**Extra Lab 4– Synchronization Between Threads in PySide6**

**Objective:**

**Lab Exercise: Synchronization Between Threads in PySide6**

**Objective**

Learn how to **synchronize multiple threads** in PySide6 using QMutex, QWaitCondition, and QSemaphore to control execution order and prevent race conditions.

**Scenario**

* One thread (ProducerThread) **produces numbers** and adds them to a shared list.
* Another thread (ConsumerThread) **processes numbers** from the list.
* **Synchronization between threads** ensures the consumer waits until data is available.

**Step 1: Install Dependencies**

Ensure PySide6 is installed:

pip install PySide6

**Step 2: Implement the Code**

This program demonstrates synchronization using **QMutex** (for exclusive access) and **QWaitCondition** (to notify waiting threads).

import sys

import time

import random

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

from PySide6.QtCore import Qt, QThread, Signal, QMutex, QWaitCondition

# Shared resource (queue of numbers)

data\_queue = []

mutex = QMutex() # Synchronization lock

condition = QWaitCondition() # Condition variable for signaling

class ProducerThread(QThread):

produced\_signal = Signal(str) # Signal to update UI

def run(self):

global data\_queue

for \_ in range(5):

time.sleep(random.uniform(1, 2)) # Simulate time delay in producing data

mutex.lock()

num = random.randint(1, 100) # Produce a random number

data\_queue.append(num) # Add to shared list

self.produced\_signal.emit(f"Produced: {num}")

condition.wakeOne() # Notify waiting consumer

mutex.unlock()

class ConsumerThread(QThread):

consumed\_signal = Signal(str) # Signal to update UI

def run(self):

global data\_queue

for \_ in range(5):

mutex.lock()

while not data\_queue: # Wait until there is data to consume

condition.wait(mutex) # Wait until producer signals

num = data\_queue.pop(0) # Consume data

self.consumed\_signal.emit(f"Consumed: {num}")

mutex.unlock()

time.sleep(random.uniform(1, 2)) # Simulate processing time

class MainWindow(QWidget):

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

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

self.setWindowTitle("Thread Synchronization in PySide6")

self.setGeometry(200, 200, 400, 300)

# UI Elements

self.status\_label = QLabel("Press 'Start' to run producer & consumer threads.")

self.log\_text = QTextEdit()

self.log\_text.setReadOnly(True)

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

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

# Layout

layout = QVBoxLayout()

layout.addWidget(self.status\_label)

layout.addWidget(self.log\_text)

layout.addWidget(self.start\_button)

self.setLayout(layout)

def start\_threads(self):

"""Start producer and consumer threads."""

self.producer = ProducerThread()

self.consumer = ConsumerThread()

self.producer.produced\_signal.connect(self.update\_log)

self.consumer.consumed\_signal.connect(self.update\_log)

self.producer.start()

self.consumer.start()

def update\_log(self, message):

"""Update log with messages from threads."""

self.log\_text.append(message)

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

app = QApplication(sys.argv)

window = MainWindow()

window.show()

sys.exit(app.exec())

**Explanation**

1. **Shared Resources & Synchronization Mechanisms**
   * **data\_queue** – Shared list holding produced numbers.
   * **QMutex** – Ensures only one thread accesses data\_queue at a time.
   * **QWaitCondition** – Makes consumer wait if data\_queue is empty.
2. **Producer-Consumer Synchronization**
   * **Producer Thread**:
     + Generates numbers and adds them to data\_queue.
     + **Signals the consumer (wakeOne())** when new data is available.
   * **Consumer Thread**:
     + Waits (wait()) if data\_queue is empty.
     + Consumes numbers when notified by producer.
3. **Thread-Safe UI Updates**
   * **Signals (produced\_signal and consumed\_signal)** update the GUI safely.

**Expected Behavior**

1. Click **"Start Threads"** to run producer and consumer.
2. Producer **generates random numbers** and adds them to data\_queue.
3. Consumer **waits** until numbers are available, then processes them.
4. Log displays:

Produced: 42

Consumed: 42

Produced: 78

Consumed: 78