Building MirageOS on Linux and Improvement to an Open Source Project Solo5

Yanzhi Li
yli16@earlham.edu
Computer Science Department
Earlham College
Richmond, Indiana

ABSTRACT
As they have come to dominate many areas of IT, most of the clouds are still loaded with old fashion software that was used in the days before cloud. There is an urgent need for a faster, smaller and more secure system that supports cloud computing and high performance computing. That’s where unikernels come in. Unikernels is a library operating system that consists of the minimal set of libraries that a target application requires to operate and gain better performance. It is lightweight, fast and much more secure than containers. It could benefit cloud computing, system optimization, high performance computing, etc. With more researchers involved, unikernels will become the next stage of system virtualization, or even the next evolution of operating systems. Unikernels run directly on a hypervisor or hardware without an intervening OS. However, Dan Williams et al. introduced a new approach that enables unikernels to run as processes on Linux, FreeBSD, etc. [16]. This is the open source project solo5. It provides an execution environment that could run applications built using various unikernels. However, since unikernel and Solo5 interface are newly developed technologies, they are unstable to use in some situations and have not yet become general for all platforms.

This paper presents two improvements to the Solo5 open source project. One is to improve the user experience and the other is to add proper support for cross-compilation of the system. The specific unikernel used is the MirageOS, which is for constructing secure, high-performance network applications across a variety of cloud computing and mobile platforms.

Keywords Linux, Library Operating System, Operating System, MirageOS

INTRODUCTION
The first unikernel-like systems were designed in the late 1990s. However, because of the hardware requirement and the low demand of their potential, unikernels were paused and faded from researchers’ sight. The technology behind unikernels has been developed over the last five years. Unlike a complete operating system such as Linux, which has an expanded kernel to support a wide range of functionality, a unikernel is stripped off all the unnecessary parts. In other words, it is a specialized, executable image that is deployed directly on hardware and linked with target application [14]. A similar idea of a lightweight runtime environment is the container which is now in great demand. However, Filipe et al. [12] pointed out that containers offer weaker isolation than VMs. Thus, the unikernel seems to become the next stage of system virtualization. Moreover, the idea of introducing the unikernels to traditional operating system like Linux has become a major branch of unikernel studies.

Traditionally, people use a specially designed compiler to wrap the application software and operating system support libraries to form an unikernel target at compile time instead of runtime. The result is a single application image that consists of everything supports and optimizes the application. All drivers, all I/O routines, and all support library functions normally provided by an operating system are included in the executable. This becomes a problem as people found out that some applications could not be implemented as unikernels easily and that this approach requires a lot of extra work to compile the target applications.

Dan et al. introduced a new approach of running unikernels as processes while retaining their VM-like isolation properties. This idea has now grown into an open source project Solo5, which is a general sandboxed execution environment suitable for running applications built using various unikernels [16]. In addition, Solo5 now supports different sandboxing technologies on diverse host operating systems and hypervisors. Their research provides a practical and accessible way to implement unikernels and utilizing their advantages. However, the Solo5 project is still in its early stage for development and requires many improvements not only in technical functionalities but also in user experience. This will be a long term and positive project as it is the pioneer of this approach.

This project aims to contribute to the Solo5 open source project. It tried to solve three issues of the project including: Proper support for cross-compiling, Support for building the system with gcc 10.x, and Forbidding users to execute target unikernels directly. First, this will help to improve the user experience as they will receive more understandable error messages. Second, while the project tried to add support for cross platform compilation, which is important as shown in many cases, it turned out that this issue is too ambitious to solve individually. Although solo5 now supports several host operating systems and processor architectures such as FreeBSD and the aarch64 architecture, the primary platform of this project is on Linux Debian 10 the x86-64 architecture.

This paper first introduces the concept of unikernels and its development, especially introducing some unikernel implementations. Then it examines the design and structure of the solo5 interface. The three issues that the project tries to solve are then presented. It explains how two of the issues were successfully solved and why one issue was beyond my capacity. Lastly, the paper discusses how the project could be developed on platforms other than Linux, the future contribution to the project and presents an analysis of its
potential. The following are the major contributions of this research project:

- Improving user experience by implementing better error messages report
- An analysis of how cross compilation could be implemented

RELATED WORK

The related work section includes two parts. The first part introduces some papers that explain the concept of unikernels, why they are important, and what benefits they have. This part also talks about some implementations of the unikernels. The second part introduces some ideas of associating the unikernels with traditional operating systems, especially the project by Dan Williams, where the Solo5 project originates from. His idea of porting MirageOS to run on the Linux/KVM hypervisor has grown into a general sandbox execution environment. This section is divided into two subsections - what are unikernels and unikernel on Linux.

1 What are Unikernels

The work in this category concerns primarily with unikernels as a separate microsystem from a complete operating system. Lankes et al. viewed unikernels as a totally independent system image to be used for extreme scale computing[8]. Projects under this category are application oriented. In other words, their unikernels care only about how to be applicable.

Unikernels are machine images constructed by using library operating systems. They are composed of the minimal set of libraries which correspond to the OS constructs required for some specific applications to run. In other words, unikernels strip away everything that is unnecessary. They are specifically designed for certain applications and do not provide any other options. Besides, unikernels can directly run on the virtual or real, hardware without an intervening operating system[16]. Since they only use a small set of the resources required by a complete operating system, they are lightweight and platform-independent.

The idea of such a microsystem came from the demand for an improvement of the traditional operating system virtualization. Anil and David in their paper stated that the traditional operating system virtualization, while being very useful, is built upon an already layered software stack and thus adds more burden to the overall system[11]. They pointed out a typical virtual machine that contains a full operating system image was reasonable in several years ago because of the high cost of building such a system. A single system to perform multiple tasks was desirable. However, nowadays most deployed VMs ultimately perform a single function such as acting as a database or Web server. The demand for a single-purpose virtual machine is a reflection of the inexpensive cost of building new virtual computers. Thus come the unikernels. The new technology is capable of delivering: system security, small footprints, high application optimization, near instant boot times, good resource utilization. Unikernels could dramatically improve the performance of the target application. A similar purposed technology of lightweight virtualization is the container, which is now widely used in many fields such as cloud computing. Google also ran most of its services in containers. However, Filipe et al. expressed concerns for the security problem of the containers in their paper[12]. The API that a container used to interact with the host OS is fundamentally difficult to secure, even with many isolation mechanisms introduced in the past few years. In addition, containers are vulnerable to DoS attack. Thus they proposed to replace containers with unikernels, which provides high isolation as a complete VM and no less efficiency.

The unikernels introduced in this section are application/function oriented. In other words, they are designed for some particular applications or providing some specific functionalities. According to Raza et al., there are two approaches to creating a new unikernel: a clean slate approach where the kernel is largely built from scratch, and a strip down approach where people stripped an existing kernel codebase of functionality that are unnecessary for the unikernel. One of the most famous unikernel implementation exploring the clean-slate design space is the MirageOS, which is extremely specialized and limited to OCaml-based applications[11]. Meanwhile, strip-down unikernels are better at porting software by preserving the general-purpose libraries and interfaces of a legacy kernel codebase. One representation of this type is the RumpRun unikernel which contains a heavily-reduced version of NetBSD[14].

In addition, unikernel communities have now developed many different unikernel implementations for other languages, some of which are able to support common applications and runtimes like nginx, redis, Node.js express, Python, etc[16]. For example, HalVM8 is an unikernel based on the famously pure and lazy Haskell language[11].

With lightweight characteristics and strong isolation, unikernels have substantial advantages for a wide class of applications. They are well suited for microservices, network function virtualization, and High-performance Computing.

2 Unikernels on Linux

Rather than designing a new unikernel for some target applications, work under this category focuses on how to port current unikernel implementation to the environment that people more familiar with, which includes the Linux, FreeBSD, and other hypervisors. Some papers proposed the idea of incorporating unikernels into current Operating System Linux and making them compatible with Linux applications. Raza et al. argued that unikernels’ advantages represent the next natural evolution for Linux[14]. While Pierre et al. pointed out that the barrier to their widespread adoption is the difficulty/impossibility to port existing applications to current unikernels, they created an unikernel HermiTux which is the first unikernel providing binary-compatibility with Linux applications[11]. These people aimed to make unikernels backward compatible so that the unikernel could be more acceptable and more people will be involved in its development. Raza et al. demonstrated that Linux can be turned into a unikernel successfully[14]. Pierre et al. built HermiTux, a unikernel that runs native Linux executables by providing binary compatibility, relieving application programmers from the effort of porting their software[11]. They all showed the possibility of merging the new technology with the complete operating systems.

Dan Williams et al. used a different approach by implementing unikernels as processes while retaining their VM-like isolation properties. Running unikernels as processes will benefit their...
progress towards production, because, as processes, they can reuse lighter-weight process or container tooling, be debugged with standard process debugging tools, and run in already-virtualized infrastructure[16]. This idea has grown into the Solo5 project. Solo5 redefines an interface between the unikernels, as user processes, and their host operating system. It is a sandboxed execution environment, where applications built with unikernels could run, and it supports different sandboxing technologies on diverse host operating systems and hypervisors. Moreover, the Solo5 backend has now become the recommended back end for several unikernel implementations, especially MirageOS.

DESIGN AND IMPLEMENTATION
This project depends on the paper by Dan Williams et al. and the Solo5 open source project. It includes two primary sections. First, it explored and built the Solo5 system on a virtual machine. Second, it worked on three issues posted on the Solo5 project, two of which were solved and one unsolved.

1 Solo5 Structure and Overview
Dan Williams et al. ported MirageOS to Linux via a unikernel base interface called Solo5[16]. Figure 1 shows an overview of how to run unikernels as processes. A tender process, which is the bridge between the unikernels and the host system, was created. The tender process is a Linux backend that is built upon the ukvm monitor, which itself contains backends to run on different systems and architectures, from Linux/KVM to FreeBSD and OpenBSD, from x86 to ARM. The tender dynamically loads the unikernel code into its own address space and configures seccomp filters to only allow certain system calls. Once inside the tender space, the unikernels are only allowed to do normal procedure calls to the hypercall implementation in the tender, which performs a system call to Linux and returns the result to the unikernels.

Figure 1: Unikernel isolation using a tender process with seccomp technology[16]

2 Solo5 Tender and Unikernels
Solo5 has five types of tender process, among which hvt(hardware virtualized tender) and spt(sandboxed process tender) are now the stable and productive ones. Hvt supports Linux, FreeBSD and OpenBSD systems and is loosely equivalent to a hosted virtual machine monitor, which isolates the guest unikernels. On Linux, the solo5-hvt requires the dependencies of KVM and access to /dev/kvm. On the other hand, Spt loads the guest into memory and uses a minimal whitelist seccomp sandbox to isolate the guest unikernel. While this approach currently supports Linux only, it is more general for Linux users because it only requires a package libseccomp >= 2.3.3 and does not rely on KVM and the underlying hardware. Since the project uses a virtual machine with Debian distribution to explore Solo5, it makes more sense to use the Spt target in this project for its lower requirement of both software and hardware. At unikernel build time, a JSON file called application manifest is specified. The tender is responsible for setting up access to the host resources declared in the application manifest. The unikernel used in this project is the MirageOS, which consists of a set of OCaml libraries that link with a runtime to form either a standalone unikernel. These libraries are managed via the OPAM tool.

3 Creating VM and Installing Solo5
VirtualBox was used to create two host operating systems Debian 10 and Fedora 32. All the dependencies were then added, which includes:

- Git Software
- Network Configuration to enable internet access
- A C11 compiler; recent versions of GCC and clang are supported
- GNU make
- Full host system headers (on Linux, kernel headers are not always installed by default)
- Pkg-config and libseccomp >= 2.3.3

Solo5 was then pulled from https://github.com/Solo5/solo5 and the configure script was used to build the system.

4 Project Issues Solved
Two open issues were solved in this project.

(1) Enable the system to be built with GCC 10.x
(2) Stop host system from executing Solo5 binaries directly on Linux

Enable the system to be built with GCC 10.x:
Issue 1 was founded when we attempt to install Solo5-bindings on Fedora. The compilation failed and gave the ERROR: Only ‘gcc’ 4.x+ is supported on Linux. This was fixed by modifying the gcc section of the configure script.

Stop host system from executing Solo5 binaries directly on Linux:
When the Solo5 binaries were executed directly (regardless of target), the host kernel would in most cases attempt to load and run the executable. However, the host would only produce a segmentation fault on Linux. This would result in bad user experience and potential security risk. We added a PT_INTERP in binding-specific linker scripts, which points to a trailing slash /nonexistent/solo5/. This method prevents the host kernel from loading the binaries in all cases and avoids the risk of exposing the host kernel to unikernel binaries from untrusted sources. In addition, permission deny or not a directory error messages will be produced when a Solo5
unikernel binary is run directly, which is more sensible and user friendly than a segmentation fault.

5 Project Issues Unsolved

This project also tried to work on the issue of adding support for cross-compilation to build the system. On the Debian VM, the ARM 64bit toolchain was used to test cross-compilation. After careful investigation, it turned out that this issue is far beyond the scope of the project for two reasons. First, the Solo5 tenders use the specific toolchain af-gcc to be built. Our attempt to override the toolchain caused the compilation crash. Currently, we could not find a way to safely override. Second, the Solo5 system supports several targets on different platforms including FreeBSD and OpenBSD. As a result, it is hard to add support for cross-compilation for Linux only. In other words, a comprehensive modification to the configure script and other files seems reasonable. With limited time and resources, this project failed to complete this task.

6 Building MirageOS

The experiments for the project results consist of two parts. First, after committing changes to the Solo5 open source project, I received feedback and advice from other contributors and my pull requests were accepted. Second, I built MirageOS via the Solo5 tender on the virtual machine and built a Mirage hello world.

CONCLUSION AND FUTURE WORK

We explore the approach of running unikernels as processes on Linux and show the Solo5 tender as the interface between unikernels and the host operating system. We first built Solo5 system on virtual machines and then worked to solve the issues of the Solo5 open source project.

Solo5 has been in development since late 2015. It supports a variety of different targets and Solo5-based unikernels on Linux/KVM, FreeBSD/vmm and Google Compute Engine. However, it has many limitations and requires long term efforts to develop. The Solo5 interfaces are not yet stable and do not support cross-compilation. The current interface and its implementations have not been designed for high I/O performance. Solo5 has the potential of introducing unikernels to different platforms. The next step is to transform the experimental targets into productive targets. We will continue to contribute to the project and work on the open issues.

ACKNOWLEDGEMENT

I would like to thank Prof.Xunfei to for her help and advice for my project and Prof. Charlie for his suggestions on unikernels.

REFERENCES


