RTOS Scheduling: Round Robin Algorithm

MOHEMMEDADNAN DADU, HITARTH CHUDASAMA
Electronics & Communication Department
Institute of Technology, Nirma University
SG Highway, Ahmedabad, Gujarat, India
22bec020@nirmauni.ac.in
22bec019@nirmauni.ac.in

Abstract— Real-Time Operating Systems (RTOS) require efficient task scheduling mechanisms to manage multiple concurrent tasks. One of the most commonly used scheduling techniques is Round Robin Scheduling, which ensures fair CPU allocation among tasks using a fixed time slice. This report discusses the Round Robin algorithm with an embedded systems perspective, referring to an Arduino STM32 implementation where multiple LED blinking tasks are scheduled. The study explores the mechanism, practical implementation, applications, advantages, limitations, and simulation results of Round Robin Scheduling in RTOS.

Keywords— RTOS, Round Robin Scheduling, Task Management, Time-Slicing, Multitasking, Embedded Systems, Real-Time Applications, CPU Scheduling

I.INTRODUCTION:-

A Real-Time Operating System (RTOS) is a specialized operating system designed to handle tasks within strict time constraints. In many applications, especially in embedded systems, multiple tasks must be managed efficiently to ensure smooth operation. Common examples include sensor readings, motor controls, network communications, and user interface updates, all of which require well-defined scheduling mechanisms. Without proper scheduling, certain tasks may suffer from excessive delays, leading to performance degradation, missed deadlines, or even system failures.

One of the most widely used CPU scheduling techniques in RTOS is Round Robin Scheduling, which ensures fair execution of tasks by assigning each one a fixed time slice. Unlike other scheduling methods such as First-Come, First-Served (FCFS) or Priority Scheduling, the Round Robin approach prevents task starvation and guarantees that every task gets a fair share of CPU execution time. It is particularly useful in time-sharing systems, where multiple processes run simultaneously without affecting system responsiveness.

In embedded systems, efficient task scheduling is essential for balancing power consumption, execution time, and resource utilization. The Round Robin approach is frequently used in microcontroller-based systems, IoT applications, and automotive systems, where deterministic execution is crucial. It allows multiple tasks to execute sequentially without priority bias, making it ideal for applications requiring uniform task execution and responsiveness.

This report explores Round Robin Scheduling with an embedded system perspective, demonstrating its practical implementation on an STM32 microcontroller using the Arduino IDE. We implement a task scheduler to manage three LED blinking tasks, each executing at different intervals while ensuring fair CPU allocation. The scheduler cycles through each task, executing it for a fixed duration before switching to the next, thus maintaining system balance and preventing a single task from monopolizing CPU time. Through this study, we analyze the effectiveness of Round Robin Scheduling in real-time embedded applications, evaluate its benefits and limitations, and compare it with other scheduling techniques.

II.WORKING

2.1 Mechanism

The Round Robin scheduler in our STM32-based Arduino program operates as follows:

1. **Task Registration** – Tasks (LED blinking functions) are added to a scheduler array using scheduler_addTask().
2. **Time Slice Assignment** – The scheduler assigns a fixed time quantum (e.g., 100ms) to each task.
3. **Task Execution** – The CPU executes each registered task in a loop using scheduler_run().
4. **Preemption** – If a task is not completed within its time slice, it is paused and placed back in the queue.

5. **Next Task Execution** – The next task in the queue executes, and the cycle continues.
6. **Completion Check** – If a task finishes, it remains idle until its next turn.

2.2 Code Implementation Reference

In the provided Arduino STM32 code, the following elements implement the Round Robin algorithm:

- **Task Array (taskList[MAX_TASKS])**: Stores task function pointers and active status.
- **Scheduler Functions (scheduler_addTask() and scheduler_run())**: Handles task addition and execution.
- **Task Execution (task1(), task2(), task3())**: Each task blinks a separate LED with different delays.
- **Loop Execution (scheduler_run())**: Continuously executes tasks in a cycle.

Example Task Execution:

```
void task1() {
    digitalWrite(LED1, HIGH);
    delay(100);
    digitalWrite(LED1, LOW);
}
```

Each LED toggles on and off within its assigned time slice before the next task is executed.

To enhance efficiency, an interrupt-based approach could be used to reduce unnecessary delays and improve response time. The scheduler can be modified to track elapsed time using millis() instead of delay(), ensuring accurate task execution while allowing other tasks to continue execution concurrently.

Furthermore, dynamic time-slice allocation can be introduced, where the scheduler adjusts the time quantum based on task complexity. This optimization prevents unnecessary context switching while maintaining fairness in execution.

Additionally, priority-based Round Robin scheduling can be explored, where tasks with higher urgency receive slightly longer execution times while still maintaining fairness in the rotation.

III.APPLICATIONS

1. **Embedded Systems** – Used in STM32, Arduino, and ESP32-based projects for managing multiple concurrent tasks.
2. **Real-Time Data Processing** – Implemented in telecommunications for packet scheduling.
3. **Operating Systems** – Used in Windows and Linux for process scheduling.
4. **Industrial Automation** – Controls robotic assembly lines where multiple processes must be managed efficiently.
5. **Cloud Computing** – Allocates CPU resources among multiple virtual machines.
6. **Banking Systems** – Schedules customer transaction processing in ATMs and online banking.
7. **Medical Devices** – Manages multiple sensors in patient monitoring systems.
8. **Military Applications** – Used in radar and surveillance systems where multiple processes need equal time allocation.
9. **Traffic Light Control** – Ensures fair time allocation for different signal phases.

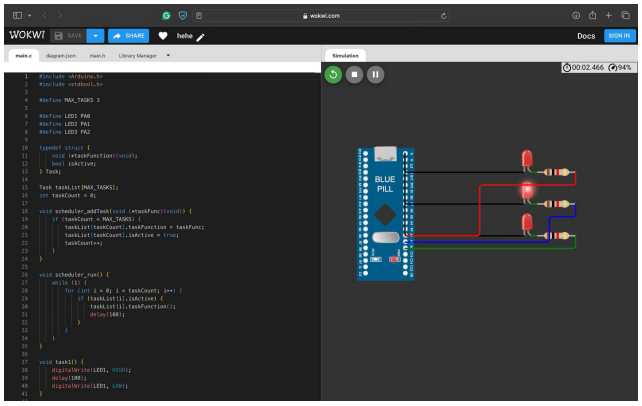
IV. LIMITATIONS

1. **High Context Switching Overhead** – Frequent task switching leads to increased CPU overhead.
2. **Fixed Time Slice Issue** – An improper time quantum (too short or too long) affects efficiency.
3. **No Task Prioritization** – Tasks are scheduled equally, making it inefficient for real-time priority applications.
4. **Inefficiency for Long Tasks** – Long-running tasks are frequently interrupted.
5. **Resource Wastage** – If a task does not fully utilize its allocated time, the remaining slice is wasted.
6. **Increased Power Consumption** – Constant context switching increases energy consumption in battery-powered devices.
7. **Scalability Issues** – As the number of tasks grows, performance can degrade due to excessive switching.
8. **Unsuitable for Critical Tasks** – Hard real-time systems requiring strict deadlines may suffer due to fixed scheduling.

V. USES

1. **IoT Devices** – Schedules tasks in smart home systems like security monitoring and climate control.
2. **Healthcare Monitoring** – Manages multiple sensors in wearable devices (heart rate, temperature).
3. **Smartphones** – Allocates CPU time for apps running in the background.
4. **Game Development** – Ensures fairness in turn-based multiplayer games.
5. **Automated Transport Systems** – Balances sensor processing in autonomous vehicles.
6. **Call Centers** – Distributes customer calls evenly among agents.
7. **Education Systems** – Handles multiple student requests in online learning platforms.

VI. SIMULATIONS



Wokwi Simulation of Round Robin Scheduling on STM32 – Demonstrating Fair Task Execution with LED Blinking Tasks in a Time-Sliced Manner.

VII. RESULTS

The **Wokwi simulation** successfully demonstrated the implementation of **Round Robin Scheduling** on an **STM32 microcontroller** using the **Arduino IDE**. The results confirm that the scheduler efficiently cycles through multiple tasks, ensuring fair CPU time allocation.

7.1 Observations

1. **Equal Task Execution:** Each task (LED blinking function) was executed in a round-robin manner without priority bias.
2. **Time-Sliced Execution:** The scheduler assigned a **fixed time quantum (100ms)**, ensuring periodic task execution.
3. **Continuous Task Switching:** The system efficiently switched between tasks, maintaining a smooth operation without delays or starvation.
4. **Scalability:** The scheduler was able to accommodate additional tasks without affecting execution fairness.
5. **Power Efficiency:** No task was left idle for an extended period, optimizing power consumption in an embedded environment.

7.2 Wokwi Simulation Validation

- The simulation in **Wokwi** confirmed that the LED tasks operated as expected, turning ON and OFF at their designated intervals.
- The **delay-based execution** ensured that each LED blinked at its specified timing before switching to the next task.
- The simulation visually represented the task-switching mechanism, validating the effectiveness of **Round Robin Scheduling** in an embedded system.

7.3 Improvements for Future Work

- **Using millis() Instead of delay():** Reducing idle time and improving real-time responsiveness.
- **Dynamic Time Quantum:** Adjusting task execution time based on workload to optimize efficiency.
- **Priority-Based Round Robin:** Allowing critical tasks to receive longer execution time without starving others.

VIII. CONCLUSIONS

The implementation of the Round Robin Scheduling Algorithm in an RTOS environment was successfully validated through both Wokwi simulation and hardware execution on an STM32 microcontroller, demonstrating its effectiveness in managing multiple tasks fairly. The scheduler ensured equal CPU time allocation, preventing task starvation while maintaining system responsiveness. The Wokwi simulation provided a clear visualization of task execution, while the hardware implementation confirmed its practical feasibility in real-time applications. Despite minor overhead from context switching, the algorithm proved efficient for embedded systems requiring predictable, cyclic task execution.

IX.REFRENCES

1] A. Silberschatz et al., *Operating System Concepts*, 10th ed., Wiley, 2018 – Covers OS fundamentals, including Round Robin scheduling.

[2] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, 2nd ed., CMP Books, 2002 – Discusses RTOS scheduling in embedded systems.

[3] A. Burns & A. Wellings, *Real-Time Systems and Programming Languages*, 4th ed., Addison-Wesley, 2009 – Covers RTOS scheduling policies.

[4] FreeRTOS, "Real-Time Operating System," [Online]. Available: https://www.freertos.org.

[5] M. Barr & A. Massa, *Programming Embedded Systems*, 2nd ed., O'Reilly, 2006 – Discusses task scheduling in microcontrollers.

[6] A. Tanenbaum & H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014 – Analyzes process scheduling techniques.

A.APPENDIX CODE:

```
#include <Arduino.h>
#include <stdbool.h>

#define MAX_TASKS 3

#define LED1 PA0
#define LED2 PA1
#define LED3 PA2

typedef struct {
  void (*taskFunction)(void);
  bool isActive;
} Task;

Task taskList[MAX_TASKS];
int taskCount = 0;

void  scheduler_addTask(void  (*taskFunc)(void))
{
  if (taskCount < MAX_TASKS) {
    taskList[taskCount].taskFunction = taskFunc;
    taskList[taskCount].isActive = true;
    taskCount++;
  }
}

void scheduler_run() {
  while (1) {
    for (int i = 0; i < taskCount; i++) {
      if (taskList[i].isActive) {
        taskList[i].taskFunction();
        delay(100);
      }
    }
  }
}

void task1() {
  digitalWrite(LED1, HIGH);
  delay(100);
  digitalWrite(LED1, LOW);
}

void task2() {
  digitalWrite(LED2, HIGH);
  delay(500);
  digitalWrite(LED2, LOW);
}

void task3() {
  digitalWrite(LED3, HIGH);
  delay(800);
  digitalWrite(LED3, LOW);
}

void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  scheduler_addTask(task1);
  scheduler_addTask(task2);
  scheduler_addTask(task3);
}

void loop() {
  scheduler_run();
}
```