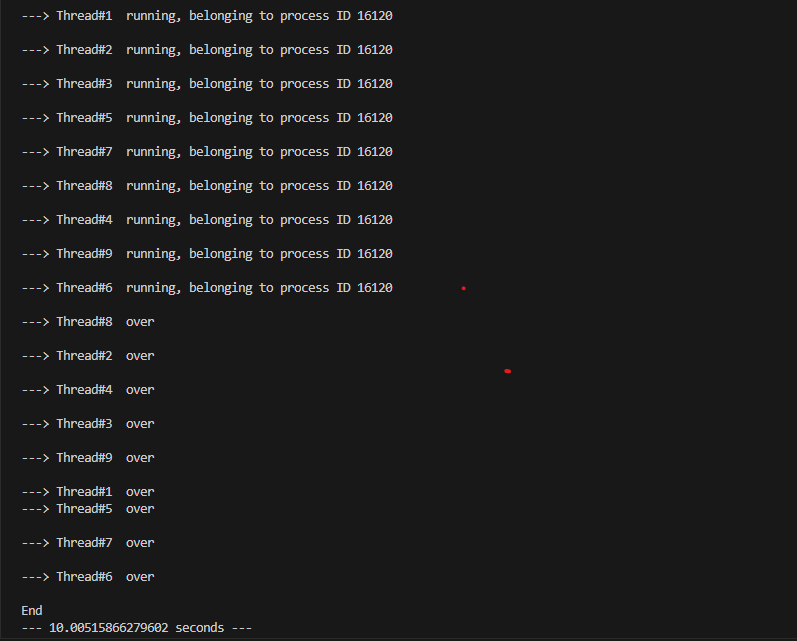
**Threads Synchronization:**

**Code1:**

OUTPUT:



All 9 threads start almost **at the same time**:

Then they finish in **random order**:

· All threads **start concurrently**.

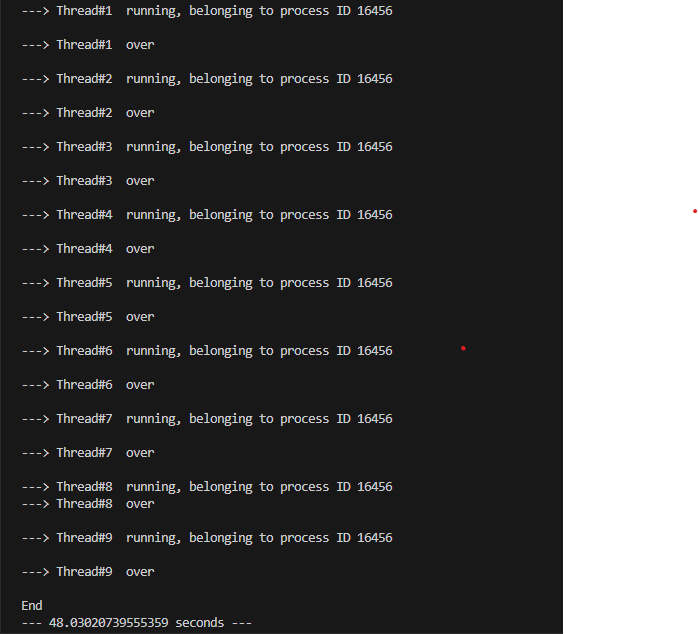
· They **sleep independently**, then finish.

· **True multithreading** (fast, efficient)

· Execution time = **max sleep duration**, not the sum.

2nd code: LOCK

OUTPUT:



Threads **start and finish one by one**:

· Only **one thread runs at a time**

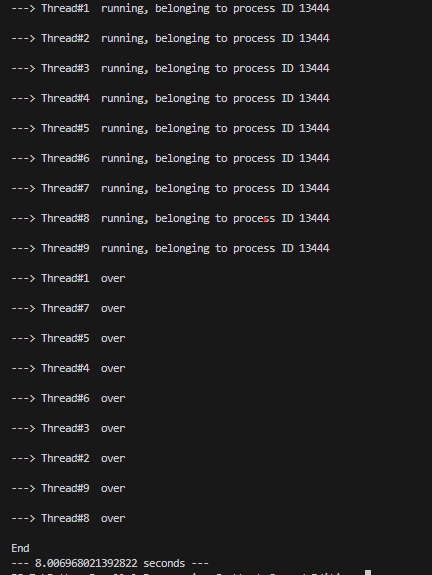
· Others **wait until the previous is done**

· Threads are **not parallel** — run **sequentially**

· Likely a Lock is blocking the entire run() method (including sleep())

3rd code LOCK

OUTPUT:



All 9 threads **start running almost instantly**:

· You **only lock the print part** — not the whole thread.

· time.sleep() is **outside** the lock, so it doesn’t block other threads.

· This allows all threads to do their work **in parallel**.

**Conclusion of the 3 Multithreading Codes:**

1. **Code 1 (MyThreadClass\_lock.py)** uses a lock around the entire thread, causing threads to run **one by one**, resulting in **sequential execution** and **very slow performance (~48s)**.
2. **Code 2 (MyThreadClass\_lock\_2.py)** uses the lock **only for print statements**, allowing threads to **run in parallel**, giving **much faster execution (~8s)**.
3. **Code 3 (first one without any lock)** also runs threads in parallel but may produce **mixed/unreadable print output** if threads print at the same time.
4. The key difference is **how and where the Lock is applied** — locking too much blocks parallelism, while locking smartly gives speed and clean output.
5. For efficient multithreading, always **minimize lock usage** and never include long tasks (like sleep) inside a lock.