Building a Linux-compatible Unikernel

How your Application runs on Unikraft

Simon Kuenzer
Lead Maintainer
CTO and Co-Founder
Unikraft GmbH
simon@unikraft.io
1

Unikraft: The Unikernel SDK
Unikraft Unikernels

- One application → Flat and single address space
- Single monolithic binary with only necessary kernel components
- Advantages from specialization
  - Performance and efficiency
  - Small TCB and memory footprint
Design Principles

■ Specialization as main driving design principle
  - Highly customizable & configurable: KPI-driven specialization

■ Philosophy: “Everything is a (micro-)library”
  - Decomposed OS primitives
    ■ Schedulers, memory allocators, VFS, network stacks, …
  - Architectures, platform support, and drivers
    ■ Virtualization environments, bare-metal
  - Application interfaces
    ■ POSIX, Linux system call ABI, language runtimes

(1) Configuration (KConfig) and Build System

(2) Library Pool
The (Micro)-Library Stack

application

musl
newlib

syscall-shim

posix-fdtab
posix-process

posix-socket
vfscore

uosched
ukboot
ukalloc

lwip
mtcp
9pfs
ramfs
ext4

uknetdev
ukblockdev

virtio-net
clock
virtio-block
memregion

KVM

XEN

netfront
memregion

virtio-net
virtio-block
clock
memregion

KVM

virtio-net
virtio-block
clock
memregion

KVM

virtio-net
virtio-block
clock
memregion

XEN

netfront
memregion

virtio-net
virtio-block
clock
memregion

KVM

virtio-net
virtio-block
clock
memregion

XEN

netfront
memregion

virtio-net
virtio-block
clock
memregion

KVM
2

Linux Application Compatibility
Linux Application Compatibility for Adoption

- Most cloud software is developed for Linux
- People are used to their software
- Remove obstacles for using Unikraft with existing application

VISION

Seamless application support

*Applications are automatically ported and benefit from lower boot times, less memory consumption, improved performance, etc.*
## Linux-compatibility Landscape

<table>
<thead>
<tr>
<th>Native</th>
<th>Binary compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Application* sources are compiled and linked together with Unikraft</td>
<td>• Application* binaries are externally built</td>
</tr>
</tbody>
</table>

### Unikraft-driven compilation
- • Port/convert application build procedure to Unikraft

### Instrumented
- • Instrument foreign build system (e.g., cross-compilation)

### Build-time linking
- • Build objects or static libraries externally and link with Unikraft

### Runtime linking/loading
- • Support for shared libraries and loading on ELF binaries
## Requirements

### API-compatibility

- POSIX, POSIX, POSIX
- API-compatible libraries and ported libraries (including libC)

### ABI-compatibility

- ELF format (shared libraries/binaries)
- Binary compatible function interfaces
  - Linux system calls
  - Library functions
- Binary compatible data representation

### Native

- Compatible system runtime environment
  - E.g., special filesystems and mount points: procfs, sysfs

### Binary compatible
# Pros & Cons

<table>
<thead>
<tr>
<th>Native (API-compatible)</th>
<th>Binary compatible (ABI-compatible)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Performance tuning and specialization of application-kernel interaction naturally possible</td>
<td>+ Source code not required</td>
</tr>
<tr>
<td>+ Source code not required</td>
<td>+ Applications are compiled the standard way, independent from Unikraft</td>
</tr>
<tr>
<td>- Compiling of the application is not independent to Unikraft (instrumentation, build system porting)</td>
<td>+ No modifications to application needed</td>
</tr>
<tr>
<td>- Risk of taking over implementation complexity of Linux to Unikraft (e.g., “netlink sockets” for getifaddrs())</td>
<td>- Less opportunities to specialize and tune kernel-application interaction</td>
</tr>
</tbody>
</table>
Binary compatibility vs. Native

- No extra optimization on native port
- Still Apple&Oranges comparison: musl vs glibc, different heap allocators

![Graph showing nginx, nginx-native, elfloader+nginx, linux+nginx performance comparison]

Nginx default index.htm, served from initrd(RAM), Unikraft: tlsf
Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, Guest-Host, 1vCPU
Optimization Potential of Native Ports

- Native port patched with improved HTTP processing

![Graph showing optimization potential for Go HTTP Application]

- Optimized native
- Binary compatibility

Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, Guest-Host, 1vCPU

FOSDEM'23 February 5, 2023  Building a Linux-compatible Unikernel  © 2023 Unikraft GmbH
Unikraft’s Implementation
Overview

Native (API-compatible)

- musl

Binary compatible (ABI-compatible)

- elfloader

Syscall mapping

- syscall_shim

Compat layer

- posix-process
- posix-user
- vfscore
- posix-mmap
Syscall Shim

- Libraries register a handler to the shim
- Shim provides two ways to handle/route system calls
  - **Compile-time**: Link application to handler functions (function calls)
  - **Runtime**: Binary system call handler (Linux-style)
- Our aim: Re-use code for both modes
Overview

Native (API-compatible)

Native (API-compatible)

musl

musl

Binary compatible (ABI-compatible)

Binary compatible (ABI-compatible)

elfloader

elfloader

syscall_shim

syscall_shim

syscall mapping

syscall mapping

posix-process

posix-process

posix-user

posix-user

vfscore

vfscore

posix-mmap

posix-mmap

Compat layer

Compat layer
Native: lib-musl

- Musl is compiled natively (build by Unikraft)
- Few patches to replace system call invocation
  - Syscall_shim resolves invocation to functions calls
  - Syscall_shim provides ENOSYS stub for unregistered system calls
- At run-time, syscall shim is out of the way
Overview

Native (API-compatible)

- musl

Binary compatible (ABI-compatible)

- elfloader

Syscall mapping

- syscall_shim

Compat layer

- posix-process
- posix-user
- vfscore
- posix-mmap
Bin. Compatic. (1/2): app-elfloader

- Loads an Linux ELF application
- Supports (today):
  - static-PIE
  - dynamically-linked using loader (needs posix-mmap)
- System calls are trapped and handled through syscall_shim
- Supported system calls selectable by choosing subsystem libraries
  - e.g., vfscore, posix-process, posix-user
Bin. Compats. (2/2): System Call handler

- syscall trap handler provided by syscall_shim
- No domain switch needed, single AS

- Because of Linux system call calling convention and assumptions:
  - Linux does not use extended registers \( \rightarrow \) we do
    - Save & restore FPU, VU, \ldots state
  - Linux does not use a TLS \( \rightarrow \) we do
    - Save & restore TLS register (application TCB)
Demo time
5

Learned Lessons
Native Application Support
Background

- We avoid linking in multiple libCs to monolithic Unikernel (specialization at compile time)
  - Use a single libC

- Provide multiple libCs: nolibc, musl, newlib

- Keep libC as vanilla as possible
  \( \rightarrow \) lower maintenance effort
Learned Lessons with libCs

- Every libC is different
  - Test code with all officially supported libCs

- Namespacing is important
  - Risk of clashing with libC(-internal) definitions
  - avoid plain declarations, like MIN(), MAX()
  - underscore prefixes may not be enough

- Careful with initialization and dependencies
  - Example: TLS and kernel prints
    - Kernel prints got their own print function

- Circular dependencies can occur
  - Example: getdents64()
  - Learned: Vanilla not always possible → patching
Example: Circular Dependency `getdents64()`

- Circular dependency (syscall_shim → musl → syscall_shim)
- musl defines `getdents64()` as macro to `getdents()`

  ```c
  #define getdents64 getdents
  ```

- vfscore implements both syscalls with:

  ```c
  #include <dirent.h> /* struct dirent, struct dirent64 */
  UK_SYSCALL_R_DEFINE(int, getdents, int, fd, struct dirent*, dirp, size_t, count)
  {/* ... */}
  UK_SYSCALL_R_DEFINE(int, getdents64, int, fd, struct dirent64*, dirp, size_t, count)
  {/* ... */}
  ```
Learned Lessons:
Binary Application Support
Background

■ Unikraft makes use of TLS
  - An artifact of supporting applications natively
  - Same register used as in Linux user space (x86: %fsbase segment register)
    → Keep bin. compat working for build-time linking

■ Unikraft makes use of extended registers (even drivers)
  - Normally no separation between kernel and application code
    → Monolithic: Everything is a function call
Learned Lessons with binary compatibility

- Linux system calling convention fits for Linux assumptions
- → Need to be able to handle two TLSes (Unikraft TLS and “userland” TLS)
  - Our solution: switch TLS on binary system calls
- → Need to handle extended register context
  - Our solution: Save & restore on binary system calls
Closing
Upcoming Features

- Improved Linux compatibility
  - posix-signals, posix-netlink, thread exit, join, wait support
- Seamless application support with kraftkit (using elfloader)

Watch out for:

- Seamless integration into kubernetes
- Running Unikraft on your infrastructure provider
- Automatically packaging of your applications

I’d ❤️ to hear your feedback: simon@unikraft.io
Join us!

- OSS project [unikraft.org](https://unikraft.org)
- Get started with kraftkit [github.com/unikraft/kraftkit](https://github.com/unikraft/kraftkit)
- Code & Contributing [github.com/unikraft](https://github.com/unikraft)

Follow us on
- Twitter: [@UnikraftSDK](https://twitter.com/UnikraftSDK)
- LinkedIn: [https://linkedin.com/company/unikraft-sdk](https://linkedin.com/company/unikraft-sdk)
Thank you!