Unikernels
No OS? No problem!

Kevin Sapper
Hochschule RheinMain
Unter den Eichen 5
Wiesbaden, Germany
kevin.b.sapper@student.hs-rm.de

ABSTRACT
Unikernels aim to reduce the layers and dependencies modern operating systems force onto applications. The concept is similar to library operating systems from the 90s but is leveraging hypervisor technology for hardware independence.

Categories and Subject Descriptors
D.4 [Software]: OPERATING SYSTEMS

Keywords
microkernel, cloud, library operating system, virtualization

1. INTRODUCTION
In recent years cloud computing made it possible to rent computing resources in possibly multiple large data centers and from possibly multiple competing providers. The enabling technology for the rise of cloud computing is operating-system virtualization which allows multiplexing of virtual machines (VMs) on a set of physical hardware. A VM usually represents a self-contained computer which boots and runs a standard full operating-system like Linux or Windows. A key advantage of this approach is the ability to run unmodified applications as if they were executing on a physical machine. Furthermore, those VMs can be centrally backed up and migrated and/or duplicated onto different physical machines. This allows for applications, that are installed on physical hosts, to be packed on fewer hosts without modifying or recompiling them. (see [4])

2. OS-VIRTUALIZATION
Despite the fact that VMs allow multi-user and multi-purpose applications and services, virtualization and cheap hardware created situations where most deployed VMs only perform single functions such as a database or a web server. This shift towards single-purpose VMs shows how easy it has become to create and deploy new VMs. (see [4])

2.1 Problems
Operating-system virtualization is obviously very useful, but at the same time it adds another layer to the already highly layered software stack that modern applications implicitly are forced to use. These layers include irrelevant optimizations (e.g. spinning disk algorithms on SSDs), backwards compatibility (e.g. POSIX), userspace processes/threads and code-runtimes (e.g. Java Virtual Machine). One obvious issue with many layers is performance. An approach to shrink the virtualization layer of VMs are OS containers like FreeBSD Jails and Linux Containers. These usually abstract at the operating-system level as opposed to the hardware level of VMs. Hence, they virtualize the userspace instead of the physical machine. Even though this architecture is somewhat lighter and improves performance, it doesn’t remove the extra layer added on top. Also, both VMs and containers provide large attack surfaces which can lead to severe system compromising. (see [4])

Another problem with virtualization is strong isolation of multi-tenant application to support the distribution of application and system code. This is especially critical on public clouds as VMs of different customers might be running on the same physical hardware. The limitations therefore are current operating systems, which are designed to be general-purpose in order to solve problems for a wide-audience. For example Linux runs on low-power mobile devices as well as high-end servers in vast data centers. Changing these principles to cater one class of users is not acceptable. (see [4])

2.2 Half the Solution
To radically reduce the amount of layers to improve performance and security in the late 90s several research groups proposed a new operating-system architecture, known as library operating system (libOS). A libOS consists of a set of libraries which allow operating the hardware or talk to network protocols. Further, they contain a set of policies to provide access control and isolation on the application layer. A core advantage of this approach is predictable performance as it allows applications to talk to the hardware resources directly, instead of repeatedly switching between user and kernel space. Hence, a libOS doesn’t have a centralized networking service. Thus network packets from different applications are mixed together when they arrive at the network device. When running multiple applications that can access the hardware directly it is a serious problem to provide strong resource isolation. Another issue occurs when porting applications that rely on interfaces such as POSIX and thus need to be rewritten. Though the major drawback and
probably the reason why library operating systems failed is the need to rewrite drivers. At the speed that new commodity PC hardware is developed and their short lifetime makes this an almost impossible task. (see [3])

2.3 Unikernels

Modern hypervisors run on commodity hardware and provide VMs with CPU time, strongly isolated virtual devices for networking, block storage, USB and PCI bridges. A library operating system running as a VM would only need to implement drivers for these virtual appliances. Further, the isolation between a libOS applications can simply be achieved by spawning one VM per application. These virtual library operating systems are called unikernels. Each unikernel VM therefore can be specialized to its particular purpose. Even though the hypervisor caters for a lot of work an unikernel would still need to take care of traditional operating system services like file system access. As each VM is highly specialized therefore the attack surface is reduced in comparison to traditional operating systems that share huge amounts of common code. (see [3])

The remainder of this paper will introduce the two unikernel implementations MirageOS and rump kernels. Whereas MirageOS (Section 3) has been developed to be used in cloud computing area, rump kernels (Section 4) aim to run anywhere even on bare-metal.

3. MIRAGEOS

The MirageOS unikernel has been developed at the University of Cambridge and is a Xen and Linux Foundation incubator project. It is entirely written in the functional programming language OCaml and thus only supports applications written in OCaml. Figure 1 shows a comparison between an application running on top of a traditional operating system and the Mirage unikernel. During compilation the Mirage compiler combines configuration files, application source code and needed kernel functionalities, e.g. TCP/IP-Stack, into one specialized unikernel. The unikernel can greatly reduce the footprint of a VM. For example a Bind9 DNS Server appliance on a Debian VM uses over 400MB, in contrast the Mirage DNS server unikernel has only 200KB. To assure communicating with external systems the mirage unikernel relies on standard network protocols. The kernel can either be executed as a user process on top of a UNIX operating system, which allows to easily debug and test applications, or it can be launched like a VM on the Xen hypervisor.

3.1 Performance & Security

Apart from a possible much smaller footprint Mirage aims for better performance and security. Traditional OS contain several hundred thousand, if not millions, lines of code that have to be executed every time it boots. The Mirage unikernel only contains the bare minimum needed to run its appliance this allows a VM to boot in only a couple of milliseconds. This can be especially useful if VMs are to be spawned based on traffic.

To offer better security an unikernel is sealed at compile time. This means any code not present during the compilation will never be run, which completely prevents code injection attacks. Therefore, the unikernel creates a set of page tables which are never both writable and executable. The sealing is achieved by a special seal hypercall to the hypervisor. This approach means that the hypervisor needs to be adjusted to support sealing and that an unikernel VM cannot expand its heap, but instead must pre-allocate all memory at startup.

Modern operating systems use Runtime Address Space Randomization (ASR) to make it harder for attackers to execute malicious code. Doing ASR at runtime requires a runtime linker which will add significant complexity into the unikernel. Luckily any changes to the unikernel appliance must result in recompiling it, hence ASR can be performed at compile time using a freshly generated linker script which avoids the need for runtime ASR. (see [4])

3.2 Virtual memory layout

Mirage is run on Xen with the special boot library PV-Boot. PVBoot will initialize the VM with one virtual CPU and jump into the entry function. Mirage applications are always single purpose, hence to minimize the OS overhead it doesn’t support multiple processes or preemptive threading. The VM will halt once the main function returns. The whole VM is laid out in a single 64-bit address space. PVBoot provides two memory page allocators. A slab allocator which
is used to support the C code of the OCaml runtime. The idea of a slab allocator is to have caches of commonly used objects which are kept in an initialized state. Therefore, the slab allocator aims to cache freed objects, as most code is in OCaml this allocator is not heavily used. The second is an extent allocator which serves continuous virtual memory in 2MB chunks. The language layout in the virtual memory is shown in Figure 2 which is divided into three regions: text and data, external I/O pages and the OCaml heaps. The OCaml heap is split into a minor heap for short lived values and large major heap for long lived values. The minor heap has a single extend of 2MB and grows in 4kB chunks and the major head has the remainder of the heap and grows in 2MB chunks.

Communication between VMs is achieved by the local VM granting memory page access right to the remote VM via the hypervisor. The PVBoot library therefore reserve virtual memory (Figure 2: foreign memory) and allocates a proxy value in the minor heap. (see [4])

3.3 Drivers

Mirage uses the driver interfaces provided by Xen which consists of a frontend driver in the VM and a backend driver in Xen that multiplexes frontend requests. These are connected through an event channel for signaling and a single memory page which contains fixed size of slots organized in a ring. Responses are written into the same slots as requests. The entire Mirage I/O throughput relies on this shared memory ring. Instead of writing data directly into the shared page, 4kB memory pages are passed by reference. Using the same granting mechanism that is applied by inter VM communication. This results in a high-performance Zero-Copy Device I/O. All drivers in Mirage are written in OCaml, therefore the shared memory ring is mapped into an OCaml `Bigarray`.

The network driver supports two communication methods. Firstly for on-host-inter-VM `vchan` transport and secondly Ethernet transport. `vchan` is a fast shared memory interconnection. The driver allocates multiple continuous pages for the I/O ring to have reasonable buffer. VMs can exchange data directly without intervention of the hypervisor. `vchan` is present in Linux since kernel version 3.3.0 which enables easy interaction between Mirage and Linux on the same host. Things get more complicated for Ethernet. While the changes for reading only require splitting header and data, writing is a bit more complicated because a variable length header has to be prepended. This is solved by allocating an explicit header page for every write operation. The entire payload must be present before writing the whole packet into device ring.

Besides networking most applications also require storage. Storage uses the same ring based shared memory I/O as networking does. For example the FAT-32 storage driver accesses the disk one sector at a time to avoid building large lists in the heap. (see [4])

3.4 Modularity

MirageOS is composed of a highly modularized dependencies system which allows users to easily develop and test their appliances in a familiar UNIX environment and afterwards recompile it to run on Xen. Figure 3 shows a module graph for the sample application `MyHomePage` which serves static web sites. Therefore, it depends on a HTTP signature which is satisfied by the Cohttp module. The Cohttp signature itself requires an TCP implementation. For developers on a UNIX system this dependency can be provided by the UnixSocket library. In a next step the developer can decide to switch to the OCaml TCP/IP stack (MirTCP). MirTCP requires Ethernet frames to be delivered which can be provided by the MirNet module. At this point things get exciting as the developer can either decide to stick with UNIX or recompile the application to link against the Xen network drivers. For the developer recompiling just requires to substitute the compiler flag `-unix to -xen.

4. RUMP KERNELS

The rump in rump kernels doesn't mean the fleshy hindquarters of an animal and is in no way to be associated with a rump steak. Instead, it defines a small or inferior remnant that is carrying on in the name of the original body after the expulsion or departure of a large number of its members. In case of the rump kernel the original body is the monolithic NetBSD OS and the remnant are the NetBSD kernel drivers. To put things into perspective, the rump kernel project's original goal was to make kernel driver development possible in userspace where a driver crash doesn't result in a crash of the whole OS which is a bliss for driver developers. The rump kernel is shown in figure 4 which consists of approximately one million lines of unmodified, battle-hardened NetBSD kernel drivers running on top of a documented hypervisor interface [1]. In order to run the unmodified drivers some platform-independent glue code is necessary which is entirely transparent to the users. The platform-specific hypervisor implementation is only about 1000 lines of code. Using this architecture allows running the standalone NetBSD kernel drivers on many platforms. Currently supported platforms are Xen, KVM, POSIX userspace, Linux userspace and every bare-metal that is supported by NetBSD. Having the drivers run anywhere is a neat thing for driver develop-
ers but it isn’t necessarily the concern of most application developers who rather like to run applications on top of a rump kernel. The solution of running unmodified POSIX applications on top of the rump kernel is as clever as simple. All that is necessary is a modified libc library where syscall traps have been replaced with equivalent rump kernel calls.

The drivers are the components of a rump kernel which are built for the target system as libraries and the final image is constructed by linking the component-libraries which are controlled by some kind of application which controls their operations. An application can put together a subset of driver which it needs to operate. For example a web server serving dynamically created content will need a TCP/IP stack and sockets support as well as memory and I/O devices access. These dependencies are resolved by the anykernel and the rump kernel hypercall interfaces. (see [1])

4.1 Anykernel & Hypercalls

The anykernel architecture is the key technology for rump kernels. The “any” in anykernels stands for the possibility to use the drivers in any configuration: monolithic, microkernel, exokernel, etc. In short, using the anykernel kernel modules can be loaded into places beyond the original OS. This is done by separating the NetBSD kernel into three layers: base, factions and drivers. The base layer contains all mandatory routines, such as allocators and synchronization, and thus must be present in every rump kernel. The other two layers are optional. The factions are further separated into devices, file systems and networking which provide basic routines. Finally, the driver layer provides the actually drivers like file system, PCI driver, firewalls, etc. To compile the anykernel no magic is required, just plain old C linkage.

In order to run the anykernel the drivers needs access to back-end resources like memory. These resources are being provided by the hypercall interface implementation. The hypercall implementation is written on top of a platform where it must be able to run C code and do stack switching. On bare-metal this requires some bootstrap code whereas in hosted environments like the POSIX userspace, that code is implicit present. (see [1])

4.2 Rump Kernel Clients

Rump kernel clients are defined as applications that request services from a rump kernel. Applications must explicitly request the services they require. There are three types of clients - local, remote and microkernel - which are shown in figure 5.

The local client exists in the same process as the rump kernel and does have full access to the kernel’s address space but typically request are made through function calls which are defined in the kernel syscall interface. The benefit of this architecture is compactness and speed. The downside is that the client is required to bring the kernel into a suitable state such as adding routing tables and mounting file systems.

Remote clients use a rump kernel but reside elsewhere. They can be either located at the same host or a remote one. Due to this separation they can only access kernel modules which are implemented by the interface. Hence, the client is not allowed to access arbitrary resources, this allows for security models with different levels. Furthermore, one rump kernel can be used by clients which is especially useful in userspace where the fork()/semantics otherwise would mean to duplicate the entire kernel. The communication between client(s) and kernel is possible via local domain or TCP sockets. The request routing policies is hereby controlled by the remote client. The drawback of remote clients is that the IPC mechanism causes an overhead which negatively affects the performance.

Microkernel clients are similar to remote clients with the difference that the requests are routed by the host kernel. The key difference is that the host kernel controls the request routing policies instead of the client. The rump kernel is implemented as microkernel server. (see [2])

5. CONCLUSIONS

Unikernels reduce the amount of layers an application has to go through to the bare minimum which results in highly specialized, fast and lean operating systems that run one application or even one module of a larger application. With this approach hundreds if not thousands of unikernels can be run on a hypervisor. This approach greatly reduces the size of VMs are a lot of dependencies can be left out. Furthermore, security bugs can only result in the compromise of one application and not the entire OS.

When using MirageOS there are some drawbacks as it completely drops compatibility for existing appliances and forces developers to use the OCaml programming languages. Both Mirage and rump kernels provide the ability to run application in userland which allows developers to efficiently test and debug their applications.

Unikernels take library operating systems to the next level by alleviating their driver problem. Further, using rump kernels also avoids rewriting applications because of a compatibility layer.

6. REFERENCES

