TITLE: -PROCESS VIZUVALIZATION TOOL

Teacher:- ANUDEEP GORAYA

SUBJECT:- CSE316

SECTION:-K23SG

1. STUDENT(S):

A. Full name	Gedala Mohan Rao
Email (personal)	Gedalamohanraolpu@gamil.com
UID/Registration number	12312147
Address of Internal Inventors	Lovely Professional University, Punjab-144411, India

2.STUDENT(S):

A. Full name	LOKIREDDY KONDA GURAVA REDDY
Email (personal)	lokireddyguravareddy93@gmail.com
UID/Registration number	12320620
Address of Internal Inventors	Lovely Professional University, Punjab-144411, India

3.STUDENT(S):

A. Full name	Polla Sai Tharun
Email (personal)	psaitharun23@gmail.com
UID/Registration number	12318347
Address of Internal Inventors	Lovely Professional University, Punjab-144411, India

GITHUB LINK :- https://github.com/mohanrao06/process-visualization-tool

Project Overview

Process Visualization Tool is a powerful educational environment developed with Streamlit for visualizing and simulating CPU scheduling algorithms interactively. The objective of the project is to present an intuitive and dynamic learning space where one can try various scheduling techniques and learn about their implications on process execution and resource usage in the system.

Operating systems depend upon scheduling algorithms for specifying the order in which the processes are given access to the CPU. It is important for students, teachers, and system-level programmers or operating system professionals to comprehend the dynamics of these algorithms. This software provides the facility to enter actual-world inputs, i.e., process arrival times, burst times, and priorities, and observe how various algorithms including First-Come-First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin perform dynamically.

Besides emulating scheduling algorithms, the tool has an integrated system monitoring dashboard showing real-time resource usage statistics. Users can view CPU utilization, memory usage, and disk activity during the execution of the scheduling algorithms, providing an all-around view of how system resources are utilized in response to process scheduling decisions. This integrated approach enables users to understand both the theoretical and practical sides of process administration.

The primary objective of this project is to give an interactive, hands-on visual aid for any learner interested in process scheduling and the behavior of an operating system to improve the learning process with the ability to provide real-time interaction and feedback.

Module-Wise Analysis

Module 1: -Visualization of Scheduling Algorithm

Purpose: The module emulates the process of scheduling processes as per user input and chosen algorithms. The module shows the sequence of process execution, with emphasis on the execution sequence, waiting time, turnaround time, and how CPU time is allocated by the system to each process.

Role: This module acts as the academic backbone of the project, providing users with a detailed grasp of how different scheduling algorithms schedule processes with different priorities. It illustrates ideas like waiting time, turnaround time, and process scheduling order, making it easier for users to visualize the fundamental concepts in scheduling theory.

Key Features:

Process Input Interface: Enables users to enter process information (e.g., arrival time, burst time, priority).

Gantt Chart Visualization: Shows process scheduling over time for the chosen algorithms.

Execution Metrics: Computes and shows execution metrics such as average waiting time, average turnaround time, and process completion order.

Module 2: System Monitoring Dashboard

Purpose: This module monitors and shows real-time system resource usage, such as CPU, memory, and disk usage, while the simulation is running. The objective is to see how system resources are utilized in real-time during the scheduling process.

Role: The module is supplemented by the visualization of process scheduling to demonstrate how various scheduling methods influence system performance. Through incorporation of real-time monitoring of resources, it indicates how scheduling would influence resource utilization, and if any algorithms are likely to introduce CPU or memory bottlenecks.

Key Features:

Real-Time Data Monitoring: Indicates CPU load, memory use, and disk usage.

Performance Insights: Informs about the influence of process scheduling on system resources.

Graphical Representation: Uses charts and graphs to visualize system usage dynamically.

Module 3: Interactive Features and Customization

Purpose: This module enables users to grasp process data and parameter experimentation, enabling them to alter the input process information and observe how the behavior of the algorithm evolves in real time. Users can experiment with varying scheduling algorithms, vary

process arrival times, burst times, and priorities, and view how these variations impact system performance and the scheduling result.

Role: The interactivity and customization capabilities enhance user participation, enabling them to try out different scenarios. The experiential nature of learning is reinforced by providing users with a direct observation of the result of varying input parameters on process scheduling and system resource utilization.

Key Features:

Dynamic Input: Facilitates users' ability to change process parameters in the course of simulation.

Algorithm Switching: Facilitates users' ability to switch from one algorithm to another in real-time and see comparable results.

Instant Feedback: Offers instant visualization updates according to user input changes.

3. Functionalities

Scheduling Algorithm Visualization:

The application receives process information (e.g., process ID, burst time, arrival time, priority) and emulates scheduling based on the selected algorithm.

A Gantt chart is drawn to graphically illustrate process execution with distinct labels indicating start and completion times.

For FCFS and SJF, the application computes waiting time, turnaround time, and contrasts performance measures such as average waiting time.

System Monitoring Dashboard:

The tool gives a dynamic representation of the system's actual resource usage in real time.

CPU Utilization: Tracks the amount of CPU time utilized during process scheduling.

Memory Usage: Tracks the memory utilized as processes are scheduled and run.

Disk Usage: Displays the disk space utilized during the simulation.

Interactive Customization:

Users can modify the arrival time, burst time, and priority of processes at any point during the simulation.

The software recalculates the schedule and re-displays the Gantt chart to reflect the impact of these modifications in real time.

Real-Time Performance Metrics:

The software refreshes the performance metrics and graphs each time the user makes changes to the input data, so that users always get the latest system status and process execution information.

4. Technology Used

Programming Languages:

Python: Python is utilized due to its ease of use, flexibility, and extensive support for libraries like Streamlit, Matplotlib, Psutil, Pandas, and Numpy. Python is suitable for developing learning tools with good data visualization features and managing intricate logic in an easy and readable way.

Libraries and Tools:

Streamlit : Streamlit streamlines interactive web application development. It is used to develop the user interface and display the actual-time simulation of process scheduling and system performance monitoring.

Matplotlib: It is used for producing different types of visualizations including Gantt charts, line plots, and bar plots to graphically show process scheduling and system resource utilization.

Psutil: This package gives real-time information about CPU, memory, and disk utilization and enables the tool to dynamically adjust these measures as the simulation proceeds.

Pandas: Processes process data (e.g., arrival time, burst time, priority) and computes necessary computations, such as waiting time and turnaround time per process for varied scheduling algorithms.

Numpy: Numpy is utilized for numerical computations involved in scheduling and process execution order to make the tool efficient even for big data.

Other Tools:

GitHub: GitHub is utilized for version control so that several team members can work on the codebase. It also facilitates issue tracking, pull requests, and access to the latest version of the project.

5. Flow Diagram

Here's the step-by-step flow of how the process works:

User Input:

The users feed the process information (e.g., Process ID, burst time, arrival time, priority) to the system.

Choose Scheduling Algorithm:

The user chooses one of the supported CPU scheduling algorithms.

Process Scheduling Computation:

The system computes the user input and simulates execution order based on the algorithm chosen.

Visualization:

A Gantt chart or any graphical representation (e.g., timeline) of the process schedule is derived to indicate the order of execution.

System Monitoring:

The system tracks CPU, memory, and disk utilization in real time during the simulation and graphs the same dynamically.

Update Display:

The Gantt chart and system performance indicators are updated in real time as per any changes in user input.

6. Revision Tracking on GitHub

- **Repository Name:** **process-visualization-tool**
- GitHub Link: https://github.com/mohanrao06/process-visualization-tool

7. Conclusion and Future Scope

Conclusion:

The Process Visualization Tool provides a novel method for learning and comprehending CPU scheduling algorithms and system resource usage. By simulating process execution in real time and monitoring the system, users can develop a better understanding of how operating systems execute processes and allocate resources. The tool is particularly useful for students learning operating systems, as well as for professionals looking for hands-on experience with scheduling algorithms.

Future Scope:

Integration of Sophisticated Scheduling Algorithms: Subsequent releases of this project might include sophisticated algorithms like Multilevel Queue Scheduling, Earliest Deadline First (EDF), Multilevel Feedback Queue Scheduling, and Lottery Scheduling, enabling better analysis of the different methods processes can be scheduled.

Improve Real-Time Monitoring: In later releases, the system can be equipped with more sophisticated performance monitoring features like disk I/O consumption, network throughput, and other resources whose consumption can be influenced by process scheduling.

Mobile and Cloud Support: The application can be made available for mobile devices so users can study on the move. Additionally, cloud support can facilitate users to model distributed systems with real-time scheduling visualizations on several nodes or machines.

Gamification and Learning Pathways: Adding gamification to the tool might encourage students to interact more deeply with the material. For instance, a "challenge mode" might allow students to compete with one another to optimize system performance on various scheduling algorithms.

8. References

- **GitHub Repository:** https://github.com/mohanrao06/process-visualization-tool
- **Streamlit Documentation:** https://docs.streamlit.io/
- **Matplotlib Documentation:** https://matplotlib.org/stable/contents.html
- **Psutil Documentation:** https://psutil.readthedocs.io/
- **Pandas Documentation:** https://pandas.pydata.org/
- **Numpy Documentation:** https://numpy.org/doc/

Problem Statement:-

PROCESS VIZUVALIZATION TOOL:-

The goal of this project is to create a **Process Visualization Tool** that simulates different CPU scheduling algorithms such as First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. The tool will use **Streamlit** for the frontend, allowing users to input process details and visualize scheduling results dynamically

CODE:-

app.py

```
import streamlit as st


# Set Page Configuration
st.set_page_config(
    page_title="Process Visualization Tool",
    layout="wide",
    initial_sidebar_state="collapsed"
)


# Custom CSS for modern styling
st.markdown("""
<style>
    /* Main container styling */
    .main {
        background: linear-gradient(135deg, #f8f9fa 0%, #eef2f5 100%);
```

```
}

/* Title styling */
.title {
    text-align: center;
    color: #2E86C1;
    margin-bottom: 0.5rem;
    font-size: 2.8rem;
    font-weight: 700;
    background: linear-gradient(to right, #2E86C1, #4AA8FF);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
}

/* Subtitle styling */
.subtitle {
    text-align: center;
    color: #555;
    font-size: 1.2rem;
    margin-bottom: 2.5rem;
}

/* Card styling */
.feature-card {
    border: none;
    padding: 25px;
    border-radius: 15px;
    margin-bottom: 25px;
    background: white;
    box-shadow: 0 6px 20px rgba(0, 0, 0, 0.08);
    transition: all 0.3s ease;
    height: 280px;
    display: flex;
    flex-direction: column;
```

```
    justify-content: space-between;
    color:black;
}

.feature-card:hover {
    transform: translateY(-5px);
    box-shadow: 0 12px 25px rgba(0, 0, 0, 0.12);
}

.card-title {
    color: #2E86C1;
    text-align: center;
    font-size: 1.5rem;
    font-weight: 600;
    margin-bottom: 1rem;
}

.card-desc {
    color: #555;
    text-align: center;
    font-size: 1rem;
    line-height: 1.5;
    margin-bottom: 1.5rem;
}

.card-button {
    background: linear-gradient(to right, #2E86C1, #4AA8FF);
    color: white;
    border: none;
    padding: 12px 24px;
    font-size: 1rem;
    border-radius: 8px;
    cursor: pointer;
    width: 100%;
```

```
        font-weight: 500;
        transition: all 0.3s ease;
        box-shadow: 0 4px 15px rgba(42, 134, 193, 0.3);
    }

    .card-button:hover {
        transform: translateY(-2px);
        box-shadow: 0 6px 20px rgba(42, 134, 193, 0.4);
    }

    /* Divider styling */
    .divider {
        border: 0;
        height: 1px;
        background: linear-gradient(to right, transparent, #2E86C1, transparent);
        margin: 2rem 0;
    }
</style>
""", unsafe_allow_html=True)

# Page Title
st.markdown("<h1 class='title'>Process Visualization Tool</h1>", unsafe_allow_html=True)
st.markdown("<p class='subtitle'>A modern tool to visualize CPU scheduling and system processes</p>", unsafe_allow_html=True)
st.markdown("<hr class='divider'>", unsafe_allow_html=True)

# Define pages with descriptions
pages = [
    {"title": "Real-time System Monitoring", "description": "View system processes and monitor resource usage in real-time with interactive dashboards.", "page": "Home"},
    {"title": "Process Scheduling", "description": "Simulate CPU scheduling algorithms like FCFS, SJF, RR, and Priority Scheduling with dynamic visualizations.", "page": "Scheduling"},
    {"title": "Analysis & Stats", "description": "View detailed Gantt charts, CPU utilization metrics, and comprehensive performance analysis.", "page": "Statistics"},
    {"title": "About Us", "description": "Browse and analyze past scheduling simulations with comparison tools.", "page": "About"},
```

```
]

# Display cards in a grid layout (2 columns, 2 rows)
cols = st.columns(2)  # 2 columns for better layout

# Loop through the pages and add each card in the grid layout
for i, page in enumerate(pages):
    with cols[i % 2]:  # Arrange in 2 columns
        st.markdown(
            f"""
            <div class="feature-card">
                <div>
                    <h2 class="card-title">{page['title']}</h2>
                    <p class="card-desc">{page['description']}</p>
                </div>
                <a href="{page['page']}" target="_self">
                    <button class="card-button">Go to {page['title']}</button>
                </a>
            </div>
            """,
            unsafe_allow_html=True
        )
```

Home.py

```
import streamlit as st
import pandas as pd
import time
import psutil  # Import psutil for system metrics
from utils.helpers import get_process_info  # Import process fetching function
st.set_page_config(
    page_title="Process Visualization Tool",
    layout="wide",
    initial_sidebar_state="collapsed"
```

```
)
# Set Page Config
st.set_page_config(page_title="Process Visualization Tool", layout="wide",
initial_sidebar_state="collapsed")

# Title and Description
st.title("□ Home - Process Visualization Tool")
st.write("Welcome to the Process Visualization Tool! □")
st.write("This page displays the currently running processes on your system.")

st.subheader("□ Running Processes")

# Initialize session state
if "df" not in st.session_state:
    st.session_state.df = get_process_info()  # Store DataFrame in session state

if "refresh_rate" not in st.session_state:
    st.session_state.refresh_rate = 1  # Default refresh rate (1 second)

# New Feature: System Overview (Process Count, CPU, Memory & Disk Usage)
st.markdown("### □ System Overview")

col1, col2, col3, col4 = st.columns(4)
with col1:
    st.metric(label="□ Total Processes", value=len(st.session_state.df))

with col2:
    cpu_usage = psutil.cpu_percent(interval=1)
    st.metric(label="□ CPU Usage (%)", value=f"{cpu_usage}%")

with col3:
    memory_usage = psutil.virtual_memory().percent
    st.metric(label="□ Memory Usage (%)", value=f"{memory_usage}%")
```

```
with col4:
    disk_usage = psutil.disk_usage("/").percent
    st.metric(label="□ Disk Usage (%)", value=f"{disk_usage}%")

# New Feature: Top 5 CPU-Consuming Processes
st.markdown("### □ Top 5 CPU-Consuming Processes")
processes = []
for proc in psutil.process_iter(attrs=["pid", "name", "cpu_percent"]):
    processes.append(proc.info)

df_top_cpu = pd.DataFrame(processes).sort_values(by="cpu_percent",
ascending=False).head(5)
st.dataframe(df_top_cpu, use_container_width=True)


# Display all running processes
st.subheader("□ All Running Processes")
st.dataframe(st.session_state.df, use_container_width=True)

# Auto-refresh logic (without user controls)
time.sleep(st.session_state.refresh_rate)
st.session_state.df = get_process_info()  # Update only data
st.rerun()  # Refresh without blinking
```

Schedling.py

```
import streamlit as st
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
from algorithms.fcfs import fcfs_scheduling, generate_gantt_chart as generate_fcfs_gantt
from algorithms.sjf import sjf_scheduling, generate_gantt_chart as generate_sjf_gantt
```

```
from algorithms.priority import priority_scheduling, generate_gantt_chart as generate_priority_gantt
from algorithms.round_robin import round_robin_scheduling, generate_gantt_chart as generate_rr_gantt


st.set_page_config(
    page_title="Process Visualization Tool",
    layout="wide",
    initial_sidebar_state="collapsed"
)
# Streamlit UI
st.title("Process Scheduling Visualization")

# Dropdown for selecting scheduling algorithm
algorithm = st.selectbox("Select Scheduling Algorithm",
                         ["FCFS", "SJF", "Round Robin", "Priority Scheduling"])

# Algorithm-specific options
if algorithm == "SJF":
    sjf_mode = st.radio("SJF Mode", ["Non-Preemptive", "Preemptive"])
elif algorithm == "Priority Scheduling":
    priority_mode = st.radio("Priority Mode", ["Non-Preemptive", "Preemptive"])

num_processes = st.number_input("Number of processes", min_value=1, value=3, step=1)

# Collect Process Data
process_data = []
for i in range(num_processes):
    pid = f'P{i+1}'
    col1, col2 = st.columns(2)
    with col1:
        arrival = st.number_input(f'Arrival Time of {pid}', min_value=0, value=i)
    with col2:
        burst = st.number_input(f'Burst Time of {pid}', min_value=1, value=i+2)
```

```
    process_data.append((pid, arrival, burst))

# Additional inputs based on selected algorithm
if algorithm == "Round Robin":
    time_quantum = st.number_input("Time Quantum", min_value=1, value=2)
elif algorithm == "Priority Scheduling":
    priorities = []
    for i in range(num_processes):
        priority = st.number_input(f"Priority of P{i+1} (lower=higher priority)",
                                   min_value=1, value=i+1)
        priorities.append(priority)

# Compute Scheduling on Button Click
if st.button("Compute Scheduling"):
    if not process_data:
        st.error("No processes entered! Please add at least one process.")
    else:
        try:
            if algorithm == "FCFS":
                start, completion, turnaround, waiting = fcfs_scheduling(process_data)
                fig = generate_fcfs_gantt(pd.DataFrame({
                    "Process": [p[0] for p in process_data],
                    "Start Time": start,
                    "Burst Time": [p[2] for p in process_data]
                }))
            elif algorithm == "SJF":
                preemptive = sjf_mode == "Preemptive"
                start, completion, turnaround, waiting, gantt = sjf_scheduling(process_data,
preemptive)
                fig = generate_sjf_gantt(gantt)
            elif algorithm == "Round Robin":
                start, completion, turnaround, waiting, gantt = round_robin_scheduling(process_data,
time_quantum)
                fig = generate_rr_gantt(gantt)
```

```
    elif algorithm == "Priority Scheduling":
        preemptive = priority_mode == "Preemptive"
        start, completion, turnaround, waiting, gantt = priority_scheduling(process_data, priorities, preemptive)
        fig = generate_priority_gantt(gantt)

    # Create DataFrame for Display
    df = pd.DataFrame({
        "Process": [p[0] for p in process_data],
        "Arrival Time": [p[1] for p in process_data],
        "Burst Time": [p[2] for p in process_data],
        "Start Time": start,
        "Completion Time": completion,
        "Turnaround Time": turnaround,
        "Waiting Time": waiting
    })

    # Add priority column if applicable
    if algorithm == "Priority Scheduling":
        df["Priority"] = priorities

    # Display DataFrame in Streamlit
    st.write("### Scheduling Table")
    st.dataframe(df)

    # Generate and Show Gantt Chart
    st.write("### Gantt Chart")
    st.pyplot(fig)

    # Calculate and display averages
    avg_turnaround = sum(turnaround) / len(turnaround)
    avg_waiting = sum(waiting) / len(waiting)
```

```
    st.write("### Performance Metrics")
    col1, col2 = st.columns(2)
    col1.metric("Average Turnaround Time", f"{avg_turnaround:.2f}")
    col2.metric("Average Waiting Time", f"{avg_waiting:.2f}")
    # In the Compute Scheduling section, update the history entry creation:
    # In the Compute Scheduling section, add throughput calculation:
    throughput = len(process_data) / max(completion) if max(completion) > 0 else 0

    # Update the history_entry to include throughput:
    history_entry = {
        'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M"),
        'algorithm': algorithm,
        'num_processes': num_processes,
        'results': {
            'avg_waiting': round(avg_waiting, 2),
            'avg_turnaround': round(avg_turnaround, 2),
            'throughput': round(throughput, 2)
        },
        'process_details': df.to_dict('records')
    }

    if 'simulation_history' not in st.session_state:
        st.session_state.simulation_history = []
    st.session_state.simulation_history.append(history_entry)

    # # Display results
    # st.subheader("Results")
    # st.dataframe(pd.DataFrame(result['table']))

    # st.subheader("Gantt Chart")
    # st.image(result['gantt'])

except Exception as e:
```

```
            st.error(f"Error computing scheduling: {e}")
            # After running a simulation in your Scheduling.py:
```

Statistics.py

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
st.set_page_config(
    page_title="Process Visualization Tool",
    layout="wide",
    initial_sidebar_state="collapsed"
)
def main():
    st.title("□ Performance Analysis")

    # Check for simulation history
    if 'simulation_history' not in st.session_state or not st.session_state.simulation_history:
        st.warning("No simulation data available. Run some scheduling simulations first.")
        return

    # Convert history to DataFrame
    history_data = []
    for entry in st.session_state.simulation_history:
        history_data.append({
            'timestamp': entry['timestamp'],
            'algorithm': entry['algorithm'],
            'avg_waiting': entry['results']['avg_waiting'],
            'avg_turnaround': entry['results']['avg_turnaround'],
            'throughput': entry['results'].get('throughput', 0),  # Handle cases where throughput might
not exist
            'num_processes': entry['num_processes']
        })
```

```
df = pd.DataFrame(history_data)

# Metrics overview
st.subheader("Performance Metrics")
col1, col2, col3 = st.columns(3)
col1.metric("Best Avg Waiting Time", f"{df['avg_waiting'].min():.2f}")
col2.metric("Best Avg Turnaround", f"{df['avg_turnaround'].min():.2f}")
col3.metric("Highest Throughput", f"{df['throughput'].max():.2f}")

# Visualization
st.subheader("Trend Analysis")

tab1, tab2, tab3 = st.tabs(["Waiting Time", "Turnaround Time", "Throughput"])

with tab1:
    fig, ax = plt.subplots()
    ax.bar(df['algorithm'], df['avg_waiting'])
    ax.set_title("Average Waiting Time Comparison")
    ax.set_ylabel("Time Units")
    st.pyplot(fig)

with tab2:
    fig, ax = plt.subplots()
    ax.plot(df['algorithm'], df['avg_turnaround'], marker='o')
    ax.set_title("Turnaround Time Trend")
    ax.set_ylabel("Time Units")
    st.pyplot(fig)

with tab3:
    fig, ax = plt.subplots()
    ax.barh(df['algorithm'], df['throughput'])
    ax.set_title("Throughput Comparison")
```

```
        ax.set_xlabel("Processes per Time Unit")
        st.pyplot(fig)

    # Raw data
    st.subheader("Simulation History Data")
    st.dataframe(df)

if __name__ == "__main__":
    main()
```

About.py

```
import streamlit as st
from datetime import datetime
st.set_page_config(
    page_title="Process Visualization Tool",
    layout="wide",
    initial_sidebar_state="collapsed"
)
def main():
    st.title("About This Project")

    st.image("https://via.placeholder.com/800x200?text=Process+Scheduling+Visualizer",
             use_container_width=True)

    st.header("□ Our Mission")
    st.markdown("""
    This tool was created to help students and professionals understand CPU scheduling algorithms
    through interactive visualization and real-time system monitoring.
    """)

    with st.expander("□ Background Information"):
        st.markdown("""
```

```
    ### What are CPU Scheduling Algorithms?
    CPU scheduling is a process that allows one process to use the CPU while
    the execution of another process is on hold due to unavailability of any
    resource like I/O etc.

    ### Why Visualization Matters
    Visualizing these algorithms helps in:
    - Understanding the theoretical concepts
    - Comparing algorithm performance
    - Seeing real-world system impacts
    """)

st.header("□□ Development Team")

col1, col2, col3 = st.columns(3)

with col1:
    st.image("https://via.placeholder.com/150?text=Team+Member", width=100)
    st.subheader("Your Name")
    st.markdown("""
    - Lead Developer
    - System Architecture
    - [GitHub](https://github.com/you)
    """)

with col2:
    st.image("https://via.placeholder.com/150?text=Team+Member", width=100)
    st.subheader("Team Member")
    st.markdown("""
    - UI/UX Design
    - Documentation
    - [LinkedIn](#)
    """)
```

```
    with col3:
        st.image("https://via.placeholder.com/150?text=Team+Member", width=100)
        st.subheader("Advisor")
        st.markdown("""
        - Project Mentor
        - Algorithm Expert
        - [University](#)
        """)

    st.header("□ Resources")
    st.markdown("""
    - **Source Code**: [GitHub Repository](https://github.com/your-repo)
    - **Documentation**: [ReadTheDocs](#)
    - **Report Issues**: [Issue Tracker](#)
    """)

    st.header("□ Technologies Used")
    st.markdown("""
    - **Python** (Streamlit, Pandas, Matplotlib)
    - **System Monitoring** (psutil)
    - **Visualization** (Plotly, Altair)
    - **Deployment** (Docker, Heroku)
    """)

    st.markdown("---")
    st.markdown("""
    *Built with □ for Operating Systems students and professionals*
    *Version 1.0 | Last Updated: {date}*
    """.format(date=datetime.now().strftime("%B %Y")))

if __name__ == "__main__":
    main()
```

Algorithms

Fcfs.py

```
import matplotlib.pyplot as plt


def fcfs_scheduling(processes):
    """
    Implements First-Come, First-Served (FCFS) Scheduling Algorithm.
    Returns: Start Time, Completion Time, Turnaround Time, Waiting Time as lists.
    """
    processes.sort(key=lambda x: x[1])  # Sort by Arrival Time
    start_time, completion_time, turnaround_time, waiting_time = [], [], [], []

    current_time = 0
    for pid, arrival, burst in processes:
        if current_time < arrival:
            current_time = arrival
        start_time.append(current_time)
        completion = current_time + burst
        completion_time.append(completion)
        turnaround_time.append(completion - arrival)
        waiting_time.append(current_time - arrival)
        current_time = completion

    return start_time, completion_time, turnaround_time, waiting_time  # □ Return only 4 lists


def generate_gantt_chart(df, bar_width=0.5):  # □ Add bar_width parameter
    fig, ax = plt.subplots(figsize=(10, 4))
    for i, (pid, st_time, bt) in enumerate(zip(df["Process"], df["Start Time"], df["Burst Time"])):
        ax.barh(y=pid, width=bt, left=st_time, height=bar_width, align="center")  # □ Use bar_width
        ax.text(st_time + bt / 2, i, f"{pid}", ha="center", va="center", color="white", fontsize=12)

    ax.set_xlabel("Time")
```

```
    ax.set_ylabel("Processes")
    ax.set_title("Gantt Chart for FCFS Scheduling")
    ax.grid(axis="x", linestyle="--")

    return fig
```

Sjf.py

```
import matplotlib.pyplot as plt

def sjf_scheduling(processes, preemptive=False):
    """
    Implements Shortest Job First (SJF) Scheduling Algorithm.
    Args:
        processes: List of tuples (pid, arrival_time, burst_time)
        preemptive: Boolean for preemptive (SJF) or non-preemptive (SJN) version
    Returns: Start Time, Completion Time, Turnaround Time, Waiting Time as lists.
    """
    processes = sorted(processes, key=lambda x: x[0])  # Sort by PID for consistent order
    n = len(processes)
    burst_remaining = [p[2] for p in processes]
    start_time = [-1] * n
    completion_time = [0] * n
    turnaround_time = [0] * n
    waiting_time = [0] * n
    current_time = 0
    completed = 0
    gantt = []

    while completed != n:
        # Find available processes
        available = []
        for i in range(n):
            if processes[i][1] <= current_time and burst_remaining[i] > 0:
```

```
                available.append((burst_remaining[i], i))

        if not available:
            current_time += 1
            continue

        if preemptive:
            # Preemptive - choose shortest remaining time
            burst, idx = min(available)
        else:
            # Non-preemptive - choose shortest job
            available.sort(key=lambda x: x[0])
            burst, idx = available[0]

        if start_time[idx] == -1:
            start_time[idx] = current_time

        # Execute for 1 time unit (preemptive) or until completion (non-preemptive)
        exec_time = 1 if preemptive else burst
        burst_remaining[idx] -= exec_time
        gantt.append((processes[idx][0], current_time, current_time + exec_time))
        current_time += exec_time

        if burst_remaining[idx] == 0:
            completed += 1
            completion_time[idx] = current_time
            turnaround_time[idx] = completion_time[idx] - processes[idx][1]
            waiting_time[idx] = turnaround_time[idx] - processes[idx][2]

    return start_time, completion_time, turnaround_time, waiting_time, gantt

def generate_gantt_chart(gantt_data, bar_width=0.5):
    """Generate Gantt chart from scheduling data"""
```

```
    fig, ax = plt.subplots(figsize=(10, 4))

    # Create a mapping from process to y-position
    processes = sorted({p[0] for p in gantt_data})
    y_pos = {p: i for i, p in enumerate(processes)}

    for pid, start, end in gantt_data:
        duration = end - start
        ax.barh(y=y_pos[pid], width=duration, left=start, height=bar_width, align='center')
        ax.text(start + duration/2, y_pos[pid], f"{pid}", ha='center', va='center', color='white',
fontsize=10)

    ax.set_yticks(range(len(processes)))
    ax.set_yticklabels(processes)
    ax.set_xlabel("Time")
    ax.set_ylabel("Processes")
    ax.set_title("Gantt Chart for SJF Scheduling")
    ax.grid(axis='x', linestyle='--')

    return fig
```

Round_robin.py

```
import matplotlib.pyplot as plt
from collections import deque

def round_robin_scheduling(processes, time_quantum):
    """
    Implements Round Robin Scheduling Algorithm.
    Args:
        processes: List of tuples (pid, arrival_time, burst_time)
        time_quantum: Time quantum for RR scheduling
    Returns: Start Time, Completion Time, Turnaround Time, Waiting Time, Gantt data
    """
```

```
processes = sorted(processes, key=lambda x: x[0])  # Sort by PID for consistent order
n = len(processes)
burst_remaining = [p[2] for p in processes]
start_time = [-1] * n
completion_time = [0] * n
turnaround_time = [0] * n
waiting_time = [0] * n
current_time = 0
gantt = []

# Create a queue for ready processes
queue = deque()
completed = 0
arrived = [False] * n

while completed != n:
    # Add newly arrived processes to queue
    for i in range(n):
        if not arrived[i] and processes[i][1] <= current_time and burst_remaining[i] > 0:
            arrived[i] = True
            queue.append(i)

    if not queue:
        current_time += 1
        continue

    # Get next process from queue
    idx = queue.popleft()

    if start_time[idx] == -1:
        start_time[idx] = current_time

    # Execute for time quantum or until completion
```

```
            exec_time = min(time_quantum, burst_remaining[idx])
            burst_remaining[idx] -= exec_time
            gantt.append((processes[idx][0], current_time, current_time + exec_time))
            current_time += exec_time

            # Check if process completed
            if burst_remaining[idx] == 0:
                completed += 1
                completion_time[idx] = current_time
                turnaround_time[idx] = completion_time[idx] - processes[idx][1]
                waiting_time[idx] = turnaround_time[idx] - processes[idx][2]
            else:
                # Re-add to queue if not completed
                queue.append(idx)

    return start_time, completion_time, turnaround_time, waiting_time, gantt

def generate_gantt_chart(gantt_data, bar_width=0.5):
    """Generate Gantt chart from scheduling data"""
    fig, ax = plt.subplots(figsize=(10, 4))

    # Create a mapping from process to y-position
    processes = sorted({p[0] for p in gantt_data})
    y_pos = {p: i for i, p in enumerate(processes)}

    for pid, start, end in gantt_data:
        duration = end - start
        ax.barh(y=y_pos[pid], width=duration, left=start, height=bar_width, align='center')
        ax.text(start + duration/2, y_pos[pid], f"{pid}", ha='center', va='center', color='white',
fontsize=10)

    ax.set_yticks(range(len(processes)))
    ax.set_yticklabels(processes)
```

```
    ax.set_xlabel("Time")
    ax.set_ylabel("Processes")
    ax.set_title("Gantt Chart for Round Robin Scheduling")
    ax.grid(axis='x', linestyle='--')

    return fig
```

priority.py

```
import matplotlib.pyplot as plt

def priority_scheduling(processes, priorities, preemptive=False):
    """
    Implements Priority Scheduling Algorithm.
    Args:
        processes: List of tuples (pid, arrival_time, burst_time)
        priorities: List of priority values (lower number = higher priority)
        preemptive: Boolean for preemptive or non-preemptive version
    Returns: Start Time, Completion Time, Turnaround Time, Waiting Time, Gantt data
    """
    processes = sorted(processes, key=lambda x: x[0])  # Sort by PID for consistent order
    n = len(processes)
    burst_remaining = [p[2] for p in processes]
    start_time = [-1] * n
    completion_time = [0] * n
    turnaround_time = [0] * n
    waiting_time = [0] * n
    current_time = 0
    completed = 0
    gantt = []

    while completed != n:
        # Find available processes
```

```
available = []
for i in range(n):
    if processes[i][1] <= current_time and burst_remaining[i] > 0:
        available.append((priorities[i], i, burst_remaining[i]))

if not available:
    current_time += 1
    continue

if preemptive:
    # Preemptive - choose highest priority (lowest number)
    available.sort()
    priority, idx, burst = available[0]
else:
    # Non-preemptive - choose highest priority among arrived processes
    available.sort()
    priority, idx, burst = available[0]

if start_time[idx] == -1:
    start_time[idx] = current_time

# Execute for 1 time unit (preemptive) or until completion (non-preemptive)
exec_time = 1 if preemptive else burst
burst_remaining[idx] -= exec_time
gantt.append((processes[idx][0], current_time, current_time + exec_time))
current_time += exec_time

if burst_remaining[idx] == 0:
    completed += 1
    completion_time[idx] = current_time
    turnaround_time[idx] = completion_time[idx] - processes[idx][1]
    waiting_time[idx] = turnaround_time[idx] - processes[idx][2]
```

```
    return start_time, completion_time, turnaround_time, waiting_time, gantt


def generate_gantt_chart(gantt_data, bar_width=0.5):
    """Generate Gantt chart from scheduling data"""
    fig, ax = plt.subplots(figsize=(10, 4))

    # Create a mapping from process to y-position
    processes = sorted({p[0] for p in gantt_data})
    y_pos = {p: i for i, p in enumerate(processes)}

    for pid, start, end in gantt_data:
        duration = end - start
        ax.barh(y=y_pos[pid], width=duration, left=start, height=bar_width, align='center')
        ax.text(start + duration/2, y_pos[pid], f"{pid}", ha='center', va='center', color='white',
fontsize=10)

    ax.set_yticks(range(len(processes)))
    ax.set_yticklabels(processes)
    ax.set_xlabel("Time")
    ax.set_ylabel("Processes")
    ax.set_title("Gantt Chart for Priority Scheduling")
    ax.grid(axis='x', linestyle='--')

    return fig
```

THANK YOU

THE END