How Python Manages Processes, Threads, and Async in Linux

12.12.2025

Agenda

1. Processes and Threads in Linux
2. Syscalls: fork(), execve(), clone()
3. Kernel Scheduling (CFS)
4. Python Processes (multiprocessing)
5. Python Threads and the GIL
6. Asyncio and epoll-based I/O
7. Comparison: Processes vs Threads vs Coroutines

Linux Process Model

A Linux process includes:

- Its own virtual address space
- Code, data, heap, and stacks
- File descriptors
- Signals and metadata (PID, PPID, state)

User space interacts with the kernel through system calls (e.g., fork, execve, clone).

Work with processes in Linux

fork() duplicates the current process:

- Creates a new task_struct in the kernel
- Uses copy-on-write (COW) for memory pages
- Child inherits file descriptors
- Return values:
  - Parent receives child PID
  - Child receives 0

execve() loads a new program into the current process:

- Replaces code, data, heap, and stack
- Keeps the same PID
- Inherits open file descriptors (unless marked close-on-exec)
- Used together with fork() to start new programs

Work with processes in Linux

Property	*fork*()	*execve*()
Creates a new PID	Yes	Completely replaces the process memory with a new program
Memory	Copies the parent's address space using copy-on-write	Inherited (unless FD_CLOEXEC is set)
File descriptors	Inherited	Starts executing a new program from its entrypoint
Execution flow	Continues from the point of fork in both parent and child	No shared memory with the old program
Typical use	Process creation	Does not return on success (the program image is replaced)
Usages by Python	multiprocessing (default on Linux)	*subprocess* (after fork), multiprocesssing with 'spawn'

clone(): Creating Threads in Linux

clone() is a flexible syscall used to create:

- Threads (shared address space)
- Lightweight processes
- Various process namespaces (containers)

Threads share:

- Virtual address space
- File descriptors
- Signal handlers
- clone() is the foundation of POSIX threads (pthread_create) and Python threads

Kernel Scheduling (CFS)

Linux uses the **Completely Fair Scheduler** (CFS) to manage tasks:

- Each process and thread is a separate schedulable task_struct
- Tasks receive CPU time based on fairness, not strict priority
- Context switches occur among all runnable tasks
- The scheduler is unaware of the Python GIL

Python Processes

Python’s multiprocessing module:

- Uses fork() by default on Linux
- Creates fully independent Python interpreters
- Achieves true CPU parallelism (no GIL sharing)
- IPC via pipes, queues, shared memory
- Each process has its own memory space and GIL

Fork Caveats in Python

When Python uses **fork()**:

- The entire interpreter state is duplicated
- Locks and threads are copied → may cause deadlocks
- Open file descriptors are inherited
- Memory is shared via Copy-on-Write until modified
- Global variables are copied but not synchronized
  Use caution with complex global state or active threads.

Python Threads

Python threads are real OS threads:

- Created via threading → pthread_create()
- Kernel creates a new task_struct using clone()
- Threads share memory, file descriptors, and GIL
- True parallelism only for I/O or C extensions that release the GIL
- Visible under */proc/<PID>/task/<TID>*

The Global Interpreter Lock (GIL)

The GIL ensures that:

- Only one thread executes Python bytecode at a time
- Threads still run in parallel at the OS level
- I/O operations release the GIL
- Many C extensions release the GIL to enable true parallelism
- The GIL limits CPU-bound multithreading, but not I/O-bound workloads.

Thread Synchronization in Python

Python provides several synchronization primitives (from threading import …):

- Lock (mutex) – exclusive access to shared data
- RLock – re-entrant lock for recursive locking
- Semaphore / BoundedSemaphore – limit concurrent access
- Event – thread signaling mechanism
- Condition – coordination between producer/consumer threads

When Python Threads Are Useful

Threads are effective for:

- I/O-bound workloads (networking, file I/O, databases)
- Programs that rely on extensions which release the GIL
- Concurrent background tasks (timers, schedulers)
- Lightweight parallelism inside a single process

Threads are **not suitable** for CPU-bound pure Python code.

Quiz

Situation	Use Threads?
Lots of I/O	
Lots of network requests	
C extensions doing heavy work	
Background helpers in a small app	
Pure Python heavy computations	
Need scalable parallelism on multiple CPU	

Quiz

Situation	Use Threads?
Lots of I/O	Yes
Lots of network requests	Yes
C extensions doing heavy work	Yes
Background helpers in a small app	Yes
Pure Python heavy computations	No
Need scalable parallelism on multiple CPU	No — use multiprocessing

Asyncio: User-Space Concurrency

Asyncio provides:

- Event loop managing scheduled tasks
- Coroutines using async / await
- Tasks as lightweight units of work
- Cooperative multitasking in a single thread
- No parallel execution of Python bytecode

Asyncio is ideal for high-concurrency I/O-bound workloads.

Asyncio and the Kernel: epoll

Asyncio relies on:

- epoll (Linux) / kqueue (BSD) / IOCP (Windows)
- Event loop calls epoll_wait() to detect ready file descriptors
- Kernel notifies when sockets/files are ready
- Event loop resumes the corresponding coroutines
- Kernel does not switch between coroutines — Python does

Asyncio = user-level scheduling + kernel-assisted I/O readiness.

Why Async Is Efficient

Asyncio is efficient because:

- No kernel context switching (user-space scheduling)
- No stacks per task → thousands of coroutines are lightweight
- epoll reports ready I/O with minimal overhead
- Cooperative multitasking avoids race conditions
- Perfect for high-concurrency I/O-bound systems

Async = high concurrency, not parallelism.

Processes vs Threads vs Coroutines

Property	Processes	Threads	Coroutines (async)
Scheduler	Kernel (CFS)	Kernel (CFS)	Python event loop
Memory	Separate address space	Shared address space	Shared address space
Switching cost	True parallelism	Limited by GIL	No parallelism
Kernel presence	Always	Always	Only for I/O (epolll)
Ideal for	CPU-bound tasks	I/O-bound or mixed workloads	Massive I/O concurrency

Key Takeaways

- Python relies on Linux syscalls (fork, execve, clone)
- Processes provide real parallelism (separate interpreters, separate GILs)
- Threads are real OS threads but limited by the GIL
- Asyncio provides user-level concurrency, ideal for massive I/O
- Each model serves a different purpose
- Choosing the right model = efficient Python programs