Taking full advantage of QEMU in the Xen userspace

Daniel P. Berrangé berrange@redhat.com

Xen Summit November 2007

Abstract

The Xen userspace has evolved over time and is now comprised of many daemons, librararies and helper programs. To get a single domain running requires all of these components to be installed & operating normally. When tracking xen-unstable development this can be a trying task as hypervisor ABIs evolve & program interactions change. By comparison to get a KVM guest running requires a single QEMU userspace binary. This paper will examine how Xen can take full advantage of QEMU to simplify the userspace management stack & bring it on a par with KVM.

Keywords

Xen, QEMU, Virtualization, libvirt, KVM, Open Source

Contents

List of Figures

1 Introduction

This section provides some introductory material describing the current Xen userspace management infrastructure. It outlines some shortcomings in the currrent code, and compares it with that available in KVM.

The current Xen userspace is comprised of a large number of daemons, libraries and helper programs. To get a domain up and running requires that all of these components be installed & their versions matched. While tracking the xen-unstable development tree it is not uncommon for hypervisor ABIs to change, for major code refactoring to take place in XenD, and for interactions between XenD and the other components to change. The interactions between all of these components make debugging and tracing operations a complex & troublesome process. This is a list of the commonly set of components needed for general operation of Xen guests, to support paravirt, fullvirt and save/restore/migration.

- **libxenctrl** provides a lowest level C library interface to the hypervisor. For most hypercalls, it essentially just exposes the hypercalls directly as simple C APIs.
- **libxenguest** an extension of the libxenctrl library providing some higher level APIs for the tasks for domain creation, save, restore and migration. These will typically result in multiple hypercalls.
- **libxenstore** provides a C library for talking to the XenStore, the virtual filesystem / datastore.
- **xenstored** is a daemon implementing the virtual filesystem, and interfacing to the XenBus communication protocol.
- **xenconsoled** is a daemon implementing the paravirtualized text console protocol.
- **xend** is a daemon implementing the core management API, both the legacy SEXPR and XML-RPC protocols, and the recommended Xen-API protocol. It also tracks and co-ordinates all management tasks
- **xen-vncfb/xen-sdlfb** are helper programs launched per guest to implement the paravirtualized graphical console protocol.
- **udev/hotplug scripts** are helper scripts to setup backend devices for disk, network and other paravirtualized devices with Domain-0 backends.
- **qemu** is a helper program providing an emulated device model for fuly virtualized guests
- **xcsave/xcrestore** are helper programs for performing save, restore and migration of guests. They are invoked from XenD

- **blktapctrl** is a daemon to co-ordinate all userspace paravirtualized disk backends
- **tapdisk** is a helper program launched for each userspace paravirtualized guest
- **pygrub** is a helper program presenting a bootloader menu for paravirtualized guests in the style of the traditional **grub** program.

Figure 1 shows the relationship between these components in the Xen 3.1.0 release.

1.1 Previous issues

In the development process leading upto the Xen 3.2.0 release a couple of issues were addressed with the management of paravirtualized guests and their text and graphics consoles. Figure 2 shows the updated architecture following this work. The key differences from Xen 3.1.0 (as seen in Figure 1) is the removal of xen-vncfb and xenconsoled as required components.

1.1.1 Graphics consoles

One of the most glaring problems in the managment space was the presence of two completely separate VNC server implements. Paravirtualized guests used the **xen-vncfb** helper program which linked to **LibVNCServer**, while fullyvirtualized guests ued the VNC server built-in to **QEMU**. Both VNC server implementations had their own strengths and weaknesses, but ultimately the **LibVNCServer** impl was more trouble with a large number of thread race conditions and very hard to understand code. When the time came to add TLS/x509 security to VNC there was little desire to implement the code twice. Following discussions at the April 2007 Xen summit & subsequently on the #xen IRC channel it was decided to adapt QEMU to serve as a VNC server for paravirtualized guests.

The approach taken for this was to implement a new machine type for the Xen-ified QEMU 'xenpv', while the existing machine type (for fullyvirtualized guests) was renamed to 'xenfv'. The 'xenpv' machine type directly implemented the Xen paravirt framebuffer backend protocol & plumbed it into QEMU. This provided by VNC and SDL display options for paravirt. Not counting the removal of **LibVNCServer**, the before & after lines-of-code for the paravirt framebuffer userspace were approximately the same. Taking into account the code for the subsequent VNC server extensions for TLS/x509 security, the unification was a clear win.

The QEMU command line required to support the paravirt graphical framebuffer looks like

```
# For SDL graphics
qemu-dm -d 5 -domain-name demo -M xenpv

# For VNC graphics
qemu-dm -d 5 -domain-name demo -M xenpv \
   -vnc 127.0.0.1:0,password -vncunused
```

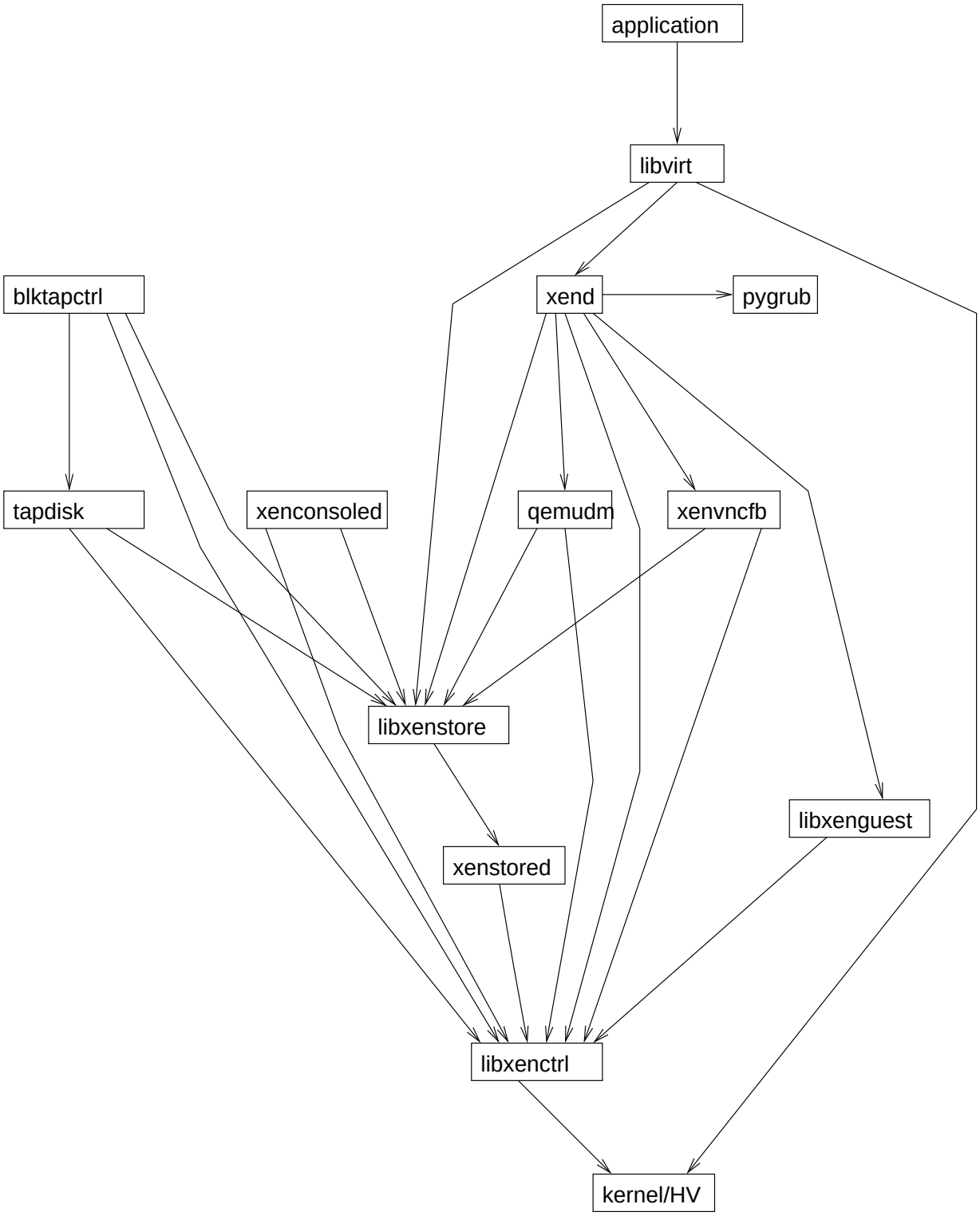

Figure 1: Xen userspace architecture in 3.1.0 release

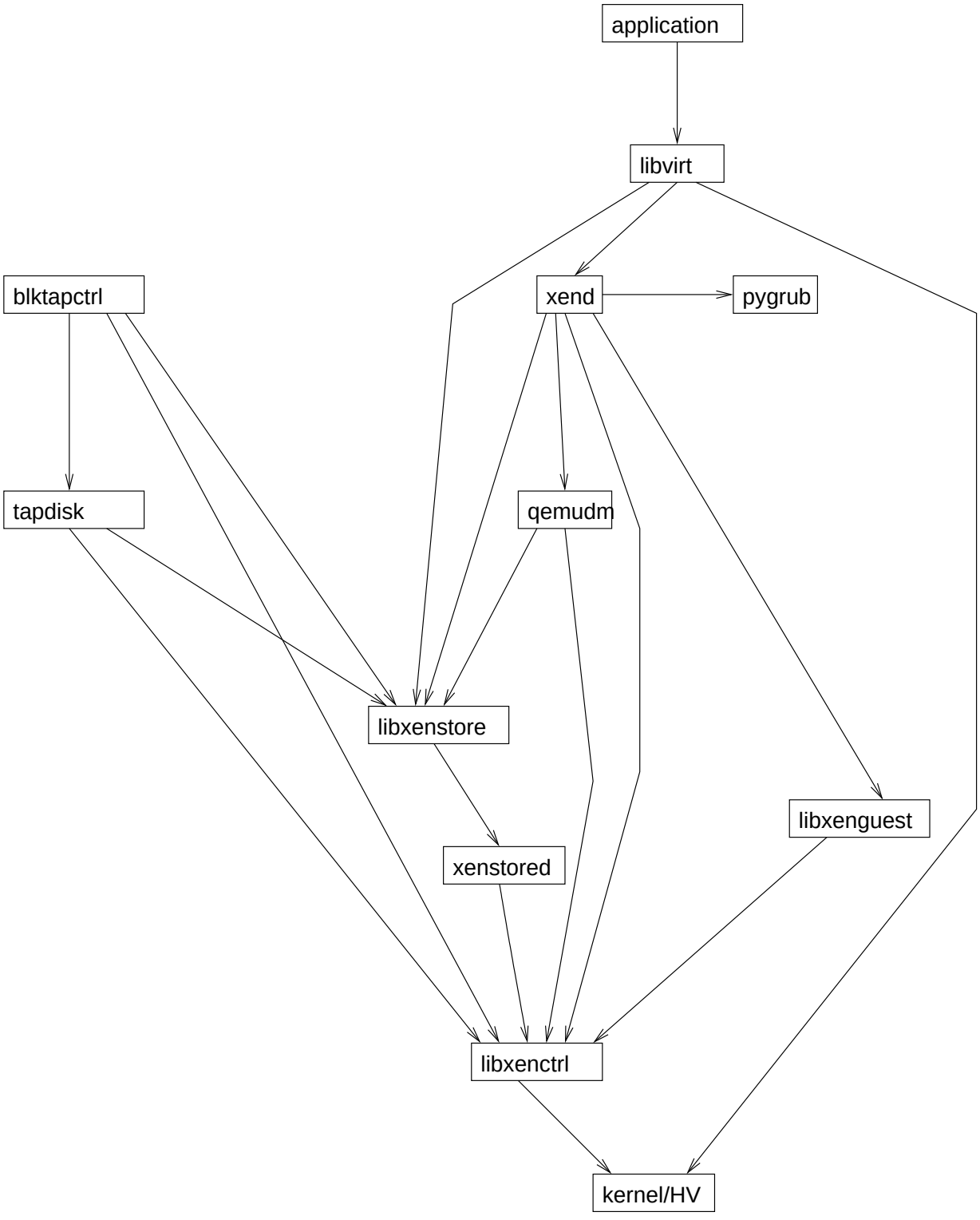

Figure 2: Xen userspace architecture in 3.2.0 release

1.1.2 Text consoles

Immediately following the implementation of the paravirt framebuffer, analysis turned to **xenconsoled**. This persistent daemon watches for new guest domains, and when one appears, connects its text console to a psuedo-TTY. Fullyvirtualized guests do not have a Xen text console, however, they do have emulated serial ports which serve a similar purpose. QEMU can expose serial ports as text files (for logging purposes), pseudo-TTYs, UNIX domain sockets, TCP sockets, and QEMU virtual consoles. It did not take much imagination to see that it would be possible to connect a paravirtualized text console upto the same infrastructure. Support for this was merged into xen-unstable at the same time as the VNC server unification, thus making another of the aforementioned management components redundant.

The QEMU command line required to support the paravirt text console looks like

```
# For file logging
qemu-dm -d 5 -domain-name demo -M xenpv \
  -serial file:/var/log/xen/console/demo.log

# For psuedo-tty
qemu-dm -d 5 -domain-name demo -M xenpv \
  -serial pty
```

1.2 Further issues

The success in unifying the userspace management of text and graphics consoles for paravirtualized and fullyvirtualized guests inevitably leads to more questions & ideas...

1.2.1 Bootloaders

When booting fullyvirtualized guests, there is a full BIOS which enables a domain to be booted from floppy, harddisk, cdrom or the network. For local hardisk boot, it is typical to see a bootloader like **grub** or **lilo** which presents a menu of available kernels installed from the guest-OS. With network boot, the PXE protocol is implemented allowing the kernel and initrd to be fetched from a remote server. In all cases the bootloader is running from the context of the guest. This means the bootloader automatically uses the guest network connectivity, and does not have to worry about the undering disk format (raw, qcow, vmdk, etc), and seemlessly appears via the guest graphical or text consoles.

When booting paravirtualized guests, there is no BIOS running - the Xen hypervisor just requires a kernel and initrd to be provided directly. The **pygrub** program was created to **simulate** a traditional bootloader. This program runs in the context of the host OS, and peeks inside the guest disks to find the bootloader config and extract the kernel & initrd. Some tricks are required for it to be able to access non-raw disks, and it is not plumbed into the guest graphical console - it is only available via a text console. There is

also an experimental network bootloader, **pypxeboot**. Again this suffers from being run in the host OS, being unable to send out packets with the guest's MAC address.

1.2.2 Direct kernel boot

As mentioned above, Xen paravirtualized guests are able to boot directly from a kernel & initrd. This is particularly useful when provisioning new guest OS images, because it is possible to automatically pass options directly to the installer via the kernel command line. This capability is not available for fullyvirtualized guests, which prevents full automation of their install processs unless using a PXE server & pre-assign configs.

1.2.3 Disk image format backends

There are a multitude of storage backends in use for virtual disks, from plain block devices, and raw files, through to advanced formats like QCow2 and VMDK. Paravirtualized guests have long supported the first two, while support for QCow2 & VMDK arrived for fullyvirtualized guests first via the QEMU device model. Xen 3.0.3 introduced the blktap driver for paravirtualized guests which essentially duplicated all of the QEMU disk format code. The motivation for this at the time, was that Xen was based on QEMU 0.8.2 which had synchronous I/O only. Since Xen 3.1.0 though, QEMU was updated to 0.9.0 with the result that there are now two implementations of disk format handlers both supporting asynchronous I/O, each with their own performance characteristics & bugs.

1.3 A brief look at the KVM userspace

The KVM virtualization project has the obvious advantage of starting with a clean slate and thus being able to learn from the history of Xen development. In its released form, it also only attempts to support fullvirtualization, using paravirtualized extensions only where applicable to improve performance. No attempt is (currently) made to provide a pure paravirtualized solution. As would be expected, the userspace management stack is considerably simpler. Figure 3 illustrates the components in the KVM userspace.

The libkvm is the equivalent of the two libxenctrl/libxenguest libraries in the Xen world. The QEMU component is a superset of the QEMU device model present in Xen. In KVM, the QEMU binary directly takes care of talking to the hypervisor to create the guest domain. In Xen, the QEMU binary merely provides the I/O emulation, while XenD takes care of actually creating the domain. Similarly, QEMU handles save, restore & migration of KVM guests, and also provides the host end of the paravirtualized drivers avoiding any need for a blktap equivalent daemon. There is no general purpose virtual filesystem on a par with xenstore.

The interesting result of having domain creation inside QEMU itself, rather than merely using it for I/O emulation, is that the task of launching a guest reduces to a simple QEMU command line:

```
# Direct kernel boot
```

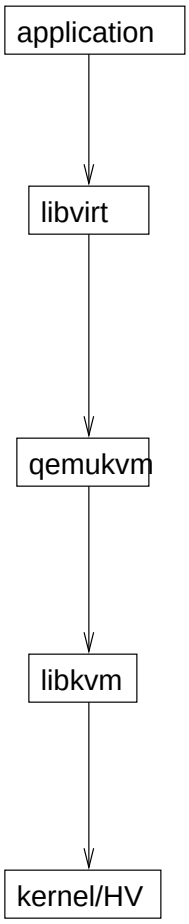

Figure 3: KVM userspace architecture

```
qemu-kvm -kernel /root/vmlinux.f8.x86_64 \
         -initrd /root/initrd.img.f8.x86_64 \
         -append "ks=http://example.org/fedora/8/x86_64/ \
         -m 500 \
         -hda /dev/VolGroup00/DemoGuest
```

The QEMU program provides a monitor shell console for interacting with the running VM. This enables USB device hotplug/remove, pause/resume of the CPU, save, restore and migration. For example

```
# Pause the CPU
(qemu) stop
# Resume the CPU
(qemu) cont
# Add a USB device from the host
(qemu) usb_add 0x4502:0x3422
# Save a snapshot
(qemu) save somesnapshot
# Migrate to a remote host
(qemu) migrate tcp:example.org:40
```

This provides a very quick turn around for developers to build & test changes to KVM - no need to continually re-install, and restart a large set of daemons (some of which can't be restarted with a host reboot). It provides a very easy way for end users to get VMs up & running without having to learn a special set of management tools. Management daemons / tools / libraries can be layered ontop of the core 'qemu-kvm' binary if desired, but this is strictly optional. Constrast with Xen, where it is neccessary to deploy as many as 4 daemons, 3 libraries and numerous helper programs.

2 Xenite - aka Xen lite

This section describes the code changes made to Xen to address some of the issues identified in the previous section. The result is a new mode of operation for qemu-dm, referred to as Xenite, or Xen lite.

The goal of Xenite is to address the issues identified in the previous section, with the aim of making the Xen domain boot process as simple as that of KVM. ie no more than launching a qemu binary with appropriate command line arguments. Figure 4 illustrates the goal for the minimal userspace architecture required under Xenite. The key difference from the Xen 3.2.0 architecture in Figure 2 is the elimination of XenD, and the direct execution of pygrub from the QEMU process. The elimination of blktap would be desirable too, but that is out-of-scope at this time.

2.1 Direct kernel boot

The first issue to be addressed is the ability to do a direct kernel boot for fullyvirtualized guests. The code for doing this was recently re-written in upstream QEMU CVS by Peter Anvin, to correctly comply with the Linux boot process in **Documentation/i386/boot.txt**. For purposes of the discussion that follows only boot protocol 2.0.2 or later (kernel ¿= 2.4.0) will be considered.

2.1.1 QEMU i386 direct kernel boot process

There are between 1 and 3 inputs when booting a kernel. The kernel image itself is compulsory; it may optionally be accompanied by an initrd image and a command line args string. The kernel image is split into two parts, a small real-mode block containing the boot sector and associated setup code, and a large protected-mode block containing the bulk of the kernel.

The real-mode block has to be loaded at an address somewhere between 0x10000 and the end of low-memory, while the protected-mode kernel is non-relocatable and must live at 0x100000 in high-memory. In the 2.04 revision of the boot protocol, (available since 2.6.20 on i386 and 2.6.22 on x86_64) the protected-mode kernel is now relocatable to anywhere in high memory. The initrd can be loaded anywhere in high memory, and is typically put right at the end of physical memory, minus an offset to allow for ACPI data tables. The kernel command line string is located in low-memory at 0x20000.

When given a kernel to boot directly, QEMU loads the various bits into guest RAM at the locations mentioned, simply using memcpy() since guest RAM is mapped directly into QEMU at a fixed offset. It then generates a fake bootsector for the first hard disk containing a few bytes of code which setup a few registers, and then jump straight to the start of the real-mode kernel at 0x10000. The real-mode kernel does its setup magic, switches to protected-mode and jumps to the protected mode kernel.

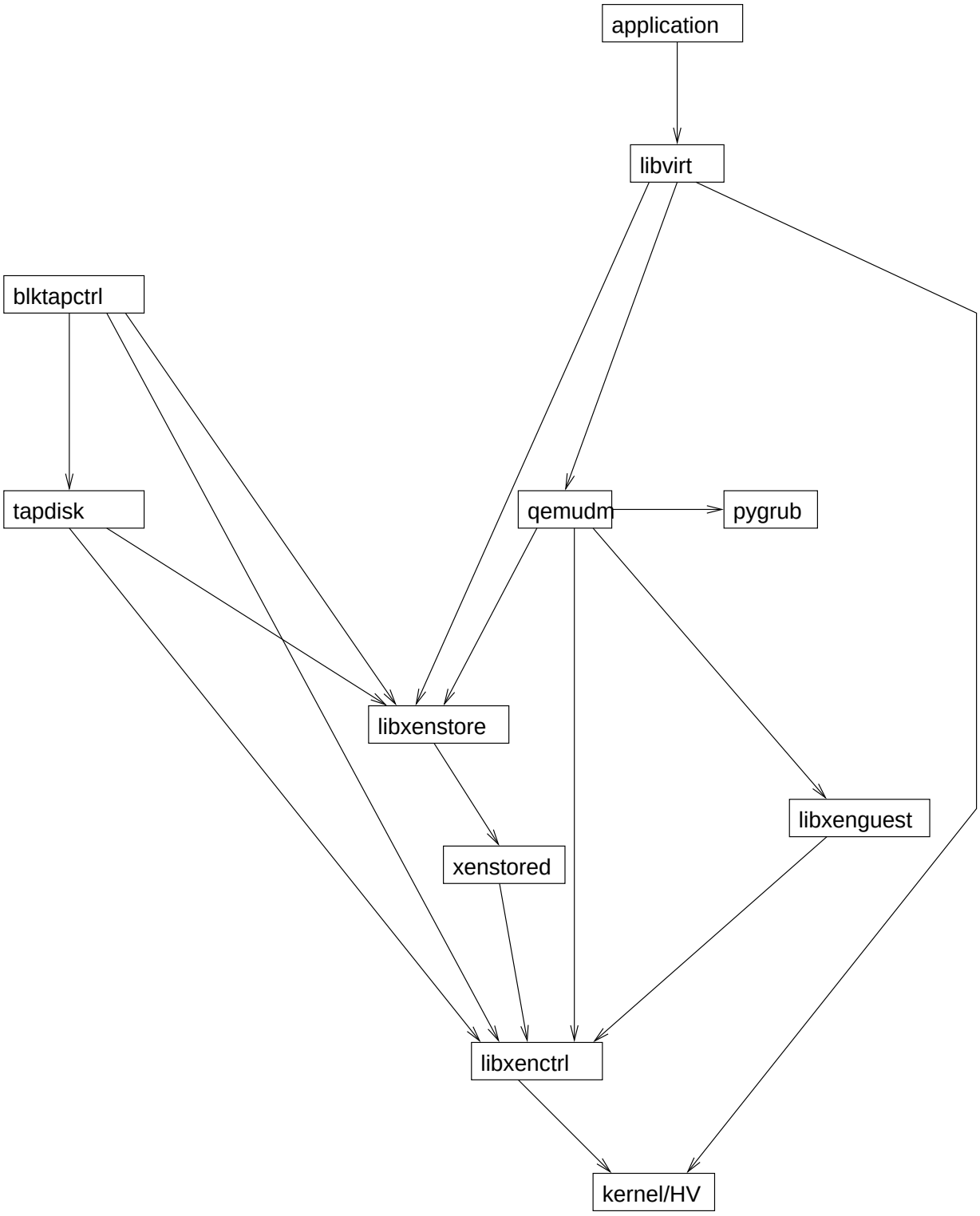

Figure 4: Xen userspace architecture with 3.2.0 derived Xenite

2.1.2 Xen i386 HVM boot process

The Xen HVM boot process is a little more complex than the upstream QEMU boot process. In upstream QEMU, the BIOS is loaded & and the emulator just starts running at 0x0 in real-mode. The BIOS takes over and just runs in more or less regular PC style. When a Xen HVM domain starts, execution begins at 0x100000 in protected-mode. The first thing which runs is the HVM firmware, which is loaded at 0x100000. The firmware does the setup things QEMU would normally care of, such as loading the VGA BIOS, the e820 map, the SMBIOS tables, the ACPI tables, VMXassist, etc. Most importantly it also sets up Xen hypercall pages for paravirt drivers to use. Once this is all done, it switches to real-mode, jumps to 0x0 and allows the regular PC boot process to run.

2.1.3 Making the two boot processes play nicely

From the descriptions of the two boot processes, two problems are pretty obvious. Both the HVM firmware, and the protected mode chunk of kernels want to live at 0x100000 in high memory. They can't both live there. Xen fullyvirtualized guest memory is not directly mapped into the QEMU process. There is a on-demand cache which which maps individual pages into QEMU as needed. This allows the guest memory size to exceed the size of the QEMU host process memory area (eg, allows a 64-bit HVM guest to run on a 64-bit hypervisor, and a 32-bit Domain-0 userspace, with more than 3GB of memory).

The second problem is fairly easy to address. Instead of directly memcpy()ing the kernel/initrd into the guest memory, QEMU provides an API cpu_physical_memory_rw to copy data into guest memory. This is abstracted in Xen to make use of the map cache. Thus the first stage of getting direct kernel boot to work is to pull in the latest upstream QEMU code from **hw/pc.c** and replace the memcpy() with calls to cpu_physical_memory_rw. There were also some I/O routines which fread() directly to guest memory which needed some wrappers.

Addressing the first problem is more challenging. If booting a recent kernel with support for relocating protected-mode, it is possible to simply load the kernel at 0x200000 instead. Then the **code32_start** field in the real-mode header is updated to point to this new location. Everything should 'just work' in this scenario. In tests is has been possible to successfully boot a fully-virtualized Fedora 8 32-bit installer kernel+initrd.

There are a huge number of distros with non-relocatable kernels in existance, which it would be very desirable to boot. Assuming that we don't wish to change the Xen hypervisor to support booting the firmware at an alternate address, we have to move the kernel elsewhere. Once the firmware has done its initial bootstrapping tasks, it is no longer needed in memory. So one idea is to move the kernel to 0x200000, then place a smaller helper routine at 0x500000, which essentially does a memove(0x100000, 0x200000, sizeof(kernel image)), and then jumps to 0x100000. The **code32_start** field is configured to point at the helper routine. So upon the switch from real-mode to protected-mode, the helper moves the non-relocatable kernel to its expected position & everything ought to be happy. Experimental code has attempted to do this, but does not really work particularly

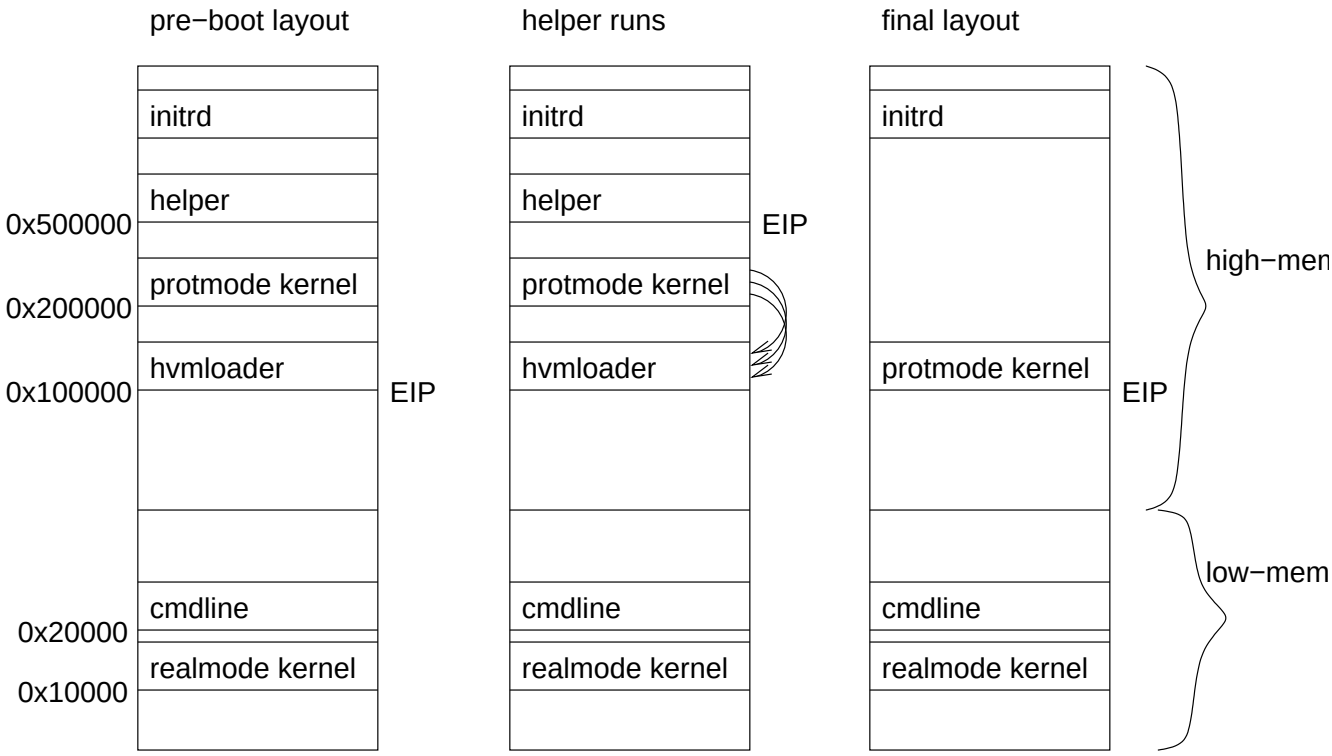

Figure 5: Memory layout when booting a non-relocatable kernel

well yet. Kernels will partially boot, but hang at some point. This suggests the general idea will probably work, but the helper routine is buggy. This ought to be solvable by someone with good knowledge of assembler.

Figure 5 illustrates the use of a helper to deal with booting non-relocatable kernels.

2.2 Booting guests without XenD

One of the very compelling advantages of KVM is that it is possible to create / destroy guest domains, simply by running/killing a single QEMU process. In the Xen userspace, XenD is responsible for creating guest domains, and then spawning QEMU to serve as the device model.

2.2.1 Stages in creating a guest VM

Analysis of the XenD code showed that the process for creating a guest consisted of the following series of stages

1. **Create the domain** This causes the hypervisor to allocate the basic domain data structures
2. **Free up memory in HV** Balloon down one or more other guest domains to free up memory, potentially including an allowance shadow memory

3. **Set max VCPU count** Tell hypervisor how many virtual CPUs to allocate to the guest

4. **Set max memory count** Tell hypervisor the maximum size of the e820 map

5. **Allocate event channels** Allocate event channels for guest to talk to xenstore and the parvirt console

6. **Write domain xenstore data** Write entries into xenstore for /local/domain/[ID] and /vm/[UUID] with basic domain metadata

7. **Load the kernel** Either load the kernel and initrd, or the HVM firmware

8. **Set HVM params** Setup a couple of HVM specific parameters

9. **Introduce to xenstore** Introduce the new domain to xenstore

10. **Write balloon xenstore data** Write data for the balloon driver into xenstore

11. **Write console xenstore data** Write data for the console event channel & mfn into /local/domain/[ID]/memory

12. **Write store xenstore data** Write data for the store event channel & mfn into /local/domain/[ID]/store

13. **Write VFB xenstore data** Write data for paravirt framebuffer into frontend and backend paths

14. **Write VBD xenstore data** Write data for virtual disks into frontend and backend paths

15. **Write VIF xenstore data** Write data for virtual NICs into frontend and backend paths

16. **Wait for hotplug** Wait for the hotplug scripts to complete setting up the backend devices

17. **Unpause the CPU** Start the guest execution by unpausing the CPU

In XenD, responsibility for these stages is split between XendDomainInfo.py, image.py, and the various server/*.py device type specific files.

2.2.2 Booting guests using QEMU

During development of Xen 3.2.0, the QEMU device model was extended to have two distinct machine types, 'xenpv' and 'xenfv'. One provides device emulation for paravirt guests and the other provides the same for fullvirt guests. The QEMU device model is launched by XenD passing in the domain ID to serve as one of the command line arguments. This provides an obvious approach to extend QEMU to support booting of guests directly, while retaining compatability with XenD. If a domain ID is provided via argv, then QEMU should assume it is attaching to an existing domain, otherwise it should create and configure a new domain.

2.3 Bootloaders for paravirtualized guest

Since paravirtualized guests do not use a BIOS, there is no scope for running a traditional bootloader inside the guest. Xen provides a psuedo-bootloader in the form of **pygrub** which runs in the host emulating the UI interface of a real **grub** bootloader. It extracts the kernel and initrd from the guest filesystem, and XenD subsquently uses these to do a direct kernel boot. In its current form, there is a key problem with pygrub - it is only visible via the text based serial console. This is only accessible if running as root on the host machine, and is not connected to the main graphical console.

For a consistent user interaction model between para and fullvirt domains, it is desirable to try and address these deficiencies. QEMU provides the ability to allocate virtual text consoles and attach them to the main graphical framebuffer. These are thus visible over the remote VNC connection to a guest. If QEMU were able to run the pygrub process directly, it could wire up its I/O into this main graphical console. So the paravirtualized guest would start off displaying the pygrub screen, and then switch to the main graphical console during boot. In a non-graphical environment, pygrub could be wired into the same output as the paravirt text console, eg as a psuedo-tty.

2.3.1 Integrating the bootloader with QEMU

The file **hw/xen_machine_pv.c** is responsible for booting up paravirtualized domains. It first creates the domain, then loads the kernel and initrd, writes all the front and backend device information into xenstore, and finally sets up the text and graphics console backend device implementations. The latter watch xenstore and attach to the guest domain at appropriate stage of the boot process. To support use of a bootloader, all of this must be delayed for a period of time.

A new file is introduced, **hw/xen_bootloader.c**, which contains a single API accepting a disk filename, and a callback. This forks a child process, and connects its STDIN/OUT/ERR to a puesdo TTY. The child process is also passed an additional file descriptor (the write end of a FIFO pipe) via which it will return details of the extracted kernel, initrd and cmdline. The parent (QEMU) attachs the master end of the psuedo TTY to a virtual QEMU console allocated with the **text_console_init** method, and sets

up a filedescriptor watch on the read-end od the FIFO pipe.

At this point the bootloader is running and waiting for user interaction. The user can connect via VNC, (or using the local SDL display) and interact with the bootloader to choose a suitable kernel. When chosen, the kernel and initrd are written out to temporary files, and their locations returned to QEMU via the FIFO pipe. At this point, the file descriptor watch QEMU has on the FIFO pipe will fire & it can read in the details of the kernel/initrd. The callback mentioned earlier is then invoked, and **hw/xen_machine_pv.c** can now carry on with its usual process for loading a kernel & initrd. The QEMU graphics console is automatically switched from the pygrub screen over to the graphical guest framebuffer during boot. The results is a reasonably convincing simulation of a guest bootloader.

2.4 Using Xenite from libvirt

3 Conclusion

This section provides a summary of the important issues discussed throughout the document. Together with a selection of ideas for followup work to improve the ability to remotely manage virtual machines.