# **🔧 PostgreSQL 17 Kernel Tuning Guide: Managing System Parameters for Optimal Performance**

[](https://medium.com/@jramcloud1?source=post_page---byline--fe097de1dcdb---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--fe097de1dcdb---------------------------------------)

Following

20 min read

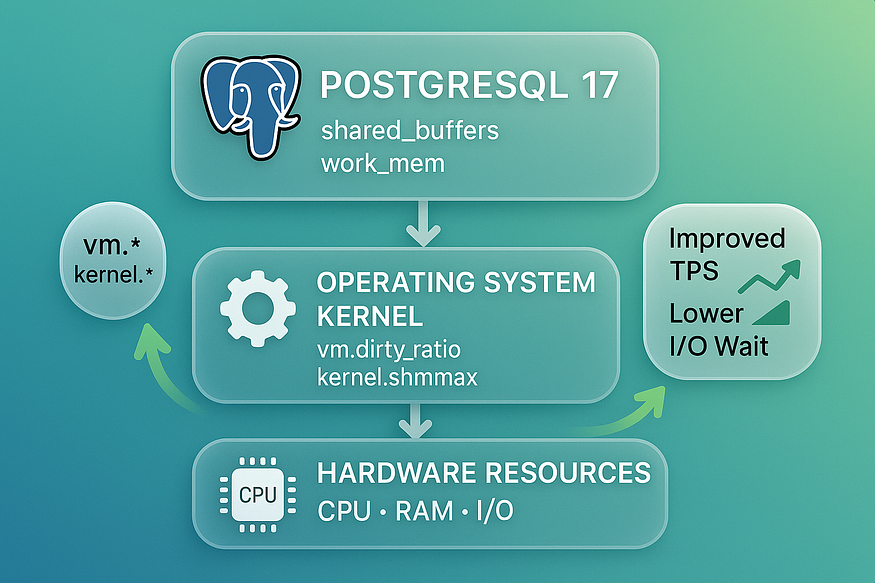
·

Jun 18, 2025

59

2

Press enter or click to view image in full size



PostgreSQL is a highly performant database system, but to fully leverage its power — especially at scale — proper tuning at the ****kernel level**** is critical. PostgreSQL 17 heavily relies on shared memory, semaphores, and system resource limits to operate smoothly under high loads.

In this guide, we’ll dive into how to configure kernel parameters on Linux to optimize PostgreSQL 17, ensure stability, and avoid common performance pitfalls.

## **⚙ Why Kernel Tuning Matters for PostgreSQL**

PostgreSQL is an advanced, high-performance relational database system that’s designed to support multiple users, complex workloads, and concurrent operations. But in order for PostgreSQL to run efficiently — especially in ****production or high-load environments**** — it relies heavily on ****Linux kernel-level system resources****.

If these system resources are not tuned properly, PostgreSQL can suffer from ****slow performance, random crashes, or complete startup failure****. That’s why ****kernel tuning**** is not just a best practice — it’s often essential.

Let’s take a closer look at what this means and why it matters.

## **🧠 How PostgreSQL Uses the Linux Kernel**

PostgreSQL is ****not a single-process**** application. When a user connects to the database, PostgreSQL ****forks a new backend process****. So, if you have 100 users connected, PostgreSQL may have over 100 backend processes running — each requiring access to shared memory, file descriptors, semaphores, and CPU time.

Here are the ****key system resources PostgreSQL depends on****:

### **1. Shared Memory Segments**

PostgreSQL uses shared memory (also called shm) to allow all backend processes to ****access common areas****, like:

* The buffer pool (shared\_buffers)
* WAL buffers
* Lock management memory
* Background writer memory

This shared memory is allocated at startup. If Linux doesn’t allow enough shared memory (based on settings like kernel.shmmax and kernel.shmall), PostgreSQL ****won’t even start**** or will crash with memory-related errors.

### **2. Semaphores**

Semaphores are used for ****inter-process coordination****. They help PostgreSQL manage locks and synchronization between the multiple backend processes running simultaneously.

Linux kernel parameters like kernel.sem define the ****maximum number of semaphores and semaphore arrays**** available on the system. If these are set too low, PostgreSQL might:

* Refuse new connections
* Fail to acquire locks
* Crash during operations that require heavy coordination

### **3. File Descriptors**

PostgreSQL reads and writes many files:

* Data files (tables, indexes, WAL segments)
* Log files
* Temporary files
* Client connection sockets

Each open file or socket requires a ****file descriptor****, and the operating system limits how many file descriptors a single process can open (controlled by nofile). If PostgreSQL exceeds this limit, it won’t be able to accept new connections, create temp files, or write logs.

### **4. Process and Virtual Memory Limits**

PostgreSQL uses many child processes for:

* Autovacuum workers
* Background writer
* Checkpointer
* WAL writer
* Replication processes

The system’s process limits (nproc) and memory behavior (like vm.swappiness) can impact how PostgreSQL scales, especially under ****high concurrency**** or ****large data volume**** scenarios.

## **⚠️ What Happens When Kernel Settings Are Too Low?**

If the Linux kernel is not tuned to handle PostgreSQL’s requirements, you may run into issues like:

* 🚫 ****Startup failures****  
  Example: “FATAL: could not create shared memory segment” error due to low shmmax.
* 🐌 ****Sluggish performance****  
  When PostgreSQL can’t get enough memory or semaphores, background tasks and queries may slow down significantly.
* 🔁 ****Connection failures****  
  If file descriptors or semaphore arrays are exhausted, new client connections will be rejected.
* 💥 ****Crashes under load****  
  During high activity (e.g., ETL jobs, bulk inserts, reporting), PostgreSQL may become unstable and crash without warning.

## **🎯 The Goal of Kernel Tuning**

The goal is simple but critical:

Align your **Linux kernel settings** with your **PostgreSQL configuration** so that the OS can support the database’s demands without bottlenecks.

That means:

* Allocating enough ****shared memory**** to match shared\_buffers
* Providing enough ****semaphores**** to handle concurrent backend activity
* Increasing the ****open file limit**** so PostgreSQL doesn’t hit “too many open files” errors
* Tuning ****swappiness**** to prevent PostgreSQL memory from being swapped to disk under pressure

By making these adjustments, you:

* 🟢 Enable PostgreSQL to start and run smoothly
* ⚡ Boost performance under load
* 🛡️ Avoid system-level crashes or stalls
* 🔄 Reduce the need for emergency interventions or restarts

## **🧠 Think of Kernel Tuning Like Database Infrastructure Insurance**

PostgreSQL’s performance isn’t just about indexes, queries, or configuration files — it starts at the ****OS level****. A misaligned kernel setting can ruin all the work you’ve done tuning PostgreSQL itself.

So tuning the kernel is like preparing the foundation of a house. Without it:

* Even the best database configuration may fail.
* Performance tweaks won’t matter if the OS can’t support them.

## **🧩 Summary**

* PostgreSQL uses ****multiple backend processes****, and relies on the Linux kernel to manage memory, semaphores, and file access.
* If the system’s resource limits are too low, PostgreSQL might ****fail to start****, become ****unreliable****, or ****crash under heavy use****.
* Proper ****kernel tuning**** ensures that PostgreSQL has what it needs to run efficiently — especially in high-load or enterprise environments.
* It is a ****necessary step**** when preparing PostgreSQL for production.

## **🧪 1️⃣ Check Current Kernel Parameters**

Before tuning your Linux system for PostgreSQL performance, it’s critical to ****check the current kernel parameter values****. Think of this as the ****diagnostic step**** — just like checking your car’s oil level or tire pressure before a long drive.

PostgreSQL depends on the operating system for shared memory, semaphores, and process handling. If system resource limits are too low, the database may crash, refuse to start, or struggle under heavy workloads. So, before making any changes, you need to know where your system currently stands.

## **🔍 Why This Step Is Important**

PostgreSQL is a ****multi-process**** database engine. It creates multiple background and user-facing processes — and all of them need access to ****shared memory**** and ****system-level synchronization tools**** (like semaphores). These resources are tightly controlled by the ****Linux kernel****, and if PostgreSQL exceeds the limits, you’ll encounter errors like:

* FATAL: could not create shared memory segment
* FATAL: semget: No space left on device
* FATAL: could not fork new process

By checking the current kernel parameter values, you’re answering questions like:

* Is my system currently configured to support PostgreSQL’s workload?
* Do I need to increase shared memory limits before restarting PostgreSQL?
* Are semaphore limits high enough to allow for hundreds of concurrent connections?

## **🖥️ How to Check Kernel Parameter Values**

You can view kernel parameter values using the sysctl command — a tool that reads and modifies system configuration at runtime. We’ll use it to inspect memory and semaphore-related settings.

Run the following commands in your terminal:

### **🔹 Check Shared Memory Maximum Size**

sudo sysctl -a | grep -i shmmax

* shmmax controls the ****maximum size (in bytes)**** of a single shared memory segment.
* PostgreSQL needs this value to be ****at least as large**** as the value of shared\_buffers.
* If shmmax is too low, PostgreSQL may fail to start.

### **🔹 Check Shared Memory Minimum Size**

sudo sysctl -a | grep -i shmmin

* shmmin is the ****minimum size**** of shared memory segments.
* It’s rarely changed, but it’s good to verify it’s not set to an unexpectedly high value.

### **🔹 Check Shared Memory Allocation Limit**

sudo sysctl -a | grep -i shmall

* shmall defines the ****total amount of shared memory**** available on the system (in memory pages).
* It determines the combined size of all shared memory segments.
* PostgreSQL needs this to be large enough to support shared\_buffers plus other memory regions.

### **🔹 Check Semaphore Set Count Limit**

sudo sysctl -a | grep -i semmni

* semmni defines the ****maximum number of semaphore sets**** the system can allocate.
* Each backend process uses semaphores to synchronize with others, so this must be high enough to handle your expected number of users and background processes.

### **🔹 Check Total Semaphores Available**

sudo sysctl -a | grep -i semmns

* semmns is the ****total number of semaphores**** available across all sets.
* This directly affects how many processes PostgreSQL can support concurrently.

## **📋 Example Output**

Let’s say you run the above commands and get the following:

kernel.shmmax = 68719476736 # 64 GB  
kernel.shmall = 16777216  
kernel.sem = 250 32000 32 128

This tells you:

* shmmax is 64 GB, which is good for a large shared\_buffers setting
* shmall allows around 64 GB of shared memory (assuming 4 KB page size)
* Semaphore values support up to 128 semaphore sets and 32,000 total semaphores

This baseline helps you decide whether adjustments are needed. If, for example, you’re setting shared\_buffers to 128 GB, you’ll know you must raise shmmax and shmall.

## **🧠 Best Practice: Record the Output**

Before changing any kernel settings:

* ****Write down the current values**** for comparison and rollback.
* ****Use version control**** or config management tools (like Ansible) in production systems to track kernel changes.
* ****Avoid guessing**** — always check what the system is doing first.

## **✅ Summary**

* PostgreSQL needs sufficient ****shared memory**** and ****semaphores**** to work properly.
* The Linux kernel controls these resources, and the defaults are often ****too low**** for high-performance databases.
* Using sysctl -a with filters like grep -i shmmax helps you quickly ****inspect current values****.
* This step gives you the ****confidence**** to move forward with safe and smart tuning — no surprises, no guesswork.

## **🛠️ 2️⃣ Key Kernel Parameters for PostgreSQL 17**

Tuning the Linux kernel is a critical task for database administrators who want to ensure PostgreSQL runs efficiently, reliably, and at scale. PostgreSQL is a powerful, multi-process system that relies heavily on kernel-level resources to manage memory, process coordination, and client connections.

By default, many Linux systems are ****not configured**** with PostgreSQL in mind. This can lead to performance issues, startup failures, or unexpected crashes — especially when PostgreSQL is handling high loads or a large number of users.

In this section, we’ll dive deep into two of the most essential kernel resources that PostgreSQL depends on:

* ****Shared Memory****
* ****Semaphores****

## **🔷 Shared Memory (SHMMAX, SHMALL, SHMMIN)**

## **📘 What is Shared Memory?**

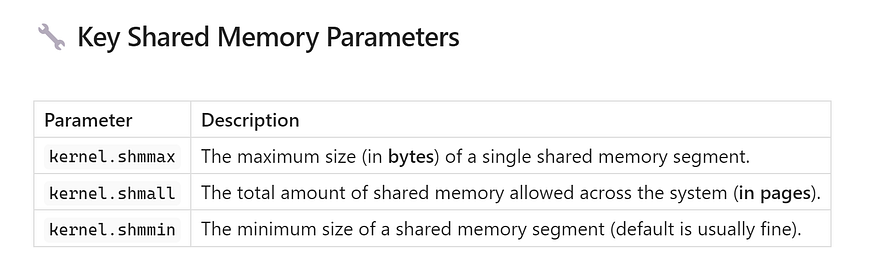
Shared memory is a special memory area that multiple processes can access at the same time. PostgreSQL uses it for:

* ****Caching data**** in memory (using shared\_buffers)
* ****Coordinating internal processes**** (like background workers, WAL writer, and autovacuum)
* ****Avoiding constant disk access**** — which improves speed

If the Linux kernel doesn’t allow enough shared memory, PostgreSQL might:

* Fail to start
* Crash unexpectedly
* Run very slowly due to increased disk I/O

Press enter or click to view image in full size



## **🧪 How to Calculate Proper Values**

Let’s say you want to allocate ****16 GB of shared memory**** for PostgreSQL’s shared\_buffers.

1. ****Set shmmax**** to the full 16 GB:

sudo sysctl -w kernel.shmmax=17179869184

(That’s 16 × 1024 × 1024 × 1024 = 17,179,869,184 bytes)

1. ****Calculate shmall**** based on page size (usually 4 KB):

17179869184 ÷ 4096 = 4,194,304

So:

sudo sysctl -w kernel.shmall=4194304

1. ****Optional (but usually safe):**** Confirm or set shmmin to 1 (default on most systems):

sudo sysctl -w kernel.shmmin=1

## **💾 Make It Permanent**

To ensure these settings persist after reboot:

1. Open the system configuration file:

sudo nano /etc/sysctl.conf

1. Add the following lines:

kernel.shmmax=17179869184  
kernel.shmall=4194304  
kernel.shmmin=1

1. Reload the kernel parameters:

sudo sysctl -p

Now, PostgreSQL will have access to the memory it needs — even after a reboot.

## **🔷 Semaphores (kernel.sem)**

## **📘 What Are Semaphores?**

Semaphores are used by PostgreSQL to ****coordinate access**** to shared resources. For example:

* Multiple processes updating the same buffer cache
* Preventing two processes from writing to the same log
* Synchronizing internal background processes

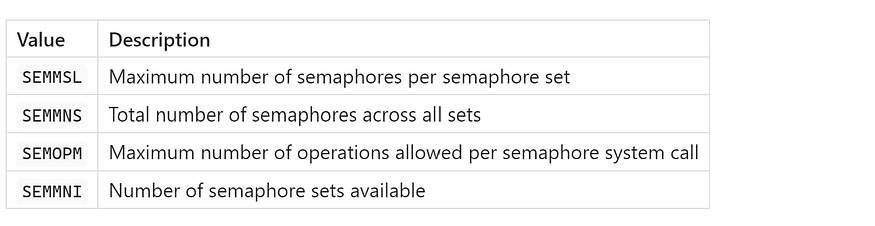
Every PostgreSQL connection and backend process requires semaphores to function safely.

## **🔧 Key Semaphore Parameters**

Linux uses one kernel parameter (kernel.sem) to define four semaphore values:

SEMMSL SEMMNS SEMOPM SEMMNI

Press enter or click to view image in full size



## **🧪 Recommended Safe Value for PostgreSQL**

A commonly used and production-safe setting is:

kernel.sem = 250 32000 100 128

This means:

* 250 semaphores per set
* 32,000 total semaphores
* 100 operations per system call
* 128 total semaphore sets

This configuration works well for most PostgreSQL deployments — even those with hundreds of concurrent users.

## **⚙️ How to Apply It**

1. Set the values immediately:

sudo sysctl -w kernel.sem="250 32000 100 128"

1. To make it permanent, add the following line to /etc/sysctl.conf:

kernel.sem=250 32000 100 128

1. Reload the configuration:

sudo sysctl -p

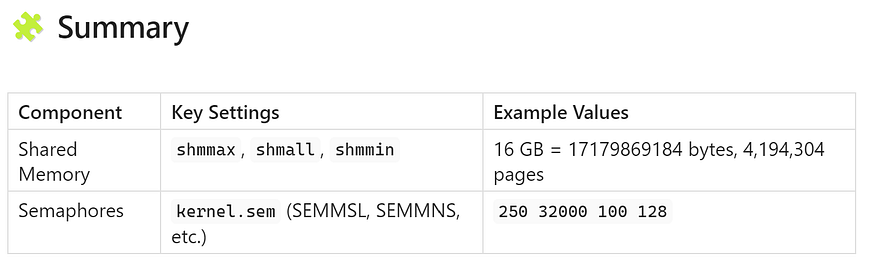
Your new semaphore settings are now active and persistent.

## **✅ Final Thoughts on Shared Memory and Semaphores**

By configuring these kernel parameters correctly:

* You ****prevent PostgreSQL startup failures**** caused by memory or semaphore limits.
* You allow PostgreSQL to ****scale**** to handle more users and bigger workloads.
* You give PostgreSQL the resources it needs to ****cache data**** and avoid slow disk access.
* You build a more ****stable and high-performance system**** from the OS level up.

Press enter or click to view image in full size



✅ Always make your changes persistent by editing /etc/sysctl.conf  
✅ Reload your settings with sudo sysctl -p after making changes

## **📈 3️⃣ Resource Limits (**ulimit**)**

PostgreSQL is a high-performance database system that works best when the ****Linux environment is tuned properly****. One important part of this tuning is setting the right ****resource limits**** for the user that runs PostgreSQL — usually the postgres user.

Linux uses a tool called ulimit to control how many files, processes, and system resources each user can use at the same time. If these limits are too low, PostgreSQL may run into problems like:

* Failing to open files or logs
* Dropping client connections
* Crashing during high load

## **🧠 What Is**ulimit**?**

ulimit stands for ****"user limit"****, and it defines how many resources a ****single user session**** is allowed to consume.

One of the most important limits for PostgreSQL is the ****number of open files**** — because PostgreSQL opens many files during normal operation, including:

* Data files for tables and indexes
* WAL (Write-Ahead Log) files
* Log files
* Temporary files for large queries
* Network sockets for client connections

## **🔍 How to Check the Current Limits**

You can see your system’s current limits by running:

ulimit -a

This command will show something like:

open files (-n) 1024

If the ****“open files”**** limit (ulimit -n) is only 1024, it’s too low for PostgreSQL in a production environment — especially if many users or connections are active at the same time.

## **🚀 PostgreSQL Best Practice**

PostgreSQL recommends increasing the open file limit to at least:

ulimit -n 100000

This allows PostgreSQL to safely manage:

* Thousands of simultaneous connections
* Background workers
* File and log operations without hitting system limits

## **⚙️ How to Raise File Descriptor Limits**

To increase the ****system-wide file descriptor limit****, run:

sudo sysctl -w fs.file-max=100000

This sets the ****maximum number of file handles**** the Linux kernel can allocate system-wide.

To make this change ****permanent****, add this line to your /etc/sysctl.conf file:

fs.file-max=100000

Then reload the system settings:

sudo sysctl -p

## **👤 How to Set**ulimit**for the**postgres**User**

To make sure the postgres user can actually use more file descriptors:

1. Edit the user limits file:

sudo nano /etc/security/limits.conf

1. Add these lines at the bottom:

postgres soft nofile 100000  
postgres hard nofile 100000

1. Make sure the system applies these settings. Edit:

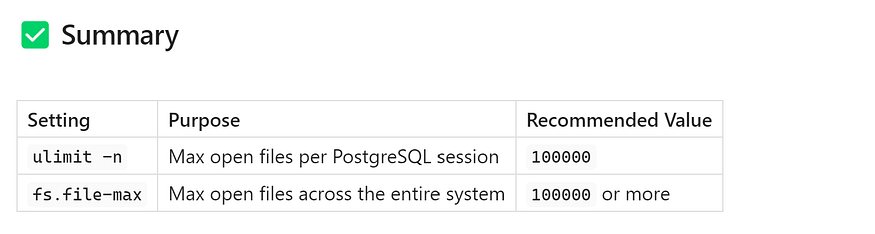
sudo nano /etc/pam.d/common-session

And ensure it includes:

session required pam\_limits.so

1. Restart the machine or log out and back in to apply changes.

Press enter or click to view image in full size



By increasing these limits, you ensure that PostgreSQL can:

* Handle more users
* Open more files
* Run without hitting system restrictions

This is a simple but essential step for a stable, scalable PostgreSQL deployment — especially in production environments.

## **🧮 4️⃣ Control Linux Memory Overcommit**

When running PostgreSQL in a Linux environment — especially in production — one of the most important but commonly overlooked aspects is how the ****Linux kernel handles memory allocation****. By default, Linux is a bit too optimistic. It may allow applications to ****request and allocate more memory than the system physically has****, assuming not all of it will be used.

This behavior is called ****memory overcommit****, and while it works fine in some desktop scenarios, it can be ****dangerous for critical services like PostgreSQL****.

Let’s explore what overcommit is, why it’s risky, and how to configure it properly for PostgreSQL.

## **🧠 What Is Memory Overcommit?**

In Linux, ****overcommitting memory**** means the system allows programs to request memory beyond what is physically and virtually available (RAM + swap).

The kernel does this because:

* Many applications reserve memory that they never fully use.
* Allowing a little over-allocation improves memory utilization.

But when ****many programs actually use**** the memory they’ve been promised, the system may ****run out of usable memory****.

When that happens, Linux responds by invoking the ****OOM Killer**** (Out-of-Memory Killer) — a last-resort mechanism that forcefully terminates processes to free up memory.

## **❌ Why This Is Dangerous for PostgreSQL**

PostgreSQL is a memory-intensive service. It needs memory for:

* ****shared\_buffers**** – PostgreSQL's cache
* ****work\_mem**** – used per sort, join, or hash operation
* ****maintenance\_work\_mem**** – used by background operations like vacuum
* Connections — each backend process also consumes memory

If the system runs out of RAM because it overcommitted too much, the kernel might kill the ****PostgreSQL process**** — even if it’s functioning normally.

You might see this in your logs:

Out of memory: Kill process 12345 (postgres) score 998 or sacrifice child  
Killed process 12345 (postgres) total-vm:4194304kB, anon-rss:500000kB, file-rss:1024kB

This can result in:

* 🟥 Database downtime
* 🟥 Crashed queries and lost sessions
* 🟥 Incomplete transactions
* 🟥 Data integrity issues in extreme cases

This is ****not acceptable in a production environment****.

## **✅ The Safe Setting:**vm.overcommit\_memory = 2

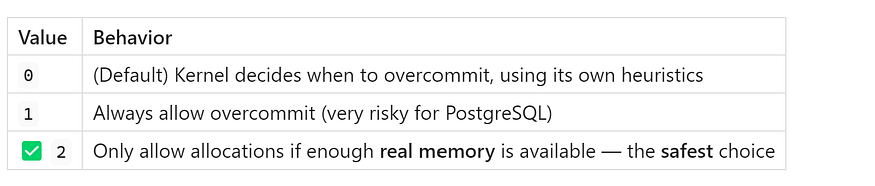
To prevent this risk, Linux provides a way to control how memory is overcommitted.

The parameter is called:

vm.overcommit\_memory

And it has three possible values:

Press enter or click to view image in full size



Setting this to ****2**** tells Linux:

“Only allow memory allocations when the system actually has enough memory to fulfill them.”

This greatly reduces the chance of hitting the OOM killer and keeps PostgreSQL safer during heavy loads or memory spikes.

## **🛠️ How to Apply the Setting**

You can apply this setting immediately and make it permanent.

### **🔹 1. Apply Immediately**

Run this command:

sudo sysctl -w vm.overcommit\_memory=2

This updates the value in the currently running kernel.

### **🔹 2. Make It Permanent**

To ensure the setting stays after reboot, add it to /etc/sysctl.conf:

sudo nano /etc/sysctl.conf

Then add this line at the end:

vm.overcommit\_memory=2

Save the file and reload the kernel settings:

sudo sysctl -p

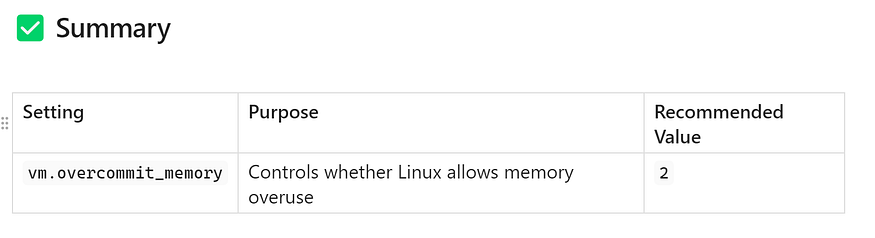
## **💡 Tip: Monitor Memory Usage**

Even with this setting, it’s smart to monitor PostgreSQL’s memory consumption using tools like:

* top, htop, or free -m for general memory usage
* pg\_stat\_activity for active queries and connection load
* Alerts in monitoring tools like ****Prometheus + Grafana****, ****Zabbix****, or ****Datadog****

This helps you avoid OOM scenarios in the first place and proactively manage resource growth.

Press enter or click to view image in full size



****Why it matters:****

* Prevents Linux from over-allocating memory beyond what’s available
* Reduces the risk of PostgreSQL being ****killed by the Out-of-Memory Killer****
* Essential for ****high-reliability database environments****

## **🚀 Final Thought**

PostgreSQL needs a ****stable memory environment**** to perform at its best. While PostgreSQL itself is highly resilient, it can’t survive being terminated by the operating system. By setting vm.overcommit\_memory=2, you’re making a ****simple but powerful change**** that can save your production system from unpredictable crashes.

## **🚀 5️⃣ Enabling Huge Pages in PostgreSQL**

When dealing with ****high-performance PostgreSQL databases****, especially those with large memory allocations (e.g., shared\_buffers set to several gigabytes), memory efficiency becomes critical. One powerful Linux feature that can boost PostgreSQL’s performance in such environments is ****Huge Pages****.

## **🧠 What Are Huge Pages?**

By default, Linux manages memory in small chunks called ****pages****, usually ****4 KB**** in size. But when a system has to manage ****gigabytes of memory****, this means managing ****millions of tiny pages**** — which increases overhead on both the CPU and the memory subsystem.

****Huge Pages**** allow the system to use ****larger memory pages**** (typically 2 MB or more). Fewer pages = less management overhead = faster performance.

## **🚀 Why PostgreSQL Benefits from Huge Pages**

PostgreSQL uses a large amount of shared memory (via shared\_buffers). If this memory is managed using normal pages, it can lead to:

* Increased CPU cycles spent managing memory
* TLB (Translation Lookaside Buffer) misses
* Slower performance under load

Enabling Huge Pages tells Linux to allocate PostgreSQL’s shared memory using ****larger, contiguous memory blocks****, reducing the overhead and improving efficiency.

Especially in systems with:

* shared\_buffers set to 1 GB or more
* High connection counts
* Frequent queries and concurrent activity  
  Huge Pages can significantly improve performance and reduce latency.

## **🔍 How to Enable Huge Pages — Step-by-Step**

## **🧪 Step 1: Check Huge Page Size**

First, verify the size of a huge page on your system:

grep Hugepagesize /proc/meminfo

You might see:

Hugepagesize: 2048 kB

This means each huge page is 2 MB.

## **🧪 Step 2: Estimate How Many Huge Pages You Need**

Now calculate how many huge pages PostgreSQL needs to store its shared memory.

### **1️⃣ Start PostgreSQL normally**

Start PostgreSQL as you normally would — no huge pages needed yet.

### **2️⃣ Get the PID of the running PostgreSQL process**

head -1 $PGDATA/postmaster.pid

This will return the main process ID (PID) of PostgreSQL.

### **3️⃣ Check how much shared memory is being used**

Replace <pid> with the PID from the previous command:

pmap <pid> | awk '/rw-s/ && /zero/ {print $2}'

This shows how much anonymous shared memory is mapped — the main candidate for Huge Page allocation.

### **4️⃣ Convert that value into huge pages**

Let’s say the above command tells you PostgreSQL uses ~6.2 GB of shared memory.

* Each huge page = 2 MB
* 6.2 GB = 6,200 MB
* Required Huge Pages = 6200 ÷ 2 = ****3100****

Add a buffer to be safe — let’s round up to ****3170****.

## **🧪 Step 3: Allocate Huge Pages**

Run this command to allocate 3170 huge pages in the Linux kernel:

sudo sysctl -w vm.nr\_hugepages=3170

To make this change ****permanent**** across reboots, add it to /etc/sysctl.conf:

vm.nr\_hugepages=3170

Then reload sysctl settings:

sudo sysctl -p

## **🧪 Step 4: Force PostgreSQL to Use Huge Pages**

Now tell PostgreSQL to use Huge Pages for its shared memory.

Edit postgresql.conf:

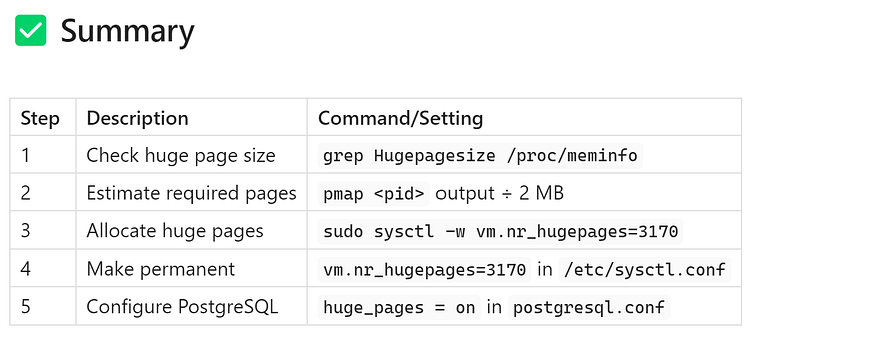
huge\_pages = on

💡 Other options:

* off – don’t use huge pages
* try – use huge pages if available, otherwise fall back
* on – fail startup if huge pages aren't available (ensures performance optimization)

****Tip:**** Use on in production to ensure the setting is enforced.

Press enter or click to view image in full size



## **🏁 Final Thoughts**

Enabling Huge Pages can lead to noticeable performance improvements — especially in memory-heavy PostgreSQL environments.

By reducing memory management overhead, Huge Pages help your database:

* Run more efficiently
* Avoid latency from TLB misses
* Reduce context switching overhead

It’s a ****one-time setup**** that brings long-term gains.

## **🔬 6️⃣ Full Example:**/etc/sysctl.conf**for PostgreSQL 17**

Once you’ve learned about all the key Linux kernel parameters that help optimize PostgreSQL — like shared memory, semaphores, file limits, overcommit settings, and Huge Pages — the final step is to ****bring everything together into one place****.

That place is the configuration file:  
📄 ****/etc/sysctl.conf****

This file contains kernel-level settings that the Linux operating system reads at boot time. When you’re preparing a PostgreSQL server — especially for high-performance production environments — it’s a best practice to ****define all required tuning parameters here**** for consistency, stability, and persistence.

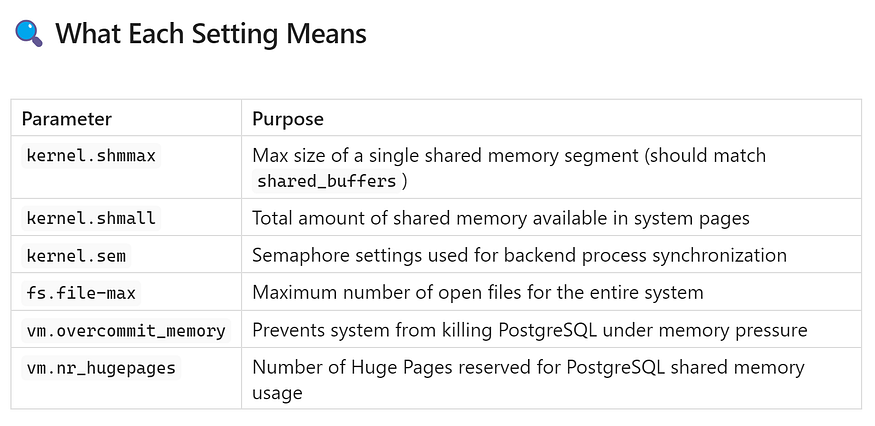
## **✅ Recommended**/etc/sysctl.conf**for PostgreSQL 17**

Below is a complete and production-grade example that includes all the critical settings we’ve discussed so far:

# Shared Memory Configuration  
kernel.shmmax=17179869184 # Max shared memory segment (16GB)  
kernel.shmall=4194304 # Total shared memory pages (for 4KB page size)

# Semaphores for inter-process communication  
kernel.sem=250 32000 100 128 # Safe default values for PostgreSQL workloads# File descriptor limits  
fs.file-max=100000 # Max number of open files system-wide# Memory overcommit protection  
vm.overcommit\_memory=2 # Only allocate memory if enough is available# Huge Pages allocation for performance  
vm.nr\_hugepages=3170 # Reserve ~6.2GB of Huge Pages (if 2MB each)

Press enter or click to view image in full size



## **🛠️ How to Apply the Settings**

After saving your edits to /etc/sysctl.conf, apply the new configuration immediately without rebooting:

sudo sysctl -p

This command reads the /etc/sysctl.conf file and updates the running kernel with the new values.

## **📌 Why This Matters**

By centralizing all kernel tunings into /etc/sysctl.conf, you:

* Ensure PostgreSQL has ****enough system resources****
* Avoid system-level memory or I/O issues
* Improve ****stability and performance****
* Make server configuration ****repeatable and consistent**** (great for DevOps and automation)

This is one of the ****most important one-time setups**** when preparing a Linux system for PostgreSQL 17 or later.

## **🧠 Final Tip**

Always reboot the server after making these changes during initial configuration to ensure:

* Memory pages are reserved correctly (especially Huge Pages)
* No other processes interfere with shared memory space

## **📝 Conclusion — The Final Word on Kernel Tuning for PostgreSQL 17**

Optimizing PostgreSQL performance doesn’t stop at database parameters like shared\_buffers, work\_mem, or autovacuum. A highly performant PostgreSQL database also depends on the ****foundation it's built upon — the Linux kernel****.

Many production issues — like slow performance, unexpected crashes, or limited concurrency — stem from misconfigured system-level settings. Fortunately, Linux gives us control over those through kernel parameters, and PostgreSQL is well-documented to benefit from tuning them.

Let’s wrap up with a complete summary of the most essential kernel tuning changes you should consider when preparing a PostgreSQL 17 server.

## **✅ PostgreSQL 17 Kernel Tuning Summary**

## **1️⃣ Tune Shared Memory (**kernel.shmmax**,**kernel.shmall**)**

PostgreSQL relies on ****shared memory**** to store cache (via shared\_buffers) and to allow its backend processes to communicate efficiently.

* kernel.shmmax sets the ****maximum size**** of a single shared memory segment. This must be large enough to fit your shared\_buffers.
* kernel.shmall defines the ****total shared memory available****, in pages. It's typically calculated based on your system's RAM.

📌 Without sufficient shared memory settings:

* PostgreSQL may fail to start
* Or it may be forced to reduce its cache, hurting performance

By aligning these with PostgreSQL’s needs, you’re ensuring it has room to breathe and cache data effectively.

## **2️⃣ Set Semaphores Properly (**kernel.sem**)**

Semaphores are a type of locking mechanism that PostgreSQL uses to manage concurrent operations across its many processes (each connection to PostgreSQL is a separate process).

Use the recommended setting:

kernel.sem = 250 32000 100 128

This allows:

* More simultaneous PostgreSQL processes to work without blocking
* Smooth concurrency for multi-user or multi-threaded workloads

📌 With incorrect semaphore values, you might experience:

* Delays in backend process creation
* Errors under high concurrency (e.g., max connection failures)

## **3️⃣ Increase File Descriptors (**fs.file-max**)**

Each open file, table, WAL log, and temporary result set uses a file descriptor. If the system-wide or per-process limit is too low, PostgreSQL will start failing under stress.

Set:

fs.file-max = 100000

Also make sure your PostgreSQL user’s ulimit -n is increased to match this.

📌 This is essential for:

* Systems with high connection counts
* Queries that touch many files or indexes
* Replication, logging, or partition-heavy workloads

## **4️⃣ Use Safe Memory Allocation (**vm.overcommit\_memory = 2**)**

By default, Linux allows more memory to be allocated than it has — a practice called ****memory overcommit****. While useful for desktops, it’s risky for database servers.

PostgreSQL may be killed by the ****OOM Killer**** if memory runs low — causing downtime and data recovery.

Set:

vm.overcommit\_memory = 2

This tells Linux to allocate memory ****only if enough real memory exists****, preventing overcommit-related crashes.

📌 This is a must-have for any PostgreSQL production setup.

## **5️⃣ Enable Huge Pages (**vm.nr\_hugepages**,**huge\_pages = on**)**

Linux normally manages memory in small 4 KB chunks (pages). PostgreSQL, especially with large shared\_buffers, can benefit from ****Huge Pages**** (usually 2 MB), which reduce memory management overhead.

To enable Huge Pages:

* Reserve pages in Linux:

vm.nr\_hugepages = <calculated\_number>

* Enable in PostgreSQL:

huge\_pages = on

📌 Benefits of Huge Pages:

* Less CPU overhead for memory operations
* Improved cache efficiency
* Better performance for large-memory systems

## **⚠️ Always Test Before Production**

“Measure twice, cut once.”

Kernel settings affect the entire operating system. While these recommendations are PostgreSQL-specific, they can impact other applications on the same server.

Before rolling out to production:

* ✅ Test changes in a staging environment
* ✅ Monitor PostgreSQL startup logs for memory errors
* ✅ Use tools like pg\_stat\_activity, htop, and Prometheus to validate improvements

## **🏁 Final Thoughts**

PostgreSQL is a powerful and reliable database engine — but it needs a well-tuned system underneath to truly shine.

By following these kernel tuning best practices, you’re ensuring your PostgreSQL 17 server:

* Starts reliably every time
* Handles memory-intensive workloads smoothly
* Avoids unexpected process crashes or bottlenecks
* Scales efficiently as your application grows

In short, you’re creating a ****solid infrastructure**** for one of the world’s best open-source databases.

🛠️ ****A few tweaks at the kernel level can save hours of debugging down the road.****