Kernel Virtual Machine (KVM)

Tuning KVM for performance

Kernel Virtual Machine (KVM)

Tuning KVM for performance

Note

Before using this information and the product it supports, read the information in "Notices" on page 17.

Second Edition (April 2012)

Contents

Tuning KVM for performance

You can improve the performance of Kernel-based Virtual Machine (KVM) by pinning processors, setting the cache mode, configuring huge pages, and tuning memory settings.

Processor pinning

Workloads that run in the guest operating systems can benefit from processor pinning. Processor pinning is when you pin the virtual processors of a guest operating system to one or more physical processors on the host.

Without processor pinning, the system can dispatch virtual processors to any physical processor in the system. Processor pinning ensures that the system only dispatches a virtual processor to one or more specified physical processors. For example, you can pin a virtual processor to a single physical processor, to the physical processors on the same core, or to the physical processors on the same socket.

Benefits and drawbacks of processor pinning

Learn about the benefits and drawbacks of pinning virtual processors to physical processors, including cache-sharing and processor availability.

Benefits of processor pinning

You can pin a virtual processor to a set of physical processors that share the same cache. Thus, the virtual processor likely finds instructions and data in the cache.

For example, a guest operating system uses multiple virtual processors. You pin those virtual processors to physical processors that share the same cache. An application that runs in the guest operating system runs several threads that do similar work. The application often finds instructions and data in the cache for each of the threads because you pinned the virtual processors to physical processors that share the same cache.

In one set of lab tests, the SPECjbb2005 workload ran in a guest operating system of KVM. The guest operating system used 16 virtual processors on a system with 16 physical processors. The SPECjbb2005 workload was configured with four JVMs and each JVM used four virtual processors. One test iteration ran without virtual processor pinning. Another test iteration ran with the following processor pinning configuration:

- Each virtual processor was pinned to a single physical processor.
- The virtual processors that were used by one JVM were pinned to physical processors that shared the same level three cache.

The test iteration that used processor pinning resulted in a 16% performance improvement compared to the test iteration that did not use processor pinning.

You can guarantee processor availability to a guest operating system by using processor pinning. For example, you have a guest operating system that must run. However, the host system does not have enough physical processors to run all of the virtual processors of all the guest operating systems. You pin the virtual processors that the guest operating system uses to a subset of physical processors. You pin the virtual processors that the other guest operating systems use to the remainder of the physical processors. Therefore, the guest operating system always has the subset of physical processors available on which to run.

Drawbacks of processor pinning

You can lose the performance benefit of sharing the same cache when you pin virtual processors that run different tasks to the same physical processors. For example, you have a guest operating system. The virtual processors running in the guest operating system run different tasks. You pin the virtual processors to the same physical processors. When one of the virtual processors runs, it fills the processor caches. When the other virtual processor runs, it finds no data in the caches.

Sometimes you can decrease performance when you pin a virtual processor to a set of physical processors. One situation is when physical processors are idle because they are pinned to inactive workloads. For example, a system contains four physical processors. Guest operating system 1 uses one virtual processor to read the stock market ticker. Guest operating system 2 uses four virtual processors to run a web server. You want the stock market ticker to always run so that the system can respond to the stock market data in real time. Therefore, you pin the virtual processor that is used by Guest operating system 1 to one physical processor. You pin the four virtual processors that Guest operating system 2 uses to the remaining three physical processors. In this configuration, the stock market ticker always runs. However, the web server has fewer physical processors on which to run, which might decrease the performance of the web server. You might accept this performance risk when the stock market is open. However, when the stock market closes, the stock market ticker no longer needs to run. Therefore, the physical processor pinned to the stock market ticker is idle instead of providing additional resource to the web server.

Related concepts:

"Cache topology for processor pinning"
Learn about general guidelines for pinning virtual processors to physical processors that share the same cache.

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Related tasks:

"Pinning processors by cache level" on page 4
You can pin virtual processors to physical processors that share the same cache level.

"Pinning processors by common threads and cores" on page 6
You can pin one or more virtual processors to one or more physical processors that share the same processor threads or cores.

Cache topology for processor pinning

Learn about general guidelines for pinning virtual processors to physical processors that share the same cache.

You can pin processors at cache level two or cache level three. In general, physical processor threads on a core share level two cache, and cores on a socket share level three cache. However, this cache level sharing can vary by chip architecture.

Lower level caches are faster but smaller than higher level caches. If you pin too many virtual processors to physical processors with the same cache level, each virtual processor might fill the cache more quickly with its instructions and data. This situation negates any performance benefits gained by sharing the cache with other virtual processors.

In general, pin the same number of virtual processors to a cache level as the number of physical processors that share that cache level.

You can find a description of the cache topology for a physical processor in the `/sys/devices/system/cpu/cpu[n]/cache/` directory for the physical processor. The directory contains several subdirectories named `index[n]`. Each index directory contains information about one of the caches of the physical processor, including the files named `level`, `size`, and `shared_cpu_map`.

Related concepts:

"Benefits and drawbacks of processor pinning" on page 1
Learn about the benefits and drawbacks of pinning virtual processors to physical processors, including cache-sharing and processor availability.

Related tasks:

"Pinning processors by cache level" on page 4
You can pin virtual processors to physical processors that share the same cache level.

Processor topology for processor pinning

Learn about the IDs and bit masks for processor cores and processor threads.

The `/sys/devices/system/cpu/` directory describes the processor topology. The `/sys/devices/system/cpu/` directory contains a subdirectory for each processor, named `cpu#`, where # is the number of a processor. Each `cpu#` directory contains a subdirectory named `topology`. Each `topology` directory contains files with information about the topology of the processor. The following example shows the files in the `topology` directory that contain information about the `cpu0` processor:

```
# ls -1 /sys/devices/system/cpu/cpu0/topology/
core_id
core_siblings
physical_package_id
thread_siblings
```

The `topology` directory can contain one or more of the following files:

`core_id` file
: Contains the ID of the core on the physical package or socket of the core. The core ID distinguishes the core only on the socket of the core. Therefore, cores that are on separate sockets can have the same core ID. For example, the core ID of a processor on socket A is core ID 1. The core ID of a processor on socket B is also core ID 1.

`core_siblings` file
: Contains a bit mask of the processors that are on cores that are on the same physical package or socket. On some architectures, the bit mask is in the hexadecimal format. On other architectures, the bit mask is in the bit format. The leftmost bit of the bit mask is for the highest processor number. The rightmost bit of the bit mask is for processor `0`.

`core_siblings_list` file
: Contains a list of processor cores that are on the same physical package or socket. The list is in a human-readable format. For example, the `core_siblings_list` file shows the `0f0f` bit mask as `0-3,8-11`.

`thread_siblings` file
: Contains a bit mask of the processors that are on threads that are on the same core. The contents of the `thread_siblings` file is like the contents of the `core_siblings` file. The difference is that the bits in the `thread_siblings` file apply to threads instead of cores.

`thread_siblings_list` file
: Contains a list of processor threads that are on the same core. This list is in a human-readable format.

Related concepts:

"Benefits and drawbacks of processor pinning" on page 1
Learn about the benefits and drawbacks of pinning virtual processors to physical processors, including cache-sharing and processor availability.

Related tasks:

"Pinning processors by common threads and cores" on page 6
You can pin one or more virtual processors to one or more physical processors that share the same processor threads or cores.

Pinning processors by cache level

You can pin virtual processors to physical processors that share the same cache level.

To pin virtual processors to physical processors that share the same cache level, complete the following steps:

1. Determine which index directory holds the information for each cache level by typing the following commands:

   ```
   # for i in `ls /sys/devices/system/cpu/cpu#/cache/`; do
   > for f in level type size; do
   > echo -n "$i $f is "
   > cat /sys/devices/system/cpu/cpu#/cache/$i/$f
   > done
   > done
   ```

   where # is the number of the processor for which you want to find cache information. For example, `cpu0`. The output might look similar to the following output:

   ```
   index0 level is 1
   index0 type is Data
   index0 size is 32K
   index1 level is 1
   index1 type is Instruction
   index1 size is 32K
   index2 level is 2
   index2 type is Unified
   index2 size is 256K
   index3 level is 3
   index3 type is Unified
   index3 size is 8192K
   ```

   In this example, the `index2` directory contains information about the level two cache.

2. Determine which physical processors share the same cache level by typing the following commands:

   ```
   # for i in `seq 0 15`; do
   > echo -n "cpu$i "
   > cat /sys/devices/system/cpu/cpu$i/cache/index#/shared_cpu_map
   > done
   ```

   where # is the number of the index directory that contains information about the level two or level three cache. For example, `index2`. The output might look similar to the following output:

   ```
   cpu0 0101
   cpu1 0202
   cpu2 0404
   cpu3 0808
   cpu4 1010
   cpu5 2020
   cpu6 4040
   cpu7 8080
   cpu8 0101
   cpu9 0202
   cpu10 0404
   cpu11 0808
   ```

```
cpu12 1010
cpu13 2020
cpu14 4040
cpu15 8080
```

In this example, `cpu0` and `cpu8` share the same cache level. Also, `cpu1` and `cpu9` share the same cache level.

3. Pin the virtual processors of the guest operating system to the physical processors by typing the following command:

```
# virsh vcpupin guest# vproc# pproc#,pproc#
```

where:

- *guest#* is the name or ID of a guest operating system. This guest operating system uses the virtual processors that you want to pin to the physical processors.
- *vproc#* is the ID number of the virtual processor that you want to pin to one or more physical processors.
- *pproc#*, *pproc#* is a list of physical processors to which you want to pin a virtual processor.

  Remember: The **virsh** command does not accept processor ranges, such as `0-3`. Instead, you must enumerate each processor number individually. For example: `0,1,2,3`

For example:

```
# virsh vcpupin guest1 0 0,8
# virsh vcpupin guest1 1 0,8
# virsh vcpupin guest1 2 1,9
# virsh vcpupin guest1 3 1,9
```

In this example, `guest1` uses four virtual processors: `0`, `1`, `2`, and `3`. The example shows virtual processor `0` and virtual processor `1` each pinned to physical processor `0` and physical processor `8`. The example also shows virtual processor `2` and virtual processor `3` each pinned to physical processor `1` and physical processor `9`. The example shows the same number of virtual processors pinned to a cache level as the number of physical processors that share that cache level.

Example

The following example shows you how to pin virtual processors to physical processors that share the same level three cache:

1. Determine which index directory holds the information for the level three cache by typing the following commands:

```
# for i in `ls /sys/devices/system/cpu/cpu0/cache/`; do
> for f in level type size; do
> echo -n "$i $f is "
> cat /sys/devices/system/cpu/cpu0/cache/$i/$f
> done
> done
```

The following output shows that the `index3` directory contains information about the level three cache:

```
index0 level is 1
index0 type is Data
index0 size is 32K
index1 level is 1
index1 type is Instruction
index1 size is 32K
index2 level is 2
index2 type is Unified
```

```
index2 size is 256K
index3 level is 3
index3 type is Unified
index3 size is 8192K
```

2. Determine which physical processors share the same level three cache by typing the following commands:

```
# for i in `seq 0 15`; do
> echo -n "cpu$i "
> cat /sys/devices/system/cpu/cpu$i/cache/index3/shared_cpu_map
> done
```

   The following output shows that `cpu0`, `cpu1`, `cpu2`, `cpu3`, `cpu8`, `cpu9`, `cpu10`, and `cpu11` share a level three cache:

```
cpu0 0f0f
cpu1 0f0f
cpu2 0f0f
cpu3 0f0f
cpu4 f0f0
cpu5 f0f0
cpu6 f0f0
cpu7 f0f0
cpu8 0f0f
cpu9 0f0f
cpu10 0f0f
cpu11 0f0f
cpu12 f0f0
cpu13 f0f0
cpu14 f0f0
cpu15 f0f0
```

3. Pin the virtual processors of `guest1` to `cpu0`, `cpu1`, `cpu2`, `cpu3`, `cpu8`, `cpu9`, `cpu10`, and `cpu11` by typing the following commands:

```
# virsh vcpupin guest1 0 0,1,2,3,8,9,10,11
# virsh vcpupin guest1 1 0,1,2,3,8,9,10,11
# virsh vcpupin guest1 2 0,1,2,3,8,9,10,11
# virsh vcpupin guest1 3 0,1,2,3,8,9,10,11
```

Related concepts:

"Benefits and drawbacks of processor pinning" on page 1
Learn about the benefits and drawbacks of pinning virtual processors to physical processors, including cache-sharing and processor availability.

"Cache topology for processor pinning" on page 2
Learn about general guidelines for pinning virtual processors to physical processors that share the same cache.

Pinning processors by common threads and cores

You can pin one or more virtual processors to one or more physical processors that share the same processor threads or cores.

Related concepts:

"Benefits and drawbacks of processor pinning" on page 1
Learn about the benefits and drawbacks of pinning virtual processors to physical processors, including cache-sharing and processor availability.

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Pinning a virtual processor to the threads of a core

You can pin a virtual processor to processor threads that are on the same processor core.

Before you start, verify that you have a `thread_siblings_list` file. The **virsh** command uses processor numbers and not bit masks. If your architecture does not have a `thread_siblings_list` file, you must manually convert the bit mask from the `thread_siblings` file to processor numbers.

To pin a virtual processor to processor threads that are on the same processor core, complete the following steps:

1. List the processor threads that are on the same core of a physical processor by typing the following command:

   ```
   # cat /sys/devices/system/cpu/cpu#/topology/thread_siblings_list
   ```

   where # is the number of the processor that runs on the core that contains the threads. For example, `cpu3`. The output might look similar to the following output:

   ```
   3,11
   ```

   In this example, processor threads `3` and `11` are on the same core.
2. Pin the virtual processor of the guest operating system to the processor threads (that are on the same core) by typing the following command:

   ```
   # virsh vcpupin guest# vproc# thread_siblings
   ```

   where:

   - *guest#* is the name or ID of a guest operating system. This guest operating system uses the virtual processor that you want to pin to the processor threads.
   - *vproc#* is the ID number of the virtual processor that you want to pin to the processor threads.
   - *thread_siblings* are the processor threads to which you want to pin the virtual processor.

   For example:

   ```
   # virsh vcpupin guest1 4 3,11
   ```

   In this example, `guest1` uses virtual processor `4`. The example shows virtual processor `4` pinned to processor threads `3` and `11`.

Related concepts:

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Pinning a virtual processor to the cores of a socket

You can pin a virtual processor to processor cores that are on the same socket.

To pin a virtual processor to processor cores that are on the same socket, complete the following steps:

1. List the processor cores that are on the same socket by typing the following command:

   ```
   # cat /sys/devices/system/cpu/cpu#/topology/core_siblings_list
   ```

   where # is the number of the processor that runs on the cores on the socket. For example, `cpu3`. The output might look similar to the following output:

   ```
   0-3,8-11
   ```

   In this example, processor cores `0`, `1`, `2`, `3`, `8`, `9`, `10`, and `11` are on the same socket.
2. Pin the virtual processor of the guest operating system to the processor cores (that are on the same socket) by typing the following command:

   ```
   # virsh vcpupin guest# vproc# core_sibling,core_sibling
   ```

   where:

   - *guest#* is the name or ID of a guest operating system. This guest operating system uses the virtual processor that you want to pin to the processor cores.

- *vproc#* is the ID number of the virtual processor that you want to pin to the processor cores.
- *core_sibling,core_sibling* is a list of processor cores to which you want to pin the virtual processor.

  Remember: The **virsh** command does not accept processor ranges, such as `0-3`. Instead, you must enumerate each processor number individually. For example: `0,1,2,3`

For example:

```
# virsh vcpupin guest1 4 0,1,2,3,8,9,10,11
```

In this example, `guest1` uses virtual processor `4`. The example shows virtual processor `4` pinned to processor cores `0`, `1`, `2`, `3`, `8`, `9`, `10`, and `11`.

Related concepts:

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Pinning sets of virtual processors to sets of threads

You can pin a set of virtual processors to a set of processor threads that are on the same processor core.

Before you start, verify that you have a `thread_siblings_list` file. The **virsh** command uses processor numbers and not bit masks. If your architecture does not have a `thread_siblings_list` file, you must manually convert the bit mask from the `thread_siblings` file to processor numbers.

To pin a set of virtual processors to a set of processor threads that are on the same processor core, complete the following steps:

1. List the processor threads that are on the same cores of the physical processors by typing the following command:

   ```
   # for i in `seq 0 15`; do
   > echo -n "cpu$i "
   > cat /sys/devices/system/cpu/cpu$i/topology/thread_siblings_list
   > done
   ```

   The output might look similar to the following output:

   ```
   cpu0 0,8
   cpu1 1,9
   cpu2 2,10
   cpu3 3,11
   cpu4 4,12
   cpu5 5,13
   cpu6 6,14
   cpu7 7,15
   cpu8 0,8
   cpu9 1,9
   cpu10 2,10
   cpu11 3,11
   cpu12 4,12
   cpu13 5,13
   cpu14 6,14
   cpu15 7,15
   ```

   In this example, processor threads `0` and `8` are on the same core. Processors `cpu0` and `cpu8` share processor threads `0` and `8`. The example also shows that processor threads `1` and `9` are on the same core. Processors `cpu1` and `cpu9` share processor threads `1` and `9`.
2. Pin the virtual processors of the guest operating system to the processor threads (that are on the same core) by typing the following command:

   ```
   # virsh vcpupin guest# vproc# thread_siblings
   ```

   where:

- *guest#* is the name or ID of a guest operating system. This guest operating system uses the virtual processors that you want to pin to the processor threads.
- *vproc#* is the ID number of the virtual processor that you want to pin to the processor threads.
- *thread_siblings* are the processor threads to which you want to pin a virtual processor.

For example:

```
# virsh vcpupin guest1 0 0,8
# virsh vcpupin guest1 1 0,8
# virsh vcpupin guest1 2 1,9
# virsh vcpupin guest1 3 1,9
```

In this example, `guest1` uses four virtual processors: `0`, `1`, `2`, and `3`. The example shows virtual processors `0` and `1` each pinned to processor threads `0` and `8`. This example also shows virtual processors `2` and `3` each pinned to processor threads `1` and `9`.

Related concepts:

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Pinning sets of virtual processors to sets of cores

You can pin a set of virtual processors to a set of processor cores that are on the same socket.

To pin a set of virtual processors to a set of processor cores that are on the same socket, complete the following steps:

1. List the processor cores that are on the same sockets of the physical processors by typing the following command:

   ```
   # for i in `seq 0 15`; do
   > echo -n "cpu$i "
   > cat /sys/devices/system/cpu/cpu$i/topology/core_siblings_list
   > done
   ```

   The output might look similar to the following output:

   ```
   cpu0 0-3,8-11
   cpu1 0-3,8-11
   cpu2 0-3,8-11
   cpu3 0-3,8-11
   cpu4 4-7,12-15
   cpu5 4-7,12-15
   cpu6 4-7,12-15
   cpu7 4-7,12-15
   cpu8 0-3,8-11
   cpu9 0-3,8-11
   cpu10 0-3,8-11
   cpu11 0-3,8-11
   cpu12 4-7,12-15
   cpu13 4-7,12-15
   cpu14 4-7,12-15
   cpu15 4-7,12-15
   ```

   In this example, processor cores `0`, `1`, `2`, `3`, `8`, `9`, `10`, and `11` are on the same socket. Processors `cpu0`, `cpu1`, `cpu2`, `cpu3`, `cpu8`, `cpu9`, `cpu10`, and `cpu11` share processor cores `0-3` and `8-11`. Similarly, processor cores `4`, `5`, `6`, `7`, `12`, `13`, `14`, and `15` are on the same socket. Processors `cpu4`, `cpu5`, `cpu6`, `cpu7`, `cpu12`, `cpu13`, `cpu14`, and `cpu15` share processor cores `4-7` and `12-15`.

2. Pin the virtual processors of the guest operating system to the processor cores (that are on the same socket) by typing the following command:

   ```
   # virsh vcpupin guest# vproc# core_sibling,core_sibling
   ```

   where:

- *guest#* is the name or ID of a guest operating system. This guest operating system uses the virtual processors that you want to pin to the processor cores.
- *vproc#* is the ID number of the virtual processor that you want to pin to the processor cores.
- *core_sibling,core_sibling* is a list of processor cores to which you want to pin a virtual processor.

  Remember: The **virsh** command does not accept processor ranges, such as `0-3`. Instead, you must enumerate each processor number individually. For example: `0,1,2,3`

For example:

```
# virsh vcpupin guest1 0 0,1,2,3,8,9,10,11
# virsh vcpupin guest1 1 0,1,2,3,8,9,10,11
# virsh vcpupin guest1 2 4,5,6,7,12,13,14,15
# virsh vcpupin guest1 3 4,5,6,7,12,13,14,15
```

In this example, `guest1` uses four virtual processors: `0`, `1`, `2`, and `3`. The example shows virtual processors `0` and `1` each pinned to processor cores `0`, `1`, `2`, `3`, `8`, `9`, `10`, and `11`. This example also shows virtual processors `2` and `3` each pinned to processor cores `4`, `5`, `6`, `7`, `12`, `13`, `14`, and `15`.

Related concepts:

"Processor topology for processor pinning" on page 3
Learn about the IDs and bit masks for processor cores and processor threads.

Storage caching modes for guest operating systems

For the best storage performance on guest operating systems that use raw disk volumes or partitions, completely avoid the page cache on the host.

To turn off the storage caching mode of KVM, set `cache=none` in the **-drive** parameter. For example:

```
qemu-kvm -drive file=/dev/mapper/ImagesVolumeGroup-Guest1,cache=none,if=virtio
```

KVM also supports the writethrough caching mode and the writeback caching mode. The writethrough caching mode is the default caching mode for guest operating systems. The writethrough caching mode provides better data integrity than the writeback caching mode and uses the page cache on the host. The writeback caching mode is useful for testing, but does not guarantee storage data integrity. Therefore, do not use the writeback caching mode for guest operating systems in production environments.

Memory tuning for guest operating systems

To enhance performance, you can disable zone reclaim and set the `swappiness` value to zero for KVM hosts.

The `zone_reclaim_mode` value

Learn about zone reclaim and about the performance benefits of disabling zone reclaim in KVM environments.

The hardware of some operating systems has a Non Uniform Memory Architecture (NUMA) penalty for remote memory access. If the NUMA penalty of an operating system is high enough, the operating system performs zone reclaim.

For example, an operating system allocates memory to a NUMA node, but the NUMA node is full. In this case, the operating system reclaims memory for the local NUMA node rather than immediately allocating the memory to a remote NUMA node. The performance benefit of allocating memory to the local node outweighs the performance drawback of reclaiming the memory. However, in some situations reclaiming memory decreases performance to the extent that the opposite is true. In other words, in these situations, allocating memory to a remote NUMA node generates better performance than reclaiming memory for the local node.

A guest operating system can sometimes cause zone reclaim to occur when you pin memory. For example, a guest operating system causes zone reclaim in the following situations:

- When you configure the guest operating system to use huge pages.
- When you use Kernel same-page merging (KSM) to share memory pages between guest operating systems.

Configuring huge pages and running KSM are both best practices for KVM environments. Therefore, to optimize performance in KVM environments, disable zone reclaim.

Related concepts:

"Huge pages" on page 12
Learn about the performance benefits of huge pages and the Translation Lookaside Buffer (TLB).

Related tasks:

"Disabling zone reclaim"
You can disable zone reclaim for a KVM host.

Related information:

Kernel same-page merging (KSM)

The `swappiness` value

Learn about the `swappiness` value and about the performance benefits of setting the `swappiness` value as low as possible in KVM environments.

The default `swappiness` value is 60. The system accepts values between zero and 100. When you set the `swappiness` value to zero on Intel Nehalem systems, in most cases the virtual memory manager removes page cache and buffer cache rather than swapping out program memory. In KVM environments, program memory likely consists of a large amount of memory that the guest operating system uses.

Intel Nahalem systems use an extended page table (EPT). EPT does not set an access bit for pages that the guest operating systems uses. Therefore, the Linux virtual memory manager cannot use the least recently used (LRU) algorithm to accurately determine which pages are not needed. In other words, the LRU algorithm cannot accurately determine which pages, that back the memory of the guest operating systems, are the best candidates to swap out. In this situation, the best performance option is to avoid swapping for as long as possible by setting the `swappiness` value to zero.

Systems with Advanced Micro Devices (AMD) processors and nested page table (NPT) technology do not have this issue.

Related tasks:

"Setting the `swappiness` value to zero" on page 12
You can set the `swappiness` value to zero for a KVM host.

Related information:

Swapping

Disabling zone reclaim

You can disable zone reclaim for a KVM host.

To disable zone reclaim for a KVM host, complete the following steps:

1. View the status of the zone reclaim setting for a KVM host by typing the following command:

   ```
   cat /proc/sys/vm/zone_reclaim_mode
   ```

   If the value of `zone_reclaim_mode` is `0`, zone reclaim is disabled. If the value of `zone_reclaim_mode` is any other value, zone reclaim is enabled. For Red Hat Enterprise Linux version 6 and later, values other than `0` and `1` are allowed
2. Disable zone reclaim by modifying the system in one of the following ways:

- Disable zone reclaim for one instance (not persistent) by typing the following command:
  ```
  echo 0 > /proc/sys/vm/zone_reclaim_mode
  ```
- To permanently disable zone reclaim, complete the following steps:
  a. Edit the /etc/sysctl.conf file by adding the following information:
     ```
     vm.zone_reclaim_mode=0
     ```
     Every time you reboot the system, the system automatically uses the settings in the sysctl.conf file.
  b. Optional: To disable zone reclaim without rebooting, run the following command after you exit the sysctl.conf file:
     ```
     sysctl -p
     ```

Related concepts:

"The zone_reclaim_mode value" on page 10
Learn about zone reclaim and about the performance benefits of disabling zone reclaim in KVM environments.

Setting the swappiness value to zero

You can set the swappiness value to zero for a KVM host.

To set the swappiness value to zero for a KVM host, complete the following steps:

1. View the swappiness value for a KVM host by running the following command:
   ```
   cat /proc/sys/vm/swappiness
   ```
2. Set the swappiness value to zero by modifying the system in one of the following ways:
   - Set the swappiness value to zero for one instance (not persistent) by typing the following command:
     ```
     echo 0 > /proc/sys/vm/swappiness
     ```
   - To permanently set the swappiness value to zero, complete the following steps:
     a. Edit the /etc/sysctl.conf file by adding the following information:
        ```
        vm.swappiness=0
        ```
        Every time you reboot the system, the system automatically uses the settings in the sysctl.conf file.
     b. Optional: To set the swappiness value to zero without rebooting, run the following command after you exit the sysctl.conf file:
        ```
        sysctl -p
        ```

Related concepts:

"The swappiness value" on page 11
Learn about the swappiness value and about the performance benefits of setting the swappiness value as low as possible in KVM environments.

Huge page configuration

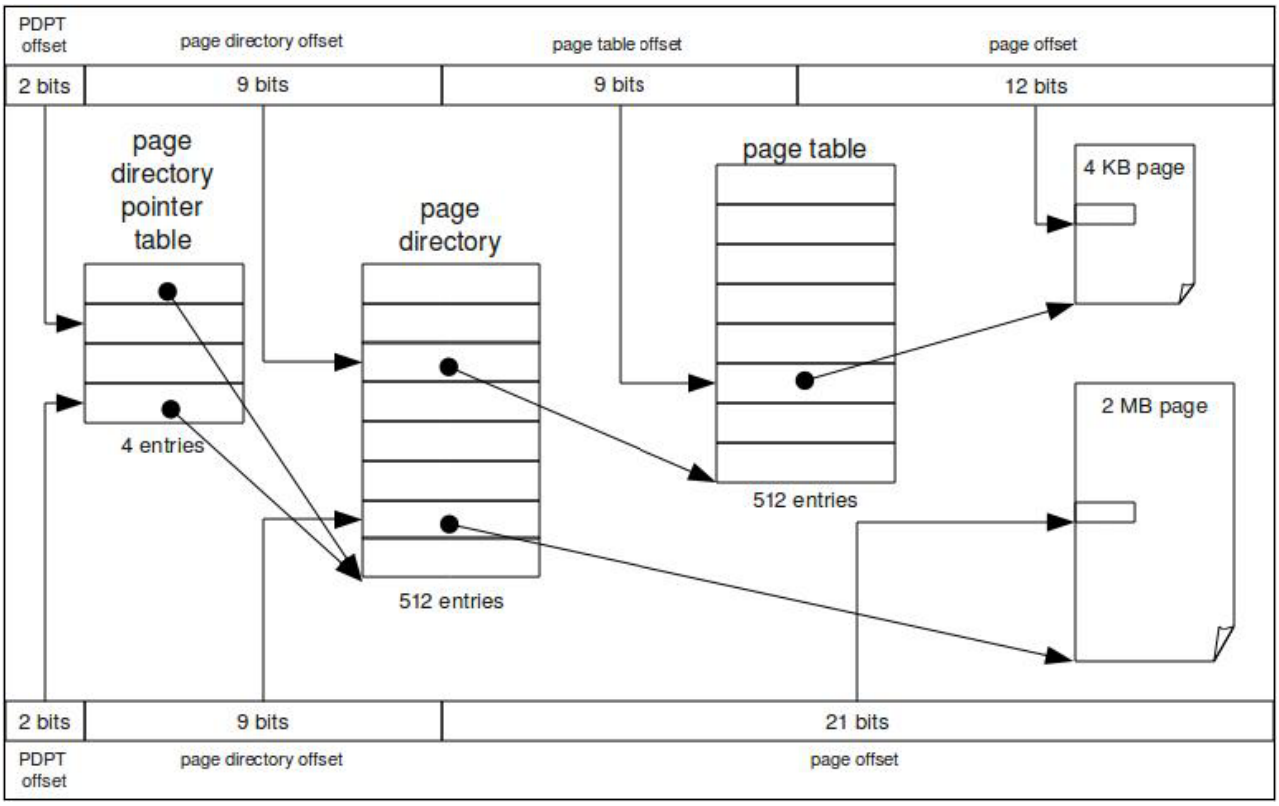
To increase performance, you can configure the KVM host to use huge pages.

Huge pages

Learn about the performance benefits of huge pages and the Translation Lookaside Buffer (TLB).

Huge pages are a feature of processor paging. Processors translate a virtual address to a physical address by separating the virtual address into bit segments. Then, the processors use the bit segments as indexes into the page tables at various levels to find the physical address. Huge pages use fewer page table levels in the address translation scheme.

The following figure shows how an i386 processor, with Physical Address Extension (PAE) enabled, translates a virtual address to a physical address:



The top half of the figure shows how the processor translates a virtual address to a physical address by using the normal, three-level, paging scheme:

1. First, the processor uses the top 2 bits of the virtual address as an index into the page directory pointer table (PDPT). The entry in the PDPT contains the base address of a page directory (PD).
2. Then, the processor uses the next upper 9 bits of the virtual address as an index into the PD. The entry in the PD contains the base address of a page table (PT).
3. Next, the processor uses the next upper 9 bits of the virtual address as an index into a PT. The entry in the PT contains the base address of the page.
4. Finally, the processor uses the lowest 12 bits of the virtual address as the offset into the 4 KB page.

The bottom half of the figure shows how the processor translates a virtual address to a physical address by using the two-level paging scheme for huge pages:

1. First, the processor uses the top 2 bits of the virtual address as an index into the PDPT. The entry in the PDPT contains the base address of a PD.
2. Then, the processor uses the next upper 9 bits of the virtual address as an index into the PD. The entry in the PD contains the base address of the page.
3. Finally, the processor uses the lowest 21 bits of the virtual address as the offset into the 2 MB page.

Different processor architectures use different paging schemes. Therefore, different processor architectures use different sizes for huge pages. For example, the Intel i386 and x86_64 processor architectures use 2 MB huge pages. The PowerPC® processor architecture uses 16 MB huge pages.

Translation Lookaside Buffer (TLB)

The TLB provides performance benefits for huge pages. The *TLB* is a cache that contains virtual-to-physical address mappings. When a processor first scans the page tables to find a physical address for a virtual address, the processor puts the result in the cache. The next time the processor needs the address, the processor finds the address in the cache instead of scanning the page tables again. The TLB provides a performance improvement because address references are often localized. In other words, the processor usually finds the mapping for a virtual address in the TLB.

However, the TLB has a maximum number of entries. A *TLB miss* occurs when a processor searches for a virtual address in the TLB and the TLB does not contain that virtual address. In this situation, the processor must scan the page table structure to find the physical address for the virtual address. Then, the processor removes an older mapping from the TLB so that the processor can store this new mapping in the TLB.

Each TLB entry contains a virtual address range the size of one page. When the page is a huge page, the address range of the TLB entry is larger than when the page is a standard page. For example, the address range of a TLB entry might be 4 KB for a standard page and 2 MB for a huge page. Therefore, the processor finds a mapping in the TLB more often for a huge page than for a standard page. The TLB contains a greater range of virtual addresses when several TLB entries contain virtual address ranges for huge pages rather than standard pages.

Applications that use large and discrete portions of memory benefit the most from huge pages because the applications generate fewer TLB misses. In other words, the processor finds a mapping in the TLB more often and scans the page table structure less often. Therefore, huge pages provide a performance improvement for applications that use large and discrete portions of memory. Applications that use many small and discrete portions of memory might not benefit from huge pages because these applications can waste memory if they use small portions of pages.

Configuring huge pages

Depending on the release, Red Hat Enterprise Linux can dynamically configure huge pages for you, or you can configure them yourself manually.

With Red Hat Enterprise Linux 6, the Linux kernel dynamically converts small adjacent pages into huge pages. The Linux kernel also dynamically converts huge pages into small pages when necessary. However, for Red Hat Enterprise Linux 5.5, you can configure huge pages manually by using the bash scripting language.

To configure huge pages for Red Hat Enterprise Linux 5.5, complete the following steps:

Note: The examples in the following procedure are written in the bash scripting language.

1. Determine the amount of memory to allocate as huge pages. For example:

   ```
   HUGE_SIZE_MB=2048
   ```

   Tip: Use units of megabytes because huge page sizes are in multiples of megabytes.
2. Identify the size of a huge page by viewing the `Hugepagesize` entry in the `/proc/meminfo` file. For example:

   ```
   HUGE_PAGE_SIZE_KB=`grep Hugepagesize /proc/meminfo | awk '{print $2}'`
   HUGE_PAGE_SIZE_MB=$((HUGE_PAGE_SIZE_KB/1024))
   ```

   In this example, the page size is in KB.
3. Calculate the number of huge pages that you need. For example:

   ```
   HUGE_PAGES_NEEDED=$(((HUGE_SIZE_MB+HUGE_PAGE_SIZE_MB-1)/HUGE_PAGE_SIZE_MB))
   ```

In this example, you divide the amount of huge page memory that you need by the size of a huge page.

4. Allocate the memory as huge pages by writing the number of huge pages to the `/proc/sys/vm/nr_hugepages` file. For example:

```
echo $HUGE_PAGES_NEEDED >/proc/sys/vm/nr_hugepages
HUGE_PAGES_ALLOCATED=`cat /proc/sys/vm/nr_hugepages`
```

5. Set the maximum size of any discrete portion of shared memory by writing the size of the huge page memory, in bytes, to the `/proc/sys/kernel/shmmax` file: For example:

```
SHMMAX_NEEDED=$((HUGE_PAGES_ALLOCATED*HUGE_PAGE_SIZE_KB*1024))
echo $SHMMAX_NEEDED > /proc/sys/kernel/shmmax
```

6. Set the total amount of shared memory that you want to allocate:
   a. Identify the size of a page, not a huge page, by using the **getconf** command. For example:

```
PAGE_SIZE=`getconf PAGE_SIZE`
```

   b. Write the number of pages, not huge pages, of the huge page memory to the `/proc/sys/kernel/shmall` file. For example:

```
SHMALL_NEEDED=$((SHMMAX_NEEDED/PAGE_SIZE))
echo $SHMALL_NEEDED > /proc/sys/kernel/shmall
```

7. Mount the `hugetlbfs` file system by typing the following commands:

```
mkdir -p /libhugetlbfs
mount -t hugetlbfs hugetlbfs /libhugetlbfs
```

8. Set QEMU to use huge pages by using the **-mem-path qemu** option. For example:

```
/usr/libexec/qemu-kvm ... -mem-path /libhugetlbfs ...
```

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