**iagram: How Multiprocessing Would Work**

When using **multiprocessing**, each process **has its own memory.** So you need **special channels** to share data.

A screenshot of a computer

AI-generated content may be incorrect.

In this case:

* Processes **cannot see each other’s memory by default.**
* You **must use Manager() to share state safely.**
* Queue() (from multiprocessing) is used to **pass data between processes.**

**When to Use Each:**

| **Threading** | **Multiprocessing** |
| --- | --- |
| ✅ Shared memory is natural. | 🚧 You need Manager.dict(), Queue(). |
| ✅ Lightweight. | 🚀 Heavy-duty parallelism (CPU-bound work). |
| ✅ Best for I/O tasks (waiting, networking). | ✅ Best for CPU-heavy tasks. |
| ⚠️ Global Interpreter Lock (GIL) limits true parallelism. | ✔️ True parallel execution of Python code. |

**Your Current Case:**

* ✅ **Temperature monitoring** is an I/O-like task (periodic sensor reads, not huge CPU work).
* ✅ **Threading** works great.
* ⚠️ You **don't need Manager.dict() unless you move to multiprocessing.**

**Key Changes:**

* 🛠 **Change from Thread to Process:**  
  We'll use multiprocessing.Process instead of threading.Thread.
* 🛠 **Manager for Shared Data:**  
  We’ll use Manager().dict() for shared\_state and Manager().Queue() for temp\_queue.
* 🛠 **Lock:**  
  We’ll use a **multiprocessing lock** instead of the threading lock.

//==============================================================================

**What’s Different from Threading:**

| **Threading Version** | **Multiprocessing Version** |
| --- | --- |
| ChannelMonitor(Thread) | ChannelMonitor(Process) |
| queue.Queue() | Manager().Queue() |
| self.shared\_state = {} | self.shared\_state = Manager().dict() |
| threading.Lock() | multiprocessing.Lock() |
| Stop: monitor.stop() | Stop: monitor.terminate() (kills the process directly) |
| Automatic shared memory | Must **explicitly** use Manager() for shared memory |

**⚠️ Important Note:**

* In **threading**, your ChannelMonitor class could have a self.running flag to cleanly stop the thread.
* In **multiprocessing**, it's harder to **signal gracefully** across processes (flags don’t work easily because of isolated memory).  
  ✅ So here I used monitor.terminate(), which **forcefully stops the process.**

For a **clean shutdown,** you'd need to implement an **inter-process event** or **special signaling.**

✅ Now You Can:

See how threading and multiprocessing differ.

Switch between the two based on your needs (CPU-bound vs. I/O-bound).

Do you want to enhance it further (e.g., graceful shutdown for processes), or explore how to handle errors or recovery between processes? 😊

//=============================================================================

**Goal: Make a CLEAN Shutdown for Multiprocessing**

Right now, we **force-stop** with terminate(). This is fine for some cases, but:

* ❌ It's abrupt.
* ❌ It can **leave resources uncleaned.**

✅ Instead, let's **signal each process to stop gracefully** (similar to how we used self.running = False in threading).

**💡 How to Do It:**

We’ll use a **multiprocessing.Event**. This is a **shared flag** that can signal **across processes** (unlike a regular variable).

**✅ Steps:**

1. **Pass an Event to each process.**
2. Each process **watches the event** inside its loop.
3. The main process **sets the event** when it wants all processes to stop.

Code:

from multiprocessing import Process, Manager, Lock, Event

import random

import time

class ChannelMonitor(Process):

def \_\_init\_\_(self, channel\_id, temp\_queue, shared\_state, lock, stop\_event, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.channel\_id = channel\_id

self.temp\_queue = temp\_queue

self.shared\_state = shared\_state

self.lock = lock

self.stop\_event = stop\_event # <-- NEW: shared stop signal

def run(self):

while not self.stop\_event.is\_set():

# Simulate reading a temperature value

temp = round(20 + random.uniform(-5, 5), 2)

timestamp = time.time()

# Update current temperature in shared state

with self.lock:

self.shared\_state[f"channel\_{self.channel\_id}\_current"] = temp

# Get the setpoint if available

setpoint\_key = f"channel\_{self.channel\_id}\_setpoint"

setpoint = self.shared\_state.get(setpoint\_key)

# Print current status

print(f"[Channel {self.channel\_id}] Temp: {temp}°C at {timestamp}")

# Check against setpoint

if setpoint is not None:

diff = abs(temp - setpoint)

if diff > 2:

print(f"[Channel {self.channel\_id}] ⚠️ Deviation from setpoint ({setpoint}°C) by {diff:.2f}°C.")

# Push the data to the queue

self.temp\_queue.put({

'channel': self.channel\_id,

'temperature': temp,

'timestamp': timestamp

})

time.sleep(2)

print(f"[Channel {self.channel\_id}] Stopped gracefully.")

class TCUMasterProcess:

def \_\_init\_\_(self, num\_channels=4):

self.num\_channels = num\_channels

self.manager = Manager()

self.temp\_queue = self.manager.Queue()

self.shared\_state = self.manager.dict()

self.lock = Lock()

self.stop\_event = Event() # <-- NEW: Event to control stopping

self.monitors = []

def start\_monitoring(self):

print("Starting TCU Monitoring (Multiprocessing)...")

for channel\_id in range(1, self.num\_channels + 1):

monitor = ChannelMonitor(

channel\_id,

self.temp\_queue,

self.shared\_state,

self.lock,

self.stop\_event # Pass the Event

)

monitor.start()

self.monitors.append(monitor)

def stop\_monitoring(self):

print("Stopping TCU Monitoring (gracefully)...")

self.stop\_event.set() # Signal all processes to stop

for monitor in self.monitors:

monitor.join() # Wait for each to finish

def process\_temperature\_data(self):

while not self.temp\_queue.empty():

data = self.temp\_queue.get()

print(f"[Processor] Received data: {data}")

def set\_temperature\_point(self, channel\_id, setpoint):

with self.lock:

self.shared\_state[f"channel\_{channel\_id}\_setpoint"] = setpoint

print(f"[Master] Setpoint for Channel {channel\_id} set to {setpoint}°C.")

def main():

tcu\_master = TCUMasterProcess()

tcu\_master.start\_monitoring()

try:

# Set some setpoints for channels

tcu\_master.set\_temperature\_point(1, 22.5)

tcu\_master.set\_temperature\_point(2, 25.0)

tcu\_master.set\_temperature\_point(3, 35.0)

# Let it run for 12 seconds

time.sleep(12)

except KeyboardInterrupt:

print("\n[Master] KeyboardInterrupt detected! Stopping monitoring...")

finally:

tcu\_master.stop\_monitoring()

tcu\_master.process\_temperature\_data()

if \_\_name\_\_ == "\_\_main\_\_":

from multiprocessing import freeze\_support

freeze\_support() # Needed on Windows

main()

//===============================================================================

**🔍 What's New:**

| **Old** | **New** |
| --- | --- |
| Used terminate() to kill processes | Use stop\_event to **signal them to exit gracefully** |
| No feedback when stopping | Each process prints: [Channel X] Stopped gracefully. |
| ❌ Abrupt stop | ✅ Clean shutdown (no risk of corruption/incomplete work) |

**✅ Benefits:**

* Processes **finish their loop** before stopping.
* You avoid possible **resource leaks** (queues, files, etc.).
* You can catch KeyboardInterrupt and **stop gracefully.**

**🔑 Key:**

* **stop\_event:** The *shared signal* all processes watch to know when to exit.
* **Lock:** Ensures that shared\_state updates are **synchronized**.
* **temp\_queue:** Passes temperature data **from processes to main thread**.
* **ChannelMonitor:** Each channel is a **separate process** that loops until stop.

//===============================================================================

┌────────────────────────────────────────────────────────┐

│ TCUMasterProcess │

│ ┌────────────────────────────────────────────────────┐ │

│ │ Shared Resources: │ │

│ │ - Manager().dict() [shared\_state] │ │

│ │ - Manager().Queue() [temp\_queue] │ │

│ │ - multiprocessing.Lock() │ │

│ │ - multiprocessing.Event() [stop\_event] │ │

│ └────────────────────────────────────────────────────┘ │

│ │

│ ┌─────────────────────┬─────────────────────┬─────────┐│

│ │ ChannelMonitor 1 │ ChannelMonitor 2 │ ... ││

│ │ (Process) │ (Process) │ ││

│ │ │ │ ││

│ │ Loop: │ Loop: │ ││

│ │ - Read temp │ - Read temp │ ││

│ │ - Lock & update │ - Lock & update │ ││

│ │ - Check setpoint │ - Check setpoint │ ││

│ │ - Queue data │ - Queue data │ ││

│ │ - Wait 2s │ - Wait 2s │ ││

│ │ - Check stop\_event ─┴───────────────┬─────┘ ││

│ │ (exit if set) │ ││

│ └─────────────────────────────────────┴───────────────┘│

│ │

│ [Main Thread] │

│ - Sets temperature points │

│ - Sleeps or runs │

│ - On shutdown: │

│ • stop\_event.set() ───▶ tells all processes to stop │

│ • join() all ChannelMonitors │

│ • process remaining data from temp\_queue │

|  |
| --- |
|  |

//=============================================================================