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 :ref:`Documentation/process/submitting-patches.rst <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 :manpage:`read(2)`/:manpage:`write(2)`-like operations have to be implemented as :manpage:`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 :manpage:`fcntl(2)` command option may be more appropriate. However, :manpage:`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 :manpage:`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 :manpage:`prctl(2)` command option may be more appropriate. As with :manpage:`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 :manpage:`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 :manpage:`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 :manpage:`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 :manpage:`xyzzy(2)` system call involves a filename argument:

```
int sys_xyzzy(const char __user *path, ..., unsigned int flags);
```

you should also consider whether an :manpage:`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 :manpage:`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 :manpage:`openat(2)` man page; for an example of AT_EMPTY_PATH, see the :manpage:`fstatat(2)` man page.)

If your new :manpage:`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 :manpage:`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 :manpage:`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 :manpage:`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 :manpage:`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, :manpage:`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 :manpage:`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. :manpage:`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