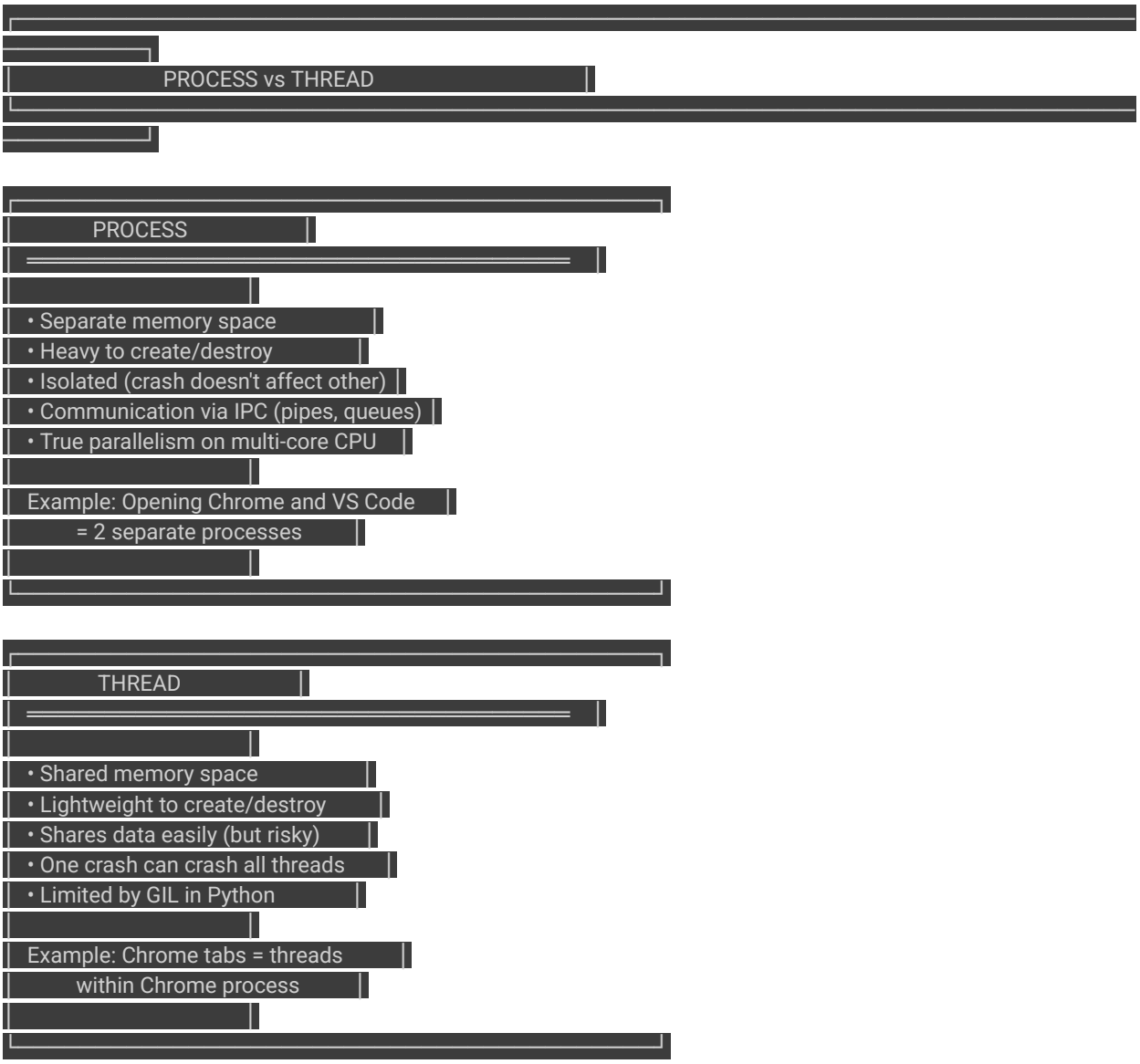
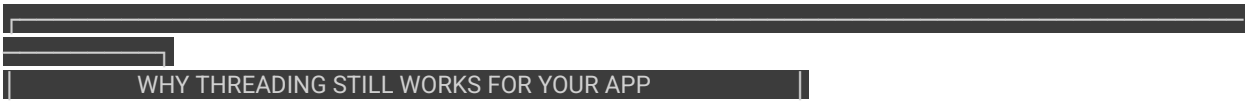
Threading, Multi-threading & Multiprocessing - Complete Guide

Part 1: Core Concepts



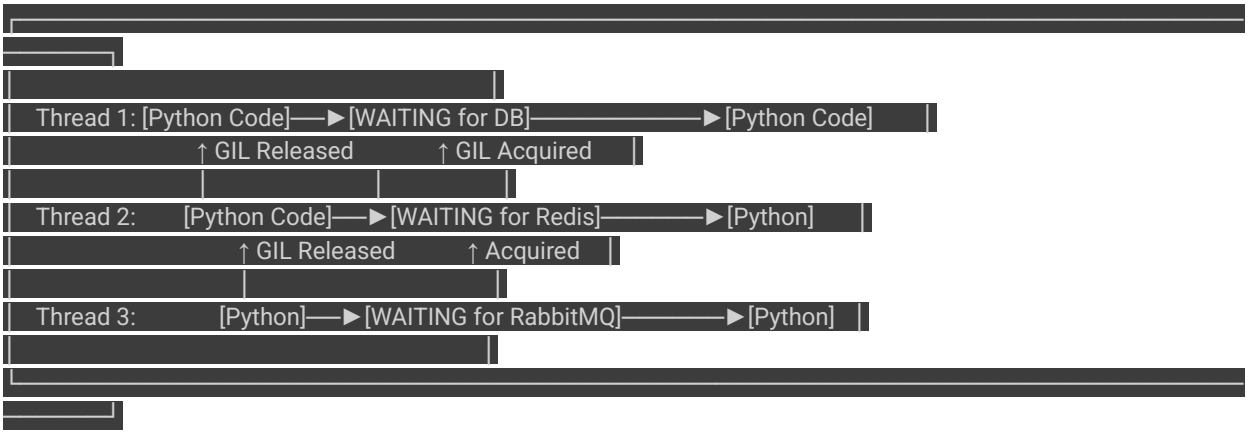
Part 2: Visual Comparison

Part 3: Python's GIL (Global Interpreter Lock)



Python has GIL = Only ONE thread executes Python code at a time

BUT... GIL is RELEASED during I/O operations!



YOUR APP IS I/O BOUND:
• Waiting for Grandstream webhooks (Network I/O)
• Waiting for PostgreSQL queries (Database I/O)
• Waiting for Redis operations (Network I/O)
• Waiting for RabbitMQ messages (Network I/O)
• Waiting for WebSocket events (Network I/O)

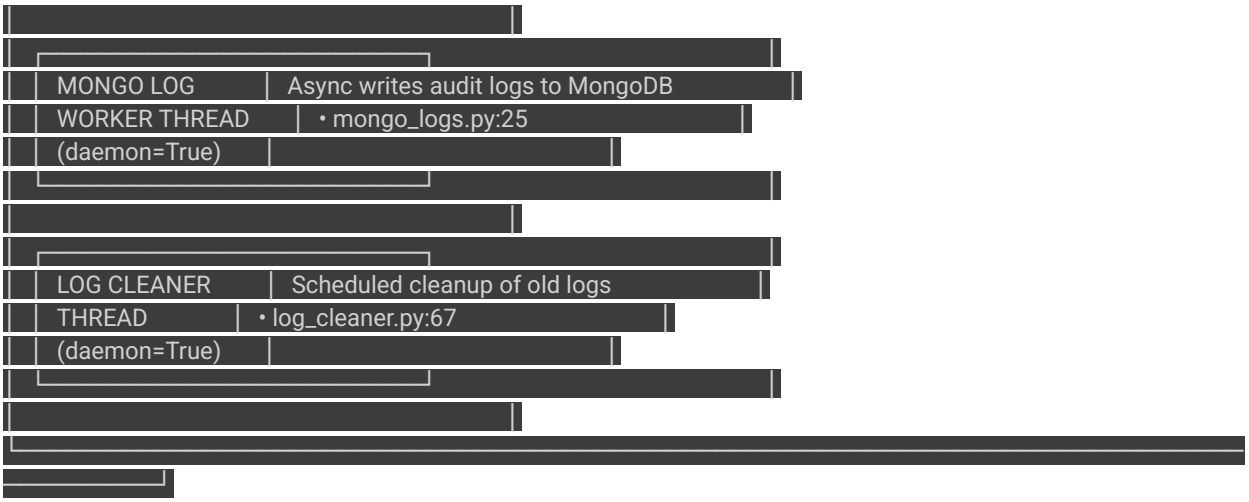
⭐ Threading is PERFECT for your use case!

Part 4: Threading in YOUR Codebase

Let me find where threading is used in your application:
Grep `"import threading|from threading|Thread\(|threading\."`32 lines of output
Grep `"start_grandstream_consumer|run_in_background|background.*thread"`13 lines of output
Read `d:\NextGenCC\NextGen-FastAPI\messaging\consumer_manager.py`
Now let me show you exactly how threading works in your codebase:

Part 5: Threading in YOUR Codebase (Real Examples)

When your FastAPI app starts, multiple threads are created:



Part 6: Code Examples from Your Codebase

Example 1: Grandstream Consumer Thread

File: consumer_manager.py

```
import threading

def start_consumers():
    # Create a new thread that runs start_grandstream_consumer function
    grandstreamThread = threading.Thread(
        target=start_grandstream_consumer,  # Function to run
        daemon=True                         # Dies when main thread dies
    )
    grandstreamThread.start()  # Start the thread
    threads.append(grandstreamThread)
```

Example 2: Thread-Local Storage for RabbitMQ

File: rabbitmq.py

```
import threading

# Each thread gets its OWN connection (not shared)
_thread_local = threading.local()

def get_thread_local_connection():
    # Check if THIS thread already has a connection
    if not hasattr(_thread_local, 'connection') or _thread_local.connection is None:
        # Create connection ONLY for this thread
        _thread_local.connection = create_connection()
    return _thread_local.connection
```

Example 3: Thread Lock for Safe Access

File: rabbitmq.py

```
import threading

# Lock prevents multiple threads from modifying connection simultaneously
_connection_lock = threading.RLock()

def get_or_create_connection():
    with _connection_lock:  # Only ONE thread can enter this block
        if _connection is None:
            _connection = create_connection()
        return _connection
```

Example 4: Timer Thread for Missed Calls

File: grandstream_consumer.py

```
import threading

# Timer = Thread that waits X seconds then executes function
missed_call_timer = threading.Timer(
    delay_seconds + 0.5,          # Wait 5.5 seconds
    process_missed_queue_timer    # Then call this function
)
missed_call_timer.start()
```

Part 7: Key Threading Concepts Used

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                    THREADING CONCEPTS IN YOUR APP                            │
└──────────────────────────────────────────────────────────────────────────────┘

1️⃣ DAEMON THREADS (daemon=True)
   ═══════════════════════════════

   threading.Thread(target=func, daemon=True)

   • Automatically killed when main program exits
   • Used for: background tasks that shouldn't block shutdown
   • Your app: ALL background threads are daemons

   If daemon=False:
   • Main program waits for thread to finish before exiting
```

• Can cause app to hang on shutdown

2️⃣ THREAD-LOCAL STORAGE (threading.local())
═══

```
_thread_local = threading.local()
_thread_local.connection = create_connection()
```

• Each thread gets its OWN copy of the variable
• Thread A's connection ≠ Thread B's connection
• Your app: Each thread has its own RabbitMQ connection

WHY? RabbitMQ connections are NOT thread-safe!

3️⃣ LOCKS (threading.Lock() / threading.RLock())
═══

```
_lock = threading.RLock()

with _lock:
    # Only ONE thread can be here at a time
    shared_resource.modify()
```

• Prevents race conditions
• RLock = Reentrant Lock (same thread can acquire multiple times)
• Your app: Protects shared RabbitMQ connection

4️⃣ EVENTS (threading.Event())
═══════════════════════════

```
_stop_event = threading.Event()

# In main thread:
_stop_event.set()  # Signal to stop

# In worker thread:
while not _stop_event.is_set():
    do_work()
```

• Used for signaling between threads
• Your app: availability_cleanup.py uses this for graceful shutdown

5️⃣ TIMERS (threading.Timer())
═══════════════════════════

```
timer = threading.Timer(5.0, my_function)
timer.start()  # Calls my_function after 5 seconds
```

• Delayed execution in background
• Your app: Delayed processing of missed calls (5s delay)

Part 8: Learning Path / Roadmap

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                    THREADING/MULTIPROCESSING LEARNING ROADMAP                │
└─────────────────────────────────────────────────────────────────────────────┘

LEVEL 1: BASICS (Start Here)
═══════════════════════════
□ Understand Process vs Thread
□ Create simple threads with threading.Thread()
□ Understand daemon vs non-daemon threads
□ Learn thread.start() and thread.join()

  Practice:
  ─────────
  import threading
  import time

  def worker(name):
      print(f"{name} started")
      time.sleep(2)
      print(f"{name} finished")

  # Create and start threads
  t1 = threading.Thread(target=worker, args=("Thread-1",))
  t2 = threading.Thread(target=worker, args=("Thread-2",))

  t1.start()
  t2.start()

  t1.join()  # Wait for t1 to finish
  t2.join()  # Wait for t2 to finish
  print("All done!")


LEVEL 2: SYNCHRONIZATION
═══════════════════════
□ Learn about race conditions
□ Use threading.Lock() to protect shared data
□ Understand deadlocks and how to avoid them
□ Learn threading.RLock() (reentrant lock)

  Practice:
```

```
    counter = 0
    lock = threading.Lock()

    def increment():
        global counter
        with lock:  # Only one thread at a time
            temp = counter
            time.sleep(0.001)  # Simulate work
            counter = temp + 1

    threads = [threading.Thread(target=increment) for _ in range(100)]
    for t in threads: t.start()
    for t in threads: t.join()
    print(f"Counter: {counter}")  # Should be 100
```

LEVEL 3: ADVANCED THREADING
═══════════════════════════
□ threading.Event() for signaling
□ threading.Condition() for complex synchronization
□ threading.Semaphore() for limiting concurrent access
□ threading.local() for thread-local storage
□ threading.Timer() for delayed execution

```
    Practice:
    ─────────
    # Thread-local storage example
    local_data = threading.local()

    def worker(value):
        local_data.x = value  # Each thread has its own 'x'
        time.sleep(1)
        print(f"Thread {value}: local_data.x = {local_data.x}")

    threads = [threading.Thread(target=worker, args=(i,)) for i in range(3)]
    for t in threads: t.start()
    for t in threads: t.join()
```

LEVEL 4: MULTIPROCESSING
════════════════════════
□ Understand when to use multiprocessing vs threading
□ Learn multiprocessing.Process()
□ Use multiprocessing.Queue() for communication
□ Use multiprocessing.Pool() for parallel tasks
□ Understand shared memory (Value, Array)

```
    Practice:
    ─────────
    from multiprocessing import Process, Queue
```

```
    def worker(q, name):
        q.put(f"Hello from {name}")

    q = Queue()
    processes = [Process(target=worker, args=(q, f"Process-{i}")) for i in range(3)]

    for p in processes: p.start()
    for p in processes: p.join()

    while not q.empty():
        print(q.get())


LEVEL 5: ASYNC/AWAIT (Modern Python)
═══════════════════════════════════════
□ Understand asyncio event loop
□ Learn async def and await keywords
□ Use asyncio.gather() for concurrent tasks
□ Understand how FastAPI uses async

    Practice:
    ─────────
    import asyncio

    async def fetch_data(name, delay):
        print(f"{name} starting")
        await asyncio.sleep(delay)  # Non-blocking wait
        print(f"{name} done")
        return f"Result from {name}"

    async def main():
        # Run 3 tasks concurrently
        results = await asyncio.gather(
            fetch_data("Task-1", 2),
            fetch_data("Task-2", 1),
            fetch_data("Task-3", 3)
        )
        print(results)

    asyncio.run(main())


LEVEL 6: PRODUCTION PATTERNS
═══════════════════════════════
□ Thread pools (concurrent.futures.ThreadPoolExecutor)
□ Process pools (concurrent.futures.ProcessPoolExecutor)
□ Producer-Consumer pattern (like your RabbitMQ setup!)
□ Background task queues (Celery, RQ)
□ Graceful shutdown handling
```

Part 9: When to Use What?