Kernel Virtual Machine (KVM)

Best practices for KVM

IBM

Kernel Virtual Machine (KVM)

Best practices for KVM

Note

Before using this information and the product it supports, read the information in “Notices” on page 39.

Second Edition (April 2012)

Contents

Best practices for KVM 1
What's new in KVM best practices 1
Best practices for device virtualization for guest operating systems 1
Best practice: Para-virtualize devices by using the VirtIO API 1
Best practice: Optimize performance by using the `virtio_blk` and `virtio_net` drivers 3
Best practice: Virtualize memory resources by using the `virtio_balloon` driver 5
Best practices for VM storage devices 5
Best practice: Use block devices for VM storage . 6
Best practice: If you cannot use block devices for VM storage, you must use disk image files . . . 6
Disk image files 6
Sparse files 7
Partitions for disk image files. 8
Creating a disk image file 10
Best practices for over-committing processor and memory resources 10
Best practice: Target system use at 80% or lower 10
Best practice: Allocate the minimum amount of processor resources 11
Best practice: Over-commit memory resources by using page sharing or ballooning 11
Page sharing 11
Kernel same-page merging (KSM) 11
Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 6 and later . 13
Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 5.4, 5.5, and 5.6 14
Determining the memory savings of KSM with Red Hat Enterprise Linux 6 and later . 14
Determining the memory savings of KSM with Red Hat Enterprise Linux 5.4, 5.5, and 5.6 16
Controlling KSM usage on a per-guest basis with Red Hat Enterprise Linux 6.1 and later . 16
Ballooning. 17
Swapping 19
Best practices for networking 19
Best practice: Use the **`tap`** networking option in QEMU 19
Best practice: Configure para-virtualized network devices 21
Adding guest para-virtualized network devices with libvirt 21
Using a host bridged network device with libvirt 22
Best practice: Use vhost-net for high-bandwidth applications in Red Hat Enterprise Linux 6.1 and later 22
Disabling vhost-net in-kernel support using libvirt 22
Best practice: Use PCI pass-through to enhance network performance 23
PCI pass-through 23
Setting up PCI pass-through. 23
Adding pass-through devices to guests with libvirt 23
Best practices for block I/O performance 24
General best practices for block I/O performance 24
Best practice: KVM guest caching modes . . . 25
Best practice: I/O schedulers 26
Best practice: Optimize block I/O performance by using the Deadline I/O scheduler 28
Best practice: If you cannot use the Deadline I/O scheduler, configure the CFQ I/O scheduler . . 31
Best practices: Asynchronous I/O model for KVM guests 33
Best practices for improving security key strength 34
General principles for improving security key strength 34
Best practice: Assess quantity of kernel entropy available just before generating keys 34
Best practice: Ensure that keys or entropy sources are not cloned 35
Best practice: Prevent or limit network access . . 35
Best practice: Seed the kernel entropy pool with node-specific data 35
Best practice: Increase the kernel entropy pool by adding device event timing uncertainty 36
Best practice: Generate keys just before they are required 36

Appendix. Code license and disclaimer information 37

Notices 39
Trademarks 40

Best practices for KVM

Learn about the best practices for Kernel-based Virtual Machine (KVM), including device virtualization for guest operating systems, local storage devices of virtual machines, over-committing processor and memory resources, networking, and block I/O performance.

What's new in KVM best practices

KVM best practices has been updated to include new topics.

New best practices for improving security key strength

Several topics containing best practices for improving security key strength based on your product type have been added to the Best practices for KVM topic collection. Learn more...

Best practices for device virtualization for guest operating systems

Learn about the performance gains of para-virtualized devices and about the drivers of the VirtIO API.

Best practice: Para-virtualize devices by using the VirtIO API

KVM can provide emulated or para-virtualized devices to the guest operating systems. Compared to emulated devices, para-virtualized devices provide lower latency and higher throughput for I/O operations of guest operating systems. KVM includes the VirtIO API to para-virtualize devices.

With all virtualization technologies, the hypervisor must provide the guest operating systems with devices that the guest operating systems require to successfully run. In the simplest case, these devices provide storage and network access. In more complicated cases, additional devices provide new functions that use the virtualization base on which the guest operating systems run.

Typically, the hypervisor provides emulated or para-virtualized devices to the guest operating systems.

Emulated devices

Emulated devices are the most common virtualized devices. Emulated devices are a core component of full virtualization solutions such as VMware.

When a hypervisor provides a guest operating system with an emulated device, the hypervisor creates a software implementation of the hardware. In KVM, a modified version of QEMU in user space provides device emulation. The hypervisor intercepts all I/O requests from the guest operating system and emulates the operation of the real I/O hardware. The guest operating system that uses the device interacts with the device as if it were actual hardware rather than software.

The hardware device that is emulated is usually an older, generic device that supports various drivers across various operating systems. This broad support is the core strength of device emulation. It provides the ability for guest operating systems to run and use emulated devices with no special drivers and with no modification to the operating system.

While emulated solutions provide a broader compatibility level than para-virtualized solutions, the performance of emulated solutions is lower than para-virtualized solutions.

Red Hat Enterprise Linux provides several emulated devices that support the following functions:

- Block I/O: emulated devices include IDE and Floppy. SCSI and USB also are valid for Red Hat Enterprise Linux 5.5 only.
- Networking: emulated devices include e1000 and rtl8139. ne2k_pci and pcnetare also are valid for Red Hat Enterprise Linux 5.5 only.
- Graphics: for Red Hat Enterprise Linux 6, emulated devices include cirrus and vga.
- Mouse input
- Serial port
- Sound cards: for Red Hat Enterprise Linux 6, emulated devices include intel-hda, ac97, and es1370. intel-hda is recommended.

Emulated devices that support graphics, mouse input, serial port, and sound cards are meant for the enablement of guest operating systems. These devices are not currently targeted for para-virtualized performance improvements.

Para-virtualized devices

Para-virtualized devices are software implementations of hardware devices. You can install a para-virtualized driver in the guest operating system. With para-virtualized devices, the hypervisor and the guest operating system use an optimized I/O interface to communicate as quickly and efficiently as possible.

Unlike emulation, para-virtualization requires that the guest operating system be modified to communicate with the hypervisor. In some solutions, such as Xen, the entire guest operating system is para-virtualized for an efficient, cooperative relationship with the hypervisor. In other solutions, such as VMware and KVM, only the device drivers are para-virtualized.

While para-virtualized solutions typically outperform emulated solutions, the compatibility level of para-virtualized solutions is lower than emulated solutions. The reason that para-virtualized solutions typically outperform emulated solutions is because emulated devices must adhere to the same requirements as the real hardware they emulate. This adherence is critical because if the emulated device does not behave like the hardware, the device drivers in the guest operating system might not work as intended. However, this adherence is not necessarily efficient when two pieces of software, the hypervisor and guest operating system, must communicate. Para-virtualized devices are not burdened with this adherence requirement. Instead, para-virtualized solutions use an API that provides high performance for I/O operations of guest operating systems.

In addition to latency and bandwidth gains, para-virtualized devices require fewer processor resources than emulated devices. Sometimes emulation code cannot be implemented efficiently. One possible reason is the lack of specialized hardware assistance. This inefficient code causes high processor load when a device is heavily used. Lower processor use is often an additional benefit of para-virtualized devices because of the optimized nature of para-virtualized devices.

VirtIO API

The VirtIO API is a high performance API that para-virtualized devices use to gain speed and efficiency. The VirtIO API specifies an interface between virtual machines and hypervisors that is independent of the hypervisor. In typical situations, VirtIO para-virtualized devices provide lower latency and higher throughput for I/O operations of guest operating systems. VirtIO para-virtualized devices are especially useful for guest operating systems that run I/O heavy tasks and applications.

Tip: Always use devices that implement the VirtIO API if the devices are available and supported for the guest operating system.

For information about the compatibility of guest operating systems and VirtIO devices on Red Hat Enterprise Linux , see KVM Para-virtualized Drivers in the *Red Hat Enterprise Linux 6 Virtualization Host Configuration and Guest Installation Guide*.

Related concepts:

"Best practice: Optimize performance by using the virtio_blk and virtio_net drivers"
The virtio_blk driver uses the VirtIO API to provide high performance for storage I/O devices, especially in large enterprise storage systems. The virtio_net driver uses the VirtIO API to provide increased network performance.

"Best practice: Virtualize memory resources by using the virtio_balloon driver" on page 5
The virtio_balloon driver provides a communication pathway between the hypervisor and the guest operating system. The hypervisor uses this communication pathway to over-commit memory resources.

Related tasks:

"Best practice: Configure para-virtualized network devices" on page 21
You can configure para-virtualized network devices for the guest operating systems.

Best practice: Optimize performance by using the virtio_blk and virtio_net drivers

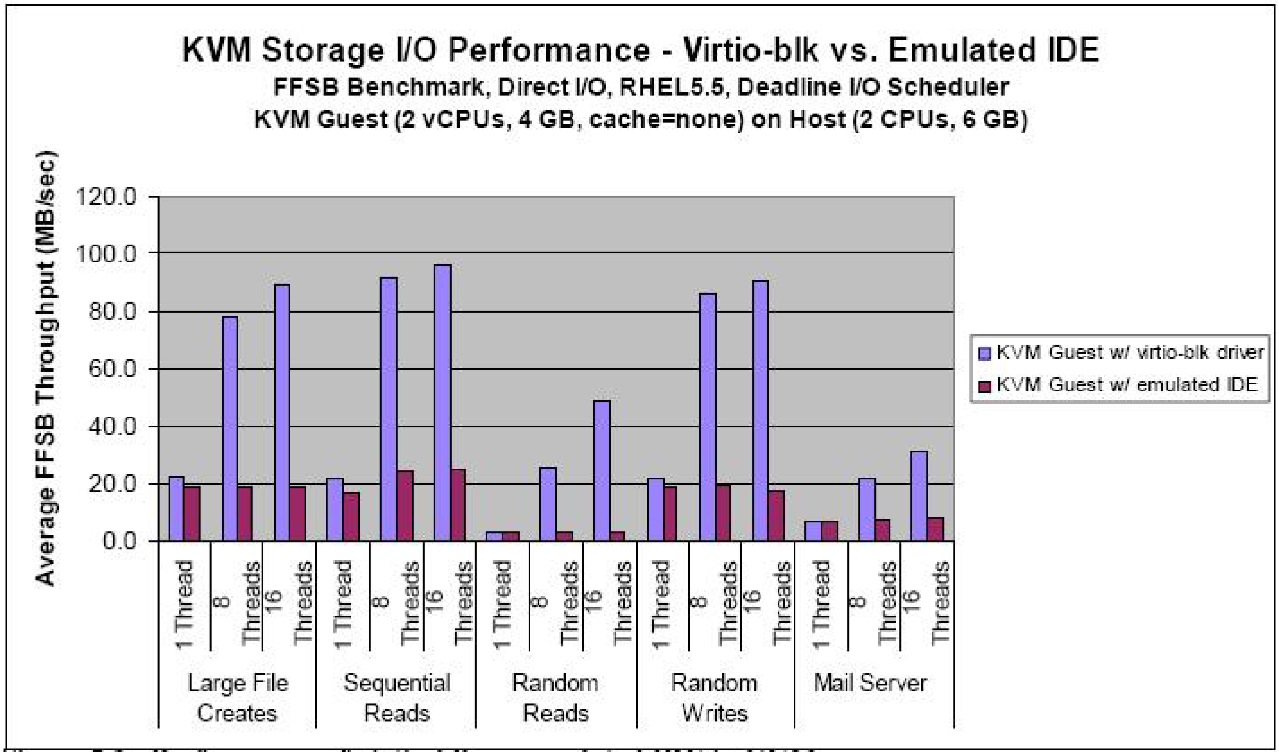
The virtio_blk driver uses the VirtIO API to provide high performance for storage I/O devices, especially in large enterprise storage systems. The virtio_net driver uses the VirtIO API to provide increased network performance.

Red Hat Enterprise Linux 5.5 provides the virtio_blk and virtio_net VirtIO performance drivers. Red Hat Enterprise Linux 5.5 also provides additional drivers, such as virtio, virtio_pci, and virtio_ring, which provide the base support for the VirtIO API.

The virtio_blk driver

The virtio_blk driver supports access to para-virtualized block devices for increased storage performance.

In a set of lab tests, the Flexible File System Benchmark (FFSB) program ran in guest operating systems and generated various I/O workloads on a large storage system. The large storage system included four IBM® DS3400 controllers that were fiber-attached, eight RAID10 disk arrays, and 192 disks. The KVM environment included a guest operating system that used two virtual processors and 4 GB of memory. The guest operating system did not use any disk I/O cache. The host used two physical processors and 2 GB of memory. The following figure shows the FFSB throughput data for direct I/O with the Deadline I/O scheduler:



The figure shows the average FFSB throughput in megabytes per second for the guest operating system run with a virtio_blk driver and the guest operating system run with an emulated IDE device. For both the virtio_blk driver and the emulated IDE device, the figure shows throughput data for the following types of operations:

- Large file creation
- Sequential reads
- Random reads
- Random writes
- Mail server, which includes random file operations such as creating files, opening files, deleting files, random reads, random writes, and other random file operations. These random file operations were performed on 100,000 files in 100 directories with file sizes ranging from 1 KB to 1 MB.

For each operation, the figure shows results for one thread, eight threads, and 16 threads.

For example, the figure shows the following average FFSB throughput for random writes on eight threads:

- Guest operating system with the virtio_blk driver: 85.8 MB/second
- Guest operating system with an emulated IDE device: 19.0 MB/second

Overall, the figure shows that devices that you para-virtualize with the virtio_blk driver outperform emulated devices, especially in multi-threaded scenarios.

The virtio_net driver

The virtio_net driver supports access to para-virtualized network devices for increased network performance.

Quantifying the performance gains of the `virtio_net` driver is difficult because of the various different usage patterns for networking. However, in general, the driver provides latency and bandwidth performance gains.

Related concepts:

"Best practice: Para-virtualize devices by using the VirtIO API" on page 1
KVM can provide emulated or para-virtualized devices to the guest operating systems. Compared to emulated devices, para-virtualized devices provide lower latency and higher throughput for I/O operations of guest operating systems. KVM includes the VirtIO API to para-virtualize devices.

"Best practices for block I/O performance" on page 24
Learn about general best practices and I/O scheduler options for optimizing block I/O performance.

Best practice: Virtualize memory resources by using the `virtio_balloon` driver

The `virtio_balloon` driver provides a communication pathway between the hypervisor and the guest operating system. The hypervisor uses this communication pathway to over-commit memory resources.

Red Hat Enterprise Linux provides the `virtio_balloon` driver of the VirtIO API. Red Hat Enterprise Linux also provides additional drivers, such as `virtio`, `virtio_pci`, and `virtio_ring`, which provide the base support for the VirtIO API.

Instead of targeting performance gains, the `virtio_balloon` driver provides a new function in a virtualized environment, which is memory over-commitment. The `virtio_balloon` driver opens a cooperative communication pathway between the hypervisor and the guest operating system. Over this pathway, the hypervisor sends a request to the virtio_balloon driver to return some of the memory of the guest operating system back to the hypervisor. The virtio_balloon driver allocates memory from the guest operating system to satisfy the request and returns the memory to the hypervisor. The hypervisor then allocates the memory elsewhere as needed, over-committing the memory. If you do not load the `virtio_balloon` driver in the guest operating system, the following results occur:

1. The guest operating system ignores memory requests from the hypervisor.
2. The hypervisor cannot over-commit the memory.

Related concepts:

"Best practice: Para-virtualize devices by using the VirtIO API" on page 1
KVM can provide emulated or para-virtualized devices to the guest operating systems. Compared to emulated devices, para-virtualized devices provide lower latency and higher throughput for I/O operations of guest operating systems. KVM includes the VirtIO API to para-virtualize devices.

"Ballooning" on page 17
You can use ballooning to over-commit the memory resources of a guest operating system. However, ballooning has some disadvantages that limit the usefulness of ballooning outside of specialized scenarios.

Best practices for VM storage devices

With KVM, you can use block devices or files as local storage devices within guest operating systems.

Files are commonly known as *disk image files* for the following reasons:

- Disk image files are files that are available to the hypervisor.
- Like block devices, the disk image files represent a local mass storage disk device to the guest operating systems.

Block devices perform better than disk image files. Unlike block devices, disk image files provide other advantages in the areas of system management and storage capacity use. In most scenarios where disk performance from the guest operating system is not critical, most users prefer to use disk image files.

Regardless of the storage device you use, the storage must be accessible by the hypervisor before the following actions can occur:

- The **qemu-kvm** emulator can specify the storage device.
- The guest operating systems can use the storage device.

With the **qemu-kvm** emulator, you can specify up to 32 disk options for each guest operation system. These disk options can be any combination of block devices and disk image files.

The disk device options that are passed by the **qemu-kvm** emulator represent local mass storage devices to the guest operating system. Therefore, the guest operating system uses and manages the disk devices as typical storage devices. You can use standard Linux tools to manage and manipulate the devices. Some standard Linux tools include:

- Partition managers
- Logical Volume Managers (LVM)
- Multiple Device (MD) drivers
- File system formatting

Storage devices within the guest operating systems are typically formatted with a file system. However, some applications can perform I/O operations to an unformatted or raw disk device.

Best practice: Use block devices for VM storage

A guest operating system that uses block devices for local mass storage typically performs better than a guest operating system that uses disk image files. The guest operating system that uses block devices achieves lower-latency and higher throughput.

A guest operating system that uses block devices performs better because of the number of software layers through which it passes. When an I/O request is targeted to the local storage of a guest operating system, the I/O request must pass through the file system and I/O subsystem of the guest operating system. Then, **qemu-kvm** moves the I/O request from the guest operating system and the hypervisor handles the I/O request. Next, the hypervisor completes the I/O request in the same way the hypervisor completes an I/O request for other processes running within the Linux operating system.

Consider the following requirements when choosing to use block devices:

- All block devices must be available and accessible to the hypervisor. The guest operating system cannot access devices that are not available from the hypervisor.
- You must activate or enable some block devices before you can use the block devices. For example, Logical Volume Manager (LVM) logical volumes and Multiple Device (MD) arrays must be running in order to use the exported devices.

KVM supports various block storage devices that **qemu-kvm** can use for the guest operating systems. Supported block devices include hard disks, hard disk partitions, LVM logical volumes, MD arrays, USB, and other devices.

Best practice: If you cannot use block devices for VM storage, you must use disk image files

Learn about disk image files, sparse image files, partitions for disk image files, and how to create a disk image file.

Disk image files

Learn about disk image files, including the benefits and drawbacks of using disk image files.

A *disk image file* is a file that represents a local hard disk to the guest operating system. This representation is a *virtual hard disk*. The size of the disk image file determines the maximum size of the virtual hard disk. A disk image file of 100 GB can produce a virtual hard disk of 100 GB.

The disk image file is in a location outside of the domain of the guest operating system and outside the domain of the virtual machine (VM). Other than the size of the disk image file, the guest operating system cannot access any other information about the disk image file. The disk image file is typically located in the file system of the hypervisor. However, disk image files can also be located across a network connection in the remote file system. For example, on a Network File System (NFS).

In general, a guest operating system that uses block devices for local mass storage typically performs better than a guest operating system that uses disk image files for the following reasons:

- Managing the file system where the disk image file is located creates additional resource demand for I/O operations
- Using sparse files in a file system (that supports sparse files) adds logic that uses additional resources and time
- Improper partitioning of mass storage using disk image files can cause unnecessary I/O operations

However, disk image files provide the following benefits:

- Containment: Many disk image files can be located in a single storage unit. For example, disk images files can be located on disks, partitions, logical volumes, and other storage units.
- Usability: Managing multiple files is easier than managing multiple disks, multiple partitions, multiple logical volumes, multiple arrays, and other storage units.
- Mobility: You can easily move files from one location or system to another location or another system.
- Cloning: You can easily copy and modify files for new VMs to use.
- Sparse files save space: Using a file system that supports sparse files conserves unaccessed disk space.
- Remote and network accessibility: Files can be located in file systems on remote systems that are connected by a network.

Sparse files

Learn about sparse image files, including the benefits of sparse image files and cautions about using sparse image files.

Some file systems, like the third extended file system (ext3) in Linux, support a feature for allocating sparse files. With this feature, you can sparsely create disk image files. Traditional (non-sparse) files preallocate all the storage data space when the file is created. A 100 GB file uses 100 GB of storage space. Sparse files are like traditional files except sparse files do not preallocate all of the data in the file. Instead, sparse files allocate only the portions of the file that are written. Areas of the file that are not written are not allocated. The sparse file allocates storage space for new data at the time a new block is written to the file. This allocation provides on-demand growth of storage space for sparse files and ensures the smallest amount of storage space consumption.

The maximum size to which a sparse file can grow equals the size of the file at creation. A sparse file of 100 GB can use 0 - 100 GB of storage space. The amount of GB the sparse file uses depends on how many different portions of the sparse file are allocated by write operations.

With sparse files, you can define a maximum amount of storage space that a guest operating system can use. At the same time, you can limit the amount of actual storage space to the amount that is in use. For example, a guest operating system uses a sparse file for a local storage device. In this situation, only those portions of the disks that are written use storage space within the sparse file. Any portions of the disks that are unaccessed do not use space. To plan for the future growth of the data stored on that device, you can create a preallocated storage area that reserves more space than the requirements. The preallocated storage area can be a partition, Logical Volume Manager (LVM) logical volume, or other storage areas. However, if the storage never grows to the reserved size, the remainder is wasted.

When you use sparse files, you can over-commit disk space. Remember, a sparse file stores a fraction of its total space allocated. Because a sparse file uses less disk space than its maximum, it can store more files in the same storage space than traditional preallocated files. For example, you have two sparse files each at 100 GB. One sparse file has 20 GB of allocated storage space and the other sparse file has 10 GB of allocated storage space. The total storage space used by both sparse files is 30 GB (20 GB + 10 GB). In comparison, you have two traditional preallocated files each at 100 GB. The total storage space used by both traditional preallocated files is 200 GB (100 GB + 100 GB).

Until a sparse file becomes fully allocated, the size of the sparse file is always less than the maximum size. A storage device is over-committed when the maximum size of one or more sparse files is greater than the available storage space. For example, you have 250 GB of available space to store the files from the previous example. When you store the two traditional preallocated files (100 GB each), the available space is 50 GB (250 GB - (100 GB + 100 GB)). In this example, you cannot add another 100 GB file in the remaining 50 GB of available space. When you store the two sparse files (both using 30 GB total), the available space is 220 GB (250 GB - 30 GB). In this example, you can add at least two 100 GB files (sparse files or traditional preallocated files) in the remaining 220 GB of available space. If the additional files are sparse files, you can add more than two additional files (like those previously described) because the storage space used is less than 100 GB.

Continuing the example, you have 250 GB of available space to store four sparse files at 100 GB each. Two sparse files have 20 GB each of allocated storage space and the other two sparse files have 10 GB each of allocated storage space. In total, you have four 100 GB files that use 60 GB of storage space. If the four 100 GB sparse files grow beyond the remaining 190 GB (250 GB - 60 GB) of available space, the guest operating system encounters errors when attempting to write to new blocks. To the guest operating system, the disk (which is really a sparse file) still has available space. However, the sparse file cannot grow because there is no available space on the device in which the sparse file is located.

Over-committing storage provides the following benefits:

- You can make better use of storage space.
- You can store a greater number of sparse files on a given storage device.

However, over-committing storage can create problems if the underlying storage device becomes full. There are no guarantees that the underlying device always has enough available space for sparsely allocated files to grow. You must exercise caution to balance the risk associated with over-committing storage with the advantages of using sparse files.

To reduce risk when over-committing storage, create plans and policies that help you perform the following tasks:

- Avoid using all available space on the underlying storage device.
- Move the disk image files of the guest operating system to another device to provide additional space.

Partitions for disk image files

Learn about the challenges of subdividing a disk image file into partitions, including the misalignment of cylinder boundaries, cylinder sizes, and the page cache.

In many scenarios, the mass storage device of a guest operating system uses an underlying disk image file. You can use standard Linux tools to divide the storage device of a guest operating system into partitions. A common practice for partitions for Linux is using three partitions as follows:

- Use one partition as the boot partition.
- Use another partition as the operating system partition.
- Use the other partition as the swap partition.

Standard partitioning rules dictate the placement of boot loaders and the first partition. With these rules, older operating systems can continue to function on newer and larger storage devices. Most current operating systems also use these standard partitioning rules by default to increase the probability of functional success.

One rule that has not changed over time is the geometry of the disk. To boot older operating systems, the number of cylinders is as small as possible to better adhere to older boot limitations, like DOS and BIOS. To achieve a small cylinder number, the `track size` value is set to 63 and the `heads per cylinder` value is set to 255. Thus, the cylinder size is 63 x 255 = 16065 sectors per cylinder.

Most standard partitioning tools use the first track of the device for the Master Boot Record (MBR) and the boot loader code. The MBR describes the offset, in sectors, where a partition begins and where the partition ends. The default starting offset of the first partition is at the beginning of the second track of the device because the MBR uses the first track. Typically, the beginning of the second track of the device is sector 63, which is the 64th sector from zero. By default, standard partitioning tools usually start and end subsequent partitions on cylinder boundaries. These boundaries typically do not align with processor and operating system architecture elements.

Linux uses a page cache to cache data from I/O devices. The page cache has a granularity of one standard processor memory page, which is 4 KB for Intel processors. The page cache linearly maps a device or partition. In other words, Linux caches the first 4 KB of the device in one page in the page cache. Then Linux caches the second 4 KB of the device into another page in the page cache. Linux caches each successive 4 KB region of the device into a different page in the page cache.

Most Linux file systems ensure that the metadata and files, that are stored within the file system, are aligned on 4 KB (page) boundaries. The second extended file system (ext2) and the third extended file system (ext3) are most commonly used. Both file systems use 4 KB alignment for all data.

Because a disk image file is a file in the file system, the disk image file is 4 KB aligned. However, when the disk image file is partitioned using the default or established partitioning rules, partitions can begin on boundaries or offsets within the image file that are not 4 KB aligned.

The probability of a partition, within a disk image file, starting on a 4 KB alignment is low for the following reasons:

- Partitions within a disk image file typically start and end on cylinder boundaries.
- Cylinder sizes typically are not multiples of 4 KB.

Within the guest operating system, disk I/O operations occur with blocks that are offset in 4 KB multiples from the beginning of the partitions. However, the offset of the partition in the disk image file is typically not 4 KB aligned with respect to the following items:

- The beginning of the disk image file.
- The file system in which the disk image file is located.

As a result, when a guest operating system initiates I/O operations of 4 KB to the partition in the disk image file, the I/O operations span two 4 KB pages of the disk image file. The disk image file is located in the page cache of the hypervisor. If an I/O operation is a write request, the hypervisor must perform a Read-Modify-Write (RMW) operation to complete the I/O request.

For example, the guest operating system initiates 4 KB write operation. The hypervisor reads a page to update the last 1 KB of data. Then, the hypervisor reads the next page to update the remaining 3 KB of data. After the updates are finished, the hypervisor writes both modified pages back to the disk. For every write I/O operation from the guest operating system, the hypervisor must perform up to two read operations and up to two write operations. These extra read and write operations produce an I/O multiplication factor of four. The additional I/O operations create a greater demand on the storage devices. This increased demand can affect the throughput and response times of all the software using the storage devices.

To avoid the I/O affects from partitions that are not 4 KB aligned, ensure that you optimally partition the disk image file. Ensure that partitions start on boundaries that adhere to the 4 KB alignment recommendation. Most standard partitioning tools default to using cylinder values to specify the start and end boundaries of each partition. However, based on default established values used for disk geometry, like `track size` and `heads per cylinder`, cylinder sizes rarely fall on 4 KB alignment values. To specify an optimal start value, you can switch the partition tool to the Advanced or Expert modes.

Creating a disk image file

You can create a disk image file for a guest operating system to use.

To create a disk image file, complete the following steps:

1. Create a disk image file by using the **`qemu-img`** command. For example:

   ```
   qemu-img create -f 100GB-VM-file.img 100GB
   ```

   This example creates a 100 GB file named `100GB-VM-file.img`. If you are using the second extended file system (ext2) or the third extended file system (ext3), the file that you create is a sparse file.
2. Specify the disk image file to the guest operating system by using the **`-drive`** parameter of the **`qemu-kvm`** command. The guest operating system uses the disk image file as a raw, or blank, disk.
3. Optional: Create partitions within the disk image file. Start all partitions, or the most active partitions, on a sector offset that is aligned 4 KB from the start of the disk image file. Use the Expert mode in Linux partitioning tools to specify the starting and ending offsets for partitions in sector values rather than cylinder values. In Expert mode, the starting sector value must be a multiple of eight sectors. Each sector is 512 bytes and 512 bytes x 8 sectors = 4 KB.) The end boundary value is less important because the file system likely ensures that multiples of 4 KB units are used.

Best practices for over-committing processor and memory resources

When over-committing processor and memory resources, you can target system use at 80%, allocate the minimum amount of processor resources, and use page sharing or ballooning instead of swapping.

One of the most alluring aspects of virtualization is that you can run many guest operating systems simultaneously on one system. Consolidating systems with virtualization has been around for quite some time with IBM mainframe virtualization coming about in the 1970s. When you load one system with the workloads of many systems, you increase the usage level of the system making the system more productive. Idle computing resources are wasteful from both an acquisition and operations perspective.

One of the challenges of driving higher system use by consolidating workloads is managing the complexity of the consolidation. Some workloads stress the storage subsystem while other workloads stress the network. Some workloads are active during the business day and other workloads are active at night. To manage this diversity, you can over-commit the processor and memory resources and then control the ability of the guest operating systems to access those resources.

Best practice: Target system use at 80% or lower

Target overall system use at 80% or lower to maximize resource use while maintaining performance.

The hypervisor technology of KVM is based on the Linux kernel. In the KVM environment, each guest operating system is a process that consists of multiple threads. Because the Linux operating system multi-tasks, it can perform the following operations:

- Linux can switch execution between processes.

- Linux can run multiple processes at the same time by using simultaneous multi-processing (SMP) hardware.

KVM uses both operations to over-commit processor resources.

Over-committing processor resources can cause performance penalties. When a processor must switch context from one process to another process, the switch affects the performance of the system. When the current use of the system is low, the performance penalty of the context-switch is low. When the current use of the system is high, the performance penalty of the context-switch is high. In this situation, the performance penalty can increase to the point of negatively affecting the overall performance of the system.

As overall system use increases, the responsiveness of the guest operating systems diminishes. By targeting overall system use at 80% or lower, you can reduce the performance penalties as follows:

- You provide the system with 20% of available resources to handle unexpected bursts in activity.
- You can minimize the negative effect on the responsiveness of the guest operating systems.

The target of 80% or lower is defined as a percentage of system use and not as a specific number of virtual processors. At any time, some guest operating systems might run with some virtual processors inactive. Running guest operating systems with inactive virtual processors minimally affects the processor use of the system, making it possible to over-commit processor resources.

Best practice: Allocate the minimum amount of processor resources

To maximize performance, allocate the minimum amount of virtual processors necessary for each guest operating system to successfully operate. Do not allocate virtual processors to guest operating systems that the guest operating systems do not need.

If you allocate many virtual processors to the guest operating systems, the system functionally works. However, this configuration has scaling issues that cause performance degradations. You can reduce the effect of these scaling issues by tuning KVM. For example, you can pin virtual processors to physical processors. However, some of the tuning techniques impart additional restrictions.

Related information:

Processor pinning

Best practice: Over-commit memory resources by using page sharing or ballooning

You can use page sharing, ballooning, or swapping to over-commit memory resources. Page sharing and ballooning outperform swapping.

There are three main technologies that you can use to over-commit memory in Red Hat Enterprise Linux 5.5: page sharing, ballooning, and swapping. The primary goal is to over-commit memory with the minimum negative effect on performance.

Page sharing

You can use page sharing to over-commit the memory resources of a guest operating system. In Red Hat Enterprise Linux 5.5, difficulty quantifying the memory gains of page sharing limits the usefulness of page sharing in production environments. However, advances in Red Hat Enterprise Linux 6 enhance the ability to manage page sharing.

Kernel same-page merging (KSM):

KSM is a kernel feature in KVM that shares memory pages between various processes, over-committing the memory.

How KSM works

The memory that KSM scans must be registered with KSM. The application that owns the memory registers the memory with KSM. In this case, the owning application is QEMU and it has been modified to register the memory with KSM. Currently, there are no other KVM applications that use KSM.

Page sharing with KSM works as follows:

1. You configure KSM to scan memory ranges.
2. KSM looks for pages that contain identical content within the defined memory ranges.
3. When KSM finds two or more identical pages, KSM identifies those pages as shareable.
4. KSM replaces the shareable pages with a single page that is write protected.
5. If later a process requests to modify one of the pages from step 3, KSM creates a page by using copy-on-write.
6. The requesting process modifies the new page.

How KSM affects processor resources

Scanning memory pages is a processor-intensive operation. Therefore, tuning KSM is critical, especially in environments with high processor use. If you configure KSM to act too aggressively, KSM might use an entire processor thread. The following actions might cause KSM to act too aggressively:

- Configuring KSM to scan too many pages per period.
- Configuring KSM to scan too often.

You can measure the processor load that KSM applies by running the **`top`** utility. Then, watch the process use of the KSM kernel thread. For Red Hat Enterprise Linux 5.4, 5.5, and 5.6, the KSM kernel thread is the `kksmd` process thread. For Red Hat Enterprise Linux 6 and later, it is the `ksmd` process thread.

How KSM affects memory resources in Red Hat Enterprise Linux 5.4, 5.5, and 5.6

For Red Hat Enterprise Linux 5.4, 5.5, and 5.6, quantifying the memory performance gained by using KSM is difficult for the following reasons:

- The version of KSM that is backported to Red Hat Enterprise Linux 5.4 is missing several statistics that you can use to evaluate the effectiveness of KSM. Without these statistics, you must aggregate information from multiple sources to determine the effects of KSM. For example, you can obtain information from the `/proc/meminfo` directory and the **`top`** utility.
- You can deploy KSM and save memory, but you cannot easily calculate how much memory KSM actually saves. Without this calculation, you cannot determine how many additional guest operating systems you can deploy without forcing the system to start swapping memory. Swapping memory creates the following complications:
  - Swapping memory can lead to a huge performance penalty.
  - The system cannot swap memory pages that are shared by KSM because they are pinned.

The difficulties associated with quantifying the memory gains of KSM limits the usefulness of KSM in production environments.

Page size restriction for KSM in Red Hat Enterprise Linux 6 and later

Red Hat Enterprise Linux 6 enables a feature called transparent huge pages. This feature allows anonymously-mapped memory, such as KVM guest memory, to be automatically backed by huge pages when possible. In Red Hat Enterprise Linux 6, guest memory that is backed by transparent huge pages cannot be shared by KSM. However, in Red Hat Enterprise Linux 6.1 this restriction no longer exists, and KVM guest memory backed by transparent huge pages can be decomposed into small pages and shared with KSM.

Page size restriction for KSM in Red Hat Enterprise Linux 5.4, 5.5, and 5.6

In Red Hat Enterprise Linux 5.4, 5.5, and 5.6, KSM can share only small pages and not huge pages. Therefore, you cannot use **libhugetlbfs** to enable huge-page backing of the memory of the guest operating system.

Related concepts:

"Swapping" on page 19
You can use swapping to over-commit the memory resources of a guest operating system. However, page sharing and ballooning are better methods.

Related information:

Huge pages

Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 6 and later:

You can control Kernel same-page merging (KSM) using the **ksm** and **ksmtuned** utilities.

In Red Hat Enterprise Linux 6, the **ksm** service can be used to start or stop KSM. When the **ksm** service is not started, KSM shares at most 2000 pages. When it is started, KSM shares up to half of all system memory.

- To check the status of the **ksm** service, enter the following command:

  ```
  service ksm status
  ```

  Status such as `ksm is running` is shown.
- To determine what run levels the **ksm** service will start under, enter the following command:

  ```
  chkconfig ksm --list
  ```

  Output similar to the following is shown: `ksm 0:off 1:off 2:off 3:off 4:off 5:off 6:off`.
- To start the **ksm** service, enter the following command:

  ```
  service ksm start
  ```
- To stop the **ksm** service, enter the following command:

  ```
  service ksm stop
  ```
- To modify the default state to enable the **ksm** service to start on boot, enter the following command:

  ```
  chkconfig ksm on
  ```
- To modify the default state to disable the **ksm** service from starting on boot, enter the following command:

  ```
  chkconfig ksm off
  ```

Available in Red Hat Enterprise Linux 6, the **ksmtuned** service is an automatic tuning utility. The **ksmtuned** service is used to adjust the KSM settings depending on the current state of memory utilization on the host system. The service is notified by libvirt whenever a guest is created and destroyed, and it adjusts the KSM settings accordingly. The service can be configured by editing `/etc/ksmtuned.conf`.

- To determine the status of the **ksmtuned** service, enter the following command:

  ```
  service ksmtuned status
  ```

  Output similar to the following is shown: `ksmtuned (pid 12423) is running...`.
- To determine what run levels the **ksmtuned** service will start under, enter the following command:

  ```
  chkconfig ksmtuned --list
  ```

  Output similar to the following is shown: `ksmtuned 0:off 1:off 2:off 3:off 4:off 5:off 6:off`.
- To start the **ksmtuned** service, enter the following command:

  ```
  service ksmtuned start
  ```

- To stop the **ksmtuned** service, enter the following command:

  ```
  service ksmtuned stop
  ```

- To modify the default state to enable the **ksmtuned** service to start on boot, enter the following command:

  ```
  chkconfig ksmtuned on
  ```

- To modify the default state to disable the **ksmtuned** service to from starting on boot, enter the following command:

  ```
  chkconfig ksmtuned off
  ```

- To force the **ksmtuned** service to readjust the current KSM control settings, enter the following command:

  ```
  service ksmtuned retune
  ```

After KSM runs for a while, determine the resulting memory savings. For instructions, see "Determining the memory savings of KSM with Red Hat Enterprise Linux 6 and later."

Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 5.4, 5.5, and 5.6:

You can start KSM, view KSM configuration settings, and stop KSM by using the **ksmctl** utility.

Red Hat Enterprise Linux 5.4 supports KSM as a backport from recent kernel releases. You control the backported version of KSM by using the **ksmctl** utility.

To run KSM, complete the following steps:

1. Activate KSM by typing the following command:

   ```
   ksmctl start pages period
   ```

   where:

   - *pages* is the number of pages to be scanned per period.
   - *period* is how often, in milliseconds, to scan.

   For example, to scan 440 pages every 5 seconds, type the following command:

   ```
   ksmctl start 440 5000
   ```

2. View the current KSM configuration settings by typing the following command:

   ```
   ksmctl info
   ```

3. Stop KSM from scanning by typing the following command:

   ```
   ksmctl stop
   ```

After KSM runs for a while, determine the resulting memory savings. For instructions, see "Determining the memory savings of KSM with Red Hat Enterprise Linux 5.4, 5.5, and 5.6" on page 16.

Determining the memory savings of KSM with Red Hat Enterprise Linux 6 and later:

Red Hat Enterprise Linux 6 provides statistics that allow you to determine the memory savings and efficiency of KSM.

Before you can determine the memory savings of KSM, you must first run KSM. For instructions, see "Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 6 and later" on page 13.

When the KSM module is loaded, statistics relevant to the operational state of KSM are available at the following location: `/sys/kernel/mm/ksm` . This directory contains a file for each of several parameters. Each file contains a single value pertaining to the current state of KSM. Understanding what these values are and how they relate to each other enables you to determine how KSM is performing.

full_scans
The value of **full_scans** indicates the number of times that all shareable memory areas have been scanned. If this number changes while other numbers (such as pages shared) do not, that indicates that KVM is not finding any new shareable memory even though it is looking for additional memory to share.

pages_shared
The value of **pages_shared** indicates how many pages KSM is using to back the shared pool that it has built up. When you multiply this value by the page size (usually 4 KB), you can determine the total amount of memory that KSM is using.

pages_sharing
The value of **pages_sharing** is the number of guest memory pages that KSM has shared using the shared pool. Some pages, such as randomized or encrypted data, can be shared few times, if at all. Other pages, such as zeroed pages, can be shared many times. KSM efficiency can be calculated by dividing the **pages_sharing** value by the**pages_shared** value. A higher result indicates more times that a page is shared by KSM, and a greater efficiency of KSM.

pages_to_scan
The value of **pages_to_scan** determines how many memory pages KSM will analyze for sharing potential on each pass. The higher this value is, the more aggressively KSM has been configured to share memory. If the **ksmtuned** service is running, it will manipulate this value as needed.

pages_unshared
The value of **pages_unshared** indicates how many pages KSM is scanning that cannot be shared because they are unique. KSM's wasted effort can be evaluated by dividing the **pages_unshared** value by the **pages_sharing** value. A lower result indicates less wasted effort on the part of KVM, and better KVM performance.

pages_volatile
The value of **pages_volatile** indicates the number of pages that have content that is changing too rapidly for memory sharing. If the number of volatile pages is high, that is an indication that the running guests are not good candidates for memory sharing.

run
The value of **run** indicates the current state of KSM.

0 Indicates that KSM is not currently running, but any previously shared pages will remain shared.

1 Indicates that KSM is currently active and attempting to share additional memory.

2 Indicates that KSM has been stopped and all previously shared pages have been unshared.

If the **ksmtuned** or **ksm** services are running, they will manipulate this value as needed.

sleep_millisecs
The value of **sleep_millisecs** value indicates how long KSM will sleep in between each scanning pass. The lower this value is the more aggressively KSM has been configured to share memory. If the **ksmtuned** service is running, it will manipulate this value based on the amount of system memory.

The aggressiveness of KSM memory scanning is determined by both the **pages_to_scan** and **sleep_millisecs** parameters. These two parameters combine to determine how often KSM wakes up to scan memory and for how long. The efficiency of KSM can be measured through analysis of the **pages_shared**, **pages_sharing**, **pages_unshared**, and **pages_volatile** metrics. You can experiment with KSM and determine your own appropriate thresholds for evaluating KSM based on these values and the calculations described.

Determining the memory savings of KSM with Red Hat Enterprise Linux 5.4, 5.5, and 5.6:

For Red Hat Enterprise Linux 5.4, 5.5, and 5.6, you can identify the memory savings of KSM by looking at the anonymous pages value and at the output of the **top** utility for the QEMU processes.

Before you can determine the memory savings of KSM, you must first run KSM. For instructions, see "Running Kernel same-page merging (KSM) with Red Hat Enterprise Linux 5.4, 5.5, and 5.6" on page 14.

To determine the memory savings of KSM, complete the following steps:

1. View the memory savings of KSM by looking at the anonymous pages line in the /proc/meminfo directory. Your output might look like the following output:

   ```
   AnonPages: 1622216 kB
   ```

   If no significant tasks are running on the host, other than KSM, most of the anonymous pages are likely the memory of guest operating systems. When you activate KSM and it starts to share memory pages, the number of anonymous pages decreases.

   The anonymous pages value is the best way to ascertain the memory savings of KSM. However, the anonymous pages value changes constantly, making it difficult to observe KSM performance. The value changes because guest operating systems fault in memory pages on demand. The hypervisor allocates memory to the guest operating system only when the guest operating system attempts to use the memory. The memory is not included in the anonymous pages value until the guest operating system faults in the memory. Over time a guest operating system grows larger until it reaches its maximum size and stops growing. The anonymous pages value changes as the guest operating system faults in pages. The anonymous pages value does not change when a guest operating system starts and runs for a long time with all of its memory faulted in.
2. View the amount of shared memory for each guest operating system by looking at the output of the **top** utility for the QEMU processes. The output might look similar to the following output:

   ```
   PID  USER     PR NI VIRT  RES  SHR S %CPU %MEM  TIME+ COMMAND
   ...
   7260 root     15 0 1240m 255m 196m S  1.0  0.4 0:25.22 qemu-kvm
   7215 root     15 0 1238m 252m 190m S  0.7  0.4 0:25.17 qemu-kvm
   ...
   ```

   As KSM shares pages, the values in the SHR column increase for the guest operating systems. The changes in the SHR column might significantly fluctuate depending on the KSM configuration settings. The values in the SHR column only partially indicate how much memory KSM saves. The values reported in the SHR column do not distinguish between consolidating 1000 pages into one page or consolidating 1000 pages into 500 pages. The values reflect only that KSM shared 1000 pages.

Controlling KSM usage on a per-guest basis with Red Hat Enterprise Linux 6.1 and later

You can disable KSM usage by explicitly identifying which guests should not share memory with other guests. This feature, introduced in Red Hat Enterprise Linux 6.1, allows you to control whether KSM is used for specific guests. Previously, if KSM was enabled, then all guests running on the system were subject to sharing memory with each other. With this feature, the system administrator can isolate one or more guests from system-wide memory sharing.

One reason to isolate a guest from KSM is to enforce ideal memory locality for guests where performance is critical. When guest performance is critical, one optimization is to set the guest's processor affinity to limit its execution to one or more CPUs. Often limiting the execution is done by containing a guest within a single NUMA node. Generally this practice lowers memory latency. When KSM is used, one or more memory pages with the same content are collapsed into a single page. This page is then shared to all guests that have a reference to one of the original pages. When this process occurs and CPU affinity is used for the guests, guests that are not on the same NUMA node as the shared page must access the page as remote memory. Remote access can decrease performance. By disabling KSM for guests where performance is critical, remote memory references can be avoided.

To control KSM usage on a per-guest basis, you must first determine which method of guest control to use. You can decide whether to call **qemu-kvm** directly or to use libvirt to handle guest management.

Calling `qemu-kvm` directly

You can invoke **qemu-kvm** directly with a command.

1. Type the following command:
   ```
   /usr/libexec/qemu-kvm -M rhel6.1.0 -enable-kvm -m 2048 -smp
    1,sockets=1,cores=1,threads=1 -name myguest ...
   ```
2. Disable KSM for that guest by adding **-redhat-disable-KSM** to the **qemu-kvm** command, as follows:
   ```
   /usr/libexec/qemu-kvm -M rhel6.1.0 -enable-kvm -m 2048 -redhat-disable-KSM
     -smp 1,sockets=1,cores=1,threads=1 -name myguest ...
   ```

Managing guests with libvirt

You can use libvirt to manage your KVM environment using **virt-install**, **virsh**, and **virt-manager** tools. Using these tools, you can instruct libvirt to disable KSM for a specific guest by modifying the guest's XML. One way to do this is to use the **virsh** tool.

1. Enter the following command:
   ```
   virsh edit myguest
   ```

   After you enter this command, an editing session of the XML that describes the guest configuration opens. The editor used is defined by the environment variable EDITOR; if the variable is not set it defaults to vi. The beginning of the XML file is similar to the following example:
   ```
   <domain type='kvm'>
    <name>myguest<\name>
    <uuid>...</uuid>
    <memory>2048</memory>
    <currentMemory>2048</currentMemory>
    <vcpu>1</vcpu>
    ...
   </domain>
   ```
2. To specify that KSM must not be used for this guest, you must add a **nosharepages** element inside of a **memoryBacking** element. Doing so makes the XML look similar to the following example:
   ```
   <domain type='kvm'>
    <name>myguest<\name>
    <uuid>...</uuid>
    <memory>2048</memory>
    <currentMemory>2048</currentMemory>
    <memoryBacking>
     <nosharepages/>
    </memoryBacking>
    <vcpu>1</vcpu>
    ...
   </domain>
   ```
3. After editing the XML file, start the guest, or stop and start the guest if it is already running.

Ballooning

You can use ballooning to over-commit the memory resources of a guest operating system. However, ballooning has some disadvantages that limit the usefulness of ballooning outside of specialized scenarios.

Ballooning is a cooperative operation between the host hypervisor and the guest operating systems.

How ballooning works

Ballooning works as follows:

1. The hypervisor sends a request to the guest operating system to return some amount of memory back to the hypervisor.
2. The `virtio_balloon` driver in the guest operating system receives the request from the hypervisor.
3. The `virtio_balloon` driver inflates a balloon of memory inside the guest operating system:
   - The guest operating system cannot use or access the memory inside the balloon.
   - The `virtio_balloon` driver tries to satisfy the request from the hypervisor. However, it is possible that the `virtio_balloon` driver might not be able to inflate the balloon of memory to fully satisfy the request. For example, when an application running in the guest operating system pins memory, the `virtio_balloon` driver might not be able to find enough available memory to satisfy the request. In this situation, the driver inflates the balloon of memory as much as possible even though the amount of memory does not fully satisfy the request.
4. The guest operating system returns the balloon of memory back to the hypervisor.
5. The hypervisor allocates the memory from the balloon elsewhere as needed.
6. If the memory from the balloon later becomes available, the hypervisor can return the memory to the guest operating system:
   a. The hypervisor sends a request to the `virtio_balloon` driver in the guest operating system.
   b. The request instructs the guest operating system to deflate the balloon of memory.
   c. The memory in the balloon becomes available to the guest operating system to allocate as needed.

Balloon management

You can initiate balloon operations in one of the following ways:

- If you start QEMU directly, you can control ballooning with the QEMU monitor.
- If you manage guest operating systems with the `libvirt` API, you can control ballooning with a management program that supports the `libvirt` API. For example, you can use the **`setmem`** command in the `virsh` management program.

Advantages of ballooning

- Ballooning can potentially save you massive amounts of memory because you can control and monitor ballooning. Unlike page sharing, the system only makes changes based on the commands that you issue to it. Then, you can monitor the memory and verify the changes.
- Ballooning can be subtle by requesting small amounts of memory, and ballooning can be aggressive by requesting large amounts of memory.
- With ballooning, the hypervisor can relieve memory pressure by inflating a memory balloon in a guest operating system and returning the memory to the hypervisor. The hypervisor is not required to allocate the memory to another guest operating system.

Disadvantages of ballooning

- Ballooning requires that you load the `virtio_balloon` driver on the guest operating system. In contrast, swapping and page sharing work independently of the guest operating system and do not require the cooperation of the guest operating system.
- The `virtio_balloon` driver is available only for guest operating systems running Red Hat Enterprise Linux 5.5 and later.
- Ballooning can potentially negatively affect the performance of the guest operating system because ballooning can remove large amounts of memory from the guest operating system. This memory removal can cause the following situations:
  – Applications running in the guest operating system can fail.
  – The amount of block I/O in the guest operating system can increase because it is harder for the guest operating system to cache data.
- Red Hat Enterprise Linux provides no mechanism that you can use to manage the ballooning. The manual administration requirement of ballooning makes ballooning impractical for deployment in a

production environment. Ballooning is not a practical technology for KVM until an autonomic management technology exists that can perform the following tasks:

1. Monitor the system.
2. Decide how and when to perform ballooning operations.

Until that time, treat the ballooning capabilities that KVM currently supports as a technology preview with limited uses outside of specialized scenarios.

Related concepts:

"Best practice: Virtualize memory resources by using the `virtio_balloon` driver" on page 5
The `virtio_balloon` driver provides a communication pathway between the hypervisor and the guest operating system. The hypervisor uses this communication pathway to over-commit memory resources.

Swapping

You can use swapping to over-commit the memory resources of a guest operating system. However, page sharing and ballooning are better methods.

Compared to page sharing and ballooning, swapping is the most mature. However, swapping negatively affects performance. Swapping is the least ideal method for over-committing the memory of a guest operating system for the following reasons:

- Swapping in general usually generates poor performance.
- When using Intel Nehalem processors with extended page tables (EPT), the Linux host cannot accurately determine which memory pages the guest operating system is least likely to use. Therefore, the Linux host cannot accurately determine which memory pages are ideal candidates to swap out.

Related information:

The swapiness value

Best practices for networking

You can use the QEMU networking options or PCI pass-through to provide networking support to guest operating systems.

Best practice: Use the `tap` networking option in QEMU

Learn about QEMU networking options and Linux bridge support.

QEMU networking options

QEMU networking support includes the following options:

User
The user option is a networking environment that supports the TCP and UDP protocols. QEMU provides services to the guest operating system such as DHCP, TFTP, SMB, and DNS. QEMU acts as a gateway and a firewall for the guest operating system such that communication from the guest operating system appears to be from the QEMU host.

You cannot initiate a connection to the guest operating system without help from QEMU. For this type of connection, QEMU provides the **`redir`** parameter. The **`redir`** parameter redirects TCP or UDP connections from a specific port on the host to a specific port on the guest operating system.

The user option is the default networking option in QEMU.

Socket
The socket option is used to connect together the network stacks of multiple QEMU processes. You create one QEMU process that listens on a specified port. Then, you create other QEMU processes that connect to the specified port.

Tap
The tap option connects the network stack of the guest operating system to a TAP network device on the host. By using a TAP device, QEMU can perform the following actions:

- Receive networking packets from the host network stack and pass the packets to the guest operating system.
- Receive networking packets from the guest operating system and inject the packets into the host network stack.

Use the tap networking option because it provides full networking capability to a guest operating system.

Linux bridge support

Perform the following tasks to add and remove TAP network devices to and from the bridges when you start and stop a guest operating system:

1. Create the bridges before you start the first guest operating system.
2. If you want the guest operating system to access the physical network, add an Ethernet device to the bridge.
3. Specify a script for configuring the tap network device and a script for unconfiguring the tap network device.

Guest operating systems that you add to the same bridge can communicate with each other. If you want multiple subnets available to the guest operating systems, define multiple bridges. In this situation, each bridge is for a unique subnet. Each bridge contains the TAP devices that are associated with the NICs of the guest operating systems that are part of the same subnet.

When using the Linux bridge, consider the form of receive offload supported by the network adapter. Receive offload aggregates multiple packets into a single packet to improve network performance. Many network adapters provide a form of receive offload in the adapter, which is often referred to as large receive offload (LRO). The Linux kernel provides a form of receive offload called generic receive offload (GRO). Linux bridges can forward GRO packets. Linux bridges cannot forward LRO packets unless the driver is compliant with GRO. Therefore, in order for guest operating systems to use receive offload the network adapter must support GRO.

QEMU VLAN

QEMU networking uses a networking technology that is like VLAN. A QEMU VLAN is not an 802.1q VLAN. Rather, a QEMU VLAN is a way for QEMU to forward packets to guest operating systems that are on the same VLAN. When you define the networking options for a guest operating system, you can specify a VLAN to which the network interface is assigned. If you do not specify a VLAN, by default QEMU assigns the interface to VLAN 0. In general, if you create more than one network interface for a guest operating system, assign the network interfaces to different VLANs.

Example

The following example shows the qemu-kvm options you can use to set up multiple interfaces:

```
-net nic,model=virtio,vlan=0,macaddr=00:16:3e:00:01:01
-net tap,vlan=0,script=/root/ifup-br0,downscript=/root/ifdown-br0
-net nic,model=virtio,vlan=1,macaddr=00:16:3e:00:01:02
-net tap,vlan=1,script=/root/ifup-br1,downscript=/root/ifdown-br1
```

The example shows two network devices configured for a guest operating system as follows:

- The **- net nic** command defines a network adapter in the guest operating system. Both network devices are para-virtualized devices which is indicated by the `model=virtio` value. Both devices also have unique MAC addresses which is indicated by the `macaddr` values. Each network device is on a different VLAN. The first device is on VLAN 0 and the second network device is on VLAN 1.
- The **-net tap** command defines how QEMU configures the host. Each network device is added to and removed from a different bridge by using scripts. The first device is added to the `br0` bridge by using the `/root/ifup-br0` script and removed from the `br0` bridge by using the `/root/ifdown-br0` script.

Similarly, the second network device is added to the br1 bridge by using the /root/ifup-br1 script and removed from the br1 bridge by using the /root/ifdown-br1 script. Each network device is also on a different VLAN. The first device is on VLAN 0 and the second network device is on VLAN 1.

Best practice: Configure para-virtualized network devices

You can configure para-virtualized network devices for the guest operating systems.

When you select a network device, choose a para-virtualized network device instead of an emulated network device. A para-virtualized network device improves throughput and latency. (If the guest operating system does not support a para-virtualized network device, then choose one of the emulated devices that the guest operating system supports.)

To configure para-virtualized network devices, complete the following steps:

1. Verify that the system supports para-virtualized devices:
   a. Obtain a list of supported devices by typing the following command:
      ```
      $ qemu-kvm -net nic,model=?
      ```
      The output might look similar to the following output:
      ```
      qemu: Supported NIC models: ne2k_pci,i82551,i82557b,i82559er,rtl8139,e1000,pcnet,virtio
      ```
   b. Verify that virtio is in the list of supported devices. The virtio value represents para-virtualized devices.
2. Specify the networking device by using the **-net nic** option in **qemu-kvm** as follows:
   - Set model=virtio to specify a para-virtualized device.
   - Specify a unique MAC address for each network device on each guest operating system to avoid MAC address collisions. You can use the MAC address prefix 00:16:3e to generate unique MAC addresses.

     Note: The 00:16:3e prefix is the prefix assigned to the guest operating systems of the Xen hypervisor. However, the prefix is meant for generating MAC addresses for guest operating systems in general.
   - With Red Hat Enterprise Linux 6, if you do not require QEMU VLAN support, you can take advantage of network offload performance capabilities. Add the **netdev** keyword to the **-net nic** option, and replace the **-net tap** option with the **-netdev tap** option.

   Example for Red Hat Enterprise Linux 5.4 and later:
   ```
   -net nic,model=virtio,vlan=0,macaddr=00:16:3e:00:01:01
   ```
   Example for Red Hat Enterprise Linux 6 and later:
   ```
   -net nic,model=virtio,macaddr=00:16:3e:00:01:01,netdev=nic-0
   -netdev tap,id=nic-0,script=/root/ifup-br0,downscript=/root/ifdown-br0
   ```

Related concepts:

"Best practice: Para-virtualize devices by using the VirtIO API" on page 1
KVM can provide emulated or para-virtualized devices to the guest operating systems. Compared to emulated devices, para-virtualized devices provide lower latency and higher throughput for I/O operations of guest operating systems. KVM includes the VirtIO API to para-virtualize devices.

Related information:

Red Hat Enterprise Virtualization for Servers

Adding guest para-virtualized network devices with libvirt

Your guest network interfaces can be para-virtualized network devices.

To enable the guest to use a para-virtualized network device, complete the following steps:

1. Shut down the guest operating system.
2. Edit the guest XML definition with the **virsh edit** command. For example:

   ```
   virsh edit guest-name
   ```
3. Change the network definition to use the “virtio” model type. For example:

   ```
   <interface type='bridge'>
     ...
     <model type='virtio'/>
     ...
   </interface>
   ```
4. Start the guest operating system.

Using a host bridged network device with libvirt

Your guest network interfaces can use bridged network devices.

To enable the guest to use a host bridged network device, complete the following steps:

1. Shut down the guest operating system.
2. Edit the guest XML definition with the **virsh edit** command. For example:

   ```
   virsh edit guest-name
   ```
3. Change the network definition to use the “bridge” option that specifies the name of the bridged network device. For example:

   ```
   <interface type='bridge'>
     ...
     <source bridge='br0'/>
     ...
   </interface>
   ```
4. Start the guest operating system.

Best practice: Use vhost-net for high-bandwidth applications in Red Hat Enterprise Linux 6.1 and later

Beginning with Red Hat Enterprise Linux 6.1, you can use an in-kernel guest networking performance enhancement called vhost-net. vhost-net moves network packets between the guest and the host system using the Linux kernel rather than QEMU. This avoids context switches from the kernel to user space to improve overall performance. Use of vhost-net might also improve network performance and latency.

To use of vhost-net support, add **vhost=on** to the **-netdev** option.

For example:

```
-net nic,model=virtio,macaddr=00:16:3e:00:01:01,netdev=nic-0
-netdev tap,id=nic-0,script=/root/ifup-br0,downscript=/root/ifdown-br0,vhost=on
```

Disabling vhost-net in-kernel support using libvirt

By default, libvirt uses the vhost-net in-kernel support if it is available.

To disable vhost-net in-kernel support, complete the following steps:

1. Shut down the guest operating system.
2. Edit the guest XML definition with the **virsh edit** command. For example:

   ```
   virsh edit guest-name
   ```
3. Change the network definition to use the “qemu” driver type. For example:

   ```
   <interface type='bridge'>
     ...
     <model type='virtio'/>
     <driver name='qemu'/>
     ...
   </interface>
   ```

4. Start the guest operating system.

Best practice: Use PCI pass-through to enhance network performance

You can assign a PCI device directly to a guest operating system instead of emulating network devices.

PCI pass-through

With PCI pass-through, you can assign a PCI device directly to one guest operating system. You do not need to emulate a network device.

When you use PCI pass-through, the PCI device becomes unavailable to the host and to all other guest operating systems. Ideally you need one networking device for each guest operating system and one networking device for the host. If you do not have enough networking devices, then you must choose which guest operating systems use PCI pass-through and which do not.

An alternative to using a complete networking device is to use a virtual networking device, or an SR-IOV device. An SR-IOV device supports I/O virtualization. You create multiple virtual function (VF) devices that communicate with the physical function (PF) device. Each VF device operates as a separate PCI device that you can use with PCI pass-through. You can create a VF device for each guest operating system and maintain access to the PF device in the host. The number of VF devices that you can create depends on the manufacturer of the device.

PCI pass-through reduces the flexibility of migrating the guest operating system to a new host. When you migrate a guest operating system, the PCI pass-through hardware on the current host must be available on the new host.

Setting up PCI pass-through

You can configure a networking device to use PCI pass-through.

Before you start, verify that your host supports either Intel VT-d or AMD IOMMU. Then, verify that this support is enabled in both the machine BIOS and the Linux kernel.

To set up PCI pass-through, complete the following steps:

1. Identify the ID and other slot information that is associated with the PCI device by typing the following command:

   ```
   lspci -nn
   ```

   Your output might include a line like the following line, which identifies the network adapter:

   ```
   15:00.0 Ethernet controller [0200]: Intel Corporation 82599EB 10-Gigabit Network \
   Connection [8086:10fb] (rev 01)
   ```

2. Unbind the device from the host by using the **echo** command. For example:

   ```
   echo "8086 10fb" > /sys/bus/pci/drivers/pci-stub/new_id
   echo 0000:15:00.0 > /sys/bus/pci/devices/0000:15:00.0/driver/unbind
   echo 0000:15:00.0 > /sys/bus/pci/drivers/pci-stub/bind
   ```

3. Shut down the guest operating system.
4. Assign the device to the guest operating system by using the `pcidevice` **qemu-kvm** option. For example:

   ```
   -pcidevice host=15:00.0
   ```

5. Start the guest operating system.
6. Test the networking device to verify that it supports PCI pass-through.

Adding pass-through devices to guests with libvirt

You can simplify using PCI pass-through by adding pass-through devices to guests with libvirt.

To add a PCI pass-through device to a guest, complete the following steps:

1. Identify the ID and other slot information that is associated with the PCI device by typing the following command:

   ```
   lspci -nn
   ```

   Your output might include a line like the following line, which identifies the network adapter:

   ```
   15:00.0 Ethernet controller [0200]: Intel Corporation 82599EB 10-Gigabit Network \
   Connection [8086:10fb] (rev 01)
   ```
2. Shut down the guest operating system.
3. Edit the guest XML definition with the **virsh edit** command. For example:

   ```
   virsh edit guest-name
   ```
4. Add a <hostdev> definition to the <devices> section, using bus, slot, and function attribute values identified in Step 1. For example:

   ```
   <devices>
     ...
     <hostdev mode='subsystem' type='pci' managed='yes'>
       <source>
         <address domain='0x000' bus='0x15' slot='0x00' function='0x0'/>
       </source>
     </hostdev>
   </devices>
   ```
5. Start the guest operating system.
6. Test the networking device to verify that it supports PCI pass-through.

Best practices for block I/O performance

Learn about general best practices and I/O scheduler options for optimizing block I/O performance.

Related concepts:

"Best practice: Optimize performance by using the virtio_blk and virtio_net drivers" on page 3
The virtio_blk driver uses the VirtIO API to provide high performance for storage I/O devices, especially in large enterprise storage systems. The virtio_net driver uses the VirtIO API to provide increased network performance.

General best practices for block I/O performance

You can use the ext3 file system and raw partitions to optimize block I/O performance.

Best practice: Use the ext3 file system

For best performance in Red Hat Enterprise Linux 5, use the third extended file system (ext3) instead of the fourth extended file system (ext4). ext4 is only partially optimized in Red Hat Enterprise Linux 5.

Red Hat Enterprise Linux 6 fully optimizes ext4. However, because the ext4 default setting for the file system barrier is ON, I/O operations that involve much metadata might experience negative performance impacts. Keeping the file system barrier enabled is a best practice for data integrity and protection in a power loss. This is especially true if you are not sure which QEMU caching mode is being used and whether the disk write cache is enabled for the KVM guest. You should consider disabling the file system barriers only if the guest is being set up with **cache = writethrough** (default) or **cache = directsync**. See "Best practice: KVM guest caching modes" on page 25 for more information.

Best practice: Use raw partitions

The modified version of QEMU that is included in KVM supports raw partitions and the qcow2 image format. Unless you need features that are supported by qcow2, such as snapshots, use raw partitions for best performance.

Related information:

KVM/QEMU Storage Stack Performance Discussion from the Linux Plumbers Conference

Best practice: KVM guest caching modes

You can optimize performance by being aware of and selecting the best guest caching mode for your environment.

The operating system maintains a *page cache* to improve the storage I/O performance. With the page cache, write operations to the storage system are considered completed after the data has been copied to the page cache. Read operations can be satisfied from the page cache if the data requested is in the cache. The page cache is copied to permanent storage using **`fsync(2)`**. Direct I/O requests bypass the page cache. In the KVM environment, both the host and guest operating systems can maintain their own page caches, resulting in two copies of data in memory. In general, it is better to bypass at least one of these page caches. If the application running in the guest is using direct I/O operations, then the guest cache would be bypassed. If the guest is set up with no caching, then the host page cache is bypassed, effectively turning all I/O operations from the guest into direct I/O operations on the host.

The *disk write cache* is intended to help improve the performance of write operations to the storage system. These write operations are considered completed after they reach the disk write cache, even though the data might not be on the physical disk media. As a result, the disk write cache might lose its contents (data) during a power failure unless the disk write cache has battery backup. To ensure that the write data is actually on physical disk media, applications must issue **`fsync(2)`**. In general, with the disk write cache enabled, the write performance is improved significantly, but the data integrity and protection in case of power loss can be ensured only if the applications and the storage stack transfer the cache data to permanent storage correctly. However, if the disk write cache is disabled, the write performance might suffer while the risk of data loss in a power failure is lessened.

In Red Hat Enterprise Linux 6, KVM and QEMU (the hardware emulator used by KVM) support the following caching modes for KVM guests:

- `writethrough`

  `writethrough` mode is the default caching mode. With caching set to `writethrough` mode, the host page cache is enabled, but the disk write cache is disabled for the guest. Consequently, this caching mode ensures data integrity even if the applications and storage stack in the guest do not transfer data to permanent storage properly (either through **`fsync`** operations or file system barriers). Because the host page cache is enabled in this mode, the read performance for applications running in the guest is generally better. However, the write performance might be reduced because the disk write cache is disabled.
- `writeback`

  With caching set to `writeback` mode, both the host page cache and the disk write cache are enabled for the guest. Because of this, the I/O performance for applications running in the guest is good, but the data is not protected in a power failure. As a result, this caching mode is recommended only for temporary data where potential data loss is not a concern.
- `none`

  With caching mode set to `none`, the host page cache is disabled, but the disk write cache is enabled for the guest. In this mode, the write performance in the guest is optimal because write operations bypass the host page cache and go directly to the disk write cache. If the disk write cache is battery-backed, or if the applications or storage stack in the guest transfer data properly (either through **`fsync`** operations or file system barriers), then data integrity can be ensured. However, because the host page cache is disabled, the read performance in the guest would not be as good as in the modes where the host page cache is enabled, such as `writethrough` mode.
- `unsafe`

Caching mode of `unsafe` ignores cache transfer operations completely. As its name implies, this caching mode should be used only for temporary data where data loss is not a concern. This mode can be useful for speeding up guest installations, but you should switch to another caching mode in production environments.

For local or direct-attached storage, it is recommended that you use `writethrough` mode, as it ensures data integrity and has acceptable I/O performance for applications running in the guest, especially for read operations. However, caching mode `none` is recommended for remote NFS storage, because direct I/O operations (O_DIRECT) perform better than synchronous I/O operations (with O_SYNC). Caching mode `none` effectively turns all guest I/O operations into direct I/O operations on the host, which is the NFS client in this environment.

Best practice: I/O schedulers

Linux offers four I/O schedulers, or elevators: the NOOP I/O scheduler, the Anticipatory I/O scheduler, the Completely Fair Queuing (CFQ) I/O scheduler, and the Deadline I/O scheduler. Each I/O scheduler is effective in different scenarios.

NOOP I/O scheduler

The NOOP I/O scheduler is the simplest I/O scheduler. The NOOP I/O scheduler is a FIFO queue that provides basic merging and sorting functions to complete I/O requests. The NOOP I/O scheduler uses performance optimizations in the block device, HBA, or controller. The NOOP I/O scheduler is most effective in scenarios with small systems with slow disks.

Anticipatory I/O scheduler

The Anticipatory I/O scheduler changes the location of data to reduce disk seek operations by performing the following tasks:

- Combining and reordering I/O requests.
- Introducing a controlled delay before dispatching I/O requests.

The Anticipatory I/O scheduler improves disk I/O performance on systems with small or slow disks. The Anticipatory I/O scheduler has the possibility of long I/O latencies. Therefore, this I/O scheduler can be effective in scenarios with small client systems or small workstations where I/O latencies are less important than interactive response times.

CFQ I/O scheduler

The CFQ I/O scheduler is one of the most sophisticated I/O schedulers. The CFQ I/O scheduler divides the available I/O bandwidth among all of the processes that issue I/O requests. The CFQ I/O scheduler maintains per-process queues for synchronous I/O requests. The maximum number of per-process queues that the CFQ I/O scheduler maintains for synchronous I/O requests is 64. The CFQ I/O scheduler batches together asynchronous requests from all processes based on the priorities of the processes. For example, the CFQ I/O scheduler maintains one process queue for all asynchronous requests from processes with the `idle` scheduling priority.

During each cycle, the CFQ I/O scheduler moves one request from each queue to the dispatch queue. After the CFQ I/O scheduler moves a request from each queue, it repeats the process and removes another request from each queue. You can configure the number of requests that the CFQ I/O scheduler moves to the dispatch queue during each cycle by setting the **`quantum`** parameter. After the CFQ I/O scheduler moves requests to the dispatch queue, the CFQ I/O scheduler sorts the requests to minimize disk seeks. Then, the CFQ I/O scheduler services the requests accordingly.

The CFQ I/O scheduler provides each queue with time to access the disk. The length of time depends on the scheduling priority of the process. You can adjust the scheduling priority of a process by using the **ionice** parameter. For example:

```
ionice  -pprocessPID -cclass -npriority_within_a_class
```

where:

- *processPID* is the ID of the process.
- *class* is the scheduling priority of the process. The *class* value can be one of the following values:
  - 1: The scheduling priority of the process is idle, which is the lowest priority.
  - 2: The scheduling priority of the process is best effort, which is the default priority.
  - 3: The scheduling priority of the process is real time, which is the highest priority.
- *priority_within_a_class* is the scheduling priority within the best effort or real-time classes. The *priority_within_a_class* value can be integers 0 through 7, with 0 as the highest priority within a class and 7 as the lowest priority within a class.

The CFQ I/O scheduler is the default I/O scheduler for Red Hat Enterprise Linux 5 or later. To improve the performance of the CFQ I/O scheduler, you can use a predefined profile customized to your environment. For example, if you have a large enterprise storage environment, you can run the following command to customize the CFQ I/O scheduler profile:

```
# tuned-adm profile enterprise-storage
```

The profile includes adjusted parameters for the CFQ I/O scheduler that increase the performance of the CFQ I/O scheduler for environments with large enterprise storage.

The CFQ I/O scheduler is most effective in the following scenarios:

- Systems that require balanced I/O performance across several I/O controllers and LUNs
- Large enterprise systems

Deadline I/O scheduler

The Deadline I/O scheduler is one of the most sophisticated I/O schedulers. The Deadline I/O scheduler dispatches I/O requests based on the length of time that the I/O requests are in the queues. Therefore, the Deadline I/O scheduler guarantees a start service time for each I/O request.

The Deadline I/O scheduler maintains deadline queues and other queues as follows:

- The Deadline I/O scheduler sorts deadline queues by the expiration times, or deadlines, of the I/O requests.
- The Deadline I/O scheduler sorts the other queues by the positions of the requests on the disks, or sector numbers.

Each set of queues, deadline queues and other queues, includes read queues and write queues as follows:

- The read queues contain read requests. Because processes often block on read operations, the Deadline I/O scheduler performs the following actions on read requests:
  - Prioritizes read requests higher than write requests. However, you can control the number of read requests that the Deadline I/O scheduler dispatches before a write request.
  - Assigns read requests shorter expiration times than write requests. By default, the Deadline I/O scheduler assigns read requests the expiration time of 500 milliseconds.
- The write queues contain write requests. By default, the Deadline I/O scheduler assigns write requests the expiration time of 5 seconds.

The Deadline I/O scheduler works as follows:

1. Based on prioritization and expiration times, the Deadline I/O scheduler determines which request from which queue to dispatch.
2. The Deadline I/O scheduler checks to see if the first request in the deadline queue expired.
3. If the first request in the deadline queue expired, then the Deadline I/O scheduler performs the following tasks:
   a. The Deadline I/O scheduler services the first request in the deadline queue immediately.
   b. To improve disk efficiency, the Deadline I/O scheduler services a batch of requests near the disk location of the request from step 3a. This batch contains requests that follow the request (from step 3a) in the queue.
4. If the first request in the deadline queue is not expired, then the Deadline I/O Scheduler performs the following tasks:
   a. The Deadline I/O scheduler services the request from the sorted queue.
   b. The Deadline I/O scheduler services a batch of requests near the disk location of the request from step 4a. This batch contains requests that follow the request (from step 4a) in the sorted queue.

The Deadline I/O scheduler is the default Linux I/O scheduler on IBM mainframe systems.

The Deadline I/O scheduler is most effective in scenarios with large enterprise systems.

Related concepts:

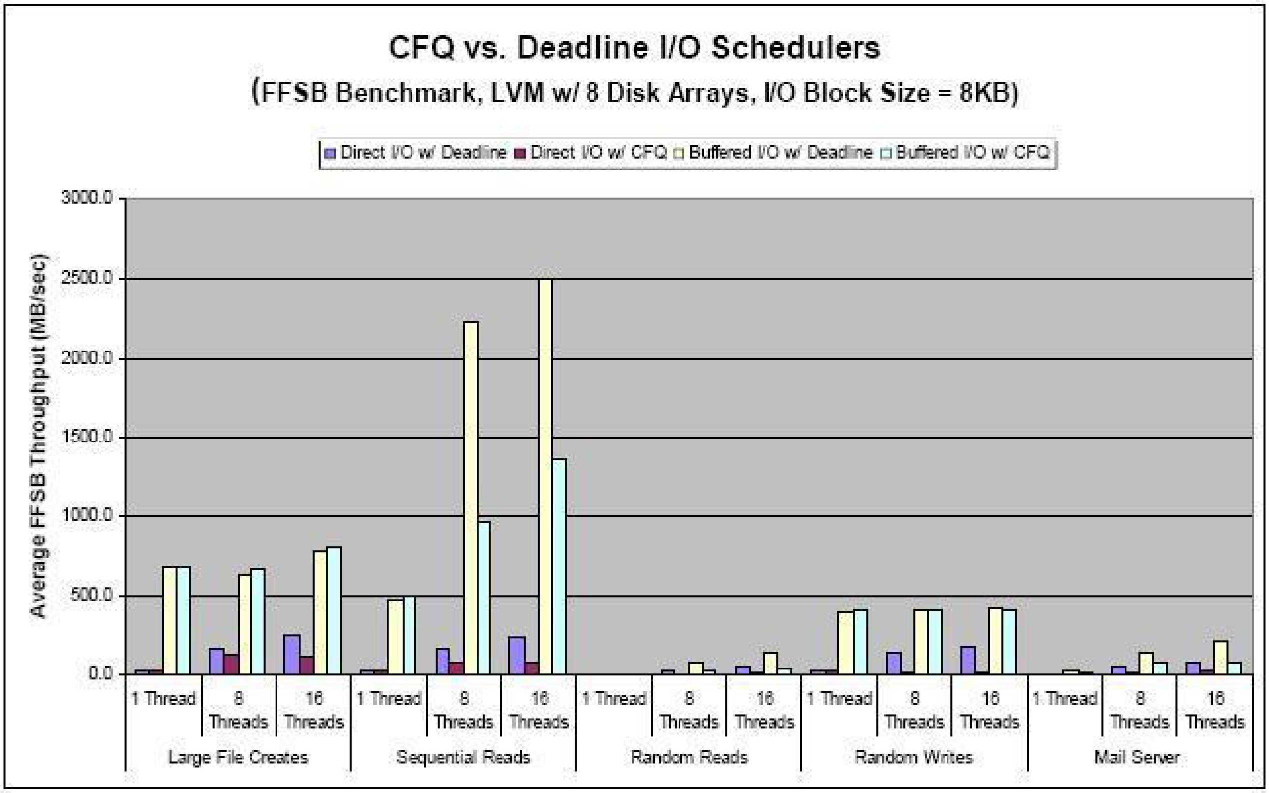
"Best practice: Optimize block I/O performance by using the Deadline I/O scheduler"
Based on lab test data, use the Deadline I/O scheduler on the KVM host and guest operating systems for I/O-bound workloads on enterprise storage systems.

"Best practice: If you cannot use the Deadline I/O scheduler, configure the CFQ I/O scheduler" on page 31
If you cannot use the Deadline I/O scheduler, you can use the Completely Fair Queuing (CFQ) I/O scheduler. You can configure the CFQ I/O scheduler to optimize block I/O performance.

Best practice: Optimize block I/O performance by using the Deadline I/O scheduler

Based on lab test data, use the Deadline I/O scheduler on the KVM host and guest operating systems for I/O-bound workloads on enterprise storage systems.

In a set of lab tests, the Flexible File System Benchmark (FFSB) program generated various I/O workloads on a large storage system. The large storage system included four IBM DS3400 controllers that were fiber-attached, eight RAID10 disk arrays, 192 disks, and an I/O block size of 8 KB. The following figure shows the FFSB throughput data for direct and buffered I/O with the Deadline and CFQ I/O schedulers:



The figure shows the average FFSB throughput in megabytes per second for each type of I/O paired with each type of I/O scheduler as follows:

- Direct I/O with the Deadline I/O scheduler
- Direct I/O with the CFQ I/O scheduler
- Buffered I/O with the Deadline I/O scheduler
- Buffered I/O with the CFQ I/O scheduler

For each combination of I/O type and I/O scheduler type, the figure shows throughput data for the following types of operations:

- Large file creation
- Sequential reads
- Random reads
- Random writes
- Mail server, which includes random file operations such as creating files, opening files, deleting files, random reads, random writes, and other random file operations. These random file operations were performed on 100,000 files in 100 directories with file sizes ranging from 1 KB to 1 MB.

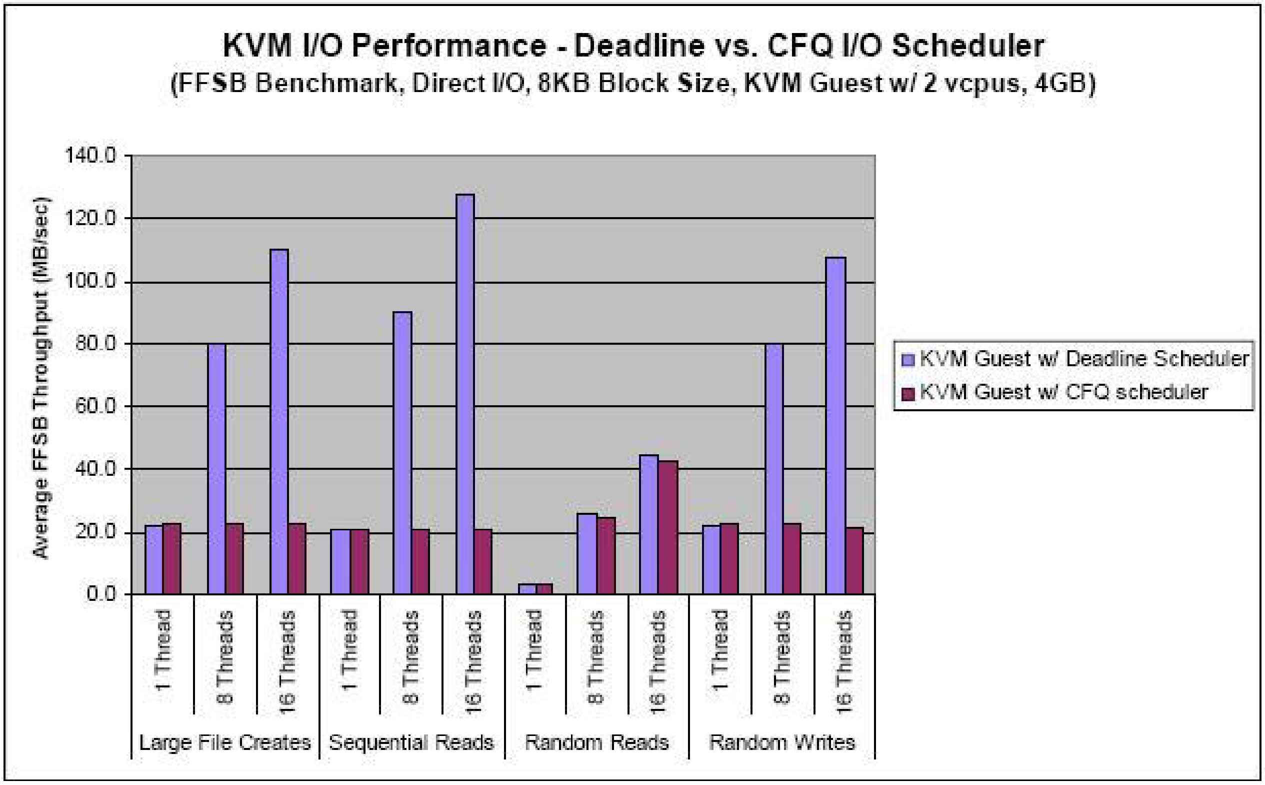
For each operation, the figure shows results for one thread, eight threads, and 16 threads.

For example, the figure shows the following average FFSB throughput for sequential reads on 16 threads:

- Direct I/O with the Deadline I/O scheduler: 237.0 MB/second
- Direct I/O with the CFQ I/O scheduler: 77.8 MB/second
- Buffered I/O with the Deadline I/O scheduler: 2498.6 MB/second
- Buffered I/O with the CFQ I/O scheduler: 1361.9 MB/second

Overall, the figure shows that the Deadline I/O scheduler outperforms the CFQ I/O scheduler, especially in multi-threaded scenarios.

In another set of lab tests, the FFSB program generated various I/O workloads on a guest operating system that used two virtual processors and 4 GB of memory. The I/O block size was 8 KB. The following figure shows the FFSB throughput data for direct I/O with the Deadline and CFQ I/O schedulers:



The figure shows the average FFSB throughput in megabytes per second for the guest operating system run with the Deadline I/O scheduler and for the guest operating system run with the CFQ I/O scheduler. For each I/O scheduler, the figure shows throughput data for the following types of operations:

- Large file creation
- Sequential reads
- Random reads
- Random writes

For each operation, the figure shows results for one thread, eight threads, and 16 threads.

For example, the figure shows the following average FFSB throughput for random writes on eight threads:

- Guest operating system with the Deadline I/O scheduler: 79.8 MB/second
- Guest operating system with the CFQ I/O scheduler: 23.1 MB/second

Overall, the figure shows that the Deadline I/O scheduler outperforms the CFQ I/O scheduler, especially in multi-threaded scenarios.

Related concepts:

"Best practice: I/O schedulers" on page 26
Linux offers four I/O schedulers, or elevators: the NOOP I/O scheduler, the Anticipatory I/O scheduler, the Completely Fair Queuing (CFQ) I/O scheduler, and the Deadline I/O scheduler. Each I/O scheduler is effective in different scenarios.

Best practice: If you cannot use the Deadline I/O scheduler, configure the CFQ I/O scheduler

If you cannot use the Deadline I/O scheduler, you can use the Completely Fair Queuing (CFQ) I/O scheduler. You can configure the CFQ I/O scheduler to optimize block I/O performance.

To improve the performance of the CFQ I/O scheduler in large storage environments, you can use the `enterprise-storage` profile on Red Hat Enterprise Linux 5 and 6. To activate this profile, run the following command:

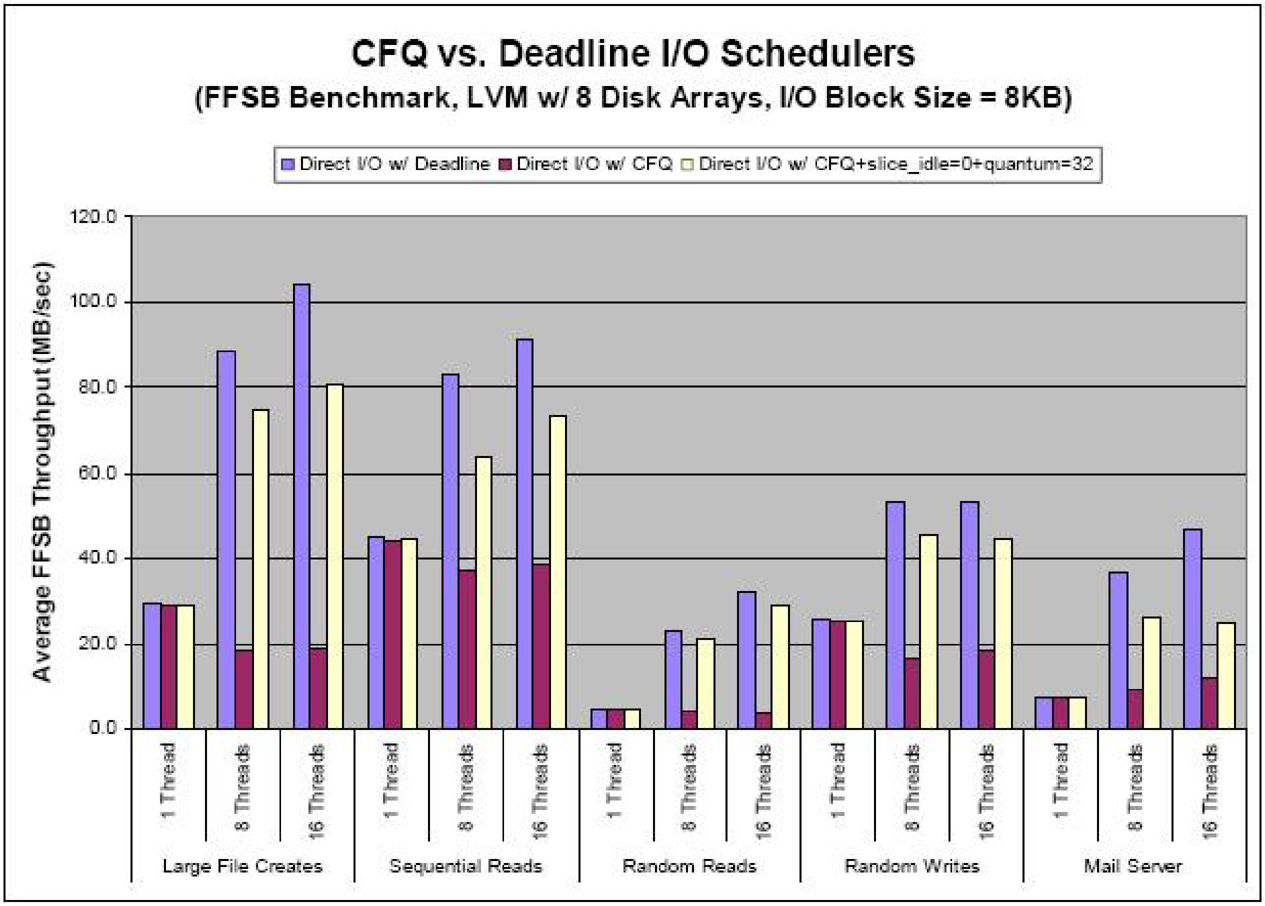
```
# tuned-adm profile enterprise-storage
```

The `enterprise-storage` profile sets the **`quantum`** and **`slice_idle`** parameters as follows:

```
/sys/block/device/queue/iosched/quantum = 32
/sys/block/device/queue/iosched/slice_idle = 0
```

where *device* is the name of an I/O device with which the CFQ scheduler interacts.

In a set of lab tests, the Flexible File System Benchmark (FFSB) program generated various I/O workloads on a large storage system. The large storage system included four IBM DS3400 controllers that were fiber-attached, eight RAID10 disk arrays, 192 disks, and an I/O block size of 8 KB. The following figure shows the FFSB throughput data for direct I/O with the Deadline and CFQ I/O schedulers:



The figure shows the average FFSB throughput in megabytes per second for direct I/O paired with the following types of I/O schedulers:

- Deadline I/O scheduler
- CFQ I/O scheduler with the default settings
- CFQ I/O scheduler with the **quantum** parameter set to 32 and the **slice_idle** parameter set to 0

For each I/O scheduler type, the figure shows throughput data for the following operations:

- Large file creation
- Sequential reads
- Random reads
- Random writes
- Mail server, which includes random file operations such as creating files, opening files, deleting files, random reads, random writes, and other random file operations. These random file operations were performed on 100,000 files in 100 directories with file sizes ranging from 1 KB to 1 MB.

For each operation, the figure shows results for one thread, eight threads, and 16 threads.

For example, the figure shows the following average FFSB throughput for large file creation on 16 threads:

- Deadline I/O scheduler: 104.0 MB/second
- CFQ I/O scheduler with the default settings: 18.9 MB/second
- CFQ I/O scheduler with the **quantum** parameter set to 32 and the **slice_idle** parameter set to 0: 80.9 MB/second

Overall, the figure shows the following results:

- The Deadline I/O scheduler outperforms the CFQ I/O scheduler, especially in multi-threaded scenarios.
- Changing the default settings of the CFQ I/O scheduler improves performance in large storage environments.

Related concepts:

"Best practice: I/O schedulers" on page 26
Linux offers four I/O schedulers, or elevators: the NOOP I/O scheduler, the Anticipatory I/O scheduler, the Completely Fair Queuing (CFQ) I/O scheduler, and the Deadline I/O scheduler. Each I/O scheduler is effective in different scenarios.

Best practices: Asynchronous I/O model for KVM guests

KVM supports thread pools for handling asynchronous I/O operations from guests for both Red Hat Enterprise Linux 5 and 6. However, in Red Hat Enterprise Linux 6, KVM guests can also take advantage of the Kernel Asynchronous I/O (AIO) support for Linux.

IBM lab tests show that using AIO support can improve guest I/O performance, especially when there are multiple threads performing I/O operations at the same time. You can enable AIO by specifying **aio=native** in the **qemu-kvm** command for a KVM guest.

Best practice: x2APIC support for Red Hat Enterprise Linux 6 guests

Red Hat Enterprise Linux 6 implements x2APIC emulation for KVM guests. Advanced programmable interrupt controllers (APICs) are used in symmetric multiprocessor (SMP) computer systems. x2APIC is a machine state register (MSR) interface to a local APIC with performance and scalability enhancements. IBM lab tests show that enabling the x2APIC support for Red Hat Enterprise Linux 6 guests can result in 2% to 5% throughput improvement for many I/O workloads. You can enable the x2APIC support by specifying the **-cpu qemu64,+x2apic** option on the **qemu-kvm** command for a KVM guest.

Best practice: NFS remote storage

When underlying storage is exported through NFS, if the NFS read and write windows sizes are not sufficiently large, I/O operations might be broken into smaller, unstable NFS I/O operations. Red Hat Enterprise Linux 6 includes support for automatically negotiating larger NFS read and write window sizes, such as 1 MB, to resolve this problem. In addition, Red Hat Enterprise Linux 6 contains fixes that reduce the number of **lseek** operations, which cause performance issues with NFS remote storage. For Red Hat Enterprise Linux 5, you can take advantage of these fixes by installing KVM package kvm-83-164.el5_5.15 or later.

For more efficient I/O handling, the latest versions of QEMU (the hardware emulator used by KVM) use vector I/O. Unfortunately, the NFS client does not provide support for vector I/O at this time. As a result, large I/O operations from KVM guests are broken into 4 KB I/O operations across NFS, negatively impacting the performance of large I/O operations. Red Hat issued an errata package (http://rhn.redhat.com/errata/RHBA-2011-1086.html) for Red Hat Enterprise Linux 6.1, which contains a workaround for this issue. A permanent solution for this issue is included in Red Hat Enterprise Linux 6.2 and later versions.

Best practice: Device-backed and file-backed virtual storage

In many virtualized environments, a virtual block device in a guest is really a file on the host file system. One advantage of this structure is that you can easily set up many virtual block devices for all guests running on the host. The disadvantage is that all I/O operations for a virtual block device are really going to a single file on the host, limiting performance. For better performance, you can pass the entire block device to the guest, assuming that no other guest is using that block device. IBM lab tests show that, on average, passing a block device directly to the guest results in the I/O performance in the guest reaching 90% or more of the bare metal host performance. These tests also show that, on average, passing a file on the host to the guest as a virtual disk results in only about 70% of host performance.

If you need to create partitions on virtual storage, you would need to ensure that partitions start on 4 KB boundary for optimal performance. To specify an optimal start value for a partition, you can switch the partition tool to the Advanced or Expert modes.

Best practices for improving security key strength

Review these general principles and best practices for improving security key strength based on your product type.

Many embedded systems, appliances, and web servers use public key cryptography, specifically in the SSL and HTTPS protocols, to secure communications. These protocols rely on a source of randomness (entropy) to generate large random prime numbers, which are the basis of public key cryptography. The lack of sufficient entropy in a system, especially in embedded systems, can cause either repeated or weak SSL and SSH keys to be generated. Communications or transactions encrypted using these keys then become vulnerable to attack.

In addition to general principles, review the best practices that most closely fit your environment and product.

Related concepts:

KVM security
You can protect and secure the Kernel-based Virtual Machine (KVM) environment by deploying KVM security features, such as configuring network isolation, securing storage devices, configuring secure remote management, isolating virtual machines with the sVirt service, preventing denial-of-service situations with control groups (cgroups), and protecting data at rest with disk-image encryption.

General principles for improving security key strength

To improve security key strength when developing applications, follow these general principles.

- Adhere to US Government Federal Information Processing Standards (FIPS) cryptography standards, such as NIST-131A, and stay up to date as the standards change over time.
- Document your entropy sources and the rationale of why the sources provide sufficient entropy.
  - Detail why each entropy source is unpredictable or cannot be influenced.
  - For libraries and applications, document what is expected from caller and environment.
  - For appliances and whole systems, perform an end-to-end analysis verifying that all contracts are matched with actual sources of entropy.
  - Finally, validate your design and rationale with a subject matter expert. This final step is important if you do not rely on standard and validated sources of entropy, such as a hardware TRNG (hardware random number generator) or verified, operating system-provided sources.
- Keep system software as up to date as practicable.
- Test entropy as part of your product release and regression testing. Such tests can be automated, and algorithms and software to test randomness are available for incorporation into your test suite. Entropy tests measure the quality of randomness and suggest an acceptable range based on Federal Information Processing Standard 140-2. Test code can be found at the US National Institute of Standards and Technology Computer Security Resource Center.

Best practice: Assess quantity of kernel entropy available just before generating keys

Assessing the quantity of kernel entropy available immediately before keys are generated helps to ensure randomness.

To check the Linux kernel entropy pool estimate, look at /proc/sys/kernel/random/entropy_avail after the first system boot, but before initial key generation. Then assess the quantity of kernel entropy available by comparing the estimated number against the recommended minimum entropy for the cryptographic mechanisms that you are using.

Best practice: Ensure that keys or entropy sources are not cloned

Ensure that manufacturing procedures do not clone keys or entropy sources.

You can prevent cloning by seeding, or initializing, entropy at manufacturing time. This practice makes entropy available on first boot, just before key generation. This recommendation is particularly applicable to embedded and virtual machines.

To seed entropy for Linux, initialize /dev/urandom with the saved entropy during the first boot before keys are generated.

The following is an example for a Fedora image on a Fedora host.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 36.

```
# Assume an existing Fedora disk image my.qcow2.
# Assume that the 2nd partition contains the root partition.

# Create image partition mapping
kpartx -av my.qcow2

# Mount the root partition assuming loop1p2 is the mapped root partition on the image.
mount /dev/mapper/loop1p2 /mnt

# Save random seed
touch /mnt/var/lib/random-seed
chmod 600 /mnt/var/lib/random-seed
dd if=/dev/urandom of=/mnt/var/lib/random-seed count=1 bs=512

# Unmount partition and remove mapping
umount /mnt
kpartx -dv my.qcow2
```

Best practice: Prevent or limit network access

Improve security by preventing Internet access or limiting network access, if possible.

You can limit opportunities for attackers to carry out attacks on a virtual machine by folllowing these suggestions:

- Prevent Internet access to systems by placing them on private networks.
- Limit network access to systems by using firewalls.

These recommendations are particularly applicable to embedded and virtual machines.

Best practice: Seed the kernel entropy pool with node-specific data

Seed, or initialize, the kernel entropy pool with node-specific data to improve key strength.

Node-specific data is data that is unique to the virtual machine instance, including the MAC address, hostname, and boot time. Seeding the kernel entropy pool with node-specific data is particularly effective if there is insufficient entropy.

To seed entropy for Linux, initialize /dev/urandom with node-specific data from a data source.

Note: Ensure that the data source contains node-specific data before using it.
For example, modify the system initialization script to include the following command:

```
cat /var/log/messages > /dev/urandom; ifconfig > /dev/urandom
```

Note: Adding device unique information to the PRNG (pseudorandom number generator) causes keys to be partially shared across devices. However, this method does not ensure enough entropy to withstand a targeted attack. In addition, it does not prevent reuse of randomness on the same device.

Best practice: Increase the kernel entropy pool by adding device event timing uncertainty

If you build your own kernel devices, you can increase the kernel entropy pool, and therefore increase entropy, by adding device event timing uncertainty.

To add device event timing uncertainty for Linux when building a kernel device, complete the following steps.

1. Add **`SA_SAMPLE_RANDOM`** to parameter 3 of request_irq() for selected drivers.
2. Rebuild the kernel.
3. Reassess the kernel entropy pool quantity and quality.

Note: Ensure that the additions do not adversely affect performance and that the entropy added is good quality.

Best practice: Generate keys just before they are required

If possible, generate keys just before they are required instead of during first boot.

Follow these suggestions for key generation:

- Remove SSL and SSH key generation from the system initialization script.
- Modify management applications to generate keys at a later time during the system configuration process, just before the system is placed on the network.

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Appendix. Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 903
11501 Burnet Road
Austin, TX 78758-3400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® and ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Printed in USA