**Creating Responsive Applications with Multithreading in PySide6-PyQt6**

**Creating Responsive Applications with Multithreading in PySide6/PyQt6**

When developing **GUI applications**, it’s essential to keep the **UI responsive** even while performing long-running tasks (e.g., file operations, API calls, data processing). Using **multithreading** properly ensures that the UI does not freeze.

1. **Why Use Multithreading?**

✅ **Prevents UI from freezing** during heavy tasks.  
✅ **Allows parallel execution** of background tasks.  
✅ **Improves user experience** by keeping interactions smooth.

In **PySide6/PyQt6**, the main thread (**GUI thread**) **should not** run blocking tasks. Instead, we use **QThread** or **QThreadPool** to handle background tasks.

**2. Using QThread to Keep UI Responsive**

A simple example where a long-running **task runs in a separate thread** while keeping the UI responsive.

**Example: Running a Background Task with QThread**

from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel

from PySide6.QtCore import QThread, Signal

import time

import sys

class WorkerThread(QThread):

"""Worker thread that performs a long-running task."""

progress = Signal(int) # Custom signal to update UI

def run(self):

"""Long-running task (runs in background)."""

for i in range(5):

time.sleep(1) # Simulate work

self.progress.emit(i + 1) # Send progress update

class MyWindow(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Responsive GUI with QThread")

self.resize(300, 200)

layout = QVBoxLayout()

self.label = QLabel("Click the button to start the task")

layout.addWidget(self.label)

self.button = QPushButton("Start Task")

self.button.clicked.connect(self.start\_thread)

layout.addWidget(self.button)

self.setLayout(layout)

def start\_thread(self):

"""Start the worker thread."""

self.worker = WorkerThread()

self.worker.progress.connect(self.update\_label) # Connect signal to update UI

self.worker.start()

def update\_label(self, value):

"""Update UI safely from worker thread."""

self.label.setText(f"Progress: {value}/5")

app = QApplication(sys.argv)

window = MyWindow()

window.show()

sys.exit(app.exec())

**Why is the UI Responsive?**

* The **worker thread runs separately** from the main GUI thread.
* UI updates are done through **signals (progress.emit(value))**, preventing direct modifications from a background thread.

**3. Using QThread with Worker Object for Better Control**



A **better approach** is to use QObject inside QThread, making it more reusable.



**Example: Running Multiple Workers Without Freezing UI**

from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel

from PySide6.QtCore import QObject, QThread, Signal

import time

import sys

class Worker(QObject):

"""Worker object that runs in a separate thread."""

progress = Signal(int)



finished = Signal()



def do\_work(self):



"""Background task that runs in the worker thread."""

for i in range(5):

time.sleep(1) # Simulate work



self.progress.emit(i + 1)



self.finished.emit()



class MyWindow(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Responsive GUI with Worker QThread")

self.resize(300, 200)

layout = QVBoxLayout()

self.label = QLabel("Click button to start task")



layout.addWidget(self.label)



self.button = QPushButton("Start Task")

self.button.clicked.connect(self.start\_worker)



layout.addWidget(self.button)

self.setLayout(layout)

def start\_worker(self):



"""Start the worker thread."""

self.thread = QThread()



self.worker = Worker()



self.worker.moveToThread(self.thread)



self.thread.started.connect(self.worker.do\_work)

self.worker.progress.connect(self.update\_label)



self.worker.finished.connect(self.thread.quit)



self.worker.finished.connect(self.worker.deleteLater)



self.thread.finished.connect(self.thread.deleteLater)

self.thread.start()

def update\_label(self, value):

"""Update UI safely from worker thread."""

self.label.setText(f"Progress: {value}/5")

app = QApplication(sys.argv)

window = MyWindow()

window.show()

sys.exit(app.exec())

**Why is this Better?**

✅ The **worker logic is separate** from QThread (better design).  
✅ We can **reuse Worker** for multiple tasks.  
✅ Ensures **safe cleanup** of threads using deleteLater().

**4. Using QThreadPool for Efficient Task Execution**

Instead of creating new QThread instances manually, we can use **QThreadPool** with QRunnable.

**Example: Running Multiple Tasks Efficiently**

from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel, QHBoxLayout  
from PySide6.QtCore import QRunnable, QThreadPool, Signal, QObject  
import time  
import sys  
  
class WorkerSignals(QObject):  
 *"""Defines custom signals for the worker."""* progress = Signal(str, int) *# Signal to send progress messages (text, worker ID)* finished = Signal(int) *# Signal to notify when task is complete (worker ID)*class Worker(QRunnable):  
 *"""Worker task that runs in a thread pool."""* def \_\_init\_\_(self, worker\_id):  
 super().\_\_init\_\_()  
 self.worker\_id = worker\_id  
 self.signals = WorkerSignals()  
  
 def run(self):  
 *"""Runs the task in a separate thread."""* for i in range(50):  
 time.sleep(0.5)  
 self.signals.progress.emit(f"Task {self.worker\_id} progress: {i+1}/50", self.worker\_id)  
 self.signals.finished.emit(self.worker\_id)  
  
class MyWindow(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setWindowTitle("QThreadPool Example")  
 self.resize(400, 300)  
  
 self.layout = QVBoxLayout()  
  
 self.label\_main = QLabel("Click to start workers")  
 self.layout.addWidget(self.label\_main)  
  
 self.button = QPushButton("Start 3 Tasks")  
 self.button.clicked.connect(self.start\_workers)  
 self.layout.addWidget(self.button)  
  
 *# Separate labels for each worker* self.worker\_labels = {}  
 for i in range(3):  
 label = QLabel(f"Task {i+1}: Waiting...")  
 self.layout.addWidget(label)  
 self.worker\_labels[i + 1] = label *# Store labels in a dictionary* self.setLayout(self.layout)  
  
 self.threadpool = QThreadPool()  
  
 def start\_workers(self):  
 *"""Starts multiple worker tasks in the thread pool."""* for i in range(3): *# Start 3 workers* worker = Worker(i + 1)  
 worker.signals.progress.connect(self.update\_label)  
 worker.signals.finished.connect(self.task\_finished)  
 self.threadpool.start(worker)  
  
 def update\_label(self, msg, worker\_id):  
 *"""Updates label with worker progress."""* self.worker\_labels[worker\_id].setText(msg)  
  
 def task\_finished(self, worker\_id):  
 *"""Marks task as completed."""* self.worker\_labels[worker\_id].setText(f"Task {worker\_id} completed!")  
  
app = QApplication(sys.argv)  
window = MyWindow()  
window.show()  
sys.exit(app.exec())

**Why Use QThreadPool?**

✅ **Efficient management** of worker threads.  
✅ **Reuses threads**, reducing overhead.  
✅ No need to manually create and delete QThread instances.

**5. Best Practices for Multithreading in PySide6/PyQt**

✔ **Never update the UI directly** from a worker thread (use signals).  
✔ Use deleteLater() to **avoid memory leaks**.  
✔ **Prefer QThreadPool** for lightweight tasks.  
✔ **Separate worker logic** from QThread to make it reusable.  
✔ Handle **thread termination safely** before closing the app.

**Conclusion**

| **Approach** | **Best For** |
| --- | --- |
| **QThread (Subclassing)** | Simple background tasks |
| **QThread + Worker (QObject)** | More reusable and manageable workers |
| **QThreadPool + QRunnable** | Lightweight, repeated tasks efficiently |

For large applications, **QThreadPool is preferred** as it manages threads efficiently without manual creation.

**Pausing, Resuming, and Cancelling Threads in PySide6/PyQt6**

In real-world applications, you may need to **pause, resume, or cancel** background tasks to improve user control and responsiveness. This is especially useful for **long-running operations** such as file downloads, data processing, or network requests.

**1. Implementing Pause, Resume, and Cancel in QThread**

We will use:

* **QThread** for running the task.
* **Flags (self.\_is\_paused, self.\_is\_running)** to control execution.
* **Signals** to communicate with the UI.

**Example: A Worker Thread with Pause, Resume, and Cancel Controls**

from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel

from PySide6.QtCore import QThread, Signal

import time

import sys

class WorkerThread(QThread):

"""Worker thread that supports pause, resume, and cancel."""

progress = Signal(int) # Signal to update UI progress

finished = Signal(str) # Signal to notify completion

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.\_is\_paused = False

self.\_is\_running = True

def run(self):

"""Background task that runs in a separate thread."""

for i in range(10):

if not self.\_is\_running:

self.finished.emit("Task Canceled")

return

while self.\_is\_paused: # Pause execution

time.sleep(0.1)

time.sleep(1) # Simulate work

self.progress.emit(i + 1) # Emit progress

self.finished.emit("Task Completed") # Emit completion message

def pause(self):

"""Pause the execution."""

self.\_is\_paused = True

def resume(self):

"""Resume execution."""

self.\_is\_paused = False

def stop(self):

"""Stop execution."""

self.\_is\_running = False

class MyWindow(QWidget):

"""Main Window with buttons to control threading."""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Pause, Resume, Cancel QThread")

self.resize(300, 200)

layout = QVBoxLayout()

self.label = QLabel("Click Start to begin")

layout.addWidget(self.label)

self.start\_button = QPushButton("Start Task")

self.start\_button.clicked.connect(self.start\_task)

layout.addWidget(self.start\_button)

self.pause\_button = QPushButton("Pause Task")

self.pause\_button.clicked.connect(self.pause\_task)

self.pause\_button.setEnabled(False)

layout.addWidget(self.pause\_button)

self.resume\_button = QPushButton("Resume Task")

self.resume\_button.clicked.connect(self.resume\_task)

self.resume\_button.setEnabled(False)

layout.addWidget(self.resume\_button)

self.cancel\_button = QPushButton("Cancel Task")

self.cancel\_button.clicked.connect(self.cancel\_task)

self.cancel\_button.setEnabled(False)

layout.addWidget(self.cancel\_button)

self.setLayout(layout)

def start\_task(self):

"""Start the worker thread."""

self.worker = WorkerThread()

self.worker.progress.connect(self.update\_label)

self.worker.finished.connect(self.task\_finished)

self.worker.start()

self.start\_button.setEnabled(False)

self.pause\_button.setEnabled(True)

self.cancel\_button.setEnabled(True)

def pause\_task(self):

"""Pause the worker thread."""

self.worker.pause()

self.pause\_button.setEnabled(False)

self.resume\_button.setEnabled(True)

def resume\_task(self):

"""Resume the worker thread."""

self.worker.resume()

self.pause\_button.setEnabled(True)

self.resume\_button.setEnabled(False)

def cancel\_task(self):

"""Stop the worker thread."""

self.worker.stop()

self.pause\_button.setEnabled(False)

self.resume\_button.setEnabled(False)

self.cancel\_button.setEnabled(False)

self.start\_button.setEnabled(True)

def update\_label(self, value):

"""Update UI safely from worker thread."""

self.label.setText(f"Progress: {value}/10")

def task\_finished(self, message):

"""Handle task completion."""

self.label.setText(message)

self.start\_button.setEnabled(True)

self.pause\_button.setEnabled(False)

self.resume\_button.setEnabled(False)

self.cancel\_button.setEnabled(False)

app = QApplication(sys.argv)

window = MyWindow()

window.show()

sys.exit(app.exec())

**2. How It Works**

**State Control Using Flags**

* self.\_is\_paused = False: Controls whether the thread should pause.
* self.\_is\_running = True: Controls whether the thread should stop.

**Key Methods**

| **Method** | **Description** |
| --- | --- |
| pause() | Sets \_is\_paused = True, making the thread wait in a loop. |
| resume() | Sets \_is\_paused = False, allowing execution to continue. |
| stop() | Sets \_is\_running = False, terminating execution. |

**3. Alternative Approach: Using QThreadPool & QRunnable**

If you want a more **efficient approach**, QThreadPool with QRunnable can also be used for pausable/cancelable tasks.