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 [5].

Since then multiple unikernels have been created. Some are written from scratch like ClickOS [7] and Unikraft [4]. Others borrow code from an existing OS like Drawbridge [8] 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 [6].

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 [5].

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 [5].

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

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

Listing 1: A sample code snippet

3 Related Work

Works [1] and [9] are relevant but different.

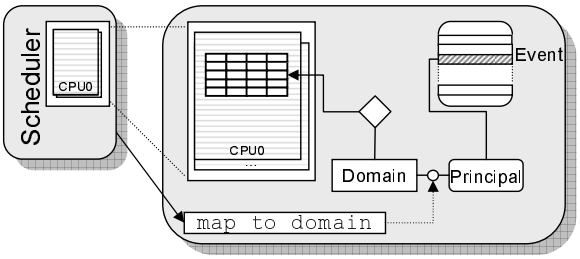

Figure 2: Sample figure automatically from Windows prn.

4 Approach

5 Conclusion

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, Oct. 19–22 2003.

[2] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, Nov. 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*, Dec. 1995.

[4] S. Kuenzer, V.-A. Badoiu, and H. Lefeuvre. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, Apr. 2021.

[5] A. Madhavapeddy, R. Mortier, and C. Rotsos. Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Mar. 2013.

[6] A. Madhavapeddy and D. 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? Dec. 2013.

[7] J. Martins, M. Ahmed, and C. Raiciu. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, Apr. 2014.

[8] D. E. Porter, S. Boyd-Wickizer, and J. 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*, Mar. 2011.

[9] 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.