FreeBSD® JOURNAL

September/October 2024

S&W PUBLISHING LLC
PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

Welcome the September/October issue of the *FreeBSD Journal*. This is our Kernel Development issue and includes articles on topics like Porting VPP to FreeBSD, Valgrind for FreeBSD, the first installment of a three-part tutorial on Character Device Drivers, and much more!

Before you dive into these great articles, we have a quick update for 2025. Starting in January, the *Journal* will shift to a quarterly schedule, continuing to bring you the same top-notch tutorials, articles, and columns. This new schedule will allow us to focus on delivering the best FreeBSD content while keeping the *Journal* accessible to everyone.

Everything else you love about the *Journal* will stay the same, and we look forward to bringing you another year of valuable FreeBSD content!

On behalf of the FreeBSD Foundation and *FreeBSD Journal* Editorial Board,

Anne Dickison
Deputy Director
FreeBSD Foundation

John Baldwin
Chair of the *FreeBSD Journal* Editorial Board

FreeBSD® JOURNAL

September/October 2024

Kernel Development

WeGet letters

by Michael W Lucas

Dear Most Useless Advice Columnist in Technology (or Anywhere),

In all of open source, kernel developers are the elite. They get to implement the really cool stuff and invent nifty new features, like ZFS and buffer caches and memory protections. Any advice on how I could become one of them?

I've read your column several times, and honesty demands that I inform you that I'm going to maximize my chances of success by listening carefully to everything you suggest, then doing the exact opposite.

—Novice But Not Naïve

Dear NBNN̈,

"Working in computing isn't enough for me. I want my failures to be truly inexplicable!"

Very well.

Many people fantasize that kernel developers are programming elite. John Baldwin, of repeated FreeBSD Core Team fame as well as the editorial board chair of this very Journal, went from writing documentation straight into kernel development. John has been unfortunate enough to know me for decades so I can confidently assure you that not only is he not an elite but the remarkable, incriminating, and noteworthy things about him have absolutely nothing to do with programming. Kernel developers must achieve a minimum competence, yes, but beyond a couple rules, there's nothing special about kernel code. Imagining that kernel programmers are an elite will sabotage you before you start, so I strongly encourage it.

Many people fantasize that kernel developers are programming elite.

If you insist on proceeding, though, if you demand you be allowed to weave yourself a chrysalis and transform into a kernel developer like a panic-prone memory-dumping file-corrupting butterfly, immediately separate your dreams from your goals. A goal is something actionable that is completely within your control to achieve. Accomplishing a dream requires other people to intervene on your behalf. Going out for a dinner date with that attractive person? Totally a dream. Asking that attractive person out for a dinner date, and when they remind you that you are inherently unlovable and should leave them the heck alone instead of stalking them like the creepy hero of a so-called "romance?" An absolutely achievable goal!

You cannot control other people. Work on goals. Never on dreams.

What goals can you set that would guide you become a kernel developer?

Start by reading the documentation. There are books like *The Design and Implementation of the FreeBSD Operating System, FreeBSD Device Drivers*, and *Designing BSD Rootkits*, which add an interesting twist to learning how the kernel works. *The FreeBSD Developers Handbook* is freely available. Fill your brain. Do the exercises. If something is beyond you, well, people have written articles and books discussing it.

Most transformations end hard right here because humans cherish cozy comfy non-actionable dreams and aren't as fond of ugly hard goals.

Note that I didn't say "ask other people how to start learning about the kernel." If you haunt the mailing lists, the forums, or the Internet's sketchier discussion boards you'll occasionally see people asking for help in learning to program the kernel. You might think that these people are looking for the list above, but the answer I give here appears on the most cursory search and comes across as *please hold my hand*. Do caterpillars ask for help weaving their chrysalis? No! They sweat and struggle so they can slither into their cramped cocoons and simmer into transcendence. You must do the work. Most transformations end hard right here because humans cherish cozy comfy non-actionable dreams and aren't as fond of ugly hard goals.

As with any other part of contributing to an open source project, you need to find a tiny piece to work on. Start with bugs. Problem reports are a gold mine for the aspiring kernel developer. As you look through possible projects, you must again separate goals from dreams. "Solve several panic bugs and get my fixes committed" is a dream. It requires that established kernel developers notice your fixes and choose to incorporate them. "Solve one reported kernel panic this month" isn't exactly a goal, because you can't guarantee that you will be able to solve it. "Spend ten hours this month working on a reported kernel panic, without taking breaks every three minutes to gripe on social media, in the work chat, or to my pet who has to put up with me even though I'm inherently unlovable." There—*that's* a goal! Complete enough of those goals and you'll develop the skill of kernel programming.

The hard part of working in the kernel, though?

Other people.

Suppose you develop patches to fix reported problems and attach them to the bug. You can't make a project member notice your work. If they notice your work, you can't make them take your patch as-is. A project member might use your patch as inspiration or a proof-of-concept and create a wholly different patch for reasons you hadn't even thought of. Making people notice you is a dream. Making yourself dang hard to ignore by submitting a whole series of quality patches is absolutely a goal.

An interesting thing about how caterpillars become butterflies. They don't. We see the caterpillar crawl into its cocoon and the butterfly emerge, so we assume that there's been a transformation when the harsh reality is, the caterpillar's chrysalis? It's a coffin. The caterpillar crawls in and melts to goo surrounding a tiny lump that's basically a self-assembling butterfly kit. The butterfly's first meal is 100% Grade A caterpillar sludge. When you submit your twentieth patch and still it feels like nobody cares, be grateful that you haven't transformed yourself into literal physical muck. Mental muck is less noticeable.

Suppose your patches get picked up? What then?

Again, it's people.

In that glorious aeon when the Sacred and Penultimately Blessed Computer Science Research Group distributed primordial BSD, a single person could achieve a good understanding of Unix. A complete install took only a few megabytes. And yes, that included the compiler and source code, what part of "complete install" was unclear? College students were expected to read and understand the code.

If you achieve your dream and become a full-on kernel developer, you'll discover that people are not a problem.

Today? By the time you finish reading the base system source code, it's changed and you get to start over. Becoming a "kernel developer" is almost impossible. You might, at best, become a trusted developer responsible for one tiny slice of the kernel. Performing maintenance will require interacting with other parts of the kernel, which means discussing your changes with the people responsible for those parts. Working in the kernel is no different than programming in userland, except people believe you've achieved a certain minimal competence.

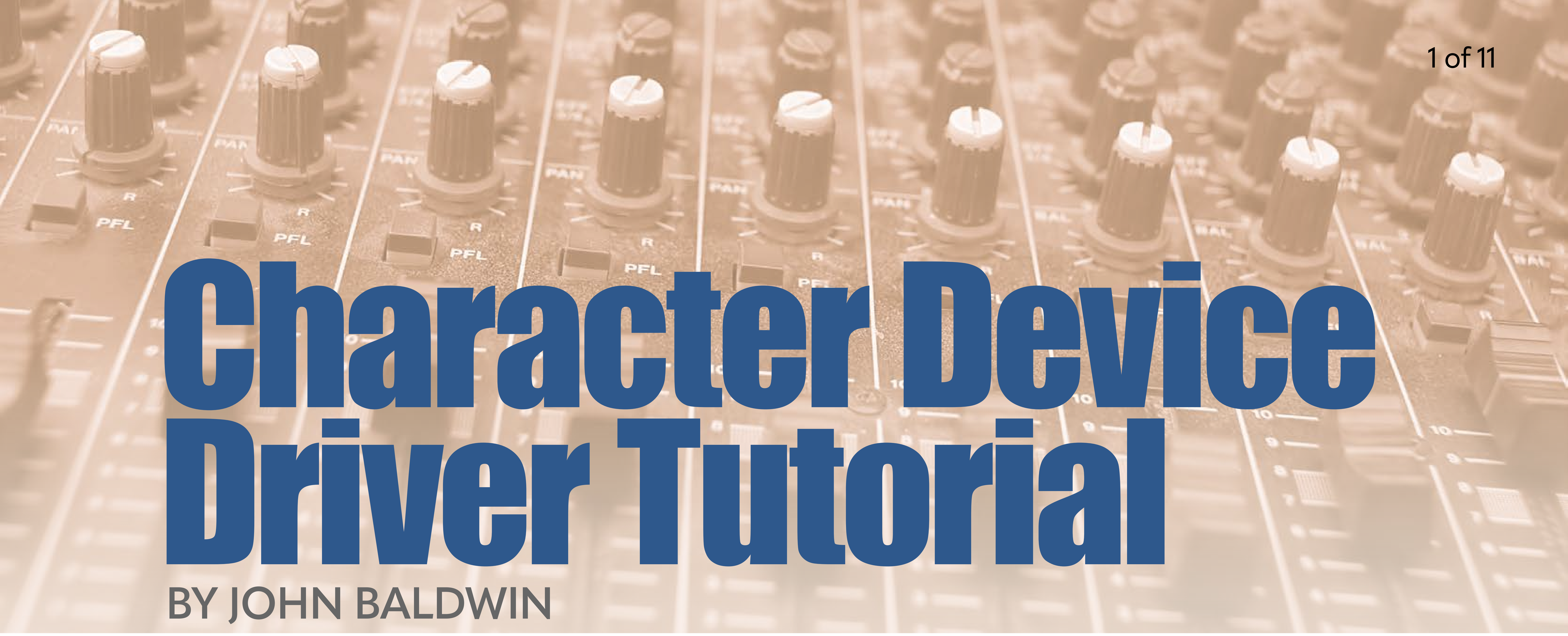
If you achieve your dream and become a full-on kernel developer, you'll discover that people are not a problem. They are the problem. Every change you make will upset someone. Users and non-kernel programmers will have this weird idea that you're the elite, that you know what you're doing, that you are less baffled than them.

As you've declared an intent to not merely ignore but reverse my thoughts, let me summarize: becoming a kernel programmer is the one true path to happiness and I wish you well. Dream on!

Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS Unlike esteemed *FreeBSD Journal* Editorial Chair and elite kernel developer John Baldwin, Michael W. Lucas remained in documentation. His latest book is *Run Your Own Mail Server*, which uses FreeBSD as a reference platform. Learn more at https://mwl.io.

Contents



Character Device Driver Tutorial

BY JOHN BALDWIN

Character devices provide pseudo files exported to userspace applications by the device filesystem (devfs(5)). Unlike standard filesystems where the semantics of various operations such as reading and writing are the same across all files within a filesystem, each character device defines its own semantics for each file operation. Character device drivers declare a character device switch (`struct cdevsw`) which includes function pointers for each file operation.

Character device switches are often implemented as part of a hardware device driver. FreeBSD's kernel provides several wrapper APIs which implement a character device on top of a simpler set of operations. The disk(9) API implements an internal character device switch on top of the methods in struct disk for example. Several device drivers provide a character device to export device behavior that doesn't map to an existing in-kernel subsystem to userspace.

Character device switches are often implemented as part of a hardware device driver.

Other character device switches are implemented purely as a software construct. For example, the `/dev/null` and `/dev/zero` character devices are not associated with any hardware device.

In a series of three articles, the first of which is this one, we will build a simple character device driver progressively adding new functionality to explore character device switches and several of the operations character device drivers can implement. The full source of each version of device driver can be found at https://github.com/bsdjhb/cdev_tutorial. We will start with a barebones driver which creates a single character device.

Lifecycle Management

A character device driver is responsible for explicitly creating and destroying character devices. Active character devices are represented by instances of `struct cdev`. Character devices are created by the make_dev_s(9) function. This function accepts a pointer to an arguments structure, a pointer to a character device object pointer, and a printf-style format string and following arguments. The format string and following arguments are used to construct the name of the character device.

The arguments structure contains a few mandatory fields and several optional fields. The structure must be initialized by a call to `make_dev_args_init()` before setting any fields.

The **mda_devsw** member must point to the character device switch. The **mda_uid**, **mda_gid**, and **mda_mode** fields should be set to the initial user ID, group ID, and permissions of the device node. Most character devices are owned by root:wheel, and the constants UID_ROOT and GID_WHEEL can be used for this. The **mda_flags** field should also be set to either MAKEDEV_NOWAIT or MAKEDEV_WAITOK. Additional flags can be included via the C or operator if needed. For our sample driver, we set MAKEDEV_CHECKNAME so that we can fail gracefully with an error if an echo device already exists rather than panicking the system.

Character devices are destroyed by passing a pointer to the character device to **destroy_dev()**. This function will block until all references to the character device have been removed. This includes waiting for any threads currently executing in character device switch methods for this device to return from those methods. Once **destroy_dev()** returns, it is safe to release any resources used by the character device. Alternatively, character devices can be destroyed asynchronously via either **destroy_dev_sched()** or **destroy_dev_sched_cb()**. These functions schedule destruction of the character device on an internal kernel thread. For **destroy_dev_sched_cb()**, the supplied callback is invoked with the supplied argument after the character device has been destroyed. This can be used to release resources used by the character device. Keep in mind that one of the resources a character device uses are the character device switch methods. This means, for example, that module unloading must wait for any character devices using functions defined in that module to be destroyed.

Alternatively, character devices can be destroyed asynchronously.

For our initial driver (Listing 1), we use a module event handler to create a /dev/echo device when the module is loaded and destroy it when the module is unloaded. After building and loading this module, the device exists but isn't able to do much as shown in Example 1. The character device switch for this driver (**echo_cdevsw**) is initialized with only two required fields: **d_version** must always be set to the constant D_VERSION, and **d_name** should be set to the driver name.

Listing 1: Barebones Driver

```
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/module.h>

static struct cdev *echodev;

static struct cdevsw echo_cdevsw = {
      .d_version =      D_VERSION,
      .d_name =         "echo"
};

static int
```

```
echodev_load(void)
{
	struct make_dev_args args;
	int error;

	make_dev_args_init(&args);
	args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
	args.mda_devsw = &echo_cdevsw;
	args.mda_uid = UID_ROOT;
	args.mda_gid = GID_WHEEL;
	args.mda_mode = 0600;
	error = make_dev_s(&args, &echodev, "echo");
	return (error);
}

static int
echodev_unload(void)
{
	if (echodev != NULL)
		destroy_dev(echodev);
	return (0);
}

static int
echodev_modevent(module_t mod, int type, void *data)
{
	switch (type) {
	case MOD_LOAD:
		return (echodev_load());
	case MOD_UNLOAD:
		return (echodev_unload());
	default:
		return (EOPNOTSUPP);
	}
}

DEV_MODULE(echodev, echodev_modevent, NULL);
```

Example 1: Using the Barebones Driver

```
# ls -l /dev/echo
crw-------  1 root wheel 0x39 Oct 25 13:06 /dev/echo
# cat /dev/echo
cat: /dev/echo: Operation not supported by device
```

Reading and Writing

Now that we have a character device, let's add some behavior. As the name "echo" implies, this device should accept input by writing to the device and echo that input back out by reading from the device. To provide this, we will add read and write methods to the character device switch.

Read and write requests for character devices are described by a **`struct uio`** object. Two of the fields in this structure are useful for character device drivers: **`uio_offset`** is the logical file offset (e.g. from lseek(2)) for the start of the request and **`uio_resid`** is the number of bytes to transfer. Data is transferred between the application buffer and an in-kernel buffer by the uiomove(9) function. This function updates members of the uio object including **`uio_offset`** and **`uio_resid`** and can be called multiple times. A request can be completed as a short operation by moving a subset of bytes to or from the application buffer.

The second version of the echo driver adds a global static buffer to use as the backing store for read and write requests. The logical file offset is treated as an offset into the global buffer. Requests are truncated to the size of the buffer, so that reading beyond the end of the buffer triggers a zero-byte read indicating EOF. Writes beyond the end of the buffer fail with the error **`EFBIG`**. To protect against concurrent access, a global sx(9) lock is used to protect the buffer. An sx(9) lock is used instead of a regular mutex since **`uiomove()`** might sleep while it faults in a page backing an application buffer. Listing 2 shows the read and write character device methods.

Listing 2: Read and Write Using a Global Buffer

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
      size_t todo;
      int error;

      if (uio->uio_offset >= sizeof(echobuf))
            return (0);

      sx_slock(&echolock);
      todo = MIN(uio->uio_resid, sizeof(echobuf) - uio->uio_offset);
      error = uiomove(echobuf + uio->uio_offset, todo, uio);
      sx_sunlock(&echolock);
      return (error);
}

static int
echo_write(struct cdev *dev, struct uio *uio, int ioflag)
{
      size_t todo;
      int error;

      if (uio->uio_offset >= sizeof(echobuf))
            return (EFBIG);

      sx_xlock(&echolock);
```

```
        todo = MIN(uio->uio_resid, sizeof(echobuf) - uio->uio_offset);
        error = uiomove(echobuf + uio->uio_offset, todo, uio);
        sx_xunlock(&echolock);
        return (error);
}
```

The body of these methods are mostly identical. One reason for this is that the arguments to `uiomove()` are the same for both read and write. This is because the uio object encodes the direction of the data transfer as part of its state.

If we load this version of the driver, we can now interact with the device by reading and writing to it. Example 2 shows a few interactions demonstrating the echo behavior. Note that the output of jot exceeded the size of the driver's 64-byte buffer, so the subsequent read of the device was truncated.

Example 2: Echoing Data Using a Global Buffer

```
# cat /dev/echo
# echo foo > /dev/echo
# cat /dev/echo
foo
# jot -c -s "" 70 48 > /dev/echo
# cat /dev/echo
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno#
```

Device Configuration via ioctl()

The fixed size of the global buffer is a weird quirk of this device. We can permit changing the buffer size by adding a custom ioctl(2) command for this device. I/O control commands are named by a command constant and accept an optional argument.

Command constants are defined by one of the `_IO`, `_IOR`, `_IOW`, or `_IOWR` macros from the <sys/ioccom.h> header. All these macros accept a group and number as the first two arguments. Both values are 8 bits. Typically, an ASCII alphabetical character is used as the group, and all commands for a given driver use the same group. FreeBSD's kernel defines several existing sets of I/O control commands. A set of generic commands that can be used with any file descriptor are defined in <sys/filio.h> using the group 'f'. Other sets are intended for use with specific types of file descriptors such as the commands in <sys/sockio.h> which are defined for sockets. For custom commands for a character device driver, do not use the 'f' group to avoid potential conflicts with the generic commands in <sys/filio.h>. Each command should use a different value for the number argument. If a command accepts an optional argument, the type of the argument must be given as the third argument to the `_IOR`, `_IOW`, or `_IOWR` macro. The `_IOR` macro defines a command that returns a value from the driver to the userspace application (the command "reads" the argument from the driver). The `_IOW` macro defines a command that passes a value to the driver (the command "writes" the argument to the driver). The `_IOWR` macro defines a command that is both read and written by the driver. The size of the argument is encoded in the command constant. This means that commands with the same group and number, but a different sized argument, will have different command constants. This is useful when implementing support for alternate userspace ABIs (for example, supporting a 32-bit userspace application on a 64-bit kernel) as the alternate ABIs will use a different command constant.

BSD kernels such as FreeBSD manage the copying of the I/O control command argument in the generic system call layer. This differs from Linux where the kernel passes the raw userspace pointer to the device driver, requiring the device driver to copy data to and from userspace. Instead, BSD kernels use the size argument encoded in the command constant to allocate an in-kernel buffer of the requested size. If the command was defined with `_IOW` or `_IOWR`, the buffer is initialized by copying the argument value in from the userspace application. If the command was defined with `_IOR`, the buffer is cleared with zeroes. After the device driver's ioctl routine completes, if the command was defined with `_IOR` or `_IOWR`, the buffer's contents are copied out to the userspace application.

For the echo driver, let's define three new control commands. The first command returns the current size of the global buffer. The second command permits setting a new size of the global buffer. The third command clears the contents of the buffer by resetting all the bytes to zero.

These commands are defined in a new echodev.h header shown in Listing 3. A header is used so that the constants can be shared with userspace applications as well as the driver. Note that the first command reads the buffer size into a size_t argument in userspace, the second command writes a new buffer size from a size_t argument in userspace, and the third command does not accept an argument. All three commands use the 'E' group and are assigned unique command numbers.

Listing 3: I/O Control Command Constants

```
#define     ECHODEV_GBUFSIZE   _IOR('E', 100, size_t)   /* get buffer size */
#define     ECHODEV_SBUFSIZE   _IOW('E', 101, size_t)   /* set buffer size */
#define     ECHODEV_CLEAR      _IO('E', 102)            /* clear buffer */
```

Supporting a dynamically sized buffer requires several driver changes. The global buffer is replaced with a global pointer to a dynamically allocated buffer, and a new global variable contains the buffer's current size. The pointer and length are initialized during module load, and the current buffer is freed during module unload. Since the buffer's size is no longer a constant, the checks for out-of-bounds reads and writes must now be done while holding the lock.

The in-kernel malloc(9) for FreeBSD requires an additional malloc type argument for both the allocation and free routines. Malloc types track allocation requests providing fine-grained statistics. These statistics are available via the -m flag to the vmstat(8) command which displays a separate line for each type. The kernel does include a general device buffer malloc type (`M_DEVBUF`) that drivers can use. However, it is best practice for drivers to define a dedicated malloc type. This is especially true for drivers in kernel modules. When a module is unloaded, malloc types defined in a kernel module are destroyed. If any allocations still reference those malloc types, the kernel emits a warning about the leaked allocations. The finer-grained statistics are also useful for debugging and performance analysis. New malloc types are defined via the `MALLOC_DEFINE` macro. The first argument provides the variable name of the new type. By convention, types are named in all uppercase and use a leading prefix of "M_". For this driver, we will use the name `M_ECHODEV`. The second argument is a short string name displayed by utilities such as vmstat(8). It is best practice to avoid whitespace characters in the short name. The third argument is a string description of the type.

Driver support for the custom control commands is implemented in the new function in Listing 4. The `cmd` argument contains the command constant for the requested command and the `data` argument points to the in-kernel buffer containing the optional command argument. The overall structure of the function is a switch statement on the `cmd` argument. The default error value for unknown commands is `ENOTTY`, even for non-tty devices. The two commands which accept a size argument cast `data` to the correct pointer type before dereferencing. The `ECHODEV_GBUFSIZE` command writes the current size to `*data`, while `ECHODEV_SBUFSIZE` reads the desired new size from `*data`.

For commands which alter the device state, the driver requires a writable file descriptor (that is, a file descriptor opened with `O_RDWR` or `O_WRONLY`). To enforce this, the `ECHODEF_SBUFSIZE` and `ECHODEV_CLEAR` commands require the `FWRITE` flag to be set in `fflag`. The `fflag` argument contains the file descriptor status flags defined in <sys/fcntl.h>. These flags map `O_RDONLY`, `O_WRONLY`, and `O_RDWR` to a combination of the `FREAD` and `FWRITE` flags. All other flags from open(2) are included directly in the file descriptor status flags. Note that a subset of these flags can be changed on an open file descriptor by fcntl(2).

Listing 4: I/O Control Handler

```
static int
echo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
      int error;

      switch (cmd) {
      case ECHODEV_GBUFSIZE:
            sx_slock(&echolock);
            *(size_t *)data = echolen;
            sx_sunlock(&echolock);
            error = 0;
            break;
      case ECHODEV_SBUFSIZE:
      {
            size_t new_len;

            if ((fflag & FWRITE) == 0) {
                  error = EPERM;
                  break;
            }

            new_len = *(size_t *)data;
            sx_xlock(&echolock);
            if (new_len == echolen) {
                  /* Nothing to do. */
            } else if (new_len < echolen) {
                  echolen = new_len;
            } else {
                  echobuf = reallocf(echobuf, new_len, M_ECHODEV,
                            M_WAITOK | M_ZERO);
                    echolen = new_len;
            }
            sx_xunlock(&echolock);
            error = 0;
            break;
    }
    case ECHODEV_CLEAR:
            if ((fflag & FWRITE) == 0) {
                    error = EPERM;
                    break;
            }

            sx_xlock(&echolock);
            memset(echobuf, 0, echolen);
            sx_xunlock(&echolock);
            error = 0;
            break;
    default:
            error = ENOTTY;
            break;
    }
    return (error);
}
```

To invoke these commands from userspace, we need a new user application. The repository contains an `echoctl` program used in Example 3. The size command outputs the current size of the buffer, the resize command sets a new buffer size, and the clear command clears the buffer contents. Note that in this example, the output from jot is no longer truncated. The last command in this example displays the dynamic allocation statistics for the driver's allocations using `M_ECHODEV`.

Example 3: Resizing the Global Buffer

```
# echoctl size
64
# echo foo > /dev/echo
# echoctl clear
# cat /dev/echo
# echoctl resize 80
# jot -c -s "" 70 48 > /dev/echo
# cat /dev/echo
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
# vmstat -m | egrep 'Type|echo'
           Type  Use Memory Req Size(s)
        echodev    1   128    2 64,128
```

Per-Instance Data

So far, our device driver has used global variables to hold its state. For a simple demonstration driver with a single device instance this is ok. However, most character devices are part of a hardware device driver and need to support multiple instances of a device within a single system. To support this, drivers define a structure containing the software context for a single device instance. In BSD kernels, this software context is named a "softc". Drivers typically define a structure type whose name uses a "_softc" suffix, and variables holding pointers to softc structures are usually named "sc".

Character devices provide straightforward support for per-instance data. `struct cdev` contains three members available for storing driver-specific data. `si_drv0` contains an integer value while `si_drv1` and `si_drv2` store arbitrary pointers. Device drivers are free to set these variables while creating character devices using the `mda_unit`, `mda_si_drv1`, and `mda_si_drv2` fields of `struct make_dev_args`. These values can then be accessed as members of the `struct cdev` argument to character device switch methods. Historically, device drivers used a unit number to track per-instance data. Modern device drivers in FreeBSD store a softc pointer in the `si_drv1` field and rarely use the other two fields.

Most character devices are part of a hardware device driver and need to support multiple instances of a device within a single system.

For our echo device driver, we define a `struct echodev_softc` type containing all of the state needed for an instance of the echo device. The device driver still stores a single global holding the softc of the single instance for use during module load and unload, but the rest of the driver accesses state via the softc pointer. These changes do not change any of the driver's functionality but do require refactoring various parts of the driver. Listing 5 shows the new softc structure type. Listing 6 demonstrates the type of refactoring needed for each character device switch method by showing the updated read method. Lastly, Listing 7 shows the updated routines used during module load and unload.

Listing 5: softc Structure

```
struct echodev_softc {
      struct cdev *dev;
      char *buf;
      size_t len;
      struct sx lock;
};
```

Listing 6: Driver Method Using softc Structure

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
      struct echodev_softc *sc = dev->si_drv1;
      size_t todo;
      int error;
```

```
        sx_slock(&sc->lock);
        if (uio->uio_offset >= sc->len) {
                error = 0;
        } else {
                todo = MIN(uio->uio_resid, sc->len - uio->uio_offset);
                error = uiomove(sc->buf + uio->uio_offset, todo, uio);
        }
        sx_sunlock(&sc->lock);
        return (error);
}
```

Listing 7: Module Load and Unload Using softc Structure

```
static int
echodev_create(struct echodev_softc **scp, size_t len)
{
        struct make_dev_args args;
        struct echodev_softc *sc;
        int error;

        sc = malloc(sizeof(*sc), M_ECHODEV, M_WAITOK | M_ZERO);
        sx_init(&sc->lock, "echo");
        sc->buf = malloc(len, M_ECHODEV, M_WAITOK | M_ZERO);
        sc->len = len;
        make_dev_args_init(&args);
        args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
        args.mda_devsw = &echo_cdevsw;
        args.mda_uid = UID_ROOT;
        args.mda_gid = GID_WHEEL;
        args.mda_mode = 0600;
        args.mda_si_drv1 = sc;
        error = make_dev_s(&args, &sc->dev, "echo");
        if (error != 0) {
                free(sc->buf, M_ECHODEV);
                sx_destroy(&sc->lock);
                free(sc, M_ECHODEV);
        }
        return (error);
}

static void
echodev_destroy(struct echodev_softc *sc)
{
        if (sc->dev != NULL)
                destroy_dev(sc->dev);
        free(sc->buf, M_ECHODEV);
        sx_destroy(&sc->lock);
        free(sc, M_ECHODEV);
}

static int
echodev_modevent(module_t mod, int type, void *data)
{
        static struct echodev_softc *echo_softc;

        switch (type) {
        case MOD_LOAD:
                return (echodev_create(&echo_softc, 64));
        case MOD_UNLOAD:
                if (echo_softc != NULL)
                        echodev_destroy(echo_softc);
                return (0);
        default:
                return (EOPNOTSUPP);
        }
}
```

Conclusion

Thanks for reading this far. The next article in this series will extend this driver to implement a FIFO buffer including support for non-blocking I/O and I/O event reporting via poll(2) and kevent(2).

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

Contents

Porting VPP to FreeBSD: Basic Usage

BY TOM JONES

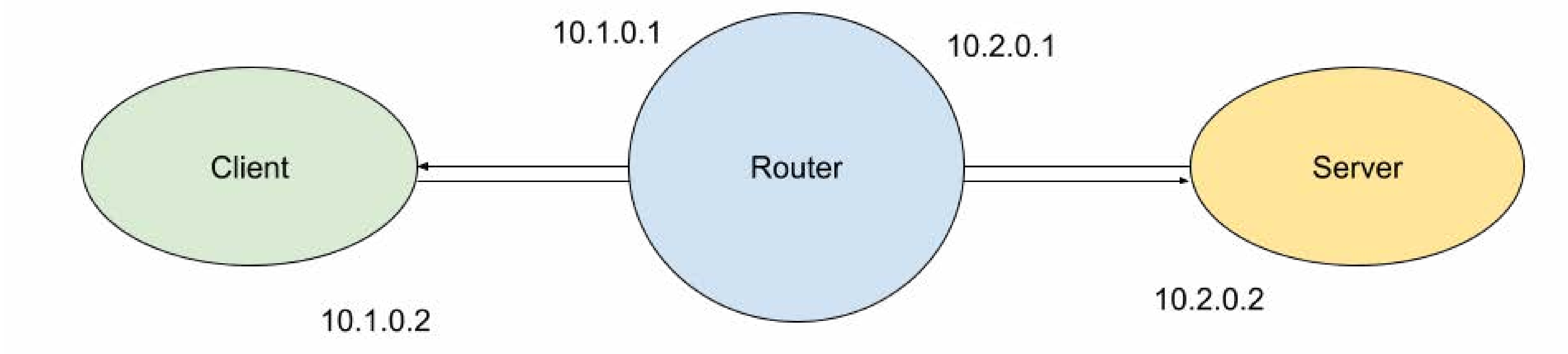
The Vector Packet Process (VPP) is a high-performance framework for processing packets in userspace. Thanks to a project by the FreeBSD Foundation and RGNets, I was sponsored to port VPP to FreeBSD and I am really happy to share some basic usage with readers of the *FreeBSD Journal*.

VPP enables forwarding and routing applications to be written in userspace with a API-controllable interface. High-performance networking is made possible by DPDK on Linux and DPDK and netmap on FreeBSD. These APIs allow direct 0 copy access to data and can be used to make forwarding applications that can significantly exceed the host's forwarding performance.

VPP is a full-network router replacement, and, as such, needs some host configuration to be usable. This article presents some complete examples of how to use VPP on FreeBSD which most users should be able to follow with a virtual machine of their own. VPP on FreeBSD also runs on real hardware.

This introduction to using VPP on FreeBSD gives an example set up showing how to do things on FreeBSD. VPP resources can be difficult to find, the documentation from the project at https://fd.io is high quality.

Lets Build a Router



VPP can be put to lots of purposes, the main one and easiest to configure is as some form of router or bridge. For our example of using VPP as a router, we need to construct a small example network with three nodes — a client, a server and the router.

To show you how VPP can be used on FreeBSD, I'm going to construct an example network with the minimum of overhead. All you need is VPP and a FreeBSD system. I'm also going to install iperf3 so we can generate and observe some traffic going through our router.

From a FreeBSD with a recent ports tree you can get our two required tools with the `pkg` command like so:

```
host # pkg install vpp iperf3
```

To create three nodes for our network, we are going to take advantage of one of FreeBSD's most powerful features, VNET jails. VNET jails give us completely isolated instances of the network stack, they are similar in operation to Linux Network Namespaces. To create a VNET, we need to add the **vnet** option when creating a jail and pass along the interfaces it will use.

Finally we will connect our nodes using **epair** interfaces. These offer the functionality of two ends of an ethernet cable — if you are familiar with **veth** interfaces on linux they offer similar functionality.

We can construct our test network with the following 5 commands:

```
host # ifconfig epair create
epair0a
host # ifconfig epair create
epair1a
# jail -c name=router persist vnet vnet.interface=epair0a vnet.interface=epair1a
# jail -c name=client persist vnet vnet.interface=epair0b
# jail -c name=server persist vnet vnet.interface=epair1b
```

The flags to take note of in these jail commands are **persist** without which the jail will be removed automatically because there are no processes running inside it, **vnet** which makes this jail a vnet jail and **vnet.interface=** which assigns the given interface to the jail.

When an interface is moved to a new vnet, all of its configuration is stripped away \-worth noting in case you configure an interface and then move it to a jail and wonder why nothing is working.

Set up peers

Before turning to VPP, let us set up the client and server sides of the network. Each of these needs to be given an ip address and the interface moved to the up state. We will also need to configure default routes for the client and server jails.

```
host # jexec client
# ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
        options=680003<RXCSUM,TXCSUM,LINKSTATE,RXCSUM_IPV6,TXCSUM_IPV6>
        groups: lo
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
epair0b: flags=1008842<BROADCAST,RUNNING,SIMPLEX,MULTICAST,LOWER_UP> metric 0
mtu 1500
        options=8<VLAN_MTU>
        ether 02:90:ed:bd:8b:0b
        groups: epair
        media: Ethernet 10Gbase-T (10Gbase-T <full-duplex>)
        status: active
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
# ifconfig epair0b inet 10.1.0.2/24 up
# route add default 10.1.0.1
add net default: gateway 10.1.0.1
```

```
host # jexec server
# ifconfig epair1b inet 10.2.0.2/24 up
# route add default 10.2.0.1
add net default: gateway 10.2.0.1
```

Our client and server jails now have ip addresses and routes towards the VPP router.

Netmap requirements

For our examples, we are going to use VPP with netmap, a high-performance userspace networking framework that ships as a default component of FreeBSD. Netmap requires a little interface configuration before it can be used — the interface needs to be in the `up` state and have the `promisc` option configured.

```
host # jexec router
# ifconfig epair0a promisc up
# ifconfig epair1a promisc up
```

Now we are able to start using VPP\!

VPP First Commands

VPP is very flexible and offers configuration by a config file, a command line interface, and an API with mature Python bindings. VPP needs a base configuration telling it where to get commands and the names of the files it uses for control if they aren't the default. We can give VPP a minimal configuration file on the command line as part of its arguments. For this example, we tell VPP to drop into interactive mode – offer us a cli, and we tell vpp to only load the plugins we will use (netmap) which is a sensible default.

If we don't disable all plugins, we will either need to set up the machine to use DPDK, or disable that plugin on its own. The syntax to do so is the same as the syntax to enable the netmap plugin.

```
host # vpp "unix { interactive} plugins { plugin default { disable } plugin
netmap_plugin.so { enable } plugin ping_plugin.so { enable } }"

       _______    _        _   _____  ___
    __/ __/ _ \  (_)__    | | / / _ \/ _ \
    _/ _// // / / / _ \   | |/ / ___/ ___/
    /_/ /____(_)_/\___/   |___/_/  /_/

vpp# show int
          Name               Idx       State  MTU (L3/IP4/IP6/MPLS)
Counter      Count
local0                        0        down          0/0/0/0
```

If all is set up, you will see the VPP banner and the default cli prompt (`vpp#`).

The VPP command line interface offers a lot of options for the creation and management of interfaces, groups like bridges, the addition of routes and tools for interrogating the performance of a VPP instance.

The syntax of the interface configuration commands is similar to the linux iproute2 commands – coming from FreeBSD these are a little alien, but they are reasonably clear once you start to get used to them.

Our VPP server hasn't been configured with any host interfaces yet, `show int` only lists the default `local0` interface.

To use our netmap interfaces with vpp, we need to create them first and then we can configure them.

The create command lets us create new interfaces, we use the netmap subcommand and the host interface.

```
vpp# create netmap name epair0a
netmap_create_if:164: mem 0x882800000
netmap-epair0a
vpp# create netmap name epair1a
netmap-epair1a
```

Each netmap interface is created with a prefix of **`netmap-`** . With the interfaces created, we can configure them for use and start using VPP as a router.

```
vpp# set int ip addr netmap-epair0a 10.1.0.1/24
vpp# set int ip addr netmap-epair1a 10.2.0.1/24
vpp# show int addr
local0 (dn):
netmap-epair0a (dn):
  L3 10.1.0.1/24
netmap-epair1a (dn):
  L3 10.2.0.1/24
```

The command **`show int addr`** (the shortened version of **`show interface address`**) confirms our ip address assignment has worked. We can then bring the interfaces up:

```
vpp# set int state netmap-epair0a up
vpp# set int state netmap-epair1a up
vpp# show int
          Name                Idx State  MTU (L3/IP4/IP6/MPLS)
Counter   Count
local0                          0   down         0/0/0/0
netmap-epair0a                  1   up           9000/0/0/0
netmap-epair1a                  2   up           9000/0/0/0
```

With our interfaces configured, we can test functionality from VPP by using the ping command:

```
vpp# ping 10.1.0.2
116 bytes from 10.1.0.2: icmp_seq=2 ttl=64 time=7.9886 ms
116 bytes from 10.1.0.2: icmp_seq=3 ttl=64 time=10.9956 ms
116 bytes from 10.1.0.2: icmp_seq=4 ttl=64 time=2.6855 ms
116 bytes from 10.1.0.2: icmp_seq=5 ttl=64 time=7.6332 ms

Statistics: 5 sent, 4 received, 20% packet loss
vpp# ping 10.2.0.2
116 bytes from 10.2.0.2: icmp_seq=2 ttl=64 time=5.3665 ms
116 bytes from 10.2.0.2: icmp_seq=3 ttl=64 time=8.6759 ms
116 bytes from 10.2.0.2: icmp_seq=4 ttl=64 time=11.3806 ms
```

```
116 bytes from 10.2.0.2: icmp_seq=5 ttl=64 time=1.5466 ms

Statistics: 5 sent, 4 received, 20% packet loss
```

And if we jump to the client jail, we can verify that VPP is acting as a router:

```
client # ping 10.2.0.2
PING 10.2.0.2 (10.2.0.2): 56 data bytes
64 bytes from 10.2.0.2: icmp_seq=0 ttl=63 time=0.445 ms
64 bytes from 10.2.0.2: icmp_seq=1 ttl=63 time=0.457 ms
64 bytes from 10.2.0.2: icmp_seq=2 ttl=63 time=0.905 ms
^C
--- 10.2.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.445/0.602/0.905/0.214 ms
```

As a final piece of initial set up, we will start up an iperf3 server in the server jail and use the client to do a TCP throughput test.:

```
server # iperf3 -s

client # iperf3 -c 10.2.0.2
Connecting to host 10.2.0.2, port 5201
[  5] local 10.1.0.2 port 63847 connected to 10.2.0.2 port 5201
[ ID]   Interval         Transfer    Bitrate         Retr  Cwnd
[  5]   0.00-1.01   sec  341 MBytes  2.84 Gbits/sec  0     1001 KBytes
[  5]   1.01-2.01   sec  488 MBytes  4.07 Gbits/sec  0     1.02 MBytes
[  5]   2.01-3.01   sec  466 MBytes  3.94 Gbits/sec  144   612 KBytes
[  5]   3.01-4.07   sec  475 MBytes  3.76 Gbits/sec  0     829 KBytes
[  5]   4.07-5.06   sec  452 MBytes  3.81 Gbits/sec  0     911 KBytes
[  5]   5.06-6.03   sec  456 MBytes  3.96 Gbits/sec  0     911 KBytes
[  5]   6.03-7.01   sec  415 MBytes  3.54 Gbits/sec  0     911 KBytes
[  5]   7.01-8.07   sec  239 MBytes  1.89 Gbits/sec  201   259 KBytes
[  5]   8.07-9.07   sec  326 MBytes  2.75 Gbits/sec  0     462 KBytes
[  5]   9.07-10.06  sec  417 MBytes  3.51 Gbits/sec  0     667 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID]   Interval        Transfer     Bitrate         Retr
[  5]   0.00-10.06  sec  3.98 GBytes  3.40 Gbits/sec  345          sender
[  5]   0.00-10.06  sec  3.98 GBytes  3.40 Gbits/sec               receiver

iperf Done.
```

VPP Analysis

Now that we have sent some traffic through VPP, the output of `show int` contains more information:

```
vpp# show int
     Name               Idx    State  MTU (L3/IP4/IP6/MPLS)       Counter          Count
local0                   0     down        0/0/0/0
netmap-epair0a           1     up          9000/0/0/0       rx packets            4006606
                                                            rx bytes           6065742126
                                                            tx packets            2004365
                                                            tx bytes            132304811
                                                            drops                       2
                                                            ip4                   4006605
netmap-epair1a           2     up          9000/0/0/0       rx packets            2004365
                                                            rx bytes            132304811
                                                            tx packets            4006606
                                                            tx bytes           6065742126
                                                            drops                       2
                                                            ip4                   2004364
```

The interface command now gives us a summary of the bytes and packets that have passed across the VPP interfaces. This can be really helpful to debug how traffic is moving around, especially if your packets are going missing.

The V in VPP stands for vector and this has two meanings in the project. VPP aims to use vectorised instructions to accelerate packet processing and it also bundles groups of packets together into vectors to optimize processing. The theory here is to take groups of packets through the processing graph together saving cache thrashing and giving optimal performance.

VPP has a lot of tooling for interrogating what is happening while packets are processed. Deep tuning is beyond this article, but a first tool to look at to understand what is happening in VPP is the **runtime** command.

Runtime data is gathered for each vector as it passes through the VPP processing graph, it collects how long it takes to transverse each node and the number of vectors processed.

To use the run time tooling, it is good to have some traffic. Start a long running iperf3 throughput test like so:

```
client # iperf3 -c 10.2.0.2 -t 1000
```

Now in the VPP jail, we can clear the gathered run time statistics so far, wait a little bit and then look at how we are doing:

```
vpp# clear runtime
... wait ~5 seconds ...
vpp# show runtime
Time 5.1, 10 sec internal node vector rate 124.30 loops/sec 108211.07
  vector rates in 4.4385e5, out 4.4385e5, drop 0.0000e0, punt 0.0000e0
          Name                 State         Calls        Vectors       Suspends     Clocks     Vectors/Call
ethernet-input                 active        18478        2265684       0            3.03e1     122.62
fib-walk                       any wait          0        0             3            1.14e4     0.00
ip4-full-reassembly-expire-wal any wait          0        0             102          7.63e3     0.00
ip4-input                      active        18478        2265684       0            3.07e1     122.62
ip4-lookup                     active        18478        2265 684      0            3.22e1     122.62
ip4-rewrite                    active        18478        2265684       0            3.05e1     122.62
ip6-full-reassembly-expire-wal any wait          0        0             102          5.79e3     0.00
ip6-mld-process                any wait          0        0             5            6.12e3     0.00
```

```
ip6-ra-process                  any wait         0         0           5          1.18e4    0.00
netmap-epair0a-output           active        8383         755477      0          1.12e1    90.12
netmap-epair0a-tx               active        8383         755477      0          1.17e3    90.12
netmap-epair1a-output           active       12473         1510207     0          1.04e1    121.08
netmap-epair1a-tx               active       12473         1510207     0          2.11e3    121.08
netmap-input                    interrupt wa  16698        2265684     0          4.75e2    135.69
unix-cli-process-0              active           0         0           13         7.34e4    0.00
unix-epoll-input                polling      478752        0           0          2.98e4    0.00
```

The columns in the **`show runtime`** output give us a great idea of what is happening in vpp. They tell us which nodes have been active since the run time counters were cleared, their current state, how many times this node was called, how much time it used, and how many vectors were processed per call. Out of the box, the maximum vector size for vpp is 255.

A final debugging task you can perform is to examine the packet processing graph in its entirety with the **`show vlib graph`** command. This command shows each node and the potential parent and child nodes which could lead to it.

Next Steps

VPP is an incredible piece of software — once the headaches of compatibility were addressed, the core parts of VPP were reasonably straightforward to port. Even with just minimal tuning, VPP is able to reach some impressive performance with netmap on FreeBSD, and it does even better if you configure DPDK. The VPP documentation is slowly getting more information about running on FreeBSD, but the developers really need example use cases of VPP on FreeBSD.

If you start from this example of a simple network, it should be reasonably straight forward to port it onto a large network with faster interfaces.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.

Contents

Enhancing FreeBSD Test Suite Parallelism with Kyua's Jail Feature

BY IGOR OSTAPENKO

Testing is a rather broad concept today. Regardless of approaches, academic views, or specific situations, it is difficult to resist the simple desire not to force the end user to test on our behalf. Even a simple error can significantly harm the business in multiple ways: reputation, time-to-market, conversion rate, expansion, etc. The industry has evolved from relying on complex manual checks before release to embracing automation whenever and wherever possible. Automated testing offers its benefits: shorter cycles, more frequent feedback, additional confidence in the result, less fear of changes, and so on. Even if automated testing is a significant part of the development process, it is just one of many practices improving software delivery.

It is difficult to resist the simple desire not to force the end user to test on our behalf.

Organizing Tests with Kyua

The FreeBSD Test Suite is typically located in `/usr/tests`, with its infrastructure built on Kyua, a testing framework created by Julio Merino. Kyua offers an expressive test suite definition language, a safe runtime engine, and powerful report generation. Tests can be written using anything or without relying on a specific library. The unit and integration tests found in the suite often utilize libraries like `atf-c(3)`, `atf-c++(3)`, `atf-sh(3)`, or `pytest`.

Kyua operates with the following hierarchy of key concepts: test suite > test program > test case.

A test suite groups several binaries (test programs) into one set with a single name. A Lua script is used to describe a test suite, which is typically saved in a special kyuafile(5). Let's consider the following existing file:

```
# cat -n /usr/tests/sys/kern/Kyuafile
       1  -- Automatically generated by bsd.test.mk.
       2
       3  syntax(2)
       4
       5  test_suite("FreeBSD")
       6
       7   atf_test_program{name="basic_signal", }
```

```
[skipped]
       34  atf_test_program{name="sonewconn_overflow", required_programs="python",
required_user="root", is_exclusive="true"}
       35  atf_test_program{name="subr_physmem_test", }
       36  plain_test_program{name="subr_unit_test", }
[skipped]
       45  atf_test_program{name="unix_seqpacket_test", timeout="15"}
       46  atf_test_program{name="unix_stream", }
       47  atf_test_program{name="waitpid_nohang", }
       48  include("acct/Kyuafile")
       49  include("execve/Kyuafile")
       50  include("pipe/Kyuafile")
```

Line #3 specifies the required version of the syntax used. Line #5 sets the name for the test suite. Usually, all **`/usr/tests/**/Kyuafile`** descriptions are collected into a single test suite named FreeBSD. If a binary is based on ATF libraries, it's registered using **`atf_test_program`**, so Kyua can leverage ATF capabilities and specifics for such a test program. If it's not based on a library supported by Kyua and simply communicates results via exit code, **`plain_test_program`** construct is used instead. There is also **`tap_test_program`** for test programs that communicate results using the good old Test Anything Protocol.

Each Kyuafile describes test binaries only within its directory. However, **`/usr/tests`** are structured the way that each test directory explicitly includes its subdirectories, as illustrated in lines #48, #49, and #50. Consequently, running tests in this directory will execute all tests from the **`sys/kern`** sub-tree, including those in **`sys/kern/acct`**, **`sys/kern/execve`**, and **`sys/kern/pipe`**:

```
# kyua test -k /usr/tests/sys/kern/Kyuafile
```

Line #1 indicates that Kyuafile is not created manually in the FreeBSD Test Suite. Instead, as with most components of the FreeBSD build system, the process is handled through a Makefile, which builds the test programs and generates the corresponding Kyuafile. To understand the process in detail, the generated **`/usr/tests/sys/kern/Kyuafile`** can be compared with its source in /**`usr/src/tests/sys/kern/Makefile`** — it's a straightforward approach.

A test program that is not registered in the Kyuafile will not be recognized by Kyua and therefore will not be executed.

There is no explicit mention of test cases in the Kyuafile because test cases are defined at a lower level, within a test program, and it requires support from the library used. It's often more convenient to group several similar tests (i.e. test cases) within a single test program binary rather than creating multiple separate test programs. Plain test programs are expected to provide only one test case, typically named "main" after the **`main()`** function. In contrast, tests based on ATF libraries can report multiple test cases. In Kyua, what we generally call a test is referred to as a test case, which is treated as a unit of execution. Consequently, a test suite described in a Kyuafile might seem to reference only a single test program, but it could contain dozens or more test cases. The **`kyua list`** command lists test cases in the format **`<test program>:<test case>`**, which can also be used with other commands, for example, to run a specific test case individually:

```
# cd /usr/tests/sys/kern
# kyua test unix_dgram:basic
```

Each test case can have optional metadata properties in the form of key/value pairs, which modify Kyua's behavior for that specific test case. The Kyuafile example above shows that the same metadata can be applied to all test cases within a test program. The illustrated properties are:

- `timeout` allows changing the default value, which is 300 seconds.
- If the binaries specified in `required_programs` are not found either by their full path or within the `PATH`, the test case will be marked as skipped with the respective message.
- `required_user="root"` will skip the test if Kyua is not running with root privileges, while `required_user="unprivileged"` will ensure that the test is run without root access rights.
- `is_exclusive="true"` specifies that the test cannot be run concurrently with other tests.

Parallelism and Jails

Kyua can run test cases in parallel when configured to do so. By default, the `parallelism` setting is set to 1, which means tests are run sequentially. This can be adjusted in `kyua.conf(5)` or specified as an option:

```
# kyua -v parallelism=8 test
```

Test cases that require exclusive access to a shared resource should be marked with `is_exclusive="true"` so Kyua knows not to run them in parallel with others. Kyua operates in two phases. First, it runs all non-exclusive test cases, which can be executed in parallel if configured so. The second phase runs all exclusive test cases sequentially. To keep the test suite efficient, it's best to avoid adding new exclusive tests when possible and create non-exclusive versions instead. Otherwise, the test suite may become too time-consuming to execute.

However, some tests utilize the `jail(8)` feature to handle scenarios that are otherwise difficult to reproduce. For example, network module testing often involves creating temporary jails to verify module behavior by poking it from the host through `epair(4)`. There are several reasons why such tests must be marked as exclusive: they usually re-use the same jail naming for convenience (while each jail in the system must have a unique name), the same IP addresses officially allocated for demo purposes are used to configure interfaces on the host side leading to potential conflicts with a shared routing table, and other related issues. While these problems could be addressed by test cases themselves, doing so would significantly increase the complexity for test authors and maintainers, and some issues might be impossible to resolve without external intervention. This is where the latest Kyua version comes into play.

To keep the test suite efficient, it's best to avoid adding new exclusive tests.

Execution Environment Concept

In 15-CURRENT Kyua provides a new concept called "execution environment". It's going to be available from 14.2-RELEASE.

By default, tests continue to run as before by spawning child processes — this is referred to as the host execution environment. A test case can opt-in to use a different execution environment by specifying a new metadata property called **`execenv`**. The general sequence of steps applied to each test case has been extended to include the following:

1. Execution environment initialization
2. Test execution
3. (optional) Test cleanup
4. Execution environment cleanup

Currently, Kyua supports only one additional execution environment — the jail environment. While it can be configured for individual test cases, the following example shows how to apply the execenv metadata property to all test cases within a test program:

```
atf_test_program{name="test_program", execenv="jail"}
```

This configuration causes Kyua to provide each test case in the **`test_program`** with its own temporary jail in which to execute. If a test case declares a cleanup routine, it will be executed within the same jail. Kyua uses `jail(8)` for creating these jails, and test cases can pass additional parameters through a new metadata property called **`execenv_jail_params`**:

```
atf_test_program{name="test_program", execenv="jail", execenv_jail_params="vnet allow.raw_
sockets"}
```

As long as the names of sub-jails do not conflict among different parent jails, and each jail can have its own VNET stack, we can easily isolate tests — such as the network tests mentioned earlier — into separate jails and run them in parallel by removing the **`is_exclusive`** flag. It depends on environment and configuration, but there are reports that **`netpfil/pf`** test suite runs 4 or 5 times faster using the same environment — taking just a few minutes instead of half an hour.

Implicit Parameters and Hierarchical Jails

Since a test case and its optional cleanup routine run in separate child processes, Kyua implicitly appends the **`persist`** parameter to keep the temporary jail alive, allowing both child processes to run within the same jail. Kyua ensures that the temporary jail is removed during the "Execution environment cleanup" step.

Spawning jails by network tests is a common practice. This raises the question of whether a test case, which is already running inside a jail, is permitted to create sub-jails. In principle, this is allowed as long as system limits are not exceeded. Each jail has a limit on the number of sub-jails that can be created. The following new read-only sysctl variables, introduced in 15-CURRENT, provide this information:

```
# sysctl security.jail.children
security.jail.children.cur: 0
security.jail.children.max: 999999
```

Apparently, the above looks to be the highest jail in the hierarchy, so called **`prison0`**, and almost a million of jails can be created according to the current and maximum values. When jail(8) is used to create a new jail, it applies the following default configuration:

```
# jail -c command=sysctl security.jail.children
security.jail.children.cur: 0
security.jail.children.max: 0
```

This indicates that no sub-jails are allowed. Obviously, test cases attempting to create new jails would fail under these conditions. To address this, Kyua assists by adding another implicit parameter that allows the maximum number of child jails, calculated as the parent jail's maximum limit minus one. Although it's possible to configure this from the test case side using the `execenv_jail_params` metadata property, it appears to be cumbersome and repetitive work.

The following formula clarifies how Kyua creates temporary jails and how this process can be modified using metadata properties:

```
jail -qc name=<name> children.max=<parent_max-1> <test case defined params> persist
```

The name of a temporary jail is derived from the test program path and test case name. For example, the test case `/usr/tests/sys/kern/unix_dgram:basic` will use a temporary jail named `kyua_usr_tests_sys_kern_unix_dgram_basic`.

kldload concerns

Since all jails, except for `prison0`, lack the privilege to load kernel modules, this creates inconveniences if a test case relies on the jail execution environment.

Kyua's original philosophy is to be usable by both developers and users. This means that a system administrator should be able to run the test suite after OS upgrade to ensure everything is functioning as expected. Clearly, such a host is not a test lab where developers can freely experiment, break things, or cause fire. Therefore, tests should be designed to avoid disrupting the normal operation of the host unless explicitly instructed. That's why the FreeBSD Test Suite has configuration variables like `allow_sysctl_side_effects` to follow this approach. Despite the fact that the suite is primarily treated as a developer tool, many existing tests adhere to this principle by checking if a required module is loaded, rather than loading it implicitly. A system administrator would not appreciate it if, for example, tests of a firewall, which is not used by a host, inadvertently affect its traffic or even make it inaccessible.

Therefore, the recommended strategy is to use `kldstat -q -m <module-name>` within the test case to check for the presence of required modules and skip the test if the module is not found. The configuration of the FreeBSD CI ensures that all necessary modules are loaded and required software packages are installed before running the test suite.

execenvs and WITHOUT_JAIL

A new engine configuration variable is provided — `execenvs`. By default, it is set to a list of all supported execution environments:

```
# kyua config
architecture = aarch64
execenvs = host jail
parallelism = 1
platform = arm64
unprivileged_user = tests
```

This variable can be manipulated through kyua.conf(5) or specified as an option for the kyua(1) CLI. For instance, the following command will execute only host-based tests and skip all others:

```
# kyua -v execenvs=host test
```

If the system is built without jail support, only the default host execution environment will be available. Consequently, any tests that require the jail execution environment will be skipped.

Getting Started Examples

The following example, based on atf-sh(3), illustrates how to configure the jail environment at the test case level. It also reminds the importance of root user privileges.

```
# cat /usr/src/tests/sys/kern/test_program.sh
atf_test_case "case1" "cleanup"
case1_head()
{
       atf_set descr 'Test that X does Y'
       atf_set require.user root
       atf_set execenv jail
       atf_set execenv.jail.params vnet allow.raw_sockets
}
case1_body()
{
       if ! kldstat -q -m tesseract; then
               atf_skip "This test requires tesseract"
       fi

       # test code...
}
case1_cleanup()
{
       # cleanup code...
}

atf_init_test_cases()
{
       atf_add_test_case "case1"
}
```

A single line addition to the Makefile is enough for this test program:

```
# grep test_program /usr/src/tests/sys/kern/Makefile
ATF_TESTS_SH+= test_program
```

The build system will prepend the `#!/usr/libexec/atf-sh` shebang line, install the script without the `.sh` extension at `/usr/tests/sys/kern/test_program`, and register it in the `Kyuafile` accordingly:

```
# grep test_program /usr/tests/sys/kern/Kyuafile
atf_test_program{name="test_program", }
```

Having multiple test cases within a single test program can lead to a Don't Repeat Yourself situation. To handle this, common metadata can be moved up to the test suite level in a Kyuafile, allowing it to apply to the entire test program rather than repeating it for each test case. However, individual test cases can still override these properties if necessary:

```
# cat /usr/src/tests/sys/kern/test_program2.sh
atf_test_case "case2"
case2_head()
{
       atf_set descr 'Test that A does B'
}
case2_body()
{...}

atf_test_case "case3"
case3_head()
{
       atf_set descr 'Test that Foo does Bar'
       atf_set execenv.jail.params vnet allow.raw_sockets
}
case3_body()
{...}

atf_init_test_cases()
{
       atf_add_test_case "case2"
       atf_add_test_case "case3"
}
```

Now the main configuration is provided on the test program level:

```
# grep test_program2 /usr/src/tests/sys/kern/Makefile
ATF_TESTS_SH+= test_program2
TEST_METADATA.test_program2+= execenv="jail",execenv_jail_params="vnet"
```

As a result, Kyua consolidates metadata defined at different levels into the following:

```
# kyua list -k /usr/tests/sys/kern/Kyuafile -v test_program2
test_program2:case2 (FreeBSD)
    description = Test that A does B
    execenv = jail
    execenv_jail_params = vnet
test_program2:case3 (FreeBSD)
    description = Test that Foo does Bar
    execenv = jail
    execenv_jail_params = vnet allow.raw_sockets
```

It's important to note the key difference in metadata property naming conventions between ATF and Kyua — dots (`execenv.jail.params`) versus underscores (`execenv_jail_params`). Additionally, names themselves may vary slightly, the `kyuafile(5)` and `atf-test-case(4)` manual pages can be compared for that.

To switch an existing test to the jail execution environment, the `is_exclusive="true"` metadata property should be negated or removed. Otherwise, the test will not benefit from parallel execution.

Further Reading

The entry point to the FreeBSD Test Suite is described in tests(7). For test authors, the following wiki page is a valuable starting point: https://wiki.freebsd.org/TestSuite/DeveloperHowTo.

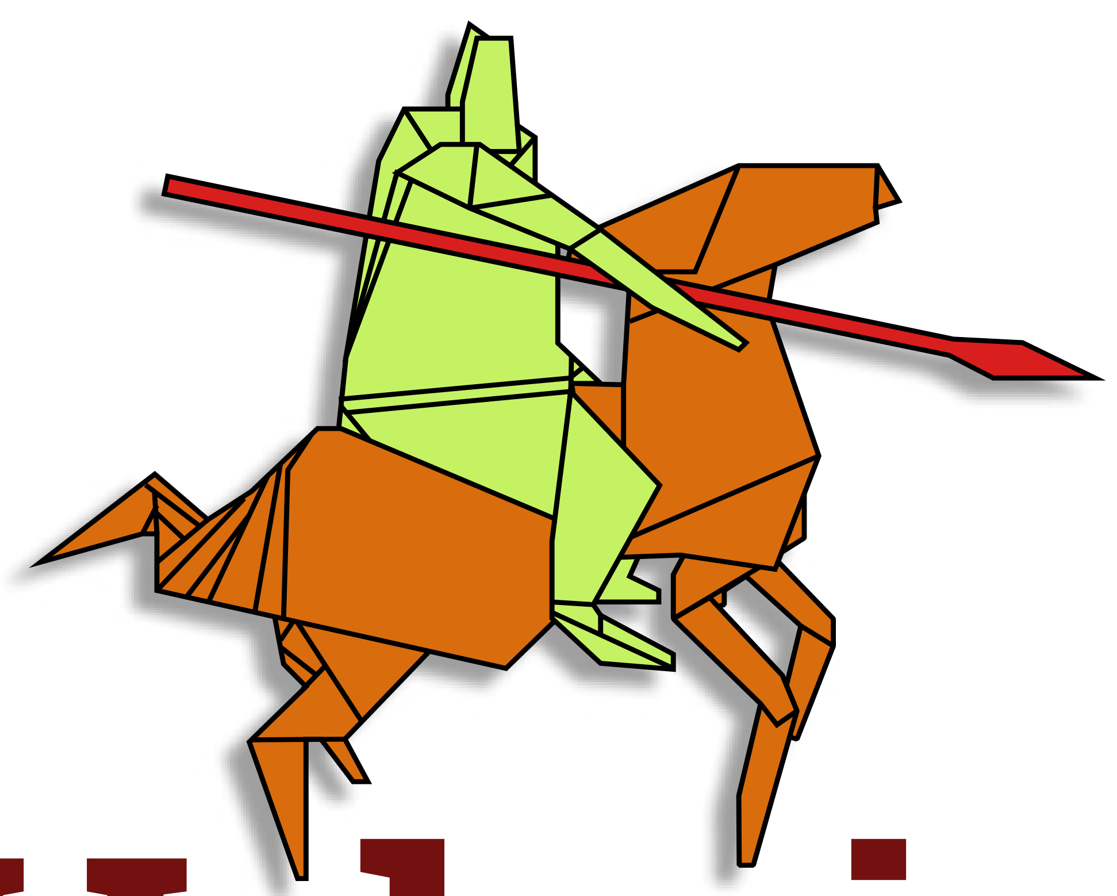
The official Kyua wiki is an excellent resource for historical aspects, design rationale, and feature overviews. Detailed information on execution environments can be found in the kyua.conf(5) and kyuafile(5) manual pages.

Also, reviewing how existing jail-based tests are written and organized is crucial to avoid reinventing the wheel. The PF test suite located in `/usr/src/tests/sys/netpfil/pf` is a great source for understanding of the established practices.

While retroactively adding tests to the existing code can be an enormous effort, incorporating tests that address bug fixes is a worthwhile opportunity to enhance the FreeBSD Test Suite, and therefore the project as a whole.

IGOR OSTAPENKO is a FreeBSD contributor with a wide range of software development experience in various areas, whether it's systems for manipulating and testing navigation devices, enterprise solutions for optimizing business processes, reverse-engineering, or B2B/B2C startups.

Contents



Valgrind on FreeBSD

BY PAUL FLOYD

I first started using Valgrind in the early 2000s. Previously, I had a fair bit of experience with Purify (now Unicom PuifyPlus) on Solaris/SPARC. To be honest, I wasn't that impressed with Valgrind. Sure, it didn't need a special build process, but it lacked the ability to interact with a debugger.

Switching briefly to FreeBSD, the first version I installed was 2.1 back in late 1995. Like Valgrind, at first, I was not that impressed. At least it was "a unix" on my home PC if I needed it. I continued to dabble with FreeBSD, installing new versions from time to time. My main home system(s) were OS/2 till the late 90s, Solaris for long time until it went into suspended animation with 11.4. I also got a MacBook in 2007 which is mainly for "desktop" stuff — I don't find developing on macOS to be a gratifying experience.

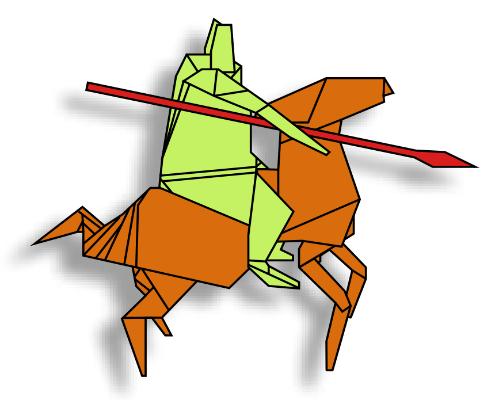
I have always been a bit of a believer in Quality. I had a short course on Product Quality at university which converted me to the cause. Much later, I got round to reading W. Edwards Deming. Though the writing is rough around the edges, the message is strong and clear. Since I'd studied Electronics, it was plainly obvious the benefits from quality processes that Japanese companies had reaped in the second half of the 20th century. I ended up working as a software developer in the domain of Electronics simulation. Not surprisingly, I carried on using tools like Valgrind.

I decided that it was time to start giving back to the open-source community.

About five years ago, I decided that it was time to start giving back to the open-source community. Since I was already expert in using Valgrind and had already dabbled a tiny bit with the source, it was the logical project for me. I hesitated a bit between working on the macOS and FreeBSD Valgrind ports. Two things put me off macOS — frequent major OS and userland changes that break everything, and the difficulty of getting help from within Apple. There are the XNU code source dumps and a few books, but after that you are on your own. I plumped for FreeBSD. That also suited me because I was looking to switch away from Solaris. There's been a lot of cross-fertilization between Illumos and FreeBSD so I thought that would ease the transition. In the meantime, macOS lingers in the official Valgrind repo, but it hasn't really been usable since version 10.12 in 2016.

History of Valgrind

Valgrind is now a bit over 20 years old. It started off on i386 Linux. Over the years, several other CPU architectures have been added (amd64, MIPS, ARM, PPC and s390x) as well as other operating systems (macOS, Solaris, and, most recently, FreeBSD).



The tools have continued to evolve over those 20 years. Since the initial version in 2002, the tools added were

2002 memcheck
2002 helgrind
2002 cachegrind
2004 massif
2006 callgrind
2008 drd
2009 exp-bbv
2018 DHAT

In addition, there are several tools that are maintained (or not) out-of-tree.

The development of Valgrind has been carried out by a small number of people - about twenty have made significant contributions. A few corporations have lent a hand. RedHat/IBM is probably the one that has contributed the most. Sun did contribute while Solaris was being actively developed. Apple also contributed until they suddenly became GLP averse.

History of Valgrind on FreeBSD

Valgrind on FreeBSD had a very long and checkered history. I won't mention everyone who has contributed (and I'm not even sure that I have the full list as some of the source code repos are no longer accessible). Doug Robson did a lot of the initial work in 2004. The next torch bearer was Stan Sedov who maintained the port from 2009 to 2011. There was a protracted push to get the FreeBSD source accepted upstream at that time, but it didn't quite make it. The upstream maintainers were quite strict with their quality bar, and the FreeBSD port kept getting close, but was never good enough. Secondly, someone needs to maintain the port, preferably a member of the Valgrind team. I don't know why that never happened. I've been maintaining the FreeBSD port since Apr 2021, and I've had a Valgrind commit bit for a bit over 4 years. Now, I'm the main contributor to Valgrind.

Valgrind on FreeBSD had a very long and checkered history

The most recent big change was adding support for aarch64. I added the port to this CPU in April 2024, in time for the 3.23 release of Valgrind.

The Valgrind Tools

Before I dive into the internals of Valgrind, I'll give a quick overview of the tools.

Memcheck

This is the tool that most people think of when they refer to Valgrind. It is the default tool. The main things memcheck does are validate that memory reads are from initialized memory and that reads and writes are within the bounds of blocks of allocated heap memory. The missing piece there is checking the bounds of stack memory — that requires instrumentation.

DRD and Helgrind

These two tools are both thread hazard detection tools. They will detect accesses to memory from different threads that do not use some sort of locking mechanism. They will also warn of errors in the use of the pthread functions. The difference between the two is that Helgrind will try to give the error context for all the threads involved with a hazard. DRD only gives details for one thread.

Callgrind and Cachegrind

These two tools are for CPU profiling. Callgrind profiles function calls. Cachegrind is historically used to profile CPU instructions with a basic cache and branch predictor model. These models were never very accurate and now they are quite unrealistic. On top of that, Valgrind does not do any speculative execution. For those reasons, the current version of Valgrind no longer uses cache simulation with Cachegrind by default. Some people like the precise nature of the instruction counts. Personally, I usually prefer sampling profilers like Google perftools, (port devel/google-perftools), Linux perf and gprofng, especially for large problems (runtimes in hours or days and memory use in the 100s of Gbytes).

Massif and DHAT

These two tools are memory profiling tools. Massif profiles memory over time. Personally, I find it is overkill. Other tools exist that can usually produce equally good profiles without the Valgrind overhead — Google perftools again, and HeapTrack (port devel/heaptrack). There is an exception to this. If your application makes heavy use of a custom allocator based on mmap or statically links with a malloc library, then those alternative tools won't work. Massif doesn't need to interpose allocation functions in a shared library, and it also has an option to profile memory at the mmap level. DHAT is the hidden gem in the Valgrind suite. This tool profiles memory accesses to heap memory. This gives you information that will allow you to see which bits of memory are heavily used, memory that remains allocated for a long time, memory that is never used. For memory blocks that aren't too large, it will also generate access histograms for that block. From that, you can see holes or unused members in structs and classes. You can also infer access patterns which might help in reordering members to get them on the same cache line.

If your application makes heavy use of a custom allocator based on mmap or statically links with a malloc library, then those alternative tools won't work.

Valgrind Basics

Non-dependencies

In order to allow Valgrind to execute all of the client code (not just from main(), but from the first instructions in the ELF file at program startup) and also to avoid any conflicts with things like stdio buffers, Valgrind does not link with libc or any external libraries. I sometimes joke that this is not so much C++ as C--. That means Valgrind has its own implementation of a subset of libc. To keep the function names distinct, it uses macros as a kind of pseudo-namespace. The Valgrind version of *printf* is *VG_(printf)* (great fun for code navigation!). This also means that we can't just add a third-party library and use it. The library needs to be ported to use Valgrind's libc subset. An example is that there is currently a bugzilla item to add support for zstd compressed DWARF sections.

Paranoid programming

Valgrind is very cautious and makes extensive use of asserts that are enabled in release builds. That makes it a little slower, but there are just so many things that can go wrong. It's best to be honest and bomb straight away rather than try to fake it and limp on.

Valgrind has extensive verbose and debug messages. You can crank up the debug/verbosity levels by repeating -v and -d up to 4 times each. In addition to that there are several more targeted trace options like —trace-syscalls=yes. Debugging in Valgrind can be quite difficult and all these outputs can be a big aid when developing features. They are also useful for support, e.g., asking the user to upload logs to the Valgrind bugzilla.

Code complexity

Valgrind itself is a bit of a beast. One of the hardest things about working on Valgrind is that it touches on so many things. There is virtualization for four families of CPUS (Intel/AMD, ARM, MIPS and PPC with a few sub-variants). Each of those has multi-thousand-page manuals. You often need to know all about opcodes to the level of every bit that they might change. You need a good knowledge of C, C++, and POSIX. You need to be able to tell which OS syscalls need special handling. Knowing the ELF standard is important - we've had issues because lld and mold do things differently. As well as ELF there is DWARF for the debuginfo. So far, I've only covered the core of Valgrind.

Despite the complexity, I don't think that Valgrind contains a huge amount of code

Despite the complexity, I don't think that Valgrind contains a huge amount of code. A clean git clone, not counting the regression tests, is about 500kloc. With the regression tests, that goes up to about 750kloc — while there are only 1000 or so regession tests, some of them are enormous, covering vast numbers of combinations of bit patterns, using scripts to generate all combinations of inputs to test.

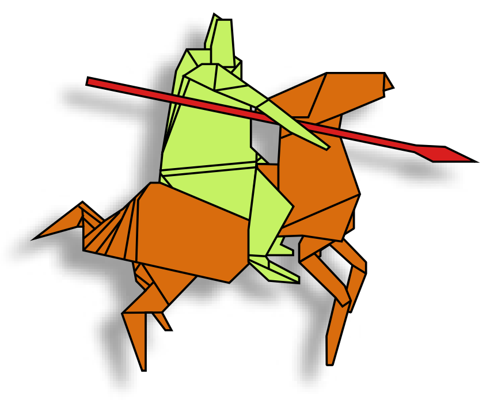
The tools themself take up barely 10% of the code. It's the CPU emulation and the "core" that dominate. The core consists of many things — libc replacement, syscall wrappers, memory management, gdb interface, DWARF reader, signal handling, internal data structures and function redirections.

One further complication when developing Valgrind is that, being entirely static, you can't build it with sanitizers. However, you can run Valgrind inside Valgrind! This requires a special build so you end up with an outer Valgrind and inner Valgrind which is the guest of the outer Valgrind, and a guest executable, guest of the inner Valgrind. Of course, that makes everything slower to another degree. I do use the free Coverity Scan service to run static analysis on FreeBSD builds of Valgrind. That mostly finds the usual kinds of false positives but has found a few real bugs including some that I added. I still need to do some work to provide code models for Valgrind's internal libc replacements, particularly the allocation functions.

Valgrind at Runtime

Guest execution

The CPU emulation in Valgrind is called VEX (not to be confused with Intel Vector EXtensions). I'm not sure of the origins of VEX, possibly "Valgrind Emulation."



When Valgrind runs, there is just one process — the host. Ptrace (as used by debuggers such as lldb and gdb) is not used. The guest (sometimes referred to as the client) executable runs within the host using Dynamic Binary Instrumentation (DBI). To perform the instrumentation, it performs dynamic recompilation using Just-In-Time (JIT) compilation. That proceeds as follows:

- Read a bunch of machine code.
- Translate these into Valgrind Intermediary Representation (IR) — this is the same sort of representation that compilers use, and by no coincidence Julian Seward also once worked on the Glasgow Haskell Compiler
- Instrument the IR depending on the needs of the tool
- Perform optimization and rewriting on the IR
- Store the JITted opcodes in a cache and execute them

Memory separation

Valgrind has its own memory manager. It maintains a strict separation of memory that is used by the host and memory that is used for the guest. Many of the tools replace the C and C++ allocation and deallocation functions. For these tools, it is the Valgrind memory manager that handles everything. Tools like cachegrind and callgrind do not replace the memory allocators (and thus, they include the allocators in their performance profiling).

Valgrind startup

Valgrind starts off in assembler in its own _start routine (no libc, remember?), and the first things that it does is create a temporary stack for itself, set up logging, and set up the heap allocator. The point I want to make here is that there is little room for mistakes. If something goes wrong, if you are lucky, you just won't get filenames and line numbers in error messages. If you're not lucky, then all you will get is a load of hex addresses in a stack trace. As you can imagine, you fail pretty quickly if you don't have a stack. Once Valgrind has done all its internal setup, it is ready to start the guest executable on the synthetic CPU. It creates another stack for itself that has a configurable size, and it starts the guest executable. From the perspective of the guest executable, it is just like it were running natively.

Handling syscalls, threads, and signals

Valgrind intercepts all system calls. Fortunately, most of them do nothing or just have a few checks (do the registers contain initialized memory?) and then get forwarded to the kernel. More complicated syscalls will have a behavior that depends on some operation code (like umtx_op and ioctl). Finally, there are syscalls that do not get forwarded to the kernel that need to be implemented by Valgrind. An example of that is 'getcontext' where Valgrind needs to fill the context from its synthetic CPU rather than letting the kernel fill it from the context of the Valgrind host.

One tricky thing is that the code running on the virtual CPU needs to stay on the virtual CPU. While Valgrind executes some guest code natively on the physical CPU, that's usually extremely limited in scope. If the control flow of the guest escapes back to the physical CPU, things will go horribly wrong. I'll give two examples of the contortions that are needed to ensure Valgrind stays in control. Firstly, thread creation. When there are calls to 'pthread_create' Valgrind needs to make sure that the OS doesn't run the function passed in the third argument. Instead, it needs to hook the third argument with a "run_thread_in_valgrind" function. Similarly, for signals Valgrind needs to ensure that guest signal handlers run under

Valgrind, and then that the return from the signal handler goes back to running under Valgrind. These things require some very hacky code. Valgrind also must do a lot of juggling of signal masks. When the guest is running, signals are blocked with the host polling and handling signals itself. When there is a syscall, signals are unmasked, the syscall performed, and signals masked again. Without this little dance, blocking syscalls would not be interruptible.

The Valgrind Port

When I started looking at the Valgrind port, it was in a bad state. As mentioned earlier, there was a push from 2009 to 2011 to get the port upstreamed. From 2011 to 2018 it slipped back to minimal maintenance.

Valgrind on amd64 was broken due to a change to add large file support to the 'stat' family of functions. A couple of people had found patches for that. I386 was broken in several ways. There were no FreeBSD-specific regression tests. Valgrind contains many tests that run on all platforms, and then all combinations of OS and CPU architecture (e.g., amd64, freebsd and amd64-freebsd). There are 600 or so of these common tests. Linux amd64 has about 200 or so tests on top of those common tests. I don't remember how many of those common tests were passing and failing, probably not much more than half. Fortunately, there was a large amount of low hanging fruit. After sorting some serious issues on i386, after about six months I had about 90% of the regression tests working. That may sound good, but there were still some serious limitations. Slogging through the remaining 10% really was a case of the last 10% taking 90% of the time

When I started looking at the Valgrind port, it was in a bad state.

War Stories

Signals leading to asserts

Signals. Oh my, I did have a hard time at first understanding all this. When running natively, signals will do the following:

- The kernel synthesizes a ucontext block which contains the address where the signal occurred and a call frame on the stack (or the alt stack), with the call frame return address set to the 'retpoline' (a small asm function for returning from signal handlers)
- The kernel transfers the running exe to the signal handler
- The signal handler does its stuff and returns
- The retpoline calls the sigreturn syscall
- The kernel gets the original address before the signal from the content and transfers execution there

On Linux, that picture holds for both non-threaded and threaded applications. On FreeBSD, once you link with libthr, the picture changes. 'thr_sighandler' replaces the user signal handler. This does some things like signal masking. It calls the user signal handler and calls sigreturn itself.

Valgrind can't let guest code execute. So, it handles all possible signals. It synthesizes its own context with a bit more information. It replaces the guest signal handler with its own *run_signal_handler_in_valgrind* function. The return address has set its own retpoline that will call a *valgrind_sigreturn* that will transfer execution of the guest back to where came from. What could possibly go wrong? As it turns out, almost everything. There have been at least two things that were broken in this flow that I've dealt with.

The first was a very small code change. Valgrind crashed when returning from guest signal handler functions on i386. After a lot of debugging, I narrowed this down to the assembly retpoline function VG_(*x86_freebsd_SUBST_FOR_sigreturn*). At some point, there must have been some change to the size of the ucontext structure. VG_(*x86_freebsd_SUBST_FOR_sigreturn*) was looking for the return address at the wrong offset - 0x14 instead of 0x1c. That meant the virtual CPU was resuming execution at some rubbish address. Boom! That soon hit an assert.

My second big battle with signals was intermittent. If a signal arrives when Valgrind is executing "ordinary" guest code on the virtual CPU, that is great because it knows exactly where to resume from. But what happens if a signal arrives during a syscall? Things are a lot more complicated because syscalls are one of the places where Valgrind is sort-of letting the guest run on the physical CPU. Valgrind can't make guest syscalls within its global lock. The syscall might block and that would cause multi-threaded processes to hang. Instead, it releases the lock and then makes the syscall. Now, if an interrupt happens in the window when the lock is down, Valgrind needs to try to figure out exactly where it happened so that it can decide whether it needs to be restarted or not. To do that, the machine code function that does the guest syscall, ML(*do_syscall_for_client_WRK*), has an associate table of addresses that correspond to setup, restart, complete, committed and finished. That worked well, but occasionally would fail with an assert. The problem was with how the syscall status gets set. On Linux, it's just in the RAX register, and that gets returned from the small assembly function, so nothing special needs to be done. On FreeBSD (and Darwin), it's saved in the carry flag. That needs a function call to set the carry flag in the synthetic CPU. And if a signal arrives when Valgrind is in the *LibVEX_GuestAMD64_put_rflag_c* function call? That case wasn't handled — resulting in the assert. Unfortunately, in C there's no easy way to tell which function the instruction pointer is executing in. You can take the address of the start of the function easily enough. But where is the end? I did consider using the Valgrind DWARF debuginfo (which should always be present and Valgrind has DWARF reading code built in). In the end, I went for an ugly and non-standard way. I took the address of a dummy function just after *LibVEX_GuestAMD64_put_rflag_c*. It happened to work on i386 and amd64 even though there is no guarantee that the compiler and linker will lay out functions in the same way that they appear in source files. Later, when I worked on the aarch64 port this did not work because the carry flag setting function uses several helper functions, and they aren't all laid out in the same order. So, I switched to setting a global variable from the assembler routine that makes guest system calls.

What happens if a signal arrives during a syscall?

GlusterFS swapcontext crashes

One more war story. This was one of the first bug reports I got after I released the rebooted FreeBSD Valgrind. A user running GlusterFS was getting crashes in Valgrind. After quite a bit of toing and froing, asking for log files and traces, I narrowed it down to the swapcontext syscall. It turned out that switched-to context has two pointers to the signal mask in the thread state that Valgrind saves. Only the first of them was getting set. Another case of several days of debugging for a one-line code change.

FreeBSD issues

The work I've done on Valgrind has also revealed a few bugs in FreeBSD. I had to debug one of those early on when I was working with i386 binaries running on amd64. I didn't have problems with i386 on i386 or amd64 on amd64 but i386 on amd64 was crashing early in the guest startup, in the link loader (lib rtld). Eventually I discovered that this was a problem with the detection of the pagesize. Normal standalone applications have this information in their auxiliary vector (auxv) as AT_PAGESZ (the actual page size) and AT_PAGESIZES (a pointer to table of possible page sizes). Valgrind synthesizes the aux for the guest, but at the time, it ignored AT_PAGESIZES. No problem, rtld has a fallback to use the HW_PAGESIZE sysctl. I386 has two possible page sizes, but amd64 has three possible page sizes. Unfortunately, what was happening was that rtld running on amd64 was using the size of three for PAGESIZES, but the i386 kernel component was using a size of two. The result was that the sysctl was returning ENOMEM.

The Elephant in the Room — Sanitizers

Why bother using Valgrind now that we have the sanitizers? I'll also turn that question around and ask why use sanitizers when we have Valgrind? Roughly Address Sanitizer and Memory Sanitizer are equivalent to Memcheck, and Thread Sanitizer is equivalent to DRD and Helgrind. UB sanitizer has no Valgrind equivalent.

There's one case when using Valgrind is simply out of the question. That is, if you are using an unsupported CPU architecture. Valgrind on FreeBSD only supports amd64, i386, and aarch64. If you are using another architecture, then Valgrind is out of the question. Next, Valgrind is lagging CPU development. That means if your application relies on using AVX512 then you can't use Valgrind.

If both sanitizers and Valgrind work on your system, which should you choose? As ever, it depends.

	Valgrind	Sanitizer
Speed	Very slow, to the point of being sometimes unusable	Slow
Stack bounds checking	No	Yes
Instrumentation required	No	Yes
Availability and support	amd64 i386 aarch64	amd64 i386 aarch64 risc-v

When I said Valgrind doesn't need instrumentation, that was a white lie. If you are using custom allocators, then you need to write some annotation for either Valgrind or the sanitizers to work correctly. Similarly, if you use custom thread locking routines like a spin lock, you need to annotate them again in both cases. Thread sanitizer does have the advantage of having built-in annotation for standard library mechanisms that don't rely on pthreads such as std::atomic.

FreeBSD is lucky to have its toolchain based on LLVM. That means memory sanitizer is easily available. GCC doesn't have memory sanitizer, making it a lot more difficult to use on Linux. Don't underestimate how big a task "instrumentation required" is. For the best results that means you should instrument all your dependent libraries. If you are a KDE application developer, that means at least the following sets of libraries: KDE, Qt, libc++. There are dozens of other dependencies (libfontconfig, libjpeg, etc.). As we Valgrind developers like to say, "good luck with that!" If you are working for a big company and you have a dedicated devops team that can set it all up, then it's not so bad. I'd be interested in hearing from anyone who has experience in using poudriere for sanitizer builds. I've also read about people with large unit test suites complaining about the excessive build time and disk space requirements when building with sanitizers, particularly as you can't do a "one stop shop" sanitizer build (address and memory sanitizers are incompatible).

My conclusion here is that you should use whichever best suits your needs.

Future Work

Unfortunately, Valgrind is a tool that bitrots very quickly. New versions of FreeBSD keep coming out with new and changed syscalls. Extra items keep getting added to the auxiliary vector. _umtx_op gets more commands. libc++ keeps finding stranger ways of using pthreads. Compilers optimize things in ways that look like they are unsafe. That means that work on Valgrind is never finished.

You can't do a "one stop shop" sanitizer build.

CPU architectures

Valgrind on FreeBSD runs on amd64, i386, and aarch64. I can't see myself adding MIPS or PPC support. RISC-V hasn't yet been added to the official Valgrind source — a port is on the way, but currently it is being held up by discussions over the implementation of vector instructions.

Bug list

The Valgrind Bugzilla has around 1000 open bugs in it. While many of these only affect Linux/macOS/Solaris, there are a good number that do affect FreeBSD.

- Helgrind produces false positives in thread local storage when there is a lot of thread creation/destruction. That is because there is a cache for the pthread stacks that include TLS. Valgrind doesn't see the recycled TLS as having different memory addresses. Linux works around this by deactivating the pthread stack cache via a GNU libc environment variable. I haven't found a way to do the same thing with FreeBSD libc.
- When the guest coredumps, it is Valgrind that generates the core file. Currently, the core file is pretty much with the same layout as a Linux core dump. That means lldb and gdb can't do much with the core file. I don't think that is a big issue as not many people use core files these days.
- The thread scheduler. Valgrind has a very rudimentary thread scheduler. Thread context switches occur at system call boundaries or every 100000 basic blocks. The default scheduler simply releases the global lock, and it's a question of luck as to which thread gets the lock. That could well be the previous thread if it is hot in the CPU cache. Linux has an optional fair scheduler based on futexes. Whilst that can't be ported directly to FreeBSD, it shouldn't be too difficult to post it using _umtx_op.
- On aarch64 there are occasional DRD false positives related to accesses in thread local storage

- The code that verifies ioctls is very limited. Almost all ioctls only get basic size checking done on their arguments. This needs to be extended, ideally also with testcases.

Conclusions

Working on Valgrind is quite a challenge. Debugging can be extremely difficult – I've often found myself doing things like debugging the guest in parallel with debugging Valgrind running the guest in parallel with using vgdb to debug the guest running in Valgrind. I've learned a lot about ELF, signals, and syscalls as well, of course, as about Valgrind itself. There's always much to learn — the nuances of aarch64 and amd64 opcodes and the multitude of tricks used in the dynamic recompilation.

PAUL FLOYD has been using FreeBSD intermittently since 2.1 and in earnest since 10.0. He's been a member of the Valgrind development team for four years. He has a PhD in Electronics and lives near Grenoble, on the edge of the French Alps working for Siemens EDA developing tools for analog electronic circuit simulation.

Write For Us!

Contact Jim Maurer with your article ideas.

(**maurer.jim@gmail.com**)

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

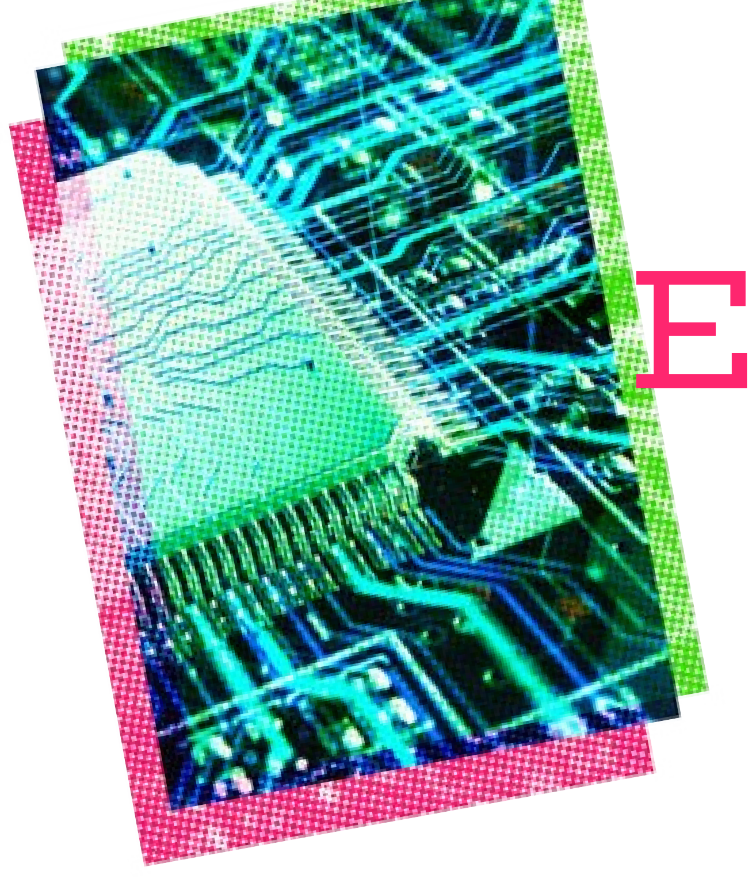
Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsdfoundation.org/donate

FreeBSD™
FOUNDATION



Embedded FreeBSD

Digression into bhyve

BY CHRISTOPHER R. BOWMAN

In the previous two columns we talked about the Digilent Arty Z7-20 with which I've been experimenting. I find this an interesting board because not only can you toggle primary pins to interface to the outside world, but you can build your own circuits into the fabric and interface them to the processor. To do all this, however, you'll need to use Xilinx/AMDs software to configure and program the chip. Xilinx calls this tool suit Vivado and you can download a version which works with the Zynq chips free of charge from their website.

So, what's the downside? There has to be another shoe waiting to drop somewhere, right? There are two really. First, Vivado only comes in versions for Windows and Linux. Second, the download alone is currently 110GB. For years, I ran the Linux version under VMWare on my Mac and it worked pretty well. But I ended up with several different versions of my virtual machine, each with different versions of the software installed. These VMs were large, starting at 30 gigs or so in the early days and growing most recently to 75 gigs in my most recent installs. Make a couple of versions of these on a couple of machines, and, well, that starts to add up, and I have trouble keeping track of which one is the most recent.

I can use the magic of ZFS to checkpoint my vendor software installs.

I decided I wanted to simplify and centralize. I want to store all the originals of vendor tools I downloaded on my networked home file server, which runs FreeBSD, naturally, and has terabytes of ZFS space. After all, I can't depend on my vendor to continue to offer these past versions after they've moved to newer versions. That's understandable, but as a hobbyist, my work doesn't always proceed at their pace, so I want to make sure I keep the original tool downloads. I wanted my home directory and my working files and projects to be stored on my network server, so they are available from all my machines and backed up like any other data on my file server. I wanted all the unzipped and installed versions of the vendor software to be stored on my file server not in my VMs. This way I can use the magic of ZFS to checkpoint my vendor software installs. My VMs are a light weight install of Linux, any Linux with which I want to experiment. If I need to upgrade or create a new VM, I know there is no work on there, it's all on my file server, so I just need a new basic install and everything else is all still on my file server. I don't have to copy over or reinstall the vendor tools. Since recent upgrades to the compute environment at my home have included a FreeBSD machine with 16 cores and 128 Gigabytes of memory, I want to run these tools on that machine in anticipation of the point in time where my designs get big enough that they take hours to synthesize. As a bonus, I get a setup where someone else looking to replicate my work needs only a single FreeBSD machine.

There seem to be two basic approaches at this point. I could try to get the installer to run under FreeBSD's Linux emulation and run these tools on my FreeBSD box natively without a Linux VM. That would be AWESOME! But I'm not sure it could be done or what problems I would run into or how long that might take me to figure out, and I was in the middle of some projects. This really seems like the best approach, and I'd love to hear if anyone has this working or wants to try it, but I chose an approach that I thought would involve less effort and be more likely to work. I'd heard a lot about bhyve and decided to investigate that. If I get a VM running a version of Linux supported by Vivado natively, I figured that would be the easiest. It only took me an evening or so of reading and another of experimentation and I was shockingly up and running.

I started with the FreeBSD handbook. It's really an amazing resource and kudos to everyone who helped make it as brilliant as it is.

I started with the FreeBSD handbook. It's really an amazing resource and kudos to everyone who helped make it as brilliant as it is. Chapter 24.6. FreeBSD as a Host with bhyve has a really good introduction to using FreeBSD as a host operating system. For the most part, I cobbled together a setup from there and a few other places on the internet. To setup, I created the basic host setup for networking:

```
# kldload vmm

# ifconfig tap0 create up
# ifconfig bridge0 create
# ifconfig bridge0 addm igb0 addm tap0
# ifconfig bridge0 up
```

I'm using the instructions in 24.6.5. Graphical UEFI Framebuffer for bhyve Guests since I don't want to muck around with setting up grub. Using the UEFI Framebuffer also allows me to export the Linux display via vnc. I can connect from my FreeBSD host if I'm using that or from any other machine on my network. Though perhaps I should think a bit more about security.

I downloaded an ISO version of CentOS from before RedHat killed it, and I use the following VM configuration to do the install:

```
# bhyve -c 4 -m 32G -w -H \
        -s 0,hostbridge \
        -s 3,ahci-cd,/u1/ISOs/CentOS/CentOS-7-x86_64-DVD-2009.iso \
        -s 4,ahci-hd,/dev/zvol/zroot/vms/centos7 \
        -s 5,virtio-net,tap0 \
        -s 29,fbuf,tcp=0.0.0.0:5900,w=1920,h=1200,wait \
        -s 30,xhci,tablet \
        -s 31,lpc -l com1,stdio \
        -l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd \
        vm0
```

With this I get a 4 core 32-gig virtual machine with the ISO, `/u1/ISOs/CentOS/CentOS-7-x86_64-DVD-2009.iso` mounted in the guest as a CDROM. With this I can run a standard GUI install of Linux which is pretty straight forward. I'm running this on a bare zvol: `/dev/zvol/zroot/vms/centos7`. I did this so I could use ZFS snapshots to snapshot my VM any time I want (initially right after a clean install) so I can roll back at any time. If I put the VM virtual disk on a ZFS files system, my snapshot would apply to anything else on that file filesystem, not just the VM virtual disk image. I also heard using bare zvols can be faster. In hindsight, I could have used a single data set per VM virtual disk image and used a file on that data set as the VM virtual disk image rather than a zvol. If there is one VM per data set, snapshots still apply to just the VM. I'm not sure which is a better approach, and subsequent reading has called into question the zvol vs. file backing speed aspect. My tool run times aren't yet long enough that I'm concerned about eking out every last ounce of performance. I've also heard the nvme devices can be faster than the ahci-hd device for guest OS that support them, but I haven't experimented there. If it becomes an issue, it's relatively easy to create a new VM now that my data and install doesn't reside on the VM.

Now I have a fairly generic CentOS VM with a Vivado installation.

Once I installed CentOS, I configured it to NFS to mount my home directory from my FreeBSD machine, and I ran the Xilinx Vivado tool installation. It's a gui install and pretty straightforward. I just make sure I'm installing to an NFS mounted directory. Given the volume of file operations, I was expecting this to be pretty painful, but it went surprisingly quickly for a tool that ends up at 66 Gigabytes installed. Granted I have a very fast local network. I don't expect to edit the tool install, but I created a snapshot — this too for peace of mind. I don't want to have to run that install again.

When I applied for a Vivado license, I copied the ethernet MAC address that my Linux guest reports. It seems to be stable across reboots, but I'd like to figure out how to configure this so that I can be sure that the MAC address on my VM always matches my Vivado license MAC address.

Now I have a fairly generic CentOS VM with a Vivado installation that I can access via VNC from any machine on my local network. At this point, I also installed a 15-year-old Linux version of *Civilization: Call to Power* to play while my FPGA builds are compiling. I was shocked at how well it works (and how addicted I am to it.)

While most of this works really well, there are a couple differences from my initial setup using VMWare on a Mac. First VMWare supports pass through to the host filesystems. The NFS mounts accomplish basically the same thing, and I can't notice much speed penalty, but I have to configure this in Linux instead of the VMWare gui. Not a big difference--I'm comfortable with that. Where I do find myself wishing for the VMWare solution, is copy and paste. If I select something in the Linux Guest gui, I can't easily copy and paste that to the machine on which I'm running the VNC viewer. This is occasionally a sore point, but since the user filesystems I'm using under Linux are all NFS mounted from FreeBSD, I can edit those files live just as easily from FreeBSD (or any other machine that's NFS mounting them). As a result, I do almost no editing or work under Linux. Mostly, I just switch to the Linux VNC session window to run the Vivado compiles, and I do everything else outside that. It works pretty well.

In the next column, we will start to use Vivado to build our first circuit.

If you've got feedback, complaints or flames on any of this I'd love to hear from you. You can contact me at articles@ChrisBowman.com.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

Contents

Adventures in TCP/IP

Pacing in the FreeBSD TCP Stack

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP sending and receiving behavior has evolved over the more than 40 years that TCP has been used. Many of the advances have helped TCP to be able to transmit a reliable stream of data at very high speeds. However, some of the enhancements (both in the stack and in the network) come with downsides. Originally, a TCP stack sends a TCP segment in response to a received TCP segment (an acknowledgment), or in response to the upper layer providing new data to send, or due to a timer running off. The TCP sender also implements a congestion control to protect the network against sending too fast. These features combined can cause a TCP endpoint to be driven most of the time by an "ACK clock" as depicted in the following figure.

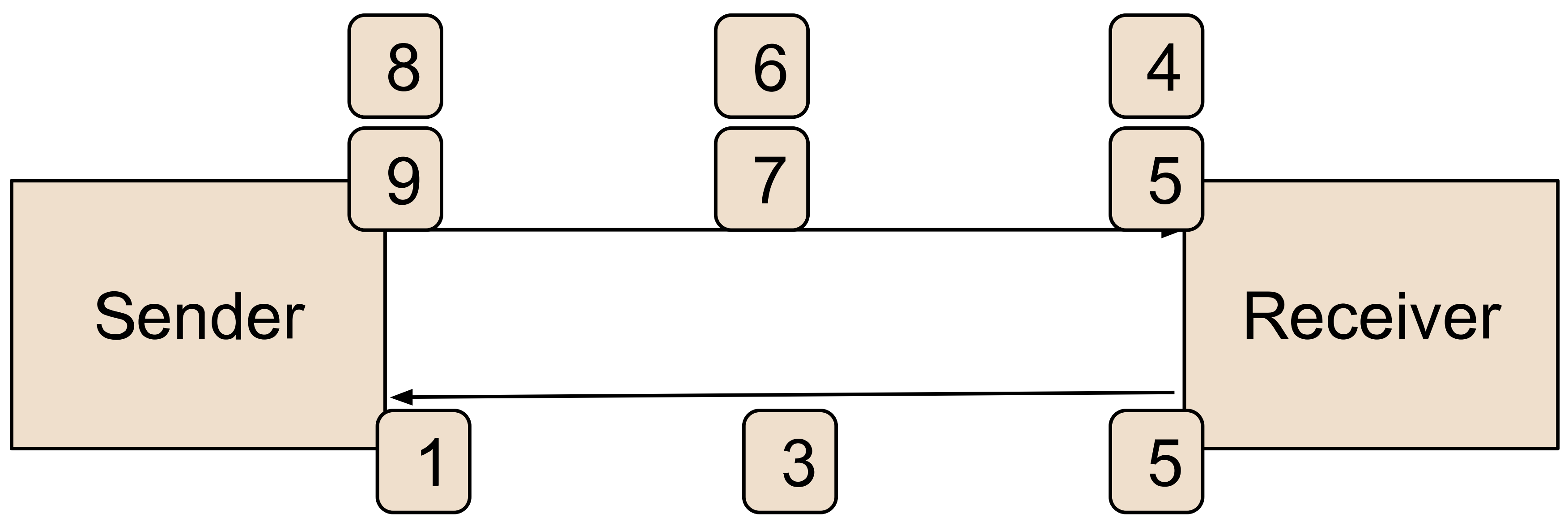


Figure 1: An example of ACK-clocking

For simplicity it is assumed that packets are numbered and acknowledged and that the receiver acknowledges every other packet to minimize the overhead. The arrival of the acknowledgment labeled "1" on the bottom arrow acknowledges packets 0 and 1 that were sent earlier (not shown in the figure), removes two of the outstanding packets, and allows two more to be sent aka packets 8 and 9. At that point the sender is blocked (due to its congestion control) waiting for the arrival of the ACK labeled "3" which will acknowledge packet 2 and 3 and again send out the next two packets. This ACK-clocking was prevalent in a large number of flows in the early Internet and can still be seen today in some circumstances. ACK-clocking forms a natural pacing of data through the Internet allowing packets to be sent through a bottleneck, and oftentimes, by the time the next packets from the sender arrives at the bottleneck, the previous packets are transmitted.

However, over the years, both the network and TCP optimizations have changed this behavior. One example of a change in behavior is often seen in cable networks. In such networks the down link bandwidth is large, but the uplink bandwidth is small (often only a fraction of the downlink bandwidth).

Due to this scarcity of the return path bandwidth, the cable modems will often keep only the last acknowledgment sent (assuming it is seeing the acknowledgments in sequence) until it decides to transmit. So, in the above example, instead of allowing ACK-1, ACK-3 and ACK-5 to be transmitted by the cable modem, it might only send ACK-5. This would then cause the sender to send one larger burst of 6 packets, instead of spacing out 3 separate bursts of two packets.

Another example of a modifying behavior can be seen in other slotted technologies (they hold off sending until their time slot is reached, and then they send out all queued packets at that time) where the acknowledgments are queued up and then all sent at once in one burst of 3 ACK packets. This type of technology would then interplay with TCP-LRO (talked about in our last column) and thus either collapse the acknowledgments into one single acknowledgment (if the old methods are being used) or queue up for simultaneous processing of all of the acknowledgments before the sending function is called. In either case, a large burst is again sent instead of a series of small two packet bursts, separated by a small increment of time (closely approximating the bottleneck bandwidth plus some propagation delay).

Another example of a modifying behavior can be seen in other slotted technologies.

Our example shows only six packets but several dozen packets can burst all in one `tcp_output()` call. This is good for CPU optimization but can cause packet loss in the network since router buffers are limited and large bursts are more likely to cause a tail drop. This loss would then reduce the congestion window and hinder overall performance.

In addition to the two examples given above, there are other reasons that TCP can become bursty (some of which have always been inherent in TCP) such as application limited periods. This is where a sending application stalls for some reason and delays the sending for some number of milliseconds. During that time, the acknowledgments arrive but there is no more data to send. But before all the data from the network is drained, which might cause a congestion control reduction due to idleness, the application sends down another large block of data to be sent. In such a case, the congestion window is open and a large sending burst can be generated.

Yet another source of sources of burstiness may come from the peer TCP implementations that might decide to acknowledge every eighth or sixteenth packet instead of following the TCP standard and acknowledging every other packet. Such large stretch acknowledgments will again cause corresponding bursts. TCP pacing, described in the next section, is a way to improve the sending of TCP segments to smooth out these bursts.

TCP Pacing

The following example illustrates TCP pacing. Let's assume that a TCP connection is used to transfer data to its peer at 12 megabits per second including the IP and TCP header. Assuming a maximum IP packet size of 1500 bytes (which is 12,000 bits), this results in sending 1000 TCP segments per second. When using pacing, sending a TCP segment every millisecond would be used. A timer that runs off every millisecond could be used to achieve this. The following figure illustrates such a sending behavior.

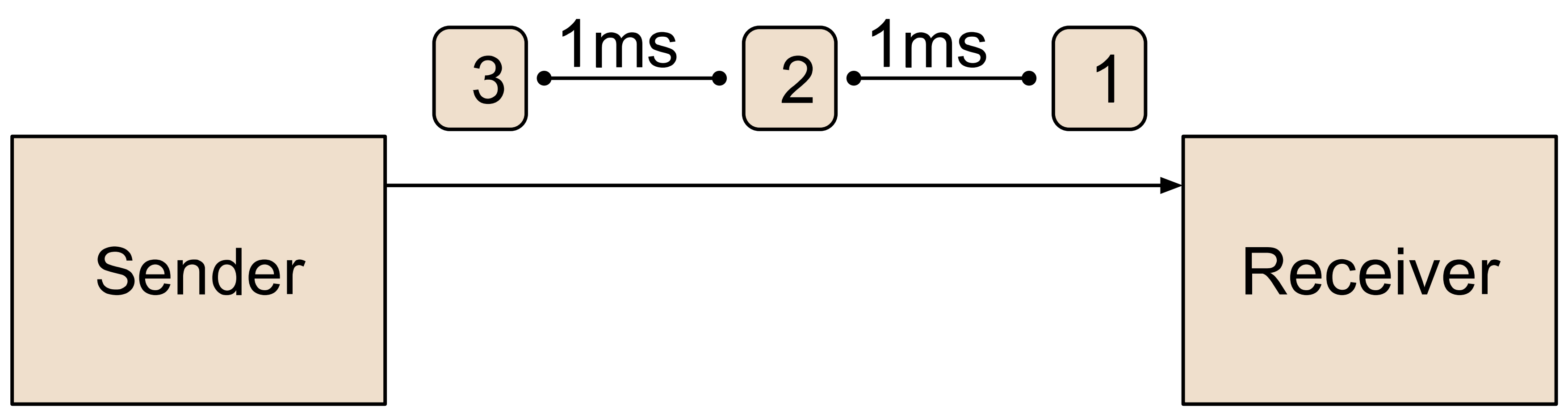


Figure 2: A TCP connection paced at 12Mbps

Doing pacing in such a manner is possible but not desirable due to the high CPU cost this would take. Instead, for efficiency reasons, when pacing, TCP sends small bursts of packets with some amount of time between them. The size of the burst is generally correlated to the speed that the stack wishes to pace at. If pacing at a higher rate, larger bursts are in order. If pacing at a lower rate, smaller bursts are used.

When designing a methodology for pacing a TCP stack, there are a number of approaches that can be taken. A common one is to have the TCP stack set a rate in a lower layer and then hand off large bursts of data to be sent to that layer. The lower layer then just multiplexes packets from various TCP connections with appropriate timers to space out the data. This is not the approach taken in the FreeBSD stack, specifically because it limits the control the TCP stack has over the sending. If a connection needs to send a retransmission, that retransmission ends up falling in behind all the packets that are in queue to be sent.

In FreeBSD, a different approach was taken in letting the TCP stack control the sending and creating of a timing system that is dedicated to calling the TCP stack to send data on a connection when the pacing interval has ended. This leaves complete control of what to send in the hands of the TCP stack, but can have some performance implications that need to be compensated for. This new subsystem created for FreeBSD is described in the next section.

High Precision Timing System

Conceptual Overview

The High Precision Timing System (HPTS) is a loadable kernel module and provides a simple interface to any TCP stack wishing to use it. Basically, there are two main functions that a TCP stack would call to get service from HPTS:

- `tcp_hpts_insert()` inserts a TCP connection into HPTS to have either `tcp_output()` or, in some cases, TCP's inbound packet processing call `tfb_do_queued_segments()` at a specified interval.
- `tcp_hpts_remove()` asks HPTS to remove a connection from HPTS. This is often used when a connection is closed or otherwise no longer going to send data.

There are some other ancillary helping functions that are available in HPTS to help with timing and other housekeeping functions, but the two functions listed above are the basic building blocks that a TCP stack uses to implement pacing.

Details

Internally, each CPU has an HPTS wheel, which is an array of lists of connections wanting service at various time points. Each slot in the wheel represents 10 microseconds. When a TCP connection is inserted, it is given the number of slots from now (i.e., 10 microsecond intervals) that need to elapse before the `tcp_output()` function is called. The wheel is managed by a combination of both a system timer (i.e., FreeBSD's callout system) and a soft timer as proposed in [1]. Basically every time a system call returns, before the return to user space, HPTS can potentially be called to see if an HPTS wheel needs to be serviced.

The HTPS system also auto tunes its FreeBSD system timer having first a minimum (defaulting to 250 microseconds) and a maximum that it can tune up. If an HPTS wheel has more connections and is getting called more often, the small amount of processing during FreeBSD system timeout will raise the length of the system timer. There is also a low connection threshold where if the number of connections drops below, then only the system timer based approach is used. This helps avoid starving out connections by keeping them on the wheel too long. HPTS attempts to yield a precision of the timer minimum aka 250 microseconds, but this is not guaranteed.

A TCP stack using HPTS to pace has some distinct responsibilities in order to collaborate with HPTS to achieve its desired pace rate including:

- Once a pacing timer has been started, the stack must not allow a send or other call to `tcp_output()` to perform any output until the pacing timer expires. The stack can look at the `TF2_HPTS_CALLS` flag in the `t_flags2` field. This flag is or'ed onto the `t_flags2` as HPTS calls the `tcp_output()` function and should be noted and cleared by the stack inside its `tcp_output()` function.
- At the expiration of a pacing timer, in the call from HPTS, the stack needs to verify the time that it has been idle. It is possible that HPTS will call the stack later than expected, and it is even possible that HPTS will call the stack early (though this is quite rare). The amount of time that the stack is late or early needs to be included in the TCP stack's next pacing timeout calculation after it has sent data.
- If the stack decides to use the FreeBSD timer system, it must also prevent timer callouts from sending data. The RACK and BBR stacks do not use the FreeBSD timer system for timeouts, and, instead, just use HPTS as well.
- If the stack is queuing packets from LRO, then HPTS may call the input function instead of `tcp_output()`. If this occurs, no other call to `tcp_output()` will be made, since it is assumed that the stack will call its output function if it is needed.

There are also a number of utilities that HTPS offers to assist a TCP stack including:

- `tcp_in_hpts()` tells the stack if it is in the HPTS system.
- `tcp_set_hpts()` sets up the CPU a connection will use, and is optional to call, since the HPTS will do this for the connection if the stack does not call this function.
- `tcp_tv_to_hptstick()` converts a `struct timeval` into the number of HPTS slots the time is.
- `tcp_tv_to_usectick()` converts a `struct timeval` into a 32-bit unsigned integer.
- `tcp_tv_to_lusectick()` converts a `struct timeval` into a 64-bit unsigned integer.
- `tcp_tv_to_msectick()` converts a `struct timeval` into a 32-bit unsigned millisecond tick.
- `get_hpts_min_sleep_time()` returns the minimum sleep time that HPTS is enforcing.
- `tcp_gethptstick()` optionally fills in a `struct timeval` and returns the current monolithic time as a 32-bit unsigned integer.
- `tcp_get_u64_usecs()` optionally fills in a `struct timeval` and returns the current monolithic time as a 64-bit unsigned integer.

`sysctl`-Variables

The HPTS system can be configured using `sysctl`-variables to change its performance characteristics. These values come defaulted to a set of "reasonable" values, but depending on the application, they might need to be changed. The values are settable under the **`net.inet.tcp.hpts`** system control node.

The following tunables are available:

Name	Default	Description
`no_wake_over_thresh`	1	When inserting a connection into HPTS, if this boolean value is true and the number of connections is larger than **`cnt_thresh`**, do not allow scheduling of a HPTS run. If the value is 0 (false), then when inserting a connection into HPTS, it may cause the HPTS system to run connections aka call **`tcp_output()`** for connections due to be scheduled.
`less_sleep`	1000	When HPTS finishes running, it knows how many slots it ran over. If the number of slots is over this value then the dynamic timer needs to be decreased.
`more_sleep`	100	When HPTS finishes running, if the number of slots run is less than this value then the dynamic sleep is increased.
`min_sleep`	250	This is the absolute minimum value that the HPTS sleep timer will lower to. Decreasing this will cause HPTS to run more using more CPU. Increasing it will cause HPTS to run less using less CPU, but it will affect precision negatively.
`max_sleep`	51200	This is the maximum sleep value (in HPTS ticks) that the timer can reach. It is typically used only when no connections are being serviced i.e., HPTS will wake up every 51200 x 10 microseconds (approximately half a second).
`loop_max`	10	This value represents how many times HPTS will loop when trying to service all its connections needing service. When HPTS starts, it pulls together a list of connections to be serviced and then starts to call **`tcp_output()`** on each connection. If it takes too long to do this, then it's possible more connections need service, so it will loop back around to again service connections. This value represents the maximum HPTS will do that loop, before being forced to sleep. Note that being called on return from a function call never causes any looping to occur; only the FreeBSD timer call is affected by this parameter.
`dyn_maxsleep`	5000	This is the maximum value that the dynamic timer can be raised to when adjusting the callout time upwards is being performed and seeing the need for **`more_sleep`**.
`dyn_minsleep`	250	This is the minimum value that the dynamic timer can lower the timeout to when adjusting the callout time down after seeing **`less_sleep`**.
`cnt_thresh`	100	This is the number of connections on the wheel that are needed to start relying more on system call returns. Above this threshold, both system call return and timeouts cause HPTS to run, below this threshold, we rely more heavily on the callout system to run HPTS.

Optimizations for Pacing in the RACK Stack

When pacing using the HPTS system, there is some performance loss as compared to a pacing system that runs below a TCP stack. This is because when you call `tcp_output()` a lot of decisions are made as to what to send. These decisions usually reference many cache lines and cover a lot of code. For example, the default TCP stack has over 1500 lines of code in the `tcp_output()` path and it includes no code to deal with pacing or burst mitigation. For the default stack without pacing, going through such a large number of lines of code and lots of cache misses is compensated easily by the fact that it might output several dozen segments in one send. Now, when you implement a pacing system that is below the TCP stack, it can readily optimize the sending of the various packets it has to do by keeping track of what and how much it needs to send next. This makes a lower layer pacing system have many fewer cache misses.

In order to obtain similar performance with a higher layer system like HPTS, it becomes up to the TCP stack to find ways to optimize the sending paths (both transmissions and retransmissions). A stack can do this by creating a "fast path" sending track. The RACK stack has implemented these fast paths so that pacing does not cost quite so much. The BBR stack currently does pace, as required by the BBRv1 specification that was implemented, but it does not (as yet) have the fast paths described below.

Retransmissions in RACK also have a fast path. This is made possible by RACK's sendmap.

Fast Path Transmissions

When RACK is pacing the first time, a send call falls through its `tcp_output()` path and it will derive the number of bytes that can be sent. This is then lowered to conform to the size of the pacing microburst that has been established, but during that reduction, a "fast send block" is set up with the amount that is left to send and pointer to where in the socket send buffer that data is. A flag is also set so that RACK will remember next time that the fast path is active. Note that if a timeout occurs, the fast path flag is cleared so that proper decisions will be made as to which retransmission needs to be sent.

At the entry to RACK's `tcp_output()` routine, the fast path flag is checked after validating that, pacing wise, it is ok to send. If the flag is set, it proceeds to use the previously saved information to send new data without all of the typical checks that the output path would normally do. This brings the cost of pacing down considerably, since much of the code and cache misses are eliminated from this fast output path.

Fast Path Retransmissions

Retransmissions in RACK also have a fast path. This is made possible by RACK's sendmap which tracks all data that has been sent. When a piece of data needs retransmission, the sendmap entry tells the fast path precisely where and how much data needs to be sent. This bypasses typical socket buffer hunting and other overhead and provides a level of efficiency even when sending retransmissions.

Conclusion

HPTS provides a novel service TCP stacks can make use of to implement pacing. In order to achieve efficiencies more equivalent to competing design approaches, both the TCP stack and the HPTS need to cooperate to minimize overhead and provide for efficient sending of packet bursts. This column only discusses the need for pacing and the infrastructure provided to do so in FreeBSD. Future columns will look at another key question when a TCP stack paces, i.e., what rate to pace at.

Reference

1. Mohit Aron, Peter Druschel: *Soft Timers: Efficient Microsecond Software Timer Support for Network Processing*. In: ACM Transactions on Computer Systems, Vol. 18, No. 3, August 2000, pp 197-228. https://dl.acm.org/doi/pdf/10.1145/319344.319167.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.

Contents

PRACTICAL PORTS

Go Paperless

BY BENEDICT REUSCHLING

During the height of the Pandemic, I was staying home and the days were basically like any other. Time passed slowly, and looking around my room, I became painfully aware of how untidy it had become. Especially around my work desk, where piles of books, notes, letters and other pieces of paper had started a race to the top. I decided to do something about it. I put on a podcast and started picking up, ordering, throwing away, and cleaning my way toward a clean desk top. This was a great feeling of accomplishment in times of a global crisis. Motivated by that, I decided to start scanning those papers to reduce the pile.

I had purchased a mobile scanner a while ago, which connects via USB and uses some proprietary, yet efficient, scanning software. The scanner was not smaller than a rolling pin, and a single piece of paper would fit. Since I had time, I continued scanning each paper, giving the resulting PDF a description and date, moving on to the next one. This was tedious, but I got through it in the end. Ordering them afterwards was another long task: taxes, insurance, contracts, receipts, bills (both sent and received), records, certificates, and more needed to end up in the proper directories.

This software re-scans PDFs and makes the text into something that one can extract and search individual words.

The scanning software was only 32-bit, and the entire scanning unit stopped working a couple of years later — just when another pile of papers was dangerously close to tumbling over. It was time to look for alternatives. I had found `textproc/py-ocrmypdf` a few months earlier. Essentially, this software re-scans PDFs and makes the text, which is sometimes a big picture, into something that one can extract and search individual words. It uses pattern recognition to figure out the words and language and the results are surprisingly good. I ran this over my existing scans and now I can do a full-text search over a document and find how expensive my flight to BSDCan 2014 had been, for example.

Introducing Paperless-ngx

Then I found paperless-ngx, which `ocrmypdf` is a part of, but a comparably small part of a greater whole. It is a document management system that you can pass documents into and it creates an archive that is fully searchable, automatically detects the content (AI is everywhere) and files it away according to rules you define. Basically, I can feed it a letter, and it figures out that it comes from my bank, files it away based on the date it detected, while **ocrmypdf**-ing it makes it searchable and adds useful metadata as well. The file ends up in a directory that can either be subject related (everything I ever got from that particular bank), or by year-month-date-description, or something completely up to you. You can also feed entire directories into paperless-ngx, and it figures out which documents it has already scanned and skips those, while the rest run through a processing pipeline. With each document, the chances grow that future documents like it get properly categorized. Plus, it comes with a nice web UI into which you can drag and drop files for scanning and find existing documents with ease. Another way to feed documents into it is via an "incoming" folder that you can share among colleagues in an office or by sending documents as attachments (remember emails?) to it.

With each document, the chances grow that future documents like it get properly categorized.

The software stack itself is impressive and may be too intimidating, even with the excellent documentation that the paperless-ngx website provides. Plenty of software and services have to work together to get a smooth scanning experience. Luckily, there is a port for it under **deskutils/py-paperless-ngx.** Even better, the port maintainer created a post-install man page detailing all the steps to get a working paperless-ngx stack going. Did I mention that I love ports maintainers? With these instructions, I was able to set up my own paperless-ngx in no time. First on a Raspberry Pi 3 and then on a Pi 4. It works, even though the Pi 3 stretches your patience because of the required processing power to get the final result. With the Pi 4 though, I've had good experiences and the scanning time is decent enough. You could run this in the office or at home with a negligible footprint on your electricity bill, while allowing other people to scan their docs without seeing those of others. If you're dealing with a lot of documents and want to have them digitized, take a look at the paperless-ngx setup we're doing here. You can thank me later...

Paperless-ngx Setup

Whether you are using a Raspberry Pi, a different embedded device, or a full-blown server does not really matter. As long as it runs FreeBSD, you can follow along. I'm not spending any time on the base installation or hardening the system, as there are plenty of other good articles available that cover that. Just make sure to do exactly that when you connect your paperless-ngx service to the network for other people to use.

Start by installing the paperless-ngx port:

```
# pkg install deskutils/py-paperless-ngx
```

You'll be greeted by the **pkg-message** after installation, advising you to take a look at the man page for further instructions. Without them, you have only the basic service, which does not do too much at this point.

Most files end up in **/var/db/paperless**, which you can probably put on a separate ZFS dataset, but in my experience, the compression savings are not worth it. But your mileage may vary and ZFS is generally a good idea for storing those precious documents.

Paperless-ngx wants to have access to a Redis instance, which is what we're installing next:

```
# pkg install redis
# service redis enable
# service redis start
```

Easy enough, having it both installed and started at boot time, as well as the current session with these three commands. If you have Redis running somewhere else in your network, you need to modify and add its credentials to `/usr/local/etc/paperless.conf`. When running on localhost, it's fine to run it without any special privileges since it won't be reachable from other hosts this way.

The configuration file is well documented with comments. Some items like `THREADS_PER_WORKER` (mine is at 1 on the RPI 4), `PAPERLESS_URL` (IP address or DNS name), and `PAPERLESS_TIME_ZONE` (I use UTF) should be modified to fit your system and network. Many other settings are fine in their defaults for your first couple of scans. You can always revisit this file and make modifications later.

Paperless-ngx is backed by a database to store various information. It's as easy to initialize as you can imagine using the following command:

```
# service paperless-migrate onestart
```

If you want to run this every time the system starts, you can execute this `service` command as well:

```
# service paperless-migrate enable
```

After that is done, we will start the backend services that paperless uses in order:

```
# service paperless-beat enable
# service paperless-consumer enable
# service paperless-webui enable
# service paperless-worker enable
```

You can find individual descriptions of these on the paperless-ngx website. Since we want to use paperless-ngx without restarting the system, we start all these services next:

```
# service paperless-beat start
# service paperless-consumer start
# service paperless-webui start
# service paperless-worker start
```

Machine learning is all the rage behind the AI hype. Paperless-ngx uses it as well, but mostly to aid in the character recognition to figure out the language of the document at hand. To do that, it uses the Natural Language Toolkit (NLTK). To download the necessary files, the following one-liner does the trick (replace the python version if necessary):

```
# su -l paperless -c '/usr/local/bin/python3.11 -m nltk.downloader \
 stopwords snowball_data punkt -d /var/db/paperless/nltkdata'
```

Documents are classified in different ways, which is the responsibility of the Celery component. This classification is done automatically upon scanning, but you can trigger it manually with this invocation:

```
# su -l paperless -c '/usr/local/bin/paperless document_create_classifier'
```

Celery also runs an optional component called Flower. It monitors a cluster of workers that Celery controls. This is an optional component and I run my instance without it. But for those who want all the bells and whistles, here is how to start it:

```
# service paperless-flower enable
# service paperless-flower start
```

Setting up the Web UI

To protect your Django-based Web UI holding all your documents scanned so far, you can set a superuser password like this:

```
# su -l paperless -c '/usr/local/bin/paperless createsuperuser'
```

I run an nginx webserver already (SSL proxy), so I can re-use that to point to my paperless-ngx website. If you don't have one already, the port also provides a ready-to-use configuration file in **`/usr/local/share/examples/paperless-ngx/nginx.conf`** that you just have to copy into your **`/usr/local/etc/nginx/`** directory. This includes an SSL configuration as well to not let people sniffing traffic figure out the login and doing other nasty things. To create a key that's valid for a whole year, run this lengthy **`openssl`** incantation (or get a key via **`lets-encrypt`**):

```
# openssl req -x509 -nodes -days 365 -newkey rsa:4096 \
 -keyout /usr/local/etc/nginx/selfsigned.key \
 -out /usr/local/etc/nginx/selfsigned.crt
```

Of course, you can make your own adjustments to the **`nginx.conf`** when necessary. When finished, enable it to start at boot time and for the current session:

```
# service nginx enable
# service nginx start
```

Voila! Now point your browser to the web URL defined in the **`paperless.conf`** and log into the application.

Basic configuration in the web UI

Before scanning your first document, I would recommend setting up a couple of items in the "Manage" section on the left, first. To begin, Correspondents are people or organizations that have sent you the paper. Think of banks, insurance companies, but also individuals. You can give them a descriptive name and configure paperless-ngx as to whether it should file a document with this correspondent if it detects certain keywords or other criteria.

Next, define document types. A contract is different from a love letter, which differs from a bill, which is not the same as certificate, and so on. This way, you can let paperless-ngx distinguish whether someone has sent you a bill or if that same person gave you a contract. Both can happen, and especially government agencies have a tendency (at least where I live) to correspond with you in different contexts, which you want to keep separate from each other. That's where paperless-ngx shines: once you defined your most active Correspondents and their typical documents, you don't need to worry about the proper classification anymore. Simply add documents and let paperless-ngx do it's work. With a bit of tweaking, you can scan a whole bunch of documents. But how do you order them? That's where storage paths come in.

These paths define where in your filesystem the documents should end up and under which directory hierarchy. I personally use `{created_year}/{correspondent}/{title}`, which means I have directories like 2024/insuranceXZY/YearlyReport.pdf. If you want to file all tax-related documents in a separate directory, define that under the storage paths section and define a rule to match when a document fits that criteria. The best part is if you change your mind about the ordering, changing the storage paths will automatically move and rename your already scanned documents within the file system without you doing a lengthy `mkdir`, `cp`, `mv`, `rm` dance.

But how do you order them? That's where storage paths come in.

Ready, set, scan

That's all for now. Drag a PDF document that you have lying around onto the web UI and see paperless-ngx start processing it. The Logs section on the left has details on how paperless-ngx choses to match correspondents and other details, which help you fine-tune your match rules. After processing is done, you can find the final result on the Dashboard or in the Documents folder. Continue scanning some more documents. They'll all end up in the `/var/db/paperless/media/documents/archive` directory (if you have not changed it in the `paperless.conf`), followed by the storage paths definition. I hope you'll find paperless-ngx as useful for your documents as I do. I'm always looking forward to the next letter I receive just to scan it with paperless-ngx. Thanks to the people creating paperless-ngx and those who made the FreeBSD port such a great installation experience.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

Contents

2024 Events Calendar

BSD Events taking place through March 2025

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

FOSDEM 2025

February 1-2, 2025
Brussels, Belgium
https://fosdem.org/2025/

FOSDEM is a two-day event organized by volunteers to promote the widespread use of free and open source software. Taking place February 1-2, 2025, FOSDEM offers open source and free software developers a place to meet, share ideas and collaborate.

SCALE 22X

March 6-9, 2025
Pasadena, CA
https://www.socallinuxexpo.org/scale/22x

SCaLE 22X – the 22nd annual Southern California Linux Expo – will take place March 6-9, 2025, in Pasadena, CA.

SCaLE is the largest community-run open-source and free software conference in North America.

AsiaBSDCon 2025

March 20-23, 2025
Tokyo, Japan
https://2025.asiabsdcon.org/

AsiaBSDCon is a conference for users and developers on BSD based systems. The next conference will be held in Tokyo, Japan, March 20-23, 2025. The conference is for anyone developing, deploying, and using systems based on FreeBSD, NetBSD, OpenBSD, DragonFlyBSD, Darwin, and MacOS X. AsiaBSDCon is a technical conference that aims to collect the best technical papers and presentations available to ensure that the latest developments in our open source community are shared with the widest possible audience.

Contents