Profiling the FreeBSD kernel boot

Colin Percival
Tarsnap Backup Inc.
Email: cperciva@tarsnap.com

***Abstract*—We describe work we have done to profile the FreeBSD kernel boot — both adding instrumentation to the kernel to collect data while the system is booting, and converting the resulting timestamp records into a graphical visualization of where time is spent in the boot process. We show results from two systems and highlight some places where it is clear that the performance of the FreeBSD boot process can be improved.**

I. Introduction

In mid-2017, the author bought a new laptop to run FreeBSD. Over the following months spent attempting to get the integrated Intel video chipset to work properly in FreeBSD, and a large number of ensuing reboots as new kernels were attempted, a question emerged: Why did rebooting FreeBSD take so long, anyway?

While a large amount of effort has been spent on optimizing the performance of FreeBSD *while it is running*, the time spent booting the system had never received much attention, despite being a frequent irritant to users. Indeed, as soon as we started investigating this, it became clear that there was a large amount of "low-hanging fruit" of time spent during the boot process which could be easily improved. Nevertheless, in the interest of completeness (which proved very important, since there turned out to be major improvements available which we had not noticed prior to commencing this work) and ensuring that future opportunities for performance gains could be easily discovered by other developers, we decided to pursue a systematic approach to collecting and visualizing information on how time was spent during the FreeBSD kernel boot.

The time spent booting FreeBSD can be divided into four parts:

1) System firmware (BIOS, UEFI, etc.) initializes the system and loads the boot loader from disk.
2) The boot loader loads the FreeBSD kernel[1].
3) The FreeBSD kernel starts additional CPUs, discovers attached devices, initializes hardware, mounts filesystems, and spawns the `init` process.
4) The `init` process launches `/etc/rc` and a set of userland startup scripts run.

Of these, we are concerned in this paper only with the third: The time between when we start running code in the kernel and when the `init` process is launched.

[1]Sometimes this step involves multiple boot loaders with increasing degrees of sophistication; in any event, one of them eventually loads the FreeBSD kernel.

II. Prior work

Most FreeBSD developers, when the need for profiling arises, immediately reach for DTrace [1]. This is an incredibly powerful tool for performance analysis, but with its power comes some limitations: DTrace makes use of traps, memory allocation, and even the kernel scheduler — none of which are available when we first start running code in the kernel. While DTrace can be a very effective tool for profiling the portion of the kernel boot which occurs after DTrace is running[2], we wanted to profile as much of the kernel boot as possible, so DTrace was insufficient for our purposes.

Users of the Linux systems will be familiar with the basic boot profiling functionality built into the Linux kernel: Lines emitted by the kernel to the system log are prefixed with a timestamp. This has the great advantage that it can be immediately obvious to end users if one step in the boot process is taking an inordinate amount of time, but it has some significant disadvantages as well. First, timestamping kernel log output means that only those points in the boot when output is printed will have timestamps recorded; the boot time `initcall_debug` option adds extra logging of the `initcall` functions called to initialize many parts of the Linux kernel, but these still provide only a coarse level of granularity. Second, the timestamps which are printed become meaningful only after the necessary initialization has been performed to be able to convert system cycle counts into wallclock times; at the start of the boot, every line is marked as having timestamp `0.000000`. Finally, while additional timestamping points — after the clock initialization has been performed — can be easily added, the act of printing additional lines to the console (with or without timestamps) can affect the time spent, making this cumbersome as a mechanism for further investigation of boot performance.

Once data has been collected, there are a number of tools available for visualization; most notably, Flame Graphs [2] are commonly used for viewing data collected by DTrace, while bootgraph [3] is commonly used to visualize the time used in the Linux kernel by `initcall` functions[3].

III. Timestamp logging

To avoid the limitations of DTrace and Linux's kernel log timestamping, we adopt a hybrid approach. Like

[2]In particular, DTrace has support for "boot-time tracing" whereby a set of directives is prepared and the boot loader is instructed to load them along with the kernel the next time the system boots.

[3]A more sophisticated tool, systemd-bootchart [4], is also available from the same author for visualizing the userland boot process.

DTrace, we record system cycle counts, as returned by the `get_cyclecount` function[4] into a buffer which is not displayed in any way but rather retained to be dumped by userland after the boot completes; but like Linux, we fix at compile time the points at which timestamps are recorded rather than using hot-patching to insert probes at run time.

The infrastructure used for recording timestamps lies in two files, `sys/tslog.h` and `kern/kern_tslog.c`. The `C` file exposes two interfaces: A function `tslog` which records a timestamped event, and a FreeBSD sysctl [5] which allows the buffer of recorded events to be read by a userland process. Since `tslog` must be able to run before kernel locking primitives are available, it uses a very simple synchronization mechanism: When a record is to be logged, a "slot" is reserved in a buffer (which is defined at compile time with a fixed size) by atomically incrementing a record-number variable; the record is then filled in with the necessary data.

When the buffer is read out via the sysctl interface, the same record-number value is used to indicate when to stop copying out the buffer; this exposes a theoretical race condition, since the sysctl could read a record which has not been fully written, but since we will be reading the buffer only after the boot process completes and will ignore any records from after that point, this race condition is of no practical relevance.

The `tslog.h` header file serves two purposes: First, to expose a more useful interface to the `tslog` function — most notably, so that the lines `TSENTER();` and `TSEXIT();` are sufficient to annotate the start and end of a function — and second to allow timestamping to be easily turned on and off at compile time. When compiled with the default options, the macros used for timestamping the FreeBSD kernel are replaced with nothing; only when the kernel is compiled with the (non-default) `TSLOG` option do the macros result in `tslog` being invoked to record timestamps.

Each timestamp record consists of five fields: The current thread ID; the type of record; pointers to one or two strings; and the system cycle count.

IV. Function tracing

Most of the instrumentation we have added to the kernel is to record the start and end of functions. Unlike DTrace, which understands enough about machine architectures to inspect the call stack and translate program counters to function names, we directly record when we enter and exit functions of interest.

The first set of functions we annotated this way are major steps which comprise the FreeBSD boot; while little time is spent in these functions directly, recording them provides a useful reference when considering finer grained steps in the boot process. These are the amd64 machine-dependent initialization function `hammer_time`[5]; the main machine-independent initialization function `mi_startup`, which runs as "process #0" and once completed becomes the "swapper" process; and the `start_init` function, which — as the name suggests — is responsible for launching the `init` process which takes over the boot procedure, continuing into userland. We also annotated two additional functions which are called from within `start_init`: The `vfs_mountroot` function, which is responsible for mounting the root filesystem, and the `vfs_mountroot_wait` function, which is responsible for waiting until the system is *ready* for the root filesystem to be mounted.

At the other extreme, we annotated the various machine-dependent `DELAY` functions, which are used to wait for a specified number of microseconds (typically to allow hardware to become ready during initialization routines when it is not yet possible for a thread to yield the CPU to the scheduler and resume later), and the `_vprintf` function, which is used by the kernel `printf` and `log` routines to print output to the console. These functions are used from many places in the kernel and turn out to be collectively responsible for as much as 80% of the time spent booting.

Between these two extremes, there are three important sets of functions which we were able to instrument automatically by adding code to the frameworks which call them.

A. SYSINITs

FreeBSD `SYSINIT` functions are like Linux `initcall` functions: They are created by placing symbols into a special ELF section, and `mi_startup` iterates over those symbols calling the appropriate functions in turn[6]. Since `SYSINIT`s are defined by a C macro, we simply redefine the macro to instead call a "wrapper" function which logs the function entry/exit and calls the original target function.

In our first attempt, this caused a problem with FreeBSD's linux KPI compatibility shim: `SYSINIT` declarations were placed within functions, at which point C does not permit a new function to be defined[7]. To remedy this problem, we switched instead to defining a single wrapper function in the C header file rather than instantiating a new wrapper function every time the macro was used.

[4]On x86 CPUs, this is provided by the `RDTSC` instruction.

[5]Other architectures will need their machine-dependent initialization functions annotated as well, but we did not have access to all the platforms upon which FreeBSD runs in order to test said changes. Since the usual mechanism for determining which thread is running is unavailable when the machine-dependent initialization code starts running, annotating these functions is slightly more complex (and thus error-prone) than simply adding `TSENTER();` and `TSEXIT();`.

[6]There are subtle differences — FreeBSD's `SYSINIT` mechanism allows for a far more fine-grained ordering to be specified, for example — but for our purposes the two systems are the same: Lists of functions which are called during kernel startup.

[7]Even before our changes this produced (upon macro expansion) somewhat questionable C code — identifiers were being created with block scope and no linkage, and then declared as being placed into a specific ELF section — but FreeBSD's build toolchain was able to cope with this odd behaviour.

B. *Devices*

After some basic kernel infrastructure has been initialized, the FreeBSD kernel proceeds[8] to examine the system buses to discover all of the hardware devices available. Drivers in turn are invited to probe for hardware which they are able to manage via their respective `DEVICE_PROBE` methods; for each device found, FreeBSD then selects a single driver (based on the strengths of matching returned by the probe functions) and calls its `DEVICE_ATTACH` method.

In order to instrument these function calls, we add prologue and epilogue code to the inline functions which look up and invoke these two methods. (There are many other methods which could be instrumented the same way, but it seemed unlikely that suspend, resume, shutdown, or detach methods would be of any relevance during the boot process.)

C. *Filesystems*

The final set of functions we found to be of note during the boot process are those used for mounting filesystems. To that end, we modified the `VFS_MOUNT` macro (which invokes the appropriate `vfs_mount` function for the filesystem type in question) to make function entry/exit timestamping calls.

V. Boot holds

Recording when we enter and exit certain functions allows us to build timelines of stacks[9] for each kernel thread; but this alone does not allow us to ascribe blame in the case where one thread is blocked waiting for another thread to finish performing an operation. It might be possible to handle this by gathering information from the kernel scheduler about when threads are scheduled, sleeping, and waking up[10], but since we were interested only in the FreeBSD kernel boot and do not expect to see any significant lock contention, we opted instead to instrument the specific locations in the boot process where the boot is blocked waiting for another thread to complete:

1) The `intr_config_hooks` `SYSINIT` waits until any hooks which were registered via the `config_intrhook_establish` function have been released; this is typically used by devices which need interrupts enabled in order to complete their initialization, and allows them to prevent the continuation of the boot process until their initialization is complete.
2) The `g_waitidle` function waits for the GEOM event queue to be empty; this ensures that all of the attached disks have been "tasted"[11] before the kernel proceeds to attempt to mount the root filesystem.
3) The `vfs_mountroot_wait` function (in addition to calling `g_waitidle`) waits for holds registered via the `root_mount_hold` function; among other things, this is used by the USB subsystem in order to ensure that the USB device tree has been fully explored (and disks discovered) before the system attempts to mount the root filesystem.

We refer to these collectively as "boot holds", and instrumented them by recording every time a boot hold is established or released, and every time a thread waits (or finishes waiting) for boot holds to be released.

We also instrument the creation of new threads, recording the name given to the thread (e.g., `g_event`); these thread names will be used later to identify the code being blamed for holding up the boot process.

VI. Visualization

After the system has finished booting, we use the newly-created `debug.tslog` sysctl to dump the buffer of timestamped events to userland. We split the data according to thread IDs, and then for each thread we convert the series of "function entry" and "function exit" records into a series of timestamped stacks for each thread.

We then identify the two threads we know that we will be interested in — the thread which runs the machine-dependent initialization and `mi_startup`, and the thread which runs `start_init` — by simply noting when those functions are entered, and pull their stack histories together. These together cover the kernel boot process and describe the time spent booting; but they do not provide any insight into the time spent during boot holds.

To fill in those gaps, we look at the periods when those threads are waiting for boot holds to be released. In each case, we then look at how other threads are acquiring and releasing boot holds of that type; we "blame" the thread which was the *last* to release its boot hold for the period of time ending when its boot hold was released and starting at the latest of (a) when the boot hold wait started; (b) when the boot hold was acquired; and (c) when the thread which released the boot hold was created. This is only a rough heuristic — it's possible for a boot hold to be established and then for work to be done in several different threads before the boot hold is finally released — but in our experience it works well enough to give a clear indication of where time is being spent. Having identified these, we then splice together the stacks to create pseudo-stacks showing e.g., that `start_init` called `vfs_mountroot`, which called `vfs_mountroot_wait`, which is waiting for `THREAD g_event`, which called `pkcs5v2_genkey`[12].

At this point we have a series of timestamped (pseudo)stacks and simply want to convert them into a more human-viewable form. Unfortunately, since flame graphs sort stacks into lexicographical order, they lose the information we have about the order in which stacks are encountered; while that is of little

[8]The process of probing and attaching drivers to devices mainly occurs within the "`configure2`" `SYSINIT`.

[9]At least, the elements from stacks which have been instrumented.

[10]FreeBSD's `schedgraph` tool displays this and other information.

[11]GEOM tasting is the process by which stacked storage structures are parsed, e.g., to determine that a raw disk is partitioned into several parts, is one part of a mirror, or contains an encrypted disk.

[12]We have not instrumented the `pkcs5v2_genkey` function in the public FreeBSD source tree, but during development we instrumented it locally; this is the function used by the GELI encrypted disk system to convert a passphrase into a derived encryption key, and to ensure the security of the disk encryption it is necessary to spend a nontrivial amount of time here.

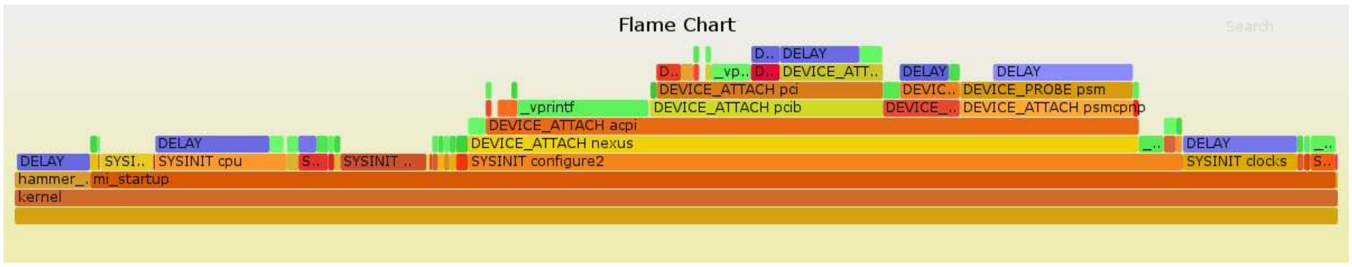

Fig. 1. FreeBSD 11.0-RELEASE on EC2 c5.4xlarge instance

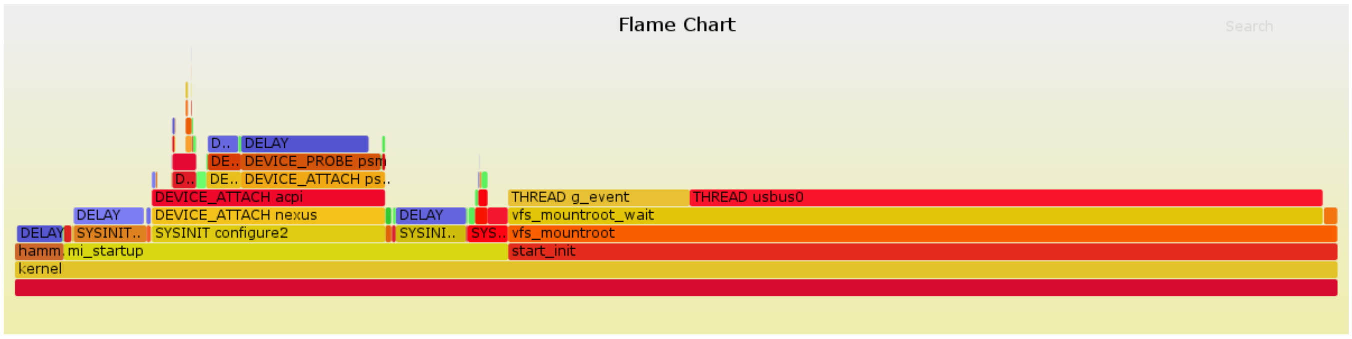

Fig. 2. FreeBSD 12.0-CURRENT on the author's laptop

importance for monitoring the steady-state of running systems (which is where flame graphs are most often used), we believe that this information is likely to be of great importance to understanding the FreeBSD boot process.

However, we do not want to simply display stacks exactly as they are logged either; there are many stacks which recur — device probe routines are called multiple times as the devices attached to a particular bus are uncovered, and some drivers make repeated `DELAY` calls separated by sending commands to hardware and reading responses — so aggregating identical stacks is still useful. To that end, we add up the number of clock cycles spent in each unique stack, while keeping track of the "average cycle count" during the times when the kernel is running in each stack; we then perform a tree traversal of the list of stacks, ordering subtrees based on the "average cycle count" spent within each, in order to place stacks in roughly the order in which they were originally encountered.

While this does not perfectly retain information — if a function repeatedly cycles between calling two other functions[13], that information is lost as we will display the block of time spent in one child function as being entirely before or entirely after the other — in practice we find that this serves as a good compromise between retaining all the available information and eliding enough to allow the most important information to be easily "eyeballed".

Finally, we pass the series of stacks (which now, rather than having timestamps attached, merely have recorded the total number of cycles spent within each stack) to a version of Brendan Gregg's flamegraph code which has been modified to operate in "flame chart" mode[14] — displaying the stacks in the order provided rather than re-sorting lexicographically. In order to highlight potential "easy wins", we also amend the colour scheme used in flame charts, marking the `DELAY` function in blue and the `_vprintf` function in green.

VII. Results

We applied our work to visualize the FreeBSD boot process on two systems: The author's laptop (FreeBSD 12.0-CURRENT, 4-thread Intel 2700 MHz CPU, 32 GB RAM, ZFS running on a pair of encrypted disks) and a virtual machine in the Amazon Elastic Compute Cloud service (FreeBSD 11.1-RELEASE, 16-thread Intel 3000 MHz CPU, 32 GB RAM, UFS running on a single virtual disk); the flame charts produced are shown above[15] as Figures 1 and 2.

In Table 3 we compare the time spent on these two systems in different parts of the boot process; all routines which took over 100 ms on either system are shown.

VIII. Security

While we expect this work to be very useful for profiling and ultimately improving the performance of the FreeBSD

[13]For example, `vfs_mountroot_wait` runs in a loop checking the idleness of the GEOM event queue and checking that there are no holds registered via `root_mount_hold`.

[14]This name originates from a tool used in the Chrome web browser for performance visualization; as with our application, it displays stacks vertically and maps time to the x axis.

[15]With the "live" SVG images generated by our code it is possible to zoom into individual parts of the chart to see smaller portions with the appropriate labels, but that functionality is lost when incorporating the images into PDF or printed form.

[16]The `hpt27xx` and `hptnr` drivers in FreeBSD 11.1-RELEASE have a bug which resulted in excessively long probe times; this bug was fixed in 12.0-CURRENT before we profiled the boot time on our laptop.

Initialization routine	Laptop	EC2 instance
`hammer_time DELAY`	647	642
`SYSINIT vm_mem`	104	469
`SYSINIT cpu DELAY`	1000	1000
`SYSINIT cpu_mp DELAY`	60	156
`SYSINIT start_aps`	3	794
`DEVICE_ATTACH acpi`	197	52
`DEVICE_PROBE acpi_timer`	19	123
`DEVICE_ATTACH pci`	214	55
`DEVICE_PROBE hpt27xx`[16]		216
`DEVICE_PROBE hptnr`		108
`DEVICE_ATTACH nvme DELAY`		250
`DEVICE_ATTACH ena DELAY`		700
`DEVICE_ATTACH atkbd DELAY`	433	430
`DEVICE_PROBE psm`	174	323
`DEVICE_PROBE psm DELAY`	1813	1182
`SYSINIT clocks DELAY`	1000	1000
`THREAD thread taskq`	288	
`THREAD g_event`	2582	
`THREAD usbus0`	9008	
`VFS_MOUNT zfs`	190	
`VFS_MOUNT ufs`		1
`_vprintf`	720	4021
Other code	371	218
Total	18823	11740

Fig. 3. Time (ms) spent in during the boot process. Lines marked `foo DELAY` correspond to the time spent calling `DELAY` from within `foo`; with that exception, times indicated do not include time spent in child functions.

system boot, care should be taken when using it. There are two particular limitations of which users should be aware.

First, because the code which records timestamped events takes pointers to strings (most commonly the names of the functions which are being entered or exited) and does not copy their contents until the records are dumped out to userland via the `debug.tslog` sysctl, the code is almost certain to fail spectacularly in the event that a kernel module is ever unloaded — since the pointers to strings will turn into pointers to garbage (or possibly pointers to non-existent pages).

Second, since the FreeBSD kernel random number generator relies upon the time spent in device attach methods as an entropy source, recording timestamps upon entering and exiting those methods and exporting those values to userland risks compromising the cryptographic security of the kernel's entropy pool.

As a result of these problems, we recommend that this functionality remain a non-default option, enabled only when developers manually add `TSLOG` to their kernel configuration.

IX. Availability

The FreeBSD kernel patches required for this work are now in 12.0-CURRENT, having been added as SVN revisions 327423 through 327432. The userland code for visualizing the boot process is available from https://github.com/cperciva/freebsd-boot-profiling.

X. Future work

Now that it is easy to visualize the time spent in the FreeBSD kernel boot, we intend to return to our original motivation, of speeding up the boot process. There are some obvious targets: The `cpu` and `clocks` `SYSINIT`s each spend 1 second spinning in order to calibrate clock frequencies; the initialization of the `atkbd` and `psm` keyboard and mouse drivers includes a total of 2–3 seconds, mostly spent on delays which were once needed for ancient PS/2 hardware but are unlikely to be necessary any longer; avoiding the need to wait for USB devices to be probed before mounting ZFS would save as much as 9 seconds on affected systems; and the console output code used by `_vprintf` can be responsible for an absurdly large amount of time — over 30% of the total time spent booting on the EC2 instance we tested. We expect that the ease of use of this framework — functions can be instrumented by adding the `TSENTER()` and `TSEXIT()` macros, and timestamps can be logged within functions via a `TSLINE()` macro which records the source file and line number — will prove useful to developers as they delve deeper into the kernel code to further track down and fix boot time performance problems.

We also hope that this work will prove useful to FreeBSD developers working with non-x86 hardware; while they will need to instrument their respective machine-dependent initialization routines (since we only instrumented `hammer_time`), the rest of the kernel should already be usefully instrumented as a result of our work.

It would also be very interesting to see if this work can be ported or otherwise reproduced on other BSD operating systems. While NetBSD and OpenBSD do not have the same `SYSINIT` and device probing/attaching mechanisms as FreeBSD, we hope that the general concepts exhibited here can be applied to instrumenting those kernels, and expect that the visualization code will be applicable.

References

[1] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD.* Prentice Hall Professional, 2011.

[2] B. Gregg, "The flame graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2909476

[3] A. van de Ven, "scripts/bootgraph.pl," in *Linux kernel version 2.6.28.* Linus Torvalds, 2008.

[4] ——, "systemd-bootchart," https://github.com/systemd/systemd-bootchart.

[5] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system.* Pearson Education, 2014.