Running Application Specific Kernel Code by a Just-In-Time Compiler

Ake Koomsin
University of Tsukuba
ake@softlab.cs.tsukuba.ac.jp

Yasushi Shinjo
University of Tsukuba
yas@cs.tsukuba.ac.jp

ABSTRACT
Kernel scripting is a technique to run an extension code in a script language in an operating system kernel. Conventional kernel scripting has two limitations. First, it affects an entire system and only privileged users are allowed to install a new script. This prohibits developers from running their own application-specific code in the kernel. Second, its performance is not sufficient for some time-sensitive applications. In this paper, we address these problems. Our system call scripting allows developers to run their own application-specific code in the kernel without the root privilege. Our system call scripting runs with less overhead because we use a Just-In-Time (JIT) compiler. To evaluate our idea, we ported the LuaJIT compiler into the FreeBSD 10.1 x86 kernel. We modified Memcached to use system call scripting that processes multiple UDP GET requests at a time. With one worker thread and under a high-load condition, we achieved a 33% reduction in the average response time and a 44% improvement in the throughput when the response value size was small.

Categories and Subject Descriptors
D.3.3 [**Language Constructs and Features**]: Frameworks; D.4.7 [**Organization and Design**]; D.4.8 [**Performance**]

General Terms
Design, Performance

Keywords
System call scripting, Scriptable operating systems, Kernel scripting, Lua programming language, JIT compilers

1. INTRODUCTION
Kernel scripting is a technique to run an extension code in a script language in an operating system kernel [18]. This allows users to install scripts that, for example, control the CPU frequency and extend the functionalities of packet filters.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PLOS'15, October 04-07, 2015, Monterey, CA, USA
© 2015 ACM. ISBN 978-1-4503-3942-1/15/10...$15.00
DOI: http://dx.doi.org/10.1145/2818302.2818305

Kernel scripting should be a good tool for adding application-specific code and kernel specialization. Since a typical kernel provides general services for typical applications, it cannot always meet the requirements of some applications, such as I/O-based applications. Kernel specialization can solve this problem. For example, if we can add specialized lightweight system calls for a network cache server, we can fully utilize the hardware capacity. Specialization should be easy and will result in performance improvement compared with using the standard interfaces.

However, conventional kernel scripting has two limitations. First, it affects an entire system and only privileged users are allowed to install a new script. This prohibits developers from running their own application-specific code in the kernel. Second, its performance is not sufficient for some time-sensitive applications. This is because a script is executed by an interpreter.

In this paper, we address these problems. We extend the concept of kernel scripting and propose *system call scripting.* Our system call scripting allows developers to run their own application-specific code in the kernel without the root privilege. Our system call scripting runs with less overhead because we use a Just-In-Time (JIT) compiler.

To evaluate our idea, we ported the LuaJIT compiler into the FreeBSD 10.1 x86 kernel. We modified Memcached to use system call scripting that processes multiple UDP GET requests at a time. With one worker thread and under a high-load condition, we achieved a 33% reduction in the average response time and a 44% improvement in the throughput when the response size was small.

The rest of this paper is organized as follows: Section 2 talks about Lua, LuaJIT and system call scripting in Lua. Section 3 evaluates our approach by using the results on Memcached. Section 4 discusses the related work. We conclude this paper in Section 5.

2. APPROACH
We used the Lua scripting language and the LuaJIT compiler to run application-specific kernel services. We modified the LuaJIT compiler to be able to run in the kernel space and make it a loadable kernel module. These infrastructures provide developers with the ability to create their own system calls on the basis of the existing ones. We call it system call scripting.

2.1 Introduction to Lua and LuaJIT

We chose the Lua language for running application-specific code in a kernel for the following reasons: First, it is a small scripting language designed to be embeddable into another application. It is commonly used in game engines. Many applications such as Wireshark, Redis and MySQL Proxy also support Lua scripting to provide additional functionalities. Second, Lua has a JIT compiler. Developers often supply scripts to the system which are in hot execution paths. We intend to take advantage of the JIT compiler to reduce the interpreter overhead.

LuaJIT[12] is a tracing JIT compiler of the Lua language. When a code path is repeatedly executed for a number of times, the tracing will start. During the tracing, the bytecodes are recorded and translated into Static Single Assignment Intermediate Representations (SSA IRs) [4]. After tracing is done, LuaJIT carries out some optimizations such as dead code elimination and constant folding. Finally, these optimized SSA IRs are translated into a native code. When the code path is going to be executed again, the interpreter jumps to the compiled code instead of interpreting the bytecodes.

When integrating into an application, the Lua engine and the application communicate with each other with what is called a "Lua state". This is also the place where the computation is done. A Lua state is a kind of stack. A Lua script and its arguments are pushed into the Lua state. Once the execution is done, the result will be at the top of the stack to be used by the application it is integrated into.

2.2 System call scripting

Writing application-specific system calls should be easy for developers. To realize this concept, we propose the following two system calls.

1. `int register_lua_syscall(char *script, size_t size)`
2. `int lua_syscall(int lua_fd, unsigned int num_args, int arg1, ..., int arg8)`

The idea behind these system calls is that it is possible to encapsulate a Lua state in a file descriptor. The first system call creates a Lua state, registers the necessary bindings, loads the supplied script into the Lua state and returns a file descriptor that points to the Lua state. The next system call takes a file descriptor that points to a Lua state and executes it with the given arguments. If the script is successfully executed, developers can return values to the user space. The current implementation limits the number of arguments to eight. This system call looks for the `run()` function from the supplied script and executes the function. Since a Lua state is represented by a file descriptor, we can use the existing `close()` system call for cleaning up.

Figure 1 shows the script that is used for receiving multiple UDP packets. We will describe the details in Section 3. The script takes six arguments namely a socket file descriptor, a read buffer address, a `struct sockaddr` array address, a

```
1  function run(sfd, buf_addr, sockaddr_addr,
       sockaddr_len_addr, recv_uaddr, max_batch)
2    local recvfrom = syscall.recvfrom
3    local copyout = util.copyout
4    local nrecv_array = {}
5
6    local nreq = 0
7    while nreq < max_batch do
8      -- 1024 is the size of buffer
9      -- 16 is the size of struct sockaddr
10     -- 4 is the size of socklen_t
11     local buf_offset = nreq * 1024
12     local sockaddr_offset = nreq * 28
13     local sockaddr_len_offset = nreq * 4
14     local byte_recv = recvfrom(sfd,
15       buf_addr + buf_offset,
16       1024, 0,
17       sockaddr_addr + sockaddr_offset,
18       sockaddr_len_addr + sockaddr_len_offset)
19     if byte_recv > 8 then
20       nreq = nreq + 1
21       nrecv_array[nreq] = byte_recv
22     elseif byte_recv > 0 then
23       -- do nothing
24     else
25       if byte_recv == -4 then
26         -- -4 is EINTR
27         -- do nothing
28       elseif byte_recv == -35 then
29         -- -35 is EAGAIN
30         break
31       else
32         return -1
33       end
34     end
35   end
36
37   if nreq > 0 then
38     copyout(nrecv_array, recv_uaddr, nreq)
39   end
40
41   return nreq
42 end
```

Figure 1: Script for receiving multiple UDP packets

`sockaddr` length array address, an array to hold bytes received and the maximum number of batching. These values are all from the user space. Lines 11 to 13 are pointer arithmetic. Line 14 calls the system call function `recvfrom()`. If the data we receive are more than 8 bytes, we record the value of the bytes received and try to receive more data. If the script encounters `EAGAIN` which means no data, the operation stops. If it receives at least a request, it copies the array that stores the bytes received to where `recv_uaddr` points to and returns the number of incoming requests.

2.3 Safe execution of Lua scripts in a kernel

Figure 2 illustrates how we restrict an execution of a script in a kernel. An arrow indicates a memory access. The scope of a script of a non-privileged user is a process. It does not interfere with other processes. A Lua script running in a kernel cannot access the kernel memory directly. It can only access the kernel memory and user process memory through bindings. The script in Figure 1 calls `copyout()`. The source address of this function is limited to that of a table variable in a Lua script. The destination address of this function is limited to the user process that invokes `lua_syscall()`.

Next, we provide safe bindings for non-privileged users. We implement a system call module for the LuaJIT which con-

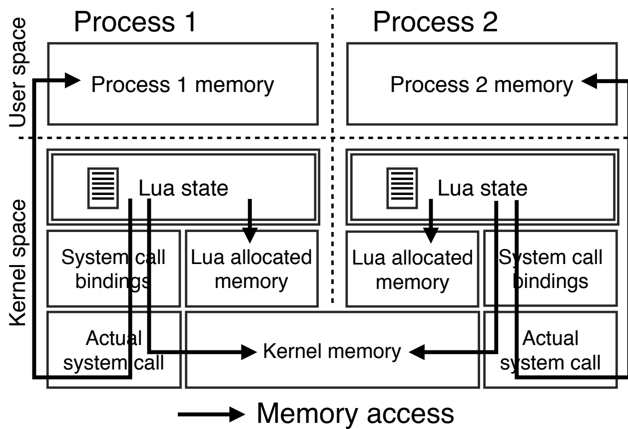

Figure 2: Restricted execution environment

tains system calls bindings such as `recv()` family and `send()` family. Further, we disallow user land programs to add new bindings.

Our bindings simply pass the provided arguments to the actual system call functions, for example, `sys_recvfrom()`. This also applies to the JIT compiler; it will just produce codes that call the bindings. This means that the regular protection mechanism of the operating system kernel works well for the safe execution of a script. While a script runs in a kernel, its memory access is limited. For example, if a script passes an illegal address to `sys_recvfrom()`, the unmodified `sys_recvfrom()` performs regular checking in the `copyout()` function and returns `EFAULT` to the script.

To prevent resource exhaustion, in the current implementation, we modified the memory allocation function to check whether the total memory exceeds a certain limit. If this happens, the process is terminated. We also make use of the FreeBSD's `callout` API [7] to limit an execution time of a script. If the script exceeds the time limit, the process is terminated.

2.4 Experience of modifying LuaJIT

The environment in the kernel land is different from that in the user land. This section gives you a brief overview of how we modified the LuaJIT compiler to make it runnable in the kernel as a kernel module.

Parsing The power operator was removed as it was not necessary. The number parser was modified to avoid generating floating point values.

Arithmetic Arithmetic-related functions, such as constant folding, were modified to handle only integers. Division and modulo operators were modified to support only integers and follow the behavior of C language.

Memory allocation LuaJIT has its own custom memory allocator. It was replaced with the kernel memory allocator.

Bytecode interpreter This is the trickiest part as it is written in an assembly language. The interpreter uses floating point operations and registers by default. We modified its logic and code flow to perform operations on integers and use only general-propose registers.

JIT compiler Like the interpreter, the JIT compiler emits floating related SSA IRs by default. We ensured that SSA IRs did not contain floating-related operations.

In addition, we implemented the system call bindings module as described in Section 2.3. However, this is not sufficient as these bindings cannot be compiled by the JIT compiler automatically. We added recording functions, which are called during tracing, to record system call invocations.

The current limitation of our modification is that it works only with the x86 kernel because of the LuaJIT internals. While LuaJIT can run on x86-64 operating systems, it uses 32-bit pointers in many places internally.

3. EVALUATION

3.1 Experimental setup

We conducted an evaluation on Memcached [6], a distributed memory object caching system. We were interested in the scenario when GET requests are sent over UDP and SET requests are sent over TCP. This scenario is used by Facebook [11]. A request over UDP is represented by one UDP packet. The original Memcached processes one UDP packet at a time.

With our system call scripting, it is possible to receive multiple UDP packets or send multiple UDP packets in one mode switch. We modified Memcached to use our system call scripting that receives multiple GET requests over UDP and sends their replies at a time. One of the scripts we use is shown in Figure 1.

We used Memaslap [19] to generate requests. We modified it to support the Facebook Test (multiple key GET requests over UDP and SET requests over TCP) with a single key. We made it run on four threads with 128 concurrencies to generate a large load on Memcached. The ratio of GET-to-SET requests was set to 9:1.

Both the original Memcached and our modified Memcached were run with a single worker thread. They were fixed to run on a CPU core. This was to avoid the effect of lock contention and thread scheduling. Their memory object size was set to 2 GB as we were interested in a no cache-miss situation. We configured our Memcached to process at most 16 requests at a time.

The Memcached and Memaslap were run on different machines. Table 1 shows the specifications of these machines. Hyperthreading and features such as TurboBoost were disabled on both machines. These machines were connected with a gigabit Ethernet cable. There were no other machines on the network.

We ran two tests in this evaluation. The first one measured the average response time. In this test, Memaslap was set to execute one million requests. The other test measured the number of transactions per second (TPS). In this test,

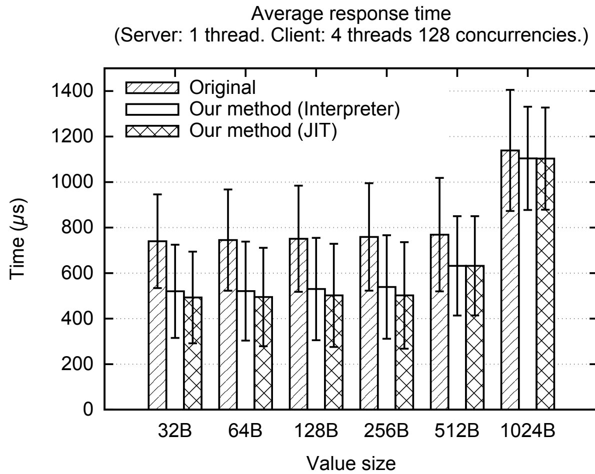

Figure 3: Average response time of GET requests

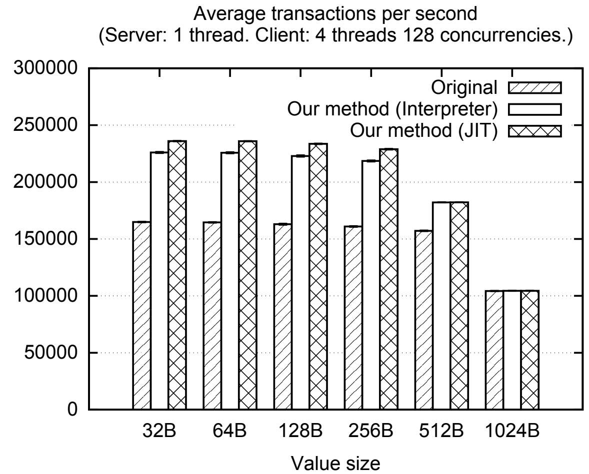

Figure 4: Transactions per second of GET requests

Memaslap was set to execute requests for a period of 30 seconds.

3.2 Experimental results

Figure 3 and Figure 4 show the average response time and the number of transactions per second of GET requests with various value sizes respectively. The error bars in these figures denote the standard deviation (Note that the standard deviation of Figure 4 is very low compared to the value). Both graphs show the results of the original Memcached, our Memcached when running with only the LuaJIT interpreter and our Memcached when the JIT compiler was enabled.

By comparing the original Memcached and our Memcached running with only the interpreter, we found that system call scripting reduced the average response time by about 30% and increased the throughput by about 37% when the value size was small. When the JIT compiler was enabled, our method further reduced the average response time by about 33% and increased the throughput by about 44% as compared to the original. As the value size increased to 1024 bytes, we found no significant difference between the original method and our method. This was attributed to the system limit.

We achieved these improvements because, under a high-load situation, there were many UDP packets waiting in the socket buffer. Our method allows draining the socket buffer. This makes requests spend less time on waiting in the socket buffer.

We can also see the effect of the JIT compiler in action in both Figure 3 and Figure 4. With the JIT compiler, we found a 3% reduction in the average response time and a 7% improvement in the throughput when the value size was small as compared to case with only the interpreter. Although these look like small improvements, they are important for latency-sensitive applications such as Memcached. Reduction in the average response time means a better user experience as a Memcached client, such as HTTP application servers, spending less time waiting for responses.

Table 1: Server and client specifications

	Server	Client
CPU	Intel i7 3820 3.6 GHz	Intel i7 950 3.06 GHz
RAM	32 GB	12 GB
NIC	Intel Ethernet Server I340 T4	
OS	FreeBSD 10.1 x86	Ubuntu 14.04 x86-64

3.3 Simplicity and usefulness

The script shown in Figure 1 is fairly straightforward to write. For those who have experience in system programming, it should not be difficult to write such code. The trickiest part is the pointer arithmetic as seen in Lines 11 to 13. This is attributed to the fact that the Lua environment treats these addresses as normal integers. It has no knowledge about C data structures. We would like to provide a better solution in a future.

Since we wanted to keep the modifications to Memcached to a minimum, we implemented only the batch-receiving script and the batch-sending script. Memcached heavily relies on libevent [13]. It is a wrapper of event notification APIs such as `epoll`, `select` and `kqueue`. Many applications rely on it for portability reasons. We did not implement other combinations such as `kevent()` and `recvfrom()`. Doing so would require a large number of changes in Memcached.

System call scripting affects the way an application is written. In case of Memcached, we had to make changes in some data structures, functions and control flows to support the processing of multiple UDP GET requests. For instance, Memcached's `conn` object contains fields that are related to receiving and responding requests. To support the processing of multiple requests, we had to decouple these fields from the `conn` object. This was a little challenging for the application. On the other hand, an application designed from scratch should have no problem doing so.

4. RELATED WORK

4.1 System call batching

System call clustering is the idea of batching a set of system calls and performing them sequentially in just one mode switch [14]. System call clustering relies on a simple system call extension `multi_call()` that sequentially performs all system calls it receives as arguments. It relies on compiler optimization that analyzes system call flows and groups them into clusters. After the analysis, it transforms each cluster into a single `multi_call()` system call.

Our approach can achieve system call clustering in a simpler way. With system call scripting, developers can create their own system calls explicitly on the basis of the existing system calls in a scripting language. In addition, our approach allows for more flexible error handling. For example, when bytes returned from `recvfrom()` on a UDP socket are fewer than what an application expects, our approach can ignore the received data without going back to the user space. System call clustering cannot handle this situation.

Another approach to batch system calls is to make system calls asynchronous. Syslet [10] was proposed for the Linux kernel to support system call batching in an asynchronous manner. A syslet consists of a chain of system calls. It supports branching and looping. Once a syslet is submitted, it will be executed asynchronously. While Syslet still relies on exception, FlexSC [17] eliminates this by separating system call invocation from execution and system call memory pages. FlexSC does not only avoid using exceptions which cause cache pollution, it also makes system call batching possible.

Rather than going asynchronous, we explore an alternative approach by making system call batching possible while still maintaining the traditional system call usage. We think that this approach makes it easier to understand what is going on as the traditional system call usage is well understood. The Memcached we modified is still event-based with non-blocking I/O. In Figure 1, the `sfd` file descriptor is non-blocking. The script is called after `kevent()` returns.

Linux provides `recvmmsg()` and `sendmmsg()` that allow the receiving and sending of multiple messages respectively in a mode switch [15, 16]. These have performance benefits for some applications. FreeBSD and NetBSD are now porting these system calls [3, 9]. With our approach, we demonstrate that it is possible to create the same functions using `recvmsg()` and `sendmsg()` without implementing new system calls.

4.2 Extensible operating systems

An extensible operating system is an operating system that can improve its flexibility by allowing the use of extensions [2, 5, 8]. For example, SPIN Operating System [1] is an operating system which applications are allowed to register hooks to parts of the operating system in order to override the default behaviors of the system for their own requirements. This research shares ideas that are similar to ours. Its approach uses the compiled language Modula-3 while we use the scripting language Lua backed with a JIT compiler. SPIN allows multiple applications to register hooks to a part of the system which can cause an overhead of determining which hook to run. On the other hand, we allow an application to run specialized code that affects a single application.

4.3 Scriptable Operating Systems with Lua

As mentioned in Section 1, Scriptable Operating System with Lua [18] is the idea of using the Lua scripting language to write extensions on an operating system. It exposes the Lua engine as pseudo-devices. The research uses the original Lua interpreter. It allows an easy implementation the scripting facility on both Linux and NetBSD.

While the research focuses on making changes that have system-wide effects, our research extends the idea and focuses on making application-scope kernel extension. To make the Lua engine available only in an application, we encapsulate the execution environment in a file descriptor. We also experiment with the idea of an in-kernel JIT compiler to reduce interpretation overhead by porting the LuaJIT into the FreeBSD 10.1 x86 kernel.

5. CONCLUSION AND FUTURE WORK

We have extended the kernel scripting mechanism and proposed system call scripting that allows developers to create their own application-specific system calls on the basis of the existing ones. This allows the execution of multiple system calls in one mode switch without root permission. We modified the LuaJIT compiler to run as a loadable kernel module to provide the scripting engine in the FreeBSD 10.1 x86. We evaluated our idea using Memcached. We modified Memcached to be able to process multiple UDP GET requests at a time by using system call scripting. We found that, under a single worker thread and a high-load condition, our approach with an interpreter reduced the average response time by up to 30% and improved the throughput by up to 37% when the response value size was small. With the JIT compiler, our approach reduced the average response time by up to 33% and increased the throughput by up to 44%. These improvements were done with two simple scripts and minimal modifications to Memcached.

In the future, we intend to explore the application-specific network stack specialization by using our scripting approach and FreeBSD's Jails virtual network stack.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility Safety and Performance in the SPIN Operating System". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 1995, pp. 267–283.

[2] R. H. Campbell and See-Mong Tan. "μChoices: An Object-oriented Multimedia Operating System". In: *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*. 1995, pp. 90–94.

[3] *Changes from NetBSD 6.0 to NetBSD 7.0*. URL: `http://ftp.netbsd.org/pub/NetBSD/NetBSD-7.0_RC3/CHANGES`.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490.

[5] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr. “Exokernel: An Operating System Architecture for Application-level Resource Management”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP ’95)*. 1995, pp. 251–266.

[6] Brad Fitzpatrick and Anatoly Vorobey. *Memcached - a distributed memory object caching system*. URL: `http://memcached.org`.

[7] *FreeBSD’s callout API*. URL: `http://www.freebsd.org/cgi/man.cgi?query=timeout&sektion=9`.

[8] Galen C. Hunt and James R. Larus. “Singularity: Rethinking the Software Stack”. In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49.

[9] *Implement recvmmsg() and sendmmsg() system calls*. URL: `https://reviews.freebsd.org/rS279204`.

[10] Ingo Molnar. *Syslets, generic asynchronous system call support*. URL: `https://lkml.org/lkml/2007/2/13/142`.

[11] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. “Scaling Memcache at Facebook”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI ’13)*. 2013, pp. 385–398.

[12] Mike Pall. *The LuaJIT Project*. URL: `http://luajit.org`.

[13] Niels Provos and Nick Mathewson. *libevent an event notification library*. URL: `http://libevent.org`.

[14] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. “Cassyopia: Compiler Assisted System Optimization”. In: *In 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. 2003, pp. 103–108.

[15] *recvmmsg - receive multiple messages on a socket*. URL: `http://man7.org/linux/man-pages/man2/recvmmsg.2.html`.

[16] *sendmmsg - send multiple messages on a socket*. URL: `http://man7.org/linux/man-pages/man2/sendmmsg.2.html`.

[17] Livio Soares and Michael Stumm. “FlexSC: Flexible System Call Scheduling with Exception-less System Calls”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. 2010, pp. 1–8.

[18] Lourival Vieira Neto, Roberto Ierusalimschy, Ana Lúcia de Moura, and Marc Balmer. “Scriptable Operating Systems with Lua”. In: *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS ’14)*. 2014, pp. 2–10.

[19] Mingqiang Zhuang and Brian Aker. *Memaslap - Load testing and benchmarking a server*. URL: `http://docs.libmemcached.org/bin/memaslap.html`.