Virtualization of BSD

Using the QNX Hypervisor

Quentin Garnier

Senior Kernel Developer
qgarnier@blackberry.com
May 2019

BlackBerry | QNX

Agenda

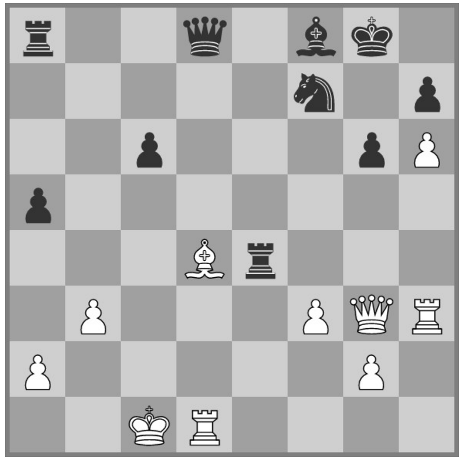
- The virtualization environment: QNX Neutrino and QNX Hypervisor
- Goals for the exercise
- Stories from the trenches

QNX Host Environment

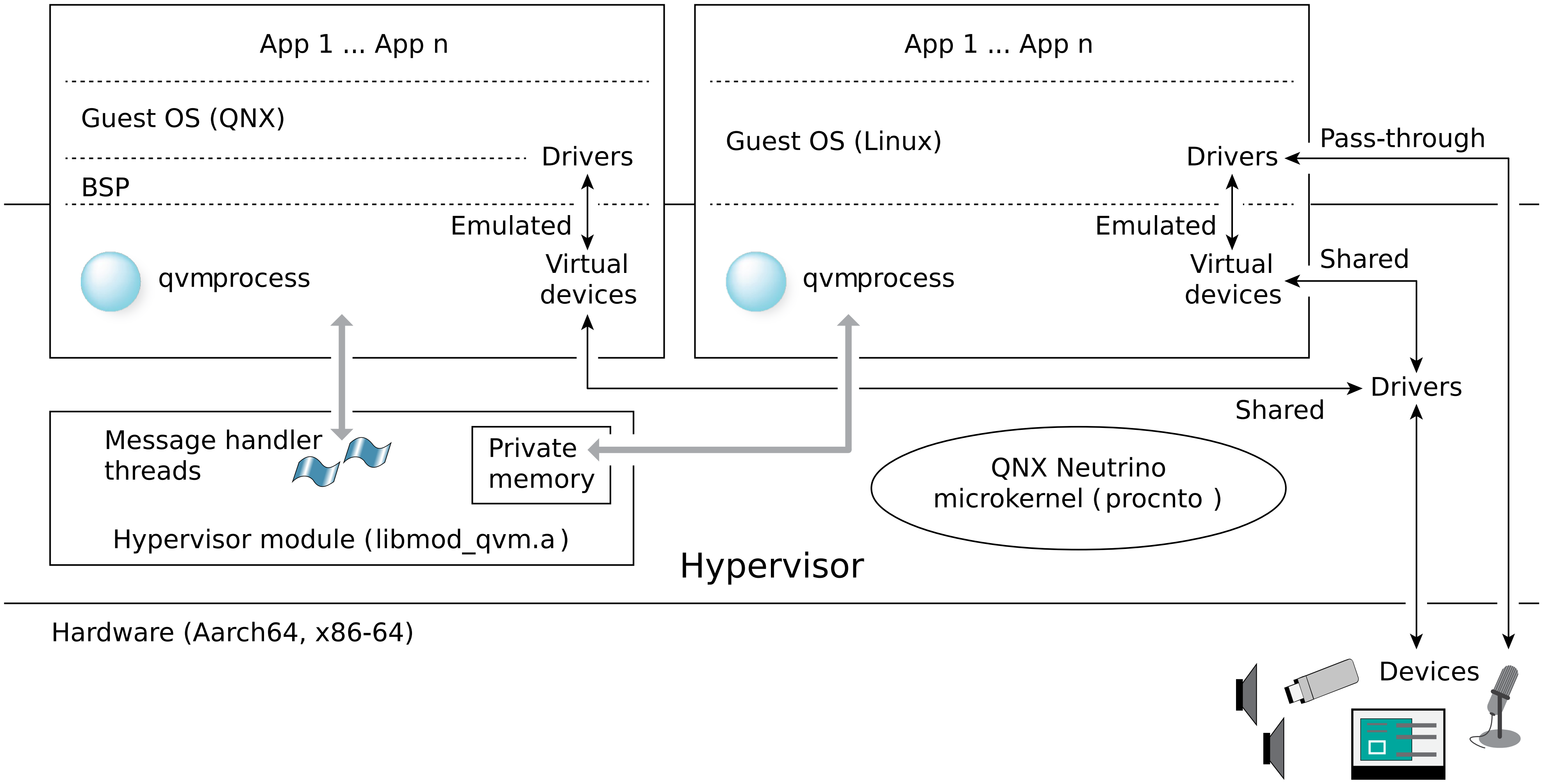
About the QNX Hypervisor design (1/2)

- Some vocabulary:
  - Host system
  - Virtualization manager (qvm)
  - Virtual machine
  - Guest system

- What the host system provides:
  - Virtualization manager
  - Drivers for possible shared hardware resources
  - Anything else the system designer wants to have



About the QNX Hypervisor design (2/2)

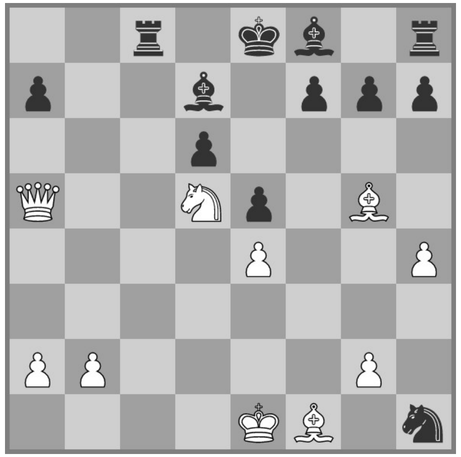


A few design choices for the Hypervisor (1/2)

- Targeting QNX guests and Linux guests
  - This is what the industry wants

- The Hypervisor runs as a process in the host system, with virtual CPUs being scheduled as normal threads
  - A special privilege elevation interface allows running guest code

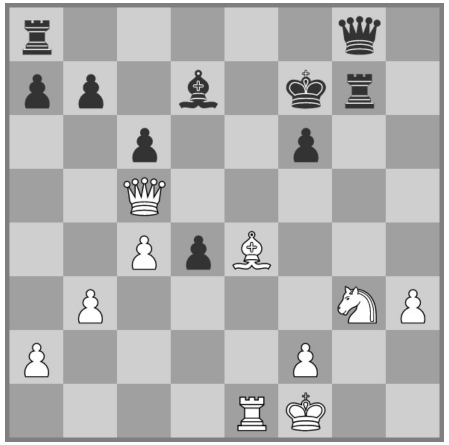
- Minimal environment, therefore no or minimal virtual firmware
  - For instance, no emulated BIOS whatsoever on x86_64



A few design choices for the Hypervisor (2/2)

- Minimal environment, therefore no or minimal virtual firmware
  - QNX on x86 is booted through Multiboot
  - Linux/x86_64 has its own protected-mode loading protocol

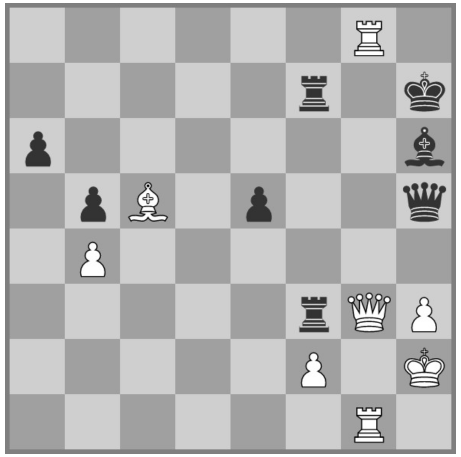
- How does time flow in a guest?
  - It’s complicated...



Virtual Machines

- As little emulated hardware as possible
  - It’s not just about being lazy: emulation is slow

- Customers either pass-through hardware or use VirtIO devices



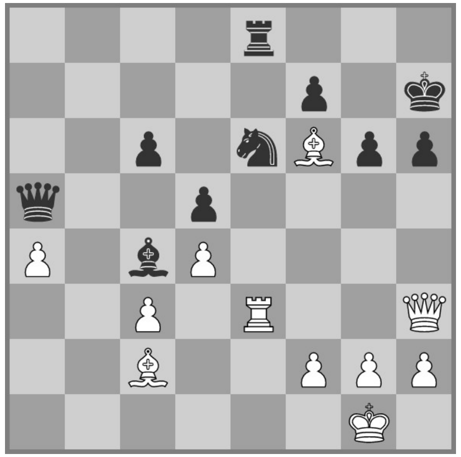
Goals

Why am I doing this?

- Emulation gaps
  - Comes from focusing on a very limited number of guests

- Finding actual bugs
  - Same cause, but it means better coverage of the existing code

- QNX is cool, hypervisors are cool, BSDs are cool.

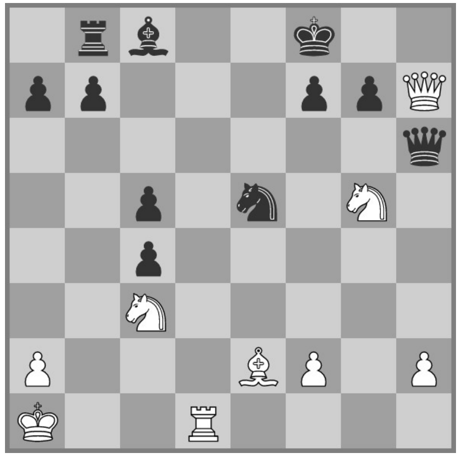


Objectives with BSD guests

- Get a multi-user prompt
  - I’ll settle for some dmesg excitement

- Find bugs in our Hypervisor, possibly in the guests
  - It’s all a matter of point of view!

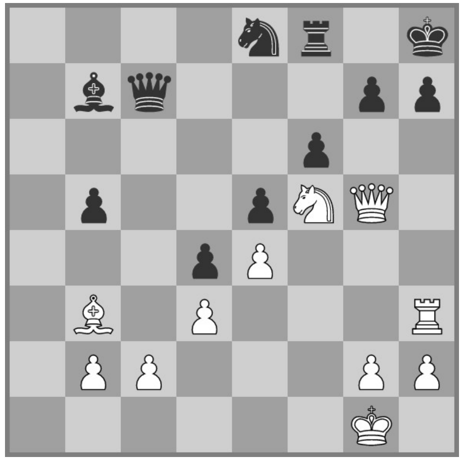
- Look at performance if time permits



Guest Experience

Booting NetBSD/amd64

- The easiest to start with:
  - Very familiar with the x86 Hypervisor code
  - Quite familiar with NetBSD internals
  - build.sh – other BSDs, take note!

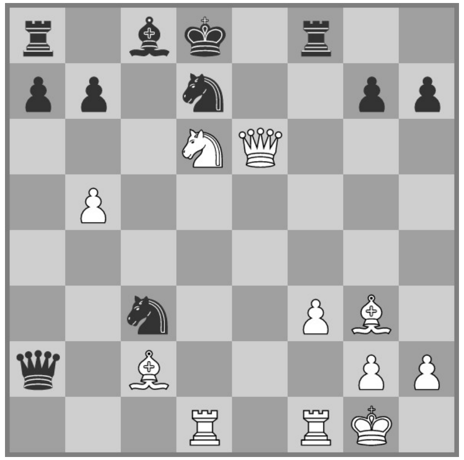


The bootloader problem

- Not just about NetBSD/amd64

- While NetBSD/i386 can be booted through Multiboot, NetBSD/amd64 has it own protocol

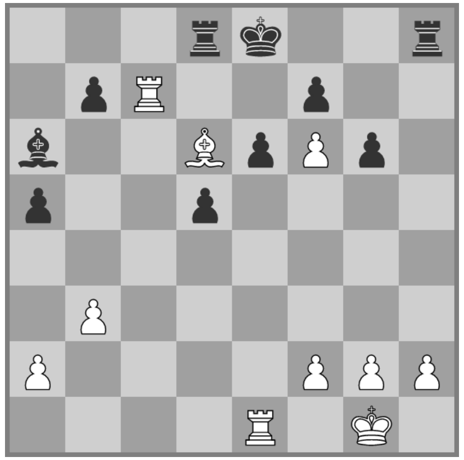
- Frustrating because:
  - It starts in protected mode
  - It wants the same data it would get through Multiboot, just in a slightly different format



Work needed to get NetBSD/amd64 booting (1/3)

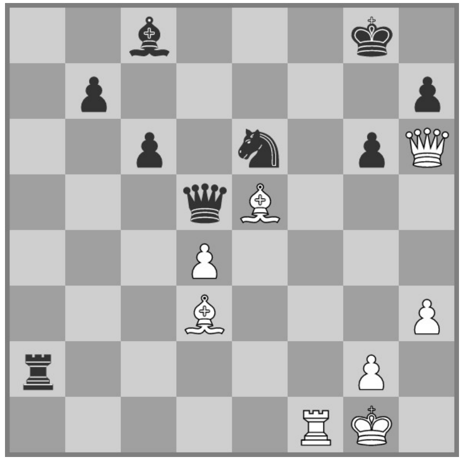
- A module to load NetBSD kernels

- Emulation of MSR 0x10 (IA32_TIME_STAMP_COUNTER)
  - In our Hypervisor, MSRs that are not passed-through or emulated result in an exception
  - This is an emulation gap



Work needed to get NetBSD/amd64 booting (2/3)

- NetBSD doesn’t handle not having MTRRs
  - MTRRs are not emulated because it’s a lot of work and not really used these days
  - However, the Linux kernel ties the support of MTRRs to its ability to use the PAT

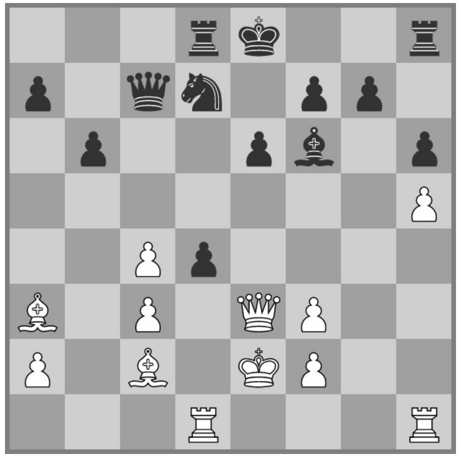


- Handling REP OUTS correctly
  - Long debugging session

Work needed to get NetBSD/amd64 booting (3/3)

- A small workaround for the virtio-block driver
  - size_max is a 32-bit unsigned

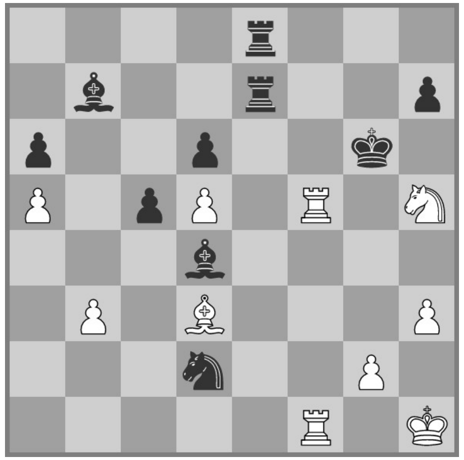
- The DSDT always exposes a PCI bridge, but it’s only created if there is a PCI device in the virtual machine
  - NetBSD would crash trying to access the non-existent PCI host controller



Booting FreeBSD/x86_64

- It wants to start in long mode, and has its own set of data structures to pass memory information and command line

- Sorry, FreeBSD folks, I skip
  - Already done the data structure spelunking for NetBSD, and I’d have to write even more code to start a guest in long mode
  - Neither are terribly difficult, and the long mode environment could be a time saver booting Linux in the future



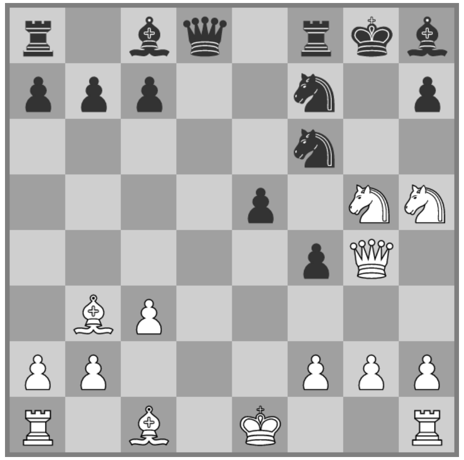
- Also, having to install FreeBSD somewhere just so I can recompile a kernel is frustrating

Booting OpenBSD/amd64

- OpenBSD was interesting in addition to NetBSD because it has its own implementation of the ACPI OSPM

- The boot protocol to get the OpenBSD kernel running is very similar to NetBSD, but everything is just slightly different.

- The only problem was comprobe1() which compares the value read from IIR to 0x38 and thinks that it means the receive buffer is not empty.
  - I haven't seen any 8250 documentation that would indicate that
  - 0x20 actually means a 64-byte FIFO on the 16750
  - 0x38 was there in revision 1.1 in NetBSD, I didn't go any further in history

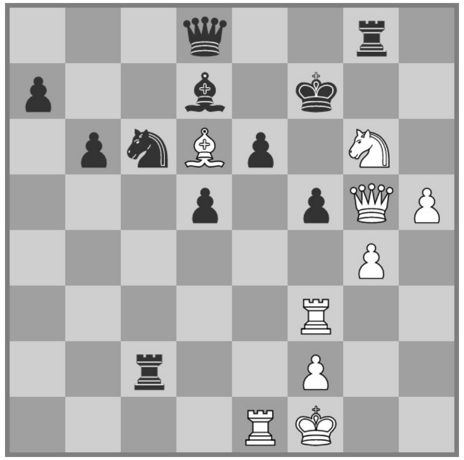


Booting FreeBSD/aarch64

- It seems the FreeBSD/aarch64 kernel wants an EFI loader

- I couldn’t figure out an easy way to get an EFI environment
  - The Linaro EFI build for the Foundation Model expects to have code running at EL3, which is not something we ever consider emulating
  - U-Boot might be an option to investigate more in the future

- Another skip, but I’m not biased against FreeBSD, okay?

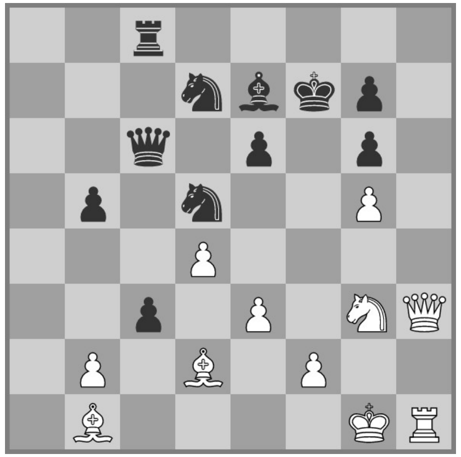


Booting NetBSD/aarch64

- NetBSD/aarch64’s locore.S is easier to read, it can work with only the pointer to the FDT in x0

- The QNX Hypervisor emulates (or pretends to) the Foundation Model, but there’s no code handling it among all the ARM_PLATFORM definitions

- NetBSD’s pl011 driver disables the transmit FIFO, which exposed a bug in our emulation

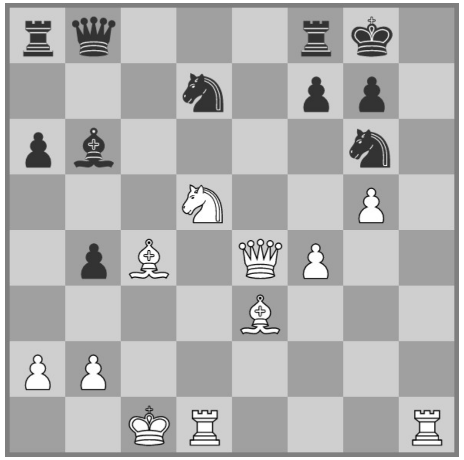


What’s next?

Performance considerations

- NetBSD/amd64 seems sluggish during boot
  - i8254_delay() is expensive, especially because of the way time flows in our Hypervisor

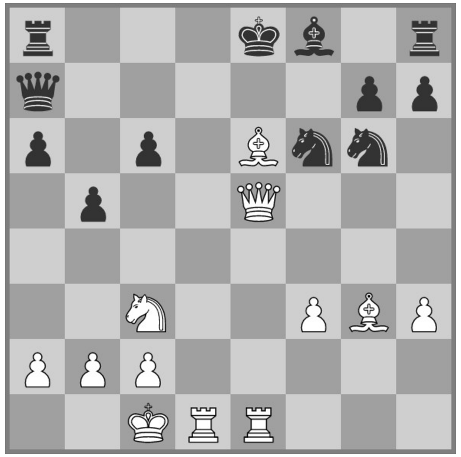
- There’s a long pause in both OpenBSD and NetBSD on amd64
  - It seems to be because of the minimal emulation of the 8042



Maybe for next time

- Get a FreeBSD to boot

- Create guest images for our test team
  - The number of problems found proved it was a worthy experiment



Questions

qgarnier@blackberry.com