Week 11: Multitasking in a Single-Threaded Environment 🧵

Objectives:

- Understand the limitations of single-threaded environments 🧐.
- Learn how to implement multitasking using event loops 🔄.
- Avoid blocking functions like `pause` to keep applications responsive 🚫⏸️.
- Discuss multitasking, multithreading, and hyperthreading concepts 💡.

Topics We'll Cover:

1. Introduction to Single-Threaded Programming 🖥️
2. Multitasking Concepts 🧠
3. The Event Loop Concept 🔄
4. Blocking vs. Non-Blocking Code 🚫⏸️
5. Practical Examples in MATLAB 💻
6. Best Practices 📝
7. Advanced Concepts 🌟

Introduction to Single-Threaded Programming 🖥️

Single-Threaded Constraints:

- Only one task executes at a time.
- Need to manage multiple tasks efficiently.
- Real-time responsiveness is critical.

Why It Matters: Understanding these constraints helps us design better applications that remain responsive and efficient.

Understanding Multitasking in Single-Threaded Environments 🧠

What is Multitasking?

- *Simulated Concurrency:* Switching between tasks rapidly to appear simultaneous.
- *Task Interleaving:* Executing parts of multiple tasks in turns.

Limitations:

- No true parallelism—only one instruction executes at any moment.
- Risk of blocking—long-running tasks can prevent others from executing.

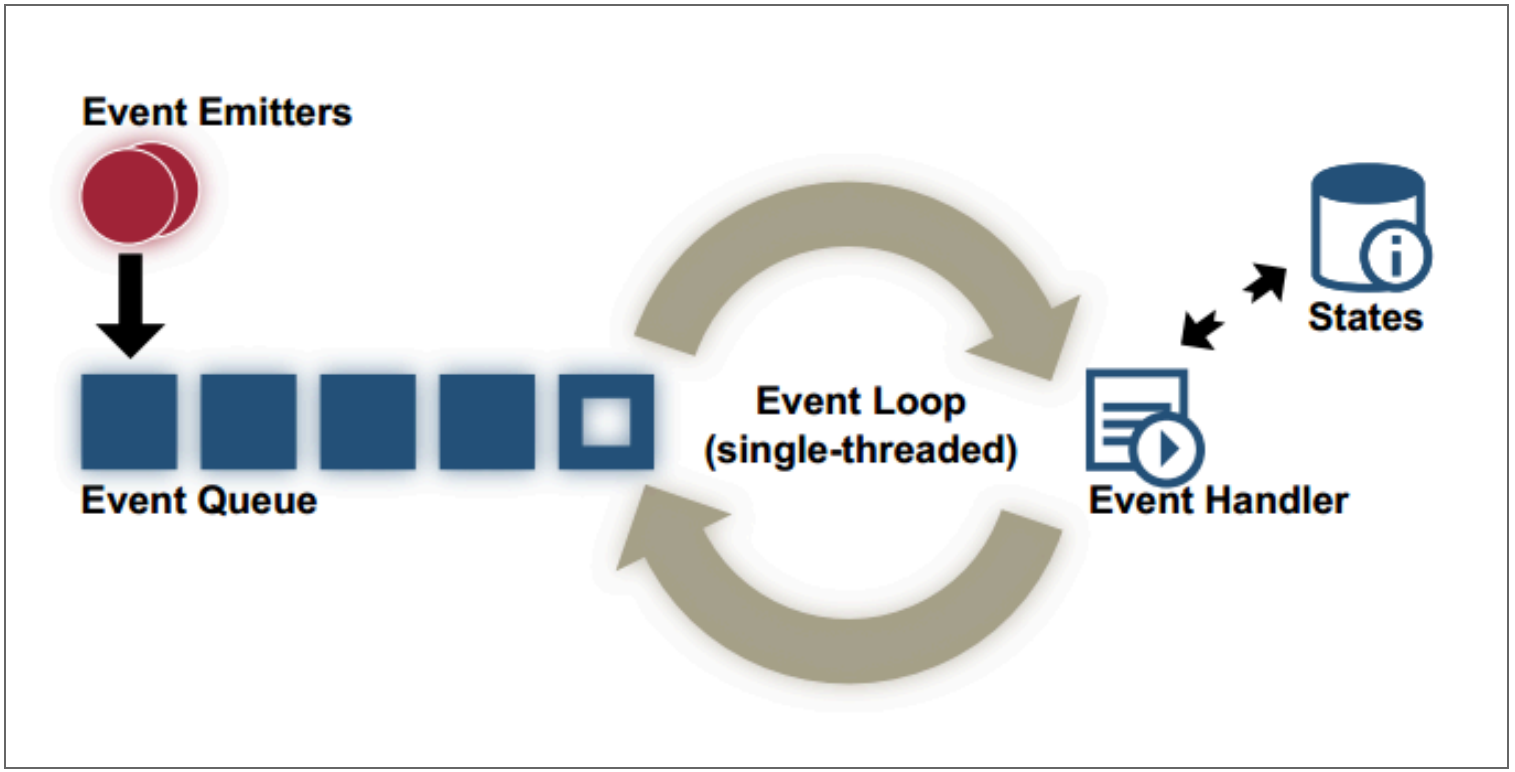
Question: How can we handle multiple tasks efficiently in a single-threaded environment?

The Event Loop Concept 🔄

What is an Event Loop?

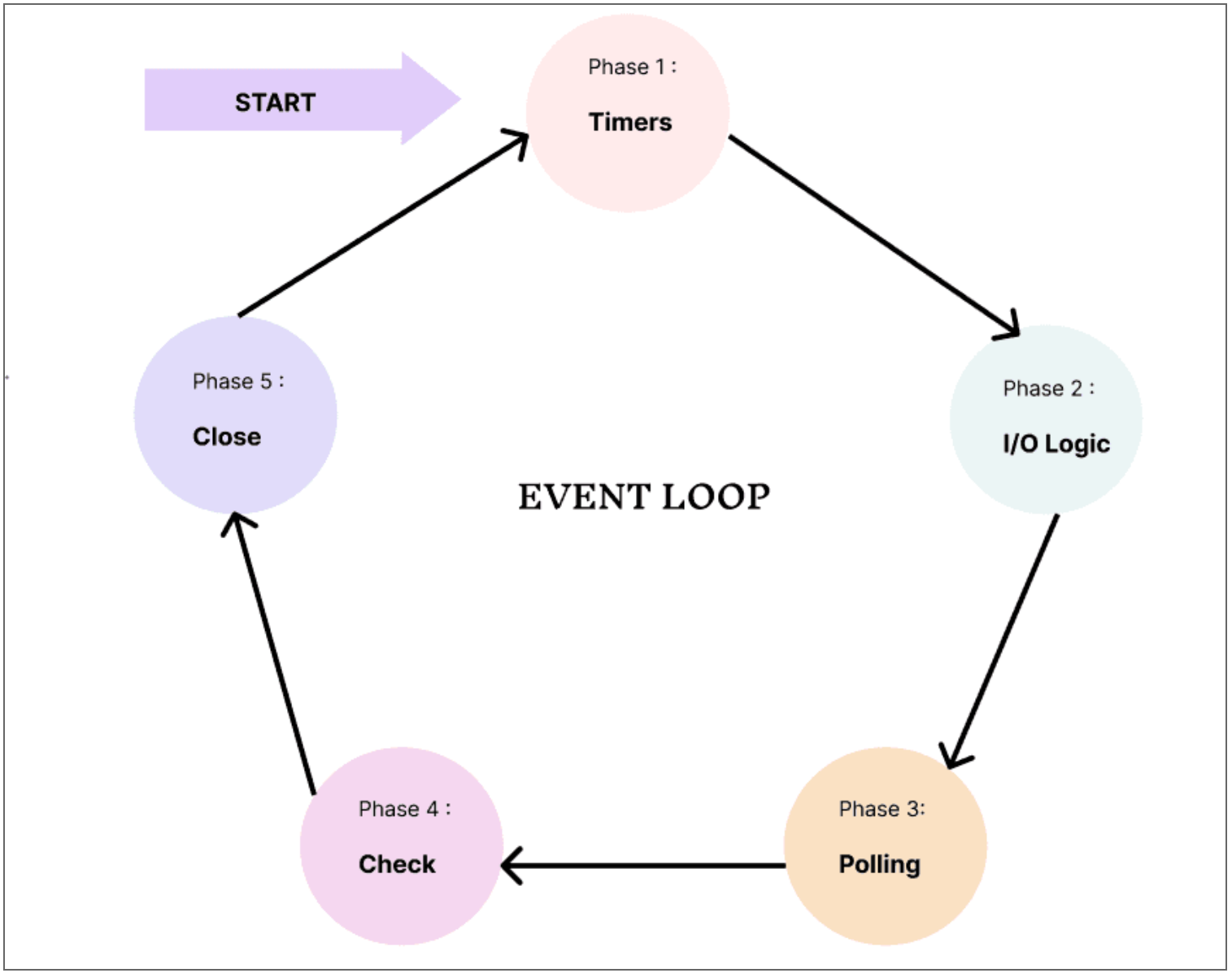
- A continuous loop that checks for and handles events or tasks.
- Keeps the application responsive by not blocking execution.

Analogy: An event loop is like a chef in a busy kitchen, juggling multiple dishes by attending to each one in quick succession, ensuring everything cooks perfectly without burning anything.


Event Emitters
States
Event Loop
(single-threaded)
Event Queue
Event Handler

The Event Loop Concept 🔄

- Can also simply poll the same tasks/events


START
Phase 1 :
Timers
Phase 2 :
I/O Logic
Phase 3:
Polling
Phase 4 :
Check
Phase 5 :
Close
EVENT LOOP

Blocking vs. Non-Blocking Code 🚫⏸️

Blocking Functions

- **Definition:** Functions that halt program execution until an operation completes.
- **Examples:** `pause`, `wait`, or any long-running computation.

Example of Blocking Code:

```
pause(5); % Program stops here for 5 seconds
```

Why Blocking is Problematic

- Other tasks can't run during the block.
- Unresponsive applications lead to poor user experience.
- Critical events may be missed.

Remember: In single-threaded environments, blocking functions halt everything!

Non-Blocking Alternatives

- Use elapsed time checks to decide when to execute tasks.
- Implement state machines to manage task progress.
- Regularly poll for conditions without stopping execution.

Example of Non-Blocking Code:

```
if (toc(startTime) >= interval)
    % Perform task
end
```

Practical Examples in MATLAB 💻

Example 1: Blinking an LED with `pause` 🚨

Problem Statement: Blink an LED using `pause` for delays.

Code Snippet:

```
function blinkLed(app)
    app.led.Enable = true;
    pause(1);
    app.led.Enable = false;
    pause(1);
end
```

Issue: The `pause` function blocks execution, preventing other tasks from running.

Consequence: The program becomes unresponsive during the pause.

Example 2: Blinking an LED Without `pause` ⏱️

Solution: Use elapsed time to schedule tasks without blocking.

Improved Code Snippet (using static variables):

```
function toggleLED(app, currentTime)
    persistent lastToggleTime;
    if isempty(lastToggleTime)
        lastToggleTime = currentTime;
    end

    % If duration is met, THEN execute task
end
```

Advantages:

- The main loop continues to run.
- Other tasks can execute without delay.

- The program remains responsive performing other tasks very quickly.

Scaling Up: Managing Multiple Tasks 📈

Objective: Create a GUI where different components update at different intervals.

Implementation Strategy:

- Assign each task a `lastRunTime`.
- Define intervals for each task.
- Check all tasks within the main loop.

Sample Loop:

```
while true
    elapsedTime = toc(startTime);
    updateTimer(app, elapsedTime);
    toggleLED1(app, elapsedTime);
    readSensor(app, elapsedTime);
```

```
    % Add more tasks as needed
end
```

Best Practices in Single-Threaded Multitasking 

Avoid Blocking Operations

- Do not use `pause`, `wait`, or long computations in tasks.
- Ensure tasks complete quickly.

Efficient Task Design

- Keep tasks short and efficient.
- Use modular code; separate tasks into functions.

State Management

- Use `persistent` variables to maintain task states.
- Consider using data structures for managing multiple tasks.

Common Pitfall to Avoid ⚠️

Can you identify what's wrong with the following code snippet?

```
function toggleLED(app, currentTime)
    if (currentTime - lastToggleTime) >= 1
        app.led.Enable = ~app.led.Enable;
        lastToggleTime = currentTime;
    end
end
```

Tips and Tricks 💡

- **Use the Debugger:** Step through your code to find issues.
- **Output Statements:** Use `disp` and `fprintf` to monitor variables.
- **Timing Functions:** Use `tic` and `toc` to measure execution time.
- **Collaborate:** Discuss your code with peers for other solutions.

Key Takeaways 🎓

- Event loops are essential for multitasking in single-threaded environments.
- Avoid blocking functions to maintain responsiveness.
- Time-based scheduling effectively manages task execution.
- State management is critical for tracking task progress.
- These concepts are directly applicable to microcontroller programming.

"*Efficient multitasking is about smart scheduling, not about doing everything at once.*"

Speaker notes