Unikernel Linux (UKL)

Jakob Schmid

Abstract

This report discusses Unikernel Linux (UKL), an approach to introduce a unikernel target into the Linux kernel. The specialized demand of cloud services has recently given rise to a resurgence of library operating systems in the form of unikernels. The authors of the UKL paper want to show that the Linux kernel can be modified to include the benefits of unikernels, while maintaining the ecosystem of applications and maintainers of Linux.

1 Introduction

Modern cloud services are often highly specialised, to the point of a microservice, where an applications is split up into a collection of loosely coupled services, that communicate through lightweight protocols and each only fullfill a single purpose. These services share a few key demands. They need to be able to communicate efficiently with each other. Since they need to communicate with other services, they are exposed to the internet to some degree and thus should be as secure as possible. Due to the single purpose nature of the microservices they often execute as a single process. Lastly the image the service is executed with and the memory usage should be as small as possible. The services are often executed in a virtual machine on rented hardware, so lower requirements in memory and storage can improve the profitability of a service. These demands have caused a resurgence of research exploring the concept of a library operating system (libOS) and the emergence of unikernels.

In a libOS a target application is linked with a set of libraries that provide all the services a regular OS would usually provide. The resulting executable can then be deployed directly to hardware. In 2013 the term *unikernel* was first introduced for libOSs for cloud services and deployment to virtual hardware [8].

Since then multiple unikernels have been created. Some are written from scratch like ClickOS [12] and Unikraft [6]. Others borrow code from an existing OS like Drawbridge [14] that uses code from Windows.

UKL uses the Linux kernel as a base but tries to keep the changes minimal so that UKL can be maintained with the (general purpose) Linux kernel.

2 Background

2.1 General-purpose monolithic operating systems

In a general-purpose monolithic operating system, such as Linux and Windows, the operating system (OS) defines an interface to use the physical resources of the system. This interface hides details about the physical resources behind high-level abstractions like processes, files, addresss spaces and interprocess communication. Applications running on the system use this interface and its abstractions to utilize the physical resources of the system. The implementation of the abstractions must be general to allow the execution of very different applications. The OS also ensures that different applications can execute concurrently, without interfering with each other. Virtual address spaces are used to allow for simultanious usage of physical memory. Access to other resources, such as files and network communication, is controlled by the OS. This protection requires that applications must not be able to modify or replace the implementation of the abstractions supplied by the OS.

2.2 Library operating systems

One of the problems of the fixed abstractions for system resources in general-purpose OSs is that it denies applications the advantage of domain-specific optimizations. For example in 1994 Cao et al. [2] showed that application-controlled file caching can reduce application running time by 45%.

In the mid 1990s the libOS was proposed as a different OS architecture where the application can control the hardware abstractions, instead of the OS [3]. In a libOS the hardware resources are not controlled and protected by a kernel. Instead a set of application-level libraries implement the mechanisms to drive hardware, communicate over the network and all other functionality a general-purpose operating system would provide for an application.

Further a set of policies are used to enforce access control and isolation in the application layer [9].

The libOS architecture allows applications to access hardware resources directly thus eliminating the need for repeated privilege transitions to move data between kernel and user space. In addition applications can now choose a library to drive the hardware that is taylored to the needs of that specific application, thus improving performance. This architecture also makes the performance more predictable. In a monolithic OS the kernel might schedule a different process after a context switch or do signal handling before handing control back to the application.

Compiling an application with only the libraries the application requires can significantly reduce the amount of code involved. This reduction in code can not only lead to a reduction in image size, but can also decrease the chance of the code containing a security vulnerability [8].

The libOS architecture has two major disadvantages: One, it is hard to achieve strong resource isolation because not all usage of hardware resources has to go through one abstraction supplied by a kernel like in a monolithic OS. Two, device drivers have to be rewritten to work as a library rather than for example a kernel module. This implementation effort has caused problems with hardware compatability for libOS projects and thus mostly limited their relevance to the research domain [8].

2.3 Unikernels

OS virtualisation can overcome both of the major disadvantages of libOSs mentioned in § 2.2. The lack of resource isolation between applications can be avoided by dedicating a seperate virtual machine (VM) to each application. This delegates resource isolation from the OS level to the hypervisor. The hypervisor offers a much simpler, less fine grained interface than a conventional OS, consisting of virtual CPUs and memory pages, rather than the process-oriented approach in conventional OSs [9]. Dedicating a seperate VM to each application also leaves more room to specialize the VM to its particular purpose. The problem of hardware compatability of libOSs can also be overcome with virtualization. The libOS only has to implement drivers for the virtual hardware devices offered by the hypervisor. The hypervisor is then responsibel for driving the ever changing physical hardware.

Madhavapeddy et al. introduced the term unikernel in a paper published in 2013 [8]. Here unikernels are described as an approach to deploy cloud services using specialised single-purpose libOS VMs running directly on the hypervisor [8].

Despite the benefits of the unikernel approach, they have not yet been widely adopted outside of the research and experimental domain. This could be attributed to the engeneering burden of porting existing software to an environment with no or only partial support for legacy software interfaces [16].

2.3.1 Performance and size

Deploying a service using a unikernel can offer significant performance advantages. Typical optimisations other than using libraries specialised for the given application inlcude avoiding ring transition overheads [10], leveraging the single address space to pass pointers rather than copying data [17], deferring preemption in latency sensitive routines, exploiting application knowledge to remove code that is never executed [8], and cross layer optimization between kernel and application code [16]. Memcached running on a unikernel TCP/IP stack demonstrated a 200% throughput improvement compared to Linux [17], unikernels deployed within a microVM have shown six to ten times shorter boot times over containers [5] and a micropython unikernel reached an image size of 1MB while requireing 8MB of memory to run [11].

2.3.2 Thread model

Unikernels are per definition concerned with applications that provide network facing services. The focus on virtual hardware and deployment to a hypervisor often implies usage in a multi-tenant datacenter. Here the provider of the datacenter is trusted not to be malisious. Other tenants and internet-connected hosts in general are seen as a potential attacker. Internally the hypervisor is used as the sole unit of isolation, rather than adopting a multi-user access control model as most general purpose OSs do. External entities the unikernel communicates with are trusted through protocol libraries such as SSL or SSH [8].

2.3.3 Security

A problem with images based on a general purpose OSs is that misconfiguration can leave unneeded or unnecessary services running. These services can significantly increase the remote attack surface. When creating an appliance using the unikernel approach only the libraries needed by the particular application are included in the final image. Thus no unnecessary services are included in the final image. Further this can also drastically deacrease the

amount of code included and thus the chance for a vulnerability in the code.

About 70% of vulnerabilities in Windows, a general purpose OS written in C/C++, that Microsoft assigns a CVE each year are memory safety issues [13]. Unikernels written in high level programming languages with a strong type system can use high level language features for memory management to negate a large portion of these vulnerabilities [7, 8]. MirageOS [8], a unikernel written in OCaml, can be sealed [4] at runtime, thus preventing any code not present at compile time from being executed. MirageOS further implements compile-time address space randomization to defend against return-oriented programming attacks and save on runtime complexity. This is justified by the fact that for unikernels reconfiguring the appliance means recompiling it, potentially for every deployment [8].

Another aspect of unikernel security is that the chance for a common exploit is lower. An exploit found in the linux kernel can potentially affect all machines running Linux due to them sharing a vast majority of their code. Each unikernel however is specialized to suit the needs of only one application thus sharing less code with other appliances using the same unikernel. The smaller amount of shared code decreases the probability of an exploit being applicable to many unikernel appliances.

2.3.4 Categorisation of unikernels

Clean slate unikernels are written from scratch. Here the unikernel designers have full control over the language and the methodology used [16]. Thus the unikernel can be specialised to an abitrary degree to service the needs of a particular class of applications. The interface available to the application can also be choosen freely.

In *strip down* unikernels an existing kernel code base is forked and all functionality not needed for the unikernel is removed. The interfaces and general purpose libraries can be perserved to some degree to make porting of existing applications written for the original kernel over to the unikernel easier. This generality however comes at the cost of loosing room for specialisation, efficient pathways and fine-tuned implementation. The creation of an out-of-tree fork of an existing OS comes with the burden of having to manually port updates from the original OS to the unikernel at regular intervals in order to keep the support for existing applications [16].

```
/* code snippet */
while (!sleep)
```

Here you can include a sample figure. Use something like

`\includegraphics[scale=.8]{template}`

to include an encapsulated postscript figure. The *scale* argument can be used for scaling the picture, although it may scale the font incorrectly.

Figure 1: Sample Figure

```
sleep++;
```

Listing 1: A sample code snippet

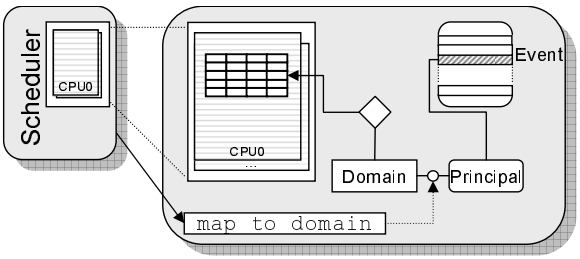

Figure 2: Sample figure automatically from Windows prn.

3 Related Work

Works [1] and [15] are relevant but different.

4 Approach

5 Conclusion

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, October 19–22 2003.

[2] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for

application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[4] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, March 2007.

[5] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, May 2017.

[6] Simon Kuenzer, Vlad-Andrei Badoiu, and Hugo Lefeuvre. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, April 2021.

[7] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel developement. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, October 2019.

[8] Anil Madhavapeddy, Richard Mortier, and Charalampos Rotsos. Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, March 2013.

[9] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? December 2013.

[10] Toshiyuki Maeda and Akinori Yonezawa. Kernel mode linux: Toward an operating system protected by a type theory. In *n Annual Asian Computing Science Conference*, pages 3–17, 2003.

[11] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating System Principles*, October 2017.

[12] Joao Martins, Mohamed Ahmed, and Costin Raiciu. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, April 2014.

[13] MSRC Team. We need a safer systems programming language. `https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/`, July 2019. Accessed 2023-07-04.

[14] Donald E. Porter, Silas Boyd-Wickizer, and Jon Howell. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, March 2011.

[15] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, pages 65–85, Ottawa, Canada, July 2005.

[16] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2019.

[17] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. Ebbrt: a framework for building per-application library operating systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, November 2016.