增加新的系统调用

================

本文档描述了在Linux内核中跟添加新的系统调用相关的内容（除了在Documentation/SubmittingPatches文件中提到的一般的方式之外的）。

系统调用的替代选项

------------------

首先要考虑的一件事是增加一个系统调用是否是一个合适的选择。尽管系统调用是用户空间和内核之间最为传统以及显著的交互点，仍然应该考虑其他的可能性——为接口选择其最适合的。

* 如果操作涉及的对象为类文件系统的，创建一个新的文件系统或设备或许是更好的选择。这么做也使得其能更为容易的作为一个新的功能在内核模块被封装，而不需要内建在主要内核区。

-如果新的功能涉及以下操作：当有事件发生的时候，内核通知用户空间并为相关的对象返回一个新的文件描述符允许用户空间使用poll/select/epoll来接收通知。

-但是，那些没有映射到类读写操作的操作，必须以ioctl请求的形式被实现，会导致不透明的API。

* 如果你只是想知道一些运行时的系统信息，一个sysfs中的新的节点（详见Documentation/filesystems/sysfs.txt）或/proc文件系统更为合适。访问这些结构需要相关的文件系统是已挂载的，但这并不是总能满足的（例如在namespaced/sandboxed/chrooted的环境下）。应该避免在debugfs增加API。
* 如果操作是针对一个特定的文件或文件描述符的，一个附加的fcntl命令选项更为合适。但是，fcntl是一个复杂的多路系统调用，因此，这个选项最适合的是当新的函数是近似于已存的fcntl功能，或是新的功能是很简单的（例如：设置/更改一个文件描述符的简单标志）。
* 如果操作是针对一个特定的任务或进程的，一个新的prctl命令选项更为合适。正像fcntl一样，这个系统调用是很复杂的，适合的情况类比上一项。

设计API：扩展前的计划

---------------------

一个组成内核API的新的系统调用必须是被无限期的支持的。正因如此，就需要对要放在内核邮件清单上的接口进行判断，同时，也需要为将来的接口的扩展做好准备。

（syscall表杂乱无章地堆放了许多历史的用例以及相应的后续的系统调用，例如：eventfd/eventfd2，dup2/dup3，inotify\_init/inotify\_init1，pipe/pipe2, renameat/renameat2，因此，需要从最开始就了解内核的历史，并未后续的扩展做好计划。）

对于简单的只有几个参数的系统调用来说，为了保证未来的可扩展性，最好添加一个flags参数。要确保用户空间程序能安全地在内核版本间使用该flags，需检查其值是否含有其他未知flags，并且拒绝以下情况（含EINVAL）的系统调用：

if (flags & ~(THING\_FLAG1 | THING\_FLAG2 | THING\_FLAG3))

return -EINVAL;

（如果没有flags的值被使用，则确保flags参数为0。）

对于包含大量参数的系统调用来说，最好将大多数的参数封装在结构体中并以一个指针进行传递。这样的结构体可以通过包含一个size参数的形式来保证未来的扩展：

struct xyzzy\_params {

u32 size; /\* 用户空间设置 p->size = sizeof(struct xyzzy\_params) \*/

u32 param\_1;

u64 param\_2;

u64 param\_3;

};

因此，后续增加的字段，例如param\_4，只要将之前的操作赋零，就能解决版本间不协调的问题：

* 解决新版本的用户空间程序调用旧版本的内核的问题，内核代码只要检查是否超出结构体大小的内存的值为零就行（更有效率的做法是检查param\_4是否为零）。
* 解决旧版本的用户空间程序调用新版本的内核的问题，内核代码可以零扩展一个小的结构体用例（更有效率的做法是将param\_4置零）。

此方法的实现可参考perf\_event\_open(2) 和 the perf\_copy\_attr() 函数 （在文件夹

kernel/events/core.c中）。

设计API：其他方面的考量

------------------------

如果你的新的系统调用允许用户空间应用内核对象，应该使用文件描述符来处理那个对象——当内核已为使用文件描述符有完整定义的语义机制时，不要试图创建一个新的用户空间对象来处理。

当你的新的系统调用返回一个新的文件描述符时，flags参数必须包含一个等值于在一个新的FD上设置O\_CLOEXEC的值。这使得用户空间于关闭xyzzy() 及调用 fcntl(fd, F\_SETFD, FD\_CLOEXEC)的时序窗口成为可能，因为另一线程中的意料之外的fork() 和 execve()可能会将描述符泄漏给执行程序。

如果你的系统调用返回一个新的文件描述符，那就应该考虑使用poll族系统调用对那个文件描述符意味着什么。创建一个读写的文件描述符，是内核告知用户空间事件发生在相应的内核对象的一种常见方式。

如果你的xyzzy(2)系统调用涉及一个文件名参数：

int sys\_xyzzy(const char \_\_user \*path, ..., unsigned int flags);

你应该考虑xyzzyat(2)是否更为合适：

int sys\_xyzzyat(int dfd, const char \_\_user \*path, ..., unsigned int flags);

这为用户空间如何明确上述问题中的文件提供了灵活性；尤其是，它允许用户空间请求一个已打开的文件描述符使用AT\_EMPTY\_PATH标志，让fxyzzy(3)操作更为自由：

- xyzzyat(AT\_FDCWD, path, ..., 0) 相当于 xyzzy(path,...)

- xyzzyat(fd, "", ..., AT\_EMPTY\_PATH) 相当于 fxyzzy(fd, ...)

（更详细的关于\*at()调用，参见openat(2)的参考手册；需查看AT\_EMPTY\_PATH的例子，参考statat(2)的参考手册。）

如果你的系统调用涉及一个描述文件内偏移量的参数，将其设置为loff\_t类型，这能使得64位的偏移量能在32位的体系结构中被支持。

如果你的xyzzy(2)系统调用涉及特权性功能，则需要被合适的Linux容量位管理（调用capable()来进行检查），就像在capabilities(7)参考手册中描述的那样。选择一个已存的容量位来管理，尽量避免将大量仅模糊相关的函数结合在同一位之下。

如果你的xyzzy(2)系统调用能操控一个进程而不是调用一个进程，那么其需要被约束（通过调用ptrace\_may\_access()），这能确保调用进程和目标进程拥有相同的权限，或者在获取到必要前提之下，可以操控目标进程。

最后，需注意一些非x86体系结构能为更容易，如果系统调用的变量是明确的64位指向奇数的参数。（例如1,3,5），被允许使用连续的成对的32位寄存器（这不适用于参数是结构体的一员而被传入指针的情况。）

API提案

--------

为了使系统调用更便于再次审查，最好将补丁集分成不同的块。这些需包括至少以下几项不同的提交项目（每一项的详述如下）：

* 系统调用的核心实现，以及原型，通用编号，Kconfig变化和回退存根实现。
* 将新的系统调用连接到特定的体系结构上，通常为x86（包括x86\_64 ， x86\_32 和 x32）。
* 在tools/testing/selftests/文件目录下放置自测试程序来说明如何在用户空间使用新的系统调用。
* 新的系统调用的草稿参考手册，或是封面邮件中的简单文本，或是作为（独立的）参考手册的仓库的补丁。

新系统调用的提案，就像是内核API的改变一样，都应该被抄送到linux-api@vger.kernel.org。

一般系统调用实现

----------------

你的新的xyzzy(2)系统调用的主入口点应叫做sys\_xyzzy()，但是你应该为这个入口点添加合适的SYSCALL\_DEFINEn()宏而不是显式调用。其中的‘n’表示了系统调用的参数的个数，并且宏随其系统调用之名，其后跟（类型，名称）键值对作为参数。使用宏使得关于系统调用的元数据能为其他工具所使用。

新的系统调用需要在include/linux/syscalls.h中添加相应的入口点，标记为asmlinkage以匹配系统调用被调用的方式：

asmlinkage long sys\_xyzzy(...);

一些体系结构（如x86）有其特殊的系统调用表，单其他的一些体系结构是有通用的系统调用表的。将你的系统调用添加到通用的系统调用表中就需要在include/uapi/asm-generic/unistd.h中添加一个入口：

#define \_\_NR\_xyzzy 292

\_\_SYSCALL(\_\_NR\_xyzzy, sys\_xyzzy)

然后需要更新\_\_NR\_syscalls的数量来匹配额外的系统调用，需要注意的是，当有很多个系统调用被添加到同一个合并窗口中是，你的系统调用的数字可能会被调整以解决冲突。

kernel/sys\_ni.c文件为每一个系统调用提供了回退存根的实现，通过返回-ENOSYS。也在此添加你的系统调用：

cond\_syscall(sys\_xyzzy);

你的新的内核功能，以及控制其的相应的系统调用，在正常情况下应是可选择的，因此为其添加一个CONFIG选项（一般来说在init/Kconfig中）。通常对于新的CONFIG选项来说：

* 包含一个对新功能的描述以及被此选项控制的系统调用。
* 让这个选项依赖于EXPERT，如果它是对一般用户隐藏的。
* 使得任何实现函数的源文件依赖于Makefile中的CONFIG选项（例如："obj-$(CONFIG\_XYZZY\_SYSCALL) += xyzzy.c"）。
* 确认在新的CONFIG选项关闭时内核仍能正常运行。

综上，你需要确定以下几点：

* 为新功能提供CONFIG选项，通常在init/Kconfig中
* 设置SYSCALL\_DEFINEn(xyzzy, ...)为入口点
* 在include/linux/syscalls.h中包含相应的原型
* 在include/uapi/asm-generic/unistd.h添加通用表入口
* 在kernel/sys\_ni.c中添加回退存根

x86系统调用实现

----------------

将新系统调用连接到x86平台，需要更新主系统调用表。假设你的系统调用没有在哪方面有特殊之处（见下），那就在arch/x86/entry/syscalls/syscall\_64.tbl目录下涉及一个“通用”入口（对于x86\_64 和 x32）：

333 common xyzzy sys\_xyzzy

以及在arch/x86/entry/syscalls/syscall\_32.tbl目录下的“i386”的入口：

380 i386 xyzzy sys\_xyzzy

需强调的是，如果在相关的合并窗口中发生一些冲突的时候，这些数字是容易发生改变的。

兼容性系统调用（一般）

---------------------

对于大多数的系统调用来说，即使用户空间程序本身是32位的，同样的64位实现也是可以被调用的；即使系统调用的参数包含指针，因为这是完全透明的进行处理的。

但是，仍然存在几种情况需要兼容层来处理32位和64位之间的大小不同的问题。

首先，如果64位内核支持32位用户空间程序，另外也能解析(\_\_user)内存以同时支持32位和64位数值。当系统调用参数是如下情况这就是必须的：

* 一个指向指针的指针
* 一个指针指向包含指针的结构体（例如struct iovec \_\_user \*）
* 一个指针指向不同大小的整型（time\_t, off\_t, long, ...）
* 一个指针指向包含不同大小的整型的结构体

第二种需要兼容层的情况是，如果系统调用的其中一个参数的类型即使在32位的体系结构中仍然需要是64位的，例如loff\_t 或 \_\_u64。在这种情况下，一个64位的数值从64位内核到32位应用时需要被分成2个32位的，这就需要在兼容层中进行重置。

（需要注意的是，一个指向明确的64位类型的指针是不需要兼容层的，例如，splice(2)的参数类型loff\_t \_\_user \*并不需要compat\_系统调用。）

系统调用的兼容性版本应叫做compat\_sys\_xyzzy()，并添加COMPAT\_SYSCALL\_DEFINEn()宏，类似于SYSCALL\_DEFINEn。这版本的实现作为64位内核的一部分进行运行，但接收32位参数值并做相应处理。（也就是说，compat\_sys\_的版本转换数值到64位的版本，然后调用sys\_版本，或者同时调用一个内实现函数。）

兼容入口点也需要一个相应的函数原型，在include/linux/compat.h目录下，来匹配系统调用被调用的情况：

asmlinkage long compat\_sys\_xyzzy(...);

如果系统调用涉及结构体在32位和64位系统中是不同的，就像struct xyzzy\_args，那么include/linux/compat.h中的头文件需同时包含一个兼容版的结构体（struct

compat\_xyzzy\_args），使得每一个可变的字段有一个合适的compat\_类型以应对其在struct xyzzy\_args中的类型。这样，compat\_sys\_xyzzy()就能使用compat\_结构体来解析32位调用中的参数。

举个例子，有这样的字段：

struct xyzzy\_args {

const char \_\_user \*ptr;

\_\_kernel\_long\_t varying\_val;

u64 fixed\_val;

/\* ... \*/

};

那么，就应该有struct compat\_xyzzy\_args：

struct compat\_xyzzy\_args {

compat\_uptr\_t ptr;

compat\_long\_t varying\_val;

u64 fixed\_val;

/\* ... \*/

};

通用系统调用表需要为兼容性版本进行调整，则include/uapi/asm-generic/unistd.h中的入口应使用\_\_SC\_COMP而不是\_\_SYSCALL：

#define \_\_NR\_xyzzy 292

\_\_SC\_COMP(\_\_NR\_xyzzy, sys\_xyzzy, compat\_sys\_xyzzy)

综上，你需要：

* 一个COMPAT\_SYSCALL\_DEFINEn(xyzzy, ...)为兼容入口点
* 在include/linux/compat.h中添加相应的原型
* （如有需要）在include/linux/compat.h中添加32位映射结构体
* 在include/uapi/asm-generic/unistd.h中添加\_\_SC\_COMP而不是\_\_SYSCALL

兼容性系统调用（x86）

--------------------

将x86体系结构中的系统调用和兼容性版本连接在一起，需要调整系统调用表中的条目。

首先，arch/x86/entry/syscalls/syscall\_32.tbl中的条目应增加一列来表明32位用户空间程序在64位内核上运行会命中一个兼容入口点：

380 i386 xyzzy sys\_xyzzy compat\_sys\_xyzzy

第二，你要搞清楚在x32 ABI版本上新的系统调用是什么样的情况。有这样一点：参数的布局应匹配64位版本还是32位版本。

如果存在指针指向指针的情况，那解决方式就是：由于x32是ILP32，所以布局应为32位版本，arch/x86/entry/syscalls/syscall\_64.tbl中的条目是分割开的以保证x32程序能够命中兼容层：

333 64 xyzzy sys\_xyzzy

...

555 x32 xyzzy compat\_sys\_xyzzy

如果不涉及指针，那么更倾向于为x32 ABI重用64位系统调用（则arch/x86/entry/syscalls/syscall\_64.tbl中的条目是不变的）。

在其他情况下，你应该检查涉及你的参数布局的数据类型准确地从x32 (-mx32)进行了映射，不论是32位还是64位。

返回到别处的系统调用

--------------------

对于大多数的系统调用来说，一旦系统调用完成用户程序就会从其停止的地方继续运行——也就是说，在下一个指令开始之前，堆栈、寄存器以及虚拟内存都会恢复到系统调用开始之前的状态。

但是，少数的系统调用不是这样的。他们会返回到不同的位置（像rt\_sigreturn），或是改变内存空间（像fork/vfork/clone），甚至是程序的结构（像execve/execveat）。

为了实现此种情况，系统调用的内核实现可能需要为内核堆栈保存并恢复额外的寄存器，以确保程序能在系统调用之后准确地运行。

这在不同的体系结构中都应是不同的，但大体上都涉及定义组合入口点使其能保存/恢复额外寄存器以及调用真正的系统调用入口点。

对于x86\_64，这是作为arch/x86/entry/entry\_64.S中的stub\_xyzzy入口点进行实现的，并且系统调用表中的条目（arch/x86/entry/syscalls/syscall\_64.tbl）应进行调整以匹配：

333 common xyzzy stub\_xyzzy

对于在64位内核上运行的32位程序来说是stub32\_xyzzy，并需要在arch/x86/entry/entry\_64\_compat.S进行实现，以及相应地调整arch/x86/entry/syscalls/syscall\_32.tbl中的系统调用表：

380 i386 xyzzy sys\_xyzzy stub32\_xyzzy

如果系统调用需要兼容层（就像在之前的章节中提到的），那么stub32\_版本的就需要调用compat\_sys\_版的系统调用而不是原本的64位版的。还有，如果x32 ABI版的实现和x86\_64版的是不通用的，那么它的系统调用表也需要调用存根以调用compat\_sys\_版本。

从完整性考虑，最好设置一个映射使得其用户模式Linux也能运行——它的系统调用表能够应用stub\_xyzzy，但是UML的构造不包括include arch/x86/entry/entry\_64.S（因为UML模拟了寄存器等）。解决这个问题就像增加一个#define到arch/x86/um/sys\_call\_table\_64.c一样简单：

#define stub\_xyzzy sys\_xyzzy

其他细节

--------

多数内核以一般方式对待系统调用，但存在例外情况其可能需要根据你的特殊的系统调用进行更新。

审计子系统就是这样一个例外的例子；它包含(arch-specific)函数，能够对一些特殊类型的系统调用进行分类，尤其是文件打开(open/openat)，程序执行(execve/exeveat)和套接字多路复用器(socketcall)操作。如果你的系统调用近似以上这些，那么审计系统就需要被更新。

一般来说，如果存在一个系统调用是近似你的新的系统调用的，那就值得花时间在内核范围进行搜索以确认在现存系统调用中不存在其他例外情况。

测试

----

一个新的系统调用显然是需要被测试的；这也有助于为审核者提供示例此系统调用是如何在用户空间程序中被使用的。一个好的将这些目的结合在一起的方式是在tools/testing/selftests/目录下创建新的文件目录来包含一个简单的自测试程序。

对于一个新的系统调用来说，显然没有libc库封装函数，因此这些测试需要调用syscall()来唤醒；另外，如果系统调用涉及新的用户空间可视的结构，则需要安装相应的头文件来编译测试程序

需确保自测试程序在所有支持的体系结构中能够成功运行。例如，检查其是否能作为一个x86\_64 (-m64)， x86\_32 (-m32) 或 x32 (-mx32) ABI 程序进行编译。

为了更为全面彻底地测试新的功能，你应该考虑在Linux Test Project增加测试，或是在xfstests project上以检测文件系统相关的改变。

- https://linux-test-project.github.io/

- git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git

参考手册

--------

所有的新的系统调用都应该有完整的参考手册，最理想的是使用groff标记，但简单的文本也是可以的。如果使用了groff，那么在为补丁集写的封面邮件中包含一个预渲染的ASCII版参考手册，将极大地方便审核者。

参考手册需抄送至linux-man@vger.kernel.org

获取更多详细信息，参考https://www.kernel.org/doc/man-pages/patches.html

资源及引用

----------

- LWN article from Michael Kerrisk on use of flags argument in system calls:

https://lwn.net/Articles/585415/

- LWN article from Michael Kerrisk on how to handle unknown flags in a system

call: https://lwn.net/Articles/588444/

- LWN article from Jake Edge describing constraints on 64-bit system call

arguments: https://lwn.net/Articles/311630/

- Pair of LWN articles from David Drysdale that describe the system call

implementation paths in detail for v3.14:

- https://lwn.net/Articles/604287/

- https://lwn.net/Articles/604515/

- Architecture-specific requirements for system calls are discussed in the

syscall(2) man-page:

http://man7.org/linux/man-pages/man2/syscall.2.html#NOTES

- Collated emails from Linus Torvalds discussing the problems with ioctl():

http://yarchive.net/comp/linux/ioctl.html

- "How to not invent kernel interfaces", Arnd Bergmann,

http://www.ukuug.org/events/linux2007/2007/papers/Bergmann.pdf

- LWN article from Michael Kerrisk on avoiding new uses of CAP\_SYS\_ADMIN:

https://lwn.net/Articles/486306/

- Recommendation from Andrew Morton that all related information for a new

system call should come in the same email thread:

https://lkml.org/lkml/2014/7/24/641

- Recommendation from Michael Kerrisk that a new system call should come with

a man page: https://lkml.org/lkml/2014/6/13/309

- Suggestion from Thomas Gleixner that x86 wire-up should be in a separate

commit: https://lkml.org/lkml/2014/11/19/254

- Suggestion from Greg Kroah-Hartman that it's good for new system calls to

come with a man-page & selftest: https://lkml.org/lkml/2014/3/19/710

- Discussion from Michael Kerrisk of new system call vs. prctl(2) extension:

https://lkml.org/lkml/2014/6/3/411

- Suggestion from Ingo Molnar that system calls that involve multiple

arguments should encapsulate those arguments in a struct, which includes a

size field for future extensibility: https://lkml.org/lkml/2015/7/30/117

- Numbering oddities arising from (re-)use of O\_\* numbering space flags:

- 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

- Recommendation from Greg Kroah-Hartman that unknown flags should be

policed: https://lkml.org/lkml/2014/7/17/577

- Recommendation from Linus Torvalds that x32 system calls should prefer

compatibility with 64-bit versions rather than 32-bit versions:

https://lkml.org/lkml/2011/8/31/244

Adding a New System Call

========================

This document describes what's involved in adding a new system call to the

Linux kernel, over and above the normal submission advice in

Documentation/SubmittingPatches.

System Call Alternatives

------------------------

The first thing to consider when adding a new system call is whether one of

the alternatives might be suitable instead. Although system calls are the

most traditional and most obvious interaction points between userspace and the

kernel, there are other possibilities -- choose what fits best for your

interface.

- If the operations involved can be made to look like a filesystem-like

object, it may make more sense to create a new filesystem or device. This

also makes it easier to encapsulate the new functionality in a kernel module

rather than requiring it to be built into the main kernel.

- If the new functionality involves operations where the kernel notifies

userspace that something has happened, then returning a new file

descriptor for the relevant object allows userspace to use

poll/select/epoll to receive that notification.

- However, operations that don't map to read(2)/write(2)-like operations

have to be implemented as ioctl(2) requests, which can lead to a

somewhat opaque API.

- If you're just exposing runtime system information, a new node in sysfs

(see Documentation/filesystems/sysfs.txt) or the /proc filesystem may be

more appropriate. However, access to these mechanisms requires that the

relevant filesystem is mounted, which might not always be the case (e.g.

in a namespaced/sandboxed/chrooted environment). Avoid adding any API to

debugfs, as this is not considered a 'production' interface to userspace.

- If the operation is specific to a particular file or file descriptor, then

an additional fcntl(2) command option may be more appropriate. However,

fcntl(2) is a multiplexing system call that hides a lot of complexity, so

this option is best for when the new function is closely analogous to

existing fcntl(2) functionality, or the new functionality is very simple

(for example, getting/setting a simple flag related to a file descriptor).

- If the operation is specific to a particular task or process, then an

additional prctl(2) command option may be more appropriate. As with

fcntl(2), this system call is a complicated multiplexor so is best reserved

for near-analogs of existing prctl() commands or getting/setting a simple

flag related to a process.

Designing the API: Planning for Extension

-----------------------------------------

A new system call forms part of the API of the kernel, and has to be supported

indefinitely. As such, it's a very good idea to explicitly discuss the

interface on the kernel mailing list, and it's important to plan for future

extensions of the interface.

(The syscall table is littered with historical examples where this wasn't done,

together with the corresponding follow-up system calls -- eventfd/eventfd2,

dup2/dup3, inotify\_init/inotify\_init1, pipe/pipe2, renameat/renameat2 -- so

learn from the history of the kernel and plan for extensions from the start.)

For simpler system calls that only take a couple of arguments, the preferred

way to allow for future extensibility is to include a flags argument to the

system call. To make sure that userspace programs can safely use flags

between kernel versions, check whether the flags value holds any unknown

flags, and reject the system call (with EINVAL) if it does:

if (flags & ~(THING\_FLAG1 | THING\_FLAG2 | THING\_FLAG3))

return -EINVAL;

(If no flags values are used yet, check that the flags argument is zero.)

For more sophisticated system calls that involve a larger number of arguments,

it's preferred to encapsulate the majority of the arguments into a structure

that is passed in by pointer. Such a structure can cope with future extension

by including a size argument in the structure:

struct xyzzy\_params {

u32 size; /\* userspace sets p->size = sizeof(struct xyzzy\_params) \*/

u32 param\_1;

u64 param\_2;

u64 param\_3;

};

As long as any subsequently added field, say param\_4, is designed so that a

zero value gives the previous behaviour, then this allows both directions of

version mismatch:

- To cope with a later userspace program calling an older kernel, the kernel

code should check that any memory beyond the size of the structure that it

expects is zero (effectively checking that param\_4 == 0).

- To cope with an older userspace program calling a newer kernel, the kernel

code can zero-extend a smaller instance of the structure (effectively

setting param\_4 = 0).

See perf\_event\_open(2) and the perf\_copy\_attr() function (in

kernel/events/core.c) for an example of this approach.

Designing the API: Other Considerations

---------------------------------------

If your new system call allows userspace to refer to a kernel object, it

should use a file descriptor as the handle for that object -- don't invent a

new type of userspace object handle when the kernel already has mechanisms and

well-defined semantics for using file descriptors.

If your new xyzzy(2) system call does return a new file descriptor, then the

flags argument should include a value that is equivalent to setting O\_CLOEXEC

on the new FD. This makes it possible for userspace to close the timing

window between xyzzy() and calling fcntl(fd, F\_SETFD, FD\_CLOEXEC), where an

unexpected fork() and execve() in another thread could leak a descriptor to

the exec'ed program. (However, resist the temptation to re-use the actual value

of the O\_CLOEXEC constant, as it is architecture-specific and is part of a

numbering space of O\_\* flags that is fairly full.)

If your system call returns a new file descriptor, you should also consider

what it means to use the poll(2) family of system calls on that file

descriptor. Making a file descriptor ready for reading or writing is the

normal way for the kernel to indicate to userspace that an event has

occurred on the corresponding kernel object.

If your new xyzzy(2) system call involves a filename argument:

int sys\_xyzzy(const char \_\_user \*path, ..., unsigned int flags);

you should also consider whether an xyzzyat(2) version is more appropriate:

int sys\_xyzzyat(int dfd, const char \_\_user \*path, ..., unsigned int flags);

This allows more flexibility for how userspace specifies the file in question;

in particular it allows userspace to request the functionality for an

already-opened file descriptor using the AT\_EMPTY\_PATH flag, effectively giving

an fxyzzy(3) operation for free:

- xyzzyat(AT\_FDCWD, path, ..., 0) is equivalent to xyzzy(path,...)

- xyzzyat(fd, "", ..., AT\_EMPTY\_PATH) is equivalent to fxyzzy(fd, ...)

(For more details on the rationale of the \*at() calls, see the openat(2) man

page; for an example of AT\_EMPTY\_PATH, see the statat(2) man page.)

If your new xyzzy(2) system call involves a parameter describing an offset

within a file, make its type loff\_t so that 64-bit offsets can be supported

even on 32-bit architectures.

If your new xyzzy(2) system call involves privileged functionality, it needs

to be governed by the appropriate Linux capability bit (checked with a call to

capable()), as described in the capabilities(7) man page. Choose an existing

capability bit that governs related functionality, but try to avoid combining

lots of only vaguely related functions together under the same bit, as this

goes against capabilities' purpose of splitting the power of root. In

particular, avoid adding new uses of the already overly-general CAP\_SYS\_ADMIN

capability.

If your new xyzzy(2) system call manipulates a process other than the calling

process, it should be restricted (using a call to ptrace\_may\_access()) so that

only a calling process with the same permissions as the target process, or

with the necessary capabilities, can manipulate the target process.

Finally, be aware that some non-x86 architectures have an easier time if

system call parameters that are explicitly 64-bit fall on odd-numbered

arguments (i.e. parameter 1, 3, 5), to allow use of contiguous pairs of 32-bit

registers. (This concern does not apply if the arguments are part of a

structure that's passed in by pointer.)

Proposing the API

-----------------

To make new system calls easy to review, it's best to divide up the patchset

into separate chunks. These should include at least the following items as

distinct commits (each of which is described further below):

- The core implementation of the system call, together with prototypes,

generic numbering, Kconfig changes and fallback stub implementation.

- Wiring up of the new system call for one particular architecture, usually

x86 (including all of x86\_64, x86\_32 and x32).

- A demonstration of the use of the new system call in userspace via a

selftest in tools/testing/selftests/.

- A draft man-page for the new system call, either as plain text in the

cover letter, or as a patch to the (separate) man-pages repository.

New system call proposals, like any change to the kernel's API, should always

be cc'ed to linux-api@vger.kernel.org.

Generic System Call Implementation

----------------------------------

The main entry point for your new xyzzy(2) system call will be called

sys\_xyzzy(), but you add this entry point with the appropriate

SYSCALL\_DEFINEn() macro rather than explicitly. The 'n' indicates the number

of arguments to the system call, and the macro takes the system call name

followed by the (type, name) pairs for the parameters as arguments. Using

this macro allows metadata about the new system call to be made available for

other tools.

The new entry point also needs a corresponding function prototype, in

include/linux/syscalls.h, marked as asmlinkage to match the way that system

calls are invoked:

asmlinkage long sys\_xyzzy(...);

Some architectures (e.g. x86) have their own architecture-specific syscall

tables, but several other architectures share a generic syscall table. Add your

new system call to the generic list by adding an entry to the list in

include/uapi/asm-generic/unistd.h:

#define \_\_NR\_xyzzy 292

\_\_SYSCALL(\_\_NR\_xyzzy, sys\_xyzzy)

Also update the \_\_NR\_syscalls count to reflect the additional system call, and

note that if multiple new system calls are added in the same merge window,

your new syscall number may get adjusted to resolve conflicts.

The file kernel/sys\_ni.c provides a fallback stub implementation of each system

call, returning -ENOSYS. Add your new system call here too:

cond\_syscall(sys\_xyzzy);

Your new kernel functionality, and the system call that controls it, should

normally be optional, so add a CONFIG option (typically to init/Kconfig) for

it. As usual for new CONFIG options:

- Include a description of the new functionality and system call controlled

by the option.

- Make the option depend on EXPERT if it should be hidden from normal users.

- Make any new source files implementing the function dependent on the CONFIG

option in the Makefile (e.g. "obj-$(CONFIG\_XYZZY\_SYSCALL) += xyzzy.c").

- Double check that the kernel still builds with the new CONFIG option turned

off.

To summarize, you need a commit that includes:

- CONFIG option for the new function, normally in init/Kconfig

- SYSCALL\_DEFINEn(xyzzy, ...) for the entry point

- corresponding prototype in include/linux/syscalls.h

- generic table entry in include/uapi/asm-generic/unistd.h

- fallback stub in kernel/sys\_ni.c

x86 System Call Implementation

------------------------------

To wire up your new system call for x86 platforms, you need to update the

master syscall tables. Assuming your new system call isn't special in some

way (see below), this involves a "common" entry (for x86\_64 and x32) in

arch/x86/entry/syscalls/syscall\_64.tbl:

333 common xyzzy sys\_xyzzy

and an "i386" entry in arch/x86/entry/syscalls/syscall\_32.tbl:

380 i386 xyzzy sys\_xyzzy

Again, these numbers are liable to be changed if there are conflicts in the

relevant merge window.

Compatibility System Calls (Generic)

------------------------------------

For most system calls the same 64-bit implementation can be invoked even when

the userspace program is itself 32-bit; even if the system call's parameters

include an explicit pointer, this is handled transparently.

However, there are a couple of situations where a compatibility layer is

needed to cope with size differences between 32-bit and 64-bit.

The first is if the 64-bit kernel also supports 32-bit userspace programs, and

so needs to parse areas of (\_\_user) memory that could hold either 32-bit or

64-bit values. In particular, this is needed whenever a system call argument

is:

- a pointer to a pointer

- a pointer to a struct containing a pointer (e.g. struct iovec \_\_user \*)

- a pointer to a varying sized integral type (time\_t, off\_t, long, ...)

- a pointer to a struct containing a varying sized integral type.

The second situation that requires a compatibility layer is if one of the

system call's arguments has a type that is explicitly 64-bit even on a 32-bit

architecture, for example loff\_t or \_\_u64. In this case, a value that arrives

at a 64-bit kernel from a 32-bit application will be split into two 32-bit

values, which then need to be re-assembled in the compatibility layer.

(Note that a system call argument that's a pointer to an explicit 64-bit type

does \*not\* need a compatibility layer; for example, splice(2)'s arguments of

type loff\_t \_\_user \* do not trigger the need for a compat\_ system call.)

The compatibility version of the system call is called compat\_sys\_xyzzy(), and

is added with the COMPAT\_SYSCALL\_DEFINEn() macro, analogously to

SYSCALL\_DEFINEn. This version of the implementation runs as part of a 64-bit

kernel, but expects to receive 32-bit parameter values and does whatever is

needed to deal with them. (Typically, the compat\_sys\_ version converts the

values to 64-bit versions and either calls on to the sys\_ version, or both of

them call a common inner implementation function.)

The compat entry point also needs a corresponding function prototype, in

include/linux/compat.h, marked as asmlinkage to match the way that system

calls are invoked:

asmlinkage long compat\_sys\_xyzzy(...);

If the system call involves a structure that is laid out differently on 32-bit

and 64-bit systems, say struct xyzzy\_args, then the include/linux/compat.h

header file should also include a compat version of the structure (struct

compat\_xyzzy\_args) where each variable-size field has the appropriate compat\_

type that corresponds to the type in struct xyzzy\_args. The

compat\_sys\_xyzzy() routine can then use this compat\_ structure to parse the

arguments from a 32-bit invocation.

For example, if there are fields:

struct xyzzy\_args {

const char \_\_user \*ptr;

\_\_kernel\_long\_t varying\_val;

u64 fixed\_val;

/\* ... \*/

};

in struct xyzzy\_args, then struct compat\_xyzzy\_args would have:

struct compat\_xyzzy\_args {

compat\_uptr\_t ptr;

compat\_long\_t varying\_val;

u64 fixed\_val;

/\* ... \*/

};

The generic system call list also needs adjusting to allow for the compat

version; the entry in include/uapi/asm-generic/unistd.h should use

\_\_SC\_COMP rather than \_\_SYSCALL:

#define \_\_NR\_xyzzy 292

\_\_SC\_COMP(\_\_NR\_xyzzy, sys\_xyzzy, compat\_sys\_xyzzy)

To summarize, you need:

- a COMPAT\_SYSCALL\_DEFINEn(xyzzy, ...) for the compat entry point

- corresponding prototype in include/linux/compat.h

- (if needed) 32-bit mapping struct in include/linux/compat.h

- instance of \_\_SC\_COMP not \_\_SYSCALL in include/uapi/asm-generic/unistd.h

Compatibility System Calls (x86)

--------------------------------

To wire up the x86 architecture of a system call with a compatibility version,

the entries in the syscall tables need to be adjusted.

First, the entry in arch/x86/entry/syscalls/syscall\_32.tbl gets an extra

column to indicate that a 32-bit userspace program running on a 64-bit kernel

should hit the compat entry point:

380 i386 xyzzy sys\_xyzzy compat\_sys\_xyzzy

Second, you need to figure out what should happen for the x32 ABI version of

the new system call. There's a choice here: the layout of the arguments

should either match the 64-bit version or the 32-bit version.

If there's a pointer-to-a-pointer involved, the decision is easy: x32 is

ILP32, so the layout should match the 32-bit version, and the entry in

arch/x86/entry/syscalls/syscall\_64.tbl is split so that x32 programs hit the

compatibility wrapper:

333 64 xyzzy sys\_xyzzy

...

555 x32 xyzzy compat\_sys\_xyzzy

If no pointers are involved, then it is preferable to re-use the 64-bit system

call for the x32 ABI (and consequently the entry in

arch/x86/entry/syscalls/syscall\_64.tbl is unchanged).

In either case, you should check that the types involved in your argument

layout do indeed map exactly from x32 (-mx32) to either the 32-bit (-m32) or

64-bit (-m64) equivalents.

System Calls Returning Elsewhere

--------------------------------

For most system calls, once the system call is complete the user program

continues exactly where it left off -- at the next instruction, with the

stack the same and most of the registers the same as before the system call,

and with the same virtual memory space.

However, a few system calls do things differently. They might return to a

different location (rt\_sigreturn) or change the memory space (fork/vfork/clone)

or even architecture (execve/execveat) of the program.

To allow for this, the kernel implementation of the system call may need to

save and restore additional registers to the kernel stack, allowing complete

control of where and how execution continues after the system call.

This is arch-specific, but typically involves defining assembly entry points

that save/restore additional registers and invoke the real system call entry

point.

For x86\_64, this is implemented as a stub\_xyzzy entry point in

arch/x86/entry/entry\_64.S, and the entry in the syscall table

(arch/x86/entry/syscalls/syscall\_64.tbl) is adjusted to match:

333 common xyzzy stub\_xyzzy

The equivalent for 32-bit programs running on a 64-bit kernel is normally

called stub32\_xyzzy and implemented in arch/x86/entry/entry\_64\_compat.S,

with the corresponding syscall table adjustment in

arch/x86/entry/syscalls/syscall\_32.tbl:

380 i386 xyzzy sys\_xyzzy stub32\_xyzzy

If the system call needs a compatibility layer (as in the previous section)

then the stub32\_ version needs to call on to the compat\_sys\_ version of the

system call rather than the native 64-bit version. Also, if the x32 ABI

implementation is not common with the x86\_64 version, then its syscall

table will also need to invoke a stub that calls on to the compat\_sys\_

version.

For completeness, it's also nice to set up a mapping so that user-mode Linux

still works -- its syscall table will reference stub\_xyzzy, but the UML build

doesn't include arch/x86/entry/entry\_64.S implementation (because UML

simulates registers etc). Fixing this is as simple as adding a #define to

arch/x86/um/sys\_call\_table\_64.c:

#define stub\_xyzzy sys\_xyzzy

Other Details

-------------

Most of the kernel treats system calls in a generic way, but there is the

occasional exception that may need updating for your particular system call.

The audit subsystem is one such special case; it includes (arch-specific)

functions that classify some special types of system call -- specifically

file open (open/openat), program execution (execve/exeveat) or socket

multiplexor (socketcall) operations. If your new system call is analogous to

one of these, then the audit system should be updated.

More generally, if there is an existing system call that is analogous to your

new system call, it's worth doing a kernel-wide grep for the existing system

call to check there are no other special cases.

Testing

-------

A new system call should obviously be tested; it is also useful to provide

reviewers with a demonstration of how user space programs will use the system

call. A good way to combine these aims is to include a simple self-test

program in a new directory under tools/testing/selftests/.

For a new system call, there will obviously be no libc wrapper function and so

the test will need to invoke it using syscall(); also, if the system call

involves a new userspace-visible structure, the corresponding header will need

to be installed to compile the test.

Make sure the selftest runs successfully on all supported architectures. For

example, check that it works when compiled as an x86\_64 (-m64), x86\_32 (-m32)

and x32 (-mx32) ABI program.

For more extensive and thorough testing of new functionality, you should also

consider adding tests to the Linux Test Project, or to the xfstests project

for filesystem-related changes.

- https://linux-test-project.github.io/

- git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git

Man Page

--------

All new system calls should come with a complete man page, ideally using groff

markup, but plain text will do. If groff is used, it's helpful to include a

pre-rendered ASCII version of the man page in the cover email for the

patchset, for the convenience of reviewers.

The man page should be cc'ed to linux-man@vger.kernel.org

For more details, see https://www.kernel.org/doc/man-pages/patches.html

References and Sources

----------------------

- LWN article from Michael Kerrisk on use of flags argument in system calls:

https://lwn.net/Articles/585415/

- LWN article from Michael Kerrisk on how to handle unknown flags in a system

call: https://lwn.net/Articles/588444/

- LWN article from Jake Edge describing constraints on 64-bit system call

arguments: https://lwn.net/Articles/311630/

- Pair of LWN articles from David Drysdale that describe the system call

implementation paths in detail for v3.14:

- https://lwn.net/Articles/604287/

- https://lwn.net/Articles/604515/

- Architecture-specific requirements for system calls are discussed in the

syscall(2) man-page:

http://man7.org/linux/man-pages/man2/syscall.2.html#NOTES

- Collated emails from Linus Torvalds discussing the problems with ioctl():

http://yarchive.net/comp/linux/ioctl.html

- "How to not invent kernel interfaces", Arnd Bergmann,

http://www.ukuug.org/events/linux2007/2007/papers/Bergmann.pdf

- LWN article from Michael Kerrisk on avoiding new uses of CAP\_SYS\_ADMIN:

https://lwn.net/Articles/486306/

- Recommendation from Andrew Morton that all related information for a new

system call should come in the same email thread:

https://lkml.org/lkml/2014/7/24/641

- Recommendation from Michael Kerrisk that a new system call should come with

a man page: https://lkml.org/lkml/2014/6/13/309

- Suggestion from Thomas Gleixner that x86 wire-up should be in a separate

commit: https://lkml.org/lkml/2014/11/19/254

- Suggestion from Greg Kroah-Hartman that it's good for new system calls to

come with a man-page & selftest: https://lkml.org/lkml/2014/3/19/710

- Discussion from Michael Kerrisk of new system call vs. prctl(2) extension:

https://lkml.org/lkml/2014/6/3/411

- Suggestion from Ingo Molnar that system calls that involve multiple

arguments should encapsulate those arguments in a struct, which includes a

size field for future extensibility: https://lkml.org/lkml/2015/7/30/117

- Numbering oddities arising from (re-)use of O\_\* numbering space flags:

- 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

- Recommendation from Greg Kroah-Hartman that unknown flags should be

policed: https://lkml.org/lkml/2014/7/17/577

- Recommendation from Linus Torvalds that x32 system calls should prefer

compatibility with 64-bit versions rather than 32-bit versions:

https://lkml.org/lkml/2011/8/31/244