FreeBSD® JOURNAL
November/December 2021

Open Channel SSD

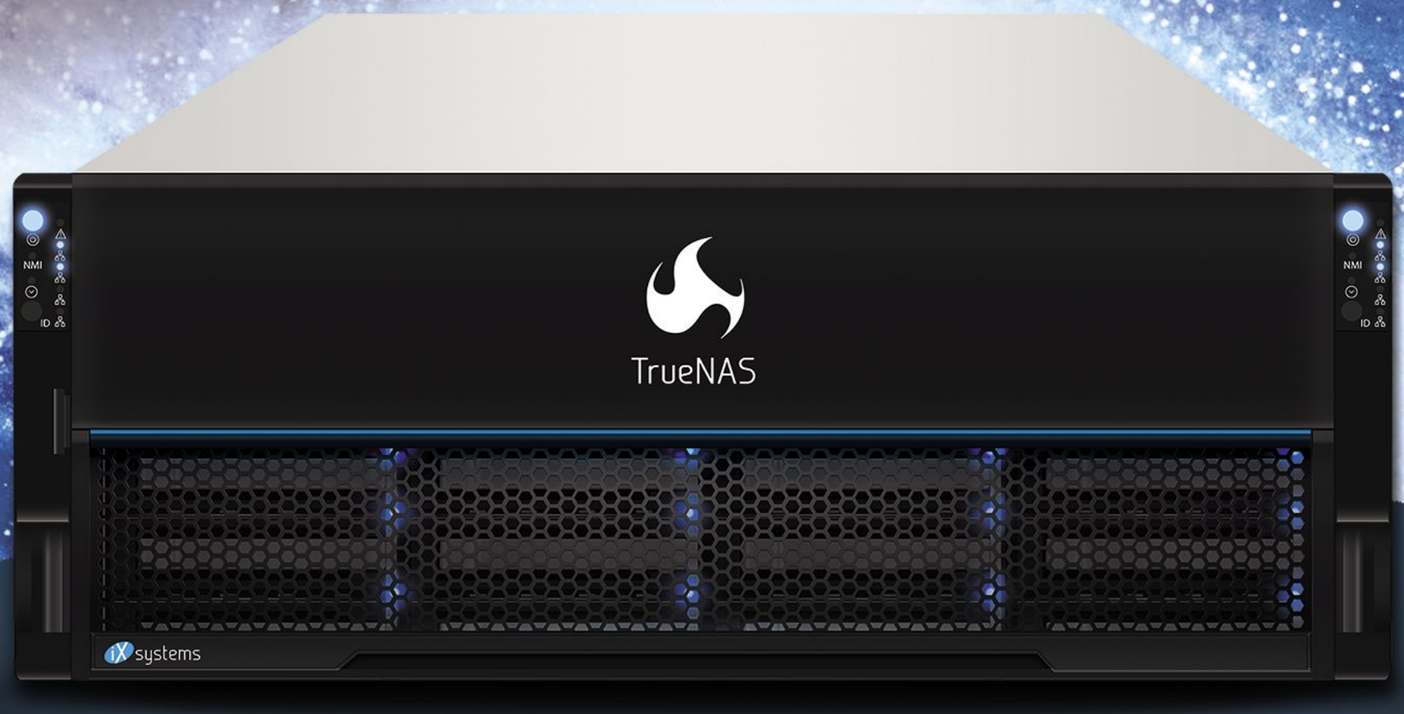
Building FreeBSD Communities

27 Years with the Perfect OS

WIP/CFT:OccamBSD

TrueNAS® M-SERIES
Powerfully Scalable Enterprise Storage

OPEN STORAGE FOR ENTERPRISE WORKLOADS



UTILIZES FLASH-OPTIMIZED ZFS TECHNOLOGY

IDEAL FOR LATENCY-SENSITIVE AND BUSINESS-CRITICAL VIRTUAL MACHINES AND PHYSICAL WORKLOADS.

- PERFORMANCE AND SCALE WITHOUT COMPROMISE
- INTELLIGENT STORAGE OPTIMIZATION
- SELF-HEALING DATA PROTECTION
- UNLIMITED SNAPSHOTS AND REPLICATION

Contact iXsystems to Learn More about what TrueNAS® can do for your business!

iXsystems.com/TrueNAS | (855) GREP-4-iX

iXsystems™

FreeBSD® JOURNAL

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

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) 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: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

On behalf of the FreeBSD Foundation, I'd like to wish everyone a very happy holiday season.

As another year comes to a close, we want to send a big thank you to the Journal's authors and columnists, the editorial board and publishing team for all of their hard work during another challenging year.

Of course, we also want to thank you, our readers! We hope you've enjoyed the FreeBSD Journal over the past year and we look forward to bringing you more high-quality content in 2022.

Deb Goodkin,
FreeBSD Foundation Executive Director

FreeBSD Foundation

FreeBSD® JOURNAL
November/December 2021

Storage

Plus

Open Channel SSD

NAND flash SSDs are widely used as primary storage devices due to their low power consumption and high performance. However, SSD's suffer from unpredictable IO latency, log-on-log problems, and resource underutilization.

BY ARKA SHARMA, AMIT KUMAR, ASHUTOSH SHARMA

Along with the adoption of SSDs, the need for more predictable IO latency is also growing. Traditional SSDs that expose a block interface to the host often fail to meet this requirement. The reason is the way NAND flash works. Typically, within an SSD, flash is divided into chips that consist of dies. A die can execute flash commands (read/write/erase) independently. Dies contain planes which can execute the same flash commands in one shot across multiple planes within the same die. Planes contain blocks which are erase units and blocks contain pages which are read/write units.

The chips can be organized into multiple channels that can independently transfer data in and out between NAND and the flash controller. As it is well known that pages in NAND can't be overwritten, a block must be erased first before its pages can be filled with new data. Blocks have a limited number of times they can be erased. This count is also called the PE(Program/Erase) count, which is different for different types of NAND. As an example, SLC NAND has a PE count of around 100,000, the MLC PE count is somewhere between 1,000 to 3,000, and the TLC PE count range is 100 to 300. Typically, SSDs internally run a Flash Translation Layer(FTL) that implements a log-structured scheme which gives the host an abstraction of in-place updates by invalidating the previous content. FTL's also implement a mapping scheme to facilitate this.

As with any log-structured implementation, fragmented writes occur over time which creates the need for garbage collection (GC) to erase invalidated data and create free blocks. In the case of SSDs, this will require moving valid pages from one block (GC source) to another block (GC destination) and then erasing the source block and marking it free. The entire task is performed transparently to the host which faces the drop in SSD performance as well as the GC operations also affect the lifetime of flash media by writing valid data to GC destination blocks. There are several studies and existing solutions to mitigate this like introducing TRIM/UNMAP which aims to invalidate data from the host in such a way that minimizes the number

of pages GC operation must move. Multi-stream SSD is a technique to attempt to store data in such a way that data with similar lifetimes is stored in the same erase block, thereby reducing fragmentation which, in turn, relaxes the GC to some degree. Workload classification is another approach of reducing fragmentation. Open channel SSD(OCSSD) is another approach to increase predictability and better resource utilization by shifting some of the FTL's responsibility to the host. Typically, the responsibilities of an SSD can be classified into following categories, data placement, I/O scheduling, media management, logical to physical(L2P) address translation, and error recovery.

OCSSDs can either transfer all (Fully host-managed Open-Channel SSD (1.2)) or some (Host-driven Open-Channel SSD (2.0)) of the responsibilities to the host. Our work is inspired by LightNVM which is Linux's implementation of open channel SSDs and Linux specifics have been modified to fit in FreeBSD's ecosystem. As in LighNVM, it is observed that a shared model of responsibilities achieves a better balance without stressing the host to a greater extent. We explore a model of OCSSDs where data placement, L2P management, I/O scheduling, and some parts of NAND management are done by the host. Some tasks like error detections and recoveries are still done on the device side. The OCSSD exposes a generic abstracted geometry of the media (NAND), wear-leveling threshold, Read/Write/Erase timings, and write constraints (min/optimal write size).

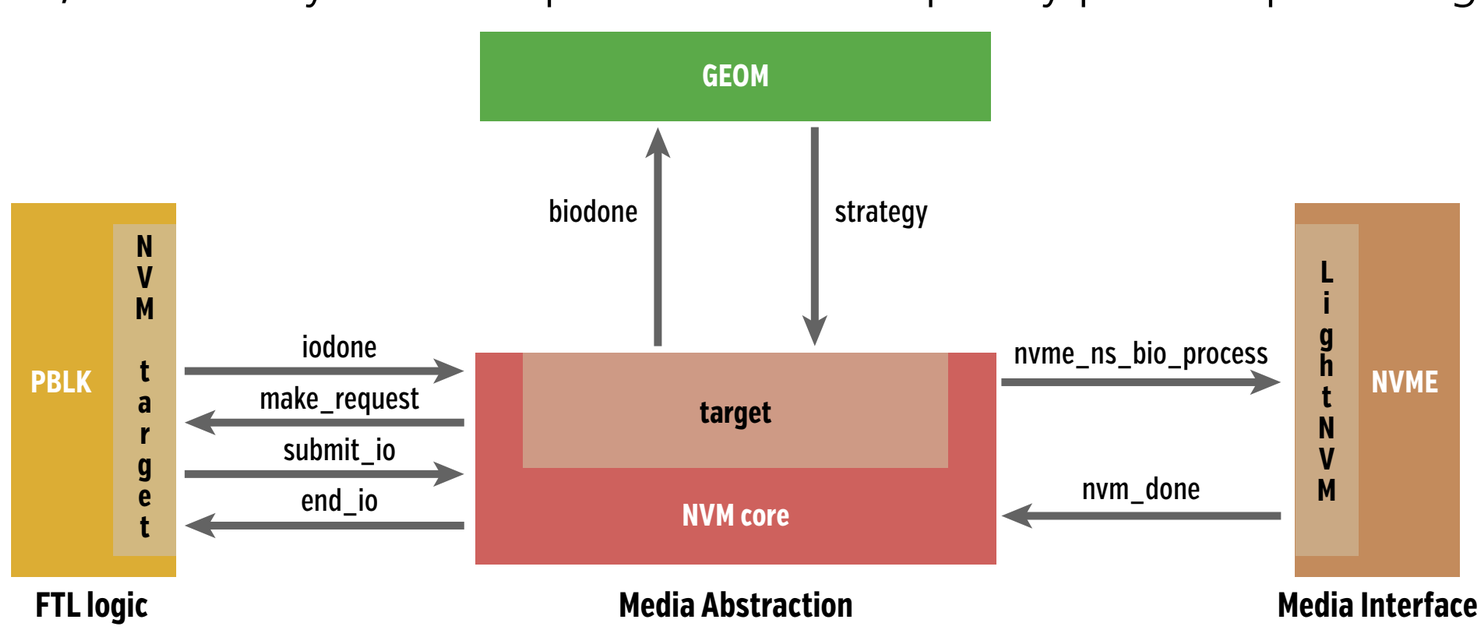
The geometry information typically depicts the parallelism within the underlying NAND media through the number of channels, chips, blocks, and pages. The host can query the state of blocks through commands and get the following information: LBA start address, current write offset within the chunk, and state of blocks (Full, Free, Open, Bad). The drive provides active feedback of chunk health, thus reminding the host to move data from those chunks when required.

Some tasks like error detections and recoveries are still done on the device side.

So far, basic read and write use cases have been tested using FIO. Garbage collection, which is one of the must-have features, hasn't yet been developed due to bandwidth unavailability. All the development efforts have been on QEMU, hence the performance benchmark data is also currently unavailable. Before we received the update about the removal of LightNVM in Linux in 5.15, we planned to on implementing this solution as a GEOM class and with some specific solution where we could consider a custom box with some NVRAM/NVDIMM/PCM as cache and that being coupled with open channel SSDs. But at this point, we have chosen to scrap these ideas. In the future, we look forward to getting involved with work related to NVMe ZNS in FreeBSD.

We have split our work into two components. The FTL part which we call pblk, and the driver which we called lighnvm, keeping the nomenclature similar to LightNVM in Linux. We followed the model of nvd to write the lightnvm driver. The lightnvm driver creates a DEVFS entry "lightnvm/control" which can be used by various tools(nvmecli) to manage the OCSSD device. We have added support for OCSSD devices in nvmecli. The underlying NVMe driver (sys/dev/nvme) initializes the device and notifies the lightnvm driver. The lightnvm driver registers the device to the lightnvm subsystem, the lightnvm system initiates the initialization process and populates the geometry of the underlying media by querying it from the device via the NVMe Geometry admin command(http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf). After the device geometry has been populated, the lightnvm subsystem registers the device along with it's geometry and other NAND attributes.

Once a user initiates the creation of an OCSSD target (via nvmecli), the lightnvm driver carves the requested space out of the OCSSD and creates a "disk" instance for the target which interfaces with geom subsystem. The IOs are intercepted by the strategy routine and forwarded to the pblk subsystem for further processing. The completion of IOs is notified by nvme to the lightnvm, which relays it to the pblk and is subsequently passed up to the geom layer.



We kept the FTL algorithm in the pblk layer largely similar to that of LightNVM. We defined the mapping units to be 4K (also called sectors), which implies that each logical page of size 4K to be mapped with a 4K part of a physical page which is typically larger than 4K. We use nvmecli to carve out the parallel units and create a target. While creating the target, we have an option to choose the target type, which allows us to select the underlying FTL (in case we have more than one) with the target.

As mentioned before, the NAND is divided into chips/dies/plane/blocks. In the context of lightnvm and keeping the terminologies consistent with OCSSD specification, we use the term group for channels, PU or parallel units for chips, and chunk for blocks. OCSSD spec also defines Physical Page Address or PPA which locates a physical page in NAND in terms of group, PU, chunk and page number within chunk. OCSSD compliant devices expose the NAND geometry via the 'geometry' command which is defined in OCSSD specification, and abstracts some of the particularities of underlying NAND media. This allows the user to choose the start and end parallel units which would be part of the target. This also enables the underlying FTL to define 'lines' which is an array of chunks across different parallel units such that the data could be striped to take advantage of the underlying NAND parallelism. This can be achieved two ways: if the target consists of PU's that are connected to different NAND channels, then the data from the SSD controller can be sent to/received from NAND simultaneously. If the PU's of the target are connected to the same channel, the data flow can't happen in parallel. However, once the data flow is complete and flash commands are being executed inside PU's, channels could be utilized for transfer data to/from other PU's. In the case where the target contains one single PU, as expected, we can't have parallelism.

For writing data, we typically write it to a cache and return the success status to geom. We have a writer thread that writes data from this cache to NAND. The size of the cache is computed such that it must accommodate the number of pages that have to be written ahead of a page before data can be read from that page. Suppose the underlying NAND has a restriction that 16 physical pages must be written ahead of a page, and let us say we want to read data from page 10. To be able to reliably read from the chunk, pages up to 26 must be written. Now, if we consider striping, it will take more time to fill those pages, as all the chunks in the line will have same restriction. Also, we must ensure that the maximum number of sectors in a chunk that can be written in a single vector write commands to be fit in cache. The reason for

this is that chunks can have program failure and to do a chunk replacement and retry the write command, we need to hold that much data in the cache. And the cache must be able to hold that much data multiplied by the number of PU's in the target. So, to avoid data loss, we need to ensure these pages fit in the cache. The L2P mapping data is maintained in three places: in the host memory which maps the entire target, at the end of the line which maps only the pages written in that line, and in the spare area of the physical page which contains the data of the logical page. As mentioned before, garbage collection has not yet been implemented due to bandwidth unavailability.

As mentioned above, we have a writer thread that reads the data from the cache and writes it to a NAND device. As we defined the mapping unit of the device to be of 4K size, we have divided the cache and the ring buffer in terms of entries with each entry corresponding to 4K of user data. We store some counters in a ring buffer which act as pointers to dictate the writer thread to pick the right ring buffer entry for flushing the data to the NAND device, acknowledging that flush is successful, and updating the L2P map so that the logical page maps to a physical page instead of to the cache entry. These counters store the cache information such as size of the cache in terms of ring buffer entries (4K), how many writable/free entries are available in the cache, how many entries are yet to be submitted to the NAND device, entries whose acknowledgment is yet to be received from the device, entries whose acknowledgment we got from the device, and entries whose physical mapping needs to be updated from cache address to the device's PPA. So, now with the help of these counters, the writer thread will calculate the ring buffer entries whose data need to be flushed to the device. Now it will check if the number of entries (which need to be flushed to the device) is greater than the minimum write pages data (a.k.a. Optimal Write Size). Let's consider Optimal Write Size as 8 sectors (8 * 4K). So, if the number of entries is less than 8, then the thread will come out and retry in the next run. But if the number of entries is greater than or equal to the 8 (Optimal Write Size), then it will read those entries from the cache. While forming the vectored write command to write data to the physical page, we create a meta-area for each page where we write the LBA of the associated page. This is done so that we can recover the mapping in case of power failure. In the current implementation, we have only one active write end, which means we will write to one single line until it is full or there is a program failure, in which case we allocate a new line and write in that. Once we have all 8 (Optimal Write Size) sectors available in the memory pages (data + meta), we will write the data to the device and update the WP (write pointer) of the device and internally in the NAND pages the LBA information will be updated in the spare area. In the case where a write request gets failed by the device, then we will add those failed IOs to a resubmit queue. Here also, the consumer of the resubmit queue is the writer thread. This time, the writer thread will read only those failed entries from the ring buffer (cache). So, now if the number of entries is less than 8 (Optimal Write Size), then we will add padding (dummy pages) and resubmit the write request to the device.

The L2P mapping data is maintained in three places: in the host memory which maps the entire target, at the end of the line which maps only the pages written in that line, and in the spare area of the physical page which contains the data of the logical page.

For the read request we receive the number of sectors requested to be read, along with the starting sector and the data buffer, encapsulated in a bio structure. Consider a read request for 8 sectors. Now, we read the L2P mapping of the first sector. If the logical address of first requested sector is mapped to cache i.e., the data resides in the cache/ring buffer, then we calculate the number of contiguous sectors whose data reside in the cache. Suppose the logical address of all 8 sectors is mapped to cache. Then we just copy the data of all 8 sectors from the cache to the pages of the read bio structure and call the bio_done to send data back to the above layer (geom).

In another scenario, where the first requested sector is mapped with the device, we calculate the number of contiguous sectors whose data reside in the device and we create a child bio for those contiguous sectors and send a read request to the device with appropiate PPA. Now suppose the logical address of all 8 sectors is mapped to the NAND device. Then we will create a child bio of 8 pages and send the read request for those 8 sectors to the device. Meanwhile, the parent (read) bio will wait until we receive the acknowledgment from the device for the read completion. After this, the read bio which was sent from GEOM will, update it's buffer with the data read in child bio, and call the bio_done to send the data back to geom.

In another scenario, where the first requested sector is mapped with the device, we calculate the number of contiguous sectors whose data reside in the device and we create a child bio for those contiguous sectors and send a read request to the device with appropiate PPA.

Now there is another hybrid case, where partial data resides in the device and the remaining data in the cache. Let's consider an example where the first two sectors are residing on the device, the third and fourth sectors are on the cache, and the remaining four sectors again reside on the device. Now, the first step is the same i.e., we find the mapping of the first sector is on the device, we find the contiguous sector count as 2. We create the child bio of two pages, we send the read request to the device using the child bio. Now, we'll find the logical address mapping of the third sector is on cache and once again we get the contiguous sectors count as 2. So, we read the two appropriate ring buffer entries and copy their data to the read (parent) bio's pages. Once again, we find the mapping of the fifth sector is on the device and the contiguous sectors count is 4. This time we create another child bio to read the remaining four sectors from the device. Now the parent (read) bio must wait until we receive the acknowledgment from the device for both child BIOs. In the end, read IO will get the data from both child BIOs and the cache, and then we call the bio_done and complete the read request.

ARKA SHARMA has working experience on various storage components like drivers, FTLs, and option ROMs. Before getting into FreeBSD in 2019, he worked in WDM mini-port and UEFI drivers.

AMIT KUMAR is a system software developer and currently works on storage products based on FreeBSD. He has been a FreeBSD user since 2019. In his spare time, he likes to explore the FreeBSD IO stack.

ASHUTOSH SHARMA currently works as a software engineer at Isilon. His main area of interest is storage subsystems. In the past, he worked on Linux md-raid.

Building FreeBSD Communities

This is some advice for running different types of community events ranging from small informal meetings to single-track conferences.

BY TOM JONES

FreeBSD is an open source community, and when there is a feature missing, we have the power to add it ourselves. That power isn't limited just to software, we can use it for social events too.

I have been involved in running technology-based meetings and groups for about 13 years. This began in University, when I helped start the student computer science society, and since then, I have run monthly meet ups, a hackerspace with weekly meetings, a tiny festival that was accidentally on Hackaday, and a Friendly Wee Tech Conference in the North East of Scotland.

FreeBSD encompasses all sorts of events. We have user group meetings (the famous NYCBug is a great example), there are semi-frequent hackathons and bugsquashes hosted by the community and user groups, and we have several conferences a year. Conferences range from the BSD DevRoom sub event at FOSDEM to three large BSD-focused events (BSDCan, EuroBSDCon, AsiaBSDCon) and some purely technology-driven events like the OpenZFS developer summit and the BSDCam unconference. All sizes of event are open for you to run, but smaller events that can be put on by one or two people are a good (and realistic) place to start.

If you have never run anything before, there is nothing to fear. I continue to be surprised at how friendly people are everywhere--even the scariest hackerspace in a secret complex in Berlin was full of really friendly people who just wanted to nerd out with like-minded people.

Informal Meetings

On the way into the pandemic, I had one really good idea. I am part of a local group of hackers that meets through a hackerspace plus a few times a year at conferences and festivals, I forced us to meet twice a week. First over Mumble, then Jitsi, and finally through a work adventure based Jitsi chat thing that allowed us to have multiple conversations focused in a single setting. Meeting frequently gave all of us a way to keep speaking to our friends, and many of us became a lot closer in the pandemic than we were before.

These informal meetings were a great way to keep everyone in touch and created a focal point beyond just chatting on IRC. Informal meetings are a great way to judge interest in an area for a FreeBSD user group. They give you focused time where you can meet with interested people--you get to know each other and plan things out. Informal meetings can have other activities bolted on to them too. For many years, the TechMeetUp group I helped organize was a pizza eating session, followed by a talk, and then a trip to the pub.

Regular, informal meetings work best when you get a core group of people to commit to attending. You can use that core group as a kernel to build out from, making the event public and advertising it as much as you can (or want). Without a core set of people, you might find you have very few attendees and things can be awkward. After running events for a number of years, I have come up with a rule that the first meeting will be exciting and new, the second meeting will be much smaller, and the third meeting will start to have people that regularly go to things.

The logic behind this is that it is easy to get attention for a new meeting, but the people who go to exciting new things don't tend to go to regular meetings. The second meeting sees a downturn because those people who were excited, have found something else to be excited by. The second meeting is normally smaller, anyone who heard about your first great meeting has probably planned to come to your second meeting, but then life has gotten in the way, or they just plain forgot. By the third meeting, you start to build a weight of common knowledge and the people who forgot or missed will remember and show up.

This means that if you want to go down the path of organizing regular meetings, then you have to take heart and steel yourself for disappointment, as it is very likely that it will take several meetings for attendance to grow and for the event to find its feet. It just takes time for word of mouth to spread.

In 2022, you will likely start informal meetings with just a regularly scheduled video call. For a call as a meeting, all you need is somewhere to meet and then get people to show up. I wouldn't plan any regular, in-person meetings in 2022 without a fallback plan for when things change.

In-person venues need to allow for people to speak and therefore work best if they are in public places. You are more likely to go and meet strangers if you don't have to go to some hidden room in the basement of a university building. Bars are popular for meetings like this, but I tend to discourage that choice, as it can exclude anyone not comfortable meeting strangers in bars. If a public university space isn't available to you, coffee shops are often a good alternative. Make sure to plan your meeting around the venue's activity schedule. There is nothing worse than getting everyone together to talk about kernel hacking and then something else begins.

Wherever you meet should have power, a source for refreshments and should be easy to get to.

Hackathons/Bugsquashes/Installfests and other Activities Days

In parallel with or as an alternative to regular meetings is the opportunity to run day-long, focused activities. I am very partial to Hackathons and development activities, but you might get the same sort of pleasure from helping others install FreeBSD or build test labs.

Day-long events can be an ego gamble. It is very upsetting to put a lot of energy into planning a hackathon and then only have one or two other people show up (ask me how I know :D).

Day long events benefit greatly from a format (how are you going to approach what you do?) and a theme (what is the core focus of what you are doing?). You can get by with one of these, but I think strongly focused events work a lot better.

This means that rather than having a hackathon, you host a 'Network Hackathon' or an 'Embedded Device' hackathon, which makes the "what you are going to do" and "how you are going to do it" clear. Installfests are a clear idea, but maybe, instead, you want to host a 'build a FreeBSD Cluster Saturday.' I have run un-themed events, and they always require a lot of explanation of the 'what will we do' type.

Virtual events of this form are straightforward to run, you need to pick a time zone and time period that allows the core people you want to turn up to be able to turn up. I have found it works well to get three or four other people to commit to a slot and then others to join if they can. In addition to a time, you need a meeting technology, which can be a video call, a voice chat, or you all can just get together in IRC.

In-person, day-long events require some planning and infrastructure. You have to cater to the needs of people for the duration of the event, so—given the computer enthusiasts that BSD folk generally are—you need a place that has power and Internet as a minimum. You need available rest room facilities, heat in the winter and cooling in the summer or a park—BSD park meets should be a thing!

You don't have to arrange food or refreshments, but you should arrange for a location that makes it possible to get refreshments or give people fair warning that they will have to look after their own basic needs. There was an OpenBSD, multi-day hackathon in a mountain cabin—a several hour hike from any food, but I think the participants were warned before they showed up.

Day-long hackathons and Installfest events can be very successful. You can track how some of them have gone in the past by looking at the 'Event' tag in the FreeBSD commit log. However, if you are the organizer, then you might spend more time managing things and looking after people than you expect--don't plan to get too much done!

Small Conference

The next step after running some single-day, activity-based events is running conferences. I don't think anyone who has run a conference would recommend that you run a conference (myself included). I also know that if you really want to run a conference, then you won't heed this advice.

Conferences are difficult to run because there are a lot more human-based, moving parts. The considerations for single-day activity events are still there, you need power, internet, food, water, and oxygen enough for everyone, but you also have to schedule and manage a lot of people.

The difference is that in a single-day activity, your entertainment is the activity, the network stack can't not show up. When you are running a conference with speakers, there is always the worry that speakers won't show up, that it will run too short or far too long, or at the absolute worst, you'll have speakers and no audience.

You have to manage the venue, speakers, attendees, volunteers and the bits and bytes on the network.

Conferences require a lot of planning and involvement before the event. A conference has a day-long schedule to fill with talks and sessions. These need to come from the community you have built up (which is why it is good to run regular events). You need to solicit presentations and sessions generally, which is normally done with a Call for Papers or CFP. The secret thing you don't see as an attendee is that organizers will also have to solicit talks directly from potential speakers that you know will do a good job.

Conferences need a theme. The major BSD and open source conferences typically have the theme of 'BSD' or 'Open Source.' These are general themes, and while they might have a big audience on a global scale, they probably don't on a local scale. While you might want to run the 'Weimar FreeBSD tmpfs Storage Appliance' conference, you limit who will attend with the level of specificity. There are already a few large BSD conferences in the year, but there is still plenty of room for smaller, single-day events focused on a topic or a geographic region.

I have found that general topics are good, and then you can gently (or not so) encourage your local BSD friends to submit. The Friendly Wee Tech Conference I run has the theme 'Tools and Infrastructure.' We managed to have a talk about building Ham radio infrastructure using HamBSD next to other great talks about interesting tooling, the security of numberplate readers and hosting stuff on NixOS.

Conferences are hard work but very gratifying. If you decided to start a conference, there is a lot of help and advice available from the community. I found Li-Wen Hsu's talk "How to Bootstrap a BSD Conference" very helpful when I was contemplating running one myself.

The community will be able to give you advice on pitfalls to avoid, who to pester for talks, and the time of the year to slot your event into the calendar.

Filling the Gap Between Events

It is good to have a place to bring together like minded people during events, but also between them. Informal community spaces give you somewhere to meet to discuss and plan your next event.

The FreeBSD project already has many of these communities. There are informal community spaces formed around mailing lists, IRC networks and the excellent FreeBSD Discord (you can join with this invite link https://discord.gg/freebsd). These are FreeBSD communities that focus on sub parts of the project. For regional or national activities, you can create similar spaces by forming regional FreeBSD or just BSD groups and meeting in whatever form you can get the most traction.

I love IRC, but there are many that have bad memories from the past or find it too obtuse to use. If you already speak to friends on Telegram or Discord, then you can start forming and planning your meetups using those tools. The way you meet really doesn't matter, only that you meet and organize and create a sense of community.

I Want to Come to Your Event

There are more possibilities for events than I can cover here. They are all very rewarding to run, even if in the buildup they are stressful, and you find yourself worrying for other people and hoping that their talks will be a success.

The building blocks of successful events and communities are consistency and good planning. Nothing appears in the world fully formed though, and if you can find some friends—new or old--to run events with, then you will have a much more enjoyable time (and it will probably be more successful). Even when events have flopped for me, I have still had a good time hanging out with friends and laughing about how our grand plans of success failed. After successful events, I have had the best conversations of my life, where people recount stories from the day from something that I helped pull together. Even with the stress of someone asking you 'when is the next one?' it is an amazing feeling and makes running events worthwhile.

I want to see user groups and meetings in every country, and the only way to do that is to get more people organizing things.

TOM JONES is a FreeBSD hacker from the North East of Scotland and has been involved in community groups and running events for more years than he wants to admit.

27 Years with the Perfect OS

FreeBSD is perfect and I have been using it for a bit over 27 years. But FreeBSD is not the only OS on my desktop.

BY PETER CZANIK

If you are a longtime FreeBSD user, you probably know everything I have to say, and, what's more, you can probably add a few more points. But hopefully, there will be some Linux or even Windows users among readers who might learn something new!

FreeBSD is not just a kernel but a complete operating system. It has everything to boot and use the system: networking utilities, text editors, development tools and more. Why is that a big deal? Well, because all these components are developed together, they work perfectly together! And a well-polished system is also easier to document. One of my favorite pieces of documentation is the FreeBSD Handbook which covers most of the operating system and is (most of the time) up to date.

Of course, not everything can be integrated into the base operating system, and this is where FreeBSD ports and packages can be useful. The ports system allows a clean separation of the base system and third-party software which allows you to install third-party software on top of a FreeBSD base system.

There are tens of thousands ready-to-use software packages to choose from. For example, all the graphical desktop applications are in ports, just as various web servers or more up-to-date development tools.

FreeBSD is flexible. It runs on anything from Raspberry Pi through desktop machines to high-end servers.

FreeBSD is flexible. It runs on anything from Raspberry Pi through desktop machines to high-end servers. You can use binaries provided by the FreeBSD project for both the base system and packages. But you can also recompile everything and carefully customize to your own environment. It's really no wonder so many appliances are FreeBSD-based.

The engineering of FreeBSD is fantastic. All small aspects of the operating system are carefully designed before implementation which results in perfect solutions in most cases, but also means slightly slower progress. If you like to use the latest and greatest hardware as your desktop, it might not yet be fully supported—if at all. This is why many people consider FreeBSD a server OS (including me), even if FreeBSD runs perfectly on the desktop—on older hardware.

The University Years

When I started university 27 years ago, the facility already had a FreeBSD server—a 486 box with 16 MB (not GB) of RAM and an SCSI hard drive. I do not recall the exact version of FreeBSD, but it was still 1.X, as version 2.0 was released only months later. It took many days to download the new version: our whole university had a 64K line at that time.

There was no Linux at the facility, and it became my task to install the first Linux server, which gave me the opportunity to see the early days of both operating systems next to each other. When it came to number of installed systems, Linux quickly won with its "good enough" attitude, as the always perfect FreeBSD was often slower to adopt new hardware or technologies. However, when it came to work, the well-designed, no-surprise implementation of FreeBSD was and still is a lot more pleasant experience—at least for me.

In the first two years, I was a regular FBSD user, but for another sixteen years, I also maintained the server, even after I had left the university. FBSD was famous for its stability, and even long after Gmail became widely available, many students and faculty asked for a username on that server.

Being Jailed!

Fortunately, it was not me, but the web servers! I had a part-time sysadmin job and was running web servers. Serving static pages is not scary but serving PHP pages takes some courage. Luckily, just when I needed to solve PHP serving for customers, jails were introduced to FreeBSD.

At first, I had a single server, and all the jails were created and configured by hand. That is not a huge problem when you can count your customers on one hand. But it becomes quite problematic when you have multiple servers and dozens of customers. So, I introduced a couple of shell scripts, later we introduced central management, LDAP, and a Windows-based management app and almost everything could be automated.

After spending a couple of hours checking the last remaining hosts, I could not find any evidence of a security incidence. FreeBSD jails are fantastic!

While many hosting companies around us continuously reported breaches affecting multiple customers, using a well-hardened, FreeBSD base system and self-build, and hardened jails on top did the trick for us. Of course, even the best hardened jail environment cannot help on a badly configured WordPress instance. Quite a few web servers were defaced but this consistently only impacted a single jail. That's not bad when you have hundreds of jails running on a single server, and at the peak, there were dozens of physical and virtual machines in the cluster. Everything was compiled by me on these servers, and I removed all options from the base system that were not mandatory for running the jails. Software inside the jail was hardened both at compile time and by configuration.

Once I left the company, the same system stayed in use for another five years without any updates. They carefully monitored system logs, and before shutting down the whole system, I got access one more time for an audit. After spending a couple of hours checking the last remaining hosts, I could not find any evidence of a security incidence. FreeBSD jails are fantastic!

The syslog-ng Years

When I joined my current workplace, one of the first tasks was to make sure that Linux distributions and FreeBSD had up-to-date syslog-ng packages. Getting a package updated is much faster if, in addition to asking for it, I also provide an updated package to the package maintainer. So, I learned the basics of FreeBSD ports from the maintainer point of view.

I am not a FreeBSD ports committer, as I only work on a single package, but I work closely with a committer and this arrangement is easier for both of us: I know the syslog-ng part better, so I can change the port to enable new features. He knows FreeBSD ports a lot better than I do and can make sure that the syslog-ng port conforms to the latest recommendations about ports.

Ten years ago, at FOSDEM, I spent part of my time at the BSD devroom. There was a talk about how to extend one of the FreeBSD-based appliances with additional packages. After the talk, I asked how syslog-ng could be integrated. I even gave my business card to the speaker. I was never contacted, but soon after that discussion, I discovered that FreeBSD-based appliances started to feature syslog-ng for logging.

Syslog-ng was known for its portability. Over the years, all the supported commercial UNIX variants disappeared and the developer team focused on Linux. My regular testing on FreeBSD helped to ensure that syslog-ng did not turn into a Linux-only software.

Occasionally, I give a try to FreeBSD on the desktop, but then I give up quickly.

A couple years ago, I learned about BastilleBSD, a jail management system for FreeBSD. Remembering the pain of implementing my own scripts two decades earlier, I really appreciated the features and ease of use that BastilleBSD provided. It now has a template system—similar to Dockerfile in the Linux world—to make creating jails easier. There is also a template for syslog-ng. You can read more about it at: https://www.syslog-ng.com/community/b/blog/posts/running-syslog-ng-in-bastille-revisited

What's Next?

Occasionally, I give a try to FreeBSD on the desktop, but then I give up quickly. I love state-of-the-art hardware but unfortunately FreeBSD does not. As an example, Windows and Linux run without problems on my AMD Ryzen 5800 + nVidia 3070 system while FreeBSD runs only in text mode—and I could not get graphics to work. So, for me, FreeBSD remains a server operating system and I really love it. And, once I have some real servers again—and not just virtual machines for development and testing—I look forward to running FreeBSD on them!

PETER CZANIK started using FreeBSD with version 1.X in 1994. He is an engineer working as open source evangelist at Balabit (a One Identity business), the company that developed syslog-ng. He assists FreeBSD and Linux distributions to maintain the syslog-ng package, follows bug trackers, helps users, and regularly speaks about sudo and syslog-ng at conferences (SCALE, All Things Open, FOSDEM, LOADays, and others). In his limited free time, he is interested in non-x86 architectures and works on one of his PPC or ARM machines.

FreeBSD® JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!

2022 Editorial Calendar

- Software and System Management (January-February)
- ARM64 is Tier 1 (March-April)
- Disaster Recovery (May-June)
- Science, Systems, and FreeBSD (July-August)
- Performance (September-October)
- Topic to be decided (November-December)

Find out more at: freebsdfoundation/journal

WIP/CFT: OccamBSD

BY TOM JONES AND MICHAEL DEXTER

WIP/CFT is a new column shepherded by Tom Jones that will cover interesting, long-running projects and work in progress you might like to know about and/or contribute to. This first installment features Tom in conversation with OccamBSD author, **Michael Dexter**.

What is OccamBSD?

FreeBSD can be compiled many different ways—the FreeBSD operating system has many components that can be built conditionally. Optionally built components are very powerful, they help keep the operating system modular and make it easy to remove features that are not required for a build, whether this is embedded or not.

OccamBSD is a tool for building small, embedded FreeBSD images. Rather than copying individual tools to make custom images or relying on external specialized build tools, OccamBSD is a shell script that uses FreeBSD's build infrastructure to create minimal images with three boot targets in mind—jails, and the bhyve and Xen Hypervisors.

The resulting minimal system contains approximately 400 files in three-dozen directories, and rather than being unrecognizable, provides a glimpse of what 4.4BSD-Lite2 looked like before the modern BSDs were born.

With OccamBSD, we have a unique opportunity to see the majority of build options in action and to explore what a "world" without "buildworld" looks like, providing a minimum userland that allows for a successful login using a bhyve virtual machine.

OccamBSD is a tool for building small, embedded FreeBSD images.

The minimum files required to boot under bhyve, with the exception of the VirtIO drivers, largely represent the code used by all FreeBSD users at all times. This narrow scope is where all auditing, documentation, and computer science education efforts should arguably begin. FreeBSD is otherwise overwhelming to a new student or user.

Why it is interesting?

If you use FreeBSD, this highlights the code you use and is a rewarding exercise.

For a FreeBSD user, OccamBSD gives you an example of a very stripped-down system and

WIP/CFT: OccamBSD

the opportunity to consider if a smaller system works for you. OccamBSD creates a learning environment that is smaller and thus easier to read and reason about compared to a full FreeBSD environment, which might be a great starting point for a course or academic work.

How can I contribute?

OccamBSD is developed on GitHub. Contributions are welcome and you can get involved by testing the tools, writing documentation, or submitting patches.

OccamBSD is happy to take new issues on github, bug fixes by pull request, and reports of success wherever you can find the developers.

https://github.com/michaeldexter/occambsd
https://github.com/michaeldexter/occambsd/issues

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting..

MICHAEL DEXTER is an OpenZFS support provider in Portland, Oregon loves to talk about the bhyve hypervisor and OpenZFS.

PRACTICAL PORTS

Importing a ZFS ZIL via iSCSI

Don't do this at work – like I did

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

I have run our CS department's PostgreSQL server on FreeBSD in a virtual machine for a number of years now with great success. The server is mainly used in the database classes and for projects requiring a database backend. I gave a talk at vBSDcon 2019 about the server which you can find on youtube.

Recently, the department that hosts the virtualization server for this machine changed their underlying storage to Ceph. This added more capacity and redundancy for them by synchronizing the I/O between three different buildings on campus. Around the same time, the database professors devised new lab exercises to let students become familiar with large sets of data. One of the exercises was to create mass data and insert it into a database table, measuring execution time with and without a table index. All well and good, but soon after that particular lab began, professors and students started complaining about poor performance. In some instances, a local postgres installation on students' laptops ran faster than on our server with more CPU and memory. For example, running a "`SELECT COUNT(*)` from `bigtable;`" with roughly 10 million rows took 2 minutes and five seconds on average. A local laptop took about a second. Running the same query a second time took 1 second on the server, proving that it was served from the much faster main memory cache.

Soon after that particular lab began, professors and students started complaining about poor performance.

I started my investigation on postgres, tuning some parameters in `postgresql.conf` and restarting the server. This had only marginal success and people still complained about long insert and query times. Since there was proof that PostgreSQL's default settings had better performance, the problem must have been storage—or I/O-related. When the VM was created, its underlying portion of the Ceph storage was transformed into a ZFS pool, which in turn provided most of it as a dataset for the postgresql database. Since a lot of students were inserting the same data and queried it afterwards, the ZFS ARC was serving those directly from memory. Not all data could fit in the ARC or was evicted from it by other queries. As soon as we hit the disk with writes, the slowdown was noticeable with large data generated by the users.

To confirm our suspicion that the underlying storage was the problem, I picked a server from our big data cluster with 64 CPUs, 384 GB RAM, 4x 512 GB NVMe and installed FreeBSD on it. Then I used "`zfs send`" to copy the dataset hosting the `postgresql` server over to this new server. After starting the `postgres` service, I had a complete copy of the server to play with on beefier hardware. Running the same `COUNT(*)`-queries on the new server proved that they were as fast (if not faster) than a student's laptop, even if they had an SSD. Clearly, performance on our virtual server was to blame. Solving this problem was not that easy though as our IT-department couldn't simply attach an SSD or NVMe to this VM to speed it up. Purchasing and installing it in the server (which meant downtime) would take longer than the remaining time in the semester.

Solving this problem was not that easy though as our IT-department couldn't simply attach an SSD or NVMe to this VM to speed it up.

My idea was to export one of the NVMe disks from the server we just tested on to the VM via iSCSI to create a tablespace. Tablespaces allow the database administrator to define where database objects should be stored on the file system. With iSCSI, storage from a server (called target) can be sent over the network to another machine (called initiator) that imports it. Instead of a network share, the iSCSI protocol lets the storage appear on the importing machine as local block storage—an important difference. This new storage is handled like any other and can be partitioned and formatted with a new filesystem just like a device attached locally.

FreeBSD has iSCSI built-in by default and only requires a few changes in configuration files to set it up. Here is the configuration on the server exporting the NVMe:

First, I created a volume of 200 GB on one of the NVMe drives called iscsi_export:

```
# zfs create -V 200g nvme/iscsi_export
```

Next, I edited /etc/ctl.conf to contain these sections:

```
portal-group pg0 {
        discovery-auth-group no-authentication
        listen ip.address.of.initiator
}

target iqn.dns-name-of-initiator:nvme {
        portal-group pg0
        chap postgres verysecurepasswordgoeshere

        lun 0 {
                path /dev/nvme/iscsi_export
                size 200G
        }
}
```

I changed the ownership and permissions on this file to `root` since it contains a cleartext password.

Upon reboot of the server, the iSCSI initiator should be started again, so I put `ctld_enable="YES"` into `/etc/rc.conf`:

```
# sysrc ctld_enable=yes
```

To activate the initiator, I started the service:

```
# service ctld start
```

This mostly follows the descriptions of the iSCSI section in the FreeBSD handbook.
Over on the VM importing the storage disk, I put the following into `/etc/iscsi.conf`:

```
        TargetAddress   = ip.address.of.initiator
        TargetName      = iqn.dns-name-of-initiator:nvme
        AuthMethod      = CHAP
        chapIName       = postgres
        chapSecret      = verysecurepasswordgoeshere
}
```

Since the `postgres` users log into this server via SSH to run postgresql's commandline utility `psql`, keeping the password in this file secure from prying eyes is important. A `chmod` of 0700 followed by a `chown` with owner and group set to `root` and `wheel` solves this. An entry to `/etc/rc.conf` is necessary to initiate the storage import upon reboot (more on that later):

```
# sysctl iscsid_enable=yes
```

Next, we can import the disk by starting the service:

```
# service iscsid start
```

Upon successful import, a new device (probably `da0` or similar) appears in `/dev`. A separate ZFS pool was created on it:

```
# zpool create nvme_ts /dev/da0
```

Yes, this is not redundant, but for our benchmarking purposes, it was sufficient enough. On the postgres side, logged in as the database superuser in `psql`, the tablespace is defined by this statement (see https://www.postgresql.org/docs/current/manage-ag-tablespaces.html for details):

```
psql#>CREATE TABLESPACE nvme LOCATION '/nvme';
```

Checking the access permissions again, but after the command is complete, the postgres database users can use the tablespace and put database objects (like tables) on it. Either by explicitly defining where the data should be stored:

```
psql#>CREATE TABLE nvme_powered_table(i int) TABLESPACE nvme_ts;
```

or setting the tablespace as default:

```
psql#>SET default_tablespace = nvme_ts;
```

With this new configuration (clearing the cache first) and reload of a fresh batch of 10 GB data into the `nvme_powered_table`, the database insert performance on the VM improved to 7 seconds (from its original more than 2 minutes). Having an NVMe tablespace is certainly nice, but we went further. This is also when trouble started...

Not Thinking Things Through

We decided to use the exported storage as a ZIL to speed up the slower writes on the Ceph-backed pool. The ZIL would acknowledge to the application (the database) that the writes have reached stable storage and would later write to the slower disk while the database continued its work. A ZIL usually does not have to be big, as the data in it gets quickly evicted. We reduced the amount of exported disk space in the `iSCSI-initiator` and re-imported the disk in the database VM. Then we configured the iSCSI disk as a ZIL with the following command:

```
# zpool add pgpool log da0
```

The device showed up and worked immediately. I/O on the pool was now quickly acknowledged as "written" and the database could continue without waiting. The ZIL trickled the write

requests to the slower Ceph storage. This boosted the database performance a good degree and we went into production.

Don't Try This At Work

What I did not realize at the time is how badly this integrates into the boot process. When FreeBSD with a ZFS-only filesystem boots, it tries to detect all the storage devices contained in the pool. At this point during the boot, the network is not yet completely configured and thus no iSCSI services are available to import the external device. When it comes to the ZIL device, it turned out that ZFS requires this to boot properly and complains about a missing disk in the pool. The boot process is halted at this early stage, even though the main vdev of the pool was available (but ZIL wasn't). You can imagine that this does not go well on a production server and only the management console of the server itself revealed what was going on.

Note that this can happen in two ways: either the iSCSI target (the server exporting the storage) goes down or loses connectivity, or the initiator (the client importing the device). Seasoned sysadmins know that during a typical day, interruptions of this kind can happen, often unannounced and unexpected. It is only a matter of time when this would have happened and now that it did, we needed a way to fix it--quickly.

Rebooting the server with a FreeBSD ISO image and selecting the Live-CD option in the installer was next. From the Live-CD's shell environment, we could mount the pool with the missing ZIL device on `/mnt` like this:

```
# zpool import -R /mnt -m pgpool
```

After the import was finished, we could inspect the remaining devices in the pool:

```
# zpool status
```

The output showed the missing cache device with its long unique numeric identifier. The next action was to remove the ZIL device from the pool:

```
# zpool remove pgpool <verylongnumericidentifier>
```

Typing in the long identifier instead of the much shorter device name serves as a good reminder to avoid this situation in the future. Once this had been done and the output of `zpool status` confirmed the removal, the pool was exported again. This is usually done upon reboot, but we did not want to take any chances.

```
# zpool export pgpool
```

After the machine rebooted, we were happy to see it complete the boot this time and gave us our familiar login prompt back. Disaster averted, but the underlying performance problem was still present.

Happy Ending

Clearly, the iSCSI export is too risky and could fail again. Although we did run like this for a whole semester, Murphy's law will let that happen at the worst time of night when sysadmins are supposed to be sleeping. Certainly, a script could safely remove the ZIL from the pool upon every shutdown. But power losses or crashes on both machines involved in the iSCSI export are not covered by this. Luckily, our IT department was finally able to provide us an SSD-backed Ceph storage as an alternative for this machine. The import is similar to iSCSI but is more stable and less prone to crashes.

Ceph on FreeBSD works, but importing this device proved to be...interesting. Ceph supports this kind of import on FreeBSD only via geom_gate, which is similar to iSCSI. After installing the `net/ceph14` package, the `rbd-ggate` command was available (rbd is the Rados Block Device of Ceph). The man page rbd-ggate(8) is rather short, listing only a few commands and switches. I was a bit worried at first as it dates back to 2014. With no recent updates, chances are that support could have been broken by a change on newer FreeBSD versions. This was unfounded, however. We only had to deal with some of the differences in how Linux and FreeBSD deal with commandline arguments. On Linux, a `--option` is used, whereas on FreeBSD a single `-option` is more common. The command initially looked like this:

```
# rbd map -t ggate volumes/ssdvolume
```

The `volumes/ssdvolume` is the path to the SSD ceph storage given to us by the IT department and maps a geom gate device upon successful import. The command failed because the `--id` of the user doing the import was not provided (username and password protects this storage from unauthorized imports by others). Here's where the mixing of single and double dashes became problematic, as the Linux-based rbd command refused to mix the `--id` with the single `-t` parameter. We found a solution by providing the ID as an environment variable like this:

```
CEPH_ARGS='--id postgresdb' rbd map -t ggate volumes/postgresdb
```

With this combination, the command ran successfully and told us

```
ggate0 created
```

This was confirmed by looking at `/dev/ggate0`. This is the imported device from Ceph, on which we could now create a new ZFS pool:

```
# zpool create ssdpool /dev/ggate0
```

Remembering what we learned from last time, we tried rebooting the machine to see how it coped with this device during boot. We were happy to see that the system did reboot without issues, and we could then re-import this new pool using:

```
# zpool import ssdpool
```

We could then create a little startup script that was executed once the system finished booting to automatically re-import this pool and activate the postgres database on it. The postgres database was cloned by snapshotting and zfs sending from the old, slower pool and receiving it on the faster ssdpool. This works quite well, and the performance difference is definitely noticeable. As I write this, the first student groups are already working on it (without their knowledge) and I have not received any complaints yet.

Lessons Learned

Measure where performance is lost and isolate the bottlenecks. Use different test cases to confirm any hypotheses about where the problem might be located. Test things before putting them into production. Ensure solutions survive a reboot of both the exporting and importing machine when dealing with storage coming over the network. Keep a FreeBSD Live-CD ISO image handy to fix things in case of disaster. Document every step and command for yourself and your peers to have them available when people are breathing down your neck while your phone is ringing by users demanding the functionality back (when already in production). Be ready to experiment and try out new things. Lastly, rely on FreeBSD to be a solid foundation in the storage space with its flexibility and options it provides for combining different solutions.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. 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.

WeGet letters

by Michael W Lucas

Dear Worst Columnist in This Journal,

My company has rack upon rack of storage servers. When I started as a sysadmin, nine-gigabyte drives were common. Now, each drive is multiple terabytes, and we're building arrays that aren't just petabytes but exabytes. We're building a data center for multiple zettabytes. What can any company be doing with all this storage?

—It's not bootleg movies, I checked

Dearest Bootleg,

That really is the question, isn't it? We have vast amounts of data storage capacity, and yet a measurable fraction of the world's manufacturing capacity is dedicated to producing more. We have entire container ships full of SSDs adrift in the Pacific Ocean, eagerly awaiting that glorious moment when they finally get to dock and offload all that blank storage. Organizations like yours order disks by the pallet. What can anyone do that generates so much data that they need yawning chasms of storage?

Unless you're working in exciting big data fields like bioinformatics or ripping holes in the universe at the Large Hadron Collider in the hope that your favorite incarnation of The Doctor will show up and tell you to stop, most of those petabytes are either data that you shouldn't have, obsolete data, or data that nobody will take responsibility for throwing away.

Organizations have a horrible habit of keeping every scrap of data that they get, even when possession of that data poses an appalling risk to the organization's health or existence. How many data breaches have you seen where a company leaked, say, Social Security numbers or credit card numbers or biological analyses of nose hair samples, and you immediately asked yourself why the company had that information in the first place? It's like a disease. Perhaps a C-level officer made the decision to gather this data, or maybe it was an unsupervised web designer infuriated with his manager who decided that the database could handle one more column. The decision to collect that kind of data comes easily but getting rid of it demands meeting after meeting. Given the choice between calling that meeting and playing NetHack, most of us cuddle our keyboards. After all, if the data gets stolen, you probably won't be the employee chosen for sacrifice at the Temple of Mass Media—and if you are, you can use that symbolic execution as a point on your resume demonstrating that you are experienced and land a better job.

Then there's the old data. Last year's expense reports. 1993's expense reports. Spreadsheets containing estimates of expenses before replacing the leaky roof on the building that the previous CEO moved the company out of. A folder labeled "blackmail photos," and while they're certainly incriminating, especially the one with the chocolate fountain and the barbeque tongs, no-

body currently employed recognizes anyone in any of the photographs. These documents are an archive of the organization's history. When the time comes that your friendly little real estate firm serendipitously discovers a cure for cancer and the CEO decides to hire a ghostwriter to chronicle the organization's amazing history, some poor bastard is going to have to dig through all those fossilized layers searching for evidence that can be misconstrued to demonstrate brilliance.

All this data could conceivably be used—one day—if a bizarre, never-to-be-repeated series of coincidences should strike that makes the long-dreaded astrological alignment of Jupiter, Pluto, and Halley's Comet with Polaris seem commonplace. It won't happen, but it could. The most pernicious data, though, is cruft that can never possibly be used, but nobody will take the responsibility to discard. Old database backups that might, possibly, be necessary. Old databases that can never be useful under any circumstances, because the software to read those backups runs only on SCO UNIX and even NetBSD has dropped *that* binary compatibility layer. Realistically, even though you have the skills to crack open what is almost certainly a bunch of comma separated values with a weird file extension, if anyone asked, you'd be much more likely to laugh and say there is no way to read that data than actually break out file(1) and strings(1) and pipe the whole mess into Perl and produce a handy Excel-compatible spreadsheet. Images of laptop hard drives from employees who fled in 2001, because their manager declared that the next person to fill that role would need that employee's files—and then refused to release those files to said replacement. Test spreadsheets that were discarded as failures. Accounting files that were eradicated for excessive honesty and replaced with IRS-friendly versions. As your organization ages it will acquire more and more of this detritus, filling drive after drive, until nobody is willing to either look at the data or take responsibility for discarding it.

We want our systems to be clean!

We want our storage tidy and elegant.

Any reasonable sysadmin finds this offensive. We want our systems to be clean! We want our storage tidy and elegant. Lugging around petabytes of the wreckage—or worse, backing up said petabytes—violates our proprieties. Many of us itch to attack this debris, discarding what is unneeded and organizing the rest. I'm forced to call out System Administration Rule #18 here: *it is cheaper for the organization to buy more storage than to pay you to clean out existing files*. Think back on those old 9-GB hard drives. Remember how many thousands or millions of files they could hold? Opening each file, assessing the contents, and deciding if it merited survival or should be cast into the outer darkness was an overwhelming task. Those drives were minuscule by today's standards. This isn't a modern problem; my first hard drive was 20 MB, and it contained more files than I could cope with. Worse, many of those files still exist. Every system I get has more hard drive capacity than the last. I'm never quite sure what files I will need, so I copy everything from the old hard drive into an archive folder on the new system. The only thing I don't have is the code for the Sinclair ZX80 maze game that Young Lucas enjoyed playing, and I'm sure that's available somewhere on the Internet. Destroying these files is a high-risk, low-gain game for any manager. If successful, the organization can avoid spending a few hundred bucks on storage. If unsuccessful, some of those antediluvian files turn out to be of vital importance and the manager's career is over. Even options like archiving to tape pose risks. While every true sysadmin archives everything in an open source format like tar, many organizations insist on using "Enterprise Backup Systems" with an appalling habit of obsoleting support for old formats.

With ample opportunity for self-humiliation and minimal potential reward, nobody is going to tackle this morass.

You cannot solve this problem.

You *can* avoid contributing to it.

Consider the data you, personally, are responsible for. Are you following your organization's data retention policy? If your organization has no data retention policy, establish one yourself. It can be as simple as telling your team, "Hey, I want to discard all logs on these systems after 60 days. Does anyone have a problem with that?" Perhaps you'll need some data longer, and other data you can throw away after a week. A good data retention policy can even keep you out of court — logs that do not exist cannot be subpoenaed. You don't want to go to court. Court is not fun, and neither lawyers nor judges understand sysadmin humor.

Or you can buy even more storage and stop worrying.

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

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *TLS Mastery*, and *$ git sync murder*. His *DNSSEC Mastery* and *Domesticate Your Badgers* should be out in early 2022, despite earnest requests from the Humane Society. For a complete list of everything he's done, query his SNMP table. Submit your questions to letters@freebsdjournal.org.

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

Events Calendar

BSD Events taking place through March 2022

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 2022

February 5-6, 2022
VIRTUAL
https://fosdem.org/2022/

FOSDEM is a two-day event organized by volunteers to promote the widespread use of free and open source software. Taking place, February 5-6, 2022, FOSDEM offers open source and free software developers a place to meet, share ideas and collaborate. Renowned for being highly developer-oriented, the event brings together some 8000+ developers from all over the world. The conference will once again be held virtually.

SCALE 19x

March 3-6, 2022
Pasadena, CA
https://www.socallinuxexpo.org/scale/19x

The 19th annual Southern California Linux Expo — will take place on March 3-6, 2022, at the Pasadena Convention Center. SCALE is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area.

FreeBSD Fridays

https://freebsdfoundation.org/freebsd-fridays/

Stay tuned for new episodes in early 2022.
Past FreeBSD Fridays sessions are available at: https://freebsdfoundation.org/freebsd-fridays/

FreeBSD Office Hours

https://wiki.freebsd.org/OfficeHours

Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.
https://www.youtube.com/c/FreeBSDProject.