Profiling the FreeBSD kernel boot

From `hammer_time` to `start_init`

Colin Percival
Tarsnap Backup Inc.
`cperciva@tarsnap.com`

March 10, 2018

Why profile the FreeBSD kernel boot?

Why *did I* profile the FreeBSD kernel boot?

- In June 2017 I bought a new laptop.
- Unlike many FreeBSD developers, I insist on running FreeBSD on my laptops.
- Video driver support in laptops has traditionally been problematic.
    1. Load the `i915kms.ko` kernel module.
    2. Read the panic message.
    3. Reboot.
    4. Try changing some code.
    5. Recompile the kernel module.
    6. GOTO 1
- Hundreds and hundreds of attempts.

Why *did I* profile the FreeBSD kernel boot?

- Around reboot number 100 I started to notice things.
- Text scrolls by as the kernel initializes itself and probes devices, but sometimes the scrolling stops for a while.
- I started wondering what the kernel was doing during these "pauses".
- Make educated guesses and sprinkle `printf("%llu\n", rdtsc());`
  - Initializing the `vm_page` array. (20 ms / GB RAM)
  - Calibrating the CPU clock frequency. (1.0 s)
  - Calibrating the local APIC timer. (1.0 s)
  - Probing and initializing `psm0`. (2.0 s)
- I realized that having a systematic way of measuring everything would be much better than annotating functions only when I became suspicious.

FreeBSD boot process

- BIOS / EFI
- FreeBSD boot loader(s)
- FreeBSD kernel initialization
  - Machine-dependent initialization (e.g., `hammer_time`)
  - `mi_startup`
  - `start_init` (including `vfs_mountroot`).
- FreeBSD userland initialization
  - `rc.d` scripts

FreeBSD boot process

- BIOS / EFI
- FreeBSD boot loader(s)
- FreeBSD kernel initialization ← I'm looking at this.
  - Machine-dependent initialization (e.g., `hammer_time`)
  - `mi_startup`
  - `start_init` (including `vfs_mountroot`).
- FreeBSD userland initialization
  - `rc.d` scripts

Linux boot profiling

- Linux prints a timestamp at the start of each line of kernel output.

```
[    2.082829] ACPI: Power Button [PWRF]
[    2.085704] input: Sleep Button as /devices/LNX...
[    2.092002] ACPI: Sleep Button [SLPF]
[    2.166920] input: ImExPS/2 Generic Explorer Mo...
[    2.302339] mousedev: PS/2 mouse device common ...
```

- This can make it very easy for users to notice if part of the kernel boot is taking a long time.
- Timestamping kernel log messages means that you only get timestamps when the kernel is printing log messages — not always the most useful moments.
- At the beginning of the Linux boot, all the timestamps logged are `0.000000` because the clocks aren't initialized yet — better to record raw CPU cycle count numbers and then translate them later.

DTrace

- DTrace is *the* way to profile anything and everything in FreeBSD!
- However, DTrace needs:
    - Traps
    - Memory allocation
    - Thread scheduling
    - probably lots more...
- A large part of what we want to profile happens before any of these basic kernel subroutines are available.
- We need to use something which is simpler and with fewer dependencies.

KTR

- KTR is a mechanism for logging "kernel events".
- You call a function; it logs whatever you give it into a buffer.
- Almost exactly what I needed, but...
    - It uses a circular buffer — good for answering "what happened just before we crashed" but bad for answering "what happened at the start of the boot process".
    - Its default buffer size is only 1024 records — we will need far more than this.
    - It can't *quite* run at the start of the boot process.
- All of these limitations could be worked around with a few lines of changes, but it was simpler to add a new subroutine for logging timestamped events which was designed for boot profiling.

TSLOG

- `sys/tslog.h` and `kern/kern_tslog.c` implement the TSLOG framework.
- Buffer fixed at compile time (default 256k records).
- To log an record, we atomically reserve a slot, then populate it with the appropriate data.
- When the buffer is full, future records are silently discarded.
- Each record consists of a cycle count, a thread ID, a record type, and one or two strings.
- Records are logged via `TS*` macros, which compile to nothing for kernels compiled without the `TSLOG` option.
- The buffer is dumped to userland via the `debug.tslog` sysctl.

Function tracing

- We can figure out most of what we want to know by knowing when we entered and exited functions.
- `TSENTER()` records that we have entered a function.
- `TSEXIT()` records that we are about to exit a function.
- Scatter these through the tree in potentially useful places!
- Top level of the boot process: `hammer_time`, `mi_startup`, `start_init`.
- Functions which get called a lot: `DELAY()`, `_vprintf`.
- `SYSINIT` routines.
- `DEVICE_PROBE` and `DEVICE_ATTACH` functions.
- `VFS_MOUNT` calls.

Annotating a function

```
void
DELAY(int n)
{

        TSENTER();
        if (delay_tc(n)) {
                TSEXIT();
                return;
        }

        init_ops.early_delay(n);
        TSEXIT();
}
```

`SYSINIT`

- `SYSINIT`s are a mechanism used by FreeBSD to specify that code should be run during the kernel startup process.

  `SYSINIT(name, order1, order2, function, cookie);`
- Similar to Linux `initcalls`.
- A record is created in a special ELF section, and linker magic makes it possible to get a list of all the `SYSINIT`s declared all over the kernel.
- `mi_startup` sorts the `SYSINIT` functions and calls them in the appropriate order.
- With the `TSLOG` kernel option, we redefine the `SYSINIT` macro to call a shim function which logs the entry/exit.

SYSINIT

```
#ifdef TSLOG
struct sysinit_tslog {
        sysinit_cfunc_t func;
        const void * data;
        const char * name;
};
static inline void
sysinit_tslog_shim(const void * data)
{
        const struct sysinit_tslog * x = data;

        TSRAW(curthread, TS_ENTER, "SYSINIT", x->name);
        (x->func)(x->data);
        TSRAW(curthread, TS_EXIT, "SYSINIT", x->name);
}
...
```

DEVICE_PROBE and DEVICE_ATTACH

- The `configure2` SYSINIT function recurses through the attached buses looking for devices.
- As the names suggest, `DEVICE_PROBE` is used to probe devices, and `DEVICE_ATTACH` is used to attach devices once they are found.
- Drivers declare their probe and attach methods via the `DEVMETHOD` macro.
  - Yes, the FreeBSD kernel is object-oriented! See `kobj(9)`.
- `DEVICE_*` are inline functions defined in `device_if.h`, which is generated at build-time from `device_if.m`.
  - Generic object method dispatch code: Look up the function pointer, then call it.
- I taught `makeobjops.awk` to add prologues and epilogues to the generated code, then annotated `device_if.m`.

VFS_MOUNT

```
#define VFS_MOUNT(MP) ({                                        \
        int _rc;                                                \
                                                                \
        TSRAW(curthread, TS_ENTER, "VFS_MOUNT",                 \
            (MP)->mnt_vfc->vfc_name);                           \
        VFS_PROLOGUE(MP);                                       \
        _rc = (*(MP)->mnt_op->vfs_mount)(MP);                   \
        VFS_EPILOGUE(MP);                                       \
        TSRAW(curthread, TS_EXIT, "VFS_MOUNT",                  \
            (MP)->mnt_vfc->vfc_name);                           \
        _rc; })
```

Boot holds

- Tracing function entry/exit points tells us what each kernel *thread* is doing at any given time.
- Once the kernel is running multiple threads, we need a bit more than this — sometimes one thread will wait for another.
- The `intr_config_hooks` `SYSINIT` waits for hooks which were established via `config_intrhook_establish`.
- The `g_waitidle` function waits for the GEOM event queue to be empty.
- The `vfs_mountroot_wait` function waits for holds registered via `root_mount_hold`.
- Extracting information from the kernel scheduler might help here, but that gets complicated fast.

Boot holds

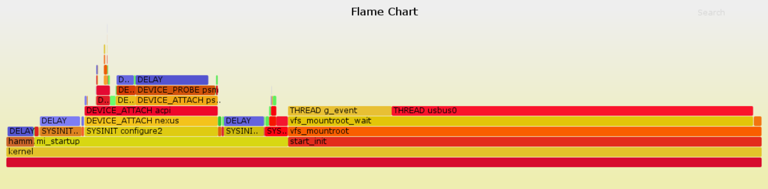
- Much simpler: Annotate the places where the "main thread" is blocked waiting for other threads to finish something.
- Record the start and end of "waits", and when "holds" are acquired and released.
- Record the identity of newly created kernel threads.
- Heuristic: Blame "blocked" time on whatever thread was the last one to release a hold, for as long as that thread held it.
- Heuristic: Assume the thread was blocking the boot process starting at the latest of when it picked up a hold and when the thread was created.

Visualization

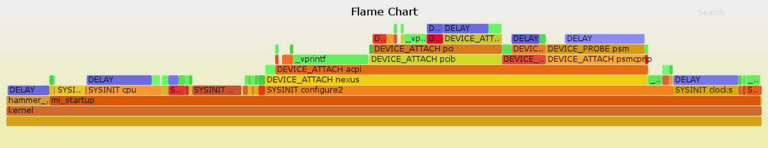
- After booting, dump all of the logged records.
- Organize them into threads and use entry/exit records to construct timestamped stacks.
- The "kernel boot process" is `thread0` (aka. `swapper`) plus `init` prior to when it enters userland.
- Where a boot hold occurs, identify the thread which we're waiting for and splice its stacks on top.
- Now we have a series of stacks covering the kernel boot process.
- Obvious visualization tool: Flame Graphs.
- Unfortunately Flame Graphs sort stacks in alphabetical order...
- Flame *Charts* are like Flame Graphs but keep the stacks in chronological order.

Flame Charts

My laptop:



Amazon EC2 c5.4xlarge:



Where's the time going?

- `hammer_time DELAY`: 640 ms.
- `SYSINIT vm_mem`: $\approx$ 20 ms / GB RAM.
- `SYSINIT cpu DELAY`: 1000 ms.
- `SYSINIT start_aps`: 3 ms on my laptop, 800 ms in EC2.
- `DEVICE_PROBE hpt*`: 320 ms.
- `DEVICE_PROBE psm`: 2000 ms on my laptop, 1500 ms in EC2.
- `SYSINIT clocks DELAY`: 1000 ms.
- `THREAD g_event`: 2600 ms on my laptop — GELI key derivation.
- `THREAD usbus0`: 9000 ms on my laptop — root mount waiting for usbus0.
- `_vprintf`: 720 ms on my laptop, 4000 ms in EC2.

Where's the time going?

- `hammer_time DELAY`: 640 ms.
- `SYSINIT vm_mem`: ≈ 20 ms / GB RAM.
- `SYSINIT cpu DELAY`: 1000 ms.
- `SYSINIT start_aps`: 3 ms on my laptop, 800 ms in EC2.
- `DEVICE_PROBE hpt*`: 320 ms.
- `DEVICE_PROBE psm`: 2000 ms on my laptop, 1500 ms in EC2.
- `SYSINIT clocks DELAY`: 1000 ms.
- `THREAD g_event`: 2600 ms on my laptop — GELI key derivation.
- `THREAD usbus0`: 9000 ms on my laptop — root mount waiting for usbus0.
- `_vprintf`: 720 ms on my laptop, 4000 ms in EC2.

Why profile the FreeBSD kernel boot?

Taking a systematic approach to profiling the kernel boot will tell you far more than simply relying on your ability to notice when it seems slow.

Availability

- `TSLOG` code is in FreeBSD HEAD.
- Visualization code is at `https://github.com/cperciva/freebsd-boot-profiling`.

Availability

- `TSLOG` code is in FreeBSD HEAD.
- Visualization code is at `https://github.com/cperciva/freebsd-boot-profiling`.

Questions?