Proceedings of the Linux Symposium

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

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

KvmFS: Virtual Machine Partitioning For Clusters and Grids

Andrey Mirtchovski
Los Alamos National Laboratory
andrey@lanl.gov

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Abstract

This paper describes KvmFS, a synthetic file system that can be used to control one or more KVM virtual machines running on a computer. KvmFS is designed to provide its functionality via an interface that can be exported to other machines for remote configuration and control. The goal of KvmFS is to allow a multi-CPU, multi-core computer to be partitioned externally in a fashion similar to today's computational nodes on a cluster. KvmFS is implemented as a file server using the 9P protocol and its main daemon can be mounted locally via the v9fs kernel module. Communication with the KvmFS occurs through standard TCP sockets. Virtual machines are controlled via commands written to KvmFS' files. Status information about KVM virtual machines is obtained by reading KvmFS. KvmFS allows us to build clusters in which more than one application can share the same SMP/Multi-core node with minimalistic full system images tailored specifically for the application.

1 Introduction

The tendency in high-performance computing is towards building processors with many computational units, or cores, with the goal of parallelizing computation so that many units are performing work at the same time. Dual and quad-core processors are already on the market, and manufacturers are hinting at 8, 32, or even 80 cores for a single CPU, with a single computational node composed of two, four, or more CPUs. This will result in applications running and contending for resources on large symmetric multiprocessor systems (SMPs) composed of hundreds of computational units.

There is a problem with this configuration, however: since clusters are currently the dominant form of node organization in the HPC world (as seen in the latest breakdown by machine type on the Top 500 list of supercomputers), most applications are designed to either run on a single 2- or 4-core machine, or so that separate parts of the program will run on separate 2- or 4-core nodes, and will communicate via some message-passing framework such as MPI. This has resulted in most of the applications running here, at Los Alamos National Laboratory, scaling to at most 8 CPUs on a single node. Furthermore, many applications assume that they are the only ones running on a single node and will not have to contend for resources. To satisfy the requirements of such applications, large SMP computers will have to be partitioned so that applications are ensured dedicated resources without contention. Fail-over and resilience, two very hot topics in High Performance Computing, also require the means to transfer an application from one machine to another in the case of hardware or software component failures on the original computer.

One solution for partitioning hardware and providing resilience has gained widespread adoption and is considered feasible for the HPC world: virtualization using hypervisors. Borrowing from the mainframe, it allows separate instances of an operating system (or indeed separate operating systems) to be run on the same hardware or parts thereof. The two major CPU manufacturers have added support for virtualization to their newest offerings, which provides even greater performance gains than previously thought.

KVM has recently emerged as a fast and reliable (with the hardware support on modern processors) subsystem for virtualizing the hardware on a computer. Our goal with KvmFS is to enable KVM to be remotely controlled by either system operators or schedulers and to allow it to be used for partitioning on clusters composed of large SMP machines, such as the ones already being proposed here at LANL.

Virtualization benefits the system administrator, as well as programmers and scientists running high-performance code on large clusters. One benefit is the full control over the operating system installation that an application requires. For example, it is not necessary to have all support libraries and software installed on all machines of a cluster, instead, the application is run in an OS instance that already contains all that is required. This greatly simplifies installations in the case where conflicting libraries and support software may be required by different applications. Another possibility is to run a completely different operating system under virtualization, something impossible in current monolithic cluster environments.

With the fast and reliable means of running applications on their own slice of a SMP, it is convenient to be able to extend the control of partitioning and virtualization across the cluster to a control or a head node. This is the niche that KvmFS fills: it provides the fast and secure means to control VMs across a cluster, or indeed a grid environment.

1.1 KVM

KVM [14] is a hypervisor support module in the Linux kernel which utilizes hardware-assisted x86 virtualization on modern Intel processors with Intel Virtualization technology or AMD's Secure Virtual Machine. By adding virtualization capabilities to a standard Linux kernel, KVM provides the benefits of the optimizations that exist in a standard kernel to virtualized programs, greatly increasing performance over "full hypervisors" such as Xen [1] or VMWare [9]. Under the KVM model, every virtual machine is a regular Linux process scheduled by the standard Linux scheduler. Its memory is allocated by the Linux memory allocator.

KVM works in conjunction with QEMU to deliver the processor's virtualization capabilities to the end user.

1.2 QEMU

QEMU [2] is a machine emulator which can run an unmodified target operating system (such as Windows or Linux) and all its applications in a virtual machine. QEMU runs on several host operating systems such as Linux, Windows, and Mac OSX.

The primary usage of QEMU is to run one operating system on another, such as Windows on Linux or Linux on Windows. Another usage is debugging, because the virtual machine can be easily stopped, and its state can be inspected, saved, and restored. Moreover, specific embedded devices can be simulated by adding new machine descriptions and new emulated devices.

Although the host and target operating systems can be different, our software will focus on Linux as the host system since Linux is the primary OS on all of our recent clusters at LANL and is widely adopted for HPC environments. Also, KVM currently exists only for the Linux kernel.

2 Design

KvmFS was created allow its users to run and control virtual machines in a heterogeneous networked environment. As such, KvmFS was designed to fulfill the following tasks:

functionality provide an interface that allows management of VMs on a cluster

scalability provide the ability for fast creation of multiple identical VMs on different nodes connected via a network

checkpoint and restart provide the ability to suspend virtual machines and resume their execution, potentially on a different node

The design of KvmFS follows the well established model of providing functionality in the form of synthetic file systems which clients operate on using standard I/O commands such as `read` and `write`. This method has proven successful in various operating systems descendant from UNIX. The `/proc` [7] file system is a very well established example. The "Plan 9" operating system further extends this concept. It presents the network communication subsystem as mountable files [13] or even the graphics subsystem and the window manager written on top of it, as a file system.

Implementations such as the above suggest that the concept is feasible and that implementing interfaces to resources in the form of a file system and exporting them to other machines is a very good way to quickly allow access to them from remote machines, especially since files are the single most exported resource in a networked environment such as a cluster.

KvmFS is structured as a two-tiered file server to which clients connect either from the local machine or across the network. The file server allows them to copy image files and boot virtual machines using those image files. The file server also allows controlling running virtual machines (start, stop, freeze), as well as migrating them from one computer to another.

The top-level directory KvmFS serves contains two files providing information about the architecture of the machine as well as starting a new session for a new VM. Each session already started is presented as a numbered subdirectory. The subdirectory itself presents files which can be used to control the execution of the VM, as well as a subdirectory which allows arbitrary image files to be copied to it and used by the VM. The KvmFS filesystem is presented in detail in section 3.

3 The KVM File System

KvmFS presents a synthetic file system to its clients. The file system can be used for starting and controlling all aspects of the runtime of the virtual machines running on the machine on which kvmfs is running.

```
clone
arch
vm#/
       ctl
       info
       id
       fs/
```

3.1 Top-level files

`Arch` is a read-only file; reading from it returns the architecture of the compute node in a format *operating-system/processor-type.*

`Clone` is a read-only file. When it is opened, KvmFS creates a new session and the corresponding session directory in the filesystem.

Reading from the file returns the name of the session directory.

`Vm#` is a directory corresponding to a session created by a KvmFS client. Even though a session may not be running, `Vm#` will exist as long as that client keeps the `clone` file open. If the virtual machine corresponging to a session is running the `clone` file may be closed without causing the `Vm#` file to disappear.

3.2 Session-level files

These files are contained in the session directory which is created when a client opens the `clone` file of a KvmFS server.

`Ctl` is used to execute and control a session's main process. Reading from the file returns the main process pid if the process is running, and –1 otherwise. The operations on the session are performed by writing to it.

Reading from `info` returns the current memory and device configuration of the virtual machine. The format of the information is identical to the commands written to `ctl` file.

`Id` is used to set and get the user-specified VM identifier.

The `fs` directory points to the temporary storage created for the virtual machine. The user can copy disk images and saved VM state files that can be used in the VM configuration.

4 KvmFS Commands

The following section describes the set of commands available for controlling KvmFS instances:

dev *name image* Specifies the device image for a specific device. *Name* is one of *hda, hdb, hdc, hdd.* If *image* is not an absolute path, it should point to a file that is copied in the `fs` directory. An optional *boot* parameter can be provided to specify that the device should be used to boot from.

net *id mac* Creates a network device with ID *id* and MAC *mac.*

loadvm *file* Loads a saved VM state from file *file.* If *file* is not an absolute path, it should point to a file in the `fs` directory.

storevm *file* Stores the state of the VM to file *file.* If *file* is not an absolute path, the file is created in the `fs` directory.

power *on|off* Turns VM power on or off.

freeze Suspends the execution of the VM.

unfreeze Resumes the execution of the VM.

clone *max-vms address-list* Creates copies of the VM on the nodes specified by *address-list*. Copies the content of the `fs` directory to the remote VMs and configures the same device configuration. If the virtual machine is already running, stores the current VM state (as in *storevm*) and loads it in the remote VMs. If `max-vms` is greater than zero, and the number of the specified sessions is bigger than `max-vms`, `clone` pushes its content to up to max-sessions and issues `clone` commands to some of them to clone themselves to the remaining VMs from the list.

The format of the address-list is:

```
address-list = \
    1*(vm-address `,'')
vm-address = node-name \
    [`!''port]
  `/'' vm-id
node-name = ANY
port = NUMBER
vm-id = ANY
```

5 Implementation

There are two ways of implementing accesses to programs or system resources as files in Linux, either using Fuse [3] or the 9P [8] protocol. We chose the 9P protocol because it is better suited for communicating with file systems over networks. 9P has also been in use for the past twenty years and is sufficiently hardened to be able to handle various workloads on environments ranging from a single machine to thousands of cluster nodes [5]. Furthermore, our team is well familiarized with 9P through the implementation of V9FS, the kernel module allowing 9P servers to be mounted on a Linux filesystem [10] [4]. It is important to point out, however, that there is no significant barrier to implementing KvmFS using FUSE.

5.1 9P

Representing operating system resources as files is a relatively old concept exploited to some extent in the original UNIX operating system, but it matured extensively with the development and release of the "Plan 9 from Bell-Labs" operating system [12].

"Plan 9 from Bell-Labs" uses a simple, yet very powerful communication protocol to facilitate communication between different parts of the system. The protocol, named "9P" [8], allows heterogeneous resource sharing by allowing servers to build a hierarchy of files corresponding to real or virtual system resources, which then clients access via common (POSIX-like) file operations by sending and receiving 9P messages. The different types of 9P messages are described in Table 1.

There are several benefits of using the 9P protocol:

Simplicity The protocol has only a handful of messages which encompass all major file operations, yet it can be implemented (including the co-routine code explained above) in around 2,000 lines of C code.

Robustness 9P has been in use in the Plan 9 operating system for over 15 years.

Architecture independence 9P has been ported to and used on all major computer architectures.

Scalability Our Xcpu [11] suite uses 9P to control and execute programs on thousands of nodes at the same time.

A 9P *session* between a server and its clients consists of requests by the clients to navigate the server's file and directory hierarchy and responses from the server to those requests. The client initiates a request by issuing a *T-message*, the server responds with an *R-messages*. A 9P *transaction* is the combined act of transmitting a request of particular type by the client and receiving a reply from the server. There may be more than one request outstanding; however, each request requires a response to complete a transaction. There is no limit on the number of transactions in progress for a single session.

Each 9P message contains a sequence of bytes representing the size of the message, the type, the tag (transaction id), control fields depending on the message type, and a UTF-8 encoded payload. Most T-messages contain a 32-bit unsigned integer called *Fid*, used by the client to identify the "current file" on the server, i.e., the last file accessed by the client. Each file in the file system served by our library has an associated element called *Qid* used to uniquely identify it in the file system.

5.2 KvmFS

KvmFS is implemented in C using the SPFS and Spclient [6] libraries for writing 9P2000-compliant user-space file servers and accessing them over a network.

9P type	Description
version	identifies the version of the protocol and indicates the maximum message size the system is prepared to handle
auth	exchanges auth messages to establish an authentication fid used by the attach message
error	indicates that a request (T-message) failed and specifies the reason for the failure
flush	aborts all outstanding requests
attach	initiates a connection to the server
walk	causes the server to change the current file associated with a fid
open	opens a file
create	creates a new file
read	reads from a file
write	writes to a file
clunk	frees a fid that is no longer needed
remove	deletes a file
stat	retrieves information about a file
wstat	modifies information about the file

Table 1: Message types in the 9P protocol

It is a single-threaded code which uses standard networking via the `socket()` routines. Although our implementation is in C, both 9P2000 and KvmFS are language-agnostic and can be reimplemented in any other programming language that has access to networking.

OS Image files used by virtual machines can grow to be quite large (sometimes up to the size of a complete system installation: several gigabytes) and can take a long time to be transferred to a remote node. To start a single VM on all the nodes of a cluster can potentially take upwards of an hour for large clusters, with literally a hundred percent of the time being spent transferring the disk images of the VM either from a head node or from a networked file system such as NFS. To alleviate this problem we can employ tree-based spawning of virtual machines via cloning. During tree-spawning, if an end node has received the complete image (or in some cases a partial image), that node can retransmit the image to another node, potentially located only a hop away on the network. To allow tree-spawns each KvmFS server can also serve as a client to another server by implementing routines which connect over 9P, create new sessions, set-up and start a new VM with the image from the local session. This reduces logarithmically the amount of fetches that need to occur from the head node and significantly increases the scale at which KvmFS can be deployed. We have tested tree-spawn algorithms for small images on several thousand nodes on LANL's clusters.

The total number of lines for KvmFS, not including the SPFS libraries, is less than two thousand lines of code. SPFS itself is 5,158 lines of code, and Spclient is another 2,381 lines of code.

6 Sample Sessions

Several examples of using KvmFS follow. The examples show systems mounted remotely using the v9fs [4] kernel module and consequently being accessed via common shell commands. In the examples below, the names `n1`, `n2`, etc., are names of nodes on our cluster.

6.1 Create a virtual machine

This example creates a virtual machine using two files copied from the home directory. `Disk.img` is set to correspond to hard drive `hda` and `vmstate` is used as a previously saved virtual machine.

```
mount -t 9p n1 /mnt/9
cd /mnt/9
tail -f clone &
cd 0
cp ~/disk.img fs/disk.img
cp ~/vmstate fs/vmstate
echo dev hda disk.img > ctl
echo net 0 00:11:22:33:44:55 > ctl
echo power on freeze > ctl
echo loadvm vmstate > ctl
echo unfreeze > ctl
```

6.2 Migrate a virtual machine to another node

This example shows the migration of a virtual machine from one node to another.

```
mount -t 9p n1 /mnt/9/1
mount -t 9p n2 /mnt/9/2
tail -f /mnt/9/2/clone &
cd /mnt/9/1/0
echo freeze > ctl
echo 'clone 0 n2!7777/0' > ctl
echo power off > ctl
```

6.3 Create clones of a virtual machine

This example shows the cloning of a virtual machine onto a new computer.

```
mount -t 9p n1 /mnt/9
cd /mnt/9/0
echo 'clone 2 n2!7777/0,\
  n3!7777/0,\
  n4!7777/0' > ctl
```

7 Conclusions And Future Work

We have described the KvmFS file system which presents an interface to virtual machines running on Linux in the form of files accessible locally or remotely. KvmFS allows us to extend the control of the partitioning and running of virtual machines on a computer beyond the system on which the virtual machines are running and onto a networked environment such as a cluster or a computational grid. KvmFS benefits large cluster environments such as the ones in use here, at the Los Alamos National Laboratory, by enabling fine-grained control over the software running on them from a centralized location. Status information regarding the parameters on currently running VMs can also easily be obtained from computers other than the ones they are executing on. Our system also allows checkpointing and migration of VMs to be controlled from a centralized source, thus enabling partitioning schedulers to be built on top of KvmFS.

Future work we have planned for KvmFS is in the area of fine-grained control of the execution parameters of virtual machines running under KvmFS such as their CPU affinity. Also, we plan to integrate KvmFS with existing schedulers at LANL to provide a seamless way of partitioning our clusters.

Another interesting issue we are exploring is exporting the resources of running virtual machines, such as their `/proc` filesystem, through the KvmFS interface so that processes running under the VM can be controlled externally or even over a network.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, and R. Neugebauer. Xen and the art of virtualization. 2004.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[3] FUSE. Filesystems in userspace. `http://fuse.sourceforge.net/`.

[4] Eric Van Hensbergen and Latchesar Ionkov. The v9fs project. `http://v9fs.sourceforge.net`.

[5] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9p2000 under linux. In *Freenix Annual Conference*, pages 83–94, 2005.

[6] L. Ionkov. Library for writing 9p2000 compliant user-space file servers. `http://sourceforge.net/projects/npfs/`.

[7] T.J. Killian. Processes as files. *USENIX Summer 1984 Conf. Proc.*, 1984.

[8] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[9] R. Meushaw and D. Simard. Nettop: Commercial technology in high-assurance applications. `http://www.vmware.com`, 2000.

[10] R. Minnich. V9fs: A private name space system for unix and its uses for distributed and cluster computing.

[11] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[13] D. Presotto and P. Winterbottom. The organization of networks in plan 9. *USENIX Winter 1993 Conf. Proc.*, pages 43–50, 1993.

[14] Qumranet. Kvm: Kernel-based virtualization driver. `http://kvm.qumranet.com/kvmwiki/Documents`.