Lovely Professional University, Phagwara, Punjab

CS316: Operating Systems

Term Paper: Multithreading Models Simulator

Submitted By: Keshav Yadav

Registration Numbers: 12320523

Roll Numbers: 02

Section: K23TG

Submitted to: Mr. Akash Pundir

Marks Obtained	
Maximum Marks	
Remarks	

Multithreading Models Simulator

Project Overview

The Multithreading Models Simulator is an interactive Python-based GUI application developed to demonstrate and visualize core multithreading models used in operating systems. Using Python's tkinter for the graphical interface and the threading module for concurrency, the simulator provides a hands-on, educational experience that helps users understand how threads behave under different multithreading architectures.
The application simulates three primary threading models — Many-to-One, One-to-Many, and Many-to-Many — with real-time graphical feedback and activity logs. Users can input the desired number of threads, initiate the simulation for any of the models, and observe how threads are scheduled, executed, and synchronized in each architecture.
Each model showcases different synchronization strategies using Python constructs like locks and semaphores. The threads are visualized on a canvas with distinct colours and labels, making it easy to differentiate between models and track individual thread behaviour.

Key Features

- Model Simulation:
  - Many-to-One: Multiple user threads mapped to a single kernel thread. Threads are executed sequentially using a shared lock.
  - One-to-Many: Each user thread mapped to a separate kernel thread, enabling true parallel execution.
  - Many-to-Many: Multiple user threads mapped to a limited pool of kernel threads, controlled via semaphores to simulate resource sharing.
- Dynamic Thread Visualization:
  - Real-time display of thread execution as moving and 2labelled circles.
  - Color-coded threads for each model (e.g., Blue for Many-to-One, Green for One-to-Many, Red for Many-to-Many).
- Real-Time Logging:
  - Thread execution details are logged live in a scrollable console.
  - Feedback includes start, execution, and completion messages

for each thread.

- User-Controlled Simulation:
  - Input field for custom thread count.
  - Dedicated buttons to run simulations for each multithreading model independently.

Expected Outcomes

- A fully interactive and educational GUI tool to explore multithreading behaviour..
- Visual demonstration of thread synchronization and kernel-user thread mappings.
- Improved understanding of thread scheduling, context switching, and resource contention.
- A foundation for extending to more advanced operating system simulations like process management or deadlock detection.

Module-Wise Breakdown

1.1. Input Module

- Purpose:
  Collects the number of threads from the user to simulate various multithreading models.
- Key Functions:
- Accepts and parses user input for thread count.
- Validates numeric input to ensure logical thread creation.
- Prepares the system for initializing simulation based on user input.

1.2. Validation Module

- Purpose:
  Ensures that the user input is valid and simulation can proceed without errors.
- Key Functions:
- Verifies the thread count is a positive integer.
- Sets a default thread count if the input is invalid or empty.
- Provides real-time feedback on input issues in the log section.

1.3. Thread Execution Module

- Purpose:

Handles the core logic for simulating the behavior of each multithreading model.

- Key Functions:
- Defines threading logic for:
  - Many-to-One Model: Threads share a single kernel-level thread using a lock.
  - One-to-Many Model: Each thread runs on its own kernel thread.
  - Many-to-Many Model: A limited pool of kernel threads serve multiple user threads via a semaphore.
- Simulates execution time using random delays.
- Logs thread activity in real time.

1.4. Visualization Module

- Purpose:

  Visually represents thread execution on a canvas for better conceptual understanding.

- Key Functions:
- Dynamically creates and positions coloured nodes (threads) on the canvas.
- Uses distinct Y-axis rows for different models (e.g., top for Many-to-One, middle for One-to-Many).
- Associates' labels and colours with thread identities and models.

1.5. Logging Module

- Purpose:

  Keeps a real-time log of thread execution status and model progress.

- Key Functions:
- Outputs start, execution, and completion messages for each thread.
- Updates the GUI with smooth scrolling and auto-refresh for visibility.
- Displays model lifecycle (start to end).

1.6. Graphical User Interface (GUI) Module

- Purpose:

  Acts as the main user interaction layer for the simulator.

- Key Functions:

- Built using tkinter.
- Includes input fields, buttons, and layout components.
- Integrates canvas for visualization and text widget for logs.
- Controls thread simulation via dedicated buttons.

1.7. Export/Extension Module (Optional/Future Scope)

- Purpose:
  To allow saving logs or extending the simulator with new features like Gantt charts or performance analysis.
- Key Functions:
- Save logs to .txt files for documentation.
- Enable screenshots of the canvas for educational purposes.
- Future: Add CPU time stats, turnaround time, or thread starvation detection.

Functionalities

1. Input Data Management
    - Allows users to input the number of threads and select simulation models (Many-to-One, One-to-Many, Many-to-Many).
    - Provides an intuitive interface for initiating thread simulations and viewing real-time execution.
2. Data Validation and Parsing
    - Automatically checks for valid numeric input for thread counts.
    - Ensures thread count is a positive integer, with default fallback logic if input is invalid.
3. Thread Execution Simulation
    - Implements distinct multithreading models:
        - Many-to-One: All threads mapped to a single kernel thread using locks.
        - One-to-Many: Each user-level thread maps to an individual kernel-level thread.
        - Many-to-Many: A pool of kernel threads (via semaphores) serves multiple user threads.
4. Visualization of Thread Activity
    - Renders real-time visual representation of thread execution using a canvas.
    - Assigns color-coded nodes to differentiate thread groups by model.
    - Dynamic movement and visualization help in understanding concurrency behaviours.
5. Real-Time Logging
    - Maintains a live activity log to show when threads start, execute, wait (if applicable), and complete.

- Enhances learning by showing the internal timeline of multithreaded execution.

6. Simulation Feedback
    - Displays model completion messages.
    - Provides visual and textual cues to help users understand thread lifecycle behavior.
7. Export and Extend Capabilities (Future Scope)
    - Additions like exporting thread logs to text files.
    - Capture visual thread execution as images for reports or assignments.
    - Extensions may include Gantt chart generation or thread starvation analysis.

Technologies Used

1. Programming Language
    - Python 3.13: Primary language for logic implementation, threading, and UI management.
2. Graphical User Interface (GUI)
    - Tkinter:
        - Used to create the interactive GUI.
        - Manages layout, input fields, buttons, and output components like canvas and text logs.
3. Thread Management
    - Python threading module:
        - Enables creation and execution of threads simulating different multithreading models.

4. Visualization

- Tkinter Canvas:
    - Used to visually display thread nodes, simulate their execution, and animate their state changes.

5. Optional Visualization Enhancement (Future Scope)
    - Matplotlib / NetworkX:
        - Can be integrated to show advanced thread/resource dependency graphs (if expanded into OS-level thread/process simulation).
6. Algorithmic Simulation Tools
    - Semaphore and Lock Mechanisms:
        - Used to control access to shared resources in Many-to-Many and Many-to-One models.
7. Development Environment
    - PyCharm:
        - IDE recommended for code development, testing, debugging, and version control integration.

Revision Tracking on GitHub

- Repository Name: Operating-Systems

GitHub Link: https://github.com/Harshvardhan0306/Operating-Systems.git

Conclusion and Future Scope

Conclusion:

The Multithreading Models Simulator effectively visualizes and simulates key multithreading models, including Many-to-One, One-to-Many, and Many-to-Many. With an interactive GUI built using Tkinter, it provides real-time visual feedback and logs of thread execution, making it easier to grasp threading behaviour in different models.

By incorporating Python's threading library along with graphical elements like canvas-based thread visualization, this tool demonstrates core concepts of concurrency, synchronization, and resource sharing in operating systems. It serves as an excellent educational aid for students, as well as a practical demonstrator for developers and educators.

Future Enhancements:

1. Advanced Thread Control

- Introduce features like thread pause/resume and priority scheduling.
- Allow users to simulate thread blocking, waiting, and signalling mechanisms.

2. Performance Monitoring

- Include CPU usage, memory stats, and execution time tracking for each thread model.
- Add live charts to compare model efficiency under different workloads.

3. Animated Visualization

- Implement real-time animated transitions for thread movements and resource usage.
- Colour-code thread states (e.g., running, waiting, terminated) for better clarity.

4. Web-Based Deployment

- Migrate the simulator to a web platform using frameworks like Flask or Streamlit to allow remote access and collaboration.

5. Educational Modules

- Add guided scenarios (e.g., producer-consumer or reader-writer problems).
- Include quizzes or tutorials alongside each model to support learning outcomes.

References

- GeeksforGeeks. (n.d.). Multithreading in Operating Systems. Retrieved from https://www.geeksforgeeks.org/
- Parvati, P. (n.d.). Multithreading and Thread Synchronization in C. Retrieved from https://www.linkedin.com/
- Brazilian Journal of Science. (n.d.). Multi-threading and Cache Memory Latency Masking. Retrieved from https://www.brazilianjournalofscience.org/
- Stack Overflow. (n.d.). Thread Scheduling and Synchronization in Java. Retrieved from https://stackoverflow.com/
- TutorialsPoint. (n.d.). Operating System - Processes. Retrieved from https://www.tutorialspoint.com/
- Programiz. (n.d.). Learn C Programming. Retrieved from https://www.programiz.com/
- Oracle. (n.d.). Java SE Documentation - Concurrency. Retrieved from https://docs.oracle.com/
- Microsoft Docs. (n.d.). Threading (C#). Retrieved from https://docs.microsoft.com/
- IBM Knowledge Center. (n.d.). Thread Synchronization. Retrieved from https://www.ibm.com/support/knowledgecenter/
- TechTarget. (n.d.). What is Thread Safety and Why is it Important? Retrieved from https://searchapparchitecture.techtarget.com/
- The Crazy Programmer. (n.d.). Operating System Tutorial. Retrieved from https://www.thecrazyprogrammer.com/

- Rajagopalan, M., Lewis, B.T., & Anderson, T.A. (Year). Thread Scheduling for Multi-Core Platforms. [Research Paper].
- Symonds, A., Arora, S.K.M., & Kaif, M.Y. (Year). Process Synchronization. [Research Paper].
- Donaldson, J.L. (Year). Implementation of Threads as an Operating System. [Research Paper].

Appendix

A. Project Overview

Title: Multithreading Models Simulator

Goals:
Simulate key multithreading models: Many-to-One, One-to-Many, and Many-to-Many.
Visualize thread scheduling and execution behavior under different models.
Include optional deadlock prevention and recovery techniques for educational analysis.

Expected Outcomes:
Thread Visualization: Real-time thread execution shown via animated GUI using Tkinter canvas.
Model Comparison: Clear distinctions between threading models and how they manage CPU/kernel interaction.
User Interactivity: Adjustable thread count and model selection through a friendly interface.

Scope:
Simulation Logic: Core simulation of threading models using Python's threading and synchronization primitives.
Visualization: GUI-based live rendering of thread activity.
Educational Utility: Demonstrates scheduling behavior and possible contention/resource management.

B. Module-Wise Technical Breakdown

Module 1: Threading Model Engine
Technology: Python threading, semaphore, lock
Functionality:
Implements logic for:
Many-to-One: Multiple user threads mapped to a single kernel thread using a shared lock.
One-to-Many: One user thread mapped to multiple kernel threads using parallel execution.

Many-to-Many: Multiple user threads managed by limited kernel threads using semaphores.
Code Snippet:
python
CopyEdit
```
def many_to_one_thread(name, index):
    with resource_lock:
        ...
def one_to_many_worker(name, index):
    ...
def many_to_many_worker(name, index):
    with semaphore:
        ...
```
Inputs: Number of threads (user-defined)
Output: Execution logs and real-time visual feedback

Module 2: Visualization Engine
Technology: Tkinter Canvas
Functionality:
Draws thread icons dynamically with color codes:
Blue: Many-to-One
Green: One-to-Many
Red: Many-to-Many
Supports animated placement and labeling
Code Snippet:
python
CopyEdit
```
def draw_thread(x, y, color, text):
    canvas.create_oval(...)
    canvas.create_text(...)
```
Visual Elements:
Threads are shown in separate rows by model
Real-time updates as threads execute

Module 3: User Interaction & GUI
Technology: Tkinter + ttk
Components:
Entry field to accept number of threads
Buttons to launch simulation for each model

Log window for live feedback
Code Snippet:
python
CopyEdit
many_to_one_button = ttk.Button(root, text="Run Many-to-One", command=many_to_one)
Impact: Enables easy switching between models, customizable thread count, and live logs

---

Module 4: Logging and Feedback System
Technology: Tkinter Text Widget
Functionality:
Real-time feedback on thread execution
Logs when a thread starts, accesses a resource, and finishes
Code Snippet:
python
CopyEdit
def log_activity(message):
    log_text.insert(...)
reports).

Functionalities

Feature	Technical Implementation
Many-to-One Simulation	Uses threading.Thread objects synchronized by a single Lock to simulate a single kernel thread.
One-to-Many Simulation	Spawns' multiple threads executed in parallel to simulate multiple kernel threads per user thread.
Many-to-Many Simulation	Controlled with Semaphore to limit concurrent access to simulate kernel thread pool behaviour.
Thread Visualization	Uses Tkinter Canvas to draw threads with color-coded ovals and labels based on the model.
Live Execution Logs	GUI-based log area shows real-time thread actions (start, access, completion).
User Input	Accepts thread count dynamically through GUI input to control simulation scale.
Interactive GUI	Buttons to launch each model, and a text log area for monitoring and educational interaction.
Thread Coordination & Sync	Demonstrates effects of synchronization primitives like Lock and Semaphore.
Educational Clarity	Highlights differences in scheduling and kernel mapping across models for deeper understanding.

Technology Stack

Component	Technology
Core Logic	Python threading, semaphore, lock
Visualization	Tkinter Canvas for GUI thread visualization
GUI	Tkinter + ttk
Logging	Tkinter Text Widget for live thread logging

Execution Plan

Phase	Timeline	Description
Phase 1: Core Simulation Engine	4 Days	Implement Many-to-One, One-to-Many, and Many-to-Many threading models.
Phase 2: GUI Layout Design	3 Days	Design the interface, including buttons, input fields, and output areas.
Phase 3: Visualization Integration	3 Days	Integrate dynamic thread drawing on the canvas with proper color-coding.
Phase 4: Real-Time Logging	2 Days	Add detailed, scrolling logs for thread activities in each model.
Phase 5: Testing & Validation	2 Days	Ensure stability across different thread counts, simulate race conditions.

Future Scope

- Web Deployment:
  - Convert the desktop GUI to a web-based simulator using Flask or Django for browser-based access.
- Model Behaviour Metrics:
  - Integrate timers and performance metrics (execution time, context switches) for each threading model.
- Interactive Animations:
  - Use libraries like Plotly or Tkinter Canvas animations for smoother and more engaging thread flows.
- AI-based Optimization:
  - Incorporate ML-based schedulers to demonstrate dynamic model selection based on load conditions.
- Multilingual Interface:
  - Support multilingual GUIs for wider accessibility in educational contexts.

Problem Statement:

Multithreading Models Simulator
As operating systems evolve to support more efficient multitasking, understanding how user threads are mapped to kernel threads becomes essential for both system developers and students. The behaviour of Many-to-One, One-to-Many, and Many-to-Many threading models significantly influences performance, concurrency, and synchronization complexity.
This project aims to simulate and visually represent these three multithreading models. It provides an interactive environment where users can choose a model, define the number of threads, and observe how threads are scheduled and executed in real time. Through visual aids and live logs, users can explore:

- Thread contention,
- Resource sharing under limited concurrency,
- Synchronization mechanisms like semaphores and locks.

By abstracting complex OS-level concepts into a graphical and interactive tool, this simulator bridges theory and practice — making it a powerful learning resource for operating systems coursework, labs, and self-paced study.

- C. Solution/Code:

```
import threading
import time
import random
import tkinter as tk
from tkinter import ttk, Canvas

# Shared resource and semaphore for synchronization
semaphore = threading.Semaphore(2)
resource_lock = threading.Lock()

# GUI Setup
root = tk.Tk()
root.title("Multithreading Models Simulator")
root.geometry("800x600")

log_text = tk.Text(root, height=15, width=80)
log_text.pack()

canvas = Canvas(root, width=700, height=300, bg="white")
canvas.pack()

thread_positions = {}  # Dictionary to track thread positions

def log_activity(message):
    log_text.insert(tk.END, message + "\n")
    log_text.see(tk.END)
    root.update_idletasks()

def draw_thread(x, y, color, text):
    canvas.create_oval(x, y, x + 40, y + 40, fill=color, outline="black")
    canvas.create_text(x + 20, y + 20, text=text, fill="white")

def get_thread_count():
    try:
        return max(1, int(thread_count_entry.get()))
    except ValueError:
        return 3  # Default value

# Many-to-One Model
def many_to_one_thread(name, index):
    with resource_lock:
        root.after(0, draw_thread, 100 + index * 60, 50, "blue", name)
        log_activity(f"{name} is running on a single kernel thread")
        time.sleep(random.uniform(1, 3))
        log_activity(f"{name} finished execution")

def many_to_one():
    def run():
```

```
        log_activity("Starting Many-to-One Model...")
        canvas.delete("all")
        num_threads = get_thread_count()
        threads = [threading.Thread(target=many_to_one_thread, args=(f"T{i}", i)) for i in range(num_threads)]
        for t in threads:
            t.start()
        for t in threads:
            t.join()
        log_activity("Many-to-One Model Completed.\n")
    threading.Thread(target=run).start()

# One-to-Many Model
def one_to_many_worker(name, index):
    root.after(0, draw_thread, 100 + index * 60, 150, "green", name)
    log_activity(f"{name} starts execution in multiple kernel threads")
    time.sleep(random.uniform(1, 3))
    log_activity(f"{name} finished execution")

def one_to_many():
    def run():
        log_activity("Starting One-to-Many Model...")
        canvas.delete("all")
        num_threads = get_thread_count()
        kernel_threads = [threading.Thread(target=one_to_many_worker, args=(f"KThread{i}", i)) for i in range(nu
        for kt in kernel_threads:
            kt.start()
        for kt in kernel_threads:
            kt.join()
        log_activity("One-to-Many Model Completed.\n")
    threading.Thread(target=run).start()

# Many-to-Many Model
def many_to_many_worker(name, index):
    with semaphore:
        root.after(0, draw_thread, 100 + index * 60, 250, "red", name)
        log_activity(f"{name} is accessing the resource")
        time.sleep(random.uniform(1, 2))
        log_activity(f"{name} released the resource")

def many_to_many():
    def run():
        log_activity("Starting Many-to-Many Model...")
        canvas.delete("all")
        num_threads = get_thread_count()
        threads = [threading.Thread(target=many_to_many_worker, args=(f"T{i}", i)) for i in range(num_threads)]
```

(parameter) index: Any

```
        for t in threads:
            t.start()
        for t in threads:
            t.join()
        log_activity("Many-to-Many Model Completed.\n")
    threading.Thread(target=run).start()

# User input for thread count
thread_count_label = tk.Label(root, text="Enter number of threads:")
thread_count_label.pack()
thread_count_entry = tk.Entry(root)
thread_count_entry.pack()

# UI Buttons
many_to_one_button = ttk.Button(root, text="Run Many-to-One", command=many_to_one)
many_to_one_button.pack()

one_to_many_button = ttk.Button(root, text="Run One-to-Many", command=one_to_many)
one_to_many_button.pack()

many_to_many_button = ttk.Button(root, text="Run Many-to-Many", command=many_to_many)
many_to_many_button.pack()

root.mainloop()
```