FUSD:

A Linux **F**ramework for **U**ser-**S**pace **D**evices

Jeremy Elson

jelson@circlemud.org
http://www.circlemud.org/jelson/software/fusd

19 August 2003
Documentation for FUSD 1.10

Contents

List of Example Programs

1 Introduction

1.1 What is FUSD?

FUSD (pronounced *fused*) is a Linux framework for proxying device file callbacks into user-space, allowing device files to be implemented by daemons instead of kernel code. Despite being implemented in user-space, FUSD devices can look and act just like any other file under /dev which is implemented by kernel callbacks.

A user-space device driver can do many of the things that kernel drivers can't, such as perform a long-running computation, block while waiting for an event, or read files from the file system. Unlike kernel drivers, a user-space device driver can *use other device drivers*—that is, access the network, talk to a serial port, get interactive input from the user, pop up GUI windows, or read from disks. User-space drivers implemented using FUSD can be much easier to debug; it is impossible for them to crash the machine, are easily traceable using tools such as `gdb`, and can be killed and restarted without rebooting even if they become corrupted. FUSD drivers don't have to be in C—Perl, Python, or any other language that knows how to read from and write to a file descriptor can work with FUSD. User-space drivers can be swapped out, whereas kernel drivers lock physical memory.

Of course, as with almost everything, there are trade-offs. User-space drivers are slower than kernel drivers because they require three times as many system calls, and additional memory copies (see section 10). User-space drivers can not receive interrupts, and do not have the full power to modify arbitrary kernel data structures as kernel drivers do. Despite these limitations, we have found user-space device drivers to be a powerful programming paradigm with a wide variety of uses (see Section 2).

FUSD is free software, distributed under a GPL-compatible license (the "new" BSD license, with the advertising clause removed).

1.2 How does FUSD work?

FUSD drivers are conceptually similar to kernel drivers: a set of callback functions called in response to system calls made on file descriptors by user programs. FUSD's C library provides a device registration function, similar to the kernel's `devfs_register_chrdev()` function, to create new devices. `fusd_register()` accepts the device name and a structure full of pointers. Those pointers are callback functions which are called in response to certain user system calls—for example, when a process tries to open, close, read from, or write to the device file. The callback functions should conform to the standard definitions of POSIX system call behavior. In many ways, the user-space FUSD callback functions are identical to their kernel counterparts.

Perhaps the best way to show what FUSD does is by example. Program 1 is a simple FUSD device driver. When the program is run, a device called `/dev/hello-world` appears under the `/dev` directory. If that device is read (e.g., using `cat`), the read returns `Hello, world!` followed by an EOF. Finally, when the driver is stopped (e.g., by hitting Control-C), the device file disappears.

On line 40 of the source, we use `fusd_register()` to create the `/dev/hello-world` device, passing pointers to callbacks for the open(), close() and read() system calls. (Lines 36–39 use the GNU C extension that allows initializer field naming; the 2.4 series of Linux kernels use also that extension for the same purpose.) The "Hello, World" read() callback itself is virtually identical to what a kernel driver for this device would look like. It can inspect and modify the user's file pointer, copy data into the user-provided buffer, control the system call return value (either positive, EOF, or error), and so forth.

The proxying of kernel system calls that makes this kind of program possible is implemented by FUSD, using a combination of a kernel module and cooperating user-space library. The kernel module implements a character device, `/dev/fusd`, which is used as a control channel between the two. fusd_register() uses this channel to send a message to the FUSD kernel module, telling the name of the device the user wants to register. The kernel module, in turn, registers that device with the kernel proper using devfs. devfs and the kernel don't know anything

Program 1 helloworld.c: A simple program using FUSD to create `/dev/hello-world`

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include "fusd.h"

#define GREETING "Hello, world!\n"

int do_open_or_close(struct fusd_file_info *file)
{
  return 0; /* attempts to open and close this file always succeed */
}

int do_read(struct fusd_file_info *file, char *user_buffer,
            size_t user_length, loff_t *offset)
{
  int retval = 0;

  /* The first read to the device returns a greeting.  The second read
   * returns EOF. */
  if (*offset == 0) {
    if (user_length < strlen(GREETING))
      retval = -EINVAL;  /* the user must supply a big enough buffer! */
    else {
      memcpy(user_buffer, GREETING, strlen(GREETING)); /* greet user */
      retval = strlen(GREETING); /* retval = number of bytes returned */
      *offset += retval; /* advance user's file pointer */
    }
  }

  return retval;
}

int main(int argc, char *argv[])
{
  struct fusd_file_operations fops = {
    open: do_open_or_close,
    read: do_read,
    close: do_open_or_close };

  if (fusd_register("/dev/hello-world", 0666, NULL, &fops) < 0)
    perror("Unable to register device");
  else {
    printf("/dev/hello-world should now exist - calling fusd_run...\n");
    fusd_run();
  }
  return 0;
}
```

unusual is happening; it appears from their point of view that the registered devices are simply being implemented by the FUSD module.

Later, when kernel makes a callback due to a system call (e.g. when the character device file is opened or read), the FUSD kernel module's callback blocks the calling process, marshals the arguments of the callback into a message and sends it to user-space. Once there, the library half of FUSD unmarshals it and calls whatever user-space callback the FUSD driver passed to fusd_register(). When that user-space callback returns a value, the process happens in reverse: the return value and its side-effects are marshaled by the library and sent to the kernel. The FUSD kernel module unmarshals this message, matches it up with a corresponding outstanding request, and completes the system call. The calling process is completely unaware of this trickery; it simply enters the kernel once, blocks, unblocks, and returns from the system call—just as it would for any other blocking call.

One of the primary design goals of FUSD is *stability*. It should not be possible for a FUSD driver to corrupt or crash the kernel, either due to error or malice. Of course, a buggy driver itself may corrupt itself (e.g., due to a buffer overrun). However, strict error checking is implemented at the user-kernel boundary which should prevent drivers from corrupting the kernel or any other user-space process—including the errant driver's own clients, and other FUSD drivers.

1.3 What FUSD *Isn't*

FUSD looks similar to certain other Linux facilities that are already available. It also skirts near a few of the kernel's hot-button political issues. So, to avoid confusion, we present a list of things that FUSD is *not*.

- **A FUSD driver is not a kernel module.** Kernel modules allow—well, modularity of kernel code. They let you insert and remove kernel modules dynamically after the kernel boots. However, once inserted, the kernel modules are actually part of the kernel proper. They run in the kernel's address space, with all the same privileges and restrictions that native kernel code does. A FUSD device driver, in contrast, is more similar to a daemon—a program that runs as a user-space process, with a process ID.
- **FUSD is not, and doesn't replace, devfs.** When a FUSD driver registers a FUSD device, it automatically creates a device file in `/dev`. However, FUSD is not a replacement for devfs—quite the contrary, FUSD creates those device files by *using* devfs. In a normal Linux system, only kernel modules proper—not user-space programs—can register with devfs (see above).
- **FUSD is not UDI.** UDI, the Uniform Driver Interface[1], aims to create a binary API for drivers that is uniform across operating systems. It's true that FUSD could conceivably be used for a similar purpose (inasmuch as it defines a system call messaging structure). However, this was not the goal of FUSD as much as an accidental side effect. We do not advocate publishing drivers in binary-only form, even though FUSD does make this possible in some cases.
- **FUSD is not an attempt to turn Linux into a microkernel.** We aren't trying to port existing drivers into user-space for a variety of reasons (not the least of which is performance). We've used FUSD as a tool to write new drivers that are much easier from user-space than they would be in the kernel; see Section 2 for use cases.

1.4 Related Work

FUSD is a new implementation, but certainly not a new idea—the theory of its operation is the same as any microkernel operating system. A microkernel (roughly speaking) is one that implements only very basic resource

[1] http://www.projectudi.org

protection and message passing in the kernel. Implementation of device drivers, file systems, network stacks, and so forth are relegated to userspace. Patrick Bridges maintains a list of such microkernel operating systems[2].

Also related is the idea of a user-space filesystem, which has been implemented in a number of contexts. Some examples include Klaus Schauser's UFO Project[3] for Solaris, and Jeremy Fitzhardinge's (no longer maintained) UserFS[4] for Linux 1.x. The UFO paper[5] is also notable because it has a good survey of similar projects that integrate user-space code with system calls.

1.5 Limitations and Future Work

In its current form, FUSD is useful and has proven to be quite stable—we use it in production systems. However, it does have some limitations that could benefit from the attention of developers. Contributions to correct any of these deficiencies are welcomed! (Many of these limitations will not make sense without having read the rest of the documentation first.)

- Currently, FUSD only supports implementation of character devices. Block devices and network devices are not supported yet.
- The kernel has 15 different callbacks in its `file_operations` structure. The current version of FUSD does not proxy some of the more obscure ones out to userspace.
- Currently, all system calls that FUSD understands are proxied from the FUSD kernel module to userspace. Only the userspace library knows which callbacks have actually been registered by the FUSD driver. For example, the kernel may proxy a write() system call to user-space even if the driver has not registered a write() callback with fusd_register().

  fusd_register() should, but currently does not, tell the kernel module which callbacks it wants to receive, per-device. This will be more efficient because it will prevent useless system calls for unsupported operations. In addition, it will lead to more logical and consistent behavior by allowing the kernel to use its default implementations of certain functions such as writev(), instead of being fooled into thinking the driver has an implementation of it in cases where it doesn't.
- It should be possible to write a FUSD library in any language that supports reads and writes on raw file descriptors. In the future, it might be possible to write FUSD device drivers in a variety of languages—Perl, Python, maybe even Java. However, the current implementation has only a C library.
- It's possible for drivers that use FUSD to deadlock—for example, if a driver tries to open itself. In this one case, FUSD returns `-EDEADLOCK`. However, deadlock protection should be expanded to more general detection of cycles of arbitrary length.
- FUSD should provide a /proc interface that gives debugging and status information, and allows parameter tuning.
- FUSD was written with efficiency in mind, but a number of important optimizations have not yet been implemented. Specifically, we'd like to try to reduce the number of memory copies by using a buffer shared between user and kernel space to pass messages.
- FUSD currently requires devfs, which is used to dynamically create device files under `/dev` when a FUSD driver registers itself. This is, perhaps, the most convenient and useful paradigm for FUSD. However, some users have asked if it's possible to use FUSD without devfs. This should be possible if FUSD drivers bind to device major numbers instead of device file names.

[2]http://www.cs.arizona.edu/people/bridges/os/microkernel.html
[3]http://www.cs.ucsb.edu/projects/ufo/index.html
[4]http://www.goop.org/ jeremy/userfs/
[5]http://www.cs.ucsb.edu/projects/ufo/97-usenix-ufo.ps

1.6 Author Contact Information and Acknowledgments

The original version of FUSD was written by Jeremy Elson (jelson@circlemud.org) and Lewis Girod at Sensoria Corporation. Sensoria no longer maintains public releases of FUSD, but the same authors have since forked the last public release and continue to maintain FUSD from the University of California, Los Angeles.

If you have bug reports, patches, suggestions, or any other comments, please feel free to contact the authors.

FUSD has two SourceForge[6]-host mailing lists: a low-traffic list for announcements (`fusd-announce`) and a list for general discussion (`fusd-devel`). Subscription information for both lists is available at the SourceForge's FUSD mailing list page.

For the latest releases and information about FUSD, please see the official FUSD home page[7].

1.7 Licensing Information

FUSD is free software, distributed under a GPL-compatible license (the "new" BSD license, with the advertising clause removed). The license is enumerated in its entirety below.

2 Why use FUSD?

One basic question about FUSD that one might ask is: what is it good for? Why use it? In this section, we describe some of the situations in which FUSD has been the solution for us.

[6] http://www.sourceforge.net

[7] http://www.circlemud.org/jelson/software/fusd

2.1 Device Driver Layering

A problem that comes up frequently in modern operating systems is contention for a single resource by multiple competing processes. In UNIX, it's the job of a device driver to coordinate access to such resources. By accepting requests from user processes and (for example) queuing and serializing them, it becomes safe for processes that know nothing about each other to make requests in parallel to the same resource. Of course, kernel drivers do this job already, but they typically operate on top of hardware directly. However, kernel drivers can't easily be layered on top of *other device drivers*.

For example, consider a device such as a modem that is connected to a host via a serial port. Let's say we want to implement a device driver that allows multiple users to dial the telephone (e.g., `echo 1-310-555-1212 > /dev/phone-dialer`). Such a driver should be layered *on top of* the serial port driver—that is, it most likely wants to write to `/dev/ttyS0`, not directly to the UART hardware itself.

While it is possible to write to a logical file from within a kernel device driver, it is both tricky and considered bad practice. In the words of kernel hacker Dick Johnson[8], "You should never write a [kernel] module that requires reading or writing to any logical device. The kernel is the thing that translates physical I/O to logical I/O. Attempting to perform logical I/O in the kernel is effectively going backwards."

With FUSD, it's possible to layer device drivers because the driver is a user-space process, not a kernel module. A FUSD implementation of our hypothetical `/dev/phone-dialer` can open `/dev/ttyS0` just as any other process would.

Typically, such layering is accomplished by system daemons. For example, the `lpd` daemon manages printers at a high level. Since it is a user-space process, it can access the physical printer devices using kernel device drivers (for example, using printer or network drivers). There a number of advantages to using FUSD instead:

- Using FUSD, a daemon/driver can create a standard device file which is accessible by any program that knows how to use the POSIX system call interface. Some trickery is possible using named pipes and FIFOs, but quickly becomes difficult because of multiplexed writes from multiple processes.
- FUSD drivers receive the UID, GID, and process ID along with every file operation, allowing the same sorts of security policies to be implemented as would be possible with a real kernel driver. In contrast, writes to a named pipe, UDP, and so forth are "anonymous."

2.2 Use of User-Space Libraries

Since a FUSD driver is just a regular user-space program, it can naturally use any of the enormous body of existing libraries that exist for almost any task. FUSD drivers can easily incorporate user interfaces, encryption, network protocols, threads, and almost anything else. In contrast, porting arbitrary C code into the kernel is difficult and usually a bad idea.

2.3 Driver Memory Protection

Since FUSD drivers run in their own process space, the rest of the system is protected from them. A buggy or malicious FUSD driver, at the very worst, can only corrupt itself. It's not possible for it to corrupt the kernel, other FUSD drivers, or even the processes that are using its devices. In contrast, a buggy kernel module can bring down any process in the system, or the entire kernel itself.

[8] http://www.uwsg.indiana.edu/hypermail/linux/kernel/0005.3/0061.html

2.4 Giving libraries language independence and standard notification interfaces

One particularly interesting application of FUSD that we've found very useful is as a way to let regular user-space libraries export device file APIs. For example, imagine you had a library which factored large composite numbers. Typically, it might have a C interface—say, a function called `int *factorize(int bignum)`. With FUSD, it's possible to create a device file interface—say, a device called `/dev/factorize` to which clients can `write(2)` a big number, then `read(2)` back its factors.

This may sound strange, but device file APIs have at least three advantages over a typical library API. First, it becomes much more language independent—any language that can make system calls can access the factorization library. Second, the factorization code is running in a different address space; if it crashes, it won't crash or corrupt the caller. Third, and most interestingly, it is possible to use `select(2)` to wait for the factorization to complete. `select(2)` would make it easy for a client to factor a large number while remaining responsive to *other* events that might happen in the meantime. In other words, FUSD allows normal user-space libraries to integrate seamlessly with UNIX's existing, POSIX-standard event notification interface: `select(2)`.

2.5 Development and Debugging Convenience

FUSD processes can be developed and debugged with all the normal user-space tools. Buggy drivers won't crash the system, but instead dump cores that can be analyzed. All of your favorite visual debuggers, memory bounds checkers, leak detectors, profilers, and other tools can be applied to FUSD drivers as they would to any other program.

3 Installing FUSD

This section describes the installation procedure for FUSD. It assumes a good working knowledge of Linux system administration.

3.1 Prerequisites

Before installing FUSD, make sure you have all of the following packages installed and working correctly:

- **Linux kernel 2.4.0 or later**. FUSD was developed under 2.4.0 and should work with any kernel in the 2.4 series.
- **devfs installed and running.** FUSD dynamically registers devices using devfs, the Linux device filesystem by Richard Gooch. For FUSD to work, devfs must be installed and running on your system. For more information about devfs installation, see the devfs home page[9].

  Note that some distributions make installation devfs easier. RedHat 7.1, for example, already has all of the necessary daemons and configuration changes integrated. devfs can be installed simply by recompiling the kernel with devfs support enabled and reconfiguring LILO to pass `"devfs=mount"` to the kernel.

3.2 Compiling FUSD as a Kernel Module

Before compiling anything, take a look at the Makefile in FUSD's home directory. Adjust any constants that are not correct. In particular, make sure `KERNEL_HOME` correctly reflects the place where your kernel sources are installed, if they aren't in the default location of `/usr/src/linux`.

[9] http://www.atnf.csiro.au/ rgooch/linux/docs/devfs.html

Then, type `make`. It should generate a directory whose name looks something like `obj.i686-linux`, or some variation depending on your architecture. Inside of that directory will be a number of files, including:

- kfusd.o – The FUSD kernel module
- libfusd.a – The C library used to talk to the kernel module
- Example programs – linked against libfusd.a

Compilation of the kernel module will fail if the dependencies described in the previous section are not satisfied. The module must be compiled again Linux kernel must be v2.4.0 or later, and the kernel must have devfs support enabled.

3.3 Testing and Troubleshooting

Once everything has been compiled, give it a try to see if it actually does something. First, use `insmod` to insert the FUSD kernel module, e.g. `insmod obj.i686-linux/kfusd.o`. A greeting message similar to "`fusd: starting, Revision: 1.50`" should appear in the kernel log (accessed using the `dmesg` command, or by typing `cat /proc/kmsg`). You can verify the module has been inserted by typing `lsmod`, or alternatively `cat /proc/modules`.

Once the module has been inserted successfully, trying running the `helloworld` example program. When run, the program should print a greeting message similar to `/dev/hello-world should now exist - calling fusd_run`. This means everything is working; the daemon is now blocked, waiting for requests to the new device. From another shell, type `cat /dev/hello-world`. You should see `Hello, world!` printed in response. Try killing the test program; the corresponding device file should disappear.

If nothing seems to be working, try looking at the kernel message log (type `dmesg` or `cat /proc/kmsg`) to see if there are any errors. If nothing seems obviously wrong, try turning on FUSD kernel module debugging by defining `CONFIG_FUSD_DEBUG` in kfusd.c, then recompiling and reinserting the module.

3.4 Installation

Typing `make install` will copy the FUSD library, header files, and man pages into `/usr/local`. The FUSD kernel module is *not* installed automatically because of variations among different Linux distributions in how this is accomplished. You may want to arrange to have the module start automatically on boot by (for example) copying it into `/lib/modules/your-kernel-version`, and adding it to `/etc/modules.conf`.

3.5 Making FUSD Part of the Kernel Proper

The earlier instructions, by default, create a FUSD kernel module. If desired, it's also very easy to build FUSD right into the kernel, instead:

1. Unpack the 2.4 kernel sources and copy all the files in the `include` and `kfusd` directories into your kernel source tree, under `drivers/char`. For example, if FUSD is in your home directory, and your kernel is in `/usr/src/linux`:

```
cp ~/fusd/kfusd/* ~/fusd/include/* /usr/src/linux/drivers/char
```

2. Apply the patch found in FUSD's `patches` directory to your kernel source tree. For example:

```
cd /usr/src/linux
patch -p0 < ~/fusd/patches/fusd-inkernel.patch
```

The FUSD in-kernel patch doesn't actually change any kernel sources proper; it just adds FUSD to the kernel configuration menu and Makefile.

3. Using your kernel configurator of choice (e.g. `make menuconfig`), turn on the FUSD options. It will be under the "Character devices" menu.

4. Build and install the kernel as usual.

4 Basic Device Creation

Enough introduction—it's time to actually create a basic device driver using FUSD!

This following sections will illustrate various techniques using example programs. To save space, interesting excerpts are shown instead of entire programs. However, the `examples` directory of the FUSD distribution contains all the examples in their entirety. They can actually be compiled and run on a system with the FUSD kernel module installed.

Where this text refers to example program line numbers, it refers to the line numbers printed alongside the excerpts in the manual—not the line numbers of the actual programs in the `examples` directory.

4.1 Using `fusd_register` to create a new device

We saw an example of a simple driver, helloworld.c, in Program 1 on page 2. Let's go back and examine that program now in more detail.

The FUSD ball starts rolling when the `fusd_register` function is called, as shown on line 40. This function tells the FUSD kernel module:

- `char *name`—The name of the device being created. The prefix (such as `/dev/`) must match the location where devfs has been mounted. Names containing slashes (e.g., `/dev/my-devices/dev1`) are legal; devfs creates subdirectories automatically.
- `mode_t mode`—The device's default permissions. This is usually specified using an octal constant with a leading 0—`0666` (readable and writable by everyone) instead of the incorrect decimal constant `666`.
- `void *device_info`—Private data that should be passed to callback functions for this device. The use of this field is described in Section 5.1.
- `struct fusd_file_operations *fops`—A structure containing pointers to the callback functions that should be called by FUSD in response to certain events.

If device registration is successful, `fusd_register` returns a *device handle*—a small integer ≥ 0. On errors, it returns -1 and sets the global variable `errno` appropriately. In reality, the device handle you get is a plain old file descriptor, as we'll see in Section 7.

Although Program 1 only calls `fusd_register` once, it can be called multiple times if the FUSD driver is handling more than one device as we'll see in Program 4.

There is intentional similarity between `fusd_register()` and the kernel's device registration functions, such as `devfs_register()` and `register_chrdev()`. In many ways, FUSD's interface is meant to mirror the kernel interface as closely as possible.

The `fusd_file_operations` structure, defined in `fusd.h`, contains a list of callbacks that are used in response to different system calls executed on a file. It is similar to the kernel's `file_operations` structure, accepting callbacks for system calls such as `open()`, `close()`, `read()`, `write()`, and `ioctl()`. For the most part, the prototypes of FUSD file operation callbacks are the same as their kernel cousins, with one important exception. The first argument of FUSD callbacks is always a pointer to a `fusd_file_info` structure; it contains information that can be used to identify the file. This structure is used instead of the kernel's `file` and `inode` structures, and will be described in more detail later.

In lines 35–38 of Program 1, we create and initialize a `fusd_file_operations` structure. A GCC-specific C extension allows us to name structure fields explicitly in the initializer. This style may look strange, but it guards against errors in the future in case the order of fields in the structure ever changes. The 2.4 kernel series uses the same trick.

After calling `fusd_register()` on line 40, the example program calls `fusd_run()` on line 44. This function turns control over to the FUSD framework. fusd_run blocks the driver until one of the devices it registered needs to be serviced. Then, it calls the appropriate callback and blocks again until the next event.

Now, imagine that a user types `cat /dev/hello-world`. What happens? Recall first what the `cat` program itself does: opens a file, reads from it until it receives an EOF (printing whatever it reads to stdout), then closes it. `cat` works the same way regardless of what it's reading—be it a a FUSD device, a regular file, a serial port, or anything else. The `strace` program is a great way to see this in action; see Appendix A for details.

4.2 The `open` and `close` callbacks

The first two callbacks that most drivers typically implement are `open` and `close`. Each of these two functions are passed just one argument—the `fusd_file_info` structure that describes the instance of the file being opened or closed. Use of the information in that structure will be covered in more detail in Section 5.

The semantics of an `open` callback's return value are exactly the same as inside the kernel:

- 0 means success, and the file is opened. If the file is allowed to open, the kernel returns a valid file descriptor to the client. Using that descriptor, other callbacks may be called for that file, including (at least) a `close` callback.
- A negative number indicates a failure, and that the file should not be opened. Such return values should *always* be the specified as a negative `errno` value such as `-EPERM`, `-EBUSY`, `-ENODEV`, `-ENOMEM`, and so on. For example, if the callback returns `-EPERM`, the caller's `open()` will return -1, with `errno` set to `EPERM`. A complete list of possible return values can be found in the Linux kernel sources, under `include/asm/errno.h`.

If an `open` callback returns 0 (success), a driver is *guaranteed* to receive exactly one `close` callback for that file later. By the same token, the close callback *will not* be called if the open fails. Therefore, `open` callbacks that can return failure must be sure to deallocate any resources they might have allocated before returning a failure.

Let's return to our example in Program 1, which creates the `/dev/hello-world` device. If a user types `cat /dev/hello-world`, `cat` will will use the `open(2)` system call to open the file. FUSD will then proxy that system call to the driver and activate the callback that was registered as the `open` callback. Recall from line 36 of Program 1 that we registered `do_open_or_close`, which appears on line 8.

In `helloworld.c`, the `open` callback always returns 0, or success. However, in a real driver, something more interesting will probably happen—permissions checks, memory allocation for state-keeping, and so forth. The corresponding *de*-allocation of those resources should occur in the `close` callback, which is called when a user application calls `close` on their file descriptor. `close` callbacks are allowed to return error values, but this does not prevent the file from actually closing.

4.3 The `read` callback

Returning to our `cat /dev/hello-world` example, what happens after the `open` is successful? Next, `cat` will try to use `read(2)`, which will get proxied by FUSD to the function `do_read` on line 13. This function takes some additional arguments that we didn't see in the open and close callbacks:

- `struct fusd_file_info *file`—The first argument to all callbacks, containing information which describes the file; see Section 5.
- `char *user_buffer`—The buffer that the callback should use to write data that it is returning to the user.
- `size_t user_length`—The maximum number of bytes requested by the user. The driver is allowed to return fewer bytes, but should never write more then `user_length` bytes into `user_buffer`.
- `loff_t *offset`—A pointer to an integer which represents the caller's offset into the file (i.e., the user's file pointer). This value can be modified by the callback; any change will be propagated back to the user's file pointer inside the kernel.

The semantics of the return value are the same as if the callback were being written inside the kernel itself:

- Positive return values indicate success. If the call is successful, and the driver has copied data into `buffer`, the return value indicates how many bytes were copied. This number should never be greater than the `user_length` argument.
- A 0 return value indicates EOF has been reached on the file.
- As in the `open` and `close` callbacks, negative values (such as -EPERM, -EPIPE, or -ENOMEM) indicate errors. Such values will cause the user's `read()` to return -1 with errno set appropriately.

The first time a read is done on a device file, the user's file pointer (`*offset`) is 0. In the case of this first read, a greeting message of `Hello, world!` is copied back to the user, as seen on line 24. The user's file pointer is then advanced. The next read therefore fails the comparison at line 20, falling straight through to return 0, or EOF.

In this simple program, we also see an example of an error return on line 22: if the user tries to do a read smaller than the length of the greeting message, the read will fail with -EINVAL. (In an actual driver, it would normally not be an error for a user to provide a smaller read buffer than the size of the available data. The right way for drivers to handle this situation is to return partial data, then move `*offset` forward so that the remainder is returned on the next `read()`. We see an example of this in Program 2.)

4.4 The `write` callback

Program 1 illustrated how a driver could return data *to* a client using the `read` callback. As you might expect, there is a corresponding `write` callback that allows the driver to receive data *from* a client. `write` takes four arguments, similar to the `read` callback:

- `struct fusd_file_info *file`—The first argument to all callbacks, containing information which describes the file; see Section 5.
- `const char *user_buffer`—Pointer to data being written by the client (read-only).
- `size_t user_length`—The number of bytes pointed to by `user_buffer`.

- `loff_t *offset`—A pointer to an integer which represents the caller's offset into the file (i.e., the user's file pointer). This value can be modified by the callback; any change will be propagated back to the user's file pointer inside the kernel.

The semantics of `write`'s return value are the same as in a kernel callback:

- Positive return values indicate success and indicate how many bytes of the user's buffer were successfully written (i.e., successfully processed by the driver in some way). The return value may be less than or equal to the `user_length` argument, but should never be greater.
- 0 should only be returned in response to a `write` of length 0.
- Negative values (such as -EPERM, -EPIPE, or -ENOMEM) indicate errors. Such values will cause the user's `write()` to return -1 with errno set appropriately.

Program 2, echo.c, is an example implementation of a device (`/dev/echo`) that uses both `read()` and `write()` callbacks. A client that tries to `read()` from this device will get the contents of the most recent `write()`. For example:

```
% echo Hello there > /dev/echo
% cat /dev/echo
Hello there
% echo Device drivers are fun > /dev/echo
% cat /dev/echo
Device drivers are fun
```

The implementation of `/dev/echo` keeps a global variable, `data`, which serves as a cache for the data most recently written to the driver by a client program. The driver does not assume the data is null-terminated, so it also keeps track of the number of bytes of data available. (These two variables appear on lines 1–2.)

The driver's `write` callback first frees any data which might have been allocated by a previous call to write (lines 26–29). Next, on line 33, it attempts to allocate new memory for the new data arriving. If the allocation fails, `-ENOMEM` is returned to the client. If the allocation is successful, the driver copies the data into its local buffer and stores its length (lines 37–38). Finally, the driver tells the user that the entire buffer was consumed by returning a value equal to the number of bytes the user tried to write (`user_length`).

The `read` callback has some extra features that we did not see in Program 1's `read()` callback. The most important is that it allows the driver to read the available data *incrementally*, instead of requiring that the first `read()` executed by the client has enough space for all the data the driver has available. In other words, a client can do two 50-byte reads, and expect the same effect as if it had done a single 100-byte read.

This is implemented using `*offset`, the user's file pointer. If the user is trying to read past the amount of data we have available, the driver returns EOF (lines 8–9). Normally, this happens after the client has finished reading data. However, in this driver, it might happen on a client's first read if nothing has been written to the driver yet or if the most recent write's memory allocation failed.

If there is data to return, the driver computes the number of bytes that should be copied back to the client—the minimum of the number of bytes the user asked for, and the number of bytes of data that this client hasn't seen yet (line 12). This data is copied back to the user's buffer (line 15), and the user's file pointer is advanced accordingly (line 16). Finally, on line 19, the client is told how many bytes were copied to its buffer.

Program 2 echo.c: Using both `read` and `write` callbacks

```
char *data = NULL;
int data_length = 0;

int echo_read(struct fusd_file_info *file, char *user_buffer,
              size_t user_length, loff_t *offset)
{
  /* if the user has read past the end of the data, return EOF */
  if (*offset >= data_length)
    return 0;

  /* only return as much data as we have */
  user_length = MIN(user_length, data_length - *offset);

  /* copy data to user starting from the first byte they haven't seen */
  memcpy(user_buffer, data + *offset, user_length);
  *offset += user_length;

  /* tell them how much data they got */
  return user_length;
}

ssize_t echo_write(struct fusd_file_info *file, const char *user_buffer,
                   size_t user_length, loff_t *offset)
{
  /* free the old data, if any */
  if (data != NULL) {
    free(data);
    data = NULL;
    data_length = 0;
  }

  /* allocate space for new data; return error if that fails */
  if ((data = malloc(user_length)) == NULL)
    return -ENOMEM;

  /* make a copy of user's data; tell the user we copied everything */
  memcpy(data, user_buffer, user_length);
  data_length = user_length;
  return user_length;
}
```

4.5 Unregistering a device with `fusd_unregister()`

All devices registered by a driver are unregistered automatically when the program exits (or crashes). However, the `fusd_unregister()` function can be used to unregister a device without terminating the entire driver. `fusd_unregister` takes one argument: a device handle (i.e., the return value from `fusd_register()`).

A device can be unregistered at any time. Any client system calls that are pending when a device is unregistered will return immediately with an error. In this case, `errno` will be set to `-EPIPE`.

5 Using Information in `fusd_file_info`

We mentioned in the previous sections that the first argument to every callback is a pointer to a `fusd_file_info` structure. This structure contains information that can be useful to driver implementers in deciding how to respond to a system call request.

The fields of `fusd_file_info` structures fall into several categories:

- *Read-only.* The driver can inspect the value, but changing it will have no effect.
  - `pid_t pid`: The process ID of the process making the request
  - `uid_t uid`: The user ID of the owner of the process making the request
  - `gid_t gid`: The group ID of the owner of the process making the request
- *Read-write.* Any changes to the value will be propagated back to the kernel and be written to the appropriate in-kernel structure.
  - `unsigned int flags`: A copy of the `f_flags` field in the kernel's `file` structure. The flags are an or'd-together set of the kernel's `O_` series of flags: `O_NONBLOCK`, `O_APPEND`, `O_SYNC`, etc.
  - `void *device_info`: The data passed to `fusd_register` when the device was registered; see Section 5.1 for details
  - `void *private_data`: A generic per-file-descriptor pointer usable by the driver for its own purposes, such as to keep state (or a pointer to state) that should be maintained between operations on the same instance of an open file. It is guaranteed to be NULL when the file is first opened. See Section 5.2 for more details.
- *Hidden fields.* The driver should not touch these fields (such as `fd`). They contain state used by the FUSD library to generate the reply sent to the kernel.

Important note: the value of the `fusd_file_info` pointer itself has *no meaning*. Repeated requests on the same file descriptor *will not* generate callbacks with identical `fusd_file_info` pointer values, as would be the case with an in-kernel driver. In other words, if a driver needs to keep state in between successive system calls on a user's file descriptor, it *must* store that state using the `private_data` field. The `fusd_file_info` pointer itself is ephemeral; the data to which it points is persistent.

Program 3 shows an example of how a driver might make use of the data in the `fusd_file_info` structure. Much of the driver is identical to helloworld.c. However, instead of printing a static greeting, this new program generates a custom message each time the device file is read, as seen on line 25. The message contains the PID of the user process that requested the read (`file->pid`).

In addition, Program 3's `open` callback does not return 0 (success) unconditionally as it did in Program 1. Instead, it checks (on line 7) to make sure the UID of the process trying to read from the device (`file->uid`) matches the UID under which the driver itself is running (`getuid()`). If they don't match, -EPERM is returned. In other words, only the user who ran the driver is allowed to read from the device that it creates. If any other user—including root!—tries to open it, a "Permission denied" error will be generated.

5.1 Registration of Multiple Devices, and Passing Data to Callbacks

Device drivers frequently expose several different "flavors" of a device. For example, a single magnetic tape drive will often have many different device files in `/dev`. Each device file represents a different combination of options such as rewind/no-rewind, or compressed/uncompressed. However, they access the same physical tape drive.

Program 3 uid-filter.c: Inspecting data in `fusd_file_info` such as UID and PID of the calling process

```
int do_open(struct fusd_file_info *file)
{
  /* If the UID of the process trying to do the read doesn't match the
   * UID of the owner of the driver, return -EPERM.  If you run this
   * driver as a normal user, even root won't be able to read from the
   * device file created! */
  if (file->uid != getuid())
    return -EPERM;

  return 0;
}

int do_read(struct fusd_file_info *file, char *user_buffer,
            size_t user_length, loff_t *offset)
{
  char buf[128];
  int len;

  /* The first read to the device returns a greeting.  The second read
   * returns EOF. */
  if (*offset != 0)
    return 0;

  /* len gets set to the number of characters written to buf */
  len = sprintf(buf, "Your PID is %d.  Have a nice day.\n", file->pid);

  /* NEVER return more data than the user asked for */
  if (user_length < len)
    len = user_length;

  memcpy(user_buffer, buf, len);
  *offset += len;
  return len;
}
```

Traditionally, the device file's *minor number* was used to communicate the desired options with device drivers. But, since devfs dynamically (and unpredictably) generates both major and minor numbers every time a device is registered, a different technique was developed. When using devfs, drivers are allowed to associate a value (of type `void *`) with each device they register. This facility takes the place of the minor number.

The devfs solution is also used by FUSD. The mysterious third argument to `fusd_register` that we mentioned in Section 4.1 is an arbitrary piece of data that can be passed to FUSD when a device is registered. Later, when a callback is activated, the contents of that argument are available in the `device_info` member of the `fusd_file_info` structure.

Program 4 shows an example of this technique, inspired by Alessandro Rubini's similar devfs tutorial published in Linux Magazine[10]. It creates a number of devices in the `/dev/drums` directory, each of which is useful for generating a different kind of "sound"—`/dev/drums/bam`, `/dev/drums/boom`, and so on. Reading from

[10] http://www.linux.it/kerneldocs/devfs/

Program 4 drums.c: Passing private data to `fusd_register` and retrieving it from `device_info`

```
1    static char *drums_strings[] = {"bam", "bum", "beat", "boom",
                                     "bang", "crash", NULL};

     int drums_read(struct fusd_file_info *file, char *user_buffer,
5                   size_t user_length, loff_t *offset)
     {
       int len;
       char sound[128];

10     /* file->device_info is what we passed to fusd_register when we
        * registered the device */
       strcpy(sound, (char *) file->device_info);
       strcat(sound, "\n");

15     /* 1st read returns the sound; 2nd returns EOF */
       if (*offset != 0)
         return 0;

       /* NEVER return more data than the user asked for */
20     len = MIN(user_length, strlen(sound));
       memcpy(user_buffer, sound, len);
       *offset += len;
       return len;
     }
25
     int main(int argc, char *argv[])
     {
       int i;
       char buf[128];
30
       for (i = 0; drums_strings[i] != NULL; i++) {
         sprintf(buf, "/dev/drums/%s", drums_strings[i]);
         if (fusd_register(buf, 0666, drums_strings[i], &drums_fops) < 0)
           fprintf(stderr, "%s register failed: %m\n", drums_strings[i]);
35     }

       fprintf(stderr, "calling fusd_run...\n");
       fusd_run();
       return 0;
40   }
```

any of these devices will return a string equal to the device's name.

The first thing to notice about `drums.c` is that it registers more than one FUSD device. In the loop starting in line 31, it calls `fusd_register()` once for every device named in `drums_strings` on line 1. When `fusd_run()` is called, it automatically watches every device the driver registered, and activates the callbacks associated with each device as needed. Although `drums.c` uses the same set of callbacks for every device it registers (as can be seen on line 33), each device could have different callbacks if desired. (Not shown is the initialization of `drums_fops`, which assigns `drums_read` to be the `read` callback.)

If `drums_read` is called for all 6 types of drums, how does it know which device it's supposed to be servicing when it gets called? The answer is in the third argument of `fusd_register()`, which we were previously ignoring. Whatever value is passed to `fusd_register()` will be passed back to the callback in the `device_info` field of the `fusd_file_info` structure. The name of the drum sound is passed to `fusd_register` on line 33, and later retrieved by the driver on line 12.

Although this example uses a string as its `device_info`, the pointer can be used for anything—a mode number, a pointer to a configuration structure, and so on.

5.2 The difference between `device_info` and `private_data`

As we mentioned in Section 5, the `fusd_file_info` structure has two seemingly similar fields, both of which can be used by drivers to store their own data: `device_info` and `private_data`. However, there is an important difference between them:

- `private_data` is stored *per file descriptor*. If 20 processes open a FUSD device (or, one process opens a FUSD device 20 times), each of those 20 file descriptors will have their own copy of `private_data` associated with them. This field is therefore useful to drivers that need to differentiate multiple requests to a single device that might be serviced in parallel. (Note that most UNIX variants, including Linux, do allow multiple processes to share a single file descriptor—specifically, if a process `open`s a file, then `fork`s. In this case, processes will also share a single copy of `private_data`.)

  The first time a FUSD driver sees `private_data` (in the `open` callback), it is guaranteed to be NULL. Any changes to it by a driver callback will only affect the state associated with that single file descriptor.

- `device_info` is kept *per device*. That is, *all* clients of a device share a *single* copy of `device_info`. Unlike `private_data`, which is always initialized to NULL, `device_info` is always initialized to whatever value the driver passed to `fusd_register` as described in the previous section. If a callback changes the copy of `device_info` in the `fusd_file_info` structure, this has no effect; `device_info` can only be set at registration time, with `fusd_register`.

In short, `device_info` is used to differentiate *devices*. `private_data` is used to differentiate *users of those devices*.

Program 5, drums2.c, illustrates the difference between `device_info` and `private_data`. Like the original drums.c, it creates a bunch of devices in `/dev/drums/`, each of which "plays" a different sound. However, it also does something new: keeps track of how many times each device has been opened. Every read to any drum gives you the name of its sound as well as your unique "user number". And, instead of returning just a single line (as drums.c did), it will keep generating more "sound" every time a `read()` system call arrives.

The trick is that we want to keep users separate from each other. For example, user one might type:

```
% more /dev/drums/bam
You are user 1 to hear a drum go 'bam'!
You are user 1 to hear a drum go 'bam'!
You are user 1 to hear a drum go 'bam'!
...
```

Meanwhile, another user in a different shell might type the same command at the same time, and get different results:

Program 5 drums2.c: Using both `device_info` and `private_data`

```
struct drum_info {
  char *name;
  int num_users;
} drums[] = {
  { "bam", 0 },
  { "bum", 0 },
  /* ... */
  { NULL, 0 }
};

int drums_open(struct fusd_file_info *file)
{
  /* file->device_info is what we passed to fusd_register when we
   * registered the device.  It's a pointer into the "drums" struct. */
  struct drum_info *d = (struct drum_info *) file->device_info;

  /* Store this user's unique user number in their private_data */
  file->private_data = (void *) ++(d->num_users);

  return 0; /* return success */
}

int drums_read(struct fusd_file_info *file, char *user_buffer,
               size_t user_length, loff_t *offset)
{
  struct drum_info *d = (struct drum_info *) file->device_info;
  int len;
  char sound[128];

  sprintf(sound, "You are user %d to hear a drum go '%s'!\n",
          (int) file->private_data, d->name);

  len = MIN(user_length, strlen(sound));
  memcpy(user_buffer, sound, len);
  return len;
}

int main(int argc, char *argv[])
{
  struct drum_info *d;
  char buf[128];

  for (d = drums; d->name != NULL; d++) {
    sprintf(buf, "/dev/drums/%s", d->name);
    if (fusd_register(buf, 0666, d, &drums_fops) < 0)
      fprintf(stderr, "%s register failed: %m\n", d->name);
  }
  /* ... */
```

```
% more /dev/drums/bam
You are user 2 to hear a drum go 'bam'!
You are user 2 to hear a drum go 'bam'!
You are user 2 to hear a drum go 'bam'!
...
```

The idea is that no matter how long those two users go on reading their devices, the driver always generates a message that is specific to that user. The two users' data are not intermingled.

To implement this, Program 5 introduces a new `drum_info` structure (lines 1-4), which keeps track of both the drum's name, and the number of time each drum device has been opened. An instance of this structure, `drums`, is initialized on lines 4-8. Note that the call to `fusd_register` (line 45) now passes a pointer to a `drum_info` structure. (This `drum_info *` pointer is shared by every instance of a client that opens a particular type of drum.)

Each time a drum device is opened, its `drum_info` structure is retrieved from `device_info` (line 15). Then, on line 18, the `num_users` field is incremented and the new user number is stored in `fusd_file_info`'s `private_data` field. To reiterate our earlier point: `device_info` *contains information global to all users of a device, while* `private_data` *has information specific to a particular user of the device.*

It's also worthwhile to note that when we increment `num_users` on line 18, a simple `num_users++` is correct. If this was a driver inside the kernel, we'd have to use something like `atomic_inc()` because a plain `i++` is not atomic. Such a non-atomic statement will result in a race condition on SMP platforms, if an interrupt handler also touches `num_users`, or in some future Linux kernel that is preemptive. Since this FUSD driver is just a plain, single-threaded user-space application, good old ++ still works.

6 Writing `ioctl` Callbacks

The POSIX API provides for a function called `ioctl`, which allows "out-of-band" configuration information to be passed to a device driver through a file descriptor. Using FUSD, you can write a device driver with a callback to handle `ioctl` requests from clients. For the most part, it's just like writing a callback for `read` or `write`, as we've seen in previous sections. From the client's point of view, `ioctl` traditionally takes three arguments: a file descriptor, a command number, and a pointer to any additional data that might be required for the command.

6.1 Using macros to generate `ioctl` command numbers

The Linux header file `/usr/include/asm/ioctl.h` defines macros that *must* be used to create the `ioctl` command number. These macros take various combinations of three arguments:

- `type`—an 8-bit integer selected to be specific to the device driver. `type` should be chosen so as not to conflict with other drivers that might be "listening" to the same file descriptor. (Inside the kernel, for example, the TCP and IP stacks use distinct numbers since an `ioctl` sent to a socket file descriptor might be examined by both stacks.)
- `number`—an 8-bit integer "command number." Within a driver, distinct numbers should be chosen for each different kind of `ioctl` command that the driver services.
- `data_type`—The name of a type used to compute how many bytes are exchanged between the client and the driver. This argument is, for example, the name of a structure.

Program 6 ioctl.h: Using the `_IO` macros to generate `ioctl` command numbers

```
/* definition of the structure exchanged between client and server */
struct ioctl_data_t {
  char string1[60];
  char string2[60];
};

#define IOCTL_APP_TYPE 71 /* arbitrary number unique to this app */

#define IOCTL_TEST2 _IO(IOCTL_APP_TYPE, 2) /* int argument */
#define IOCTL_TEST3 _IOR(IOCTL_APP_TYPE, 3, struct ioctl_data_t)
#define IOCTL_TEST4 _IOW(IOCTL_APP_TYPE, 4, struct ioctl_data_t)
#define IOCTL_TEST5 _IOWR(IOCTL_APP_TYPE, 5, struct ioctl_data_t)
```

The macros used to generate command numbers are:

- `_IO(int type, int number)` – used for a simple ioctl that sends nothing but the type and number, and receives back nothing but an (integer) retval.
- `_IOR(int type, int number, data_type)` – used for an ioctl that reads data *from* the device driver. The driver will be allowed to return `sizeof(data_type)` bytes to the user.
- `_IOW(int type, int number, data_type)` – similar to _IOR, but used to write data *to* the driver.
- `_IORW(int type, int number, data_type)` – a combination of `_IOR` and `_IOW`. That is, data is both written to the driver and then read back from the driver by the client.

Program 6 is an example header file showing the use of these macros. In real programs, the client executing an ioctl and the driver that services it must share the same header file.

6.2 Example client calls and driver callbacks

Program 7 shows a client program that executes `ioctls` using the ioctl command numbers defined in Program 6. The `ioctl_data_t` is application-specific; our simple test program defines it as a structure containing two arrays of characters. The first `ioctl` call (line 10) sends the command `IOCTL_TEST3`, which retrieves strings *from* the driver. The second `ioctl` uses the command `IOCTL_TEST4` (line 18), which sends strings *to* the driver.

The portion of the FUSD driver that services these calls is shown in Program 8.

The ioctl example header file and test programs shown in this document (Programs 6, 7, and 8) are actually contained in a larger, single example program included in the FUSD distribution called `ioctl.c`. That source code shows other variations on calling and servicing `ioctl` commands.

7 Integrating FUSD With Your Application Using `fusd_dispatch()`

The example applications we've seen so far have something in common: after initialization and device registration, they call `fusd_run()`. This gives up control of the program's flow, turning it over to the FUSD library instead. This worked fine for our simple example programs, but doesn't work in a real program that needs to wait for events other than FUSD callbacks. For this reason, our framework provides another way to activate callbacks that does not require the driver to give up control of its `main()`.

Program 7 ioctl-client.c: A program that makes `ioctl` requests on a file descriptor

```
    int fd, ret;
    struct ioctl_data_t d;

    if ((fd = open("/dev/ioctltest", O_RDWR)) < 0) {
      perror("client: can't open ioctltest");
      exit(1);
    }

    /* test3: make sure we can get data FROM a driver using ioctl */
    ret = ioctl(fd, IOCTL_TEST3, &d);
    printf("ioctl test3: got retval=%d\n", ret);
    printf("ioctl test3: got string1='%s'\n", d.string1);
    printf("ioctl test3: got string2='%s'\n", d.string2);

    /* test4: make sure we can send data TO a driver using an ioctl */
    sprintf(d.string1, TEST4_STRING1);
    sprintf(d.string2, TEST4_STRING2);
    ret = ioctl(fd, IOCTL_TEST4, &d);
```

7.1 Using `fusd_dispatch()`

Recall from Section 4.1 that `fusd_register` returns a *file descriptor* for every device that is successfully registered. This file descriptor can be used to activate device callbacks "manually," without passing control of the application to `fusd_run()`. Whenever the file descriptor becomes readable according to `select(2)`, it should be passed to `fusd_dispatch()`, which in turn will activate callbacks in the same way that `fusd_run()` does. In other words, an application can:

1. Save the file descriptors returned by `fusd_register()`;
2. Add those FUSD file descriptors to an `fd_set` that is passed to `select`, along with any other file descriptors that might be interesting to the application; and
3. Pass every FUSD file descriptor that `select` indicates is readable to `fusd_dispatch`.

`fusd_dispatch()` returns 0 if at least one callback was successfully activated. On error, -1 is returned with `errno` set appropriately. `fusd_dispatch()` will never block—if no messages are available from the kernel, it will return -1 with `errno` set to `EAGAIN`.

7.2 Helper Functions for Constructing an `fd_set`

The FUSD library provides two (optional) utility functions that can make it easier to write applications that integrate FUSD into their own `select()` loops. Specifically:

- `void fusd_fdset_add(fd_set *set, int *max)`—is meant to help construct an `fd_set` that will be passed as the "readable fds" set to select. This function adds the file descriptors of all previously registered FUSD devices to the fd_set `set`. It assumes that `set` has already been initialized by the caller. The integer `max` is updated to reflect the largest file descriptor number in the set. `max` is not changed if the value passed to `fusd_fdset_add` is already larger than the largest FUSD file descriptor added to the set.

Program 8 ioctl-server.c: A driver that handles `ioctl` requests

```
/* This function is run by the driver */
int do_ioctl(struct fusd_file_info *file, int cmd, void *arg)
{
  struct ioctl_data_t *d;

  if (_IOC_TYPE(cmd) != IOCTL_APP_TYPE)
    return 0;

  switch (cmd) {
  case IOCTL_TEST3: /* returns data to the client */
    d = arg;
    strcpy(d->string1, TEST3_STRING1);
    strcpy(d->string2, TEST3_STRING2);
    return 0;
    break;

  case IOCTL_TEST4: /* gets data from the client */
    d = arg;
    printf("ioctl server: test4, string1: got '%s'\n", d->string1);
    printf("ioctl server: test4, string2: got '%s'\n", d->string2);
    return 0;
    break;
  default:
    printf("ioctl server: got unknown cmd, sigh, this is broken\n");
    return -EINVAL;
    break;
  }

  return 0;
}
```

- `void fusd_dispatch_fdset(fd_set *set)`—is meant to be called on the `fd_set` that is *returned* by select. It assumes that `set` contains a set file descriptors that `select()` has indicated are readable. `fusd_dispatch_fdset()` calls `fusd_dispatch` on every descriptor in `set` that is a valid FUSD descriptor. Non-FUSD descriptors in `set` are ignored.

The excerpt of `drums3.c` shown in Program 9 demonstrates the use of these helper functions. This program is similar to the earlier drums.c example: it creates a number of musical instruments such as `/dev/drums/bam` and `/dev/drums/boom`. However, in addition to servicing its musical callbacks, the driver also prints a prompt to standard input asking how "loud" the drums should be. Instead of turning control of `main()` over to `fusd_run()` as in the previous examples, `drums3` uses `select()` to simultaneously watch its FUSD file descriptors and standard input. It responds to input from both sources.

On lines 2–5, an `fd_set` and its associated "max" value are initialized to contain stdin's file descriptor. On line 9, we use `fusd_fdset_add` to add the FUSD file descriptors for all registered devices. (Not shown in this excerpt is the device registration, which is the same as the registration code we saw in `drums.c`.) On line 13 we call select, which blocks until one of the fd's in the set is readable. On lines 17 and 18, we check to see if standard input is readable; if so, a function is called which reads the user's response from standard input and prints a new prompt. Finally, on line 21, we call `fusd_dispatch_fdset`, which in turn will activate the callbacks for

Program 9 drums3.c: Waiting for both FUSD and non-FUSD events in a `select` loop

```
  /* initialize the set */
  FD_ZERO(&fds);

  /* add stdin to the set */
  FD_SET(STDIN_FILENO, &fds);
  max = STDIN_FILENO;

  /* add all FUSD fds to the set */
  fusd_fdset_add(&fds, &max);

  while (1) {
    tmp = fds;
    if (select(max+1, &tmp, NULL, NULL, NULL) < 0)
      perror("selecting");
    else {
      /* if stdin is readable, read the user's response */
      if (FD_ISSET(STDIN_FILENO, &tmp))
        read_volume(STDIN_FILENO);

      /* call any FUSD callbacks that have messages waiting */
      fusd_dispatch_fdset(&tmp);
    }
  }
```

devices that have pending system calls waiting to be serviced.

It's worth reiterating that drivers are not required to use the FUSD helper functions `fusd_fdset_add` and `fusd_dispatch_fdset`. If it's more convenient, a driver can manually save all of the file descriptors returned by `fusd_register`, construct its own `fd_set`, and then call `fusd_dispatch` on each descriptor that is readable. This method is sometimes required for integration with other frameworks that want to take over your `main()`. For example, the GTK user interface framework[11] is event-driven and requires that you pass control of your `main` to it. However, it does allow you to give it a file descriptor and a function pointer, saying "Call this callback when `select` indicates this file descriptor has become readable." A GTK application that implements FUSD devices can work by giving GTK all the FUSD file descriptors individually, and calling `fusd_dispatch()` when GTK calls the associated callbacks.

8 Implementing Blocking System Calls

All of the example drivers that we've seen until now have had an important feature missing: they never had to *wait* for anything. So far, a driver's response to a system call has always been immediately available—allowing the driver to response immediately. However, real devices are often not that lucky: they usually have to wait for something to happen before completing a client's system call. For example, a driver might be waiting for data to arrive from the serial port or over the network, or even waiting for a user action.

In situations like this, a basic capability most device drivers must have is the ability to *block* the caller. Blocking operations are important because they provide a simple interface to user programs that does flow control, rather than something more expensive like continuous polling. For example, user programs expect to be able to execute

[11] http://www.gtk.org

Program 10 console-read.c: A simple blocking system call

```
int do_read(struct fusd_file_info *file, char *user_buffer,
            size_t user_length, loff_t *offset)
{
  char buf[128];

  if (*offset > 0)
    return 0;

  /* print a prompt */
  printf("Got a read from pid %d.  What do we say?\n> ", file->pid);
  fflush(stdout);

  /* get a response from the console */
  if (fgets(buf, sizeof(buf) - 1, stdin) == NULL)
    return 0;

  /* compute length of the response, and return */
  user_length = MIN(user_length, strlen(buf));
  memcpy(user_buffer, buf, user_length);
  *offset += user_length;
  return user_length;
}
```

a statement like `read(fd, buf, sizeof(buf))`, and expect the read call to block (stop the flow of the calling program) until data is available. This is much simpler and more efficient than polling repeatedly.

In the following sections, we'll describe how to block and unblock system calls for devices that use FUSD.

8.1 Blocking the caller by blocking the driver

The easiest (but least useful) way to block a client's system call is simply to block the driver, too. For example, consider Program 10, which implements a device called `/dev/console-read`. Whenever a process tries to read from this device, the driver prints a prompt to standard input, asking for a reply. (The prompt appears in the shell the driver was run in, not the shell that's trying to read from the device.) When the user enters a line of text, the response is returned to the client that did the original `read()`. By blocking the driver waiting for the reply, the client that issued the system call is blocked as well.

Blocking the driver this way is safe—unlike programming in the kernel proper, where doing something like this would block the entire system. It's also easy to implement, as seen from the example above. However, it makes the driver unresponsive to system call requests that might be coming from other clients. If another process tries to do anything at all with a blocked driver's device—even an `open()`—it will block until the driver wakes up again. This limitation makes blocking drivers inappropriate for any device driver that expects to service more than one client at a time.

8.2 Blocking the caller using `-FUSD_NOREPLY`; unblocking it using `fusd_return()`

If a device driver expects more than one client at a time—as is often the case—a slightly different programming model is needed for system calls that can potentially block. Instead of blocking, the driver immediately sends a message to the FUSD framework that says, in essence, "Don't unblock the client that issued this system call, but continue sending additional system call requests that might be coming from other clients." Driver callbacks can send this message to FUSD by returning the special value `-FUSD_NOREPLY` instead of a normal system call return value.

Before a callback blocks the caller by returning `-FUSD_NOREPLY`, it must save the `fusd_file_info` pointer that was provided to the callback as its first argument. Later, when an event occurs which allows the client's blocked system call to complete, the driver should call `fusd_return()`, which will unblock the calling process and complete its system call. `fusd_return()` takes two arguments:

- The `fusd_file_info` pointer that the callback saved earlier; and
- The system call's return value (in other words, the value that would have been returned by the callback function had it not returned `-FUSD_NOREPLY`). FUSD itself *does not* examine the return value passed as the second argument to `fusd_return`; it simply propagates that value back to the kernel as the return value of the blocked system call.

Drivers should never call `fusd_return` more than once on a single `fusd_file_info` pointer. Doing so will have undefined results, similar to calling `free()` twice on the same pointer.

It also bears repeating that a callback can call *either* call fusd_return() explicitly *or* return a normal return value (i.e., not `-FUSD_NOREPLY`), but not both.

`-FUSD_NOREPLY` and `fusd_return()` make it easy for a driver to block a process, then unblock it later when data becomes available. When the callback returns `-FUSD_NOREPLY`, the driver is freed up to wait for other events, even though the process making the system call is still blocked. The driver can then wait for something to happen that unblocks the original caller—for example, another FUSD event, data from a serial port, or data from the network. (Recall from Section 7 that a FUSD driver can simultaneously wait for both FUSD and non-FUSD events.)

FUSD includes an example program, `pager.c`, which demonstrates these techniques. The pager driver implements a simple notification interface which lets any number of "waiters" wait for a signal from a "notifier." All the waiters wait by trying to read from `/dev/pager/notify`. Those reads will block until a notifier writes the string `page` to `/dev/pager/input`. It's easy to try the application out—run the driver, and then open three other shells. In two of them, type `cat /dev/pager/notify`. The reads will block. Then, in the third shell, type `echo page > /dev/pager/input`—the other two shells should become unblocked.

Let's take a look at how this application is implemented, step by step.

8.2.1 Keeping Per-Client State

The first thing to notice about `pager.c` is that it keeps *per-client state*. That is, for every file descriptor open to the driver, a structure is allocated that has information relating to that file descriptor. Previous driver examples were, for the most part, *reactive*—they received requests, and immediately generated responses. Since there was never more than one request outstanding, there was no need to keep a list of them. The pager application is the first one that must keep track of an arbitrary number of requests that might be outstanding at the same time. The first excerpt of `pager.c`, which appears in Program 11, shows the code which creates this per-client state. Lines 1–6 define a structure, `pager_client`, which keeps all the information we need about each client attached to the driver. The `open` callback for `/dev/pager/notify`, shown on lines 12–31, allocates memory for an instance of this structure and adds it to a linked list. (If the memory allocation fails, an error is returned to the client on line 18; this will prevent the file from opening.) Note on line 25 that we use the `private_data` field to store a pointer to the client state; this allows the structure to be retrieved when later callbacks on this file descriptor arrive. The memory is deallocated when the file is closed; we'll see that in a later section.

Program 11 pager.c (Part 1): Creating state for every client using the driver

```
/* per-client structure to keep track of who has an open FD to us */
struct pager_client {
  int last_page_seen;   /* seq. no. of last page this client has seen */
  struct fusd_file_info *read; /* outstanding read request, if any */
  struct fusd_file_info *polldiff; /* outstanding polldiff request */
  struct pager_client *next;   /* to construct the linked list */
};

struct pager_client *client_list = NULL; /* list of clients (open FDs) */
int last_page = 0; /* seq. no. of the most recent page to arrive */

/* open on /dev/pager/notify: create state for this client */
static int pager_notify_open(struct fusd_file_info *file)
{
  /* create state for this client */
  struct pager_client *c = malloc(sizeof(struct pager_client));

  if (c == NULL)
    return -ENOBUFS;

  /* initialize fields of this client state */
  memset(c, 0, sizeof(struct pager_client));
  c->last_page_seen = last_page;

  /* save the pointer to this state so it gets returned to us later */
  file->private_data = c;

  /* add this client to the client list */
  c->next = client_list;
  client_list = c;

  return 0;
}
```

Another thing to notice about the open callback is the use of the `last_page_seen` variable. The driver gives a sequence number to every page it receives; `last_page_seen` stores the number of the most recent page seen by a client. When a new client arrives (i.e., it opens `/dev/pager/notify`), its `last_page_seen` state is set equal to the page that has most recently arrived; this forces a new client to wait for the *next* page, rather than immediately being notified of a page that has arrived in the past.

8.2.2 Blocking and completing reads

The next part of `pager.c` is shown in Program 12. The `pager_notify_read` function seen on line 1 is registered as the `read` callback for the `/dev/pager/notify` device. It blocks the read request using the technique we described earlier: it stores the `fusd_file_info` pointer in that client's state structure, and returns `-FUSD_NOREPLY`. (Note that the pointer to the client's state structure comes from the `private_data` field of `fusd_file_info`, where the open callback stored it.)

Program 12 pager.c (Part 2): Block clients' `read` requests, and later completing the blocked reads

```
ssize_t pager_notify_read(struct fusd_file_info *file, char *buffer,
                          size_t len, loff_t *offset)
{
  struct pager_client *c = (struct pager_client *) file->private_data;

  if (c == NULL || c->read != NULL) {
    fprintf(stderr, "pager_read's arguments are confusd, alas");
    return -EINVAL;
  }

  c->read = file;
  pager_notify_complete_read(c);
  return -FUSD_NOREPLY;
}

void pager_notify_complete_read(struct pager_client *c)
{
  /* if there is no outstanding read, do nothing */
  if (c == NULL || c->read == NULL)
    return;

  /* if there are no outstanding pages, do nothing */
  if (c->last_page_seen >= last_page)
    return;

  /* bring this client up to date with the most recent page */
  c->last_page_seen = last_page;

  /* and notify the client by unblocking the read (read returns 0) */
  fusd_return(c->read, 0);
  c->read = NULL;
}

ssize_t pager_input_write(struct fusd_file_info *file,
                          const char *buffer, size_t len, loff_t *offset)
{
  struct pager_client *c;

  /* ... */

  CASE("page") {
    last_page++;

    for (c = client_list; c != NULL; c = c->next) {
      pager_notify_complete_polldiff(c);
      pager_notify_complete_read(c);
    }
  }
```

`pager_notify_complete_read` *unblocks* previously blocked reads. This function first checks to see that there is, in fact, a blocked read (line 19). It then checks to see if a page has arrived that the client hasn't seen yet (line 23). Finally, it updates the client state and unblocks the blocked read by calling `fusd_return`. Note the second argument to `fusd_return` is a 0; as we saw in Section 4.3, a 0 return value to a `read` system call means EOF. (The system call will be unblocked regardless of the return value.)

`pager_notify_complete_read` is called every time a new page arrives. New pages are processed by `pager_input_write` (line 34), which is the `write` callback for `/dev/pager/input`. After recording the fact that a new page has arrived, it calls `pager_notify_complete_read` for each client that has an open file descriptor. This will complete the reads of any clients who have not yet seen this new data, and have no effect on clients that don't have outstanding reads.

There is another interesting point to notice about `pager_notify_read`. On line 12, after it stores the blocked system call's pointer, but before we return `-FUSD_NOREPLY`, it calls the completion function. This has the effect of returning any data that might already be available back to the caller immediately. If that happens, we will end up calling `fusd_return` *before* we return `-FUSD_NOREPLY`. This probably seems strange, but it's legal. Recall that a callback can call fusd_return() explicitly *or* return a normal (not `-FUSD_NOREPLY`) return value, but not both; the order doesn't matter.

8.2.3 Using `fusd_destroy()` to clean up client state

Finally, let's take a look at one last aspect of the pager program: how it cleans up the per-client state when a client leaves. This is mostly straightforward, with one exception: a client may have an outstanding read request out when a close request comes in. Normally, a client can't make another system call request while a previous system call is still blocked. However, the `close` system call is an exception: it gets called when a client dies (for example, if it receives an interrupt signal). If a `close` comes in while another system call is still outstanding, the state associated with the outstanding request should be freed to avoid a memory leak. The `fusd_destroy` function is used to do this, seen on linen 12-14 of Program 13.

8.3 Retrieving a blocked system call's arguments from a `fusd_file_info` pointer

In the previous section, we showed how the `fusd_return` function can be used to specify the return value of a system call that was previously blocked. However, many system calls have side effects in addition to returning a value—for example, in a `read()` request, the data being returned has to be copied into the caller's buffer. To facilitate this, FUSD provides accessor functions that let drivers retrieve the arguments that had been passed to its callbacks at the time the call was originally issued. For example, the `fusd_get_read_buffer()` function will return a pointer to the data buffer that is provided with `read()` callbacks. Drivers can use these accessor functions to affect change to a client *before* calling `fusd_return()`.

The following accessor functions are available, all of which take a single `fusd_file_info *` argument:

- `int char *fusd_get_read_buffer`—The destination buffer for data that a driver is returning to a process doing a `read()`.
- `const char *fusd_get_write_buffer`—The source buffer containing data sent to the driver by a process doing a `write()`.
- `fusd_get_length`—The length (in bytes) of the buffer for either a `read()` or a `write()`.
- `loff_t fusd_get_offset`—The file descriptor's byte offset, typically used in `read()` and `write()` callbacks.

Program 13 pager.c (Part 3): Cleaning up when a client leaves

```
/* close on /dev/pager/notify: destroy state for this client */
static int pager_notify_close(struct fusd_file_info *file)
{
  struct pager_client *c;

  if ((c = (struct pager_client *) file->private_data) != NULL) {

    /* take this client off our client list */
    client_list_remove(c);

    /* if there is a read outstanding, free the state */
    if (c->read != NULL) {
      fusd_destroy(c->read);
      c->read = NULL;
    }
    /* destroy any outstanding polldiffs */
    if (c->polldiff != NULL) {
      fusd_destroy(c->polldiff);
      c->polldiff = NULL;
    }

    /* get rid of the struct */
    free(c);
    file->private_data = NULL;
  }
  return 0;
}
```

- `int fusd_get_ioctl_request`—An ioctl's request "command number" (i.e., the first argument of an ioctl).
- `int fusd_get_ioctl_arg`—The second argument of an ioctl for non-data-bearing `ioctl` requests (i.e., `_IO` commands).
- `void *fusd_get_ioctl_buffer`—The data buffer for data-bearing `ioctl` requests (`_IOR`, `_IOW`, and `_IORW` commands).
- `int fusd_get_poll_diff_cached_state`—See Section 9.

We got away without using these accessor functions in our `pager.c` example because the pager doesn't actually return data—it just blocks and unblocks `read` calls. However, the FUSD distribution contains another example program, `logring`, that demonstrates their use.

`logring` makes it easy to access the most recent (and only the most recent) output from a process. It works just like `tail -f` on a log file, except that the storage required never grows. This can be useful in embedded systems where there isn't enough memory or disk space for keeping complete log files, but the most recent debugging messages are sometimes needed (e.g., after an error is observed).

`logring` uses FUSD to implement a character device, `/dev/logring`, that acts like a named pipe that has a finite, circular buffer. The size of the buffer is given as a command-line argument. As more data is written into the buffer, the oldest data is discarded. A process that reads from the logring device will first read the existing buffer, then block and see new data as it's written, similar to monitoring a log file using `tail -f`.

You can run this example program by typing `logring <logsize>`, where `logsize` is the size of the circular buffer in bytes. Then, type `cat /dev/logring` in a shell. The `cat` process will block, waiting for data. From another shell, write to the logring (e.g., `echo Hi there > /dev/logring`). The `cat` process will see the message appear.

(This example program is based on *emlog*, a (real) Linux kernel module with identical functionality. If you find logring useful, but want to use it on a system that does not have FUSD, check out the original emlog[12].)

9 Implementing `select`able Devices

One important feature that almost every device driver in a system should have is support for the `select(2)` system call. `select` allows clients to assemble a set of file descriptors and ask to be notified when one of them becomes readable or writable. This simple feature is deceptively powerful—it allows clients to wait for any number of a set of possible events to occur. This is fundamentally different than (for example) a blocking read, which only unblocks on one kind of event. In this section, we'll describe how FUSD can be used to create a device whose state can be queried by a client's call to `select(2)`.

This section is limited to a discussion what a FUSD driver writer needs to know to implement a selectable device. Details of the FUSD implementation required to support this feature are described in Section 11.1

9.1 Poll state and the `poll_diff` callback

FUSD's implementation of selectable devices depends on the concept of *poll state*. A file descriptor's poll state is a bitmask that describes its current properties—readable, writable, or exception raised. These three states correspond to `select(2)`'s three `fd_sets`. FUSD has constants used to describe these states:

- `FUSD_NOTIFY_INPUT`—Input is available; a read will not block.
- `FUSD_NOTIFY_OUTPUT`—Output space is available; a write will not block.
- `FUSD_NOTIFY_EXCEPT`—An exception has occurred.

These constants can be combined with C's bitwise-or operator. For example, a descriptor that is both readable and writable is expressed as `FUSD_NOTIFY_INPUT | FUSD_NOTIFY_OUTPUT`. 0 means a file descriptor is not readable, not writable, and not in the exception set.

For a FUSD device to be selectable, its driver must implement a callback called `poll_diff`. This callback is very different than the others; it is not a "direct line" between the client and the driver as is the case with a call such as `ioctl`. A driver's response to `poll_diff` is *not* the return value seen by a client's call to `select`. When a client tries to `select` on a set of file descriptors, the kernel collects the responses from all the appropriate callbacks—`poll` for file descriptors managed by kernel drivers, and `poll_diff` callbacks those managed by FUSD drivers—and synthesizes all of that information into the return value seen by the client.

FUSD keeps a cache of the poll state it has most recently received from each FUSD device driver, initially assumed to be 0. This state is returned to clients trying to `select()` on devices managed by those drivers. Under certain conditions, FUSD sends a query to the driver in order to ensure that the kernel's poll state cache is up to date. This query takes the form of a `poll_diff` callback activation, which is given a single argument: the poll

[12] http://www.circlemud.org/jelson/software/emlog

state that FUSD currently has cached. The driver should consult its internal data structures to determine the actual, current poll state (i.e., whether or not buffers have readable data). Then:

- If the FUSD cache is incorrect (that is, the current true poll state is different than FUSD's cached state), the current poll state should be returned immediately.
- If the FUSD cache is up to date (that is, it matches the real current state), the callback should save the `fusd_file_info` pointer and return `-FUSD_NOREPLY`. Later, when the poll state changes, the driver can call `fusd_return()` to update FUSD's cache.

In other words, when a driver's `poll_diff` callback is activated, the kernel is effectively saying to the driver, "Here is what I think the current poll state of this file descriptor is; let me know when that state *changes*." The driver can either respond immediately (if the kernel's cache is already known to be out of date), or return `-FUSD_NOREPLY` if no update is immediately necessary. Later, when the poll state changes (for example, if new data arrives that makes a device readable), the driver can used its saved `fusd_file_info` pointer to send a poll state update to the kernel.

When a FUSD driver sends a poll state update, it might (or might not) have the effect of waking up a client that was blocked in `select(2)`. On the same note, it's worth reiterating that a `-FUSD_NOREPLY` response to a `poll_diff` callback *does not* necessarily block the client—other descriptors in the client's `select` set might be readable, for example.

9.2 Receiving a `poll_diff` request when the previous one has not been returned yet

Calls such as `read` and `write` are synchronous from the standpoint of an individual client—a request is made, and the requester blocks until a reply is received. This means that there can't ever be more than a single `read` request outstanding for a single client at a time. (The driver as a whole may be keeping track of many outstanding `read` requests in parallel, but no two of them will be from the same client file descriptor.)

As we mentioned in the previous section, the `poll_diff` callback is different from other callbacks. It is not part of a synchronous request/reply sequence that causes the client to block. It is also an interface to the *kernel*, not directly to the client. So, it *is* possible to receive a `poll_diff` request while there is already one outstanding. This happens if the kernel's poll state cache changes, causing it to notify the driver that it has a new cached value.

This is easy to handle; the client should simply

1. Destroy the old (now out-of-date) `poll_diff` request using the `fusd_destroy` function we saw in Section 8.2.3.
2. Either respond to or save the new `poll_diff` request, exactly as described in the previous section.

The next section will show an example of this technique.

9.3 Adding `select` support to `pager.c`

Given the explanation of `poll_diff` in the previous sections, it might seem that implementing a selectable device is a daunting task. It's actually not as bad as it sounds—the example code may well be shorter than its explanation!

Program 14 shows the implementation of `poll_diff` in `pager.c`, which makes its notification interface (`/dev/pager/notify`) selectable. It is decomposed into a "top half" and "bottom half" function, exactly as we did for the blocking `read` implementation in Program 12. First, on lines 1–20, we see the the callback for `poll_diff` callback itself. It is virtually identical to the `read` callback in Program 12. The main difference is

Program 14 pager.c (Part 4): Supporting `select(2)` by implementing a `poll_diff` callback

```
ssize_t pager_notify_polldiff(struct fusd_file_info *file,
                              unsigned int cached_state)
{
  struct pager_client *c = (struct pager_client *) file->private_data;

  if (c == NULL)
    return -EINVAL;

  /* if we're already holding a polldiff request that we haven't
   * replied to yet, destroy the old one and hold onto only the new
   * one */
  if (c->polldiff != NULL) {
    fusd_destroy(c->polldiff);
    c->polldiff = NULL;
  }

  c->polldiff = file;
  pager_notify_complete_polldiff(c);
  return -FUSD_NOREPLY;
}

void pager_notify_complete_polldiff(struct pager_client *c)
{
  int curr_state, cached_state;

  /* if there is no outstanding polldiff, do nothing */
  if (c == NULL || c->polldiff == NULL)
    return;

  /* figure out the "current" state: i.e. whether or not the pager
   * is readable for this client based on the last page it saw */
  if (c->last_page_seen < last_page)
    curr_state = FUSD_NOTIFY_INPUT; /* readable */
  else
    curr_state = 0; /* not readable or writable */

  /* cached_state is what the kernel *thinks* the state is */
  cached_state = fusd_get_poll_diff_cached_state(c->polldiff);

  /* if the state is not what the kernel thinks it is, notify the
     kernel of the change */
  if (curr_state != cached_state) {
    fusd_return(c->polldiff, curr_state);
    c->polldiff = NULL;
  }
}
```

that it first checks (on line 12) to see if a `poll_diff` request is already outstanding when a new request comes in. If so, the out-of-date request is destroyed using `fusd_destroy`, as we described in Section 9.2.

The bottom half is shown on lines 22-46. First, on lines 32–35, it computes the current poll state—if a page has arrived that the client hasn't seen yet, the file is readable; otherwise, it isn't. Next, the driver compares the current poll state with the poll state that the kernel has cached. If the kernel's cache is out of date, the current state is returned to the kernel. Otherwise, it does nothing.

As with the `read` callback we saw previously, notice that `pager_notify_complete_polldiff` is called in two different cases:

1. It is called immediately from the `pager_notify_polldiff` callback itself. This causes the current poll state to be returned to the kernel immediately when the request arrives if the driver already knows the kernel's state needs to be updated.
2. It is called when new data arrives that causes the poll state to change. Refer back to Program 12 on page 27; in the callback that receives new pages, notice on line 45 that the `poll_diff` completion function is called alongside the `read` completion function.

With this `poll_diff` implementation, it is possible for a client to open `/dev/pager/notify`, and block in a `select(2)` system call. If another client writes `page` to `/dev/pager/input`, the first client's `select` will unblock, indicating the file has become readable.

For additional example code, take a look at the `logring` example program we first mentioned in Section 8.3. It also supports `select` by implementing a similar `poll_diff` callback.

10 Performance of User-Space Devices

This section hasn't been written yet. I have some pretty graphs and whatnot, but no time to write about them here before the release.

11 FUSD Implementation Notes

In this section, we describe some of the details of how FUSD is implemented. It's not necessary to understand these details in order to use FUSD. However, these notes can be useful for people who are trying to understand the FUSD framework itself—hackers, debuggers, or the generally curious.

11.1 The situation with `poll_diff`

In-kernel device drivers support select by implementing a callback called `poll`. This driver's callback is supposed to do two things. First, it should return the current state of a file descriptor—a combination of being readable, writable, or having exceptions. Second, it should provide a pointer to one of the driver's internal wait queues that will be awakened whenever the state changes. The `poll` call itself should never block—it should just instantaneously report what the *current* state is.

FUSD's implementation of selectable devices is different, but attempts to maintain three properties that we thought to be most important from the point of view of a client using `select`. Specifically:

1. The `select(2)` call itself should never become blocked. For example, if one file descriptor in its set isn't readable, that shouldn't prevent it from reporting other file descriptors that are.

2. If `select(2)` indicates a file descriptor is readable (or writable), a read (or write) on that file descriptor shouldn't block.

3. Clients should be allowed to seamlessly `select` on any set of file descriptors, even if that set contains a mix of both FUSD and non-FUSD devices.

The FUSD kernel module keeps a cache of the driver's most recent answer for each file descriptor, initially assumed to be 0. When the kernel module's internal `poll` callback is activated, it:

1. Dispatches a *non*-blocking `poll_diff` to the associated user-space driver, asking for a cache update—if and only if there isn't already an outstanding poll diff request out that has the same value.

2. Immediately returns the cached value to the kernel

In addition, the cached value's readable bit is cleared on every read; the writable bit is cleared on every write. This is necessary to prevent old poll state—which says "device is readable"—from being returned out of the cache when it might be invalid. FUSD assumes that any read to a device can make it potentially unreadable. This mechanism is what causes an updated poll diff to be sent to a client before the previous one has been returned.

(this section isn't finished yet; fancy time diagrams coming someday)

11.2 Restartable System Calls

No time to write this section yet...

A Using `strace`

This section hasn't been written yet. Contributions are welcome.