Downloaded from orbit.dtu.dk on: Feb 16, 2019

DTU Library

Testing Infrastructure for Operating System Kernel Development

Walter, Maxwell; Karlsson, Sven

Published in:
Proceedings of the 7th Swedish Workshop on Multicore Computing (MCC'14)

Publication date:
2014

Link back to DTU Orbit

Citation (APA):
Walter, M., & Karlsson, S. (2014). Testing Infrastructure for Operating System Kernel Development. In Proceedings of the 7th Swedish Workshop on Multicore Computing (MCC'14)

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Testing Infrastructure for Operating System Kernel Development

Maxwell Walter
maxw@dtu.dk
Technical University of Denmark

Sven Karlsson
svea@dtu.dk
Technical University of Denmark

ABSTRACT

Testing is an important part of system development, and to test effectively we require knowledge of the internal state of the system under test. Testing an operating system kernel is a challenge as it is the operating system that typically provides access to this internal state information. Multi-core kernels pose an even greater challenge due to concurrency and their shared kernel state. In this paper, we present a testing framework that addresses these challenges by running the operating system in a virtual machine, and using virtual machine introspection to both communicate with the kernel and obtain information about the system. We have also developed an in-kernel testing API that we can use to develop a suite of unit tests in the kernel. We are using our framework for for the development of our own multi-core research kernel.

1. INTRODUCTION

Testing is an integral part of the development process. Unit testing helps ensure that individual components work as expected, and system level testing ensures that these components work together properly. Writing tests and having them available during the development cycle provides developers with knowledge of the correctness of their system. In addition, having an available test suite with good coverage allows developers to maintain a known good code base by performing regression testing.

When developing tests for programs, having more information allows for more complete and more accurate tests. The operating system controls program execution, and is responsible for providing access to this information. This makes OS kernel testing more challenging, especially in the early stages of development. Multi-core kernels increase the difficulty by adding concurrency issues, shared and private state, and state changes from multiple locations. To address these difficulties, we are creating a testing framework in conjunction with our new research operating system. We are developing our framework simultaneously with our operating system kernel to ensure that testing is a fundamental component of the system design.

Our testing framework consists of two primary components. The first component leverages *virtual machine* (VM) technology to run our operating system kernel in a virtualized environment. VMs provide an emulation of a physical computer system, and provide us with a method of inspecting the system's internal state. The second component is a kernel level testing API which provides a common way for developers to create, run, and report on tests.

In this paper, we present our testing framework and a discussion of how VMs can be used to aid in the testing of multi-core operating system kernels. The contributions this paper presents are a VM based framework for monitoring the status of an executing OS kernel for testing purposes, and addresses some of the difficulty of testing multi-core kernels. We also contribute an in-kernel API to facilitate testing and communication with test software. We are currently using our framework in the development of our own research multi-core OS kernel. In section 2 we provide background information and related work. In section 3 we describe the design of our system and section 4 describes our implementation of this architecture. We present our conclusions in section 5.

2. BACKGROUND AND PRIOR WORK

All operating system vendors test their systems. The Linux kernel, for instance, relies mostly on the community for testing on real hardware, which is a manual and labor intensive process. Automated systems such as Autotest and The Linux Test Project aim to reduce the amount of manual effort required [2, 1]. In contrast to our approach, these projects are designed to minimize the amount of manual labor required by increasing automation, rather than aiding test development.

Part of the reason for manual testing is that knowledge of the internal state of the system is required for accurate testing. Acquiring this information from an operating system kernel in an automated way can be difficult. Buchacker et al. present a testing framework that runs the kernel in userspace to allow easy access to its internals [3]. In recent years VMs have been proposed as a mechanism for testing operating system kernels, as they provide access to the internal state of the system [6]. Rather than run the kernel as a userspace application, we are able to run the kernel on simulated hardware for more accurate results.

Obtaining system state information is a common problem in other areas of system research. Knauerhase et al. use OS snapshots to acquire system state information about multi-core applications for the purpose of performance tuning [8]. Internal state information is also important to the intrusion detection community to identify potentially compromised systems without alerting software running on them [9, 12].

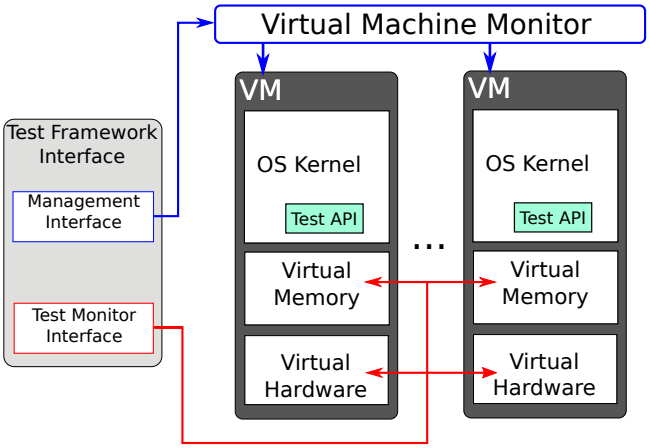

Figure 1: The architecture of our test framework.

To this end Garinkel et al. introduce a method they called *virtual machine introspection* (VMI) to passively extract information from the kernel without its knowledge [4]. Rather than assume the kernel is adversarial, as in the case of a compromised system, we allow the kernel and the test environment to cooperate. This provides us with more information, and more accurate information, as we do not have to assume that some information may be false.

3. DESIGN

In this section we present the architecture of our testing framework, which is shown in figure 1. The first component of the framework is the management and monitoring interfaces. These interfaces are responsible for interacting with the VM portion of the framework. The second component is the virtual execution environment itself which runs the OS kernel. We use the term *virtual machine monitor* (VMM) to refer to component that is responsible for creating and running the virtual machines, and *virtual machine* (VM) to refer to the actual virtualized execution. The final component is the in-kernel test API, which serves as a base on which to build kernel unit tests.

3.1 Management Interface

The *Management Interface* provides a way for the test application to communicate with the VMM and provides the ability to:

- `Create` new VMs with arbitrary hardware configurations. Machines may be created with different memory sizes, CPU count and connection topology, and connected hardware.
- `Start`, `stop`, and `destroy` the virtual machines that have been created.
- `Pause` the virtual machine. Pausing the VM gives the test application time to sample and inspect the system in a more detailed manner. It also gives the application the ability to modify the internal state of the kernel without having to worry about timing or synchronization issues.

3.2 Test Monitor Interface and Communication

In order for our testing infrastructure to report on the in-kernel tests and examine the validity of the system as a whole, we need to be able to communicate with the system. We categorize the information we require into explicit communication, where the kernel actively sends messages to an external monitor, and implicit communication, where information is obtained without action on the part of the kernel. Information such as user-space/kernel-space transitions and page faults are useful for tracking the current state of the kernel, while other information such as segment violations and system faults can be useful for determining if the kernel is executing properly. Events such as double faults indicate that the kernel is not behaving properly and cannot be counted on for accurate information in the case of explicit communication.

An important advantage afforded by the use of VMs is that it allows passive observation of the kernel's state by way of VMI. In the case of explicit communication, where the kernel actively sends messages, the very act of sending a message changes the kernel's internal state and can affect testing. In multi-core settings, with concurrency and timing differences, this can lead to indeterminate effects. Passive observation solves this by allowing access to kernel information without altering it.

3.3 Kernel Test API

The final system component is the test API which serves two purposes. First, it provides locations in the kernel that can be easily observed. The locations of various data structures in the test API can be exported to the monitoring framework for observation. The second purpose is to provide a common interface on which to build in-kernel unit tests. This interface includes the tests themselves, as well as a way to collect and run tests as groups. Using a common interface ensures that all tests can be interacted with in the same manner, which makes the task of collecting testing information and reporting on test status easier.

4. IMPLEMENTATION

This section describes the implementation of our testing framework. It explains our choice of VMM and VM, as well as the modifications required to the VM to obtain the information we require. We provide a more detailed description of the communication methods available in our framework. Finally, we describe our in-kernel testing API in more detail.

4.1 VM and VMM Control

We chose to use libvirt as our interface to the virtual execution environment which, when combined with QEMU, provides our VMM. Libvirt is a toolkit that provides a generic interface for interacting with virtualization subsystems on Linux [11]. Using libvirt we can change the VM that we are using without impacting the management interface. Additionally, it enables us to use different virtualization environments with the same management interface.

We chose QEMU as our VM due to its ability to simulate a number of architectures of interest including AMD64 and AArch64. QEMU is capable of executing at near bare-metal

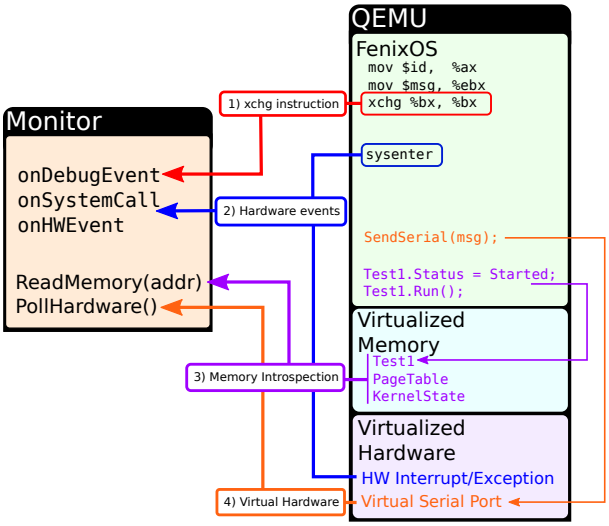

Figure 2: Communication methods available to the testing framework.

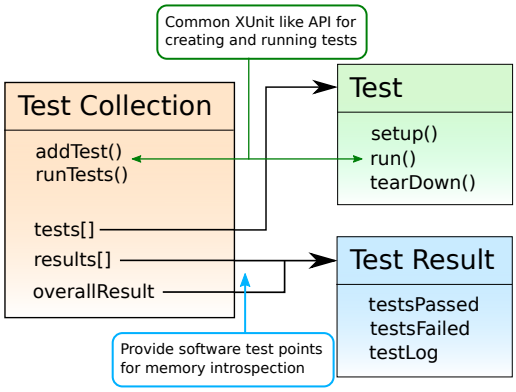

Figure 3: In-kernel test API.

speeds using the kernel virtual machine (KVM) module[7], and has support for very fast virtual storage and network devices. This allows for very fast execution which helps improve test results and speeds up testing. QEMU supports multiple processing cores, up to the number of physical cores in the system. In addition, projects such as Manifold build a simulator from QEMU supporting hundreds of hardware contexts [13].

We modified QEMU to provide information that is not accessible though the libvirt interface. One modification we added is the ability of a client to alter the memory of a VM. Libvirt allows clients to read the contents of a core's physical or virtual memory, but it does not allow writing. We also modified QEMU to enable it to send messages to a client on hardware interrupts, execution of the `syscall` and `sysret` instructions, and the execution of certain specific assembly instructions.

4.2 Framework Communication

Figure 2 shows the four methods of communication that can be used by the VM to provide information to the test monitor.

xchg Instruction: We need a way to retrieve information from the kernel during early boot, before memory and hardware has been initialized. In addition, we would like a mechanism for sending lightweight messages to the monitor for tracing purposes. To accomplish this, we modified QEMU to send a message to the monitor when it executes an `xchg` instruction and both operands are the same 8-bit register. This is a technique used in simulators like Bochs to trigger a breakpoint [10].

The system may also be configured to pause on `xchg` events for sampling or further in-depth analysis. This is important in multi-core scenarios as state can change quickly from a number of different sources. The ability to pause the entire system on an event gives us the ability to sample more accurately.

Hardware Events: In order to perform whole system testing we need to track the system state. This requires information on hardware events that affect the system state, such as user/kernel space transitions, page faults, and hardware exceptions. Using the `xchg` instruction to send a message on each of these events would be possible, but would require significant additions to the kernel source code and would introduce performance penalties when running on real systems.

We chose instead to modify QEMU to send events to the monitor on all hardware interrupts, exceptions, and system call routines. This does not require any changes the kernel source code and gives us more accurate information when something goes wrong in the kernel. For example, in the case of a double fault we cannot rely on the kernel to send events as the kernel is no longer executing properly.

As with the `xchg` instruction events, the VM can be configured to pause on any particular kind of event. This is even more useful in the case of faults as it gives the test framework time to capture the state of the system in the face of errors. Without pausing, the kernel may destroy its state, or the system may reset before enough information could be obtained.

Memory Introspection: One of the most important communication methods is memory introspection. Memory introspection gives the framework access to the system's virtual and physical memory without the system's knowledge. With this we are able to passively observe the state of the kernel without altering it, provided we know or can find the address of the objects we are interested in observing. Addresses can be obtained either during compilation, or as part of the messages received from the kernel.

Virtual Hardware Communication: Virtual hardware communication is a communication method where the system being tested communicates with the framework via virtualized hardware provided by the VM. Hardware peripherals such as network cards and serial ports can provide bidirectional communication and can be reused on actual hardware provided a proper drivers exist.

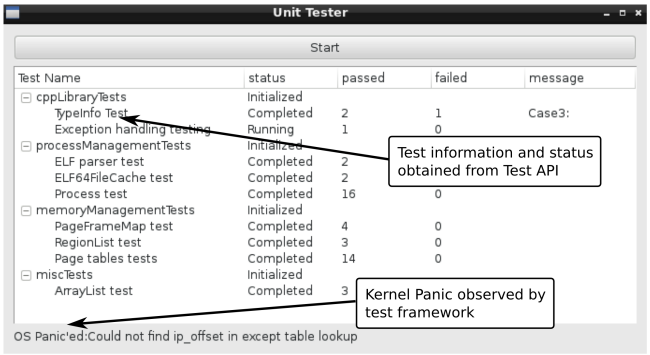

Figure 4: GUI application using the testing framework.

4.3 Kernel Testing API

The in-kernel testing API is a set of classes that are used to implement a unit test framework in the kernel. This unit test framework is modeled after the xUnit family of testing APIs which are familiar to many developers [5]. It consists of a results object, an abstract test interface, and a test collection class, as shown in figure 3. The test class interface provides a mechanism to set up the system state for test execution and restore the system after testing is completed. If the test is destructive or has failed in a way that prevents normal system operation, the test can indicate this to the framework and appropriate measures can be taken.

The test collection interface also provides a single mechanism for collecting tests into batches and running them together. This provides a convenient wrapper for tests and a single point of communication. We can also use this structure for memory introspection assuming we know where the test collections are located in memory. This allows the framework to watch tests execute and report on the status of tests as they complete.

5. CONCLUSIONS AND EXPERIENCES

The framework that we have developed has been incorporated into our development cycle and is being used for continuous testing. We are also using it to ensure that development efforts of different system components do not interfere. Figure 4 shows a GUI application developed using our framework that we used during the development of our virtual memory system. We have developed a minimal C++ runtime to support features such as exceptions and dynamic casting, and small changes can have far reaching consequences. Without adequate test coverage it was difficult to identify when breakage occurred. Debugging becomes more difficult the longer the period of time between when breakage occurs and when it is identified. Having tests available during development allowed us to immediately determine when changes to the C++ runtime impacted the other systems.

We are also able to use our framework as a simulation tool to verify the kernel as it is developed. Certain aspects of the system such as boot up are difficult to verify as there is little output to indicate what is happening unless something goes wrong. We could use a debugger attached to the simulator to step through the execution, but that is time consuming and slow. It is also difficult to get a view of the entire system using a debugger in a multi-core scenario. To obtain a view of the entire system we use the GUI application mentioned previously.

Finally, we are also using the framework to aid in analysis during debugging. As an example, early kernel assembly code uses the `xchg` instruction to indicate an error during initialization. It also sends an identifier with the message that the monitor can use to determine the error that occurred. The helps pinpoint the locations of errors during testing to aid in subsequent debugging.

6. ACKNOWLEDGMENTS

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 332913 for project COPCAMS.

7. REFERENCES

[1] The linux test project. `http://linux-test-project.github.io/`.

[2] J. Admanski and S. Howard. Autotest-testing the untestable. In *Proceedings of the Linux Symposium*, 2009.

[3] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *IEEE International Symposium on High Assurance Systems Engineering*, 2001.

[4] T. Garinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. *Proceedings of Network and Distributed Systems Security Symposium*, 2003.

[5] P. Hamill. *Unit Test Frameworks, chapter Chapter 3: The xUnit Family of Unit Test Frameworks*. O'Reilly, 2004.

[6] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference*, 2008.

[7] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[8] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 28(3), 2008.

[9] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *Euromicro Conference*, 2004.

[10] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996.

[11] libvirt Developers. libvirt virtualization api. `http://libvirt.org/index.html`.

[12] K. Nance, B. Hay, and M. Bishop. virtual machine introspection. *IEEE Computer Society*, 2008.

[13] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili. Manifold: A parallel simulation framework for multicore systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.