MECT: Real-Time Services on Linux GPOS

107474-Joseane Pereira
109050-Gabriel Costa
Universidade de Aveiro, DETI

November 5, 2024

Contents

1 Introduction **3**

2 Architecture **3**
2.1 System Overview 3
2.2 Data Structures and Access Methods 3
2.3 Synchronization Mechanisms 4

3 Discussion **4**
3.1 Priority Assignment and Activation Period Rationale 5
3.2 System Schedulability 6
3.3 Task Execution Sequence and Relevant Events 6
3.3.1 Task Execution Timeline 7

4 Tests **8**

5 Results **8**

6 Analysis **8**

7 Conclusion **8**

1 Introduction

The aim of this project is to apply the Linux Real-Time Services and the Real-Time Model to the the development of a real-life inspired real-time application.

Following the typical structure of embedded software, the project encompasses a set of cooperating tasks, involving synchronization, shared resources, access to a real-time database, etc.

2 Architecture

2.1 System Overview

In this project, the real-time monitoring system is structured around five primary tasks, each with distinct roles:

- **Sound Generation Task**: This task generates sound signals that simulate the acoustic output of a motor. The generated sound data serves as the foundation for speed measurement and fault detection tasks.
- **Measuring Speed Task**: The purpose of this task is to calculate the motor's rotational speed based on the generated sound signals. The task calculates the most powerful frequency, which corresponds to the frequency of the motor, and then converts it to RPM, representing the motor speed.
- **Bearing Issues Task**: This task is responsible for identifying potential issues in the motor. It processes the sound data to detect specific low-frequency signals that may indicate these issues. By monitoring frequencies below 200 Hz and checking if their amplitude is higher than 20% of the motor frequency, this task can flag conditions that suggest operational concerns.
- **Print RTDB Task**: Outputs from the above tasks are stored in a Cyclic Access Buffer (CAB), which allows the data to be periodically stored and retrieved. This task prints the results at regular intervals and shows the accumulated data from the buffer on a plot, providing a real-time overview of motor speed and any detected faults.
- **Playback**: This task allows for real-time audio playback of the sound generated by the first task. This functionality provides an additional verification layer, letting us monitor the sound data directly to ensure accurate sound generation and processing.

2.2 Data Structures and Access Methods

The primary data structure used in this project is a CAB. It is implemented to manage data flow between the sound generation task and the subsequent processing tasks in a synchronized and efficient manner.

A second key data structure is the Real-Time Database (RTDB), which stores information generated by both the Measuring Speed task and the Bearing Issues task.

- **Buffer Allocation**: The CAB allocates dedicated buffers to store sound samples as they are generated. The task searches for an available buffer—one that is not currently in use—before writing data to it.

- **Data Access by Processing Tasks**: To read data from the CAB, processing tasks first identify the buffer that was most recently updated. Before accessing this buffer, they increment a counter that tracks the number of tasks currently using it. This counter mechanism prevents new data from being written to the buffer while it is actively in use. Once the tasks have finished reading, they decrement the counter, signaling that the buffer is available for future writes.

2.3 Synchronization Mechanisms

The implementation relies on multiple threads that perform specific tasks concurrently. To ensure proper coordination, various synchronization methods are applied. Below are the synchronization techniques identified:

- **Clock-based Periodic Synchronization**: All threads use the function "clock_nanosleep" with the "TIMER_ABSTIME" flag and "CLOCK_MONOTONIC" as the reference clock to maintain periodic execution intervals.

  Each thread calculates the time of its next execution cycle and then waits for this specific absolute time, which is computed using a custom function "TsAdd" to increment the target time with the thread's predefined period.

- **Mutex and Locking in CAB Functions**: The CAB structure uses mutexes to control access to shared resources. This prevents tasks that are reading from the buffer from simultaneously modifying the counter, which tracks the number of tasks currently using the buffer, preventing unpredictable behavior.

3 Discussion

Each task in the system is assigned a specific priority and activation rate to ensure timely execution and prevent task starvation. The priorities, activation rates, and periods are configured to meet real-time requirements and ensure system schedulability. The following table outlines the task priorities and activation periods.

Task	**Priority**	**Activation Period**	**Description**
Sound Generation	1	3.2s	Generates sound signals representing the motor's acoustic output.
Speed Measurement	2	3.8s	Analyzes sound data to measure motor speed by detecting the dominant frequency in the spectrum.
Bearing Issues	3	6.8s	Detects potential faults in the motor by checking for low-frequency noise (below 200 Hz).
Playback	4	4.9s	Plays back the generated sound for monitoring and debugging purposes.
Database Print	5	4.8s	Prints the real-time database contents, including speed and fault information.

Table 1: Task Priorities and Activation Periods

3.1 Priority Assignment and Activation Period Rationale

Priority Assignment Rationale: Tasks are prioritized based on their importance in the data processing chain:

- **Sound Generation** has the highest priority (1) because it provides the data required by subsequent tasks.
- **Speed Measurement** (priority 2) and **Bearing Issues** (priority 3) are secondary since they need the generated sound data to operate correctly.
- **Playback** (priority 4) and **Database Print** (priority 5) are dependent on the other tasks, so they have lower priorities.

Activation Period Rationale: The periods are selected to balance the need for fresh data and processing time.

- **Sound Generation** has the shortest period to ensure continuous data flow and it's value also comes from the fact that we need to be able to track the changes in the motor (working faster or slower, without big leaps).
- **Speed Measurement** has a slightly longer period than the previous task as to avoid unnecessary waiting time.
- **Bearing Issues** has a longer period because bearing issues are less frequent, which means that this task doesn't have to execute so frequently.
- **Playback** and **Database Print** activate slightly less frequently because they are dependent on the other tasks.

3.2 System Schedulability

To confirm that all tasks meet their deadlines, we perform a schedulability analysis using the system's utilization factor.

Utilization Factor (U): Given that each task is independent and periodic, we can calculate the utilization factor U using:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} <= n * (2^{\frac{1}{n}} - 1)$$

where C_i is the computation time and T_i is the period of each task. Assuming each task completes within its assigned period, this calculation helps ensure that the system remains schedulable. Empirical validation was conducted by observing actual execution times to confirm that all tasks meet their deadlines.

For our system, the utilization factor is calculated as follows:

$$U = \frac{4.69 \times 10^{-3}}{3.2} + \frac{5.19 \times 10^{-3}}{3.8} + \frac{2.31 \times 10^{-3}}{6.8} + \frac{0.10 \times 10^{-3}}{4.8} + \frac{0.03 \times 10^{-3}}{4.8} = 2.44 \times 10^{-3}$$

$$5 * (2^{-\frac{1}{5}} - 1) = 0.74$$

$$2.44 \times 10^{-3} <= 0.74$$

The utilization factor is far below 0.74, indicating that one execution per period is guaranteed.

3.3 Task Execution Sequence and Relevant Events

The system follows a well-defined sequence to maintain real-time performance. Below is the order of task execution and relevant interactions:

1. **Sound Generation**:
   - Generates sound signals simulating motor noise and stores the wave in the Cyclic Access Buffer (CAB).
2. **Speed Measurement**:
   - Reads wave data from the CAB to calculate the motor's speed based on the dominant frequency.
   - Converts the frequency data to RPM and records it in the Real-Time Database (RTDB).
3. **Bearing Issues**:
   - Checks for low-frequency noise in the generated wave (indicating potential motor faults).
   - Logs the fault frequency and amplitude in the RTDB if an issue is detected.
4. **Playback**:

- Retrieves the wave from the CAB and plays it back for real-time monitoring.

5. **Database Print**:

    - Prints the contents of the RTDB, including motor frequency, RPM, and detected faults.
    - Every three cycles, generates a Gnuplot chart showing RPM and fault data over time.

3.3.1 Task Execution Timeline

Figure 1 illustrates the task activation times, periods, and interactions in the worst case scenario, which is when all tasks are ready at the same time.

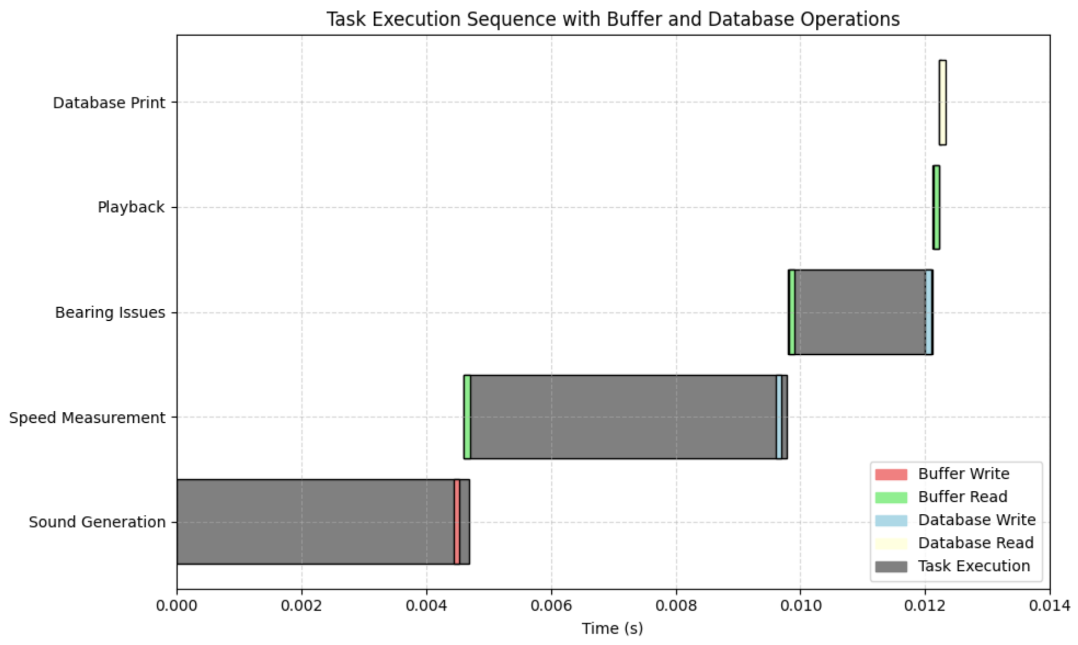

Figure 1: Gantt chart of the task execution sequence

The chart shows:

- Task Start/Finish Times: Each task's execution times are represented, demonstrating how tasks interact and avoid overlaps.
- Buffer and Database Operations: Annotations illustrate when tasks read from or write to the CAB and RTDB.
- Synchronization Events: Events like CAB writes and RTDB updates are highlighted to clarify data flow.

The chart does not display future task executions because, as previously mentioned, the system is schedulable. This means all tasks will finish before their deadlines. Also, since the deadline (equal to the task period) is significantly longer than the actual execution time, it would be challenging to visually capture the exact duration of each task in the chart.

4 Tests

The Sound Generation task produces a waveform with a frequency range from 2 kHz to 5 kHz, varying with every execution.

After five cycles, the Sound Generation task begins to introduce substantial noise to the generated sound, simulating that there is something wrong with the motor.

5 Results

In the Measuring Speed task, the most powerful frequency detected always falls within the expected range, confirming that frequency analysis is functioning as expected.

The noise generated before is promptly detected by the Bearing Issues task, with its presence illustrated in the graph generated by the RTDB printing task and displayed in the terminal. The Playback task further confirms this change, as the audio output shifts from the initial smooth waveform to a noticeably distorted sound, verifying that the noise has been successfully introduced and detected.

6 Analysis

The current implementation of the Sound Generation task serves as a periodic simulation of motor sound collection. For improved tracking accuracy in motor speed estimation, it would be beneficial for this task to operate at a higher frequency. By increasing the frequency of the Sound Generation task, the Motor Speed task—and consequently other tasks dependent on it—would gain access to more granular data, reducing the likelihood of missing critical state changes in motor behavior.

In a real-world application, the Bearing Issues task could likely operate at a much lower frequency. Since motor faults are relatively rare, checking for bearing issues at longer intervals, such as once per minute, would suffice. However, for demonstration purposes, we decided to use a lower frequency in order to be able to prove that the task works as expected.

The current priority of the Playback task is higher than that of the RTDB Print task. Given that the Playback task serves primarily as an additional debugging tool, it would be more appropriate for it to have a lower priority than the RTDB Print task.

Regarding the CAB implementation, each buffer contains a property called `position`, which is used by the Playback task to replay the sound generated by the Sound Generation task. Although the Playback task is the only one utilizing this property, directly modifying it within the task is not ideal. Instead, best practice would be to copy the `position` value into a local variable for manipulation. This approach promotes safer handling and reduces the chance of unintended side effects within the buffer's state.

7 Conclusion

In this project, a multi-threaded system was successfully implemented to generate, process, and monitor audio signals in real-time. Through a coordinated set of tasks, each functioned as intended: the Sound Generation task produced audio with variable frequencies, which were accurately detected and analyzed by the Measuring Speed and Bearing Issues tasks. The synchronization mechanisms, including mutexes in the CAB, prevented data races and ensured stable inter-thread communication. Additionally, the Playback task provided an audible confirmation of sound changes, which were corroborated by visualizations of signal data in the database printing task.

Overall, this project demonstrated effective use of periodic task scheduling, synchronization, and signal processing to achieve reliable sound generation and real-time monitoring.