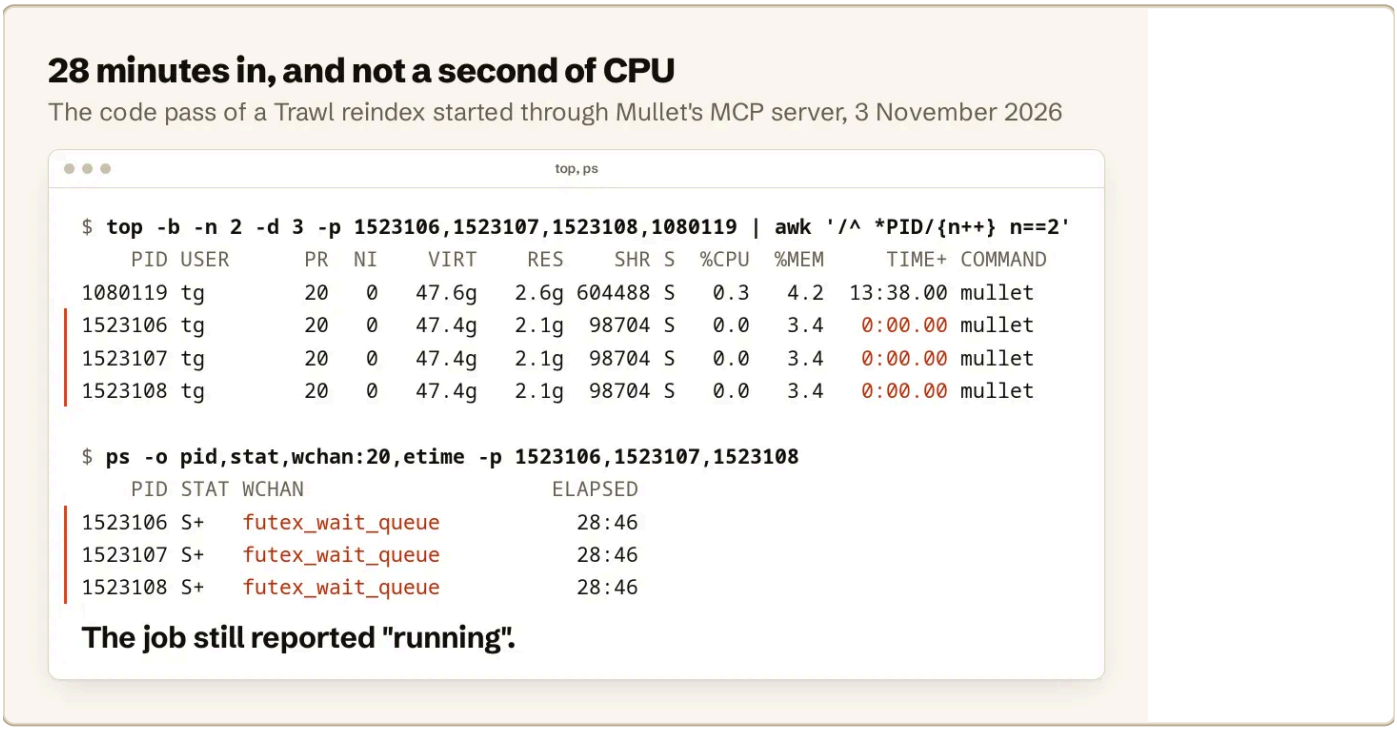
Why a forked worker can wait for ever

A Trawl reindex started through Mullet's MCP server hung at 0% CPU, because its forked workers inherited a lock the main thread was holding. Pools started off the main thread now use spawn, and a test fails instead of hanging.



Published
3 November 2026

Reading time
3 minutes

About
Mullet

Tags
Python, Linux

More on: Mullet

Reindexing a Trawl pool through Mullet's MCP server no longer hangs in its code pass. The cause is a trap worth knowing if you write Python: forking a process that has more than one thread copies every lock in whatever state it's in. If another thread held one at that moment, the child can wait for it for ever. A worker at 0% CPU, asleep in `futex_wait`, is the signature.

What it looked like

On 3 November a reindex started through Mullet's MCP tool finished its document pass normally: 5,515 files and 19,155 chunks in about 14 minutes. Then the code pass started three worker processes. After 28 minutes they still showed 0:00 of CPU time. They were in state S, waiting in `futex_wait_queue`, and the job still said "running".

The same pool reindexed from the command line, with `mullet reindex --pool <pool> --rebuild`, gave 39,666 chunks in 329 seconds with no errors. The pool was fine. What differed was where the workers were started from.

The cause

The MCP server talks to its client over standard input. Its main thread waits in `sys.stdin.readline()` for the next request, and while it waits it holds stdin's lock (`mullet/server/mcp.py`, line 351).

A reindex runs on a job thread, and the worker pools were forked from there. Fork copies the process with that lock still taken, but only the thread that called fork comes along, so in the child nothing will ever release it. Multiprocessing closes stdin in every new worker as it starts, closing waits for the lock, and the worker waits for ever.

Two pools did this: the code chunker's (`mullet/knowledge/code/chunker.py`, line 249) and the one that builds the BM25 keyword index (`mullet/knowledge/engine/search/searcher.py`, line 875). Fix only the first and the reindex hangs in the second instead.

The command line never hit it, because `mullet reindex` forks from its main thread and no thread is holding stdin's lock.

The fix

Pools started off the main thread now use the spawn start method. A spawned worker is a fresh interpreter rather than a copy of its parent, so it inherits no locks at all. The choice is one function in a new module, `mullet/knowledge/process_pools.py`, which both pools pass to `ProcessPoolExecutor` as `mp_context`:

```
def pool_context() -> Optional[BaseContext]:
    if threading.current_thread() is threading.main_thread():
        return None
    return multiprocessing.get_context("spawn")
```

On the main thread it returns `None`, the platform default, so the command line still forks and keeps its quicker start. Spawn has a cost on the server path: each spawned BM25 worker imports torch from scratch.

A job thread that dies now ends its job as "failed", instead of leaving it "running" for ever.

A test that fails instead of hanging

A hang is awkward to test, because a test that hangs never reports. The new test starts a child process laid out like the server: its main thread blocks on standard input, and a reindex of a small temporary pool runs on another thread. The

test waits with a timeout, so if the bug comes back the test fails rather than sitting there.

The fix was merged on 3 November 2026, in pull request #2 of Mullet's development repository (merge commit e6322ee), with the full suite passing locally: 18,667 passed, 42 skipped.

The signature

A worker at 0% CPU, in state S and waiting in `futex_wait`, forked from a process with more than one thread: look for a lock another thread held at the moment of the fork. The remedy is to start that pool with spawn, at least whenever it's started from anything but the main thread.