Improving performance

This article provides information on basic system diagnostics relating to performance as well as steps that may be taken to reduce resource consumption or to otherwise optimize the system with the end-goal being either perceived or documented improvements to a system's performance.

Related articles

Improving performance/Boot process

Pacman/Tips and tricks#Performance

SSH#Speeding up SSH

Openoffice#Speed up OpenOffice

Laptop

Preload

Cpulimit

Contents [hide]

The basics

Know your system

The best way to tune a system is to target bottlenecks, or subsystems which limit overall speed. The system specifications can help identify them.

- If the computer becomes slow when large applications (such as OpenOffice.org and Firefox) run at the same time, check if the amount of RAM is sufficient. Use the following command, and check the "available" column:

```
$ free -h
```

- If boot time is slow, and applications take a long time to load at first launch (only), then the hard drive is likely to blame. The speed of a hard drive can be measured with the `hdparm` command:

Note: **hdparm** indicates only the pure read speed of a hard drive, and is not a valid benchmark. A value higher than 40MB/s (while idle) is however acceptable on an average system.

```
# hdparm -t /dev/sdX
```

- If CPU load is consistently high even with enough RAM available, then try to lower CPU usage by disabling running **daemons** and/or processes. This can be monitored in several ways, for example with **htop**, `pstree` or any other **system monitoring** tool:

```
$ htop
```

- If applications using direct rendering are slow (i.e those which use the GPU, such as video players, games, or even a **window manager**), then improving GPU performance should help. The first step is to verify if direct rendering is actually enabled. This is indicated by the `glxinfo` command, part of the **mesa-demos** package:

```
$ glxinfo | grep direct
```

```
direct rendering: Yes
```

- When running a **desktop environment**, disabling (unused) visual desktop effects may reduce GPU usage. Use a more lightweight environment or create a **custom environment** if the current does not meet the hardware and/or personal requirements.

Benchmarking

The effects of optimization are often difficult to judge. They can however be measured by **benchmarking** tools.

Storage devices

Multiple hardware paths

This article or section needs language, wiki syntax or style improvements. See Help:Style for reference.

Reason: Subjective writing (Discuss in **Talk:Improving performance#**)

An internal hardware path is how the storage device is connected to your motherboard. There are different ways to connect to the motherboard such as TCP/IP through a NIC, plugged in directly using PCIe/PCI, Firewire, Raid Card, USB, etc. By spreading your storage devices across these multiple connection points you maximize the capabilities of your motherboard, for example 6 hard-drives connected via USB would be much much slower than 3 over USB and 3 over Firewire. The reason is that each entry path into the motherboard is like a pipe, and there is a set limit to how much can go through that pipe at any one time. The good news is that the motherboard usually has several pipes.

More Examples

1. Directly to the motherboard using pci/PCIe/ata
2. Using an external enclosure to house the disk over USB/Firewire
3. Turn the device into a network storage device by connecting over tcp/ip

Note also that if you have a 2 USB ports on the front of your machine, and 4 USB ports on the back, and you have 4 disks, it would probably be fastest to put 2 on front/2 on back or 3 on back/1 on front. This is because internally the front ports are likely a separate Root

Hub than the back, meaning you can send twice as much data by using both than just 1. Use the following commands to determine the various paths on your machine.

USB Device Tree

```
$ lsusb -tv
```

PCI Device Tree

```
$ lspci -tv
```

Partitioning

Make sure that your partitions are **properly aligned**.

Multiple drives

If you have multiple disks available, you can set them up as a software **RAID** for serious speed improvements.

Creating **swap** on a separate disk can also help quite a bit, especially if your machine swaps frequently.

Layout on HDDs

If using a traditional spinning HDD, your partition layout can influence the system's performance. Sectors at the beginning of the drive (closer to the outside of the disk) are faster than those at the end. Also, a smaller partition requires less movements from the drive's head, and so speed up disk operations. Therefore, it is advised to create a small partition (10GB, more or less depending on your needs) only for your system, as near to the beginning of the drive as possible. Other data (pictures, videos) should be kept on a separate partition, and this is usually achieved by separating the home directory (`/home/user`) from the system (`/`).

Choosing and tuning your filesystem

Choosing the best filesystem for a specific system is very important because each has its own strengths. The **File systems** article provides a short summary of the most popular ones. You can also find relevant articles in **Category:File systems**.

Mount options

The **noatime** option is known to improve performance of the filesystem.

Other mount options are filesystem specific, therefore see the relevant articles for the filesystems:

- **Ext3**
- **Ext4#Improving performance**
- **JFS Filesystem#Optimizations**
- **XFS**
- **Btrfs#Defragmentation**, **Btrfs#Compression**, and `btrfs(5)`
- **ZFS#Tuning**

Reiserfs

The `data=writeback` mount option improves speed, but may corrupt data during power loss. The `notail` mount option increases the space used by the filesystem by about 5%, but also improves overall speed. You can also reduce disk load by putting the journal and data on separate drives. This is done when creating the filesystem:

```
# mkreiserfs –j /dev/sda1 /dev/sdb1
```

Replace `/dev/sda1` with the partition reserved for the journal, and `/dev/sdb1` with the partition for data. You can learn more about reiserfs with this **article**.

Tuning kernel parameters

There are several key tunables affecting the performance of block devices, see **sysctl#Virtual memory** for more information.

Input/output schedulers

Background information

The input/output *(I/O)* scheduler is the kernel component that decides in which order the block I/O operations are submitted to storage devices. It is useful to remind here some specifications of two main drive types because the goal of the I/O scheduler is to optimize the way these are able to deal with read requests:

- An HDD has spinning disks and a head that moves physically to the required location. Therefore, random latency is quite high ranging between 3 and 12ms (whether it is a high end server drive or a laptop drive and bypassing the disk controller write buffer) while sequential access provides much higher throughput. The typical HDD throughput is about 200 I/O operations per second *(IOPS)*.

- An SSD does not have moving parts, random access is as fast as sequential one, typically under 0.1ms, and it can handle multiple concurrent requests. The typical SSD throughput is greater than 10,000 IOPS, which is more than needed in common workload situations.

If there are many processes making I/O requests to different storage parts, thousands of IOPS can be generated while a typical HDD can handle only about 200 IOPS. There is a queue of requests that have to wait for access to the storage. This is where the I/O schedulers plays an optimization role.

The scheduling algorithms

One way to improve throughput is to linearize access: by ordering waiting requests by their logical address and grouping the closest ones. Historically this was the first Linux I/O scheduler called **elevator**.

One issue with the elevator algorithm is that it is not optimal for a process doing sequential access: reading a block of data, processing it for several microseconds then reading next block and so on. The elevator scheduler does not know that the process is about to read another block nearby and, thus, moves to another request at some other location. The **anticipatory** I/O scheduler overcomes the problem: it pauses for a few milliseconds in anticipation of another close-by read operation before dealing with another request.

While these schedulers try to improve total throughput, they might leave some unlucky requests waiting for a very long time. As an example, imagine the majority of processes make requests at the beginning of the storage space while an unlucky process makes a request at the other end of storage. This potentially infinite postponement of the process is called starvation. To improve fairness, the **deadline** algorithm was developed. It has a queue ordered by address, similar to the elevator, but if some request sits in this queue for too long then it moves to an "expired" queue ordered by expire time. The scheduler checks the expire queue first and processes requests from there and only then moves to the elevator queue. Note that this fairness has a negative impact on overall throughput.

The **Completely Fair Queuing *(CFQ)*** approaches the problem differently by allocating a timeslice and a number of allowed requests by queue depending on the priority of the process submitting them. It supports **cgroup** that allows to reserve some amount of I/O to a specific collection of processes. It is in particular useful for shared and cloud hosting: users who paid for some IOPS want to get their share whenever needed. Also, it idles at the end of synchronous I/O waiting for other nearby operations, taking over this feature from the *anticipatory* scheduler and bringing some enhancements. Both the *anticipatory* and the *elevator* schedulers were decommissioned from the Linux kernel replaced by the more advanced alternatives presented above.

The **Budget Fair Queuing *(BFQ)*** is based on CFQ code and brings some enhancements. It does not grant the disk to each process for a fixed time-slice but assigns a "budget" measured in number of sectors to the process and uses heuristics. It is a relatively complex scheduler, it may be more adapted to rotational drives and slow SSDs because its high per-operation overhead, especially if associated with a slow CPU, can slow down fast devices. The objective of BFQ on personal systems is that for interactive tasks, the storage device is virtually as responsive as if it was idle. In its default configuration it focuses on delivering the lowest latency rather than achieving the maximum throughput.

Kyber is a recent scheduler inspired by active queue management techniques used for network routing. The implementation is based on "tokens" that serve as a mechanism for limiting requests. A queuing token is required to allocate a request, this is used to prevent starvation of requests. A dispatch token is also needed and limits the operations of a certain priority on a given device. Finally, a target read latency is defined and the scheduler tunes itself to reach this latency goal. The implementation of the algorithm is relatively simple and it is deemed efficient for fast devices.

Kernel's I/O schedulers

While some of the early algorithms have now been decommissioned, the official Linux kernel supports a number of I/O schedulers which can be split into two categories:

- The **single-queue schedulers** are available by default with the kernel:
  - **NOOP** is the simplest scheduler, it inserts all incoming I/O requests into a simple FIFO queue and implements request merging. In this algorithm, there is no re-ordering of the request based on the sector number. Therefore it can be used if the ordering is dealt with at another layer, at the device level for example, or if it does not matter, for SSDs for instance.
  - *Deadline*
  - *CFQ*
- The **multi-queue scheduler** mode can be activated at boot time as described in **#Changing I/O scheduler**. This **Multi-Queue Block I/O Queuing Mechanism *(blk-mq)*** maps I/O queries to multiple queues, the tasks are distributed across threads and therefore CPU cores. Within this framework the following schedulers are available:
  - *None*, no queuing algorithm is applied.
  - *mq-deadline* is the adaptation of the deadline scheduler to multi-threading.
  - *Kyber*
  - *BFQ*

  Warning: The multi-queue scheduler framework and its related algorithms are under active development, the state of some issues can be seen in the **bfq forum** and **FS#57496**. In particular, users reported USB drives to stop working - **[1][2][3]**.

Note: The best choice of scheduler depends on both the device and the exact nature of the workload. Also, the throughput in MB/s is not the only measure of performance: deadline or fairness deteriorate the overall throughput but improve system responsiveness.

Changing I/O scheduler

Note: The block multi-queue *(blk-mq)* mode must be enabled at boot time to be able to access the latest *BFQ* and *Kyber* schedulers. This is done by adding `scsi_mod.use_blk_mq=1` to the **kernel parameters**. The single-queue schedulers are no longer available once in this mode.

To see the available schedulers for a device and the active one, in brackets:

```
$ cat /sys/block/sda/queue/scheduler
```

```
mq-deadline kyber [bfq] none
```

or for all devices:

```
$ cat /sys/block/sd*/queue/scheduler
```

```
mq-deadline kyber [bfq] none
[mq-deadline] kyber bfq none
mq-deadline kyber [bfq] none
```

To change the active I/O scheduler to *bfq* for device *sda*, use:

```
# echo bfq > /sys/block/sda/queue/scheduler
```

SSDs can handle many IOPS and tend to perform best with simple algorithm like *noop* or *deadline* while *BFQ* is well adapted to HDDs. The process to change I/O scheduler, depending on whether the disk is rotating or not can be automated and persist across reboots. For example the **udev** rule below sets the scheduler to *mq-deadline* for non-rotational drives, it covers the naming schemes for SATA SSD, eMMC and NVMe SSD and to *bfq* for SATA HDD.

```
/etc/udev/rules.d/60-ioschedulers.rules
```

```
# set scheduler for non-rotating disks
ACTION=="add|change", KERNEL=="sd[a-z]|mmcblk[0-9]*|nvme[0-9]*", ATTR{queue/rotational}=="0", ATTR{queue/scheduler}="mq-
deadline"
# set scheduler for rotating disks
ACTION=="add|change", KERNEL=="sd[a-z]", ATTR{queue/rotational}=="1", ATTR{queue/scheduler}="bfq"
```

Then reboot or force **udev#Loading new rules**.

Tuning I/O scheduler

Each of the kernel's I/O scheduler has its own tunables, such as the latency time, the expiry time or the FIFO parameters. They are helpful in adjusting the algorithm to a particular combination of device and workload. This is typically to achieve a higher throughput or a lower latency for a given utilization. The tunables and their description can be found within the **kernel documentation files**.

To list the available tunables for a device, in the example below *sdb* which is using *deadline*, use:

```
$ ls /sys/block/sdb/queue/iosched

fifo_batch  front_merges  read_expire  write_expire  writes_starved
```

To improve *deadline*'s throughput at the cost of latency, one can increase `fifo_batch` with the command:

```
# echo 32 > /sys/block/sdb/queue/iosched/fifo_batch
```

Power management configuration

When dealing with traditional rotational disks (HDD's) you may want to **lower or disable power saving features** completely.

Reduce disk reads/writes

Avoiding unnecessary access to slow storage drives is good for performance and also increasing lifetime of the devices, although on modern hardware the difference in life expectancy is usually negligible.

Note: A 32GB SSD with a mediocre 10x write amplification factor, a standard 10000 write/erase cycle, and **10GB of data written per day**, would get an **8 years life expectancy**. It gets better with bigger SSDs and modern controllers with less write amplification. Also compare **[4]** when considering whether any particular strategy to limit disk writes is actually needed.

Show disk writes

The `iotop` package can sort by disk writes, and show how much and how frequently programs are writing to the disk. See `iotop(8)` for details.

Relocate files to tmpfs

Relocate files, such as your browser profile, to a **tmpfs** file system, for improvements in application response as all the files are now stored in RAM:

- Refer to **Profile-sync-daemon** for syncing browser profiles. Certain browsers might need special attention, see e.g. **Firefox on RAM**.
- Refer to **Anything-sync-daemon** for syncing any specified folder.
- Refer to **Makepkg#Improving compile times** for improving compile times when building packages.

Compiling in tmpfs

See **Makepkg#Building from files in memory**.

Optimize the filesystem

Filesystems may provide performance improvements instructions for each filesystem, e.g. **Ext4#Improving performance**.

Swap space

See **Swap#Performance**.

Storage I/O scheduling with ionice

Many tasks such as backups do not rely on a short storage I/O delay or high storage I/O bandwidth to fulfil their task, they can be classified as background tasks. On the other hand quick I/O is necessary for good UI responsiveness on the desktop. Therefore it is beneficial to reduce the amount of storage bandwidth available to background tasks, whilst other tasks are in need of storage I/O. This can be achieved by making use of the linux I/O scheduler CFQ, which allows setting different priorities for processes.

The I/O priority of a background process can be reduced to the "Idle" level by starting it with

```
# ionice -c 3 command
```

See ionice(1) and [5] for more information.

CPU

Overclocking

Overclocking improves the computational performance of the CPU by increasing its peak clock frequency. The ability to overclock depends on the combination of CPU model and motherboard model. It is most frequently done through the BIOS. Overclocking also has disadvantages and risks. It is neither recommended nor discouraged here.

Many Intel chips will not correctly report their clock frequency to acpi_cpufreq and most other utilities. This will result in excessive messages in dmesg, which can be avoided by unloading and blacklisting the kernel module `acpi_cpufreq`. To read their clock speed use *i7z* from the i7z package. To check for correct operation of an overclocked CPU, it is recommended to do **stress testing**.

Frequency scaling

See **CPU frequency scaling**.

Alternative CPU scheduler

This article or section needs expansion.

Reason: MuQSS is not the only alternative scheduler. (Discuss in **Talk:Improving performance#**)

The default CPU scheduler in the mainline Linux kernel is **CFS**.

An alternative scheduler designed to be used on desktop computers is MuQSS, developed by **Con Kolivas**, which is focused on desktop interactivity and responsiveness. MuQSS is available either as a stand-alone patch or as part of a wider patchset, the **-ck** patchset. See **Linux-ck** and **Linux-pf** for more information on the patchset.

Real-time kernel

Some applications such as running a TV tuner card at full HD resolution (1080p) may benefit from using a **realtime kernel**.

Adjusting priorities of processes

See also `nice(1)` and `renice(1)`.

Ananicy

Ananicy is a daemon, available in the `ananicy-git`AUR package, for auto adjusting the nice levels of executables. The nice level represents the priority of the executable when allocating CPU resources.

cgroups

See **cgroups**.

Cpulimit

Cpulimit is a program to limit the CPU usage percentage of a specific process. After installing `cpulimit`, you may limit the CPU usage of a processes' PID using a scale of 0 to 100 times the number of CPU cores that the computer has. For example, with eight CPU cores the precentage range will be 0 to 800. Usage:

```
$ cpulimit -l 50 -p 5081
```

irqbalance

The purpose of `irqbalance` is distribute hardware interrupts across processors on a multiprocessor system in order to increase performance. It can be **controlled** by the provided `irqbalance.service`.

Graphics

Xorg configuration

Graphics performance may depend on the settings in `xorg.conf(5)`; see the **NVIDIA**, **ATI** and **Intel** articles. Improper settings may stop Xorg from working, so caution is advised.

Mesa configuration

The performance of the Mesa drivers can be configured via **drirc**. GUI configuration tools are available:

- **adriconf (Advanced DRI Configurator)** — Tool to set options and configure applications using the standard drirc file used by the Mesa drivers.
  https://github.com/jlHertel/adriconf/ || adriconfAUR
- **DRIconf** — Configuration applet for the Direct Rendering Infrastructure. It allows customizing performance and visual quality settings of OpenGL drivers on a per-driver, per-screen and/or per-application level.
  https://dri.freedesktop.org/wiki/DriConf/ || driconf

Overclocking

As with CPUs, overclocking can directly improve performance, but is generally recommended against. There are several packages in the **AUR**, such as amdoverdrivectrlAUR (ATI) and nvclockAUR (NVIDIA).

See **AMDGPU#Overclocking** or **NVIDIA/Tips and tricks#Enabling overclocking**.

RAM and swap

Clock frequency and timings

RAM can run at different clock frequencies and timings, which can be configured in the BIOS. Memory performance depends on both values. Selecting the highest preset presented by the BIOS usually improves the performance over the default setting. Note that increasing the frequency to values not supported by both motherboard and RAM vendor is overclocking, and similar risks and disadvantages apply, see **#Overclocking**.

Root on RAM overlay

If running off a slow writing medium (USB, spinning HDDs) and storage requirements are low, the root may be run on a RAM overlay ontop of read only root (on disk). This can vastly improve performance at the cost of a limited writable space to root. See liverootAUR.

Zram or zswap

The **zram** kernel module (previously called **compcache**) provides a compressed block device in RAM. If you use it as swap device, the RAM can hold much more information but uses more CPU. Still, it is much quicker than swapping to a hard drive. If a system often falls back to swap, this could improve responsiveness. Using zram is also a good way to reduce disk read/write cycles due to swap on SSDs.

Similar benefits (at similar costs) can be achieved using **zswap** rather than zram. The two are generally similar in intent although not operation: zswap operates as a compressed RAM cache and neither requires (nor permits) extensive userspace configuration.

Example: To set up one lz4 compressed zram device with 32GiB capacity and a higher-than-normal priority (only for the current session):

```
# modprobe zram
# echo lz4 > /sys/block/zram0/comp_algorithm
# echo 32G > /sys/block/zram0/disksize
# mkswap --label zram0 /dev/zram0
# swapon --priority 100 /dev/zram0
```

To disable it again, either reboot or run

```
# swapoff /dev/zram0
# rmmod zram
```

A detailed explanation of all steps, options and potential problems is provided in the official documentation of the module **here**.

The **`systemd-swap`** package provides a `systemd-swap.service` unit to automatically initialize zram devices. Configuration is possible in `/etc/systemd/swap.conf`.

The package **`zramswap`**AUR provides an automated script for setting up such swap devices with optimal settings for your system (such as RAM size and CPU core number). The script creates one zram device per CPU core with a total space equivalent to the RAM available, so you will have a compressed swap with higher priority than regular swap, which will utilize multiple CPU cores for compressing data. To do this automatically on every boot, **enable** `zramswap.service`.

Swap on zRAM using a udev rule

The example below describes how to set up swap on zRAM automatically at boot with a single udev rule. No extra package should be needed to make this work.

First, enable the module:

```
/etc/modules-load.d/zram.conf
```

```
zram
```

Configure the number of /dev/zram nodes you need.

```
/etc/modprobe.d/zram.conf

options zram num_devices=2
```

Create the udev rule as shown in the example.

```
/etc/udev/rules.d/99-zram.rules

KERNEL=="zram0", ATTR{disksize}="512M" RUN="/usr/bin/mkswap /dev/zram0", TAG+="systemd"
KERNEL=="zram1", ATTR{disksize}="512M" RUN="/usr/bin/mkswap /dev/zram1", TAG+="systemd"
```

Add /dev/zram to your fstab.

```
/etc/fstab

/dev/zram0 none swap defaults 0 0
/dev/zram1 none swap defaults 0 0
```

Using the graphic card's RAM

In the unlikely case that you have very little RAM and a surplus of video RAM, you can use the latter as swap. See **Swap on video RAM**.

Network

- Kernel networking: see **Sysctl#Improving performance**
- NIC: see **Network configuration#Set device MTU and queue length**
- DNS: consider using a caching DNS resolver, see **Domain name resolution#Resolvers**
- Samba: see **Samba#Improve throughput**

Watchdogs

According to **wikipedia:Watchdog_timer**:

> A watchdog timer [...] is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly resets the watchdog timer [...]. If, [...], the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal [...] used to initiate corrective [...] actions [...] typically include placing the computer system in a safe state and restoring normal system operation.

Many users need this feature due to their system's mission-critical role (i.e. servers), or because of the lack of power reset (i.e. embedded devices). Thus, this feature is required for a good operation in some situations. On the other hand, normal users (i.e. desktop and laptop) do not need this feature and can disable it.

To disable watchdog timers (both software and hardware), append `nowatchdog` to your boot parameters.

To check the new configuration do:

```
# cat /proc/sys/kernel/watchdog
```

or use:

```
# wdctl
```

After you disabled watchdogs, you can *optionally* avoid the loading of the module responsible of the hardware watchdog, too. Do it by **blacklisting** the related module, e.g. `iTCO_wdt`.

Note: Some users **reported** the `nowatchdog` parameter does not work as expected but they have successfully disabled the watchdog (at least the hardware one) by blacklisting the above-mentioned module.

Either action will speed up your boot and shutdown, because one less module is loaded. Additionally disabling watchdog timers increases performance and **lowers power consumption**.

See **[6]**, **[7]**, **[8]**, and **[9]** for more information.

See also

- **Red Hat Performance Tuning Guide**
- **Linux Performance Measurements using vmstat**

Categories: Hardware | System administration