Live Updating in Unikernels

Live Updating IncludeOS While Preserving State

Alf-Andre Walla

Thesis submitted for the degree of
Master in
60 credits

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Live Updating in Unikernels

Live Updating IncludeOS While Preserving State

Alf-Andre Walla

Live Updating in Unikernels

Alf-Andre Walla

1st August 2017

Abstract

Dynamic software updating (DSU) is quite old by computer science standards, the term first appearing in 1983 addressing the upgrading of programs while they are running [12]. This thesis introduces dynamic software updating for the unikernel IncludeOS. It is one of very few operating system-based DSUs, simply called *LiveUpdate*, and it is written in modern C++. The *LiveUpdate* DSU will be evaluated updating live services running on a variety of cloud platforms, such as an IRC server, a SQlite database port, and a specialized test aimed to test preserving TCP connections.

The author will show that *LiveUpdate* is a capable and fast DSU that makes a big-picture difference for cloud providers, enabling novel features in cloud settings.

LiveUpdate is also highly relevant in the IoT-space as a live updating mechanism for the future billions of small devices that are going to be everywhere and in peoples homes.

Contents

List of Figures

Preface

The idea for this project was Alfred Bratterud saying live update was the holy grail for high availability.

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

Dynamic software updating allows systems to avoid the downtime associated with software updates [12].

Unikernels are specialized single-purpose operating systems. They are magnitudes smaller than general-purpose operating systems, resource efficient and have low potential attack surface simply from being small [21]. They are often used in the cloud, usually as single-purpose micro-services [3]. They can also be used in small Internet-of-Things (IoT) devices. Both are Internet-enabled and require security updates regularly to keep up with the latest threats. It can be a drag to do the updates in the case of cloud, due to cloud orchestration layers and long waiting queues. Maybe the cloud service must have no downtime. For the Internet-of-Things it can simply be that the device is managed by people unfamiliar with automatic updates and the dangers of ever-increasing cyber threats.

If there was a way to update services without stopping the systems or the service they were running, it would help everyone, and keeping an increasing percentage of Internet-enabled devices up-to-date is critical for the future. There is a large amount of research and material written on the impending disaster that is unpatched IoT-devices. We can already see this with smart phones, where not even 5 year old devices stop receiving updates from vendors or even earlier if the update channel is through service providers. However, what if the users are not trained in how to update their devices, such as their Internet-enabled fridge? Automatic live updates may help lower the threshold enough that more devices perform updates regularly, unattended. Vendors could, for example, remotely update all of its devices regularly, at the cost of some additional application complexity.

Dynamic software updating, also called *live updating*, is a common feature in non-stop systems [12], but while not strictly necessary elsewhere can be helpful nonetheless. If users did not have to experience updates at all, it could make updates wholly unattended for systems where they would normally interrupt the user. Dynamic software updating can also add and remove functionality from whole systems, or even replace one system with another while persisting selected state.

LiveUpdate as a dynamic software updating system is intended as a proof-of-concept practical implementation for library operating systems and Unikernels. It is the only modern operating system DSU out there (2017) that fully live updates operating systems, that the author is aware of. The reference implementation is for the IncludeOS unikernel. It takes a running IncludeOS service, stores selected state for later and then performs a fast system hotswap. The goal is to have various methods of safely updating small systems and services without service interruption.

1.2 Problem statement

How can live services in the cloud, or small always-on Internet-of-things devices be updated without causing service disruption?

- How can we verify the update binary is at minimum a complete and loadable ELF binary?
- How can a system be updated without persisting data, maintaining immutability?
- How can we present an API that leaves minimal room for data-type or versioning errors?
- How can we do automatic rollbacks in case something happens during or after the update?
- How can updates be done so that they leave no footprint after the update happened?
- How can we verify system consistency during and after the update?
- How fast can we live update, and does it meet real-time requirements?

1.3 Scope

This thesis is limited to live updating the IncludeOS unikernel. The author will be live updating a variety of common services, some more stateful than others, and observing the correctness and time spent during the update. There will also be experiments made to observe automatic rollback to a known good image during CPU exceptions, kernel panics and user-defined failures. There will not be updating of binaries for general-purpose operating systems (such as Linux).

IncludeOS is not fully able to run on bare metal as of writing this, nor on any non-x86 IoT-devices. IncludeOS has been shown to run on a Raspberry Pi with x86 support, however, the author unfortunately did not have this device when writing the thesis.

1.4 Response

This thesis introduces a specialized form of live updating process for Unikernels, written in modern C++. We store and restore state while maintaining, among other things, established TCP connections. The process described allows uninterrupted networking, given that implementors tailor their program to support a second code path during OS service initialization which will be restoring state stored from the old running version.

This thesis will show that service writers for single-purpose single address-space library operating systems (Unikernels) will be able to live update any service into any other service while preserving any and all state they require, and seamlessly resume execution in the order of milliseconds (and sometimes microseconds). The feature will have several consistency checks built in, and there are security precautions taken to remove traces of updates, but does not cover the methods of transferring updates to live systems. There is also support for automatic rollback to a known good kernel, if rollback is enabled. If not it will simply rollback to the image that was originally hosted on the system, which is a hypervisor feature accessed through a regular CPU reset.

Chapter 2

Previous Work

2.1 Kitsune and Ekiden

LiveUpdate is not originally based upon any existing DSU solutions, but it has a striking resemblance to Kitsune and Ekiden which also uses the state-transfer DSU style [10]. Kitsune goes much further and implements a domain-specific language specifically for state transformation. The author doesn't see any issues with using the same methods, or even porting Kitsune to work in Unikernels.

> Kitsune's updating mechanism updates the whole program, not individual functions. This mechanism is more flexible than most prior approaches and places no restrictions on data representations or allowed compiler optimizations. Second, Kitsune makes the important aspects of updating explicit in the program text, making the program's semantics easy to understand while minimizing programmer effort. Finally, the programmer can write simple specifications to direct Kitsune to generate code that traverses and transforms old-version state for use by new code; such state transformation is often necessary, and is significantly more difficult in prior DSU systems [10].

Part II

Background

Chapter 3

Cloud

3.1 Virtual Machines

A virtual machine is a separate computer running segregated from a host computer on the same hardware. The virtual machine can host anything from full operating systems down to custom unikernels just like the host machine running on real hardware. However, virtual machines are guests that run inside hypervisors which govern virtual hardware and memory as well as handling (trapping) privileged instructions. See figure 3.1.

This new virtualization is the abstraction that allows for separating the hardware from the operating system, simplifying things like migration that used to be very complex and enabling new complex features like live migration. It used to be that operating systems were installed for the hardware of the current machine, and couldn't be moved. With hardware virtualization this changes because modern hypervisors expose a generic backwards compatible hardware interface with generic extension card hardware, such as *Intel e1000* and later on para-virtualized hardware which works the same way on all architectures as long as they support *MMIO*.

For hardware virtualization to work, it requires the ability to trap into a hypervisor when executing sensitive instructions. On modern systems with better virtualization support trapping can also be done when writing to memory-mapped I/O (MMIO) locations, writing to model-specific registers (MSRs) and executing hyper-calls (VMCALL) etc. There is also typically hardware support for entering and leaving virtual machines, hardware paging support and direct device management through IOMMU hardware. It is important to note that a hypervisor need not implement any of these extra features, and can do its job just fine as long as the underlying architecture can at least trap on all sensitive instructions. For the common *Intel i386* (all the way up to i586) architecture this was not the case until Intel's VT-x (cpuid: vmx) and AMD's AMD-V (cpuid: svm), because of lack of support for trapping on sensitive instructions. The same was true for the initial 64-bit variant *AMD64*, which couldn't run virtual machines in software emulation. All of these problems were rectified with hardware supported virtualization, which also made virtualization very performant compared to software emulation and made virtualization available to

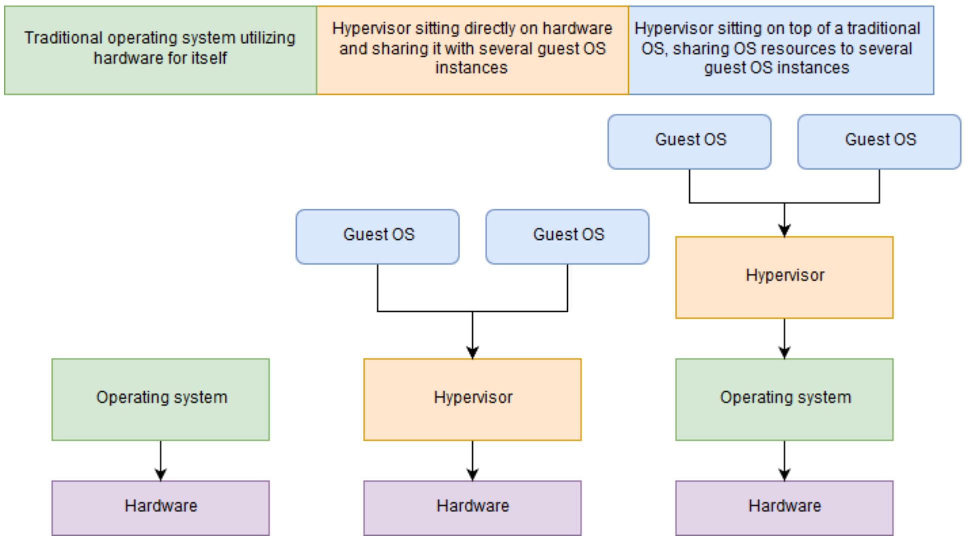

Traditional operating system utilizing hardware for itself
Hypervisor sitting directly on hardware and sharing it with several guest OS instances
Hypervisor sitting on top of a traditional OS, sharing OS resources to several guest OS instances
Guest OS
Guest OS
Guest OS
Guest OS
Hypervisor
Operating system
Hypervisor
Operating system
Hardware
Hardware
Hardware

Figure 3.1: An overview of running directly on hardware vs virtualized hardware

normal users. Today almost everyone has hardware virtualization support in their computers, and even laptops. This thesis was written on a laptop with VMX support, which allowed the author to quality- and fact-check a few things while writing, having access to full Linux as a Windows userspace application.

Requirements for Virtual Machines were originally defined in full in a paper by Popek and Goldberg as early as 1974 [29]. The biggest change from this paper and until today is that *equivalence* is no longer a desired property with the advance of both paravirtual interfaces and unikernels in the cloud. For the purposes of emulating other machines equivalence is still important, such as when running ARM machine code on x86.

We can think of virtual machines as programs that cannot directly interact with the host system, only indirectly through the rules given by the CPUs instruction set and the permissiveness of the hypervisor. It's possible to present virtual machines with an environment that is visibly equal to that of a system running on actual hardware. Much like how someone living inside The Matrix can do whatever they want within the rules of that system (and never really interact with the outside), when running a program inside a virtual machine that program can only execute CPU instructions and interact with hardware of the given architecture. Occasionally a trapping instruction is executed and the hypervisors decides whether or not the instruction is allowed and what its effects should be, and it will most likely be emulating the real hardware as closely as possible.

Today most hypervisors are made for full virtualization, which is desirable only if one intends to run an unmodified guest. Specialized operating systems for the cloud called *unikernels* want essentially to run inside virtual machines as if they were normal userspace single-purpose programs, running inside a layer of hardware isolation the architecture and hypervisor provides.

As a thought experiment; imagine running a program in a virtual machine that does timing operations. What stops the hypervisor from increasing the CPUs tick counter once every 2 years, effectively slowing down the world inside the virtual machine to a crawl? The guest system running inside the hypervisor has no other way to tell what speed it's running relative to. It would not be able to tell the time was going slow or fast compared to normal without outside help.

In reality, hypervisors function much like classic operating systems task schedulers, giving each virtual machine a share of the time, unless it's sleeping and waiting to be interrupted by hardware.

There is typically a limited list of features we would expect to have access to as an operating system:

- Basic memory protection features to help discover bugs, protect from basic exploits and segment the memory address space into meaningful parts.
- A way to hard-reset the system as a sort of off-and-on-again feature.
- A register we can count with to have a working stack (really only need 1 dedicated register).
- Access to the current date and time (otherwise we would have to get it via network communication, eg. via NTP).
- A CPU tick- or time-based invariant counter, which allows us to create a timer system as well as do some finer-grained scheduling.
- Hardware and software interrupts, allowing for asynchronous operations and communication, or optionally a hypercall API that exposes the equivalent functionality.
- A way to sleep until the next event or interrupt happens, the foundation for tick-less kernels [6].
- Network communication either via a paravirtual software interface or a regular hardware interface.

It is also very convenient to have some method for logging or remote inspection. Serial port output, for example, or simply logging remotely using the BSD Syslog Protocol [19]. The hardware (or hypervisor) does not need to present this to the operating system, but it is still an integral part of the whole.

A running guest cannot escape the virtual environment. There is no real way to escape a virtual machine into the hypervisor or into other virtual machines without exploiting rare hypervisor security holes. There are also major ongoing efforts into removing entire classes of bugs like these at a hardware level [16][35]. As a thought exercise imagine if all hypervisors were slightly different and had randomized memory layouts, how would you then exploit a known issue in a particular hypervisor? How do you even identify which hypervisor you are running inside if it doesn't willingly expose CPUID leafs that identify it? There are known timing algorithms that can at least show that you are (probably) in a virtual machine. Other than that, you can really only guess. Additionally, hypervisors rarely have bugs in the core functionality, but sometimes can have them in new drivers for new virtual hardware. Such virtual hardware has to be explicitly enabled by system administrators, as the core functionality of a current day hypervisor is simply the functionality of a generic variant of the architecture itself and a few generic block and network drivers.

3.1.1 Paravirtualization

Paravirtualization is a method of presenting devices or hardware-like interfaces used in virtual machines. They are different from real hardware in that guest operating systems need special drivers and they are intended to be very performant in comparison to emulating real hardware. Virtual machine guests know they are in a virtual environment if they can see paravirtual devices or CPU features. Paravirtualization can closely follow best practices and performance paths of an architecture as it is only software.

Each virtualization system typically has its own set of paravirtual drivers, especially for networking. VMware has the vmxnet, vmxnet2 and vmxnet3 paravirtual networking drivers. KVM on Linux has virtioblock for block devices, virtionet for networking, virtiorandom for a randomness source and many other devices. Anyone can implement such a driver, as long as they implement both the host side (qemu) and guest side (guest operating system, eg. Linux) variants of the driver.

The most prolific open source paravirtual specification on Linux is the Virtio standard:

> Virtio: a series of efficient, well-maintained Linux drivers which can be adapted for various different hypervisor implementations using a shim layer. This includes a simple extensible feature mechanism for each driver. We also provide an obvious ring buffer transport implementation called vring, which is currently used by KVM and lguest. This has the subtle effect of providing a path of least resistance for any new hypervisors: supporting this efficient transport mechanism will immediately reduce the amount of work which needs to be done. Finally, we provide an implementation which presents the vring transport and device configuration as a PCI device: this means guest operating systems merely need a new PCI driver, and hypervisors need only add vring support to the virtual devices they implement [33].

3.2 Cloud computing

3.2.1 Introduction

Cloud computing was popularized in 2006 with Amazons *Elastic compute cloud*.

Cloud is a broad term encompassing infrastructure, platforms, architectures and services as solutions running in a distributed system such as a local network or even the Internet.

There are several types of cloud and types of services that can be provided on them, however, for LiveUpdate the most relevant types are *software-as-a-service* and *function-as-a-service*.

3.2.2 Software-as-a-Service

Rented or free software hosted locally (expiration) or in a cloud (thin client). The users data is also typically stored in the cloud, and can be accessed from all of the users the devices where the software is available. Googles Gmail is email software provided as a service.

3.2.3 Function-as-a-Service and Serverless Computing

Serverless computing allows running code in the cloud without provisioning or managing a platform. The cost is typically measured in the time it takes to run the code. The code to be run is likely to be restricted to a few high-level all-in-one languages such as Node.js, Python, C# and Java. This is both because the languages can be modified by the cloud provider to be more safe to run on the underlying platform and the languages are reasonably platform independent. Certain system calls may be disabled, such as socket debugging, for security reasons. Others may be disabled because they lack equivalents on all platforms.

Amazon Lambda [11] is one of the few serverless *code-without-provisioning* services in the wild. Code running on Amazon Lambda must be written in a stateless style to allow the compute cloud to scale up functions after demand. Output from functions are logged to Amazon Cloudwatch, which is a monitoring service for the AWS Cloud.

Another example is the Google Cloud Vision API, which is single-purpose and is only used to identify descriptive words for images using a REST API.

Chapter 4

Unikernels

4.1 Introduction

4.1.1 Introduction

Unikernels are specialized single-adress-space machine images constructed from library operating systems [20].

> Unikernels are specialized OS kernels that are written in a high-level language and act as individual software components. A full application (or appliance) consists of a set of running unikernels working together as a distributed system [21].

The original quote from 2013 is slightly outdated today as unikernels are also perfect for use in the Internet-of-things, running on bare metal, and does not need to be running in a distributed system. Unikernel.org does not limit the definition in the same way.

The first computers were programmed with punched tapes. People brought the full programs with them (self-contained). These full programs contained parts that were commonly happening on many programs, which turned into the concept of reusable code: subroutines. These repeatable problems could be effectivized independently of the programs. Later, on top of this, we got the operating system which took care of the details of the underlying system and functions as a larger abstraction of reusable code. Time-sharing was invented to solve the issues with many people wanting to share the same very expensive hardware, mainly mainframes. Today, in the cloud, virtual machines are rarely shared with other users, they are commonly single-purpose and rarely have need of things like sound-cards, keyboards etc. In the cloud today, there is less need for multi-user multi-process general-purpose operating systems. Modern general-purpose operating systems have an enormous amount of time and work dedicated to protecting kernel from user-space [8], and vice versa [7]. They also have built-in terminals, users and permissions systems, multiple processes with multiple threads, a large public system call interface and on top of this there is usually a large distribution with its own set of libraries and services. And on top of all of this, there are kernel subsystems that

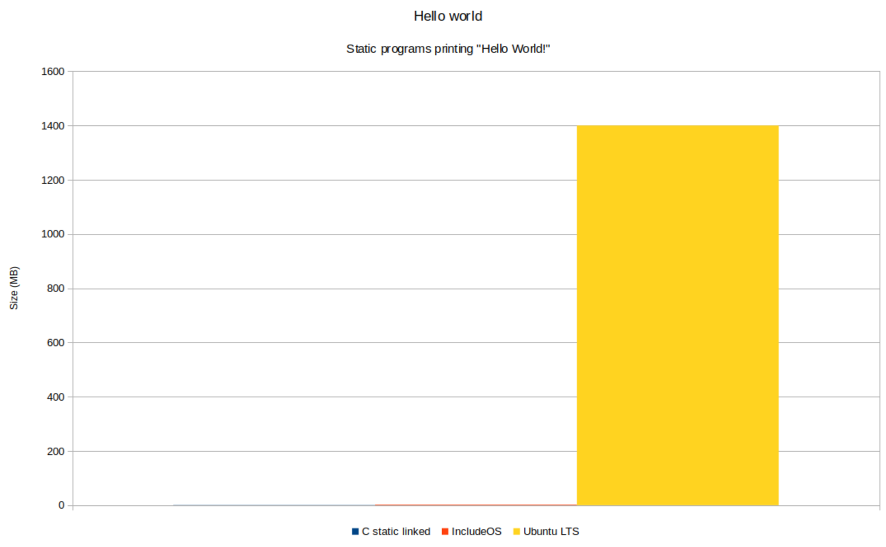


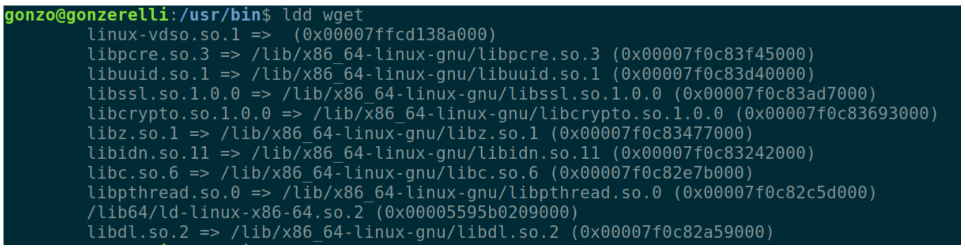
Figure 4.1: Sizes of 'Hello World!' binaries

receive little to no attention [8]. Unikernels solves the problem of the bloated operating system, reducing attack surface and complexity greatly, but comes with its own caveats which will be discussed later.

Unikernels are essentially specialized library operating systems for IoT or cloud. The operating system library is linked together with a single-purpose program, forming a minimal bootable image. Most of the dead/unreferenced code removal is done by the linker, while build-system techniques will cover the remaining issues, such as also including the minimal necessary device drivers and platform extensions. A full fledged unikernel web-service without any baked-in images or other web-content can be 2MB or less. In contrast, Ubuntu homepage states minimum requirements for an Ubuntu server is 5GB disk space. Unikernels don't need writable disks. An ideal web-service is the one that you can restart at will, with low boot time if something goes wrong, that doesn't require any state. Web-servers are ideal for this because they are very low on state. The only necessary state is per-client request timeout, the TCP's *TIME_WAIT* for each connection and optionally SYN queue state that would fend off throttled connection attempts.

A full Linux distribution has hundreds of millions of lines of code, and while most of it remains unused and is only loaded when needed, the kernel itself is 140k LOC and the distribution itself comes with hundreds of pre-installed programs. Unikernels are magnitudes smaller in comparison simply because of how the end result kernel images are assembled. For a comparison see figure 4.1.

Traditional operating systems have a rather fixed layout, with a full set of features, all built into one or more larger images loaded at boot time. Linux, for example, has an external driver image library, from where it loads selected drivers found during boot. Library operating systems can be



```
gonzo@gonzerelli:/usr/bin$ ldd wget
        linux-vdso.so.1 =>  (0x00007ffcd138a000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f0c83f45000)
        libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007f0c83d40000)
        libssl.so.1.0.0 => /lib/x86_64-linux-gnu/libssl.so.1.0.0 (0x00007f0c83ad7000)
        libcrypto.so.1.0.0 => /lib/x86_64-linux-gnu/libcrypto.so.1.0.0 (0x00007f0c83693000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f0c83477000)
        libidn.so.11 => /lib/x86_64-linux-gnu/libidn.so.11 (0x00007f0c83242000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0c82e7b000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f0c82c5d000)
        /lib64/ld-linux-x86-64.so.2 (0x00005595b0209000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f0c82a59000)
```

Figure 4.2: Dependencies of wget, a rather small Linux program

big and contain many features, but when linked into a service as a library it will only bring with it the absolutely minimum amount of features needed to perform the task required by the service. Because of that, unikernels are different in many ways from traditional operating systems. There is no need for a system call ABI or superfluous things like shell/terminal support, both of which makes remote exploit creation easier. Unikernel binary images can be very small, such as the LiveUpdate test binary being 2MiB, and they have just enough code in them to perform their specific task.

Unikernels tend to have extremely low boot times. IncludeOS for example can boot in 125 milliseconds during a boot on bare metal and as an unmodified guest in a vanilla hypervisor. 120ms of that time is spent on calibrating timers by measuring the CPU frequency against a fixed timer. On uKVM this boot time is reduced to around 400 microseconds. See chapter 9 (Boot times) for more information.

Looking at the dependencies of *wget*, a rather common, small downloader on Linux (and other systems) we can see (figure 4.2) that it doesn't use all that many libraries to do its job.

However, should we want to make a *downloader service*, and nothing more, on say Ubuntu, then this program is just one among thousands of programs. We can use *ls /usr/bin | wc -l* to see a lower bound on the number of available default programs (2400 on Ubuntu 17.04). Additionally, all these programs make use of, as referenced in the figure for *wget*, many libraries that also add more unused code to the mix. Altogether, there are hundreds of millions of lines of code present for this hypothetical *downloader service*. This is all on top of a general purpose operating system, which has hundreds of old and new interfaces to do system operations. Additionally, best practices in security on general-purpose operating systems can be complicated and time-consuming.

4.1.2 Specialized OS

Unikernels typically serve a singular purpose (eg. a single application), have a single address space (no separated kernel and user address spaces), and are often single-threaded by default, unless explicitly enabled. Unikernels are always built into a single stand-alone bootable image. The images don't have external dependencies and they also run and behave largely the same on all hypervisors and platforms (given at least minimal

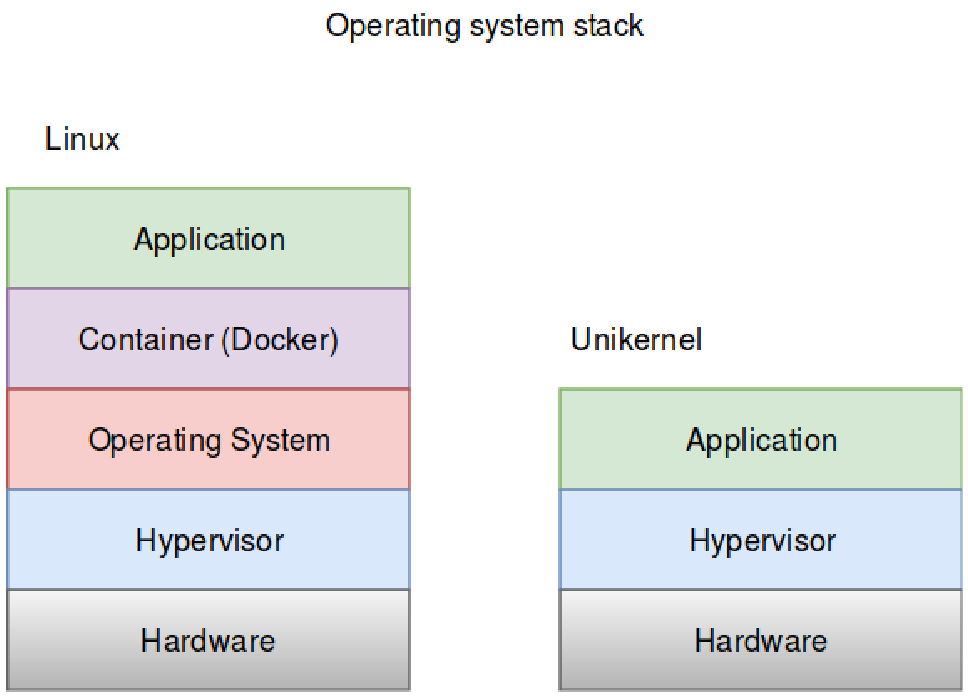


Figure 4.3: Operating system stack

support for that platform).

Binaries contain only what is needed to boot and run services on top of the OS libraries. This reduction in code and data happens both at the linker stage, where it determines what parts of the code was referenced and (vice versa) what parts went unreferenced and can be dropped from the final binary, as well as when selecting drivers and libraries from the build system. See figure 4.3.

Platforms have minor differences that set them apart, such as how to auto-configure the network or which network driver is used. As an example, on Linux (libvirt, qemu, OpenStack) as well as on all platforms that have VirtualBox there is *VirtioNet* for networking. For VMware the network driver is called *vmxnet3*, and requires a driver. Otherwise, the platforms function the same with the differences being mostly in paravirtual feature extensions (that can be checked via CPUID leafs).

For reference; bits and pieces of the current KVM paravirtual interface [18]:

```
flag                               || value || meaning
=============================================================================
KVM_FEATURE_CLOCKSOURCE            ||     0 || kvmclock available at msrs
                                   ||       || 0x11 and 0x12.
------------------------------------------------------------------------------
KVM_FEATURE_NOP_IO_DELAY           ||     1 || not necessary to perform delays
                                   ||       || on PIO operations.
------------------------------------------------------------------------------
KVM_FEATURE_MMU_OP                 ||     2 || deprecated.
------------------------------------------------------------------------------
KVM_FEATURE_CLOCKSOURCE2           ||     3 || kvmclock available at msrs
                                   ||       || 0x4b564d00 and 0x4b564d01
------------------------------------------------------------------------------
KVM_FEATURE_ASYNC_PF               ||     4 || async pf can be enabled by
                                   ||       || writing to msr 0x4b564d02
------------------------------------------------------------------------------
KVM_FEATURE_CLOCKSOURCE_STABLE_BIT ||    24 || host will warn if no guest-side
                                   ||       || per-cpu warps are expected in
                                   ||       || kvmclock.
```

The Qemu-KVM paravirtual interface exposes functionality that increases performance for KVM-aware guests, such as PV-EOI which lets guests skip having to write End-Of-Interrupt to the virtual APIC hardware if a certain bit is set when handling interrupts. The PV-EOI feature alone saw calls to *VMEXIT* cut in half for interrupt-intensive workloads according to RedHat [32].

Unikernels have two differing approaches. There is the clean-slate type and the compatibility-/portability- oriented unikernel.

For the clean-slate type of Unikernel the API is language-modern, often asynchronous just like the hardware, and performant. It will follow the best practices in memory- and type-safety and follow the guidelines of the systems language for the kernel. The same goes, optionally, for a high-level language on top of the kernel. See figure 4.4.

POSIX-conformant (legacy/portability oriented) unikernels implement the POSIX standard system calls as well as many Linux system calls so that applications that would ordinarily run on Linux can be run with minor or no changes on the unikernel. See figure 4.5. POSIX-conformant unikernels do not need to use BSD drivers as shown in the figure. It is there to show that it's possible to make a unikernel that does not reinvent the wheel, so to speak. *Rumprun* is such a unikernel, which implements all the POSIX system calls, and supports BSD drivers using a BSD shim layer [27].

4.1.3 Single address space

Unikernels are implemented as a single process. There is no scheduling between different processes with different users, permissions and environ-

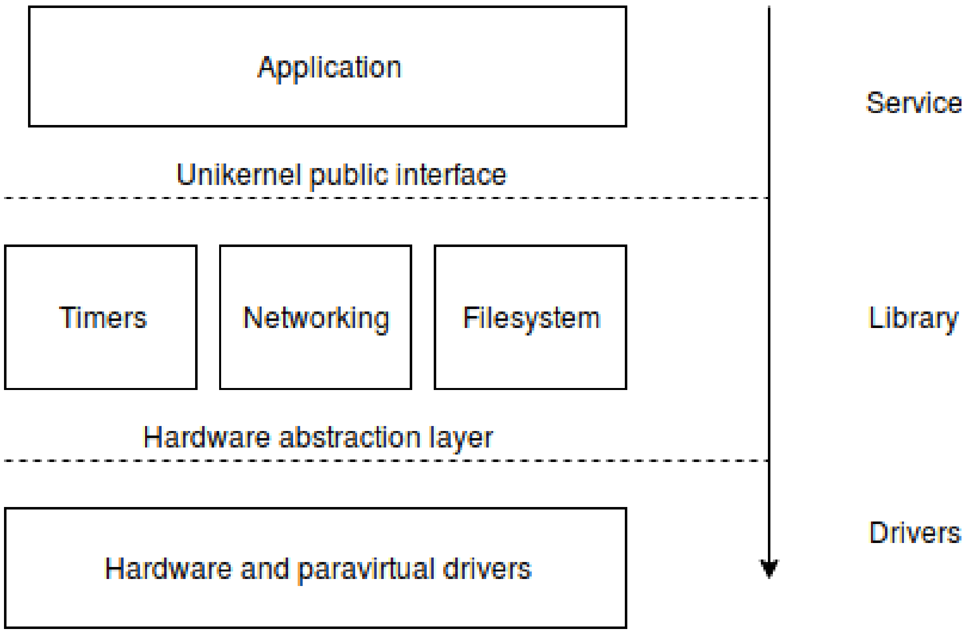

Application
Service
Unikernel public interface
Timers
Networking
Filesystem
Library
Hardware abstraction layer
Drivers
Hardware and paravirtual drivers

Figure 4.4: Clean slate Unikernel overview

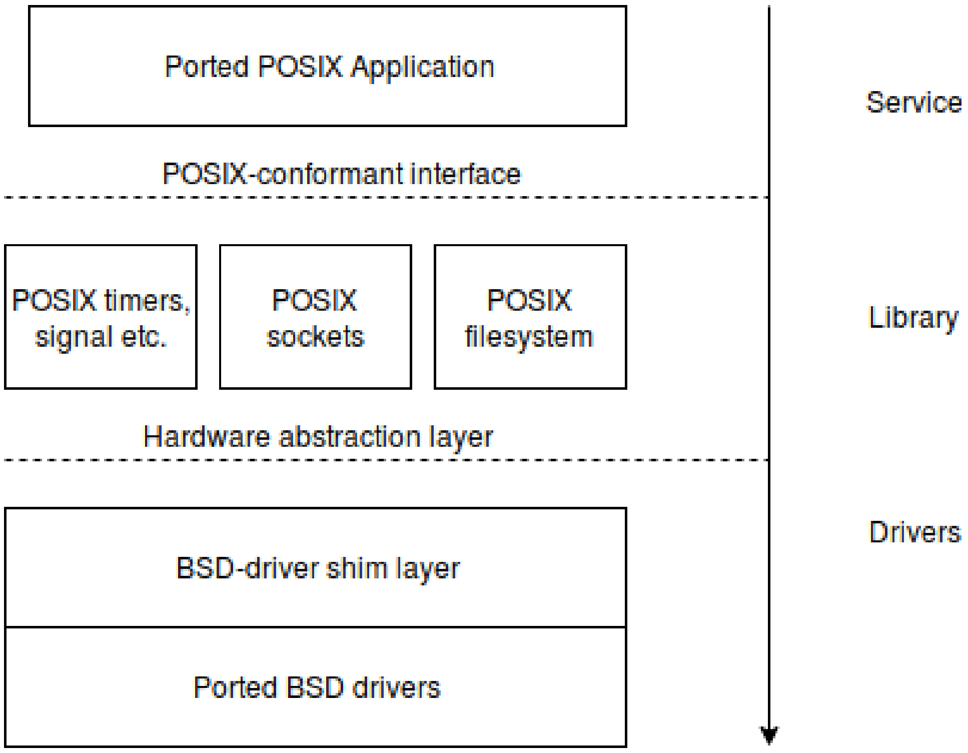


Figure 4.5: POSIX-conformant Unikernel overview

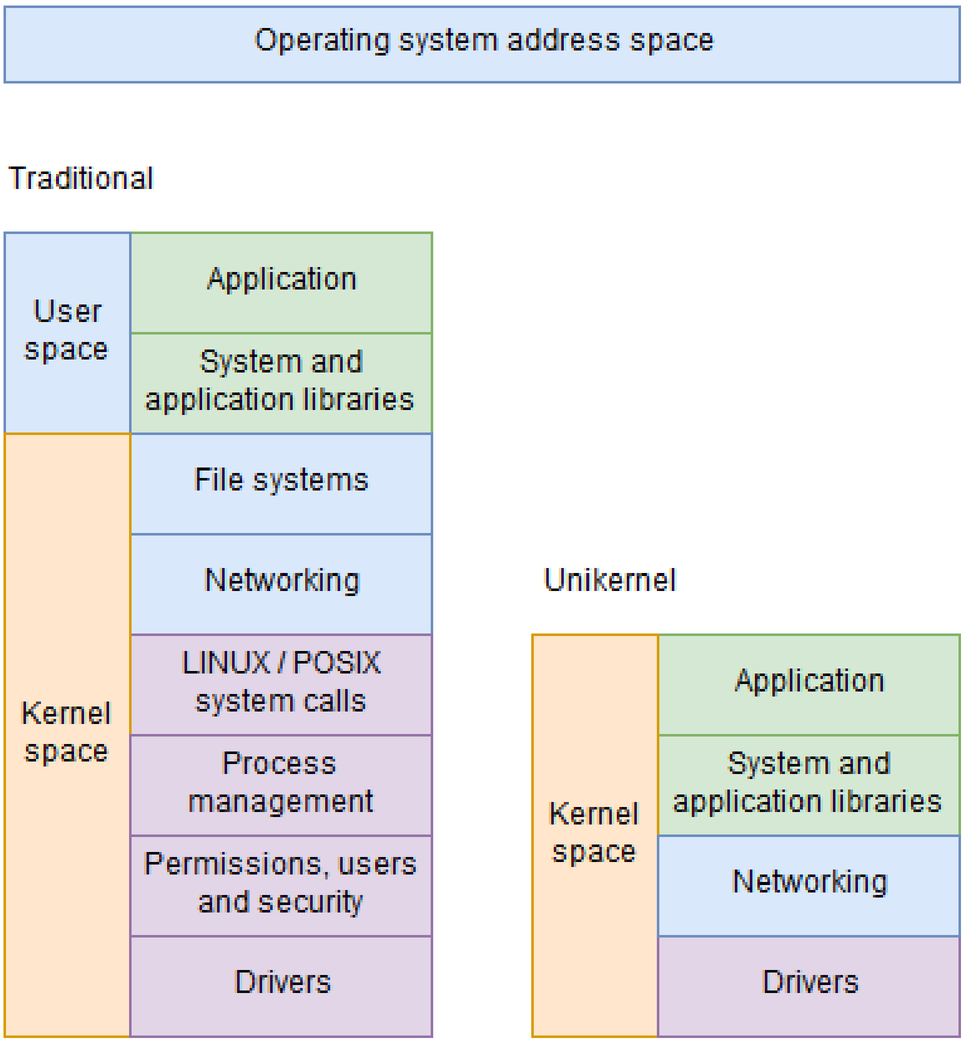

Operating system address space
Traditional
User space
Application
System and application libraries
Kernel space
File systems
Networking
LINUX / POSIX system calls
Process management
Permissions, users and security
Drivers
Unikernel
Kernel space
Application
System and application libraries
Networking
Drivers

Figure 4.6: Address spaces in Operating Systems

ments. There is also no context switching between kernel- and user-mode. The classical kernel-userspace MMU-enforced isolation is unnecessary in unikernels because there is no kernel address space to protect. Memory-safe and type-safe programming techniques from a modern programming language replaces the isolation between user's and kernel's address space [24]. See figure 4.6.

There are performance benefits to avoiding context switches, especially inside virtual machines where each privileged instruction traps into the hypervisor. Context switches can take several microseconds to complete, and scheduling out processes will increase cache pollution which is going to be costly [34][17]. Each system call on a general-purpose operating system requires two context switches (to and from kernel space). Double that for virtual machine guests. The kernel bypass project shows multiple projects that avoid the kernel round-trip to boost specialized workloads [23].

It's possible to use *Link-time optimization* (LTO) to optimize the whole unikernel as a single unit, including the OS libraries. This allows compilers to optimize the code more.

Services can be built for immutable infrastructure (eg. contains no file system drivers with write functionality). Since the build chains for unikernels always build one image with everything needed, it's also possible to append a read-only file-system directly into the image itself. On IncludeOS this feature is called simply *memdisk*, and it's accessed just like any other filesystem.

One of unikernels' drawbacks are that they can't expose an arbitrary programming language to the user, such as on a Linux distribution. Users are limited to whatever language the unikernel exposes on top of its kernel. It's common to write the core kernel in C/C++ (and other systems languages) and then build an interface on top using a high-level language, such as JavaScript. Some unikernels use the same language for both kernel and service, such as *MirageOS* which uses OCaml for everything.

4.1.4 Security

The attack surface is greatly reduced due to the image only containing what is used. Unikernels are also several magnitudes smaller than general-purpose operating systems with programs running on top.

There is no default shell or terminal server on a unikernel. Why would there be? If you wanted to inquire about the systems status, throw together a status page and host it with a read-only web-server using the unikernels API.

When running unikernels inside virtual machines, the strong isolation provided by the hypervisor is an extra layer of security. It's also possible to run unikernels as ring-3 unprivileged kernels, given that both the hypervisor and guest OS supports this.

Memory can be randomized and made read-only even without unprivileged kernels making it hard to reliably create exploits. Maybe a state actor can get into one particular unikernel given a specific image, but if the next image generated from the very same source is completely different due to randomization, then how can exploits be reliable even when created for a specific commit point of the unikernel?

It has been on the authors mind for some time now to have a linker that didn't do deterministic building. OpenBSD has recently gained the *Kernel relinking* feature, where on each boot it is relinked with all the object files in random order and with random offsets [31]. This will also be applied to unikernels in the near future, either when building the image itself or on each boot. The linking process might be too time-consuming to do on each boot.

Link-time optimization (LTO), sometimes called *Whole program optimalization*, can inline more code, remove duplicate and dead code quite aggressively both reducing attack surface and also complicating the program flow.

4.1.5 Maturity

Most, if not all, unikernels are works in progress even at this point (2017) and should not be used in production environments, bar very narrow use-cases such as for simple routing or load balancing. While they offer potentially better security than traditional operating systems, it is still the case that a fully updated Linux LTS is going to be more reliable security-wise. Linux also gets fast responses to open vulnerabilities, while Unikernels are more likely to just get updated on a normal work-week development schedule. Unikernel projects also have varying degrees of testing and code coverage, while Linux, which may seem to lack at least basic code coverage and automatic testing, is still one of (if not) the most tested and hardened operating system(s) out there.

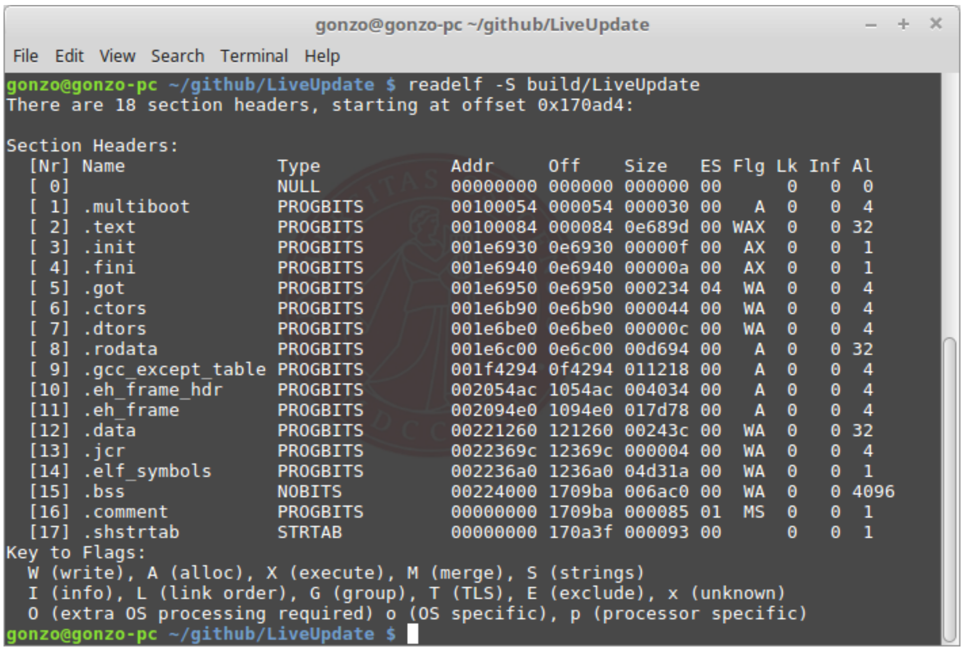
4.2 Immutable infrastructure

Immutable infrastructure is about not making changes to parts in a system once they are deployed. Instead, if changes are needed, one should simply deploy a new part with the changes.

Unikernels are almost naturally immutable, even at the address-space level with immutable address space layout support on specialized hypervisors. With a read-only filesystem, a single-purpose kernel and locked down memory the unikernel could be said to be immutable, as the configuration will not change. It is natural to deploy a new unikernel once some configuration changes, even without a read-only filesystem and immutable address space layout.

4.3 Hypervisors for unikernels - Unikernel Monitors

There is an ongoing effort to create hypervisors that present minimal interfaces to guest kernels, such as Bareflank, uKVM and others. These hypervisors will help minimize the potential attack surface of computer


gonzo@gonzo-pc ~/github/LiveUpdate
File Edit View Search Terminal Help

```
gonzo@gonzo-pc ~/github/LiveUpdate $ readelf -S build/LiveUpdate
There are 18 section headers, starting at offset 0x170ad4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .multiboot        PROGBITS        00100054 000054 000030 00   A  0   0  4
  [ 2] .text             PROGBITS        00100084 000084 0e689d 00 WAX  0   0 32
  [ 3] .init             PROGBITS        001e6930 0e6930 00000f 00  AX  0   0  1
  [ 4] .fini             PROGBITS        001e6940 0e6940 00000a 00  AX  0   0  1
  [ 5] .got              PROGBITS        001e6950 0e6950 000234 04  WA  0   0  4
  [ 6] .ctors            PROGBITS        001e6b90 0e6b90 000044 00  WA  0   0  4
  [ 7] .dtors            PROGBITS        001e6be0 0e6be0 00000c 00  WA  0   0  4
  [ 8] .rodata           PROGBITS        001e6c00 0e6c00 00d694 00   A  0   0 32
  [ 9] .gcc_except_table PROGBITS        001f4294 0f4294 011218 00   A  0   0  4
  [10] .eh_frame_hdr     PROGBITS        002054ac 1054ac 004034 00   A  0   0  4
  [11] .eh_frame         PROGBITS        002094e0 1094e0 017d78 00   A  0   0  4
  [12] .data             PROGBITS        00221260 121260 00243c 00  WA  0   0 32
  [13] .jcr              PROGBITS        0022369c 12369c 000004 00  WA  0   0  4
  [14] .elf_symbols      PROGBITS        002236a0 1236a0 04d31a 00  WA  0   0  1
  [15] .bss              NOBITS          00224000 1709ba 006ac0 00  WA  0   0 4096
  [16] .comment          PROGBITS        00000000 1709ba 000085 01  MS  0   0  1
  [17] .shstrtab         STRTAB          00000000 170a3f 000093 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
gonzo@gonzo-pc ~/github/LiveUpdate $
```

Figure 4.7: ELF Segment layout of a LiveUpdate binary

networks in the future, enables special features to reduce common security problems and enable fast booting of unikernel guests as low as in sub milliseconds [38].

Specialized hypervisors can also assist in other ways, such as enabling immutable address space layout [22].

4.3.1 uKVM

An IBM research team is currently working on a specialized hypervisor called uKVM. This hypervisor is called a unikernel monitor [38]. The monitor is intended to be specially made for a certain unikernel, although currently there is also an IncludeOS port to uKVM. uKVM enables magnitudes faster boot time compared to ordinary PC emulators (traditional hypervisors).

uKVM loads its guests according to the ELF binary section layout (see figure 4.7), and disables execution privilege on the areas of memory known to be non-executable. While this is not perfect, it is a major step in the right direction. With some extra communication between hypervisor and guest it should be possible to further lock down the guest memory address space and make it even more immutable.

All segments without the W-flag are read-only. It makes up a significant portion of the service binaries (see figure 4.7), including any memory-based read-only filesystems such as *memdisk* (not shown on figure).

4.4 Unprivileged kernels

Unprivileged kernels do not yet exist in the wild, nevertheless they will fill an important niche.

In high security scenarios kernel privileges are not desirable in guest operating systems. Here, the kernel will get loaded in by the hypervisor or some system service, for example, as a regular ELF binary. Each loadable section is loaded through an ELF loader on the hypervisor and placed where it should go in memory, according to the layout. Each read-only section is marked read-only by the hypervisor and can never be written to again by the unprivileged guest. The hypervisor also takes care of setting up paging, since it requires ring0 privileges. Once this is done, the hypervisor can pass off execution to the unprivileged guest kernel. This reduces attack surface and increases security.

Specialized type-1 hypervisors which do not emulate normal bare metal are required. The guest operating system will also have access to an interface in which it can further lock down its own access to memory. In that case it is not possible for the guest to live update its own environment, because the VM simply cannot replace itself. Instead, to do this, it is necessary to have help from the local VMM (or a remote VMM to live migrate to) via hypercalls or other mechanisms for communicating with the hypervisor. The hypervisor should go through a procedure for verifying the authenticity and consistency of the update binary, for example by checking the signing and performing a redundancy check.

4.5 IncludeOS

IncludeOS is a unikernel originally started by Alfred Bratterud at HÃ¸gskolen i Oslo- og Akershus as a PhD project to research aspects of unikernels [3]. Alfred was experimenting with thousands of small mini VM guests to see how many he could host, how to control them efficiently and where the bottlenecks were. This work was done with modern C++ with which core guidelines has the *zero-overhead-principle* and *zero-cost abstractions* [4]. From there he would go on to start building IncludeOS.

4.5.1 Early versions

Early versions of IncludeOS were 32-bit only and used Electronic Arts free *EASTL* C++ standard library. It did not have C++ exception support, and while EASTL was performant it was also very limited compared to the whole library we can expect on the major platforms. Despite this, it had a basic IPv4 network stack with UDP and very limited TCP support. Images were typically 600KiB at the time. Very small for being a bootable standalone operating system with a networking stack. At that time IncludeOS could only be run as a VM guest in Qemu.

4.5.2 Current day - a modern unikernel

Today IncludeOS is a full-fledged library operating system with full C/C++ support using the LLVM C++ standard library. This has increased the binary image sizes, which is now at 2MiB. The operating system also has a full IPv4 stack with modern TCP, a read-only file system with exFAT support, support for multiple cloud platforms, support for symmetric multiprocessing (SMP) using x86 virtual APIC cores, modern x86 hardware support, support for many para-virtual drivers and para-virtual CPU features, x86 64-bit (amd64) support, which is now the default, several ported networking drivers and a large test suite. On top of this it has a modern HTTP framework with WebSockets and support for TLS streams (such as encrypted WebSockets, or HTTPS). It also recently gained support for running as a guest inside the uKVM unikernel monitor, thanks to researchers at IBM.

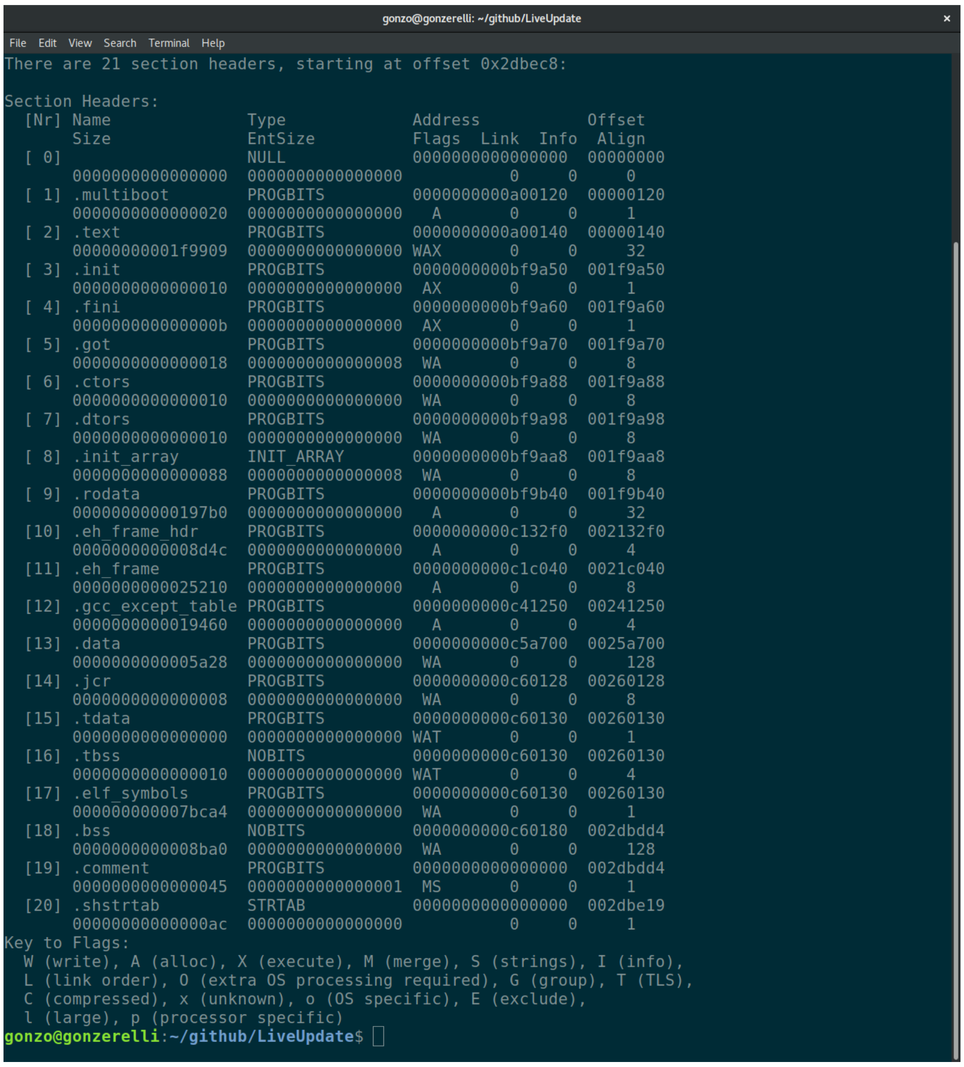
4.5.3 Anatomy of an IncludeOS binary

An IncludeOS binary is first and foremost an ELF binary (see figure 4.8), with a bootloader attached at the front when needed. On 64-bit there is a 32-bit chainloader that hotswaps in the 64-bit kernel, due to Qemu's multiboot support being limited to 32-bit. The 64-bit IncludeOS binary does not consist of 2 ELF binaries combined, rather, the 32-bit chainloader is a separate kernel and the 64-bit ELF binary is chainloaded from the 32-bit kernel as if it was a Linux kernel module. This increases the 64-bit boot time slightly.

The (legacy) bootloader is only required when booting on hypervisors without multiboot support or when booting on bare metal.

The IncludeOS memory layout is largely standard. The various sections are familiar. *.multiboot* is the 32-byte multiboot header as specified by the *GNU Multiboot Specification* [25]. *.text* is the executable code, loaded as-is. *.init* and *.fini* as well as *.got*, *.ctors*, *.dtors*, *.init_array* and *.fini_array* are all C and C++ constructors and destructor functions. *.rodata* is the read-only data section. *.eh_frame_hdr* and *.eh_frame* as well as *.gcc_except_table* is for C++ exception support. *.data* is writable data, such as a persistent (static) variable that isn't constant. *.tdata* and *.tbss* are for Thread-Local-Storage which is storage used by threads. The storage is instantiated per-thread and so each thread has its own copy of all the thread local storage.

One unique section of the OS binary is the *.elf_symbols* section, which contains pruned ELF symbols. These symbols are filtered down to just C and C++ functions by a special program and then inserted into the ELF binary after linking. They are then used to enable features such as backtrace or live stack sampling.


gonzo@gonzerelli: ~/github/LiveUpdate
File Edit View Search Terminal Help

```
There are 21 section headers, starting at offset 0x2dbec8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .multiboot        PROGBITS         0000000000a00120  00000120
       0000000000000020  0000000000000000   A       0     0     1
  [ 2] .text             PROGBITS         0000000000a00140  00000140
       00000000001f9909  0000000000000000 WAX       0     0     32
  [ 3] .init             PROGBITS         0000000000bf9a50  001f9a50
       0000000000000010  0000000000000000  AX       0     0     1
  [ 4] .fini             PROGBITS         0000000000bf9a60  001f9a60
       000000000000000b  0000000000000000  AX       0     0     1
  [ 5] .got              PROGBITS         0000000000bf9a70  001f9a70
       0000000000000018  0000000000000008  WA       0     0     8
  [ 6] .ctors            PROGBITS         0000000000bf9a88  001f9a88
       0000000000000010  0000000000000000  WA       0     0     8
  [ 7] .dtors            PROGBITS         0000000000bf9a98  001f9a98
       0000000000000010  0000000000000000  WA       0     0     8
  [ 8] .init_array       INIT_ARRAY       0000000000bf9aa8  001f9aa8
       0000000000000088  0000000000000008  WA       0     0     8
  [ 9] .rodata           PROGBITS         0000000000bf9b40  001f9b40
       00000000000197b0  0000000000000000   A       0     0     32
  [10] .eh_frame_hdr     PROGBITS         0000000000c132f0  002132f0
       0000000000008d4c  0000000000000000   A       0     0     4
  [11] .eh_frame         PROGBITS         0000000000c1c040  0021c040
       0000000000025210  0000000000000000   A       0     0     8
  [12] .gcc_except_table PROGBITS         0000000000c41250  00241250
       0000000000019460  0000000000000000   A       0     0     4
  [13] .data             PROGBITS         0000000000c5a700  0025a700
       0000000000005a28  0000000000000000  WA       0     0     128
  [14] .jcr              PROGBITS         0000000000c60128  00260128
       0000000000000008  0000000000000000  WA       0     0     8
  [15] .tdata            PROGBITS         0000000000c60130  00260130
       0000000000000000  0000000000000000 WAT       0     0     1
  [16] .tbss             NOBITS           0000000000c60130  00260130
       0000000000000010  0000000000000000 WAT       0     0     4
  [17] .elf_symbols      PROGBITS         0000000000c60130  00260130
       000000000007bca4  0000000000000000  WA       0     0     1
  [18] .bss              NOBITS           0000000000c60180  002dbdd4
       0000000000008ba0  0000000000000000  WA       0     0     128
  [19] .comment          PROGBITS         0000000000000000  002dbdd4
       0000000000000045  0000000000000001  MS       0     0     1
  [20] .shstrtab         STRTAB           0000000000000000  002dbe19
       00000000000000ac  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
gonzo@gonzerelli:~/github/LiveUpdate$
```

Figure 4.8: ELF sections of the LiveUpdate test service binary

Chapter 5

Dynamic software updating

5.1 Software updates

Every computer is connected to the Internet today, exposing it to possible remote attacks. Almost every device today is running on some sort of software that manages the underlying hardware. If the software has bugs that has gotten fixed or general changes are made to the code, the device will have to be updated to apply these changes.

Mobile phones, desktops, servers. All these are exposed to the Internet in some way, and have to have regular updates that fix security holes. The more secure our devices are, the less attacks globally because of herd-immunity, as attacks are largely executed from infected computers and devices.

Desktop computers also have to update. People generally tend to avoid updates just because they are so disruptive. Traditional software update is seen as a necessary evil for some, and for others it is simply ignored. For Windows 10, Microsoft has changed how they apply updates: After enough time has passed, users are forced to update. Their machines are restarted, and the lengthy update process begins and cannot be stopped until it has fully completed. It's not a popular change, but it reflects the reality that people weren't updating their machines regularly. With live updates, maybe users wouldn't have to know as long as the permission was explicitly given.

5.2 Dynamic Software Updating - DSU

Dynamic Software Updating (DSU) refers to live updating running programs using various methods while retaining at least part of the programs state, and also covers aspects of update verification, how to do live rollback to known good state and so on. In other words, it can be thought of as live patching of running programs or systems. DSU is a relatively small field of research with a very tiny presence in actual deployed software. The author does know about active DSUs for operating systems, but they are not full DSUs, as we will see later.

With live updating, it's possible to retain program state and even TCP connections across updates. This can reduce service disruption enough to make live updating worth the additional effort even when it's not strictly needed. It will, though, always have risks associated with how updates are built and applied. These situations can be solved with rollbacks or if that fails, failovers.

The most common usage for live updating is to correct serious bugs or disable broken features without disrupting a running service. Programs today that actually use live updating is typically high availability services that should not be disrupted. Still, most high availability applications don't use live updating and is instead typically phased out during an update. The service is then resumed on another updated machine almost instantly. Open connections are kept on the old system until none remain, at which point the old system is fully taken out of service. This method is safe, but also cumbersome, and requires a separate system that has updates applied. There are also many scenarios in which the phasing out of a system can't easily work, such as when updating very expensive or unique machinery.

While live updating is not much used, most programs and services can benefit from it. Most desktop operating systems today have to be restarted at least weekly or biweekly to apply important security updates. This would not have been necessary with live updating. The same can be said for any program or system which benefits from not having to start over. When the update happens, and how it is applied varies between existing implementations.

As an example; during development the author developed an IRC server which was live updated during the development process. The IRC server was hosted on an OpenStack cloud, where it takes a relatively long time (minutes) to upload and deploy new images each time something is changed in the program. The author continually made improvements to the program and updated it live, saving a great deal of time mostly as an experiment.

> DSU is appealing because of its generality: in principle any program can be updated in a fine-grained way. There is no need for redundant hardware or special-purpose software architectures, and application state is naturally preserved between updated versions, so that current processing is not compromised or interrupted. DSU can also be used naturally to support dynamic profiling, debugging, and "fix-and-continue" software development [26].

5.2.1 DSU approaches

Today iOS and Android programs are persisted when closed on our mobile devices. Once the programs are reactivated they must restore their state in the same manner a *state transfer-based* DSU performs upgrades [9]. LiveUpdate is a state transfer-based DSU.

> State transfer updates work by launching a new process running the updated program version and transferring program state from the running process to the updated version [9].

Other DSUs often provide *in-place updating*, such as Linux live patching [14]. In-place updating is essentially writing a dynamic patch, load patch into system/program and finally, redirect old functionality to the new functionality from the new patch.

In-place updating brings with it a few problems, such as inhibiting compiler-optimizations, stack reconstruction issues (when to update) and

> reasoning about the behavior of the updated program places additional cognitive burden on the programmer [9].

In-place updating also tends to add runtime overhead, while for state transfer-based updating overhead is only in the update process itself. The cost of transferring lots of state can be high, as shown in experiments later on.

5.3 Existing DSUs for operating systems

Unikernels are fundamentally different from general-purpose operating systems, and comparing existing DSUs to a DSU built for unikernels is like comparing apples to oranges. Additionally, there are also very few DSUs targeting operating systems in use. They also don't fulfill all the normal requirements for dynamic software updating as we will see.

5.3.1 Dynamic Kernel Patching - kpatch

kpatch is a limited purpose Linux-specific DSU started at Red Hat [14][13].

Uses noop at the beginning of kernel functions traditionally used for kernel tracing features to redirect kernel functions to new patched functions installed via kpatch as small kernel objects/modules. This works because kernel modules are relocatable.

For Linux, *kpatch* meets its needs. Linux images are static fixed things, and the division between kernel and userland means that there has to be a well defined and public kernel system call interface. You could even go as far as storing the known locations of kernel functions for a given version of Linux, and no matter which distribution it was run underneath, the functions would continue to have known locations presenting a security risk. On the other hand, knowing the addresses is what makes *kpatch* work, as the entry points are used when *kpatch* writes long jump instructions to redirect kernel functions to new patched versions. Linux also typically has a huge UNIX-like ecosystem on top of it, and also normal operating system features like loading and executing ELF programs.

This feature was merged into Linux for version 4.0, and is currently available to all. It is not a true DSO, because you cannot update any part of the kernel, however it really does not require reboot and patching is near instant.

It’s always been a struggle to get users, businesses and data-centers to prioritize security over ease-of-use. With live kernel patching Linux system administrators can now just login remotely, live-patch the kernel and not have to reboot. It is clearly a superior and much wanted solution to the problem of having to reboot and restart everything, which could in many cases be a very costly and dangerous disruption to servicing users. It is, unfortunately, only capable of live patching a single function at a time.

5.3.2 Directly boot into a new kernel - kexec

kexec is not a DSU, but it does something that is fairly close to a step in the process of what LiveUpdate does. *kexec* loads a new Linux kernel and then allows you to boot directly into it, skipping some initialization stages, which reduces the boot time [2].

From the *kexec* manual:

> kexec is a system call that enables you to load and boot into another kernel from the currently running kernel. kexec performs the function of the boot loader from within the kernel. The primary difference between a standard system boot and a kexec boot is that the hardware initialization normally performed by the BIOS or firmware (depending on architecture) is not performed during a kexec boot. This has the effect of reducing the time required for a reboot [37].

5.3.3 Live updating Linux

There has been lots of talk about how to make Linux a full-fledged DSU, and (surprisingly) the accepted method is much like what LiveUpdate does. Save state to memory, boot into new kernel, restore state. Linux, however, has millions upon millions of lines of code, all systems and subsystems of which there is no concept of serializing and deserializing state. It was estimated that it may takes 10 years or more to fully integrate full-fledged DSU support into Linux [5].

5.3.4 AutoPod

> Potter et al. [30] perform operating system upgrades by migrating running applications to an updated operating system instance. They facilitate process migration by running each application within pods - isolated environments that provide a virtual machine abstraction. These can be viewed as state transfer updates where the state of the updated operating system is its running applications [9].

Chapter 6

LiveUpdate

6.1 Origin

LiveUpdate as a dynamic software updating system was intended as a proof-of-concept for library operating systems and Unikernels. It is the only modern operating system DSU out there (2017) that can fully live update operating systems, and is also in use, that the author is aware of. The reference implementation is for the IncludeOS unikernel. It takes a running IncludeOS service, stores selected state for later and then performs a kernel hotswap. Assuming the service writer stored all the state he needed after the update to resume the system properly, it will resume running as if nothing happened after the update. The goal is to have various methods of safely updating small systems and services without service interruption.

LiveUpdate is a *state-transfer based* DSU, meaning it stores only the state, the minimal amount of data, that service writers deem necessary to resume service operation seamlessly after an update. During the live update the operating system is fully replaced with an updated operating system. LiveUpdate is written in modern C++11, which is the same language used for the IncludeOS unikernel. This work is not a general solution to updating general-purpose operating systems, however it does in theory apply to all operating systems that are derivatives of single-address-space library operating systems.

Operating systems that run on VM/JIT-based programming languages, such as Java, JavaScript or .NET/Mono, may not have a need for a state-transfer based DSU as they can employ methods that work directly on the interpreted bytecode and modifies programs in real-time with much more ease [1]. Unfortunately, VM-based operating systems cannot have bytecode all the way down to the hardware interfaces, and such will also need other methods of updating the lower level assembly and systems languages. Obscure operating systems that run on VM-based programming languages are typically single-user, single-purpose and could even be single-address-space. The more general-purpose the operating system is, the harder it is to implement dynamic software updating for it [5].

Due to the way the updates are applied, it is possible to update any service to any other service. It is therefore possible to, for example, take a service that does nothing but receive OTA live updates and transform it to another service that no longer has the capability to receive updates, but instead has new functionality required to host a website. This process happens seamlessly in but a few milliseconds. As a consequence this may open up new possibilities for cloud providers due to how fast they would be able to deploy new services as needed. Public clouds today are known to be very latency heavy, often spending minutes to just take down one VM and replace it with another. This functionality would be unique for cloud environments, with the exception of metered lambdas (functions as a service).

6.1.1 Upsides

LiveUpdate adds no run-time overhead to systems during normal operation. There is memory overhead during updates, dependent on how much state systems need to persist across the live updates. Also, the more state the user needs to store the more time the update takes.

LiveUpdate can update one arbitrary system into another arbitrary system. There is no need to know the location or names of functions. It is theoretically possible to live update from an IncludeOS service into a Linux distribution. The running Linux instance would not, however, understand that there was state preserved and it would just be overwritten.

LiveUpdate does not rely on systemcalls or hypercalls on privileged kernels, and as such can work on an operating system with no system ABI, which is considered a security feature. Known locations in memory, or known steps to perform system calls makes it easier to construct attacks.

LiveUpdate may work on unprivileged kernels, supporting unikernel monitors and other specialized hypervisors, that are running guests in user-mode. A close but not quite unprivileged scenario is shown later on when LiveUpdate is being run inside uKVM.

It's possible to do continual development (*fix-and-continue*) onto a live running program, even going as far as inserting code that gives feedback on the programs state live, something that usually requires a full redeployment [26][12]. It's also possible to live update just to add or supplement debugging information, or to perform performance metrics.

Relatively low (milliseconds) live update process start to completion time. Public clouds, especially, are known to have very long queues on taking down and then booting up images each time it is updated. On modern hypervisors the updates are near instant, with sub-millisecond times.

LiveUpdate makes it possible to write high availability services, such as a high availability proxy, where systems have real-time and non-stop requirements (no service disruption allowed). High availability services normally use rolling upgrades.

Live updating in general makes it possible to randomize or re-seed an image with randomness on a scheduled interval, to make it nigh impossible to gain any knowledge of locations of internal structures. Unfortunately, this is quite hard to do presently due to a lack of functionality in current linkers, but there are known efforts presently to create linker plugins for LLVM that adds layout randomizing features. It is non-trivial to write exploits that has to always work with custom binary code that changes often. It should be noted that for kernels running in privileged modes it is possible to create exploit kits that implement, for example, entire x86 hardware and drivers. Basically a new kernel, and then take over the system that way. For virtual machines there isn't much to do inside a sandbox, though. When running on hardware (such as IoT-devices) it can be a problem.

Allows for new novel ideas in cloud hosting, such as creating a *base service* that starts primitive images only capable of receiving/downloading another image which it then replaces itself with live. These future updates then provide the actual functionality desired, such as a web service. This method is already being pursued by IncludeOS, see the *Mothership* section, and has shown great potential.

Live updating reduces problems like deteriorating system performance and memory fragmentation over time. For example, the new heap layout after an update will very typically be less fragmented and smaller than before the update. It is similar to the benefits of rebooting systems after weeks or months of uptime.

6.1.2 Downsides

LiveUpdate requires a system to set aside a portion of (physical) memory to be used when live updating. This is not so much an issue on virtual machines, as hypervisors will always have paging enabled. When paging is enabled we can have holes in memory not in use, such as the free memory between the live update storage area and the system heap. This free memory will not be assigned pages. Still, additional physical memory must be set aside for each system that enables live updates, even when its only used during the update.

Services are updated when they themselves initiate a live update sequence. At that point the program should not be doing anything else. Non-maskable interrupts could interrupt the live update process, such as CPU exceptions. Infinite loops and such prevent systems from responding at all in non-preemptive systems. Unikernel monitors will typically do the hotswapping portion on the hypervisor side, and the monitor could provide a watchdog timer for the whole update process.

Having to serialize what you want restored after the update increases the complexity of the service. There needs to be a serialization step as well as a deserialization step. This can easily introduce bugs and inconsistencies between updates.

It's possible to introduce bugs and weaknesses into a running programs code, for example by updating it with a program that fixes one bug but introduces another.

Unfortunately, it is not possible to store the type of the data when serializing state, because differences in compilers store this type-data differently, even across different versions of the same compiler. Due to this, the strongest type-safety would have be compiler-assisted or by processing the source code, akin to what Googles *Protocol Buffers* does.

Live updating carries with it the expectation that the updated program will continue to function as normal after the update. If this is not the case, doing rolling upgrades would have been better to begin with, as it's easier to verify that the system about to be switched in is functioning.

It can be quite complicated to show or prove that there are no side-effects for a given time when the update process could be performed. Unikernels do not usually enable multiprocessing, and don't share hardware and other kernel features with other programs, simplifying things somewhat. Not to mention proofs are complicated to begin with. There are DSUs out there that employ novel methods that aid in verifying the consistency and timing of updates, and even though they only work for userspace programs the theory behind their methods still apply.

If LiveUpdate is set to perform a rollback during a crash, and the rollback causes a CPU exception or kernel panic, it would cause the rollback to fail and the system to hard-reset. Once the system hard-resets all state is lost and the host-side hypervisor (for cloud) or the IoT-device will begin loading the image that was originally booted from disk (as it is a normal boot).

The update process requires an indeterminate amount of time to complete, depending on available CPU time, hardware, OS version and build parameters of the new image and so on. A reasonable estimate can be given based on testing the image, but estimates might not be sufficient on certain real-time systems. Any number of hardware events or (in the case of virtualization) hypervisor or host-system work can delay the live update process. It has been observed that multitasking on Linux affects when the hypervisor gets scheduled in, resulting in extra milliseconds added to the live update completion time.

Rolling back to a previous version of serialized state when the stored state is a newer version that failed can make rollback impossible, or at the very least causes loss of state.

LiveUpdate adds complexity (attack surface) to the operating system. It also usually means there is a public method of transporting updates to the system (eg. OTA updates), which can be used as a way in. Images used in updates should also be cryptographically signed so that even if the distribution network or DNS servers are taken over, the services won't accept forged update images.

6.2 Update process

The actual live updating process is the most critical portion of Live Update, as if things go wrong there is a chance the system has to attempt a rollback, which comes with its own risks.

The IncludeOS live update procedure can be summarized as:

- Retrieve or load new updated binary image which will replace the currently running program
- Validate that the new binary is an ELF binary, has the correct length and that it has an entry point to jump to
- Store program data serialized in a fixed memory area determined by user (storage area)
- Validate that the update storage area did not run out of physical memory or cross heap area
- Overwrite old service with new image and jump to it by executing a platform-specific *hotswap function*
- New kernel discovers it has soft-reset information, validates it, and skips certain initialization procedures to boot faster
- Validate that the update storage area remains untouched
- Restore things from storage area, such as established TCP connections
- Verify that heap did not enter storage area during resume phase
- Zero out the old storage area for security reasons

During the live update process the service will store some of its state in a fixed memory location, which will be deserialized again when the new service has completed the soft-resetting process. The stored state uses CRC32 as a self-consistency check, as well as a canary at the end of the area to verify the memory is still writable (not outside physical RAM). See figure 6.1. It consists of typed data provided by the user, serialized by LiveUpdate and appended to the storage area. Each entry has a header and a data portion.

There is a free memory hole in between the regular service and the live update storage area. One could think that it is wasteful to have to dedicate a large unused memory section just to separate these ranges, however due to how paging works the unused memory is not actually used memory, as it will never be referenced, and so it will just be a chunk of copy-on-write pages filled with zeros. This applies even when the guest kernel does not use paging, as it is only required that the host system or hypervisor has paging enabled.

The recommended memory layout on 64-bit x86 for unikernels is to give each VM over 3GB memory, and then place the LiveUpdate storage area after the 4GB device area. Since 0xC0000000 to 0xFFFFFFFF is used for devices, MMIO and other hardware things, it is not possible for the heap (in a flat memory configuration) to enter into the LiveUpdate area. In more complex setups with paging, it could be done much simpler by just dedicating N amount of pages to the LiveUpdate area permanently. This could be done, for example, through *mmap*. The effective usage when measured on the host system (usually Linux when doing these tests) will be showing only a few megabytes of memory is actually in use.

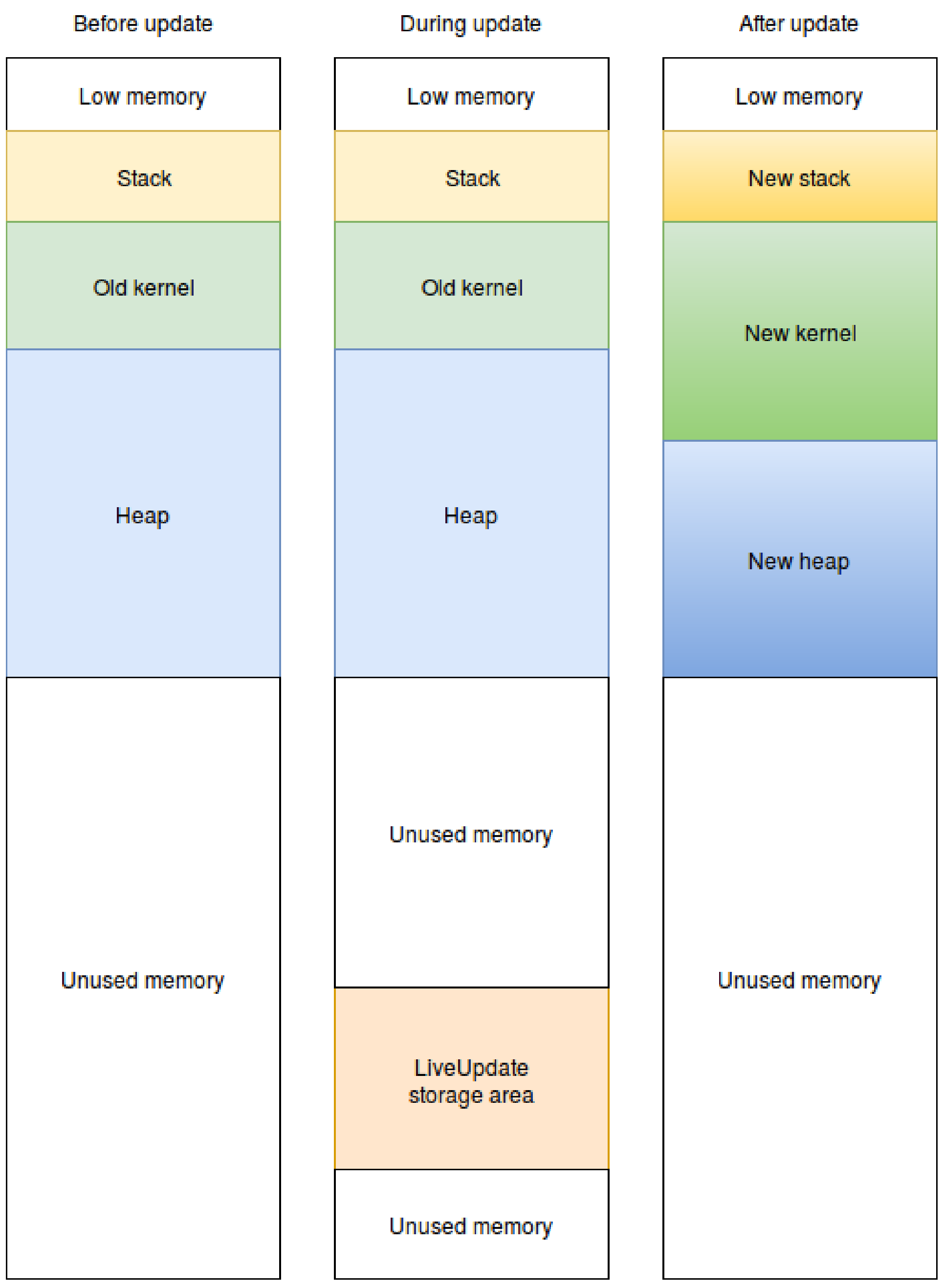


Figure 6.1: Memory layout of LiveUpdate operation

```
13742 root 20 0 1710M 93052 12324 S 0.0 3.0 0:00.00 qemu-system-x86_64 ...
13742 root 20 0 1710M 99M 12324 S 0.0 3.3 0:00.00 qemu-system-x86_64 ...
```

As shown in the *htop* output above, the resident memory for a VM running a simple LiveUpdate service rises by 6MB when live updating to itself. The state storage area is rather small comparatively and contributes very little. In this test case no state was written, so the contribution would have been zero. The 6MiB rise would be the live updating process as well as receiving a binary over TCP. The VM was given 1GiB memory, which is shown in the virtual column (1710MiB). Since the VM can only have 1GB the remainder is reserved to Qemu.

6.2.1 The design

In the beginning there was only the thought that a running kernel could be replaced with another. So, the first live updated system didn't preserve its state. A small service was written that sent its binary from the host system over a TCP connection, copied the binary over the existing running kernel using a special kernel hotswap function, jump to the ELF program header entry address and then it would consequently crash. After a few tweaks in the boot portion of the kernel to make things behave when hardware was already initialized from before, it worked. Unfortunately, it behaved erratically when the binary was changed even one bit. This was because the image was replacing an area where the CPU was already executing code on. The solution was to create a freestanding function to do the *kernel hotswapping* from. The function had to have absolute addressing and use absolutely no external functions, heap or stack from the old environment.

This new function is simply called *hotswap*. The hotswap function doesn't naturally reside in its fixed area of memory (0x8000), so it has to be copied there manually prior to calling it. The function cannot call any functions from the old kernel (or new kernel), because the environment changes during its execution, so it has to be written in plain C/C++ and it can't use any non-freestanding features. After the new kernel has been installed it will pass a few parameters to the entry point function and then jump to the new kernels entry point. This entry point is usually a plain old C function, and only requires the CPU to be in 32-bit protected mode. The new kernel then recognizes and preserves these values, passing them all the way into *OS::start* where they get processed. There is an extra stage on 64-bit systems where long-mode is exited before the jump to the new kernel.

The new kernel will begin its normal booting process. This process usually takes a bit of time, and since the hardware is always going to be the same across updates, something the author just calls soft-reset was invented. A special function was added to the unikernel kernel that stores a small struct somewhere in memory containing information about the current hardware. Since the hardware never changes across updates, this means there is potential for speedups across updates as these bits of data don't have to be recalculated or re-initialized after each update. After an update the OS then sees this structure, verifies it, applies it and then resumes booting, but can now also skip the most time-consuming processes during initialization. The soft-reset process uses only 2.6% as much time as a normal boot, mostly due to skipping CPU frequency sampling. CPU freq. sampling on its own takes 180ms, and a normal boot on the authors machine takes about 186ms. LiveUpdates soft-reset takes only 5ms on the authors machine, which is a clear improvement that could make live update work even in environments with real-time requirements.

Occasionally an incomplete binary would be received which was erroneously accepted by LiveUpdate. To remedy this the binary was to be scanned for an ELF header, which, if not found would cause an exception to be thrown. After the ELF header was discovered the effective length of the ELF binary was calculated and compared to the length of the update binary. There are no redundancy checks in the ELF format, so LiveUpdate is unable to verify the consistency of the update binary. It would likely be done by attaching some kind of checksum as metadata to incoming update binaries.

The minimum example that applies an update, stores no state across the update, and immediately changes execution to the updated service, is a one-liner that hands LiveUpdate an ELF binary:

```
void Service::start()
{
    liu::LiveUpdate::begin(storage_location, update_binary);
}
```

For IncludeOS, everything (code and data) is moved around with even small changes. This is because the kernel and service is built together into a tightly optimized and largely anonymous amalgamation mainly due to code inlining. C++ in particular with template programming will inline more code than other programming languages. In addition, it is not certain that a kernel function would not just be inlined and "ceased to exist" after linker stage. The IncludeOS kernel and service is linked together into an amalgamation that has no division between kernel and userland. LiveUpdate uses the state-transfer based DSU approach. Since the OS and the service code is tightly integrated it is possible to replace all the code in a running service, but still retain connections and data. Everything above the hardware layer is accessible to the service, and of course it's also possible to serialize hardware automatically, but unnecessary. This opens up interesting but probably obscure use-cases, such as having a unikernel monitor connected to clients, where it would upload programs to these clients realtime and have them update themselves into running these programs. The connection would be retained, and the clients would continue waiting for more programs to run, effectively having 100% uptime on tasks. The closest existing counterpart for this is Function-as-a-Service. Updating a system to only perform a specific function makes it more secure and harder to attack. This is the primary niche that unikernels fill today.

Security-wise, LiveUpdate adds risks in that the OS has to be allowed to overwrite itself with new/different executable code. On unikernel monitors this changes abit as the guest typically will ask the hypervisor to perform the hotswap. That adds an extra layer of security both because the hypervisor maintains memory access rights and because the hypervisor can validate any signing and accept the new binary to be hotswapped in.

To make LiveUpdate work on other unikernels other than IncludeOS, a C-based API may need to be exposed, and a wrapper for the language used by the unikernel needs to be written to interact with the C or C++ interface.

6.3 Update times

The live update procedure is much like *kexec*, but it skips all the time-consuming phases of booting, and unikernels typically don't have any complicated startup procedures. Linux, unfortunately, does have to initialize all of its subsystems still. Also, due to storing state across updates, LiveUpdate can avoid things like having to re-authenticate with other services.

When live updating, a built-in feature in IncludeOS called soft-reset is employed, added by the author to support LiveUpdate. The kernel has special code at boot that checks whether a magic code was provided when calling start or not. In addition to the magic code there is also a pointer to data, which is used only if the magic value is known. This code path is usually used to detect and enable multiboot features if the magic value is correct. However, in the case of Live Update another magic value is provided, making the kernel call the soft-reset codepath with a pointer to its pre-update data. The validation function checks that the soft-reset structure has sane values, that the checksum is correct and then proceeds to load certain values directly into the operating systems structures. During boot, the kernel will occasionally have to do some very time consuming work such as calibrating timers or calculating CPU frequency to a high degree of precision out of necessity. These tasks can be skipped entirely during soft-reset, and reduces the boot time by 2 orders of magnitude from hundreds of milliseconds to a few milliseconds. The values loaded from soft-reset consists of, for example, the known CPU frequency, which doesn't change between updates, or APIC timer tick frequency, which is also a constant. These stages of the boot process are the major contributors to boot time, and is skipped entirely when soft-reset is used.

It should be mentioned that even if saving startup time was not possible, soft-reset is still necessary. It conserves important information lost during a normal boot, such as the upper limit of physical memory. Without these values the system cannot operate optimally, and so it is an important part of LiveUpdate. Though it certainly is possible to live update without soft-reset, not knowing how much physical memory your service has is a scary thing, since writing outside of physical memory is a no-operation. It is for this reason that LiveUpdate actually verifies the end of its storage area by writing a canary value and then verifying by reading the memory back right after (with optimizations off). If the canary fails, a C++ exception is thrown and the service writer can then choose to either lower the address

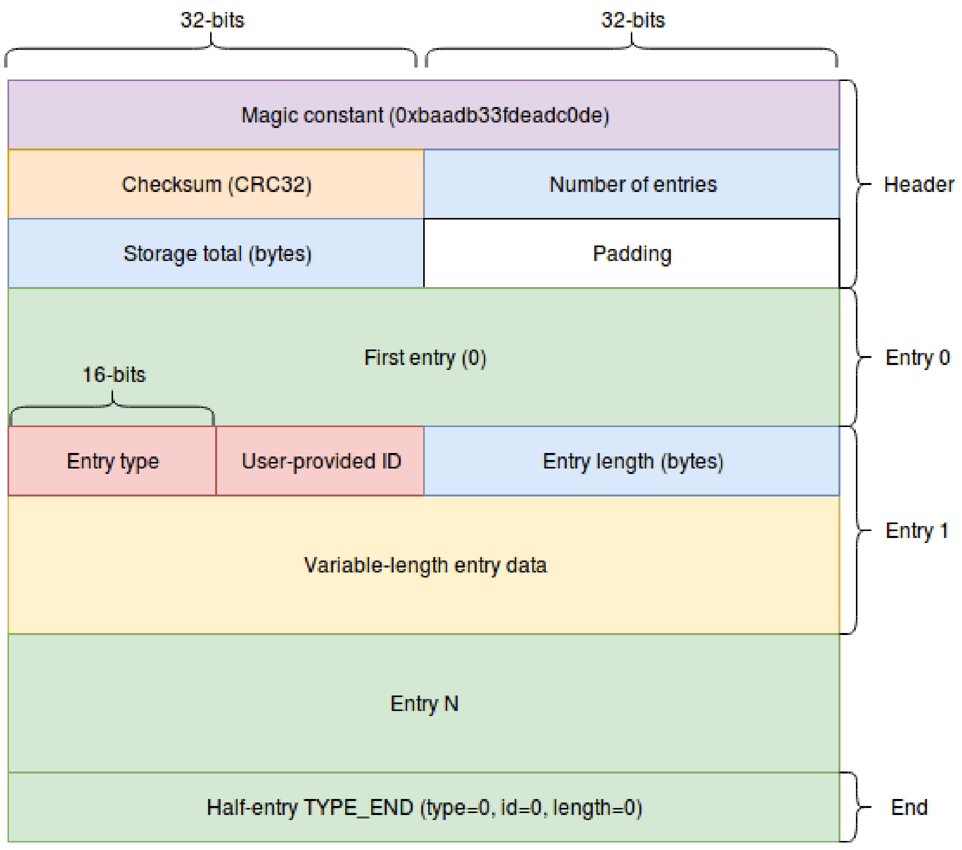


Figure 6.2: In-memory storage structures

of the storage memory area and try again or gracefully give up the update process.

6.4 Limitations

The LiveUpdate project does not cover authentication, authorization, secure updates, secure state transfers, or binary signing and versioning. The serialization system does provide mechanisms for working with and maintaining backwards compatibility for versions of program state.

6.5 Storage

LiveUpdate uses a fixed memory location to store all the programs state during the update process. The location is chosen by the service writer, and should be located in a physical memory location that is sufficiently far from the end of the heap in single address space operating systems.

LiveUpdate has no concept of size of the storage area and it is because it can never be inside an operating systems memory anyways. The operating system and the live update system are independent during the update process. Even if we knew how much storage was needed down to the last byte there is no way to know with any degree of certainty the memory layout and heap size of the new operating system currently being installed. In other words, the memory footprint of the future operating system is unknown until after the update and after fully booting and initializing the new system. Because of this, LiveUpdate will always start from a safe region in memory and grow towards the boundary of physical RAM.

If after the update the heap has entered the storage area, then the state will be fully discarded. The same applies when the heap enters the LiveUpdate area during the resume process. LiveUpdate does update the system properly, but it will lose its state. It is therefore important that programmers place the LiveUpdate storage location far out in RAM, so that both heap and storage can grow without interfering with each others.

Should the heap enter the storage area while the storing process is happening, the storing process will cancel and an error will be thrown. The updating process will not have affected the programs state, and the program can remain running having missed the update. Poorly written services may be prevented from ever updating their systems because the storage area and the heap always grow into each other.

There is also a memory write validation check performed at the tail end of the storage data, which makes sure that there is actual physical RAM there. When writing outside of RAM nothing happens, so if writing to the area does nothing (its all zeros after), then we can be sure that we are outside the RAM allocated to the guest, or if running on bare metal, outside actual physical RAM. If running inside a virtual machine and there is not enough physical memory, or running on an IoT device with low physical memory, a canary will detect we are outside RAM and the update process will be canceled gracefully. The system returns to normal operation, not losing any of its state but also not able to live update.

Figure 6.2 shows the internal storage structures used when serializing the systems state during an update. The beginning of the structure holds a magic constant that identifies the beginning of the storage area. A CRC32 checksum acts as a data consistency check and the total bytes field is the length of the whole area. The number of entries field is the number of serialized entries appended to the area. After the structure ends the first entry begins, and at the end a mandatory *TYPE_END* half-entry is appended to mark the end.

6.6 Usage

6.6.1 Storing state

A user-provided storage callback function is called when *liu::LiveUpdate::begin(...)* is called. This storage function passes with it a storage object which is the service writers interface to the *LiveUpdate* state storage.

```
liu::LiveUpdate::begin(live_update_area, save_state)
...
void save_state(liu::Storage& storage, const liu::buffer_t*)
```

```
{
   // save integer value 1024 as id 0
   storage.add_int(0, 1024);
   // delineates end of a variable-length structure
   storage.put_marker(0);
   // ..
   // save some other structures
}
```

The storage object can be thought of as an opaque interface for writing serialized state. Each time state is stored, an internal writing position is moved that always points to the end of the writable storage area.

Internally, each bit of state stored in the storage area has a mandatory type, an ID and a length, which is called the storage entry header. Only a few individual types are recognized:

```
enum storage_type
{
  TYPE_END     = 0,
  TYPE_MARKER  = 1,
  TYPE_INTEGER = 2,

  TYPE_STRING  = 10,
  TYPE_BUFFER  = 11,
  TYPE_VECTOR  = 12,
  TYPE_STR_VECTOR = 13,

  TYPE_TCP = 100,
};

struct storage_entry
{
  int16_t   type;
  uint16_t  id;
  int       len;

  ...

};
```

The types *END*, *MARKER* and *INTEGER* needs only the storage header, and always consumes 8 bytes. *END* is a special type that marks the end of the state storage area. The type is intentionally 0. Integers store their integer value in the length field, which saves some bytes, as integers are very common state.

Markers help define variable-length structures in the serialized state. Markers do not store any custom data and only have an ID. They have special resume functionality that lets the service writer jump to the next structure in the serialized state. When a marker is placed, specifying an ID gives writers the ability to skip to the first marker with the given ID. Otherwise, if no ID is given it will receive the ID 0 internally.

Otherwise, the interface can store all the primitive types as well as C++ vectors, string and has a special case for string vectors. For anything more complex service writers will have to serialize the objects themselves. This is commonly done by just adding structs directly with the *add_type<struct_name>* template function. It will internally store it as a buffer, where the length of the buffer is the size of the struct. When restored, the length is checked to see if the structures are at least the same length.

Unfortunately, there is no simple way to verify that the types that go in are the same types that comes out. Specifically, the internal *typeid* used for types in C++ is the de-facto type system with no additional compiler assistance. The problem is that even if LiveUpdate stored this, for another system to resume from this state it would have to have been compiled with the same compiler, and probably the exact same version, with the same version of the C and C++ standard library. This is not realistic. So, type-validation is limited to what service writers themselves are willing to do to verify types and values, and length checking. There are DSUs out there that store type information via compiler assistance, in theory [36], and in practice [26]. This enables type safety across updates.

For the more complex types, such as vectors, string vectors and TCP connections, extra structures are used to help serialize the data. For vectors, a segment header is used to store vector data such as the number of items in the vector and how big they are. The storage entry header has the same function as before, the length field is the length of the whole thing. The remaining data is stored sequentially in memory. This makes deserializing fast as it can be treated as a C-array when reconstructing the vector.

For the string vector all the elements are variable-length, so we have to introduce an extra level. A structure of length and a VLA-field that points to the data is provided for each entry. That way, when reconstructing the string-vector, as each string is reconstructed, it is put in the vector is already has reserved space for all the data.

TCP connections are more complicated, but the same processes apply. Store all the normal members of the TCP state: the TCB, the current and previous TCP state and RTTM, and many others. Then, store the read queue which is a single buffer. And finally, for each element in the write queue, store them with the string-vector method. TCP connections don't have a huge memory footprint. There are experiments further down showing that we can serialize and deserialize a great number of TCP connections quickly.

6.6.2 Resuming state

A user-provided resume callback function is called when resuming which provides the user with an interface that can restore state. The state is restored in the same order it was saved.

```
liu::LiveUpdate::resume(live_update_area, resume_state);
...
void resume_state(liu::Restore& thing)
{
   // restore the saved integer
   int value = thing.as_int();
   // go to the next stored state
   thing.go_next();
   // go past the end of the first marker with id 1
   storage.pop_marker(1);
   // ..
   // restore some other structures
}
```

When the resume function is called, a Restore instance is provided which is implemented as an iterator interface into the serialized storage. It's only possible to iterate forwards. For each position, the service writer can retrieve type, id and length. If an unexpected type or id was found, it is recommended to jump to known markers instead of throwing exceptions. This way at least partial state may be recovered, and it prevents total state loss just because there was a slight error in the user-side of the serialization or deserialization process.

After retrieving each bit of state, *go_next()* has to be manually called on the iterator to go to the next bit of state. The reasoning for this is simply in the naming. When deserializing an integer *as_int()* is called, which as the name suggests deserializes and returns an integer for the current position in the state. There are other reasons too. What if the service writer wanted to jump to a marker? By going next automatically it's possible to skip a marker if it's the next item.

```
liu::LiveUpdate::resume_from_heap(live_update_area, resume_state);
```

The function *resume_from_heap* is used when the state storage area resides within the systems heap already. This is the case when the state has been stored somewhere temporarily, such as on disk, or was received over network. In those cases the length of the state is known already, and the receiver is typically in a state waiting for the state storage. The receiver may then wish to resume the system from the data. The normal *resume* function has extra checks to prevent the user from accidentally writing into the LiveUpdate storage area, however in this case the storage area is likely to be safe within the heap.

6.6.3 Handling variable-length structures

When building services with live updating support versioning has to be considered. The most painless way to handle new data members in old structures is by always appending the new members to the end of the structures. This can be achieved by: Read the original structure in full, read extra members one by one until the structure end marker was reached. Jump to the end of the marker by using *pop_marker*. The last action will iterate past the structure, using the marker, so that even if a rollback to an older version with a shorter structure is performed, the resuming process will not fail with an exception because structures have changed size. Whether or not a partial restore is ideal depends on the service.

```
struct state
{
   // this structure is understood by all versions
   struct v1
   {
     int x;
   };
   // structure for a newer version
   struct v2
   {
     int y;
   };
   // this integer is also always present
   int integer_value;
};

// save state on a new version
void save_state(liu::Storage& storage, const liu::buffer_t*)
{
   // save v1 struct as id 1
   storage.add<state::v1> (1, program_state.v1);
   // save v2 struct as id 2
   storage.add<state::v2> (2, program_state.v2);
   // mark the end of a variable-length structure
   storage.put_marker(1);

   // store an always-present integer with id 100
   storage.add_int (100, program_state.integer_value);
}
```

Now let's assume the system crashes and a rollback is performed. We can then partially load the old state, and we can assume in this case that the partial state restored will be enough for the system to make a decision on whether or not to keep the state. If the state is kept the system resumes on that state, otherwise the system would boot normally.

```
// resume state on an older system version
void resume_state(liu::Restore& thing)
{
   // restore the v1 structure
   program_state.v1 = thing.as_type<state::v1> ();
```

```
    // jump past the end of the whole structure (using marker 1)
    storage.pop_marker(1);

    // restore the always-present integer
    program_state.integer_value = thing.as_int(); thing.go_next();
}
```

The old system read the *v1 version* of the state, and then skipped past any remaining state still within that marker. It continues by reading the always-present integer value at the end of the state structure. We can see that no matter which version of the state was stored, we can reliably restore the state at least partially for each version, with some planning. The system will have to have some way of deciding whether or not the state is within parameters to confidently resume the system. If the state has suspicious values, perhaps due to restoring from a newer version of the state, it may be prudent to throw away the state and boot the system normally.

6.7 Failure and recovery

In the event of failure to deserialize the stored state, eg. the call to resume() throws an exception, the user might want to rollback to an earlier version. In that case he can just start a rollback if he has a known good image to use for that. The service writer may also ignore the failure, and simply proceed without restoring any state. In that case the service would boot as normal, but no state would be maintained across updates and it would be as if starting up a new VM with an updated service image.

6.7.1 Rollback

A problematic issue during the work on this thesis has been working on the ability to rollback changes should something happen. There is not currently any way to do a guaranteed rollback to a previous kernel with the current state intact. For many reasons, a rollback should try to do the least amount of things and just instantly go back to a known good version. For example, a new version could run out of memory due to a memory leak, in which case trying to save state before doing the rollback would be fatal, ending in an infinite loop of kernel panics into failed rollbacks. This is mitigated by just performing a hardware reset should the panic function re-enter itself, but resetting hardware means resetting memory, and so the service disruption will be greater than when only doing a soft-reset during a rollback. In addition, there is no hope of recovering any saved state, should anyone try to resume from memory.

That said, LiveUpdate does have rollback functionality, which can be triggered by the service itself should it find the environment insane, or even when CPU exceptions occur, which are usually something out of programmers control. In IncludeOS it's possible to have the operating system call a function when it panics, and when LiveUpdate is linked in, that function is being set to do automatic rollback if there is a known good kernel set.

If a known good kernel is not set (via the LiveUpdate API), then the rollback system will just hard-reset the system instead, hoping that a hardware reset loads a known good kernel. If LiveUpdate had no rollback functionality at all, a good alternative would still have been to just hardware reset into a known good kernel. This is because most VMMs require a physically stored kernel image to be able to load guests.

Implementing automatic rollbacks as a service writer is fairly straightforward:

- Load a known good image from any source
- Store the image somewhere safe, such as just before the live update storage area begins
- Tell LiveUpdate where the image is via *set_rollback_location()*

Once LiveUpdate knows about the rollback buffer, the rest is automatic. Any CPU exception, assertion failure or kernel panic will trigger automatic rollback:

```
void store_rollback_binary(binary_location, size)
{
    liu::LiveUpdate::set_rollback_blob(binary_location, size);
    // after this point, any CPU exception or recoverable system
    // failure will trigger a rollback
}
```

After setting the rollback binary, it's possible to manually begin an immediate rollback process:

```
void immediately_begin_rollback()
{
    liu::LiveUpdate::rollback_now();
    // the rollback function never returns here
}
```

Rollbacks work on the uKVM (solo5 platform) unikernel monitor, on regular PC using Qemu, VMware, VirtualBox and also works on the IncludeOS-supported cloud vendors, such as Googles GCE, VMware vCloud and OpenStack. Rollbacks use the regular live updating path with no state saving.

6.7.2 System lockup

There are cases where, for example, interrupts are off, no CPU exceptions can happen and the service is now sleeping or looping somewhere forever. The system is then stuck, and can only be restored by hard rebooting it from an outside source, such as through a cloud-providers web-based interfaces.

As a potential solution, this can be solved by having the cloud-providers monitors periodically ping services (or monitor network traffic), and if they fail to respond within a certain time, hard reboot them. This problem can occur with any operating system on any architecture, even on desktop systems. The author does not know if anyone does this presently.

In non-preemptive (largely event-based) systems, a simple loop can stall the whole system. It could even look like the loop was legitimate by adding the x86 pause instruction into it, so that it looks like a spinlock loop:

```
void stall_system_forever()
{
while (true) asm("pause");
}
```

6.8 Live migration

Migrating from one system to another is simple enough from LiveUpdate, but it does require some outside assistance, just like with other live migration systems. For one, when you transfer a VM from one machine to another they have to have the same IP and routing tables need to be updated to make the network flow resume properly. Other live migrations transfer the entire VM state, often multiple times because the state changed while transferring. CPU extensions and system features can conflict from system to system, and they ideally have to be the same (hence qemu supporting virtual CPUs like kvm64 or qemu64). LiveUpdate cannot incrementally send changed memory, but the state it needs to send is typically very minimal, and it also doesn't care about CPU extensions or system features as it does a partial reboot (soft-reset).

So, what remains is to prove that migrating with LiveUpdate does not take too long. If it takes too long it might have to start sending changed state, but there is no way to tell what that is, since the state-serialization function is user-provided. The solution might be to simply start over resending state if the process takes too long.

Pseudo-code for a potential method of ensuring that the process doesn't take too long, but still live migrates in the end:

```
void live_migrate(data, timeout)
{
    // start transferring data
    TCP.send(data);
    // if the transfer takes too long ...
    Timers::oneshot(timeout,
    [data, timeout] (int) {
        // close the connection
        TCP.close();
        // start over from the beginning
        live_migrate(data, timeout);
    });
}
```

On the other side the receiving end receives an optional binary and a mandatory state. There are separate ways the remote system could resume the state. One is for the remote system to reboot into the new binary, then resume the provided state. This adds delay, in addition to the time already spent receiving the state. Another, given that the remote system is already booted with the relevant binary, and is waiting for state, is to just straight up resume the state. This method is fast, does not require hypervisor assistance and is safer since it does not require live updating.

```
void restore_from_state(state)
{
    liu::LiveUpdate::resume(state, resume_function);
}
```

The total time required is still dependent on the time it takes to deserialize and apply state, but given that the live migration takes place on a local network, the total time would be limited to transfer time + validating, deserializing and applying state. This could be anything from 1ms to 10ms depending on the complexity of the state of the migrating service.

6.9 Persisting state

6.9.1 Locally, across updates

By storing data in high memory, way outside of kernel, service code and heap areas, an implementer can keep this data persisted across updates without having to store and restore it. It will remain untouched across updates, given that the updates succeed.

As an example, a human genome project service could be doing map reduce on 5GB of data. That data could be safely stored after the 4GB 32-bit boundry in 64-bit. If a bug is then fixed in the service program itself, the program can be safely updated without modifying the data, and the new program can immediately resume working on the data.

6.9.2 Remotely, as backup

LiveUpdate can persist state to a remote server without making any changes or implementing new features. The feature, however, will not be ideal and fall flat on its face should the state grow over time or simply be too big from the start. The reason, is that it is not possible to know what state changed over time, how that will affect the saved state as a whole and how to partition this into smaller parts so that we can persist partial state changes, which scales better.

To support persisting state with some degree of scalability further work would have to be made either to have LiveUpdate support this directly, or serialize partial state some other way, manually, for example via *Google Protocol Buffers*.

A very rudimentary example of persisting state would be to transfer the state either periodically (such as every second or minute) or every time something worth persisting happens. If the service goes down, the persisted service could know this by doing keep-alives on the connection between them, however external assistance is still needed to reconfigure the network to forward traffic to the backup system.

Bottom line, the author argues that LiveUpdate is not ideal for persisting state as-is, and will need modifications either in LiveUpdate to add new support for persisting partial state, or modifying the service itself to use a more flexible state persisting approach.

6.10 Deep freeze

Interesting topics always pop up when order of magnitudes can be saved here and there. It is possible to put systems into a deep sleep with only the absolute minimal state required stored on each system. This is possible because LiveUpdate already knows what state the program needs to resume operation. This will enable providers to have many more inactive services in a *deep freeze*. Typical examples would be Web-servers with few or no active clients at certain times of the day, or databases that just aren't very active in general because they are used for very few things. This is most likely already being done as cloud providers will save resources by doing this, but saving only the absolute minimal state can be interesting.

On IncludeOS doing this can be achieved in many ways. In an ideal world we have hypervisor assistance, but for now we will just store the state via networking. That is, when shutting down a service, we send its data over TCP which is connected to a local file on the host system, then when we restart the service some time later, we will just pass the state over TCP and have LiveUpdate restore this state for us.

We assume the service is currently not in use, which is why we want to shut it off in the first place. Because of this, we can just store the system state whenever we want and just turn the guest off in any manner. The resuming process is, however, time-sensitive as it is likely triggered because someone is trying to access the service and it is currently stored in deep freeze. To resume it, the hypervisor has to boot the OS the normal way, and it may not even be on the same physical machine (or the guest may not have the same parameters) so we can't make as many assumptions that would save boot time. After booting up, the service has to have its stored sent to it one way or another, ideally during boot just like LiveUpdate normally does it, but unfortunately, this requires hypervisor assistance, so for now we will just assume the state is being transferred over network or maybe read from disk. After the state is transferred over, we have to LiveUpdate into the new state. This can be done in a number of ways, but the simplest is to just manually resume from the loaded state by calling *LiveUpdate::resume* with the address of the state. LiveUpdate will then proceed to use the same process as it would be doing normally when the system is resuming from a live updating process. In other words, no live updating happens, but it still requires the system to boot back normally, obtain its old state somehow, and then simply resume from that state. An experiment has been provided in chapter 3 showing one way to do this, however inefficient it is. It is possible to make this process extremely efficient with hypervisor assistance.

Pseudo-code example showing order of events:

```
// this function is called when we receive an external signal to
// enter deep freeze
void signal_deep_freeze()
{
    // first, we store the state somewhere
    auto size = liu::LiveUpdate::store(storage_location, storage_function);
    // then we write it to a file or device on disk
    FileSystem.write("/deep/freeze", storage_location, size);
    // now we can just turn off the system
    OS::shutdown();
}
```

At this point we are done saving state and turning the system off. Now, at some point later the hypervisor boots up this service again.

```
void Service::start()
{
    // first, we have to check if there was some state stored on disk
    auto file = FileSystem.stat("/deep/freeze");
    // if the file exists, just read the whole thing into memory ...
    if (file.is_valid()) {

        // read whole thing to memory
        auto state = file.read();
        // begin LiveUpdate resume process immediately
        liu::LiveUpdate::resume(state, resume_function);
        // the process is complete and the resume was successful
    }
    else {
        // assume we did not resume from a deep freeze,
        // start service normally...
    }
}
```

6.11 Serverless FaaS using LiveUpdate

LiveUpdate can help facilitate serverless FaaS on unikernels that would run on existing infrastructure, such as public clouds. This is done simply by replacing functionality on nodes with low access frequency, or by scaling horizontally: starting new nodes on as-needed basis.

This feature will require orchestration so that a list of all current used and unused service nodes is maintained. Each time a function is required, the system would live update a free node to have this functionality, and as the functionality goes unused over time, the node would again be updated to receive new functionality as needed. The nodes have no fixed functionality, but is rather receiving the code as the execution is requested. Load balancing will make sure the requests are routed to the appropriate destinations.

```
======================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Function-as-a-service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Network configured
               IP:  10.0.0.42
               Netmask:  255.255.255.0
               Gateway:  10.0.0.1
               DNS Server:  10.0.0.1
LiveUpdate server listening on port 666
Receiving blob on port 666
[ BufferStore ] Allocating 256 new buffers (512 total)
* Blob size: 2456312 b  stored at 0x1e88080
* Live updating from 0x1e88080 (len=2456312)
```

At this point a new binary with other functionality is sent to the service.

```
======================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Function-as-a-service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2  3  5  7  11  13  17  19  23  29
```

We can see that the service has *new functionality* which lists the 10 first primes. In practice this new functionality would likely be implemented as something that could be completed and returned in a request-response manner. Like a remote function. And implemented using a commonly understood layer above TCP, such as HTTP.

Part III

Experiments

Chapter 7

Equipment

All experiments are done with an quad-core i4770k at 4.1ghz, on Linux 4.10 running newest *QEMU* from trunk.

```
$ uname -a
Linux gonzerelli 4.10.0-24-generic #28-Ubuntu SMP Wed Jun 14 08:14:34 UTC 2017
x86_64 x86_64 x86_64 GNU/Linux

$ qemu-system-x86_64 --version
QEMU emulator version 2.9.50 (v2.9.0-1051-gc6e84fbd44-dirty)
```

Chapter 8

Long-mode Exit

Many of the experiments done show that there is a difference between 64-bit and 32-bit. I also speculate that the cause is the time it takes to exit long-mode during live update on 64-bit. An experiment was made showing that LME has a fixed cost on 64-bit, and that without LME 64-bit would be slightly faster than 32-bit. (see: 8.1)

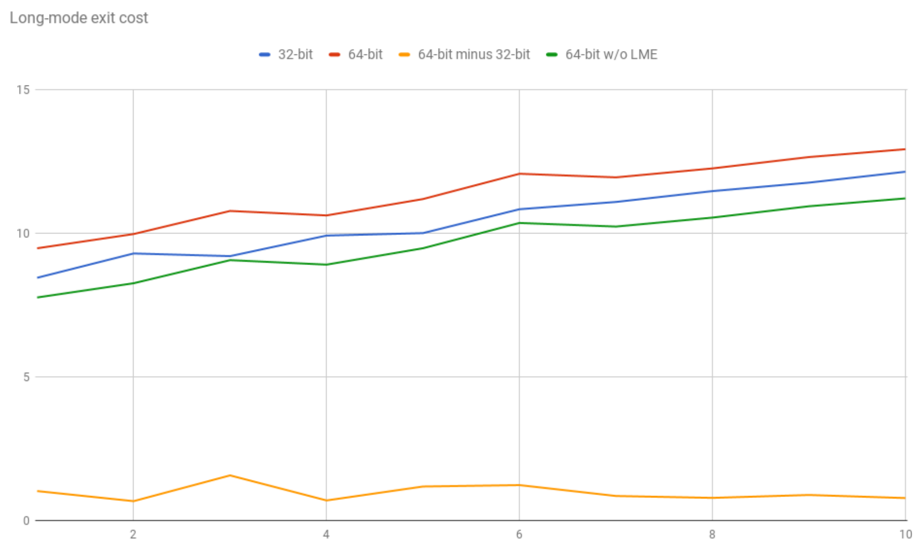


Figure 8.1: Cost of long-mode exit, 10 separate experiments

Chapter 9

Boot times

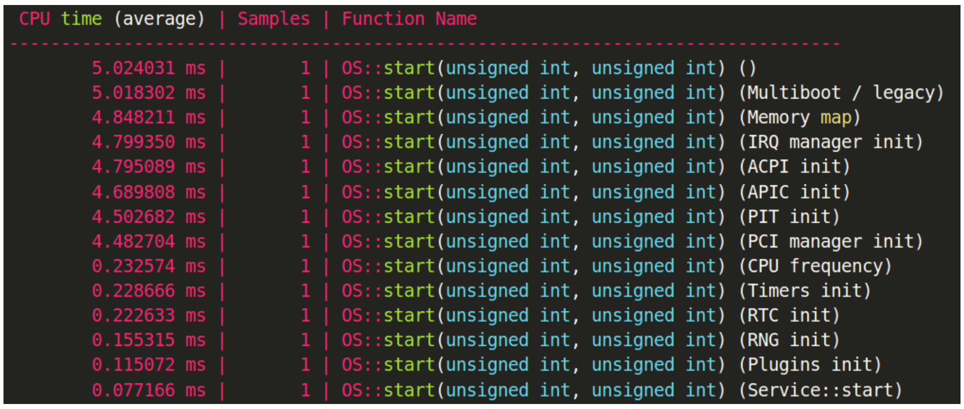
9.1 Basic service

9.1.1 Boot time-points

The service boot time charts (see: 9.1 and 9.2) show how much time has elapsed since startup until a specific time point has been reached. To find out how much each point actually used, we subtract the time point subsequent to the time point we are interested in, such as *Timers init* (0.1692 ms) - *CPU frequency* (180.2ms) = 180 ms. We can see that initializing PCI devices still takes 4ms even during soft-reset, but that the majority of the time spent during boot is in sampling the CPU frequency, which is skipped during soft-reset.

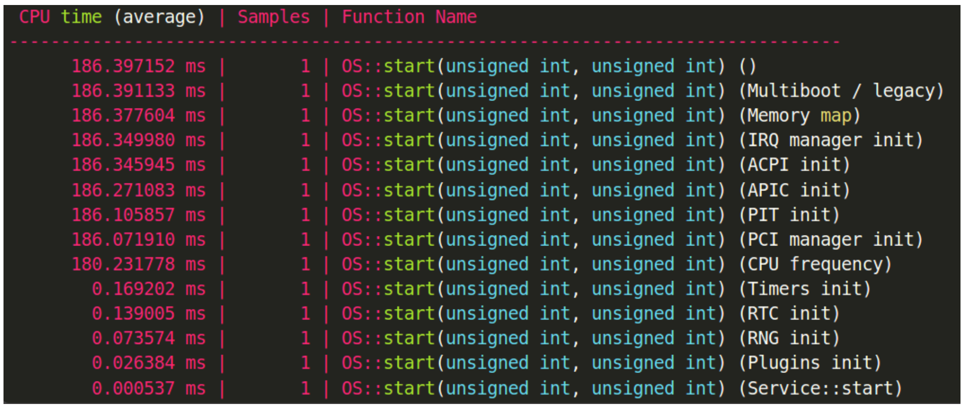
9.1.2 Automated measurements

By measuring update time on a basic Unikernel service, taking 30 samples in 10 separate experiments, we can investigate the time spent live updating a service and see whether the time needed is predictable. See figures 9.3 and 9.4. An important thing to keep in mind is that to guarantee the robustness of the test, the binary image used in the update process



```
 CPU time (average) | Samples | Function Name
-----------------------------------------------------------------------------------------------
       5.024031 ms |       1 | OS::start(unsigned int, unsigned int) ()
       5.018302 ms |       1 | OS::start(unsigned int, unsigned int) (Multiboot / legacy)
       4.848211 ms |       1 | OS::start(unsigned int, unsigned int) (Memory map)
       4.799350 ms |       1 | OS::start(unsigned int, unsigned int) (IRQ manager init)
       4.795089 ms |       1 | OS::start(unsigned int, unsigned int) (ACPI init)
       4.689808 ms |       1 | OS::start(unsigned int, unsigned int) (APIC init)
       4.502682 ms |       1 | OS::start(unsigned int, unsigned int) (PIT init)
       4.482704 ms |       1 | OS::start(unsigned int, unsigned int) (PCI manager init)
       0.232574 ms |       1 | OS::start(unsigned int, unsigned int) (CPU frequency)
       0.228666 ms |       1 | OS::start(unsigned int, unsigned int) (Timers init)
       0.222633 ms |       1 | OS::start(unsigned int, unsigned int) (RTC init)
       0.155315 ms |       1 | OS::start(unsigned int, unsigned int) (RNG init)
       0.115072 ms |       1 | OS::start(unsigned int, unsigned int) (Plugins init)
       0.077166 ms |       1 | OS::start(unsigned int, unsigned int) (Service::start)
```

Figure 9.1: Measurements during soft-reset



```
CPU time (average) | Samples | Function Name
--------------------------------------------------------------------------------
     186.397152 ms |       1 | OS::start(unsigned int, unsigned int) ()
     186.391133 ms |       1 | OS::start(unsigned int, unsigned int) (Multiboot / legacy)
     186.377604 ms |       1 | OS::start(unsigned int, unsigned int) (Memory map)
     186.349980 ms |       1 | OS::start(unsigned int, unsigned int) (IRQ manager init)
     186.345945 ms |       1 | OS::start(unsigned int, unsigned int) (ACPI init)
     186.271083 ms |       1 | OS::start(unsigned int, unsigned int) (APIC init)
     186.105857 ms |       1 | OS::start(unsigned int, unsigned int) (PIT init)
     186.071910 ms |       1 | OS::start(unsigned int, unsigned int) (PCI manager init)
     180.231778 ms |       1 | OS::start(unsigned int, unsigned int) (CPU frequency)
       0.169202 ms |       1 | OS::start(unsigned int, unsigned int) (Timers init)
       0.139005 ms |       1 | OS::start(unsigned int, unsigned int) (RTC init)
       0.073574 ms |       1 | OS::start(unsigned int, unsigned int) (RNG init)
       0.026384 ms |       1 | OS::start(unsigned int, unsigned int) (Plugins init)
       0.000537 ms |       1 | OS::start(unsigned int, unsigned int) (Service::start)
```

Figure 9.2: Measurements during normal boot

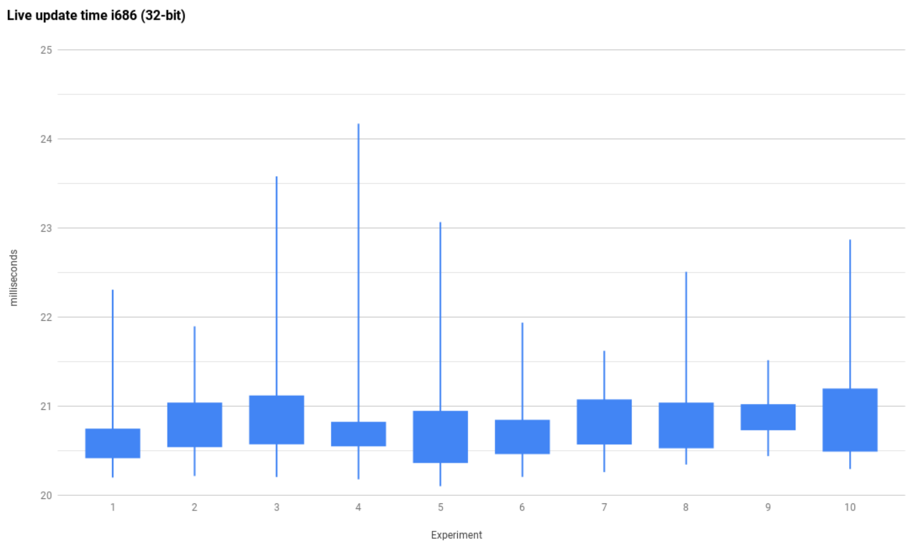


Figure 9.3: LiveUpdate time on i686 (32-bit)

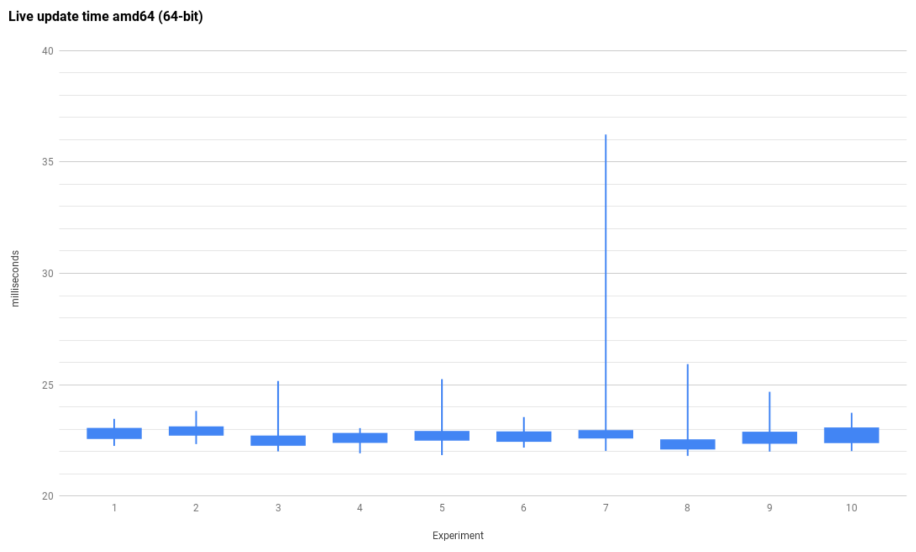


Figure 9.4: LiveUpdate time on x86_64 (64-bit)

is carried on during the sequential live updates, and so it affects the live updating time. The unikernel binary carried across live updates does affect boot times.

There is a 2 millisecond difference in update time between 32-bit (20.5ms average) and 64-bit (22.5ms average). This is not surprising because to live update 64-bit long mode has to be exited, and after the update a new page-table has to rebuilt. On 32-bit there is no such thing and after the update a simple jump to *_start* is performed. There is a large spike in one of the 300 samples taken on 64-bit which may have been the host operating system (Linux) doing other work during the live update time. It is not known for certain what caused the spike, which measured 36ms. The second highest spike in comparison is 25ms on 64-bit.

By disabling *soft-reset* we can see that the effect is quite large. The process is on average suddenly over 200 milliseconds. See figure 9.5.

Without soft-reset the CPU frequency has to be calculated, and the APIC timer has to be calibrated. The APIC timer calibration does not affect the outcome here, however, as it is an asynchronous operation running in the background during boot. It will have a smaller impact.

With modern hardware support, we can utilize CRC32 to get really low update times. See figure 9.6.

This is unfortunately not usable in all setups, and even if I select the appropriate hardware/software method based on CPUID, most local setups don't run with hardware emulation, and even fewer compile with SSE4.2 enabled in the compiler. As such, this option will most likely be strictly limited to production environments where the virtual machine guest is running on a hypervisor on a high-end server.

The hardware CRC32 support was not optimized when the experiments was first made. A second experiment with hardware CRC32 has been done

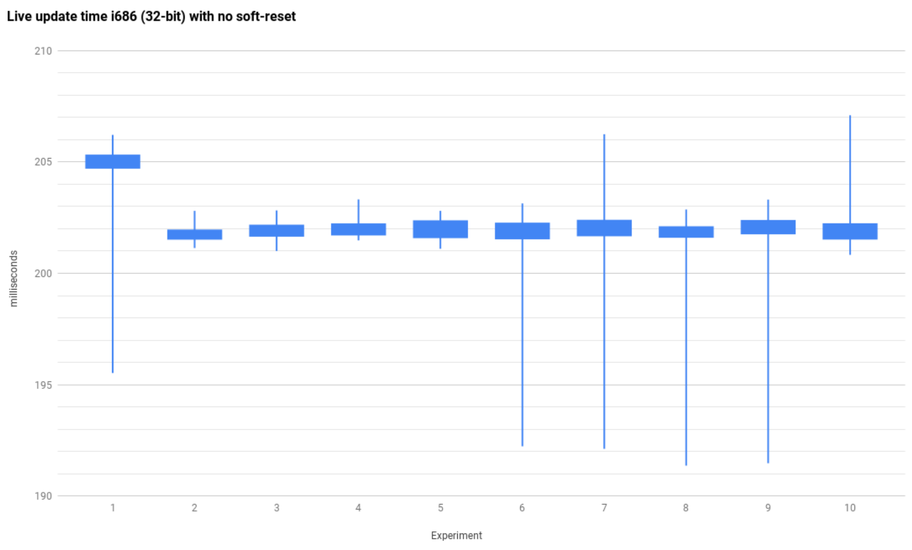


Figure 9.5: LiveUpdate time on i686 (32-bit) with soft-reset disabled

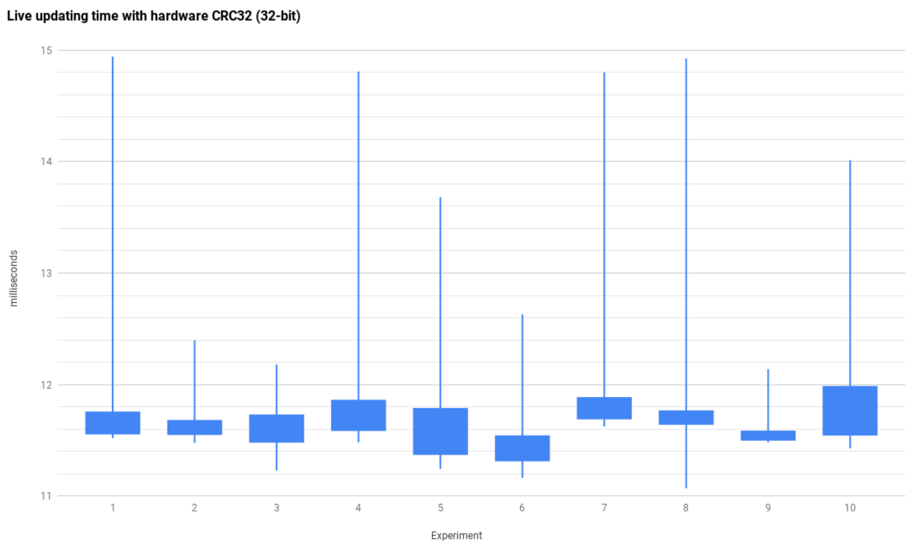


Figure 9.6: LiveUpdate time on i686 (32-bit), 30 samples on each experiment with hardware CRC32

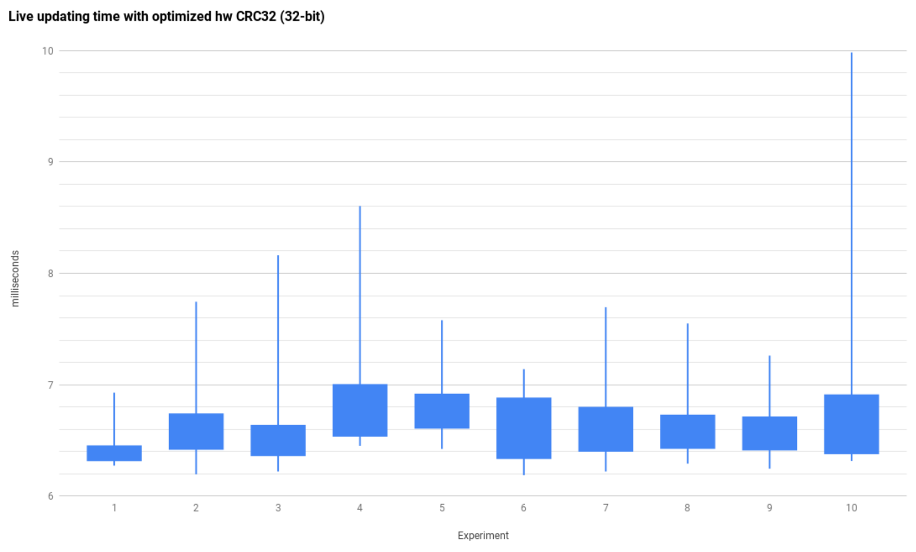


Figure 9.7: LiveUpdate time on i686 (32-bit), 30 samples on each experiment with optimized hw CRC32

to show that simply unrolling loops and using the higher bit instructions (such as *_mm_crc32_u32*) can reduce boot time all the way down to 6.1 milliseconds on 32-bit x86. See figure 9.7. This translates into 8ms update for the same experiment on 64-bit, as the difference is largely in exiting long mode. The experiment has a 2x 9ms spikes which may have been the Linux host operating system doing something during the experiment.

Finally, the last experiments for boot times on x86, x86_64 and finally uKVM-64 (later on) are done without carrying the binary across updates. It is suspected that the graph will show that the live update times are affected greatly by the reduced state. This experiment will show the lowest possible boot times.

The experiments were successful, showing that the update times when all conditions are perfect, such as hardware virtualization support, modern CPU instruction set extensions, small binaries and very little state stored across updates, that the update times can go as low as 5ms on 32-bit, and under 6ms on 64-bit. See figures 9.8 and 9.9. The experiments highlight just how costly it is to carry just 2MB extra state, due to how time-expensive checksumming is even with hardware support.

9.1.3 Effect on TCP flow

It is conceivable that the TCP flow is affected during live update process, reducing the speed or even causing loss that will trigger congestion control mechanisms which could be disastrous for throughput. An experiment has been made to measure this by sending 512MB of data to the service and updating once during this transfer.

The experiment was run many times with many different options. The

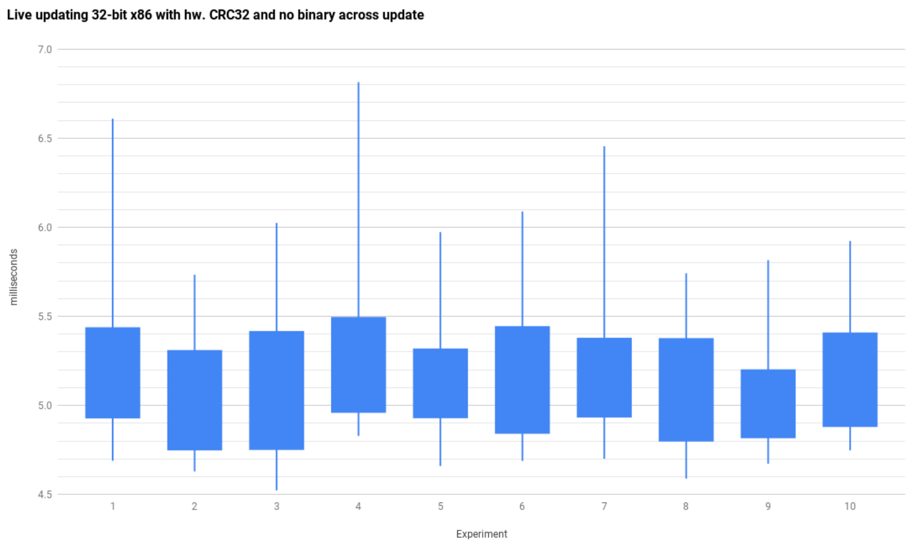


Figure 9.8: LiveUpdate time on i686 (32-bit), 30 samples on each experiment with hardware CRC32 and no binary stored across update

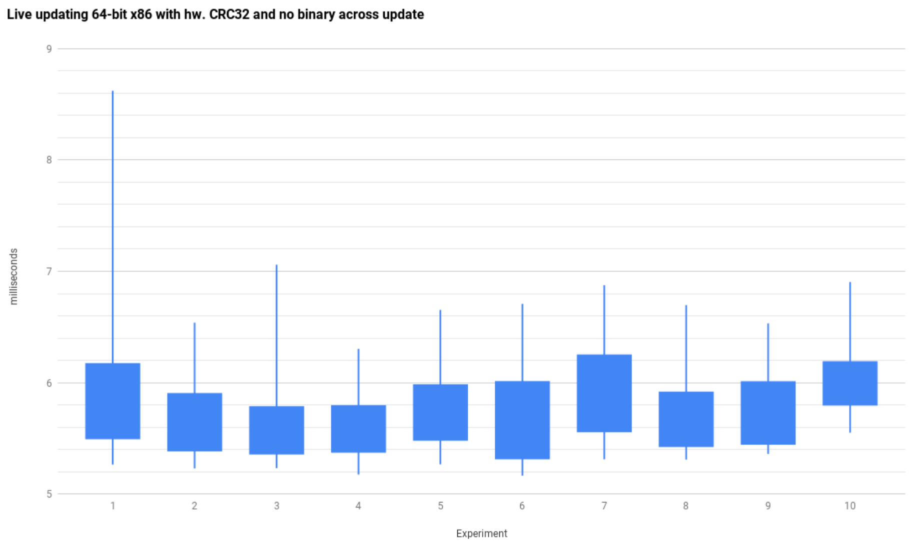


Figure 9.9: LiveUpdate time on x86_64 (64-bit), 30 samples on each experiment with hardware CRC32 and no binary stored across update

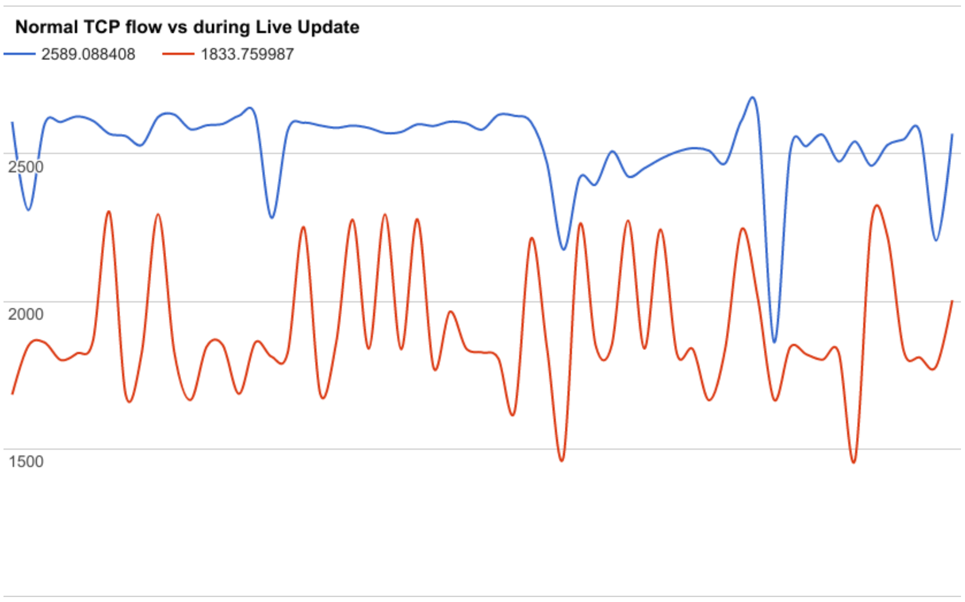


Figure 9.10: TCP flow measurements

first experiment (figure 9.10) shows that TCP throughput was reduced when a live update operation happened during transfer. To see the actual effect on TCP flow during a live update, another experiment was made that measures TCP flow exactly as it happens.

Without flushing the TCP flow is disrupted for 600ms during live updating. See figure 9.11.

The second TCP flow experiment shows that during a live update the packet flow is disrupted and actually reaches zero for 400+ milliseconds, most likely because Linux TCP congestion control kicks in. Surprisingly, somewhere near 25% of the total transfer time was spent in congestion control. Upon reading about Linux' *New Reno* algorithm, the hole is consistent with IncludeOS's lack of TCP reordering support. In other words, IncludeOS was not able to fill holes and Linux had to retransmit everything that was already in the air, causing a huge delay increasing the total transfer time by up to 200% (as estimated by the graph). There were a few cases where the congestion control didn't kick in and the effect on TCP was reduced and it almost flowed normally. The author speculates normal TCP flow will happen more frequently on live wires with latency that exceeds the live update downtime.

It was hypothesized that the TCP flow would be much better the network ring buffers were flushed just before updating. Turns out (9.12), it had almost no effect, and the effect may even be slightly negative. This may be because all it did was increase the time it took to live update, as flushing the queues is an expensive and time-consuming operation.

By accident I discovered that turning interrupts off at an earlier point had a strong effect on TCP flow. (9.13) This is most likely because the operating system is no longer telling the host system (Linux, in this case) that the queues have been serviced and we are ready for more,

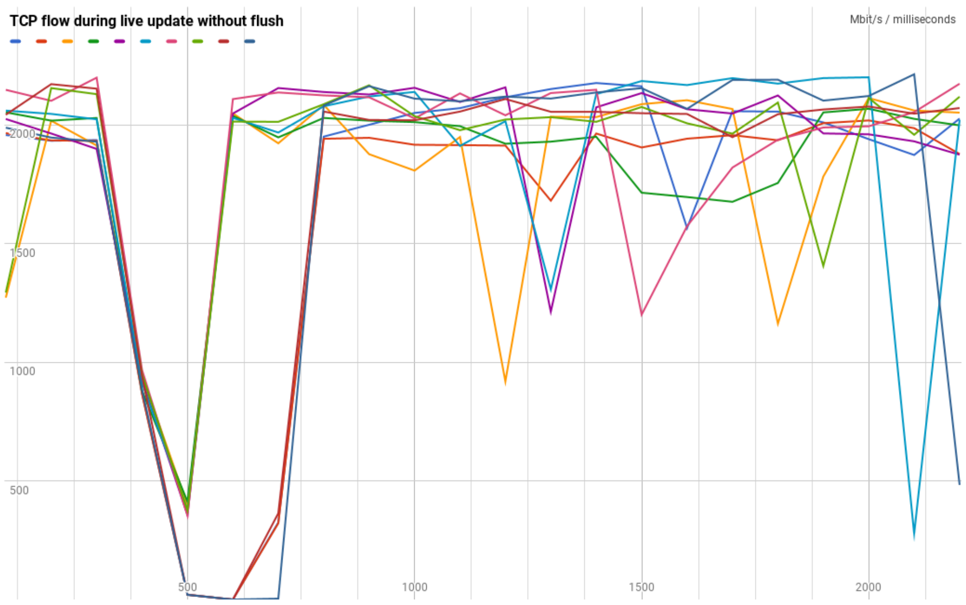


Figure 9.11: TCP flow during update with no flushing

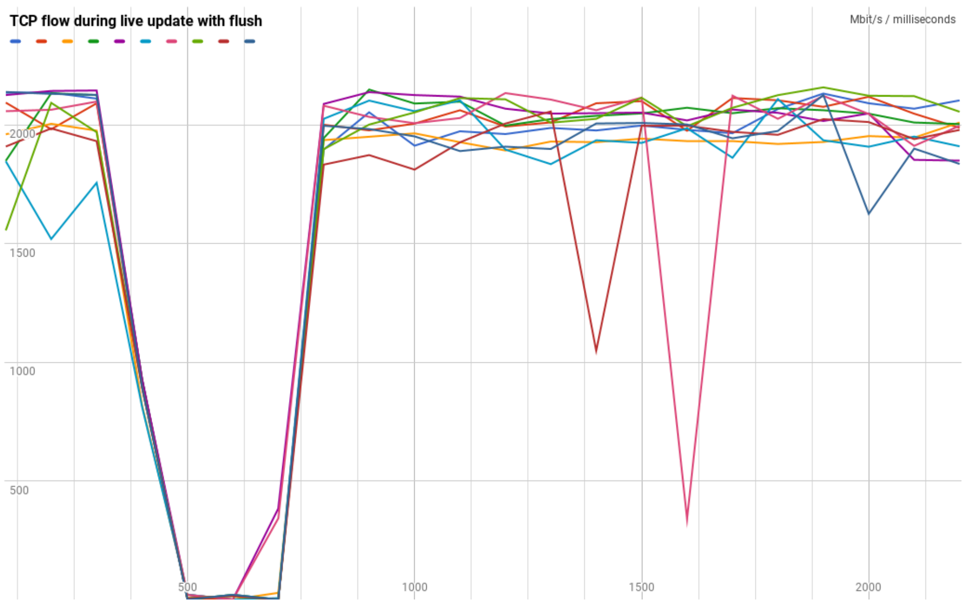


Figure 9.12: TCP flow during update with flushing

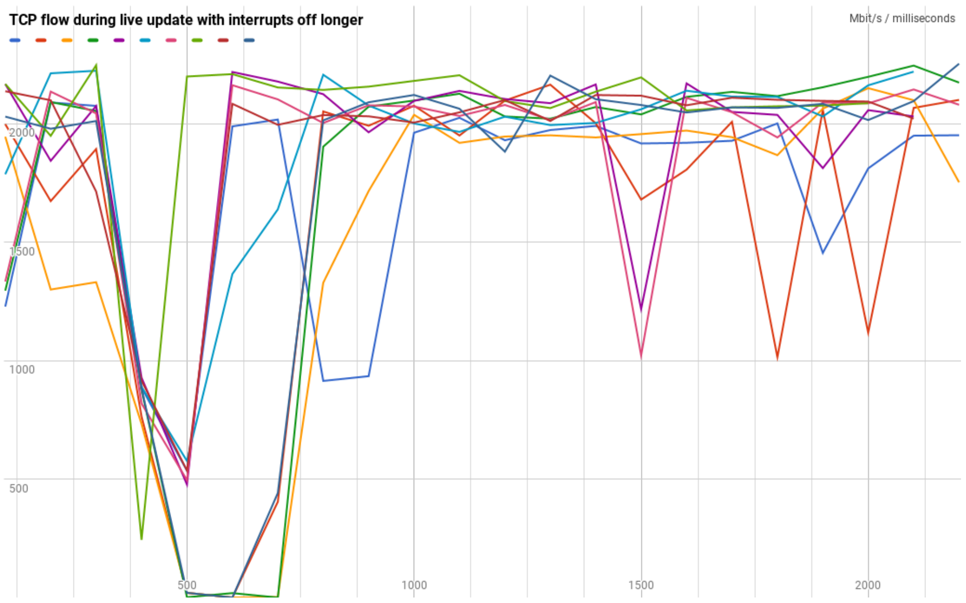


Figure 9.13: TCP flow during update with interrupts off earlier, samples every 100ms

which probably causes Linux/Qemu to start getting ready to move more stuff. It could have accelerated Linux/Qemu moving buffers around in anticipation of the guest operating system servicing network queues, which in this case happens way later due to live updating.

The last graph (9.14) is the same experiment as the third, but with 50ms time between taking samples instead of 100ms resulting in a more fine-grained graph. Same as the third graph, it shows that the time spent in congestion control is much less with interrupts off at an earlier point.

Without TCP reordering there will still be TCP-flow issues, however the effect has been reduced greatly due to turning off interrupt servicing as early as possible.

9.1.4 Effects of adding state

The author thought adding thousands of items to state would make the process very slow, eventually turning making it take so long that the service could be considered unresponsive to the outside. As shown by the experiment9.15 that measures the effect of storing an increasing number of 4-byte integers to the update process, the effect turned out to be rather small. The final experiment with 45000 integers is only storing a measly 176KiB of data, but it must be pointed out that the experiment is intended only to measure the cost of storing and retrieving individual bits of state.

The next experiment was designed to measure the costs of storing raw data, measured in megabytes, however it took forever to run. It took over 700ms to live update with a single buffer with only 10MiB data. The problem turned out to be the CRC32 algorithm being very slow. So slow, in fact, that it was pretty much unusable. (See: 9.16)

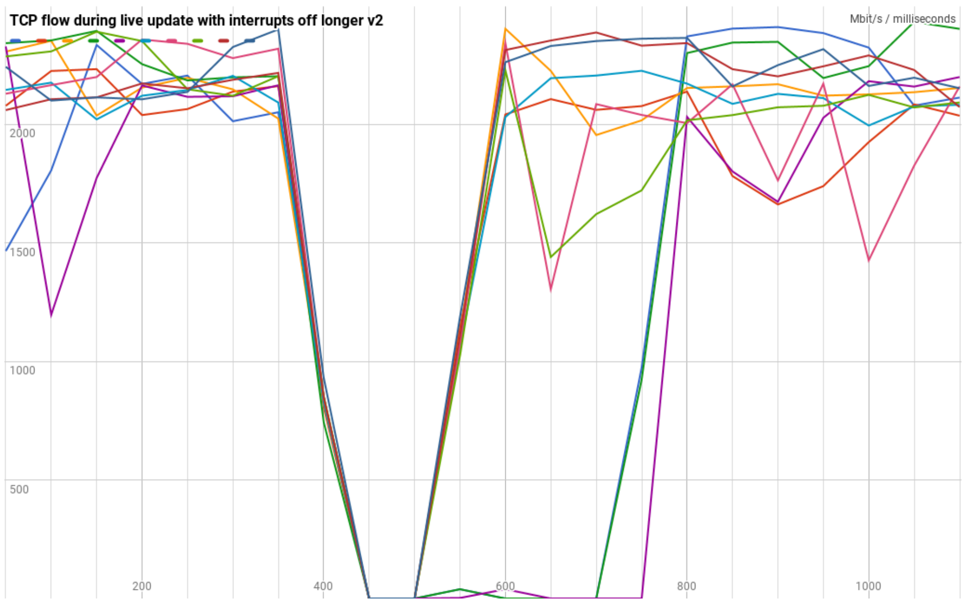


Figure 9.14: TCP flow during update with interrupts off earlier, samples every 50ms

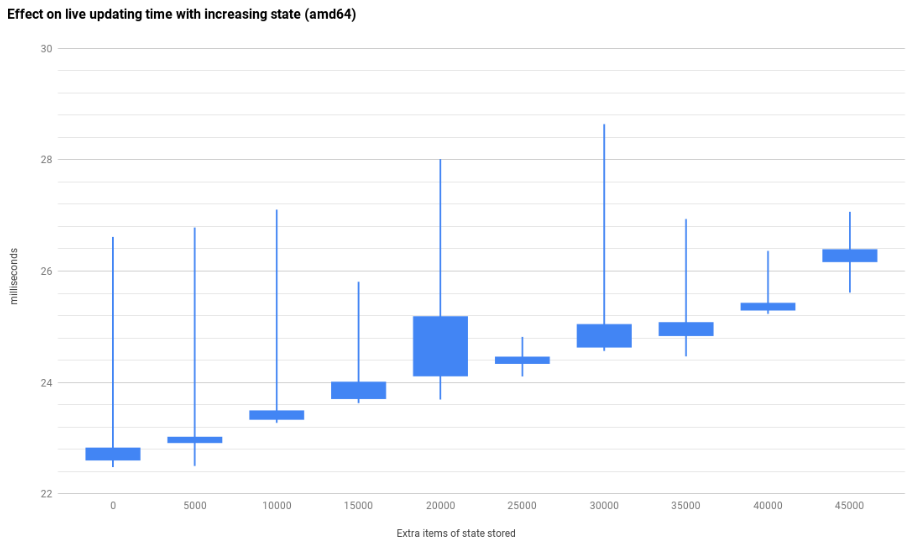


Figure 9.15: Effects of adding more and more state to the live updating process, running on qemu-KVM (64-bit)

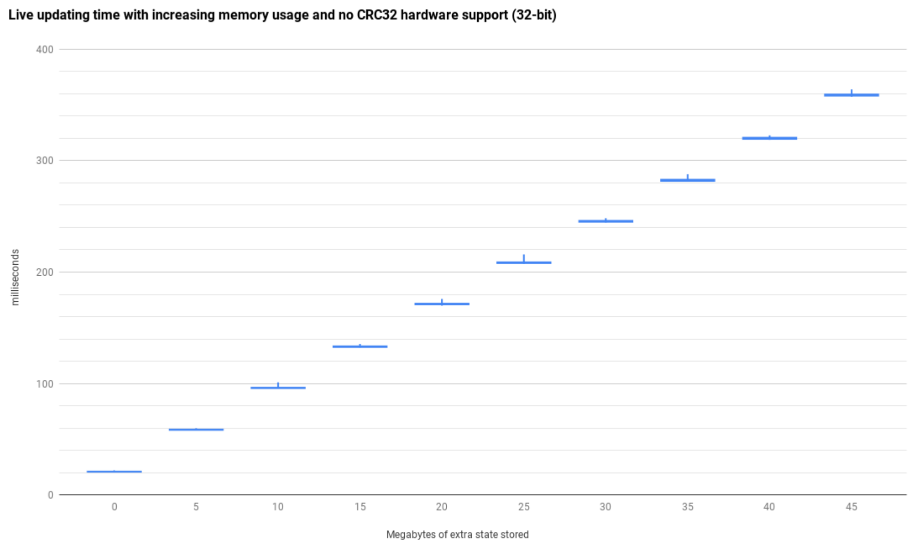


Figure 9.16: Effects of using more and more memory in the live updating process, without hardware CRC32 support

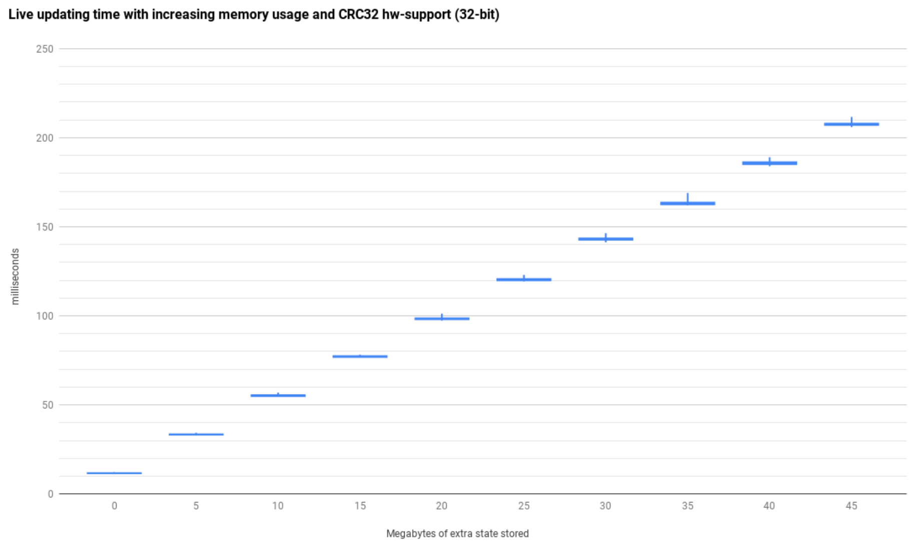


Figure 9.17: Effects of using more and more memory in the live updating process, with hardware CRC32 support

There is hardware support on CPUs with SSE4.2 support, and adding the algorithm is trivial, but in the virtual machine-space it is by no means a guarantee to have such support. The situation improved vastly with a faster hardware-assisted CRC32 algorithm. (See: 9.17)

Seeing as there is no guarantee for hardware-assisted CRC32 support, it may be necessary to allow system service writers to disable checksumming the whole of the state storage, and instead only checksum the header. In almost all cases there is not going to be a problem with the state after an update has happened, as someone has to explicitly overwrite it, which usually happens when the heap grows into the live update area. If LiveUpdate always verifies the state storage header, then even if the heap should grow into the state storage area, it may just be able to catch it happening.

Another more precise solution would be to always checksum up to 1MB of state, which would catch almost all cases where the heap has grown into the state storage area. For now, the LiveUpdate system will continue to checksum everything, so that systems have a chance to remain online should updates fail and state is lost.

9.2 LiveUpdate test service

The LiveUpdate Testing Service is the authoritative implementation of how to store and restore state. It performs many extra validation steps and consistency checks to continually confirm that the environment is sane before and after updates. It also handles exceptions and performs a rollback should a severe exception happen. The service also validates that failed rollbacks correctly resets the virtual machine, which causes the hypervisor to restart with the original provided image. The test service is intended to use all the various features of *LiveUpdate*.

The service accepts binaries used for rollbacks on TCP port 667. When a binary is received, any detectable faults will result in an attempt to live update to the rollback binary. No state is saved when doing this, as the state of the system could be such that saving state caused further faults. The first exception could have happened when storing state. If an exception occurs during rollback the CPU is reset, which will make the hypervisor intervene and load the original binary used when booting the first time.

9.3 IRC server

The IncludeOS IRC daemon is a hobby project that implements the IRC server protocol [15] and also partially the P10 Undernet server-to-server protocol. The server can be live updated, as proven by the authors continual development of the IRC server while it was still running. The server is still running as of this writing. It should be mentioned that should the server crash, it would have to have to be manually started again, and it would start using a very old image that is unlikely to support being live updated, due to having an older interface. It would, however, not fail

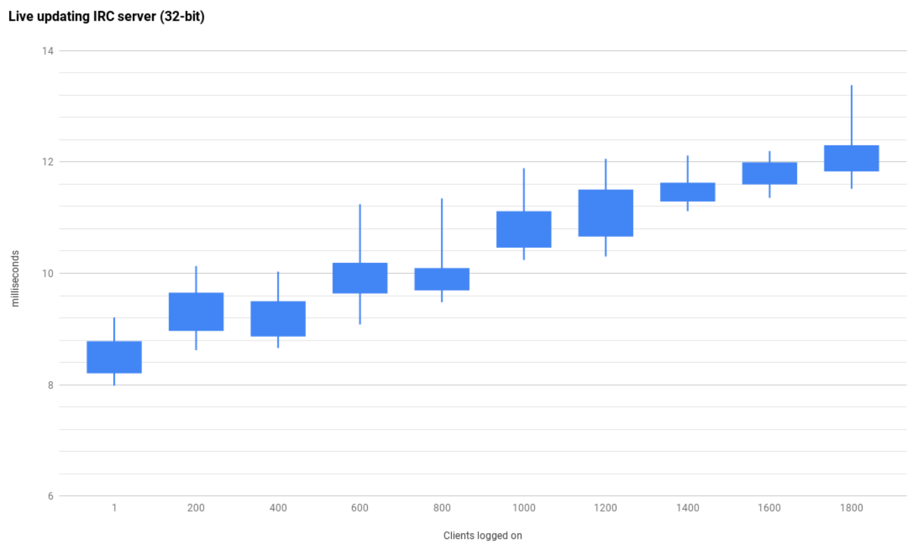


Figure 9.18: Live updating 32-bit IRC server with increasing number of connected clients

the actual live updating. It would simply not be able to retain any state, including doing a soft-reset due to checksum verification used as a sort of versioning tool.

The IRC server stores some private state, all clients, their connections and all the IRC network channels. Among the private state, the servers uptime is maintained across updates, and it has been running a few months (largely idle) as of writing this.

I have done some experiments live updating the IRC server with progressively more logged in clients.9.18 9.19 The graph shows that the number of active clients has an effect on update times, but that the effect is not very large. All clients on the IRC server have open channels, private state such as nickname, user and hostname as well as an open TCP connection. Each experiment consists of 30 samples each, and that is why there are error bars on each number of clients.

The clients are not interacting with each other during the tests. A possible experiment could be to generate an increasing level of traffic between the clients and measure that.

An interesting observation is that it takes 13ms to live update a server with 1800 clients on 64-bit, while only 12ms on 32-bit. To see whether it will remain like this no matter how many experiments are run, I performed some experiments on averages and did a T-test (see figure 9.20). The results show that there is a significant difference in means, as the T-test minimum is 0 and maximum is 0.000 000 006 480 515 434.

The test revealed to me that there was a fixed update time difference between 32-bit and 64-bit. It was later revealed to be the time it took to exit long mode and setup new page tables, as shown in an earlier experiment.

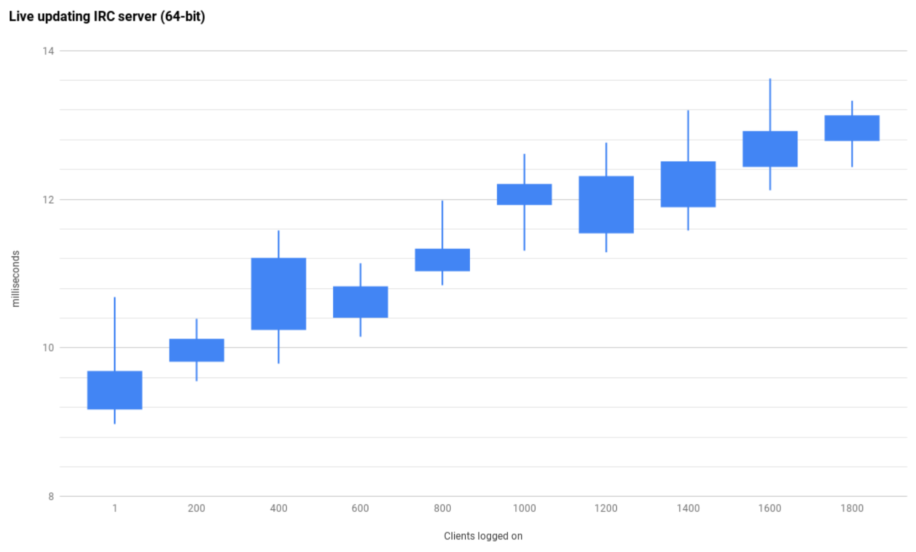


Figure 9.19: Live updating 64-bit IRC server with increasing number of connected clients

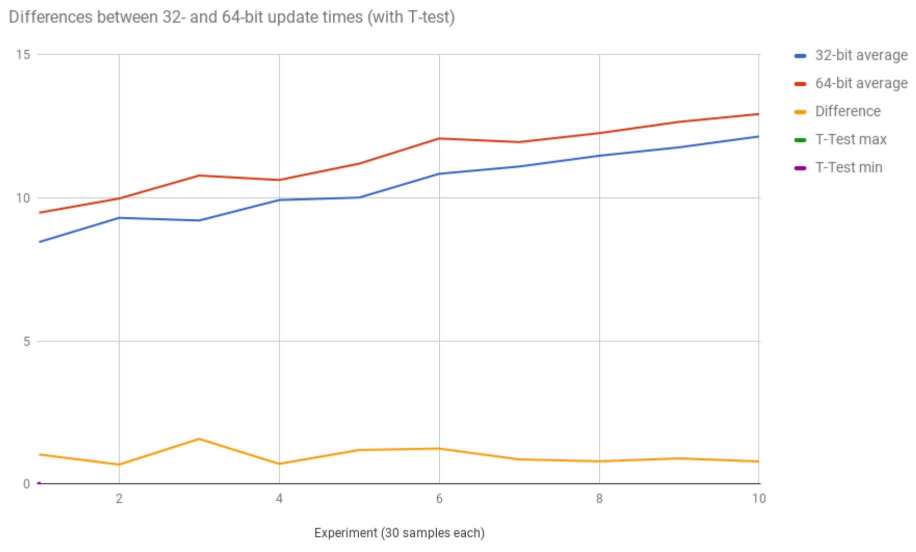


Figure 9.20: IRC server update averages

9.4 SQlite

Live updating SQlite ended up being much harder than live updating services made for the unikernel. SQlite is a self-contained embedded SQL database engine written in C, ported to IncludeOS by the author. SQlite uses writable static data (WSD) to globally initialize some of its internals. This has to be specially serialized and stored by LiveUpdate during data serialization phase. SQlite also has been configured to use a fixed area of memory outside of normal heap which it will use for all its allocation purposes. The area is then re-used after the live update unmodified. This means that we cannot update SQlite if it changes how it stores memory. Or how its database structures are stored. The update is near instant.

Not all features of SQlite are restored properly, nor are future features stored and restored, since they will each have to be written state storing/restoring code for individually. In particular, there are features in SQlite that may not apply to unikernels, which can be skipped.

An experiment was performed that shows the preservation of a small database across a live update. Unfortunately, the experiment cannot (yet) be expanded to do more complex database manipulation due to unknown issues with the internals of SQLite. SQLite is written in "old-style" C, and all the state isn't neatly placed in one place. With enough time, the large codebase can be properly scanned for global state which is then hoovered up and put in a state-collection function. The program had to be manually patched just to be able to do this simplified example. It does show that its definitely possible to live update any service while persisting a (single) SQLite database across the update.

Another attempt to solve this problem uses a feature called *OMIT_WSD*, which lets the programmer take full control of SQLites static data. Even with that control, the same issues appeared in SQLite, such as the inability to use the *COUNT* function. As such, the porting of SQLite was a limited success. It can preserve a database, but it has issues outside of the data itself that prevents proper usage. The experiment has been put on ice.

The actual experiment is constructed to to receive SQL commands over TCP. When a live update occurs, the connection should be preserved and operate as normal after the update, and that also goes for the database. The experiment would have shown that the database was consistent before and after the update.

9.5 Live migration

Live migrations typically need hypervisor and orchestration support. By using meaningful substitutes we can create an experiment that shows how to migrate a live service. The live migration is done in steps. Start new system and have it wait for state over TCP. Connect to new system from old system. Save state to memory area and then transfer it to new system. New system receives the state and then resumes execution with it. This is not the whole picture for an actual live migration as the experiment doesn't reroute the network and forward old connections to the new system. But, it is considered out of the scope of just showing how LiveUpdate can transfer state to a waiting system which then promptly resumes execution based on that state with some careful programming.

Output from the origin system:

```
================================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Live migration service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Network configured
                IP:  10.0.0.40
                Netmask:  255.255.255.0
                Gateway:  10.0.0.1
                DNS Server:  10.0.0.1
This is the origin service @ 52:54:00:12:34:56
Connecting to new system...
Connected to new instance, saving state...
Sending data...
     [ Kernel ] Soft shutdown signalled
     [ Kernel ] Stopping service
     [ Kernel ] Powering off
```

Output from the new system:

```
================================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Live migration service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Network configured
                IP:  10.0.0.43
                Netmask:  255.255.255.0
                Gateway:  10.0.0.1
                DNS Server:  10.0.0.1
This is the new service @ 12:34:56:12:34:56
Waiting for state...
Got connection from old instance
Received 52 bytes
Restored integer 1337
Success
     [ Kernel ] Soft shutdown signalled
     [ Kernel ] Stopping service
     [ Kernel ] Powering off
```

The new system received and verified the state received over TCP. The system was then restored with the state.

9.6 Deep freeze

Same as with live migration, but instead we will store the systems state to disk, bake the state into a new unikernel binary and then run that binary and verify that it resumes correctly, after some time has passed. Unfortunately, IncludeOS does not have a writable filesystem (nor a write support in the block drivers), so the state has to be transferred over TCP to the host system, stored to disk there, baked into the system binary and then the system can be booted up. At boot the system will check the filesystem for a file called */state.file*, and if found it will synchronously try to resume state from that data. Since this happens in the same place where a LiveUpdate resume would ordinarily happen, it will have the same effect on the system.

The origin system is saving its state (which would ordinarily go to disk or sent via a hypercall) and transferring it to the host system:

```
========================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Live migration service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [ Memdisk ] Initializing
        [ FAT ] Initializing FAT12 filesystem
                [ofs=0  size=3 (1536 bytes)]

    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Network configured
                IP:  10.0.0.40
                Netmask:  255.255.255.0
                Gateway:  10.0.0.1
                DNS Server:  10.0.0.1
This is the origin service @ 52:54:00:12:34:56
Connecting to new system...
Connected to new instance, saving state...
Sending data...
     [ Kernel ] Soft shutdown signalled
     [ Kernel ] Stopping service
     [ Kernel ] Powering off
```

The origin system is then turning itself off as it has nothing to do.

At some later point in time, the system is then booted back up. Without hypervisor support the state has to be baked into the read-only filesystem.

```
========================================================================
 IncludeOS v0.9.3-3712-g4a0fcfdf-dirty (x86_64 / 64-bit)
 +--> Running [ Live migration service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [ Memdisk ] Initializing
        [ FAT ] Initializing FAT12 filesystem
```

```
                  [ofs=0  size=5 (2560 bytes)]

Reading state from filesystem...
Restored integer 1337
Success
```

The system found and verified the state. The system was then resumed with the state.

9.7 Mender client

LiveUpdate was featured in a Mender client [28] demo, appearing in Linux Embedded Con 2017 in Portland, SA. For the presentation, LiveUpdate was the final stage in a process where an update image was securely transferred from a mender GUI to the mender client, and then given to LiveUpdate which performed the live updating. The live demo showed an IncludeOS webserver with a bug, a CPU-usage graph randomly spiking when it shouldn't (by design). The presenter then live updated a fixed Web-server image (containing mender client, live update and a Web-server) and then as if nothing had happened the bug was patched out, while the system was running.

9.8 Starbase

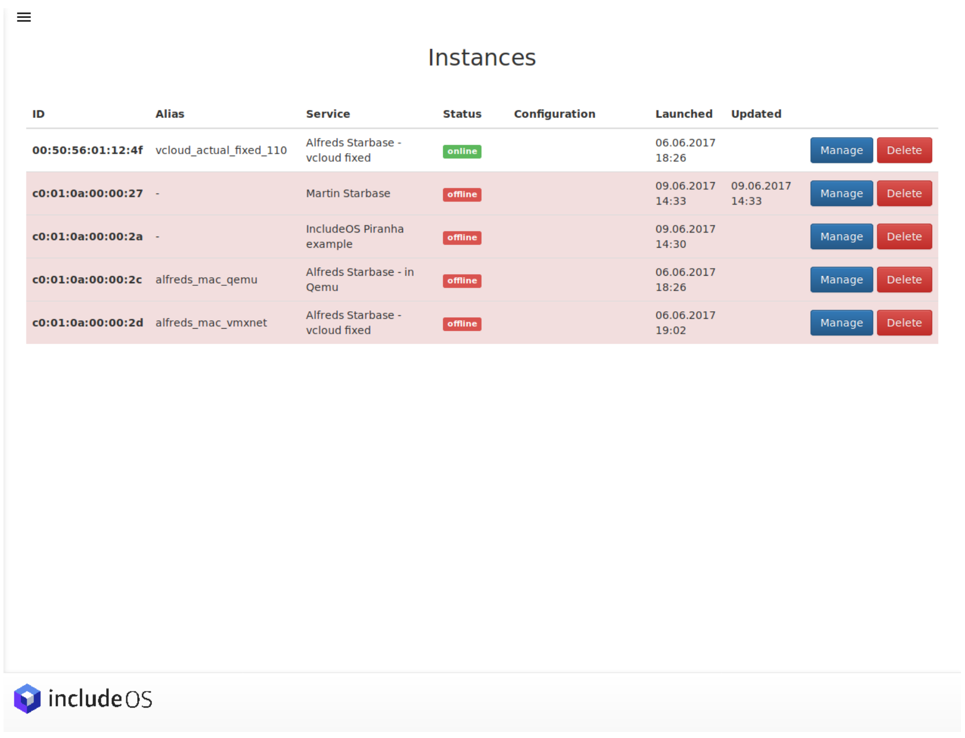
A more complex use of LiveUpdate was featured at a small VMware conference at BaseFarm in Oslo. There, IncludeOS team members showed how using service base images that only had a networking stack and LiveUpdate could be remotely updated to add and remove functionality. During a demo, it was shown how a load balancer could be hard-configured from load balancing 3 nodes and increasing it to 5. This was done by building a new load balancer image remotely and then sending it with a built-in configuration over a WebSocket connection. This image was then live updated into the service, and the service now had new functionality and/or configuration. The service reported its status uninterrupted to a remote tracker (called a *Mothership*) during the demo. The total update time was in the span of milliseconds after transmitting each update asynchronously. See figure 9.21.

Starbases are the individual nodes capable of receiving updates that transform their function, using LiveUpdate.

Starbase can be a large part of the orchestration for serverless FaaS with LiveUpdate. It is, for now, intended to allow users to deploy binaries to starbases (nodes) at will. Usually users only have access to nodes they themselves deployed anyways, so serverless FaaS makes less sense in this scenario.

Example:

Serverless FaaS would require an opaque infrastructure (a cloud essentially) with the proper interfaces to deploy code. It would also need


Instances

ID	Alias	Service	Status	Configuration	Launched	Updated		
00:50:56:01:12:4f	vcloud_actual_fixed_110	Alfreds Starbase - vcloud fixed	online		06.06.2017 18:26		Manage	Delete
c0:01:0a:00:00:27	-	Martin Starbase	offline		09.06.2017 14:33	09.06.2017 14:33	Manage	Delete
c0:01:0a:00:00:2a	-	IncludeOS Piranha example	offline		09.06.2017 14:30		Manage	Delete
c0:01:0a:00:00:2c	alfreds_mac_qemu	Alfreds Starbase - in Qemu	offline		06.06.2017 18:26		Manage	Delete
c0:01:0a:00:00:2d	alfreds_mac_vmxnet	Alfreds Starbase - vcloud fixed	offline		06.06.2017 19:02		Manage	Delete

Figure 9.21: Screenshot of work-in-progress Mothership front-end

protections against abuse, such as blocking all outgoing ports and such. The mothership could be extended to accept an URL and the function code, which it then wraps around a basic service with networking and file-system, builds and prepares for deployment. When an outside user accesses the URL the mothership then deploys the function to a free node, load balancers forward the users request to this node, the function is executed, and the response is formed directly from the node itself. This means the first request has a fixed time delay: transmission time of function binary (or bytecode), live update time, function processing time and response transmission time.

9.9 uKVM - A unikernel monitor

IBM researchers ported uKVM/Solo5 to IncludeOS. uKVM is a modern minimalistic unikernel hypervisor (without all the PC-platform stuff). Solo5 is their library that implements the uKVM hypercall interface in C. Experiments were done to measure boot times. Unfortunately, uKVM has an ELF loader and appropriately marks guest pages with access rights based on ELF section attributes. This means LiveUpdate can't modify memory at will, such as overwriting one kernel with another, even if its exactly the same kernel. It simply won't have the privileges.

At request, the uKVM team added a *kexec*-like [37] command *solo5_exec* to their *uKVM-includeos64* branch which enables live updating with uKVM through a hypercall. An experiment was performed to measure boot time

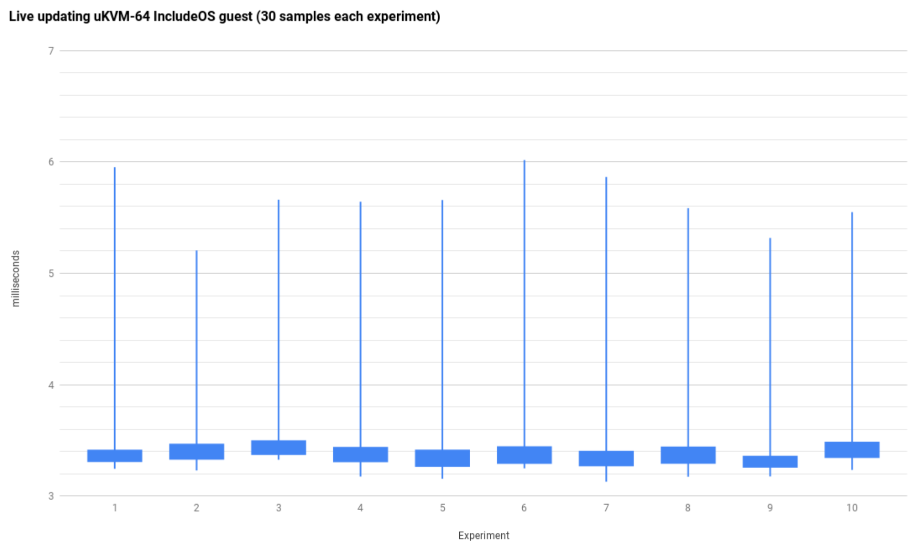


Figure 9.22: uKVM-64 IncludeOS-branch update times

on this specialized hypervisor. (9.22) The experiment shows that update times are greatly improved, going as low as 3.2 milliseconds. Hardware CRC32 was enabled.

The experiment was run as if the environment was unprivileged, showing that LiveUpdate can work in unprivileged guest operating systems given that there is a hypercall (or equivalent) that can hotswap the kernel for the guest (and consequently re-apply access bits).

The total update times being 3.2ms is not too surprising. There is still lots of checksumming going on, and IncludeOS is still doing much of it internally. uKVM sets up proper paging so some of the IncludeOS sanity checks are redundant. Another experiment was done where the extraneous checksumming was disabled. Keep in mind that symbols and LiveUpdate consistency checks (including checksumming) is still enabled.

Another factor that reduces the live updating time on uKVM further is the omission of the Long-mode exit (LME) needed on other platforms. See also figure: 8.1

The new experiment (9.23) shows that the update times go as low as 2.8ms consistently. I noticed again that there were regular spikes in each separate experiment of exactly one sample. I suspect this is because the first time solo5 updates an application it stores the executable in */tmp/randomized_name*, and writing to disk incurs a time cost. The reasoning behind this is most likely that uKVM should be able to reboot into the updated executable if something happens, unlikely with a normal hypervisor and LiveUpdate where any CPU reset will have the original (first) image booted. Considering that we don't know whether an executable is working or not until we actually boot it, this is both a blessing and a curse. It's a blessing because the updated binaries are persisted and become the de-facto new guest base image, and a curse because should we

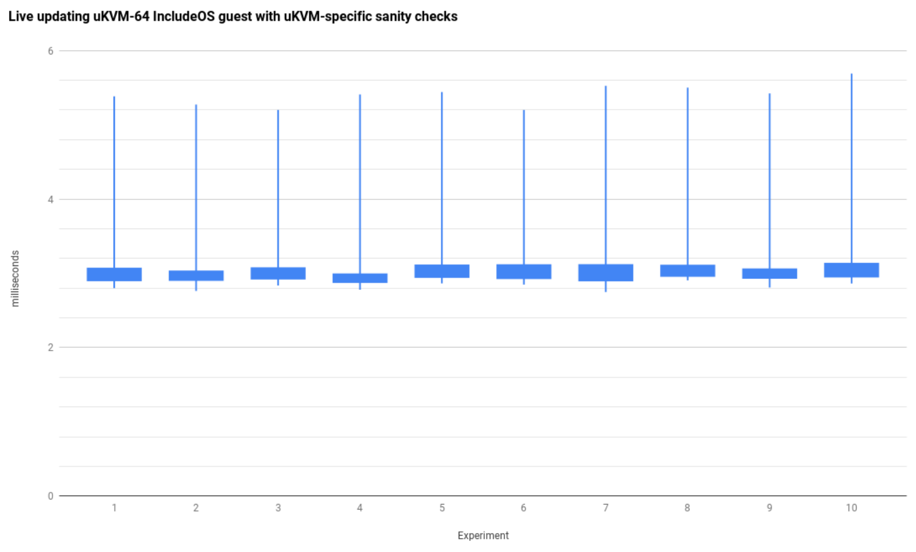


Figure 9.23: uKVM-64 IncludeOS-branch update times with only uKVM-specific sanity checks enabled

update a faulty binary we would never be able to boot again, because there is nothing to hard-reset to. This feature in uKVM is rather new and so it is not unexpected to encounter unresolved problems like this.

The last two experiments show the minimal possible update time with uKVM. By removing the update binary from the equation we can see (figure 9.24 9.25) that the update times can go as low as 400 microseconds. This should meet real-time requirements.

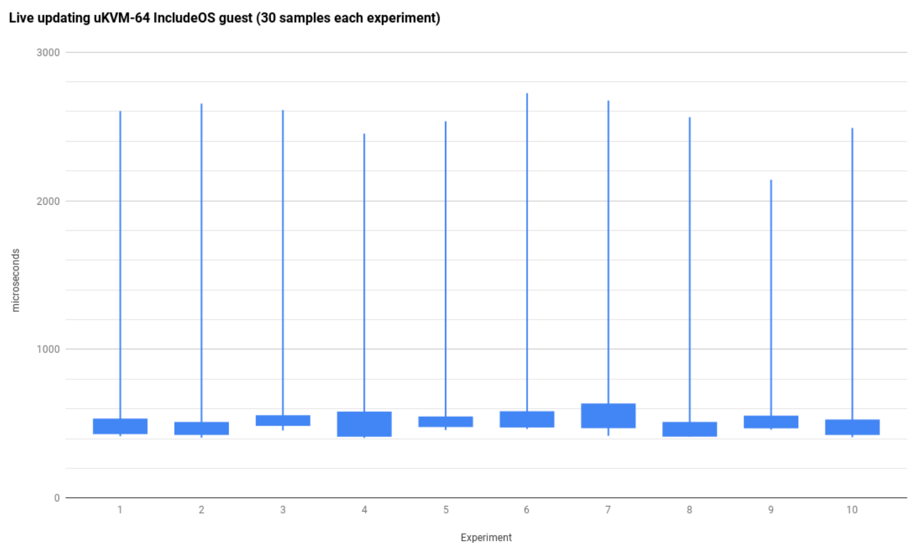


Figure 9.24: uKVM-64 IncludeOS-branch update times with no update binary preserved across updates

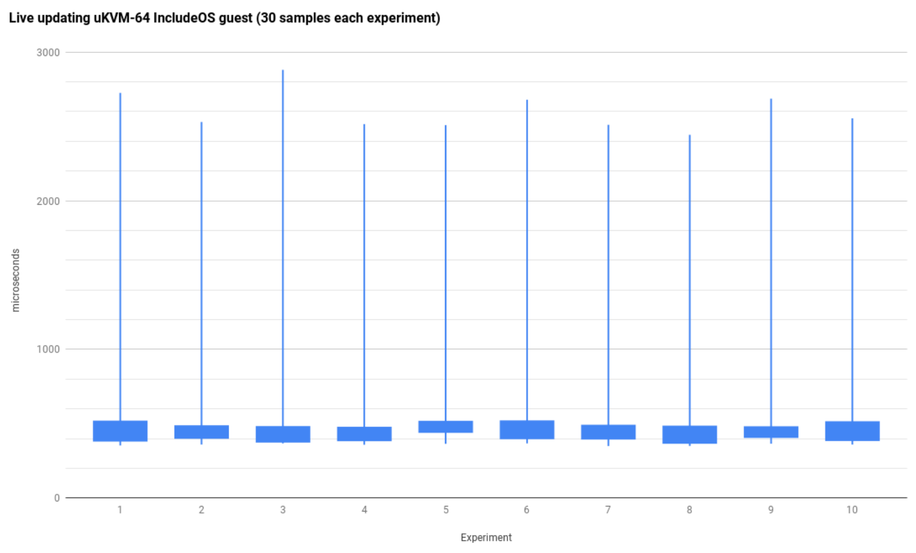


Figure 9.25: uKVM-64 IncludeOS-branch update times with no update binary preserved across updates

Part IV

Discussion

Chapter 10

Findings

It has been shown that LiveUpdate makes it possible to live update a library operating system while maintaining selected state. Furthermore, we have seen that LiveUpdate was used in a series of services, such as an IRC server, where it successfully stored and restored the state of thousands of TCP connections and maintained program state and consistency across updates. It has been observed that the live updating process can range from 400 microseconds and up to 20ms for smaller programs, depending on hardware configuration, how much state is stored, the hypervisor and scheduling.

We have seen that when storing lots of individual bits of state, the update time is not affected greatly. Adding 45000 individual bits of state only added 3ms to the overall update time.

When adding raw data, the update time was greatly affected, due to the time-cost of checksumming. Without hardware CRC32 it took 700ms to live update only 10MB of raw data as a single piece of state. When hardware CRC32 was enabled the live update time for the same amount (10MB) was reduced to 60ms. A 10x improvement, but only available with the SSE4.2 CPU instruction set.

We have seen that LiveUpdate can store and restore thousands of TCP connections. Active connections in the middle of transmissions can lose packets during the update which makes the TCP enter congestion control.

It has been shown how one could serialize a running environments state, migrate it to a remote host, and then resume operation. This will require help from hypervisors or a small unikernel service that sits idle waiting to receive an update. The experiment had the origin system save and transmit the state over the network and then shutdown. The new system received the state over TCP and resumed from it. In a separate experiment deep freeze was implemented using the read-only filesystem. A new binary was created with the state baked in. When the service was booted up it would find the state in the filesystem and resume from it. In the cases of live migration and deep freeze the state transfer was minimal, going as low as 52 bytes. In comparison a VirtualBox VM guest snapshot on the authors computer was using 1 GiB.

The uKVM experiment showing that live update could work on a specialized hypervisor (a unikernel monitor) was important for LiveUpdate, as it showed that not only does it work on multiple platforms, it also works on hypervisors in at least a semi-unprivileged environment. The uKVM environment is not completely unprivileged, but the experiments were run as if it was.

Part V

Conclusion and Future work

10.1 Conclusion

It has been shown that LiveUpdate can calculate the effective length of ELF binaries and using that with verification of the ELF header of the update binaries it can be reasonably sure that the binary is complete. Any redundancy checks would have to be done outside of the LiveUpdate environment as ELF binaries in general do not have redundancy checks built in.

By storing the data in memory during updates, systems can be live updated without persisting state or data. It has been shown that a variety of state, including TCP-connections, has been preserved across in-memory live updates.

Using template programming, user-provided unique identifiers, length checks and support for variable-length structures the programmer can be reasonably sure that the data he puts in is the data he gets out. Users do not get full type-safety without compiler assistance or code pre-/post-processing.

Rollback support adds some protection against catastrophically failed updates. Rollbacks are automatically performed when a CPU exception happens, making it possible for the system to recover some state by downgrading. Systems may be too far gone during panics to be able to recover, in which case the system is rebooted.

The temporary update storage area used between updates is zeroed out after completion. This makes sure LiveUpdate leaves no trace during normal operation.

LiveUpdate performs sanity and redundancy checks during and after updates. This increases the update time somewhat, but will detect state consistency issues, such as the accidental overwriting of the temporary update storage area.

LiveUpdate can update a running IncludeOS unikernel in as little as 5 milliseconds with the right hardware instruction set extensions, replacing it with any other kernel. It can go as low as 400 microseconds when live updating on the uKVM unikernel monitor, meeting strict real-time requirements.

LiveUpdate takes the premise of unikernels and builds live updating, live migration, deep freeze and serverless function-as-a-service on top of them. The lack of tooling support for unikernels and orchestration makes live migration and deep freeze experiment-only for now.

10.2 Future work

- Create a more finely tuned rollback system that deals with problems across architectures, update versioning and how to measure the consistency of an operating system before and after live updating.
- Investigate soft-reset optimizations that could make the LiveUpdate process time from start to end faster. Most of the work will be in storing and restoring hardware state to save expensive calls hardware

calls (which are even more expensive traps into the hypervisor in virtual machines).

- Build orchestration around seamless live-migrations of unikernels using the base concepts introduced in the live migration experiment.
- Build orchestration or hypervisor support around freezing and unfreezing of systems, based on the deep freeze experiment.
- Extend the LiveUpdate hotswapping function to load any ELF binary. In other words, implement a full-blown ELF loader, which allows LiveUpdate to boot into a Linux kernel, or any other non-flat operating system kernel.
- By loading two separate kernels into memory, where the first kernel retains CPU exceptions as well as some ability to act as watchdog. A sort of *shadow kernel*.
- Add stricter/full runtime type-safety checks to enforce type-safety across updates using compiler assistance.

10.3 Acknowledgements

I would like to thank Alfred Bratterud for his outstanding help, who has gone out of his way to help me. I (officially) owe you one. I cannot thank Rico Antonio Felix enough for proofreading this thesis, on his own volition. I would also like to thank Paal Engelstad for supervising the thesis and the IncludeOS team for creating the best unikernel, where it all started.

Bibliography

[1] 'A Technique for Dynamic Updating of Java Software'. In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. ICSM '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 649–. ISBN: 0-7695-1819-2. URL: http://dl.acm.org/citation.cfm?id=876882.879757.

[2] Eric Biederman Andy Pfiffer. *kexec for 2.5.73 available*. 2002. URL: https://lwn.net/Articles/37573/ (visited on 20/06/2017).

[3] A. Bratterud et al. 'IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services'. In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[4] *C++ Core Guidelines*. 2017. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[5] Jonathan Corbet. *A rough patch for live patching*. 2015. URL: https://lwn.net/Articles/634649/ (visited on 20/06/2017).

[6] Jonathan Corbet. *Clockevents and dyntick*. 2007. URL: https://lwn.net/Articles/223185/ (visited on 14/07/2017).

[7] Jonathan Corbet. *Supervisor mode access prevention*. 2012. URL: https://lwn.net/Articles/517475/ (visited on 26/07/2017).

[8] Jake Edge. *State of the Kernel Self Protection Project*. 2016. URL: https://lwn.net/Articles/698827/ (visited on 26/07/2017).

[9] C. M. Hayden et al. 'State transfer for clear and efficient runtime updates'. In: *2011 IEEE 27th International Conference on Data Engineering Workshops*. Apr. 2011, pp. 179–184. DOI: 10.1109/ICDEW.2011.5767632.

[10] Christopher M. Hayden et al. 'Kitsune: Efficient, General-purpose Dynamic Software Updating for C'. In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 249–264. ISSN: 0362-1340. DOI: 10.1145/2398857.2384635. URL: http://doi.acm.org/10.1145/2398857.2384635.

[11] Scott Hendrickson et al. 'Serverless Computation with openLambda'. In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 33–39. URL: http://dl.acm.org/citation.cfm?id=3027041.3027047.

[12] Michael Hicks and Scott Nettles. 'Dynamic Software Updating'. In: *ACM Trans. Program. Lang. Syst.* 27.6 (Nov. 2005), pp. 1049–1096. ISSN: 0164-0925. DOI: 10.1145/1108970.1108971. URL: http://doi.acm.org/10.1145/1108970.1108971.

[13] Seth Jennings Josh Poimboeuf. *Introducing kpatch: Dynamic Kernel Patching*. 2014. URL: http://rhelblog.redhat.com/2014/02/26/kpatch/ (visited on 20/06/2017).

[14] Seth Jennings Josh Poimboeuf. *kpatch - dynamic kernel patching*. 2017. URL: https://github.com/dynup/kpatch (visited on 20/06/2017).

[15] C. Kalt. *Internet Relay Chat: Client Protocol*. RFC 2812. Apr. 2000.

[16] Tom Lendacky. *x86: Secure Memory Encryption (AMD)*. 2017. URL: https://lwn.net/Articles/714740/ (visited on 26/07/2017).

[17] Chuanpeng Li, Chen Ding and Kai Shen. 'Quantifying the Cost of Context Switch'. In: *Proceedings of the 2007 Workshop on Experimental Computer Science*. ExpCS '07. San Diego, California: ACM, 2007. ISBN: 978-1-59593-751-3. DOI: 10.1145/1281700.1281702. URL: http://doi.acm.org/10.1145/1281700.1281702.

[18] MultiMedia LLC. *Linux Kernel Documentation KVM CPUID bits*. 2010. URL: https://www.kernel.org/doc/Documentation/virtual/kvm/cpuid.txt (visited on 14/05/2017).

[19] C. Lonvick. *The BSD syslog Protocol*. RFC 3164. Aug. 2001.

[20] Anil Madhavapeddy. *Unikernels - Rethinking Cloud Infrastructure*. 2017. URL: http://unikernel.org/ (visited on 28/07/2017).

[21] Anil Madhavapeddy and David J. Scott. 'Unikernels: Rise of the Virtual Library Operating System'. In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: http://doi.acm.org/10.1145/2557963.2566628.

[22] Anil Madhavapeddy et al. 'Unikernels: Library Operating Systems for the Cloud'. In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: http://doi.acm.org/10.1145/2499368.2451167.

[23] Marek Majkowski. *Kernel Bypass*. 2015. URL: https://blog.cloudflare.com/kernel-bypass/ (visited on 28/07/2017).

[24] matildah. *Single address spaces: design flaw or feature?* 2016. URL: https://matildah.github.io/posts/2016-01-30-unikernel-security.html (visited on 28/07/2017).

[25] *Multiboot Specification version 0.6.96*. 2017. URL: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[26] Iulian Neamtiu et al. 'Practical Dynamic Software Updating for C'. In: *SIGPLAN Not.* 41.6 (June 2006), pp. 72–83. ISSN: 0362-1340. DOI: 10.1145/1133255.1133991. URL: http://doi.acm.org/10.1145/1133255.1133991.

[27] *On rump kernels and the Rumprun unikernel*. 2015. URL: https://blog.xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/.

[28] *Open source OTA software*. 2017. URL: https://mender.io/.

[29] Gerald J. Popek and Robert P. Goldberg. 'Formal Requirements for Virtualizable Third Generation Architectures'. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073.

[30] S. Potter and J. Nieh. 'AutoPod: Unscheduled System Updates with Zero Data Loss'. In: *Second International Conference on Autonomic Computing (ICAC'05)*. June 2005, pp. 367–368. DOI: 10.1109/ICAC.2005.16.

[31] Theo de Raadt. *Kernel relinking status*. 2017. URL: http://undeadly.org/cgi?action=article&sid=20170701170044 (visited on 28/07/2017).

[32] *RedHat Support Chapter 8. Virtualization*. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/6.4_Release_Notes/virtualization.html. Accessed: 2017-05-14.

[33] Rusty Russell. 'Virtio: Towards a De-facto Standard for Virtual I/O Devices'. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 95–103. ISSN: 0163-5980. DOI: 10.1145/1400097.1400108. URL: http://doi.acm.org/10.1145/1400097.1400108.

[34] Benoit Sigoure. *How long does it take to make a context switch?* 2010. URL: http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html (visited on 28/07/2017).

[35] Brijesh Singh. *x86: Secure Encrypted Virtualization (AMD)*. 2017. URL: https://lwn.net/Articles/716165/ (visited on 26/07/2017).

[36] Gareth Stoyle et al. 'Mutatis Mutandis: Safe and Predictable Dynamic Software Updating'. In: *ACM Trans. Program. Lang. Syst.* 29.4 (Aug. 2007). ISSN: 0164-0925. DOI: 10.1145/1255450.1255455. URL: http://doi.acm.org/10.1145/1255450.1255455.

[37] Unknown. *kexec - directly boot into a new kernel*. 2017. URL: https://linux.die.net/man/8/kexec (visited on 20/06/2017).

[38] Dan Williams and Ricardo Koller. 'Unikernel Monitors: Extending Minimalism Outside of the Box'. In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 71–76. URL: http://dl.acm.org/citation.cfm?id=3027041.3027053.

Appendix A

LiveUpdate API

```
/**
 * Master thesis
 * by Alf-Andre Walla 2016-2017
 *
**/
#pragma once
#ifndef LIVEUPDATE_HEADER_HPP
#define LIVEUPDATE_HEADER_HPP

#include <net/tcp/connection.hpp>
#include <delegate>
#include <string>
#include <vector>
struct storage_entry;
struct storage_header;

namespace liu
{
struct Storage;
struct Restore;
typedef std::vector<char> buffer_t;

/**
 * The beginning and the end of the LiveUpdate process is the begin() and resume() fun
 * begin() is called with a provided fixed memory location for where to store all seri
 * and after an update is_resumable, with the same fixed memory location, will return
 * resume() can then be called with this same location, and it will call handlers for
 * unless no such handler is registered, in which case it just calls the default handl
 * to the call to resume(). The call to resume returns true if everything went well.
**/
struct LiveUpdate
{
  // The buffer_t parameter is the update blob (the new kernel) and can be null.
  // If the parameter is null, you can assume that it's currently not a live update.
  typedef delegate<void(Storage&, const buffer_t*)> storage_func;
  typedef delegate<void(Restore&)> resume_func;

  // Start a live update process, storing all user-defined data
  // at @location, which can then be resumed by the future service after updat
  static void begin(void* location, buffer_t blob, storage_func = nullptr);

  // In the event that LiveUpdate::begin() fails,
  // call this function in the C++ exception handler:
  static void restore_environment();

  // Only store user data, as if there was a live update process
  // Throws exception if process or sanity checks fail
  static size_t store(void* location, storage_func);

  // Returns true if there is stored data from before at @location.
  // It performs an extensive validation process to make sure the data is
  // complete and consistent
  static bool is_resumable(void* location);

  // Register a user-defined handler for what to do with @id from storage
  static void on_resume(uint16_t id, resume_func custom_handler);

  // Attempt to restore existing stored entries from fixed location.
  // Returns false if there was nothing there. or if the process failed
  // to be sure that only failure can return false, use is_resumable first
  static bool resume(void* location, resume_func default_handler);

  // When explicitly resuming from heap, heap overrun checks are disabled
  static bool resume_from_heap(void* location, resume_func default_handler);

  // Retrieve the recorded length, in bytes, of a valid storage area
  // Throws std::runtime_error when something bad happens
  // Never returns zero
  static size_t stored_data_length(void* location);

  // Set location of known good blob to rollback to if something happens
  static void set_rollback_blob(const void*, size_t) noexcept;
  // Returns true if a backup rollback blob has been set
  static bool has_rollback_blob() noexcept;
  // Immediately start a rollback, not saving any state other than
  // performing a soft-reset to reduce downtime to a minimum
  // Never returns, and upon failure intentionally hard-resets the OS
  static void rollback_now(const char* reason);
};

////////////////////////////////////////////////////////////////////////////////

```

```
// IMPORTANT:
// Calls to resume can fail even if is_resumable validates everything correctly.
// this is because when the user restores all the saved data, it could grow into
// the storage area used by liveupdate, if enough data is stored, and corrupt it.
// All failures are of type std::runtime_error. Make sure to give VM enough RAM!

////////////////////////////////////////////////////////////////////////////////

/**
 * The Storage object is passed to the user from the handler given to the
 * call to begin(), starting the liveupdate process. When the handler is
 * called the system is ready to serialize data into the given @location.
 * By using the various add_* functions, the user stores data with @uid
 * as a marker to be able to recognize the object when restoring data.
 * IDs don't have to have specific values, and the user is free to use any value.
 *
 * When using the add() function, the type cannot be verified on the other side,
 * simply because type_info isn't guaranteed to work across updates. A new update
 * could have been compiled with a different compiler.
 *
**/
struct Storage
{
  typedef net::tcp::Connection_ptr Connection_ptr;
  typedef uint16_t uid;

  template <typename T>
  inline void add(uid, const T& type);

  // storing as int saves some storage space compared to all the other types
  void add_int    (uid, int value);
  void add_string(uid, const std::string&);
  void add_buffer(uid, const buffer_t&);
  void add_buffer(uid, const void*, size_t length);
  // store vectors of PODs or std::string
  template <typename T>
  inline void add_vector(uid, const std::vector<T>& vector);
  // store a TCP connection
  void add_connection(uid, Connection_ptr);

  Storage(storage_header& sh) : hdr(sh) {}
  void add_vector (uid, const void*, size_t count, size_t element_size);
  void add_string_vector (uid, const std::vector<std::string>&);

  // markers are used to delineate the end of variable-length structures
  void put_marker(uid);

private:
  storage_header& hdr;
};

/**
 * A Restore object is given to the user by restore handlers,
 * during the resume() process. The user should know what type
 * each id is, and call the correct as_* function. The object
 * will still be validated, and an error is thrown if there was
 * a type mismatch in most cases.
 *
 * It's possible to restore many objects from the same handler by
 * using go_next(). In that way, a user can restore complicated objects
 * completely without leaving the handler. go_next() will throw if there
 * is no next object to go to.
 *
**/
struct Restore
{
  typedef net::tcp::Connection_ptr Connection_ptr;

  bool  is_end()    const noexcept;
  bool  is_int()    const noexcept;
  bool  is_marker() const noexcept;
  int            as_int()    const;
  std::string    as_string() const;
  buffer_t       as_buffer() const;
  Connection_ptr as_tcp_connection(net::TCP&) const;

  template <typename S>
  inline const S& as_type() const;

  template <typename T>
  inline std::vector<T> as_vector() const;

  int16_t     get_type() const noexcept;
  uint16_t    get_id()   const noexcept;
  int         length()   const noexcept;
  const void* data()     const noexcept;

  uint16_t    next_id()  const noexcept;
  // go to the next storage entry
  void        go_next();

  // go *past* the first marker found (or end reached)
  // if a marker is found, the markers id is returned
  // if the end is reached, 0 is returned
  uint16_t pop_marker();
  // go *past* the first marker found (or end reached)
  // if a marker is found, verify that @id matches
  // if the end is reached, @id is not used
  void pop_marker(uint16_t id);

  // cancel and exit state restoration process
  // NOTE: resume() will still return true
  void cancel();  // pseudo: "while (!is_end()) go_next()"

  // NOTE:
  // it is safe to immediately use is_end() after any call to:
  // go_next(), pop_marker(), pop_marker(uint16_t), cancel()

  Restore(storage_entry*& ptr) : ent(ptr) {}
  Restore(const Restore&);
private:
  const void* get_segment(size_t, size_t&) const;
  std::vector<std::string> rebuild_string_vector() const;
  storage_entry*& ent;
};

/// various inline functions

template <typename S>
inline const S& Restore::as_type() const {
  if (sizeof(S) != length()) {
    throw std::runtime_error("Mismatching length for id " + std::to_string(get_id()));
  }
  return *reinterpret_cast<const S*> (data());
}
template <typename T>
inline std::vector<T> Restore::as_vector() const
{
  size_t count = 0;
  auto*  first = (T*) get_segment(sizeof(T), count);
  return std::vector<T> (first, first + count);
}
template <>
inline std::vector<std::string> Restore::as_vector() const
{
  return rebuild_string_vector();
}

template <typename T>
inline void Storage::add(uid id, const T& thing)
{
  add_buffer(id, &thing, sizeof(T));
}
template <typename T>
```

```
inline void Storage::add_vector(uid id, const std::vector<T>& vector)
{
  add_vector(id, vector.data(), vector.size(), sizeof(T));
}
template <>
inline void Storage::add_vector(uid id, const std::vector<std::string>& vecto
{
  add_string_vector(id, vector);
}

} // liu

#endif
```