Unikernels

Yiying Zhang
Announcements

- Project progress report due on Feb 16th
- Second quiz next Thur
  - Covers everything from container (1/23) to unikernels (today)
Unikernels!

Amir Chaudhry

… on behalf of a merry crew: Anil Madhavapeddy, Thomas Gazagnaire, David Scott, Thomas Leonard, Richard Mortier, Magnus Skjegstad, David Sheets, Balraj Singh, Jon Crowcroft, Mindy Preston, and many others!

PolyConf
July 2015
Software today…

…is an **application** …

… on top of an **Operating System**.
Software today…

… is built locally… … but deployed remotely.
Software today…

… needing more tools.
Software today…

…is complex!

Even though most apps are single-purpose
Complexity is the enemy...

- More layers -> tricky config
- Duplication -> inefficiency
- Large sizes -> long boot times
- More stuff -> larger attack surface
Why build for **clouds** as we do for **desktops**?

Can we do **better**?
Can we do better?

Disentangle applications from the OS
Break up OS functionality into modular libraries
Link only the system functionality your app needs
Target alternative platforms from a single codebase
The Rise of the Unikernel

Unikernels are specialised virtual machine images built from a modular stack adding system libraries and configuration to application code. Every application is compiled into its own specialised OS that runs on the cloud or embedded devices.

MirageOS
MirageOS

- Configuration Files
- Application Binary
- Language Runtime
- Kernel Threads
- User Processes
- Filesystem
- Network Stack

Mirage Compiler

- Application Code
- Mirage Runtime

unikernel
MirageOS

Familiar development cycle

Broad deployment scenarios
Lec2: Different OS Structures

User-Mode

Kernel-Mode

Monolithic Kernel

MicroKernel

ExoKernel (Library OS)

OS
App
Logic
Traditional Library OS

- Most OS functionalities implemented in the user space as libraries
- The kernel-space OS part only ensures protection and multiplexing
- Applications get to access hardware resources directly (faster)
- But isolation is hard
- And a lot of software (esp. device drivers) need to be rewritten

What if instead we run libOS on hypervisor (as VM)?
VM, Container, and Unikernel

Virtual Machines

Containers

Unikernels
VM, Container, and Unikernel

++ Strong isolation/security
- Heavy-weight

++ Weak isolation/security
- Light-weight

++ Strong isolation/security
++ Light-weight

Virtual Machines

Containers

Unikernels

+ Strong isolation/security
- Heavy-weight

+ Weak isolation/security
- Light-weight

+ Strong isolation/security
+ Light-weight
Unikernel Designs

• Integrating configurations into the compilation process
  • All related services, applications packed into a single application
  • Features not used are not compiled => extensive dead-code elimination

• Single-purpose libOS VMs perform only what the application needs and rely on hypervisor for isolation and resource multiplexing

• Within a unikernel VM, there’s no privilege difference between application and libOS (single address space)
Unikernel Benefits

• Lightweight
  • Only what the application uses is compiled and deployed

• Better security
  • Isolates lib OSes on hypervisor
  • Small attack surface
  • Single type-safe language, page table sealing, compile-time address space randomization

• Fits many new cloud environments well
  • Serverless, microservices, NFV
MirageOS

- Single (type-safe) language for everything
- Implemented in OCaml, runs on Xen
  - OCaml: a high-level language that is type safe (statically checked) and uses managed memory
- Single address space (process), with one virtual CPU
- Modified language runtime’s memory management and concurrency
- VMs cannot be cloned by COW an existing image
- Seal VM page tables (no writable and executable, no heap expansion)
MirageOS Memory Layout

- External I/O pages for communicating with other VMs
- Fast minor heap for short-lived values
- Larger major heap for long-lived values

Figure 2: Specialised virtual memory layout of a 64-bit Mirage unikernel running on Xen.
Zero-Copy Device I/O

Figure 4: Example zero-copy write for an HTTP GET. The application writes its request into an I/O page, and the network stack segments it into fragments to write to the device ring. When a response arrives the pages are collected and the write thread notified.
Results - Boot Time

![Graph showing boot time vs memory size for Mirage and Linux PV]

Figure 6: Boot time using an asynchronous Xen toolstack.
Results - Running DNS Server

Figure 10: DNS performance with increasing zone size.
## Results - Image Size

<table>
<thead>
<tr>
<th>Appliance</th>
<th>Binary size (MB)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard build</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>DNS</td>
<td>0.449</td>
<td>0.184</td>
</tr>
<tr>
<td>Web Server</td>
<td>0.673</td>
<td>0.172</td>
</tr>
<tr>
<td>OpenFlow switch</td>
<td>0.393</td>
<td>0.164</td>
</tr>
<tr>
<td>OpenFlow controller</td>
<td>0.392</td>
<td>0.168</td>
</tr>
</tbody>
</table>

Table 2: Sizes of Mirage unikernels, before and after dead-code elimination. Configuration and data are compiled directly into the unikernel.

- Typical Linux image size is ~400MB
Other Unikernels

- OSv: new OS designed to run single application
- Rumprun: running unmodified POSIX software as a unikernel
- runtime.js: libOS that runs JavaScript
- IncludeOS: libOS for running C++ code on virtual hardware
- ClickOS: A high-performance, virtualized software middlebox platform (e.g., for NFV)
- Clive: an OS designed to work in distributed and cloud environments, written in Go
- Earlang on Xen

Find more info at [http://unikernel.org/projects/](http://unikernel.org/projects/)
Virtualization is a mixed bag

- Good for isolation, but...

- Tooling for VMs not designed for lightweight (e.g., lightVM)
- How do you debug black-box VMs?
- Poor VM performance due to vmexits
- Deployment issues on already-virtualized infrastructure
Why not run unikernels as processes?

• Unikernels are a **single process** anyway!

• Many benefits as a process
  • Better **performance**
  • Common tooling (gdb, perf, etc.)
  • ASLR
  • Memory sharing
  • Architecture independence

• **Isolation** by limiting process interface to host
  • 98% reduction in accessible kernel functions
Unikernel architecture

• ukvm unikernel monitor
  • Userspace process
  • Uses Linux/KVM

• Setup and loading
• Exit handling
Unikernel as process architecture

• **Tender**: modified ukvm unikernel monitor
  - Userspace process
  - Uses `seccomp` to restrict interface

• Setup and loading
• “Exit” handling

[Diagram showing the process architecture with steps:]
1. Set up I/O fds
2. Load unikernel
3. Configure seccomp

Linux
Unikernel isolation comes from the interface

• Direct mapping between 10 hypercalls and system call/resource pairs

• 6 for I/O
  • Network: packet level
  • Storage: block level

• vs. >350 syscalls
Implementation: nabla

• Extended Solo5 unikernel ecosystem and ukvm
• Prototype supports:
  • MirageOS
  • IncludeOS
  • Rumprun

• https://github.com/solo5/solo5
**Results**

- Unique kernel functions accessed: normal processes and VMs have 5-6x and 2-3x more than Nabla.
- Application throughput: Nabla 101% - 245% higher than ukvm.
- CPU utilization: Nabla has 12% reduction over ukvm.
- Startup time: ukvm takes 30-370% longer.
Conclusion

- Library OS (Exokernel) is an old idea aiming to expose more hardware interface directly to applications running in user space

- Unikernels: run app+libOS as VMs on hypervisor
  - Better isolation
  - Much more lightweight
  - But need a lot of reimplementation and can’t use existing tooling

- Unikernels as processes: run app+libOS as processes on host OS, limit interface to OS for security
  - Can use existing tooling, more flexible
  - Even more lightweight

{Better fit for modern cloud environments}