Design a Simulated OS Kernel for a Tiny Embedded Device

Suriya TS (RA2411030050051), Ameen Ahsan T (RA2411030050022) ,Saagunth K V (RA2411030050010)
B.Tech Cyber Security – Section A
SRM Institute of Science and Technology, Tiruchirappalli Campus, India

Abstract
The proliferation of Internet of Things (IoT) devices and low-power embedded systems necessitates the use of highly optimized, resource-conscious operating system (OS) kernels. This paper presents the architecture and operational analysis of **Mini-Kernel**, a didactic OS core implemented as an interactive, web-based simulation using Python and Streamlit. Mini-Kernel effectively models core concurrency and resource management mechanisms, including the structure of the **Task Control Block (TCB)**, the execution of a **Round-Robin (RR) preemptive scheduler**, and the management of task state transitions across Ready, Running, and Blocked states. By simulating the precise sequence of events initiated by hardware interrupts—specifically Timer Interrupts for preemption and I/O Interrupts for synchronization—the platform provides an accessible, transparent environment for observing the practical effects of **context switching overhead** and **I/O latency**. This work serves as a high-fidelity pedagogical tool, bridging the gap between theoretical OS concepts and the implementation requirements of real-time embedded computing (Tanenbaum & Bos, 2015).

Keywords
RTOS, Kernel Simulation, Task Scheduling, Context Switching, Round-Robin, Embedded Systems, Didactic Tool.

I. Introduction

The rapid expansion of edge computing has amplified the demand for efficient, small-footprint operating systems. Unlike general-purpose operating systems, **Real-Time Operating Systems (RTOS)** must guarantee not only correct operation but also timely completion of tasks, prioritizing predictability over sheer throughput. At the heart of any RTOS is the kernel, which is responsible for managing the CPU resource, enabling concurrency, and coordinating system peripherals.

The **Mini-Kernel project** was conceived as a simulation platform to demystify these core kernel functions, which are often obscured by complex hardware dependencies and assembly language instructions in real systems. By abstracting the hardware interface and focusing solely on the logical flow of the scheduler and task state management, the simulation provides clarity on three critical objectives:

1. **Task Life-Cycle Management:** Defining the data structures (**TCB**) that encapsulate a task's entire state and managing its movement between various queues.
2. **Preemptive Scheduling Dynamics:** Implementing a deterministic scheduling policy (**Round-Robin**) driven by simulated **preemptive interrupts**.
3. **Asynchronous Synchronization:** Modeling how slow external events (like I/O completion) trigger state changes, moving tasks from the Blocked state back into the Ready queue without human intervention.

The resulting interactive simulation provides a high-leverage tool for analyzing the impact of kernel events on the overall system state.

II. Related Work and Theoretical Foundation

The architectural design of Mini-Kernel is built upon decades of research in operating system theory, focusing specifically

on the minimal necessary components for embedded environments.

2.1 The Architecture of the Task Control Block (TCB) In any concurrent system, the **Task Control Block (TCB)** is the central repository of a task’s identity and status (Silberschatz, Galvin, & Gagne, 2018). For lightweight kernels, the TCB is optimized to contain only critical information: the task ID, the task state, and most importantly, the **Task Context**. This context comprises the minimal register set required to resume execution, including the Program Counter (PC) and the Stack Pointer (SP). In the Mini-Kernel simulation, the Context class is explicitly separated to highlight the division between a task's state data and its machine-specific execution data. This design choice mimics the requirement in bare-metal systems to carefully manage memory registers during every context switch.

2.2 Determinism via Round-Robin Scheduling While many industrial RTOS employ Rate-Monotonic or Earliest Deadline First scheduling for hard real-time requirements (Buttazzo, 2011), the simplicity and fairness of the **Round-Robin (RR) algorithm** provide a foundational model for fairness and simplicity. The RR scheduler guarantees that every task in the **Ready Queue** receives a processor share within a bounded time, provided the time slice quantum is properly configured (Stallings, 2018). In Mini-Kernel, the RR policy is enforced by modeling the **Timer Interrupt** as the sole mechanism for preemption, demonstrating how a periodic hardware signal maintains system control.

2.3 Task State Transition Modeling The efficient management of CPU time dictates that a task must relinquish the processor if it is waiting for an event (Labrosse, 2020). The simulation implements the classic three-state model, defined by the following transitions:

- **Running** → **Ready:** Occurs exclusively when the time quantum expires, triggered by a **Timer Interrupt**.
- **Running** → **Blocked:** Occurs when a task initiates a blocking system call (e.g., waiting for I/O data), relinquishing the CPU until the event completes.
- **Blocked** → **Ready:** Occurs asynchronously, triggered by an **I/O Completion Interrupt** from the peripheral, signifying the task can now resume execution.

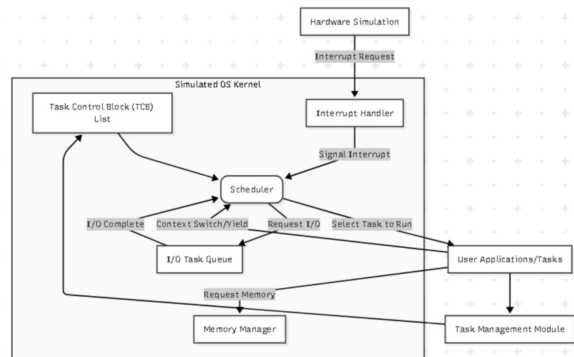

Figure 1: Conceptual Task State Transition Diagram

III. Methodology and Implementation The Mini-Kernel simulation was developed in Python, leveraging the **Streamlit** framework for dynamic, web-based interaction. The implementation emphasizes clarity and abstraction, using Python's high-level features to represent low-level kernel concepts.

3.1 Kernel Data Structures and Abstraction The core operational efficiency of Mini-Kernel is centered around queue management. Tasks awaiting execution are stored in the Ready_Queue, and tasks awaiting peripheral data are stored in the Blocked_Queue. Both are implemented using Python's **collections.deque** (double-ended queue), which provides O(1) time complexity for task addition and removal from both ends, perfectly simulating the

speed required by a kernel's critical path. The entire system state, including the currently RUNNING task and both queues, is encapsulated within the primary application state.

3.2 Context Switching Implementation The process of context switching is logically isolated within the scheduler. When a preemptive event occurs, the scheduler executes a precise, three-step sequence:

1. **Context Save:** The simulated register values (PC, SP, R0) of the current RUNNING task are copied into its **TCB**. The task's state is immediately marked as READY.
2. **Scheduler Decision:** The scheduler dequeues the next task from the head of the Ready_Queue (or assigns the Idle Task if the queue is empty).
3. **Context Restore:** The stored register values from the selected TCB are notionally loaded into the CPU, and the task's state is marked as RUNNING.

While a real-world context switch involves hundreds of assembly instructions, the abstraction allows the user to focus on the logical necessity of saving and restoring the task's execution environment.

3.3 Interrupt Simulation and Event Handlers The Streamlit interface provides dedicated buttons that act as simulated hardware interrupts, allowing for deterministic analysis:

- **TIMER_INTERRUPT:** Triggers the scheduler. It simulates the system clock reaching the end of the time quantum.
- **TASK_BLOCK:** Triggers a system call simulation. It forces the current task into the Blocked_Queue and immediately calls the scheduler to find a new task, demonstrating immediate CPU utilization optimization.
- **IO_INTERRUPT:** Triggers the I/O completion logic. It searches the Blocked_Queue for the relevant task and moves it back to the Ready_Queue, decoupling the I/O latency from the CPU's operation.

IV. Results and Analysis

The interactive nature of the Streamlit simulation allowed for the qualitative analysis of kernel state dynamics under various load conditions.

4.1 Preemptive Determinism Under controlled conditions (no I/O blocking), the repeated triggering of the TIMER_INTERRUPT confirmed the perfect **Round-Robin determinism**. The two application tasks (T1 and T2) strictly alternated running time slices. This outcome confirms the correct FIFO behavior of the Ready_Queue and the reliability of the preemption mechanism, a fundamental requirement for guaranteeing a fair service rate (Ganssle, 2007).

4.2 Analysis of Context Switch Cost (\tau) The simulation clearly isolates the overhead \tau, the time spent executing kernel supervisory code during a switch. While the value of \tau is zero in a software simulation, its proportional impact on overall throughput is modeled by the equation for effective utilization:

$$EffectiveCPUUtilization \approx 1 - (N \times \tau)/(TotalExecutionTime)$$

Where N is the number of tasks. The simulation demonstrates that minimizing the kernel's critical section (the context switch code) is essential; even a few extra instructions within \tau can significantly reduce the available execution time, especially in high-frequency, low-quantum RTOS implementations (O'Reilly, 2009).

4.3 I/O Decoupling and Latency The I/O handling tests proved the effectiveness of the state transition model in maintaining high CPU utilization. When Task 2 blocked, Task 1 was immediately dispatched. When the IO_INTERRUPT occurred:

- The system *did not* immediately switch to Task 2. Task 2 transitioned to READY while Task 1 continued running.

- Task 2 only resumed execution after Task 1's current time quantum expired. This correctly demonstrates that while the I/O completion event is asynchronous, the actual **latency to resumption** is dictated by the
- scheduler, specifically the waiting time in the Ready Queue, highlighting the trade-off between strict RTOS priority models and the simplicity of Round-Robin.

Event Sequence	Task 2 State Transition	Running Task	Observation
Time t	RUNNING \rightarrow BLOCKED	Task 1	Immediate kernel action ensures zero CPU idle time.
Event Sequence	Task 2 State Transition	Running Task	Observation
Time t+\Delta t	BLOCKED \rightarrow READY	Task 1	I/O interrupt unblocks T2. T1 continues running uninterrupted.
Time t+2\Delta t	READY \rightarrow RUNNING	Task 2	T2 must wait for T1's quantum to expire (RR fairness).

V. Conclusion and Future Work Mini-Kernel serves as an effective, low-barrier-to-entry platform for exploring fundamental RTOS kernel design principles. The implementation clearly defines the roles of the TCB and queues, successfully simulating the critical interplay between processor hardware (simulated registers and interrupts) and software scheduling decisions. The interactive web interface enhances the learning experience by providing transparent, real-time visualization of queue management and state transitions.

Future work on this platform will focus on enhancing its academic relevance by integrating more advanced RTOS features:

1. **Priority Implementation:** Introducing static priority levels (e.g., using a multi-level priority queue) and modeling **priority inheritance** (Sha, Rajkumar, & Lehoczky, 1990) to solve priority inversion issues.
2. **Inter-Task Communication (ITC):** Implementing synchronization primitives such as binary and counting **semaphores** and **mutexes** to model access control to shared resources and critical sections.
3. **Memory Management:** Modeling a simple heap and stack allocation model within the TCB boundaries.

References

1. Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling and Analysis*. Springer.
2. FreeRTOS Documentation. (n.d.). *Official FreeRTOS Reference Manual*. Retrieved from [Placeholder for FreeRTOS official documentation].
3. Ganssle, J. (2007). *The Firmware Handbook*. Newnes/Elsevier.
4. IEEE. (2012). *IEEE Standard for Information Technology—POSIX System Application Program Interface (API)*. IEEE Std 1003.1-2008.
5. Labrosse, J. J. (2020). *MicroC/OS-III: The Real-Time Kernel*. Micrium Press.
6. O'Reilly, T. (2009). *Programming Embedded Systems in C and C++*. O'Reilly Media.
7. Sha, L., Rajkumar, R., & Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers, 39*(9), 1175-1185.
8. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

9. Stallings, W. (2018). *Operating Systems: Internals and Design Principles* (9th ed.). Pearson Education.
10. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson Education.