Scriptable Operating Systems with Lua

Lourival Vieira Neto
The NetBSD Foundation
lneto@NetBSD.org

Roberto Ierusalimschy
Departamento de Informática, PUC-Rio
roberto@inf.puc-rio.br

Ana Lúcia de Moura
Departamento de Informática, PUC-Rio
amoura@inf.puc-rio.br

Marc Balmer
The NetBSD Foundation
mbalmer@NetBSD.org

Abstract

Extensible operating system is a design based on the idea that operating systems can be adapted to meet user requirements by allowing user extensions. In a different scenario, that of application development, there is a paradigm that supports that complex systems should allow users to write *scripts* to tailor an application to their needs. In this paper we propose the concept of *scriptable operating system*, which applies scripting development paradigm to the concept of extensible operating systems. Scriptable operating systems support that operating systems can adequately provide extensibility by allowing users to script their kernel. We also present an implementation of a kernel-scripting environment that allows users to dynamically extend Linux and NetBSD operating systems using the scripting language Lua. To evaluate this environment, we extended both OS kernels to allow users to script CPU frequency scaling and network packet filtering using Lua.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks; D.4.7 [*Organization and Design*]

Keywords scriptable operating system, kernel scripting, Lua programming language

1. Introduction

Extensible operating systems were introduced in the late 60's as a design approach that supports that operating systems (OS) can improve their flexibility by allowing the use of extensions [20]. The basic idea is that general-purpose OS cannot anticipate the demands of all their applications and therefore should be able to adapt their behavior to meet specific or new requirements [30].

Operating system extensibility can be provided in different ways, from the adjustment of system parameters — as supported by sysctl [23] and sysfs [24] — to dynamically injecting or linking code to the kernel. The later approach allows users to have direct access to OS internals and to create new policies and mechanisms; it has been extensively explored in the last 20 years in research operating systems such as Exokernel [8], SPIN [6], VINO [34], μChoices [7] and Singularity [13]. Most current general-purpose OS provide this kind of extensibility by allowing privileged users to dynamically load kernel modules.

In a different scenario, that of the development of customizable applications, there is an important trend to split complex systems in two parts — *core* and *configuration* — and to combine system programming languages and *scripting* languages to develop those parts [15, 26]. System programming languages, such as C and C++, are typically compiled and statically typed; they are used to develop the application core components. Scripting languages, such as Lua, Tcl, and Python, are typically interpreted and dynamically typed. In this scenario, they are used to implement the configuration part, which is responsible for connecting the application core components, tailoring the application to satisfy its users customization demands. The use of a scripting language to support application customization brings significant benefits. First, it increases productivity, favoring an agile development environment [26]. Second, the use of a full-fledged language allows users to develop run-time configuration procedures that are impossible to implement by merely choosing parameter values [15].

In this work we present an experiment to improve operating system flexibility that combines extensible operating systems with extension scripting languages. To explore our combined approach, which we call *scriptable operating systems*, we initially developed Lunatik, a small subsystem that provides a programming and execution environment for OS kernel scripting based on the Lua programming language [14, 15]. We then evolved this first experiment to a more reliable infrastructure — *Lua in the NetBSD kernel* — which is now part of the official NetBSD distribution. These two kernel-scripting implementations do not provide a fully scriptable OS. Instead, they provide frameworks that help making kernel subsystems scriptable.

The rest of the paper is organized as follows. Section 2 discusses our concept of a scriptable operating system and some issues involved. Section 3 provides the motivation for choosing Lua as an adequate scripting language for operating system kernels. Section 4 describes our kernel-scripting environment based on Lua. In Section 5 we evaluate this environment presenting extensions we implemented for Linux and NetBSD kernel subsystems. Section 6 discusses some related work. Finally, Section 7 presents our conclusions.

2. Scriptable Operating Systems

An operating system can be made scriptable both in its user space portion (that is, system programs) or in its kernel. In fact, most op-

DLS '14, October 20–24 2014, Portland, OR, USA.

ACM 978-1-4503-3211-8/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661088.2661096

erating systems have scriptable system programs, such as BSD Init, which uses a collection of shell scripts to configure user-level initialization [23]. Ideally, a scriptable operating system should support both user space and kernel scripting facilities, providing a common scripting-language environment to customize and configure the whole system. However, the key and fundamental idea behind our concept of a scriptable operating system is *kernel scripting*: scriptable OSs must provide extensibility by allowing users to dynamically load scripts into their kernel. Because this is also the most challenging and important part of making an OS scriptable, it is the focus of this paper.

OS kernel scripting can be provided in two different ways: *embedding* and *extending* a scripting language [21, 25]. Embedding a scripting language means that kernel subsystems behave as host programs, invoking the script interpreter as a library to execute user-defined scripts. Extending a scripting language means that kernel subsystems behave as libraries to user-defined scripts, which detain the execution control flow. Extending a language treats the kernel as a library: control flow begins in the script and flows into the kernel; embedding a language treats the kernel as a framework: control flow begins in the kernel and flows into scripts.

To provide extensibility by embedding a scripting language, kernel developers need to modify their subsystems to make them execute user extension scripts by invoking the script interpreter. Through those scripts, users can adapt the operating system behavior to their demands, defining appropriate policies and mechanisms.

Some possible uses for kernel extension by embedding a scripting language are:

Process scheduling: by scripting the kernel, users can define alternative algorithms and policies for process and/or thread scheduling. This facility could support, for instance, the implementation of process quality of service (QoS). This kind of extension was successfully explored by the Bossa system [4], which uses a domain-specific language (DSL) suitable for this scenario.

Packet filtering: kernel scripting permits the use of more sophisticated rules for filtering and processing network packets, as an alternative to simple rule tables. This facility is provided through domain-specific languages by some extensible operating systems such as Exokernel [8].

Device drivers: kernel developers (and users) can implement and customize device drivers using scripts. A developer could, for instance, implement a driver for a new Universal Serial Bus (USB) peripheral or create her own keyboard layouts.

Extending a scripting language to support kernel scripting requires the development of binding libraries to expose kernel functions and data structures to user scripts. Providing scripts loaded into kernel space with direct access to kernel internals avoids the use of conventional user-level interfaces, which typically involve successive costly context switches and buffer copying.

Some possible scenarios for kernel scripting by extending a scripting language are:

Web servers: a web server could improve its performance by loading a script to process the HTTP/SPDY [5, 9] protocol stacks inside the kernel. This use case has been explored in many extensible operating systems [3, 19, 32, 35].

File systems: kernel scripting allows users to define their own abstractions for disk and network data storage. A user could, for instance, define her own disk-block layouts or implement her own protocols for a remote file storage facility. User-defined disk-block layouts are provided by Exokernel with the use of a DSL [8].

Network protocols: kernel scripting can be used for implementing network protocol daemons. This facility is specially useful for protocols that reside in lower network layers. A user-level daemon for the Link Layer Discovery Protocol (LLDP) [2], for instance, requires the implementation of user-level raw sockets. The application layer Real-time Transport Protocol (RTP) [31], which has much in common with transport layer protocols, is another interesting example.

Besides improving the flexibility of operating systems, kernel scripting also provides an interesting environment for prototyping and quick and easy experimentation of new kernel algorithms and mechanisms. This kind of environment allows the use of rapid development methodologies [26] by kernel developers themselves.

Traditional operating systems use several domain-specific languages for their configuration. As an example, NetBSD uses DSLs for system initialization (rc.conf(5)), packet filtering (npf.conf(8)) and kernel static configuration (autoconf(9)). The scriptable OS approach replaces these different DSLs by a single scripting language engine, simplifying the system architecture.

2.1 Examples of kernel scripting

The Linux kernel subsystem *CPUfreq* [27] manages CPU frequency and voltage by using dynamic frequency scaling mechanisms, provided by modern processors. A collection of built-in policies controls the power consumption of the system according to specific requirements. The routine that implements the active policy is called periodically to scale the CPU frequency to satisfy its associated requirements.

The CPUfreq built-in policy *Ondemand* controls CPU frequency with the purpose of saving energy while minimizing performance loss. This control is based on the CPU load over a period of time: when the CPU load rises above an "up" threshold, the control scales the frequency to the maximum value available; when the CPU load decreases below a "down" threshold, the control scales the frequency to the highest value available that is lower than or equal to twenty percent of the current frequency.

Kernel scripting can provide support for requirements that are not satisfied by the built-in policies. As an example, a user-defined script can control the CPU frequency to prevent overheating, while trying to preserve system performance and save power. To implement this new policy, the script can extend the Ondemand algorithm to consider the CPU temperature before analyzing the CPU utilization: if the current temperature is higher than or equal to one hundred degrees Celsius, the policy routine disregards the performance evaluation and decreases the frequency to cool the processor. Figure 1 shows a Lua script that implements this CPU frequency scaling policy. In this script, function `cpufreq.target` is responsible for changing the CPU frequency. It receives the CPU identification number, the frequency target and a specification of whether the frequency to be set should be the lowest frequency above or equal to the target value (`'>='`) or the highest frequency below or equal to the target value (`'<='`).

Kernel scripting can also provide and extend facilities for processing network packets. The NetBSD kernel subsystem *NPF* [29] allows users to define packet-filtering rules using a domain-specific language. These user-defined rules are applied to network traffic, but only on layers three and four. To perform deep packet inspection — that is, to inspect network layers above layer four — a kernel-scripting environment can allow users to associate an extension script with a regular NPF rule.

Suppose a new vulnerability is discovered in a specific implementation of the Secure Shell (SSH) protocol [36], currently employed by some servers behind a firewall. To avoid compromising these servers, we can use a Lua script that inspects the outgoing traffic and filters the SSH packets that originate from servers run-

```
up          =  80
down        =  30
overheated = 100

function throttle(cpu, cur, max, min)
  -- get utilization since last check
  local load = get_load(cpu)

  -- get temperature
  local temp = acpi.get_temp(cpu)

  if temp >= overheated then
    -- decrease frequency by 20%
    cpufreq.target(cpu, cur * 80 / 100, '<=')
  else
    if load > up then
      -- rise frequency to the maximum value
      cpufreq.target(cpu, max, '>=')

    elseif load < down then
      -- decrease frequency by 20%
    cpufreq.target(cpu, cur * 80 / 100, '<=')
    end
  end
end
```

Figure 1. Lua script for controlling CPU frequency

ning the vulnerable SSH implementation. Blocking the traffic coming from these servers protects them from accesses that can exploit their vulnerability.

Figure 2 shows a Lua extension script that implements this filtering facility. To activate the filtering, we can associate the script to a NPF rule that applies it to outgoing packets from TCP connections on port 22.

Function `filter` receives the header (`hdr`) and payload (`pld`) of a network packet. Because it only receives TCP packets originated on port 22, it can assume that the payload contains an SSH message. In the SSH protocol, when a connection has been established, both sides send an identification string; the script parses this string to verify whether the message has been sent by a server running the vulnerable SSH implementation. If so, it signalizes that the packet should be dropped.

```
function filter(hdr, pld)
  -- get a segment of the payload
  local seg = pld:segment(0, 255)

  -- convert segment data to string
  local str = tostring(seg)

  -- pattern to capture the software version
  local pattern = 'SSH%-[^-%G]+%-([^-%G]+)'

  -- get the software version
  local software_version = str:match(pattern)

  if software_version == 'OpenSSH_6.4' then
    -- reject the packet
    return false
  end

  -- accept the packet
  return true
end
```

Figure 2. Lua script for inspecting SSH packets

To extract the SSH implementation version, function `filter` converts the first 255 bytes of the payload to a Lua string and uses a pattern to locate and extract the SSH version. The pattern used by the script matches strings beginning with "SSH-" (the '%' character works as an escape in Lua patterns), followed by one or more printable characters with the exception of hyphen and whitespace, an hyphen, one or more printable characters with the exception of hyphen and whitespace (which specifies the version).

2.2 Addressing Scriptable Operating System Issues

Because scriptable operating systems allow users to load and run code in privileged (kernel) mode, they involve the same class of issues experienced by previous work on extensible operating systems [30, 33]. Because typical scripting development practices are applied, they also involve issues that are present in regular scriptable applications scenarios.

Scriptable operating systems issues are mostly related to maintaining the integrity of the system, providing ease of development and enforcing effectiveness and efficiency of kernel scripts. Among those issues, the main concern when providing scripting facilities to an OS kernel is to preserve its integrity. Kernel scripts must not be allowed to cause any harm. In other words, they should not be allowed to introduce malfunctioning, intentionally or not, either to the system itself or to the applications running on it. In practice, scripts can compromise the system integrity in many ways, such as:

Correctness: kernel scripts could introduce erroneous behavior to the system, like crashing or corrupting it.

Isolation: kernel scripts loaded by a specific user could corrupt resources owned by other users or compromise system fairness.

Liveliness: kernel scripts could fall into an endless loop, block an execution flow, or run for so long that it could compromise the whole system responsiveness.

Conventional operating systems typically try to guarantee system integrity by allowing only privileged users to load and run kernel extensions. On the other hand, extensible operating systems usually allow users to load and run unprivileged code inside their kernels. Scriptable operating systems can use both approaches, providing different privilege levels for different instances of the kernel-embedded interpreter. However, due to the higher-level nature of a scripting environment, kernel scripts, either privileged or not, cannot be fully responsible for guaranteeing system integrity. In particular, even privileged kernel scripts should not be responsible for managing memory allocation or explicit synchronization. This responsibility should be confined to the system-programing language code to preserve the separation of roles that is typical of scripting environments. That is, we should prevent kernel scripts to compromise the system integrity due to the problems of managing memory allocation (e.g., null pointer dereference, memory leak) and explicit synchronization (e.g., deadlock, starvation). The system language should be used to implement the core and low-level operations, such as memory allocation and synchronization, and the scripting language should be used to implement the high-level and configuration part, such as resource allocation policies.

Extensible operating systems frequently use programming-language resources to prevent extensions malfunctioning and guarantee system integrity. For example, Exokernel provides a collection of domain-specific extension languages to allow users to create their own disk and network abstractions; these languages are very restricted to prevent extensions from causing harm to the system (they are not Turing-complete languages). Singularity[13] provides tools for static code analysis, also to prevent extension violations.

Scripting languages usually provide protection features, such as type safety, automatic memory management and protected calls. Some scripting languages also allow the use of sandboxing techniques for protecting resources. As an example, Lua can restrict the set of available libraries, the amount of memory used and the number of instructions executed by each instance of its interpreter. This kind of sandboxing can also be used to enforce overall system integrity.

Another key feature of a scriptable operating system is ease of development. Scripting languages are in essence very high-level programming languages. Usually, they are dynamically typed and provide high-level data structures, operations and APIs. Some scripting languages are also extensible, allowing their customization to specific application domains. This customization permits the implementation of domain-specific languages, but with the advantage of sharing a common syntax and APIs among different domains. Scripting languages are also often used to promote productivity and are aimed at non-programmers too. Lua, for instance, is used as an extension language for Wikipedia, World of Warcraft, and Wireshark; in these applications Lua is typically used either by non-programmers or non-system programmers.

Effectiveness and efficiency of the kernel scripts are also extremely important for scriptable operating systems. OS kernels are specially sensitive to timing issues; therefore, kernel scripts must be reasonably efficient. They should not introduce an overhead that could compromise overall system performance. One of the main arguments for extending OS kernels is to avoid the overhead imposed by successive context switches between kernel and user space. Scripts that intend to benefit from running in kernel space must then not introduce an overhead that is higher than that imposed by context switching. Some modern scripting language interpreters are noticeably fast and have proven efficiency in benchmark tests. However, if efficiency is a real concern in a specific scenario, an strategy to improve scriptable systems efficiency is implementing performance critical tasks using the system's programming language and providing proper bindings for the scripting language.

Apart from performance issues, it is also important that kernel scripts be effective. Kernel scripts must be able to implement useful extensions. This requirement can be achieved by creating proper bindings between the scripting language and the kernel. To permit the creation of such bindings, the scripting language must be able to be both embedded and extended. It must be customizable to support kernel-suitable programming interfaces and data structures; it must also provide adequate interfaces to allow kernel execution flows to call script procedures and access script data structures. Bindings have also an important role to make scripts easier to program and preserve system integrity, because they provide the abstraction and protection levels for the APIs and data structures exposed to kernel scripts.

Creating proper bindings is one of the most difficult tasks when making an OS kernel scriptable. Proper bindings are those suitable for addressing the issues presented in this section. That is, bindings that satisfy the requirements of maintaining the integrity of the system, providing ease of development and enforcing effectiveness and efficiency of kernel scripts. OS kernels typically behave like a large set of complex and interdependent programs, so the difficulty of this task is inherent to the complex nature of these systems. However, this difficulty can be attenuated if the scripting language provides adequate resources for embedding and extending it.

3. Why Lua?

Lua is an extensible extension programming language, designed both to customize and be customized by applications [15, 17]. Unlike most scripting languages, Lua has been specifically designed as an *embedded* language: it is implemented as a regular C library with a well-defined API [14, 18].

The Lua C API provides a pragmatic way to bind scripts to a host program in both directions [18]. Through the Lua C API, a host application can get and set variables and call functions defined in a script; this ability is what makes Lua an extension language. The Lua C API also allows a host program to export functions and variables to a script, adding new facilities to the Lua environment; this ability is what makes Lua an extensible language.

Extending an operating system is somewhat different than extending a user-level application, because kernels are susceptible to a particular set of constraints. OS kernels are typically written in a limited subset of the C programming language. There is no support for floating-point types, because context switches of the floating-point unit are overly expensive. There is also no support for the entire C standard library; instead, there are only a few standard functions available. As an example, kernel code cannot use the traditional `free`/`malloc` functions, as they need the kernel to be implemented.

According to the ISO C standard [1], a *freestanding environment* is one that does not assume an operating system. Programs that comply with this environment have only a limited subset of the C standard headers available, namely `float.h`, `iso646.h`, `limits.h`, `stdarg.h`, `stdbool.h`, `stddef.h` and `stdint.h`. Ideally, operating system kernels and their extensions should be freestanding compliant. Moreover, because general-purpose operating systems are designed to run over several hardware platforms, an embedded language interpreter should not have platform dependent issues, such as endianness.

Because portability is one of the main design goals of Lua, its core is almost freestanding compliant, depending on few additional standard headers. These dependencies, however, can be easily tracked in the source code, because they are confined to a single configuration header file. Floating-point dependent code is also confined to this header file. Furthermore, all OS dependent code is placed outside the Lua core, in external libraries. The Lua core, for instance, does not link to C memory-allocation functions; it allocates memory by calling a function that is passed as an argument when creating a new interpreter state. Lua is written in pure ISO C [1] and has no hardware platform dependencies.

Another constraint on OS kernels is size. An operating system kernel remains loaded in memory from the moment the system is started until it is shut down. Therefore, kernel size is a relevant issue. Binary images of OS kernels usually have less than 5 MB (common Linux distributions have about 3 MB). Like the kernel itself, an embedded interpreter must also have a small size and be reasonably efficient.

Compared to other scripting languages, Lua has a very small footprint. The Lua 5.1.4 standalone interpreter, together with all Lua standard libraries, has 258 KB on Ubuntu 10.10; other languages, such as Python and Perl, have a few megabytes[1], the same order of magnitude of a full OS kernel.

Finally, because an OS kernel has unrestricted access to the entire hardware, special constraints should be put on kernel extensions to prevent damages or unwanted access to system resources. The embedded language must provide some means to isolate extension code and restrain its access to the kernel environment.

Lua provides programming support to enforce access restrictions upon scripts. Like most scripting languages, Lua has automatic memory management, which prevents scripts from directly manipulating memory through pointers. Lua also supports multiple state instances; its C code has no global variables at all. This implementation provides full state isolation, which in turn provides the

[1] On Ubuntu 10.10, Python 2.6.5 has 2.21 MB and Perl 5.10.1 has 1.17 MB.

means to isolate extensions from each other and from the kernel itself. Because different sets of libraries can be provided for those states, it is possible to create independent protection domains, with different privilege levels.

4. A Kernel-scripting Environment based on Lua

Our programming and execution environment for kernel scripting consists of four basic components: the Lua interpreter properly embedded in the kernel, for executing Lua scripts; a programming interface, used by kernel developers to make their subsystems scriptable; a user interface, for loading and running scripts in the kernel-embedded Lua interpreter; and Lua bindings, for sharing functions and data structures between the kernel and user-defined scripts. Figure 3 outlines the architecture and operation of this environment.

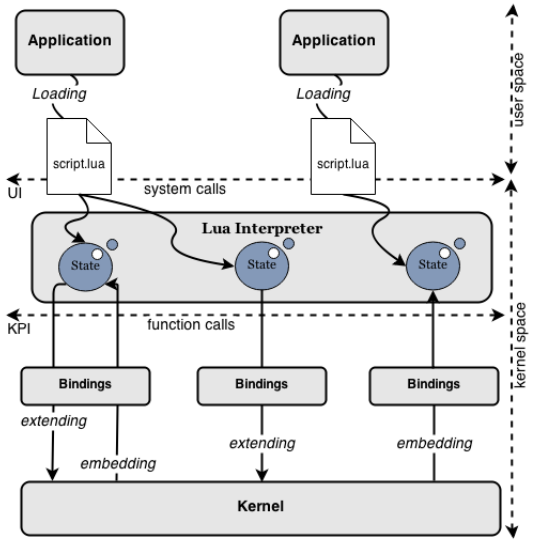

Figure 3. Architecture and operation of kernel scripting with Lua

4.1 Operation overview

Let us consider the scripting extension of the CPUfreq subsystem, discussed in Section 2.1. Suppose a user wants to activate the CPU frequency controller shown in Figure 1. To do so, she needs to load this script into the kernel-embedded Lua interpreter. To load her script, she uses the command-line tool provided by the User Interface (UI), which allows her to dynamically interact with the Lua interpreter.

Once the script has been loaded into the kernel, the CPUfreq subsystem uses its embedding binding to periodically call function `throttle`, which implements the user-defined policy. Besides allowing kernel subsystems to call Lua scripts, embedding bindings are also responsible for handling errors during the execution of those scripts. As an example, if the execution of function `throttle` fails, CPUfreq's embedding binding can invoke a default routine to process the frequency scaling.

To obtain current CPU temperature, function `throttle` uses an extension binding, which provides this information through function `acpi.get_temp`. An extension binding also allows the script to set the CPU frequency, through function `cpufreq.target`. Function `get_load` is a user-defined function which also uses an extension binding to obtain information to compute the current CPU load.

Kernel extension bindings are typically developed as loadable kernel modules (LKM), and are provided to scripts as regular Lua extension libraries, implemented in C. They can be loaded into kernel Lua states by the kernel subsystem (through the Kernel Programming Interface), by the User Interface (through a command), and also by the script itself (through function `require`, provided by the Lua standard library).

4.2 Kernel-embedded Lua

The main component of our kernel-scripting environment is the Lua interpreter, properly embedded in the OS kernel. Although some changes were necessary to embed Lua in the Linux and NetBSD kernels, all those changes were non-intrusive, involving only the modification of some macros in the Lua configuration header file and the replacement of some facilities from the C standard library that are not present in a kernel environment.

The most significant change we had to make concerned the use of floating-point types. As we discussed before, OS kernels do not provide support for floating-point types. We substituted the standard Lua number type, defined as `double`, for the integer type `intmax_t`; this change required only the redefinition of nine macros in file `luaconf.h`. We chose the integer type `intmax_t` for the convenience of having the largest integer type available in the underlying platform.

The Lua interpreter does not depend on the C standard library for memory allocation. Instead, it allows host programs to provide their own implementations of memory allocators. We implemented allocator functions for both Linux and NetBSD using the memory allocation primitives available in their kernels. Both memory allocation functions have less than eighteen lines of code.

Another change we had to make was the replacement of the pair of functions *setjmp*/*longjmp*, used by Lua for exception handling. We replaced these functions by equivalent functions available in the Linux and NetBSD kernels. This change required only the redefinition of three macros in file `luaconf.h`.

Besides the Lua interpreter, we also embedded the Lua basic library and some Lua standard libraries that do not depend entirely on operating systems resources or floating-point types (the *debug*, *coroutine*, *table* and *string* libraries). The only modifications we needed to make was the removal of some OS-dependent facilities from the basic and debug libraries, and the removal of floating-point formats from the string library.

4.3 User Interface

The User Interface (UI) has two parts: one running in user space and the other inside the kernel. The user-level component consists of a command-line tool and a *pseudo-device* descriptor file. The kernel component is the corresponding pseudo-device driver. The user-level tool resembles a Lua stand-alone interpreter, but instead of executing Lua scripts in user space, it executes them in the kernel-embedded Lua interpreter.

The user-level command interface is actually a front-end for the pseudo-device driver. When a user issues a command, the UI user-level component forwards it, by invoking an *ioctl* system call, to a handler function registered by the pseudo-device driver. This handler function, running inside the kernel, provides the actual commands for handling kernel Lua states and for loading and running scripts inside these states.

The pseudo-device driver allows only privileged access; that is, it processes only requests submitted by privileged users. Before processing any command submitted from user space, the handler function checks the user credentials. If the user has administrative privileges, the command is processed; otherwise, an access error is returned.

4.4 Kernel Programming Interface

To make a kernel subsystem scriptable, a developer needs to implement proper bindings for both extending and embedding Lua inside the kernel. In both cases, she will use the Lua C API [16] and the Kernel Programming Interface (KPI), which is implemented by the pseudo-device driver presented in the previous section. Among other facilities, the KPI provides functions to create and destroy Lua states, to synchronize the access to these states, and to load extension libraries into them.

In our scripting environment, Lua scripts are loaded into *kernel Lua states*, created through the KPI. Those states are similar to Lua states created with the regular Lua C API, but they are adapted to the kernel environment, which requires synchronization primitives and the use of special memory allocators.

When they are created, new kernel Lua states are registered in the pseudo-device driver, so that they can be accessed from user space, through UI commands. Because the system-call handler function and the kernel subsystem execute in concurrent flows, their access to kernel Lua states must be coordinated with synchronization mechanisms. Synchronization is also necessary if a Lua state can be accessed by concurrent control flows inside a subsystem or shared among different subsystems.

The synchronization of kernel Lua states is based on mutual exclusion mechanisms provided by the underlying OS kernel. Besides storing a regular Lua execution state, a kernel Lua state has also a mutual exclusion handle. Synchronized access to a state is enforced by programming discipline: *locking* and *unlocking* the state, respectively, before and after using it. The KPI provides synchronization primitives for both operations.

Extension bindings must be registered in the pseudo-driver to permit Lua scripts to dynamically load them through function `require`, like regular Lua libraries. The KPI provides this registering operation.

5. Evaluation

One of the main reasons for the large adoption of scripting languages in several domains is based on a tradeoff between systems performance and programmers performance. Because of this tradeoff, it is hard to do an objective evaluation of the alleged productivity gains we can get from the use of scripting languages [28].

An objective evaluation of our system suffers the same difficulty. Therefore, in this section we will not argue that kernel scripting is better (or worse) than other forms of extensions for operating systems, but instead we will argue that kernel scripting is a viable form of extension, comparable to its alternatives.

To evaluate our kernel-scripting environment, we implemented extensions for two kernel subsystems: the Linux subsystem CPUfreq and the NetBSD subsystem NPF (a packet filter). We chose these two subsystems because they have some favorable characteristics for scriptable extensions. First, they already provide support for extension modules, which can notably benefit from running inside the kernel. Second, they do not provide OS core services, such as process scheduling. Therefore, we could develop our extensions in a non-intrusive way, without modifying core components of the OS kernel.

We based our evaluation on the following questions, derived from the requirements discussed in Section 2.2:

Usefulness: do the scripts implement useful extensions?

Simplicity: are the scripts easily developed?

Performance: do the scripts perform adequately? Do they compromise the overall system performance?

Reliability: do the scripts compromise the system reliability?

Our first experiment was the extension of the Linux CPUfreq subsystem. CPUfreq supports the dynamic addition of new CPU frequency controllers in the form of loadable kernel modules. We used this feature to develop a CPUfreq module that supports the dynamic addition of new frequency controllers in the form of kernel-loadable user scripts. Our CPUfreq module implements both embedding and extension bindings: it calls periodically a Lua function that adjusts the CPU frequency, and also exposes the necessary kernel resources to the Lua code.

The main reason for executing CPUfreq controllers in kernel space is to react promptly to the need of changing the CPU frequency. Such a requirement is difficult to achieve if the controller executes in user space. As an example, the CPUfreq built-in controller Ondemand adjusts the CPU frequency according to the CPU load. To react quickly when more processing power is required, the Ondemand controller sets its rescheduling interval to a value that is two hundred times the latency for changing the CPU frequency. In our experiment, we used an Intel Pentium M CPU 1.6 GHz, leading to scheduling intervals around 20 ms.

We implemented a Lua script version of Ondemand, *Ondemand.lua*, using the same rescheduling interval set by the original controller. The average execution time of the Lua version was 8 μs, which is 0.04% of its rescheduling interval. Thus, we did not introduce a measurable overhead using a CPU frequency controller implemented in Lua.

Our Lua script version of Ondemand has a total of 47 lines of Lua code (including the Lua function `get_load`, which uses the extension binding to access CPU load information). Although it is a simplified version of the original controller, it has less than 8% of the amount of lines of that controller (618 lines of C code[2]). CPUfreq's Lua bindings have 138 lines of C code, which makes our whole implementation less than 30% of the original controller's size.

The Lua version of Ondemand can also be configured at runtime. For instance, using the UI interface, a user can load a script that modifies the value of the global variable `up`, changing the CPU load threshold above which the CPU frequency is risen. She can also redefine function `get_load` to only consider the CPU time spent by user processes. In our experiment, we used a Lua version of the Ondemand policy to be able to compare it with the original CPUfreq built-in controller. However, different useful extensions can also be developed, such as the overheating-prevention script discussed in Section 2.1.

Like CPUfreq, the NetBSD NPF subsystem supports the dynamic addition of new packet-filtering rule procedures in the form of loadable kernel modules. In our second experiment we developed a NPF module that supports the addition of new packet-filtering rules in the form of kernel-loadable user scripts. Our NPF module implements an embedding binding that allows the module to call a Lua function to filter network traffic; its extension binding exposes the necessary kernel resources to the Lua code.

Packet filters applied to layers two to four are typically implemented in kernel space to not impose inadequate latency to network applications. By implementing a filtering facility through a kernel-loadable script, we can satisfy this low-latency requirement for higher-layers inspections too. Our NPF extension script filters SSH traffic to block access to vulnerable servers; it was shown in Figure 2, in Section 2.1. We executed this packet-filtering script in a virtual machine running on an Intel Core i3 CPU 3.10 GHz. A 100 Mbps virtual network connected the virtual machine to its host. Using Iperf [10], a TCP bandwidth measurement tool, we measured the network bandwidth in two scenarios: with and without the script, The average bandwidth for both cases was around

[2] On Linux kernel 2.6.25.

96 Mbps. Thus, we did not introduce a measurable overhead by using a packet-filtering rule implemented in Lua.

The SSH protocol filtering script has 22 lines of Lua code. We cannot implement an equivalent filtering facility using only NPF rules. The embedding binding has around 200 lines of C code. Our filtering script uses Luadata, a Lua extension library specially developed for our kernel-scripting environment. Luadata exposes kernel memory safely to Lua code, and allows Lua scripts to apply *data layouts* to memory blocks, so that they can access delimited fields inside those blocks. The use of a full-fledged language, along with adequate libraries (such as the Lua string library, and Luadata) allowed us to easily implement the SSH-filtering facility.

Our kernel-scripting environment allows only privileged users to load and run extension scripts inside the kernel. Thus, kernel scripts cannot compromise system integrity more than loadable kernel modules do. Our environment also limit the number of instructions executed by the Lua interpreter, thus preventing an extension script to monopolize kernel execution time. The embedding bindings create individual Lua execution states, providing isolation among extensions. Extension scripts are also sandboxed; that is, they can use only a restricted set of extension bindings.

Independent research groups have used Lunatik, our first implementation of a kernel-scripting environment based on Lua, to experiment with OS kernel extensions. The Computer Networks Research Group at the University of Basel developed bindings for the Linux subsystem Netfilter to allow users to implement facilities for processing network packets using Lua scripts [11]. In their experiments, they implemented network address translation (NAT) with Lua, using a router running on an Intel Celeron CPU 2.4 GHz and a 100 Mbps local network, and measured the maximum throughput of both their NAT implementation using Lua and the Linux built-in NAT implementation. For both cases, the measured throughput was around 90 Mbps. Their analysis also showed that the Lua script and the built-in implementation had approximately the same latency and saturation interval. Their NAT implementation has 19 lines of Lua code.

Research groups at the University of Paderborn and at the University of Mainz used Lunatik to introduce scripting facilities into the pNFS file system, allowing pNFS clients to implement file layouts using Lua for configuring storage strategies [12]. They measured the execution time of several file-layout scripts using an Intel Xeon CPU 3.30 GHz. The average execution time was 1 μs for the simplest script and 8 μs for a full-featured one. In their experiment, a simple file-layout script had only 8 lines of Lua code.

Both research groups enforced system reliability by sandboxing; that is, through restricting the extension bindings exposed to the kernel scripts.

6. Related Work

Many extensible operating systems use programming language resources to provide extensibility. Some of them, such as SPIN [6] and Singularity [13], use system languages; SPIN uses a subset of Modula-3 and Singularity uses an extension of C#. This is also the case of most conventional operating systems, which usually use C for their loadable kernel modules. Some systems use instead a set of domain-specific languages; one such system is Exokernel [8], which uses restricted languages suitable for specific tasks such as creating hard disk and network abstractions. This is also the case of some conventional operating systems, which use DSLs for providing packet filtering.

What distinguishes our approach from both kinds of extensible systems is the level of extensibility provided. Extending the operating system kernel through scripting stands halfway between providing limited domain-specific languages and providing a full-featured system programming language. Because the level of extensibility is closely related to the issues we discussed in Section 2.2, the choice of a language for extending the OS kernel is a trade-off among these factors.

When compared to system languages, scripting languages are usually easier for developing extensions and for enforcing protection. However, they are also usually less useful and efficient. These downsides can be mitigated, respectively, by providing proper bindings and by applying optimization techniques.

When compared to some DSLs, scripting languages are usually more useful and efficient. On the other hand, DSLs can be easier for developing extensions and for enforcing protection. These downsides can be mitigated with the use of proper bindings and by applying sandboxing techniques.

Scripting languages have also the advantage of providing a common language core that can be used in many different domains. Once a kernel-scripting environment has been provided to support some extension (e.g., a packet filtering facility), it can be reused in several other scenarios, both for writing other extensions — device drivers, network protocols, disk abstractions — and for experimentation. Moreover, having only one extension language engine facilitates the task of guaranteeing system integrity.

Besides extensible systems that use system languages and those that use domain-specific languages, there is also an extensible system that actually uses a scripting language. The μChoices operating system [7] provides a scripting language similar to Tcl for writing kernel extensions, allowing users to load and run scripts inside its kernel. Using scripts, users can aggregate system calls in batches to avoid context switches and thus improve system performance. Extension scripts are loaded into instances of a kernel-embedded Tcl interpreter that execute in independent system processes [22]. A set of extension bindings exposes to the scripts the necessary resources for extending the kernel. What distinguishes our approach from μChoices is our support of scripting by *embedding* the language interpreter, in addition to scripting by *extending* it.

Finally, another important point that distinguishes our approach from most previous extensible operating systems is that we have been focusing on extending existing general-purpose operating systems through kernel scripting, instead of implementing a whole scriptable operating system from scratch.

7. Conclusions

In this paper we presented our concept of a *scriptable operating system*, which results from applying the idea of extensibility through scripting to the concept of extensible operating systems. Based on this concept, we developed a scripting environment that allows the extension of kernel subsystems through user scripts, dynamically loaded and executed inside the kernel.

Our kernel-scripting environment uses Lua, a popular scripting language, with minimal changes. The ease of embedding Lua in both Linux and NetBSD kernels attested its notable portability, showing that it can be used even in hostile environments such as OS kernels.

Previous works have already explored the idea of extending an OS through user scripts; however, most of them provide restricted domain-specific languages, suitable for specific tasks. Our environment, instead, provides a general-purpose, full-fledged programming language, which not only allows the development of more sophisticated extensions but also provides an interesting programming environment for kernel developers themselves. As far as we know, it is also the first environment that provides kernel extensibility by both embedding and extending a scripting language. Moreover, we worked on top of existing general-purpose operating systems, not on systems initially designed for scripting.

We have already implemented our scripting environment for two general-purpose operating systems: Linux and NetBSD. Our

NetBSD implementation is now part of the official NetBSD distribution. Our first Linux implementation, Lunatik, has been used by different research groups to support the extension of kernel subsystems [11, 12].

The positive evaluation of our kernel-scripting experiments showed that our proposal of providing OS extensibility through scripting based on Lua is a viable and interesting approach for extending OS kernels. We believe that this approach can help innovation in OS development. First, it delivers a higher-level programming environment to the kernel, providing support for easy and agile prototyping and experimentation. Second, it allows application developers, who typically have no kernel-programming skills, to experiment with the OS kernel for their own needs.

Acknowledgments

Lua in the NetBSD kernel was partially developed as a Google Summer of Code project, sponsored by The NetBSD Foundation. We also thank the NetBSD developers for all given support. During the development of Lunatik, Lourival Vieira Neto had a grant from CAPES (the Brazilian Agency for the Improvement of Higher Education) and Roberto Ierusalimschy a grant from CNPq (the Brazilian Research Council).

References

[1] ISO C standard 1999. Technical report, 1999. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`. ISO/IEC 9899:1999 draft.

[2] IEEE Standard for Local and Metropolitan Area Networks—Station and Media Access Control Connectivity Discovery. *IEEE Standard 802.1AB*, 2009.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999.

[4] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming os schedulers with domain-specific languages and aspects: New approaches for os kernel engineering. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6. Citeseer, 2002.

[5] M. Belshe and R. Peon. SPDY protocol—Draft 3.1. URL `http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1`.

[6] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN—an extensible microkernel for application-specific operating system services. *ACM SIGOPS Operating Systems Review*, 29(1):74–77, 1995. ISSN 0163-5980.

[7] R. H. Campbell and S.-M. Tan. μChoices: an object-oriented multimedia operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, 1995 (HotOS-V)*, pages 90–94. IEEE, 1995.

[8] D. Engler. *The Exokernel operating system architecture*. PhD thesis, MIT, 1998.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1, 1999.

[10] M. Gates, A. Warshavsky, A. Tirumala, J. Ferguson, Jim Dugan, F. Qin, K. Gibbs, J. Estabrook, A. Gallatin, S. Hemminger, J. Nathan, and G. Renker. Iperf, 2008. URL `http://iperf.sourceforge.net`.

[11] A. Graf. PacketScript—a Lua Scripting Engine for in-Kernel Packet Processing. Master's thesis, Computer Science Department, University of Basel, July 2010.

[12] M. Grawinkel, T. Suss, G. Best, I. Popov, and A. Brinkmann. Towards Dynamic Scripted pNFS Layouts. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 13–17. IEEE, 2012.

[13] G. Hunt and J. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007. ISSN 0163-5980.

[14] R. Ierusalimschy. *Programming in Lua*. Lua.org, third edition, 2013.

[15] R. Ierusalimschy, L. de Figueiredo, and W. Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[16] R. Ierusalimschy, L. de Figueiredo, and W. Celes. *Lua 5.1 Reference Manual*. Lua.org, first edition, 2006.

[17] R. Ierusalimschy, L. de Figueiredo, and W. Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 2–26. ACM, 2007.

[18] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Passing a language through the eye of a needle. *Communications of the ACM*, 54(7):38–43, 2011.

[19] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. In *Proceedings of the 7th Workshop on Systems Support for Worldwide Applications (ACM SIGOPS European workshop)*, pages 141–148. ACM, 1996.

[20] B. Lampson. On reliable and extendable operating systems. In *Proceedings of the Second NATO Conference on Techiniques in Software Engineering*, 1969.

[21] G. Lefkowitz. Extending vs. embedding—there is only one correct decision, 2003. URL `http://twistedmatrix.com/users/glyph/rant/extendit.html`.

[22] Y. Li, S.-m. Tan, M. L. Sefika, R. H. Campbell, and W. S. Liao. Dynamic Customization in the μChoices Operating System. In *Proceedings of Reflection'96*, 1996.

[23] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[24] P. Mochel and M. Murphy. sysfs—The filesystem for exporting kernel objects, 2009. URL `http://kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[25] H. Muhammad and R. Ierusalimschy. C APIs in extension and extensible languages. *Journal of Universal Computer Science*, 13(6):839–853, 2007.

[26] J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[27] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230, 2006.

[28] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[29] M. Rasiukevicius. NPF—Progress and Perspective. *AsiaBSDCon 2014*, page 21, 2014.

[30] S. Savage and B. Bershad. Issues in the design of an extensible operating system. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[31] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, 2003.

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, volume 96, pages 213–227, 1996.

[33] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Issues in extensible operating systems. *Computer Science Technical Report TR-18-97, Harvard University*, 2(2.2):1, 1997.

[34] C. Small and M. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical report, 1994.

[35] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *USENIX Annual Technical Conference, General Track*, pages 189–202, 2001.

[36] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol, 2006. URL `http://www.ietf.org/rfc/rfc4253.txt`.