QEMU Internals

Table of Contents

1 Introduction

1.1 Features

QEMU is a FAST! processor emulator using a portable dynamic translator.

QEMU has two operating modes:

- Full system emulation. In this mode, QEMU emulates a full system (usually a PC), including a processor and various peripherals. It can be used to launch an different Operating System without rebooting the PC or to debug system code.
- User mode emulation (Linux host only). In this mode, QEMU can launch Linux processes compiled for one CPU on another CPU. It can be used to launch the Wine Windows API emulator (`http://www.winehq.org`) or to ease cross-compilation and cross-debugging.

As QEMU requires no host kernel driver to run, it is very safe and easy to use.

QEMU generic features:

- User space only or full system emulation.
- Using dynamic translation to native code for reasonable speed.
- Working on x86 and PowerPC hosts. Being tested on ARM, Sparc32, Alpha and S390.
- Self-modifying code support.
- Precise exceptions support.
- The virtual CPU is a library (`libqemu`) which can be used in other projects (look at ‘`qemu/tests/qruncom.c`’ to have an example of user mode `libqemu` usage).

QEMU user mode emulation features:

- Generic Linux system call converter, including most ioctls.
- clone() emulation using native CPU clone() to use Linux scheduler for threads.
- Accurate signal handling by remapping host signals to target signals.

QEMU full system emulation features:

- QEMU can either use a full software MMU for maximum portability or use the host system call mmap() to simulate the target MMU.

1.2 x86 emulation

QEMU x86 target features:

- The virtual x86 CPU supports 16 bit and 32 bit addressing with segmentation. LDT/GDT and IDT are emulated. VM86 mode is also supported to run DOSEMU.
- Support of host page sizes bigger than 4KB in user mode emulation.
- QEMU can emulate itself on x86.
- An extensive Linux x86 CPU test program is included ‘`tests/test-i386`’. It can be used to test other x86 virtual CPUs.

Current QEMU limitations:

- No SSE/MMX support (yet).

- No x86-64 support.
- IPC syscalls are missing.
- The x86 segment limits and access rights are not tested at every memory access (yet). Hopefully, very few OSes seem to rely on that for normal use.
- On non x86 host CPUs, `double`s are used instead of the non standard 10 byte `long double`s of x86 for floating point emulation to get maximum performances.

1.3 ARM emulation

- Full ARM 7 user emulation.
- NWFPE FPU support included in user Linux emulation.
- Can run most ARM Linux binaries.

1.4 MIPS emulation

- The system emulation allows full MIPS32/MIPS64 Release 2 emulation, including privileged instructions, FPU and MMU, in both little and big endian modes.
- The Linux userland emulation can run many 32 bit MIPS Linux binaries.

Current QEMU limitations:

- Self-modifying code is not always handled correctly.
- 64 bit userland emulation is not implemented.
- The system emulation is not complete enough to run real firmware.
- The watchpoint debug facility is not implemented.

1.5 PowerPC emulation

- Full PowerPC 32 bit emulation, including privileged instructions, FPU and MMU.
- Can run most PowerPC Linux binaries.

1.6 SPARC emulation

- Full SPARC V8 emulation, including privileged instructions, FPU and MMU. SPARC V9 emulation includes most privileged and VIS instructions, FPU and I/D MMU. Alignment is fully enforced.
- Can run most 32-bit SPARC Linux binaries, SPARC32PLUS Linux binaries and some 64-bit SPARC Linux binaries.

Current QEMU limitations:

- IPC syscalls are missing.
- Floating point exception support is buggy.
- Atomic instructions are not correctly implemented.
- Sparc64 emulators are not usable for anything yet.

2 QEMU Internals

2.1 QEMU compared to other emulators

Like bochs [3], QEMU emulates an x86 CPU. But QEMU is much faster than bochs as it uses dynamic compilation. Bochs is closely tied to x86 PC emulation while QEMU can emulate several processors.

Like Valgrind [2], QEMU does user space emulation and dynamic translation. Valgrind is mainly a memory debugger while QEMU has no support for it (QEMU could be used to detect out of bound memory accesses as Valgrind, but it has no support to track uninitialised data as Valgrind does). The Valgrind dynamic translator generates better code than QEMU (in particular it does register allocation) but it is closely tied to an x86 host and target and has no support for precise exceptions and system emulation.

EM86 [4] is the closest project to user space QEMU (and QEMU still uses some of its code, in particular the ELF file loader). EM86 was limited to an alpha host and used a proprietary and slow interpreter (the interpreter part of the FX!32 Digital Win32 code translator [5]).

TWIN [6] is a Windows API emulator like Wine. It is less accurate than Wine but includes a protected mode x86 interpreter to launch x86 Windows executables. Such an approach has greater potential because most of the Windows API is executed natively but it is far more difficult to develop because all the data structures and function parameters exchanged between the API and the x86 code must be converted.

User mode Linux [7] was the only solution before QEMU to launch a Linux kernel as a process while not needing any host kernel patches. However, user mode Linux requires heavy kernel patches while QEMU accepts unpatched Linux kernels. The price to pay is that QEMU is slower.

The new Plex86 [8] PC virtualizer is done in the same spirit as the qemu-fast system emulator. It requires a patched Linux kernel to work (you cannot launch the same kernel on your PC), but the patches are really small. As it is a PC virtualizer (no emulation is done except for some priveledged instructions), it has the potential of being faster than QEMU. The downside is that a complicated (and potentially unsafe) host kernel patch is needed.

The commercial PC Virtualizers (VMWare [9], VirtualPC [10], TwoOStwo [11]) are faster than QEMU, but they all need specific, proprietary and potentially unsafe host drivers. Moreover, they are unable to provide cycle exact simulation as an emulator can.

2.2 Portable dynamic translation

QEMU is a dynamic translator. When it first encounters a piece of code, it converts it to the host instruction set. Usually dynamic translators are very complicated and highly CPU dependent. QEMU uses some tricks which make it relatively easily portable and simple while achieving good performances.

The basic idea is to split every x86 instruction into fewer simpler instructions. Each simple instruction is implemented by a piece of C code (see '`target-i386/op.c`'). Then a compile time tool ('`dyngen`') takes the corresponding object file ('`op.o`') to generate a dynamic code generator which concatenates the simple instructions to build a function (see '`op.h:dyngen_code()`').

In essence, the process is similar to [1], but more work is done at compile time.

A key idea to get optimal performances is that constant parameters can be passed to the simple operations. For that purpose, dummy ELF relocations are generated with gcc for each constant parameter. Then, the tool (‘`dyngen`’) can locate the relocations and generate the appriopriate C code to resolve them when building the dynamic code.

That way, QEMU is no more difficult to port than a dynamic linker.

To go even faster, GCC static register variables are used to keep the state of the virtual CPU.

2.3 Register allocation

Since QEMU uses fixed simple instructions, no efficient register allocation can be done. However, because RISC CPUs have a lot of register, most of the virtual CPU state can be put in registers without doing complicated register allocation.

2.4 Condition code optimisations

Good CPU condition codes emulation (`EFLAGS` register on x86) is a critical point to get good performances. QEMU uses lazy condition code evaluation: instead of computing the condition codes after each x86 instruction, it just stores one operand (called `CC_SRC`), the result (called `CC_DST`) and the type of operation (called `CC_OP`).

`CC_OP` is almost never explicitely set in the generated code because it is known at translation time.

In order to increase performances, a backward pass is performed on the generated simple instructions (see `target-i386/translate.c:optimize_flags()`). When it can be proved that the condition codes are not needed by the next instructions, no condition codes are computed at all.

2.5 CPU state optimisations

The x86 CPU has many internal states which change the way it evaluates instructions. In order to achieve a good speed, the translation phase considers that some state information of the virtual x86 CPU cannot change in it. For example, if the SS, DS and ES segments have a zero base, then the translator does not even generate an addition for the segment base.

[The FPU stack pointer register is not handled that way yet].

2.6 Translation cache

A 16 MByte cache holds the most recently used translations. For simplicity, it is completely flushed when it is full. A translation unit contains just a single basic block (a block of x86 instructions terminated by a jump or by a virtual CPU state change which the translator cannot deduce statically).

2.7 Direct block chaining

After each translated basic block is executed, QEMU uses the simulated Program Counter (PC) and other cpu state informations (such as the CS segment base value) to find the next basic block.

In order to accelerate the most common cases where the new simulated PC is known, QEMU can patch a basic block so that it jumps directly to the next one.

The most portable code uses an indirect jump. An indirect jump makes it easier to make the jump target modification atomic. On some host architectures (such as x86 or PowerPC), the `JUMP` opcode is directly patched so that the block chaining has no overhead.

2.8 Self-modifying code and translated code invalidation

Self-modifying code is a special challenge in x86 emulation because no instruction cache invalidation is signaled by the application when code is modified.

When translated code is generated for a basic block, the corresponding host page is write protected if it is not already read-only (with the system call `mprotect()`). Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU then invalidates all the translated code in the page and enables write accesses to the page.

Correct translated code invalidation is done efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining.

Although the overhead of doing `mprotect()` calls is important, most MSDOS programs can be emulated at reasonnable speed with QEMU and DOSEMU.

Note that QEMU also invalidates pages of translated code when it detects that memory mappings are modified with `mmap()` or `munmap()`.

When using a software MMU, the code invalidation is more efficient: if a given code page is invalidated too often because of write accesses, then a bitmap representing all the code inside the page is built. Every store into that page checks the bitmap to see if the code really needs to be invalidated. It avoids invalidating the code when only data is modified in the page.

2.9 Exception support

longjmp() is used when an exception such as division by zero is encountered.

The host SIGSEGV and SIGBUS signal handlers are used to get invalid memory accesses. The exact CPU state can be retrieved because all the x86 registers are stored in fixed host registers. The simulated program counter is found by retranslating the corresponding basic block and by looking where the host program counter was at the exception point.

The virtual CPU cannot retrieve the exact `EFLAGS` register because in some cases it is not computed because of condition code optimisations. It is not a big concern because the emulated code can still be restarted in any cases.

2.10 MMU emulation

For system emulation, QEMU uses the mmap() system call to emulate the target CPU MMU. It works as long the emulated OS does not use an area reserved by the host OS (such as the area above 0xc0000000 on x86 Linux).

In order to be able to launch any OS, QEMU also supports a soft MMU. In that mode, the MMU virtual to physical address translation is done at every memory access. QEMU uses an address translation cache to speed up the translation.

In order to avoid flushing the translated code each time the MMU mappings change, QEMU uses a physically indexed translation cache. It means that each basic block is indexed with its physical address.

When MMU mappings change, only the chaining of the basic blocks is reset (i.e. a basic block can no longer jump directly to another one).

2.11 Hardware interrupts

In order to be faster, QEMU does not check at every basic block if an hardware interrupt is pending. Instead, the user must asynchrously call a specific function to tell that an interrupt is pending. This function resets the chaining of the currently executing basic block. It ensures that the execution will return soon in the main loop of the CPU emulator. Then the main loop can test if the interrupt is pending and handle it.

2.12 User emulation specific details

2.12.1 Linux system call translation

QEMU includes a generic system call translator for Linux. It means that the parameters of the system calls can be converted to fix the endianness and 32/64 bit issues. The IOCTLs are converted with a generic type description system (see ‘`ioctls.h`’ and ‘`thunk.c`’).

QEMU supports host CPUs which have pages bigger than 4KB. It records all the mappings the process does and try to emulated the `mmap()` system calls in cases where the host `mmap()` call would fail because of bad page alignment.

2.12.2 Linux signals

Normal and real-time signals are queued along with their information (`siginfo_t`) as it is done in the Linux kernel. Then an interrupt request is done to the virtual CPU. When it is interrupted, one queued signal is handled by generating a stack frame in the virtual CPU as the Linux kernel does. The `sigreturn()` system call is emulated to return from the virtual signal handler.

Some signals (such as SIGALRM) directly come from the host. Other signals are synthetized from the virtual CPU exceptions such as SIGFPE when a division by zero is done (see `main.c:cpu_loop()`).

The blocked signal mask is still handled by the host Linux kernel so that most signal system calls can be redirected directly to the host Linux kernel. Only the `sigaction()` and `sigreturn()` system calls need to be fully emulated (see ‘`signal.c`’).

2.12.3 clone() system call and threads

The Linux clone() system call is usually used to create a thread. QEMU uses the host clone() system call so that real host threads are created for each emulated thread. One virtual CPU instance is created for each thread.

The virtual x86 CPU atomic operations are emulated with a global lock so that their semantic is preserved.

Note that currently there are still some locking issues in QEMU. In particular, the translated cache flush is not protected yet against reentrancy.

2.12.4 Self-virtualization

QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.

Achieving self-virtualization is not easy because there may be address space conflicts. QEMU solves this problem by being an executable ELF shared object as the ld-linux.so ELF interpreter. That way, it can be relocated at load time.

2.13 Bibliography

[1] `http://citeseer.nj.nec.com/piumarta98optimizing.html`, Optimizing direct threaded code by selective inlining (1998) by Ian Piumarta, Fabio Riccardi.

[2] `http://developer.kde.org/~sewardj/`, Valgrind, an open-source memory debugger for x86-GNU/Linux, by Julian Seward.

[3] `http://bochs.sourceforge.net/`, the Bochs IA-32 Emulator Project, by Kevin Lawton et al.

[4] `http://www.cs.rose-hulman.edu/~donaldlf/em86/index.html`, the EM86 x86 emulator on Alpha-Linux.

[5] `http://www.usenix.org/publications/library/proceedings/usenix-nt97/full_papers/chernoff/chernoff.pdf`, DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT, by Anton Chernoff and Ray Hookway.

[6] `http://www.willows.com/`, Windows API library emulation from Willows Software.

[7] `http://user-mode-linux.sourceforge.net/`, The User-mode Linux Kernel.

[8] `http://www.plex86.org/`, The new Plex86 project.

[9] `http://www.vmware.com/`, The VMWare PC virtualizer.

[10] `http://www.microsoft.com/windowsxp/virtualpc/`, The VirtualPC PC virtualizer.

[11] `http://www.twoostwo.org/`, The TwoOStwo PC virtualizer.

3 Regression Tests

In the directory '`tests/`', various interesting testing programs are available. They are used for regression testing.

3.1 '`test-i386`'

This program executes most of the 16 bit and 32 bit x86 instructions and generates a text output. It can be compared with the output obtained with a real CPU or another emulator. The target `make test` runs this program and a `diff` on the generated output.

The Linux system call `modify_ldt()` is used to create x86 selectors to test some 16 bit addressing and 32 bit with segmentation cases.

The Linux system call `vm86()` is used to test vm86 emulation.

Various exceptions are raised to test most of the x86 user space exception reporting.

3.2 '`linux-test`'

This program tests various Linux system calls. It is used to verify that the system call parameters are correctly converted between target and host CPUs.

3.3 '`qruncom.c`'

Example of usage of `libqemu` to emulate a user mode i386 CPU.

4 Index

(Index is nonexistent)