iX KOMPAKT
Ein Sonderheft des Magazins für professionelle Informationstechnik
Herbst 2017

CONTAINER UND VIRTUALISIERUNG

Zusatzmaterial auf DVD

Docker fürs Rechenzentrum
Windows und Container
Kubernetes
Migration in die Cloud
Beispielcode

DVD ROM

Datenträger enthält **Info- und Lehrprogramme** gemäß § 14 JuSchG

Handreichungen für Administratoren und Anwender

Orchestrierung: Docker im Cluster verwalten

Hyper-V 2017: Microsoft greift im Virtualisierungsmarkt an

Automatisierung: Tutorial für Kubernetes-Administratoren

Sicherheit: Grundlagen der Container-Abschottung

Proxmox VE 5.0: Virtualisierung und Container im Open-Source-Paket

FreeBSD Bhyve: Unix-Derivat im produktiven Einsatz

Container und Jails: Schlanke Alternative zum Emulator

VMware oder VirtualBox? Zweikampf auf dem virtuellen Desktop

Internet der Dinge: Virtualisierung in der Industrie

Ein „Connected to /dev/ttyp4 (speed 115200)“ bestätigt die Verbindung. Die Eingabetaste bringt den Installer-Dialog zum Vorschein. Hier kann der Anwender über „[S]hell“ einen *reboot* der VM durchführen, bei der die üblichen Kernel-Meldungen durchlaufen. Die VM ist recht nutzlos und man sollte sie daher aus einem anderen Terminal via *vmctl stop 1* beenden.

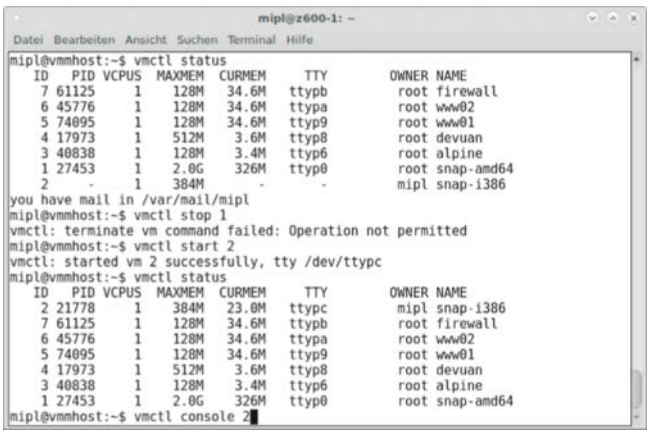
Aktuelle OpenBSD-Gäste erhalten über das im Kernel enthaltene „VMM control interface“ (*vmmci*-Treiber) die Shutdown-Anfrage des Hosts und fahren sich daraufhin herunter. Ältere Systeme muss der Nutzer aus der VM heraus oder hart beenden. Liegen bei ersten Experimenten mehrere VM-Leichen im Speicher, kann er als letzten Ausweg alle VMs gemeinsam per *vmctl reset vms* abschießen.

Über welche weiteren Parameter der Systemverwalter VMs beim Start über das Terminal konfigurieren kann, zeigt die Manualpage zu *vmctl*. Gerade beim Experimentieren mit VMM helfen zwei weitere Befehle: *rcctl restart vmd* lädt den Hypervisor samt Konfigurationsdatei komplett neu; ferner ist es praktisch, in einem zweiten Terminal eventuelle Fehlermeldungen per *tail -f /var/log/messages* mitzulesen.

Konfigurierte VMs

Benötigt man VMs regelmäßig, lassen sie sich umfangreicher und vor allem komfortabler in der */etc/vm.conf* konfigurieren. Die Datei muss der Nutzer neu anlegen, ein Beispiel liegt wie bei OpenBSD üblich unter */etc/examples/vm.conf*. Die Syntax entspricht dabei der, die der Systemverwalter bereits von *pf* (Paketfilter), *httpd* (Webserver) und anderen Diensten kennt – das erleichtert die Administration. Auch in der *vm.conf* kann er also Makros festlegen, es gibt globale und VM-spezifische Definitionen und zusätzlich einen Abschnitt zum Einrichten eines virtuellen Switches. Die Datei kann man optional per *include <Pfad>* logisch aufteilen, um beispielsweise für jede VM eine eigene *.conf*-Datei zu erstellen.

Der Kasten zur vm.conf zeigt ein Beispiel für eine */etc/vm.conf*, die zwei ähnliche VMs definiert. Selbst ohne tiefe Kenntnisse der OpenBSD-spezifischen *.conf*-Syntax ist die Datei verständlich: Zunächst definiert man Makros, die im einfachsten Fall Variablendeklarationen entsprechen. Ein Makro leitet der Anwender in der späteren Konfiguration durch ein *$*-Zeichen ein, woraufhin es den zuvor definierten Text ersetzt. Der erste der zwei Pfade in Listing 1 gibt das Verzeichnis für die virtuellen Maschinen an. */home/vm* bietet sich an, je nach Geschmack kann aber jedes beliebige Verzeichnis dienen – die



VMs können ebenfalls unprivilegierte Anwender nutzen, die sie dann starten, beenden und die Konsole öffnen dürfen (Abb. 2).

/home-Partition erhält allerdings bei einer standardmäßigen OpenBSD-Installation den meisten Platz. Das zweite Verzeichnis ist der *httpd*-Webserver des Hosts, auf den der Host des Autors einige OpenBSD-Repositories spiegelt. Auch diese Dateien können an anderen Orten liegen, wichtig ist nur, den kompletten Inhalt von beispielsweise */pub/OpenBSD/6.1/amd64/** eines OpenBSD-Mirrors verfügbar zu halten, selbst wenn man Dateien wie *.iso* oder *pxeboot* hier nicht benötigt.

Unter OpenBSD lassen sich virtuelle Switche (*switchd*, Pseudo-Gerät) definieren und über einen SDN-Controller (Software-Defined Networking) in Form von *switchctl* konfigurieren und steuern. Die Kommunikation erfolgt über das OpenFlow-Protokoll Version 1.3. VMM richtet über einen *switch*-Block ein solches Gerät automatisch als Bridge ein. Die Dokumentation nennt ihn gerne *uplink*. Ihr fügt der Nutzer der Netzwerkkarte des Hosts hinzu, die *ifconfig* anzeigt. Verbindet man VMs über *interface { switch “uplink” }* mit dem Switch, tauchen sie als eigenständige Netzwerkgeräte im LAN auf. Dabei gibt es einen Stolperstein: Setzt der Anwender für den Host DHCP ein, fängt dessen *dhclient* DHCP-Pakete ab. Letztere erreichen so niemals eine VM. Den Host sollte man also statisch konfigurieren, andernfalls muss man sich mit einer Konstruktion aus einem per NAT angebundenen *vether0*-Gerät, NAT im Paketfilter *pf* und einem lokalen DHCP-Server auseinandersetzen. Damit lassen sich VMs dann sogar auf einem Notebook übers WLAN ins Netz bringen. Näheres zeigt die OpenBSD Networking FAQ (siehe „Alle Links“).

Damit das System die Pakete der VMs vom und ins LAN transportieren kann, muss der Nutzer IP-Forwarding per *sysctl net.inet.ip.forwarding=1* einschalten. Die Einstellung kann er permanent sichern, indem er die Zeile ohne das *sysctl* in die */etc/sysctl.conf* einträgt. Ferner muss man beachten, dass OpenBSD standardmäßig vier *tap*-Geräte zur Verfügung stellt. Will der Anwender mehr als vier VMs starten, muss er zusätzliche per *MAKEDEV* generieren:

```
cd /dev
ls tap*
sh MAKEDEV tap4
sh MAKEDEV tap5
...
```

Nun definiert der Systemverwalter die VMs in der */etc/vm.conf*, eingeleitet durch *vm* und einen frei wählbaren Namen sowie eingefasst in geschweifte Klammern. Die Konfiguration erfolgt über bislang sieben Schlüsselworte. Die wichtigsten zeigt Listing 1. *memory* legt die Größe des Arbeitsspeichers fest. Ohne Angabe vergibt VMM standardmäßig 512 MByte; seit Anfang August hob ein Commit von Mike Larkin die Obergrenze von maximal 3855 MByte auf, sodass man nun maximal Werte bis zu MAXDSIZ der jeweiligen Architektur einsetzen kann – bei amd64 sind das 32 GByte.

boot verweist auf den zu startenden Kernel- oder das BIOS-Image. Bei OpenBSD ist *bsd.rd* der RAMDisk-Kernel mit dem Installer. Über *disk* richtet man virtuelle Laufwerke ein, leere Festplatten-Images lassen sich mit *dd* oder per *vmctl* erstellen. Für die beiden exemplarischen virtuellen Maschinen aus Listing 1 geschieht das folgendermaßen:

```
mkdir -p /home/vm
mkdir /home/vm/snap-amd64
mkdir /home/vm/snap-i386
vmctl create /home/vm/snap-amd64/disk1.img -s 8G
vmctl create /home/vm/snap-i386/disk1.img -s 8G
```

Die zweite *disk*-Direktive zeigt auf *install61.fs*, wobei es sich um das Installations-Image handelt. Als letzten Schritt bindet der Systemverwalter mit dem *interface*-Statement die *vio0*-Netz-

werkkarte der virtuellen Maschine an den zuvor definierten virtuellen Switch an.

In Listing 1 weist die *snap-i386*-VM einige Besonderheiten auf. Das Schlüsselwort *disable* verhindert, dass VMM die VM beim Systemstart oder dem Start von *vmd* hochfährt – sie muss man also von Hand ins Leben rufen. Die beiden Verweise auf den RAMDisk-Kernel und das Installations-Image sind auskommentiert, weil die VM bereits auf *disk1.img* vorliegt und nun direkt hiervon starten soll. Schließlich erhält ein Nutzer per *owner mipl* das Recht, die VM zu starten, zu beenden und sich mit ihrer Konsole zu verbinden. Ganzen Nutzergruppen kann der Administrator dieses Recht per *owner :<Gruppe>* übertragen.

Ein Systemneustart oder *rcctl start vmd* startet den Hypervisor und sämtliche nicht per *disable* abgeschalteten VMs. Zu Debuggingzwecken lässt sich *vmd -dvvvv* einsetzen. Nutzer sollten sie nicht als Daemon starten, denn die Software neigt zu sehr ausführlichen Meldungen. Syntaxfehler in */etc/vm.conf* zeigt *vmd -n* an, wobei man mit *-f* eine alternative Konfigurationsdatei spezifizieren darf. Informationen zu den VMs bietet *vmctl status*. Im Beispiel läuft die amd64-VM, während die i386-VM zwar definiert ist, aber nicht läuft. Der Nutzer *mipl* kann sie via *vmctl start <ID>* starten, sie per *stop* beenden oder sich über *console* mit der Konsole verbinden. Für die amd64-VM hat er all diese Rechte gemäß */etc/vm.conf* nicht.

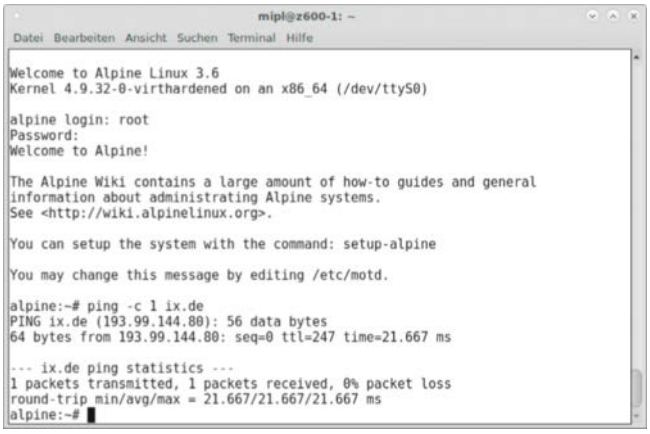
In der amd64-VM kann man nun OpenBSD installieren, anschließend muss *root* sie per *vmctl stop <ID>* beenden. Wie bei der i386-VM kommentiert man die beiden Zeilen zu Install-Kernel und -Image aus und fordert anschließend *vmd* mit einem *vmctl reload vmd* dazu auf, die Konfiguration neu einzulesen. Dabei und bei jedem Neustart startet die amd64-VM direkt vom virtuellen Festplatten-Image.

Noch ein Tipp: Auf dem Host sollte der Systemverwalter die *.profile* von *root* um die Zeile *PS1="\u@\h:\w# „* ergänzen, um das schlichte #-Prompt durch ein *root@vmmhost:~#* zu ersetzen. Für unprivilegierte Nutzer ersetzt dies das # durch *$*. Auf diese Weise bemerkt der Administrator sofort, wenn er nicht in einer VM, sondern auf dem Host arbeitet und vermeidet so unangenehme Fehler.

OpenBSD und Linux-Gäste

Seit OpenBSD 6.1 unterstützt VMM ein virtuelles BIOS und somit das Starten von anderen Betriebssystemen, wobei sich das zur Zeit noch auf wenige Linux-Distributionen beschränkt. Am erfolgversprechendsten ist Alpine Linux, eine minimalistische, auf *musl* – ein kleiner und schneller *glibc*-Ersatz – und BusyBox basierende Distribution. Als Linux-System ist sie im BSD-Lager recht beliebt, weil sie speziell auf Sicherheit, Einfachheit und Ressourcenschonung ausgelegt ist. Daher benutzt sie statt *systemd* das ebenfalls von Gentoo eingesetzte und zur BSD-Welt kompatible OpenRC-Init. Die Konfiguration einer Alpine-VM ist ebenfalls in Listing 1 enthalten. Die Installation erfolgt wie bei der i386-VM beschrieben: Zunächst startet man vom auf virtuelle Maschinen optimierten ISO-Image. Im Live-System kann *root* ohne Passwort die Installation per *setup-alpine* anwerfen.

Der Befehl kopiert Alpine auf das virtuelle Festplatten-Image, anschließend muss man das ISO-Image in der */etc/vm.conf* auskommentieren. Alpine Linux läuft unter OpenBSDs VMM, allerdings tauchen in */var/log/messages* regelmäßig Fehler auf – und hin und wieder kann die VM abstürzen oder einfrieren. Mit dem im Herbst kommenden OpenBSD 6.2 dürften Linux-Gäste stabiler laufen. Der Versuch, mit Devuan GNU/Linux einen *systemd*-befreiten Fork von Debian „Jessie“ zum Laufen zu bekommen, scheitert leider bereits beim Umschalten auf die serielle Konsole.



Linux-Gäste wie hier Alpine Linux laufen unter VMM, sofern sie sich über eine serielle Konsole installieren lassen. Ein Framebuffer fehlt VMM noch (Abb. 3).

Für den produktiven Einsatz ist VMM aus OpenBSD 6.1-current wegen kleiner Reibereien gerade mit virtueller Hardware aus den 80ern (i8259 ohne FiFo) noch nicht ganz fit, die Programmierer entwickeln die Software aber intensiv weiter (Abb. 4).

Ausblick

Zur Zeit geht die Entwicklung von OpenBSDs VMM mit großen Schritten voran. Der Einsatz von *pledge* zum Einschränken auf die ausschließlich benötigten Systemfunktionen, Separation von Privilegien und ein konsequentes fork+exec-Modell sollen VMM äußerst sicher machen, sodass unprivilegierte Nutzer VMs ohne Gefahren für die restlichen Systeme einsetzen können. Technisch ist die Entwicklung der virtuellen Switche als Alternative zum Bridging interessant und vereinfacht den Aufbau virtueller Netzwerke. Die Interface-Gruppen können sogar mit dem Paketfilter *pf* interagieren. Der SMP-Support für VMs steckt allerdings noch in den Kinderschuhen.

Der OpenBSD-Hypervisor, an dem neben weiteren Entwicklern vor allem Mike Larkin und Reyk Floeter arbeiten, öffnet viele neue Einsatzgebiete für das auf Sicherheit getrimmte Betriebssystem. Mit der kommenden Version 6.2 sollten Unternehmen VMM zumindest mit OpenBSD-Gäste produktiv einsetzen können. Die Option, mit Alpine Linux Linux-Dienste auf einem sicheren Host zu betreiben, eröffnet für Hosting-Aufgaben neue Wege. (fo)

Literatur

[1] Michael Plura; Emsig; Virtualisierung: VMs unter FreeBSD und OpenBSD mit Bhyve und vmm; *iX* 6/2017, S. 60

Michael Plura
lebt in Schweden und ist freier Autor mit den Schwerpunkten IT-Sicherheit, Virtualisierung und freie Betriebssysteme.

QEMU für den Betriebssystemzoo

Abgezapft

Michael Plura

Der Quick Emulator, oder auch QEMU, steckt in vielen Virtualisierungsdiensten, von KVM und Xen über den Emulator im Android-SDK bis hin zu VirtualBox. Eigenständig eingesetzt kann QEMU wesentlich mehr und sogar architekturfremde Gäste ausführen – Nutzer müssen sich jedoch auf Hürden gefasst machen.

Virtualisierung ist zu einem selbstverständlichen Werkzeug auf dem Desktop, dem Server und im Rechenzentrum geworden. Meist konzentrieren sich der Hypervisor auf dem Host und die zugehörigen virtuellen Maschinen (VMs) ausschließlich auf Systeme mit Intel- oder AMD-Prozessoren, also die x86-Architektur. Für letztere gibt es Software fürs Büro wie VMwares Workstation, Parallels Desktop und Oracles VirtualBox, aber auch Werkzeuge für die Infrastruktur wie die KVM (Kernel-based Virtual Machine) und Xen bis hin zu vollständigen Virtualisierungsumgebungen wie VMwares ESX/ESXi vSphere, Microsofts Hyper-V, Oracles VM Server oder Citrix' XenServer. Solche Dienste für den Server sind Typ-1-Hypervisoren und laufen ohne ein darunterliegendes Betriebssystem direkt auf der Hardware. Bei Desktop-Anwendungen handelt es sich um Typ-2-Hypervisoren, die Funktionen und Gerätetreiber des Host-Betriebssystems bedienen. Oracles VM wagt einen Spagat, denn für x86 gehört sie zum Typ-2, für SPARC hingegen zum Typ-1.

Ferner stellt Oracles SPARC-Hypervisor eine der wenigen Ausnahmen dar und zeigt, dass Virtualisierung genauso auf Nicht-x86-Architekturen funktionieren kann. Voraussetzung ist ein SPARC-V9-Prozessor (UltraSPARC T1 und M5 aufwärts). SPARC-Systeme lassen sich in Logical Domains (LDOMs) partitionieren, auf denen Nutzer vollständige Betriebssysteme wie Solaris, Illumos, Ubuntu oder OpenBSD installieren können. Diese Partitionierung stellt den Ursprung der Virtualisierung dar und erzeugt sogenannte LPARs (Logical Partitions), die einen Teil der Hardware als separates System definieren. IBM arbeitete an dem Konzept bereits unter CP/CMS-40 (Control Program/Cambridge Monitor System) auf Mainframes der S/360- und S370-Baureihe um 1970. Eine weitere Ausnahme ist Xen/ARM, das einen ARM v7-A oder ARM v8-A voraussetzt und die volle Hardware-Virtualisierung eines ARM Cortex A15 nutzt. Bei KVM gibt es Bestrebungen, Hardware-Virtualisierung neben x86/x86_64 auch für PPC 440/970, S/390, ARM (Cortex A15, AArch64) und MIPS32 zu implementieren.

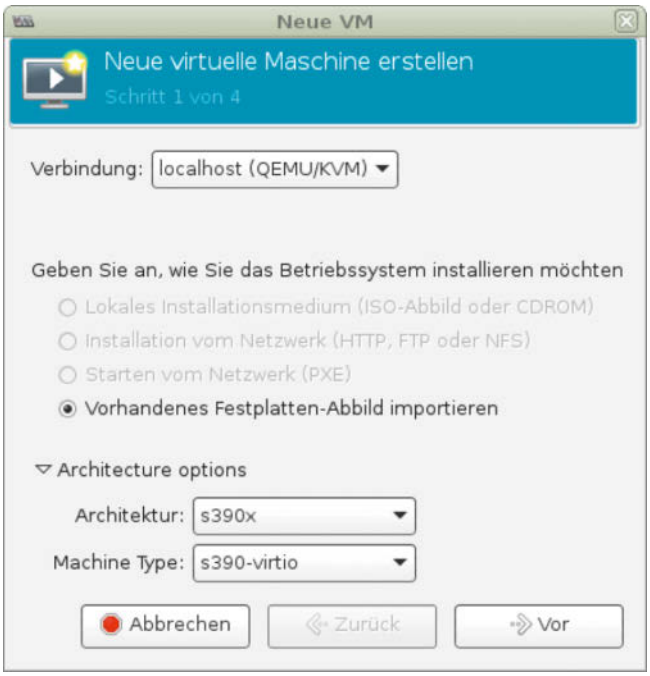
Eines haben alle bisher genannten Dienste gemeinsam: Host- und Gastsystem mögen zwar unterschiedliche Betriebssysteme verwenden, ihre Architektur muss dennoch identisch sein. Auf einem x86 gibt es ausschließlich x86-Gäste, auf einem SPARC-Host bleibt es bei SPARC-VMs. Das ist konzeptionell so gewollt, denn die Virtualisierung konzentriert sich vor allem auf eine hohe Ausführungsgeschwindigkeit. Darum führen die Host-CPUs fast den kompletten Maschinencode des Gasts nativ aus – einzig privilegierte Befehle und nicht-privilegierte Befehle mit variablem Verhalten muss die Software emulieren, um den Hypervisor nicht zu kompromittieren. Ein x86- kann beispielsweise kein SPARC-System virtualisieren, weil die beteiligten Prozessoren mit vollständig unterschiedlichen Befehlssätzen arbeiten. Alternativ könnten Entwickler alle Befehle als Ausnahme behandeln, also müsste die Anwendung den gesamten Befehlssatz emulieren. Bei so einer Virtualisierung spricht man von der Software-Emulation. Weil bei ihr die Host-CPU für jeden einzelnen Maschinenbefehl des Gastsystems viele Befehle ausführen muss, müssen Nutzer bei einer Emulation grundsätzlich mit einem deutlichen Geschwindigkeitsverlust leben.

Deutliche Leistungsverluste

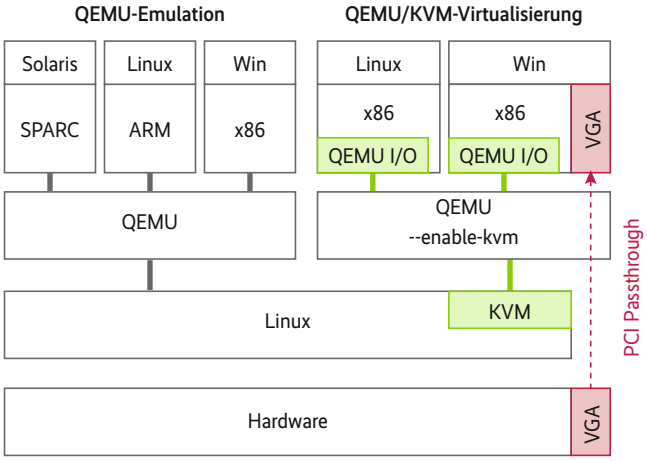
Ein einfacher Test zeigt den Unterschied. Dazu berechnet der Basic Calculator *bc* Pi mit 1000 Stellen (*4 * arctan(1)*, *„-l“* benutzt die mathematische Bibliothek) und das System misst die benötigte Zeit mit *time*:

```
time echo "scale=1000; 4*a(1)" | bc -l
```

Ein 64-bittiges Debian GNU/Linux 9 „Stretch“ benötigt dafür nativ auf einem HP Z600 (2 * E5630@2.53GHz) 0,78 Sekunden (user time). Virtualisiert per QEMU/KVM steigt der Wert auf 0,82 Sekunden, die volle Emulation in QEMU ohne KVM hingegen verschlingt neben mehreren Minuten Startzeit 4,97 Sekunden für die Berechnung. Dasselbe gilt in der Praxis: Reine Virtualisierung läuft mit wenigen Prozent Leistungseinbußen gegenüber dem Host-System, während eine Emulation eher um den Faktor fünf (reine CPU-Leistung) bis zwanzig (I/O-Aufgaben) spürbar langsamer ist. Und dabei ist QEMU sogar recht effizient, da die Software den Code der Gast-Architektur zu großen Teilen nur einmal in Code des Hosts übersetzt und ihn anschließend in einem 32 MByte großen Translation Cache zwischenspeichert.



Teile des Quick Emulator QEMU stecken in vielen Virtualisierungsprodukten, Frameworks wie der Virt-Manager nutzen ihn automatisch (Abb. 1).



QEMU emuliert viele Rechnerarchitekturen, kann mit Hilfe von KVM aber auch virtualisieren und via PCI Passthrough Hardware direkt an Gäste durchreichen (Abb. 2).

Für die Virtualisierung eines x86-Systems auf einem x86-Host eignet sich QEMU ohne KVM daher nicht. Hier greift der Systemverwalter besser zu QEMU als Bestandteil von Xen oder KVM in Kombination mit einem Frontend wie dem Virt-Manager.

QEMU separat einzusetzen bietet sich jedoch dann an, wenn Nutzer ein Gastsystem auf Basis einer nicht mit dem Host identischen Architektur benötigen. Entwickler könnten beispielsweise auf einer X86_64-Workstation Code für ein ARM-Board schreiben wollen oder Administratoren einen alten Sun SPARC- auf einen x86_64-Server migrieren müssen. QEMU bietet sich hierfür an, denn neben i386/x86_64 bildet der Emulator eine Reihe anderer Plattformen für Gastsysteme nach: ARM, MIPS, PowerPC, SPARC, Xtensa sowie teilweise auch Alpha, Etrax CRIS, SH4 und sogar M68k.

CPU-Vielfalt für freie Systeme

In fast allen Open-Source-Betriebssystemen lässt sich QEMU über die Paketverwaltung installieren. Den Namen des Paketes können Nutzer teils nicht intuitiv erraten: Bei Debian, SUSE oder Arch lautet er „qemu“, bei Gentoo schon „app-emulation/qemu“, bei RHEL/CentOS „qemu-kvm“ und bei Fedora gar „@virtualization“. Die Versionen sind vor allem bei Debian und RHEL/CentOS recht veraltet. Im Juni 2017 ist QEMU 2.9 aktuell, Debian „Jessie“ liefert eine gepatchte Version 2.5 (2015), Debian „Stretch“ immerhin die 2.8 von Ende 2016. Bei den BSDs bringt FreeBSD 11 ein QEMU 2.8 mit, während OpenBSD mit Version 2.9 dem jeweils aktuellen Entwicklungsstand folgt. macOSx installiert „qemu“ über Homebrew, Windows-Anwender müssen sich die jeweiligen binären Installer von der Projektseite herunterladen.

Bei QEMU ist es wichtig, eine aktuelle Version einzusetzen. Häufig ändert sich einiges an der Befehlssyntax, was inbeson-

dere Anleitungen im Internet schnell veralten lässt. Viele Funktionen gerade im Bereich ARM haben die Entwickler erst vor kurzem hinzugefügt. Hinzu kommt: Je nach Linux-Distribution ist QEMU nicht nur veraltet, sondern teilweise auch aufgebläht – oder eingeschränkt. Das zugehörige Paket konfiguriert immer ein Maintainer, der manche Funktionen als nicht einsatzwürdig erachtet und daher nicht mitübersetzt. „Simple DirectMedia Layer“ via *-display sdl*, eine plattformunabhängige Schnittstelle (API) für Grafik-, Sound- und Eingabegeräte steht beispielsweise unter RHEL ebenso wenig zur Verfügung wie die Emulation einer Standard-Grafikkarte durch *-vga std*. Stattdessen legt das RHEL-QEMU den Schwerpunkt auf SPICE und QXL. Das mag im Enterprise-Umfeld und vor allem auf einem Headless-Server zwar sinnvoll sein, spezielle Anforderungen bleiben aber außen vor. So wie bei RHEL viele Funktionen fehlen, stellt Debian GNU/Linux 8 das andere Extrem dar, bei dem die Entwickler sicherheitshalber gleich alle Funktionen mitliefern:

```
ldd $(which qemu-system-x86_64) | wc -l
```

liefert eine Liste von fast hundert referenzierten Bibliotheken: von X11-Libs über SDL, PulseAudio, DirectFB bis hin zu Codecs wie FLAC, Ogg, Vorbis. Aus diesem Grund ist es durchaus sinnvoll, QEMU selbst aus den Quellen zu kompilieren.

QEMU für den Einsatz anpassen

Um QEMU unter Linux zu übersetzen müssen Nutzer neben den üblichen Entwicklerwerkzeugen auch diverse Bibliotheken installieren. Für Debian GNU/Linux und Ubuntu sind erforderlich:

```
apt-get install build-essential libglib2.0-dev libfdt-dev \
                                                libpixman-1-dev zlib1g-dev
```

RHEL und CentOS 7 benötigen hingegen:

```
yum groupinstall "Development Tools"
yum install glib2-devel libfdt-devel pixman-devel zlib-devel
```

Zusätzlich empfohlene Pakete finden sich im QEMU-Wiki (siehe „Alle Links“). Wer QEMU selbst weiterentwickeln will, sollte „bleeding-edge“ aus dem Git-Repository verwenden und muss zusätzlich das Paket *git* installieren:

```
git clone git://git.qemu-project.org/qemu.git
cd qemu
git submodule init
git submodule update --recursive
```

Reine QEMU-Anwender sollten das stabile Release einsetzen, aktuell Version 2.9. Sie lässt sich als xz-Archiv herunterladen und auspacken:

```
wget http://download.qemu-project.org/qemu-2.9.0.tar.xz
tar xvJf qemu-2.9.0.tar.xz
rm qemu-2.9.0.tar.xz
mv qemu-2.9.0/ qemu/
cd qemu
```

In beiden Fällen liegt der Quellcode im Verzeichnis *qemu/*. Konfigurieren lässt sich die Software mit dem üblichen *./configure*. Ein angehängtes *–help* listet neben der beeindruckenden Anzahl an Emulationszielen (*–target list=*) auch alle weiteren Optionen. Für Experimente mag sich das komplette Paket eignen, in der Praxis benötigt ein Nutzer jedoch bloß wenige, wenn nicht gar nur ein einziges Zielsystem. Um beispielsweise eine alte Sun Workstation („sparc“ 32-Bit) zu emulieren, reicht ein:

```
./configure --disable-kvm --target-list=sparc-softmmu
```

Da es für die Zielplattform keine KVM-Virtualisierung gibt, lässt sie sich per *–disable-kvm* abschalten. Mehrere Zielplattformen müssen Nutzer als Liste aufführen *–target-list=“i386-softmmu sparc-softmmu“*. Beachten müssen Anwender allerdings, dass sie auf einem x86-System mit i386 oder x86_64 als Zielplattform die KVM-Virtualisierung nicht mehr deaktivieren sollten. Wie üblich startet *make* den eigentlichen Compiler-Lauf.

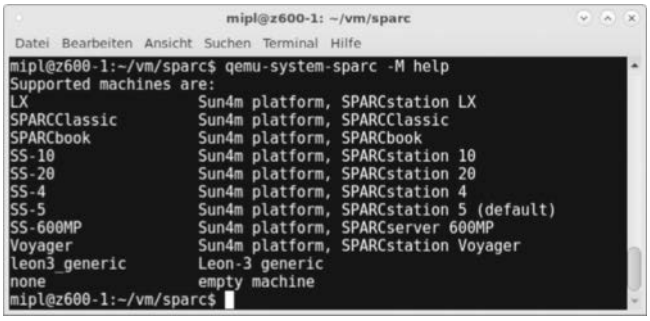
Die QEMU-Dokumentation empfiehlt, anschließend die Binaries im System unter */usr/local/bin* zu installieren und so allen Benutzern zur Verfügung zu stellen. Die Binaries finden sich in nach Architektur getrennten Verzeichnissen, bei SPARC zum Beispiel unter *~/qemu/sparc-softmmu/qemu-system-sparc -version*.

Testen kann man die erzeugten Binaries mit einigen etwas angestaubten Test-Images des QEMU-Projektes. Sie stehen für Zielplattformen und die User-Mode-Emulation zur Verfügung. Weitere Tests wie für I/O-Funktionen sollten Nutzer ebenfalls durchführen:

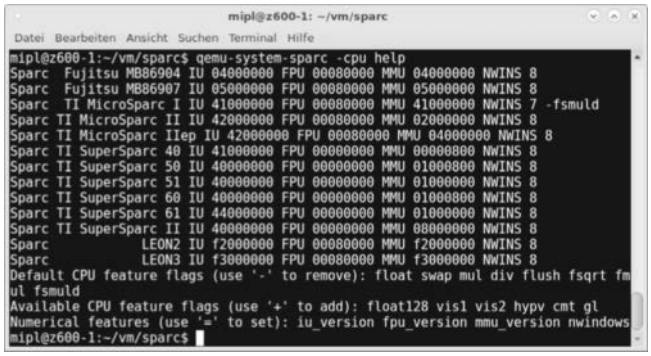
```
cd ~/qemu/tests/qemu-iotests
./check -qcow2 [<test-case>]
```

Mit Version 2.9 läuft der I/O-Test unter Debian „Stretch“ bis auf Test 087 sauber durch, unter Ubuntu 16.04 bricht er bei Schritt 45 ab – nicht gerade beruhigend. Um einen konkreten Testlauf des erzeugten SPARC-Binaries durchzuführen, verwendet man ein 4,4 MByte großes *sparc-test-0.2.tar.gz*-Test-Image, das sich auf dem Listing-Server findet:

```
tar xvzf sparc-test-0.2.tar.gz
cd sparc-test
cp ~/qemu/pc-bios/openbios-sparc32 .
```



Über die Hilfe, die Manualpages und eine etwas knappe Online-Dokumentation tastet sich der QEMU-Nutzer langsam an alle benötigten Parameter heran, um wie hier eine 32-bittige SPARC-Maschine der Sun4m-Familie zu erzeugen (Abb. 3).



Bei vielen Systemen erlaubt QEMU die genaue Definition der zu emulierenden CPU(s) bis hin zu speziellen CPU-Flags (Abb. 4).