) Tutte le nuove opzioni `Kconfig` hanno un messaggio di aiuto.

8) La patch è stata accuratamente revisionata rispetto alle più importanti configurazioni `Kconfig`. Questo è molto difficile da fare correttamente - un buono lavoro di testa sarà utile.

9) Verificare con sparse.

10) Usare `make checkstack` e `make namespacecheck` e correggere tutti i problemi rilevati.

Note: `checkstack` non evidenzia esplicitamente i problemi, ma una funzione che usa più di 512 byte sullo stack è una buona candidata per una correzione.

---

11) Includete commenti kernel-doc per documentare API globali del kernel. Usate `make htmldocs` o `make pdfdocs` per verificare i commenti kernel-doc ed eventualmente correggerli.

12) La patch è stata verificata con le seguenti opzioni abilitate contemporaneamente: CONFIG_PREEMPT, CONFIG_DEBUG_PREEMPT, CONFIG_DEBUG_SLAB, CONFIG_DEBUG_PAGEALLOC, CONFIG_DEBUG_MUTEXES, CONFIG_DEBUG_SPINLOCK, CONFIG_DEBUG_ATOMIC_SLEEP, CONFIG_PROVE_RCU e CONFIG_DEBUG_OBJECTS_RCU_HEAD.

13) La patch è stata compilata e verificata in esecuzione con, e senza, le opzioni CONFIG_SMP e CONFIG_PREEMPT.

14) Se la patch ha effetti sull' IO dei dischi, eccetera: allora dev' essere verificata con, e senza, l' opzione CONFIG_LBDAF.

15) Tutti i percorsi del codice sono stati verificati con tutte le funzionalità di lockdep abilitate.

16) Tutti i nuovi elementi in /proc sono documentati in Documentation/.

17) Tutti i nuovi parametri d' avvio del kernel sono documentati in Documentation/admin-guide/kernel-parameters.rst.

18) Tutti i nuovi parametri dei moduli sono documentati con MODULE_PARM_DESC().

19) Tutte le nuove interfacce verso lo spazio utente sono documentate in Documentation/ABI/. Leggete Documentation/ABI/README per maggiori informazioni. Le patch che modificano le interfacce utente dovrebbero essere inviate in copia anche a linux-api@vger.kernel.org.

20) Verifica che il kernel passi con successo make headers_check

21) La patch è stata verificata con l'iniezione di fallimenti in slab e nell'allocazione di pagine. Vedere Documentation/fault-injection/.

    Se il nuovo codice è corposo, potrebbe essere opportuno aggiungere l' iniezione di fallimenti specifici per il sottosistema.

22) Il nuovo codice è stato compilato con gcc -W (usate make EXTRA_CFLAGS=-W). Questo genererà molti avvisi, ma è ottimo per scovare bachi come "warning: comparison between signed and unsigned".

23) La patch è stata verificata dopo essere stata inclusa nella serie di patch -mm; questo al fine di assicurarsi che continui a funzionare assieme a tutte le altre patch in coda e i vari cambiamenti nei sottosistemi VM, VFS e altri.

24) Tutte le barriere di sincronizzazione {per esempio, barrier(), rmb(), wmb()} devono essere accompagnate da un commento nei sorgenti che ne spieghi la logica: cosa fanno e perché.

25) Se la patch aggiunge nuove chiamate ioctl, allora aggiornate Documentation/userspace-api/ioctl/ioctl-number.rst.

26) Se il codice che avete modificato dipende o usa una qualsiasi interfaccia o funzionalità del kernel che è associata a uno dei seguenti simboli Kconfig, allora verificate che il kernel compili con diverse configurazioni dove i simboli sono disabilitati e/o =m (se c' è la possibilità) [non tutti contemporaneamente, solo diverse combinazioni casuali]:

    CONFIG_SMP, CONFIG_SYSFS, CONFIG_PROC_FS, CONFIG_INPUT, CONFIG_PCI, CONFIG_BLOCK, CONFIG_PM, CONFIG_MAGIC_SYSRQ, CONFIG_NET,

CONFIG_INET=n (ma l'ultimo con CONFIG_NET=y).

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/kernel-docs.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Indice di documenti per le persone interessate a capire e/o scrivere per il kernel Linux

---

Note: Questo documento contiene riferimenti a documenti in lingua inglese; inoltre utilizza dai campi *ReStructuredText* di supporto alla ricerca e che per questo motivo è meglio non tradurre al fine di garantirne un corretto utilizzo. Per questi motivi il documento non verrà tradotto. Per favore fate riferimento al documento originale in lingua inglese.

Ed infine, qui ci sono alcune guide più tecniche che son state messe qua solo perché non si è trovato un posto migliore.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/applying-patches.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Applicare patch al kernel Linux

Note: Questo documento è obsoleto. Nella maggior parte dei casi, piuttosto che usare `patch` manualmente, vorrete usare Git. Per questo motivo il documento non verrà tradotto.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/adding-syscalls.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Aggiungere una nuova chiamata di sistema

Questo documento descrive quello che è necessario sapere per aggiungere nuove chiamate di sistema al kernel Linux; questo è da considerarsi come un' aggiunta ai soliti consigli su come proporre nuove modifiche Inviare patch: la guida essenziale per vedere il vostro codice nel kernel.

Alternative alle chiamate di sistema

La prima considerazione da fare quando si aggiunge una nuova chiamata di sistema è quella di valutare le alternative. Nonostante le chiamate di sistema siano il punto di interazione fra spazio utente e kernel più tradizionale ed ovvio, esistono altre possibilità - scegliete quella che meglio si adatta alle vostra interfaccia.

- Se le operazioni coinvolte possono rassomigliare a quelle di un filesystem, allora potrebbe avere molto più senso la creazione di un nuovo filesystem o dispositivo. Inoltre, questo rende più facile incapsulare la nuova funzionalità in un modulo kernel piuttosto che essere sviluppata nel cuore del kernel.
  - Se la nuova funzionalità prevede operazioni dove il kernel notifica lo spazio utente su un avvenimento, allora restituire un descrittore di file all' oggetto corrispondente permette allo spazio utente di utilizzare `poll`/`select`/`epoll` per ricevere quelle notifiche.
  - Tuttavia, le operazioni che non si sposano bene con operazioni tipo *`read(2)`*/*`write(2)`* dovrebbero essere implementate come chiamate *`ioctl(2)`*, il che potrebbe portare ad un' API in un qualche modo opaca.
- Se dovete esporre solo delle informazioni sul sistema, un nuovo nodo in sysfs (vedere `Documentation/filesystems/sysfs.rst`) o in procfs potrebbe essere sufficiente. Tuttavia, l' accesso a questi meccanismi richiede che il filesystem sia montato, il che potrebbe non essere sempre vero (per esempio, in ambienti come namespace/sandbox/chroot). Evitate d' aggiungere nuove API in debugfs perché questo non viene considerata un' interfaccia di 'produzione' verso lo spazio utente.
- Se l' operazione è specifica ad un particolare file o descrittore, allora potrebbe essere appropriata l' aggiunta di un comando *`fcntl(2)`*. Tuttavia, *`fcntl(2)`* è una chiamata di sistema multiplatrice che nasconde una notevole complessità, quindi è ottima solo quando la nuova funzione assomiglia a quelle già esistenti in *`fcntl(2)`*, oppure la nuova funzionalità è veramente semplice (per esempio, leggere/scrivere un semplice flag associato ad un descrittore di file).
- Se l' operazione è specifica ad un particolare processo, allora potrebbe essere appropriata l' aggiunta di un comando *`prctl(2)`*. Come per *`fcntl(2)`*,

questa chiamata di sistema è un complesso multiplatore quindi è meglio usarlo per cose molto simili a quelle esistenti nel comando `prctl` oppure per leggere/scrivere un semplice flag relativo al processo.

Progettare l'API: pianificare le estensioni

Una nuova chiamata di sistema diventerà parte dell'API del kernel, e dev'essere supportata per un periodo indefinito. Per questo, è davvero un'ottima idea quella di discutere apertamente l'interfaccia sulla lista di discussione del kernel, ed è altrettanto importante pianificarne eventuali estensioni future.

(Nella tabella delle chiamate di sistema sono disseminati esempi dove questo non fu fatto, assieme ai corrispondenti aggiornamenti - `eventfd/eventfd2`, `dup2/dup3`, `inotify_init/inotify_init1`, `pipe/pipe2`, `renameat/renameat2` – quindi imparate dalla storia del kernel e pianificate le estensioni fin dall'inizio)

Per semplici chiamate di sistema che accettano solo un paio di argomenti, il modo migliore di permettere l'estensibilità è quello di includere un argomento *flags* alla chiamata di sistema. Per assicurarsi che i programmi dello spazio utente possano usare in sicurezza *flags* con diverse versioni del kernel, verificate se *flags* contiene un qualsiasi valore sconosciuto, in qual caso rifiutate la chiamata di sistema (con `EINVAL`):

```
if (flags & ~(THING_FLAG1 | THING_FLAG2 | THING_FLAG3))
    return -EINVAL;
```

(Se *flags* non viene ancora utilizzato, verificate che l'argomento sia zero)

Per chiamate di sistema più sofisticate che coinvolgono un numero più grande di argomenti, il modo migliore è quello di incapsularne la maggior parte in una struttura dati che verrà passata per puntatore. Questa struttura potrà funzionare con future estensioni includendo un campo *size*:

```
struct xyzzy_params {
    u32 size; /* userspace sets p->size = sizeof(struct xyzzy_
↪params) */
    u32 param_1;
    u64 param_2;
    u64 param_3;
};
```

Fintanto che un qualsiasi campo nuovo, diciamo `param_4`, è progettato per offrire il comportamento precedente quando vale zero, allora questo permetterà di gestire un conflitto di versione in entrambe le direzioni:

- un vecchio kernel può gestire l'accesso di una versione moderna di un programma in spazio utente verificando che la memoria oltre la dimensione della struttura dati attesa sia zero (in pratica verificare che `param_4 == 0`).
- un nuovo kernel può gestire l'accesso di una versione vecchia di un programma in spazio utente estendendo la struttura dati con zeri (in pratica `param_4 = 0`).

Vedere *perf_event_open(2)* e la funzione perf_copy_attr() (in kernel/events/core.c) per un esempio pratico di questo approccio.

Progettare l'API: altre considerazioni

Se la vostra nuova chiamata di sistema permette allo spazio utente di fare riferimento ad un oggetto del kernel, allora questa dovrebbe usare un descrittore di file per accesso all'oggetto - non inventatevi nuovi tipi di accesso da spazio utente quando il kernel ha già dei meccanismi e una semantica ben definita per utilizzare i descrittori di file.

Se la vostra nuova chiamata di sistema *xyzzy(2)* ritorna un nuovo descrittore di file, allora l'argomento *flags* dovrebbe includere un valore equivalente a O_CLOEXEC per i nuovi descrittori. Questo rende possibile, nello spazio utente, la chiusura della finestra temporale fra le chiamate a xyzzy() e fcntl(fd, F_SETFD, FD_CLOEXEC), dove un inaspettato fork() o execve() potrebbe trasferire il descrittore al programma eseguito (Comunque, resistete alla tentazione di riutilizzare il valore di O_CLOEXEC dato che è specifico dell'architettura e fa parte di una enumerazione di flag O_* che è abbastanza ricca).

Se la vostra nuova chiamata di sistema ritorna un nuovo descrittore di file, dovreste considerare che significato avrà l'uso delle chiamate di sistema della famiglia di *poll(2)*. Rendere un descrittore di file pronto per la lettura o la scrittura è il tipico modo del kernel per notificare lo spazio utente circa un evento associato all'oggetto del kernel.

Se la vostra nuova chiamata di sistema *xyzzy(2)* ha un argomento che è il percorso ad un file:

```
int sys_xyzzy(const char __user *path, ..., unsigned int flags);
```

dovreste anche considerare se non sia più appropriata una versione *xyzzyat(2)*:

```
int sys_xyzzyat(int dfd, const char __user *path, ..., unsigned int␣
↪flags);
```

Questo permette più flessibilità su come lo spazio utente specificherà il file in questione; in particolare, permette allo spazio utente di richiedere la funzionalità su un descrittore di file già aperto utilizzando il *flag* AT_EMPTY_PATH, in pratica otterremmo gratuitamente l'operazione *fxyzzy(3)*:

```
- xyzzyat(AT_FDCWD, path, ..., 0) is equivalent to xyzzy(path,...)
- xyzzyat(fd, "", ..., AT_EMPTY_PATH) is equivalent to fxyzzy(fd, ..
↪.)
```

(Per maggiori dettagli sulla logica delle chiamate *at(), leggete la pagina man *openat(2)*; per un esempio di AT_EMPTY_PATH, leggere la pagina man *fstatat(2)*).

Se la vostra nuova chiamata di sistema *xyzzy(2)* prevede un parametro per descrivere uno scostamento all'interno di un file, usate loff_t come tipo cosicché scostamenti a 64-bit potranno essere supportati anche su architetture a 32-bit.

Se la vostra nuova chiamata di sistema *xyzzy(2)* prevede l'uso di funzioni riservate, allora dev'essere gestita da un opportuno bit di privilegio (verificato con una chiamata a `capable()`), come descritto nella pagina man *capabilities(7)*. Scegliete un bit di privilegio già esistente per gestire la funzionalità associata, ma evitate la combinazione di diverse funzionalità vagamente collegate dietro lo stesso bit, in quanto va contro il principio di *capabilities* di separare i poteri di root. In particolare, evitate di aggiungere nuovi usi al fin-troppo-generico privilegio `CAP_SYS_ADMIN`.

Se la vostra nuova chiamata di sistema *xyzzy(2)* manipola altri processi oltre a quello chiamato, allora dovrebbe essere limitata (usando la chiamata `ptrace_may_access()`) di modo che solo un processo chiamante con gli stessi permessi del processo in oggetto, o con i necessari privilegi, possa manipolarlo.

Infine, state attenti che in alcune architetture non-x86 la vita delle chiamate di sistema con argomenti a 64-bit viene semplificata se questi argomenti ricadono in posizioni dispari (pratica, i parametri 1, 3, 5); questo permette l'uso di coppie contigue di registri a 32-bit. (Questo non conta se gli argomenti sono parte di una struttura dati che viene passata per puntatore).

Proporre l'API

Al fine di rendere le nuove chiamate di sistema di facile revisione, è meglio che dividiate le modifiche i pezzi separati. Questi dovrebbero includere almeno le seguenti voci in *commit* distinti (ognuno dei quali sarà descritto più avanti):

- l'essenza dell'implementazione della chiamata di sistema, con i prototipi, i numeri generici, le modifiche al Kconfig e l'implementazione *stub* di ripiego.
- preparare la nuova chiamata di sistema per un'architettura specifica, solitamente x86 (ovvero tutti: x86_64, x86_32 e x32).
- un programma di auto-verifica da mettere in `tools/testing/selftests/` che mostri l'uso della chiamata di sistema.
- una bozza di pagina man per la nuova chiamata di sistema. Può essere scritta nell'email di presentazione, oppure come modifica vera e propria al repositorio delle pagine man.

Le proposte di nuove chiamate di sistema, come ogni altro modifica all'API del kernel, deve essere sottomessa alla lista di discussione linux-api@vger.kernel.org.

Implementazione di chiamate di sistema generiche

Il principale punto d'accesso alla vostra nuova chiamata di sistema *xyzzy(2)* verrà chiamato `sys_xyzzy()`; ma, piuttosto che in modo esplicito, lo aggiungerete tramite la macro `SYSCALL_DEFINEn`. La 'n' indica il numero di argomenti della chiamata di sistema; la macro ha come argomento il nome della chiamata di sistema, seguito dalle coppie (tipo, nome) per definire i suoi parametri. L'uso di questa macro permette di avere i metadati della nuova chiamata di sistema disponibili anche per altri strumenti.

Il nuovo punto d'accesso necessita anche del suo prototipo di funzione in `include/linux/syscalls.h`, marcato come asmlinkage di modo da abbinargli il modo in cui quelle chiamate di sistema verranno invocate:

```
asmlinkage long sys_xyzzy(...);
```

Alcune architetture (per esempio x86) hanno le loro specifiche tabelle di chiamate di sistema (syscall), ma molte altre architetture condividono una tabella comune di syscall. Aggiungete alla lista generica la vostra nuova chiamata di sistema aggiungendo un nuovo elemento alla lista in `include/uapi/asm-generic/unistd.h`:

```
#define __NR_xyzzy 292
__SYSCALL(__NR_xyzzy, sys_xyzzy)
```

Aggiornate anche il contatore __NR_syscalls di modo che sia coerente con l'aggiunta della nuove chiamate di sistema; va notato che se più di una nuova chiamata di sistema viene aggiunga nella stessa finestra di sviluppo, il numero della vostra nuova syscall potrebbe essere aggiustato al fine di risolvere i conflitti.

Il file `kernel/sys_ni.c` fornisce le implementazioni *stub* di ripiego che ritornano `-ENOSYS`. Aggiungete la vostra nuova chiamata di sistema anche qui:

```
COND_SYSCALL(xyzzy);
```

La vostra nuova funzionalità del kernel, e la chiamata di sistema che la controlla, dovrebbero essere opzionali. Quindi, aggiungete un' opzione `CONFIG` (solitamente in `init/Kconfig`). Come al solito per le nuove opzioni `CONFIG`:

- Includete una descrizione della nuova funzionalità e della chiamata di sistema che la controlla.
- Rendete l'opzione dipendente da EXPERT se dev'essere nascosta agli utenti normali.
- Nel Makefile, rendere tutti i nuovi file sorgenti, che implementano la nuova funzionalità, dipendenti dall' opzione CONFIG (per esempio `obj-$(CONFIG_XYZZY_SYSCALL) += xyzzy.o`).
- Controllate due volte che sia possibile generare il kernel con la nuova opzione CONFIG disabilitata.

Per riassumere, vi serve un *commit* che includa:

- un' opzione `CONFIG``per la nuova funzione, normalmente in ``init/Kconfig`
- `SYSCALL_DEFINEn(xyzzy, ...)` per il punto d' accesso
- il corrispondente prototipo in `include/linux/syscalls.h`
- un elemento nella tabella generica in `include/uapi/asm-generic/unistd.h`
- *stub* di ripiego in `kernel/sys_ni.c`

Implementazione delle chiamate di sistema x86

Per collegare la vostra nuova chiamate di sistema alle piattaforme x86, dovete aggiornate la tabella principale di syscall. Assumendo che la vostra nuova chiamata di sistema non sia particolarmente speciale (vedere sotto), dovete aggiungere un elemento *common* (per x86_64 e x32) in arch/x86/entry/syscalls/syscall_64.tbl:

```
333   common   xyzzy     sys_xyzzy
```

e un elemento per *i386* `arch/x86/entry/syscalls/syscall_32.tbl`:

```
380   i386     xyzzy     sys_xyzzy
```

Ancora una volta, questi numeri potrebbero essere cambiati se generano conflitti durante la finestra di integrazione.

Chiamate di sistema compatibili (generico)

Per molte chiamate di sistema, la stessa implementazione a 64-bit può essere invocata anche quando il programma in spazio utente è a 32-bit; anche se la chiamata di sistema include esplicitamente un puntatore, questo viene gestito in modo trasparente.

Tuttavia, ci sono un paio di situazione dove diventa necessario avere un livello di gestione della compatibilità per risolvere le differenze di dimensioni fra 32-bit e 64-bit.

Il primo caso è quando un kernel a 64-bit supporta anche programmi in spazio utente a 32-bit, perciò dovrà ispezionare aree della memoria (`__user`) che potrebbero contenere valori a 32-bit o a 64-bit. In particolar modo, questo è necessario quando un argomento di una chiamata di sistema è:

- un puntatore ad un puntatore
- un puntatore ad una struttura dati contenente a sua volta un puntatore (ad esempio `struct iovec __user *`)
- un puntatore ad un tipo intero di dimensione variabile (`time_t`, `off_t`, `long`, …)
- un puntatore ad una struttura dati contenente un tipo intero di dimensione variabile.

Il secondo caso che richiede un livello di gestione della compatibilità è quando uno degli argomenti di una chiamata a sistema è esplicitamente un tipo a 64-bit anche su architetture a 32-bit, per esempio `loff_t` o `__u64`. In questo caso, un valore che arriva ad un kernel a 64-bit da un' applicazione a 32-bit verrà diviso in due valori a 32-bit che dovranno essere riassemblati in questo livello di compatibilità.

(Da notare che non serve questo livello di compatibilità per argomenti che sono puntatori ad un tipo esplicitamente a 64-bit; per esempio, in *`splice(2)`* l' argomento di tipo `loff_t __user *` non necessita di una chiamata di sistema `compat_`)

La versione compatibile della nostra chiamata di sistema si chiamerà `compat_sys_xyzzy()`, e viene aggiunta utilizzando la macro

COMPAT_SYSCALL_DEFINEn() (simile a SYSCALL_DEFINEn). Questa versione dell'implementazione è parte del kernel a 64-bit ma accetta parametri a 32-bit che trasformerà secondo le necessità (tipicamente, la versione compat_sys_ converte questi valori nello loro corrispondente a 64-bit e può chiamare la versione sys_ oppure invocare una funzione che implementa le parti comuni).

Il punto d'accesso *compat* deve avere il corrispondente prototipo di funzione in include/linux/compat.h, marcato come asmlinkage di modo da abbinargli il modo in cui quelle chiamate di sistema verranno invocate:

```
asmlinkage long compat_sys_xyzzy(...);
```

Se la chiamata di sistema prevede una struttura dati organizzata in modo diverso per sistemi a 32-bit e per quelli a 64-bit, diciamo struct xyzzy_args, allora il file d'intestazione then the include/linux/compat.h deve includere la sua versione *compatibile* (struct compat_xyzzy_args); ogni variabile con dimensione variabile deve avere il proprio tipo compat_ corrispondente a quello in struct xyzzy_args. La funzione compat_sys_xyzzy() può usare la struttura compat_ per analizzare gli argomenti ricevuti da una chiamata a 32-bit.

Per esempio, se avete i seguenti campi:

```
struct xyzzy_args {
    const char __user *ptr;
    __kernel_long_t varying_val;
    u64 fixed_val;
    /* ... */
};
```

nella struttura struct xyzzy_args, allora la struttura struct compat_xyzzy_args dovrebbe avere:

```
struct compat_xyzzy_args {
    compat_uptr_t ptr;
    compat_long_t varying_val;
    u64 fixed_val;
    /* ... */
};
```

La lista generica delle chiamate di sistema ha bisogno di essere aggiustata al fine di permettere l'uso della versione *compatibile*; la voce in include/uapi/asm-generic/unistd.h dovrebbero usare __SC_COMP piuttosto di __SYSCALL:

```
#define __NR_xyzzy 292
__SC_COMP(__NR_xyzzy, sys_xyzzy, compat_sys_xyzzy)
```

Riassumendo, vi serve:

- un COMPAT_SYSCALL_DEFINEn(xyzzy, ...) per il punto d'accesso *compatibile*
- un prototipo in include/linux/compat.h
- (se necessario) una struttura di compatibilità a 32-bit in include/linux/compat.h

- una voce `__SC_COMP`, e non `__SYSCALL`, in `include/uapi/asm-generic/unistd.h`

Compatibilità delle chiamate di sistema (x86)

Per collegare una chiamata di sistema, su un' architettura x86, con la sua versione *compatibile*, è necessario aggiustare la voce nella tabella delle syscall.

Per prima cosa, la voce in `arch/x86/entry/syscalls/syscall_32.tbl` prende un argomento aggiuntivo per indicare che un programma in spazio utente a 32-bit, eseguito su un kernel a 64-bit, dovrebbe accedere tramite il punto d' accesso compatibile:

```
380   i386    xyzzy    sys_xyzzy    __ia32_compat_sys_xyzzy
```

Secondo, dovete capire cosa dovrebbe succedere alla nuova chiamata di sistema per la versione dell' ABI x32. Qui C' è una scelta da fare: gli argomenti possono corrisponde alla versione a 64-bit o a quella a 32-bit.

Se c' è un puntatore ad un puntatore, la decisione è semplice: x32 è ILP32, quindi gli argomenti dovrebbero corrispondere a quelli a 32-bit, e la voce in `arch/x86/entry/syscalls/syscall_64.tbl` sarà divisa cosicché i programmi x32 eseguano la chiamata *compatibile*:

```
333   64      xyzzy    sys_xyzzy
...
555   x32     xyzzy    __x32_compat_sys_xyzzy
```

Se non ci sono puntatori, allora è preferibile riutilizzare la chiamata di sistema a 64-bit per l' ABI x32 (e di conseguenza la voce in arch/x86/entry/syscalls/syscall_64.tbl rimane immutata).

In ambo i casi, dovreste verificare che i tipi usati dagli argomenti abbiano un'esatta corrispondenza da x32 (-mx32) al loro equivalente a 32-bit (-m32) o 64-bit (-m64).

Chiamate di sistema che ritornano altrove

Nella maggior parte delle chiamate di sistema, al termine della loro esecuzione, i programmi in spazio utente riprendono esattamente dal punto in cui si erano interrotti – quindi dall' istruzione successiva, con lo stesso *stack* e con la maggior parte del registri com' erano stati lasciati prima della chiamata di sistema, e anche con la stessa memoria virtuale.

Tuttavia, alcune chiamata di sistema fanno le cose in modo differente. Potrebbero ritornare ad un punto diverso (`rt_sigreturn`) o cambiare la memoria in spazio utente (`fork`/`vfork`/`clone`) o perfino l' architettura del programma (`execve`/`execveat`).

Per permettere tutto ciò, l' implementazione nel kernel di questo tipo di chiamate di sistema potrebbero dover salvare e ripristinare registri aggiuntivi nello *stack* del kernel, permettendo così un controllo completo su dove e come l' esecuzione dovrà continuare dopo l' esecuzione della chiamata di sistema.

Queste saranno specifiche per ogni architettura, ma tipicamente si definiscono dei punti d'accesso in *assembly* per salvare/ripristinare i registri aggiuntivi e quindi chiamare il vero punto d'accesso per la chiamata di sistema.

Per l'architettura x86_64, questo è implementato come un punto d'accesso `stub_xyzzy` in `arch/x86/entry/entry_64.S`, e la voce nella tabella di syscall (`arch/x86/entry/syscalls/syscall_64.tbl`) verrà corretta di conseguenza:

```
333   common   xyzzy     stub_xyzzy
```

L'equivalente per programmi a 32-bit eseguiti su un kernel a 64-bit viene normalmente chiamato `stub32_xyzzy` e implementato in `arch/x86/entry/entry_64_compat.S` con la corrispondente voce nella tabella di syscall `arch/x86/entry/syscalls/syscall_32.tbl` corretta nel seguente modo:

```
380   i386     xyzzy     sys_xyzzy    stub32_xyzzy
```

Se una chiamata di sistema necessita di un livello di compatibilità (come nella sezione precedente), allora la versione `stub32_` deve invocare la versione `compat_sys_` piuttosto che quella nativa a 64-bit. In aggiunta, se l'implementazione dell'ABI x32 è diversa da quella x86_64, allora la sua voce nella tabella di syscall dovrà chiamare uno *stub* che invoca la versione `compat_sys_`,

Per completezza, sarebbe carino impostare una mappatura cosicché *user-mode* Linux (UML) continui a funzionare – la sua tabella di syscall farà riferimento a stub_xyzzy, ma UML non include l'implementazione in `arch/x86/entry/entry_64.S` (perché UML simula i registri eccetera). Correggerlo è semplice, basta aggiungere una #define in `arch/x86/um/sys_call_table_64.c`:

```
#define stub_xyzzy sys_xyzzy
```

Altri dettagli

La maggior parte dei kernel tratta le chiamate di sistema allo stesso modo, ma possono esserci rare eccezioni per le quali potrebbe essere necessario l'aggiornamento della vostra chiamata di sistema.

Il sotto-sistema di controllo (*audit subsystem*) è uno di questi casi speciali; esso include (per architettura) funzioni che classificano alcuni tipi di chiamate di sistema – in particolare apertura dei file (`open`/`openat`), esecuzione dei programmi (`execve`/`exeveat`) oppure multiplatori di socket (`socketcall`). Se la vostra nuova chiamata di sistema è simile ad una di queste, allora il sistema di controllo dovrebbe essere aggiornato.

Più in generale, se esiste una chiamata di sistema che è simile alla vostra, vale la pena fare una ricerca con `grep` su tutto il kernel per la chiamata di sistema esistente per verificare che non ci siano altri casi speciali.

Verifica

Una nuova chiamata di sistema dev' essere, ovviamente, provata; è utile fornire ai revisori un programma in spazio utente che mostri l' uso della chiamata di sistema. Un buon modo per combinare queste cose è quello di aggiungere un semplice programma di auto-verifica in una nuova cartella in `tools/testing/selftests/`.

Per una nuova chiamata di sistema, ovviamente, non ci sarà alcuna funzione in libc e quindi il programma di verifica dovrà invocarla usando `syscall()`; inoltre, se la nuova chiamata di sistema prevede un nuova struttura dati visibile in spazio utente, il file d' intestazione necessario dev' essere installato al fine di compilare il programma.

Assicuratevi che il programma di auto-verifica possa essere eseguito correttamente su tutte le architetture supportate. Per esempio, verificate che funzioni quando viene compilato per x86_64 (-m64), x86_32 (-m32) e x32 (-mx32).

Al fine di una più meticolosa ed estesa verifica della nuova funzionalità, dovreste considerare l'aggiunta di nuove verifica al progetto 'Linux Test', oppure al progetto xfstests per cambiamenti relativi al filesystem.

- https://linux-test-project.github.io/
- git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git

Pagine man

Tutte le nuove chiamate di sistema dovrebbero avere una pagina man completa, idealmente usando i marcatori groff, ma anche il puro testo può andare. Se state usando groff, è utile che includiate nella email di presentazione una versione già convertita in formato ASCII: semplificherà la vita dei revisori.

Le pagine man dovrebbero essere in copia-conoscenza verso linux-man@vger.kernel.org Per maggiori dettagli, leggere https://www.kernel.org/doc/man-pages/patches.html

Non invocate chiamate di sistema dal kernel

Le chiamate di sistema sono, come già detto prima, punti di interazione fra lo spazio utente e il kernel. Perciò, le chiamate di sistema come `sys_xyzzy()` o `compat_sys_xyzzy()` dovrebbero essere chiamate solo dallo spazio utente attraverso la tabella syscall, ma non da nessun altro punto nel kernel. Se la nuova funzionalità è utile all'interno del kernel, per esempio dev'essere condivisa fra una vecchia e una nuova chiamata di sistema o dev' essere utilizzata da una chiamata di sistema e la sua variante compatibile, allora dev'essere implementata come una funzione di supporto (*helper function*) (per esempio `kern_xyzzy()`). Questa funzione potrà essere chiamata dallo *stub* (`sys_xyzzy()`), dalla variante compatibile (`compat_sys_xyzzy()`), e/o da altri parti del kernel.

Sui sistemi x86 a 64-bit, a partire dalla versione v4.17 è un requisito fondamentale quello di non invocare chiamate di sistema all' interno del kernel. Esso usa una diversa convenzione per l' invocazione di chiamate di sistema dove `struct pt_regs` viene decodificata al volo in una funzione che racchiude la chiamata di sistema

la quale verrà eseguita successivamente. Questo significa che verranno passati solo i parametri che sono davvero necessari ad una specifica chiamata di sistema, invece che riempire ogni volta 6 registri del processore con contenuti presi dallo spazio utente (potrebbe causare seri problemi nella sequenza di chiamate).

Inoltre, le regole su come i dati possano essere usati potrebbero differire fra il kernel e l'utente. Questo è un altro motivo per cui invocare `sys_xyzzy()` è generalmente una brutta idea.

Eccezioni a questa regola vengono accettate solo per funzioni d'architetture che surclassano quelle generiche, per funzioni d'architettura di compatibilità, o per altro codice in arch/

Riferimenti e fonti

- Articolo di Michael Kerris su LWN sull'uso dell'argomento flags nelle chiamate di sistema: https://lwn.net/Articles/585415/
- Articolo di Michael Kerris su LWN su come gestire flag sconosciuti in una chiamata di sistema: https://lwn.net/Articles/588444/
- Articolo di Jake Edge su LWN che descrive i limiti degli argomenti a 64-bit delle chiamate di sistema: https://lwn.net/Articles/311630/
- Una coppia di articoli di David Drysdale che descrivono i dettagli del percorso implementativo di una chiamata di sistema per la versione v3.14:
  - https://lwn.net/Articles/604287/
  - https://lwn.net/Articles/604515/
- Requisiti specifici alle architetture sono discussi nella pagina man *`syscall(2)`*: http://man7.org/linux/man-pages/man2/syscall.2.html#NOTES
- Collezione di email di Linux Torvalds sui problemi relativi a `ioctl()`: http://yarchive.net/comp/linux/ioctl.html
- "Come non inventare interfacce del kernel", Arnd Bergmann, http://www.ukuug.org/events/linux2007/2007/papers/Bergmann.pdf
- Articolo di Michael Kerris su LWN sull'evitare nuovi usi di CAP_SYS_ADMIN: https://lwn.net/Articles/486306/
- Raccomandazioni da Andrew Morton circa il fatto che tutte le informazioni su una nuova chiamata di sistema dovrebbero essere contenute nello stesso filone di discussione di email: https://lkml.org/lkml/2014/7/24/641
- Raccomandazioni da Michael Kerrisk circa il fatto che le nuove chiamate di sistema dovrebbero avere una pagina man: https://lkml.org/lkml/2014/6/13/309
- Consigli da Thomas Gleixner sul fatto che il collegamento all'architettura x86 dovrebbe avvenire in un *commit* differente: https://lkml.org/lkml/2014/11/19/254
- Consigli da Greg Kroah-Hartman circa la bontà d'avere una pagina man e un programma di auto-verifica per le nuove chiamate di sistema: https://lkml.org/lkml/2014/3/19/710

- Discussione di Michael Kerrisk sulle nuove chiamate di sistema contro le estensioni *prctl(2)*: https://lkml.org/lkml/2014/6/3/411
- Consigli da Ingo Molnar che le chiamate di sistema con più argomenti dovrebbero incapsularli in una struttura che includa un argomento *size* per garantire l'estensibilità futura: https://lkml.org/lkml/2015/7/30/117
- Un certo numero di casi strani emersi dall'uso (riuso) dei flag O_*:
  - commit 75069f2b5bfb ("vfs: renumber FMODE_NONOTIFY and add to uniqueness check")
  - commit 12ed2e36c98a ("fanotify: FMODE_NONOTIFY and __O_SYNC in sparc conflict")
  - commit bb458c644a59 ("Safer ABI for O_TMPFILE")
- Discussion from Matthew Wilcox about restrictions on 64-bit arguments: https://lkml.org/lkml/2008/12/12/187
- Raccomandazioni da Greg Kroah-Hartman sul fatto che i flag sconosciuti dovrebbero essere controllati: https://lkml.org/lkml/2014/7/17/577
- Raccomandazioni da Linus Torvalds che le chiamate di sistema x32 dovrebbero favorire la compatibilità con le versioni a 64-bit piuttosto che quelle a 32-bit: https://lkml.org/lkml/2011/8/31/244

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/magic-number.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

I numeri magici di Linux

Questo documento è un registro dei numeri magici in uso. Quando aggiungete un numero magico ad una struttura, dovreste aggiungerlo anche a questo documento; la cosa migliore è che tutti i numeri magici usati dalle varie strutture siano unici.

È **davvero** un'ottima idea proteggere le strutture dati del kernel con dei numeri magici. Questo vi permette in fase d'esecuzione di (a) verificare se una struttura è stata malmenata, o (b) avete passato a una procedura la struttura errata. Quest'ultimo è molto utile - particolarmente quando si passa una struttura dati tramite un puntatore void *. Il codice tty, per esempio, effettua questa operazione con regolarità passando avanti e indietro le strutture specifiche per driver e discipline.

Per utilizzare un numero magico, dovete dichiararlo all'inizio della struttura dati, come di seguito:

```
struct tty_ldisc {
        int     magic;
        ...
};
```

Per favore, seguite questa direttiva quando aggiungerete migliorie al kernel! Mi ha risparmiato un numero illimitato di ore di debug, specialmente nei casi più ostici dove si è andati oltre la dimensione di un vettore e la struttura dati che lo seguiva in memoria è stata sovrascritta. Seguendo questa direttiva, questi casi vengono identificati velocemente e in sicurezza.

Registro dei cambiamenti:

```
                                        Theodore Ts'o
                                        31 Mar 94

La tabella magica è aggiornata a Linux 2.1.55.

                                        Michael Chastain
                                        <mailto:mec@shout.net>
                                        22 Sep 1997

Ora dovrebbe essere aggiornata a Linux 2.1.112. Dato che
siamo in un momento di congelamento delle funzionalità
(*feature freeze*) è improbabile che qualcosa cambi prima
della versione 2.2.x.  Le righe sono ordinate secondo il
campo numero.

                                        Krzysztof G. Baranowski
                                        <mailto: kgb@knm.org.pl>
                                        29 Jul 1998

Aggiornamento della tabella a Linux 2.5.45. Giusti nel congelamento
delle funzionalità ma è comunque possibile che qualche nuovo
numero magico s'intrufoli prima del kernel 2.6.x.

                                        Petr Baudis
                                        <pasky@ucw.cz>
                                        03 Nov 2002

Aggiornamento della tabella magica a Linux 2.5.74.

                                        Fabian Frederick
                                        <ffrederick@users.sourceforge.
↪net>
                                        09 Jul 2003
```

Nome magico	Numero	Struttura	File
PG_MAGIC	‘P’	pg_{read,write}_hdr	include/linu

Table 1 – continued from previous page

Nome magico	Numero	Struttura	File
CMAGIC	0x0111	user	include/linu
MKISS_DRIVER_MAGIC	0x04bf	mkiss_channel	drivers/net/
HDLC_MAGIC	0x239e	n_hdlc	drivers/char
APM_BIOS_MAGIC	0x4101	apm_user	arch/x86/ker
CYCLADES_MAGIC	0x4359	cyclades_port	include/linu
DB_MAGIC	0x4442	fc_info	drivers/net/
DL_MAGIC	0x444d	fc_info	drivers/net/
FASYNC_MAGIC	0x4601	fasync_struct	include/linu
FF_MAGIC	0x4646	fc_info	drivers/net/
ISICOM_MAGIC	0x4d54	isi_port	include/linu
PTY_MAGIC	0x5001		drivers/char
PPP_MAGIC	0x5002	ppp	include/linu
SSTATE_MAGIC	0x5302	serial_state	include/linu
SLIP_MAGIC	0x5302	slip	drivers/net/
STRIP_MAGIC	0x5303	strip	drivers/net/
X25_ASY_MAGIC	0x5303	x25_asy	drivers/net/
SIXPACK_MAGIC	0x5304	sixpack	drivers/net/
AX25_MAGIC	0x5316	ax_disp	drivers/net/
TTY_MAGIC	0x5401	tty_struct	include/linu
MGSL_MAGIC	0x5401	mgsl_info	drivers/char
TTY_DRIVER_MAGIC	0x5402	tty_driver	include/linu
MGSLPC_MAGIC	0x5402	mgslpc_info	drivers/char
TTY_LDISC_MAGIC	0x5403	tty_ldisc	include/linu
USB_SERIAL_MAGIC	0x6702	usb_serial	drivers/usb/
FULL_DUPLEX_MAGIC	0x6969		drivers/net/
USB_BLUETOOTH_MAGIC	0x6d02	usb_bluetooth	drivers/usb/
RFCOMM_TTY_MAGIC	0x6d02		net/bluetoot
USB_SERIAL_PORT_MAGIC	0x7301	usb_serial_port	drivers/usb/
CG_MAGIC	0x00090255	ufs_cylinder_group	include/linu
RPORT_MAGIC	0x00525001	r_port	drivers/char
LSEMAGIC	0x05091998	lse	drivers/fc4/
GDTIOCTL_MAGIC	0x06030f07	gdth_iowr_str	drivers/scsi
RIEBL_MAGIC	0x09051990		drivers/net/
NBD_REQUEST_MAGIC	0x12560953	nbd_request	include/linu
RED_MAGIC2	0x170fc2a5	(any)	mm/slab.c
BAYCOM_MAGIC	0x19730510	baycom_state	drivers/net/
ISDN_X25IFACE_MAGIC	0x1e75a2b9	isdn_x25iface_proto_data	drivers/isdn
ECP_MAGIC	0x21504345	cdkecpsig	include/linu
LSOMAGIC	0x27091997	lso	drivers/fc4/
LSMAGIC	0x2a3b4d2a	ls	drivers/fc4/
WANPIPE_MAGIC	0x414C4453	sdla_{dump,exec}	include/linu
CS_CARD_MAGIC	0x43525553	cs_card	sound/oss/cs
LABELCL_MAGIC	0x4857434c	labelcl_info_s	include/asm/
ISDN_ASYNC_MAGIC	0x49344C01	modem_info	include/linu
CTC_ASYNC_MAGIC	0x49344C01	ctc_tty_info	drivers/s390
ISDN_NET_MAGIC	0x49344C02	isdn_net_local_s	drivers/isdn
SAVEKMSG_MAGIC2	0x4B4D5347	savekmsg	arch/*/amiga
CS_STATE_MAGIC	0x4c4f4749	cs_state	sound/oss/cs

Table 1 – continued from previous page

Nome magico	Numero	Struttura	File
SLAB_C_MAGIC	0x4f17a36d	kmem_cache	mm/slab.c
COW_MAGIC	0x4f4f4f4d	cow_header_v1	arch/um/driv
I810_CARD_MAGIC	0x5072696E	i810_card	sound/oss/i8
TRIDENT_CARD_MAGIC	0x5072696E	trident_card	sound/oss/tr
ROUTER_MAGIC	0x524d4157	wan_device	[in wanrouter
SAVEKMSG_MAGIC1	0x53415645	savekmsg	arch/*/amiga
GDA_MAGIC	0x58464552	gda	arch/mips/in
RED_MAGIC1	0x5a2cf071	(any)	mm/slab.c
EEPROM_MAGIC_VALUE	0x5ab478d2	lanai_dev	drivers/atm/
HDLCDRV_MAGIC	0x5ac6e778	hdlcdrv_state	include/linu
PCXX_MAGIC	0x5c6df104	channel	drivers/char
KV_MAGIC	0x5f4b565f	kernel_vars_s	arch/mips/in
I810_STATE_MAGIC	0x63657373	i810_state	sound/oss/i8
TRIDENT_STATE_MAGIC	0x63657373	trient_state	sound/oss/tr
M3_CARD_MAGIC	0x646e6f50	m3_card	sound/oss/ma
FW_HEADER_MAGIC	0x65726F66	fw_header	drivers/atm/
SLOT_MAGIC	0x67267321	slot	drivers/hotp
SLOT_MAGIC	0x67267322	slot	drivers/hotp
LO_MAGIC	0x68797548	nbd_device	include/linu
OPROFILE_MAGIC	0x6f70726f	super_block	drivers/oprc
M3_STATE_MAGIC	0x734d724d	m3_state	sound/oss/ma
VMALLOC_MAGIC	0x87654320	snd_alloc_track	sound/core/m
KMALLOC_MAGIC	0x87654321	snd_alloc_track	sound/core/m
PWC_MAGIC	0x89DC10AB	pwc_device	drivers/usb/
NBD_REPLY_MAGIC	0x96744668	nbd_reply	include/linu
ENI155_MAGIC	0xa54b872d	midway_eprom	drivers/atm/
CODA_MAGIC	0xC0DAC0DA	coda_file_info	fs/coda/coda
DPMEM_MAGIC	0xc0ffee11	gdt_pci_sram	drivers/scsi
YAM_MAGIC	0xF10A7654	yam_port	drivers/net/
CCB_MAGIC	0xf2691ad2	ccb	drivers/scsi
QUEUE_MAGIC_FREE	0xf7e1c9a3	queue_entry	drivers/scsi
QUEUE_MAGIC_USED	0xf7e1cc33	queue_entry	drivers/scsi
HTB_CMAGIC	0xFEFAFEF1	htb_class	net/sched/sc
NMI_MAGIC	0x48414d4d455201	nmi_s	arch/mips/in

Da notare che ci sono anche dei numeri magici specifici per driver nel *sound memory management*. Consultate `include/sound/sndmagic.h` per una lista completa. Molti driver audio OSS hanno i loro numeri magici costruiti a partire dall' identificativo PCI della scheda audio - nemmeno questi sono elencati in questo file.

Il file-system HFS è un altro grande utilizzatore di numeri magici - potete trovarli qui `fs/hfs/hfs.h`.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l' unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original

Documentation/process/volatile-considered-harmful.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Perché la parola chiave “volatile” non dovrebbe essere usata

Spesso i programmatori C considerano volatili quelle variabili che potrebbero essere cambiate al di fuori dal thread di esecuzione corrente; come risultato, a volte saranno tentati dall'utilizzare *volatile* nel kernel per le strutture dati condivise. In altre parole, gli è stato insegnato ad usare *volatile* come una variabile atomica di facile utilizzo, ma non è così. L'uso di *volatile* nel kernel non è quasi mai corretto; questo documento ne descrive le ragioni.

Il punto chiave da capire su *volatile* è che il suo scopo è quello di sopprimere le ottimizzazioni, che non è quasi mai quello che si vuole. Nel kernel si devono proteggere le strutture dati condivise contro accessi concorrenti e indesiderati: questa è un'attività completamente diversa. Il processo di protezione contro gli accessi concorrenti indesiderati eviterà anche la maggior parte dei problemi relativi all'ottimizzazione in modo più efficiente.

Come *volatile*, le primitive del kernel che rendono sicuro l'accesso ai dati (spinlock, mutex, barriere di sincronizzazione, ecc) sono progettate per prevenire le ottimizzazioni indesiderate. Se vengono usate opportunamente, non ci sarà bisogno di utilizzare *volatile*. Se vi sembra che *volatile* sia comunque necessario, ci dev'essere quasi sicuramente un baco da qualche parte. In un pezzo di codice kernel scritto a dovere, *volatile* può solo servire a rallentare le cose.

Considerate questo tipico blocco di codice kernel:

```
spin_lock(&the_lock);
do_something_on(&shared_data);
do_something_else_with(&shared_data);
spin_unlock(&the_lock);
```

Se tutto il codice seguisse le regole di sincronizzazione, il valore di un dato condiviso non potrebbe cambiare inaspettatamente mentre si trattiene un lock. Un qualsiasi altro blocco di codice che vorrà usare quel dato rimarrà in attesa del lock. Gli spinlock agiscono come barriere di sincronizzazione - sono stati esplicitamente scritti per agire così - il che significa che gli accessi al dato condiviso non saranno ottimizzati. Quindi il compilatore potrebbe pensare di sapere cosa ci sarà nel dato condiviso ma la chiamata spin_lock(), che agisce come una barriera di sincronizzazione, gli imporrà di dimenticarsi tutto ciò che sapeva su di esso.

Se il dato condiviso fosse stato dichiarato come *volatile*, la sincronizzazione rimarrebbe comunque necessaria. Ma verrà impedito al compilatore di ottimizzare gli accessi al dato anche _dentro_ alla sezione critica, dove sappiamo che in realtà nessun altro può accedervi. Mentre si trattiene un lock, il dato condiviso non è *volatile*. Quando si ha a che fare con dei dati condivisi, un'opportuna sincronizzazione rende inutile l'uso di *volatile* - anzi potenzialmente dannoso.

L'uso di *volatile* fu originalmente pensato per l'accesso ai registri di I/O mappati in memoria. All'interno del kernel, l'accesso ai registri, dovrebbe essere protetto dai lock, ma si potrebbe anche desiderare che il compilatore non “ottimizzi” l'accesso

ai registri all'interno di una sezione critica. Ma, all'interno del kernel, l'accesso alla memoria di I/O viene sempre fatto attraverso funzioni d'accesso; accedere alla memoria di I/O direttamente con i puntatori è sconsigliato e non funziona su tutte le architetture. Queste funzioni d'accesso sono scritte per evitare ottimizzazioni indesiderate, quindi, di nuovo, *volatile* è inutile.

Un'altra situazione dove qualcuno potrebbe essere tentato dall'uso di *volatile*, è nel caso in cui il processore è in un'attesa attiva sul valore di una variabile. Il modo giusto di fare questo tipo di attesa è il seguente:

```
while (my_variable != what_i_want)
    cpu_relax();
```

La chiamata cpu_relax() può ridurre il consumo di energia del processore o cedere il passo ad un processore hyperthreaded gemello; funziona anche come una barriera per il compilatore, quindi, ancora una volta, *volatile* non è necessario. Ovviamente, tanto per puntualizzare, le attese attive sono generalmente un atto antisociale.

Ci sono comunque alcune rare situazioni dove l'uso di *volatile* nel kernel ha senso:

- Le funzioni d'accesso sopracitate potrebbero usare *volatile* su quelle architetture che supportano l'accesso diretto alla memoria di I/O. In pratica, ogni chiamata ad una funzione d'accesso diventa una piccola sezione critica a se stante, e garantisce che l'accesso avvenga secondo le aspettative del programmatore.
- I codice *inline assembly* che fa cambiamenti nella memoria, ma che non ha altri effetti espliciti, rischia di essere rimosso da GCC. Aggiungere la parola chiave *volatile* a questo codice ne previene la rimozione.
- La variabile jiffies è speciale in quanto assume un valore diverso ogni volta che viene letta ma può essere lette senza alcuna sincronizzazione. Quindi jiffies può essere *volatile*, ma l'aggiunta ad altre variabili di questo è sconsigliata. Jiffies è considerata uno "stupido retaggio"(parole di Linus) in questo contesto; correggerla non ne varrebbe la pena e causerebbe più problemi.
- I puntatori a delle strutture dati in una memoria coerente che potrebbe essere modificata da dispositivi di I/O può, a volte, essere legittimamente *volatile*. Un esempio pratico può essere quello di un adattatore di rete che utilizza un puntatore ad un buffer circolare, questo viene cambiato dall'adattatore per indicare quali descrittori sono stati processati.

Per la maggior parte del codice, nessuna delle giustificazioni sopracitate può essere considerata. Di conseguenza, l'uso di *volatile* è probabile che venga visto come un baco e porterà a verifiche aggiuntive. Gli sviluppatori tentati dall'uso di *volatile* dovrebbero fermarsi e pensare a cosa vogliono davvero ottenere.

Le modifiche che rimuovono variabili *volatile* sono generalmente ben accette - purché accompagnate da una giustificazione che dimostri che i problemi di concorrenza siano stati opportunamente considerati.

Riferimenti

[1] http://lwn.net/Articles/233481/

[2] http://lwn.net/Articles/233482/

Crediti

Impulso e ricerca originale di Randy Dunlap

Scritto da Jonathan Corbet

Migliorato dai commenti di Satyam Sharma, Johannes Stezenbach, Jesper Juhl, Heikki Orsila, H. Peter Anvin, Philipp Hahn, e Stefan Richter.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/process/clang-format.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

clang-format

`clang-format` è uno strumento per formattare codice C/C++/⋯secondo un gruppo di regole ed euristiche. Come tutti gli strumenti, non è perfetto e non copre tutti i singoli casi, ma è abbastanza buono per essere utile.

`clang-format` può essere usato per diversi fini:

- Per riformattare rapidamente un blocco di codice secondo lo stile del kernel. Particolarmente utile quando si sposta del codice e lo si allinea/ordina. Vedere *it_clangformatreformat*.
- Identificare errori di stile, refusi e possibili miglioramenti nei file che mantieni, le modifiche che revisioni, le differenze, eccetera. Vedere *it_clangformatreview*.
- Ti aiuta a seguire lo stile del codice, particolarmente utile per i nuovi arrivati o per coloro che lavorano allo stesso tempo su diversi progetti con stili di codifica differenti.

Il suo file di configurazione è `.clang-format` e si trova nella cartella principale dei sorgenti del kernel. Le regole scritte in quel file tentano di approssimare le lo stile di codifica del kernel. Si tenta anche di seguire il più possibile Stile del codice per il kernel Linux. Dato che non tutto il kernel segue lo stesso stile, potreste voler aggiustare le regole di base per un particolare sottosistema o cartella. Per farlo, potete sovrascriverle scrivendole in un altro file `.clang-format` in una sottocartella.

Questo strumento è già stato incluso da molto tempo nelle distribuzioni Linux più popolari. Cercate `clang-format` nel vostro repositorio. Altrimenti, potete scaricare una versione pre-generata dei binari di LLVM/clang oppure generarlo dai codici sorgenti:

http://releases.llvm.org/download.html

Troverete più informazioni ai seguenti indirizzi:

https://clang.llvm.org/docs/ClangFormat.html

https://clang.llvm.org/docs/ClangFormatStyleOptions.html

Revisionare lo stile di codifica per file e modifiche

Eseguendo questo programma, potrete revisionare un intero sottosistema, cartella o singoli file alla ricerca di errori di stile, refusi o miglioramenti.

Per farlo, potete eseguire qualcosa del genere:

```
# Make sure your working directory is clean!
clang-format -i kernel/*.[ch]
```

E poi date un' occhiata a *git diff*.

Osservare le righe di questo diff è utile a migliorare/aggiustare le opzioni di stile nel file di configurazione; così come per verificare le nuove funzionalità/versioni di `clang-format`.

`clang-format` è in grado di leggere diversi diff unificati, quindi potrete revisionare facilmente delle modifiche e *git diff*. La documentazione si trova al seguente indirizzo:

https://clang.llvm.org/docs/ClangFormat.html#
script-for-patch-reformatting

Per evitare che `clang-format` formatti alcune parti di un file, potete scrivere nel codice:

```
int formatted_code;
// clang-format off
    void    unformatted_code  ;
// clang-format on
void formatted_code_again;
```

Nonostante si attraente l' idea di utilizzarlo per mantenere un file sempre in sintonia con `clang-format`, specialmente per file nuovi o se siete un manutentore, ricordatevi che altre persone potrebbero usare una versione diversa di `clang-format` oppure non utilizzarlo del tutto. Quindi, dovreste trattenervi dall' usare questi marcatori nel codice del kernel; almeno finché non vediamo che `clang-format` è diventato largamente utilizzato.

Riformattare blocchi di codice

Utilizzando dei plugin per il vostro editor, potete riformattare una blocco (selezione) di codice con una singola combinazione di tasti. Questo è particolarmente utile: quando si riorganizza il codice, per codice complesso, macro multi-riga (e allineare le loro "barre"), eccetera.

Ricordatevi che potete sempre aggiustare le modifiche in quei casi dove questo strumento non ha fatto un buon lavoro. Ma come prima approssimazione, può essere davvero molto utile.

Questo programma si integra con molti dei più popolari editor. Alcuni di essi come vim, emacs, BBEdit, Visaul Studio, lo supportano direttamente. Al seguente indirizzo troverete le istruzioni:

https://clang.llvm.org/docs/ClangFormat.html

Per Atom, Eclipse, Sublime Text, Visual Studio Code, XCode e altri editor e IDEs dovreste essere in grado di trovare dei plugin pronti all'uso.

Per questo caso d'uso, considerate l'uso di un secondo `.clang-format` che potete personalizzare con le vostre opzioni. Consultare *it_clangformatextra*.

Cose non supportate

`clang-format` non ha il supporto per alcune cose che sono comuni nel codice del kernel. Sono facili da ricordare; quindi, se lo usate regolarmente, imparerete rapidamente a evitare/ignorare certi problemi.

In particolare, quelli più comuni che noterete sono:

- Allineamento di `#define` su una singola riga, per esempio:

```
#define TRACING_MAP_BITS_DEFAULT       11
#define TRACING_MAP_BITS_MAX           17
#define TRACING_MAP_BITS_MIN           7
```

  contro:

```
#define TRACING_MAP_BITS_DEFAULT 11
#define TRACING_MAP_BITS_MAX 17
#define TRACING_MAP_BITS_MIN 7
```

- Allineamento dei valori iniziali, per esempio:

```
static const struct file_operations uprobe_events_ops = {
        .owner          = THIS_MODULE,
        .open           = probes_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = seq_release,
        .write          = probes_write,
};
```

contro:

```
static const struct file_operations uprobe_events_ops = {
        .owner = THIS_MODULE,
        .open = probes_open,
        .read = seq_read,
        .llseek = seq_lseek,
        .release = seq_release,
        .write = probes_write,
};
```

Funzionalità e opzioni aggiuntive

Al fine di minimizzare le differenze fra il codice attuale e l'output del programma, alcune opzioni di stile e funzionalità non sono abilitate nella configurazione base. In altre parole, lo scopo è di rendere le differenze le più piccole possibili, permettendo la semplificazione della revisione di file, differenze e modifiche.

In altri casi (per esempio un particolare sottosistema/cartella/file), lo stile del kernel potrebbe essere diverso e abilitare alcune di queste opzioni potrebbe dare risultati migliori.

Per esempio:

- Allineare assegnamenti (`AlignConsecutiveAssignments`).
- Allineare dichiarazioni (`AlignConsecutiveDeclarations`).
- Riorganizzare il testo nei commenti (`ReflowComments`).
- Ordinare gli `#include` (`SortIncludes`).

Piuttosto che per interi file, solitamente sono utili per la riformattazione di singoli blocchi. In alternativa, potete creare un altro file `.clang-format` da utilizzare con il vostro editor/IDE.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
../../../riscv/patch-acceptance

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

arch/riscv linee guida alla manutenzione per gli sviluppatori

Introduzione

L'insieme di istruzioni RISC-V sono sviluppate in modo aperto: le bozze in fase di sviluppo sono disponibili a tutti per essere revisionate e per essere sperimentare nelle implementazioni. Le bozze dei nuovi moduli o estensioni possono cambiare in fase di sviluppo - a volte in modo incompatibile rispetto a bozze precedenti. Questa flessibilità può portare a dei problemi di manutenzioni per il supporto RISC-V nel kernel Linux. I manutentori Linux non amano l'abbandono del codice, e il processo di sviluppo del kernel preferisce codice ben revisionato e testato rispetto a quello sperimentale. Desideriamo estendere questi stessi principi al codice relativo all'architettura RISC-V che verrà accettato per l'inclusione nel kernel.

In aggiunta alla lista delle verifiche da fare prima di inviare una patch

Accetteremo le patch per un nuovo modulo o estensione se la fondazione RISC-V li classifica come "Frozen" o "Retified". (Ovviamente, gli sviluppatori sono liberi di mantenere una copia del kernel Linux contenente il codice per una bozza di estensione).

In aggiunta, la specifica RISC-V permette agli implementatori di creare le proprie estensioni. Queste estensioni non passano attraverso il processo di revisione della fondazione RISC-V. Per questo motivo, al fine di evitare complicazioni o problemi di prestazioni, accetteremo patch solo per quelle estensioni che sono state ufficialmente accettate dalla fondazione RISC-V. (Ovviamente, gli implementatori sono liberi di mantenere una copia del kernel Linux contenente il codice per queste specifiche estensioni).

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Note: Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

---

Come scrivere la documentazione del kernel

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Note: Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

Introduzione

Il kernel Linux usa Sphinx per la generazione della documentazione a partire dai file reStructuredText che si trovano nella cartella `Documentation`. Per generare la documentazione in HTML o PDF, usate comandi `make htmldocs` o `make pdfdocs`. La documentazione così generata sarà disponibile nella cartella `Documentation/output`.

I file reStructuredText possono contenere delle direttive che permettono di includere i commenti di documentazione, o di tipo kernel-doc, dai file sorgenti. Solitamente questi commenti sono utilizzati per descrivere le funzioni, i tipi e l'architettura del codice. I commenti di tipo kernel-doc hanno una struttura e formato speciale, ma a parte questo vengono processati come reStructuredText.

Inoltre, ci sono migliaia di altri documenti in formato testo sparsi nella cartella `Documentation`. Alcuni di questi verranno probabilmente convertiti, nel tempo, in formato reStructuredText, ma la maggior parte di questi rimarranno in formato testo.

Installazione Sphinx

I marcatori ReST utilizzati nei file in Documentation/ sono pensati per essere processati da `Sphinx` nella versione 1.3 o superiore.

Esiste uno script che verifica i requisiti Sphinx. Per ulteriori dettagli consultate *Verificare le dipendenze Sphinx*.

La maggior parte delle distribuzioni Linux forniscono Sphinx, ma l'insieme dei programmi e librerie è fragile e non è raro che dopo un aggiornamento di Sphinx, o qualche altro pacchetto Python, la documentazione non venga più generata correttamente.

Un modo per evitare questo genere di problemi è quello di utilizzare una versione diversa da quella fornita dalla vostra distribuzione. Per fare questo, vi raccomandiamo di installare Sphinx dentro ad un ambiente virtuale usando `virtualenv-3` o `virtualenv` a seconda di come Python 3 è stato pacchettizzato dalla vostra distribuzione.

Note:

1) Le versioni di Sphinx inferiori alla 1.5 non funzionano bene con il pacchetto Python docutils versione 0.13.1 o superiore. Se volete usare queste versioni, allora dovere eseguire `pip install 'docutils==0.12'`.

2) Viene raccomandato l'uso del tema RTD per la documentazione in HTML. A seconda della versione di Sphinx, potrebbe essere necessaria l'installazione tramite il comando `pip install sphinx_rtd_theme`.

3) Alcune pagine ReST contengono delle formule matematiche. A causa del modo in cui Sphinx funziona, queste espressioni sono scritte utilizzando LaTeX. Per una corretta interpretazione, è necessario aver installato texlive con i pacchetti amdfonts e amsmath.

Riassumendo, se volete installare la versione 1.7.9 di Sphinx dovete eseguire:

```
$ virtualenv sphinx_1.7.9
$ . sphinx_1.7.9/bin/activate
(sphinx_1.7.9) $ pip install -r Documentation/sphinx/requirements.
↪txt
```

Dopo aver eseguito `. sphinx_1.7.9/bin/activate`, il prompt cambierà per indicare che state usando il nuovo ambiente. Se aprite un nuova sessione, prima di generare la documentazione, dovrete rieseguire questo comando per rientrare nell' ambiente virtuale.

Generazione d' immagini

Il meccanismo che genera la documentazione del kernel contiene un' estensione capace di gestire immagini in formato Graphviz e SVG (per maggior informazioni vedere *Figure ed immagini*).

Per far si che questo funzioni, dovete installare entrambe i pacchetti Graphviz e ImageMagick. Il sistema di generazione della documentazione è in grado di procedere anche se questi pacchetti non sono installati, ma il risultato, ovviamente, non includerà le immagini.

Generazione in PDF e LaTeX

Al momento, la generazione di questi documenti è supportata solo dalle versioni di Sphinx superiori alla 1.4.

Per la generazione di PDF e LaTeX, avrete bisogno anche del pacchetto `XeLaTeX` nella versione 3.14159265

Per alcune distribuzioni Linux potrebbe essere necessario installare anche una serie di pacchetti `texlive` in modo da fornire il supporto minimo per il funzionamento di `XeLaTeX`.

Verificare le dipendenze Sphinx

Esiste uno script che permette di verificare automaticamente le dipendenze di Sphinx. Se lo script riesce a riconoscere la vostra distribuzione, allora sarà in grado di darvi dei suggerimenti su come procedere per completare l' installazione:

```
$ ./scripts/sphinx-pre-install
Checking if the needed tools for Fedora release 26 (Twenty Six) are␣
↪available
```

(continues on next page)

(continued from previous page)

```
Warning: better to also install "texlive-luatex85".
You should run:

        sudo dnf install -y texlive-luatex85
        /usr/bin/virtualenv sphinx_1.7.9
        . sphinx_1.7.9/bin/activate
        pip install -r Documentation/sphinx/requirements.txt

Can't build as 1 mandatory dependency is missing at ./scripts/
 ↪sphinx-pre-install line 468.
```

L'impostazione predefinita prevede il controllo dei requisiti per la generazione di documenti html e PDF, includendo anche il supporto per le immagini, le espressioni matematiche e LaTeX; inoltre, presume che venga utilizzato un ambiente virtuale per Python. I requisiti per generare i documenti html sono considerati obbligatori, gli altri sono opzionali.

Questo script ha i seguenti parametri:

--no-pdf
: Disabilita i controlli per la generazione di PDF;

--no-virtualenv
: Utilizza l'ambiente predefinito dal sistema operativo invece che l'ambiente virtuale per Python;

Generazione della documentazione Sphinx

Per generare la documentazione in formato HTML o PDF si eseguono i rispettivi comandi `make htmldocs` o `make pdfdocs`. Esistono anche altri formati in cui è possibile generare la documentazione; per maggiori informazioni potere eseguire il comando `make help`. La documentazione così generata sarà disponibile nella sottocartella `Documentation/output`.

Ovviamente, per generare la documentazione, Sphinx (`sphinx-build`) dev'essere installato. Se disponibile, il tema *Read the Docs* per Sphinx verrà utilizzato per ottenere una documentazione HTML più gradevole. Per la documentazione in formato PDF, invece, avrete bisogno di `XeLaTeX` e di ``convert(1)` disponibile in ImageMagick (https://www.imagemagick.org). Tipicamente, tutti questi pacchetti sono disponibili e pacchettizzati nelle distribuzioni Linux.

Per poter passare ulteriori opzioni a Sphinx potete utilizzare la variabile make `SPHINXOPTS`. Per esempio, se volete che Sphinx sia più verboso durante la generazione potete usare il seguente comando `make SPHINXOPTS=-v htmldocs`.

Potete eliminare la documentazione generata tramite il comando `make cleandocs`.

Scrivere la documentazione

Aggiungere nuova documentazione è semplice:

1. aggiungete un file `.rst` nella sottocartella `Documentation`
2. aggiungete un riferimento ad esso nell' indice (TOC tree) in `Documentation/index.rst`.

Questo, di solito, è sufficiente per la documentazione più semplice (come quella che state leggendo ora), ma per una documentazione più elaborata è consigliato creare una sottocartella dedicata (o, quando possibile, utilizzarne una già esistente). Per esempio, il sottosistema grafico è documentato nella sottocartella `Documentation/gpu`; questa documentazione è divisa in diversi file `.rst` ed un indice `index.rst` (con un `toctree` dedicato) a cui si fa riferimento nell' indice principale.

Consultate la documentazione di Sphinx e reStructuredText per maggiori informazione circa le loro potenzialità. In particolare, il manuale introduttivo a reStructuredText di Sphinx è un buon punto da cui cominciare. Esistono, inoltre, anche alcuni costruttori specifici per Sphinx.

Guide linea per la documentazione del kernel

In questa sezione troverete alcune linee guida specifiche per la documentazione del kernel:

- Non esagerate con i costrutti di reStructuredText. Mantenete la documentazione semplice. La maggior parte della documentazione dovrebbe essere testo semplice con una strutturazione minima che permetta la conversione in diversi formati.
- Mantenete la strutturazione il più fedele possibile all' originale quando convertite un documento in formato reStructuredText.
- Aggiornate i contenuti quando convertite della documentazione, non limitatevi solo alla formattazione.
- Mantenete la decorazione dei livelli di intestazione come segue:
  1. = con una linea superiore per il titolo del documento:

     ```
     ======
     Titolo
     ======
     ```

  2. = per i capitoli:

     ```
     Capitoli
     ========
     ```

  3. \- per le sezioni:

     ```
     Sezioni
     -------
     ```

4. ~ per le sottosezioni:

```
Sottosezioni
~~~~~~~~~~~~
```

Sebbene RST non forzi alcun ordine specifico (*Piuttosto che imporre un numero ed un ordine fisso di decorazioni, l'ordine utilizzato sarà quello incontrato*), avere uniformità dei livelli principali rende più semplice la lettura dei documenti.

- Per inserire blocchi di testo con caratteri a dimensione fissa (codici di esempio, casi d'uso, eccetera): utilizzate `::` quando non è necessario evidenziare la sintassi, specialmente per piccoli frammenti; invece, utilizzate `.. code-block:: <language>` per blocchi più lunghi che beneficeranno della sintassi evidenziata. Per un breve pezzo di codice da inserire nel testo, usate ``.

Il dominio C

Il **Dominio Sphinx C** (denominato c) è adatto alla documentazione delle API C. Per esempio, un prototipo di una funzione:

```
.. c:function:: int ioctl( int fd, int request )
```

Il dominio C per kernel-doc ha delle funzionalità aggiuntive. Per esempio, potete assegnare un nuovo nome di riferimento ad una funzione con un nome molto comune come `open` o `ioctl`:

```
.. c:function:: int ioctl( int fd, int request )
   :name: VIDIOC_LOG_STATUS
```

Il nome della funzione (per esempio ioctl) rimane nel testo ma il nome del suo riferimento cambia da `ioctl` a `VIDIOC_LOG_STATUS`. Anche la voce nell'indice cambia in `VIDIOC_LOG_STATUS`.

Notate che per una funzione non c'è bisogno di usare `c:func:` per generarne i riferimenti nella documentazione. Grazie a qualche magica estensione a Sphinx, il sistema di generazione della documentazione trasformerà automaticamente un riferimento ad una `funzione()` in un riferimento incrociato quando questa ha una voce nell'indice. Se trovate degli usi di `c:func:` nella documentazione del kernel, sentitevi liberi di rimuoverli.

Tabelle a liste

Raccomandiamo l'uso delle tabelle in formato lista (*list table*). Le tabelle in formato lista sono liste di liste. In confronto all'ASCII-art potrebbero non apparire di facile lettura nei file in formato testo. Il loro vantaggio è che sono facili da creare o modificare e che la differenza di una modifica è molto più significativa perché limitata alle modifiche del contenuto.

La `flat-table` è anch'essa una lista di liste simile alle `list-table` ma con delle funzionalità aggiuntive:

- column-span: col ruolo `cspan` una cella può essere estesa attraverso colonne successive
- raw-span: col ruolo `rspan` una cella può essere estesa attraverso righe successive
- auto-span: la cella più a destra viene estesa verso destra per compensare la mancanza di celle. Con l'opzione `:fill-cells:` questo comportamento può essere cambiato da *auto-span* ad *auto-fill*, il quale inserisce automaticamente celle (vuote) invece che estendere l'ultima.

opzioni:

- `:header-rows:` [int] conta le righe di intestazione
- `:stub-columns:` [int] conta le colonne di stub
- `:widths:` [[int] [int] ⋯] larghezza delle colonne
- `:fill-cells:` invece di estendere automaticamente una cella su quelle mancanti, ne crea di vuote.

ruoli:

- `:cspan:` [int] colonne successive (*morecols*)
- `:rspan:` [int] righe successive (*morerows*)

L'esempio successivo mostra come usare questo marcatore. Il primo livello della nostra lista di liste è la *riga*. In una *riga* è possibile inserire solamente la lista di celle che compongono la *riga* stessa. Fanno eccezione i *commenti* (`..`) ed i *collegamenti* (per esempio, un riferimento a `:ref:`last row <last row>`` / *last row*)

```
.. flat-table:: table title
   :widths: 2 1 1 3

   * - head col 1
     - head col 2
     - head col 3
     - head col 4

   * - column 1
     - field 1.1
     - field 1.2 with autospan

   * - column 2
     - field 2.1
     - :rspan:`1` :cspan:`1` field 2.2 - 3.3

   * .. _`it last row`:

     - column 3
```

Che verrà rappresentata nel seguente modo:

Table 2: table title

head col 1	head col 2	head col 3	head col 4
column 1	field 1.1	field 1.2 with autospan	
column 2	field 2.1	field 2.2 - 3.3	
column 3			

Figure ed immagini

Se volete aggiungere un' immagine, utilizzate le direttive `kernel-figure` e `kernel-image`. Per esempio, per inserire una figura di un' immagine in formato SVG:

```
.. kernel-figure::  ../../../doc-guide/svg_image.svg
   :alt:    una semplice immagine SVG

   Una semplice immagine SVG
```

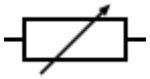

Fig. 1: Una semplice immagine SVG

Le direttive del kernel per figure ed immagini supportano il formato **DOT**, per maggiori informazioni

- DOT: http://graphviz.org/pdf/dotguide.pdf
- Graphviz: http://www.graphviz.org/content/dot-language

Un piccolo esempio (*Esempio DOT*):

```
.. kernel-figure::  ../../../doc-guide/hello.dot
   :alt:    ciao mondo

   Esempio DOT
```

Tramite la direttiva `kernel-render` è possibile aggiungere codice specifico; ad esempio nel formato **DOT** di Graphviz.:

```
.. kernel-render:: DOT
   :alt: foobar digraph
   :caption: Codice **DOT** (Graphviz) integrato

   digraph foo {
    "bar" -> "baz";
   }
```

La rappresentazione dipenderà dei programmi installati. Se avete Graphviz installato, vedrete un' immagine vettoriale. In caso contrario, il codice grezzo verrà rappresentato come *blocco testuale* (*Codice DOT (Graphviz) integrato*).

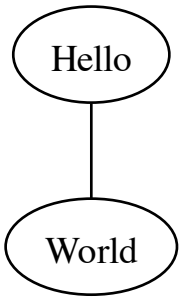

Fig. 2: Esempio DOT

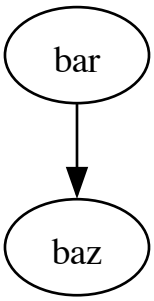

Fig. 3: Codice **DOT** (Graphviz) integrato

La direttiva *render* ha tutte le opzioni della direttiva *figure*, con l'aggiunta dell'opzione `caption`. Se `caption` ha un valore allora un nodo *figure* viene aggiunto. Altrimenti verrà aggiunto un nodo *image*. L'opzione `caption` è necessaria in caso si vogliano aggiungere dei riferimenti (*Integrare codice SVG*).

Per la scrittura di codice **SVG**:

```
.. kernel-render:: SVG
   :caption: Integrare codice **SVG**
   :alt: so-nw-arrow

   <?xml version="1.0" encoding="UTF-8"?>
   <svg xmlns="http://www.w3.org/2000/svg" version="1.1" ...>
      ...
   </svg>
```

Fig. 4: Integrare codice **SVG**

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Note: Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

Scrivere i commenti in kernel-doc

Nei file sorgenti del kernel Linux potrete trovare commenti di documentazione strutturanti secondo il formato kernel-doc. Essi possono descrivere funzioni, tipi di dati, e l'architettura del codice.

Note: Il formato kernel-doc può sembrare simile a gtk-doc o Doxygen ma in realtà è molto differente per ragioni storiche. I sorgenti del kernel contengono decine di migliaia di commenti kernel-doc. Siete pregati d'attenervi allo stile qui descritto.

La struttura kernel-doc è estratta a partire dai commenti; da questi viene generato il dominio Sphinx per il C con un'adeguata descrizione per le funzioni ed i tipi di dato con i loro relativi collegamenti. Le descrizioni vengono filtrare per cercare i riferimenti ed i marcatori.

Vedere di seguito per maggiori dettagli.

Tutte le funzioni esportate verso i moduli esterni utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL` dovrebbero avere un commento kernel-doc. Quando l'in-

tenzione è di utilizzarle nei moduli, anche le funzioni e le strutture dati nei file d'intestazione dovrebbero avere dei commenti kernel-doc.

È considerata una buona pratica quella di fornire una documentazione formattata secondo kernel-doc per le funzioni che sono visibili da altri file del kernel (ovvero, che non siano dichiarate utilizzando `static`). Raccomandiamo, inoltre, di fornire una documentazione kernel-doc anche per procedure private (ovvero, dichiarate "static") al fine di fornire una struttura più coerente dei sorgenti. Quest'ultima raccomandazione ha una priorità più bassa ed è a discrezione dal manutentore (MAINTAINER) del file sorgente.

Sicuramente la documentazione formattata con kernel-doc è necessaria per le funzioni che sono esportate verso i moduli esterni utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL`.

Cerchiamo anche di fornire una documentazione formattata secondo kernel-doc per le funzioni che sono visibili da altri file del kernel (ovvero, che non siano dichiarate utilizzando "static")

Raccomandiamo, inoltre, di fornire una documentazione formattata con kernel-doc anche per procedure private (ovvero, dichiarate "static") al fine di fornire una struttura più coerente dei sorgenti. Questa raccomandazione ha una priorità più bassa ed è a discrezione dal manutentore (MAINTAINER) del file sorgente.

Le strutture dati visibili nei file di intestazione dovrebbero essere anch'esse documentate utilizzando commenti formattati con kernel-doc.

Come formattare i commenti kernel-doc

I commenti kernel-doc iniziano con il marcatore `/**`. Il programma `kernel-doc` estrarrà i commenti marchiati in questo modo. Il resto del commento è formattato come un normale commento multilinea, ovvero con un asterisco all'inizio d'ogni riga e che si conclude con `*/` su una riga separata.

I commenti kernel-doc di funzioni e tipi dovrebbero essere posizionati appena sopra la funzione od il tipo che descrivono. Questo allo scopo di aumentare la probabilità che chi cambia il codice si ricordi di aggiornare anche la documentazione. I commenti kernel-doc di tipo più generale possono essere posizionati ovunque nel file.

Al fine di verificare che i commenti siano formattati correttamente, potete eseguire il programma `kernel-doc` con un livello di verbosità alto e senza che questo produca alcuna documentazione. Per esempio:

```
scripts/kernel-doc -v -none drivers/foo/bar.c
```

Il formato della documentazione è verificato della procedura di generazione del kernel quando viene richiesto di effettuare dei controlli extra con GCC:

```
make W=n
```

Documentare le funzioni

Generalmente il formato di un commento kernel-doc per funzioni e macro simil-funzioni è il seguente:

```
/**
 * function_name() - Brief description of function.
 * @arg1: Describe the first argument.
 * @arg2: Describe the second argument.
 *        One can provide multiple line descriptions
 *        for arguments.
 *
 * A longer description, with more discussion of the function␣
↪function_name()
 * that might be useful to those using or modifying it. Begins with␣
↪an
 * empty comment line, and may include additional embedded empty
 * comment lines.
 *
 * The longer description may have multiple paragraphs.
 *
 * Context: Describes whether the function can sleep, what locks it␣
↪takes,
 *          releases, or expects to be held. It can extend over␣
↪multiple
 *          lines.
 * Return: Describe the return value of function_name.
 *
 * The return value description can also have multiple paragraphs,␣
↪and should
 * be placed at the end of the comment block.
 */
```

La descrizione introduttiva (*brief description*) che segue il nome della funzione può continuare su righe successive e termina con la descrizione di un argomento, una linea di commento vuota, oppure la fine del commento.

Parametri delle funzioni

Ogni argomento di una funzione dovrebbe essere descritto in ordine, subito dopo la descrizione introduttiva. Non lasciare righe vuote né fra la descrizione introduttiva e quella degli argomenti, né fra gli argomenti.

Ogni `@argument:` può estendersi su più righe.

Note: Se la descrizione di `@argument:` si estende su più righe, la continuazione dovrebbe iniziare alla stessa colonna della riga precedente:

```
* @argument: some long description
*            that continues on next lines
```

or:

```
* @argument:
*         some long description
*         that continues on next lines
```

Se una funzione ha un numero variabile di argomento, la sua descrizione dovrebbe essere scritta con la notazione kernel-doc:

```
* @...: description
```

Contesto delle funzioni

Il contesto in cui le funzioni vengono chiamate viene descritto in una sezione chiamata `Context`. Questo dovrebbe informare sulla possibilità che una funzione dorma (*sleep*) o che possa essere chiamata in un contesto d' interruzione, così come i *lock* che prende, rilascia e che si aspetta che vengano presi dal chiamante.

Esempi:

```
* Context: Any context.
* Context: Any context. Takes and releases the RCU lock.
* Context: Any context. Expects <lock> to be held by caller.
* Context: Process context. May sleep if @gfp flags permit.
* Context: Process context. Takes and releases <mutex>.
* Context: Softirq or process context. Takes and releases <lock>,␣
 ↪BH-safe.
* Context: Interrupt context.
```

Valore di ritorno

Il valore di ritorno, se c' è, viene descritto in una sezione dedicata di nome `Return`.

Note:

1) La descrizione multiriga non riconosce il termine d' una riga, per cui se provate a formattare bene il vostro testo come nel seguente esempio:

   ```
   * Return:
   * 0 - OK
   * -EINVAL - invalid argument
   * -ENOMEM - out of memory
   ```

   le righe verranno unite e il risultato sarà:

   ```
   Return: 0 - OK -EINVAL - invalid argument -ENOMEM - out of␣
    ↪memory
   ```

Quindi, se volete che le righe vengano effettivamente generate, dovete utilizzare una lista ReST, ad esempio:

```
* Return:
* * 0             - OK to runtime suspend the device
* * -EBUSY        - Device should not be runtime suspended
```

2) Se il vostro testo ha delle righe che iniziano con una frase seguita dai due punti, allora ognuna di queste frasi verrà considerata come il nome di una nuova sezione, e probabilmente non produrrà gli effetti desiderati.

Documentare strutture, unioni ed enumerazioni

Generalmente il formato di un commento kernel-doc per struct, union ed enum è:

```
/**
 * struct struct_name - Brief description.
 * @member1: Description of member1.
 * @member2: Description of member2.
 *           One can provide multiple line descriptions
 *           for members.
 *
 * Description of the structure.
 */
```

Nell' esempio qui sopra, potete sostituire `struct` con `union` o `enum` per descrivere unioni ed enumerati. `member` viene usato per indicare i membri di strutture ed unioni, ma anche i valori di un tipo enumerato.

La descrizione introduttiva (*brief description*) che segue il nome della funzione può continuare su righe successive e termina con la descrizione di un argomento, una linea di commento vuota, oppure la fine del commento.

Membri

I membri di strutture, unioni ed enumerati devo essere documentati come i parametri delle funzioni; seguono la descrizione introduttiva e possono estendersi su più righe.

All' interno d' una struttura o d' un unione, potete utilizzare le etichette `private:` e `public:`. I campi che sono nell' area `private:` non verranno inclusi nella documentazione finale.

Le etichette `private:` e `public:` devono essere messe subito dopo il marcatore di un commento `/*`. Opzionalmente, possono includere commenti fra `:` e il marcatore di fine commento `*/`.

Esempio:

```
/**
 * struct my_struct - short description
 * @a: first member
 * @b: second member
 * @d: fourth member
 *
 * Longer description
 */
struct my_struct {
    int a;
    int b;
/* private: internal use only */
    int c;
/* public: the next one is public */
    int d;
};
```

Strutture ed unioni annidate

È possibile documentare strutture ed unioni annidate, ad esempio:

```
/**
 * struct nested_foobar - a struct with nested unions and structs
 * @memb1: first member of anonymous union/anonymous struct
 * @memb2: second member of anonymous union/anonymous struct
 * @memb3: third member of anonymous union/anonymous struct
 * @memb4: fourth member of anonymous union/anonymous struct
 * @bar: non-anonymous union
 * @bar.st1: struct st1 inside @bar
 * @bar.st2: struct st2 inside @bar
 * @bar.st1.memb1: first member of struct st1 on union bar
 * @bar.st1.memb2: second member of struct st1 on union bar
 * @bar.st2.memb1: first member of struct st2 on union bar
 * @bar.st2.memb2: second member of struct st2 on union bar
 */
struct nested_foobar {
  /* Anonymous union/struct*/
  union {
    struct {
      int memb1;
      int memb2;
  }
    struct {
      void *memb3;
      int memb4;
    }
  }
  union {
    struct {
```

(continues on next page)

(continued from previous page)

```
      int memb1;
      int memb2;
    } st1;
    struct {
      void *memb1;
      int memb2;
    } st2;
  } bar;
};
```

Note:

1) Quando documentate una struttura od unione annidata, ad esempio di nome `foo`, il suo campo `bar` dev' essere documentato usando `@foo.bar:`

2) Quando la struttura od unione annidata è anonima, il suo campo `bar` dev' essere documentato usando `@bar:`

Commenti in linea per la documentazione dei membri

I membri d' una struttura possono essere documentati in linea all' interno della definizione stessa. Ci sono due stili: una singola riga di commento che inizia con `/**` e finisce con `*/`; commenti multi riga come qualsiasi altro commento kernel-doc:

```
/**
 * struct foo - Brief description.
 * @foo: The Foo member.
 */
struct foo {
      int foo;
      /**
       * @bar: The Bar member.
       */
      int bar;
      /**
       * @baz: The Baz member.
       *
       * Here, the member description may contain several␣
↪paragraphs.
       */
      int baz;
      union {
              /** @foobar: Single line description. */
              int foobar;
      };
      /** @bar2: Description for struct @bar2 inside @foo */
      struct {
```

(continues on next page)

(continued from previous page)

```
             /**
              * @bar2.barbar: Description for @barbar inside @foo.
↪bar2
              */
             int barbar;
     } bar2;
};
```

Documentazione dei tipi di dato

Generalmente il formato di un commento kernel-doc per typedef è il seguente:

```
/**
 * typedef type_name - Brief description.
 *
 * Description of the type.
 */
```

Anche i tipi di dato per prototipi di funzione possono essere documentati:

```
/**
 * typedef type_name - Brief description.
 * @arg1: description of arg1
 * @arg2: description of arg2
 *
 * Description of the type.
 *
 * Context: Locking context.
 * Return: Meaning of the return value.
 */
 typedef void (*type_name)(struct v4l2_ctrl *arg1, void *arg2);
```

Marcatori e riferimenti

All'interno dei commenti di tipo kernel-doc vengono riconosciuti i seguenti *pattern* che vengono convertiti in marcatori reStructuredText ed in riferimenti del dominio Sphinx per il C.

> **Attention:** Questi sono riconosciuti **solo** all'interno di commenti kernel-doc, e **non** all'interno di documenti reStructuredText.

`funcname()`
: Riferimento ad una funzione.

`@parameter`
: Nome di un parametro di una funzione (nessun riferimento, solo formattazione).

%CONST
Il nome di una costante (nessun riferimento, solo formattazione)

``literal``
Un blocco di testo che deve essere riportato così com'è. La rappresentazione finale utilizzerà caratteri a `spaziatura fissa`.

Questo è utile se dovete utilizzare caratteri speciali che altrimenti potrebbero assumere un significato diverso in kernel-doc o in reStructuredText

Questo è particolarmente utile se dovete scrivere qualcosa come `%ph` all'interno della descrizione di una funzione.

$ENVVAR
Il nome di una variabile d'ambiente (nessun riferimento, solo formattazione).

&struct name
Riferimento ad una struttura.

&enum name
Riferimento ad un'enumerazione.

&typedef name
Riferimento ad un tipo di dato.

&struct_name->member or &struct_name.member
Riferimento ad un membro di una struttura o di un'unione. Il riferimento sarà la struttura o l'unione, non il memembro.

&name
Un generico riferimento ad un tipo. Usate, preferibilmente, il riferimento completo come descritto sopra. Questo è dedicato ai commenti obsoleti.

Riferimenti usando reStructuredText

Per fare riferimento a funzioni e tipi di dato definiti nei commenti kernel-doc all'interno dei documenti reStructuredText, utilizzate i riferimenti dal dominio Sphinx per il C. Per esempio:

```
See function :c:func:`foo` and struct/union/enum/typedef␣
↪:c:type:`bar`.
```

Nonostante il riferimento ai tipi di dato funzioni col solo nome, ovvero senza specificare struct/union/enum/typedef, potreste preferire il seguente:

```
See :c:type:`struct foo <foo>`.
See :c:type:`union bar <bar>`.
See :c:type:`enum baz <baz>`.
See :c:type:`typedef meh <meh>`.
```

Questo produce dei collegamenti migliori, ed è in linea con il modo in cui kernel-doc gestisce i riferimenti.

Per maggiori informazioni, siete pregati di consultare la documentazione del dominio Sphinx per il C.

Commenti per una documentazione generale

Al fine d'avere il codice ed i commenti nello stesso file, potete includere dei blocchi di documentazione kernel-doc con un formato libero invece che nel formato specifico per funzioni, strutture, unioni, enumerati o tipi di dato. Per esempio, questo tipo di commento potrebbe essere usato per la spiegazione delle operazioni di un driver o di una libreria

Questo s'ottiene utilizzando la parola chiave `DOC:` a cui viene associato un titolo.

Generalmente il formato di un commento generico o di visione d'insieme è il seguente:

```
/**
 * DOC: Theory of Operation
 *
 * The whizbang foobar is a dilly of a gizmo. It can do whatever you
 * want it to do, at any time. It reads your mind. Here's how it␣
↪works.
 *
 * foo bar splat
 *
 * The only drawback to this gizmo is that is can sometimes damage
 * hardware, software, or its subject(s).
 */
```

Il titolo che segue `DOC:` funziona da intestazione all'interno del file sorgente, ma anche come identificatore per l'estrazione di questi commenti di documentazione. Quindi, il titolo dev'essere unico all'interno del file.

Includere i commenti di tipo kernel-doc

I commenti di documentazione possono essere inclusi in un qualsiasi documento di tipo reStructuredText mediante l'apposita direttiva nell'estensione kernel-doc per Sphinx.

Le direttive kernel-doc sono nel formato:

```
.. kernel-doc:: source
   :option:
```

Il campo *source* è il percorso ad un file sorgente, relativo alla cartella principale dei sorgenti del kernel. La direttiva supporta le seguenti opzioni:

export: *[source-pattern …]*
: Include la documentazione per tutte le funzioni presenti nel file sorgente (*source*) che sono state esportate utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL` in *source* o in qualsiasi altro *source-pattern* specificato.

  Il campo *source-patter* è utile quando i commenti kernel-doc sono stati scritti nei file d'intestazione, mentre `EXPORT_SYMBOL` e `EXPORT_SYMBOL_GPL` si trovano vicino alla definizione delle funzioni.

  Esempi:

```
.. kernel-doc:: lib/bitmap.c
   :export:

.. kernel-doc:: include/net/mac80211.h
   :export: net/mac80211/*.c
```

internal: ***[source-pattern …]***
Include la documentazione per tutte le funzioni ed i tipi presenti nel file sorgente (*source*) che **non** sono stati esportati utilizzando `EXPORT_SYMBOL` o `EXPORT_SYMBOL_GPL` né in *source* né in qualsiasi altro *source-pattern* specificato.

Esempio:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
   :internal:
```

identifiers: ***[function/type …]***
Include la documentazione per ogni *function* e *type* in *source*. Se non vengono esplicitamente specificate le funzioni da includere, allora verranno incluse tutte quelle disponibili in *source*.

Esempi:

```
.. kernel-doc:: lib/bitmap.c
   :identifiers: bitmap_parselist bitmap_parselist_user

.. kernel-doc:: lib/idr.c
   :identifiers:
```

functions: ***[function …]***
Questo è uno pseudonimo, deprecato, per la direttiva ‘identifiers’.

doc: ***title***
Include la documentazione del paragrafo `DOC:` identificato dal titolo (*title*) all’interno del file sorgente (*source*). Gli spazi in *title* sono permessi; non virgolettate *title*. Il campo *title* è utilizzato per identificare un paragrafo e per questo non viene incluso nella documentazione finale. Verificate d’avere l’intestazione appropriata nei documenti reStructuredText.

Esempio:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
   :doc: High Definition Audio over HDMI and Display Port
```

Senza alcuna opzione, la direttiva kernel-doc include tutti i commenti di documentazione presenti nel file sorgente (*source*).

L’estensione kernel-doc fa parte dei sorgenti del kernel, la si può trovare in `Documentation/sphinx/kerneldoc.py`. Internamente, viene utilizzato lo script `scripts/kernel-doc` per estrarre i commenti di documentazione dai file sorgenti.

Come utilizzare kernel-doc per generare pagine man

Se volete utilizzare kernel-doc solo per generare delle pagine man, potete farlo direttamente dai sorgenti del kernel:

```
$ scripts/kernel-doc -man $(git grep -l '/\*\*' -- :^Documentation␣
↪:^tools) | scripts/split-man.pl /tmp/man
```

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Note: Per leggere la documentazione originale in inglese: Documentation/doc-guide/index.rst

Includere gli i file di intestazione uAPI

Qualche volta è utile includere dei file di intestazione e degli esempi di codice C al fine di descrivere l'API per lo spazio utente e per generare dei riferimenti fra il codice e la documentazione. Aggiungere i riferimenti ai file dell'API dello spazio utente ha ulteriori vantaggi: Sphinx genererà dei messaggi d'avviso se un simbolo non viene trovato nella documentazione. Questo permette di mantenere allineate la documentazione della uAPI (API spazio utente) con le modifiche del kernel. Il programma *parse_headers.pl* genera questi riferimenti. Esso dev'essere invocato attraverso un Makefile, mentre si genera la documentazione. Per avere un esempio su come utilizzarlo all'interno del kernel consultate `Documentation/userspace-api/media/Makefile`.

parse_headers.pl

NOME

parse_headers.pl - analizza i file C al fine di identificare funzioni, strutture, enumerati e definizioni, e creare riferimenti per Sphinx

SINTASSI

parse_headers.pl [<options>] <C_FILE> <OUT_FILE> [<EXCEPTIONS_FILE>]

Dove <options> può essere: –debug, –usage o –help.

OPZIONI

-debug

Lo script viene messo in modalità verbosa, utile per il debugging.

-usage

Mostra un messaggio d’aiuto breve e termina.

-help

Mostra un messaggio d’aiuto dettagliato e termina.

DESCRIZIONE

Converte un file d’intestazione o un file sorgente C (C_FILE) in un testo ReStructuredText incluso mediante il blocco ..parsed-literal con riferimenti alla documentazione che descrive l’API. Opzionalmente, il programma accetta anche un altro file (EXCEPTIONS_FILE) che descrive quali elementi debbano essere ignorati o il cui riferimento deve puntare ad elemento diverso dal predefinito.

Il file generato sarà disponibile in (OUT_FILE).

Il programma è capace di identificare *define*, funzioni, strutture, tipi di dato, enumerati e valori di enumerati, e di creare i riferimenti per ognuno di loro. Inoltre, esso è capace di distinguere le #define utilizzate per specificare i comandi ioctl di Linux.

Il file EXCEPTIONS_FILE contiene due tipi di dichiarazioni: **ignore** o **replace**.

La sintassi per ignore è:

ignore **tipo nome**

La dichiarazione **ignore** significa che non verrà generato alcun riferimento per il simbolo **name** di tipo **tipo**.

La sintassi per replace è:

replace **tipo nome nuovo_valore**

La dichiarazione **replace** significa che verrà generato un riferimento per il simbolo **name**di tipo **tipo**, ma, invece di utilizzare il valore predefinito, verrà utilizzato il valore **nuovo_valore**.

Per entrambe le dichiarazioni, il **tipo** può essere uno dei seguenti:

ioctl

La dichiarazione ignore o replace verrà applicata su definizioni di ioctl come la seguente:

#define VIDIOC_DBG_S_REGISTER _IOW(‘V’, 79, struct v4l2_dbg_register)

define

La dichiarazione ignore o replace verrà applicata su una qualsiasi #define trovata in C_FILE.

typedef

La dichiarazione ignore o replace verrà applicata ad una dichiarazione typedef in C_FILE.

struct

La dichiarazione ignore o replace verrà applicata ai nomi di strutture in C_FILE.

enum

La dichiarazione ignore o replace verrà applicata ai nomi di enumerati in C_FILE.

symbol

La dichiarazione ignore o replace verrà applicata ai nomi di valori di enumerati in C_FILE.

Per le dichiarazioni di tipo replace, il campo **new_value** utilizzerà automaticamente i riferimenti :c:type: per **typedef**, **enum** e **struct**. Invece, utilizzerà :ref: per **ioctl**, **define** e **symbol**. Il tipo di riferimento può essere definito esplicitamente nella dichiarazione stessa.

ESEMPI

ignore define _VIDEODEV2_H

Ignora una definizione #define _VIDEODEV2_H nel file C_FILE.

ignore symbol PRIVATE

In un enumerato come il seguente:

enum foo { BAR1, BAR2, PRIVATE };

Non genererà alcun riferimento per **PRIVATE**.

replace symbol BAR1 :c:type:`foo` replace symbol BAR2 :c:type:`foo`

In un enumerato come il seguente:

enum foo { BAR1, BAR2, PRIVATE };

Genererà un riferimento ai valori BAR1 e BAR2 dal simbolo foo nel dominio C.

BUGS

Riferire ogni malfunzionamento a Mauro Carvalho Chehab <mchehab@s-opensource.com>

COPYRIGHT

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/kernel-hacking/index.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Guida all'hacking del kernel

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Note: Per leggere la documentazione originale in inglese: Documentation/kernel-hacking/hacking.rst

---

Original
Documentation/kernel-hacking/hacking.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

L'inaffidabile guida all'hacking del kernel Linux

Author
Rusty Russell

Introduzione

Benvenuto, gentile lettore, alla notevole ed inaffidabile guida all'hacking del kernel Linux ad opera di Rusty. Questo documento descrive le procedure più usate ed i concetti necessari per scrivere codice per il kernel: lo scopo è di fornire ai programmatori C più esperti un manuale di base per sviluppo. Eviterò dettagli implementativi: per questo abbiamo il codice, ed ignorerò intere parti di alcune procedure.

Prima di leggere questa guida, sappiate che non ho mai voluto scriverla, essendo esageratamente sotto qualificato, ma ho sempre voluto leggere qualcosa di simile, e quindi questa era l'unica via. Spero che possa crescere e diventare un compendio di buone pratiche, punti di partenza e generiche informazioni.

Gli attori

In qualsiasi momento ognuna delle CPU di un sistema può essere:

- non associata ad alcun processo, servendo un' interruzione hardware;
- non associata ad alcun processo, servendo un softirq o tasklet;
- in esecuzione nello spazio kernel, associata ad un processo (contesto utente);
- in esecuzione di un processo nello spazio utente;

Esiste un ordine fra questi casi. Gli ultimi due possono avvicendarsi (preempt) l'un l'altro, ma a parte questo esiste una gerarchia rigida: ognuno di questi può avvicendarsi solo ad uno di quelli sottostanti. Per esempio, mentre un softirq è in esecuzione su d'una CPU, nessun altro softirq può avvicendarsi nell'esecuzione, ma un' interruzione hardware può. Ciò nonostante, le altre CPU del sistema operano indipendentemente.

Più avanti vedremo alcuni modi in cui dal contesto utente è possibile bloccare le interruzioni, così da impedirne davvero il diritto di prelazione.

Contesto utente

Ci si trova nel contesto utente quando si arriva da una chiamata di sistema od altre eccezioni: come nello spazio utente, altre procedure più importanti, o le interruzioni, possono far valere il proprio diritto di prelazione sul vostro processo. Potete sospendere l' esecuzione chiamando `schedule()`.

Note: Si è sempre in contesto utente quando un modulo viene caricato o rimosso, e durante le operazioni nello strato dei dispositivi a blocchi (*block layer*).

Nel contesto utente, il puntatore `current` (il quale indica il processo al momento in esecuzione) è valido, e `in_interrupt()` (`include/linux/preempt.h`) è falsa.

Warning: Attenzione che se avete la prelazione o i softirq disabilitati (vedere di seguito), `in_interrupt()` ritornerà un falso positivo.

Interruzioni hardware (Hard IRQs)

Temporizzatori, schede di rete e tastiere sono esempi di vero hardware che possono produrre interruzioni in un qualsiasi momento. Il kernel esegue i gestori d'interruzione che prestano un servizio all'hardware. Il kernel garantisce che questi gestori non vengano mai interrotti: se una stessa interruzione arriva, questa verrà accodata (o scartata). Dato che durante la loro esecuzione le interruzioni vengono disabilitate, i gestori d'interruzioni devono essere veloci: spesso si limitano esclusivamente a notificare la presa in carico dell'interruzione, programmare una 'interruzione software' per l'esecuzione e quindi terminare.

Potete dire d'essere in una interruzione hardware perché `in_irq()` ritorna vero.

Warning: Attenzione, questa ritornerà un falso positivo se le interruzioni sono disabilitate (vedere di seguito).

Contesto d'interruzione software: softirq e tasklet

Quando una chiamata di sistema sta per tornare allo spazio utente, oppure un gestore d'interruzioni termina, qualsiasi 'interruzione software' marcata come pendente (solitamente da un'interruzione hardware) viene eseguita (`kernel/softirq.c`).

La maggior parte del lavoro utile alla gestione di un'interruzione avviene qui. All'inizio della transizione ai sistemi multiprocessore, c'erano solo i cosiddetti 'bottom half' (BH), i quali non traevano alcun vantaggio da questi sistemi. Non appena abbandonammo i computer raffazzonati con fiammiferi e cicche, abbandonammo anche questa limitazione e migrammo alle interruzioni software 'softirqs'.

Il file `include/linux/interrupt.h` elenca i differenti tipi di 'softirq'. Un tipo di softirq molto importante è il timer (`include/linux/timer.h`): potete programmarlo per far si che esegua funzioni dopo un determinato periodo di tempo.

Dato che i softirq possono essere eseguiti simultaneamente su più di un processore, spesso diventa estenuante l'averci a che fare. Per questa ragione, i tasklet (`include/linux/interrupt.h`) vengo usati più di frequente: possono essere registrati dinamicamente (il che significa che potete averne quanti ne volete), e garantiscono che un qualsiasi tasklet verrà eseguito solo su un processore alla volta, sebbene diversi tasklet possono essere eseguiti simultaneamente.

Warning: Il nome 'tasklet' è ingannevole: non hanno niente a che fare con i 'processi' ('tasks'), e probabilmente hanno più a che vedere con qualche pessima vodka che Alexey Kuznetsov si fece a quel tempo.

Potete determinate se siete in un softirq (o tasklet) utilizzando la macro `in_softirq()` (`include/linux/preempt.h`).

Warning: State attenti che questa macro ritornerà un falso positivo se *botton half lock* è bloccato.

Alcune regole basilari

Nessuna protezione della memoria
Se corrompete la memoria, che sia in contesto utente o d' interruzione, la macchina si pianterà. Siete sicuri che quello che volete fare non possa essere fatto nello spazio utente?

Nessun numero in virgola mobile o MMX
Il contesto della FPU non è salvato; anche se siete in contesto utente lo stato dell' FPU probabilmente non corrisponde a quello del processo corrente: vi incasinerete con lo stato di qualche altro processo. Se volete davvero usare la virgola mobile, allora dovrete salvare e recuperare lo stato dell' FPU (ed evitare cambi di contesto). Generalmente è una cattiva idea; usate l' aritmetica a virgola fissa.

Un limite rigido dello stack
A seconda della configurazione del kernel lo stack è fra 3K e 6K per la maggior parte delle architetture a 32-bit; è di 14K per la maggior parte di quelle a 64-bit; e spesso è condiviso con le interruzioni, per cui non si può usare. Evitare profonde ricorsioni ad enormi array locali nello stack (allocateli dinamicamente).

Il kernel Linux è portabile
Quindi mantenetelo tale. Il vostro codice dovrebbe essere a 64-bit ed indipendente dall'ordine dei byte (endianess) di un processore. Inoltre, dovreste minimizzare il codice specifico per un processore; per esempio il codice assembly dovrebbe essere incapsulato in modo pulito e minimizzato per facilitarne la migrazione. Generalmente questo codice dovrebbe essere limitato alla parte di kernel specifica per un' architettura.

ioctl: non scrivere nuove chiamate di sistema

Una chiamata di sistema, generalmente, è scritta così:

```
asmlinkage long sys_mycall(int arg)
{
        return 0;
}
```

Primo, nella maggior parte dei casi non volete creare nuove chiamate di sistema. Create un dispositivo a caratteri ed implementate l' appropriata chiamata ioctl. Questo meccanismo è molto più flessibile delle chiamate di sistema: esso non dev' essere dichiarato in tutte le architetture nei file `include/asm/unistd.h` e `arch/kernel/entry.S`; inoltre, è improbabile che questo venga accettato da Linus.

Se tutto quello che il vostro codice fa è leggere o scrivere alcuni parametri, considerate l'implementazione di un'interfaccia `sysfs()`.

All'interno di una ioctl vi trovate nel contesto utente di un processo. Quando avviene un errore dovete ritornare un valore negativo di errno (consultate `include/uapi/asm-generic/errno-base.h`, `include/uapi/asm-generic/errno.h` e `include/linux/errno.h`), altrimenti ritornate 0.

Dopo aver dormito dovreste verificare se ci sono stati dei segnali: il modo Unix/Linux di gestire un segnale è di uscire temporaneamente dalla chiamata di sistema con l'errore `-ERESTARTSYS`. La chiamata di sistema ritornerà al contesto utente, eseguirà il gestore del segnale e poi la vostra chiamata di sistema riprenderà (a meno che l'utente non l'abbia disabilitata). Quindi, dovreste essere pronti per continuare l'esecuzione, per esempio nel mezzo della manipolazione di una struttura dati.

```
if (signal_pending(current))
        return -ERESTARTSYS;
```

Se dovete eseguire dei calcoli molto lunghi: pensate allo spazio utente. Se **davvero** volete farlo nel kernel ricordatevi di verificare periodicamente se dovete *lasciare* il processore (ricordatevi che, per ogni processore, c'è un sistema multiprocesso senza diritto di prelazione). Esempio:

```
cond_resched(); /* Will sleep */
```

Una breve nota sulla progettazione delle interfacce: il motto dei sistemi UNIX è "fornite meccanismi e non politiche"

La ricetta per uno stallo

Non è permesso invocare una procedura che potrebbe dormire, fanno eccezione i seguenti casi:

- Siete in un contesto utente.
- Non trattenete alcun spinlock.
- Avete abilitato le interruzioni (in realtà, Andy Kleen dice che lo schedulatore le abiliterà per voi, ma probabilmente questo non è quello che volete).

Da tener presente che alcune funzioni potrebbero dormire implicitamente: le più comuni sono quelle per l'accesso allo spazio utente (*_user) e quelle per l'allocazione della memoria senza l'opzione `GFP_ATOMIC`

Dovreste sempre compilare il kernel con l'opzione `CONFIG_DEBUG_ATOMIC_SLEEP` attiva, questa vi avviserà se infrangete una di queste regole. Se **infrangete** le regole, allora potreste bloccare il vostro scatolotto.

Veramente.

Alcune delle procedure più comuni

printk()

Definita in include/linux/printk.h

printk() fornisce messaggi alla console, dmesg, e al demone syslog. Essa è utile per il debugging o per la notifica di errori; può essere utilizzata anche all'interno del contesto d'interruzione, ma usatela con cautela: una macchina che ha la propria console inondata da messaggi diventa inutilizzabile. La funzione utilizza un formato stringa quasi compatibile con la printf ANSI C, e la concatenazione di una stringa C come primo argomento per indicare la "priorità":

```
printk(KERN_INFO "i = %u\n", i);
```

Consultate include/linux/kern_levels.h per gli altri valori KERN_; questi sono interpretati da syslog come livelli. Un caso speciale: per stampare un indirizzo IP usate:

```
__be32 ipaddress;
printk(KERN_INFO "my ip: %pI4\n", &ipaddress);
```

printk() utilizza un buffer interno di 1K e non s'accorge di eventuali sforamenti. Accertatevi che vi basti.

Note: Saprete di essere un vero hacker del kernel quando inizierete a digitare nei vostri programmi utenti le printf come se fossero printk :)

Note: Un'altra nota a parte: la versione originale di Unix 6 aveva un commento sopra alla funzione printf: "Printf non dovrebbe essere usata per il chiacchiericcio". Dovreste seguire questo consiglio.

---

copy_to_user() / copy_from_user() / get_user() / put_user()

Definite in include/linux/uaccess.h / asm/uaccess.h

[DORMONO]

put_user() e get_user() sono usate per ricevere ed impostare singoli valori (come int, char, o long) da e verso lo spazio utente. Un puntatore nello spazio utente non dovrebbe mai essere dereferenziato: i dati dovrebbero essere copiati usando suddette procedure. Entrambe ritornano -EFAULT oppure 0.

copy_to_user() e copy_from_user() sono più generiche: esse copiano una quantità arbitraria di dati da e verso lo spazio utente.

Warning: Al contrario di:c:func:*put_user()* e get_user(), queste funzioni ritornano la quantità di dati copiati (0 è comunque un successo).

[Sì, questa stupida interfaccia mi imbarazza. La battaglia torna in auge anno dopo anno. –RR]

Le funzioni potrebbero dormire implicitamente. Queste non dovrebbero mai essere invocate fuori dal contesto utente (non ha senso), con le interruzioni disabilitate, o con uno spinlock trattenuto.

kmalloc()/kfree()

Definite in `include/linux/slab.h`

[POTREBBERO DORMIRE: LEGGI SOTTO]

Queste procedure sono utilizzate per la richiesta dinamica di un puntatore ad un pezzo di memoria allineato, esattamente come malloc e free nello spazio utente, ma `kmalloc()` ha un argomento aggiuntivo per indicare alcune opzioni. Le opzioni più importanti sono:

GFP_KERNEL
: Potrebbe dormire per librarare della memoria. L'opzione fornisce il modo più affidabile per allocare memoria, ma il suo uso è strettamente limitato allo spazio utente.

GFP_ATOMIC
: Non dorme. Meno affidabile di `GFP_KERNEL`, ma può essere usata in un contesto d'interruzione. Dovreste avere **davvero** una buona strategia per la gestione degli errori in caso di mancanza di memoria.

GFP_DMA
: Alloca memoria per il DMA sul bus ISA nello spazio d'indirizzamento inferiore ai 16MB. Se non sapete cos'è allora non vi serve. Molto inaffidabile.

Se vedete un messaggio d'avviso per una funzione dormiente che viene chiamata da un contesto errato, allora probabilmente avete usato una funzione d'allocazione dormiente da un contesto d'interruzione senza `GFP_ATOMIC`. Dovreste correggerlo. Sbrigatevi, non cincischiate.

Se allocate almeno `PAGE_SIZE``(``asm/page.h` o `asm/page_types.h`) byte, considerate l'uso di `__get_free_pages()` (`include/linux/gfp.h`). Accetta un argomento che definisce l'ordine (0 per per la dimensione di una pagine, 1 per una doppia pagina, 2 per quattro pagine, eccetra) e le stesse opzioni d'allocazione viste precedentemente.

Se state allocando un numero di byte notevolemnte superiore ad una pagina potete usare `vmalloc()`. Essa allocherà memoria virtuale all'interno dello spazio kernel. Questo è un blocco di memoria fisica non contiguo, ma la MMU vi darà l'impressione che lo sia (quindi, sarà contiguo solo dal punto di vista dei processori, non dal punto di vista dei driver dei dispositivi esterni). Se per qualche strana ragione avete davvero bisogno di una grossa quantità di memoria fisica contigua, avete un problema: Linux non ha un buon supporto per questo caso d'uso perché, dopo un po' di tempo, la frammentazione della memoria rende l'operazione difficile. Il modo migliore per allocare un simile blocco all'inizio dell'avvio del sistema è attraverso la procedura `alloc_bootmem()`.

Prima di inventare la vostra cache per gli oggetti più usati, considerate l'uso di una cache slab disponibile in `include/linux/slab.h`.

`current()`

Definita in `include/asm/current.h`

Questa variabile globale (in realtà una macro) contiene un puntatore alla struttura del processo corrente, quindi è valido solo dal contesto utente. Per esempio, quando un processo esegue una chiamata di sistema, questo punterà alla struttura dati del processo chiamate. Nel contesto d'interruzione in suo valore **non è NULL**.

`mdelay()/udelay()`

Definite in `include/asm/delay.h` / `include/linux/delay.h`

Le funzioni `udelay()` e `ndelay()` possono essere utilizzate per brevi pause. Non usate grandi valori perché rischiate d'avere un overflow - in questo contesto la funzione `mdelay()` è utile, oppure considerate `msleep()`.

`cpu_to_be32()/be32_to_cpu()/cpu_to_le32()/le32_to_cpu()`

Definite in `include/asm/byteorder.h`

La famiglia di funzioni `cpu_to_be32()` (dove "32" può essere sostituito da 64 o 16, e "be" con "le") forniscono un modo generico per fare conversioni sull'ordine dei byte (endianess): esse ritornano il valore convertito. Tutte le varianti supportano anche il processo inverso: `be32_to_cpu()`, eccetera.

Queste funzioni hanno principalmente due varianti: la variante per puntatori, come `cpu_to_be32p()`, che prende un puntatore ad un tipo, e ritorna il valore convertito. L'altra variante per la famiglia di conversioni "in-situ", come `cpu_to_be32s()`, che convertono il valore puntato da un puntatore, e ritornano void.

`local_irq_save()/local_irq_restore()`

Definite in `include/linux/irqflags.h`

Queste funzioni abilitano e disabilitano le interruzioni hardware sul processore locale. Entrambe sono rientranti; esse salvano lo stato precedente nel proprio argomento `unsigned long flags`. Se sapete che le interruzioni sono abilite, potete semplicemente utilizzare `local_irq_disable()` e `local_irq_enable()`.

local_bh_disable()/local_bh_enable()

Definite in include/linux/bottom_half.h

Queste funzioni abilitano e disabilitano le interruzioni software sul processore locale. Entrambe sono rientranti; se le interruzioni software erano già state disabilitate in precedenza, rimarranno disabilitate anche dopo aver invocato questa coppia di funzioni. Lo scopo è di prevenire l'esecuzione di softirq e tasklet sul processore attuale.

smp_processor_id()

Definita in include/linux/smp.h

get_cpu() nega il diritto di prelazione (quindi non potete essere spostati su un altro processore all'improvviso) e ritorna il numero del processore attuale, fra 0 e NR_CPUS. Da notare che non è detto che la numerazione dei processori sia continua. Quando avete terminato, ritornate allo stato precedente con put_cpu().

Se sapete che non dovete essere interrotti da altri processi (per esempio, se siete in un contesto d'interruzione, o il diritto di prelazione è disabilitato) potete utilizzare smp_processor_id().

__init/__exit/__initdata

Definite in include/linux/init.h

Dopo l'avvio, il kernel libera una sezione speciale; le funzioni marcate con __init e le strutture dati marcate con __initdata vengono eliminate dopo il completamento dell'avvio: in modo simile i moduli eliminano questa memoria dopo l'inizializzazione. __exit viene utilizzato per dichiarare che una funzione verrà utilizzata solo in fase di rimozione: la detta funzione verrà eliminata quando il file che la contiene non è compilato come modulo. Guardate l'header file per informazioni. Da notare che non ha senso avere una funzione marcata come __init e al tempo stesso esportata ai moduli utilizzando EXPORT_SYMBOL() o EXPORT_SYMBOL_GPL() - non funzionerà.

__initcall()/module_init()

Definite in include/linux/init.h / include/linux/module.h

Molte parti del kernel funzionano bene come moduli (componenti del kernel caricabili dinamicamente). L'utilizzo delle macro module_init() e module_exit() semplifica la scrittura di codice che può funzionare sia come modulo, sia come parte del kernel, senza l'ausilio di #ifdef.

La macro module_init() definisce quale funzione dev'essere chiamata quando il modulo viene inserito (se il file è stato compilato come tale), o in fase di avvio : se il file non è stato compilato come modulo la macro module_init() diventa equivalente a __initcall(), la quale, tramite qualche magia del linker, s'assicura che la funzione venga chiamata durante l'avvio.

La funzione può ritornare un numero d'errore negativo per scatenare un fallimento del caricamento (sfortunatamente, questo non ha effetto se il modulo è compilato come parte integrante del kernel). Questa funzione è chiamata in contesto utente con le interruzioni abilitate, quindi potrebbe dormire.

`module_exit()`

Definita in `include/linux/module.h`

Questa macro definisce la funzione che dev' essere chiamata al momento della rimozione (o mai, nel caso in cui il file sia parte integrante del kernel). Essa verrà chiamata solo quando il contatore d' uso del modulo raggiunge lo zero. Questa funzione può anche dormire, ma non può fallire: tutto dev' essere ripulito prima che la funzione ritorni.

Da notare che questa macro è opzionale: se non presente, il modulo non sarà removibile (a meno che non usiate 'rmmod -f').

`try_module_get()/module_put()`

Definite in `include/linux/module.h`

Queste funzioni maneggiano il contatore d' uso del modulo per proteggerlo dalla rimozione (in aggiunta, un modulo non può essere rimosso se un altro modulo utilizzo uno dei sui simboli esportati: vedere di seguito). Prima di eseguire codice del modulo, dovreste chiamare `try_module_get()` su quel modulo: se fallisce significa che il modulo è stato rimosso e dovete agire come se non fosse presente. Altrimenti, potete accedere al modulo in sicurezza, e chiamare `module_put()` quando avete finito.

La maggior parte delle strutture registrabili hanno un campo owner (proprietario), come nella struttura `struct file_operations`. Impostate questo campo al valore della macro `THIS_MODULE`.

Code d' attesa `include/linux/wait.h`

[DORMONO]

Una coda d' attesa è usata per aspettare che qualcuno vi attivi quando una certa condizione s' avvera. Per evitare corse critiche, devono essere usate con cautela. Dichiarate una `wait_queue_head_t`, e poi i processi che vogliono attendere il verificarsi di quella condizione dichiareranno una `wait_queue_entry_t` facendo riferimento a loro stessi, poi metteranno questa in coda.

Dichiarazione

Potere dichiarare una `wait_queue_head_t` utilizzando la macro `DECLARE_WAIT_QUEUE_HEAD()` oppure utilizzando la procedura `init_waitqueue_head()` nel vostro codice d'inizializzazione.

Accodamento

Mettersi in una coda d'attesa è piuttosto complesso, perché dovete mettervi in coda prima di verificare la condizione. Esiste una macro a questo scopo: `wait_event_interruptible()` (`include/linux/wait.h`). Il primo argomento è la testa della coda d'attesa, e il secondo è un'espressione che dev'essere valutata; la macro ritorna 0 quando questa espressione è vera, altrimenti `-ERESTARTSYS` se è stato ricevuto un segnale. La versione `wait_event()` ignora i segnali.

Svegliare una procedura in coda

Chiamate `wake_up()` (`include/linux/wait.h`); questa attiverà tutti i processi in coda. Ad eccezione se uno di questi è impostato come `TASK_EXCLUSIVE`, in questo caso i rimanenti non verranno svegliati. Nello stesso header file esistono altre varianti di questa funzione.

Operazioni atomiche

Certe operazioni sono garantite come atomiche su tutte le piattaforme. Il primo gruppo di operazioni utilizza `atomic_t` (`include/asm/atomic.h`); questo contiene un intero con segno (minimo 32bit), e dovete utilizzare queste funzione per modificare o leggere variabili di tipo `atomic_t`. `atomic_read()` e `atomic_set()` leggono ed impostano il contatore, `atomic_add()`, `atomic_sub()`, `atomic_inc()`, `atomic_dec()`, e `atomic_dec_and_test()` (ritorna vero se raggiunge zero dopo essere stata decrementata).

Sì. Ritorna vero (ovvero != 0) se la variabile atomica è zero.

Da notare che queste funzioni sono più lente rispetto alla normale aritmetica, e quindi non dovrebbero essere usate a sproposito.

Il secondo gruppo di operazioni atomiche sono definite in `include/linux/bitops.h` ed agiscono sui bit d'una variabile di tipo `unsigned long`. Queste operazioni prendono come argomento un puntatore alla variabile, e un numero di bit dove 0 è quello meno significativo. `set_bit()`, `clear_bit()` e `change_bit()` impostano, cancellano, ed invertono il bit indicato. `test_and_set_bit()`, `test_and_clear_bit()` e `test_and_change_bit()` fanno la stessa cosa, ad eccezione che ritornano vero se il bit era impostato; queste sono particolarmente utili quando si vuole impostare atomicamente dei flag.

Con queste operazioni è possibile utilizzare indici di bit che eccedono il valore `BITS_PER_LONG`. Il comportamento è strano sulle piattaforme big-endian quindi è meglio evitarlo.

Simboli

All' interno del kernel, si seguono le normali regole del linker (ovvero, a meno che un simbolo non venga dichiarato con visibilita limitata ad un file con la parola chiave `static`, esso può essere utilizzato in qualsiasi parte del kernel). Nonostante ciò, per i moduli, esiste una tabella dei simboli esportati che limita i punti di accesso al kernel. Anche i moduli possono esportare simboli.

EXPORT_SYMBOL()

Definita in `include/linux/export.h`

Questo è il classico metodo per esportare un simbolo: i moduli caricati dinamicamente potranno utilizzare normalmente il simbolo.

EXPORT_SYMBOL_GPL()

Definita in `include/linux/export.h`

Essa è simile a `EXPORT_SYMBOL()` ad eccezione del fatto che i simboli esportati con `EXPORT_SYMBOL_GPL()` possono essere utilizzati solo dai moduli che hanno dichiarato una licenza compatibile con la GPL attraverso `MODULE_LICENSE()`. Questo implica che la funzione esportata è considerata interna, e non una vera e propria interfaccia. Alcuni manutentori e sviluppatori potrebbero comunque richiedere `EXPORT_SYMBOL_GPL()` quando si aggiungono nuove funzionalità o interfacce.

EXPORT_SYMBOL_NS()

Definita in `include/linux/export.h`

Questa è una variate di *EXPORT_SYMBOL()* che permette di specificare uno spazio dei nomi. Lo spazio dei nomi è documentato in *Spazio dei nomi dei simboli*

EXPORT_SYMBOL_NS_GPL()

Definita in `include/linux/export.h`

Questa è una variate di *EXPORT_SYMBOL_GPL()* che permette di specificare uno spazio dei nomi. Lo spazio dei nomi è documentato in *Spazio dei nomi dei simboli*

Procedure e convenzioni

Liste doppiamente concatenate `include/linux/list.h`

Un tempo negli header del kernel c'erano tre gruppi di funzioni per le liste concatenate, ma questa è stata la vincente. Se non avete particolari necessità per una semplice lista concatenata, allora questa è una buona scelta.

In particolare, `list_for_each_entry()` è utile.

Convenzione dei valori di ritorno

Per codice chiamato in contesto utente, è molto comune sfidare le convenzioni C e ritornare 0 in caso di successo, ed un codice di errore negativo (eg. `-EFAULT`) nei casi fallimentari. Questo potrebbe essere controintuitivo a prima vista, ma è abbastanza diffuso nel kernel.

Utilizzate `ERR_PTR()` (`include/linux/err.h`) per codificare un numero d'errore negativo in un puntatore, e `IS_ERR()` e `PTR_ERR()` per recuperarlo di nuovo: così si evita d'avere un puntatore dedicato per il numero d'errore. Da brividi, ma in senso positivo.

Rompere la compilazione

Linus e gli altri sviluppatori a volte cambiano i nomi delle funzioni e delle strutture nei kernel in sviluppo; questo non è solo per tenere tutti sulle spine: questo riflette cambiamenti fondamentati (eg. la funzione non può più essere chiamata con le funzioni attive, o fa controlli aggiuntivi, o non fa più controlli che venivano fatti in precedenza). Solitamente a questo s'accompagna un'adeguata e completa nota sulla lista di discussone linux-kernel; cercate negli archivi. Solitamente eseguire una semplice sostituzione su tutto un file rendere le cose **peggiori**.

Inizializzazione dei campi d'una struttura

Il metodo preferito per l'inizializzazione delle strutture è quello di utilizzare gli inizializzatori designati, come definiti nello standard ISO C99, eg:

```
static struct block_device_operations opt_fops = {
        .open               = opt_open,
        .release            = opt_release,
        .ioctl              = opt_ioctl,
        .check_media_change = opt_media_change,
};
```

Questo rende più facile la ricerca con grep, e rende più chiaro quale campo viene impostato. Dovreste fare così perché si mostra meglio.

Estensioni GNU

Le estensioni GNU sono esplicitamente permesse nel kernel Linux. Da notare che alcune delle più complesse non sono ben supportate, per via dello scarso sviluppo, ma le seguenti sono da considerarsi la norma (per maggiori dettagli, leggete la sezione "C Extensions" nella pagina info di GCC - Sì, davvero la pagina info, la pagina man è solo un breve riassunto delle cose nella pagina info).

- Funzioni inline
- Istruzioni in espressioni (ie. il costrutto ({ and })).
- Dichiarate attributi di una funzione / variabile / tipo (__attribute__)
- typeof
- Array con lunghezza zero
- Macro varargs
- Aritmentica sui puntatori void
- Inizializzatori non costanti
- Istruzioni assembler (non al di fuori di 'arch/' e 'include/asm/')
- Nomi delle funzioni come stringhe (__func__).
- __builtin_constant_p()

Siate sospettosi quando utilizzate long long nel kernel, il codice generato da gcc è orribile ed anche peggio: le divisioni e le moltiplicazioni non funzionano sulle piattaforme i386 perché le rispettive funzioni di runtime di GCC non sono incluse nell'ambiente del kernel.

C++

Solitamente utilizzare il C++ nel kernel è una cattiva idea perché il kernel non fornisce il necessario ambiente di runtime e gli header file non sono stati verificati. Rimane comunque possibile, ma non consigliato. Se davvero volete usarlo, almeno evitate le eccezioni.

NUMif

Viene generalmente considerato più pulito l'uso delle macro negli header file (o all'inizio dei file .c) per astrarre funzioni piuttosto che utlizzare l'istruzione di pre-processore `#if' all'interno del codice sorgente.

Mettere le vostre cose nel kernel

Al fine d'avere le vostre cose in ordine per l'inclusione ufficiale, o anche per avere patch pulite, c'è del lavoro amministrativo da fare:

- Trovare di chi è lo stagno in cui state pisciando. Guardare in cima ai file sorgenti, all'interno del file `MAINTAINERS`, ed alla fine di tutti nel file `CREDITS`. Dovreste coordinarvi con queste persone per evitare di duplicare gli sforzi, o provare qualcosa che è già stato rigettato.

  Assicuratevi di mettere il vostro nome ed indirizzo email in cima a tutti i file che create o che mangeggiate significativamente. Questo è il primo posto dove le persone guarderanno quando troveranno un baco, o quando **loro** vorranno fare una modifica.

- Solitamente vorrete un'opzione di configurazione per la vostra modifica al kernel. Modificate `Kconfig` nella cartella giusta. Il linguaggio Config è facile con copia ed incolla, e c'è una completa documentazione nel file `Documentation/kbuild/kconfig-language.rst`.

  Nella descrizione della vostra opzione, assicuratevi di parlare sia agli utenti esperti sia agli utente che non sanno nulla del vostro lavoro. Menzionate qui le incompatibilità ed i problemi. Chiaramente la descrizione deve terminare con "if in doubt, say N" (se siete in dubbio, dite N) (oppure, occasionalmente, `Y'); questo è per le persone che non hanno idea di che cosa voi stiate parlando.

- Modificate il file `Makefile`: le variabili CONFIG sono esportate qui, quindi potete solitamente aggiungere una riga come la seguete "obj-$(CONFIG_xxx) += xxx.o". La sintassi è documentata nel file `Documentation/kbuild/makefiles.rst`.

- Aggiungete voi stessi in `CREDITS` se avete fatto qualcosa di notevole, solitamente qualcosa che supera il singolo file (comunque il vostro nome dovrebbe essere all'inizio dei file sorgenti). `MAINTAINERS` significa che volete essere consultati quando vengono fatte delle modifiche ad un sottosistema, e quando ci sono dei bachi; questo implica molto di più di un semplice impegno su una parte del codice.

- Infine, non dimenticatevi di leggere `Documentation/process/submitting-patches.rst` e possibilmente anche `Documentation/process/submitting-drivers.rst`.

Trucchetti del kernel

Dopo una rapida occhiata al codice, questi sono i preferiti. Sentitevi liberi di aggiungerne altri.

`arch/x86/include/asm/delay.h`:

```
#define ndelay(n) (__builtin_constant_p(n) ? \
        ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul))␣
↪: \
        __ndelay(n))
```

include/linux/fs.h:

```
/*
 * Kernel pointers have redundant information, so we can use a
 * scheme where we can return either an error code or a dentry
 * pointer with the same return value.
 *
 * This should be a per-architecture thing, to allow different
 * error and pointer decisions.
 */
 #define ERR_PTR(err)    ((void *)((long)(err)))
 #define PTR_ERR(ptr)    ((long)(ptr))
 #define IS_ERR(ptr)     ((unsigned long)(ptr) > (unsigned long)(-
 ↪1000))
```

arch/x86/include/asm/uaccess_32.h::

```
#define copy_to_user(to,from,n)                         \
        (__builtin_constant_p(n) ?                      \
         __constant_copy_to_user((to),(from),(n)) :     \
         __generic_copy_to_user((to),(from),(n)))
```

arch/sparc/kernel/head.S::

```
/*
 * Sun people can't spell worth damn. "compatability" indeed.
 * At least we *know* we can't spell, and use a spell-checker.
 */

/* Uh, actually Linus it is I who cannot spell. Too much murky
 * Sparc assembly will do this to ya.
 */
C_LABEL(cputypvar):
        .asciz "compatibility"

/* Tested on SS-5, SS-10. Probably someone at Sun applied a spell-
 ↪checker. */
        .align 4
C_LABEL(cputypvar_sun4m):
        .asciz "compatible"
```

arch/sparc/lib/checksum.S::

```
/* Sun, you just can't beat me, you just can't.  Stop trying,
 * give up.  I'm serious, I am going to kick the living shit
 * out of you, game over, lights out.
 */
```

Ringraziamenti

Ringrazio Andi Kleen per le sue idee, le risposte alle mie domande, le correzioni dei miei errori, l'aggiunta di contenuti, eccetera. Philipp Rumpf per l'ortografia e per aver reso più chiaro il testo, e per alcuni eccellenti punti tutt'altro che ovvi. Werner Almesberger per avermi fornito un ottimo riassunto di `disable_irq()`, e Jes Sorensen e Andrea Arcangeli per le precisazioni. Michael Elizabeth Chastain per aver verificato ed aggiunto la sezione configurazione. Telsa Gwynne per avermi insegnato DocBook.

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
Documentation/kernel-hacking/locking.rst

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

L'inaffidabile guida alla sincronizzazione

Author
Rusty Russell

Introduzione

Benvenuto, alla notevole ed inaffidabile guida ai problemi di sincronizzazione (locking) nel kernel. Questo documento descrive il sistema di sincronizzazione nel kernel Linux 2.6.

Dato il largo utilizzo del multi-threading e della prelazione nel kernel Linux, chiunque voglia dilettarsi col kernel deve conoscere i concetti fondamentali della concorrenza e della sincronizzazione nei sistemi multi-processore.

Il problema con la concorrenza

(Saltatelo se sapete già cos'è una corsa critica).

In un normale programma, potete incrementare un contatore nel seguente modo:

```
contatore++;
```

Questo è quello che vi aspettereste che accada sempre:

Table 3: Risultati attesi

Istanza 1	Istanza 2
leggi contatore (5)	
aggiungi 1 (6)	
scrivi contatore (6)	
	leggi contatore (6)
	aggiungi 1 (7)
	scrivi contatore (7)

Questo è quello che potrebbe succedere in realtà:

Table 4: Possibile risultato

Istanza 1	Istanza 2
leggi contatore (5)	
	leggi contatore (5)
aggiungi 1 (6)	
	aggiungi 1 (6)
scrivi contatore (6)	
	scrivi contatore (6)

Corse critiche e sezioni critiche

Questa sovrapposizione, ovvero quando un risultato dipende dal tempo che intercorre fra processi diversi, è chiamata corsa critica. La porzione di codice che contiene questo problema è chiamata sezione critica. In particolar modo da quando Linux ha incominciato a girare su macchine multi-processore, le sezioni critiche sono diventate uno dei maggiori problemi di progettazione ed implementazione del kernel.

La prelazione può sortire gli stessi effetti, anche se c' è una sola CPU: interrompendo un processo nella sua sezione critica otterremo comunque la stessa corsa critica. In questo caso, il thread che si avvicenda nell' esecuzione potrebbe eseguire anch' esso la sezione critica.

La soluzione è quella di riconoscere quando avvengono questi accessi simultanei, ed utilizzare i *lock* per accertarsi che solo un' istanza per volta possa entrare nella sezione critica. Il kernel offre delle buone funzioni a questo scopo. E poi ci sono quelle meno buone, ma farò finta che non esistano.

Sincronizzazione nel kernel Linux

Se posso darvi un suggerimento: non dormite mai con qualcuno più pazzo di voi. Ma se dovessi darvi un suggerimento sulla sincronizzazione: **mantenetela semplice**.

Siate riluttanti nell' introduzione di nuovi *lock*.

Abbastanza strano, quest' ultimo è l' esatto opposto del mio suggerimento su quando **avete** dormito con qualcuno più pazzo di voi. E dovreste pensare a prendervi un cane bello grande.

I due principali tipi di *lock* nel kernel: spinlock e mutex

Ci sono due tipi principali di *lock* nel kernel. Il tipo fondamentale è lo spinlock (`include/asm/spinlock.h`), un semplice *lock* che può essere trattenuto solo da un processo: se non si può trattenere lo spinlock, allora rimane in attesa attiva (in inglese *spinning*) finché non ci riesce. Gli spinlock sono molto piccoli e rapidi, possono essere utilizzati ovunque.

Il secondo tipo è il mutex (`include/linux/mutex.h`): è come uno spinlock, ma potreste bloccarvi trattenendolo. Se non potete trattenere un mutex il vostro processo si auto-sospenderà; verrà riattivato quando il mutex verrà rilasciato. Questo significa che il processore potrà occuparsi d' altro mentre il vostro processo è in attesa. Esistono molti casi in cui non potete permettervi di sospendere un processo (vedere *Quali funzioni possono essere chiamate in modo sicuro dalle interruzioni?*) e quindi dovrete utilizzare gli spinlock.

Nessuno di questi *lock* è ricorsivo: vedere *Stallo: semplice ed avanzato*

I *lock* e i kernel per sistemi monoprocessore

Per i kernel compilati senza `CONFIG_SMP` e senza `CONFIG_PREEMPT` gli spinlock non esistono. Questa è un' ottima scelta di progettazione: quando nessun altro processo può essere eseguito in simultanea, allora non c' è la necessità di avere un *lock*.

Se il kernel è compilato senza `CONFIG_SMP` ma con `CONFIG_PREEMPT`, allora gli spinlock disabilitano la prelazione; questo è sufficiente a prevenire le corse critiche. Nella maggior parte dei casi, possiamo considerare la prelazione equivalente ad un sistema multi-processore senza preoccuparci di trattarla indipendentemente.

Dovreste verificare sempre la sincronizzazione con le opzioni `CONFIG_SMP` e `CONFIG_PREEMPT` abilitate, anche quando non avete un sistema multi-processore, questo vi permetterà di identificare alcuni problemi di sincronizzazione.

Come vedremo di seguito, i mutex continuano ad esistere perché sono necessari per la sincronizzazione fra processi in contesto utente.

Sincronizzazione in contesto utente

Se avete una struttura dati che verrà utilizzata solo dal contesto utente, allora, per proteggerla, potete utilizzare un semplice mutex (`include/linux/mutex.h`). Questo è il caso più semplice: inizializzate il mutex; invocate mutex_lock_interruptible() per trattenerlo e mutex_unlock() per rilasciarlo. C'è anche mutex_lock() ma questa dovrebbe essere evitata perché non ritorna in caso di segnali.

Per esempio: `net/netfilter/nf_sockopt.c` permette la registrazione di nuove chiamate per setsockopt() e getsockopt() usando la funzione nf_register_sockopt(). La registrazione e la rimozione vengono eseguite solamente quando il modulo viene caricato o scaricato (e durante l'avvio del sistema, qui non abbiamo concorrenza), e la lista delle funzioni registrate viene consultata solamente quando setsockopt() o getsockopt() sono sconosciute al sistema. In questo caso `nf_sockopt_mutex` è perfetto allo scopo, in particolar modo visto che setsockopt e getsockopt potrebbero dormire.

Sincronizzazione fra il contesto utente e i softirq

Se un softirq condivide dati col contesto utente, avete due problemi. Primo, il contesto utente corrente potrebbe essere interroto da un softirq, e secondo, la sezione critica potrebbe essere eseguita da un altro processore. Questo è quando spin_lock_bh() (`include/linux/spinlock.h`) viene utilizzato. Questo disabilita i softirq sul processore e trattiene il *lock*. Invece, spin_unlock_bh() fa l'opposto. (Il suffisso '_bh' è un residuo storico che fa riferimento al "Bottom Halves", il vecchio nome delle interruzioni software. In un mondo perfetto questa funzione si chiamerebbe 'spin_lock_softirq()').

Da notare che in questo caso potete utilizzare anche spin_lock_irq() o spin_lock_irqsave(), queste fermano anche le interruzioni hardware: vedere *Contesto di interruzione hardware*.

Questo funziona alla perfezione anche sui sistemi monoprocessore: gli spinlock svaniscono e questa macro diventa semplicemente local_bh_disable() (`include/linux/interrupt.h`), la quale impedisce ai softirq d'essere eseguiti.

Sincronizzazione fra contesto utente e i tasklet

Questo caso è uguale al precedente, un tasklet viene eseguito da un softirq.

Sincronizzazione fra contesto utente e i timer

Anche questo caso è uguale al precedente, un timer viene eseguito da un softirq. Dal punto di vista della sincronizzazione, tasklet e timer sono identici.

Sincronizzazione fra tasklet e timer

Qualche volta un tasklet od un timer potrebbero condividere i dati con un altro tasklet o timer

Lo stesso tasklet/timer

Dato che un tasklet non viene mai eseguito contemporaneamente su due processori, non dovete preoccuparvi che sia rientrante (ovvero eseguito più volte in contemporanea), perfino su sistemi multi-processore.

Differenti tasklet/timer

Se un altro tasklet/timer vuole condividere dati col vostro tasklet o timer, allora avrete bisogno entrambe di spin_lock() e spin_unlock(). Qui spin_lock_bh() è inutile, siete già in un tasklet ed avete la garanzia che nessun altro verrà eseguito sullo stesso processore.

Sincronizzazione fra softirq

Spesso un softirq potrebbe condividere dati con se stesso o un tasklet/timer.

Lo stesso softirq

Lo stesso softirq può essere eseguito su un diverso processore: allo scopo di migliorare le prestazioni potete utilizzare dati riservati ad ogni processore (vedere *Dati per processore*). Se siete arrivati fino a questo punto nell' uso dei softirq, probabilmente tenete alla scalabilità delle prestazioni abbastanza da giustificarne la complessità aggiuntiva.

Dovete utilizzare spin_lock() e spin_unlock() per proteggere i dati condivisi.

Diversi Softirqs

Dovete utilizzare spin_lock() e spin_unlock() per proteggere i dati condivisi, che siano timer, tasklet, diversi softirq o lo stesso o altri softirq: uno qualsiasi di essi potrebbe essere in esecuzione su un diverso processore.

Contesto di interruzione hardware

Solitamente le interruzioni hardware comunicano con un tasklet o un softirq. Spesso questo si traduce nel mettere in coda qualcosa da fare che verrà preso in carico da un softirq.

Sincronizzazione fra interruzioni hardware e softirq/tasklet

Se un gestore di interruzioni hardware condivide dati con un softirq, allora avrete due preoccupazioni. Primo, il softirq può essere interrotto da un'interruzione hardware, e secondo, la sezione critica potrebbe essere eseguita da un' interruzione hardware su un processore diverso. Questo è il caso dove spin_lock_irq() viene utilizzato. Disabilita le interruzioni sul processore che l' esegue, poi trattiene il lock. spin_unlock_irq() fa l' opposto.

Il gestore d'interruzione hardware non ha bisogno di usare spin_lock_irq() perché i softirq non possono essere eseguiti quando il gestore d'interruzione hardware è in esecuzione: per questo si può usare spin_lock(), che è un po' più veloce. L' unica eccezione è quando un altro gestore d' interruzioni hardware utilizza lo stesso *lock*: spin_lock_irq() impedirà a questo secondo gestore di interrompere quello in esecuzione.

Questo funziona alla perfezione anche sui sistemi monoprocessore: gli spinlock svaniscono e questa macro diventa semplicemente local_irq_disable() (`include/asm/smp.h`), la quale impedisce a softirq/tasklet/BH d' essere eseguiti.

spin_lock_irqsave() (`include/linux/spinlock.h`) è una variante che salva lo stato delle interruzioni in una variabile, questa verrà poi passata a spin_unlock_irqrestore(). Questo significa che lo stesso codice potrà essere utilizzato in un' interruzione hardware (dove le interruzioni sono già disabilitate) e in un softirq (dove la disabilitazione delle interruzioni è richiesta).

Da notare che i softirq (e quindi tasklet e timer) sono eseguiti al ritorno da un' interruzione hardware, quindi spin_lock_irq() interrompe anche questi. Tenuto conto di questo si può dire che spin_lock_irqsave() è la funzione di sincronizzazione più generica e potente.

Sincronizzazione fra due gestori d’ interruzioni hardware

Condividere dati fra due gestori di interruzione hardware è molto raro, ma se succede, dovreste usare spin_lock_irqsave(): è una specificità dell’ architettura il fatto che tutte le interruzioni vengano interrotte quando si eseguono di gestori di interruzioni.

Bigino della sincronizzazione

Pete Zaitcev ci offre il seguente riassunto:

- Se siete in un contesto utente (una qualsiasi chiamata di sistema) e volete sincronizzarvi con altri processi, usate i mutex. Potete trattenere il mutex e dormire (`copy_from_user*(` o `kmalloc(x,GFP_KERNEL)`).
- Altrimenti (== i dati possono essere manipolati da un’ interruzione) usate spin_lock_irqsave() e spin_unlock_irqrestore().
- Evitate di trattenere uno spinlock per più di 5 righe di codice incluse le chiamate a funzione (ad eccezione di quell per l’ accesso come readb()).

Tabella dei requisiti minimi

La tabella seguente illustra i requisiti **minimi** per la sincronizzazione fra diversi contesti. In alcuni casi, lo stesso contesto può essere eseguito solo da un processore per volta, quindi non ci sono requisiti per la sincronizzazione (per esempio, un thread può essere eseguito solo su un processore alla volta, ma se deve condividere dati con un altro thread, allora la sincronizzazione è necessaria).

Ricordatevi il suggerimento qui sopra: potete sempre usare spin_lock_irqsave(), che è un sovrainsieme di tutte le altre funzioni per spinlock.

.	IRQ Handler A	IRQ Handler B	Softir A	Softir B	Taskl A	Taskl B	Time A	Time B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	SLIS	None								
Softirq A	SLI	SLI	SL							
Softirq B	SLI	SLI	SL	SL						
Tasklet A	SLI	SLI	SL	SL	None					
Tasklet B	SLI	SLI	SL	SL	SL	None				
Timer A	SLI	SLI	SL	SL	SL	SL	None			
Timer B	SLI	SLI	SL	SL	SL	SL	SL	None		
User Context A	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBI	SLBI	None	
User Context B	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBI	SLBI	MLI	None

Table: Tabella dei requisiti per la sincronizzazione

SLIS	spin_lock_irqsave
SLI	spin_lock_irq
SL	spin_lock
SLBH	spin_lock_bh
MLI	mutex_lock_interruptible

Table: Legenda per la tabella dei requisiti per la sincronizzazione

Le funzioni *trylock*

Ci sono funzioni che provano a trattenere un *lock* solo una volta e ritornano immediatamente comunicato il successo od il fallimento dell' operazione. Posso essere usate quando non serve accedere ai dati protetti dal *lock* quando qualche altro thread lo sta già facendo trattenendo il *lock*. Potrete acquisire il *lock* più tardi se vi serve accedere ai dati protetti da questo *lock*.

La funzione spin_trylock() non ritenta di acquisire il *lock*, se ci riesce al primo colpo ritorna un valore diverso da zero, altrimenti se fallisce ritorna 0. Questa funzione

può essere utilizzata in un qualunque contesto, ma come spin_lock(): dovete disabilitare i contesti che potrebbero interrompervi e quindi trattenere lo spinlock.

La funzione mutex_trylock() invece di sospendere il vostro processo ritorna un valore diverso da zero se è possibile trattenere il lock al primo colpo, altrimenti se fallisce ritorna 0. Nonostante non dorma, questa funzione non può essere usata in modo sicuro in contesti di interruzione hardware o software.

Esempi più comuni

Guardiamo un semplice esempio: una memoria che associa nomi a numeri. La memoria tiene traccia di quanto spesso viene utilizzato ogni oggetto; quando è piena, l'oggetto meno usato viene eliminato.

Tutto in contesto utente

Nel primo esempio, supponiamo che tutte le operazioni avvengano in contesto utente (in soldoni, da una chiamata di sistema), quindi possiamo dormire. Questo significa che possiamo usare i mutex per proteggere la nostra memoria e tutti gli oggetti che contiene. Ecco il codice:

```
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/mutex.h>
#include <asm/errno.h>

struct object
{
        struct list_head list;
        int id;
        char name[32];
        int popularity;
};

/* Protects the cache, cache_num, and the objects within it */
static DEFINE_MUTEX(cache_lock);
static LIST_HEAD(cache);
static unsigned int cache_num = 0;
#define MAX_CACHE_SIZE 10

/* Must be holding cache_lock */
static struct object *__cache_find(int id)
{
        struct object *i;

        list_for_each_entry(i, &cache, list)
                if (i->id == id) {
                        i->popularity++;
```

(continues on next page)

(continued from previous page)

```
                        return i;
                }
        return NULL;
}

/* Must be holding cache_lock */
static void __cache_delete(struct object *obj)
{
        BUG_ON(!obj);
        list_del(&obj->list);
        kfree(obj);
        cache_num--;
}

/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
        list_add(&obj->list, &cache);
        if (++cache_num > MAX_CACHE_SIZE) {
                struct object *i, *outcast = NULL;
                list_for_each_entry(i, &cache, list) {
                        if (!outcast || i->popularity < outcast->
↪popularity)
                                outcast = i;
                }
                __cache_delete(outcast);
        }
}

int cache_add(int id, const char *name)
{
        struct object *obj;

        if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                return -ENOMEM;

        strscpy(obj->name, name, sizeof(obj->name));
        obj->id = id;
        obj->popularity = 0;

        mutex_lock(&cache_lock);
        __cache_add(obj);
        mutex_unlock(&cache_lock);
        return 0;
}

void cache_delete(int id)
{
        mutex_lock(&cache_lock);
```

(continues on next page)

(continued from previous page)

```
        __cache_delete(__cache_find(id));
        mutex_unlock(&cache_lock);
}

int cache_find(int id, char *name)
{
        struct object *obj;
        int ret = -ENOENT;

        mutex_lock(&cache_lock);
        obj = __cache_find(id);
        if (obj) {
                ret = 0;
                strcpy(name, obj->name);
        }
        mutex_unlock(&cache_lock);
        return ret;
}
```

Da notare che ci assicuriamo sempre di trattenere cache_lock quando aggiungiamo, rimuoviamo od ispezioniamo la memoria: sia la struttura della memoria che il suo contenuto sono protetti dal *lock*. Questo caso è semplice dato che copiamo i dati dall' utente e non permettiamo mai loro di accedere direttamente agli oggetti.

C' è una piccola ottimizzazione qui: nella funzione cache_add() impostiamo i campi dell' oggetto prima di acquisire il *lock*. Questo è sicuro perché nessun altro potrà accedervi finché non lo inseriremo nella memoria.

Accesso dal contesto utente

Ora consideriamo il caso in cui cache_find() può essere invocata dal contesto d' interruzione: sia hardware che software. Un esempio potrebbe essere un timer che elimina oggetti dalla memoria.

Qui di seguito troverete la modifica nel formato *patch*: le righe - sono quelle rimosse, mentre quelle + sono quelle aggiunte.

```
--- cache.c.usercontext 2003-12-09 13:58:54.000000000 +1100
+++ cache.c.interrupt   2003-12-09 14:07:49.000000000 +1100
@@ -12,7 +12,7 @@
         int popularity;
 };

-static DEFINE_MUTEX(cache_lock);
+static DEFINE_SPINLOCK(cache_lock);
 static LIST_HEAD(cache);
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10
@@ -55,6 +55,7 @@
```

(continues on next page)

(continued from previous page)

```
 int cache_add(int id, const char *name)
 {
         struct object *obj;
+        unsigned long flags;

         if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                 return -ENOMEM;
@@ -63,30 +64,33 @@
         obj->id = id;
         obj->popularity = 0;

-        mutex_lock(&cache_lock);
+        spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
         return 0;
 }

 void cache_delete(int id)
 {
-        mutex_lock(&cache_lock);
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
         __cache_delete(__cache_find(id));
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
 }

 int cache_find(int id, char *name)
 {
         struct object *obj;
         int ret = -ENOENT;
+        unsigned long flags;

-        mutex_lock(&cache_lock);
+        spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
         if (obj) {
                 ret = 0;
                 strcpy(name, obj->name);
         }
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
         return ret;
 }
```

Da notare che spin_lock_irqsave() disabiliterà le interruzioni se erano attive, altrimenti non farà niente (quando siamo già in un contesto d' interruzione); dunque

queste funzioni possono essere chiamante in sicurezza da qualsiasi contesto.

Sfortunatamente, cache_add() invoca kmalloc() con l'opzione GFP_KERNEL che è permessa solo in contesto utente. Ho supposto che cache_add() venga chiamata dal contesto utente, altrimenti questa opzione deve diventare un parametro di cache_add().

Esporre gli oggetti al di fuori del file

Se i vostri oggetti contengono più informazioni, potrebbe non essere sufficiente copiare i dati avanti e indietro: per esempio, altre parti del codice potrebbero avere un puntatore a questi oggetti piuttosto che cercarli ogni volta. Questo introduce due problemi.

Il primo problema è che utilizziamo cache_lock per proteggere gli oggetti: dobbiamo renderlo dinamico così che il resto del codice possa usarlo. Questo rende la sincronizzazione più complicata dato che non avviene più in un unico posto.

Il secondo problema è il problema del ciclo di vita: se un'altra struttura mantiene un puntatore ad un oggetto, presumibilmente si aspetta che questo puntatore rimanga valido. Sfortunatamente, questo è garantito solo mentre si trattiene il *lock*, altrimenti qualcuno potrebbe chiamare cache_delete() o peggio, aggiungere un oggetto che riutilizza lo stesso indirizzo.

Dato che c'è un solo *lock*, non potete trattenerlo a vita: altrimenti nessun altro potrà eseguire il proprio lavoro.

La soluzione a questo problema è l'uso di un contatore di riferimenti: chiunque punti ad un oggetto deve incrementare il contatore, e decrementarlo quando il puntatore non viene più usato. Quando il contatore raggiunge lo zero significa che non è più usato e l'oggetto può essere rimosso.

Ecco il codice:

```
--- cache.c.interrupt   2003-12-09 14:25:43.000000000 +1100
+++ cache.c.refcnt  2003-12-09 14:33:05.000000000 +1100
@@ -7,6 +7,7 @@
 struct object
 {
         struct list_head list;
+        unsigned int refcnt;
         int id;
         char name[32];
         int popularity;
@@ -17,6 +18,35 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

+static void __object_put(struct object *obj)
+{
+        if (--obj->refcnt == 0)
+                kfree(obj);
+}
```

(continues on next page)

(continued from previous page)

```
+
+static void __object_get(struct object *obj)
+{
+        obj->refcnt++;
+}
+
+void object_put(struct object *obj)
+{
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
+        __object_put(obj);
+        spin_unlock_irqrestore(&cache_lock, flags);
+}
+
+void object_get(struct object *obj)
+{
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
+        __object_get(obj);
+        spin_unlock_irqrestore(&cache_lock, flags);
+}
+
 /* Must be holding cache_lock */
 static struct object *__cache_find(int id)
 {
@@ -35,6 +65,7 @@
 {
         BUG_ON(!obj);
         list_del(&obj->list);
+        __object_put(obj);
         cache_num--;
 }

@@ -63,6 +94,7 @@
         strscpy(obj->name, name, sizeof(obj->name));
         obj->id = id;
         obj->popularity = 0;
+        obj->refcnt = 1; /* The cache holds a reference */

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
@@ -79,18 +111,15 @@
         spin_unlock_irqrestore(&cache_lock, flags);
 }

-int cache_find(int id, char *name)
+struct object *cache_find(int id)
```

(continues on next page)

(continued from previous page)

```
 {
         struct object *obj;
-        int ret = -ENOENT;
         unsigned long flags;

         spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
-        if (obj) {
-                ret = 0;
-                strcpy(name, obj->name);
-        }
+        if (obj)
+                __object_get(obj);
         spin_unlock_irqrestore(&cache_lock, flags);
-        return ret;
+        return obj;
 }
```

Abbiamo incapsulato il contatore di riferimenti nelle tipiche funzioni di 'get' e 'put' . Ora possiamo ritornare l' oggetto da cache_find() col vantaggio che l' utente può dormire trattenendo l' oggetto (per esempio, copy_to_user() per copiare il nome verso lo spazio utente).

Un altro punto da notare è che ho detto che il contatore dovrebbe incrementarsi per ogni puntatore ad un oggetto: quindi il contatore di riferimenti è 1 quando l' oggetto viene inserito nella memoria. In altre versione il framework non trattiene un riferimento per se, ma diventa più complicato.

Usare operazioni atomiche per il contatore di riferimenti

In sostanza, `atomic_t` viene usato come contatore di riferimenti. Ci sono un certo numbero di operazioni atomiche definite in `include/asm/atomic.h`: queste sono garantite come atomiche su qualsiasi processore del sistema, quindi non sono necessari i *lock*. In questo caso è più semplice rispetto all' uso degli spinlock, benché l' uso degli spinlock sia più elegante per casi non banali. Le funzioni atomic_inc() e atomic_dec_and_test() vengono usate al posto dei tipici operatori di incremento e decremento, e i *lock* non sono più necessari per proteggere il contatore stesso.

```
--- cache.c.refcnt  2003-12-09 15:00:35.000000000 +1100
+++ cache.c.refcnt-atomic   2003-12-11 15:49:42.000000000 +1100
@@ -7,7 +7,7 @@
 struct object
 {
         struct list_head list;
-        unsigned int refcnt;
+        atomic_t refcnt;
         int id;
         char name[32];
         int popularity;
```

(continues on next page)

(continued from previous page)

```
@@ -18,33 +18,15 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

-static void __object_put(struct object *obj)
-{
-        if (--obj->refcnt == 0)
-                kfree(obj);
-}
-
-static void __object_get(struct object *obj)
-{
-        obj->refcnt++;
-}
-
 void object_put(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&cache_lock, flags);
-        __object_put(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        if (atomic_dec_and_test(&obj->refcnt))
+                kfree(obj);
 }

 void object_get(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&cache_lock, flags);
-        __object_get(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        atomic_inc(&obj->refcnt);
 }

 /* Must be holding cache_lock */
@@ -65,7 +47,7 @@
 {
         BUG_ON(!obj);
         list_del(&obj->list);
-        __object_put(obj);
+        object_put(obj);
         cache_num--;
 }

@@ -94,7 +76,7 @@
         strscpy(obj->name, name, sizeof(obj->name));
         obj->id = id;
```

(continues on next page)

(continued from previous page)

```
         obj->popularity = 0;
-        obj->refcnt = 1; /* The cache holds a reference */
+        atomic_set(&obj->refcnt, 1); /* The cache holds a␣
 ↪reference */

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
@@ -119,7 +101,7 @@
         spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
         if (obj)
-                __object_get(obj);
+                object_get(obj);
         spin_unlock_irqrestore(&cache_lock, flags);
         return obj;
 }
```

Proteggere l'oggetto stesso

In questo esempio, assumiamo che gli oggetti (ad eccezione del contatore di riferimenti) non cambino mai dopo la loro creazione. Se vogliamo permettere al nome di cambiare abbiamo tre possibilità:

- Si può togliere static da `cache_lock` e dire agli utenti che devono trattenere il *lock* prima di modificare il nome di un oggetto.
- Si può fornire una funzione cache_obj_rename() che prende il *lock* e cambia il nome per conto del chiamante; si dirà poi agli utenti di usare questa funzione.
- Si può decidere che `cache_lock` protegge solo la memoria stessa, ed un altro *lock* è necessario per la protezione del nome.

Teoricamente, possiamo avere un *lock* per ogni campo e per ogni oggetto. In pratica, le varianti più comuni sono:

- un *lock* che protegge l'infrastruttura (la lista `cache` di questo esempio) e gli oggetti. Questo è quello che abbiamo fatto finora.
- un *lock* che protegge l'infrastruttura (inclusi i puntatori alla lista negli oggetti), e un *lock* nell'oggetto per proteggere il resto dell'oggetto stesso.
- *lock* multipli per proteggere l'infrastruttura (per esempio un *lock* per ogni lista), possibilmente con un *lock* per oggetto.

Qui di seguito un'implementazione con "un lock per oggetto":

```
--- cache.c.refcnt-atomic   2003-12-11 15:50:54.000000000 +1100
+++ cache.c.perobjectlock   2003-12-11 17:15:03.000000000 +1100
@@ -6,11 +6,17 @@

 struct object
 {
```

(continues on next page)

(continued from previous page)

```
+        /* These two protected by cache_lock. */
         struct list_head list;
+        int popularity;
+
         atomic_t refcnt;
+
+        /* Doesn't change once created. */
         int id;
+
+        spinlock_t lock; /* Protects the name */
         char name[32];
-        int popularity;
 };

 static DEFINE_SPINLOCK(cache_lock);
@@ -77,6 +84,7 @@
         obj->id = id;
         obj->popularity = 0;
         atomic_set(&obj->refcnt, 1); /* The cache holds a␣
↪reference */
+        spin_lock_init(&obj->lock);

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
```

Da notare che ho deciso che il contatore di popolarità dovesse essere protetto da `cache_lock` piuttosto che dal *lock* dell'oggetto; questo perché è logicamente parte dell'infrastruttura (come `struct list_head` nell'oggetto). In questo modo, in __cache_add(), non ho bisogno di trattenere il *lock* di ogni oggetto mentre si cerca il meno popolare.

Ho anche deciso che il campo id è immutabile, quindi non ho bisogno di trattenere il lock dell'oggetto quando si usa __cache_find() per leggere questo campo; il *lock* dell'oggetto è usato solo dal chiamante che vuole leggere o scrivere il campo name.

Inoltre, da notare che ho aggiunto un commento che descrive i dati che sono protetti dal *lock*. Questo è estremamente importante in quanto descrive il comportamento del codice, che altrimenti sarebbe di difficile comprensione leggendo solamente il codice. E come dice Alan Cox: "Lock data, not code".

Problemi comuni

Stallo: semplice ed avanzato

Esiste un tipo di baco dove un pezzo di codice tenta di trattenere uno spinlock due volte: questo rimarrà in attesa attiva per sempre aspettando che il *lock* venga rilasciato (in Linux spinlocks, rwlocks e mutex non sono ricorsivi). Questo è facile da diagnosticare: non è uno di quei problemi che ti tengono sveglio 5 notti a parlare da solo.

Un caso un pochino più complesso; immaginate d'avere una spazio condiviso fra un softirq ed il contesto utente. Se usate spin_lock() per proteggerlo, il contesto utente potrebbe essere interrotto da un softirq mentre trattiene il lock, da qui il softirq rimarrà in attesa attiva provando ad acquisire il *lock* già trattenuto nel contesto utente.

Questi casi sono chiamati stalli (*deadlock*), e come mostrato qui sopra, può succedere anche con un solo processore (Ma non sui sistemi monoprocessore perché gli spinlock spariscano quando il kernel è compilato con `CONFIG_SMP=n`. Nonostante ciò, nel secondo caso avrete comunque una corruzione dei dati).

Questi casi sono facili da diagnosticare; sui sistemi multi-processore il supervisione (*watchdog*) o l'opzione di compilazione `DEBUG_SPINLOCK` (`include/linux/spinlock.h`) permettono di scovare immediatamente quando succedono.

Esiste un caso più complesso che è conosciuto come l'abbraccio della morte; questo coinvolge due o più *lock*. Diciamo che avete un vettore di hash in cui ogni elemento è uno spinlock a cui è associata una lista di elementi con lo stesso hash. In un gestore di interruzioni software, dovete modificare un oggetto e spostarlo su un altro hash; quindi dovrete trattenete lo spinlock del vecchio hash e di quello nuovo, quindi rimuovere l'oggetto dal vecchio ed inserirlo nel nuovo.

Qui abbiamo due problemi. Primo, se il vostro codice prova a spostare un oggetto all'interno della stessa lista, otterrete uno stallo visto che tenterà di trattenere lo stesso *lock* due volte. Secondo, se la stessa interruzione software su un altro processore sta tentando di spostare un altro oggetto nella direzione opposta, potrebbe accadere quanto segue:

CPU 1	CPU 2
Trattiene *lock* A -> OK	Trattiene *lock* B -> OK
Trattiene *lock* B -> attesa	Trattiene *lock* A -> attesa

Table: Conseguenze

Entrambe i processori rimarranno in attesa attiva sul *lock* per sempre, aspettando che l'altro lo rilasci. Sembra e puzza come un blocco totale.

Prevenire gli stalli

I libri di testo vi diranno che se trattenete i *lock* sempre nello stesso ordine non avrete mai un simile stallo. La pratica vi dirà che questo approccio non funziona all'ingrandirsi del sistema: quando creo un nuovo *lock* non ne capisco abbastanza del kernel per dire in quale dei 5000 *lock* si incastrerà.

I *lock* migliori sono quelli incapsulati: non vengono esposti nei file di intestazione, e non vengono mai trattenuti fuori dallo stesso file. Potete rileggere questo codice e vedere che non ci sarà mai uno stallo perché non tenterà mai di trattenere un altro *lock* quando lo ha già. Le persone che usano il vostro codice non devono nemmeno sapere che voi state usando dei *lock*.

Un classico problema deriva dall'uso di *callback* e di *hook*: se li chiamate mentre trattenete un *lock*, rischiate uno stallo o un abbraccio della morte (chi lo sa cosa farà una *callback*?).

Ossessiva prevenzione degli stalli

Gli stalli sono un problema, ma non così terribile come la corruzione dei dati. Un pezzo di codice trattiene un *lock* di lettura, cerca in una lista, fallisce nel trovare quello che vuole, quindi rilascia il *lock* di lettura, trattiene un *lock* di scrittura ed inserisce un oggetto; questo genere di codice presenta una corsa critica.

Se non riuscite a capire il perché, per favore state alla larga dal mio codice.

corsa fra temporizzatori: un passatempo del kernel

I temporizzatori potrebbero avere dei problemi con le corse critiche. Considerate una collezione di oggetti (liste, hash, eccetera) dove ogni oggetto ha un temporizzatore che sta per distruggerlo.

Se volete eliminare l'intera collezione (diciamo quando rimuovete un modulo), potreste fare come segue:

```
/* THIS CODE BAD BAD BAD BAD: IF IT WAS ANY WORSE IT WOULD USE
   HUNGARIAN NOTATION */
spin_lock_bh(&list_lock);

while (list) {
        struct foo *next = list->next;
        del_timer(&list->timer);
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

Primo o poi, questo esploderà su un sistema multiprocessore perché un temporizzatore potrebbe essere già partiro prima di spin_lock_bh(), e prenderà il *lock* solo

dopo spin_unlock_bh(), e cercherà di eliminare il suo oggetto (che però è già stato eliminato).

Questo può essere evitato controllando il valore di ritorno di del_timer(): se ritorna 1, il temporizzatore è stato già rimosso. Se 0, significa (in questo caso) che il temporizzatore è in esecuzione, quindi possiamo fare come segue:

```
retry:
        spin_lock_bh(&list_lock);

        while (list) {
                struct foo *next = list->next;
                if (!del_timer(&list->timer)) {
                        /* Give timer a chance to delete this */
                        spin_unlock_bh(&list_lock);
                        goto retry;
                }
                kfree(list);
                list = next;
        }

        spin_unlock_bh(&list_lock);
```

Un altro problema è l'eliminazione dei temporizzatori che si riavviano da soli (chiamando add_timer() alla fine della loro esecuzione). Dato che questo è un problema abbastanza comune con una propensione alle corse critiche, dovreste usare del_timer_sync() (`include/linux/timer.h`) per gestire questo caso. Questa ritorna il numero di volte che il temporizzatore è stato interrotto prima che fosse in grado di fermarlo senza che si riavviasse.

Velocità della sincronizzazione

Ci sono tre cose importanti da tenere in considerazione quando si valuta la velocità d'esecuzione di un pezzo di codice che necessita di sincronizzazione. La prima è la concorrenza: quante cose rimangono in attesa mentre qualcuno trattiene un *lock*. La seconda è il tempo necessario per acquisire (senza contese) e rilasciare un *lock*. La terza è di usare meno *lock* o di più furbi. Immagino che i *lock* vengano usati regolarmente, altrimenti, non sareste interessati all'efficienza.

La concorrenza dipende da quanto a lungo un *lock* è trattenuto: dovreste trattenere un *lock* solo il tempo minimo necessario ma non un istante in più. Nella memoria dell'esempio precedente, creiamo gli oggetti senza trattenere il *lock*, poi acquisiamo il *lock* quando siamo pronti per inserirlo nella lista.

Il tempo di acquisizione di un *lock* dipende da quanto danno fa l'operazione sulla *pipeline* (ovvero stalli della *pipeline*) e quant'è probabile che il processore corrente sia stato anche l'ultimo ad acquisire il *lock* (in pratica, il *lock* è nella memoria cache del processore corrente?): su sistemi multi-processore questa probabilità precipita rapidamente. Consideriamo un processore Intel Pentium III a 700Mhz: questo esegue un'istruzione in 0.7ns, un incremento atomico richiede 58ns, acquisire un *lock* che è nella memoria cache del processore richiede 160ns, e un

trasferimento dalla memoria cache di un altro processore richiede altri 170/360ns (Leggetevi l'articolo di Paul McKenney's Linux Journal RCU article).

Questi due obiettivi sono in conflitto: trattenere un *lock* per il minor tempo possibile potrebbe richiedere la divisione in più *lock* per diverse parti (come nel nostro ultimo esempio con un *lock* per ogni oggetto), ma questo aumenta il numero di acquisizioni di *lock*, ed il risultato spesso è che tutto è più lento che con un singolo *lock*. Questo è un altro argomento in favore della semplicità quando si parla di sincronizzazione.

Il terzo punto è discusso di seguito: ci sono alcune tecniche per ridurre il numero di sincronizzazioni che devono essere fatte.

Read/Write Lock Variants

Sia gli spinlock che i mutex hanno una variante per la lettura/scrittura (read/write): `rwlock_t` e `struct rw_semaphore`. Queste dividono gli utenti in due categorie: i lettori e gli scrittori. Se state solo leggendo i dati, potete acquisire il *lock* di lettura, ma per scrivere avrete bisogno del *lock* di scrittura. Molti possono trattenere il *lock* di lettura, ma solo uno scrittore alla volta può trattenere quello di scrittura.

Se il vostro codice si divide chiaramente in codice per lettori e codice per scrittori (come nel nostro esempio), e il *lock* dei lettori viene trattenuto per molto tempo, allora l'uso di questo tipo di *lock* può aiutare. Questi sono leggermente più lenti rispetto alla loro versione normale, quindi nella pratica l'uso di `rwlock_t` non ne vale la pena.

Evitare i *lock*: Read Copy Update

Esiste un metodo di sincronizzazione per letture e scritture detto Read Copy Update. Con l'uso della tecnica RCU, i lettori possono scordarsi completamente di trattenere i *lock*; dato che nel nostro esempio ci aspettiamo d'avere più lettore che scrittori (altrimenti questa memoria sarebbe uno spreco) possiamo dire che questo meccanismo permette un'ottimizzazione.

Come facciamo a sbarazzarci dei *lock* di lettura? Sbarazzarsi dei *lock* di lettura significa che uno scrittore potrebbe cambiare la lista sotto al naso dei lettori. Questo è abbastanza semplice: possiamo leggere una lista concatenata se lo scrittore aggiunge elementi alla fine e con certe precauzioni. Per esempio, aggiungendo `new` ad una lista concatenata chiamata `list`:

```
new->next = list->next;
wmb();
list->next = new;
```

La funzione wmb() è una barriera di sincronizzazione delle scritture. Questa garantisce che la prima operazione (impostare l'elemento `next` del nuovo elemento) venga completata e vista da tutti i processori prima che venga eseguita la seconda operazione (che sarebbe quella di mettere il nuovo elemento nella lista). Questo è importante perché i moderni compilatori ed i moderni processori possono, entrambe, riordinare le istruzioni se non vengono istruiti altrimenti:

vogliamo che i lettori non vedano completamente il nuovo elemento; oppure che lo vedano correttamente e quindi il puntatore `next` deve puntare al resto della lista.

Fortunatamente, c'è una funzione che fa questa operazione sulle liste `struct list_head`: list_add_rcu() (`include/linux/list.h`).

Rimuovere un elemento dalla lista è anche più facile: sostituiamo il puntatore al vecchio elemento con quello del suo successore, e i lettori vedranno l'elemento o lo salteranno.

```
list->next = old->next;
```

La funzione list_del_rcu() (`include/linux/list.h`) fa esattamente questo (la versione normale corrompe il vecchio oggetto, e non vogliamo che accada).

Anche i lettori devono stare attenti: alcuni processori potrebbero leggere attraverso il puntatore `next` il contenuto dell'elemento successivo troppo presto, ma non accorgersi che il contenuto caricato è sbagliato quando il puntatore `next` viene modificato alla loro spalle. Ancora una volta c'è una funzione che viene in vostro aiuto list_for_each_entry_rcu() (`include/linux/list.h`). Ovviamente, gli scrittori possono usare list_for_each_entry() dato che non ci possono essere due scrittori in contemporanea.

Il nostro ultimo dilemma è il seguente: quando possiamo realmente distruggere l'elemento rimosso? Ricordate, un lettore potrebbe aver avuto accesso a questo elemento proprio ora: se eliminiamo questo elemento ed il puntatore `next` cambia, il lettore salterà direttamente nella spazzatura e scoppierà. Dobbiamo aspettare finché tutti i lettori che stanno attraversando la lista abbiano finito. Utilizziamo call_rcu() per registrare una funzione di richiamo che distrugga l'oggetto quando tutti i lettori correnti hanno terminato. In alternative, potrebbe essere usata la funzione synchronize_rcu() che blocca l'esecuzione finché tutti i lettori non terminano di ispezionare la lista.

Ma come fa l'RCU a sapere quando i lettori sono finiti? Il meccanismo è il seguente: innanzi tutto i lettori accedono alla lista solo fra la coppia rcu_read_lock()/rcu_read_unlock() che disabilita la prelazione così che i lettori non vengano sospesi mentre stanno leggendo la lista.

Poi, l'RCU aspetta finché tutti i processori non abbiano dormito almeno una volta; a questo punto, dato che i lettori non possono dormire, possiamo dedurre che un qualsiasi lettore che abbia consultato la lista durante la rimozione abbia già terminato, quindi la *callback* viene eseguita. Il vero codice RCU è un po' più ottimizzato di così, ma questa è l'idea di fondo.

```
--- cache.c.perobjectlock   2003-12-11 17:15:03.000000000 +1100
+++ cache.c.rcupdate    2003-12-11 17:55:14.000000000 +1100
@@ -1,15 +1,18 @@
 #include <linux/list.h>
 #include <linux/slab.h>
 #include <linux/string.h>
+#include <linux/rcupdate.h>
 #include <linux/mutex.h>
 #include <asm/errno.h>
```

(continues on next page)

(continued from previous page)

```
 struct object
 {
-        /* These two protected by cache_lock. */
+        /* This is protected by RCU */
         struct list_head list;
         int popularity;

+        struct rcu_head rcu;
+
         atomic_t refcnt;

         /* Doesn't change once created. */
@@ -40,7 +43,7 @@
 {
         struct object *i;

-        list_for_each_entry(i, &cache, list) {
+        list_for_each_entry_rcu(i, &cache, list) {
                 if (i->id == id) {
                         i->popularity++;
                         return i;
@@ -49,19 +52,25 @@
         return NULL;
 }

+/* Final discard done once we know no readers are looking. */
+static void cache_delete_rcu(void *arg)
+{
+        object_put(arg);
+}
+
 /* Must be holding cache_lock */
 static void __cache_delete(struct object *obj)
 {
         BUG_ON(!obj);
-        list_del(&obj->list);
-        object_put(obj);
+        list_del_rcu(&obj->list);
         cache_num--;
+        call_rcu(&obj->rcu, cache_delete_rcu);
 }

 /* Must be holding cache_lock */
 static void __cache_add(struct object *obj)
 {
-        list_add(&obj->list, &cache);
+        list_add_rcu(&obj->list, &cache);
         if (++cache_num > MAX_CACHE_SIZE) {
                 struct object *i, *outcast = NULL;
```

(continues on next page)

(continued from previous page)

```
                 list_for_each_entry(i, &cache, list) {
@@ -104,12 +114,11 @@
 struct object *cache_find(int id)
 {
         struct object *obj;
-        unsigned long flags;

-        spin_lock_irqsave(&cache_lock, flags);
+        rcu_read_lock();
         obj = __cache_find(id);
         if (obj)
                 object_get(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        rcu_read_unlock();
         return obj;
 }
```

Da notare che i lettori modificano il campo popularity nella funzione __cache_find(), e ora non trattiene alcun *lock*. Una soluzione potrebbe essere quella di rendere la variabile `atomic_t`, ma per l'uso che ne abbiamo fatto qui, non ci interessano queste corse critiche perché un risultato approssimativo è comunque accettabile, quindi non l'ho cambiato.

Il risultato è che la funzione cache_find() non ha bisogno di alcuna sincronizzazione con le altre funzioni, quindi è veloce su un sistema multi-processore tanto quanto lo sarebbe su un sistema mono-processore.

Esiste un'ulteriore ottimizzazione possibile: vi ricordate il codice originale della nostra memoria dove non c'erano contatori di riferimenti e il chiamante semplicemente tratteneva il *lock* prima di accedere ad un oggetto? Questo è ancora possibile: se trattenete un *lock* nessuno potrà cancellare l'oggetto, quindi non avete bisogno di incrementare e decrementare il contatore di riferimenti.

Ora, dato che il '*lock* di lettura'di un RCU non fa altro che disabilitare la prelazione, un chiamante che ha sempre la prelazione disabilitata fra le chiamate cache_find() e object_put() non necessita di incrementare e decrementare il contatore di riferimenti. Potremmo esporre la funzione __cache_find() dichiarandola non-static, e quel chiamante potrebbe usare direttamente questa funzione.

Il beneficio qui sta nel fatto che il contatore di riferimenti no viene scritto: l'oggetto non viene alterato in alcun modo e quindi diventa molto più veloce su sistemi molti-processore grazie alla loro memoria cache.

Dati per processore

Un' altra tecnica comunemente usata per evitare la sincronizzazione è quella di duplicare le informazioni per ogni processore. Per esempio, se volete avere un contatore di qualcosa, potreste utilizzare uno spinlock ed un singolo contatore. Facile e pulito.

Se questo dovesse essere troppo lento (solitamente non lo è, ma se avete dimostrato che lo è devvero), potreste usare un contatore per ogni processore e quindi non sarebbe più necessaria la mutua esclusione. Vedere DEFINE_PER_CPU(), get_cpu_var() e put_cpu_var() (`include/linux/percpu.h`).

Il tipo di dato `local_t`, la funzione cpu_local_inc() e tutte le altre funzioni associate, sono di particolare utilità per semplici contatori per-processore; su alcune architetture sono anche più efficienti (`include/asm/local.h`).

Da notare che non esiste un modo facile ed affidabile per ottenere il valore di un simile contatore senza introdurre altri *lock*. In alcuni casi questo non è un problema.

Dati che sono usati prevalentemente dai gestori d' interruzioni

Se i dati vengono utilizzati sempre dallo stesso gestore d' interruzioni, allora i *lock* non vi servono per niente: il kernel già vi garantisce che il gestore d' interruzione non verrà eseguito in contemporanea su diversi processori.

Manfred Spraul fa notare che potreste comunque comportarvi così anche se i dati vengono occasionalmente utilizzati da un contesto utente o da un' interruzione software. Il gestore d' interruzione non utilizza alcun *lock*, e tutti gli altri accessi verranno fatti così:

```
spin_lock(&lock);
disable_irq(irq);
...
enable_irq(irq);
spin_unlock(&lock);
```

La funzione disable_irq() impedisce al gestore d' interruzioni d' essere eseguito (e aspetta che finisca nel caso fosse in esecuzione su un altro processore). Lo spinlock, invece, previene accessi simultanei. Naturalmente, questo è più lento della semplice chiamata spin_lock_irq(), quindi ha senso solo se questo genere di accesso è estremamente raro.

Quali funzioni possono essere chiamate in modo sicuro dalle interruzioni?

Molte funzioni del kernel dormono (in sostanza, chiamano schedule()) direttamente od indirettamente: non potete chiamarle se trattenere uno spinlock o avete la prelazione disabilitata, mai. Questo significa che dovete necessariamente essere nel contesto utente: chiamarle da un contesto d'interruzione è illegale.

Alcune funzioni che dormono

Le più comuni sono elencate qui di seguito, ma solitamente dovete leggere il codice per scoprire se altre chiamate sono sicure. Se chiunque altro le chiami dorme, allora dovreste poter dormire anche voi. In particolar modo, le funzioni di registrazione e deregistrazione solitamente si aspettano d'essere chiamante da un contesto utente e quindi che possono dormire.

- Accessi allo spazio utente:
  - copy_from_user()
  - copy_to_user()
  - get_user()
  - put_user()
- kmalloc(GFP_KERNEL) <kmalloc>`
- mutex_lock_interruptible() and mutex_lock()

  C'è anche mutex_trylock() che però non dorme. Comunque, non deve essere usata in un contesto d'interruzione dato che la sua implementazione non è sicura in quel contesto. Anche mutex_unlock() non dorme mai. Non può comunque essere usata in un contesto d'interruzione perché un mutex deve essere rilasciato dallo stesso processo che l'ha acquisito.

Alcune funzioni che non dormono

Alcune funzioni possono essere chiamate tranquillamente da qualsiasi contesto, o trattenendo un qualsiasi *lock*.

- printk()
- kfree()
- add_timer() e del_timer()

Riferimento per l’ API dei Mutex

mutex_init

mutex_init (mutex)

initialize the mutex

Parameters

mutex
the mutex to be initialized

Description

Initialize the mutex to unlocked state.

It is not allowed to initialize an already locked mutex.

bool **mutex_is_locked**(struct mutex *lock)

is the mutex locked

Parameters

struct mutex *lock
the mutex to be queried

Description

Returns true if the mutex is locked, false if unlocked.

enum mutex_trylock_recursive_enum **mutex_trylock_recursive**(struct mutex *lock)

trylock variant that allows recursive locking

Parameters

struct mutex *lock
mutex to be locked

Description

This function should not be used, _ever_. It is purely for hysterical GEM raisins, and once those are gone this will be removed.

Return

- MUTEX_TRYLOCK_FAILED - trylock failed,
- MUTEX_TRYLOCK_SUCCESS - lock acquired,
- MUTEX_TRYLOCK_RECURSIVE - we already owned the lock.

void **mutex_lock**(struct mutex *lock)

acquire the mutex

Parameters

struct mutex *lock
the mutex to be acquired

Description

Lock the mutex exclusively for this task. If the mutex is not available right now, it will sleep until it can get it.

The mutex must later on be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locked. memset()-ing the mutex to 0 is not allowed.

(The CONFIG_DEBUG_MUTEXES .config option turns on debugging checks that will enforce the restrictions and will also do deadlock debugging)

This function is similar to (but not equivalent to) down().

void **mutex_unlock**(struct mutex *lock)

release the mutex

Parameters

struct mutex *lock
the mutex to be released

Description

Unlock a mutex that has been locked by this task previously.

This function must not be used in interrupt context. Unlocking of a not locked mutex is not allowed.

This function is similar to (but not equivalent to) up().

void **ww_mutex_unlock**(struct ww_mutex *lock)

release the w/w mutex

Parameters

struct ww_mutex *lock
the mutex to be released

Description

Unlock a mutex that has been locked by this task previously with any of the ww_mutex_lock* functions (with or without an acquire context). It is forbidden to release the locks after releasing the acquire context.

This function must not be used in interrupt context. Unlocking of a unlocked mutex is not allowed.

int **mutex_lock_interruptible**(struct mutex *lock)

Acquire the mutex, interruptible by signals.

Parameters

struct mutex *lock
The mutex to be acquired.

Description

Lock the mutex like mutex_lock(). If a signal is delivered while the process is sleeping, this function will return without acquiring the mutex.

Context

Process context.

Return

0 if the lock was successfully acquired or `-EINTR` if a signal arrived.

int **mutex_lock_killable**(struct mutex *lock)

Acquire the mutex, interruptible by fatal signals.

Parameters

struct mutex *lock
The mutex to be acquired.

Description

Lock the mutex like mutex_lock(). If a signal which will be fatal to the current process is delivered while the process is sleeping, this function will return without acquiring the mutex.

Context

Process context.

Return

0 if the lock was successfully acquired or `-EINTR` if a fatal signal arrived.

void **mutex_lock_io**(struct mutex *lock)

Acquire the mutex and mark the process as waiting for I/O

Parameters

struct mutex *lock
The mutex to be acquired.

Description

Lock the mutex like mutex_lock(). While the task is waiting for this mutex, it will be accounted as being in the IO wait state by the scheduler.

Context

Process context.

int **mutex_trylock**(struct mutex *lock)

try to acquire the mutex, without waiting

Parameters

struct mutex *lock
the mutex to be acquired

Description

Try to acquire the mutex atomically. Returns 1 if the mutex has been acquired successfully, and 0 on contention.

This function must not be used in interrupt context. The mutex must be released by the same task that acquired it.

NOTE

this function follows the spin_trylock() convention, so it is negated from the down_trylock() return values! Be careful about this when converting semaphore users to mutexes.

int **atomic_dec_and_mutex_lock**(atomic_t *cnt, struct mutex *lock)
return holding mutex if we dec to 0

Parameters

atomic_t *cnt
the atomic which we are to dec

struct mutex *lock
the mutex to return holding if we dec to 0

Description

return true and hold lock if we dec to 0, return false otherwise

Riferimento per l'API dei Futex

struct **futex_q**
The hashed futex queue entry, one per waiting task

Definition

```
struct futex_q {
  struct plist_node list;
  struct task_struct *task;
  spinlock_t *lock_ptr;
  union futex_key key;
  struct futex_pi_state *pi_state;
  struct rt_mutex_waiter *rt_waiter;
  union futex_key *requeue_pi_key;
  u32 bitset;
};
```

Members

list
priority-sorted list of tasks waiting on this futex

task
the task waiting on the futex

lock_ptr
the hash bucket lock

key
the key the futex is hashed on

pi_state
optional priority inheritance state

rt_waiter
rt_waiter storage for use with requeue_pi

requeue_pi_key
the requeue_pi target futex key

bitset
bitset for the optional bitmasked wakeup

Description

We use this hashed waitqueue, instead of a normal wait_queue_entry_t, so we can wake only the relevant ones (hashed queues may be shared).

A futex_q has a woken state, just like tasks have TASK_RUNNING. It is considered woken when plist_node_empty(`q->list`) || q->lock_ptr == 0. The order of wakeup is always to make the first condition true, then the second.

PI futexes are typically woken before they are removed from the hash list via the rt_mutex code. See unqueue_me_pi().

struct futex_hash_bucket *__hash_futex__(union futex_key *key)
Return the hash bucket in the global hash

Parameters

union futex_key *key
Pointer to the futex key for which the hash is calculated

Description

We hash on the keys returned from get_futex_key (see below) and return the corresponding hash bucket in the global hash.

int __match_futex__(union futex_key *key1, union futex_key *key2)
Check whether two futex keys are equal

Parameters

union futex_key *key1
Pointer to key1

union futex_key *key2
Pointer to key2

Description

Return 1 if two futex_keys are equal, 0 otherwise.

struct hrtimer_sleeper *__futex_setup_timer__(ktime_t *time, struct hrtimer_sleeper *timeout, int flags, u64 range_ns)
set up the sleeping hrtimer.

Parameters

ktime_t *time
ptr to the given timeout value

struct hrtimer_sleeper *timeout
the hrtimer_sleeper structure to be set up

int flags
futex flags

u64 range_ns
optional range in ns

Return

Initialized hrtimer_sleeper structure or NULL if no timeout
value given

int **get_futex_key**(u32 __user *uaddr, bool fshared, union futex_key *key, enum futex_access rw)

Get parameters which are the keys for a futex

Parameters

u32 __user *uaddr
virtual address of the futex

bool fshared
false for a PROCESS_PRIVATE futex, true for PROCESS_SHARED

union futex_key *key
address where result is stored.

enum futex_access rw
mapping needs to be read/write (values: FUTEX_READ, FUTEX_WRITE)

Return

a negative error code or 0

Description

The key words are stored in **key** on success.

For shared mappings (when **fshared**), the key is:

(inode->i_sequence, page->index, offset_within_page)

[also see get_inode_sequence_number()]

For private mappings (or when **!fshared**), the key is:

(current->mm, address, 0)

This allows (cross process, where applicable) identification of the futex without keeping the page pinned for the duration of the FUTEX_WAIT.

lock_page() might sleep, the caller should not hold a spinlock.

int **fault_in_user_writeable**(u32 __user *uaddr)

Fault in user address and verify RW access

Parameters

u32 __user *uaddr
pointer to faulting user space address

Description

Slow path to fixup the fault we just took in the atomic write access to **uaddr**.

We have no generic implementation of a non-destructive write to the user address. We know that we faulted in the atomic pagefault disabled section so we can as well avoid the #PF overhead by calling get_user_pages() right away.

struct *futex_q* ***futex_top_waiter**(struct futex_hash_bucket *hb, union futex_key *key)

Return the highest priority waiter on a futex

Parameters

struct futex_hash_bucket *hb
the hash bucket the futex_q's reside in

union futex_key *key
the futex key (to distinguish it from other futex futex_q's)

Description

Must be called with the hb lock held.

void **wait_for_owner_exiting**(int ret, struct task_struct *exiting)

Block until the owner has exited

Parameters

int ret
owner's current futex lock status

struct task_struct *exiting
Pointer to the exiting task

Description

Caller must hold a refcount on **exiting**.

int **futex_lock_pi_atomic**(u32 __user *uaddr, struct futex_hash_bucket *hb, union futex_key *key, struct futex_pi_state **ps, struct task_struct *task, struct task_struct **exiting, int set_waiters)

Atomic work required to acquire a pi aware futex

Parameters

u32 __user *uaddr
the pi futex user address

struct futex_hash_bucket *hb
the pi futex hash bucket

union futex_key *key
the futex key associated with uaddr and hb

struct futex_pi_state **ps
the pi_state pointer where we store the result of the lookup

struct task_struct *task
the task to perform the atomic lock work for. This will be "current" except in the case of requeue pi.

struct task_struct **exiting
Pointer to store the task pointer of the owner task which is in the middle of exiting

int set_waiters
force setting the FUTEX_WAITERS bit (1) or not (0)

Return

- 0 - ready to wait;
- 1 - acquired the lock;
- <0 - error

Description

The hb->lock and futex_key refs shall be held by the caller.

exiting is only set when the return value is -EBUSY. If so, this holds a refcount on the exiting task on return and the caller needs to drop it after waiting for the exit to complete.

void **__unqueue_futex**(struct *futex_q* *q)
Remove the futex_q from its futex_hash_bucket

Parameters

struct futex_q *q
The futex_q to unqueue

Description

The q->lock_ptr must not be NULL and must be held by the caller.

void **requeue_futex**(struct *futex_q* *q, struct futex_hash_bucket *hb1, struct futex_hash_bucket *hb2, union futex_key *key2)
Requeue a futex_q from one hb to another

Parameters

struct futex_q *q
the futex_q to requeue

struct futex_hash_bucket *hb1
the source hash_bucket

struct futex_hash_bucket *hb2
the target hash_bucket

union futex_key *key2
the new key for the requeued futex_q

void **requeue_pi_wake_futex**(struct *futex_q* *q, union futex_key *key, struct futex_hash_bucket *hb)
Wake a task that acquired the lock during requeue

Parameters

struct futex_q *q
the futex_q

union futex_key *key
the key of the requeue target futex

struct futex_hash_bucket *hb
the hash_bucket of the requeue target futex

Description

During futex_requeue, with requeue_pi=1, it is possible to acquire the target futex if it is uncontended or via a lock steal. Set the futex_q key to the requeue target futex so the waiter can detect the wakeup on the right futex, but remove it from the hb and NULL the rt_waiter so it can detect atomic lock acquisition. Set the q->lock_ptr to the requeue target hb->lock to protect access to the pi_state to fixup the owner later. Must be called with both q->lock_ptr and hb->lock held.

int **futex_proxy_trylock_atomic**(u32 __user *pifutex, struct futex_hash_bucket *hb1, struct futex_hash_bucket *hb2, union futex_key *key1, union futex_key *key2, struct futex_pi_state **ps, struct task_struct **exiting, int set_waiters)

Attempt an atomic lock for the top waiter

Parameters

u32 __user *pifutex
the user address of the to futex

struct futex_hash_bucket *hb1
the from futex hash bucket, must be locked by the caller

struct futex_hash_bucket *hb2
the to futex hash bucket, must be locked by the caller

union futex_key *key1
the from futex key

union futex_key *key2
the to futex key

struct futex_pi_state **ps
address to store the pi_state pointer

struct task_struct **exiting
Pointer to store the task pointer of the owner task which is in the middle of exiting

int set_waiters
force setting the FUTEX_WAITERS bit (1) or not (0)

Description

Try and get the lock on behalf of the top waiter if we can do it atomically. Wake the top waiter if we succeed. If the caller specified set_waiters, then direct futex_lock_pi_atomic() to force setting the FUTEX_WAITERS bit. hb1 and hb2 must be held by the caller.

exiting is only set when the return value is -EBUSY. If so, this holds a refcount on the exiting task on return and the caller needs to drop it after waiting for the exit to complete.

Return

- 0 - failed to acquire the lock atomically;
- >0 - acquired the lock, return value is vpid of the top_waiter
- <0 - error

int **futex_requeue**(u32 __user *uaddr1, unsigned int flags, u32 __user *uaddr2, int nr_wake, int nr_requeue, u32 *cmpval, int requeue_pi)

Requeue waiters from uaddr1 to uaddr2

Parameters

u32 __user *uaddr1
source futex user address

unsigned int flags
futex flags (FLAGS_SHARED, etc.)

u32 __user *uaddr2
target futex user address

int nr_wake
number of waiters to wake (must be 1 for requeue_pi)

int nr_requeue
number of waiters to requeue (0-INT_MAX)

u32 *cmpval
uaddr1 expected value (or `NULL`)

int requeue_pi
if we are attempting to requeue from a non-pi futex to a pi futex (pi to pi requeue is not supported)

Description

Requeue waiters on uaddr1 to uaddr2. In the requeue_pi case, try to acquire uaddr2 atomically on behalf of the top waiter.

Return

- >=0 - on success, the number of tasks requeued or woken;
- <0 - on error

void **queue_me**(struct *futex_q* *q, struct futex_hash_bucket *hb)

Enqueue the futex_q on the futex_hash_bucket

Parameters

struct futex_q *q
The futex_q to enqueue

struct futex_hash_bucket *hb
The destination hash bucket

Description

The hb->lock must be held by the caller, and is released here. A call to queue_me() is typically paired with exactly one call to unqueue_me(). The exceptions involve

the PI related operations, which may use unqueue_me_pi() or nothing if the unqueue is done as part of the wake process and the unqueue state is implicit in the state of woken task (see futex_wait_requeue_pi() for an example).

int **unqueue_me**(struct *futex_q* *q)

Remove the futex_q from its futex_hash_bucket

Parameters

struct futex_q *q
The futex_q to unqueue

Description

The q->lock_ptr must not be held by the caller. A call to unqueue_me() must be paired with exactly one earlier call to queue_me().

Return

- 1 - if the futex_q was still queued (and we removed unqueued it);
- 0 - if the futex_q was already removed by the waking thread

int **fixup_owner**(u32 __user *uaddr, struct *futex_q* *q, int locked)

Post lock pi_state and corner case management

Parameters

u32 __user *uaddr
user address of the futex

struct futex_q *q
futex_q (contains pi_state and access to the rt_mutex)

int locked
if the attempt to take the rt_mutex succeeded (1) or not (0)

Description

After attempting to lock an rt_mutex, this function is called to cleanup the pi_state owner as well as handle race conditions that may allow us to acquire the lock. Must be called with the hb lock held.

Return

- 1 - success, lock taken;
- 0 - success, lock not taken;
- <0 - on error (-EFAULT)

void **futex_wait_queue_me**(struct futex_hash_bucket *hb, struct *futex_q* *q,
struct hrtimer_sleeper *timeout)

queue_me() and wait for wakeup, timeout, or signal

Parameters

struct futex_hash_bucket *hb
the futex hash bucket, must be locked by the caller

struct futex_q *q
the futex_q to queue up on

struct hrtimer_sleeper *timeout
the prepared hrtimer_sleeper, or null for no timeout

int **futex_wait_setup**(u32 __user *uaddr, u32 val, unsigned int flags, struct *futex_q* *q, struct futex_hash_bucket **hb)

Prepare to wait on a futex

Parameters

u32 __user *uaddr
the futex userspace address

u32 val
the expected value

unsigned int flags
futex flags (FLAGS_SHARED, etc.)

struct futex_q *q
the associated futex_q

struct futex_hash_bucket **hb
storage for hash_bucket pointer to be returned to caller

Description

Setup the futex_q and locate the hash_bucket. Get the futex value and compare it with the expected value. Handle atomic faults internally. Return with the hb lock held and a q.key reference on success, and unlocked with no q.key reference on failure.

Return

- 0 - uaddr contains val and hb has been locked;
- <1 - -EFAULT or -EWOULDBLOCK (uaddr does not contain val) and hb is unlocked

int **handle_early_requeue_pi_wakeup**(struct futex_hash_bucket *hb, struct *futex_q* *q, union futex_key *key2, struct hrtimer_sleeper *timeout)

Detect early wakeup on the initial futex

Parameters

struct futex_hash_bucket *hb
the hash_bucket futex_q was original enqueued on

struct futex_q *q
the futex_q woken while waiting to be requeued

union futex_key *key2
the futex_key of the requeue target futex

struct hrtimer_sleeper *timeout
the timeout associated with the wait (NULL if none)

Description

Detect if the task was woken on the initial futex as opposed to the requeue target futex. If so, determine if it was a timeout or a signal that caused the wakeup and

return the appropriate error code to the caller. Must be called with the hb lock held.

Return

- 0 = no early wakeup detected;
- <0 = -ETIMEDOUT or -ERESTARTNOINTR

int **futex_wait_requeue_pi**(u32 __user *uaddr, unsigned int flags, u32 val, ktime_t *abs_time, u32 bitset, u32 __user *uaddr2)

Wait on uaddr and take uaddr2

Parameters

u32 __user *uaddr
the futex we initially wait on (non-pi)

unsigned int flags
futex flags (FLAGS_SHARED, FLAGS_CLOCKRT, etc.), they must be the same type, no requeueing from private to shared, etc.

u32 val
the expected value of uaddr

ktime_t *abs_time
absolute timeout

u32 bitset
32 bit wakeup bitset set by userspace, defaults to all

u32 __user *uaddr2
the pi futex we will take prior to returning to user-space

Description

The caller will wait on uaddr and will be requeued by futex_requeue() to uaddr2 which must be PI aware and unique from uaddr. Normal wakeup will wake on uaddr2 and complete the acquisition of the rt_mutex prior to returning to userspace. This ensures the rt_mutex maintains an owner when it has waiters; without one, the pi logic would not know which task to boost/deboost, if there was a need to.

We call schedule in futex_wait_queue_me() when we enqueue and return there via the following– 1) wakeup on uaddr2 after an atomic lock acquisition by futex_requeue() 2) wakeup on uaddr2 after a requeue 3) signal 4) timeout

If 3, cleanup and return -ERESTARTNOINTR.

If 2, we may then block on trying to take the rt_mutex and return via: 5) successful lock 6) signal 7) timeout 8) other lock acquisition failure

If 6, return -EWOULDBLOCK (restarting the syscall would do the same).

If 4 or 7, we cleanup and return with -ETIMEDOUT.

Return

- 0 - On success;
- <0 - On error

long **sys_set_robust_list**(struct robust_list_head __user *head, size_t len)
Set the robust-futex list head of a task

Parameters

struct robust_list_head __user * head
pointer to the list-head

size_t len
length of the list-head, as userspace expects

long **sys_get_robust_list**(int pid, struct robust_list_head __user *__user *head_ptr, size_t __user *len_ptr)
Get the robust-futex list head of a task

Parameters

int pid
pid of the process [zero for current task]

struct robust_list_head __user * __user * head_ptr
pointer to a list-head pointer, the kernel fills it in

size_t __user * len_ptr
pointer to a length field, the kernel fills in the header size

void **futex_exit_recursive**(struct task_struct *tsk)
Set the tasks futex state to FUTEX_STATE_DEAD

Parameters

struct task_struct *tsk
task to set the state on

Description

Set the futex exit state of the task lockless. The futex waiter code observes that state when a task is exiting and loops until the task has actually finished the futex cleanup. The worst case for this is that the waiter runs through the wait loop until the state becomes visible.

This is called from the recursive fault handling path in do_exit().

This is best effort. Either the futex exit code has run already or not. If the OWNER_DIED bit has been set on the futex then the waiter can take it over. If not, the problem is pushed back to user space. If the futex exit code did not run yet, then an already queued waiter might block forever, but there is nothing which can be done about that.

Approfondimenti

- `Documentation/locking/spinlocks.rst`: la guida di Linus Torvalds agli spinlock del kernel.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.

  L'introduzione alla sincronizzazione a livello di kernel di Curt Schimmel è davvero ottima (non è scritta per Linux, ma approssimativamente si adatta a tutte le situazioni). Il libro è costoso, ma vale ogni singolo spicciolo per capire la sincronizzazione nei sistemi multi-processore. [ISBN: 0201633388]

Ringraziamenti

Grazie a Telsa Gwynne per aver formattato questa guida in DocBook, averla pulita e aggiunto un po' di stile.

Grazie a Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul, Tim Waugh, Pete Zaitcev, James Morris, Robert Love, Paul McKenney, John Ashby per aver revisionato, corretto, maledetto e commentato.

Grazie alla congrega per non aver avuto alcuna influenza su questo documento.

Glossario

prelazione
: Prima del kernel 2.5, o quando `CONFIG_PREEMPT` non è impostato, i processi in contesto utente non si avvicendano nell'esecuzione (in pratica, il processo userà il processore fino al proprio termine, a meno che non ci siano delle interruzioni). Con l'aggiunta di `CONFIG_PREEMPT` nella versione 2.5.4 questo è cambiato: quando si è in contesto utente, processi con una priorità maggiore possono subentrare nell'esecuzione: gli spinlock furono cambiati per disabilitare la prelazioni, anche su sistemi monoprocessore.

bh
: Bottom Half: per ragioni storiche, le funzioni che contengono '_bh' nel loro nome ora si riferiscono a qualsiasi interruzione software; per esempio, spin_lock_bh() blocca qualsiasi interuzione software sul processore corrente. I *Bottom Halves* sono deprecati, e probabilmente verranno sostituiti dai tasklet. In un dato momento potrà esserci solo un *bottom half* in esecuzione.

contesto d'interruzione
: Non è il contesto utente: qui si processano le interruzioni hardware e software. La macro in_interrupt() ritorna vero.

contesto utente
: Il kernel che esegue qualcosa per conto di un particolare processo (per esempio una chiamata di sistema) o di un thread del kernel. Potete identificare il processo con la macro `current`. Da non confondere con lo spazio utente. Può essere interrotto sia da interruzioni software che hardware.

interruzione hardware
Richiesta di interruzione hardware. in_irq() ritorna vero in un gestore d'interruzioni hardware.

interruzione software / softirq
Gestore di interruzioni software: in_irq() ritorna falso; in_softirq() ritorna vero. I tasklet e le softirq sono entrambi considerati 'interruzioni software'.

In soldoni, un softirq è uno delle 32 interruzioni software che possono essere eseguite su più processori in contemporanea. A volte si usa per riferirsi anche ai tasklet (in pratica tutte le interruzioni software).

monoprocessore / UP
(Uni-Processor) un solo processore, ovvero non è SMP. (`CONFIG_SMP=n`).

multi-processore / SMP
(Symmetric Multi-Processor) kernel compilati per sistemi multi-processore (`CONFIG_SMP=y`).

spazio utente
Un processo che esegue il proprio codice fuori dal kernel.

tasklet
Un'interruzione software registrabile dinamicamente che ha la garanzia d'essere eseguita solo su un processore alla volta.

timer
Un'interruzione software registrabile dinamicamente che viene eseguita (circa) in un determinato momento. Quando è in esecuzione è come un tasklet (infatti, sono chiamati da `TIMER_SOFTIRQ`).

* Documentazione della API del kernel

Questi manuali forniscono dettagli su come funzionano i sottosistemi del kernel dal punto di vista degli sviluppatori del kernel. Molte delle informazioni contenute in questi manuali sono prese direttamente dai file sorgenti, informazioni aggiuntive vengono aggiunte solo se necessarie (o almeno ci proviamo —probabilmente *non* tutto quello che è davvero necessario).

Documentazione dell'API di base

Utilità di base

Warning: In caso di dubbi sulla correttezza del contenuto di questa traduzione, l'unico riferimento valido è la documentazione ufficiale in inglese. Per maggiori informazioni consultate le *avvertenze*.

Original
../../../core-api/symbol-namespaces

Translator
Federico Vaga <federico.vaga@vaga.pv.it>

Spazio dei nomi dei simboli

Questo documento descrive come usare lo spazio dei nomi dei simboli per strutturare quello che viene esportato internamente al kernel grazie alle macro della famiglia EXPORT_SYMBOL().

1. Introduzione

Lo spazio dei nomi dei simboli è stato introdotto come mezzo per strutturare l'API esposta internamente al kernel. Permette ai manutentori di un sottosistema di organizzare i simboli esportati in diversi spazi di nomi. Questo meccanismo è utile per la documentazione (pensate ad esempio allo spazio dei nomi SUBSYSTEM_DEBUG) così come per limitare la disponibilità di un gruppo di simboli in altre parti del kernel. Ad oggi, i moduli che usano simboli esportati da uno spazio di nomi devono prima importare detto spazio. Altrimenti il kernel, a seconda della configurazione, potrebbe rifiutare di caricare il modulo o avvisare l'utente di un'importazione mancante.

2. Come definire uno spazio dei nomi dei simboli

I simboli possono essere esportati in spazi dei nomi usando diversi meccanismi. Tutti questi meccanismi cambiano il modo in cui EXPORT_SYMBOL e simili vengono guidati verso la creazione di voci in ksymtab.

2.1 Usare le macro EXPORT_SYMBOL

In aggiunta alle macro EXPORT_SYMBOL() e EXPORT_SYMBOL_GPL(), che permettono di esportare simboli del kernel nella rispettiva tabella, ci sono varianti che permettono di esportare simboli all'interno di uno spazio dei nomi: EXPORT_SYMBOL_NS() ed EXPORT_SYMBOL_NS_GPL(). Queste macro richiedono un argomento aggiuntivo: lo spazio dei nomi. Tenete presente che per via dell'espansione delle macro questo argomento deve essere un simbolo di preprocessore. Per esempio per esportare il simbolo *usb_stor_suspend* nello spazio dei nomi *USB_STORAGE* usate:

```
EXPORT_SYMBOL_NS(usb_stor_suspend, USB_STORAGE);
```

Di conseguenza, nella tabella dei simboli del kernel ci sarà una voce rappresentata dalla struttura *kernel_symbol* che avrà il campo *namespace* (spazio dei nomi) impostato. Un simbolo esportato senza uno spazio dei nomi avrà questo campo impostato a *NULL*. Non esiste uno spazio dei nomi di base. Il programma *modpost* e il codice in kernel/module.c usano lo spazio dei nomi, rispettivamente, durante la compilazione e durante il caricamento di un modulo.

2.2 Usare il simbolo di preprocessore DEFAULT_SYMBOL_NAMESPACE

Definire lo spazio dei nomi per tutti i simboli di un sottosistema può essere logorante e di difficile manutenzione. Perciò è stato fornito un simbolo di preprocessore di base (DEFAULT_SYMBOL_NAMESPACE), che, se impostato, diventa lo spazio dei simboli di base per tutti gli usi di EXPORT_SYMBOL() ed EXPORT_SYMBOL_GPL() che non specificano esplicitamente uno spazio dei nomi.

Ci sono molti modi per specificare questo simbolo di preprocessore e il loro uso dipende dalle preferenze del manutentore di un sottosistema. La prima possibilità è quella di definire il simbolo nel *Makefile* del sottosistema. Per esempio per esportare tutti i simboli definiti in usb-common nello spazio dei nomi USB_COMMON, si può aggiungere la seguente linea in drivers/usb/common/Makefile:

```
ccflags-y += -DDEFAULT_SYMBOL_NAMESPACE=USB_COMMON
```

Questo cambierà tutte le macro EXPORT_SYMBOL() ed EXPORT_SYMBOL_GPL(). Invece, un simbolo esportato con EXPORT_SYMBOL_NS() non verrà cambiato e il simbolo verrà esportato nello spazio dei nomi indicato.

Una seconda possibilità è quella di definire il simbolo di preprocessore direttamente nei file da compilare. L'esempio precedente diventerebbe:

```
#undef  DEFAULT_SYMBOL_NAMESPACE
#define DEFAULT_SYMBOL_NAMESPACE USB_COMMON
```

Questo va messo prima di un qualsiasi uso di EXPORT_SYMBOL.

3. Come usare i simboli esportati attraverso uno spazio dei nomi

Per usare i simboli esportati da uno spazio dei nomi, i moduli del kernel devono esplicitamente importare il relativo spazio dei nomi; altrimenti il kernel potrebbe rifiutarsi di caricare il modulo. Il codice del modulo deve usare la macro MODULE_IMPORT_NS per importare lo spazio dei nomi che contiene i simboli desiderati. Per esempio un modulo che usa il simbolo usb_stor_suspend deve importare lo spazio dei nomi USB_STORAGE usando la seguente dichiarazione:

```
MODULE_IMPORT_NS(USB_STORAGE);
```

Questo creerà un'etichetta *modinfo* per ogni spazio dei nomi importato. Un risvolto di questo fatto è che gli spazi dei nomi importati da un modulo possono essere ispezionati tramite modinfo:

```
$ modinfo drivers/usb/storage/ums-karma.ko
[...]
import_ns:      USB_STORAGE
[...]
```

Si consiglia di posizionare la dichiarazione MODULE_IMPORT_NS() vicino ai metadati del modulo come MODULE_AUTHOR() o MODULE_LICENSE(). Fate riferimento alla sezione 5. per creare automaticamente le importazioni mancanti.

4. Caricare moduli che usano simboli provenienti da spazi dei nomi

Quando un modulo viene caricato (per esempio usando *insmod*), il kernel verificherà la disponibilità di ogni simbolo usato e se lo spazio dei nomi che potrebbe contenerli è stato importato. Il comportamento di base del kernel è di rifiutarsi di caricare quei moduli che non importano tutti gli spazi dei nomi necessari. L'errore verrà annotato e il caricamento fallirà con l'errore EINVAL. Per caricare i moduli che non soddisfano questo requisito esiste un'opzione di configurazione: impostare MODULE_ALLOW_MISSING_NAMESPACE_IMPORTS=y caricherà i moduli comunque ma emetterà un avviso.

5. Creare automaticamente la dichiarazione MODULE_IMPORT_NS

La mancanza di un'importazione può essere individuata facilmente al momento della compilazione. Infatti, modpost emetterà un avviso se il modulo usa un simbolo da uno spazio dei nomi che non è stato importato. La dichiarazione MODULE_IMPORT_NS() viene solitamente aggiunta in un posto ben definito (assieme agli altri metadati del modulo). Per facilitare la vita di chi scrive moduli (e i manutentori di sottosistemi), esistono uno script e un target make per correggere le importazioni mancanti. Questo può essere fatto con:

```
$ make nsdeps
```

Lo scenario tipico di chi scrive un modulo potrebbe essere:

```
- scrivere codice che dipende da un simbolo appartenente ad uno␣
↪spazio
  dei nomi non importato
- eseguire `make`
- aver notato un avviso da modpost che parla di un'importazione
  mancante
- eseguire `make nsdeps` per aggiungere import nel posto giusto
```

Per i manutentori di sottosistemi che vogliono aggiungere uno spazio dei nomi, l'approccio è simile. Di nuovo, eseguendo *make nsdeps* aggiungerà le importazioni mancanti nei moduli inclusi nel kernel:

```
- spostare o aggiungere simboli ad uno spazio dei nomi (per esempio
  usando EXPORT_SYMBOL_NS())
- eseguire `make` (preferibilmente con allmodconfig per coprire␣
↪tutti
  i moduli del kernel)
- aver notato un avviso da modpost che parla di un'importazione
  mancante
- eseguire `make nsdeps` per aggiungere import nel posto giusto
```

Potete anche eseguire nsdeps per moduli esterni. Solitamente si usa così:

```
$ make -C <path_to_kernel_src> M=$PWD nsdeps
```

* Documentazione specifica per architettura

Questi manuali forniscono dettagli di programmazione per le diverse implementazioni d’ architettura.

Warning: TODO ancora da tradurre

CHAPTER

THREE

한국어번역

NOTE: This is a version of Documentation/process/howto.rst translated into korean This document is maintained by Minchan Kim <minchan@kernel.org> If you find any difference between this document and the original file or a problem with the translation, please contact the maintainer of this file.

Please also note that the purpose of this file is to be easier to read for non English (read: korean) speakers and is not intended as a fork. So if you have any comments or updates for this file please try to update the original English file first.

이문서는 Documentation/process/howto.rst 의한글번역입니다.

역자: 김민찬 <minchan@kernel.org> 감수: 이제이미 <jamee.lee@samsung.com>

---

* 어떻게리눅스커널개발을하는가

이문서는커널개발에있어가장중요한문서이다. 이문서는리눅스커널개발자가되는법과리눅스커널개발커뮤니티와일하는법을담고있다. 커널프로그래밍의기술적인측면과관련된내용들은포함하지않으려고하였지만올바른길로여러분을안내하는데는도움이될것이다.

이문서에서오래된것을발견하면문서의아래쪽에나열된메인테이너에게패치를보내달라.

* 소개

자, 여러분은리눅스커널개발자가되는법을배우고싶은가? 아니면상사로부터”이장치를위한리눅스드라이버를작성하시오”라는말을들었는가? 이문서의목적은여러분이겪게될과정과커뮤니티와협력하는법을조언하여여러분의목적을달성하기위해필요한것모두를알려주기위함이다.

커널은대부분은 C 로작성되어있고몇몇아키텍쳐의의존적인부분은어셈블리로작성되어있다. 커널개발을위해 C 를잘이해하고있어야한다. 여러분이특정아키텍쳐의 low-level 개발을할것이아니라면어셈블리 (특정아키텍쳐) 는잘알아야할필요는없다. 다음의참고서적들은기본에충실한 C 교육이나수년간의경험에견주지는못하지만적어도참고용도로는좋을것이다

- “The C Programming Language”by Kernighan and Ritchie [Prentice Hall]
- “Practical C Programming”by Steve Oualline [O’Reilly]
- “C: A Reference Manual”by Harbison and Steele [Prentice Hall]

커널은 GNU C 와 GNU 툴체인을사용하여작성되었다. 이툴들은 ISO C89 표준을따르는반면표준에있지않은많은확장기능도가지고있다. 커널은표준 C 라이브러리와는관계없이 freestanding C 환경이어서 C 표준의일부는지원되지않는다. 임의의 long long 나누기나 floating point 는지원되지않는다. 때론이런이유로커널이그런확장기능을가진툴체인을가지고만들어졌다는것이이해하기어려울수도있고게다가불행하게도그런것을정확하게설명하는어떤참고문서도있지않다. 정보를얻기위해서는 gcc info (*info gcc*) 페이지를살펴보라.

여러분은기존의개발커뮤니티와협력하는법을배우려고하고있다는것을기억하라. 코딩, 스타일, 함수에관한훌륭한표준을가진사람들이모인다양한그룹이있다. 이표준들은오랜동안크고지역적으로분산된팀들에의해가장좋은방법으로일하기위하여찾은것을기초로만들어져왔다. 그표준들은문서화가잘되어있기때문에가능한한미리많은표준들에관하여배우려고시도하라. 다른사람들은여러분이나여러분의회사가일하는방식에적응하는것을원하지는않는다.

* 법적문제

리눅스커널소스코드는 GPL 로배포 (release) 되었다. 소스트리의메인디렉토리에있는라이센스에관하여상세하게쓰여있는 COPYING 이라는파일을봐라. 리눅스커널라이센싱규칙과소스코드안의 SPDX 식별자사용법은 Documentation/process/license-rules.rst 에설명되어있다. 여러분이라이센스에관한더깊은문제를가지고있다면리눅스커널메일링리스트에묻지말고변호사와연락하라. 메일링리스트들에있는사람들은변호사가아니기때문에법적문제에관하여그들의말에의지해서는안된다.

GPL 에관한잦은질문들과답변들은다음을참조하라.

> https://www.gnu.org/licenses/gpl-faq.html

* 문서

리눅스커널소스트리는커널커뮤니티와협력하는법을배우기위해훌륭한다양한문서들을가지고있다. 새로운기능들이커널에들어가게될때, 그기능을어떻게사용하는지에관한설명을위하여새로운문서파일을추가하는것을권장한다. 커널이유저스페이스로노출하는인터페이스를변경하게되면변경을설명하는메뉴얼페이지들에대한패치나정보를 mtk.manpages@gmail.com 의메인테이너에게보낼것을권장한다.

다음은커널소스트리에있는읽어야할파일들의리스트이다.

Documentation/admin-guide/README.rst
: 이파일은리눅스커널에관하여간단한배경설명과커널을설정하고빌드하기위해필요한것을설명한다. 커널에입문하는사람들은여기서시작해야한다.

Documentation/process/changes.rst
: 이파일은커널을성공적으로빌드하고실행시키기위해필요한다양한소프트웨어패키지들의최소버젼을나열한다.

Documentation/process/coding-style.rst
: 이문서는리눅스커널코딩스타일과그렇게한몇몇이유를설명한다. 모든새로운코드는이문서에가이드라인들을따라야한다. 대부분의메인테이너들은이규칙을따르는패치들만을받아들일것이고많은사람들이그패치가올바른스타일일경우만코드를검토할것이다.

Documentation/process/submitting-patches.rst 와 Documentation/process/submitting-drivers.rst
: 이파일들은성공적으로패치를만들고보내는법을다음의내용들로굉장히상세히설명하고있다 (그러나다음으로한정되진않는다).

- Email 내용들
- Email 양식
- 그것을누구에게보낼지

이러한규칙들을따르는것이성공 (역자주: 패치가받아들여지는것) 을보장하진않는다 (왜냐하면모든패치들은내용과스타일에관하여면밀히검토되기때문이다). 그러나규칙을따르지않는다면거의성공하지도못할것이다.

올바른패치들을만드는법에관한훌륭한다른문서들이있다.

"The Perfect Patch"
https://www.ozlabs.org/~akpm/stuff/tpp.txt

"Linux kernel patch submission format"
https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html

Documentation/process/stable-api-nonsense.rst
이문서는의도적으로커널이불변하는 API 를갖지않도록결정한이유를설명하며다음과같은것들을포함한다.

- 서브시스템 shim-layer(호환성을위해?)
- 운영체제들간의드라이버이식성
- 커널소스트리내에빠른변화를늦추는것 (또는빠른변화를막는것)

이문서는리눅스개발철학을이해하는데필수적이며다른운영체제에서리눅스로전향하는사람들에게는매우중요하다.

Documentation/admin-guide/security-bugs.rst
여러분들이리눅스커널의보안문제를발견했다고생각한다면이문서에나온단계에따라서커널개발자들에게알리고그문제를해결할수있도록도와달라.

Documentation/process/management-style.rst
이문서는리눅스커널메인테이너들이그들의방법론에녹아있는정신을어떻게공유하고운영하는지를설명한다. 이것은커널개발에입문하는모든사람들 (또는커널개발에작은호기심이라도있는사람들) 이읽어야할중요한문서이다. 왜냐하면이문서는커널메인테이너들의독특한행동에관하여흔히있는오해들과혼란들을해소하고있기때문이다.

Documentation/process/stable-kernel-rules.rst
이문서는안정적인커널배포가이루어지는규칙을설명하고있으며여러분들이이러한배포들중하나에변경을하길원한다면무엇을해야하는지를설명한다.

Documentation/process/kernel-docs.rst
커널개발에관계된외부문서의리스트이다. 커널내의포함된문서들중에여러분이찾고싶은문서를발견하지못할경우이리스트를살펴보라.

Documentation/process/applying-patches.rst
패치가무엇이며그것을커널의다른개발브랜치들에어떻게적용하는지에관하여자세히설명하고있는좋은입문서이다.

커널은소스코드그자체에서또는이것과같은 ReStructuredText 마크업 (ReST) 을통해자동적으로만들어질수있는많은문서들을가지고있다. 이것은커널내의 API 에대한모든설명, 그리고락킹을올바르게처리하는법에관한규칙을포함하고있다.

모든그런문서들은커널소스디렉토리에서다음커맨드를실행하는것을통해 PDF 나 HTML 의형태로만들어질수있다:

```
make pdfdocs
make htmldocs
```

ReST 마크업을사용하는문서들은 Documentation/output 에생성된다. 해당문서들은다음의커맨드를사용하면 LaTeX 이나 ePub 로도만들어질수있다:

```
make latexdocs
make epubdocs
```

* 커널개발자가되는것

여러분이리눅스커널개발에관하여아무것도모른다면 Linux KernelNewbies 프로젝트를봐야한다.

https://kernelnewbies.org

그곳은거의모든종류의기본적인커널개발질문들 (질문하기전에먼저아카이브를찾아봐라. 과거에이미답변되었을수도있다) 을할수있는도움이될만한메일링리스트가있다. 또한실시간으로질문할수있는 IRC 채널도가지고있으며리눅스커널개발을배우는데유용한문서들을보유하고있다.

웹사이트는코드구성, 서브시스템들, 그리고현재프로젝트들 (트리내, 외부에존재하는) 에관한기본적인정보들을가지고있다. 또한그곳은커널컴파일이나패치를하는법과같은기본적인것들을설명한다.

여러분이어디서시작해야할진모르지만커널개발커뮤니티에참여할수있는일들을찾길원한다면리눅스커널 Janitor 프로젝트를살펴봐라.

https://kernelnewbies.org/KernelJanitors

그곳은시작하기에훌륭한장소이다. 그곳은리눅스커널소스트리내에간단히정리되고수정될수있는문제들에관하여설명한다. 여러분은이프로젝트를대표하는개발자들과일하면서자신의패치를리눅스커널트리에반영하기위한기본적인것들을배우게될것이며여러분이아직아이디어를가지고있지않다면다음에무엇을해야할지에관한방향을배울수있을것이다.

리눅스커널코드에실제변경을하기전에반드시그코드가어떻게동작하는지이해하고있어야한다. 코드를분석하기위하여특정한툴의도움을빌려서라도코드를직접읽는것보다좋은것은없다 (대부분의자잘한부분들은잘코멘트되어있다). 그런툴들중에특히추천할만한것은 Linux Cross-Reference project 이며그것은자기참조방식이며소스코드를인덱스된웹페이지들의형태로보여준다. 최신의멋진커널코드저장소는다음을통하여참조할수있다.

https://elixir.bootlin.com/

* 개발프로세스

리눅스커널개발프로세스는현재몇몇다른메인커널 “브랜치들” 과서브시스템에특화된커널브랜치들로구성된다. 몇몇다른메인브랜치들은다음과같다.

- 리누스의메인라인트리
- 여러메이저넘버를갖는다양한안정된커널트리들
- 서브시스템을위한커널트리들
- 통합테스트를위한 linux-next 커널트리

메인라인트리

메인라인트리는 Linus Torvalds 가관리하며 https://kernel.org 또는소스저장소에서참조될수있다. 개발프로세스는다음과같다.

- 새로운커널이배포되자마자 2 주의시간이주어진다. 이기간동은메인테이너들은큰 diff 들을 Linus 에게제출할수있다. 대개이패치들은몇주동안 linux-next 커널내에이미있었던것들이다. 큰변경들을제출하는데선호되는방법은 git(커널의소스관리툴, 더많은정보들은 https://git-scm.com/ 에서참조할수있다) 를사용하는것이지만순수한패치파일의형식으로보내는것도무관하다.
- 2 주후에 -rc1 커널이릴리즈되며여기서부터의주안점은새로운커널을가능한한안정되게하는것이다. 이시점에서의대부분의패치들은회귀 (역자주: 이전에는존재하지않았지만새로운기능추가나변경으로인해생겨난버그) 를고쳐야한다. 이전부터존재한버그는회귀가아니므로, 그런버그에대한수정사항은중요한경우에만보내져야한다. 완전히새로운드라이버 (혹은파일시스템) 는 -rc1 이후에만받아들여진다는것을기억해라. 왜냐하면변경이자체내에서만발생하고추가된코드가드라이버외부의다른부분에는영향을주지않으므로그런변경은회귀를일으킬만한위험을가지고있지않기때문이다. -rc1 이배포된이후에 git 를사용하여패치들을 Linus 에게보낼수있지만패치들은공식적인메일링리스트로보내서검토를받을필요가있다.
- 새로운 -rc 는 Linus 가현재 git tree 가테스트하기에충분히안정된상태에있다고판단될때마다배포된다. 목표는새로운 -rc 커널을매주배포하는것이다.
- 이러한프로세스는커널이 "준비 (ready)" 되었다고여겨질때까지계속된다. 프로세스는대체로 6 주간지속된다.

커널배포에있어서언급할만한가치가있는리눅스커널메일링리스트의 Andrew Morton 의글이있다.

> "커널이언제배포될지는아무도모른다. 왜냐하면배포는알려진버그의상황에따라배포되는것이지미리정해놓은시간에따라배포되는것은아니기때문이다."

여러메이저넘버를갖는다양한안정된커널트리들

세개의버젼넘버로이루어진버젼의커널들은 -stable 커널들이다. 그것들은해당메이저메인라인릴리즈에서발견된큰회귀들이나보안문제들중비교적작고중요한수정들을포함한다. 주요 stable 시리즈릴리즈는세번째버젼넘버를증가시키며앞의두버젼넘버는그대로유지한다.

이것은가장최근의안정적인커널을원하는사용자에게추천되는브랜치이며, 개발/실험적버젼을테스트하는것을돕고자하는사용자들과는별로관련이없다.

-stable 트리들은 "stable" 팀 <stable@vger.kernel.org> 에의해관리되며거의매번격주로배포된다.

커널트리문서들내의 Documentation/process/stable-kernel-rules.rst 파일은어떤종류의변경들이 -stable 트리로들어왔는지와배포프로세스가어떻게진행되는지를설명한다.

서브시스템커널트리들

다양한커널서브시스템의메인테이너들—그리고많은커널서브시스템개발자들—은그들의현재개발상태를소스저장소로노출한다. 이를통해다른사람들도커널의다른영역에어떤변화가이루어지고있는지알수있다. 급속히개발이진행되는영역이있고그렇지않은영역이있으므로, 개발자는다른개발자가제출한수정사항과자신의수정사항의충돌이나동일한일을동시에두사람이따로진행하는사태를방지하기위해급속히개발이진행되고있는영역에작업의베이스를맞춰줄것이요구된다.

대부분의이러한저장소는 git 트리지만, git 이아닌 SCM 으로관리되거나, quilt 시리즈로제공되는패치들도존재한다. 이러한서브시스템저장소들은 MAINTAINERS 파일에나열되어있다. 대부분은 https://git.kernel.org 에서볼수있다.

제안된패치는서브시스템트리에커밋되기전에메일링리스트를통해리뷰된다 (아래의관련섹션을참고하기바란다). 일부커널서브시스템의경우, 이리뷰프로세스는 patchwork 라는도구를통해추적된다. patchwork 은등록된패치와패치에대한코멘트, 패치의버젼을볼수있는웹인터페이스를제공하고, 메인테이너는패치를리뷰중, 리뷰통과, 또는반려됨으로표시할수있다. 대부분의이러한 patchwork 사이트는 https://patchwork.kernel.org/ 에나열되어있다.

통합테스트를위한 linux-next 커널트리

서브시스템트리들의변경사항들은 mainline 트리로들어오기전에통합테스트를거쳐야한다. 이런목적으로, 모든서브시스템트리의변경사항을거의매일받아가는특수한테스트저장소가존재한다:

> https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git

이런식으로, linux-next 커널을통해다음머지기간에메인라인커널에어떤변경이가해질것인지간략히알수있다. 모험심강한테스터라면 linux-next 커널에서테스트를수행하는것도좋을것이다.

* 버그보고

https://bugzilla.kernel.org 는리눅스커널개발자들이커널의버그를추적하는곳이다. 사용자들은발견한모든버그들을보고하기위하여이툴을사용할것을권장한다. kernel bugzilla 를사용하는자세한방법은다음을참조하라.

> https://bugzilla.kernel.org/page.cgi?id=faq.html

메인커널소스디렉토리에있는 admin-guide/reporting-bugs.rst 파일은커널버그라고생각되는것을보고하는방법에관한좋은템플릿이며문제를추적하기위해서커널개발자들이필요로하는정보가무엇들인지를상세히설명하고있다.

* 버그리포트들의관리

여러분의해킹기술을연습하는가장좋은방법중의하는다른사람들이보고한버그들을수정하는것이다. 여러분은커널을더욱안정화시키는데도움을줄뿐만이아니라실제있는문제들을수정하는법을배우게되고그와함께여러분들의기술은향상될것이며다른개발자들이여러분의존재에대해알게될것이다. 버그를수정하는것은개발자들사이에서점수를얻을수있는가장좋은방법중의하나이다. 왜냐하면많은사람들은다른사람들의버그들을수정하기위하여시간을낭비하지않기때문이다.

이미보고된버그리포트들을가지고작업하기위해서 https://bugzilla.kernel.org 를참조하라.

* 메일링리스트들

위의몇몇문서들이설명하였지만핵심커널개발자들의대다수는리눅스커널메일링리스트에참여하고있다. 리스트에등록하고해지하는방법에관한자세한사항은다음에서참조할수있다.

http://vger.kernel.org/vger-lists.html#linux-kernel

웹상의많은다른곳에도메일링리스트의아카이브들이있다. 이러한아카이브들을찾으려면검색엔진을사용하라. 예를들어:

http://dir.gmane.org/gmane.linux.kernel

여러분이새로운문제에관해리스트에올리기전에말하고싶은주제에관한것을아카이브에서먼저찾아보기를강력히권장한다. 이미상세하게토론된많은것들이메일링리스트의아카이브에기록되어있다.

각각의커널서브시스템들의대부분은자신들의개발에관한노력들로이루어진분리된메일링리스트를따로가지고있다. 다른그룹들이무슨리스트를가지고있는지는 MAINTAINERS 파일을참조하라.

많은리스트들은 kernel.org 에서호스트되고있다. 그정보들은다음에서참조될수있다.

http://vger.kernel.org/vger-lists.html

리스트들을사용할때는올바른예절을따를것을유념해라. 대단하진않지만다음 URL 은리스트 (혹은모든리스트) 와대화하는몇몇간단한가이드라인을가지고있다.

http://www.albion.com/netiquette/

여러사람들이여러분의메일에응답한다면 CC: 즉수신리스트는꽤커지게될것이다. 아무이유없이 CC 에서어떤사람도제거하거나리스트주소로만회신하지마라. 메일을보낸사람으로서하나를받고리스트로부터또하나를받아두번받는것에익숙하여있으니 mail-header 를조작하려고하지말아라. 사람들은그런것을좋아하지않을것이다.

여러분의회신의문맥을원래대로유지해야한다. 여러분들의회신의윗부분에 “John 커널해커는작성했다….” 를유지하며여러분들의의견을그메일의윗부분에작성하지말고각인용한단락들사이에넣어라.

여러분들이패치들을메일에넣는다면그것들은 Documentation/process/submitting-patches.rst 에나와있는데로명백히 (plain) 읽을수있는텍스트여야한다. 커널개발자들은첨부파일이나압축된패치들을원하지않는다. 그들은여러분들의패치의각라인단위로코멘트를하길원하며압축하거나첨부하지않고보내는것이그렇게할수있는유일한방법이다. 여러분들이사용하는메일프로그램이스페이스나탭문자들을조작하지않는지확인하라. 가장좋은첫테스트는메일을자신에게보내보고스스로그패치를적용해보라. 그것이동작하지않는다면여러분의메일프로그램을고치던가제대로동작하는프로그램으로바꾸어라.

무엇보다도메일링리스트의다른구독자들에게보여주려한다는것을기억하라.

* 커뮤니티와협력하는법

커널커뮤니티의목적은가능한한가장좋은커널을제공하는것이다. 여러분이받아들여질패치를제출하게되면그패치의기술적인이점으로검토될것이다. 그럼여러분들은무엇을기대하고있어야하는가?

- 비판
- 의견
- 변경을위한요구
- 당위성을위한요구
- 침묵

기억하라. 이것들은여러분의패치가커널로들어가기위한과정이다. 여러분의패치들은비판과다른의견을받을수있고그것들을기술적인레벨로평가하고재작업하거나또는왜수정하면안되는지에관하여명료하고간결한이유를말할수있어야한다. 여러분이제출한것에어떤응답도있지않다면몇일을기다려보고다시시도해라. 때론너무많은메일들속에묻혀버리기도한다.

여러분은무엇을해서는안되는가?

- 여러분의패치가아무질문없이받아들여지기를기대하는것
- 방어적이되는것
- 의견을무시하는것
- 요청된변경을하지않고패치를다시제출하는것

가능한한가장좋은기술적인해답을찾고있는커뮤니티에서는항상어떤패치가얼마나좋은지에관하여다른의견들이있을수있다. 여러분은협조적이어야하고기꺼이여러분의생각을커널내에맞추어야한다. 아니면적어도여러분의것이가치있다는것을증명하여야한다. 잘못된것도여러분이올바른방향의해결책으로이끌어갈의지가있다면받아들여질것이라는점을기억하라.

여러분의첫패치에여러분이수정해야하는십여개정도의회신이오는경우도흔하다. 이것은여러분의패치가받아들여지지않을것이라는것을의미하는것이아니고개인적으로여러분에게감정이있어서그러는것도아니다. 간단히여러분의패치에제기된문제들을수정하고그것을다시보내라.

* 커널커뮤니티와기업조직간의차이점

커널커뮤니티는가장전통적인회사의개발환경과는다르다. 여기에여러분들의문제를피하기위한목록이있다.

여러분들이제안한변경들에관하여말할때좋은것들:

- “이것은여러문제들을해결합니다.”
- “이것은 2000 라인의코드를줄입니다.”
- “이것은내가말하려는것에관해설명하는패치입니다.”
- “나는 5 개의다른아키텍쳐에서그것을테스트했으므로…”
- “여기에일련의작은패치들이있으므로…”
- “이것은일반적인머신에서성능을향상함으로…”

여러분들이말할때피해야할좋지않은것들:

- “우리는그것을 AIX/ptx/Solaris 에서이러한방법으로했다. 그러므로그것은좋은것임에틀림없다…”
- “나는 20 년동안이것을해왔다. 그러므로…”
- “이것은돈을벌기위해나의회사가필요로하는것이다.”
- “이것은우리의엔터프라이즈상품라인을위한것이다.”
- “여기에나의생각을말하고있는 1000 페이지설계문서가있다.”
- “나는 6 달동안이것을했으니…”
- “여기에 5000 라인짜리패치가있으니…”
- “나는현재뒤죽박죽인것을재작성했다. 그리고여기에…”
- “나는마감시한을가지고있으므로이패치는지금적용될필요가있다.”

커널커뮤니티가전통적인소프트웨어엔지니어링개발환경들과또다른점은얼굴을보지않고일한다는점이다. 이메일과 irc 를대화의주요수단으로사용하는것의한가지장점은성별이나인종의차별이없다는것이다. 리눅스커널의작업환경에서는단지이메일주소만알수있기때문에여성과소수민족들도모두받아들여진다. 국제적으로일하게되는측면은사람의이름에근거하여성별을추측할수없게하기때문에차별을없애는데도움을준다. Andrea 라는이름을가진남자와 Pat 이라는이름을가진여자가있을수도있는것이다. 리눅스커널에서작업하며생각을표현해왔던대부분의여성들은긍정적인경험을가지고있다.

언어장벽은영어에익숙하지않은몇몇사람들에게문제가될수도있다. 언어의훌륭한구사는메일링리스트에서올바르게자신의생각을표현하기위하여필요하다. 그래서여러분은이메일을보내기전에영어를올바르게사용하고있는지를체크하는것이바람직하다.

* 여러분의변경을나누어라

리눅스커널커뮤니티는한꺼번에굉장히큰코드의묶음 (chunk) 을쉽게받아들이지않는다. 변경은적절하게소개되고, 검토되고, 각각의부분으로작게나누어져야한다. 이것은회사에서하는것과는정확히반대되는것이다. 여러분들의제안은개발초기에일찍이소개되야한다. 그래서여러분들은자신이하고있는것에관하여피드백을받을수있게된다. 커뮤니티가여러분들이커뮤니티와함께일하고있다는것을느끼도록만들고커뮤니티가여러분의기능을위한쓰레기장으로써사용되지않고있다는것을느끼게하자. 그러나메일링리스트에한번에 50 개의이메일을보내지는말아라. 여러분들의일련의패치들은항상더작아야한다.

패치를나누는이유는다음과같다.

1) 작은패치들은여러분의패치들이적용될수있는확률을높여준다. 왜냐하면다른사람들은정확성을검증하기위하여많은시간과노력을들이기를원하지않는다. 5 줄의패치는메인테이너가거의몇초간힐끗보면적용될수있다. 그러나 500 줄의패치는정확성을검토하기위하여몇시간이걸릴수도있다 (걸리는시간은패치의크기혹은다른것에비례하여기하급수적으로늘어난다).

   패치를작게만드는것은무엇인가잘못되었을때디버그하는것을쉽게만든다. 즉, 그렇게만드는것은매우큰패치를적용한후에조사하는것보다작은패치를적용한후에 (그리고몇몇의것이깨졌을때) 하나씩패치들을제거해가며디버그하기쉽도록만들어준다.

2) 작은패치들을보내는것뿐만아니라패치들을제출하기전에재작성하고간단하게 (혹은간단한게재배치하여) 하는것도중요하다.

여기에커널개발자 Al Viro 의이야기가있다.

> “학생의수학숙제를채점하는선생님을생각해보라. 선생님은학생들이답을얻을때까지겪은시행착오를보길원하지않는다. 선생님들은간결하고가장뛰어난답을보길원한다. 훌륭한학생은이것을알고마지막으로답을얻기전중간과정들을제출하진않는다.
>
> 커널개발도마찬가지이다. 메인테이너들과검토하는사람들은문제를풀어나가는과정속에숨겨진과정을보길원하진않는다. 그들은간결하고멋진답을보길원한다.”

커뮤니티와협력하며뛰어난답을찾는것과여러분들의끝마치지못한작업들사이에균형을유지해야하는것은어려울지도모른다. 그러므로프로세스의초반에여러분의작업을향상시키기위한피드백을얻는것뿐만아니라여러분들의변경들을작은묶음으로유지해서심지어는여러분의작업의모든부분이지금은포함될준비가되어있지않지만작은부분은벌써받아들여질수있도록유지하는것이바람직하다.

또한완성되지않았고 “나중에수정될것이다.” 와같은것들을포함하는패치들은받아들여지지않을것이라는점을유념하라.

* 변경을정당화해라

여러분들의나누어진패치들을리눅스커뮤니티가왜반영해야하는지를알도록하는것은매우중요하다. 새로운기능들이필요하고유용하다는것은반드시그에합당한이유가있어야한다.

* 변경을문서화해라

여러분이패치를보내려할때는여러분이무엇을말하려고하는지를충분히생각하여이메일을작성해야한다. 이정보는패치를위한 ChangeLog 가될것이다. 그리고항상그내용을보길원하는모든사람들을위해보존될것이다. 패치는완벽하게다음과같은내용들을포함하여설명해야한다.

- 변경이왜필요한지
- 패치에관한전체설계접근 (approach)
- 구현상세들
- 테스트결과들

이것이무엇인지더자세한것을알고싶다면다음문서의 ChageLog 항을봐라.

> “The Perfect Patch”
>
> http://www.ozlabs.org/~akpm/stuff/tpp.txt

이모든것을하는것은매우어려운일이다. 완벽히소화하는데는적어도몇년이걸릴수도있다. 많은인내와결심이필요한계속되는개선의과정이다. 그러나가능한한포기하지말라. 많은사람들은이전부터해왔던것이고그사람들도정확하게여러분들이지금서있는그곳부터시작했었다.

“개발프로세스”(https://lwn.net/Articles/94386/) 섹션을작성하는데있어참고할문서를사용하도록허락해준 Paolo Ciarrocchi 에게감사한다. 여러분들이말해야할것과말해서는안되는것의목록중일부를제공해준 Randy Dunlap 과 Gerrit Huizenga 에게감사한다. 또한검토와의견그리고공헌을아끼지않은 Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, and Alex Shepard 에게도감사를전한다. 그들의도움이없었다면이문서는존재하지않았을것이다.

메인테이너: Greg Kroah-Hartman <greg@kroah.com>

CHAPTER

FOUR

JAPANESE TRANSLATIONS

NOTE: This is a version of Documentation/process/howto.rst translated into Japanese. This document is maintained by Tsugikazu Shibata <tshibata@ab.jp.nec.com> If you find any difference between this document and the original file or a problem with the translation, please contact the maintainer of this file.

Please also note that the purpose of this file is to be easier to read for non English (read: Japanese) speakers and is not intended as a fork. So if you have any comments or updates for this file, please try to update the original English file first.

この文書は、Documentation/process/howto.rst の和訳です。

翻訳者：Tsugikazu Shibata <tshibata@ab.jp.nec.com>

---

* Linux カーネル開発のやり方

これは上のトピック (Linux カーネル開発のやり方) の重要な事柄を網羅したドキュメントです。ここには Linux カーネル開発者になるための方法と Linux カーネル開発コミュニティと共に活動するやり方を学ぶ方法が含まれています。カーネルプログラミングに関する技術的な項目に関することは何も含めないようにしていますが、カーネル開発者となるための正しい方向に向かう手助けになります。

もし、このドキュメントのどこかが古くなっていた場合には、このドキュメントの最後にリストしたメンテナにパッチを送ってください。

* はじめに

あなたは Linux カーネルの開発者になる方法を学びたいのでしょうか? それとも上司から「このデバイスの Linux ドライバを書くように」と言われたのかもしれません。この文書の目的は、あなたが踏むべき手順と、コミュニティと一緒にうまく働くヒントを書き下すことで、あなたが知るべき全てのことを教えることです。また、このコミュニティがなぜ今うまくまわっているのかという理由も説明しようと試みています。

カーネルは少量のアーキテクチャ依存部分がアセンブリ言語で書かれている以外の大部分は C 言語で書かれています。C 言語をよく理解していることはカーネル開発に必要です。低レベルのアーキテクチャ開発をするのでなければ、(どんなアーキテクチャでも) アセンブリ

(訳注: 言語) は必要ありません。以下の本は、C 言語の十分な知識や何年もの経験に取って代わるものではありませんが、少なくともリファレンスとしては良い本です。

- “The C Programming Language”by Kernighan and Ritchie [Prentice Hall]
- 『プログラミング言語C第 2 版』(B.W. カーニハン/D.M. リッチー著石田晴久訳) [共立出版]
- “Practical C Programming”by Steve Oualline [O’Reilly]
- 『C 実践プログラミング第 3 版』(Steve Oualline 著望月康司監訳谷口功訳) [オライリージャパン]
- “C: A Reference Manual”by Harbison and Steele [Prentice Hall]
- 『新・詳説 C 言語 H&S リファレンス』(サミュエル P ハービソン/ガイ L スティール共著斉藤信男監訳)[ソフトバンク]

カーネルは GNU C と GNU ツールチェインを使って書かれています。カーネルは ISO C89 仕様に準拠して書く一方で、標準には無い言語拡張を多く使っています。カーネルは標準 C ライブラリに依存しない、C 言語非依存環境です。そのため、C の標準の中で使えないものもあります。特に任意の long long の除算や浮動小数点は使えません。カーネルがツールチェインや C 言語拡張に置いている前提がどうなっているのかわかりにくいことが時々あり、また、残念なことに決定的なリファレンスは存在しません。情報を得るには、gcc の info ページ (info gcc) を見てください。

あなたは既存の開発コミュニティと一緒に作業する方法を学ぼうとしていることに思い出してください。そのコミュニティは、コーディング、スタイル、開発手順について高度な標準を持つ、多様な人の集まりです。地理的に分散した大規模なチームに対してもっともうまくいくとわかったことをベースにしながら、これらの標準は長い時間をかけて築かれてきました。これらはきちんと文書化されていますから、事前にこれらの標準について事前にできるだけたくさん学んでください。また皆があなたやあなたの会社のやり方に合わせてくれると思わないでください。

* 法的問題

Linux カーネルのソースコードは GPL ライセンスの下でリリースされています。ライセンスの詳細については、ソースツリーのメインディレクトリに存在する、COPYING のファイルを見てください。もしライセンスについてさらに質問があれば、Linux Kernel メーリングリストに質問するのではなく、どうぞ法律家に相談してください。メーリングリストの人達は法律家ではなく、法的問題については彼らの声明はあてにするべきではありません。

GPL に関する共通の質問や回答については、以下を参照してください-

https://www.gnu.org/licenses/gpl-faq.html

* ドキュメント

Linux カーネルソースツリーは幅広い範囲のドキュメントを含んでおり、それらはカーネルコミュニティと会話する方法を学ぶのに非常に貴重なものです。新しい機能がカーネルに追加される場合、その機能の使い方について説明した新しいドキュメントファイルも追加することを勧めます。カーネルの変更が、カーネルがユーザ空間に公開しているインターフェイスの変更を引き起こす場合、その変更を説明するマニュアルページのパッチや情報をマニュアルページのメンテナ mtk.manpages@gmail.com に送り、CC を linux-api@vger.kernel.org に送ることを勧めます。

以下はカーネルソースツリーに含まれている読んでおくべきファイルの一覧です-

README
このファイルは Linux カーネルの簡単な背景とカーネルを設定 (訳注 configure) し、生成 (訳注 build) するために必要なことは何かが書かれています。カーネルに関して初めての人はここからスタートすると良いでしょう。

Documentation/process/changes.rst
このファイルはカーネルをうまく生成 (訳注 build) し、走らせるのに最小限のレベルで必要な数々のソフトウェアパッケージの一覧を示しています。

Documentation/process/coding-style.rst
これは Linux カーネルのコーディングスタイルと背景にある理由を記述しています。全ての新しいコードはこのドキュメントにあるガイドラインに従っていることを期待されています。大部分のメンテナはこれらのルールに従っているものだけを受け付け、多くの人は正しいスタイルのコードだけをレビューします。

Documentation/process/submitting-patches.rst と Documentation/process/submitting-drivers.rst
これらのファイルには、どうやってうまくパッチを作って投稿するかについて非常に詳しく書かれており、以下を含みます (これだけに限らないけれども)

- Email に含むこと
- Email の形式
- だれに送るか

これらのルールに従えばうまくいくことを保証することではありませんが (すべてのパッチは内容とスタイルについて精査を受けるので)、ルールに従わなければ間違いなくうまくいかないでしょう。

この他にパッチを作る方法についてのよくできた記述は-

“The Perfect Patch”
http://www.ozlabs.org/~akpm/stuff/tpp.txt

“Linux kernel patch submission format”
https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html

Documentation/process/stable-api-nonsense.rst
このファイルはカーネルの中に不変の API を持たないことにした意識的な決断の背景にある理由について書かれています。以下のようなことを含んでいます-

- サブシステムとの間に層を作ること (コンパチビリティのため?)
- オペレーティングシステム間のドライバの移植性
- カーネルソースツリーの素早い変更を遅らせる (もしくは素早い変更を妨げる)

このドキュメントは Linux 開発の思想を理解するのに非常に重要です。そして、他の OS での開発者が Linux に移る時にとても重要です。

Documentation/admin-guide/security-bugs.rst
もし Linux カーネルでセキュリティ問題を発見したように思ったら、このドキュメントのステップに従ってカーネル開発者に連絡し、問題解決を支援してください。

Documentation/process/management-style.rst
このドキュメントは Linux カーネルのメンテナ達がどう行動するか、彼らの手法の背景にある共有されている精神について記述しています。これはカーネル開発の初心者なら（もしくは、単に興味があるだけの人でも）重要です。なぜならこのドキュメントは、カーネルメンテナ達の独特な行動についての多くの誤解や混乱を解消するからです。

Documentation/process/stable-kernel-rules.rst
このファイルはどのように stable カーネルのリリースが行われるかのルールが記述されています。そしてこれらのリリースの中のどこかで変更を取り入れてもらいたい場合に何をすれば良いかが示されています。

Documentation/process/kernel-docs.rst
カーネル開発に付随する外部ドキュメントのリストです。もしあなたが探しているものがカーネル内のドキュメントでみつからなかった場合、このリストをあたってみてください。

Documentation/process/applying-patches.rst
パッチとはなにか、パッチをどうやって様々なカーネルの開発ブランチに適用するのかについて正確に記述した良い入門書です。

カーネルはソースコードそのものや、このファイルのようなリストラクチャードテキストマークアップ (ReST) から自動的に生成可能な多数のドキュメントをもっています。これにはカーネル内 API の完全な記述や、正しくロックをかけるための規則などが含まれます。

これら全てのドキュメントを PDF や HTML で生成するには以下を実行します -

```
make pdfdocs
make htmldocs
```

それぞれメインカーネルのソースディレクトリから実行します。

ReST マークアップを使ったドキュメントは Documentation/output に生成されます。Latex と ePub 形式で生成するには -

```
make latexdocs
make epubdocs
```

* カーネル開発者になるには

もしあなたが、Linux カーネル開発について何も知らないのならば、KernelNewbies プロジェクトを見るべきです

> https://kernelnewbies.org

このサイトには役に立つメーリングリストがあり、基本的なカーネル開発に関するほとんどどんな種類の質問もできます (既に回答されているようなことを聞く前にまずはアーカイブを調べてください)。またここには、リアルタイムで質問を聞くことができる IRC チャネルや、Linux カーネルの開発に関して学ぶのに便利なたくさんの役に立つドキュメントがあります。

Web サイトには、コードの構成、サブシステム、現在存在するプロジェクト (ツリーにあるもの無いものの両方) の基本的な管理情報があります。ここには、また、カーネルのコンパイルのやり方やパッチの当て方などの間接的な基本情報も記述されています。

あなたがどこからスタートして良いかわからないが、Linux カーネル開発コミュニティに参加して何かすることをさがしているのであれば、Linux kernel Janitor’s プロジェクトにいけば良いでしょう -

> https://kernelnewbies.org/KernelJanitors

ここはそのようなスタートをするのにうってつけの場所です。ここには、Linux カーネルソースツリーの中に含まれる、きれいにし、修正しなければならない、単純な問題のリストが記述されています。このプロジェクトに関わる開発者と一緒に作業することで、あなたのパッチを Linux カーネルツリーに入れるための基礎を学ぶことができ、そしてもしあなたがまだアイディアを持っていない場合には、次にやる仕事の方向性が見えてくるかもしれません。

もしあなたが、すでにひとまとまりコードを書いていて、カーネルツリーに入れたいと思っていたり、それに関する適切な支援を求めたい場合、カーネルメンターズプロジェクトはそのような皆さんを助けるためにできました。ここにはメーリングリストがあり、以下から参照できます -

> https://selenic.com/mailman/listinfo/kernel-mentors

実際に Linux カーネルのコードについて修正を加える前に、どうやってそのコードが動作するのかを理解することが必要です。そのためには、特別なツールの助けを借りてでも、それを直接よく読むことが最良の方法です (ほとんどのトリッキーな部分は十分にコメントしてありますから)。そういうツールで特におすすめなのは、Linux クロスリファレンスプロジェクトです。これは、自己参照方式で、索引がついた web 形式で、ソースコードを参照することができます。この最新の素晴しいカーネルコードのリポジトリは以下で見つかります -

> https://elixir.bootlin.com/

* 開発プロセス

Linux カーネルの開発プロセスは現在幾つかの異なるメインカーネル「ブランチ」と多数のサブシステム毎のカーネルブランチから構成されます。これらのブランチとは -

- メインの 4.x カーネルツリー
- 4.x.y -stable カーネルツリー
- サブシステム毎のカーネルツリーとパッチ
- 統合テストのための 4.x -next カーネルツリー

4.x カーネルツリー

4.x カーネルは Linus Torvalds によってメンテナンスされ、https://kernel.org の pub/linux/kernel/v4.x/ ディレクトリに存在します。この開発プロセスは以下のとおり -

- 新しいカーネルがリリースされた直後に、2 週間の特別期間が設けられ、この期間中に、メンテナ達は Linus に大きな差分を送ることができます。このような差分は通常 -next カーネルに数週間含まれてきたパッチです。大きな変更は git(カーネルのソース管理ツール、詳細は http://git-scm.com/ 参照) を使って送るのが好ましいやり方ですが、パッチファイルの形式のまま送るのでも十分です。
- 2 週間後、-rc1 カーネルがリリースされ、この後にはカーネル全体の安定性に影響をあたえるような新機能は含まない類のパッチしか取り込むことはできません。新しいドライバ (もしくはファイルシステム) のパッチは -rc1 の後で受け付けられることもあることを覚えておいてください。なぜなら、変更が独立していて、追加されたコードの外の領域に影響を与えない限り、退行のリスクは無いからです。-rc1 がリリースされた後、Linus へパッチを送付するのに git を使うこともできますが、パッチはレビューのために、パブリックなメーリングリストへも同時に送る必要があります。
- 新しい -rc は Linus が、最新の git ツリーがテスト目的であれば十分に安定した状態にあると判断したときにリリースされます。目標は毎週新しい -rc カーネルをリリースすることです。
- このプロセスはカーネルが「準備ができた」と考えられるまで継続します。このプロセスはだいたい 6 週間継続します。

Andrew Morton が Linux-kernel メーリングリストにカーネルリリースについて書いたことをここで言っておくことは価値があります -

> 「カーネルがいつリリースされるかは誰も知りません。なぜなら、これは現実に認識されたバグの状況によりリリースされるのであり、前もって決められた計画によってリリースされるものではないからです。」

4.x.y -stable カーネルツリー

バージョン番号が 3 つの数字に分かれているカーネルは -stable カーネルです。これには、4.x カーネルで見つかったセキュリティ問題や重大な後戻りに対する比較的小さい重要な修正が含まれます。

これは、開発/実験的バージョンのテストに協力することに興味が無く、最新の安定したカーネルを使いたいユーザに推奨するブランチです。

もし、4.x.y カーネルが存在しない場合には、番号が一番大きい 4.x が最新の安定版カーネルです。

4.x.y は“stable”チーム <stable@vger.kernel.org> でメンテされており、必要に応じてリリースされます。通常のリリース期間は 2 週間毎ですが、差し迫った問題がなければもう少し長くなることもあります。セキュリティ関連の問題の場合はこれに対してだいたいの場合、すぐにリリースがされます。

カーネルツリーに入っている、Documentation/process/stable-kernel-rules.rst ファイルにはどのような種類の変更が -stable ツリーに受け入れ可能か、またリリースプロセスがどう動くかが記述されています。

サブシステム毎のカーネルツリーとパッチ

それぞれのカーネルサブシステムのメンテナ達は—そして多くのカーネルサブシステムの開発者達も—各自の最新の開発状況をソースリポジトリに公開しています。そのため、自分とは異なる領域のカーネルで何が起きているかを他の人が見られるようになっています。開発が早く進んでいる領域では、開発者は自身の投稿がどのサブシステムカーネルツリーを元にしているか質問されるので、その投稿とすでに進行中の他の作業との衝突が避けられます。

大部分のこれらのリポジトリは git ツリーです。しかしその他の SCM や quilt シリーズとして公開されているパッチキューも使われています。これらのサブシステムリポジトリのアドレスは MAINTAINERS ファイルにリストされています。これらの多くは https://git.kernel.org/ で参照することができます。

提案されたパッチがこのようなサブシステムツリーにコミットされる前に、メーリングリストで事前にレビューにかけられます（以下の対応するセクションを参照）。いくつかのカーネルサブシステムでは、このレビューは patchwork というツールによって追跡されます。Patchwork は web インターフェイスによってパッチ投稿の表示、パッチへのコメント付けや改訂などができ、そしてメンテナはパッチに対して、レビュー中、受付済み、拒否というようなマークをつけることができます。大部分のこれらの patchwork のサイトは https://patchwork.kernel.org/ でリストされています。

統合テストのための 4.x -next カーネルツリー

サブシステムツリーの更新内容がメインラインの 4.x ツリーにマージされる前に、それらは統合テストされる必要があります。この目的のため、実質的に全サブシステムツリーからほぼ毎日プルされてできる特別なテスト用のリポジトリが存在します-

> https://git.kernel.org/?p=linux/kernel/git/next/linux-next.git

このやり方によって、-next カーネルは次のマージ機会でどんなものがメインラインカーネルにマージされるか、おおまかなの展望を提供します。-next カーネルの実行テストを行う冒険好きなテスターは大いに歓迎されます。

* バグレポート

https://bugzilla.kernel.org は Linux カーネル開発者がカーネルのバグを追跡する場所です。ユーザは見つけたバグの全てをこのツールで報告すべきです。どう kernel bugzilla を使うかの詳細は、以下を参照してください -

> https://bugzilla.kernel.org/page.cgi?id=faq.html

メインカーネルソースディレクトリにあるファイル admin-guide/reporting-bugs.rst はカーネルバグらしいものについてどうレポートするかの良いテンプレートであり、問題の追跡を助けるためにカーネル開発者にとってどんな情報が必要なのかの詳細が書かれています。

* バグレポートの管理

あなたのハッキングのスキルを訓練する最高の方法のひとつに、他人がレポートしたバグを修正することがあります。あなたがカーネルをより安定化させるこに寄与するということだけでなく、あなたは現実の問題を修正することを学び、自分のスキルも強化でき、また他の開発者があなたの存在に気がつきます。バグを修正することは、多くの開発者の中から自分が功績をあげる最善の道です、なぜなら多くの人は他人のバグの修正に時間を浪費することを好まないからです。

すでにレポートされたバグのために仕事をするためには、https://bugzilla.kernel.org に行ってください。もし今後のバグレポートについてアドバイスを受けたいのであれば、bugme-new メーリングリスト (新しいバグレポートだけがここにメールされる) または bugme-janitor メーリングリスト (bugzilla の変更毎にここにメールされる) を購読できます。

> https://lists.linux-foundation.org/mailman/listinfo/bugme-new
>
> https://lists.linux-foundation.org/mailman/listinfo/bugme-janitors

* メーリングリスト

上のいくつかのドキュメントで述べていますが、コアカーネル開発者の大部分は Linux kernel メーリングリストに参加しています。このリストの登録/脱退の方法については以下を参照してください-

> http://vger.kernel.org/vger-lists.html#linux-kernel

このメーリングリストのアーカイブは web 上の多数の場所に存在します。これらのアーカイブを探すにはサーチエンジンを使いましょう。例えば-

> http://dir.gmane.org/gmane.linux.kernel

リストに投稿する前にすでにその話題がアーカイブに存在するかどうかを検索することを是非やってください。多数の事がすでに詳細に渡って議論されており、アーカイブにのみ記録されています。

大部分のカーネルサブシステムも自分の個別の開発を実施するメーリングリストを持っています。個々のグループがどんなリストを持っているかは、MAINTAINERS ファイルにリストがありますので参照してください。

多くのリストは kernel.org でホストされています。これらの情報は以下にあります -

> http://vger.kernel.org/vger-lists.html

メーリングリストを使う場合、良い行動習慣に従うようにしましょう。少し安っぽいが、以下の URL は上のリスト (や他のリスト) で会話する場合のシンプルなガイドラインを示しています -

> http://www.albion.com/netiquette/

もし複数の人があなたのメールに返事をした場合、CC: で受ける人のリストはだいぶ多くなるでしょう。正当な理由がない限り、CC: リストから誰かを削除をしないように、また、メーリングリストのアドレスだけにリプライすることのないようにしましょう。1 つは送信者から、もう 1 つはリストからのように、メールを 2 回受けることになってもそれに慣れ、しゃれたメールヘッダーを追加してこの状態を変えようとしないように。人々はそのようなことは好みません。

今までのメールでのやりとりとその間のあなたの発言はそのまま残し、“John Kernelhacker wrote …:”の行をあなたのリプライの先頭行にして、メールの先頭でなく、各引用行の間にあなたの言いたいことを追加するべきです。

もしパッチをメールに付ける場合は、Documentation/process/submitting-patches.rstに提示されているように、それはプレーンな可読テキストにすることを忘れないようにしましょう。カーネル開発者は添付や圧縮したパッチを扱いたがりません。彼らはあなたのパッチの行毎にコメントを入れたいので、そうするしかありません。あなたのメールプログラムが空白やタブを圧縮しないように確認しましょう。最初の良いテストとしては、自分にメールを送ってみて、そのパッチを自分で当ててみることです。もしそれがうまく行かないなら、あなたのメールプログラムを直してもらうか、正しく動くように変えるべきです。

何をおいても、他の購読者に対する敬意を表すことを忘れないでください。

* コミュニティと共に働くこと

カーネルコミュニティのゴールは可能なかぎり最高のカーネルを提供することです。あなたがパッチを受け入れてもらうために投稿した場合、それは、技術的メリットだけがレビューされます。その際、あなたは何を予想すべきでしょうか?

- 批判
- コメント
- 変更の要求
- パッチの正当性の証明要求
- 沈黙

思い出してください、これはあなたのパッチをカーネルに入れる話です。あなたは、あなたのパッチに対する批判とコメントを受け入れるべきで、それらを技術的レベルで評価して、パッチを再作成するか、なぜそれらの変更をすべきでないかを明確で簡潔な理由の説明を提供してください。もし、あなたのパッチに何も反応がない場合、たまにはメールの山に埋もれて見逃され、あなたの投稿が忘れられてしまうこともあるので、数日待って再度投稿してください。

あなたがやるべきでないことは?

- 質問なしにあなたのパッチが受け入れられると想像すること
- 守りに入ること
- コメントを無視すること
- 要求された変更を何もしないでパッチを出し直すこと

可能な限り最高の技術的解決を求めているコミュニティでは、パッチがどのくらい有益なのかについては常に異なる意見があります。あなたは協調的であるべきですし、また、あなたのアイディアをカーネルに対してうまく合わせるようにすることが望まれています。もしくは、最低限あなたのアイディアがそれだけの価値があるとすすんで証明するようにしなければなりません。正しい解決に向かって進もうという意志がある限り、間違うことがあっても許容されることを忘れないでください。

あなたの最初のパッチに単に 1 ダースもの修正を求めるリストの返答になることも普通のことです。これはあなたのパッチが受け入れられないということでは **ありません**、そしてあなた自身に反対することを意味するのでも **ありません**。単に自分のパッチに対して指摘された問題を全て修正して再送すれば良いのです。

* カーネルコミュニティと企業組織のちがい

カーネルコミュニティは大部分の伝統的な会社の開発環境とは異ったやり方で動いています。以下は問題を避けるためにできると良いことのリストです。

あなたの提案する変更について言うときのうまい言い方 -

- “これは複数の問題を解決します”
- “これは 2000 行のコードを削除します”
- “以下のパッチは、私が言おうとしていることを説明するものです”
- “私はこれを 5 つの異なるアーキテクチャでテストしたのですが…”
- “以下は一連の小さなパッチ群ですが…”
- “これは典型的なマシンでの性能を向上させます…”

やめた方が良い悪い言い方 -

- “このやり方で AIX/ptx/Solaris ではできたので、できるはずだ…”
- “私はこれを 20 年もの間やってきた、だから…”
- “これは私の会社が金儲けをするために必要だ”
- “これは我々のエンタープライズ向け商品ラインのためである”
- “これは私が自分のアイディアを記述した、1000 ページの設計資料である”
- “私はこれについて、6 ケ月作業している…”
- “以下は…に関する 5000 行のパッチです”
- “私は現在のぐちゃぐちゃを全部書き直した、それが以下です…”
- “私は〆切がある、そのためこのパッチは今すぐ適用される必要がある”

カーネルコミュニティが大部分の伝統的なソフトウェアエンジニアリングの労働環境と異なるもう一つの点は、やりとりに顔を合わせないということです。email と irc を第一のコミュニケーションの形とする一つの利点は、性別や民族の差別がないことです。Linux カーネルの職場環境は女性や少数民族を受容します。なぜなら、email アドレスによってのみあなたが認識されるからです。国際的な側面からも活動領域を均等にするようにします。なぜならば、あなたは人の名前で性別を想像できないからです。ある男性がアンドレアという名前で、女性の名前はパットかもしれません (訳注 Andrea は米国では女性、それ以外 (欧州など) では男性名として使われることが多い。同様に、Pat は Patricia (主に女性名) や Patrick (主に男性名) の略称)。Linux カーネルの活動をして、意見を表明したことがある大部分の女性は、前向きな経験をもっています。

言葉の壁は英語が得意でない一部の人には問題になります。メーリングリストの中で、きちんとアイディアを交換するには、相当うまく英語を操れる必要があることもあります。そのため、自分のメールを送る前に英語で意味が通じているかをチェックすることをお薦めします。

* 変更を分割する

Linux カーネルコミュニティは、一度に大量のコードの塊を喜んで受容することはありません。変更は正確に説明される必要があり、議論され、小さい、個別の部分に分割する必要があります。これはこれまで多くの会社がやり慣れてきたことと全く正反対のことです。あなたのプロポーザルは、開発プロセスのとても早い段階から紹介されるべきです。そうすればあなたは自分のやっていることにフィードバックを得られます。これは、コミュニティからみれば、あなたが彼らと一緒にやっているように感じられ、単にあなたの提案する機能のゴミ捨て場として使っているのではない、と感じられるでしょう。しかし、一度に 50 もの email をメーリングリストに送りつけるようなことはやってはいけません、あなたのパッチ群はいつもどんな時でもそれよりは小さくなければなりません。

パッチを分割する理由は以下 -

1) 小さいパッチはあなたのパッチが適用される見込みを大きくします、カーネルの人達はパッチが正しいかどうかを確認する時間や労力をかけないからです。5 行のパッチはメンテナがたった 1 秒見るだけで適用できます。しかし、500 行のパッチは、正しいことをレビューするのに数時間かかるかもしれません (時間はパッチのサイズなどにより指数関数に比例してかかります)

   小さいパッチは何かあったときにデバッグもとても簡単になります。パッチを 1 個 1 個取り除くのは、とても大きなパッチを当てた後に (かつ、何かおかしくなった後で) 解剖するのに比べればとても簡単です。

2) 小さいパッチを送るだけでなく、送るまえに、書き直して、シンプルにする (もしくは、単に順番を変えるだけでも) ことも、とても重要です。

以下はカーネル開発者の Al Viro のたとえ話です -

> “生徒の数学の宿題を採点する先生のことを考えてみてください、先生は生徒が解に到達するまでの試行錯誤を見たいとは思わないでしょう。先生は簡潔な最高の解を見たいのです。良い生徒はこれを知っており、そして最終解の前の中間作業を提出することは決してないのです
>
> カーネル開発でもこれは同じです。メンテナ達とレビューア達は、問題を解決する解の背後になる思考プロセスを見たいとは思いません。彼らは単純であざやかな解決方法を見たいのです。”

あざやかな解を説明するのと、コミュニティと共に仕事をし、未解決の仕事を議論することのバランスをキープするのは難しいかもしれません。ですから、開発プロセスの早期段階で改善のためのフィードバックをもらうようにするのも良いですが、変更点を小さい部分に分割して全体ではまだ完成していない仕事を (部分的に) 取り込んでもらえるようにすることも良いことです。

また、でき上がっていないものや、”将来直す”ようなパッチを、本流に含めてもらうように送っても、それは受け付けられないことを理解してください。

* あなたの変更を正当化する

あなたのパッチを分割するのと同時に、なぜその変更を追加しなければならないかを Linux コミュニティに知らせることはとても重要です。新機能は必要性と有用性で正当化されなければなりません。

* あなたの変更を説明する

あなたのパッチを送付する場合には、メールの中のテキストで何を言うかについて、特別に注意を払ってください。この情報はパッチの ChangeLog に使われ、いつも皆がみられるように保管されます。これは次のような項目を含め、パッチを完全に記述するべきです -

- なぜ変更が必要か
- パッチ全体の設計アプローチ
- 実装の詳細
- テスト結果

これについて全てがどのようにあるべきかについての詳細は、以下のドキュメントの ChangeLog セクションを見てください -

> **“The Perfect Patch”**
> http://www.ozlabs.org/~akpm/stuff/tpp.txt

これらはどれも、実行することが時にはとても困難です。これらの例を完璧に実施するには数年かかるかもしれません。これは継続的な改善のプロセスであり、多くの忍耐と決意を必要とするものです。でも諦めないで、実現は可能です。多数の人がすでにできていますし、彼らも最初はあなたと同じところからスタートしたのですから。

Paolo Ciarrocchi に感謝、彼は彼の書いた“Development Process”(https://lwn.net/Articles/94386/) セクションをこのテキストの原型にすることを許可してくれました。Rundy Dunlap と Gerrit Huizenga はメーリングリストでやるべきこととやってはいけないことのリストを提供してくれました。以下の人々のレビュー、コメント、貢献に感謝。Pat Mochel, Hanna Linder, Randy Dunlap, Kay Sievers, Vojtech Pavlik, Jan Kara, Josh Boyer, Kees Cook, Andrew Morton, Andi Kleen, Vadim Lobanov, Jesper Juhl, Adrian Bunk, Keri Harris, Frans Pop, David A. Wheeler, Junio Hamano, Michael Kerrisk, と Alex Shepard 彼らの支援なしでは、このドキュメントはできなかったでしょう。

Maintainer: Greg Kroah-Hartman <greg@kroah.com>

CHAPTER

FIVE

DISCLAIMER

Translation's purpose is to ease reading and understanding in languages other than English. Its aim is to help people who do not understand English or have doubts about its interpretation. Additionally, some people prefer to read documentation in their native language, but please bear in mind that the *only* official documentation is the English one: linux_doc.

It is very unlikely that an update to linux_doc will be propagated immediately to all translations. Translations' maintainers - and contributors - follow the evolution of the official documentation and they maintain translations aligned as much as they can. For this reason there is no guarantee that a translation is up to date. If what you read in a translation does not sound right compared to what you read in the code, please inform the translation maintainer and - if you can - check also the English documentation.

A translation is not a fork of the official documentation, therefore translations' users should not find information that differs from the official English documentation. Any content addition, removal or update, must be applied to the English documents first. Afterwards and when possible, the same change should be applied to translations. Translations' maintainers accept only contributions that are merely translation related (e.g. new translations, updates, fixes).

Translations try to be as accurate as possible but it is not possible to map one language directly to all other languages. Each language has its own grammar and culture, so the translation of an English statement may need to be adapted to fit a different language. For this reason, when viewing translations, you may find slight differences that carry the same message but in a different form.

If you need to communicate with the Linux community but you do not feel comfortable writing in English, you can ask the translation's maintainers for help.

BIBLIOGRAPHY

[it-c-language] http://www.open-std.org/jtc1/sc22/wg14/www/standards

[it-gcc] https://gcc.gnu.org

[it-clang] https://clang.llvm.org

[it-icc] https://software.intel.com/en-us/c-compilers

[it-gcc-c-dialect-options] https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html

[it-gnu-extensions] https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html

[it-gcc-attribute-syntax] https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html

[it-n2049] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf

INDEX