A framework prototype for multithreading implementation over microcontrollers

Everton Nagel¹, Ricardo M. Czekster¹, Thais Webber¹, and César Marcon²

¹ University of Santa Cruz do Sul, Santa Cruz do Sul, Brazil
² PPGCC, PUCRS University, Porto Alegre, Brazil, e-mail: cesar.marcon@pucrs.br

Abstract—The applications requirements are becoming more rigorous, demanding the execution of concurrent tasks that must also take into account modularity and flexibility. A fundamental part of the operating system development concerns the implementation of scheduling algorithms. In an embedded system context, it is essential to consider the scheduling algorithm heavily influences the application behavior. The restricted and finite hardware resources force to evaluate the use of flexible algorithms to guarantee efficiency. Currently, projects for embedded operating systems do exist for microcontrollers’ devices that implement scheduling algorithms; however, the developer cannot change or add new scheduling policies without implementing kernel tweaks and modifications. The alternatives are not flexible when choosing the scheduling algorithm according to the application needs. This imposes restrictions to many systems, forcing them to run specific static scheduling algorithms because no other options are available. This work concerns the design and development of a framework for implementing a microkernel with a modular scheduler unit, allowing the execution of tailored algorithms according to the application profile. The idea is to provide a flexible platform to select the most appropriated algorithm conveniently. We have employed lightweight hardware to implement multithreading patterns corresponding to sets of concurrent tasks, demonstrating the strengths of adopting our approach. Our results show the use of modern techniques combining modularity, multithreading, and scheduling methods for embedded systems yield best executions when compared to its sequential counterparts.

Index Terms— microkernel frameworks, embedded computing, scheduling applications, low cost computing.

I. INTRODUCTION

Multithreading is a programming technique widely addressed to develop parallel and concurrent applications [1]. Any modern Operating System (OS) encompasses instructions to implement parallel and distributed applications [2]. In every Embedded System (ES) platform, the use of threads is crucial due to the efficient handling of scarce resources allowing faster executions in a reasonable fashion [3]. The scheduling policy is a recurrent issue in OS implementation because it affects the execution efficiency directly [4]. It aims to use the target architecture resources rationally according to the application instruction set. In these settings, developers must thoroughly study the applications’ characteristics to set which scheduling algorithm is suitable to accomplish specific tasks. For embedded architectures, it is common to employ a unified abstract level, aiming to minimize complexity and processing costs [5]. Therefore, every microkernel project for low-cost architectures must consider simplicity in its design.

Microcontrollers are widely adopted in the embedded contexts because they are integrated computers in a chip [6].

These devices comprise only necessary resources for developing low-cost applications. However, due to the increasing demand required by the current applications, the designer shifts the application implementation from the hardware towards the software. Consequently, many Application Programming Interfaces (APIs) are used to aid programmers by offering concurrent directives to developers as well as allowing the selection of scheduling policies that produce efficient application executions [7].

An OS tailored to general applications needs to provide a reasonable service level when selecting scheduling policies targeting efficient execution. The literature on this subject provides several discussions about algorithmic solutions to the kernel’s scheduler module [8][9][10]. However, when considering embedded OSs, more specifically microkernels of microcontrollers, it is widespread to offer only a single static algorithm implemented for specific executions [11]. There is no flexibility for developers to switch scheduling policies effortlessly, without having to resort to several source code modifications.

Users should be able to develop their applications and expect reasonable performance given the static scheduling algorithm without considering the scheduler module. In our point of view, this is a huge problem found in several embedded frameworks and architectures. This paper discusses the design and implementation of a framework that provides a solution to such issues, allowing the development of multithreaded applications having a microcontroller as target architecture. The idea is to devise a microkernel consisting of a modular and scalable scheduler, which allows executing concurrent tasks according to a set of scheduling policies. We aim to offer flexibility and easiness to developers by allowing them to alternate among scheduling algorithms in design time. For every policy, the framework aids the generation of specific firmware targeted to the ES where the designer can inspect the application behavior at execution time. Such premises let examining and assessing the best scheduling algorithm that is a close match for the application’s requirements, letting smoothly concurrent and parallel executions.

II. MULTITHREADED FRAMEWORK DESCRIPTION

There is an enormous interest in scheduling embedded OSs to achieve peak performance using scarce resources. We have analyzed several frameworks in search for practical proofs of concepts or experimental versions implemented over commercial architectures. We have studied the software’s internal organization and have perused the documentation to understand the scheduler, its algorithms, and implementation as well as flexibility characteristics coded.
in APIs. We also observed other quality issues such as modularity levels, functionalities set, community support, advanced scheduling policies, working examples, and licenses.

A. Related work

FreeRTOS [12] is one of the most acknowledged embedded Real-Time OSs (RTOSs), which was developed to manage low-cost architectures including microcontrollers. FreeRTOS is vastly used by both industry and academy, as versions are found in many commercial products [13]. The system supports several architectures from different vendors, and some specific portability software are written specially to support each architecture. FreeRTOS is written in the C Programming Language and its kernel’s modules implements three source code files: (i) one responsible for task management (scheduling), (ii) one describing lists also used to organize the tasks, and (iii) another file to handle other structures such as semaphores, critical sections and so on. Each task has an integer (greater or equal to zero) used as a priority flag. The scheduling can be set up as preemptive or cooperative. The preemptive scheduler interrupts current tasks and uses priority levels to choose the next task to be executed inside the quantum. If two or more tasks have the same priority, the scheduler chooses a Round-Robin policy to handle those requests.

AtomThreads is a lightweight and understandable framework with a portable RTOS for ESs [13][14]. The platform offers essential resources to facilitate the design of multitask applications, providing a primitive real-time scheduler and synchronization mechanisms to help the development of complex applications. Other functionalities such as file systems, communication protocols, and drivers must be attached separately, depending on the project requirements. The framework currently supports a few sets of architectures in a preemptive scheduler. Tasks are created with a priority between 0 (highest) and 255. While several tasks have the same priority, just as FreeRTOS, AtomThreads considers a Round-Robin execution; however, for some blocking cases, for instance, while waiting for a semaphore, the kernel allows the current thread to be scheduled before completing the execution quantum.

TinyOS [15] executes over wireless sensors and actuators, having extensive and complex source code for handling a multitude of drivers and communication devices from several vendors. TinyOS is an RTOS that uses NestC as programming language due to event handling mechanisms and cooperative tasks implementations.

Contiki [16] is an OS tailored for Internet-of-Things (IoT) and network applications. The dispatch scheduler is cooperative, forcing application tasks implementation to switch contexts explicitly. Almeida [17] presents an interest work employing low-cost microcontrollers with an abstract software layer and driver management facility. The kernel implements a cooperative scheduler, where each task must inform whether it should be rescheduled for execution. An approach taken in [18] has considered different implementations of OSs for low-end smart devices.

FreeRTOS encompasses a large variety of architectures, with many examples, whereas AtomThreads provides only essential RTOS functions. However, the latter has a simple source code structure using ANSI-C standard definitions. Additionally, both frameworks implement the HAL (Hardware Abstraction Layer) similarly, but only FreeRTOS keeps it within the same kernel folder, offering customized algorithms for handling the heap memory.

Our current work demonstrates the interest in designing and implementing microkernels for ESs. We want to address AtomThreads is very easy to understand, making it an excellent starting point for more advanced studies. Besides, AtomThreads has unit testing mechanisms to assert the functioning when porting to different architectures. The drawback, when comparing to FreeRTOS, is the scheduler implementation, which is intermingled in the same place of the kernel, making it very hard (regarding implementation) to change policies according to application demands. It is worth mentioning that, for both cases, one must specify the set of tasks must be executed before the schedule initializa-

tion. This limitation is imposed to meet deadlines of real-time tasks. Although AtomThreads is less popular than FreeRTOS, many industrial and academic projects use it in numerous applications due to its reliability.

B. Framework details

The framework proposed here does not switch the scheduling algorithm in execution time. Instead, the framework compiles all the desired firmware as each one executes the application code using the corresponding methods to deal with the target multitasking application. This procedure customizes and adapts the scheduling policy for each application execution characteristics. We intend to provide a mechanism to allow choosing the scheduling algorithm at design time, automating firmware generation without resorting to needless modifications in the source code.

Previously studied frameworks work with a single and monolithically designed scheduling policy. However, we stress the algorithm must be generic enough to execute for a broad range of cases gracefully. The efficient CPU usage in ES projects can be achieved employing specific scheduling algorithms for handling the application behavior, enabling to fulfill load balancing and the performance required by the application. Fig. 1 shows a modular approach for a microkernel. Two parts comprise the scheduling module: the first part addresses specific functions requested by the scheduling policies, whereas the second part deals with rules for deciding if threads should return or not to the list of threads ready for timely execution.

![Modular microkernel scheduler.](image-url)

C. Framework details

Our proposal, together with design choices, enables the developers to implement their solutions with a solid basis.
on code reuse, a fundamental requirement in current ESs projects. One must know the characteristics of the application under execution and switch the scheduling policy accordingly, so the execution performs acceptably. The development platform is a low-cost architecture, an Atmega1284P microcontroller device [19]. The essential framework features are a terminal, a messaging system for debugging, a fault tolerance module, and other technical structures such as generic lists support, and memory reorganization to provide multithreading execution. The framework uses the Thread Control Block (TCB), a standard OS structure to store metadata concerning threads as well as other functions.

The implementation of the framework deals with short-term scheduling practices, specific for single applications with concurrent tasks that take into account strict performance requirements. The framework contains two TCB lists, the Ready List and the Waiting List; and two pointers, one for the TCB under execution and another for the Idle Thread. The kernel executes this thread when the Ready List is empty; i.e., no thread is available for execution. The framework implements the following scheduling policies:

1) **Round-Robin (RR)**, i.e., a low-complex scheduling policy that appends threads in the Ready List in a First Come First Served (FCFS) basis, and whose reduced code enables fast context switches;

2) **Dynamic Priority (DP)**, a scheduling policy that changes the thread priority level with the execution flow. Each thread is positioned in the Ready List according to its current priority level. Each quantum a thread is executed, the dispatcher decrements the priority level. Whenever the thread reaches the minimum priority level, DP restores its priority to the original setting;

3) **Static Priority (SP)**, a scheduling policy that gives a fixed priority level for each thread assigned to the Ready List. If there are several threads with the same priority level, the behavior will be similar to the RR policy as well.

RR and DP policies use a mechanism to handle threads returning from the Waiting List, signaling the TCB with a wake-up flag. The value of this variable is checked when reinserting the thread back to the Ready List. If it is active, the thread is positioned at the beginning of the Ready List, recovering its execution as soon as possible to accomplish time-precision within the defined delay interval. The dispatcher removes this signal upon execution, adjusting the algorithm behavior for all subsequent executions, where the thread moves from an Executing status to a Ready status. DP is a valid alternative to prevent starvation and balance CPU usage among threads through priority manipulation.

For all policies, the use of the Idle Thread is identical. When the dispatcher is called, the Ready List of threads is verified. If the list is empty, the Idle Thread is used, keeping execution until another thread arrives. Fig. 2 shows the flowchart when creating threads for execution, where the final stage (grayed in the figure) calls the scheduler and position the thread into the Waiting List. Later, a system call can resume the thread and move it to the Ready List according to the scheduling policy.

The delay mechanism designed for the framework is crucial for ESs where the information exchanges with the external environment are not atomic. Some routines execute only at specific time intervals, such as sensing or other information sources that must not be under continuous processing, for instance. The delay mechanism is implemented as a kernel system call because it cooperatively moves the current thread under execution to the Waiting List and performs a context switch. This delay mechanism frees the CPU to execute the instruction set of other threads in the Ready List. The delay is dependent on the quantum and clock system. The system call inspects whether the desired time is less than the quantum value. In this case, the effort required to leave and come back from the Waiting List must be advantageous, using the delay macro provided by the HAL instead.

### D. Initialization

Specific procedures complemented by a constructor attribute initialize the HAL signaling that it must be called before the main function. Therefore, developers can set up specific initialization details needed by applications ensuring seamless executions. The loading features follow this order: terminal, specific APIs, I/O callbacks, critical section functions, kernel pre-initialization (Ready List, Waiting List, and Idle Thread), system clock, and, lastly, architec-

---

**Fig. 2 Flowchart for creating new threads for execution.**
ture interruptions are enabled. Kernel pre-initialization and system clock variables are located at higher levels.

The kernel initialization procedure must be called in the main function explicitly. Ideally, the kernel must be started after the registration of defense procedures and after the addition of some application threads. The no return attribute specifies the function must not backtrack to the place where it was called (in this case, the main function). This attribute is necessary to ensure the kernel is started only once, where the dispatcher moves the first thread ready for execution to the Ready List. The TCB is signaled as started, and the Stack Pointer is assigned to the origin of the Stack Memory. Then, the scheduler takes charge of the execution according to the policy. Two verifications are carried out to prevent errors and act on failures: the first guarantee the Ready List is not empty; hence, the kernel initialization may advance; the second inspects the system for verifying the unconditional error status, calling specific defenses when triggered.

E. Context switching

HAL initializes controls and finalizes processes respecting every architectural detail, whereas the intermediate execution depends on the scheduling policy within the kernel module written to meet these demands. The context switching is performed by two manners: (i) pre-emptively, through quantum interruption; or (ii) cooperatively, through system calls. Both mechanisms use the naked attribute (i.e., it avoids the generator of entry and exit codes for a function), depending on the target architecture, which prevents the compiler from generating any entry or exit code. The objective is to save essential registers in Stack Memory and restore them upon subroutine return. This procedure is customized to guarantee data integrity when switching thread contexts.

The first stage disables interruptions globally within the application and saves CPU registers and other crucial data in the thread’s Stack Memory, also storing the Stack Pointer (SP) and the Program Counter (PC). Then, the scheduler is triggered, as observed in the gray square of Fig. 3. In the HAL’s point of view, the scheduler objective is only to select the next thread to be executed. Thus, after the scheduler execution, the HAL performs the second stage — restoring the SP of the new thread to be executed and reinitializes the register responsible for controlling the quantum value, so that a new iteration can start. It is worth noticing the quantum is reinitialized regardless of a preemptive or cooperative context switch. As shown in Fig. 3, the last stage of context switching is forked into two possible ways. If the thread selected by the scheduler is still not running, it must be initialized, enabling interruptions and performing a function call. However, for the next context switches, the already contextualized threads will restore its registers; retrieve the SP and PC, resuming the computations.

Fig. 3 Flowchart of a context switching in HAL’s point of view.

Fig. 4 presents the internal flowchart of context switching from the kernel’s point of view. It is noticeable the execution differs because it must consider the scheduling policy to be used for each case.

The three gray squares in Fig. 4 are explained next: (i) tasks responsible for sending back threads that have concluded the waiting time into the Ready List; (ii) it is used to send back to the Ready List the current thread under execution; and (iii) it runs for all threads, where the dispatcher is used to move the first thread ready for execution. At the end of this process, the control returns to the HAL where the second stage of the context switching resumes the execution. All process is repeated perpetually after the kernel initialization, except for some brief moments where the quantum interrupts are disabled; i.e., the case where critical sections are defined.

III. METHODOLOGY AND EXPERIMENTAL RESULTS

Our first requirement is to understand how the proposed framework will behave when executing different applications. Therefore, we have devised a methodology to execute the same algorithm for all threads where we have chosen to use different memory regions for each one. Our objective is to evaluate scheduling policies and investigate their effect on resource usage. Therefore, the threads will be created using a shared function that always computes the same results and preemptively performs context switches. The difference between each execution is the time required to finish computation of the evaluation algorithm and the number of times the CPU has gained execution within a given period.

For our experimental results, we have selected the Bub-
ble Sort as the main algorithm to exemplify the gains provided by our framework. Each thread works with a statically allocated array containing the same unsorted values. The address of each array is passed as a parameter in thread creation time; hence, when a thread starts executing, the reference for the data structured is read from the Stack Memory.

The experiment comprehends four threads, each one working with a 2048 bytes array. Since the integer is 16-bit width, the sorting algorithm works with 1024 integer positions. It is a known fact in ESs design that one must take into consideration the memory requirements for each thread stacks and other necessary resources. In light of this, we are investigating memory allocation, context switching, and thread execution using two monitoring methods:

- **End of Execution**: every time a thread ends executing the evaluation algorithm; i.e., the time spent between each initialization and sorting of the array;
- **Time Cycles**: this method performs a continuous measurement; after 60 seconds of execution the measurement reports how many times each thread has acquired the CPU.

The two monitoring types used the terminal application to measure results for a set of iterations. The same algorithm has been executed 50 times to collect mean values and confidence level intervals. We have installed a general-purpose computer connected to the serial interface to receive/store data for later statistical analysis.

Fig. 5 shows we have added a fifth thread named Monitor Thread into the system to infer the Time Cycles measurement correctly using the delay mechanism. Monitor Thread uses polling to measure the status of the evaluation thread execution, keeping itself in the Waiting List all times that it is idle; i.e., it does not significantly skew the results during time capture intervals.

Fig. 5 exemplifies the initialization of an entire application in our framework enabling the execution of several threads. We initially register a so-called defense action (i.e., a special code call whenever some logical error condition is detected), and then we define each thread details and end our procedure by calling kernel initialization procedures, leaving the task of choosing the next task for the scheduler. Each evaluation scenario was executed for long durations, enabling the correct estimation of the execution. As mentioned earlier, we have chosen to compute the confidence intervals with 50 samples and 95% confidence level, deemed sufficient for an ES application with such requirements.

### A. Round Robin (RR)

RR policy, in this context, was used as a comparing base with sequential execution to show the benefits when considering multithreading applications over single thread executions. Fig. 6 presents three charts with averages and error

```c
/* bubble sort threads - working example */
tid = k_thread("sort_1", 1, sort_thread,
    &sort_thread_arg[0], STACK_SZ, stack[0]);
k_thread_resume(tid);

tid = k_thread("sort_2", 1, sort_thread,
    &sort_thread_arg[1], STACK_SZ, stack[1]);
k_thread_resume(tid);

tid = k_thread("sort_3", 1, sort_thread,
    &sort_thread_arg[2], STACK_SZ, stack[2]);
k_thread_resume(tid);

tid = k_thread("sort_4", 1, sort_thread,
    &sort_thread_arg[3], STACK_SZ, stack[3]);
k_thread_resume(tid);

/* monitor thread */
tid = k_thread("monitor_thread", 2,
    monitor_thread, NULL, 400, NULL);
k_thread_resume(tid);

/* initialize kernel to start threads execution */
kernel_start();
```

Fig. 5 Example for an evaluation application used for the Time Cycles measurement method.
Fig. 6 Comparison between sequential execution and RR policy. 

Fig. 7 Comparing CPU gains using RR policy and polling. (where there are no context switch intervals) should present a shorter turnaround time; however, due to the compiler optimizations, the round-robin executions presented better results given the characteristic of the test algorithm that performs frequent memory accesses.

The second measuring mechanism uses polling and only RR policy because sequential executions do not benefit from kernel implementation improvements when concurrent behavior is present. Fig. 7 shows our results with the same quantum variation (1 ms and 10 ms). It also shows the average number each thread has acquired the CPU for 60 seconds. It depicts a significant difference when varying the quantum regarding time and CPU gains.

### B. Dynamic Priority (DP)

Fig. 8 exemplifies the scheduler using the DP policy on the evaluation algorithm. The experiment employs four threads starting with priorities 4a, 3a, 2a, and 1a, where the letters represent a second priority level (used here only to

Fig. 8 An example of the general algorithmic behavior of the DP policy.
explain the process), which is applied to account for the position of the thread at Ready List, since multiple threads could potentially behave the same way as execution progresses.

As soon as the kernel initializes, the scheduler promptly moves the highest priority task (at the beginning of Ready List) for execution, in this case, task 4a. This thread is decremented in one point, becoming 3b, since there is another task with 3a priority ready for execution (it assumes a value to be executed after the next one is scheduled for execution, assuring fairness execution for all tasks). This process is continuous for each thread, and when a priority equals one, then it is restored to the original value.

The next analysis shows the DP mechanism with monitoring based on End of Execution.

Fig. 9 exhibits three charts, each one representing an evaluation scenario regarding the priority of threads. The evaluations demonstrate the impact of a different set of priorities in concurrent executions.

Fig. 10 illustrates the results of the Time Cycles methodology, where CPU gains are monitored every 60 seconds. The experiment repeated the same sets of priority levels as shown in Fig. 9. It is noticeable the inverse pattern in the histograms because threads with more priorities are gaining the CPU more often, as verified by the time needed for execution of each thread in Fig. 9.

On the one hand, the DP policy allows fair and balanced CPU usage among concurrent threads, a clear distinction in comparison with RR policy. Additionally, some critical applications must ensure CPU usage over other threads, without causing starvation. On the other hand, the context switching is more complex, and this complexity varies ac-
according to the number of threads and assigned priorities.

C. Static Priority (SP)

The main problem when using SP policy concerns starvation, i.e., some threads may not have the actual chance of gaining the CPU at any time. A possible solution for this problem is the use of a delay mechanism applied to the threads with higher priorities, placing them on the Waiting List after executing the evaluation algorithm, allowing threads with fewer priority levels to gain the CPU eventually. Fig. 11 shows six charts for monitoring using the End of Execution method, e.g., all scenarios are setting the same priorities for its threads (i.e., 2, 1, 1, 1); however, for each scenario, the delay time is applied to thread 1 after executing an iteration of the evaluation algorithm. This delay value is increased by 500 ms on each subsequent chart, starting with 500 ms in Fig. 11 (a) until 3000 ms as shown in figure 11 (f).

Thread 1 of Fig. 11 has a proper execution model, which is the reason for the times obtained are sensibly faster than the others are. Threads 2, 3 and 4 have the same priority. Thus, they share CPU usage while Thread 1 is waiting. It is
noticeable the average execution time of Threads 2, 3, and 4 has changed as the delay time of Thread 1 raises.

Fig. 12 shows the experimental results from averages obtained with Time Cycles methodology, where the same priorities and delay times as in Fig. 11 are being used. Figure 12 (a) shows the first thread accesses more the CPU because the priority is higher than others. However, combined with Fig. 12 (cases b, c, d, and e) depicts that according to the delay values raises for Thread 1 the remainder threads increases its CPU gains. Besides, the last chart of Fig. 12(f) shows the lower priority thread overcomes the higher one.

D. Discussion

Tab. 1 shows all the experimental results obtained when considering a quantum of 1 ms, and 50 samples (N) for a confidence interval (C) of 95%, which enable us to analyze measurement ranges. Fig. 13(a) compares our results for each policy and evaluation scenario measured employing the End of Execution method, whereas Fig. 13(b) uses Time Cycles.

We stress that we have used RR in a broad manner, mainly because it is de facto the simplest scheduling available for implementation and offers a good comparing base
when investigating other scheduling algorithms. It is hard to compare different scheduling algorithms because of delay parameters and other application design decisions. Additionally, we emphasize the developers must deeply inspect applications’ workloads to determine the best tradeoffs when considering balancing and concurrency.

This section discussed our implemented scheduling policies with a general analysis covering all used evaluation scenarios for a given application. Flexible schedulers can alternate scheduling policies and modify thread/scheduling procedures for a better CPU load balancing according to the application profile. It is also worth to mention the implemented infrastructure is not restricted to only these policies; the way it was coded, it allows the addition of modules that could support other algorithms, customizing scheduling policies in accordance to the application demands.

IV. FINAL CONSIDERATIONS

Developing multipurpose ES firmware is a complex task since the requirements must meet the intricate application’s demands. It is crucial to use scarce and restrict OSs resources to manage concurrent tasks while offering modularity and scalability features for developers. Low-cost architectures, i.e., the case of a myriad of microcontrollers, do not support to install conventional OSs mainly due to the code size. Our work bridges this gap by offering a customized framework that uses multithreading to reduce the time needed for executing applications at a reasonable computational cost.

The literature offers tools to develop ES applications using low-cost architectures. Our related works have shown interesting examples that provide multitasking; however, the use of threads shifts the burden of reliable implementations towards the developers. We have demonstrated that it is possible to implement a flexible scheduler module for allowing the adjustment of policies according to the application needs where developers change the source code in specific locations. Our approach has shown the execution of a known sorting algorithm where we have applied different scheduling policies with priorities and mechanisms to handle multithreading specifications. It remains to be further investigated other parallel algorithms, problems with many synchronizations and memory accesses.

We envision some future works such as the implementation of other synchronization mechanisms (i.e., semaphores and critical sections) to enhance the utilization of the Waiting List and increase CPU work. One could also consider devising new APIs and high-performance libraries to offer more options to the developers when coding applications targeted to microcontrollers. Moreover, our framework allows adding other scheduling policies and other algorithms, trying to determine the best policy according to the application. One could investigate changing scheduling policy in execution time and also supporting other architectures.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of UNISC’s research council.

REFERENCES


