So you want to add a syscall?

Brooks Davis

Computer Science Laboratory
SRI International
Walla Walla, WA, USA
brooks.davis@sri.com

***Abstract*—Adding a new system call is notionally simple, but there are numerous edge cases that can confuse even senior developers. In this paper I cover the process of adding a system call including special handing for ABI compatibility layers like freebsd32. I also cover the extra requirements to support upcoming CHERI-extended architectures.**

I. Introduction

These days, adding as system call to the FreeBSD kernel and libc is a relatively straightforward task, and it's even fairly well documented in the FreeBSD Wiki.[1] Historically, it has been a bit of a fraught process, particularly when dealing with 32-bit compatibility (e.g., i386 and amd64). This paper covers the basics of adding a system call and explains the ins and outs of ABI compatibility, covering both 32-bit (i386 and non-i386) as well as 64-bit system-call support on systems that support CHERI [2], [3].

The basic process is simple. We add a single declaration to `sys/kern/syscalls.master` from which we generate kernel function declarations, system-call table entries, and make(1) variable declarations used to generate stubs in libc. With declarations in hand, we add an implementation to the kernel and a 32-bit compatibility implementation if required. For a simple system call, all that is left is to add a manual page. Later sections cover each of these steps along with common exceptions from the easy path.

Where things get trickier is knowing when to add a 32-bit compatibility implementation. Historically, even senior developers got this wrong with surprising regularity. Today, adding a `syscalls.master` entry will cause declarations to be generated for 32-bit compatibility if required in virtually every case so the process is streamlined. I'll cover the various cases that cause 32-bit compatibility to be required in the future and the various edge cases later in this paper.

Adding support for 64-bit system calls on CHERI is both easier and more complicated. All integer types are exactly the same as in CheriABI [1], so some system calls don't need translations, but pointers are 128-bits and include bounds so (fairly simple) compatibility wrappers are required for every system call that takes pointer arguments or has argument types that contain pointers.

[1]https://wiki.freebsd.org/AddingSyscalls

II. Why add a system call?

The main reason to add a system call is to access resources the kernel controls. For example, we use a wide range of system calls to access file systems and through them, their underlying storage. In examining how systems calls work below, we'll look at one of them: pwritev(2). The pwritev(2) system call takes an array of `struct iovec` pointing to data to write to a file descriptor.

Another reason to add a system call is to allow the kernel to act as a trusted intermediary. For example, sendmsg(2) can be used to send a file descriptor from one process to another over a Unix domain socket. This leads into another reason: avoiding excessive context switches. By passing a file descriptor, another process can access the contents of a file directly rather than having to ask the original holder to work on its behalf. Another example is the sendfile(2) system call that instructs the kernel to send (some of) the contents of a file to a network socket.

One final reason to add a system call is if an existing system call exists but its interface is insufficient. The most dramatic example of this is wait(2), wait4(2), and wait6(2), which are used to implement several more wait variants.

A. Why not add a system call?

So why not add a system call. First and foremost, system calls are forever[2]. FreeBSD supports most i386 binaries compiled against versions dating back to before the 1.0 release. This compatibility has non-trivial cost of adding extra code to support obsolete interfaces.

The other reason not to add a system call is that another interface may be more appropriate. In particular, anything managing a file or device via file descriptor might be a good candidate for an ioctl(1) command. Management data might make sense to provide via a sysctl(1). As a project we're somewhat more willing to punt on compatibility for read-only sysctl values, but only if consumers aren't critical.

B. How do system calls work?

Before we start adding system calls, we need to understand how they work. As we walk through key steps, we'll use the

[2]A very small number of system calls have been removed, most notably the system calls used to implement the Kernel Scheduler Entities (KSE) threading model in FreeBSD 5.

pwritev(2) system call as is illustrates many of the edge cases. Its declaration is:

```
ssize_t pwritev(int fd, struct iovec *iovp,
    u_int iovcnt, off_t offset);
```

C. Userspace stub

Most userspace programs call system calls by calling function stubs in libc. This insulates them from the details of the architecture-dependent system call implementation and allow interposition on system calls. At the very bottom of libc there is a function prefixed with `__sys_` which makes the actual call (for most syscalls there is an _ prefixed alias and an unprefixed alias, the latter should be used by programs and the former can be used in libc and libthr when an uninterposed versions are required). On amd64, the stub for pwritev(2) dissassembles as:

```
<__sys_pwritev>:
/* @generated by libc/sys/Makefile.inc */
#include "compat.h"
#include "SYS.h"
RSYSCALL(pwritev)
    b8 22 01 00 00          mov    $0x122,%eax
    49 89 ca                mov    %rcx,%r10
    0f 05                   syscall
    0f 82 b8 ed ff ff       jb     138778 <.cerror>
    c3                      ret
```

The `mov $0x122,%eax` stores `pwritev`'s system call number and `syscall` triggers a system call exception to enter the kernel.

D. Kernel overview

In the kernel the trap handler is invoked leading to `syscallenter()` being called to handle the system call. It calls `cpu_fetch_syscall_args()` to fill in a per-thread `struct syscall_args` (more on this later). `syscallenter()` also performs a number of tracing, auditing, and security operations, calls the implementation (`sys_pwritev()` in this case), and then calls `cpu_set_syscall_retval()` to update trapframe registers for return. After `syscallenter()` returns, `syscallret()` handles more tracing, debug interactions, and prepares to return to userspace. Of these, the bits that interest us mosts are return values and argument handling.

E. Return values

As return values are fairly simple, we'll get them out of the way first. Most system calls follow a convention where they return `0` or a positive value on success and `-1` on error with `errno` set to the error value. In userspace this is accomplished by setting an architecture specific status register to in indicate success or failure. If failure is indicated the `cerror()` function is called to retrieve the error value and set `errno`. The kernel will have set the return value register(s) for the function already.

Commonly, the kernel sets the return value register by setting `td->td_retval[0]` to the return value (it is `0` by default) with `cpu_set_syscall_retval()` setting the actual register entries in the trapframe. To return an error, the implementation simply returns a non-zero error value (e.g., `EINVAL`).

Exceptions to the trivial return pattern are the `pipe(2)` system call, which returns two values via `td->td_retval[0]` and `td->td_retval[1]` and system calls like lseek(2) that return 64-bit values. The latter set via `td->td_uretoff.tdu_off` in the normal case. For 32-bit architectures this splits the value across two registers automatically. There is one further twist for 32-bit compatibility. Because the native `tdu_off` aliases only `td_retval[0]` in the 64-bit implementation we need to split the value up again. This is done in `freebsd32_lseek` with:

```
off_t pos = td->td_uretoff.tdu_off;
td->td_retval[RETVAL_LO] = pos & 0xffffffff;
td->td_retval[RETVAL_HI] = pos >> 32;
```

F. In-kernel argument handling

Kernel argument is straight forward at its core, but there are a number of wrinkles related to ABI compatibility. First, let's start with the simple case. The `cpu_fetch_syscall_args()` function fills in a `struct syscall_args` from values store in the trap frame. The definition of `struct syscall_args` is:

```
struct syscall_args {
        u_int code;
        u_int original_code;
        struct sysent *callp;
        register_t args[8];
};
```

The `code` member is set to the system call number, `original_code` is used for system(2) and __system(2), `callp` points to the system call structure, and `args` holds arguments.

In FreeBSD each argument in the `args` array is the size of an native integer register (64 or 32-bits). Arguments are passed to the implementation by casting the `args` array to the implementation's user argument pointer (UAP) argument, which is a structure that aliases appropriately with the `args` array. For `sys_pwritev` the usable members of the argument are:

```
struct pwritev_args {
        int fd;
        struct iovec *iovp;
        u_int iovcnt;
        off_t offset;
};
```

You can see its use in the `sys_pwritev()` implementation:

```
int
sys_pwritev(struct thread *td, struct pwrite_args *uap)
{
        struct uio *auio;
        int error;

        error = copyinuio(uap->iovp, uap->iovcnt,
            &auio);
        if (error)
                return (error);
```

```
        error = kern_pwritev(td, uap->fd, auio,
            uap->offset);
        free(auio, M_IOV);
        return (error);
}
```

The real implementation lies in `kern_pwritev()` and is beyond the scope of this paper.

On a little-endian 64-bit system, the above declaration of `struct pwritev_args` would map to the array unchanged:

```
struct pwritev_args {
        int fd;
args[0] 01  00  00  00  00  00  00  00
        struct iovec *iovp;
args[1] 90  e9  ff  ff  ff  7f  00  00
        u_int iovcnt;
args[2] 02  00  00  00  00  00  00  00
        off_t offset;
args[3] 00  00  00  00  00  00  00  00
};
```

However on a big-endian system padding is required for the 32-bit members:

```
struct pwritev_args {
        int pl0;        int fd;
args[0] 00  00  00  00  00  00  00  01
        struct iovec *iovp;
args[1] 00  00  7f  ff  ff  ff  e9  90
        int pl2;        u_int iovcnt;
args[2] 00  00  00  00  00  00  00  02
        off_t offset;
args[3] 00  00  00  00  00  00  00  00
};
```

In practice, we always pad explicitly so the little-endian case looks notionally like:

```
struct pwritev_args {
        int fd;         int pr0;
args[0] 01  00  00  00  00  00  00  00
        struct iovec *iovp;
args[1] 90  e9  ff  ff  ff  7f  00  00
        u_int iovcnt;   int pr2;
args[2] 02  00  00  00  00  00  00  00
        off_t offset;
args[3] 00  00  00  00  00  00  00  00
};
```

In reality the generated definitions are uglier and look like:

```
struct pwritev_args {
        char fd_l_[PADL_(int)]; int fd;
            char fd_r_[PADR_(int)];
        char iovp_l_[PADL_(struct iovec *)];
            struct iovec *iovp;
            char iovp_r_[PADR_(struct iovec *)];
        char iovcnt_l_[PADL_(u_int)]; u_int iovcnt;
            char iovcnt_r_[PADR_(u_int)];
        char offset_l_[PADL_(off_t)]; off_t offset;
            char offset_r_[PADR_(off_t)];
};
```

Where the `PADL_` and `PADR_` macros expand as appropriate depending on the argument size and endianness of the target architecture.

1) 32-bit compatibility: The implementation of 32-bit compatibility uses clever tricks to limit the number of wrappers or shims that need to be written. The `args` array remains an array of 64-bit arguments, but only bottom 32-bits will ever be non-zero. This means that unsigned integer arguments (`size_t`, `unsigned long`, etc) and even pointers require no translation and can remain the same type in the UAP due to the implied zero extension.

Some types do require translation. First, any signed type that changes from 32-bit to 64-bit requires manual sign extension. Example include `ssize_t` and (on i386) `time_t`. Next, values that are always 64-bits such as `off_t` will be split between two registers and need to be glued back together. Finally, if a pointer points to a type whose ABI changes, the object in question must be translated for just by the kernel.

Putting this all together, the freebsd32 UAP for `pwritev` looks like this on i386:

```
struct freebsd32_pwritev_args {
        int fd;         int pr0;
args[0] 01  00  00  00  00  00  00  00
        struct iovec32 *iovp;
args[1] 50  db  ff  ff  00  00  00  00
        u_int iovcnt;   int pr2;
args[2] 02  00  00  00  00  00  00  00
        uint32_t offset1; int pr3;
args[3] 00  00  00  00  00  00  00  00
        uint32_t offset2; int pr4;
args[4] 00  00  00  00  00  00  00  00
};
```

For non-i386 32-bit systems where 64-bit values must be strongly aligned, a padding argument is also required as the arguments are passed in aligned register pairs so the structure actually looks like:

```
struct freebsd32_pwritev_args {
        int fd;         int pr0;
args[0] 01  00  00  00  00  00  00  00
        struct iovec32 *iovp;
args[1] 50  db  ff  ff  00  00  00  00
        u_int iovcnt;   int pr2;
args[2] 02  00  00  00  00  00  00  00
#ifndef i386
        int _pad;       int pr3;
#endif
args[3] 00  00  00  00  00  00  00  00
        uint32_t offset1; int pr4;
args[4] 00  00  00  00  00  00  00  00
        uint32_t offset2; int pr4;
args[4] 00  00  00  00  00  00  00  00
};
```

All of this is generated automatically from the central `syscall.master` entry. The `freebsd32_` variants are generated only when required as are the implementation declarations. Putting it all together, the 32-bit compatibility implementation looks almost identical to `sys_pwritev()` with a special copyinuio implementation to make a native `struct uio` from the 32-bit iovec and to glue the `offset` argument back together using the `PAIR32TO64` macro:

```
int
freebsd32_pwritev(struct thread *td,
    struct pwrite_args *uap)
{
        struct uio *auio;
        int error;

        error = freebsd32_copyinuio(uap->iovp,
            uap->iovcnt, &auio);
        if (error)
```

```
		return (error);
	error = kern_pwritev(td, uap->fd, auio,
	    PAIR32TO64(off_t, uap->offset));
	free(auio, M_IOV);
	return (error);
}
```

2) CHERI and 64-bit compatibility: Before we discuss 64-bit compatibility for CHERI systems, we need a brief introduction to CHERI. CHERI is an architectural extension that adds a new hardware type, the capability. Capabilities grant access to regions of address space. On CHERI systems, all accesses to memory are via capabilities, either explicitly via new instructions or implicitly via a default data capability (DDC) or program counter capability (PCC). CHERI capabilities contain addresses, bounds, and permissions. Bounds and permission may be reduced, but not increased and any attempt to directly manipulate the bits of a capability in memory or registers clears a validity tag. On 64-bit platforms, CHERI capabilities are 128-bits in memory with a 64-bit address, floating point compressed bounds, permissions, and with the tag stored to the side. CHERI has been ported to Armv8, RISC-V, and MIPS64 (now obsolete) with an early sketch for x86_64 in progress.

CheriBSD is a fork of FreeBSD with support for CHERI. When targeting an architecture with CHERI support, the default ABI (CheriABI) uses capabilities in place of integers for all pointers. Further, kernel functions that access userspace use capabilities without exception. As a result, all compatibility ABIs must transform their integer pointers into appropriate capabilities. Since the default ABI uses capabilities, `struct syscall_args` is modified such that the]texttt{args} member takes a `syscallarg_t`, a new type that can hold a capability on CHERI-aware systems and is a `register_t` elsewhere. Padding requirements for compatibility ABIs are similar to those for 32-bits, except that more padding is required due to the larger size, and that pointer arguments require manual handling. This means that all system calls that take pointers require handling.

Other than pointers, CheriABI is exactly the same as freebsd64 so that simplifies some aspects of the process. TABLE I shows the key differences between the 4 main ABIs FreeBSD supports.

The 64-bit compatibility implementation of `pwritev()` is similar to the 32-bit one, except that we've added a layer of indirection to share more code between the default, 32-bit, and 64-bit implementations with the help of a function pointer for `copyinuio()`:

```
int
user_pwritev(struct thread *td, int fd,
    struct iovec * __capability iovp, u_int iovcnt,
    off_t offset, copyinuio_t *copyinuio_f)
{
	struct uio *auio;
	int error;

	error = copyinuio_f(iovp, iovcnt, &auio);
	if (error)
		return (error);
	error = kern_pwritev(td, fd, auio, offset);
	free(auio, M_IOV);
	return (error);
}
```

Note the `__capability` annotation means that the `iovp` pointer is a capability even in a hybrid kernel (where only select pointers are capabilities). With this extra bit of indirection `sys_pwritev()` becomes:

```
int
sys_pwritev(struct thread *td,
    struct pwritev_args *uap)
{
	return (user_pwritev(td, uap->fd, uap->iovp,
	    uap->iovcnt, uap->offset, copyinuio));
}
```

and `freebsd64_pwritev()` is:

```
int
freebsd64_pwritev(struct thread *td,
    struct freebsd64_pwritev_args *uap)
{
	return (user_pwritev(td, uap->fd,
	    (struct iovec *__capability)
	    __USER_CAP_ARRAY(uap->iovp, uap->iovcnt),
	    uap->iovcnt, uap->offset,
	    freebsd64_copyinuio));
}
```

This differs from `sys_pwritev()` in that a capability must be created (here we use the `__USER_CAP_ARRAY` macro, which sets bounds on an array when the type is known along with the number of elements) and `freebsd64_copyinuio()` is passed as the `copyin()` function. The essential change vs 32-bit compatibility is the derivation of a capability to the `struct iovec64` array.

III. Adding a System Call

As outlined in the introduction, the actual process of adding a system call is straightforward. We declare it in `syscalls.master`, run a script to update generated files for relevant ABIs, add an implementation, any userspace bits, and a manual page.

A. syscalls.master and generated files

The first step in the process is to define the system call interface and declare it in `sys/kern/syscalls.master`. First, we'll discuss the entry for `pwritev(2)`:

```
290	AUE_PWRITEV	STD|CAPENABLED {
		ssize_t pwritev(
		    int fd,
		    _In_reads_(iovcnt)
		    _Contains_long_ptr_
		    struct iovec *iovp,
		    u_int iovcnt,
		    off_t offset
		);
	}
```

The core of the declaration declares the arguments and the return value with C function declaration syntax. On the first line we see the number `290`, which is the system call number used in the userspace stub (there disassembled as `0x122`). How should you choose a system-call number? If the system call will be added to FreeBSD directly, you should add it to the

ABI feature	i386	Other 32-bit	64-bit	CheriABI
long size	32-bit	32-bit	64-bit	64-bit
time_t size	32-bit	64-bit	64-bit	64-bit
uint64_t alignment	32-bit	64-bit	64-bit	64-bit
void * alignment	32-bit	32-bit	64-bit	128-bit

TABLE I
KEY DIFFERENCES BETWEEN ABIS IN FREEBSD AND CHERIBSD

end of the list incrementing the maximum system-call number. If it will be used locally, you can use any of the system call entries marked with `RESERVED`. Should you need to add more local system calls than `RESERVED` permits, there may be other `COMPAT` calls you could reuse or you could ask the project to add more reserved entries. If you need to duplicate the whole set of system call for some reason, starting over at `1000` is likely enough.

The `AUE_PWRITEV` entry is the audit type. Audit entries are allocated by the OpenBSM project.[3] For system calls that do not require auditing `AUE_NULL` may be used. The `STD|CAPENABLED` field says that this is a standard (always present) system call and that it is allowed to operate in Capsicum capability mode. It is allowed in capability mode because it uses no global namespaces, only file descriptors.

In addition to ordinary function declaration syntax, we use two types of annotations for pointer arguments. First, `_In_reads_(iovcnt)` indicates that we read `iovcnt` `struct iovec` objects from userspace. This Microsoft SAL annotation describes the 1st-order memory footprint of the system call, and is useful for generating system-call trace frameworks or interposers. Second, `_Contains_long_ptr_` indicates that the objects contains `long` values (e.g., `ssize_t`) and pointers. This allows ABI compatibility descriptions to be generated. The subset of SAL that we use as well as the `_Contains_` values are described in comment at the top of `syscalls.master` along with other values such as alternatives to `STD`.

Once an entry is added, you can run `make sysent` at the top of the source tree (you may need to build world first). This will update generated files. For the default ABI this set of files and their purposes are listed in table II

Compatibility ABIs have a corresponding set of files minus `syscall.mk` since the set of syscalls is the same in all FreeBSD ABIs. This differs from Linux, where each ABI starts with system call numbers aligning with the most popular Unix implementation at the time of the port.

B. Main implementation

Once you've run `make sysent`, you can add a `sys_foo()` implementation. If it's a `STD` syscall, the file must always be compiled in. For optional system calls (e.g., the audit or mac frameworks), current practice is always to add the system-call table entry, but have the implementation return `ENOSYS` when the option isn't enabled. For example, when the kernel is compiled without the `AUDIT` option, the implementation of `auditctl(2)` is:

[3]https://github.com/openbsm/openbsm

```
int
sys_auditctl(struct thread *td,
    struct auditctl_args *uap)
{
        return (ENOSYS);
}
```

Another convention is that most system calls have mostly trivial `sys_foo()` implementation, which calls `kern_foo()` to implement the actual system call. This is useful for compat system calls where a `freebsd32_foo()` calls into the actual implementation.

C. 32-bit compatibility

If `make sysent` modified `freebsd32_proto.h` then the system call needs a compatibility implementation. If you've used `_Contains_` annotations correctly and not used any always-64-bit types other than `dev_t`, `id_t`, or `off_t` then the tool used by `make sysent` (`sys/tools/makesyscalls.lua`) will generate declarations and system-call table entries for `freebsd32_foo()` if it is required.

D. 64-bit compatibility

As with 32-bit compatibility, `make sysent` will generated declarations for `freebsd64_foo()` functions as required. As discussed above, 64-bit compatibility on CheriBSD is similar to 32-bit compatibility except that `long` type are the same size and capabilities need to be derived from integer pointers using `__USER_CAP` macros. While it's possible to just use `__USER_CAP_UNBOUND` to derive a pointer to the whole program address space (as determined by the thread's DDC), it's better practice to use argument information to bound pointers. In some cases this can protect against bugs in the kernel or userspace despite the 64-bit program not using capabilities directly.

E. Userspace bits

For most systems calls, the entries in `syscalls.mk` ensure that appropriate functions are generated in libc. All that is required is to add them to `lib/libc/sys/Symbol.map`. Entries in the `Symbol.map` file should be added to a per-major-release block (for FreeBSD 14 this is `FBSD_1.7`). Only the main system call name should be added not the `_foo` or `__sys_foo` symbol.

Some system calls do require some userspace implementation. For example, the `exit()` function calls a number of cleanup and teardown routines before install calling the `_exit()` system call stub. The default behavior is overridden in `lib/libc/sys/Makefile.inc` by adding `exit.o` to

File	Purpose
sys/kern/init_sysent.c	declares system call table
sys/kern/syscalls.c	number to name translation table
sys/kern/systrace_args.c	tracing
sys/sys/syscall.h	name to number macros (e.g., SYS_pwritev)
sys/sys/syscall.mk	list of object files in MIASM variable
sys/sys/sysproto.h	kernel prototypes (e.g., sys_pwritev(), struct sys_pwritev_args)

TABLE II
GENERATED SYSTEM CALL FILES

`NOASM`, disabling the default stubs and adding `_exit.o` to `PSEUDO` enabling a reduced stub.

In addition to any libc implementation details, a manpage should be added (usually under `lib/libc/sys`) and an entry added to the `MAN2` variable to enable it. Writing manpages is beyond the scope of this paper, but starting with a manpage from a simile system call is usually a good place to start.

IV. Conclusions and guidance for new system calls

Having read this paper, the reader should be ready to add a new system call (subject to understanding the subsystem(s) it interacts with). The process of adding system calls is increasingly standardized with guardrails provided by `gensyscalls.lua`. Before you apply your new knowledge, I encourage you to think long and hard about whether a system call is actually required. Once you've concluded it is, make sure to seek review early and often. System calls are forever, so it's important to make sure they have the right interface before they make it into a release or the critical path of the boot process.

References

[1] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 379–393, New York, NY, USA, 2019. ACM.

[2] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020.

[3] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*, June 2014.