ADMIN:DOCS

Linux Administration #9: System Monitoring and Performance

Linux performance monitoring and system administration: CPU and memory analysis, I/O metrics with iostat/vmstat, log diagnostics via journalctl, network monitoring, and automated alerting pipelines.

Category: Linux administration **Published:** 12.07.2025 **Last reviewed:** 12.07.2025

You already learned in Article #8 of our Linux administration series how to manage virtual machines with KVM, run Docker containers, and set up LXC containers. Now you might be wondering: "How do I actually tell whether my system or my virtual machines are running properly? How do I notice when something is wrong?"

That is exactly where **system monitoring** comes in -- an absolutely essential area of Linux administration that you as an aspiring administrator must master.

> **Important note:** This article is aimed at technology-interested Linux beginners, learners with the goal of understanding and implementing Linux administration, and readers who value practical, understandable, and thoroughly explained content.

Why is monitoring so important?

Imagine you manage a Linux server in a company. Suddenly colleagues complain that the website is loading slowly. Or even worse: the system stops responding entirely.

Without the right monitoring knowledge you are facing a puzzle:

- Is the CPU overloaded?

- Is the RAM full?
- Is a process blocking the disk?
- Is there a network problem?

> **You will need this later to:** work professionally as a Linux administrator, solve problems before they become critical, and quickly find the right cause during outages.

How this article is structured

We start with the **fundamentals of system resources** -- you will learn what CPU load, memory usage, and I/O metrics really mean. Then we work through the most important monitoring tools with practical examples and typical use cases.

In the second part of the article I will show you how to systematically analyze log files and set up automated monitoring. Finally we will solve realistic performance problems together and build a simple monitoring dashboard.

Practical focus: Every section contains concrete examples that you can try directly on your system. You will not just learn the theory but apply everything hands-on.

> **You will need this later to:** pass the LPIC-1 exam, impress in your job as a Linux administrator, and manage your own systems professionally.

Understanding system resources

Interpreting CPU load correctly

CPU load is probably the first metric you will monitor as a Linux administrator. But beware: this is where the most misinterpretations happen!

What do the load average values really mean?

When you enter the `uptime` command you see three mysterious numbers:

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

```
uptime
14:23:45 up 2 days,  3:42,  2 users,  load average: 0.15, 0.25, 0.30
```

> ⚠ **Typical misinterpretation:** Many people think load 1.0 always means 100% utilization. That is only true for a single CPU core!

□□ Practical example:

You have a quad-core processor (4 cores). Load 1.0 therefore means that one process is using 100% on one CPU core. Load 2.0 means that one process is using 100% on two CPU cores.

This means:

- Load 4.0 = 100% utilization of all cores
- Load 2.0 = 50% utilization of all cores
- Load 8.0 = 200% utilization = system overloaded!

```
nproc
4

cat /proc/cpuinfo | grep "processor" | wc -l
4
```

> **□□ You will need this later to:** understand when your system is truly overloaded. A load of 3.0 is completely normal on an 8-core system but critical on a single-core system.

Difference between user, system, and idle time

With the `top` command you see a detailed breakdown of CPU time:

```
top
...
%Cpu(s):  12.5 us,  3.1 sy,  0.0 ni, 84.2 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
```

What do these values mean?

Abbreviation	Meaning	Explanation
us	User	Time for user programs (Firefox, LibreOffice, etc.)
sy	System	Time for kernel operations (filesystem, network, etc.)
ni	Nice	Time for processes with lower priority
id	Idle	CPU is waiting and has nothing to do
wa	I/O Wait	CPU is waiting for disk or network
hi	Hardware IRQ	Time for hardware interrupts
si	Software IRQ	Time for software interrupts
st	Steal	Time that other VMs "stole" (only in virtualization)

□□ Practical example:

Your system is slow and you see:

```
%Cpu(s):   5.2 us,  2.1 sy,  0.0 ni, 12.5 id, 80.2 wa,  0.0 hi,  0.0 si,  0.0 st
```

Diagnosis: High wa value (80.2%) means the CPU is constantly waiting for the disk. The problem is not the CPU but an I/O bottleneck!

> **Common mistake:** Many people only look at overall CPU load and overlook that the real problem is with the disk.

> **You will need this later to:** diagnose performance problems correctly. High CPU load can have very different causes.

Practical example:

Analyzing `cat /proc/loadavg`

Besides `uptime` you can also read the load average values directly from the `/proc` filesystem:

```
cat /proc/loadavg
0.15 0.25 0.30 2/267 12345
```

What do these numbers mean?

- `0.15` = load of the last 1 minute
- `0.25` = load of the last 5 minutes
- `0.30` = load of the last 15 minutes

- `2/267` = 2 running processes out of 267 total processes
- `12345` = PID of the last started process

> **You will need this later to:** use load values in your own monitoring scripts:

```
#!/bin/bash
# Simple load monitoring script
LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
CPU_CORES=$(nproc)
LOAD_PERCENT=$(echo "scale=2; $LOAD_1MIN / $CPU_CORES * 100" | bc)

echo "Current load: $LOAD_1MIN ($LOAD_PERCENT% of $CPU_CORES cores)"

if (( $(echo "$LOAD_1MIN > $CPU_CORES" | bc -l) )); then
    echo "WARNING: System overloaded!"
fi
```

> ⚠ **Typical misinterpretation:** Many people think load 1.0 always means 100% utilization. That is only true for a single CPU core! With 4 cores load 4.0 = 100% utilization.

Deciphering memory usage

This is where the most common monitoring mistake of all happens: Linux beginners think their system has no free memory left, even though everything is completely normal.

Why Linux memory always looks "full"

Look at this typical output of `free -h`:

```
free -h
               total        used        free      shared  buff/cache   available
Mem:           7.7Gi       2.3Gi       1.2Gi       228Mi       4.2Gi       4.6Gi
Swap:          2.0Gi          0B       2.0Gi
```

Panic reaction of a beginner: "Help! Only 1.2 GB free out of 7.7 GB!"

> ⚠ **That is completely wrong!** Linux uses "unused" memory intelligently as cache for files and programs. This makes your system faster.

The correct interpretation:

- **total:** Total RAM (7.7 GB)
- **used:** Currently used by programs (2.1 GB)
- **free:** Completely unused memory (1.2 GB)
- **buff/cache:** Memory used as cache (4.4 GB) -- will be released immediately when needed!
- **available:** Actually available memory (5.1 GB) -- this is the important value!

> □□ **Key takeaway:** The available value shows you how much memory is really available. In our example that is 5.1 GB -- completely sufficient!

□□ Practical example:

You start a memory-hungry program.

Linux automatically frees cache memory:

```
# Before starting a large program
free -h
               total        used        free      shared  buff/cache   available
Mem:           7.7Gi       2.1Gi       1.2Gi       234Mi       4.4Gi       5.1Gi

# After starting
free -h
               total        used        free      shared  buff/cache   available
Mem:           7.7Gi       4.8Gi       0.3Gi       234Mi       2.6Gi       2.4Gi
```

> □□ **What happened?** The program needed 2.7 GB of memory. Linux automatically released 1.8 GB from the cache. Everything works perfectly!

Difference between "used", "free", "available" and "cached"

For deeper analysis look into `/proc/meminfo`:

- cat /proc/meminfo
- MemTotal: 8052748 kB
- MemFree: 1234567 kB
- MemAvailable: 5234567 kB
- Buffers: 123456 kB
- Cached: 4567890 kB
- SwapCached: 0 kB
- Active: 3456789 kB
- Inactive: 2345678 kB
- Dirty: 12345 kB
- Writeback: 0 kB
- Slab: 345678 kB

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

The most important values explained:

Value	Meaning	When critical?
MemTotal	Total RAM	--
MemFree	Completely unused RAM	Irrelevant for evaluation
MemAvailable	Available RAM (including releasable cache)	< 10% = critical
Buffers	Buffers for filesystem metadata	--
Cached	Cache for file contents	Will be released automatically
SwapCached	Swap memory in RAM cache	0 = system is swapping
Active	Recently used memory	--
Inactive	Memory not used for a longer time	--
Dirty	Changed data, not yet written to disk	100MB = I/O congestion
Writeback	Data is currently being written	0 = I/O running
Slab	Kernel data structures	500MB = unusual

 Typical mistake: Beginners add Cached to MemFree and think that would be the available memory. That is not correct! Always use MemAvailable.

 You will need this later to: recognize memory leaks, diagnose memory bottlenecks, and understand when your system truly has too little RAM.

Understanding disk space and I/O

Disk monitoring is complex because two completely different problems can occur: **lack of space and performance issues.**

Inodes vs. space -- both can be "full"

A common problem that confuses beginners: You get the error message `No space left on device` even though `df` still shows free disk space.

□□ Practical example:

Let us look at both values:

```
# Check disk space
df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        20G   15G  4.2G  79% /

# Check inodes
df -i /
Filesystem      Inodes   IUsed   IFree IUse% Mounted on
/dev/sda1      1310720 1310720       0  100% /
```

Spot the problem? Disk space is still there `4.2 GB free`, but the inodes are 100% used up!

What are inodes?:

- Every file needs an inode (index node)
- Inodes store metadata: permissions, timestamps, location
- The number of inodes is set when formatting
- Many small files = many inodes consumed

> ⚠ **Typical scenario:** Log files are rotated and millions of small files are created. Disk space is still there but no inodes are free.

Finding a solution:

```
# Find directory with the most files
find /var/log -type f | wc -l
1234567

# Clean up old log files
find /var/log -name "*.log.*" -mtime +30 -delete
```

> **You will need this later to:** understand and solve mysterious "Disk full" errors even when space appears to be available.

What do the different I/O metrics mean?

With `iostat` you get detailed information about disk performance:

```
iostat -x 1
Device            r/s     w/s     rkB/s     wkB/s   rrqm/s   wrqm/s  %util
sda              12.34   45.67    123.45    456.78     0.12     2.34   85.6
```

Every column explained:

Column	Meaning	Critical value
r/s	Read operations per second	> 200 on HDD
w/s	Write operations per second	> 100 on HDD
rkB/s	Kilobytes read per second	--
wkB/s	Kilobytes written per second	--

Column	Meaning	Critical value
rrqm/s	Read requests merged	Low = bad
wrqm/s	Write requests merged	Low = bad
%util	Disk utilization	> 80% = bottleneck

□□ Practical example:

Your system is slow and `iostat` shows:

```
Device            r/s     w/s     rkB/s     wkB/s   %util
sda             234.5    12.3   2345.67    123.45   98.7
```

Diagnosis: Very high read rate (234.5 r/s) and 98.7% utilization. The disk is the bottleneck!

Finding the cause:

```
# Which process is causing the most I/O?
iotop -o
```

> ⚠ **Common mistake:** Only looking at transfer rate (rkB/s, wkB/s) and ignoring the number of operations (r/s, w/s). Many small accesses can block a disk just as much as few large ones.

> **You will need this later to:** identify I/O bottlenecks, optimize slow applications, and decide whether you need an SSD.

Practical example:

Disk full but `df` still shows space

```
# You get this error message:
touch /tmp/testfile
touch: cannot touch '/tmp/testfile': No space left on device

# But df still shows space:
df -h /tmp
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        10G   8.5G  1.2G  88% /tmp

# The problem: inodes are full!
df -i /tmp
Filesystem      Inodes   IUsed   IFree IUse% Mounted on
/dev/sda1      655360  655360       0  100% /tmp
```

Solution: Delete old small files to free inodes:

```
# Find directory with the most files
find /tmp -type f | head -1000 | xargs ls -la

# Delete old temporary files
find /tmp -name "*.tmp" -mtime +7 -delete
```

Practical rule:

```
# Warning at 90% usage
df -h | awk '$5 > 90 {print "WARNING: " $1 " is " $5 " full!"}'

# Critical at 95% usage
df -h | awk '$5 > 95 {print "CRITICAL: " $1 " is " $5 " full!"}'
```


```
# Warning at 90% usage
df -h | awk '$5 > 90 {print "WARNING: " $1 " is " $5 " full!"}'

# Critical at 95% usage
df -h | awk '$5 > 95 {print "CRITICAL: " $1 " is " $5 " full!"}'
```


> **You will need this later to:** avoid system outages from full disks and prevent performance problems.

Mastering important monitoring commands

`top` is probably the first monitoring command you learn as a Linux administrator. It shows you in real time what is happening on your system. But most beginners only use a fraction of its capabilities.

Understanding every value in the `top` output

Start `top` and let us look at the output together:

```
top
top - 14:23:45 up 2 days,  3:42,  2 users,  load average: 0.15, 0.25, 0.30
Tasks: 267 total,   1 running, 266 sleeping,   0 stopped,   0 zombie
%Cpu(s):  12.5 us,  3.1 sy,  0.0 ni, 84.2 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7876.2 total,   1234.5 free,   2345.6 used,   4296.1 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   5123.4 avail Mem

□PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1234 user      20   0  123456  45678  12345 S  25.3   0.6   1:23.45 firefox
   5678 root      20   0   67890  23456   7890 R  12.1   0.3   0:45.67 python3
   9012 user      20   0   34567  12345   4567 S   5.2   0.2   0:12.34 chrome
```

Header line explained line by line:

Line 1 -- System time and load:

- top - 14:23:45 up 2 days, 3:42, 2 users, load average: 0.15, 0.25, 0.30
- 14:23:45 = Current time
- up 2 days, 3:42 = System has been running for 2 days and 3:42 hours
- 2 users = Two users are logged in
- load average: 0.15, 0.25, 0.30 = Average load of the last 1, 5, 15 minutes

Line 2 -- Process status:

- Tasks: 267 total, 1 running, 266 sleeping, 0 stopped, 0 zombie

```
┌ 267 total = 267 processes total
```

- 1 running = One process is currently running (usually top itself)
- 266 sleeping = 266 processes are waiting for events

- 0 stopped = No stopped processes (Ctrl+Z)
- zombie = No zombie processes (good!)

> ⚠ Watch out for zombie processes: If values here are > 0 you have a problem. Zombie processes are "dead" processes that were not properly terminated.

Line 3 -- CPU utilization:

```
%Cpu(s):  12.5 us,  3.1 sy,  0.0 ni, 84.2 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
```

Line 4+5 -- Memory information:

- MiB Mem : 7876.2 total, 1234.5 free, 2345.6 used, 4296.1 buff/cache
- MiB Swap: 2048.0 total, 2048.0 free, 0.0 used. 5123.4 avail Mem

- 7876.2 total = Total RAM in MiB
- 1234.5 free = Completely free RAM
- 2345.6 used = RAM used by programs
- 4296.1 buff/cache = RAM used as cache
- 5123.4 avail Mem = Most important value: available RAM

> **You will need this later to:** quickly see whether your system has enough memory. When avail Mem drops below 10% of total RAM it becomes critical.

Interactive commands in `top` (k, r, M, P, etc.)

`top` is not just a passive display tool -- you can control it interactively:

The most important keyboard shortcuts:

Key	Function	When to use?
M	Sort by memory usage	Find memory hogs
P	Sort by CPU usage	Identify CPU hogs
T	Sort by runtime	Find long-running processes
k	Kill process	Stop hanging processes
r	Change process priority (renice)	Prioritize important processes
1	Show all CPU cores individually	Check multi-core utilization
c	Show full command line	See parameters of processes
q	Quit `top`	--

Practical exercise: Identify memory hogs

Start `top`:

- top

Press **M** to sort by memory usage:

```
PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1234 user      20   0 2345678 567890 123456 S   5.3  7.2   12:34.56 firefox
5678 user      20   0 1234567 234567  67890 S   2.1  3.0    5:67.89 chrome
9012 root      20   0  567890 123456  45678 S   0.5  1.6    1:23.45 systemd
```

Press **c** to see the full command line:

- COMMAND
- firefox --new-window https://admindocs.de
- chrome --disable-extensions --incognito

- /lib/systemd/systemd --switched-root --system

What do the columns mean?

Column	Meaning	Critical values
PID	Process ID	--
USER	User who started the process	--
PR	Priority (lower = more important)	< 0 = realtime
NI	Nice value (-20 to +19)	-20 = highest priority
VIRT	Virtual memory (can be larger than RAM)	--
RES	Resident memory (actually in RAM)	**Most important value!**
SHR	Shared memory (shared with other processes)	--
S	Status (R=running, S=sleeping, Z=zombie)	Z = problem!
%CPU	CPU usage in percent	> 100% with multi-core
%MEM	RAM usage in percent	> 10% = noticeable

> ⚠ **Typical mistake:** Many people look at the VIRT column and get scared. Firefox shows 2.3 GB virtual memory? That is normal! Only RES matters -- the actually used RAM.

`htop` -- The user-friendly alternative

`htop` is a modern alternative to `top` with a better interface:

```
# Install htop (if not present)
$ sudo apt install htop
```

```
# Debian/Ubuntu
$ sudo dnf install htop

# Fedora/RHEL
# Start htop
$ htop
```

Advantages of htop:

- Color display for better overview
- Mouse support
- Easier navigation with arrow keys
- Process tree view `F5`
- Built-in help `F1`

□□ Practical example:

Kill a hanging process in `htop`:

- Start htop
- Navigate with arrow keys to the problematic process
- Press `F9` (Kill)
- Choose the signal (usually `SIGTERM` or `SIGKILL`)
- Confirm with `Enter`

> ⚠ **Warning:** Use SIGKILL (signal 9) only as a last resort. It ends processes immediately without cleaning up. Try SIGTERM (signal 15) first.

> □□ **You will need this later to:** identify and safely kill hanging processes, find CPU and memory leaks, and monitor system load.

`iostat` -- Analyzing disk performance

`iostat` is your most important tool for analyzing disk performance. It belongs to the `sysstat` package which you may need to install separately.

Installation and basics

```
# Install sysstat (contains iostat)
sudo apt install sysstat

# Debian/Ubuntu
sudo dnf install sysstat

# Fedora/RHEL
# Simple iostat call
iostat
Linux 5.15.0 (hostname)    07/11/2025    _x86_64_    (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
□□   12.34    0.12    3.45    2.34    0.00   81.75

Device             tps    kB_read/s    kB_read    kB_wrtn/s    kB_wrtn
sda              15.67       123.45    1234567       456.78    4567890
```

Every column of the `iostat` output explained

For detailed analysis use `iostat -x`:

```
iostat -x 1 5

# -x = extended statistics
# 1  = update every second
# 5  = 5 measurements, then stop
```

Understanding the output:

```
Device            r/s     w/s     rkB/s     wkB/s   rrqm/s   wrqm/s  %rrqm  %wrqm
avgrq-sz    avgqu-sz     await     r_await     w_await     svctm     %util
sda              12.34   45.67    123.45    456.78     0.12     2.34   0.97  4.87
25.6        0.85      14.5        8.2         16.8       2.1      12.2
```

The most important columns explained:

Column	Meaning	Good values	Bad values
r/s	Read operations per second	< 100 (HDD)	> 300 (HDD)
w/s	Write operations per second	< 50 (HDD)	> 200 (HDD)
rkB/s	Kilobytes read per second	--	--
wkB/s	Kilobytes written per second	--	--
rrqm/s	Read requests merged per second	> 5	< 1
wrqm/s	Write requests merged per second	> 10	< 1
avgrq-sz	Average request size	> 32 KB	< 8 KB
avgqu-sz	Average queue length	< 2	> 10
await	Average wait time (ms)	< 10 (SSD), < 20 (HDD)	> 50
%util	Disk utilization	< 80%	> 95%

> □□ **You will need this later to:** understand whether your disk is the bottleneck and whether an SSD is worth it.

What do `%iowait` and `await` really mean?

These two values confuse many administrators:

```
┌ %iowait (from top or iostat):
├ Shows how much time the CPU waits because processes are waiting for I/O
├ Not disk utilization!
├ High iowait with low CPU load = I/O problem
└ High iowait with high CPU load = possibly normal
```

```
┌ await (from iostat -x):
├ Average time an I/O request takes (in milliseconds)
├ Includes wait time in queue + actual processing time
└ Most important value for I/O performance!
```

□□ Practical example: Identify a slow disk

You start an application and it loads very slowly.

Let us check if the disk is to blame:

```
iostat -x 1
Device            r/s     w/s     rkB/s     wkB/s   await     %util
sda             234.5    12.3   2345.67    123.45    45.2      98.7
```

Diagnosis:

- `r/s` = 234.5: Very many read operations (> 100 is a lot for HDD)
- `await` = 45.2ms: Very high wait time (> 20ms is bad for HDD)
- `%util` = 98.7%: Disk is practically 100% utilized

Finding the cause:

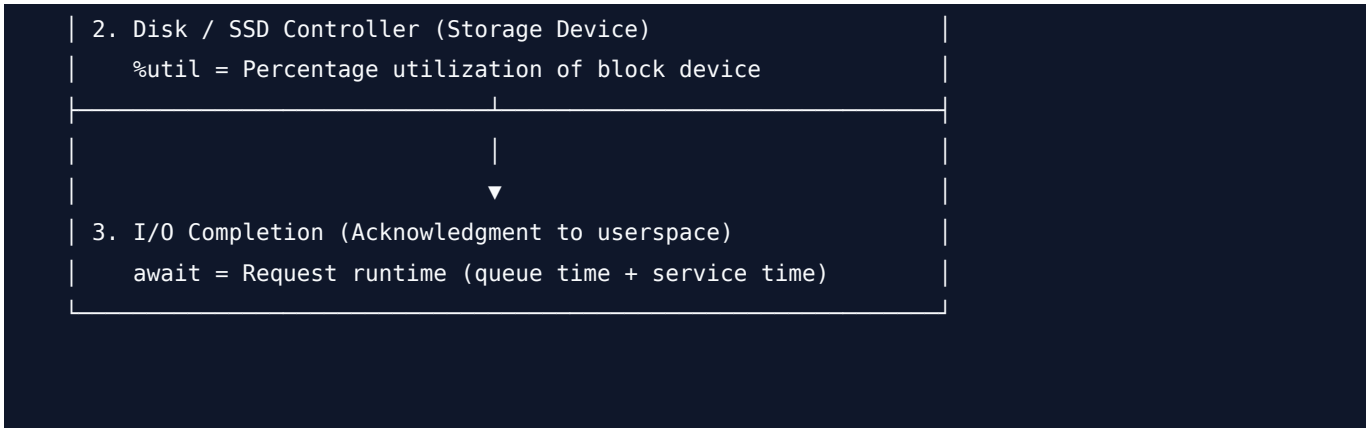
```
# Which process is causing the I/O load?
sudo iotop -o
Total  DISK READ :     2.35 M/s | Total DISK WRITE :     123.45 K/s
Actual DISK READ:      2.35 M/s | Actual DISK WRITE:     123.45 K/s
TID    PRIO  USER      DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
1234   be/4  user      2.34 M/s      0.00 B/s  0.00 % 89.12 % find /home -name "*.jpg"
```

Solution: The `find` command is searching the entire home directory and blocking the disk.

> ⚠ **Common mistake:** Only looking at transfer rate (rkB/s, wkB/s). A disk can also be blocked by many small accesses even if the transfer rate is low.

I/O queue visualized

```
┌─ I/O Request Flow (avgqu-sz, %util, await) ──────────────┐
│ Application sends I/O requests: [App1] [App2] [App3]     │
│                              │                           │
│                              ▼                           │
│ 1. I/O Queue (Kernel Block Layer)                        │
│    avgqu-sz = Number of waiting requests in queue        │
├──────────────────────────────┬───────────────────────────┤
│                              │                           │
│                              ▼                           │
```



> **You will need this later to:** identify I/O bottlenecks, choose the right disk for your application, and systematically diagnose performance problems.

`vmstat` -- Virtual memory and system performance

`vmstat` (Virtual Memory Statistics) gives you a comprehensive overview of system performance. It combines CPU, memory, I/O, and system information in a compact display.

All columns of the `vmstat` output in detail

```
vmstat 1 5

# 1 = update every second
# 5 = 5 measurements, then stop

procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 1234567  67890 4567890   0    0    12    45  123  456 12  3 84  1  0
 2  0      0 1230000  67890 4567890   0    0    15    48  134  467 15  4 80  1  0
```

Column explanation:

`procs` (Processes):

Column	Meaning	Critical values
r	Runnable processes (waiting for CPU)	> Number of CPU cores
b	Blocked processes (waiting for I/O)	> 2

`Memory` (Memory in KB):

Column	Meaning	Critical values
swpd	Used swap memory	> 0 = system is swapping
free	Free memory	< 10% of total RAM
buff	Buffer memory	--
cache	Cache memory	--

`swap` (Swap activity in KB/s):

Column	Meaning	Critical values
si	Swap in (from disk to RAM)	> 0 = problem!
so	Swap out (from RAM to disk)	> 0 = problem!

`io` (I/O activity in blocks/s):

Column	Meaning	Critical values
bi	Blocks in (read)	> 1000
bo	Blocks out (written)	> 1000

`system` (System activity):

Column	Meaning	Critical values

in	Interrupts per second	> 5000
cs	Context switches per second	> 10000

`cpu` (CPU time in %):

Column	Meaning	Critical values
us	User time	--
sy	System time	> 30%
id	Idle time	< 20%
wa	I/O wait	> 20%
st	Steal time (only in VMs)	> 5%

When is swapping problematic?

Swapping is one of the most common performance killers. Here is how to spot it:

□□ Practical example:

System suddenly becomes very slow:

- vmstat 1
- procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
- r b swpd free buff cache si so bi bo in cs us sy id wa st
- 3 2 567890 12345 23456 345678 234 456 123 234 567 890 45 25 15 15 0
- 4 3 678901 11234 23456 345678 345 567 234 345 678 901 50 30 10 10 0

Warning signs:

- swpd > 0: System is using swap memory
- si > 0: Data is being loaded from swap back into RAM
- so > 0: Data is being swapped from RAM to disk
- b > 0: Processes are waiting for I/O (due to swap access)

- wa = 15%: CPU is waiting for I/O (swap is slow)

> **What is happening here?** The system has too little RAM and must constantly move data back and forth between RAM and disk. This is extremely slow!

Immediate measures:

```
# Find and kill memory hogs
top -o %MEM

# Clear swap cache (caution!)
sudo swapoff -a && sudo swapon -a
```

> ⚠ **Warning:** `swapoff -a` can crash the system if not enough RAM is free!

Recognizing a memory leak

A memory leak shows in `vmstat` through continuously decreasing free memory:

- vmstat 5
- procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
- r b swpd free buff cache si so bi bo in cs us sy id wa st
- 1 0 0 2000000 67890 1000000 0 0 5 10 100 200 10 2 87 1 0
- 1 0 0 1800000 67890 1000000 0 0 5 10 100 200 10 2 87 1 0
- 1 0 0 1600000 67890 1000000 0 0 5 10 100 200 10 2 87 1 0
- 1 0 0 1400000 67890 1000000 0 0 5 10 100 200 10 2 87 1 0

Diagnosis: Free memory (`free`) is continuously decreasing from 2 GB to 1.4 GB while cache and buffer remain constant. A process is "eating" memory!

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

Finding the culprit:

```
# Sort processes by memory usage
ps aux --sort=-%mem | head -10
```

> **You will need this later to:** recognize memory leaks early, diagnose swap problems, and evaluate overall system performance.

Why `vmstat 1` is better than `vmstat`

Typical mistake: Many people use `vmstat` without parameters and get distorted values.

```
# WRONG - shows average values since system start
vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 1234567  67890 4567890   0    0     8    15   89  156  5  1 93  1  0

# RIGHT - shows current values every second
vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  2      0 1234567  67890 4567890   0    0   234   456  567  890 45 25 15 15  0
```

The difference: Without parameters `vmstat` shows average values since the last system start. For a performance problem you need the **current** values!

> **You will need this later to:** diagnose performance problems in real time and not be confused by historical average values.

`netstat` and `ss` -- Monitoring network connections

Monitoring network connections is an essential part of Linux administration. You need to know which services are listening on which ports, which connections are active, and whether suspicious activities are taking place.

Difference between `netstat` and the modern `ss`

`netstat` is the traditional tool you will probably learn first:

- Standard on Unix systems for decades
- Simple syntax and familiar parameters
- Slowly being replaced by ss

`ss` (Socket Statistics) is the modern successor:

- Significantly faster, especially with many connections
- More details and better filter options
- Standard in modern Linux distributions

> **You will need this later to:** master both tools since you will encounter both in practice. Older systems often only have netstat, newer ones prefer ss.

All socket states (LISTEN, ESTABLISHED, etc.) explained

TCP connections go through various states. Understanding them helps with diagnosis:

- netstat -tn
- Active Internet connections (w/o servers)
- Proto Recv-Q Send-Q Local Address Foreign Address State

- tcp 0 0 192.168.1.100:22 192.168.1.50:54321 ESTABLISHED
- tcp 0 0 192.168.1.100:80 203.0.113.10:45678 TIME_WAIT
- tcp 0 0 192.168.1.100:443 198.51.100.20:12345 CLOSE_WAIT

The most important TCP states:

State	Meaning	Normal?
LISTEN	Port is waiting for incoming connections	Yes
ESTABLISHED	Active, working connection	Yes
TIME_WAIT	Connection ended, waiting for cleanup	Yes, but many = problem
CLOSE_WAIT	Connection is being terminated	Many = application problem
FIN_WAIT1	Connection is actively being terminated	Briefly normal
FIN_WAIT2	Waiting for termination confirmation	Briefly normal
SYN_SENT	Connection establishment in progress	Many = network problem
SYN_RECV	Connection request received	Many = possible attack

Common problems:

Too many TIME_WAIT connections:

```
netstat -tn | grep TIME_WAIT | wc -l
15432
```

Problem: Application opens too many connections too quickly.

Many CLOSE_WAIT connections:

```
$ netstat -tn | grep CLOSE_WAIT | wc -l
234
```

Problem: Application does not close connections properly.

Practical example:

Find open ports and suspicious connections

Step 1: List all listening ports

```
netstat -tlnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1234/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
5678/master
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
9012/apache2
tcp6       0      0 :::22                   :::*                    LISTEN      1234/sshd
tcp6       0      0 ::1:25                  :::*                    LISTEN
5678/master
```

Understanding the output:

- **Proto:** Protocol (tcp, tcp6, udp, udp6)
- **Local Address:** IP:Port being listened on
- **State:** Connection status (LISTEN = waiting for connections)
- **PID/Program name:** Which process uses the port

Key findings:

- `Port 22` `SSH` is open on all interfaces (0.0.0.0:22)
- `Port 25` `SMTP` only local (127.0.0.1:25)
- `Port 80` `HTTP` is publicly accessible (0.0.0.0:80)

Step 2: Identify unknown ports

```
# Port 1337 should not be open!
netstat -tlnp | grep :1337
tcp        0      0 0.0.0.0:1337            0.0.0.0:*               LISTEN
6666/suspicious
```

Step 3: Investigate the suspicious process

```
$ ps aux | grep 6666
user     6666  0.1  0.5  12345  6789 ?        S    14:30   0:01 /tmp/suspicious
```

Solution:

```
# Suspicious: program in /tmp!
ls -la /tmp/suspicious
-rwxr-xr-x 1 user user 123456 Jul 11 14:30 /tmp/suspicious
```

Step 4: Check active connections to this port

```
netstat -tn | grep :1337
tcp        0      0 192.168.1.100:1337     203.0.113.50:54321     ESTABLISHED
```

> ⚠ **Warning:** An unknown process is listening on port 1337 and has an active connection to an external IP. This could be malware!

Modern alternative with `ss`

```
# Equivalent to netstat -tlnp
ss -tlnp

# Extended filter options
ss -t state established '( dport = :80 or dport = :443 )'

# Connections to a specific port
ss -tn sport = :1337
```

Common mistake: using `netstat` without parameters

BAD -- shows too much and is slow:

```
$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 hostname.local:ssh      client.local:54321     ESTABLISHED
```

```
tcp        0      0 hostname.local:http     google.com:80          TIME_WAIT
...
(hundreds of lines)
```

Problems:

- DNS resolution makes the output slow
- Too much information, confusing
- No process information

BETTER -- search specifically for what you need:

```
# Only listening TCP ports with process info, numeric
$ netstat -tlnp

# Or with ss (more modern)
$ ss -tlnp
```

Network monitoring checklist:

Daily checks:

```
# 1. Which services are listening?
ss -tlnp | grep LISTEN

# 2. Unusual connections?
ss -tn | grep -v "127.0.0.1\|::1"

# 3. Too many TIME_WAIT?
ss -tn state time-wait | wc -l
```

```
# 4. Suspicious ports?
ss -tlnp | grep -E ":(1337|4444|6666|31337)"
```

During performance problems:

```
# Count connections per state
ss -tn | awk '{print $1}' | sort | uniq -c

# Top connections by remote IP
ss -tn | awk '{print $4}' | cut -d: -f1 | sort | uniq -c | sort -nr | head -10
```

> **You will need this later to:** diagnose network problems, recognize security incidents, and monitor network performance. And as an administrator to recognize suspicious network activity, identify performance problems from too many connections, and monitor network services.

Log files -- systematic analysis

Understanding the systemd journal

The systemd journal is Linux's modern logging system and has largely replaced the traditional log files in `/var/log`. As a Linux administrator you must master `journalctl` -- it is your most important tool for analyzing system problems.

`journalctl` -- Your most important log tool

`journalctl` is the command you use to access the systemd journal. Unlike traditional log files, the journal stores structured data that you can filter and search with precision.

Basic usage:

```
# Show all journal entries (oldest first)
journalctl

# Newest entries first (like tail -f)
journalctl -r

# Only the last 20 lines
journalctl -n 20

# Live mode: Follow new entries in real time
journalctl -f
```

Why is `journalctl` so important? Traditional log files are just text files. The systemd journal stores additional metadata such as timestamps, priorities, process IDs, and service names. This makes analysis much more precise.

> **You will need this later to:** quickly diagnose system problems without having to scroll through hundreds of log lines.

Most important parameters: `-f`, `-u`, `--since`, `--until`

Live monitoring with `-f`:

```
# Follow new log entries in real time
journalctl -f

# Combined with other filters
journalctl -f -u ssh.service
```

Service-specific logs with `-u`:

```
# Only logs from the SSH service
journalctl -u ssh.service

# Multiple services at once
journalctl -u ssh.service -u apache2.service

# All services with "network" in the name
journalctl -u "*network*"
```

Time-based filters with `--since` and `--until`:

```
# Logs from the last hour
journalctl --since "1 hour ago"

# Logs from today
journalctl --since "today"

# Logs from yesterday
journalctl --since "yesterday" --until "today"

# Specific time range
journalctl --since "2025-07-12 10:00:00" --until "2025-07-12 12:00:00"

# Logs from the last 10 minutes
journalctl --since "10 minutes ago"
```

Combined filters:

```
# SSH logs from the last 2 hours
journalctl -u ssh.service --since "2 hours ago"

# Error logs from today
journalctl -p err --since "today"

# Live monitoring for critical errors only
journalctl -f -p crit
```

Understanding priority levels:

```
# Only errors and critical messages
journalctl -p err

# All priorities from "warning" upward
journalctl -p warning
```

Priority	Number	Meaning	When to use?
emerg	0	System is unusable	System outage
alert	1	Immediate action required	Critical errors
crit	2	Critical conditions	Hardware problems
err	3	Error conditions	Service failures
warning	4	Warnings	Potential problems
notice	5	Normal but significant events	Service starts
info	6	Informational messages	Normal activity

Priority	Number	Meaning	When to use?
debug	7	Debug messages	Development

□□ Practical example:

Analyzing system startup problems

Imagine your system starts slowly or a service is not working.

Here is the systematic approach:

Step 1: Analyze the boot process

```
# All logs from the last boot
journalctl -b

# Logs from the previous boot (if the system was restarted)
journalctl -b -1

# Only errors from the last boot
journalctl -b -p err
```

Step 2: Identify time-consuming services

```
# Boot time analysis
systemd-analyze blame
Startup finished in 2.547s (kernel) + 8.234s (userspace) = 10.781s
graphical.target reached after 8.234s in userspace

□ 3.456s NetworkManager.service
□ 2.123s mysql.service
□ 1.789s apache2.service
```

```
 0.987s ssh.service
```

Step 3: Examine the problematic service in detail

```
# MySQL service from the last boot
journalctl -b -u mysql.service

# Output might look like this:
Jul 12 14:30:15 hostname systemd[1]: Starting MySQL Community Server...
Jul 12 14:30:16 hostname mysqld[1234]: [Warning] World-writable config file
'/etc/mysql/my.cnf' is ignored
Jul 12 14:30:17 hostname mysqld[1234]: [ERROR] Can't start server: Bind on TCP/IP port:
Address already in use
Jul 12 14:30:17 hostname systemd[1]: mysql.service: Main process exited, code=exited,
status=1/FAILURE
Jul 12 14:30:17 hostname systemd[1]: mysql.service: Failed with result 'exit-code'.
```

Step 4: Identify the root cause

```
# Which process is blocking port 3306?
netstat -tlnp | grep :3306
tcp        0      0 0.0.0.0:3306            0.0.0.0:*               LISTEN
5678/old_mysql

# Old MySQL process is still running!
ps aux | grep 5678
mysql    5678  0.1  2.3  123456  234567 ?      S    14:25   0:05 /usr/sbin/old_mysql
```

Solution: The old MySQL process is blocking the port. After terminating it, the service starts normally.

> ⚠ **Typical pitfall:** Many people only look at the error message "Can't start server" but the actual cause is in the line before: "Address already in use".

> □□ **Tip:** How to store the journal persistently: By default, the systemd journal stores logs only temporarily in RAM. After a reboot they are gone!

Check the problem:

```
# Is the journal persistent?
ls -la /var/log/journal/
ls: cannot access '/var/log/journal/': No such file or directory

# Or does only temporary storage exist?
ls -la /run/log/journal/
drwxr-sr-x+ 3 root systemd-journal 4096 Jul 12 14:30 .
```

Enable persistent storage:

```
# Create the directory
sudo mkdir -p /var/log/journal

# Set correct permissions
sudo chown root:systemd-journal /var/log/journal
sudo chmod 2755 /var/log/journal

# Restart systemd-journald
sudo systemctl restart systemd-journald
```

```
# Verify it works
ls -la /var/log/journal/
drwxr-sr-x+ 3 root systemd-journal 4096 Jul 12 14:35 .
drwxr-sr-x+ 3 root systemd-journal 4096 Jul 12 14:35 a1b2c3d4e5f6...
```

Alternative: Configuration via `/etc/systemd/journald.conf`:

```
sudo nano /etc/systemd/journald.conf

# Change this line:
Storage=persistent

# Set storage limits (optional)
SystemMaxUse=500M
RuntimeMaxUse=100M

# Activate the configuration
sudo systemctl restart systemd-journald
```

Why is this important? Without persistent storage you lose all logs during a system failure -- exactly when you need them most!

Useful journal management:

```
# Current disk usage
journalctl --disk-usage
Archived and active journals take up 234.5M in the file system.

# Clean up old logs (older than 2 weeks)
sudo journalctl --vacuum-time=2weeks
```

```
# Enforce storage limit (max 500MB)
sudo journalctl --vacuum-size=500M

# Limit number of journal files
sudo journalctl --vacuum-files=10
```

Common mistake: Many people use `journalctl` without parameters and get an overwhelming output with thousands of lines. Always use filters!

> **You will need this later to:** systematically diagnose system problems, identify service failures, and resolve boot issues. In the LPIC-1 exam, `journalctl` is a frequent topic.

Traditional log files in /var/log

Although the systemd journal offers modern logging features, many services and applications still use traditional log files in `/var/log`. As a Linux administrator you must master both systems as they complement each other.

Which file contains what? (syslog, auth.log, kern.log, etc.)

The `/var/log` directory contains various specialized log files. Each has a specific purpose:

- ls -la /var/log/
- rw-r----- 1 syslog adm 123456 Jul 12 14:30 syslog
- rw-r----- 1 syslog adm 45678 Jul 12 14:30 auth.log
- rw-r----- 1 syslog adm 67890 Jul 12 14:30 kern.log
- rw-r----- 1 syslog adm 34567 Jul 12 14:30 daemon.log
- rw-r----- 1 syslog adm 78901 Jul 12 14:30 mail.log
- rw-r--r-- 1 root root 23456 Jul 12 14:30 dpkg.log
- rw-r--r-- 1 root root 12345 Jul 12 14:30 apt/history.log

The most important log files at a glance:

File	Content	When to use?
syslog	General system messages	Overview of system activity
auth.log	Authentication and authorization	Login problems, SSH attacks
kern.log	Kernel messages	Hardware problems, driver errors
daemon.log	Messages from system services	Service-specific problems
mail.log	E-mail system (Postfix, Sendmail)	Mail server diagnostics
apache2/error.log	Apache web server errors	Website problems
mysql/error.log	MySQL database errors	Database problems
dpkg.log	Package installation and updates	Tracking software installation
apt/history.log	Package updates	Tracking software installation

□□ Practical example: Analyzing important system logs

```
# General system activity
tail -f /var/log/syslog
Jul 12 14:30:15 hostname systemd[1]: Started Daily apt download activities.
Jul 12 14:30:16 hostname cron[1234]: (root) CMD (test -x /usr/sbin/anacron || ( cd / &&
run-parts --report /etc/cron.daily ))

# Authentication and login attempts
tail -f /var/log/auth.log
Jul 12 14:30:17 hostname sshd[5678]: Accepted publickey for user from 192.168.1.50 port
54321 ssh2
Jul 12 14:30:18 hostname sudo[9012]: user : TTY=pts/0 ; PWD=/home/user ; USER=root ;
COMMAND=/bin/systemctl status ssh

# Kernel and hardware
tail -f /var/log/kern.log
Jul 12 14:30:19 hostname kernel: [12345.678901] usb 2-1: new high-speed USB device number
3 using ehci-pci
```

```
Jul 12 14:30:20 hostname kernel: [12345.789012] usb-storage 2-1:1.0: USB Mass Storage
device detected
```

> **You will need this later to:** quickly find the right log file for a specific problem without having to search through all logs.

Understanding and configuring log rotation

Log files would grow indefinitely without rotation and eventually fill up the disk. The `logrotate` system prevents this automatically.

How log rotation works:

```
# Current log file
/var/log/syslog

# After the first rotation
/var/log/syslog      (new, empty file)
/var/log/syslog.1    (old contents, compressed)

# After the second rotation
/var/log/syslog      (new, empty file)
/var/log/syslog.1    (yesterday)
/var/log/syslog.2.gz (day before yesterday, compressed)

# After several rotations
/var/log/syslog
/var/log/syslog.1
/var/log/syslog.2.gz
/var/log/syslog.3.gz
/var/log/syslog.4.gz
```

Understanding logrotate configuration:

```
# Main configuration
cat /etc/logrotate.conf

# Global settings
weekly
rotate 4
create
compress
include /etc/logrotate.d

# Service-specific configuration
$ cat /etc/logrotate.d/rsyslog
/var/log/syslog
/var/log/mail.info
/var/log/mail.warn
/var/log/mail.err
/var/log/daemon.log
/var/log/kern.log
/var/log/auth.log
/var/log/user.log
{
□weekly
□missingok
□rotate 52
□compress
□delaycompress
□notifempty
□create 640 syslog adm
□postrotate
□□/usr/lib/rsyslog/rsyslog-rotate
□endscript
}
```

Important logrotate options:

Option	Meaning	Example
weekly	Rotate every week	`daily`, `monthly`

Option	Meaning	Example
rotate 52	Keep 52 old versions	1 year with weekly rotation
compress	Compress old logs	Saves disk space
delaycompress	Only compress from the 2nd rotation onward	For running processes
notifempty	Do not rotate empty files	Prevents unnecessary rotations
create 640 syslog adm	Create new file with specific permissions	Security and access
postrotate	Commands to run after rotation	Reload the service

Manual log rotation testing:

```
# Run logrotate for all configurations
sudo logrotate /etc/logrotate.conf

# Only for a specific configuration
sudo logrotate /etc/logrotate.d/rsyslog

# Test mode (shows what would happen)
sudo logrotate -d /etc/logrotate.conf

# Forced rotation (even if not needed)
sudo logrotate -f /etc/logrotate.d/rsyslog
```

> ⚠ **Warning:** Without log rotation, log files can fill up the entire disk and crash the system!

□□ Practical example:

Finding failed login attempts

A common administrative scenario: You suspect brute-force attacks on your system and want to analyze failed login attempts.

Step 1: Find failed SSH logins

```
# All failed SSH logins
grep "Failed password" /var/log/auth.log
Jul 12 14:25:30 hostname sshd[1234]: Failed password for root from 203.0.113.10 port 45678
ssh2
Jul 12 14:25:32 hostname sshd[1235]: Failed password for admin from 203.0.113.10 port
45679 ssh2
Jul 12 14:25:34 hostname sshd[1236]: Failed password for user from 203.0.113.10 port 45680
ssh2
Jul 12 14:25:36 hostname sshd[1237]: Failed password for invalid user test from
203.0.113.10 port 45681 ssh2
```

Step 2: Identify attacker IPs

```
# Top 10 attacker IPs
grep "Failed password" /var/log/auth.log | awk '{print $11}' | sort | uniq -c | sort -nr |
head -10
 234 203.0.113.10
  45 198.51.100.20
  23 192.0.2.30
  12 203.0.113.40
   8 198.51.100.50
```

Step 3: Analyze the time distribution

```
# Attacks per hour
grep "Failed password" /var/log/auth.log | awk '{print $1" "$2" "$3}' | cut -d: -f1 | sort
```

```
| uniq -c
  45 Jul 12 14
  67 Jul 12 15
 123 Jul 12 16
  89 Jul 12 17
```

Step 4: Detect user account targeting

```
# Which usernames are being attacked?
grep "Failed password" /var/log/auth.log | awk '{print $9}' | sort | uniq -c | sort -nr
 156 root
  67 admin
  34 user
  23 test
  12 guest
```

Step 5: Check for successful logins after attacks

```
# Successful logins from suspicious IPs
$ grep "Accepted" /var/log/auth.log | grep "203.0.113.10"

# (Hopefully empty!)
# All successful logins today
grep "Accepted" /var/log/auth.log | grep "$(date '+%b %d')"
Jul 12 14:30:17 hostname sshd[5678]: Accepted publickey for user from 192.168.1.50 port
54321 ssh2
```

Automated monitoring script:

```
#!/bin/bash
# Simple brute-force detection script

LOGFILE="/var/log/auth.log"
THRESHOLD=10

echo "=== Brute-Force Analysis ==="
echo "Date: $(date)"
echo

# IPs with more than $THRESHOLD failed logins
echo "Suspicious IPs (> $THRESHOLD failed attempts):"
grep "Failed password" $LOGFILE | \
awk '{print $11}' | \
sort | uniq -c | \
awk -v threshold=$THRESHOLD '$1 > threshold {print $1 " attempts from " $2}' | \
sort -nr

echo
echo "Last 5 failed logins:"
grep "Failed password" $LOGFILE | tail -5
```

> ⚠ **Common mistake:** Only checking `/var/log/auth.log` and missing that the attacks could be in `/var/log/auth.log.1` or older files due to log rotation.

Better search across all rotated logs:

```
# Search all auth.log files (including rotated ones)
zgrep "Failed password" /var/log/auth.log*
```

Preventive measures based on log analysis:

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

```
# Install and configure Fail2ban
sudo apt install fail2ban

# Manually block suspicious IPs
sudo iptables -A INPUT -s 203.0.113.10 -j DROP

# Harden SSH configuration
sudo nano /etc/ssh/sshd_config

# PermitRootLogin no
# MaxAuthTries 3
# AllowUsers user1 user2
```

Warning: Never simply delete logs

Typical beginner mistake:

```
# NEVER DO THIS!
sudo rm /var/log/syslog
sudo rm /var/log/auth.log
```

Why this is problematic:

- Running processes: Services may still be writing to the files
- Permissions: New files will have incorrect permissions
- Forensics: In case of security incidents, the logs are gone
- Compliance: Many regulations require log retention

Correct methods for log cleanup:

```
# Empty the log file (but do not delete it)
sudo truncate -s 0 /var/log/syslog

# Or with cat
sudo cat /dev/null > /var/log/syslog

# Use logrotate for immediate rotation
sudo logrotate -f /etc/logrotate.d/rsyslog

# Delete old rotated logs (safely)
sudo find /var/log -name "*.log.*.gz" -mtime +30 -delete
```

Monitor log file sizes:

```
# Find the largest log files
sudo du -sh /var/log/* | sort -hr | head -10
2.3G    /var/log/journal
456M    /var/log/apache2
123M    /var/log/syslog
 67M    /var/log/auth.log
 45M    /var/log/kern.log

# Automatic warning for large logs
find /var/log -name "*.log" -size +100M -exec ls -lh {} \;
```

> **You will need this later to:** analyze security incidents, meet compliance requirements, and diagnose system problems retroactively. To also systematically analyze traditional log files, detect brute-force attacks, and configure log rotation correctly. Many older systems and applications still use these traditional logs.

Log analysis with shell tools

While `journalctl` is perfect for systemd logs, you need the classic shell tools for traditional log files and complex analyses. `grep`, `awk`, and `sed` are your most important tools for log analysis -- they are available on every Linux system and extremely powerful.

Using `grep`, `awk`, `sed` for log analysis

`grep` -- Finding and filtering lines:

`grep` is your most important tool for searching log files. It finds lines that contain specific patterns.

```
# Simple text search
grep "error" /var/log/syslog
Jul 12 14:30:15 hostname systemd[1]: Failed to start some.service.
Jul 12 14:30:16 hostname kernel: [12345.678] USB disconnect, address 1

# Ignore case
grep -i "ERROR" /var/log/syslog

# Count lines instead of displaying them
grep -c "Failed password" /var/log/auth.log
234

# Show line numbers
grep -n "kernel" /var/log/syslog
1234:Jul 12 14:30:16 hostname kernel: [12345.678] USB disconnect

# Show context (2 lines before and after the match)
grep -C 2 "error" /var/log/syslog
```

Important `grep` options:

Option	Meaning	When to use?

-i	Ignore case	Troubleshooting
-v	Inverted (lines WITHOUT pattern)	Hide unwanted entries
-c	Only count matches	Creating statistics
-n	Show line numbers	Context in large files
-A 5	5 lines after the match	See follow-up events
-B 5	5 lines before the match	See root cause events
-C 5	5 lines before and after the match	Full context

`awk` -- Extracting and processing columns:

`awk` is perfect for structured log files where you need specific columns.

```
# Extract only IP addresses from auth.log
grep "Failed password" /var/log/auth.log | awk '{print $11}'
203.0.113.10
198.51.100.20
192.0.2.30

# Combine timestamp and IP
grep "Failed password" /var/log/auth.log | awk '{print $1" "$2" "$3" - "$11}'
Jul 12 14:25:30 - 203.0.113.10
Jul 12 14:25:32 - 203.0.113.10

# Filter columns with conditions
awk '$5 > 90 {print "⚠ WARNING: " $1 " is " $5 " full!"}' <(df -h | tail -n +2)
⚠ WARNING: /dev/sda1 is 95% full!
```

`sed` -- Replacing and editing text:

`sed` is ideal for cleaning and formatting log output.

```
# Anonymize IP addresses
grep "Failed password" /var/log/auth.log | sed 's/[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-
9]\{1,3\}\.[0-9]\{1,3\}/XXX.XXX.XXX.XXX/g'

# Show only specific lines (lines 10-20)
sed -n '10,20p' /var/log/syslog

# Remove empty lines
sed '/^$/d' /var/log/some.log

# Replace multiple spaces with one
sed 's/  */ /g' /var/log/some.log
```

> **You will need this later to:** perform complex log analyses, create reports, and prepare log data for further processing.

Practical example:

Identifying the top 10 error sources

Imagine your system has performance problems and you want to identify the most common error sources. Here is a systematic analysis:

Step 1: Collect all error messages

```
# All lines with "error", "failed", "warning" from syslog
grep -iE "(error|failed|warning)" /var/log/syslog > /tmp/errors.log

# Also from other important logs
grep -iE "(error|failed|warning)" /var/log/kern.log >> /tmp/errors.log
grep -iE "(error|failed|warning)" /var/log/daemon.log >> /tmp/errors.log
```

Step 2: Categorize error types

```
# Most common error messages (first 5 words)
awk '{print $5" "$6" "$7" "$8" "$9}' /tmp/errors.log | sort | uniq -c | sort -nr | head -
10
  45 Failed to start some.service
  23 kernel: USB disconnect, address
  18 systemd: Failed to reload
  12 NetworkManager: device eth0: link
   8 cron: (root) CMD failed
   6 apache2: [error] [client]
   4 mysql: [Warning] World-writable config
   3 postfix: warning: hostname verification
   2 sshd: error: Bind to
   1 cups: Job failed, will
```

Step 3: Analyze the time distribution

```
# Errors per hour
awk '{print $3}' /tmp/errors.log | cut -d: -f1 | sort | uniq -c | sort -nr
  67 16
  45 15
  34 14
  23 17
  12 13
```

Step 4: Service-specific analysis

```
# Which services cause the most problems?
awk '{print $5}' /tmp/errors.log | cut -d: -f1 | sort | uniq -c | sort -nr | head -5
  89 systemd
  45 kernel
  23 NetworkManager
  18 apache2
  12 cron
```


```
# Which services cause the most problems?
awk '{print $5}' /tmp/errors.log | cut -d: -f1 | sort | uniq -c | sort -nr | head -5
  89 systemd
  45 kernel
  23 NetworkManager
  18 apache2
  12 cron
```


Step 5: Detailed analysis of the top problems

```
# Analyze all systemd errors in detail
grep "systemd" /tmp/errors.log | awk '{for(i=6;i<=NF;i++) printf "%s ", $i; print ""}' |
sort | uniq -c | sort -nr | head -5
  34 Failed to start some.service: Unit some.service not found.
  23 Failed to reload apache2.service: Job for apache2.service failed.
  18 some.service: Main process exited, code=exited, status=1/FAILURE
  12 Timed out waiting for device dev-disk-by\x2duuid-12345.device.
   4 Failed to mount /mnt/backup: No such file or directory
```

One-liners for common log analyses

Here are proven one-liners that you as an administrator regularly need:

Security analyses:

```
# Top 10 attacker IPs (SSH)
grep "Failed password" /var/log/auth.log | awk '{print $11}' | sort | uniq -c | sort -nr |
head -10

# Successful root logins
grep "Accepted.*root" /var/log/auth.log | awk '{print $1" "$2" "$3" from "$11}'

# Sudo commands from the last 24h
grep "$(date '+%b %d')" /var/log/auth.log | grep "sudo.*COMMAND" | awk
'{for(i=14;i<=NF;i++) printf "%s ", $i; print ""}'

# Unusual login times (at night)
grep "Accepted" /var/log/auth.log | awk '$3 ~ /^0[0-5]:/ {print $0}'
```

Performance analyses:

```
# Services that are restarted most frequently
grep "Started|Stopped" /var/log/syslog | awk '{print $6}' | sort | uniq -c | sort -nr |
head -10

# Kernel errors from the last week
grep "$(date -d '7 days ago' '+%b %d')" /var/log/kern.log | grep -i error

# Memory warnings
grep -i "out of memory|oom|killed process" /var/log/kern.log

# Disk errors
grep -i "i/o error|read error|write error" /var/log/kern.log
```

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

System monitoring:

```
# Most frequent cron jobs
grep "CRON" /var/log/syslog | awk '{print $6}' | sort | uniq -c | sort -nr | head -10

# Network interface problems
grep -i "link|carrier|duplex" /var/log/kern.log | tail -20

# USB device activity
grep "usb" /var/log/kern.log | grep "$(date '+%b %d')" | tail -10

# Temperature warnings
grep -i "temperature|thermal|overheat" /var/log/kern.log
```

> **Typical mistake:** Searching logs without a timestamp filter

Common beginner mistake:

```
# BAD - searches all logs going back years
grep "error" /var/log/syslog
(thousands of lines from months/years)
```

BETTER -- work with a time filter:

```
# Only logs from today
```

```
grep "$(date '+%b %d')" /var/log/syslog | grep -i error

# Last 100 lines, then filter
tail -100 /var/log/syslog | grep -i error

# Specific time range
sed -n '/Jul 12 14:00/,/Jul 12 16:00/p' /var/log/syslog | grep -i error

# With awk for precise time filtering
awk '/Jul 12 14:/ && /error/ {print}' /var/log/syslog
```

Why is this important?

Without a time filter you get:

- Irrelevant old entries
- Overwhelming data volumes
- Slow execution on large log files
- Confusion between current and historical problems

Advanced time filter techniques:

```
# Logs from the last 2 hours (with GNU date)
SINCE=$(date -d '2 hours ago' '+%b %d %H:')
grep "^$SINCE" /var/log/syslog

# Combine multiple days
grep -E "$(date '+%b %d')|$(date -d 'yesterday' '+%b %d')" /var/log/syslog

# Time range between two points
awk '/Jul 12 09:00/,/Jul 12 17:00/' /var/log/syslog | grep -i error
```

□□ Practical example: Monitoring dashboard with shell tools

```
#!/bin/bash
# Simple log analysis dashboard

echo "=== System Log Analysis $(date) ==="
echo

echo "□□Errors from the last hour:"
LAST_HOUR=$(date -d '1 hour ago' '+%b %d %H:')
grep "^$LAST_HOUR" /var/log/syslog | grep -iE "(error|failed|warning)" | wc -l
echo

echo "□□Top 5 error sources today:"
grep "$(date '+%b %d')" /var/log/syslog | grep -iE "(error|failed)" |
awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head -5
echo

echo "□□SSH attacks today:"
grep "$(date '+%b %d')" /var/log/auth.log | grep "Failed password" |
awk '{print $11}' | sort | uniq -c | sort -nr | head -3
echo

echo "□□Critical kernel messages:"
grep "$(date '+%b %d')" /var/log/kern.log | grep -iE "(error|critical|fatal)" | tail -3
```

> ⚠ **Performance tip:** For very large log files use zgrep for compressed logs and tail -f for live monitoring instead of grep over the entire file.

Combined analysis pipeline:

```
# Complex analysis in a single pipeline
grep "$(date '+%b %d')" /var/log/syslog |
```

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

```
    grep -i error |
    awk '{print $5}' |
    cut -d: -f1 |
    sort | uniq -c |
    sort -nr |
    head -10 |
    awk '{printf "%-20s %s errors", $2, $1}'
```

> **You will need this later to:** efficiently navigate through large volumes of logs, quickly find relevant information, and create automated log analyses. Also to perform complex log analyses, investigate security incidents, and systematically diagnose performance problems. These shell tools are available on every Linux system and form the foundation for professional log analysis.

Setting up automated monitoring

Creating simple monitoring scripts

As a Linux administrator, you don't want to constantly run `top`, `iostat`, or `journalctl` manually. Automated monitoring scripts continuously watch your system and warn you before problems become critical.

Bash script: Monitor CPU usage

Let's start with a simple script that monitors CPU usage:

```
#!/bin/bash
# cpu_monitor.sh - Simple CPU monitoring
# Configuration
CPU_THRESHOLD=80
```

```
LOGFILE="/var/log/cpu_monitor.log"
HOSTNAME=$(hostname)

# Read current load
LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
CPU_CORES=$(nproc)

# Convert load to percentage
LOAD_PERCENT=$(echo "scale=2; $LOAD_1MIN / $CPU_CORES * 100" | bc)

# Timestamp for log
TIMESTAMP=$(date '+%Y-%m-%d %H:%M:%S')

# Check and log
if (( $(echo "$LOAD_PERCENT > $CPU_THRESHOLD" | bc -l) )); then
    echo "[$TIMESTAMP] WARNING: CPU load at ${LOAD_PERCENT}% (threshold: ${CPU_THRESHOLD}%)" |
tee -a $LOGFILE

    # Top 5 processes with highest CPU usage
    echo "[$TIMESTAMP] Top 5 CPU consumers:" >> $LOGFILE
    ps aux --sort=-%cpu | head -6 | tail -5 >> $LOGFILE
    echo "---" >> $LOGFILE

    # Return value for further actions
    exit 1
else
    echo "[$TIMESTAMP] CPU load normal: ${LOAD_PERCENT}%" >> $LOGFILE
    exit 0
fi
```

Make the script executable and test it:

```
# Create script and set permissions
sudo nano /usr/local/bin/cpu_monitor.sh
sudo chmod +x /usr/local/bin/cpu_monitor.sh

# Run first test
sudo /usr/local/bin/cpu_monitor.sh
```

```
[2025-07-12 16:30:15] CPU load normal: 23.45%

# Check log file
sudo tail -5 /var/log/cpu_monitor.log
[2025-07-12 16:30:15] CPU load normal: 23.45%
```

> **You will need this later to:** detect CPU problems before the system becomes overloaded and users start complaining.

Defining thresholds and triggering actions

Monitoring without actions is useless. Here we extend the script with different thresholds and actions:

```
#!/bin/bash
# advanced_cpu_monitor.sh - Advanced CPU monitoring
# Configuration
WARNING_THRESHOLD=70
CRITICAL_THRESHOLD=90
LOGFILE="/var/log/cpu_monitor.log"
EMAIL="admin@example.com"

# Define functions
send_alert() {
    local level=$1
    local message=$2
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    # Write log
    echo "[$timestamp] $level: $message" | tee -a $LOGFILE

    # Send email (if mailutils installed)
    if command -v mail >/dev/null 2>&1; then
        echo "$message" | mail -s "[$level] CPU alarm on $(hostname)" $EMAIL
```

```
    fi
    }

    get_top_processes() {
    echo "Top 5 CPU consumers:"
    ps aux --sort=-%cpu | head -6 | tail -5 | awk '{printf "%-10s %5s%% %sn", $1, $3, $11}'
    }

    # Main logic
    LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
    CPU_CORES=$(nproc)
    LOAD_PERCENT=$(echo "scale=1; $LOAD_1MIN / $CPU_CORES * 100" | bc)

    # Check thresholds
    if (( $(echo "$LOAD_PERCENT > $CRITICAL_THRESHOLD" | bc -l) )); then
    MESSAGE="CRITICAL: CPU load at ${LOAD_PERCENT}% (limit: ${CRITICAL_THRESHOLD}%)
    $(get_top_processes)"
    send_alert "CRITICAL" "$MESSAGE"

    # Emergency action: identify resource-intensive processes
    echo "[$timestamp] Emergency analysis started" >> $LOGFILE
    iostat -x 1 3 >> $LOGFILE 2>&1

    exit 2

    elif (( $(echo "$LOAD_PERCENT > $WARNING_THRESHOLD" | bc -l) )); then
    MESSAGE="WARNING: CPU load at ${LOAD_PERCENT}% (warning threshold: ${WARNING_THRESHOLD}%)
    $(get_top_processes)"
    send_alert "WARNING" "$MESSAGE"
    exit 1

    else

    # Log only when needed (every 10 minutes)
    if [ $(($(date +%M) % 10)) -eq 0 ]; then
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] CPU load normal: ${LOAD_PERCENT}%" >> $LOGFILE
    fi
    exit 0
    fi
```

Extended configuration with different actions:

Threshold	Action	Purpose
< 70%	Normal logging (every 10 min)	Baseline documentation
70-89%	Warning + email + top processes	Early warning
≥ 90%	Critical alarm + detailed analysis	Immediate action

□□ Practical example:

Send email on high load

For email notifications you need a working mail system:

Step 1: Set up a simple mail system

```
# Install mailutils
sudo apt install mailutils postfix

# Debian/Ubuntu
sudo dnf install mailx postfix

# Fedora/RHEL
# Configure Postfix for local mails
sudo dpkg-reconfigure postfix

# Choose: "Local only"
# System mail name: your-hostname.local
```

Step 2: Test mail function

```
# Send test mail
```

```
echo "Test message" | mail -s "Test subject" root

# Check mail queue
mailq

# Read local mail
mail
```

Step 3: Monitoring script with mail integration

```
#!/bin/bash
# mail_cpu_monitor.sh - CPU monitoring with email

send_mail_alert() {
    local subject=$1
    local body=$2
    local recipient="root"

    # Local admin
    # Collect detailed system info
    {
        echo "Hostname: $(hostname)"
        echo "Time: $(date)"
        echo "Uptime: $(uptime)"
        echo ""
        echo "$body"
        echo ""
        echo "=== Current System Info ==="
        echo "Load Average: $(cat /proc/loadavg)"
        echo "Memory: $(free -h | grep '^Mem:')"
        echo "Disk: $(df -h / | tail -1)"
        echo ""
        echo "=== Top Processes ==="
        ps aux --sort=-%cpu | head -10
    } | mail -s "$subject" $recipient
}

# Main script
```

```
    LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
    CPU_CORES=$(nproc)
    LOAD_PERCENT=$(echo "scale=1; $LOAD_1MIN / $CPU_CORES * 100" | bc)

    if (( $(echo "$LOAD_PERCENT > 80" | bc -l) )); then
    send_mail_alert "High CPU load on $(hostname)"
    "CPU usage: ${LOAD_PERCENT}%

    The system shows unusually high CPU load.
    Please check the running processes and take action if necessary."
    fi
```

Tip: Why you need hysteresis (on and off thresholds)

Problem without hysteresis:

```
    # BAD - alarm spam possible
    if [ $CPU_LOAD -gt 80 ]; then
    send_alert "CPU high!"
    fi
```

What happens: CPU fluctuates between 79% and 81% → hundreds of emails!

Solution with hysteresis:

```
    #!/bin/bash
    # hysteresis_monitor.sh - Monitoring with hysteresis

    ALERT_FILE="/tmp/cpu_alert_active"
    WARNING_THRESHOLD=80
    RECOVERY_THRESHOLD=70
```

```
CURRENT_LOAD=$(get_cpu_load)

# Your load function

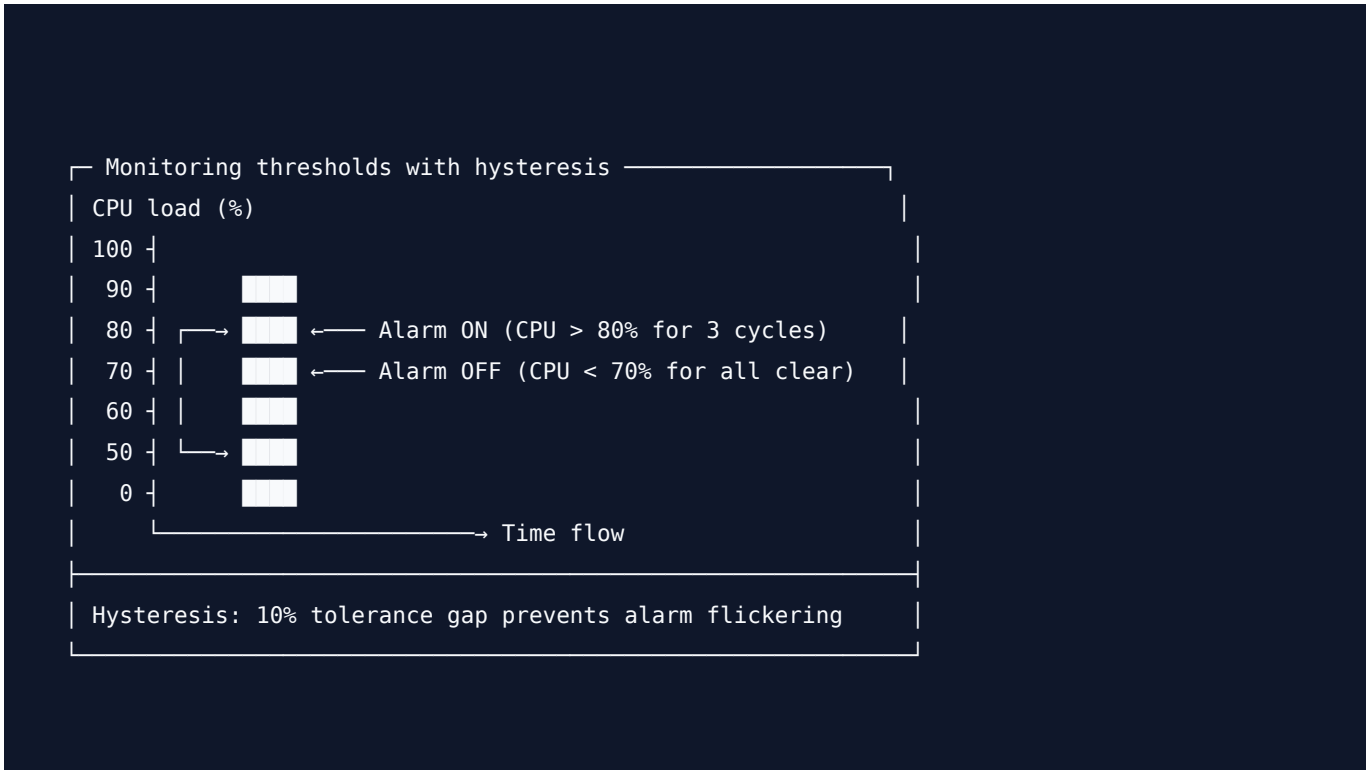
if [ $CURRENT_LOAD -gt $WARNING_THRESHOLD ]; then

# Trigger alarm (only if not already active)
    if [ ! -f $ALERT_FILE ]; then
        send_alert "CPU load critical: ${CURRENT_LOAD}%"
        touch $ALERT_FILE
        echo "$(date): Alarm activated at ${CURRENT_LOAD}%" >> $ALERT_FILE
    fi

elif [ $CURRENT_LOAD -lt $RECOVERY_THRESHOLD ]; then

# Reset alarm (only if active)
    if [ -f $ALERT_FILE ]; then
        send_alert "CPU load normal again: ${CURRENT_LOAD}%"
        rm -f $ALERT_FILE
    fi
fi
```

Hysteresis principle visualized:



Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

Why is this important?:

- Prevents alarm spam with fluctuating values
- Reduces false positives from short spikes
- Professional monitoring always works with hysteresis
- Better user experience for administrators

Advanced monitoring scripts

Memory monitoring:

```
#!/bin/bash
# memory_monitor.sh - RAM monitoring

get_memory_usage() {

# Available memory in percent
    local total=$(grep MemTotal /proc/meminfo | awk '{print $2}')
    local available=$(grep MemAvailable /proc/meminfo | awk '{print $2}')
    local used_percent=$(echo "scale=1; (($total - $available) / $total) * 100" | bc)
    echo $used_percent
}

MEMORY_USAGE=$(get_memory_usage)
THRESHOLD=85

if (( $(echo "$MEMORY_USAGE > $THRESHOLD" | bc -l) )); then
    {
        echo "Memory usage critical: ${MEMORY_USAGE}%"
        echo ""
        echo "=== Memory Details ==="
        free -h
        echo ""
        echo "=== Top Memory Consumers ==="
        ps aux --sort=-%mem | head -10
        echo ""
        echo "=== Swap Status ==="
```

```
    swapon --show
} | mail -s "High memory usage on $(hostname)" root
fi
```

Disk space monitoring:

```
#!/bin/bash
# disk_monitor.sh - Disk space monitoring

check_disk_usage() {

# Check all mounted filesystems
df -h | awk 'NR>1 {
    gsub(/%/, "", $5)
    if ($5 > 90) {
        print "CRITICAL: " $1 " is " $5 "% full (" $4 " free)"
    } else if ($5 > 80) {
        print "WARNING: " $1 " is " $5 "% full (" $4 " free)"
    }
}'
}

DISK_ALERTS=$(check_disk_usage)

if [ -n "$DISK_ALERTS" ]; then
{
    echo "Disk space warning:"
    echo "$DISK_ALERTS"
    echo ""
    echo "=== Full Disk Overview ==="
    df -h
    echo ""
    echo "=== Largest Files in /var/log ==="
    du -sh /var/log/* 2>/dev/null | sort -hr | head -5
} | mail -s "Disk space warning on $(hostname)" root
fi
```

> ⚠ **Common mistake:** Writing monitoring scripts without error handling. If `bc` is not installed or `/proc/loadavg` is not readable, the script fails.

Robust error handling:

```
#!/bin/bash
# robust_monitor.sh - Monitoring with error handling
# Check dependencies
check_dependencies() {
    local missing=""

    for cmd in bc awk mail; do
        if ! command -v $cmd >/dev/null 2>&1; then
            missing="$missing $cmd"
        fi
    done

    if [ -n "$missing" ]; then
        echo "ERROR: Missing commands:$missing" >&2
        exit 1
    fi
}

# Safe load reading
get_load_safe() {
    if [ -r /proc/loadavg ]; then
        cat /proc/loadavg | awk '{print $1}' 2>/dev/null
    else
        echo "0.0"
    fi
}

# Main script
check_dependencies

LOAD=$(get_load_safe)
if [ "$LOAD" = "0.0" ]; then
```

```
    echo "WARNING: Could not read load" >&2
    exit 1
fi
```

> **You will need this later to:** create reliable monitoring systems that work even during problems and warn you in time about critical situations.

Cron jobs for regular checks

Monitoring scripts are only useful if they run regularly. Cron is the standard scheduler in Linux and allows you to automatically run your monitoring scripts at specified times.

Run monitoring scripts on a schedule

Cron syntax basics:

```
# Crontab format:
# Minute Hour Day Month Weekday Command
#   0-59  0-23  1-31  1-12   0-7
# Examples:
# */5 * * * *     - Every 5 minutes
# 0 */2 * * *     - Every 2 hours
# 0 9 * * 1-5     - Daily at 9:00, Monday to Friday
# 30 2 * * 0      - Sundays at 2:30
# 0 0 1 * *       - 1st of every month at midnight
```

Edit and manage crontab:

```
# Edit your own crontab
$ crontab -e

# Show current crontab
$ crontab -l

# Edit root crontab (for system monitoring)
$ sudo crontab -e

# Edit another user's crontab
$ sudo crontab -e -u username
```

□□ Practical example: Set up monitoring crontab

```
# System crontab for monitoring (sudo crontab -e)
# CPU check every 5 minutes
*/5 * * * * /usr/local/bin/cpu_monitor.sh >/dev/null 2>&1

# Memory check every 10 minutes
*/10 * * * * /usr/local/bin/memory_monitor.sh >/dev/null 2>&1

# Disk check every 30 minutes
*/30 * * * * /usr/local/bin/disk_monitor.sh >/dev/null 2>&1

# Full system check daily at 6:00
0 6 * * * /usr/local/bin/daily_system_check.sh

# Log cleanup weekly on Sunday at 3:00
0 3 * * 0 /usr/local/bin/log_cleanup.sh
```

□□ **You will need this later to:** automatically monitor your system without constantly having to enter commands manually.

Log rotation for monitoring data

Monitoring scripts generate their own log files that can fill up the disk without rotation. Here you set up log rotation for your monitoring logs:

Logrotate configuration for monitoring:

```
# Create file: /etc/logrotate.d/monitoring
$ sudo nano /etc/logrotate.d/monitoring

# Content:
/var/log/cpu_monitor.log
/var/log/memory_monitor.log
/var/log/disk_monitor.log
/var/log/system_check.log
{
□daily
□missingok
□rotate 30
□compress
□delaycompress
□notifempty
□create 640 root adm
□postrotate

# Optional: notify monitoring service
□□/bin/kill -HUP $(cat /var/run/rsyslogd.pid 2>/dev/null) 2>/dev/null || true
□endscript
}
```

Understanding the logrotate configuration:

Option	Meaning	Why important?
daily	Daily rotation	Monitoring logs grow fast
rotate 30	Keep for 30 days	Sufficient for trend analysis
compress	Compress old logs	Save disk space
delaycompress	Only compress from 2nd rotation	For running processes
notifempty	Don't rotate empty files	Avoid unnecessary rotations
create 640 root adm	New file with correct permissions	Ensure security

Test logrotate manually:

```
# Test mode (shows what would happen)
sudo logrotate -d /etc/logrotate.d/monitoring

# Forced rotation for testing
sudo logrotate -f /etc/logrotate.d/monitoring

# Check if it worked
$ ls -la /var/log/cpu_monitor.log*
-rw-r----- 1 root adm    1234 Jul 12 16:30 cpu_monitor.log
-rw-r----- 1 root adm    5678 Jul 11 16:30 cpu_monitor.log.1.gz
```

□□ Practical example:

Daily system health report

Here we create a comprehensive script that sends a system health report via email daily:

```
#!/bin/bash
# daily_system_check.sh - Daily system health report
# Configuration
REPORT_FILE="/tmp/system_report_$(date +%Y%m%d).txt"
EMAIL="admin@localhost"
HOSTNAME=$(hostname)

# Create report header
create_header() {
    cat << EOF > $REPORT_FILE
========================================
DAILY SYSTEM HEALTH REPORT
========================================
Hostname: $HOSTNAME
Date: $(date '+%Y-%m-%d %H:%M:%S')
Uptime: $(uptime -p)

EOF
}

# Check CPU status
check_cpu() {
    echo "=== CPU STATUS ===" >> $REPORT_FILE

    # Load Average
    LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
    CPU_CORES=$(nproc)
    LOAD_PERCENT=$(echo "scale=1; $LOAD_1MIN / $CPU_CORES * 100" | bc)

    echo "Load Average: $(cat /proc/loadavg)" >> $REPORT_FILE
    echo "CPU usage: ${LOAD_PERCENT}% (${CPU_CORES} cores)" >> $REPORT_FILE

    # Warning on high load
    if (( $(echo "$LOAD_PERCENT > 80" | bc -l) )); then
        echo "WARNING: High CPU load!" >> $REPORT_FILE
    else
        echo "CPU load normal" >> $REPORT_FILE
    fi

    echo "" >> $REPORT_FILE
```

```
}

# Check memory status
check_memory() {
    echo "=== MEMORY STATUS ===" >> $REPORT_FILE

    # Read memory info
    local total=$(grep MemTotal /proc/meminfo | awk '{print $2}')
    local available=$(grep MemAvailable /proc/meminfo | awk '{print $2}')
    local used_percent=$(echo "scale=1; (($total - $available) / $total) * 100" | bc)

    echo "Memory overview:" >> $REPORT_FILE
    free -h >> $REPORT_FILE
    echo "" >> $REPORT_FILE
    echo "Memory usage: ${used_percent}%" >> $REPORT_FILE

    # Warning on high memory usage
    if (( $(echo "$used_percent > 85" | bc -l) )); then
        echo "WARNING: High memory usage!" >> $REPORT_FILE
        echo "Top 5 memory consumers:" >> $REPORT_FILE
        ps aux --sort=-%mem | head -6 | tail -5 >> $REPORT_FILE
    else
        echo "Memory usage normal" >> $REPORT_FILE
    fi

    echo "" >> $REPORT_FILE
}

# Check disk status
check_disk() {
    echo "=== DISK STATUS ===" >> $REPORT_FILE

    # Disk usage
    df -h >> $REPORT_FILE
    echo "" >> $REPORT_FILE

    # Warnings for full disks
    local disk_warnings=$(df -h | awk 'NR>1 {gsub(/%/, "", $5); if ($5 > 90) print "CRITICAL:
    " $1 " is " $5 "% full"; else if ($5 > 80) print "WARNING: " $1 " is " $5 "% full"}')

    if [ -n "$disk_warnings" ]; then
        echo "$disk_warnings" >> $REPORT_FILE
    else
        echo "All disks have sufficient space" >> $REPORT_FILE
```

```
    fi

    echo "" >> $REPORT_FILE
    }

    # Check service status
    check_services() {
    echo "=== IMPORTANT SERVICES ===" >> $REPORT_FILE

    # List of important services
    local services=("ssh" "cron" "rsyslog" "systemd-journald")

    for service in "${services[@]}"; do
        if systemctl is-active --quiet $service; then
            echo "$service: Running" >> $REPORT_FILE
        else
            echo "$service: Stopped or faulty" >> $REPORT_FILE
        fi
    done

    echo "" >> $REPORT_FILE
    }

    # Recent errors from logs
    check_recent_errors() {
    echo "=== RECENT SYSTEM ERRORS ===" >> $REPORT_FILE

    # Errors from the last 24 hours
    local yesterday=$(date -d 'yesterday' '+%b %d')
    local today=$(date '+%b %d')

    echo "Critical errors from the last 24h:" >> $REPORT_FILE
    grep -E "($yesterday|$today)" /var/log/syslog | grep -iE "(error|critical|failed)" | tail
    -10 >> $REPORT_FILE

    if [ $? -ne 0 ] || [ ! -s /tmp/recent_errors.tmp ]; then
        echo "No critical errors found" >> $REPORT_FILE
    fi

    echo "" >> $REPORT_FILE
    }

    # Security check
    check_security() {
```

```
    echo "=== SECURITY CHECK ===" >> $REPORT_FILE

    # Failed SSH logins from the last 24h
    local failed_logins=$(grep "$(date '+%b %d')" /var/log/auth.log 2>/dev/null | grep "Failed
    password" | wc -l)
    echo "Failed SSH logins today: $failed_logins" >> $REPORT_FILE

    if [ $failed_logins -gt 10 ]; then
        echo "WARNING: Many failed login attempts!" >> $REPORT_FILE
        echo "Top attacker IPs:" >> $REPORT_FILE
        grep "$(date '+%b %d')" /var/log/auth.log 2>/dev/null | grep "Failed password" | awk
    '{print $11}' | sort | uniq -c | sort -nr | head -5 >> $REPORT_FILE
    else
        echo "Normal number of failed logins" >> $REPORT_FILE
    fi

    echo "" >> $REPORT_FILE
    }

    # Report footer
    create_footer() {
    cat << EOF >> $REPORT_FILE
    ========================================
    Report created: $(date '+%Y-%m-%d %H:%M:%S')
    Next report: $(date -d 'tomorrow' '+%Y-%m-%d 06:00')
    ========================================
    EOF
    }

    # Main function
    main() {

    # Create report
    create_header
    check_cpu
    check_memory
    check_disk
    check_services
    check_recent_errors
    check_security
    create_footer

    # Send report via email
    if command -v mail >/dev/null 2>&1; then
```

```
    mail -s "Daily system report: $HOSTNAME" $EMAIL < $REPORT_FILE
    echo "Report sent via email to: $EMAIL"
else
    echo "WARNING: mail command not available. Report saved to: $REPORT_FILE"
fi

# Delete report files older than 7 days
find /tmp -name "system_report_*.txt" -mtime +7 -delete 2>/dev/null
}

# Run script
main
```

Set up cron job for daily report:

```
# Add to root crontab (sudo crontab -e)
0 6 * * * /usr/local/bin/daily_system_check.sh >/dev/null 2>&1

# Or with logging of script execution
0 6 * * * /usr/local/bin/daily_system_check.sh >> /var/log/system_check.log 2>&1
```

Warning: Why monitoring scripts themselves need to be monitored

The problem: What happens when your monitoring scripts fail?

```
#!/bin/bash
# monitor_the_monitor.sh - Meta-monitoring

MONITOR_LOG="/var/log/monitor_status.log"
LAST_RUN_FILE="/tmp/last_monitor_run"
```

```
check_monitor_health() {
    local script_name=$1
    local max_age_minutes=$2
    local script_log=$3

    # Check if the script ran recently
    if [ -f "$LAST_RUN_FILE" ]; then
        local last_run=$(cat $LAST_RUN_FILE)
        local current_time=$(date +%s)
        local age_minutes=$(( (current_time - last_run) / 60 ))

        if [ $age_minutes -gt $max_age_minutes ]; then
            echo "$(date): WARNING: $script_name has not run for $age_minutes minutes!" >>
$MONITOR_LOG
            return 1
        fi
    else
        echo "$(date): WARNING: $script_name has never been executed!" >> $MONITOR_LOG
        return 1
    fi

    # Check if the script produces errors
    if [ -f "$script_log" ]; then
        local recent_errors=$(tail -100 $script_log | grep -i error | wc -l)
        if [ $recent_errors -gt 5 ]; then
            echo "$(date): WARNING: $script_name produces many errors ($recent_errors)!" >>
$MONITOR_LOG
            return 1
        fi
    fi

    return 0
}

# Check monitoring scripts
check_monitor_health "cpu_monitor.sh" 10 "/var/log/cpu_monitor.log"
check_monitor_health "memory_monitor.sh" 15 "/var/log/memory_monitor.log"
check_monitor_health "disk_monitor.sh" 35 "/var/log/disk_monitor.log"
```

Monitoring scripts with heartbeat:

```
#!/bin/bash
# cpu_monitor_with_heartbeat.sh - CPU monitor with heartbeat

HEARTBEAT_FILE="/tmp/cpu_monitor_heartbeat"

# Update heartbeat
echo $(date +%s) > $HEARTBEAT_FILE

# Normal monitoring logic
LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')

# ... rest of the script
```

Monitor cron jobs:

```
# Check cron job status
sudo systemctl status cron

# Show cron logs
grep CRON /var/log/syslog | tail -10

# Check recent cron executions
sudo journalctl -u cron --since "1 hour ago"
```

> **Common mistake:** Using monitoring scripts in cron without absolute paths:

```
# WRONG - PATH is restricted in cron
*/5 * * * * cpu_monitor.sh

# CORRECT - Use absolute paths
*/5 * * * * /usr/local/bin/cpu_monitor.sh

# Or define PATH in crontab
PATH=/usr/local/bin:/usr/bin:/bin
*/5 * * * * cpu_monitor.sh
```

Debugging cron jobs:

```
# Cron job with logging
*/5 * * * * /usr/local/bin/cpu_monitor.sh >> /var/log/cron_debug.log 2>&1

# Test environment in cron
* * * * * env > /tmp/cron_env.txt

# Manual test with cron environment
sudo -u root env -i /bin/bash -c '/usr/local/bin/cpu_monitor.sh'
```

Robust cron job configuration:

```
# Complete crontab with error handling
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin
MAILTO=admin@localhost
```

```
# CPU monitor every 5 minutes
*/5 * * * * /usr/local/bin/cpu_monitor.sh || echo "CPU monitor failed: $(date)" >>
/var/log/cron_errors.log

# Memory monitor every 10 minutes
*/10 * * * * /usr/local/bin/memory_monitor.sh || echo "Memory monitor failed: $(date)" >>
/var/log/cron_errors.log

# Daily system check at 6:00
0 6 * * * /usr/local/bin/daily_system_check.sh || echo "System check failed: $(date)" >>
/var/log/cron_errors.log

# Meta-monitoring every 30 minutes
*/30 * * * * /usr/local/bin/monitor_the_monitor.sh
```

> □□ **You will need this later to:** build reliable monitoring systems that work even during problems and monitor themselves. Also to set up automated monitoring that runs reliably and warns you in time about problems. Cron jobs are the foundation for professional system monitoring and allow you to administer proactively rather than reactively.

Solving performance problems systematically

Systematic diagnosis method

When your Linux system runs slowly, the most common beginner mistake is **guessing instead of diagnosing systematically**. You open various tools at random, look here and there, but without a clear plan. This leads to confusion and wasted time.

As a professional Linux administrator you need a **structured method** to identify and solve performance problems. Here you will learn the proven **USE method** and a practical **diagnosis workflow**.

The USE method: Utilization, Saturation, Errors

The USE method is a systematic approach to performance analysis, developed by Brendan Gregg. It divides every resource into three categories:

`U` - Utilization:

- What percentage of the resource is being used?
- Example: CPU at 80% utilization

`S` - Saturation:

- How much work is waiting for the resource?
- Example: 10 processes waiting for CPU time

`E` - Errors:

- What errors occur with the resource?
- Example: Disk I/O errors

Example: Disk I/O errors

USE method for the most important Linux resources:

Resource	Utilization	Saturation	Errors
CPU	`top` (%CPU), `vmstat` (us+sy)	`vmstat` (r), Load Average	`dmesg`, `/var/log/kern.log`
Memory	`free` (used%), `top` (%MEM)	`vmstat` (si+so), Swap usage	`dmesg` (OOM), `/var/log/kern.log`
Disk I/O	`iostat` (%util)	`iostat` (avgqu-sz)	`dmesg`, `iostat` (error columns)
Network	`iftop`, `netstat`	`netstat` (Recv-Q, Send-Q)	`dmesg`, `/var/log/kern.log`

You will need this later to: analyze performance problems methodically instead of randomly trying various tools.

□□ Practical example:

Step-by-step diagnosis of a slow system

Imagine users are complaining: "The system is very slow today!" Here is your systematic approach:

Step 1: Get a quick overview

```
# First orientation - What is going on?
$ uptime
 16:45:23 up 5 days,  2:15,  3 users,  load average: 4.23, 3.87, 2.45

# Load is high! With 4 CPU cores, 4.23 = 105% utilization
$ nproc
4
```

First finding: System is overloaded (load > number of CPU cores)

Step 2: CPU analysis (USE method)

```
# Utilization: How is the CPU loaded?
$ top -n 1 | head -3
top - 16:45:24 up 5 days,  2:15,  3 users,  load average: 4.23, 3.87, 2.45
Tasks: 234 total,   4 running, 230 sleeping,   0 stopped,   0 zombie
%Cpu(s): 85.2 us, 12.3 sy,  0.0 ni,  1.2 id,  1.3 wa,  0.0 hi,  0.0 si,  0.0 st
```

Analysis of CPU times:

- `85.2% us` (user): Very high! User programs are consuming a lot of CPU
- `12.3% sy` (system): Normal
- `1.2% id` (idle): Very low! CPU has barely any idle time
- `1.3% wa` (iowait): Low, so no I/O problem

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

Intermediate conclusion: CPU problem, not an I/O problem

Step 3: Check saturation

```
# Saturation: How many processes are waiting for CPU?
vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 6  0      0 1234567 123456 2345678   0    0     5    12  234  567 85 12  2  1  0
 7  0      0 1230000 123456 2345678   0    0     3    15  245  578 87 11  1  1  0
 5  0      0 1235000 123456 2345678   0    0     2    10  223  556 83 13  3  1  0
```

Analysis:

- `r` = 6-7: 6-7 processes waiting for CPU (with 4 cores = overload!)
- `b` = 0: No processes waiting for I/O
- `si/so` = 0: No swapping

Intermediate conclusion: CPU saturation confirmed -- too many processes competing for CPU time

Step 4: Identify the culprit

```
# Which processes consume the most CPU?
top -o %CPU -n 1 | head -15
PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1234 user      20   0  567890 123456  45678 R  45.2   1.5   12:34.56 python3
   5678 user      20   0  234567  67890  23456 R  23.4   0.8    8:45.67 node
   9012 user      20   0  345678  89012  34567 R  18.7   1.1    6:23.45 java
   3456 user      20   0  123456  34567  12345 R  12.3   0.4    4:56.78 gcc
   7890 root      20   0   45678  12345   6789 S   3.2   0.2    1:23.45 systemd
```

Analysis:

- `python3`: 45.2% CPU -- main culprit!
- `node`: 23.4% CPU -- second largest consumer
- `java`: 18.7% CPU -- third largest consumer
- `gcc`: 12.3% CPU -- compilation in progress

Step 5: Detailed analysis of the culprits

```
# What is the Python process doing?
ps aux | grep 1234
user     1234  45.2  1.5 567890 123456 ?     R    14:30  12:34 python3
/home/user/scripts/heavy_calculation.py

# Process details
cat /proc/1234/cmdline
python3/home/user/scripts/heavy_calculation.py

# How long has the process been running?
ps -o pid,etime,cmd -p 1234
  PID     ELAPSED CMD
 1234    02:15:23 python3 /home/user/scripts/heavy_calculation.py
```

Findings:

- Python script has been running for over 2 hours
- Performing heavy calculations
- Blocking the system for other users

Step 6: Check errors

```
# Check kernel errors
$ dmesg | tail -20
[12345.678] No errors found

# Check system logs
$ journalctl --since "1 hour ago" -p err
-- No entries --
```

Result: No hardware errors, pure performance problem

Diagnosis workflow as ASCII diagram



Practical diagnosis checklist

Phase 1: Quick overview (30 seconds)

```
# System status
uptime

# Check load average
free -h

# Memory overview
df -h

# Disk space
```

Phase 2: Resource analysis (2 minutes)

```
# CPU analysis
top -n 1 | head -10

# Top processes
vmstat 1 3

# CPU saturation
# Memory analysis
free -h

# Memory details
ps aux --sort=-%mem | head -10

# Memory hogs
# I/O analysis
iostat -x 1 3
```

```
# Disk performance
iotop -o -n 3

# I/O-intensive processes
```

Phase 3: Detailed analysis (5 minutes)

```
# Process details
ps aux | grep <PID>

# Process information
cat /proc/<PID>/cmdline

# Full command line
ls -la /proc/<PID>/fd/

# Open files
# System logs
journalctl --since "1 hour ago" -p warning
dmesg | tail -20
```

> **Typical mistake:** Treating symptoms instead of causes

Common beginner mistake:

```
# WRONG - treating the symptom
sudo killall python3

# Kill all Python processes
```

```
sudo reboot

# Restart the system
```

Why this is problematic:

- Data loss: Running work is lost
- No solution: Problem recurs
- No learning effect: Cause remains unknown

RIGHT -- analyze the cause:

```
# 1. Analyze the process
ps aux | grep 1234
cat /proc/1234/cmdline
ls -la /proc/1234/

# 2. Understand process behavior
strace -p 1234 -c

# Analyze system calls
lsof -p 1234

# Check open files
# 3. Contact the user
who

# Who is logged in?
w

# What are the users doing?
# 4. Coordinated solution
kill -TERM 1234

# Graceful termination
```

```
kill -KILL 1234

# Only if necessary
```

Advanced diagnosis techniques

CPU profiling for complex problems:

```
# Track CPU usage over time
pidstat -p 1234 1 10

# Process 1234, 10 measurements
# All threads of a process
top -H -p 1234

# Thread view
# Check CPU affinity
taskset -p 1234

# Which CPU cores is the process running on?
```

Memory leak diagnosis:

```
# Memory usage over time
while true; do
    ps -o pid,vsz,rss,comm -p 1234
    sleep 10
done

# Analyze memory maps
cat /proc/1234/smaps | grep -E "(Size|Rss|Pss)"
```

I/O pattern analysis:

```
# Which files are being accessed?
lsof -p 1234

# Understand I/O patterns
strace -p 1234 -e read,write,open,close -f
```

Documentation of the diagnosis

Why documentation is important:

- Recurring problems: Solution is known
- Trend analysis: Are problems accumulating?
- Team communication: Colleagues can help
- Compliance: Evidence for audits

Simple diagnosis documentation:

```
#!/bin/bash
# diagnose_log.sh - Document the diagnosis

LOGFILE="/var/log/performance_diagnosis.log"
TIMESTAMP=$(date '+%Y-%m-%d %H:%M:%S')

{
echo "=== PERFORMANCE DIAGNOSIS $TIMESTAMP ==="
echo "Problem: $1"
echo "User: $USER"
echo ""
echo "System status:"
```

```
    uptime
    free -h
    echo ""
    echo "Top processes:"
    ps aux --sort=-%cpu | head -10
    echo ""
    echo "Solution: $2"
    echo "Status: $3"
    echo "=========================="
    echo ""
} >> $LOGFILE

# Usage:
# ./diagnose_log.sh "System slow" "Python script terminated" "Resolved"
```

> ⚠ **Important note:** Before you kill processes, always try to find out what they are doing and whether they are important. A kill -9 can destroy data or interrupt important system functions.

Safe process termination:

```
# 1. Graceful termination (SIGTERM)
kill -TERM 1234
sleep 10

# 2. Check if terminated
ps -p 1234 >/dev/null && echo "Still running" || echo "Terminated"

# 3. Only if necessary: Force termination (SIGKILL)
kill -KILL 1234
```

□□ **You will need this later to:** diagnose performance problems systematically and professionally without damaging the system or losing important data. Also to solve performance problems methodically and purposefully as a Linux administrator instead of randomly trying various tools. This systematic approach distinguishes professional administrators from beginners and leads to faster, better solutions.

Practical exercises and troubleshooting

Complete system health check

As a Linux administrator you need a structured routine to regularly check the state of your system. A systematic system health check helps you identify problems before they become critical and gives you a complete overview of system health.

5-minute routine check

You should perform this quick check daily or when you suspect problems. It gives you an overview of the most important system metrics in just a few minutes:

```
#!/bin/bash
# quick_health_check.sh - 5-minute system check

echo "========================================"
echo "QUICK SYSTEM HEALTH CHECK"
echo "========================================"
echo "Hostname: $(hostname)"
echo "Date: $(date '+%Y-%m-%d %H:%M:%S')"
echo "Uptime: $(uptime -p)"
echo ""

# 1. Load Average and CPU
echo "=== CPU STATUS ==="
```

```
LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
CPU_CORES=$(nproc)
LOAD_PERCENT=$(echo "scale=1; $LOAD_1MIN / $CPU_CORES * 100" | bc)

echo "Load Average: $(cat /proc/loadavg)"
echo "CPU cores: $CPU_CORES"
echo "CPU usage: ${LOAD_PERCENT}%"

if (( $(echo "$LOAD_PERCENT > 80" | bc -l) )); then
    echo "⚠ WARNING: High CPU load!"
    echo "Top 3 CPU consumers:"
    ps aux --sort=-%cpu | head -4 | tail -3
else
    echo "✓ CPU load normal"
fi
echo ""

# 2. Memory status
echo "=== MEMORY STATUS ==="
free -h
echo ""

# Calculate memory usage
TOTAL_MEM=$(grep MemTotal /proc/meminfo | awk '{print $2}')
AVAIL_MEM=$(grep MemAvailable /proc/meminfo | awk '{print $2}')
USED_PERCENT=$(echo "scale=1; (($TOTAL_MEM - $AVAIL_MEM) / $TOTAL_MEM) * 100" | bc)

echo "Memory usage: ${USED_PERCENT}%"

if (( $(echo "$USED_PERCENT > 85" | bc -l) )); then
    echo "⚠ WARNING: High memory usage!"
    echo "Top 3 memory consumers:"
    ps aux --sort=-%mem | head -4 | tail -3
else
    echo "✓ Memory usage normal"
fi

# Swap check
SWAP_USED=$(free | grep Swap | awk '{print $3}')
if [ "$SWAP_USED" -gt 0 ]; then
    echo "⚠ WARNING: System using swap ($SWAP_USED KB)"
else
    echo "✓ No swap used"
fi
```

```
    echo ""

    # 3. Disk status
    echo "=== DISK STATUS ==="
    df -h
    echo ""

    # Disk warnings
    DISK_WARNINGS=$(df -h | awk 'NR>1 {
    gsub(/%/, "", $5)
    if ($5 > 95) {
    print "CRITICAL: " $1 " is " $5 "% full"
    } else if ($5 > 85) {
    print "⚠WARNING: " $1 " is " $5 "% full"
    }
    }')

    if [ -n "$DISK_WARNINGS" ]; then
    echo "$DISK_WARNINGS"
    else
    echo " All disks have sufficient space"
    fi
    echo ""

    # 4. Important services
    echo "=== SERVICE STATUS ==="
    SERVICES=("ssh" "cron" "rsyslog" "systemd-journald")

    for service in "${SERVICES[@]}"; do
    if systemctl is-active --quiet $service 2>/dev/null; then
    echo "$service: Active"
    else
    echo "$service: Inactive or not found"
    fi
    done
    echo ""

    # 5. Basic network check
    echo "=== NETWORK STATUS ==="

    # Ping Google DNS (if internet available)
    if ping -c 1 8.8.8.8 >/dev/null 2>&1; then
    echo " Internet connection: OK"
    else
```

```
    echo "⚠ No internet connection or DNS problem"
    fi

    # Check listening ports
    LISTENING_PORTS=$(ss -tlnp | grep LISTEN | wc -l)
    echo "Listening ports: $LISTENING_PORTS"
    echo ""

    # 6. Recent critical events
    echo "=== RECENT CRITICAL EVENTS ==="
    RECENT_ERRORS=$(journalctl --since "1 hour ago" -p err --no-pager -q | wc -l)
    if [ "$RECENT_ERRORS" -gt 0 ]; then
    echo "⚠ $RECENT_ERRORS errors in the last hour"
    echo "Last 3 errors:"
    journalctl --since "1 hour ago" -p err --no-pager -q | tail -3
    else
    echo "✓ No critical errors in the last hour"
    fi
    echo ""

    echo "========================================"
    echo "SYSTEM HEALTH CHECK COMPLETED"
    echo "========================================"
```

> **You will need this later to:** quickly get an overview of the system state without having to call various tools individually.

Weekly deep analysis

Once a week you should perform a more in-depth analysis that also reveals trends and less obvious problems:

```
    #!/bin/bash
    # weekly_deep_check.sh - Weekly deep analysis
```

```
REPORT_FILE="/tmp/weekly_system_report_$(date +%Y%m%d).txt"

{
echo "========================================"
echo "WEEKLY SYSTEM DEEP ANALYSIS"
echo "========================================"
echo "Hostname: $(hostname)"
echo "Date: $(date '+%Y-%m-%d %H:%M:%S')"
echo "Kernel: $(uname -r)"
echo "Distribution: $(lsb_release -d 2>/dev/null | cut -f2 || echo 'Unknown')"
echo ""

# 1. Advanced CPU analysis
echo "=== ADVANCED CPU ANALYSIS ==="
echo "CPU info:"
lscpu | grep -E "(Model name|CPU(s)|Thread|Core|Socket)"
echo ""

echo "Load trend of the last 15 minutes:"
uptime
echo ""

echo "CPU time distribution:"
vmstat 1 3 | tail -1
echo ""

# 2. Memory deep analysis
echo "=== MEMORY DEEP ANALYSIS ==="
echo "Detailed memory info:"
free -h
echo ""

echo "Memory fragmentation:"
cat /proc/buddyinfo | head -3
echo ""

echo "Top 10 memory consumers:"
ps aux --sort=-%mem | head -11
echo ""

# 3. I/O performance analysis
echo "=== I/O PERFORMANCE ANALYSIS ==="
echo "Disk performance (3 measurements):"
```

```
iostat -x 1 3 | tail -n +4
echo ""

echo "Inode usage:"
df -i
echo ""

# 4. Network analysis
echo "=== NETWORK ANALYSIS ==="
echo "Network interfaces:"
ip addr show | grep -E "(inet |UP|DOWN)"
echo ""

echo "Listening services:"
ss -tlnp | grep LISTEN
echo ""

echo "Active connections (Top 10):"
ss -tn | awk 'NR>1 {print $4}' | cut -d: -f1 | sort | uniq -c | sort -nr | head -10
echo ""

# 5. Security analysis
echo "=== SECURITY ANALYSIS ==="
echo "Failed SSH logins (last 7 days):"
FAILED_LOGINS=$(grep "Failed password" /var/log/auth.log* 2>/dev/null | grep "$(date -d '7
days ago' '+%b')" | wc -l)
echo "Count: $FAILED_LOGINS"

if [ "$FAILED_LOGINS" -gt 50 ]; then
    echo "⚠WARNING: Many failed login attempts!"
    echo "Top attacker IPs:"
    grep "Failed password" /var/log/auth.log* 2>/dev/null | grep "$(date -d '7 days ago'
'+%b')" | awk '{print $11}' | sort | uniq -c | sort -nr | head -5
fi
echo ""

echo "Sudo activities (last 7 days):"
SUDO_COUNT=$(grep "sudo.*COMMAND" /var/log/auth.log* 2>/dev/null | grep "$(date -d '7 days
ago' '+%b')" | wc -l)
echo "Number of sudo commands: $SUDO_COUNT"
echo ""

# 6. System updates
echo "=== SYSTEM UPDATE STATUS ==="
```

```
    if command -v apt >/dev/null 2>&1; then
        echo "Available updates (Debian/Ubuntu):"
        apt list --upgradable 2>/dev/null | wc -l
    elif command -v dnf >/dev/null 2>&1; then
        echo "Available updates (Fedora/RHEL):"
        dnf check-update -q | wc -l
    fi
    echo ""

    # 7. Log analysis
    echo "=== LOG ANALYSIS ==="
    echo "Kernel warnings (last 7 days):"
    dmesg -T | grep -i "warning|error" | tail -10
    echo ""

    echo "Systemd service errors (last 7 days):"
    journalctl --since "7 days ago" -p err --no-pager -q | tail -10
    echo ""

    # 8. Resource trends
    echo "=== RESOURCE TRENDS ==="
    echo "Largest directories in /var:"
    du -sh /var/* 2>/dev/null | sort -hr | head -10
    echo ""

    echo "Oldest processes:"
    ps -eo pid,etime,comm --sort=etime | tail -10
    echo ""

    echo "========================================="
    echo "DEEP ANALYSIS COMPLETED"
    echo "Report saved to: $REPORT_FILE"
    echo "========================================="

    } | tee $REPORT_FILE

    # Send report by email (if configured)
    if command -v mail >/dev/null 2>&1; then
        mail -s "Weekly system report: $(hostname)" root < $REPORT_FILE
        echo "Report sent by email."
    fi

    # Delete old reports (older than 30 days)
    find /tmp -name "weekly_system_report_*.txt" -mtime +30 -delete 2>/dev/null
```

Automating the weekly analysis:

```
# Add to root crontab (sudo crontab -e)
0 7 * * 1 /usr/local/bin/weekly_deep_check.sh >/dev/null 2>&1

# Every Monday at 7:00 AM
```

Documentation of results

Why documentation is important:

- Trend detection: Problems often develop gradually
- Baseline values: What is "normal" for your system?
- Compliance: Evidence for audits and reports
- Team communication: Colleagues can follow developments

Creating a simple monitoring dashboard:

```
#!/bin/bash
# create_dashboard.sh - Simple text dashboard

DASHBOARD_FILE="/var/www/html/system_dashboard.html"
TEMP_FILE="/tmp/dashboard_temp.html"

# HTML header
cat << 'EOF' > $TEMP_FILE
<!DOCTYPE html>
<html>
<head>
<title>System Dashboard</title>
<meta charset="UTF-8">
<meta http-equiv="refresh" content="300">
```

```
    <style>
        body { font-family: monospace; margin: 20px; background: #f5f5f5; }
        .container { max-width: 1200px; margin: 0 auto; }
        .section { background: white; margin: 10px 0; padding: 15px; border-radius: 5px; }
        .ok { color: green; }
        .warning { color: orange; }
        .critical { color: red; }
        .header { background: #333; color: white; text-align: center; padding: 20px; }
        pre { background: #f8f8f8; padding: 10px; overflow-x: auto; }
    </style>
    </head>
    <body>
    <div class="container">
        <div class="header">
            <h1>System Dashboard</h1>
            <p>Hostname: $(hostname) | Last update: $(date)</p>
        </div>
    EOF

    # Add system status
    cat << EOF >> $TEMP_FILE
        <div class="section">
            <h2>System Overview</h2>
            <pre>$(uptime)</pre>
            <pre>$(free -h)</pre>
        </div>

        <div class="section">
            <h2>CPU Status</h2>
            <pre>$(top -bn1 | head -5)</pre>
        </div>

        <div class="section">
            <h2>Disk Status</h2>
            <pre>$(df -h)</pre>
        </div>

        <div class="section">
            <h2>Top Processes (CPU)</h2>
            <pre>$(ps aux --sort=-%cpu | head -10)</pre>
        </div>

        <div class="section">
            <h2>Top Processes (Memory)</h2>
```

```
    <pre>$(ps aux --sort=-%mem | head -10)</pre>
    </div>

    <div class="section">
    <h2>Network Status</h2>
    <pre>$(ss -tlnp | grep LISTEN | head -10)</pre>
    </div>

    <div class="section">
    <h2>Recent System Events</h2>
    <pre>$(journalctl --since "1 hour ago" -p warning --no-pager -q | tail -10)</pre>
    </div>
    EOF

    # HTML footer
    cat << 'EOF' >> $TEMP_FILE
    </div>
    </body>
    </html>
    EOF

    # Activate dashboard
    sudo mv $TEMP_FILE $DASHBOARD_FILE
    sudo chown www-data:www-data $DASHBOARD_FILE 2>/dev/null || true
    echo "Dashboard created: $DASHBOARD_FILE"
```

Automatically update the dashboard:

```
    # Cron job for dashboard update (every 5 minutes)
    */5 * * * * /usr/local/bin/create_dashboard.sh >/dev/null 2>&1
```

Collecting monitoring data in CSV format:

```
#!/bin/bash
# collect_metrics.sh - Collect metrics for trend analysis

CSV_FILE="/var/log/system_metrics.csv"
TIMESTAMP=$(date '+%Y-%m-%d %H:%M:%S')

# Create header (only on first run)
if [ ! -f "$CSV_FILE" ]; then
    echo "Timestamp,Load_1min,CPU_Percent,Memory_Percent,Disk_Root_Percent,Processes_Total,Processes_Running" > $CSV_FILE
fi

# Collect metrics
LOAD_1MIN=$(cat /proc/loadavg | awk '{print $1}')
CPU_CORES=$(nproc)
CPU_PERCENT=$(echo "scale=1; $LOAD_1MIN / $CPU_CORES * 100" | bc)

TOTAL_MEM=$(grep MemTotal /proc/meminfo | awk '{print $2}')
AVAIL_MEM=$(grep MemAvailable /proc/meminfo | awk '{print $2}')
MEMORY_PERCENT=$(echo "scale=1; (($TOTAL_MEM - $AVAIL_MEM) / $TOTAL_MEM) * 100" | bc)

DISK_PERCENT=$(df / | awk 'NR==2 {gsub(/%/, "", $5); print $5}')

PROCESSES_TOTAL=$(ps aux | wc -l)
PROCESSES_RUNNING=$(ps aux | awk '$8 ~ /^R/ {count++} END {print count+0}')

# Write data to CSV
echo "$TIMESTAMP,$LOAD_1MIN,$CPU_PERCENT,$MEMORY_PERCENT,$DISK_PERCENT,$PROCESSES_TOTAL,$PROCESSES_RUNNING" >> $CSV_FILE

# Delete old data (older than 90 days)
if [ $(wc -l < $CSV_FILE) -gt 12960 ]; then
    # 90 days * 24h * 6 (every 10 min)
    tail -12960 $CSV_FILE > ${CSV_FILE}.tmp && mv ${CSV_FILE}.tmp $CSV_FILE
fi
```

Analyzing CSV data:

```
# Average CPU load of the last 24h
$ tail -144 /var/log/system_metrics.csv | awk -F, '{sum+=$3; count++} END {print "Average
CPU load: " sum/count "%"}'

# Highest memory usage of the last week
$ tail -1008 /var/log/system_metrics.csv | awk -F, 'BEGIN{max=0} {if($4>max) max=$4} END
{print "Highest memory usage: " max "%"}'

# Trend analysis (rising/falling)
$ tail -20 /var/log/system_metrics.csv | awk -F, '{print $3}' | awk '{if(NR==1) prev=$1;
else {if($1>prev) up++; else down++; prev=$1}} END {print "CPU trend: "
(up>down?"rising":"falling")}'
```

 Important note: Perform system health checks without baseline values. You need to know what is "normal" for your system to recognize deviations.

Determining baseline values:

```
#!/bin/bash
# establish_baseline.sh - Determine baseline values

echo "Determining baseline values for $(hostname)..."
echo "Collecting data over 1 hour (every 5 minutes)..."

BASELINE_FILE="/var/log/system_baseline.txt"

{
echo "========================================="
echo "SYSTEM BASELINE for $(hostname)"
echo "Period: $(date) to $(date -d '+1 hour')"
echo "========================================="
```

```
    echo ""
} > $BASELINE_FILE

# 12 measurements over 1 hour
for i in {1..12}; do
    {
        echo "=== Measurement $i/12 - $(date) ==="
        echo "Load: $(cat /proc/loadavg | awk '{print $1}')"
        echo "CPU: $(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d% -f1)"
        echo "Memory: $(free | grep Mem | awk '{printf "%.1f", ($3/$2)*100}')"
        echo "Disk: $(df / | awk 'NR==2 {gsub(/%/, "", $5); print $5}')"
        echo ""
    } >> $BASELINE_FILE

    echo "Measurement $i/12 completed..."
    [ $i -lt 12 ] && sleep 300

    # Wait 5 minutes
done

# Baseline summary
{
    echo "========================================="
    echo "BASELINE SUMMARY"
    echo "========================================="

    # Calculate averages
    AVG_LOAD=$(grep "Load:" $BASELINE_FILE | awk '{sum+=$2; count++} END {printf "%.2f",
sum/count}')
    AVG_MEMORY=$(grep "Memory:" $BASELINE_FILE | awk '{sum+=$2; count++} END {printf "%.1f",
sum/count}')
    AVG_DISK=$(grep "Disk:" $BASELINE_FILE | awk '{sum+=$2; count++} END {printf "%.1f",
sum/count}')

    echo "Average load: $AVG_LOAD"
    echo "Average memory usage: $AVG_MEMORY%"
    echo "Average disk usage: $AVG_DISK%"
    echo ""
    echo "Recommended thresholds:"
    echo "Load warning: $(echo "$AVG_LOAD * 1.5" | bc)"
    echo "Memory warning: $(echo "$AVG_MEMORY + 20" | bc)%"
    echo "Disk warning: $(echo "$AVG_DISK + 15" | bc)%"
    echo ""
    echo "Baseline created: $(date)"
```

```
echo "========================================"
} >> $BASELINE_FILE

echo "Baseline determination completed!"
echo "Results in: $BASELINE_FILE"
```

> **You will need this later to:** make informed decisions about thresholds and distinguish real problems from normal fluctuations. Also to set up professional system monitoring that helps you detect problems early and continuously monitor the health of your Linux systems. These structured checks are the foundation for reliable IT infrastructures.

Most important monitoring commands

Area	Command	Purpose	Key parameters
CPU	`top`	Live process monitoring	`M` (Memory), `P` (CPU), `k` (kill)
CPU	`htop`	User-friendly alternative	`F9` (kill), `F5` (tree view)
CPU	`uptime`	Display load average	--
Memory	`free -h`	Memory overview	`-h` (human readable)
Memory	`vmstat 1 3`	Virtual memory	`1 3` (1 sec, 3 measurements)
Disk	`df -h`	Disk space	`-h` (human readable), `-i` (inodes)
Disk	`iostat -x 1 3`	I/O performance	`-x` (extended), `1 3` (interval)
Network	`ss -tlnp`	Network connections	`-t` (TCP), `-l` (listen), `-n` (numeric)
Network	`netstat -tlnp`	Alternative to ss	Same parameters
Logs	`journalctl -f`	Live log following	`-f` (follow), `-u` (unit), `--since`

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

Area	Command	Purpose	Key parameters
Logs	`grep -i error /var/log/syslog`	Log searching	`-i` (ignore case), `-C 3` (context)

Emergency troubleshooting checklist

For performance problems -- step by step

Quick overview (30 seconds)

```
uptime

# Check load average
free -h

# Memory status
df -h

# Disk space
```

Identify main problem (2 minutes)

```
top -n 1

# Show top processes
vmstat 1 3

# CPU/Memory saturation
iostat -x 1 3

# I/O performance
```

Find the culprit (2 minutes)

```
ps aux --sort=-%cpu | head -10

# CPU hogs
ps aux --sort=-%mem | head -10

# Memory hogs
iotop -o

# I/O-intensive processes
```

Check logs (1 minute)

```
journalctl --since "1 hour ago" -p err
dmesg | tail -20
```

Command Reference (Cheatsheet)

Command / Syntax	Category	Function & Description
`top` / `htop`	CPU & Processes	Interactive process monitor with CPU & memory usage
`free -h`	Memory	Shows used, cached, and available RAM
`vmstat 1 5`	Performance	Outputs 5 reports on processes, memory, swap & I/O per second

Command / Syntax	Category	Function & Description
`iostat -xz 1`	Disk I/O	Detailed I/O statistics per block device with wait times (`%util` , `await`)
`iotop -o`	Disk I/O	Shows processes with active disk activity in real time
`df -h` / `df -i`	File system	Shows disk space and inode usage of all mounts
`du -sh *	sort -h`	Disk space	Lists directory sizes sorted
`journalctl -u <Unit> -f`	Logs	Follows log entries of a systemd service live (`follow`)
`journalctl -p err..emerg`	Logs	Filters system journal by errors and emergencies
`dmesg -T --level=err,warn`	Kernel	Shows kernel error messages with readable timestamps
`iftop` / `nload`	Network	Real-time bandwidth monitoring per interface
`ss -s`	Network	Outputs aggregated socket state statistics

Further Resources

Resource	Description
Linux Performance Tools (Brendan Gregg)	Reference for Linux tracing and performance methodology
systemd journalctl Manual	Official manual for log analysis with journalctl
Linux Administration #8: Virtualization	Previous part: KVM, Libvirt, virsh, virt-install & containers
Linux Administration #3: Processes	Fundamental knowledge on process states, signals & cgroups
Command-line processor on Linux	Architecture of shell, TTY, pipes and I/O streams

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring

Conclusion

System monitoring and performance monitoring are the key to proactive operation of stable Linux infrastructures. By being able to precisely analyze CPU bottlenecks, memory pressure (OOM), I/O bottlenecks, and network anomalies with standard tools like `vmstat`, `iostat`, `free`, `iotop`, and `journalctl`, you can methodically resolve disruptions before they affect the availability of your services.

> **Practice tip:** Never rely on a single tool for performance problems. Always combine `vmstat 1` for the overall view (run queue, context switches, swap), `iostat -xz 1` for disk wait times (`%util`), and `journalctl -p err` for error messages to isolate the actual cause beyond doubt.

In the final tenth part of our administration series we will address the lifecycle of the system:

Continue with: Linux Administration #10: Boot Management and System Start

Source: https://admindocs.de/en/linux-administration/linux-administration-system-monitoring-and-performance-monitoring