Speeding up the Booting Time of a Toro Appliance

Matias E. Vara Larsen

www.torokernel.io
matiasevara@gmail.com
Toro is an embedded kernel including five units:
- Process
- Memory
- Filesystem
- Networking
- Devices, e.g., Block Device, Network Device

Each unit provides minimalist APIs accessible from the embedded application.
Toro is an embedded kernel including five units:
- Process
- Memory
- Filesystem
- Networking
- Devices, e.g., Block Device, Network Device
Each unit provides minimalist APIs accessible from the embedded application.
Application-oriented Kernel

- User application and kernel units are compiled in a single binary
- The application includes only the component required
Application-oriented Kernel

- User application and kernel units are compiled in a single binary.
- The application includes only the component required.

```plaintext
program HelloWorld;
uses
    Memory,
    Filesystem,
    Ext2,
    E1000;
begin
    //
    // Your Code
    //
    end.
```
Application-oriented Kernel

Toro Kernel
- Process
- Memory
- Devices
- Filesystem
- Networking

Uses

Microservice

CloudIt.sh
- Launches
- Uses

VM

Builder

Image
Application-oriented Kernel

“It’s all talk until the code runs.” - Ward Cunningham
Application-oriented Kernel

Toro Kernel
- Process
- Memory
- Devices
- Filesystem
- Networking

Uses

Microservice

Builder

Image (~ 4MB)

Booting time is 1,5s

VM

Launches

CloudIt.sh

Uses

Toro.elf

Time to build a new image is about 1s
Application-oriented Kernel

Toro Kernel
- Process
- Memory
- Devices
- Filesystem
- Networking

Uses

Microservice

Toro.elf

CloudIt.sh

VM

Launches

Image (~ 4MB)

Uses

Builder

Booting time is 1.5s

Time to build a new image is about 1s

These timings can be improved to enhance continuous deployment of microservices
Booting in Toro

- VMM Initialization
- Bootloader
- Kernel Initialization
Booting in Toro

- Initialization of the device model
- BIOS
- Other stuff

VMM Initialization

Bootloader

Kernel Initialization
Boot in Toro

1. VMM Initialization
   - Initialization of the device model
   - BIOS
   - Other stuff

2. Bootloader
   - Initialize hardware
   - Initialize processors, e.g., setup and enable paging, enable long mode, etc
   - Load the kernel into memory. In this case the image’s size is very important

3. Kernel Initialization
Booting in Toro

This presentation deals with different approaches to improve these times.
Outline

- Speeding Up the Bootloader
- Speeding Up the Virtual Machine Monitor (VMM)
- Evaluation
- Conclusion
- QA
Speeding Up the Bootloader

• Context:
  – The generated image is a copy of the kernel in memory
  – The bootloader just read from the disk the image and then it writes it to memory

• Problem:
  – The resulting image is huge
  – The bootloader is still complex

• Proposal:
  – Load Toro by using the “-kernel” option in QEMU/KVM (see Issue #223 at Github)
Kernel Binary (elf32)

Multiboot Header

MultiBootloader

.text

.data

$ kvm -kernel Toro.elf

QEMU/KVM

Memory
Kernel Binary (elf32)

- Multiboot Header
- MultiBootloader
- .text
- .data

QEMU/KVM

$ kvm -kernel Toro.elf

Memory
Example of QEMU/KVM using a Multiboot Header:

```
$ kvm -kernel Toro.elf
```

Diagram:
- Kernel Binary (elf32)
- Multiboot Header
- MultiBootloader
- .text
- .data
- Loads
- Memory
  - MultiBootloader
  - .text
  - .data

Process:
1. QEMU/KVM loads the Multiboot Header and MultiBootloader.
2. The MultiBootloader loads the .text and .data sections.
Kernel Binary (elf32)

- Multiboot Header
- MultiBootloader
- .text
- .data

MultiBootloader (MutibootMain())

.text (KernelMain())

.data

QEMU/KVM

$ kvm -kernel Toro.elf

Processor is already in protected mode
Speeding Up the Bootloader

• Benefits:
  - Reduce image size since it is only an elf32 binary from 4MB to 130kb
  - Reduce bootloader complexity since QEMU loads the kernel into memory and yield the CPU to protected mode
  - Reduce booting time from 1.5s to 0.5s
Speeding Up the Bootloader

- **Benefits:**
  - Reduce image size since it is only an elf32 binary from 4MB to 130kb
  - Reduce bootloader complexity since QEMU loads the kernel into memory and yield the CPU to protected mode
  - Reduce booting time from 1.5s to 0.5s

- **Drawbacks:**
  - VMM has to support the loading of a multiboot kernel
  - Supports only elf32, so some magic is needed to make it work with elf64
  - We still have to jump to long mode
Speeding Up the Bootloader

• Benefits:
  - Reduce image size since it is only an elf32 binary from 4MB to 130kb
  - Reduce bootloader complexity since QEMU loads the kernel into memory and yield the CPU to protected mode
  - Reduce booting time from 1.5s to 0.5s

• Drawbacks:
  - VMM has to support the loading of a multiboot kernel
  - Supports only elf32, so some magic is needed to make it work with elf64
  - We still have to jump to long mode

Qemu-lite works around these but project seems discontinued (Port of Toro at Issue #192)
Outline

• Speeding Up the Bootloader
• **Speeding Up the VMM**
• Evaluation
• Conclusion
• QA
Speeding Up the VMM

- We study three approaches to improve the time spent in VMM initialization
- We focus on KVM/QEMU-based VMM
- These approaches are: QBoot, NEMU and Firecraker
- These approaches simplifies some aspect of the VMM, e.g., loading the of the kernel, hardware initialization or device model
Bare-metal host

Linux Kernel

VMM

Guest (Mode)

Device Emulation

BIOS

KVM Driver

In-kernel device emulation

Linux Kernel

Bare-metal host
QBoot:
- Minimal x86 firmware for QEMU to boot Linux
- [https://github.com/bonzini/qboot](https://github.com/bonzini/qboot)
- “A couple hardware initialization routines written mostly from scratch but with good help from SeaBIOS source code”
- Limit of 8 MB for vmlinuz+initrd+cmdline

```
$ kvm -bios bios.bin -kernel Toro.elf
```
NEMU[1]:
- Based on QEMU only for x86-64 and aarch64
- Reduced device model by focusing on non-emulated devices to reduce the VMM’s footprint and the attack surface
- Proposes a new machine type named ‘virt’ which is thinner and only boots from UEFI

Firecracker:
- Simple VMM implemented in Rust developed by Amazon Web Services to accelerate the speed and efficiency of services like AWS Lambda and AWS Fargate
- Sets vCPU to long mode, sets pages tables the Linux way and expects kernel to be in vmlinux format (64-bit ELF uncompressed)
Evaluation

• We measured the time that takes the kernel to start to execute, i.e., the time since the VM is launched until the KernelMain() is executed

• We compared these times by using the presented solutions

• See Issue #276 at Github for more information
Results

4 cores Intel(R) Atom(TM) CPU C2550 @ 2.40GHz
8 GB of physical memory

<table>
<thead>
<tr>
<th>Approach</th>
<th>Image</th>
<th>Binary</th>
<th>Binary with QBoot</th>
</tr>
</thead>
<tbody>
<tr>
<td>QEMU/KVM (2.5.0)</td>
<td>1457 ms</td>
<td>452 ms</td>
<td>132 ms</td>
</tr>
<tr>
<td>NEMU (#39af42)</td>
<td></td>
<td>309 ms</td>
<td>95 ms</td>
</tr>
<tr>
<td>Firecracker (0.14.0)</td>
<td></td>
<td>17ms</td>
<td></td>
</tr>
</tbody>
</table>

$ echo “Hello World!”
avg: 2.629263ms

https://blog.iron.io/the-overhead-of-docker-run/
Conclusion

- Booting time improved by a factor $x_{11}$ when using multiboot and QBoot
- Booting time improved by a factor $x_{85}$ when using Firecracker
- Trade-off between the needed work to adapt the kernel and minimizing booting time
QA

- http://www.torokernel.io
- torokernel@gmail.com
- Twitter @torokernel
- Torokernel wiki at github
  - My first Three examples with Toro
- Test Toro in 5 minutes (or less...)
  - torokernel-docker-qemu-webservices at Github
QA

- http://www.torokernel.io
- torokernel@gmail.com
- @torokernel
- Torokernel wiki at github
- My first Three examples with Toro
- Test Toro in 5 minutes (or less...)
- torokernel-docker-qemu-webservices at Github

That’s all folks!