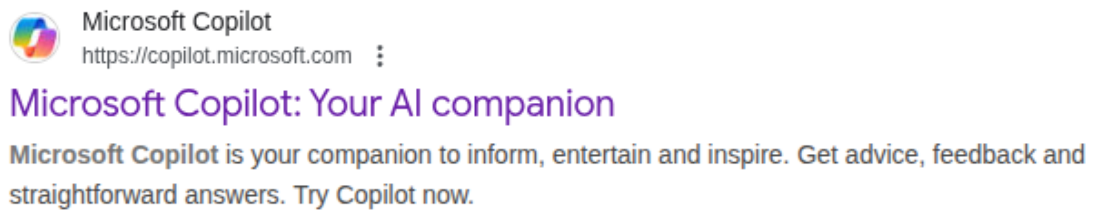
glibc Process Information

20251101 Sat

Copyright (c) 2025 Young W. Lim.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

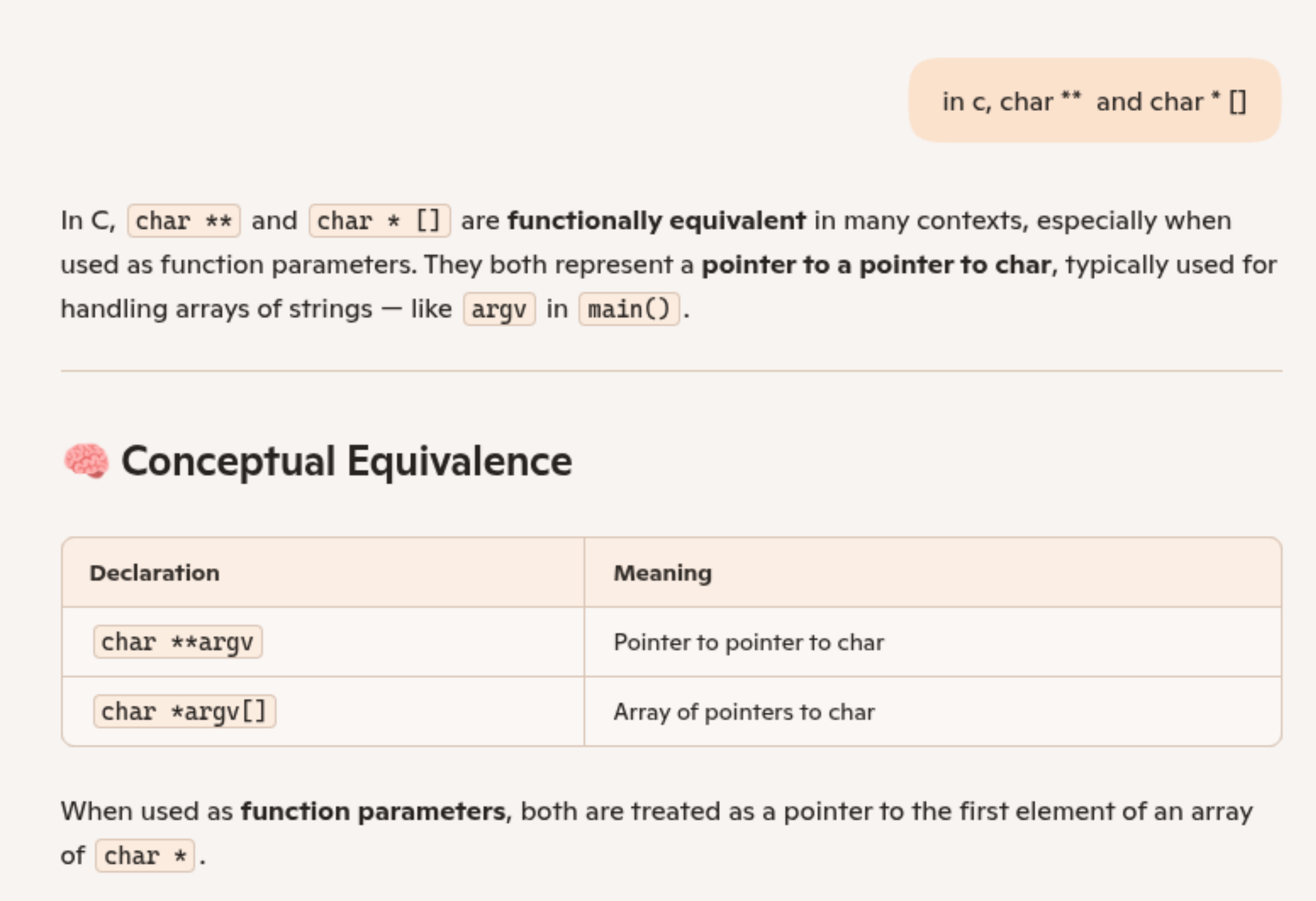
I used Microsoft Copilot to prepare this document



Microsoft Copilot
https://copilot.microsoft.com

Microsoft Copilot: Your AI companion

Microsoft Copilot is your companion to inform, entertain and inspire. Get advice, feedback and straightforward answers. Try Copilot now.



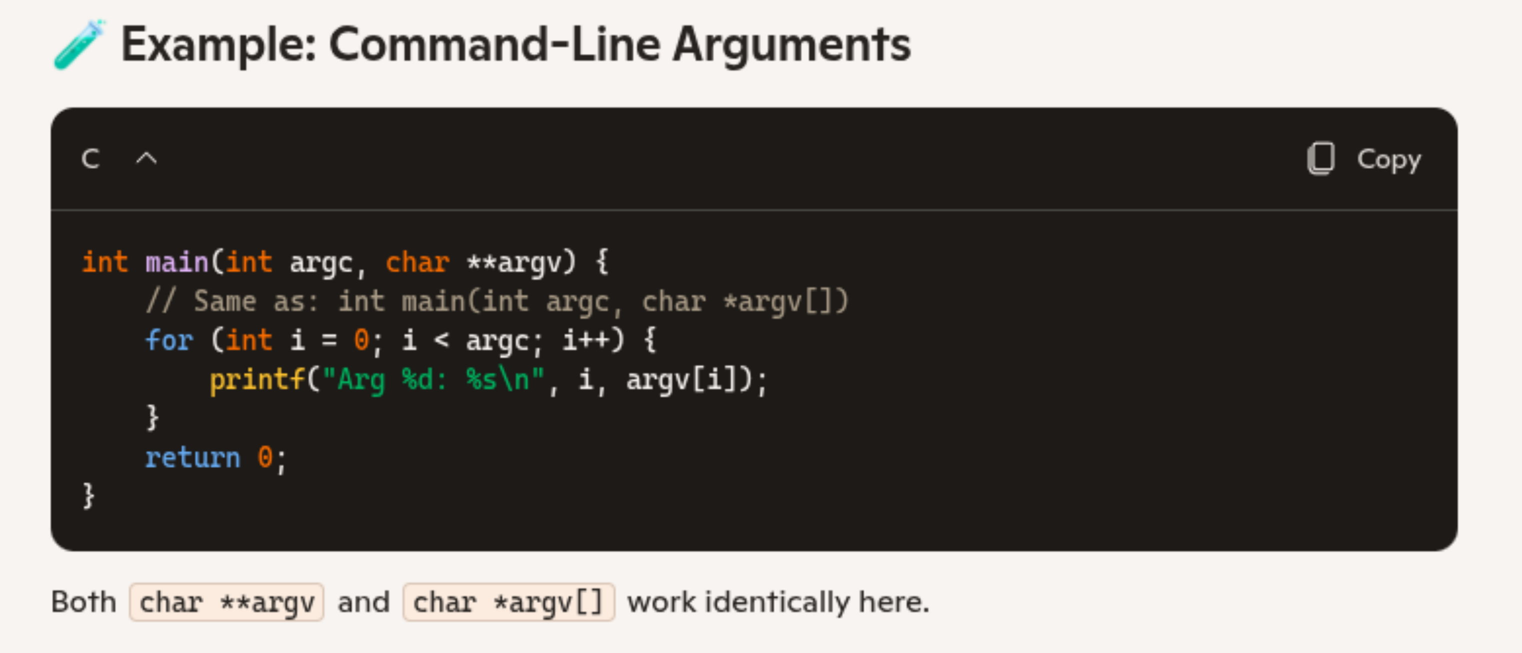
in c, char ** and char * []

In C, `char **` and `char * []` are **functionally equivalent** in many contexts, especially when used as function parameters. They both represent a **pointer to a pointer to char**, typically used for handling arrays of strings — like `argv` in `main()`.

🧠 Conceptual Equivalence

Declaration	Meaning
`char **argv`	Pointer to pointer to char
`char *argv[]`	Array of pointers to char

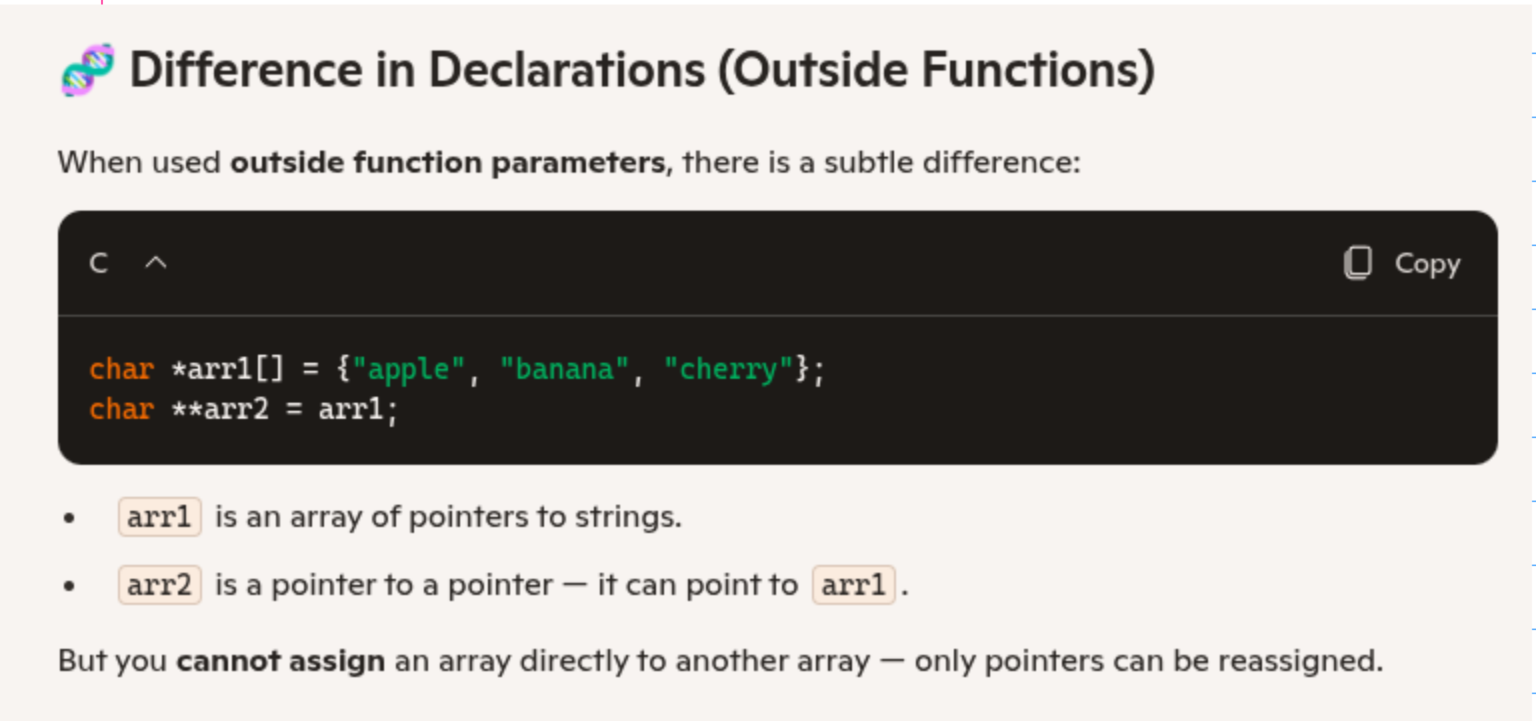
When used as **function parameters**, both are treated as a pointer to the first element of an array of `char *`.



🧪 Example: Command-Line Arguments

```c
int main(int argc, char **argv) {
    // Same as: int main(int argc, char *argv[])
    for (int i = 0; i < argc; i++) {
        printf("Arg %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Both `char **argv` and `char *argv[]` work identically here.



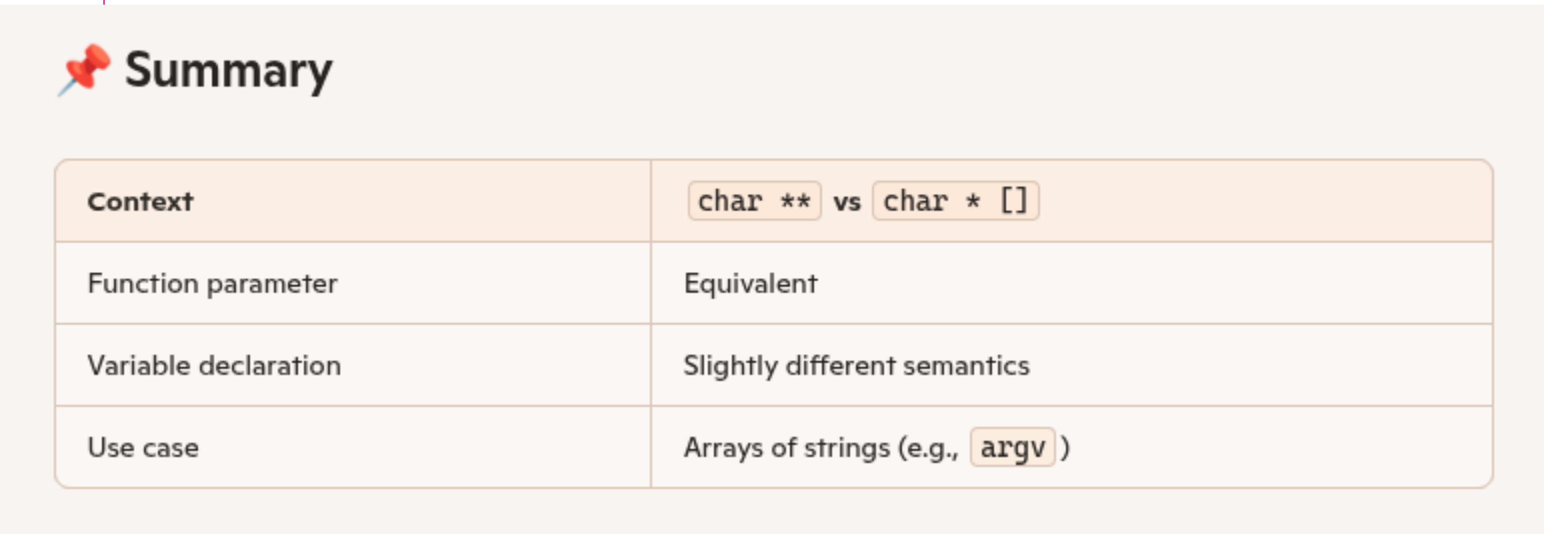
🧬 Difference in Declarations (Outside Functions)

When used **outside function parameters**, there is a subtle difference:

```c
char *arr1[] = {"apple", "banana", "cherry"};
char **arr2 = arr1;
```

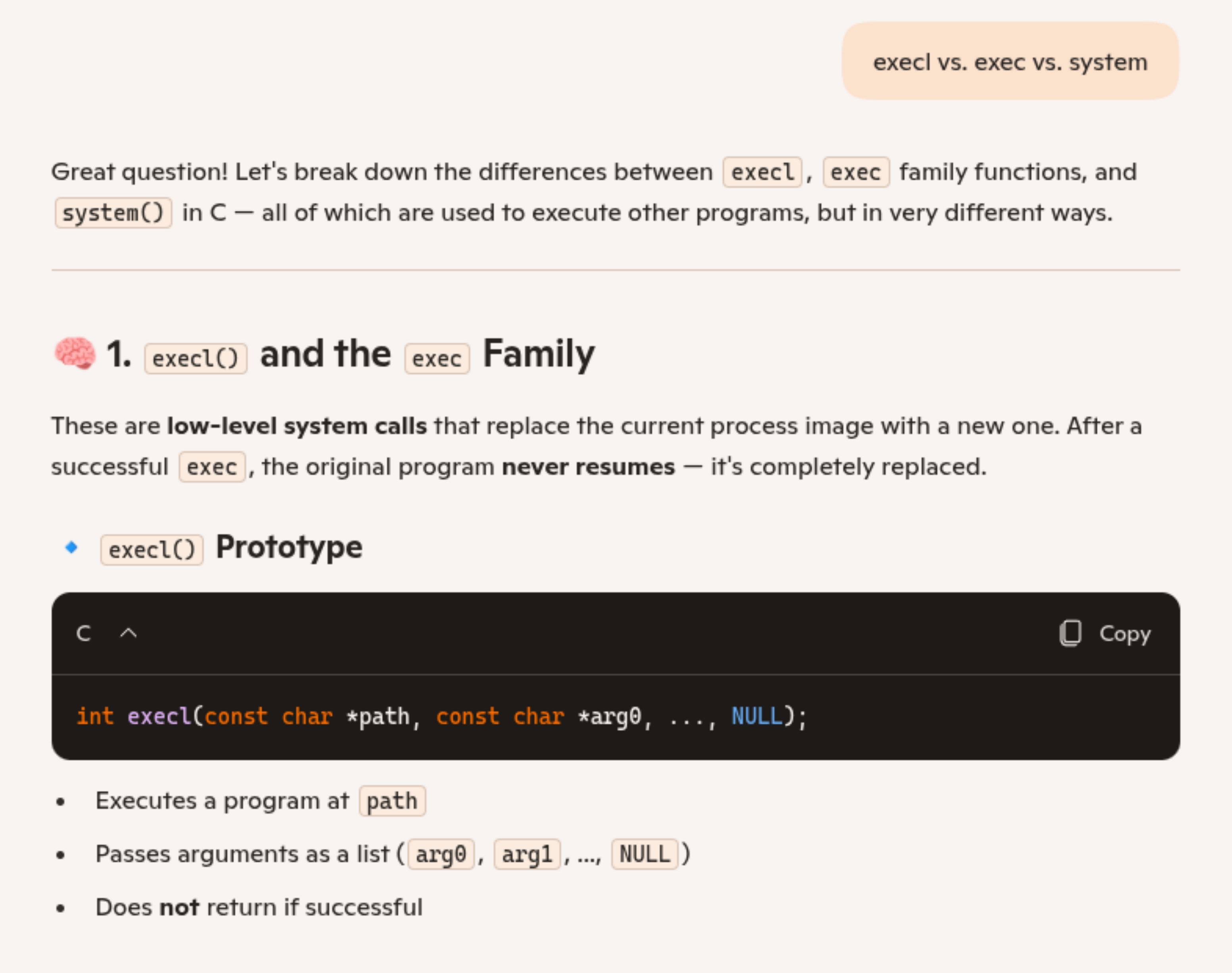
- `arr1` is an array of pointers to strings.
- `arr2` is a pointer to a pointer — it can point to `arr1`.

But you **cannot assign** an array directly to another array — only pointers can be reassigned.



📌 Summary

Context	`char **` **vs** `char * []`
Function parameter	Equivalent
Variable declaration	Slightly different semantics
Use case	Arrays of strings (e.g., `argv`)



execl vs. exec vs. system

Great question! Let's break down the differences between `execl`, `exec` family functions, and `system()` in C — all of which are used to execute other programs, but in very different ways.

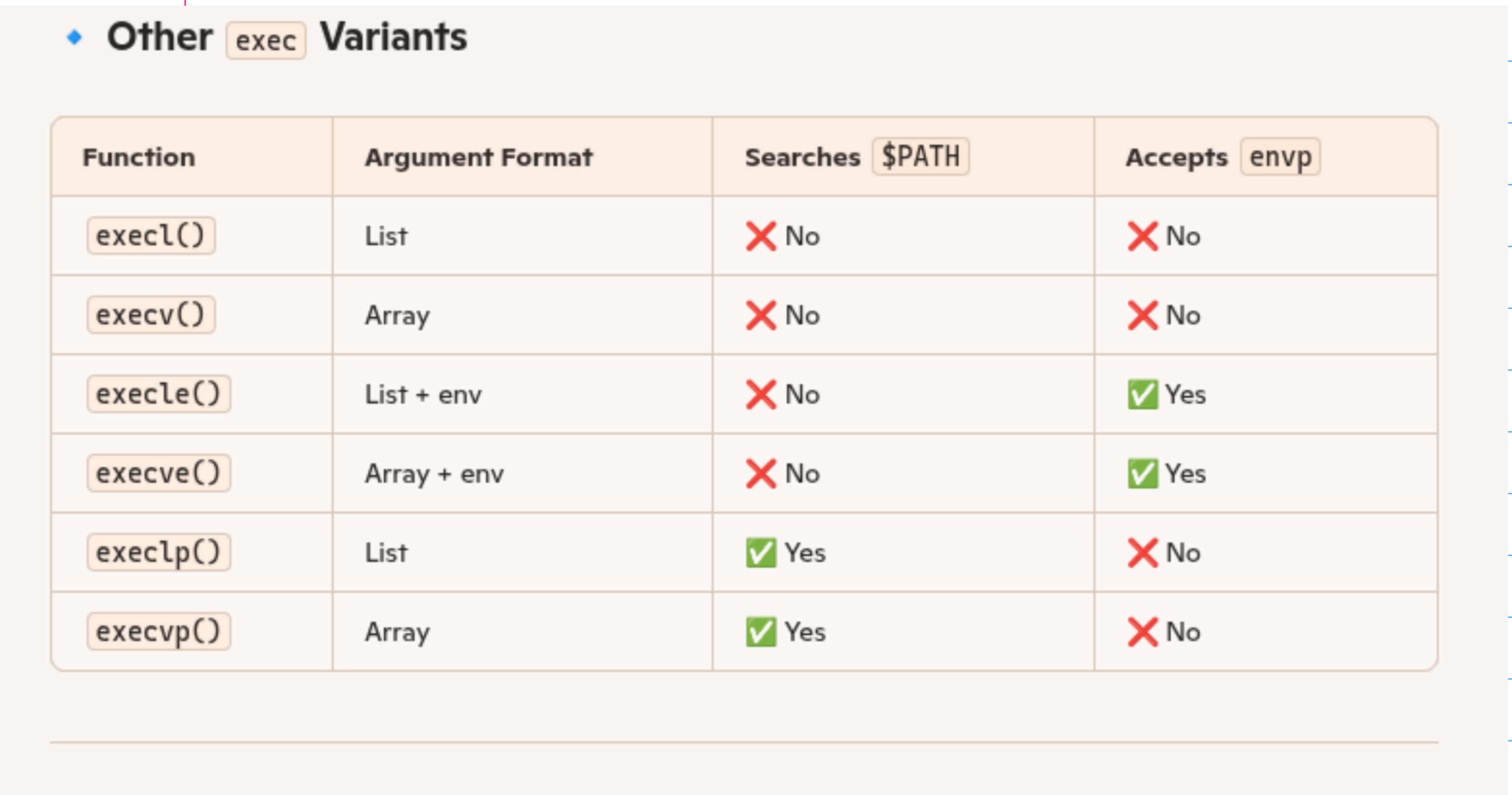
🧠 1. `execl()` and the `exec` Family

These are **low-level system calls** that replace the current process image with a new one. After a successful `exec`, the original program **never resumes** — it's completely replaced.

🔹 `execl()` Prototype

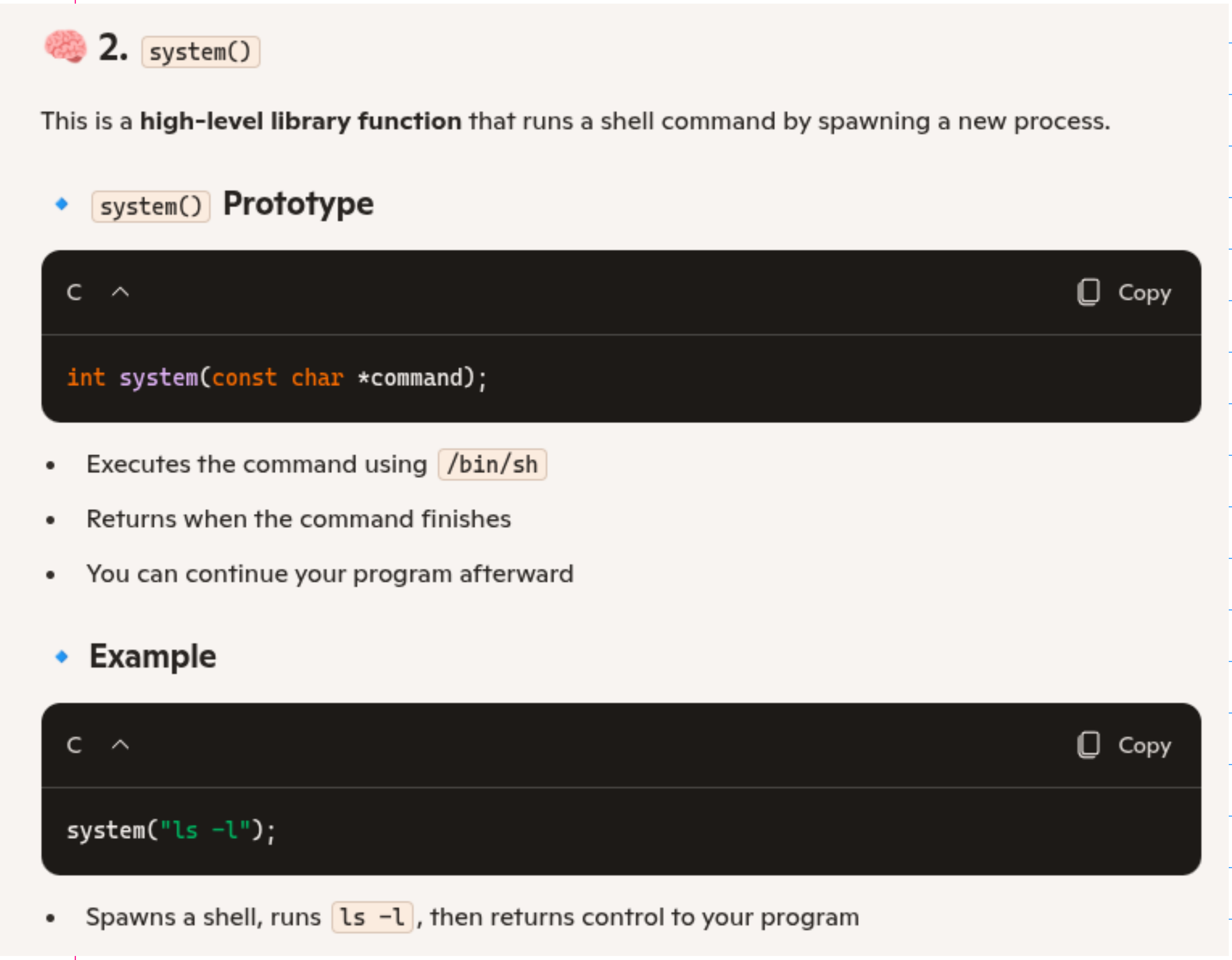
```c
int execl(const char *path, const char *arg0, ..., NULL);
```

- Executes a program at `path`
- Passes arguments as a list (`arg0`, `arg1`, ..., `NULL`)
- Does **not** return if successful



🔹 Other `exec` Variants

Function	Argument Format	Searches `$PATH`	Accepts `envp`
`execl()`	List	❌ No	❌ No
`execv()`	Array	❌ No	❌ No
`execle()`	List + env	❌ No	✅ Yes
`execve()`	Array + env	❌ No	✅ Yes
`execlp()`	List	✅ Yes	❌ No
`execvp()`	Array	✅ Yes	❌ No



🧠 2. `system()`

This is a **high-level library function** that runs a shell command by spawning a new process.

🔹 `system()` Prototype

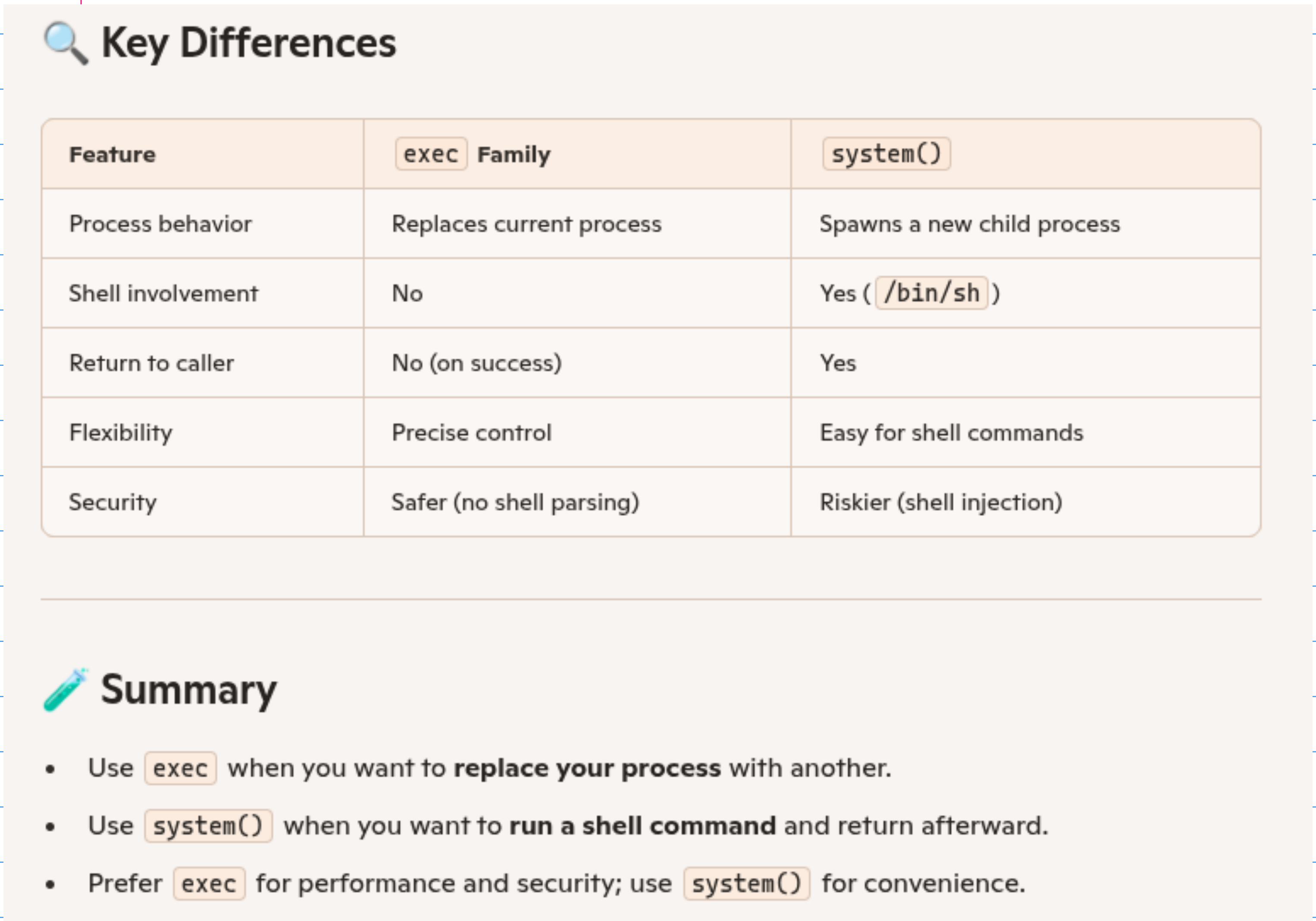
```c
int system(const char *command);
```

- Executes the command using `/bin/sh`
- Returns when the command finishes
- You can continue your program afterward

🔹 Example

```c
system("ls -l");
```

- Spawns a shell, runs `ls -l`, then returns control to your program

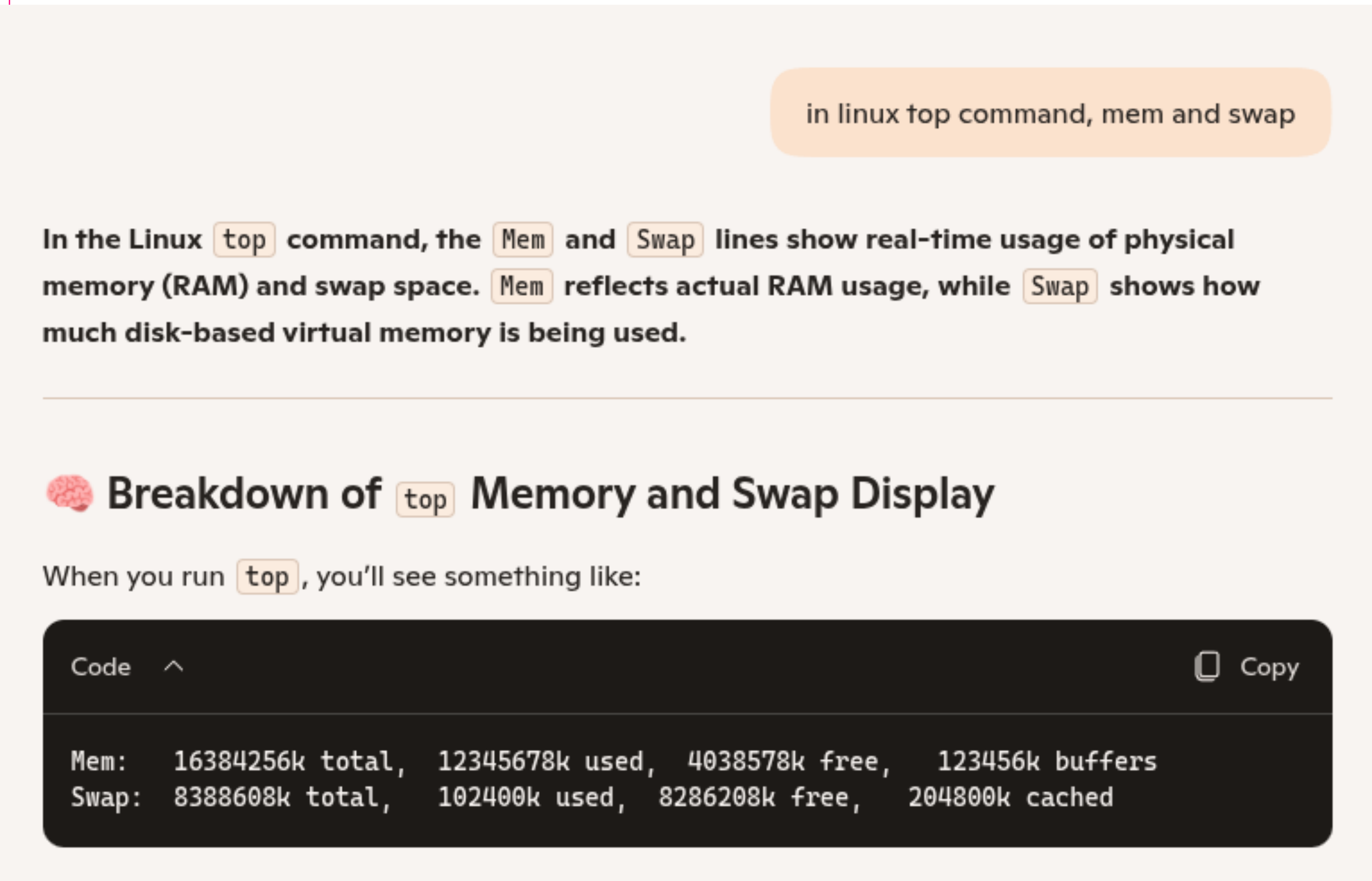


🔍 Key Differences

Feature	`exec` Family	`system()`
Process behavior	Replaces current process	Spawns a new child process
Shell involvement	No	Yes (`/bin/sh`)
Return to caller	No (on success)	Yes
Flexibility	Precise control	Easy for shell commands
Security	Safer (no shell parsing)	Riskier (shell injection)

🧪 Summary

- Use `exec` when you want to **replace your process** with another.
- Use `system()` when you want to **run a shell command** and return afterward.
- Prefer `exec` for performance and security; use `system()` for convenience.



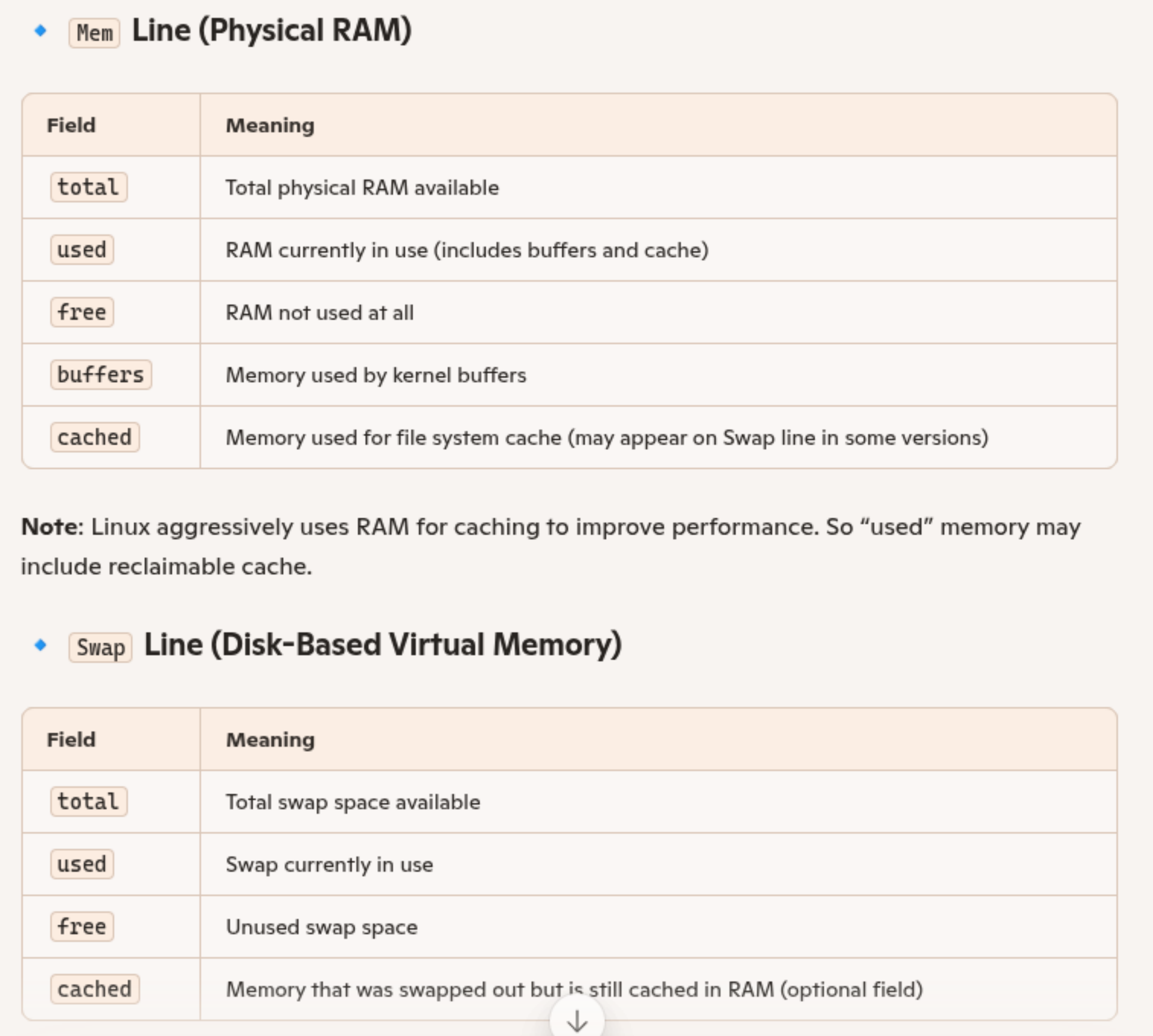
in linux top command, mem and swap

In the Linux `top` command, the `Mem` and `Swap` lines show real-time usage of physical memory (RAM) and swap space. `Mem` reflects actual RAM usage, while `Swap` shows how much disk-based virtual memory is being used.

🧠 Breakdown of `top` Memory and Swap Display

When you run `top`, you'll see something like:

```
Mem:   16384256k total,  12345678k used,  4038578k free,   123456k buffers
Swap:  8388608k total,   102400k used,  8286208k free,   204800k cached
```



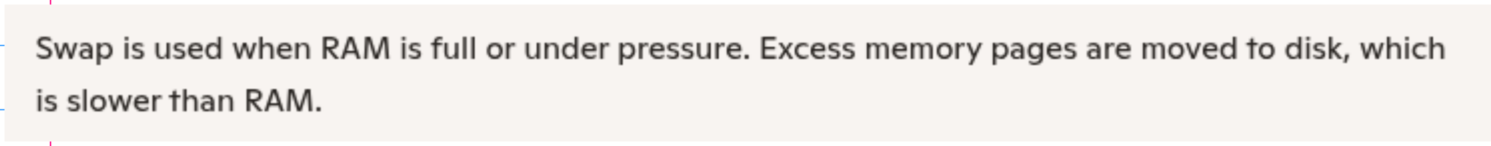
🔹 `Mem` Line (Physical RAM)

Field	Meaning
`total`	Total physical RAM available
`used`	RAM currently in use (includes buffers and cache)
`free`	RAM not used at all
`buffers`	Memory used by kernel buffers
`cached`	Memory used for file system cache (may appear on Swap line in some versions)

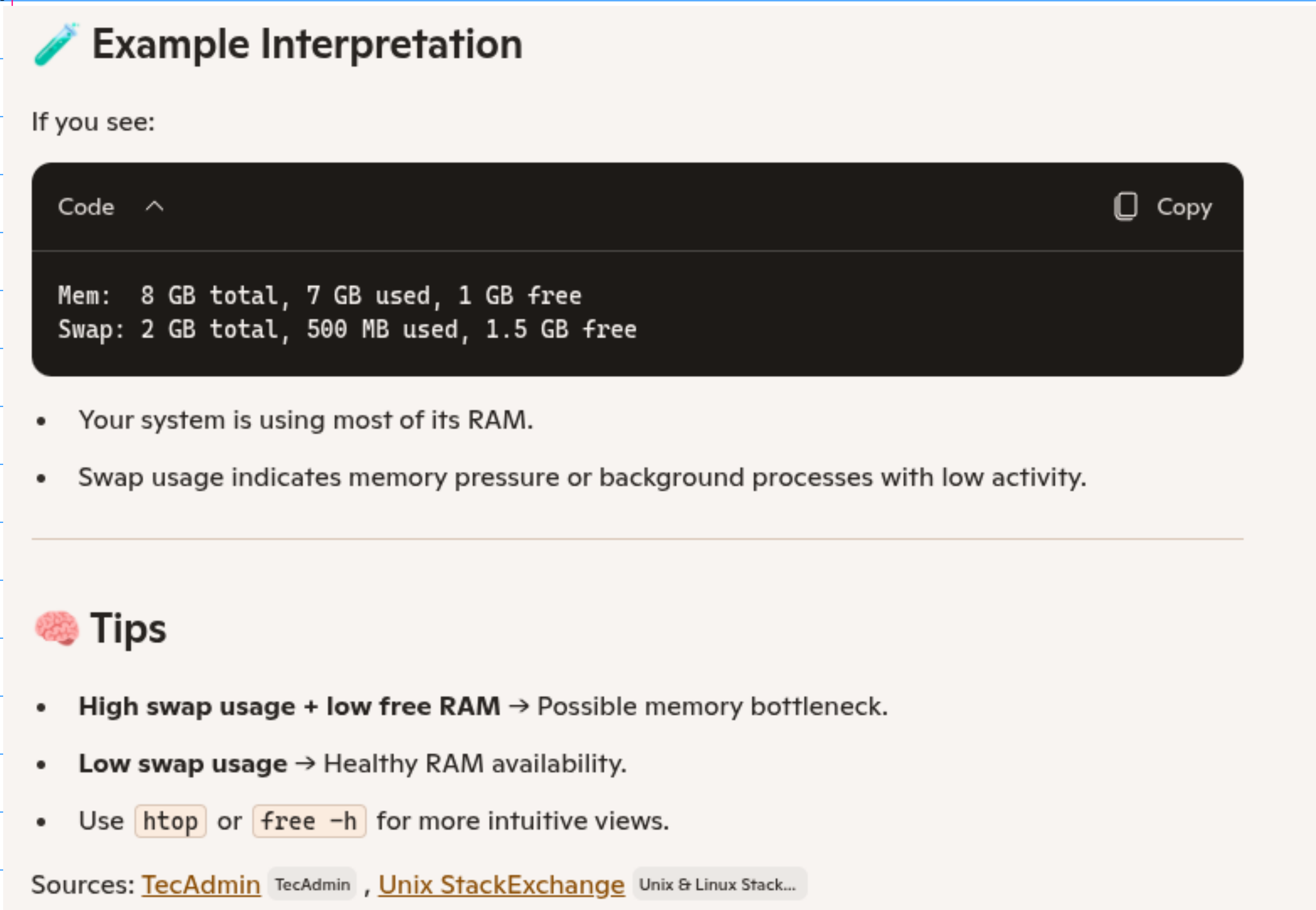
Note: Linux aggressively uses RAM for caching to improve performance. So "used" memory may include reclaimable cache.

🔹 `Swap` Line (Disk-Based Virtual Memory)

Field	Meaning
`total`	Total swap space available
`used`	Swap currently in use
`free`	Unused swap space
`cached`	Memory that was swapped out but is still cached in RAM (optional field)



Swap is used when RAM is full or under pressure. Excess memory pages are moved to disk, which is slower than RAM.



🧪 Example Interpretation

If you see:

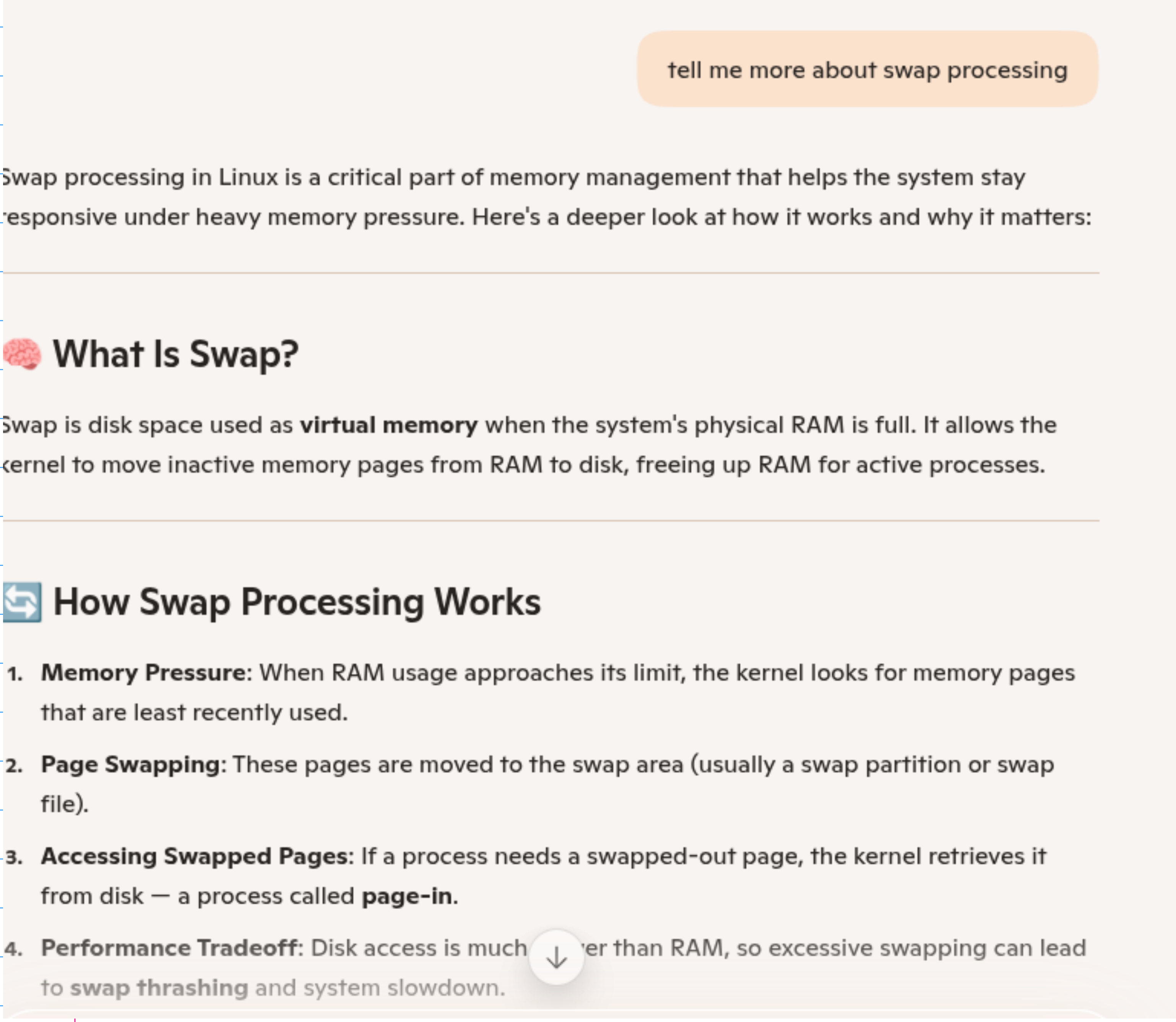
```
Mem:  8 GB total, 7 GB used, 1 GB free
Swap: 2 GB total, 500 MB used, 1.5 GB free
```

- Your system is using most of its RAM.
- Swap usage indicates memory pressure or background processes with low activity.

🧠 Tips

- **High swap usage + low free RAM** → Possible memory bottleneck.
- **Low swap usage** → Healthy RAM availability.
- Use `htop` or `free -h` for more intuitive views.

Sources: TecAdmin TecAdmin , Unix StackExchange Unix & Linux Stack...



tell me more about swap processing

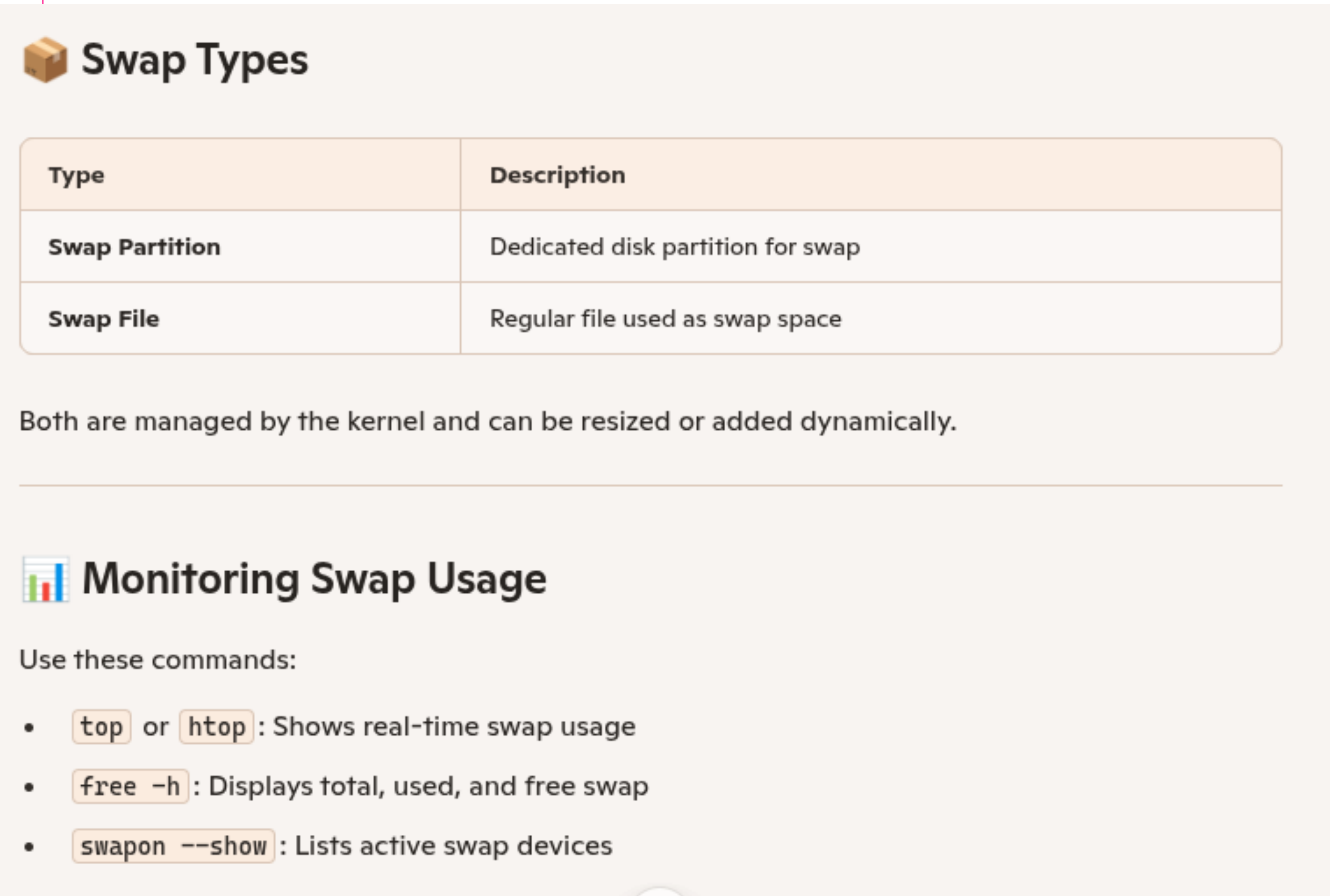
Swap processing in Linux is a critical part of memory management that helps the system stay responsive under heavy memory pressure. Here's a deeper look at how it works and why it matters:

🧠 What Is Swap?

Swap is disk space used as **virtual memory** when the system's physical RAM is full. It allows the kernel to move inactive memory pages from RAM to disk, freeing up RAM for active processes.

🔄 How Swap Processing Works

1. **Memory Pressure**: When RAM usage approaches its limit, the kernel looks for memory pages that are least recently used.
2. **Page Swapping**: These pages are moved to the swap area (usually a swap partition or swap file).
3. **Accessing Swapped Pages**: If a process needs a swapped-out page, the kernel retrieves it from disk — a process called **page-in**.
4. **Performance Tradeoff**: Disk access is much [illegible]er than RAM, so excessive swapping can lead to **swap thrashing** and system slowdown.



📦 Swap Types

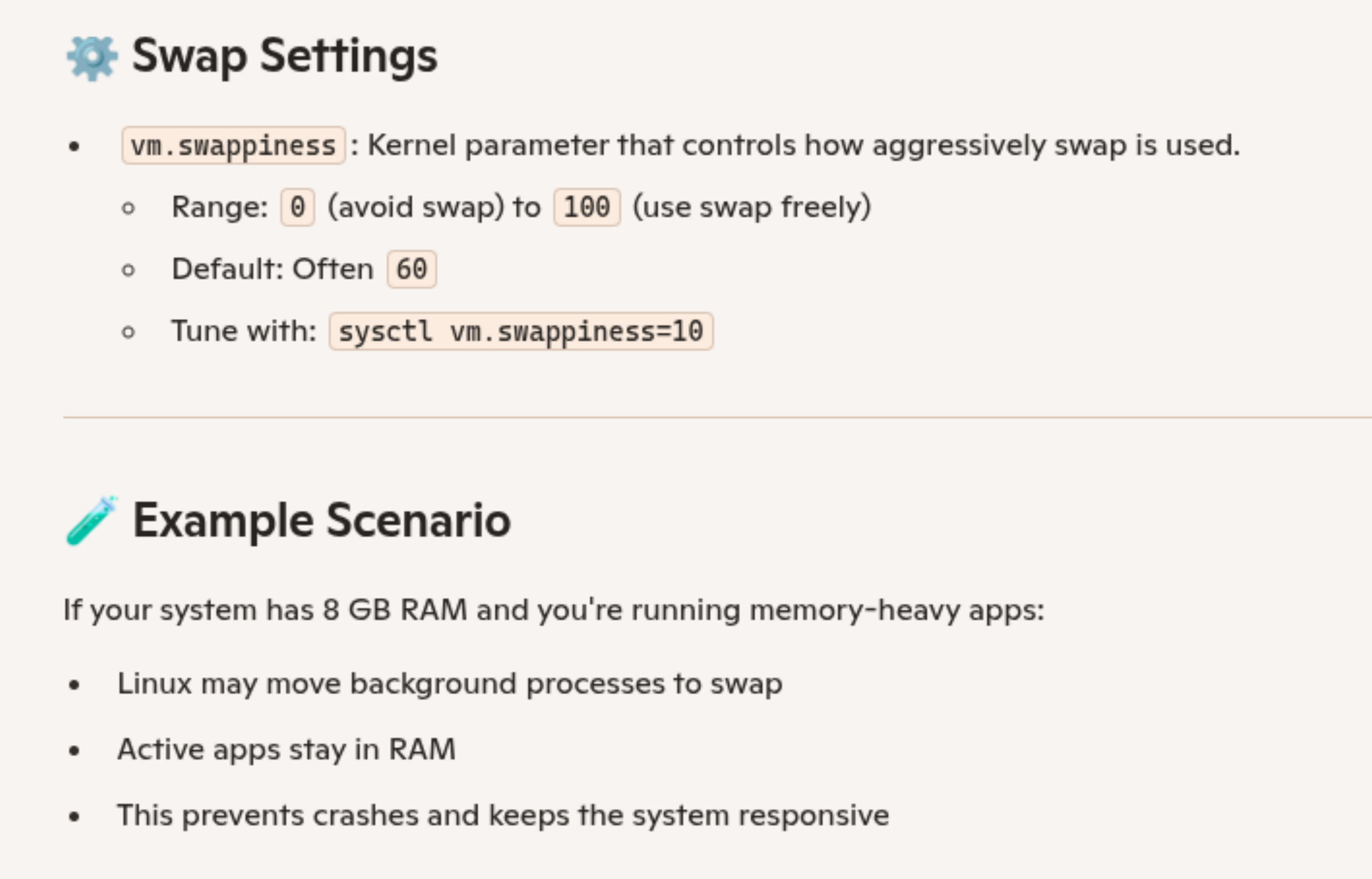
Type	Description
Swap Partition	Dedicated disk partition for swap
Swap File	Regular file used as swap space

Both are managed by the kernel and can be resized or added dynamically.

📊 Monitoring Swap Usage

Use these commands:

- `top` or `htop`: Shows real-time swap usage
- `free -h`: Displays total, used, and free swap
- `swapon --show`: Lists active swap devices



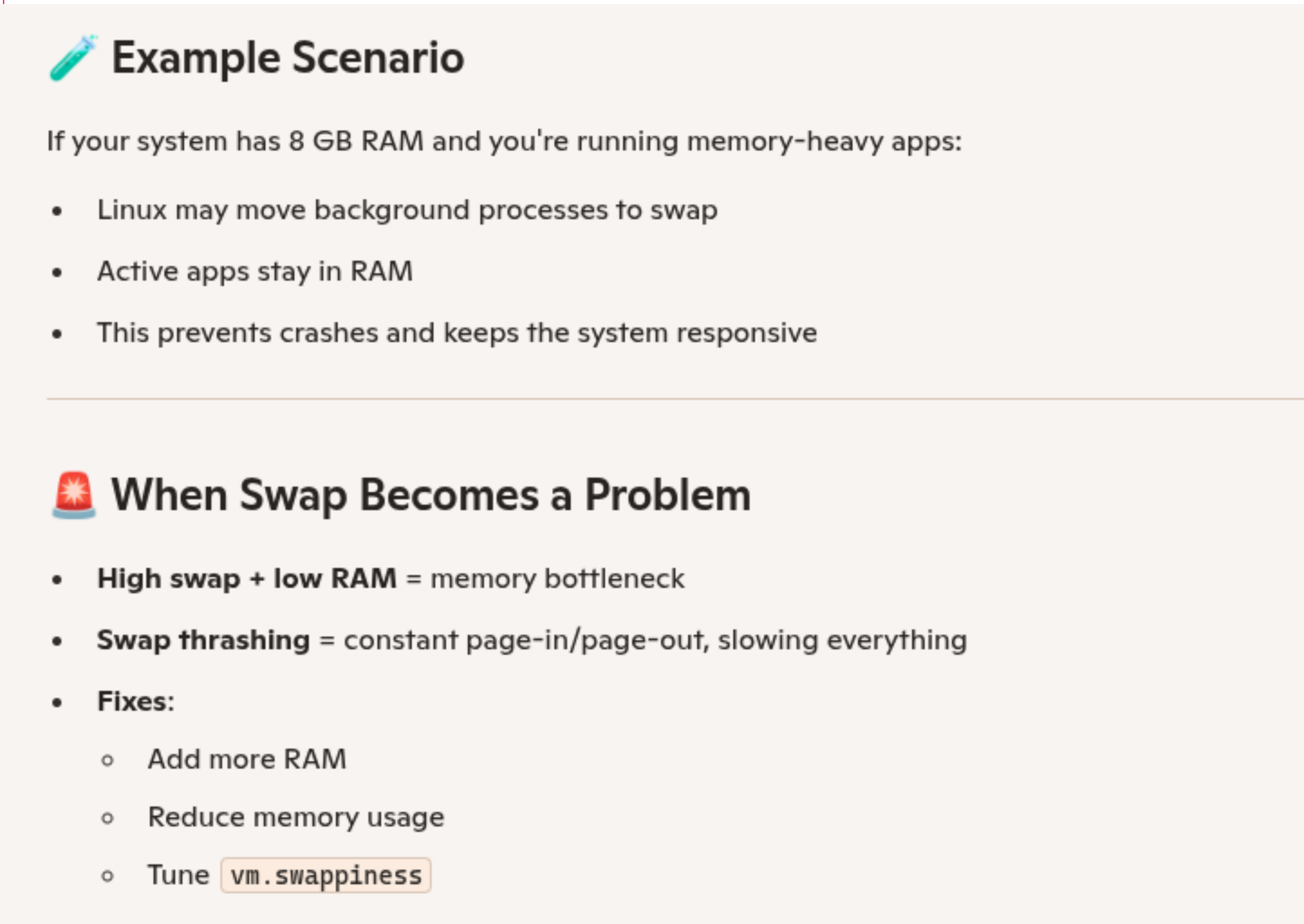
⚙️ Swap Settings

- `vm.swappiness`: Kernel parameter that controls how aggressively swap is used.
  - Range: `0` (avoid swap) to `100` (use swap freely)
  - Default: Often `60`
  - Tune with: `sysctl vm.swappiness=10`

🧪 Example Scenario

If your system has 8 GB RAM and you're running memory-heavy apps:

- Linux may move background processes to swap
- Active apps stay in RAM
- This prevents crashes and keeps the system responsive



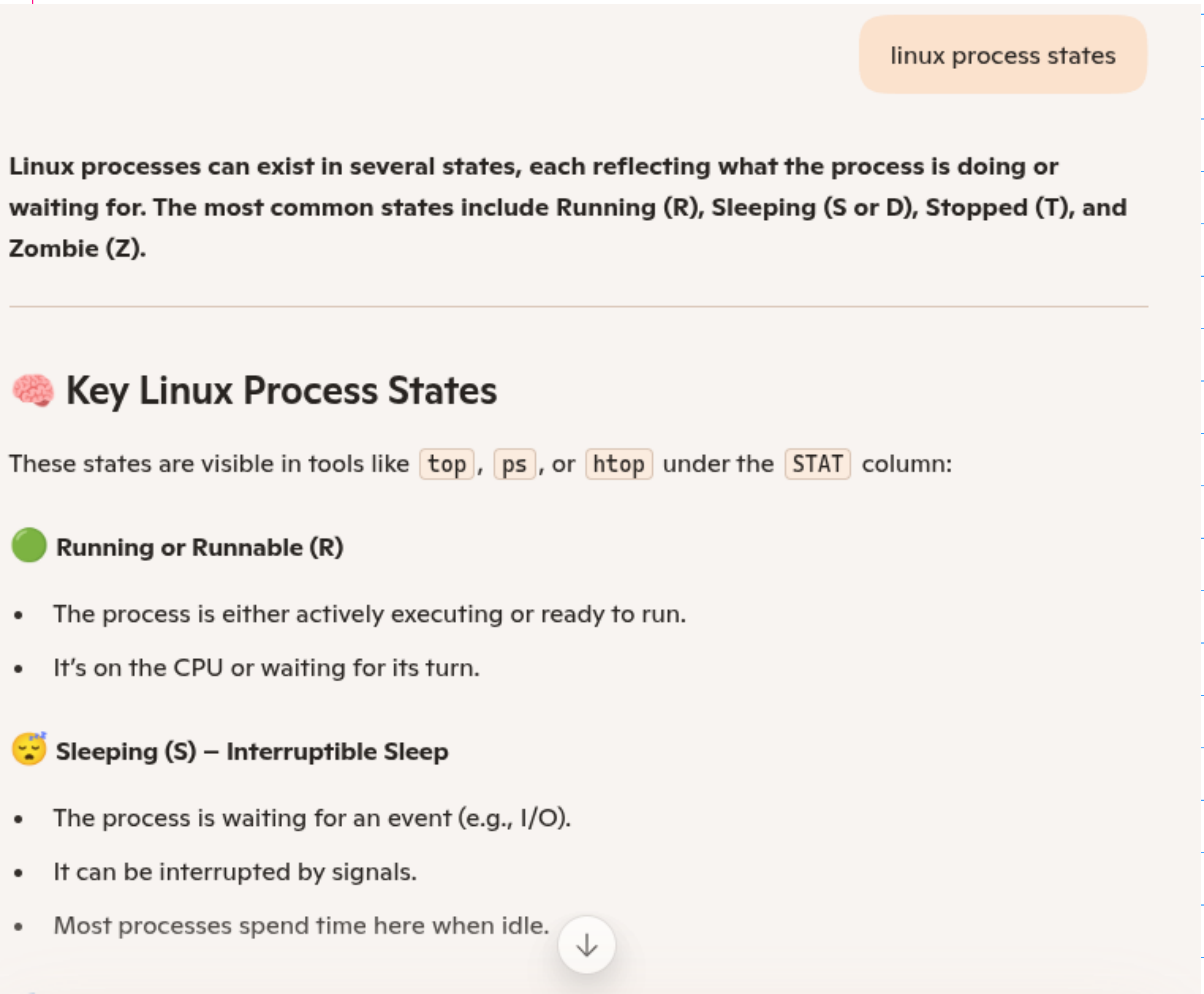
🧪 Example Scenario

If your system has 8 GB RAM and you're running memory-heavy apps:

- Linux may move background processes to swap
- Active apps stay in RAM
- This prevents crashes and keeps the system responsive

🚨 When Swap Becomes a Problem

- **High swap + low RAM** = memory bottleneck
- **Swap thrashing** = constant page-in/page-out, slowing everything
- **Fixes:**
  - Add more RAM
  - Reduce memory usage
  - Tune `vm.swappiness`



linux process states

Linux processes can exist in several states, each reflecting what the process is doing or waiting for. The most common states include Running (R), Sleeping (S or D), Stopped (T), and Zombie (Z).

🧠 Key Linux Process States

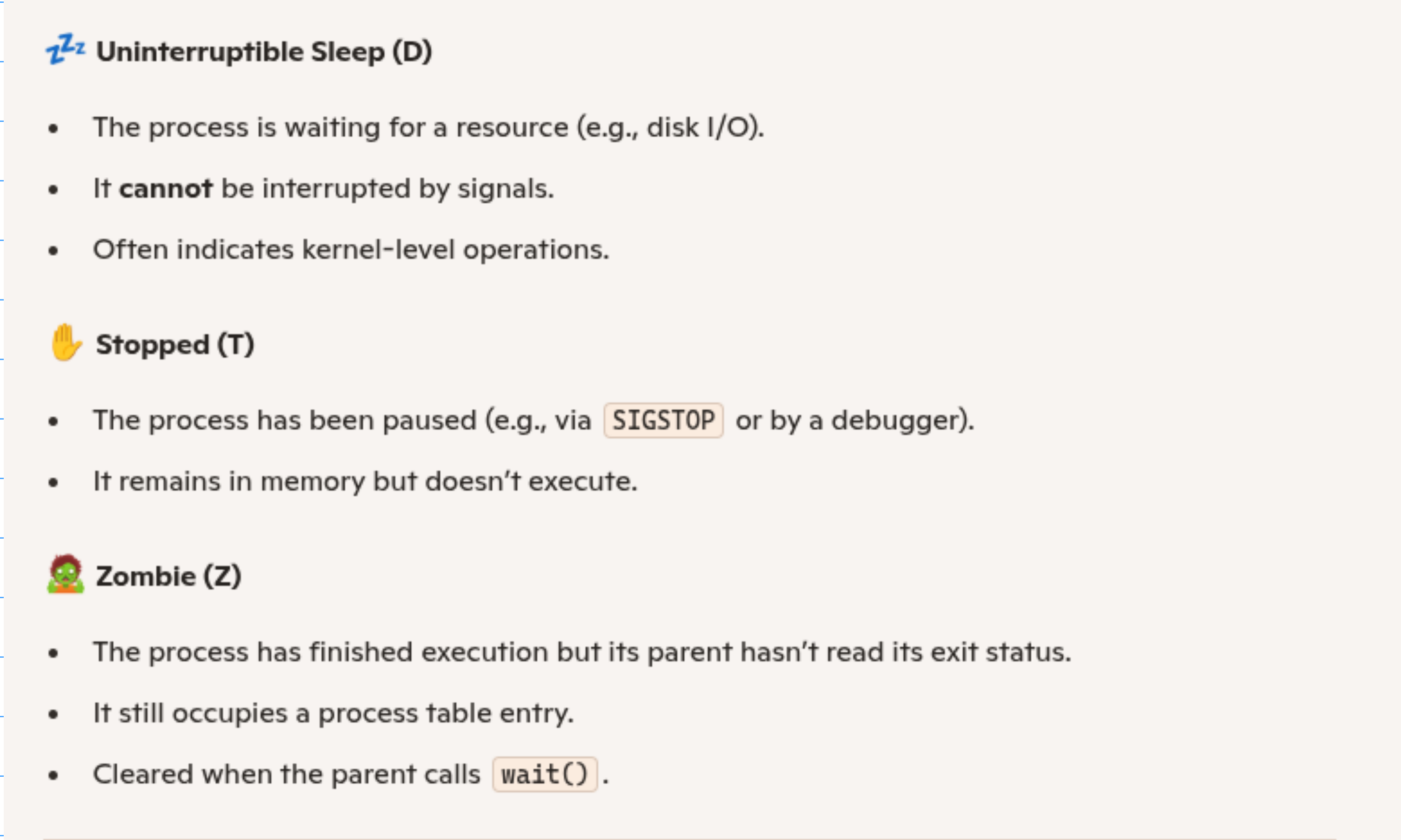
These states are visible in tools like `top`, `ps`, or `htop` under the `STAT` column:

🟢 Running or Runnable (R)

- The process is either actively executing or ready to run.
- It's on the CPU or waiting for its turn.

😴 Sleeping (S) – Interruptible Sleep

- The process is waiting for an event (e.g., I/O).
- It can be interrupted by signals.
- Most processes spend time here when idle.



💤 Uninterruptible Sleep (D)

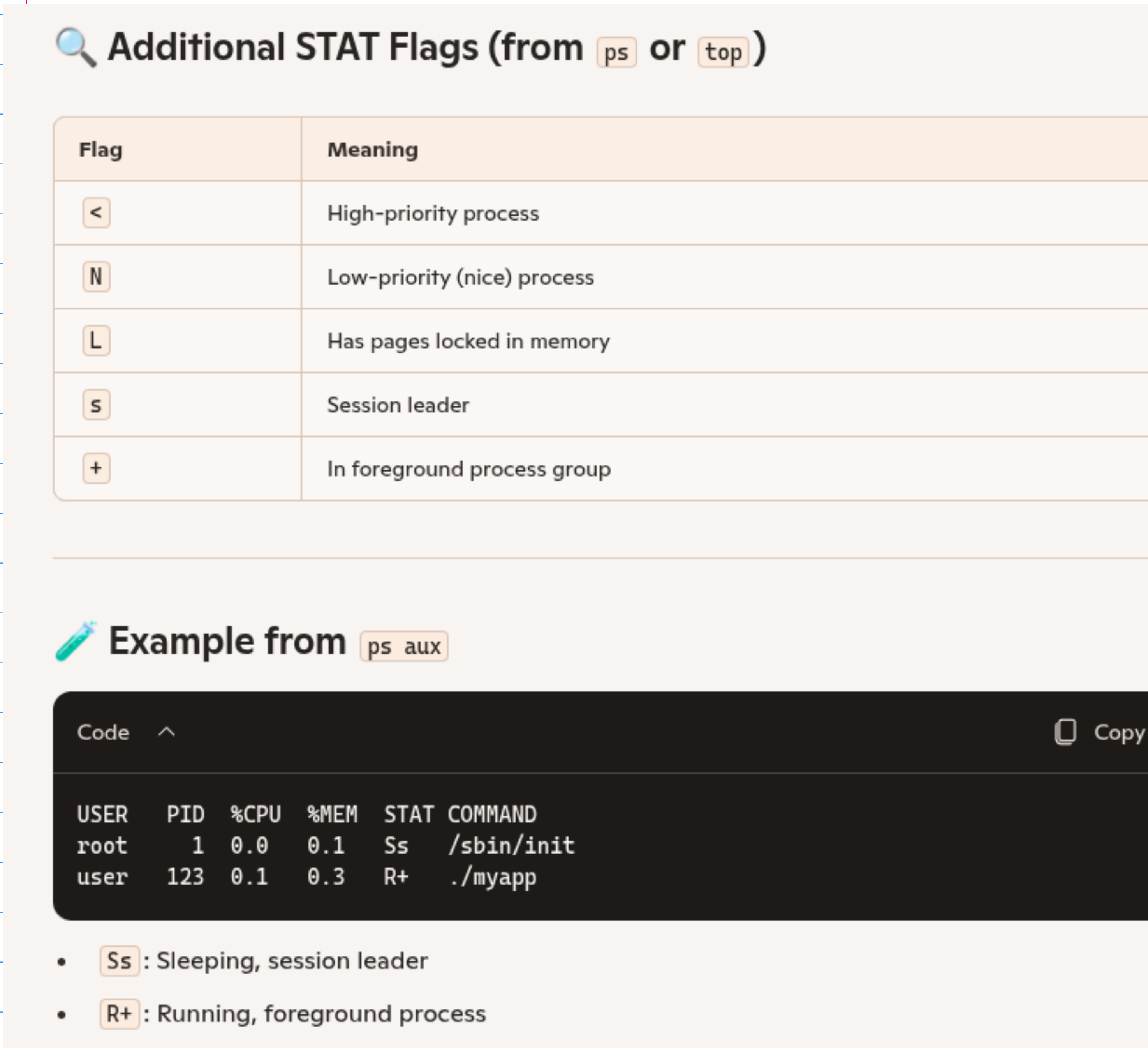
- The process is waiting for a resource (e.g., disk I/O).
- It **cannot** be interrupted by signals.
- Often indicates kernel-level operations.

✋ Stopped (T)

- The process has been paused (e.g., via `SIGSTOP` or by a debugger).
- It remains in memory but doesn't execute.

🧟 Zombie (Z)

- The process has finished execution but its parent hasn't read its exit status.
- It still occupies a process table entry.
- Cleared when the parent calls `wait()`.



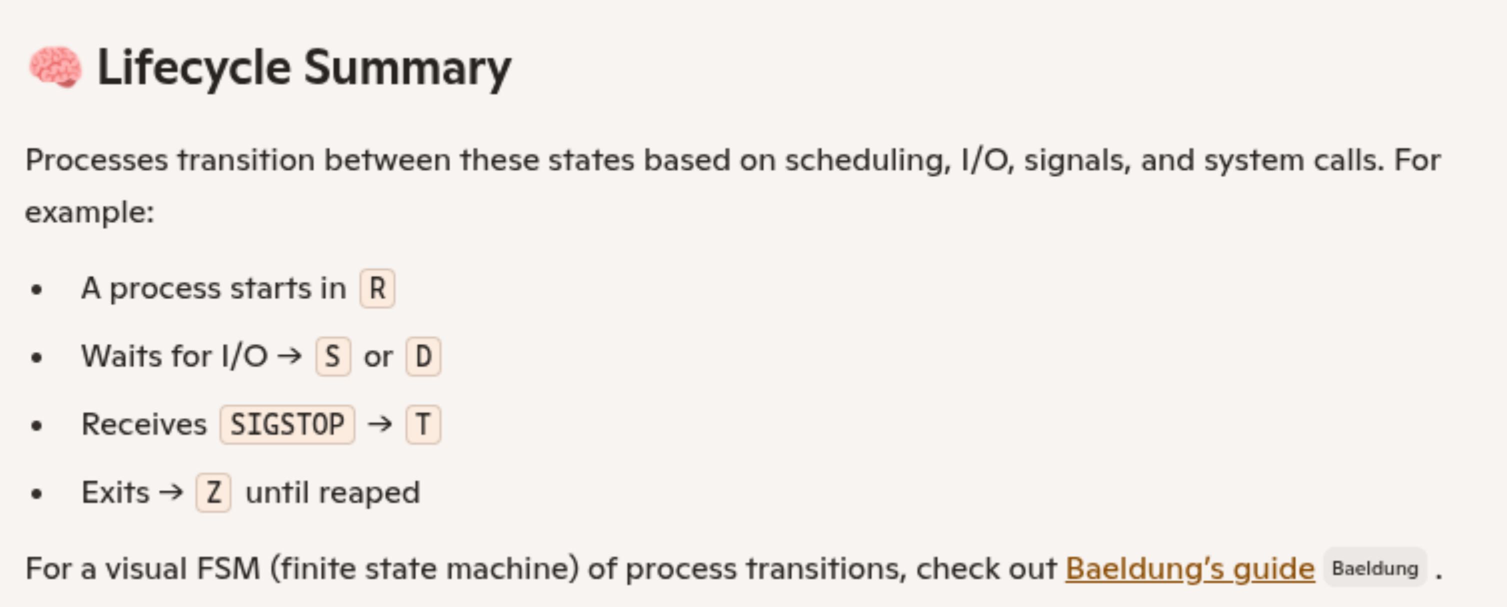
🔍 Additional STAT Flags (from `ps` or `top`)

Flag	Meaning
`<`	High-priority process
`N`	Low-priority (nice) process
`L`	Has pages locked in memory
`s`	Session leader
`+`	In foreground process group

🧪 Example from `ps aux`

```
USER   PID  %CPU  %MEM  STAT COMMAND
root     1  0.0   0.1   Ss   /sbin/init
user   123  0.1   0.3   R+   ./myapp
```

- `Ss`: Sleeping, session leader
- `R+`: Running, foreground process

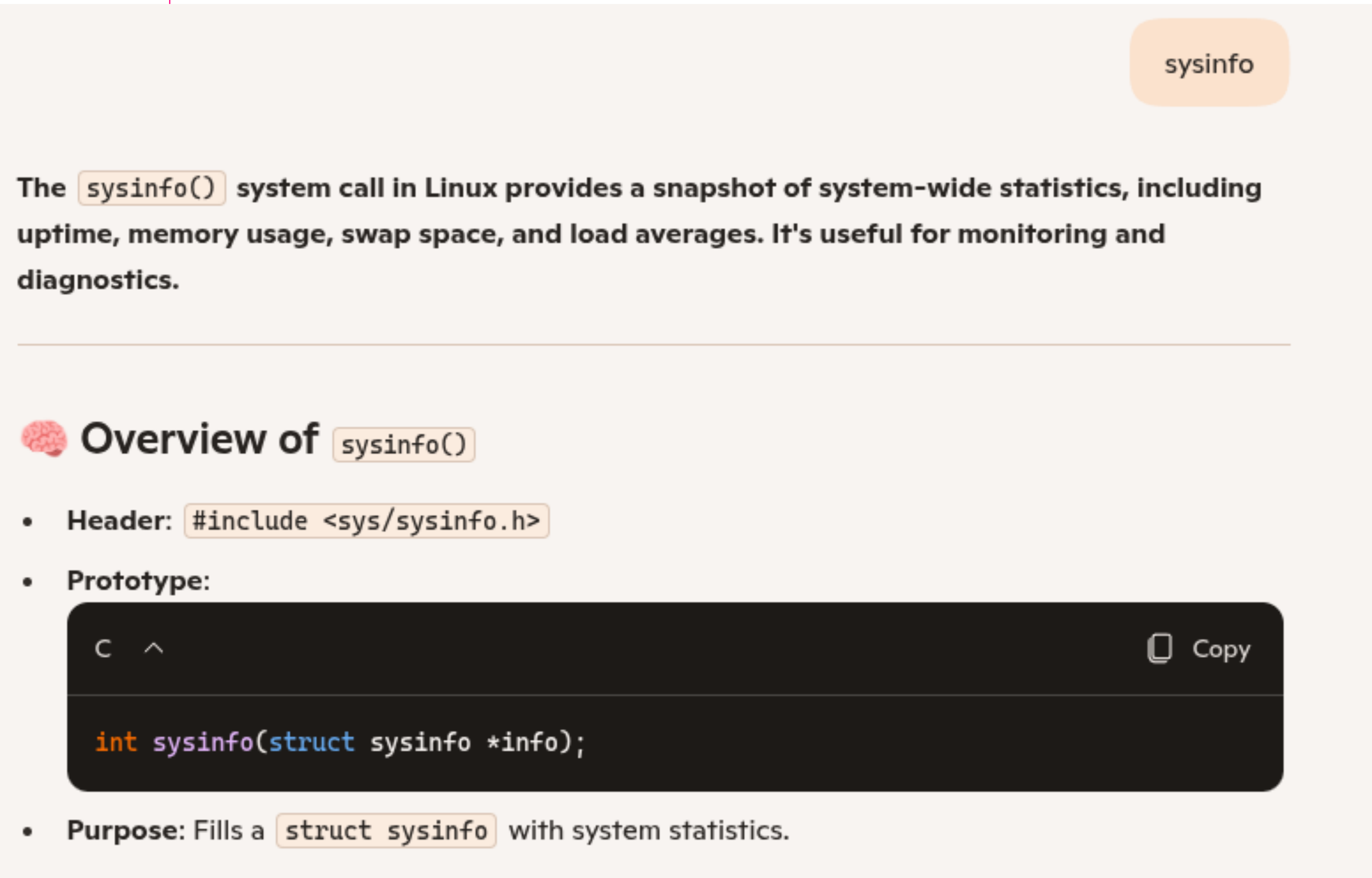


🧠 Lifecycle Summary

Processes transition between these states based on scheduling, I/O, signals, and system calls. For example:

- A process starts in `R`
- Waits for I/O → `S` or `D`
- Receives `SIGSTOP` → `T`
- Exits → `Z` until reaped

For a visual FSM (finite state machine) of process transitions, check out Baeldung's guide Baeldung .



sysinfo

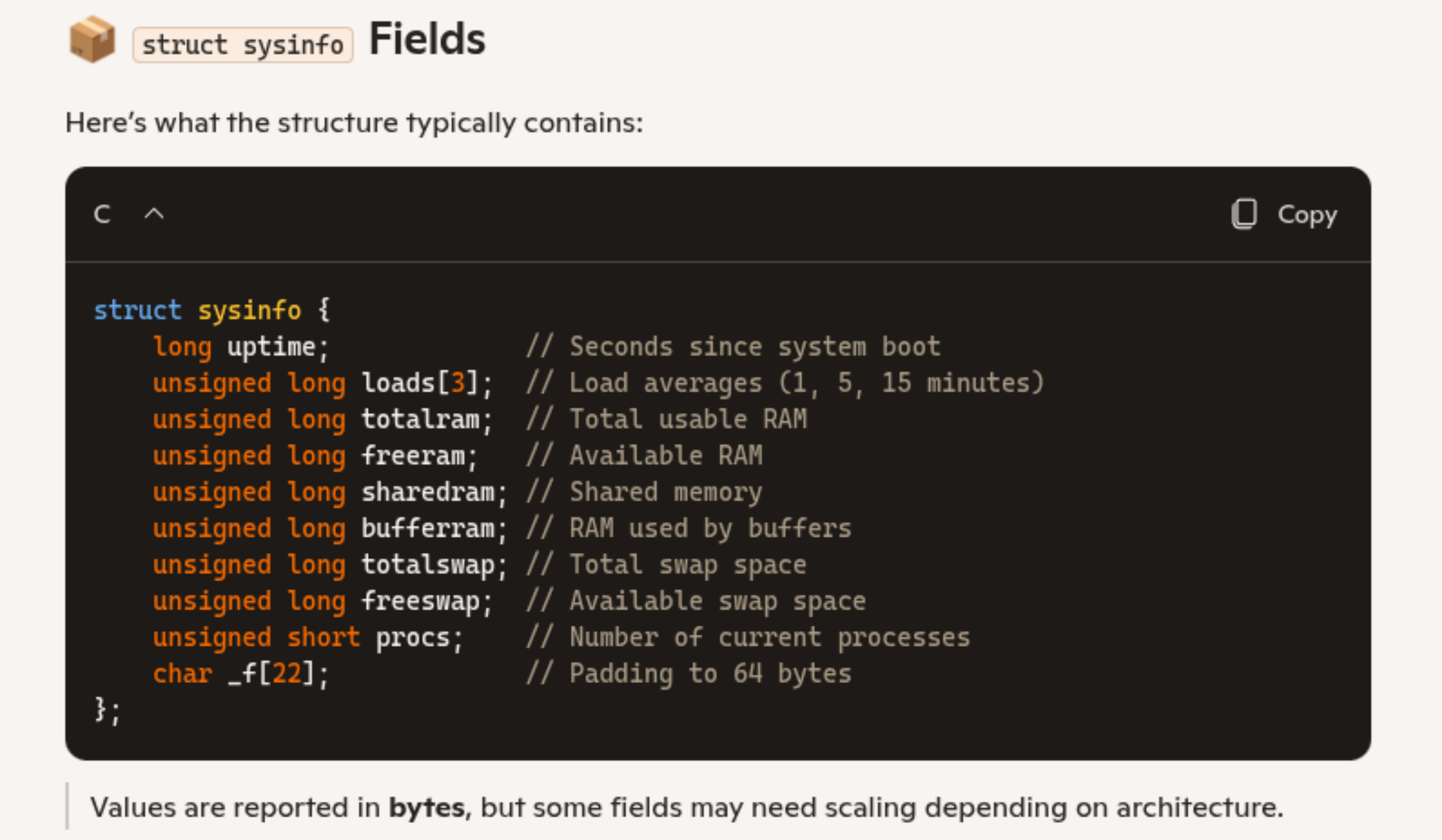
The `sysinfo()` system call in Linux provides a snapshot of system-wide statistics, including uptime, memory usage, swap space, and load averages. It's useful for monitoring and diagnostics.

🧠 Overview of `sysinfo()`

- **Header:** `#include <sys/sysinfo.h>`
- **Prototype:**

```c
int sysinfo(struct sysinfo *info);
```

- **Purpose:** Fills a `struct sysinfo` with system statistics.

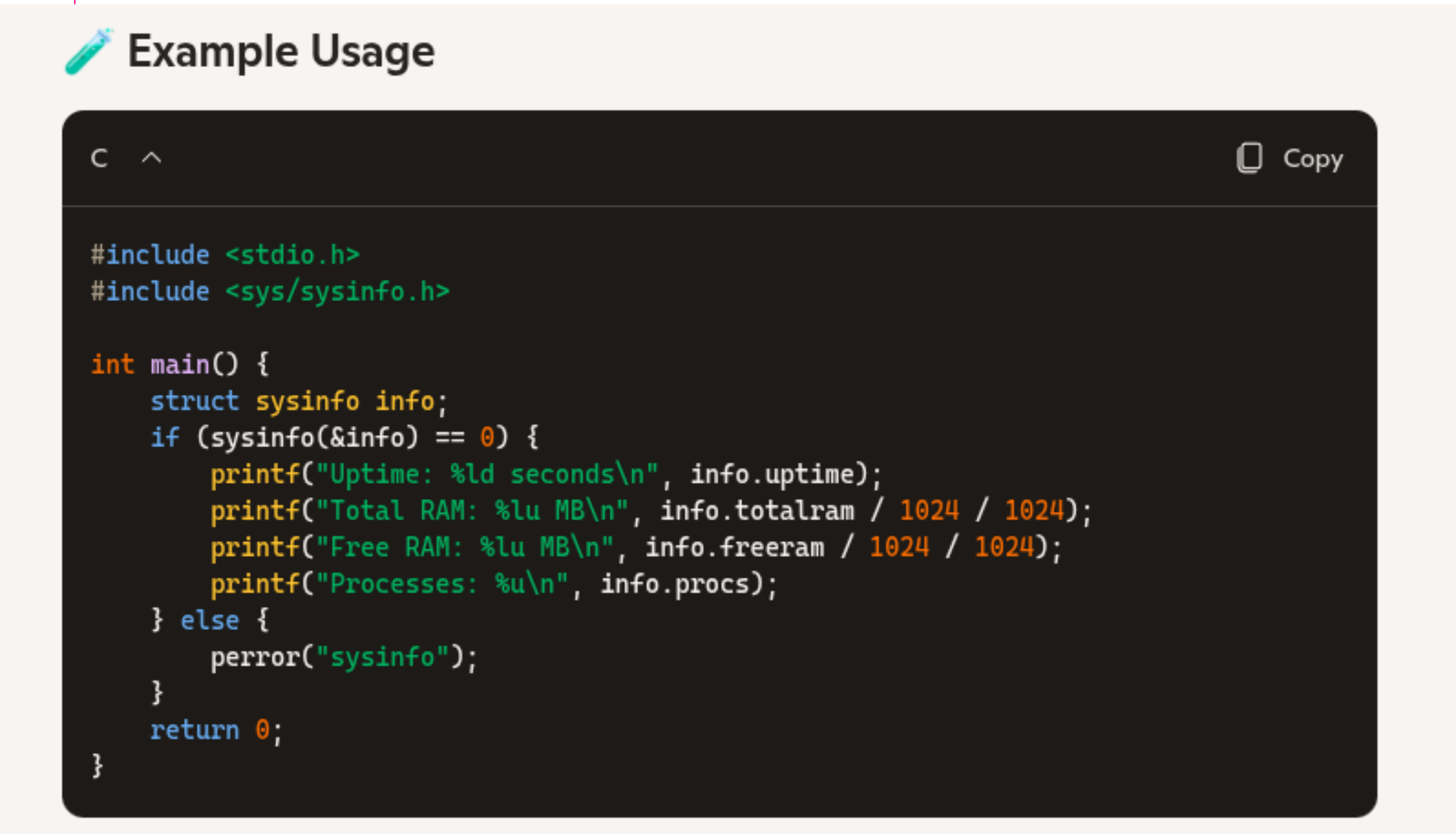


📦 `struct sysinfo` Fields

Here's what the structure typically contains:

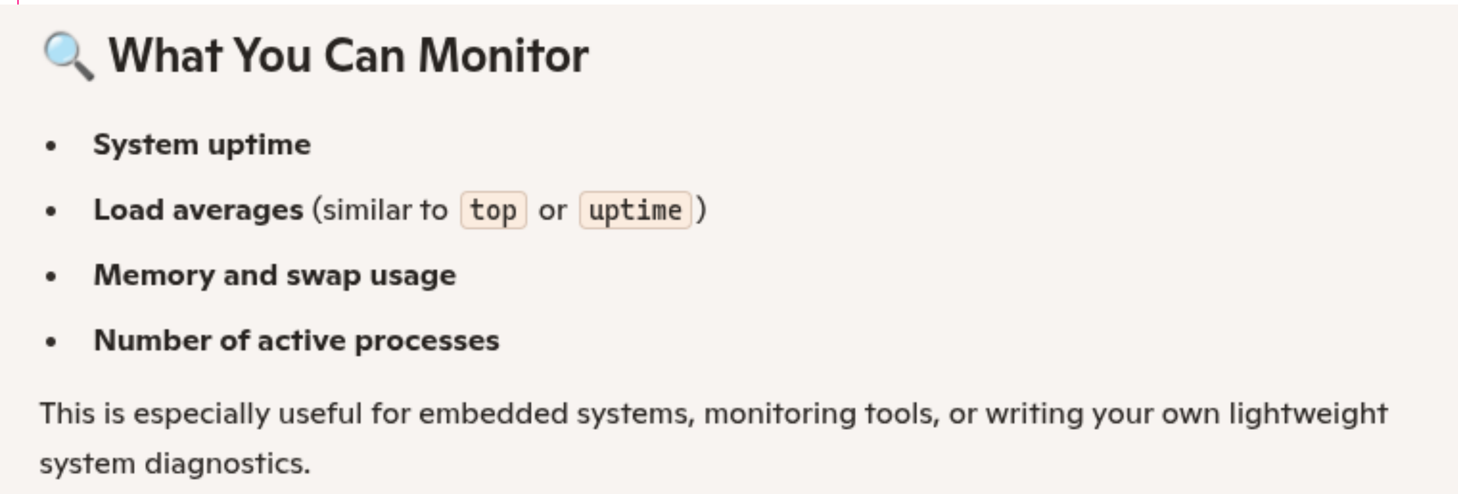
```c
struct sysinfo {
    long uptime;             // Seconds since system boot
    unsigned long loads[3];  // Load averages (1, 5, 15 minutes)
    unsigned long totalram;  // Total usable RAM
    unsigned long freeram;   // Available RAM
    unsigned long sharedram; // Shared memory
    unsigned long bufferram; // RAM used by buffers
    unsigned long totalswap; // Total swap space
    unsigned long freeswap;  // Available swap space
    unsigned short procs;    // Number of current processes
    char _f[22];             // Padding to 64 bytes
};
```

> Values are reported in **bytes**, but some fields may need scaling depending on architecture.



🧪 Example Usage

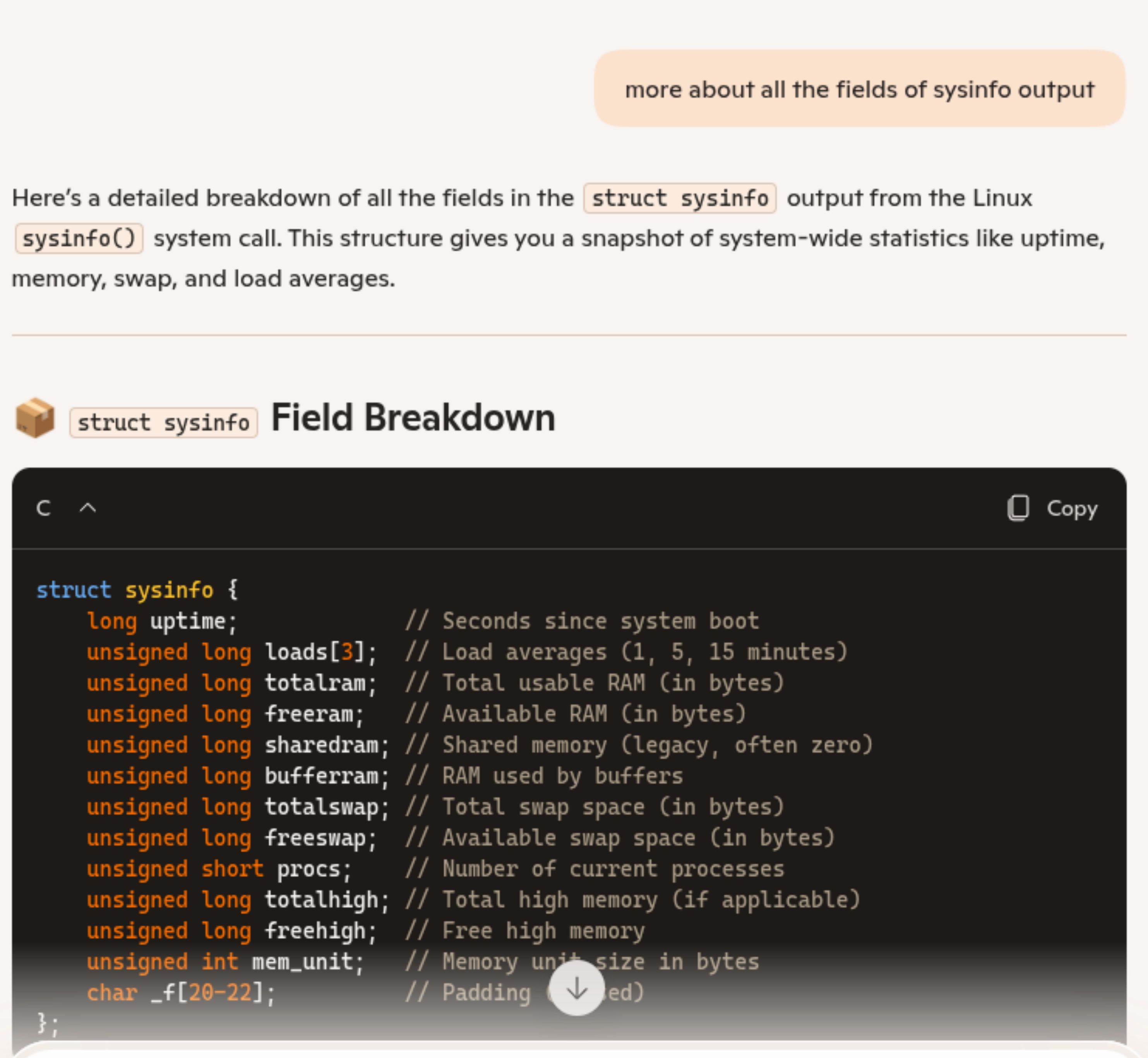
```c
#include <stdio.h>
#include <sys/sysinfo.h>

int main() {
    struct sysinfo info;
    if (sysinfo(&info) == 0) {
        printf("Uptime: %ld seconds\n", info.uptime);
        printf("Total RAM: %lu MB\n", info.totalram / 1024 / 1024);
        printf("Free RAM: %lu MB\n", info.freeram / 1024 / 1024);
        printf("Processes: %u\n", info.procs);
    } else {
        perror("sysinfo");
    }
    return 0;
}
```



🔍 What You Can Monitor

- **System uptime**
- **Load averages** (similar to `top` or `uptime`)
- **Memory and swap usage**
- **Number of active processes**

This is especially useful for embedded systems, monitoring tools, or writing your own lightweight system diagnostics.

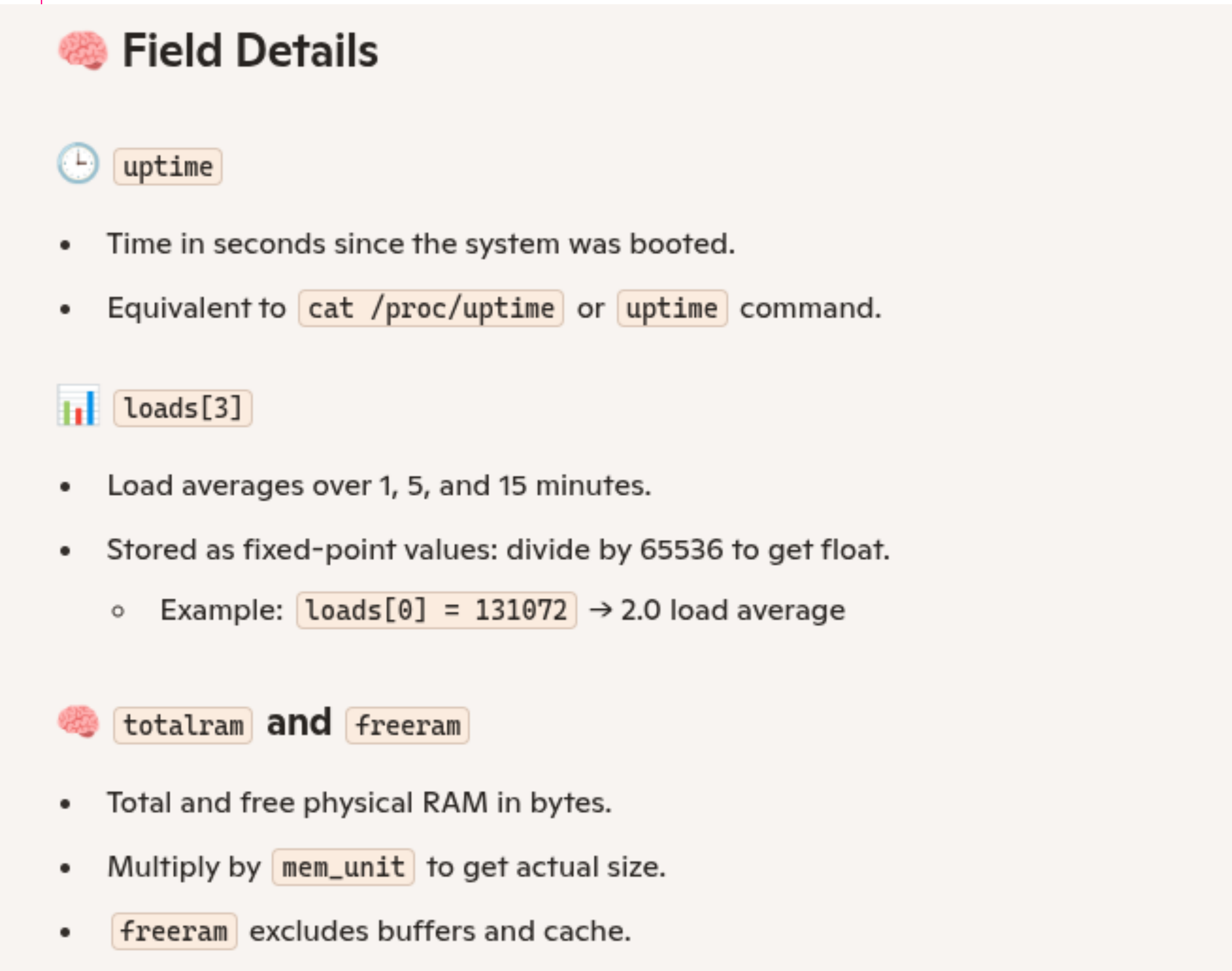


more about all the fields of sysinfo output

Here's a detailed breakdown of all the fields in the `struct sysinfo` output from the Linux `sysinfo()` system call. This structure gives you a snapshot of system-wide statistics like uptime, memory, swap, and load averages.

📦 `struct sysinfo` Field Breakdown

```c
struct sysinfo {
    long uptime;             // Seconds since system boot
    unsigned long loads[3];  // Load averages (1, 5, 15 minutes)
    unsigned long totalram;  // Total usable RAM (in bytes)
    unsigned long freeram;   // Available RAM (in bytes)
    unsigned long sharedram; // Shared memory (legacy, often zero)
    unsigned long bufferram; // RAM used by buffers
    unsigned long totalswap; // Total swap space (in bytes)
    unsigned long freeswap;  // Available swap space (in bytes)
    unsigned short procs;    // Number of current processes
    unsigned long totalhigh; // Total high memory (if applicable)
    unsigned long freehigh;  // Free high memory
    unsigned int mem_unit;   // Memory unit size in bytes
    char _f[20-22];          // Padding (unused)
};
```



🧠 Field Details

🕒 `uptime`

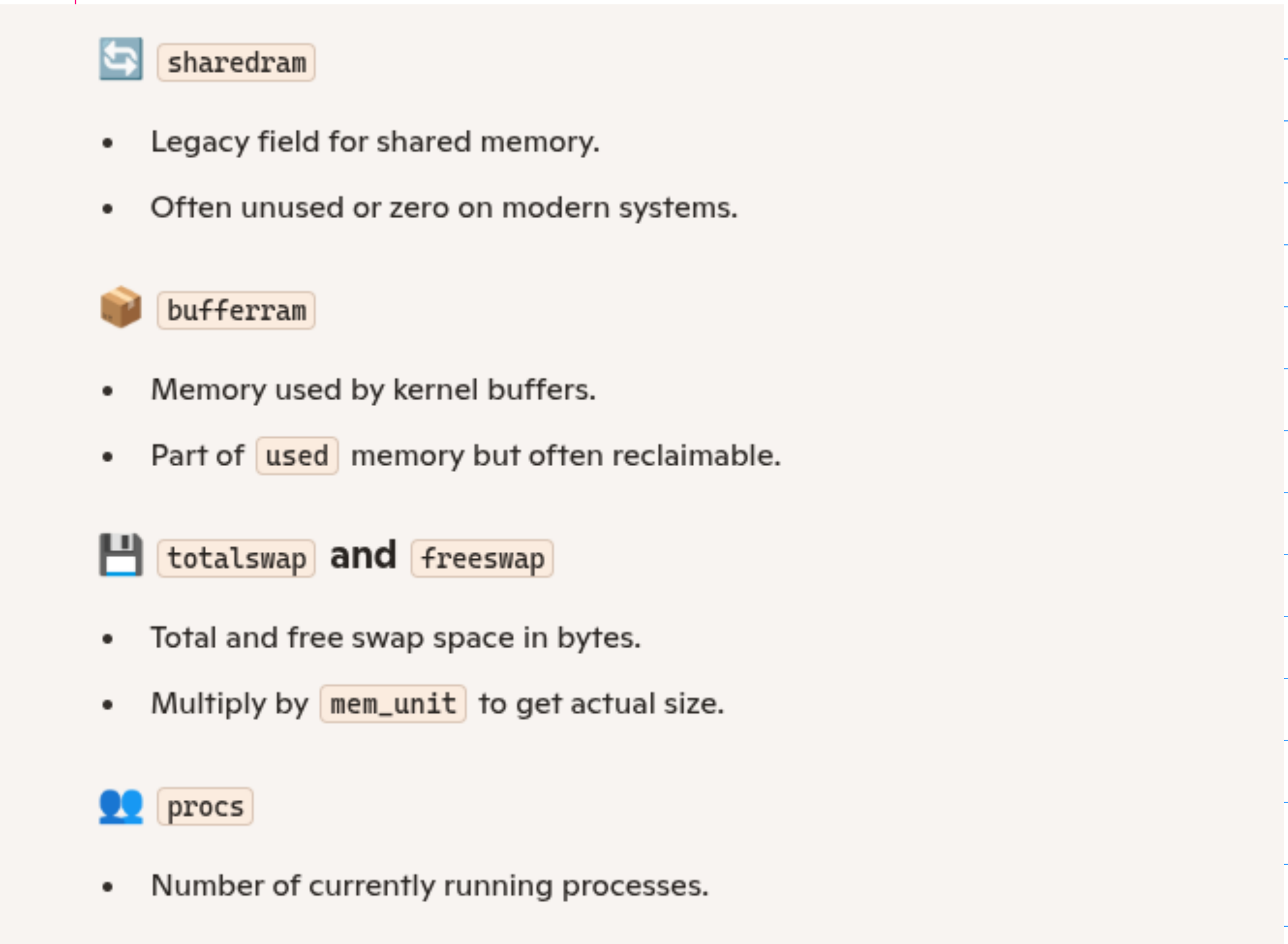
- Time in seconds since the system was booted.
- Equivalent to `cat /proc/uptime` or `uptime` command.

📊 `loads[3]`

- Load averages over 1, 5, and 15 minutes.
- Stored as fixed-point values: divide by 65536 to get float.
  - Example: `loads[0] = 131072` → 2.0 load average

🧠 `totalram` and `freeram`

- Total and free physical RAM in bytes.
- Multiply by `mem_unit` to get actual size.
- `freeram` excludes buffers and cache.



🔄 `sharedram`

- Legacy field for shared memory.
- Often unused or zero on modern systems.

📦 `bufferram`

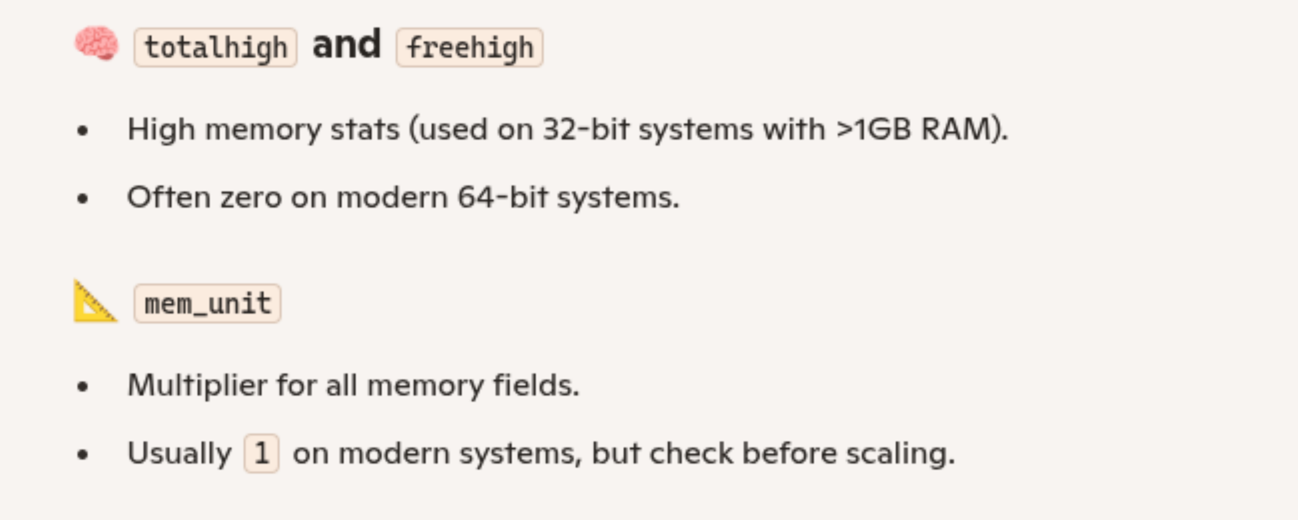
- Memory used by kernel buffers.
- Part of `used` memory but often reclaimable.

💾 `totalswap` and `freeswap`

- Total and free swap space in bytes.
- Multiply by `mem_unit` to get actual size.

👥 `procs`

- Number of currently running processes.

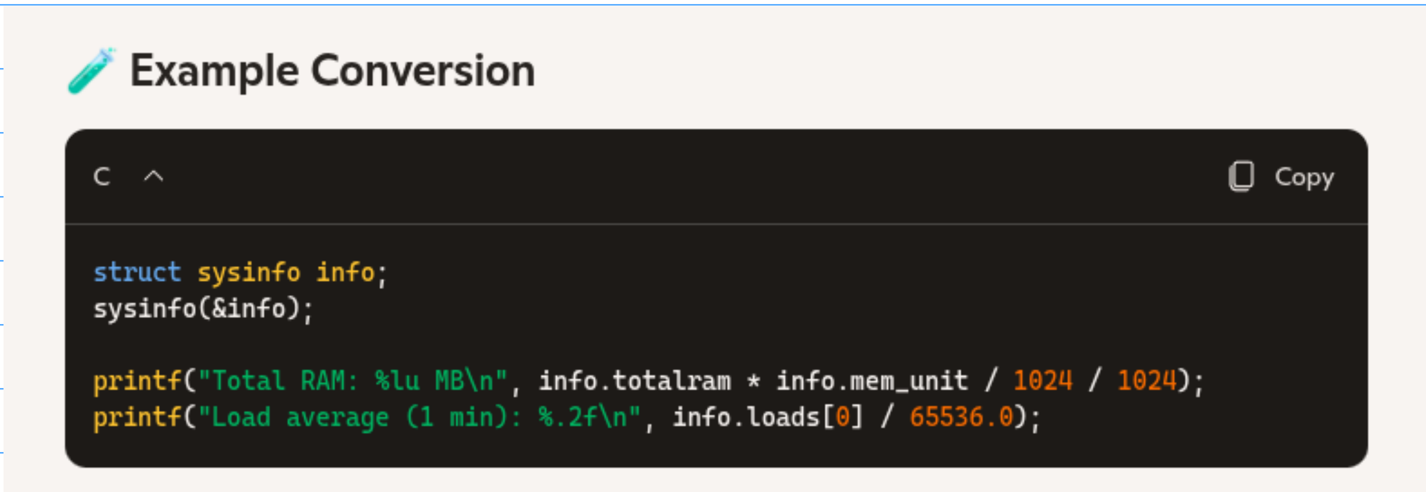


🧠 `totalhigh` and `freehigh`

- High memory stats (used on 32-bit systems with >1GB RAM).
- Often zero on modern 64-bit systems.

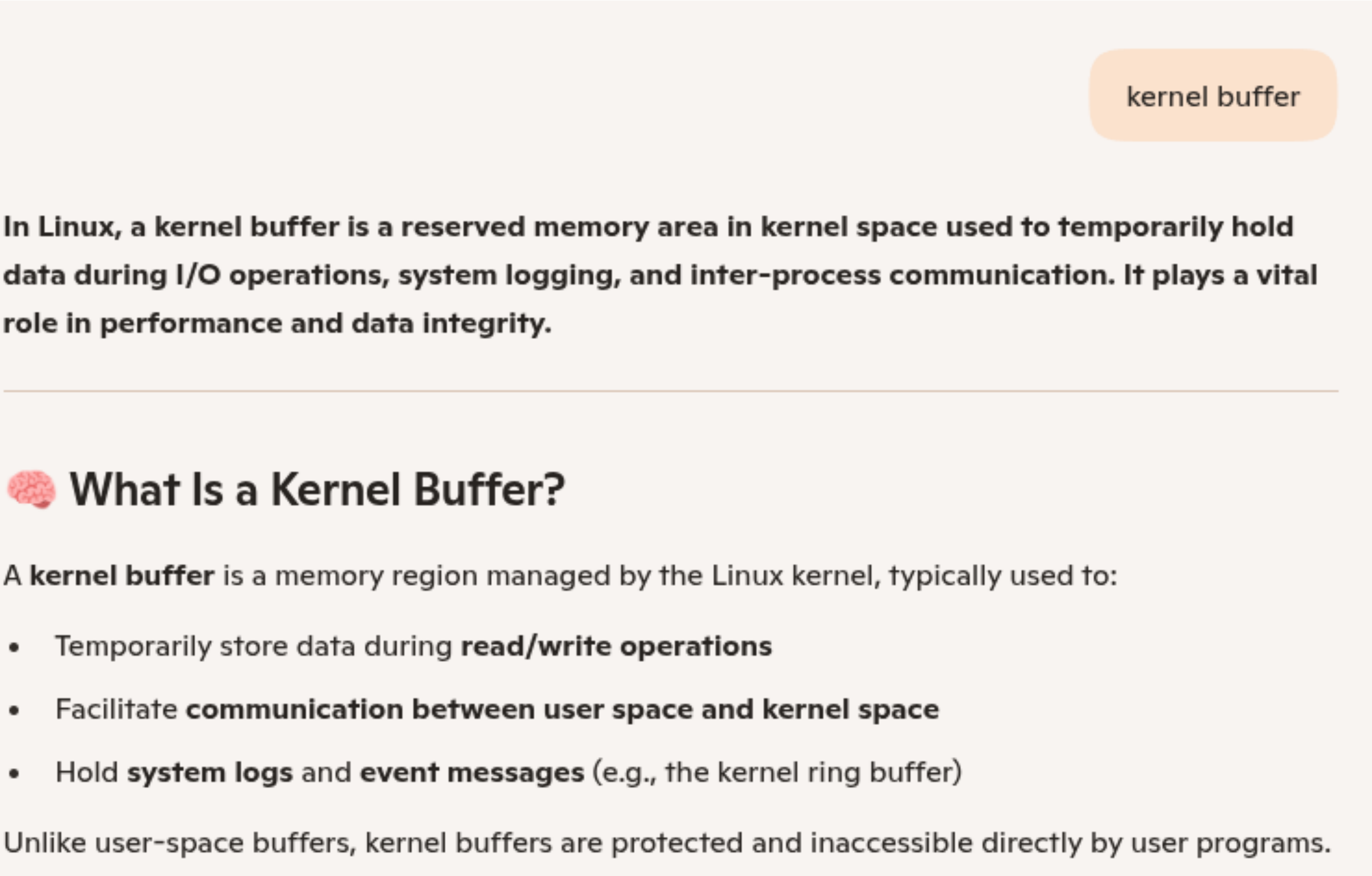
📐 `mem_unit`

- Multiplier for all memory fields.
- Usually `1` on modern systems, but check before scaling.



🧪 Example Conversion

```c
struct sysinfo info;
sysinfo(&info);

printf("Total RAM: %lu MB\n", info.totalram * info.mem_unit / 1024 / 1024);
printf("Load average (1 min): %.2f\n", info.loads[0] / 65536.0);
```



kernel buffer

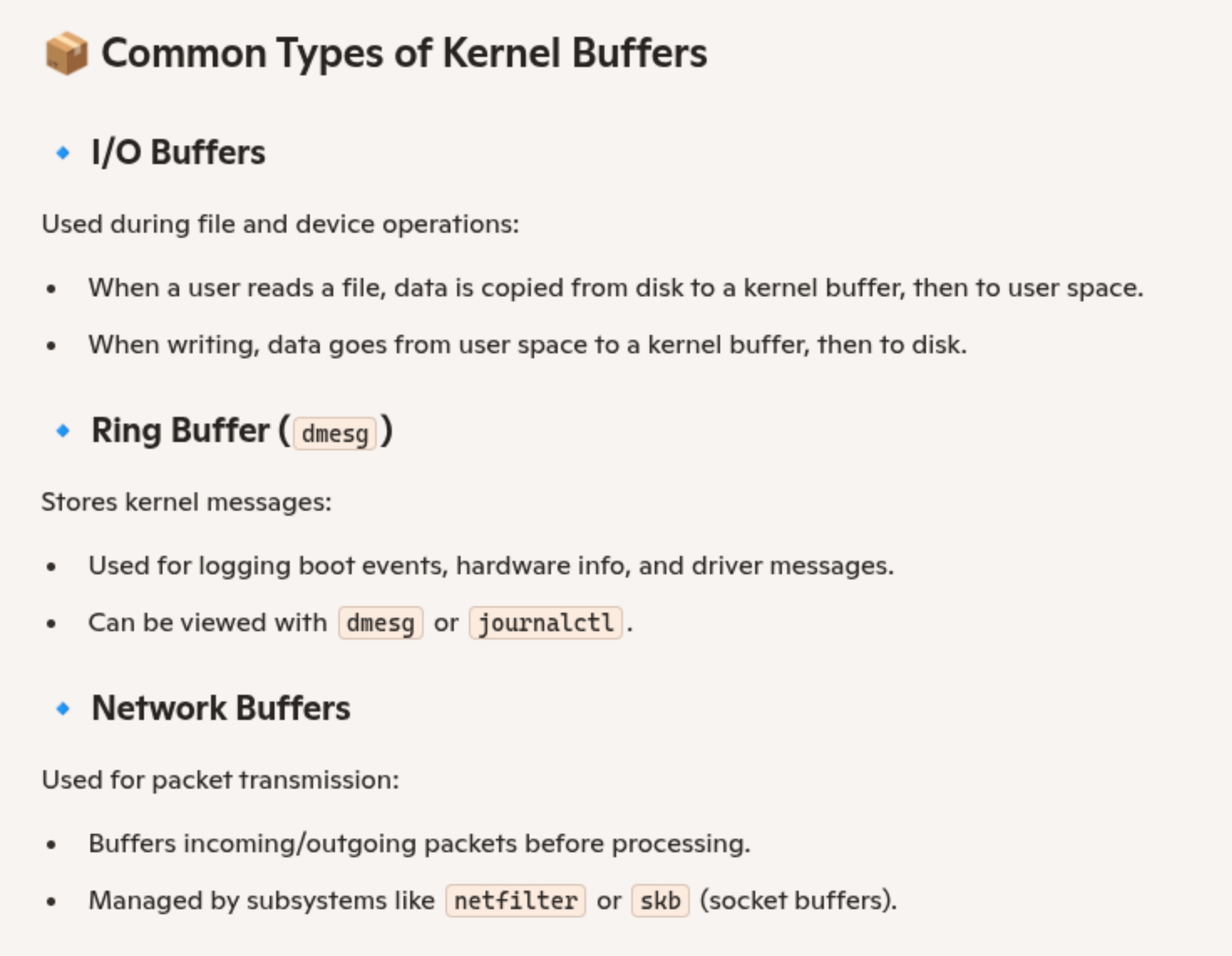
In Linux, a kernel buffer is a reserved memory area in kernel space used to temporarily hold data during I/O operations, system logging, and inter-process communication. It plays a vital role in performance and data integrity.

🧠 What Is a Kernel Buffer?

A **kernel buffer** is a memory region managed by the Linux kernel, typically used to:

- Temporarily store data during **read/write operations**
- Facilitate **communication between user space and kernel space**
- Hold **system logs** and **event messages** (e.g., the kernel ring buffer)

Unlike user-space buffers, kernel buffers are protected and inaccessible directly by user programs.



📦 Common Types of Kernel Buffers

🔹 I/O Buffers

Used during file and device operations:

- When a user reads a file, data is copied from disk to a kernel buffer, then to user space.
- When writing, data goes from user space to a kernel buffer, then to disk.

🔹 Ring Buffer (`dmesg`)

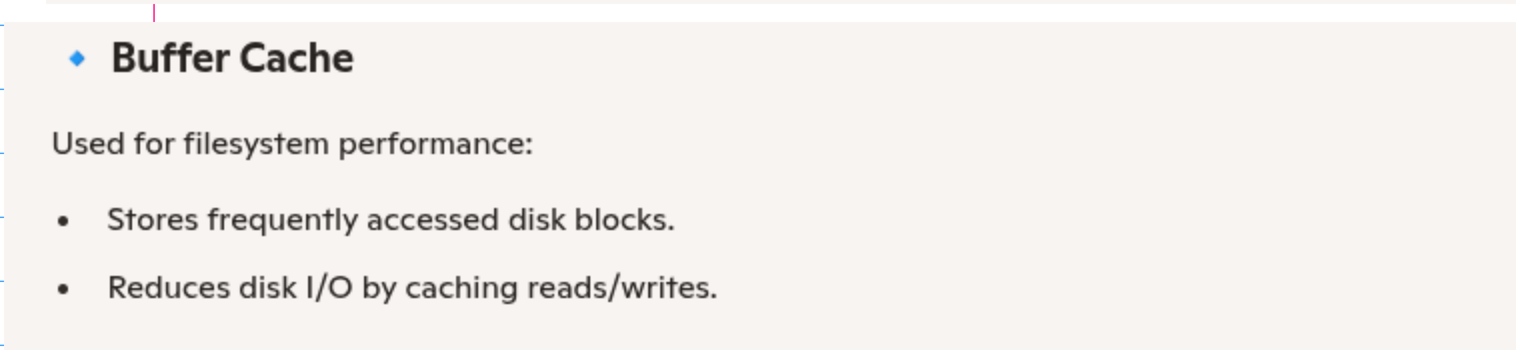
Stores kernel messages:

- Used for logging boot events, hardware info, and driver messages.
- Can be viewed with `dmesg` or `journalctl`.

🔹 Network Buffers

Used for packet transmission:

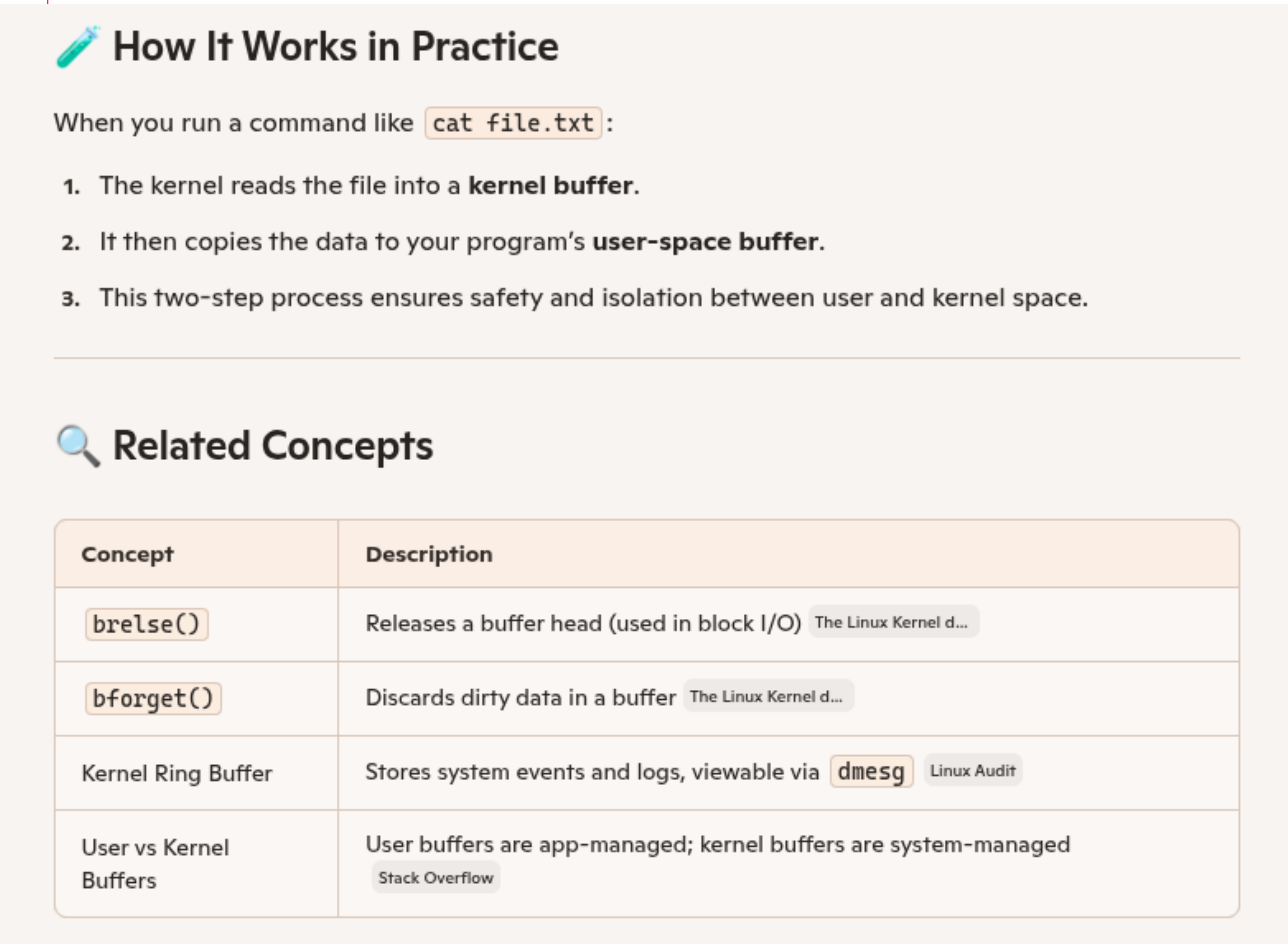
- Buffers incoming/outgoing packets before processing.
- Managed by subsystems like `netfilter` or `skb` (socket buffers).



🔹 Buffer Cache

Used for filesystem performance:

- Stores frequently accessed disk blocks.
- Reduces disk I/O by caching reads/writes.



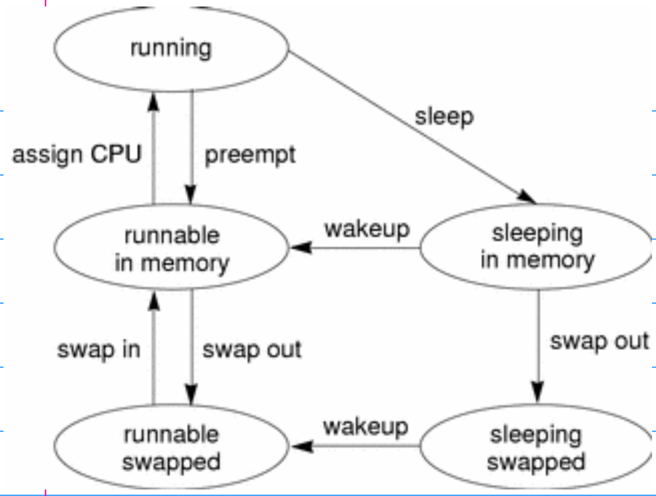
🧪 How It Works in Practice

When you run a command like `cat file.txt`:

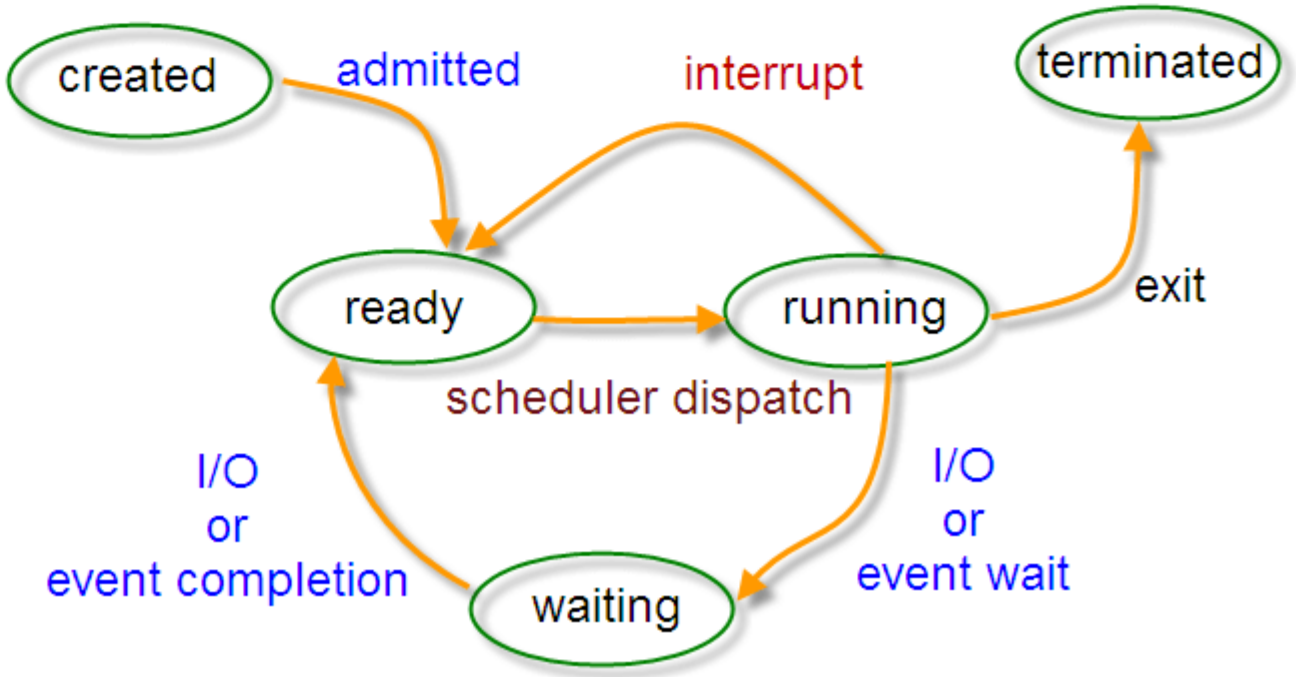
1. The kernel reads the file into a **kernel buffer**.
2. It then copies the data to your program's **user-space buffer**.
3. This two-step process ensures safety and isolation between user and kernel space.

🔍 Related Concepts

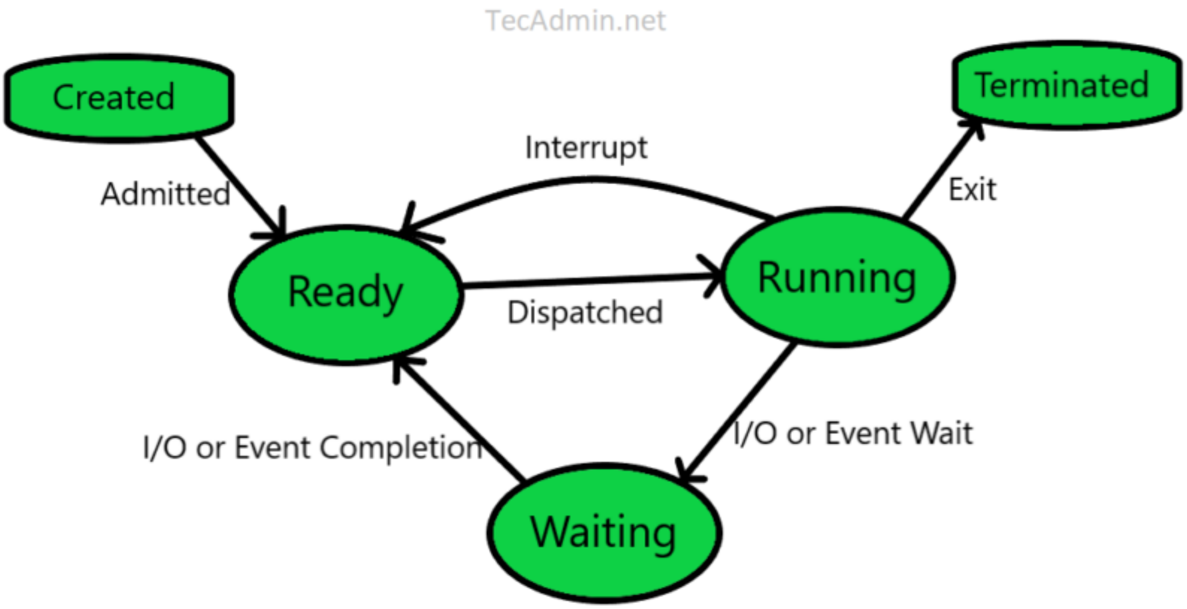
Concept	**Description**
`brelse()`	Releases a buffer head (used in block I/O) The Linux Kernel d...
`bforget()`	Discards dirty data in a buffer The Linux Kernel d...
Kernel Ring Buffer	Stores system events and logs, viewable via `dmesg` Linux Audit
User vs Kernel Buffers	User buffers are app-managed; kernel buffers are system-managed Stack Overflow

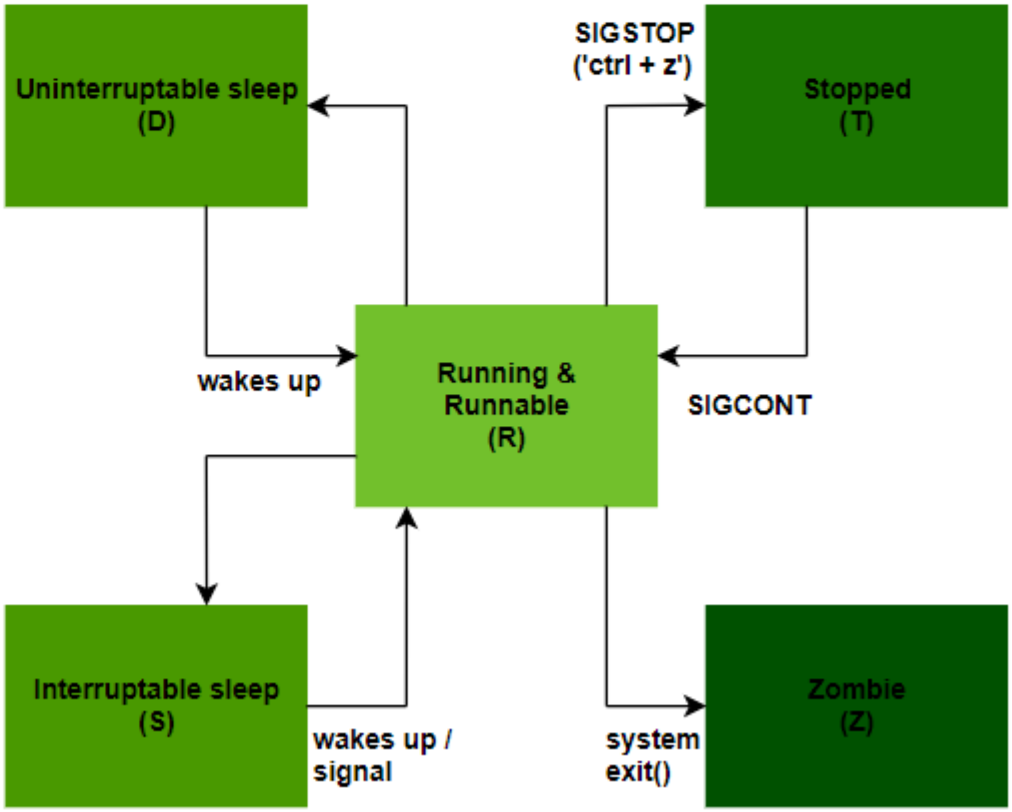


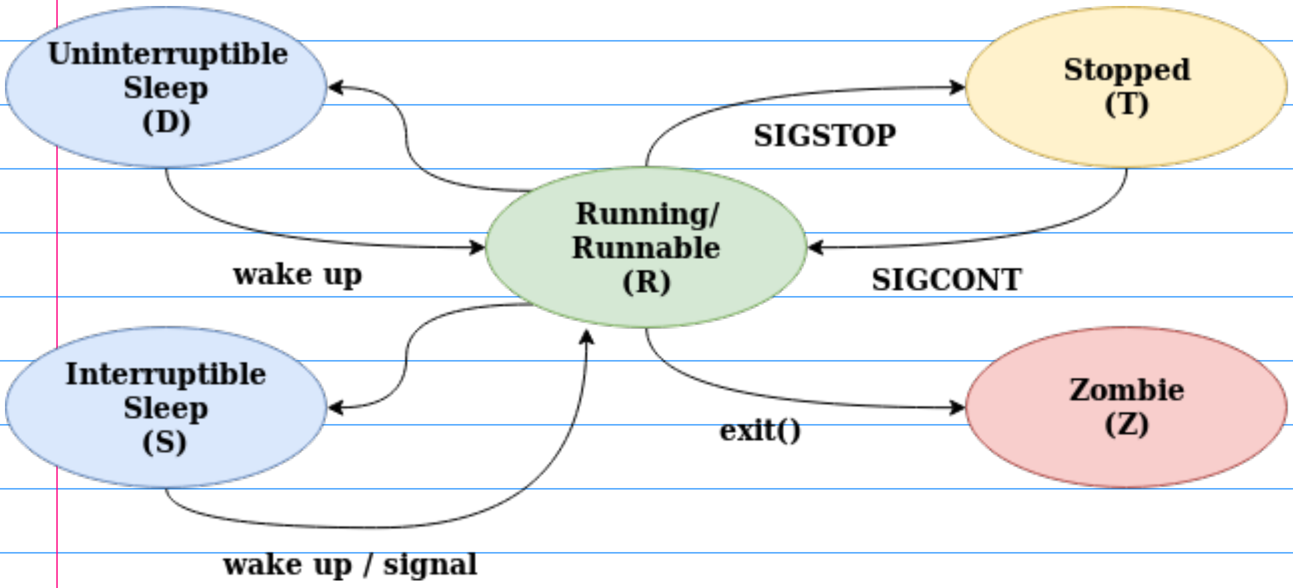
Process State

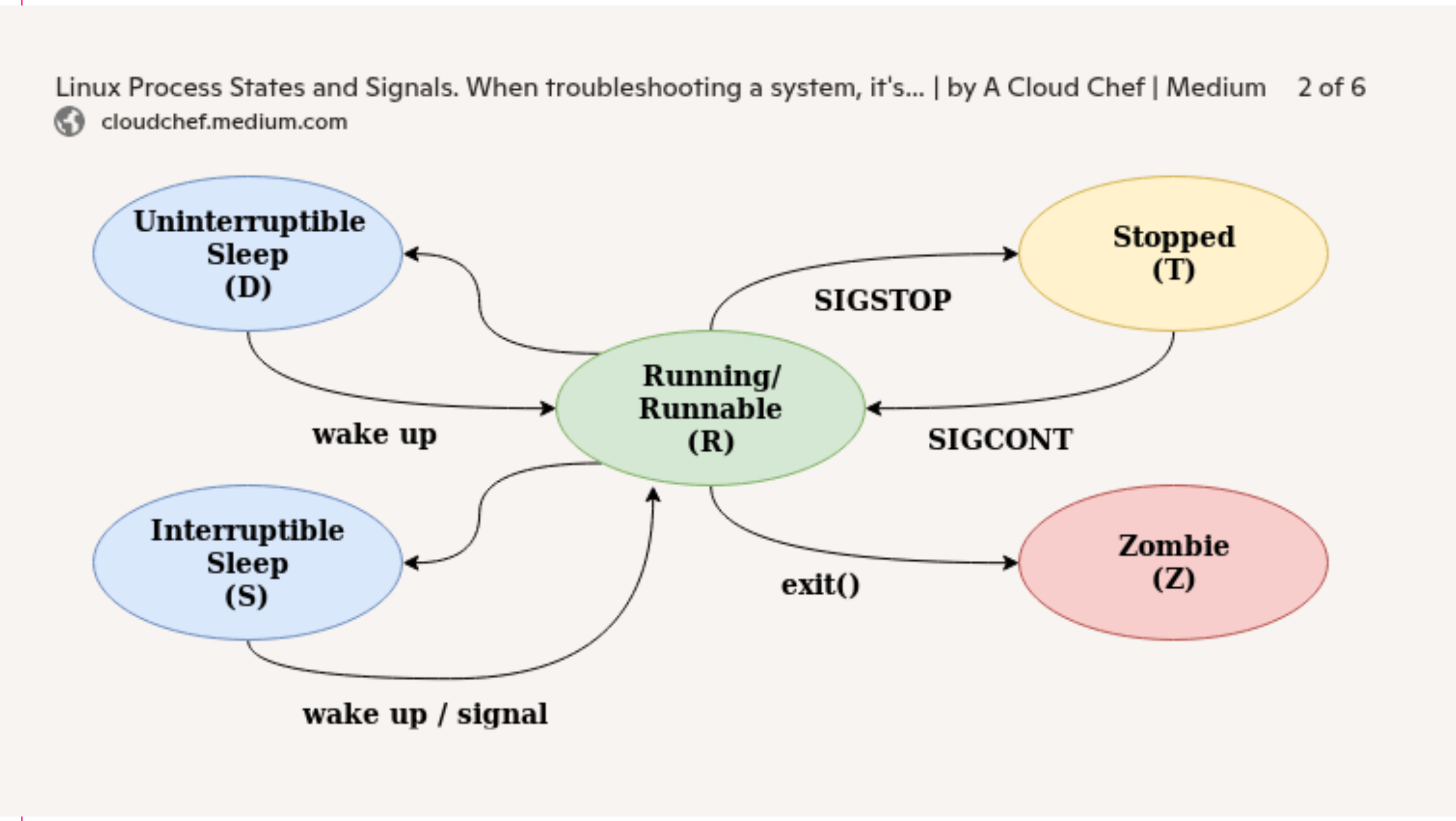


Linux Process States



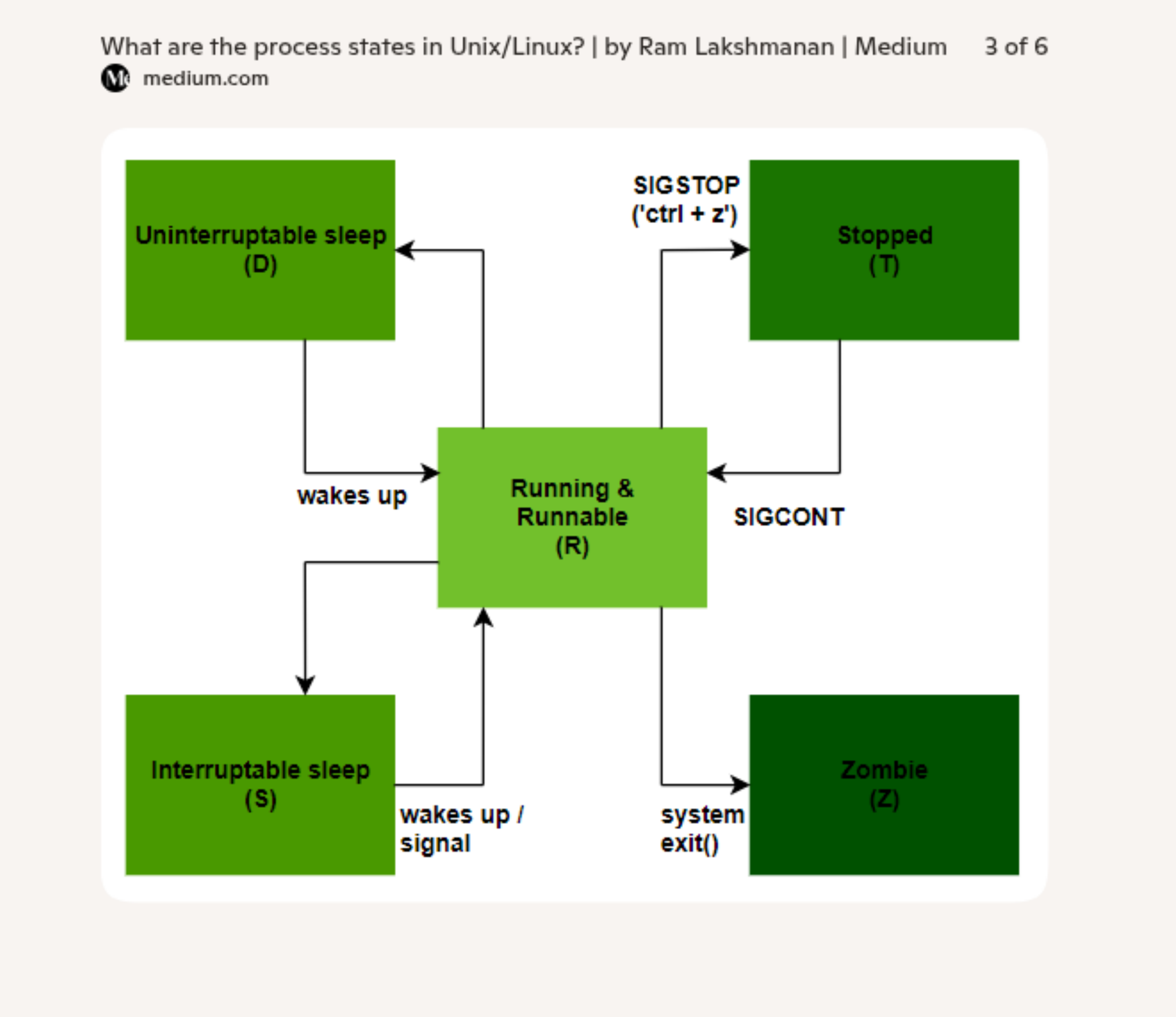

Uninterruptable sleep
(D)
SIGSTOP
('ctrl + z')
Stopped
(T)
wakes up
Running &
Runnable
(R)
SIGCONT
Interruptable sleep
(S)
wakes up /
signal
system
exit()
Zombie
(Z)


Uninterruptible
Sleep
(D)
Stopped
(T)
SIGSTOP
Running/
Runnable
(R)
wake up
SIGCONT
Interruptible
Sleep
(S)
Zombie
(Z)
exit()
wake up / signal



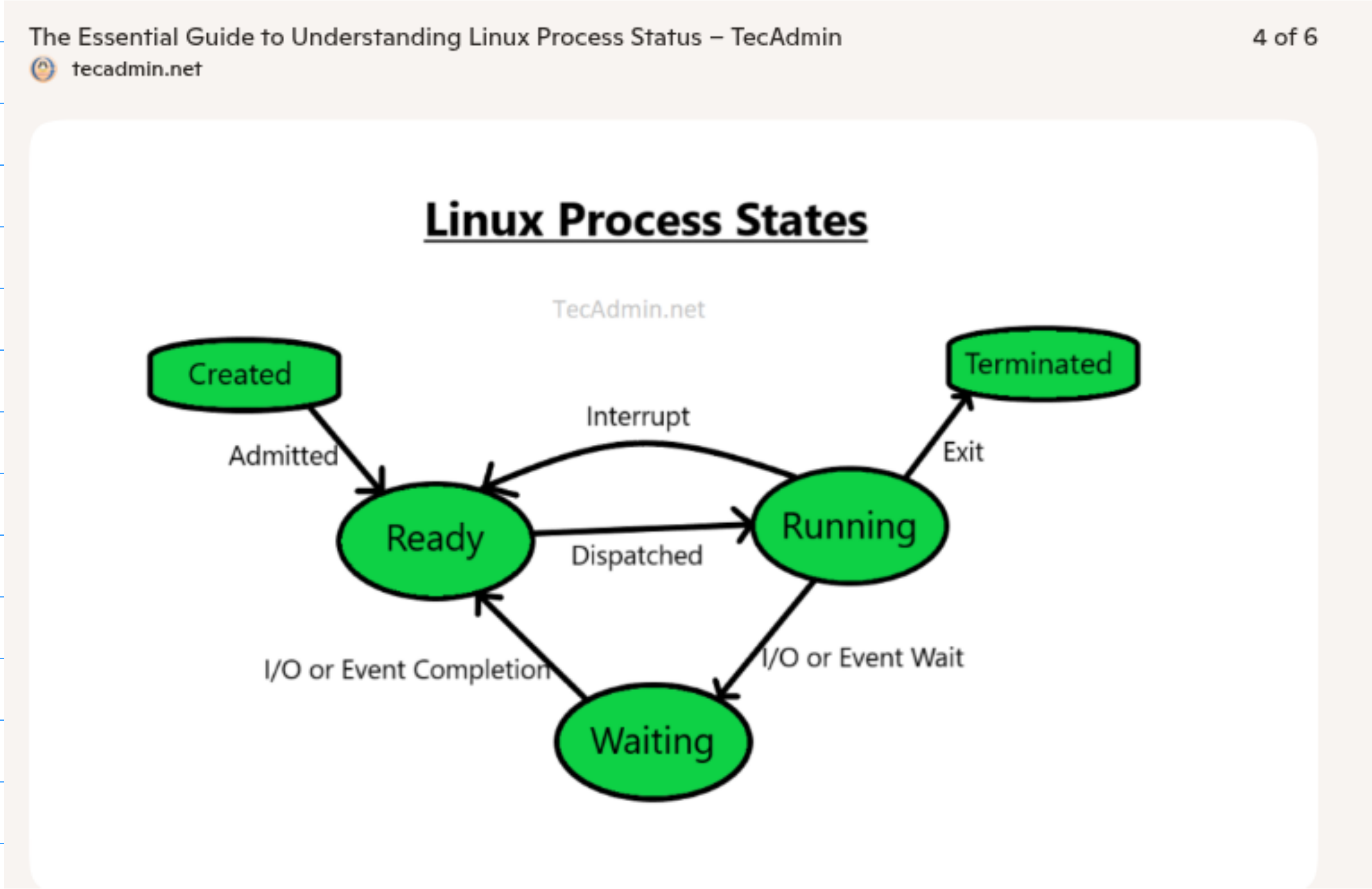
Linux Process States and Signals. When troubleshooting a system, it's... | by A Cloud Chef | Medium 2 of 6

cloudchef.medium.com



What are the process states in Unix/Linux? | by Ram Lakshmanan | Medium 3 of 6

medium.com

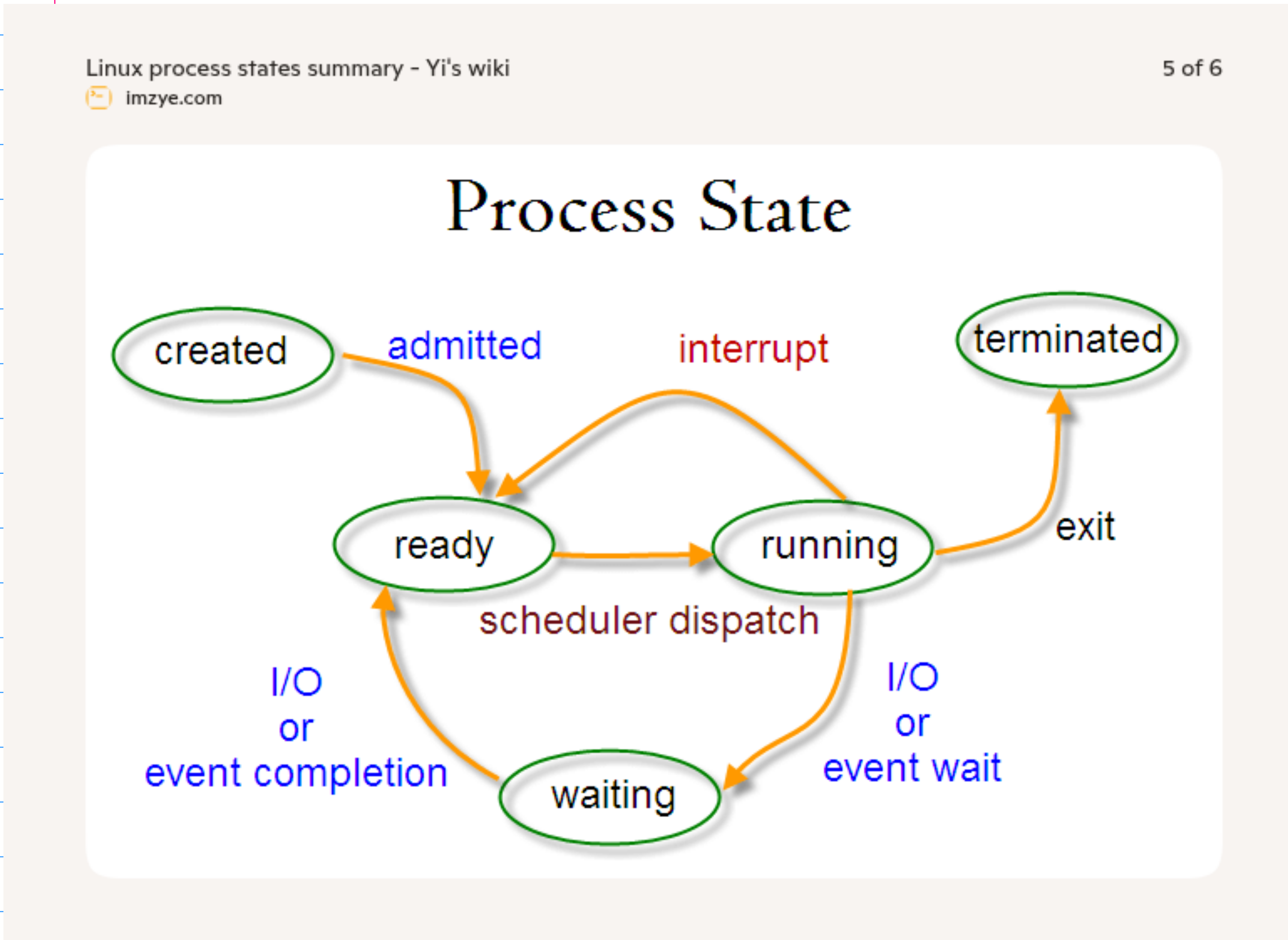


The Essential Guide to Understanding Linux Process Status – TecAdmin

tecadmin.net

4 of 6

Linux Process States

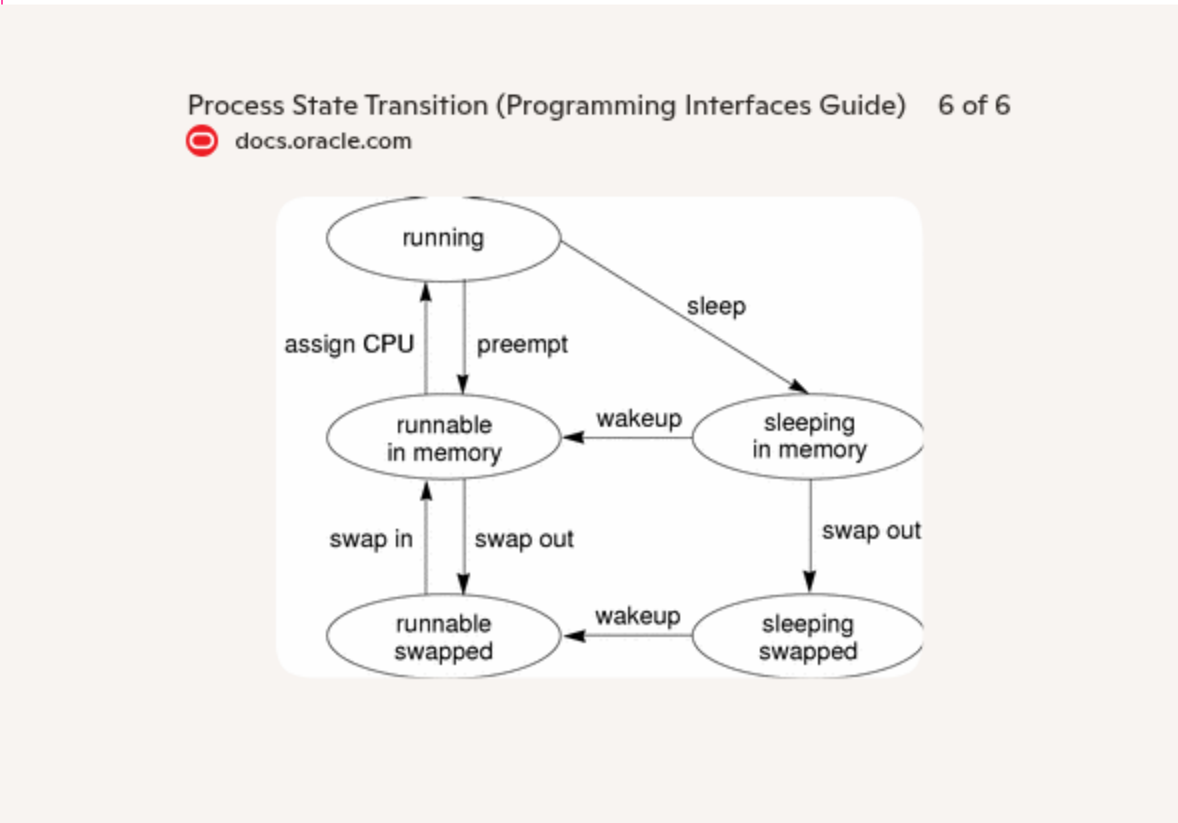


Linux process states summary - Yi's wiki

imzye.com

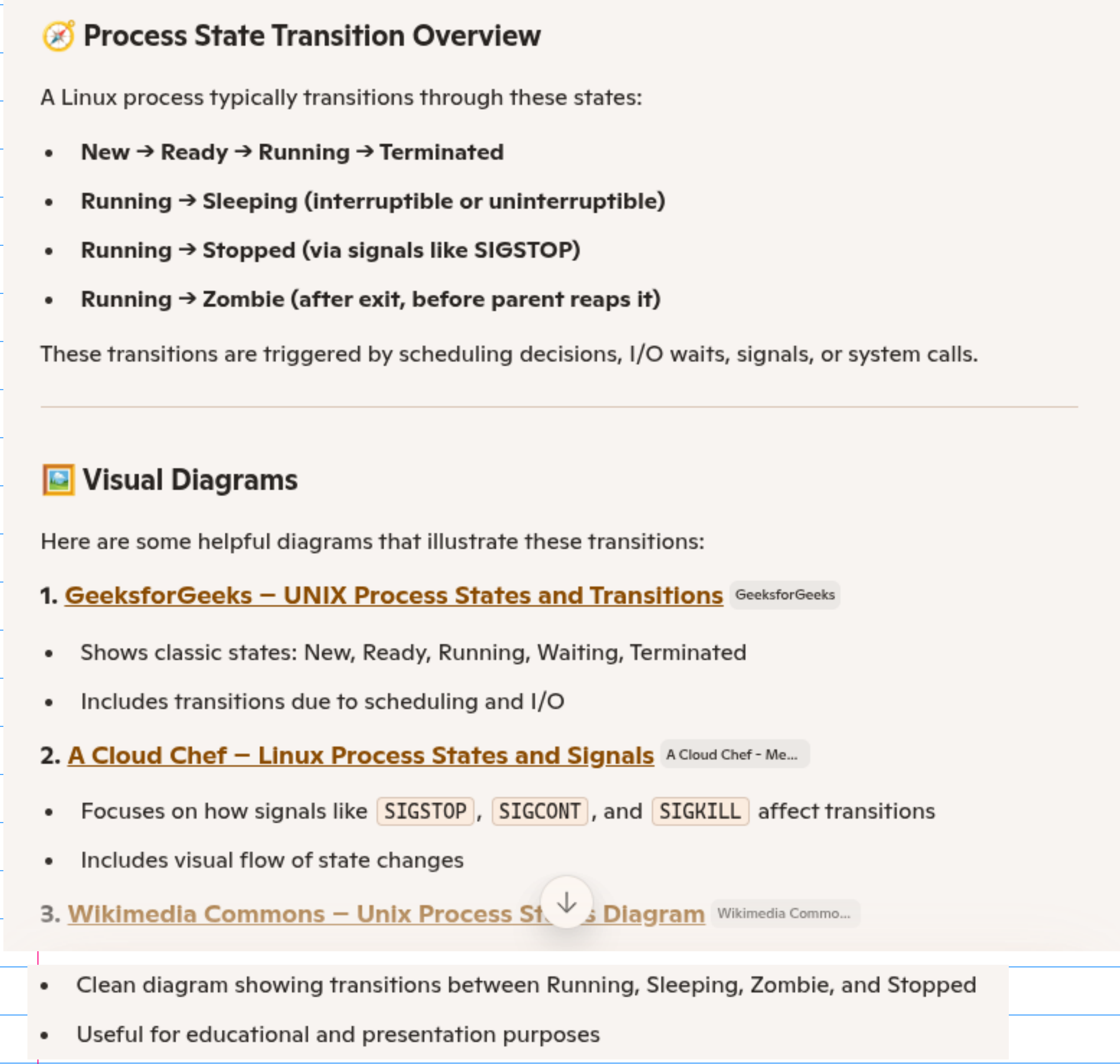
5 of 6

Process State



Process State Transition (Programming Interfaces Guide) 6 of 6

docs.oracle.com



🧭 Process State Transition Overview

A Linux process typically transitions through these states:

- **New → Ready → Running → Terminated**
- **Running → Sleeping (interruptible or uninterruptible)**
- **Running → Stopped (via signals like SIGSTOP)**
- **Running → Zombie (after exit, before parent reaps it)**

These transitions are triggered by scheduling decisions, I/O waits, signals, or system calls.

🖼️ Visual Diagrams

Here are some helpful diagrams that illustrate these transitions:

1. **GeeksforGeeks – UNIX Process States and Transitions** GeeksforGeeks

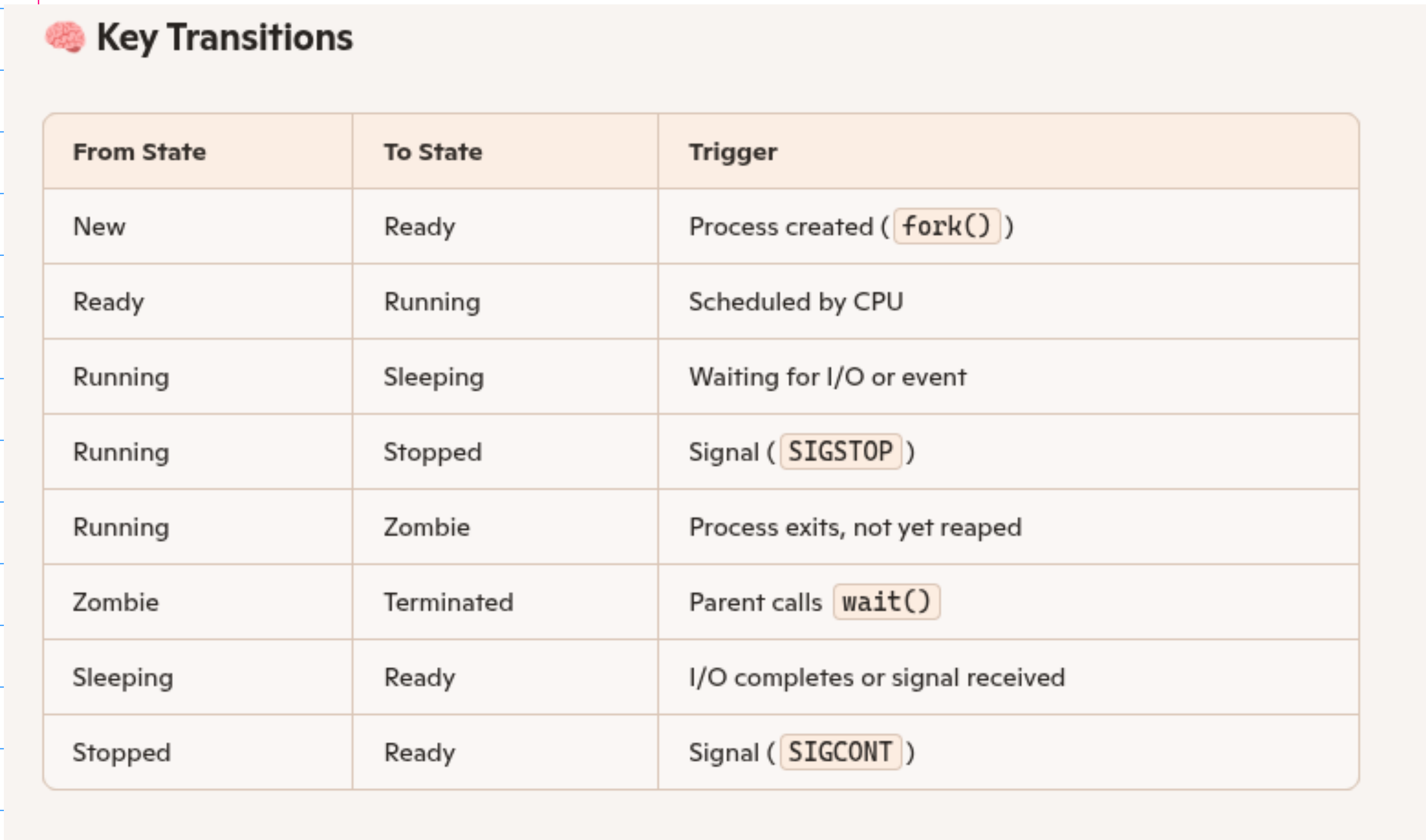
- Shows classic states: New, Ready, Running, Waiting, Terminated
- Includes transitions due to scheduling and I/O

2. **A Cloud Chef – Linux Process States and Signals** A Cloud Chef - Me...

- Focuses on how signals like `SIGSTOP`, `SIGCONT`, and `SIGKILL` affect transitions
- Includes visual flow of state changes

3. **Wikimedia Commons – Unix Process States Diagram** Wikimedia Commo...

- Clean diagram showing transitions between Running, Sleeping, Zombie, and Stopped
- Useful for educational and presentation purposes



🧠 Key Transitions

From State	To State	Trigger
New	Ready	Process created (`fork()`)
Ready	Running	Scheduled by CPU
Running	Sleeping	Waiting for I/O or event
Running	Stopped	Signal (`SIGSTOP`)
Running	Zombie	Process exits, not yet reaped
Zombie	Terminated	Parent calls `wait()`
Sleeping	Ready	I/O completes or signal received
Stopped	Ready	Signal (`SIGCONT`)