Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

lguest: Implementing the little Linux hypervisor

Rusty Russell
IBM OzLabs
`rusty@rustcorp.com.au`

Abstract

Lguest is a small x86 32-bit Linux hypervisor for running Linux under Linux, and demonstrating the paravirtualization abilities in Linux since 2.6.20. At around 5,000 lines of code including utilities, it also serves as an excellent springboard for mastering the theory and practice of x86 virtualization.

This talk will cover the philosophy of lguest and then dive into the implementation details as they stand at this point in time. Operating System experience is required, but x86 knowledge isn't. By the time the talk is finished, you should have a good grounding in the range of implementation issues facing all virtualization technologies on Intel, such as Xen and KVM. You should also be inspired to create your own hypervisor, using your own pets as the logo.

1 Introduction

Around last year's OLS I was having discussions with various technical people about Linux support for paravirtualization, and Xen in particular. (Paravirtualization is where the guest knows it's being run under a hypervisor, and changes its behaviour).

I wanted the Linux kernel to support Xen, without wedding the Linux kernel to its interface: it seemed to me that now Xen showed that Free Software virtualization wasn't hard, we'd see other virtualization technologies worth supporting. There was already VMWare's proposed VMI standard, for example, but that wasn't a proven ABI either.

The result was *paravirt_ops*. This is a single structure which encapsulates all the sensitive instructions which a hypervisor might want to override. This was very similar to the VMI proposal by Zach Amsden, but some of the functions which Xen or VMI wanted were non-obvious to me.

So, I decided to write a trivial, self-contained Linux-on-Linux hypervisor. It would live in the Linux kernel source, run the same kernel for guests as for host, and be as simple as possible. This would serve as a third testbed for `paravirt_ops`.

2 Post-rationale for lguest

There turned out to be other benefits to writing such a hypervisor.

- It turned out to be around 5,000 lines, including the 1,000 lines of userspace code. This means it is small enough to be read and understood by kernel coders.
- It provides a complete in-tree example of how to use `paravirt_ops`.
- It provides a simple way to demonstrate the effects of a new Linux paravirtualization feature: you only need to patch one place to show the new feature and how to use it.
- It provides a testbed for new ideas. As other hypervisors rush to "productize" and nail down APIs and ABIs, lguest can be changed from kernel to kernel. Remember, lguest only promises to run the matching guests and host (i.e., no ABI).

My final point is more social than technical. I said that Xen had shown that Free Software paravirtualization was possible, but there was also some concern that its lead and "buzz" risked sucking up the groundwater from under other Free hypervisors: why start your own when Xen is so far ahead? Yet this field desperately needs more independent implementations and experimentation. Creating a tiny hackable hypervisor seemed to be the best way to encourage that.

As it turned out, I needn't have worried too much. The KVM project came along while I was polishing my patches, and slid straight into the kernel. KVM uses a similar "linux-centric" approach to lguest. But on the bright side, writing lguest taught me far more than I ever thought I'd know about the horribly warty x86 architecture.

3 Comparing With Other Hypervisors

As you'd expect from its simplicity, lguest has fewer features than any other hypervisor you're likely to have heard of. It doesn't currently support SMP guests, suspend and resume, or 64-bit. Glauber de Oliveira Costa and Steven Rostedt are hacking on lguest64 furiously, and suspend and resume are high on the TODO list.

Lguest only runs matching host and guest kernels. Other hypervisors aim to run different Operating Systems as guests. Some also do full virtualization, where unmodified OSs can be guests, but both KVM and Xen require newer chips with virtualization features to do this.

Lguest is slower than other hypervisors, though not always noticeably so: it depends on workload.

On the other hand, lguest is currently 5,004 lines for a total of 2,009 semicolons. (Note that the documentation patch adds another 3,157 lines of comments.) This includes the 983 lines (408 semicolons) of userspace code.

The code size of KVM and Xen are hard to compare to this: both have *features*, such as 64-bit support. Xen includes IA-64 support, and KVM includes all of qemu (yet doesn't use most of it).

Nonetheless it is instructive to note that KVM 19 is 274,630 lines for a total of 99,595 semicolons. Xen unstable (14854:039daabebad5) is 753,242 lines and 233,995 semicolons (the 53,620 lines of python don't carry their weight in semicolons properly, however).

4 Lguest Code: A Whirlwind Tour

lguest consists of five parts:

1. The guest `paravirt_ops` implementation,
2. The launcher which creates and supplies external I/O for the guest,
3. The switcher which flips the CPU between host and guest,
4. The host module (lg.ko) which sets up the switcher and handles the kernel side of things for the launcher, and
5. The awesome documentation which spans the code.[1]

4.1 Guest Code

How does the kernel know it's an lguest guest? The first code the x86 kernel runs are is `startup_32` in `head.S`. This tests if paging is already enabled: if it is, we know we're under some kind of hypervisor. We end up trying all the registered `paravirt_probe` functions, and end up in the one in `drivers/lguest/lguest.c`. Here's the guest, file-by-file:

drivers/lguest/lguest.c Guests know that they can't do privileged operations such as disable interrupts: they have to ask the host to do such things via *hypercalls*. This file consists of all the replacements for such low-level native hardware operations: we replace the `struct paravirt_ops` pointers with these.

drivers/lguest/lguest_asm.S The guest needs several assembler routines for low-level things and placing them all in `lguest.c` was a little ugly.

drivers/lguest/lguest_bus.c Lguest guests use a very simple bus for devices. It's a simple array of device descriptors contained just above the top of normal memory. The lguest bus is 80% tedious boilerplate code.

drivers/char/hvc_lguest.c A trivial console driver: we use lguest's DMA mechanism to send bytes out, and register a DMA buffer to receive bytes in. It is assumed to be present and available from the very beginning of boot.

drivers/block/lguest_blk.c A simple block driver which appears as `/dev/lgba`, `lgbb`, `lgbc`, etc. The mechanism is simple: we place the information about the request in the device page,

[1]Documentation was awesome at time of this writing. It may have rotted by time of reading.

then use the `SEND_DMA` hypercall (containing the data for a write, or an empty "ping" DMA for a read).

drivers/net/lguest_net.c This is very simple, a virtual network driver. The only trick is that it can talk directly to multiple other recipients (i.e., other guests on the same network). It can also be used with only the host on the network.

4.2 Launcher Code

The launcher sits in the Documentation/lguest directory: as lguest has no ABI, it needs to live in the kernel tree with the code. It is a simple program which lays out the "physical" memory for the new guest by mapping the kernel image and the virtual devices, then reads repeatedly from `/dev/lguest` to run the guest. The read returns when a signal is received or the guest sends DMA out to the launcher.

The only trick: the Makefile links it statically at a high address, so it will be clear of the guest memory region. It means that each guest cannot have more than 2.5G of memory on a normally configured host.

4.3 Switcher Code

Compiled as part of the "lg.ko" module, this is the code which sits at `0xFFC00000` to do the low-level guest-host switch. It is as simple as it can be made, but it's naturally very specific to x86.

4.4 Host Module: lg.ko

It is important to that lguest be "just another" Linux kernel module. Being able to simply insert a module and start a new guest provides a "low commitment" path to virtualization. Not only is this consistent with lguest's experimental aims, but it has potential to open new scenarios to apply virtualization.

drivers/lguest/lguest_user.c This contains all the `/dev/lguest` code, whereby the userspace launcher controls and communicates with the guest. For example, the first write will tell us the memory size, pagetable, entry point, and kernel address offset. A read will run the guest until a signal is pending (`-EINTR`), or the guest does a DMA out to the launcher. Writes are also used to get a DMA buffer registered by the guest, and to send the guest an interrupt.

drivers/lguest/io.c The I/O mechanism in lguest is simple yet flexible, allowing the guest to talk to the launcher program or directly to another guest. It uses familiar concepts of DMA and interrupts, plus some neat code stolen from futexes.

drivers/lguest/core.c This contains `run_ guest()` which actually calls into the host↔guest switcher and analyzes the return, such as determining if the guest wants the host to do something. This file also contains useful helper routines, and a couple of non-obvious setup and teardown pieces which were implemented after days of debugging pain.

drivers/lguest/hypercalls.c Just as userspace programs request kernel operations via a system call, the guest requests host operations through a "hypercall." As you'd expect, this code is basically one big switch statement.

drivers/lguest/segments.c The x86 architecture has segments, which involve a table of descriptors which can be used to do funky things with virtual address interpretation. The segment handling code consists of simple sanity checks.

drivers/lguest/page_tables.c The guest provides a virtual-to-physical mapping, but the host can neither trust it nor use it: we verify and convert it here to point the hardware to the actual guest pages when running the guest. This technique is referred to as *shadow pagetables*.

drivers/lguest/interrupts_and_traps.c This file deals with Guest interrupts and traps. There are three classes of interrupts:

1. Real hardware interrupts which occur while we're running the guest,
2. Interrupts for virtual devices attached to the guest, and
3. Traps and faults from the guest.

Real hardware interrupts must be delivered to the host, not the guest. Virtual interrupts must be delivered to the guest, but we make them look just like real hardware would deliver them. Traps from the guest can be set up to go directly back into the guest, but sometimes the host wants to see them first, so we also have a way of "reflecting" them into the guest as if they had been delivered to it directly.

4.5 The Documentation

The documentation is in seven parts, as outlined in `drivers/lguest/README`. It uses a simple script in `Documentation/lguest` to output interwoven code and comments in literate programming style. It took me two weeks to write (although it did lead to many cleanups along the way). Currently the results take up about 120 pages, so it is appropriately described throughout as a heroic journey. From the README file:

Our Quest is in seven parts:

Preparation: In which our potential hero is flown quickly over the landscape for a taste of its scope. Suitable for the armchair coders and other such persons of faint constitution.

Guest: Where we encounter the first tantalising wisps of code, and come to understand the details of the life of a Guest kernel.

Drivers: Whereby the Guest finds its voice and become useful, and our understanding of the Guest is completed.

Launcher: Where we trace back to the creation of the Guest, and thus begin our understanding of the Host.

Host: Where we master the Host code, through a long and tortuous journey. Indeed, it is here that our hero is tested in the Bit of Despair.

Switcher: Where our understanding of the intertwined nature of Guests and Hosts is completed.

Mastery: Where our fully fledged hero grapples with the Great Question: "What next?"

5 Benchmarks

I wrote a simple extensible GPL'd benchmark program called virtbench.[2] It's a little primitive at the moment, but it's designed to guide optimization efforts for hypervisor authors. Here are the current results for a native run on a UP host with 512M of RAM and the same configuration running under lguest (on the same Host, with 3G of RAM). Note that these results are continually improving, and are obsolete by the time you read them.

Test Name	Native	Lguest	Factor
Context switch via pipe	2413 ns	6200 ns	2.6
One Copy-on-Write fault	3555 ns	9822 ns	2.8
Exec client once	302 us	776 us	2.6
One fork/exit/ wait	120 us	407 us	3.7
One int-0x80 syscall	269 ns	266 ns	1.0
One syscall via libc	127 ns	276 ns	2.2
Two PTE updates	1802 ns	6042 ns	3.4
256KB read from disk	33333 us	41725 us	1.3
One disk read	113 us	185 us	1.6
Inter-guest ping-pong	53850 ns	149795 ns	2.8
Inter-guest 4MB TCP	16352 us	334437 us	20
Inter-guest 4MB sendfile	10906 us	309791 us	28
Kernel Compile	10m39	13m48s	1.3

Table 1: Virtbench and kernel compile times

6 Future Work

There is an infinite amount of future work to be done. It includes:

1. More work on the I/O model.
2. More optimizations generally.
3. `NO_HZ` support.
4. Framebuffer support.
5. 64-bit support.
6. SMP guest support.
7. A better web page.

[2] `http://ozlabs.org/~rusty/virtbench`

7 Conclusion

Lguest has shown that writing a hypervisor for Linux isn't difficult, and that even a minimal hypervisor can have reasonable performance. It remains to be seen how useful lguest will be, but my hope is that it will become a testing ground for Linux virtualization technologies, a useful basis for niche hypervisor applications, and an excellent way for coders to get their feet wet when starting to explore Linux virtualization.