

# **PROCESS SYNCHRONIZATION VISUALISER**

MINI PROJECT REPORT

By

**SURIYA M (RA2211047010098)**

**DEEPAK DEVAKUMAR S(RA2211047010076)**

**MUGUNTHAN S(RA2211047010070)**

Under the guidance of

**Dr. K. SURESH**

**Assistant Professor**

*In partial fulfilment for the Course*

Of

**21CSC202J - OPERATING SYSTEMS**

in the Department of Computational Intelligence



**FACULTY OF ENGINEERING AND TECHNOLOGY**

**SCHOOL OF COMPUTING**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR**

**MAY 2024**

# **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

## **BONAFIDE CERTIFICATE**

Certified that this minor project report for the course **21CSC202J - OPERATING SYSTEMS** entitled in " **PROCESS SYNCHRONIZATION VISUALISER**" is the bonafide work of **SURIYA M** (RA2211047010098) **DEEPAK DEVAKUMAR** (RA2211047010076) and **MUGUNTHAN S**(RA2211047010070) who carried out the work under my supervision.

### **SIGNATURE**

Dr. K.Suresh

**Assistant Professor**

Department of CINTEL

SRM Institute of Science and Technology

Kattankulathur

### **SIGNATURE**

Dr Annie Uthra R

**ProfessorHead of the**

**Department**

Department of CINTEL

SRM Institute of Science and Technology

Kattankulathur

**DEPARTMENT OF COMPUTATIONAL INTELLIGENCE**  
**SCHOOL OF COMPUTING**  
**College of Engineering and Technology**  
**SRM Institute of Science and Technology**

**MINI PROJECT REPORT**

**EVEN Semester, 2023-2024**

Lab code & Sub Name : 21CSC202J & OPERATING SYSTEMS

Year & Semester : II & III

Project Title : **Query handler using a round-robin algorithm**

Name of the Supervisor : **Dr. K SURESH**

Team Members :  
1) Mugunthan S (RA2211047010070)  
2) Deepak Devakumar S (RA2211047010076)  
3) Suriya M(RA2211047010098)

PARTICULARS	MAX MARK	MARKS OBTAINED  NAME:  REGISTER NO:
Implementation	10	
Review	02	
Project Review	08	
Total	20	

## TABLE OF CONTENTS

S.NO	CONTENTS	PAGE NO
1	Objective	5
2	Abstract	6
3	Introduction	7
4	Requirements	8
5	Working Principle	9
6	Architecture And Implementation	10
7	Flowchart	17
8	Result and Discussion	18
9	Conclusion	21
10	References	22

## OBJECTIVE

The project aims to provide practical insights into real-time process monitoring, producer-consumer interactions, and effective race condition prevention, utilizing the Tkinter framework for user interface development in Python. The programs serve as educational tools to explore and understand fundamental concepts in concurrent programming

### 1. Module 1:

- Provide a real-time graphical representation of running processes, displaying arrival and closing times dynamically.
- Enhance understanding of process lifecycle and real-time system monitoring.

### 2. Module 2:

- Simulate a simplified producer-consumer scenario, emphasizing challenges related to process arrival times, waiting times, and burst times in a tabular format.
- Illustrate resource-sharing complexities in concurrent systems.

### 3. Module 3:

- Demonstrate the impact of race conditions in concurrent programming and showcase the effectiveness of semaphores in preventing conflicts during resource access.
- Enhance comprehension of race condition issues and solutions in concurrent application development.

## **ABSTRACT**

An operating system is software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

Process synchronization is a crucial concept in computer science and operating systems. It involves coordinating multiple processes or threads to access shared resources in an orderly manner. Synchronization mechanisms like mutexes, semaphores, and barriers prevent issues like race conditions and ensure data integrity by controlling access to shared resources and ensuring orderly execution, making programs more stable and reliable. This project report presents the development and implementation of three distinct programs using the Tkinter framework in Python. The "Process Monitor" provides real-time graphical insights into running processes, capturing arrival and closing times dynamically. The "Producer-Consumer Simulator" illustrates challenges in concurrent programming through a graphical interface, detailing process arrival and execution times in a tabular format. The "Race Condition Demo with Semaphore" highlights the impact of race conditions and demonstrates the effective use of semaphores to prevent conflicts in shared resource access.

## INTRODUCTION

Process synchronization is a foundational concept in computer science and operating systems, particularly in environments with multiple concurrent processes or threads. Concurrency, referring to the simultaneous execution of processes, poses challenges when these processes access shared resources concurrently, potentially leading to data corruption or inconsistencies. The concept of a critical section defines a part of the code where a process interacts with shared resources, and the objective is to ensure that only one process executes its critical section at a time to avoid conflicts. Race conditions, arising from uncontrolled access to shared resources, can result in unpredictable program behavior. Achieving mutual exclusion, where only one process executes its critical section at any given time, is vital for preventing conflicts, and techniques like locks, semaphores, and mutexes are employed for this purpose. Various synchronization mechanisms, including locks, semaphores, and condition variables, are used to control access to shared resources and facilitate signaling between processes. Deadlock, a situation where processes are unable to proceed due to mutual waiting for resources, can be prevented through careful design and the judicious use of synchronization mechanisms. Overall, process synchronization is crucial for maintaining order, preventing race conditions, and ensuring the predictable execution of concurrent processes.



# REQUIREMENTS

The system requirements for understanding and implementing the concepts of process synchronization, concurrency, and operating systems, as well as for running the provided Python programs, include:

## **1. Hardware Requirements:**

A computer with sufficient processing power and memory to run the chosen operating system and programming environment.

## **2. Operating System:**

A modern operating system that supports multi-process and multi-threaded environments. Common choices include Linux, Windows, or macOS.

## **3. Programming Language:**

Knowledge of a programming language that supports concurrent programming. Python, C, Java, and others are commonly used for illustrating these concepts.

## **4. Development Environment:**

An integrated development environment (IDE) or text editor is suitable for the chosen programming language. For Python, IDEs like PyCharm, Visual Studio Code, or Jupyter Notebooks are popular.

## **5. Python Libraries (for the provided Python code):**

Tkinter: A standard GUI toolkit for Python.

psutil: A cross-platform library for retrieving information on running processes and system utilization.

## **6. Access to a Command Line Interface (CLI):**

For running and testing programs, a command line interface or terminal is useful. This is especially relevant for running Python scripts or compiling and executing programs in languages like C.

Remember that specific requirements may vary based on the complexity of the programs or concepts you are exploring. It's advisable to check the documentation and system requirements of any specific tools or libraries you plan to use.

## WORKING PRINCIPLE

The project encompasses three interconnected programs developed using Tkinter in Python, each designed to elucidate distinct principles in computer science and concurrent programming.

### ➤ **Process Monitor:**

Utilizing Tkinter, the Process Monitor captures real-time data on running processes. By employing the Psutil library, the program extracts information on process arrival and closing times, updating a dynamic graph to reflect the system's current state. This provides users with an instantaneous visual representation of process dynamics, aiding in system analysis and resource management.

### ➤ **Producer-Consumer Simulator:**

The Producer-Consumer Simulator emulates a simplified resource-sharing scenario. Tkinter facilitates the creation of a graphical interface where producers and consumers interact. The program utilizes threading to manage process arrival times, waiting times, and burst times, presenting the intricacies of concurrent execution. A tabular format visually communicates the interactions, offering a clear insight into the challenges of producer-consumer relationships.

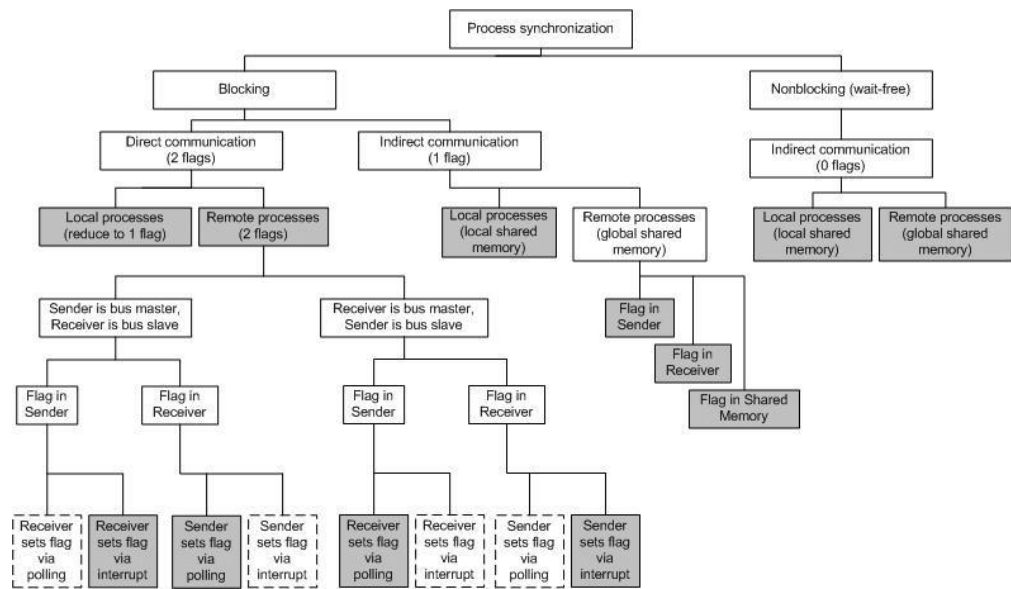
### ➤ **Race Condition Demo with Semaphore:**

Employing Tkinter, the Race Condition Demo with Semaphore illustrates the consequences of race conditions in concurrent programming. The program integrates a semaphore to control access to a shared resource, preventing conflicts and ensuring orderly execution. Through buttons for incrementing and decrementing, users trigger processes, and the application logs detailed information in a table, demonstrating how semaphores effectively mitigate race condition issues.

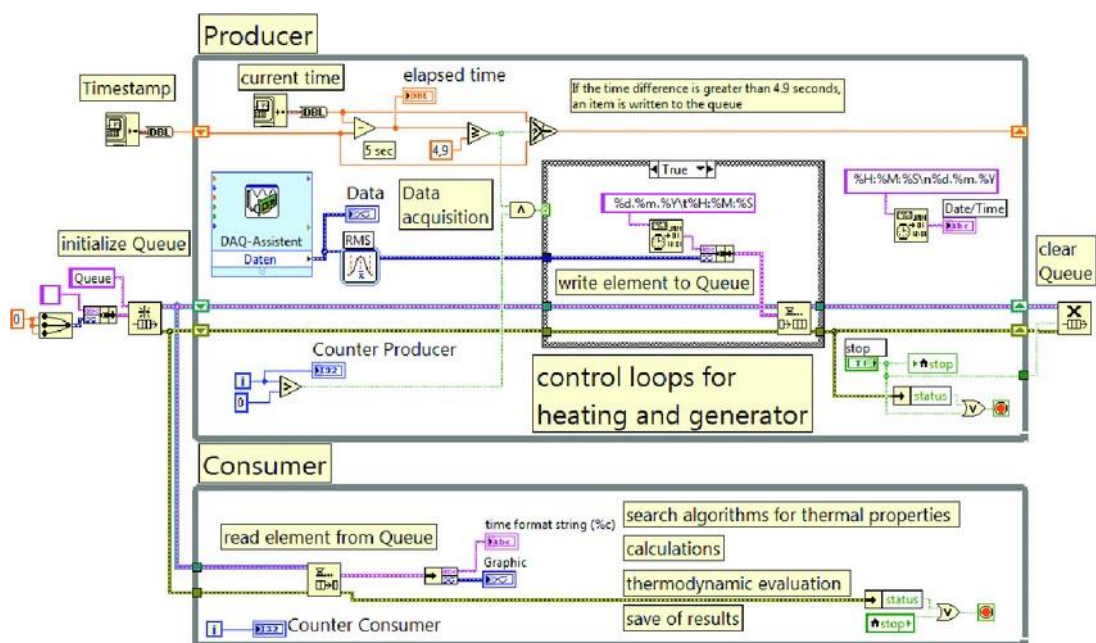
These programs collectively serve as educational tools, providing hands-on experience and insights into real-time system monitoring, producer-consumer interactions, and the practical application of synchronization mechanisms like semaphores in concurrent programming. The Tkinter framework facilitates the creation of interactive graphical interfaces, enhancing user engagement and understanding.

# ARCHITECTURE AND DESIGN

## Process Synchronization:



## Producer-Consumer Simulator:



# IMPLEMENTATION

## ❖ Real-time Process Synchronization Program:

```
import tkinter as tk
from tkinter import ttk
import psutil
import threading
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from datetime import datetime

class ProcessMonitor:
    def __init__(self, root):
        self.root = root
        self.root.title("Process Monitor")
        self.root.geometry("800x600")
        self.root.configure(bg="#f0e68c") # Set background color to light brown

        # Increase font size
        custom_font = ("Arial", 14)

        self.label = ttk.Label(root, text="Real Time Process graph", foreground="#ff0000",
                                font=custom_font, background="#f0e68c")
        self.label.place(x=280, y=320)

        self.label_processes = ttk.Label(root, text="Number of Processes: 0", font=custom_font,
                                           background="#f0e68c")
        self.label_processes.place(x=150, y=370)

        self.tree = ttk.Treeview(root, columns=("Arrival Time", "Closing Time"), show="headings",
                                   selectmode="browse")
        self.tree.heading("Arrival Time", text="Arrival Time")
        self.tree.heading("Closing Time", text="Closing Time")
        self.tree.column("Arrival Time", width=150)
        self.tree.column("Closing Time", width=150)
        self.tree.place(x=250, y=420)

        self.figure, self.ax = plt.subplots(figsize=(5, 3), tight_layout=True)
        self.canvas = FigureCanvasTkAgg(self.figure, master=root)
        self.canvas.get_tk_widget().pack(pady=10)
        self.canvas.draw()

        self.is_running = True
        self.thread = threading.Thread(target=self.update_data)
        self.thread.start()

        self.prev_num_processes = 0
```

```

self.processes_data = []

def update_data(self):
    x_data = [0]
    y_data = [0]

    while self.is_running:
        processes = psutil.process_iter()
        num_processes = 0
        for _ in processes:
            num_processes += 1

        x_data.append(x_data[-1] + 1)
        y_data.append(num_processes)

        if len(x_data) > 10: # Limit data points for better visualization
            x_data = x_data[-10:]
            y_data = y_data[-10:]

        self.label_processes.config(text=f"Number of Processes: {num_processes}")

        # Check for a decrease in the number of processes
        if num_processes < self.prev_num_processes:
            reduction = self.prev_num_processes - num_processes
            reduction_text = f"({reduction} processes closed)"
            self.label_processes.config(text=f"Number of Processes: {num_processes} {reduction_text}")

        # Check for newly arrived processes
        if num_processes > self.prev_num_processes:
            for _ in range(num_processes - self.prev_num_processes):
                arrival_time = datetime.now().strftime("%H:%M:%S")
                self.processes_data.append((arrival_time, "-"))

        # Check for closed processes
        if num_processes < self.prev_num_processes:
            for _ in range(self.prev_num_processes - num_processes):
                closing_time = datetime.now().strftime("%H:%M:%S")
                self.processes_data[-1] = (self.processes_data[-1][0], closing_time)

        # Display up to 5 processes in the table
        for i, (arrival_time, closing_time) in enumerate(self.processes_data[-5:]):
            self.tree.insert("", i, values=(arrival_time, closing_time))

        self.prev_num_processes = num_processes

    self.ax.clear()
    self.ax.plot(x_data, y_data, marker='o')
    self.ax.set_xlabel("Time")
    self.ax.set_ylabel("Number of Processes")
    self.canvas.draw()

    self.root.update()
    self.root.after(1000) # Update every second

```

```

def stop_monitoring(self):
    self.is_running = False
    self.thread.join()

if __name__ == "__main__":
    root = tk.Tk()
    monitor = ProcessMonitor(root)
    root.protocol("WM_DELETE_WINDOW", monitor.stop_monitoring)
    root.mainloop()

```

### ❖ **Producer-Consumer Simulator Program:**

```

import tkinter as tk
from tkinter import ttk
import threading
import time
from queue import Queue
import random

class ProducerConsumerSimulator:
    def __init__(self, root):
        self.root = root
        self.root.title("Producer-Consumer Simulator")
        self.root.geometry("800x400")
        self.root.configure(bg="#8B4513") # Set background color to dark brown

        # Create variables for user input
        self.num_processes_var = tk.StringVar()

        # Set up GUI components
        ttk.Label(root, text="Number of Processes:").pack(pady=10)
        entry_num_processes = ttk.Entry(root, textvariable=self.num_processes_var)
        entry_num_processes.pack(pady=10)

        start_button = ttk.Button(root, text="Start", command=self.start_simulation)
        start_button.pack(pady=20)

        # Create a treeview for displaying process information
        self.tree = ttk.Treeview(root, columns=("Process", "Arrival Time", "Waiting Time", "Burst Time",
        "Producer", "Consumer", "Resource Type"),
        show="headings", selectmode="browse")
        self.tree.heading("Process", text="Process")
        self.tree.heading("Arrival Time", text="Arrival Time (s)")
        self.tree.heading("Waiting Time", text="Waiting Time (s)")
        self.tree.heading("Burst Time", text="Burst Time (s)")
        self.tree.heading("Producer", text="Producer")
        self.tree.heading("Consumer", text="Consumer")
        self.tree.heading("Resource Type", text="Resource Type")
        self.tree.column("Process", width=80, anchor="center")
        self.tree.column("Arrival Time", width=120, anchor="center")

```

```

self.tree.column("Waiting Time", width=120, anchor="center")
self.tree.column("Burst Time", width=120, anchor="center")
self.tree.column("Producer", width=80, anchor="center")
self.tree.column("Consumer", width=80, anchor="center")
self.tree.column("Resource Type", width=120, anchor="center")
self.tree.pack(pady=20)

self.is_running = False
self.queue = Queue()

def start_simulation(self):
    num_processes = int(self.num_processes_var.get())

    if num_processes <= 0:
        tk.messagebox.showerror("Error", "Please enter a valid number of processes.")
        return

    if self.is_running:
        tk.messagebox.showinfo("Info", "Simulation is already running.")
        return

    self.is_running = True
    self.tree.delete(*self.tree.get_children())

    # Start the simulation in a separate thread
    threading.Thread(target=self.simulate_processes, args=(num_processes,), daemon=True).start()

def simulate_processes(self, num_processes):
    for process_id in range(1, num_processes + 1):
        arrival_time = time.time()
        producer_id = random.randint(1, 5)
        consumer_id = random.randint(1, 5)
        resource_type = random.choice(["TypeA", "TypeB", "TypeC"])

        self.queue.put((process_id, arrival_time, producer_id, consumer_id, resource_type))

        # Simulate burst time (waiting time)
        time.sleep(1)

        waiting_time = time.time() - arrival_time

        # Update the GUI in the main thread
        self.root.after(0, self.update_treeview, process_id, arrival_time, waiting_time, producer_id,
        consumer_id, resource_type)

    self.is_running = False

def update_treeview(self, process_id, arrival_time, waiting_time, producer_id, consumer_id,
resource_type):
    self.tree.insert("", "end", values=(process_id, f"{arrival_time:.2f} s", f"{waiting_time:.2f} s",
    "Simulated Burst", producer_id, consumer_id, resource_type))

if __name__ == "__main__":

```

```
root = tk.Tk()
app = ProducerConsumerSimulator(root)
root.mainloop()
```

### ❖ Race Condition Demo Program:

```
import tkinter as tk
from tkinter import ttk
from threading import Thread, Semaphore
import time
import random

class RaceConditionDemo:
    def __init__(self, root):
        self.root = root
        self.root.title("Race Condition Demo with Semaphore")
        self.root.geometry("600x400")

        self.shared_resource = 0
        self.semaphore = Semaphore()

        self.label = ttk.Label(root, text="Shared Resource: 0", font=("Arial", 14))
        self.label.pack(pady=10)

        # Create buttons for incrementing the shared resource
        self.increment_button = ttk.Button(root, text="Increment",
command=self.increment_shared_resource)
        self.increment_button.pack(pady=5)

        # Create buttons for decrementing the shared resource
        self.decrement_button = ttk.Button(root, text="Decrement",
command=self.decrement_shared_resource)
        self.decrement_button.pack(pady=5)

        # Create a table to display relative timings and sample processes
        columns = ("Relative Time", "Process ID", "Number of Processes", "Operation", "Result")
        self.tree = ttk.Treeview(root, columns=columns, show="headings", height=10)
        for col in columns:
            self.tree.heading(col, text=col)
        self.tree.pack(pady=10)

        self.process_counts = {"Increment": 0, "Decrement": 0}

    def increment_shared_resource(self):
        with self.semaphore:
            current_value = self.shared_resource
            time.sleep(1)
            self.shared_resource = current_value + 1
            self.update_label("Increment", current_value, "+1")
```



```

def decrement_shared_resource(self):
    with self.semaphore:
        current_value = self.shared_resource
        time.sleep(1)
        self.shared_resource = current_value - 1
        self.update_label("Decrement", current_value, "-1")

def update_label(self, operation, current_value, result):
    timestamp = time.strftime("%H:%M:%S")
    process_id = random.randint(1000, 9999)
    relative_time = time.time() # Get relative time

    process_type = "Increment" if operation == "Increment" else "Decrement"
    self.process_counts[process_type] += 1
    num_processes = self.process_counts[process_type]

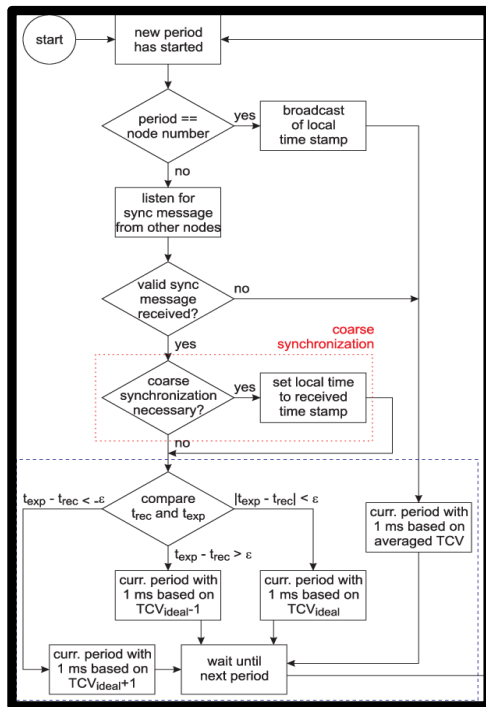
    self.tree.insert("", "end", values=(f"{timestamp}", f"{process_id}", num_processes, operation,
result))
    self.tree.yview(tk.END) # Automatically scroll to the bottom of the table
    self.label.config(text=f"Shared Resource: {self.shared_resource}")

if __name__ == "__main__":
    root = tk.Tk()
    app = RaceConditionDemo(root)
    root.mainloop()

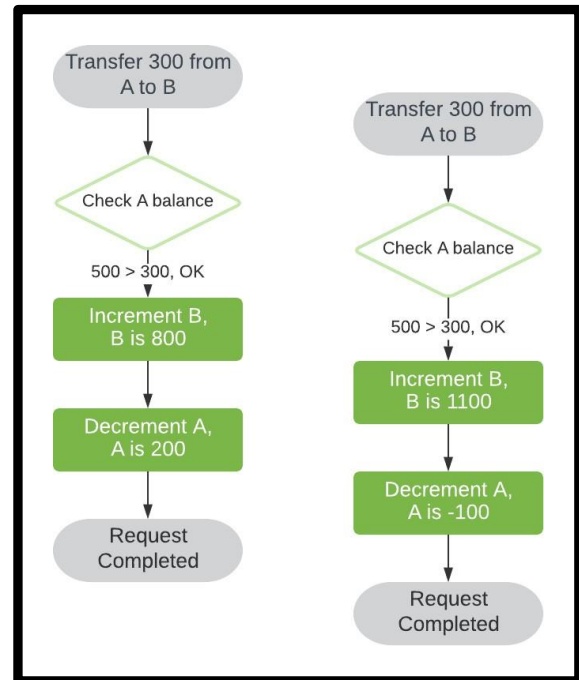
```

# FLOWCHART

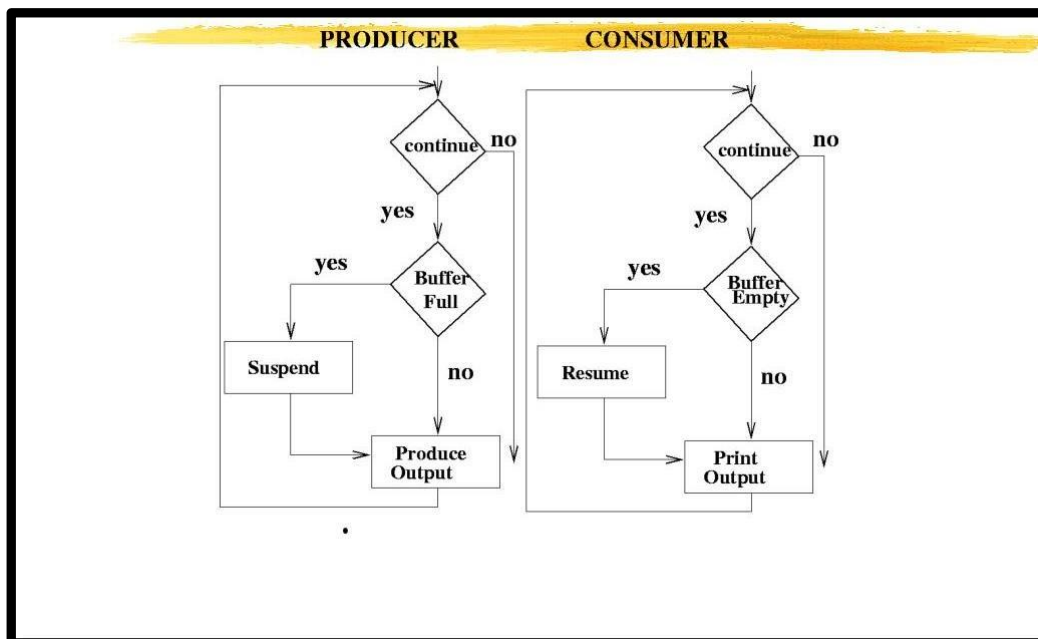
## Module 1:



## Module 2:

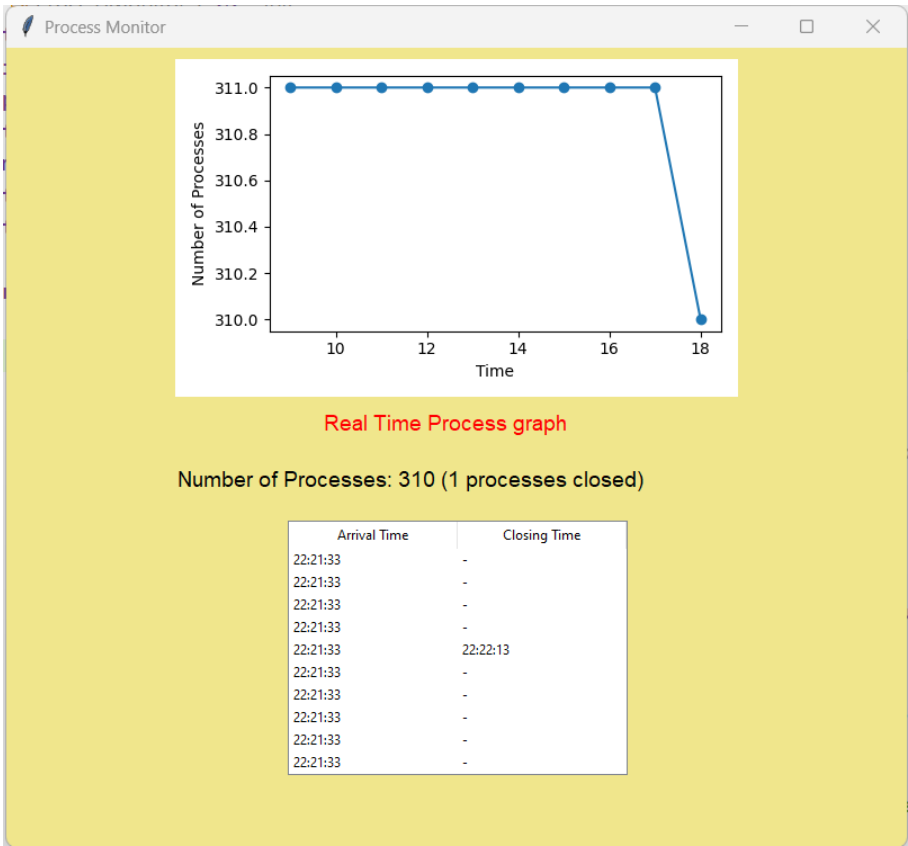


## Module 3:



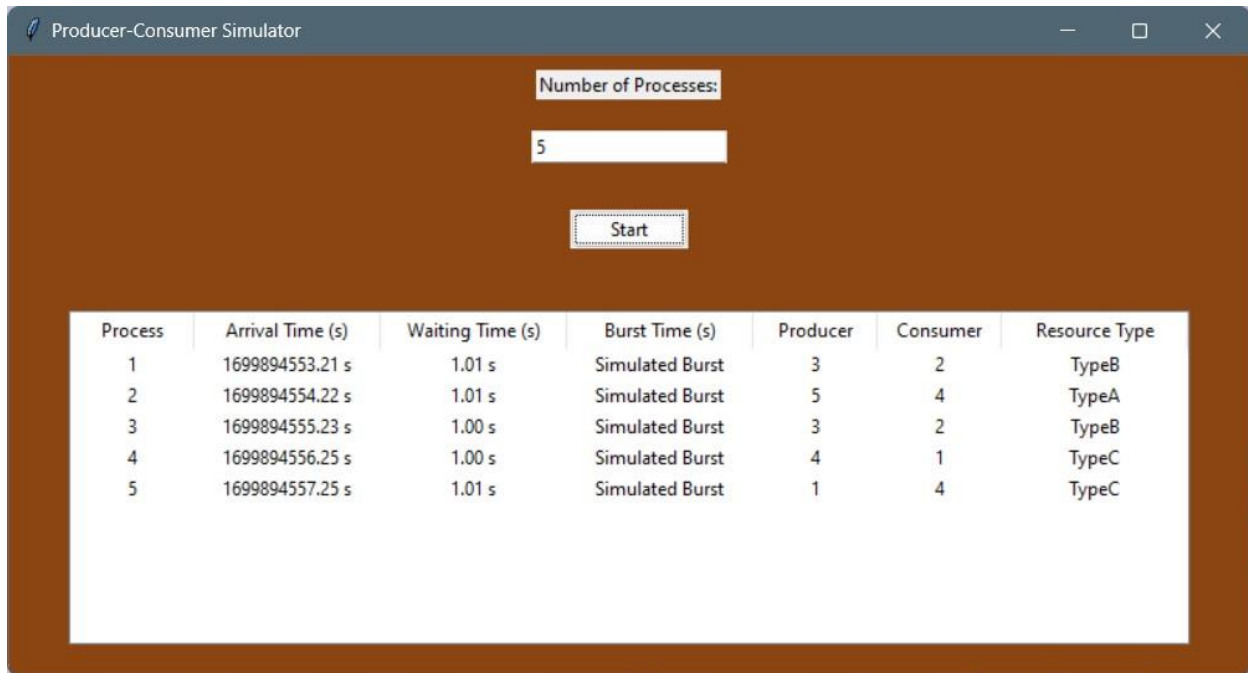
# RESULTS AND DISCUSSION

## Process Monitor with Graphical Representation:



The Process Monitor program is a Python Tkinter application that visually displays real-time information about the number of active processes on a system. The GUI includes a dynamic graph illustrating the fluctuating process count over time. Additionally, it reports the current number of processes and notifies users if the count decreases, providing insights into potential process terminations. The program utilizes the psutil library to gather system process data and runs a concurrent thread for real-time updates. This tool aids users in monitoring and understanding the dynamic behavior of processes on their system through a user-friendly graphical interface.

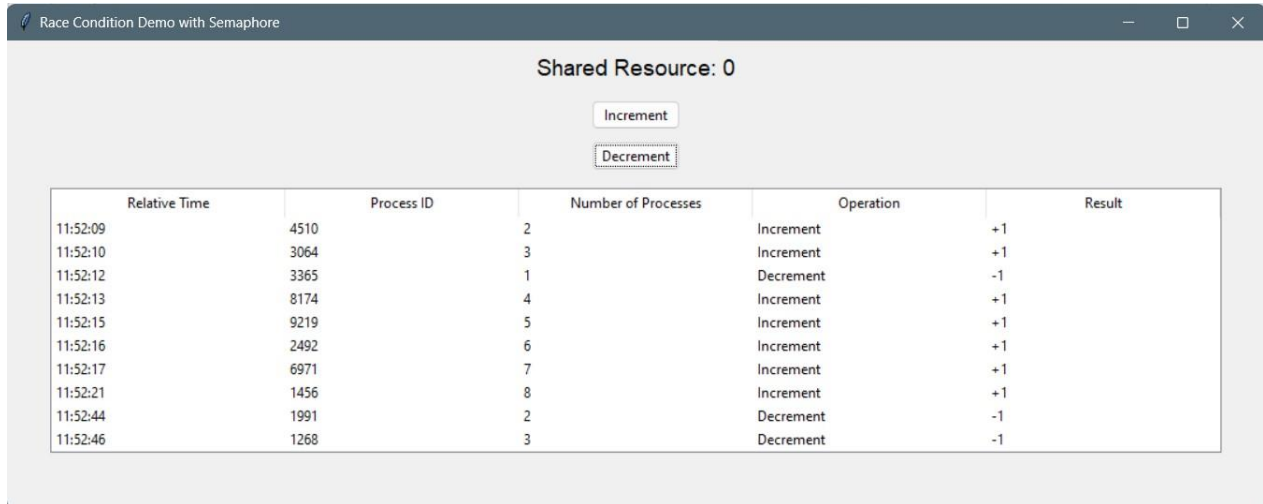
## **Producer Consumer Demonstration:**



Process	Arrival Time (s)	Waiting Time (s)	Burst Time (s)	Producer	Consumer	Resource Type
1	1699894553.21 s	1.01 s	Simulated Burst	3	2	TypeB
2	1699894554.22 s	1.01 s	Simulated Burst	5	4	TypeA
3	1699894555.23 s	1.00 s	Simulated Burst	3	2	TypeB
4	1699894556.25 s	1.00 s	Simulated Burst	4	1	TypeC
5	1699894557.25 s	1.01 s	Simulated Burst	1	4	TypeC

The Producer-Consumer Simulator is a Python Tkinter application modeling the classic synchronization problem. It simulates concurrent execution of producer and consumer processes, sharing resources in a virtual environment. The GUI allows users to input the number of simulated processes, initiating a dynamic simulation where each process has an arrival time, waiting time, and interacts with randomly assigned producers and consumers. The table dynamically updates, providing real-time information on process details, enhancing understanding of synchronization challenges in resource sharing. The program offers a practical illustration of essential concepts in concurrent programming through an interactive and informative graphical interface.

## Race Condition Demonstration:



Shared Resource: 0

Increment

Decrement

Relative Time	Process ID	Number of Processes	Operation	Result
11:52:09	4510	2	Increment	+1
11:52:10	3064	3	Increment	+1
11:52:12	3365	1	Decrement	-1
11:52:13	8174	4	Increment	+1
11:52:15	9219	5	Increment	+1
11:52:16	2492	6	Increment	+1
11:52:17	6971	7	Increment	+1
11:52:21	1456	8	Increment	+1
11:52:44	1991	2	Decrement	-1
11:52:46	1268	3	Decrement	-1

The "Race Condition Demo with Semaphore" program is a Tkinter-based Python application that illustrates the concept of race conditions and how they can be mitigated using semaphores. A race condition occurs when multiple processes attempt to access a shared resource simultaneously, leading to unpredictable and undesirable behavior. In this demonstration, the shared resource is a numerical value that can be incremented or decremented.

The program features a graphical user interface with buttons for incrementing and decrementing the shared resource. To address race conditions, a threading semaphore is employed, ensuring that only one process can access the shared resource at a time. This prevents conflicts and guarantees the orderly execution of operations.

The application also includes a table displaying real-time information about the processes, such as the relative time of execution, a unique process ID, the number of processes that have interacted with the resource, the type of operation (increment or decrement), and the result of the operation.

## CONCLUSION

In conclusion, the discussed concepts encompass crucial aspects of computer science and system design, addressing challenges related to concurrency, synchronization, and real-time visualization. Process synchronization is fundamental for ensuring orderly execution in multi-process or multi-threaded environments. It tackles issues such as race conditions and data inconsistencies, emphasizing mutual exclusion and the use of synchronization mechanisms like locks and semaphores.

The real-time process visualizer program provides a tangible representation of concurrent processes, offering insights into system behavior through a dynamic graphical interface. Utilizing Tkinter and psutil, enhances user understanding of real-time fluctuations in process counts, notifying of any reductions. This tool serves as a practical aid in monitoring and comprehending system dynamics.

Similarly, the Producer-Consumer Simulator offers hands-on experience in concurrent programming, specifically addressing resource-sharing challenges. With Tkinter, the program dynamically simulates processes with arrival and waiting times, engaging producers and consumers. The graphical table updates in real time, providing a visual representation of resource interactions. This simulation not only illustrates synchronization concepts but also emphasizes the practical complexities of coordinating processes in a shared environment.

Both programs contribute to a holistic understanding of these concepts. They serve as educational tools for students and professionals, allowing them to experiment with and visualize theoretical concepts in a practical context. The graphical interfaces enhance accessibility, aiding users in comprehending the intricate dynamics of concurrent systems.

## REFERENCES

- <https://www.researchgate.net/>
- <https://www.geeksforgeeks.org/>
- <https://www.youtube.com/>
- <https://www.guru99.com/>
- <https://www.knowledgehut.com/>
- <https://docs.python.org/3/library/tkinter.htm>
- <https://www.javatpoint.com/>
- <https://www.tutorialandexample.com/>