Super Containers: Unikernels and Virtual Machines

South Carolina Linux Users Group

Brad Whitehead, Chief Scientist – Formularity

July 11, 2023

Good evening. I'm Brad Whitehead. I'm the Chief Scientist at Formularity.

Who is Brad Whitehead?

- Former Partner and Master Technology Architect with Accenture
  - National Scale Biometric Identification and Border Management Systems
    - US VISIT (one of CBP's Biometric Border Control programs)
    - DHS Transportation Worker's Identification Card (TWIC)
    - TSA PreCheck
    - Republic of India's Aadhaar Program
- Presently Co-Founder and Chief Scientist of Formularity
  - Secure Electronic Enrollment Forms for the Government, Healthcare, Finance, and Legal Industries
  - We Offer a Hosted Solution, in Addition to Our Primary, On-Premise Products
  - We Take the Security of Our Client's Information VERY Seriously
  - Currently Open Sourcing Aspects of the Aadhaar Biometric Processing Technology
  - **We Constantly Investigate and Explore New Advances in Information Security and Assurance**

Prior to helping to founding Formularity, I was a Partner and Master Technology Architect at Accenture. My specialty, if you will, was national level biometric identification and border management systems. I was Accenture's Chief Architect on US VISIT, the basis of he fingerprint identification system that Customs and Border Protection (CBP) uses when you enter the country, the Transportation Worker's Identification Card (TWIC) system. You need a TWIC card to work at a US Port or to drive a truck with hazardous chemicals, like gasoline. And TSA Pre-Check. I'm sure everybody here is familiar with Pre-Check. These are all fingerprint identification systems. They are also "guns and badges" systems, where the Government tells you what you and can't do, based on your fingerprints. However, my biggest source of pride is having been Accenture's Chief Architect on the largest human rights and social enablement program in the world. This is the Republic of India's Aadhaar biometric identification system.

My present employer, Formularity, develops electronic enrolment forms for large scale benefits and services management systems. Think – enrollment in a National Health Care program ;-)

Since we handle personally identifiable information, we take security very seriously. As part of this, we routinely review and assess new technologies in security, with a eye toward applying it to our business. This evening I'm going to be discussing one of these promising technologies that we have our eye on and have done some investigation – SuperContainers!

“Super Container”

- 1400 Times more Secure than a well-configured Docker Container
- Boots 37 times Faster than that Docker Container
- Can run 10 times more Microservices on the same physical hardware
- Can be managed by Kubernetes or Apache Mesos

Super Containers - That’s not their real name, but what else would you call a container that’s:

- 1400 Times More Secure Than A Well-configured Docker Container
- Boots 37 Times Faster Than That Docker Container
- Can Run 10 Times More Microservices On The Same Physical Hardware
- Can Be Managed By Kubernetes Or Apache Mesos

What Is This Incredible New Technology?

It’s a Unikernel Image running on a Lightweight Virtual Machine Hypervisor

What Is This Incredible New Technology?
It’s A Unikernel Image Running On A Lightweight Virtual Machine Hypervisor

What's A Unikernel?

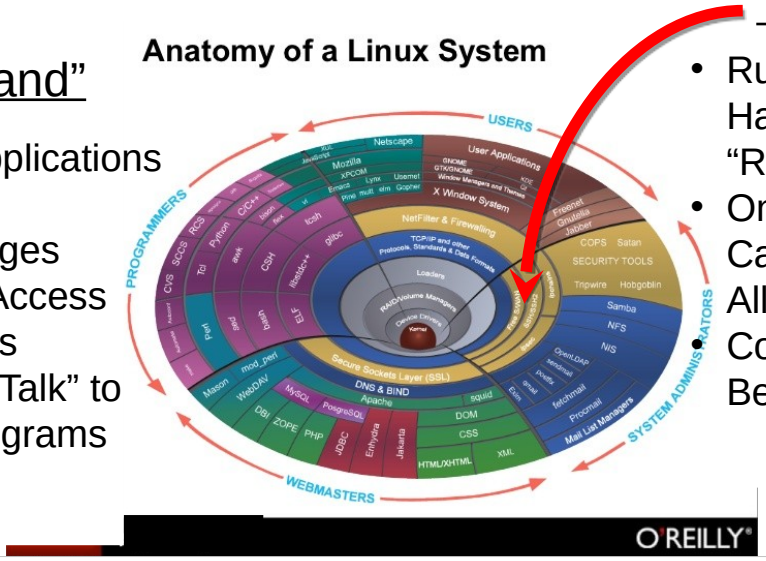
- To answer that question, we have to take a look at the structure of a modern Operating System
- Doesn't matter if it's Microsoft Windows, Linux, UNIX, ~~Mac OS X~~ ~~OS X~~ macOS, etc.
- All the mainstream Operating Systems have the same fundamental anatomy

To answer this, let's take a look at a modern operating system
Doesn't matter if we choose Windows, Linux, UNIX, or whatever Apple's calling their desktop operating system these days!
They are all basically the same

The Anatomy of An Operating System

"UserLand"

- Where Applications Run
- No Privileges
- Can Not Access Resources
- Can Not "Talk" to Other Programs



"The Kernel"

- Runs in Special Hardware Mode – "Ring 0"
- Only Program That Can Access or Allocate Resources
- Copies Data Between Programs

Anatomy of an Operating System
User Interface (CLI or GUI)
Standard set of tools and applications ("Userland")
Kernel (privileged, separated from Userland by hardware)
Monolithic (Linux even includes a web server in the kernel!)
Famous flame war between Torvald Linus and Dr. Andrew Tanenbaum, 1992 Usenet
Microkernel (Mach, Minix)
Context switches
data copying

Growth of Operating Systems

- Linux Kernel is now 30 Million Lines Of Code!
- Windows is estimated at 50 Million Lines Of Code!
- With an industry average Of 15-50 Defects Per 1000 Lines Of Code*:
  - Linux(Just the Kernel) = 330,000 To 1.1 Million Defects
  - Windows = 750,000 To 2.5 Million Defects

*(Steve McConnell, “Code Complete 2”, 2005)

Operating system kernels have grown enormously over time!
Linux 30 million SLOC
Windows estimated to be 50 million SLOC
Now, Steve McConnell has conducted a number of studies on the defect rates in modern system software. He estimates that there are between 15 and 50 defects per 1000 lines of code!
Now not all defects result in errors. A defect is anything that does not meet the requirements or is not as intended by the developer. So a defect could be as simple as a misspelled word in an error message. Or as serious as a buffer overrun or a “use after free” pointer error!
Using McConnell’s figures, the Linux kernel - just the kernel - has 330,000 to 1.1 million defects
Windows has 750,000 to 2.5 million defects

It Gets Worse!

- The “Userland” Support Software is often 10 to 20 times larger than the Kernel!
- Red Hat Enterprise Linux (RHEL) Userland is approximately 420 Million Lines Of Code
  - Try not to think about the 6.3 to 21 Million Defects running on your Bank’s Server!

That was just the kernel. The kernel needs support software to boot and to perform standard services. This support software, along with all the other applications that come with an operating system distribution, is that userland I referred to a moment ago. Red Hat Enterprise Linux is the leader in Linux software for businesses. Their userland software is around 420 million lines of code. Again, taking McConnell’s numbers, each of your your bank servers could have 6.3 to 21 million defects running!

Can It Get Even Worse?!?!

- The Kernel Is Full of Junk!
- A large number of Device Drivers are routinely compiled into the kernel, regardless of the actual hardware in the computer
  - There are Device Drivers for hardware that no longer exists
  - Amazon Ami Images ~~Have~~ Had Drivers for Floppy Disks and Audio Cards
    - In 2015, The Venom Vulnerability (CVE-2015-3456) used a flaw in the Floppy Disk Controller (FDC) Driver to compromise both Physical and Virtual Machines

Why is the kernel so big!?!? For one thing, a large number of device drivers are routinely compiled into the kernel. That way, whether you load the operating system on an IBM server or a SuperMicro, it just works - magic! There are device drivers in the kernel for hardware that no longer exists. For a number of years, Amazon's standard Linux images had floppy disk and audio card drivers compiled into them. How many floppy drives are there in an Amazon data center? Who uses those audio cards? In 2015, the Venom malware used a defect in the floppy disk driver to compromise both the VM in which it was running, as well as the physical host. How many other device driver defects are out there, waiting to be exploited?

Can It Get Even Worse?!?! (Continued)

- Likewise, there are thousands of Storage and Communications Protocols in the Kernel that will not be used in your Application
- Linux recognizes 7 different Executable Formats, even though the vast majority of applications (including yours) are in ELF Format
- **Each of these Extra, Unused Chunks Of Code (with its 15-50 Defects/1000 SLOC) is a Potential Hack waiting to happen!**

It's not just device drivers. There are a large number of file system drivers and communications protocols compiled into the kernel. Many, most, of these are esoteric and probably won't be needed by your applications. But they are still sitting there, taking up space, processing power, and the reliability budget

What If We Cut Out All The Parts We Don't Need?

- Code Traces show that the average Application uses less than 0.08% of the total code in the Kernel!
- Take the Standard C Library as an example:
  - The C Library contains thousands of Functions, but a modern linker only includes the actual Functions (and Code) That an Application uses
- Could we do the same with our Operating System?

When you code trace an average application, you find that it only uses 0.08% of the code in the kernel! Wouldn't it be great if we could jettison the extra 99.92% of the kernel we don't need?

Most of our modern libraries and linkers do this type of jettisoning. The C library has thousands of functions, but when it's linked into an executable, only the functions that are actually used are linked into the code.

Wouldn't it be great if our kernel only contained the functions we needed?

What About Actors (Microservices)?

- Run a Single Application
- As a Single User
- With a known set of Hardware Drivers
- 1 Or 2 Communications Protocols
- Important:
  - Speed (Startup and Latency)
  - Reliability
  - Security (from Unauthorized Access - “Hacking”)
  - Repeatability (Multiple Identical Servers)

So what are the functions we need?

Let’s make up a list. Let’s assume our application is a microservice or perhaps a Internet of Things application

- Run A Single Application
- As A Single User
- Known Set Of Hardware Drivers
- 1 Or 2 Communications Protocols
- Speed (Startup And Latency)
- Reliability
- Security (From Unauthorized Access - “Hacking”)
- Repeatability (Multiple Identical Servers)

Keeping Only The Parts of the Operating System We Actual Use

- What does it buy us?:
  - Let’s start with Security:
    - Greatly Reduced Attack Surface (99.92% Reduction)
    - Potentially a small enough subset to be Mathematically Verifiable
    - We don’t need any Userland Applications (Bye-bye 410 Million Lines of Potentially Flawed Code!)
    - No ability to run Malicious Code or Hacking Tools on our Server or IoT Device

So, when we strip away 99.92% of the kernel code, what do we get?
Well, first - Security
Reduced attack surface – .08% of typical
Small enough to possibly be mathematically verified
No tools (no shell, etc.)

More Benefits

We can Statically Link everything (including the Kernel Functions) and our software becomes Immutable

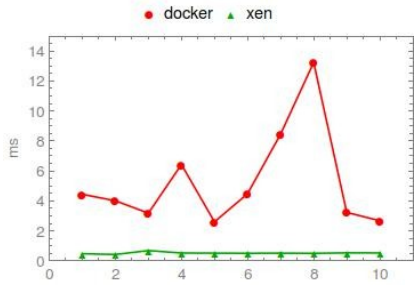
- No Injection Attacks
- No Re-Configuration Attacks
- Vastly Reduced “Return Oriented Programming” (ROP) Vulnerability

Increased Reliability and Improved Security means Reduced Devops Costs!

We can statically link everything together
At that point, our code becomes immutable
Modules and dynamic libraries can’t be added, code can’t be injected

Increased Performance

- Smaller, less memory intensive Images mean more Virtual Machines per Hardware Server
  - 5 Megabyte Virtual Machines = 10,000 VMs per Physical Server
  - Smaller than most Docker Containers
- 6 Millisecond Boot Times
- 45 Microsecond Throughput Times because:
  - No Context Switches
  - No Information Copying
  - Single Address Space



We also get huge performance gains!

Smaller instances or more VMs per instance - 5MB per VM, 10K VMs/hardware server

6 millisecond boot times

Since all the code is running in Ring 0 as privileged code, there are no context switches and no need to copy memory between kernel and applications

Server-less Functions (with servers)! – 45 microsecond response

How To Include Only The Needed Code?

- Again, The C Library analogy is the key
  - The C Library is actually a “Middleware Layer”
  - It converts standard C Function Calls into equivalent Kernel System Calls
  - Instead of handing the Function Call off as a System Call, What If we extended the C Library to include the appropriate Kernel Code?
    - Instead of the C Library passing a “Printf()” Call to the Kernel, the Library can include the machine instructions to do he actual I/O

How do we include only the required kernel code?

Again, let’s look at the the C Library. The C Library presents a standard POSIX interface to the application’s C code.

When a function in the library is called, the library prepares all the parameters and then executes specific operating system calls.

Why don’t we combine the C library and the kernel code it calls? Then, when we link in the C library code, we get both the POSIX interface code and the underlying kernel functionality

In a “Library Operating System”

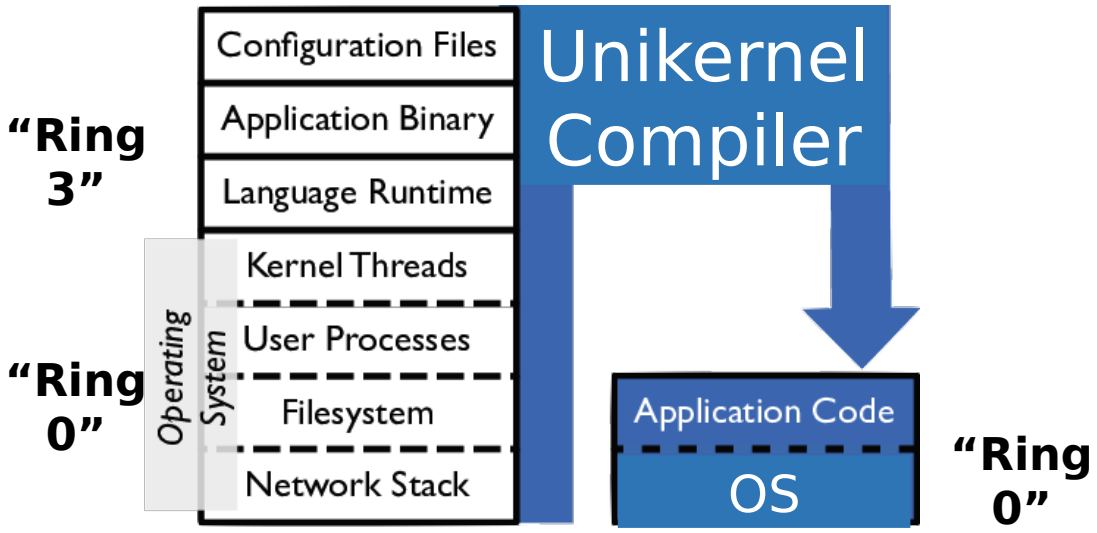
- Common Operating System Functions, Drivers, And Protocols are Written As A Library Of Functions
- When You Link these “Library Operating System” Functions to your Application, you have a single Executable that runs directly on hardware or a Virtual Machine…

…In other words, you have a Unikernel!

This approach is called, obviously enough, a “library operating system”

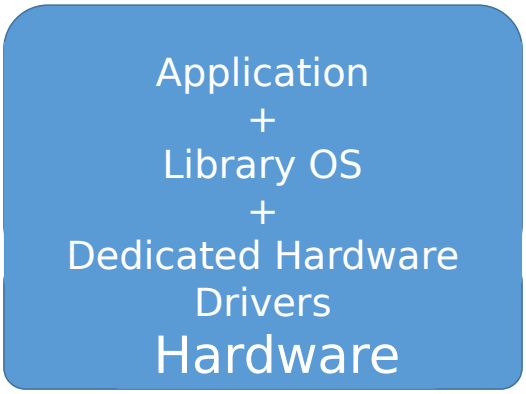
- Common Operating System Functions, Drivers, And Protocols Are Written As A Library Of Functions
- When You Link These “Library Operating System” Functions To Your Application, You Have A Single Executable That Runs Directly On Hardware Or A Hypervisor…
- in other words, you have a Unikernel!

What Does A Unikernel Look Like??



Visually, this is what a unikernel looks like:
On the left, you have the conventional software stack. The bottom four layers are operating system code. They run in Ring 0 and can access the hardware directly. The top three layers are userland and application code. They are unprivledged and run in Ring 3.
On the right hand side is our unikernel. The unikernel compiler has extracted only the operating system functionality we need and combined it with our application code. For the most part, we don't need any userland code. Our new unikernel runs in Ring 0

For IoT, It's Even Simpler



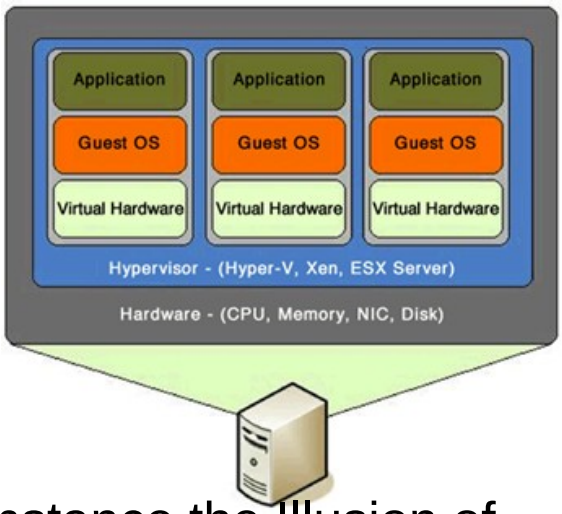
The situation is even better for IoT applications. Especially since we know exactly what hardware we are running on!

However, Unikernels are only half the answer to Small, Fast, Secure Containers

The other half is the Light(er) Weight Virtual Machine

Unikernels Are Only Half The Answer To Small, Fast, Secure Containers
The Other Half is the Light(er) Weight Virtual Machine

Virtual Machines



- Virtual Machine Monitor (VMM) Or “Hypervisor” typically sits between the real Hardware and Multiple Operating Systems
- Gives each Operating System Instance the Illusion of running on its own Hardware – A “Virtual Machine”
- **Strong** Physical Isolation between Operating Systems

I don’t want to spend a lot of time on virtual machines. I assume most of you are familiar with them, at least to the degree we need in order to discuss super containers.

- Virtual Machine Monitor (VMM) Or “Hypervisor” Typically Sits Between The Real Hardware And Multiple Operating Systems
- Gives Each Operating System Instance The Illusion Of Running On Its Own Hardware – A “Virtual Machine”
- Strong Physical Isolation Between Operating Systems

Containers Have Changed The Way We Develop and Deploy Software

- Applications are deployed as complete Images, ready to run, instead of being “Installed”
- Containers are Replaced, rather than being “Patched”
- Containers support the concept of “Microservices”, allowing complex Applications to be built from Single-Function Services wired together through Orchestration Managers
- Multiple Containers can be started and stopped in response to Traffic Loads

We don’t develop software the way we used to. Wide-spread use of containers have changed the development and deployment front, based on their characteristics

- Applications Are Deployed As Complete Images, Ready To Run, Instead Of Being Installed
- Containers Are Replaced, Rather Than Being “Patched”
- Containers Support The Concept Of “Microservices”, Allowing Complex Applications To Be Built From Single-function Services Wired Together Through Orchestration Managers
- Multiple Containers Can Be Started And Stopped In Response To Traffic Loads

Drawbacks of Containers

- Limited Isolation between Containers – Not a Security Mechanism
- Difficult to strip down Userland and Container Images
  - Bloat consumes Memory and Processing Resources
- Differences between the Production and Development Environments

But Containers have a number of drawbacks...

- Limited Isolation Between Containers – Not A Security Mechanism
- Difficult To Strip Down Userland And Container Images
  - Bloat Consumes Memory And Processing Resources
- Differences In Production And Development Environments - Containers are generally “sold” on the fact that development and production are supposed to be the identical environments, but in reality, they never are. You have development and debugging tools in the Development Container that you (better!) remove from Production. Hence the two environments are different. The difference may not make a different (should not make a difference), but then again an accidentally unsatisfied dependency may be discovered in Production at 4AM

Meanwhile, Virtual Machine Technology Has Not Stood Still

Recent optimizations to both the Xen and the Linux “Kernel-based Virtual Machine” (KVM) Hypervisors, as well as new Hypervisors like Firecracker have:

- Significantly reduced the start-up time of a Virtual Machine
- Reduced performance-robbing overhead

This New Generation Of Hypervisors Are Called **“LightVMs”**

- Significantly Reduced The Start-up Time Of A Virtual Machine
- Reduced Performance-Robbing Overhead

This New Generation Of Hypervisors Are Called “LightVMs”

Next Generation LightVMs - Speed

- Combined with Unikernels, these LightVMs can launch Microservices in as little as 4 milliseconds
- This is comparable to the Linux kernel’s Exec/Fork times of approximately 1 millisecond and significantly faster than a Docker-type container’s start-up time of 150 milliseconds

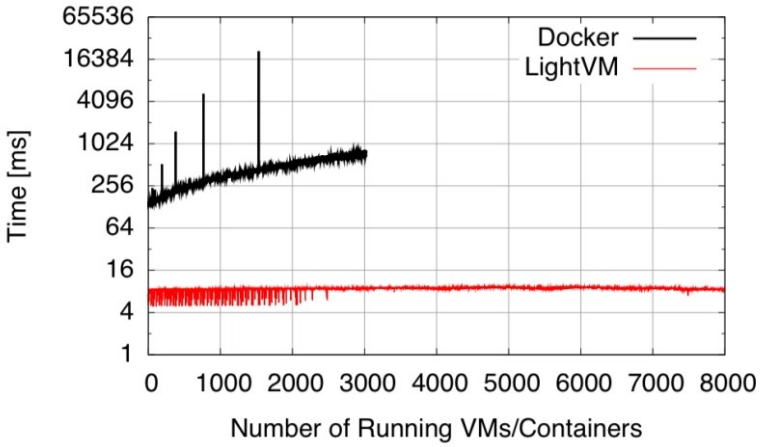
- Combined With Unikernels, These LightVMs Can Launch Microservices In As Little As 4 Milliseconds
- This Is Comparable To The Linux Kernel’s Exec/Fork Times Of Approximately 1 Millisecond And Significantly Faster Than A Docker-type Container’s Start-up Time Of 150 Milliseconds

Next Generation LightVMs - Size

- Additionally, the reduced footprint of the Unikernel requires only about 1/10th the memory of a Docker-type Container running on a Debian kernel
- Since memory is quite often the limiting factor in properly designed Microservices, this means that 10 times more Unikernel/LightVM Microservice Instances can be run on the same physical hardware.

- Additionally, The Reduced Footprint Of The Unikernel Requires Only About 1/10th The Memory Of A Docker-type Container Running On A Debian Kernel
- Since Memory Is Quite Often The Limiting Factor In Properly Designed Microservices, This Means That 10 Times More Unikernel/LightVM Microservice Instances Can Be Run On The Same Physical Hardware.

LightVM Unikernel vs Docker



LightVM Boot Times On A 64-Core Machine With 128GB Memory vs Docker Containers

Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and FelipeHuici. 2017. My VM is Lighter (and Safer) than your Container. In Proceedings of SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles, Shanghai, China, October 28, 2017 (SOSP '17),16 pages.https://doi.org/10.1145/3132747.313276

The top line are Docker Containers being launched, starting at 150 milliseconds, going to 1 full second at the 3000th container launched. The line doesn't extend beyond 3000 containers because the physical machine ran out of memory (128GB) at that point.

Now, notice the red line. Those are equivalent functionality unikernels. They all launch at a fairly consistent 4 milliseconds. Memory exhaustion now occurs at north of 8000 unikernel VMs on the same hardware.

Remember the old "C10K challenge" - How many http sessions a single web server could support? Well the new equivalent is "VM100K" - running 100,000 VMs on the same physical host. Think what that does to cloud computing economics. Amazon going from 300 VMs per physical server to 100,000 VMs!

Practical Unikernels

Unikernels have, until recently, been the province of laboratories and research projects

This has changed as Unikernel technology has matured:

- More complete Function Libraries
- More mainstream programming languages are now supported

- More Complete Function Libraries
- Mainstream Programming Languages

One Approach – Reuse - AnyKernel

- NetBSD, A version of UNIX, is famous for its ability to be ported to new Hardware
- It's a Monolithic Kernel, but has been internally structured into well defined Functions and Layers
- A Library of NetBSD Functions has been created, called "The Anykernel" concept
- The Anykernel concept allows existing Application Code, designed for Linux or UNIX (POSIX) Operating Systems to be statically linked with Operating System Functions and Drivers, forming a Unikernel!

- NetBSD, A Version Of UNIX, Is Famous For Its Ability To Be Ported To New Hardware
- It's A Monolithic Kernel, But Has Been Internally Structured Into Well Defined Functions And Layers
- A Library Of NetBSD Functions Has Been Created, Called "The Anykernel" Concept
- The Anykernel Concept Allows Existing Application Code, Designed For The Linux Or UNIX (POSIX) Operating Systems To Be Statically Linked With Operating System Functions And Drivers, Forming A Unikernel!

Another Approach – Ground Up – NanoVMs' OPS

- What access does a modern Cloud Application require?
  - A Packet Interface for network communications
  - A Block Interface for storage
  - A Serial Port to output console data
- NanoVMs Corp. wrote these interfaces (and other necessary POSIX interfaces) from scratch in C++
- Vast majority of existing C and C++ Applications will link successfully with NanoVMs' OPS Unikernel

- What Access Does A Modern Cloud Application Require?
  - A Packet Interface For Network Communications
  - A Block Interface For Some Storage
  - A Serial Port To Output Console Data
- NanoVMs Team Wrote These Interfaces (And Other Necessary POSIX Interfaces) From Scratch In C++
- Vast Majority Of Existing C And C++ Applications Will Link Successfully With NanoVMs' OPS Unikernel

Practical Unikernels and Library Operating Systems

- MirageOS (Written in OCaml)
- ClickOS (Runs Click NFV language)
- HaLVM (Written in Haskell)
- Ling (Written in Erlang)
- Hermitux – (Written in C++)
- Rusty Hermit – (Written in Rust)
- RumpKernel (NetBSD AnyKernel - Written in C/C++)
- NanoVMs' OPS (Written in C/C++)

With The last three, you can develop Unikernel Applications in Python, Ruby, Node, Java, Rust, Fortran, and more...

Practical Unikernels and Library Operating Systems
MirageOS
RumpKernel
ClickOS (runs Click NFV language)
HaLVM (Haskell)
Hermitux – (Written in C++)
Rusty Hermit – (Written in Rust)
RumpKernel (NetBSD AnyKernel - Written in C/C++)
NanoVMs' OPS (Written in C/C++)

With The last three, you can develop Unikernel Applications in Python, Ruby, Node, Java, Rust, Fortran, and more...

VM Unikernel Applications Can Be Managed With The Same Tools As Containers

- CoreOS ‘rkt’ can run Unikernel VMs with Docker Swarm, Kubernetes, or Apache Mesos management engines
- Kubernetes:
  - Kubevirt
  - Virtlet
  - RancherVM

The good news is that managing our paradigm-shifting unikernels does not require any further paradigm shift ;-)
We can manage unikernels using the same tools as containers; Kubernetes, Mesos, Swarm, etc., using adapters
These adapters include:
Kubevirt
Virtlet
and RancherVM
CoreOS rkt can natively manage virtual machines, as well as containers

Drawbacks
“Every Rose Has Its Thorn”

- Unikernels in LightVMs is a new paradigm
- Lack of experience
- Limited selection of Libraries and Build Tools
- Existing Applications may require modification
- May be more difficult to develop and debug*

* red herring

Drawbacks?

Hardware or hypervisor specific drivers

Existing applications may not run correctly in a shared memory model

The development and debug arguments might be relavent to how we used to develop software, but with the newer practices that have come about because of Containers, developing and debugging a Unikernel application is actually easier than a container app.

Olivia just sent me an article that seems to imply that we can even do remote gdb debugging.

Further Resources

- Worried about IoT DDoS? Think Unikernels, Levine, Idit, 4/14/2017 (https://github.com/solo-io/unik/wiki/Worried-about-IoT-DDoS%3F-Think-Unikernels)
- Enterprise IoT Security and Scalability: How Unikernels can Improve the Status Quo, Duncan, Bob; Happe, Andreas; Bratterud, Alfred; IEEE Xplore, 3/20/2107 (https://ieeexplore.ieee.org/document/7881647)
- Unikernels + connected devices, Ryd, Thomas, 9/8/2016 (https://mender.io/blog/unikernels-connected-devices
- Unikernels Are Unfit for Production - Bryan Cantril https://www.tritondatacenter.com/blog/unikernels-are-unfit-for-production
- What is a unikernel, and why does it matter?, Hewitt Packard Enterprise, 10/2/2017 (https://www.hpe.com/us/en/insights/articles/what-is-a-unikernel-and-why-does-it-matter-1710.html)
- Debunking Unikernel Criticisms, Oliver, Kiran; Jackson, Joab, 10/21/2016 (https://thenewstack.io/utilizing-unikernels-within-internet-things/)
- Making operating systems safer and faster with 'unikernels', University of Cambridge, 1/28/2016 (https://www.cam.ac.uk/research/news/making-operating-systems-safer-and-faster-with-unikernels)
- A unikernel experiment: A VM for every URL, Skjegstad, Magnus, 3/25/2015 (http://www.skjegstad.com/blog/2015/03/25/mirageos-vm-per-url-experiment/)
- Unikernel, Wikipedia, 1/5/2018 (https://en.wikipedia.org/wiki/Unikernel)
- My VM is lighter (and safer) than your container, Manco et al., SOSP'17 (http://cnp.neclab.eu/projects/lightvm/lightvm.pdf)
- Unikernel Monitors: Extending Minimalism Outside of the Box , Williams, Koller, 6/20/2016 (https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_williams.pdf)
- https://github.com/cetic/unikernels – EXCELLENT overview of unikernels, microkernels, monolithic kernels, and containers. Presents some surprising, but limited, benchmarks

Resources

Copies of the Slides May Be Downloaded From

the

Formularity Website

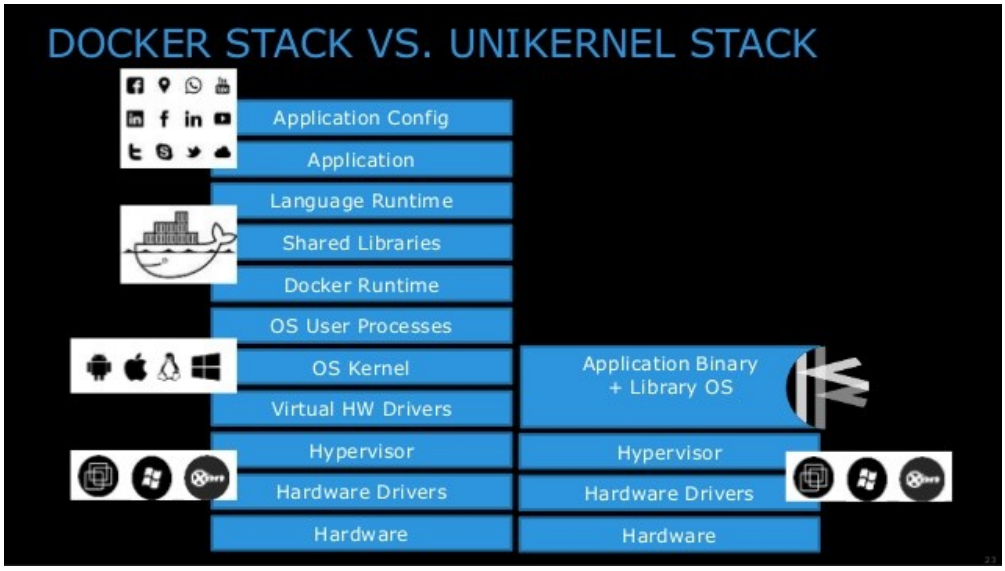
https://formularity.com

Given the security problems of current full operating systems, I truly believe that unikernels are a very promising technology to enhanced business (microservice) and IoT device security. Thank you! Copies of these slides and my talking notes on the Formularity website. Are there any questions?…

Shall we temp the demo gods and try a few demonstrations on how easy unikernels can be compiled and run?

Backup Slides

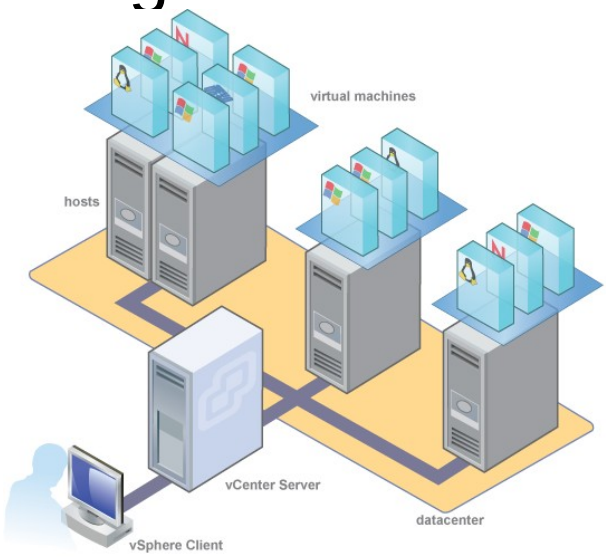
Containers vs Unikernels



So, this is what our microservices computing stack looks like with unikernels. We cut out a lot of unnecessary, defect-ridden layers, and merge other layers. We have a single executable, with no dependencies, ready to run on physical hardware, or under any hypervisor

Cloud Computing

- Virtual Machines Are The “Fuel” Of Cloud Computing
- Multiple “Virtual Machines”, Each With Its Own Operating System
- Each Virtual Machine Isolated And Managed By The Virtual Machine Monitor Or Hypervisor



- Virtual Machines Are The “Fuel” Of Cloud Computing
- Multiple “Virtual Machines”, Each With Its Own Operating System
- Each Virtual Machine Isolated And Managed By The Virtual Machine Monitor Or Hypervisor

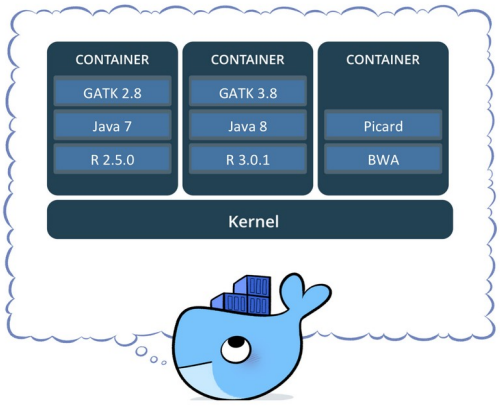
Drawbacks to Current Virtual Machines

- Size - Each VM Requires Its Own Operating System, Userland Software, And A Certain Amount Of Dedicated Memory, Making VMs **BIG**:
  - VMware – Max Of 380 VMs Per Physical Host
  - AWS Xen ~ 10 VMs Per Physical Core
  - AWS Nitro – 6000 VMs On A 36 Core Processor
- Speed - Slow To Startup – Boot Times Measured In Seconds And Minutes

- Size - Each VM Requires Its Own Operating System, Userland Software, And A Certain Amount Of Dedicated Memory, Making VMs BIG:
  - VMware – Max Of 380 VMs Per Physical Host
  - AWS Xen ~ 10 VMs Per Physical Core
  - AWS Nitro – 6000 VMs On A 36 Core Processor
- Speed - Slow To Startup – Boot Times Measured In Seconds And Minutes

Containers: Alternate to Virtual Machine

- A Container Is A Package That Bundles Up An Application And All Its Dependent Userland Software (Such As Libraries And Services) Into A Single Image
- A Container Runs Like A Pseudo-Virtual Machine, **Weakly** Isolated From The Host Processes And Other Containers
- All Containers On A Host Use The Host's Kernel
- While There Are Several Different Container Formats, Docker Is The Most Common



- A Container Is A Package That Bundles Up An Application And All Its Dependent Userland Software (Such As Libraries And Services) Into A Single Image
- A Container Runs Like A Pseudo-Virtual Machine, Weakly Isolated From The Host Processes And Other Containers
- All Containers On A Host Use The Host's Kernel
- While There Are Several Different Container Formats, Docker Is The Most Common

Notice in the diagram how there are two different versions of Java running side-by-side! Dependency isolation is a major feature of containers

Advantages of Containers

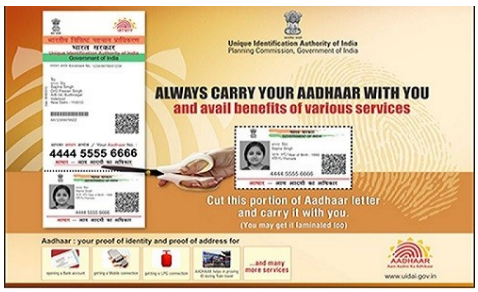
- Neatly Solves The Library Dependency And Versioning Problem (“DLL Hell”)
- Since The Kernel Is Already Running, Containers “Boot” In Milliseconds
- Less Dedicated Memory Is Required For A Container Than A Conventional VM
- “Orchestration” Software Has Been Developed To Deploy And Manage Containers
  - Kubernetes, Apache Mesos, Docker Swarm, et al
  - Google, Netflix

- Neatly Solves The Library Dependency And Versioning Problem (“DLL Hell”) - This is a Windows term, but the concept still applies to Linux and other operating systems that support dynamic library loading
- Since The Kernel Is Already Running, Containers “Boot” In Milliseconds
- Less Dedicated Memory Is Required For A Container Than A Conventional VM
- “Orchestration” Software Has Been Developed To Deploy And Manage Containers
  - Kubernetes, Apache Mesos, Docker Swarm, et al

UIDAI

Unique Identification Authority of India
Planning Commission, Government of India

AADHAAR



So, is anybody familiar with Aadhaar? India has the second largest population in the world – approximately 1.1 billion people. It is also one of the wealthest countries. Unfortunately, that wealth is very unevenly distributed. 80% of the population falls below the poverty line. Fortunately, India is very socially responsible and has large social programs to help the poorer citizens with food rations, cooking gas, guaranteed employment, etc. However, wherever you have lots of cash being distributed, you have lots of corruption and graft. Many of the people for whom these programs are intended have no “official” identity. No drivers license, no passport, often no fixed address. These people are then easily cheated out of their Government rights. The Unique Identification Authority of India (UIDAI) was founded under the Gandhi/Singh Congress Party administration and it continues under the Modi BJP administration. UIDAI’s Aadhaar program voluntarily enrolled over one billion Indians into a biometrically-based identification system. No longer can a corrupt supervisor fail to pay his or her workers. These workers can now open bank accounts and have their wages directly deposited, with an audit trail. Likewise, a rice merchant can’t give rice to his or her friends and then claim it went to people with ration cards.
As part of Aadhaar, we registered all 10 fingers and both irises of over one billion Indian residents. Actually, collecting and enrolling the biometrics was the easy part. After we collected a set of biometrics, we had to compare the new set with every other set that had already been enrolled, to keep people from accidentally or intentionally creating two or more identities

Accenture was one of the three original Biometric Service Providers for Aadhaar. We had to process 1 million enrollments per day. Day One was easy – Compare 1 million sets of biometrics against 1 million. Child’s play.

AADHAAR – “The Last Day”

Comparing 1 Million New Enrollees…

…Against 1 Billion Existing Enrollees

$1x10^6$ X $1x10^9$ = $1x10^{15}$

1 Quadrillion Comparisons in a day!

If you are using a database for workflow, that’s:

11 Billion Transactions/Second!

(It’s Even More If You Individually Log Each of 10 Fingers, 2 Irises, and a Face)

Now, there is no indexing or hashing system for fingerprints. The system must compare each fingerprint or iris print against every existing print. What a database person would call a full table scan. Now, consider the “last day” of Aadhaar – we have to compare our daily 1 million enrollees against the 1 billion existing enrollees! 1 Quadrillion comparisons or 11 billion transactions per second!

I took over as Acccenture’s Chief Architect on the program when we ran into scaling issues. Initially, we started by using the industry leading biometric middleware. This product uses an Oracle database as its workflow engine.

When we started processing over 500 thousand biometric enrollments per day, we hit the transaction limits of the Oracle system. And I don’t mean the limits for our particular hardware infrastructure, I mean the absolute limits of the Oracle database. Oracle scaled, at the time, using an architecture they call Real Application Clusters.