Advanced AVR Microkernel

Ricardo Fontão
FEUP
Porto, Portugal
up201806317@edu.fe.up.pt
25%

Ivo Saavedra
FEUP
Porto, Portugal
up201707093@edu.fe.up.pt
25%

Diogo Rosário
FEUP
Porto, Portugal
up201806582@edu.fe.up.pt
25%

Gonçalo Teixeira
FEUP
Porto, Portugal
up201806562@edu.fe.up.pt
25%

***Abstract*—In this article, we will explain the steps that were taken to develop a microkernel for the Arduino Uno R3 with the ATMega328P microcontroller. The planned features were context switching, dynamic priorities based on deadlines (Earliest Deadline First - EDF), Mutexes and the Priority-Inheritance Protocol (PIP). At the end of the project, we conclude that all the proposed features were implemented correctly.**

***Index Terms*—Microcontroller, Kernel, AVR, EDF, PIP, Mutex**

I. Introduction

The aim of this project is the implementation of a tick-based dynamic priority multi-stack microkernel for AVR microcontrollers, more specifically for the Arduino Uno R3. A multi-stack microkernel with full preemption was implemented along with context switching, EDF scheduling, and mutexes with priority inheritance protocol.

II. Hardware details

The microkernel implementation discussed works exclusively on the ATMega328P microcontroller since some platform-specific code is present. The specific hardware used for the development was the Arduino Uno R3, which uses the ATMega328P controller. The ATMega328P is an 8-bit microcontroller with 32 KBytes of Flash memory, 1KBytes of EEPROM and 2 KBytes of SRAM. To help with debugging and task visualization, the group used an Arduino hat with programmable buttons and LEDs, as well as some other unused utilities. The Arduino Uno R3 also contains a serial port capable of sending arbitrary strings which can be read by the Arduino serial monitor on a computer.

III. Implementation

The implementation of the microkernel was carried out in the C programming language with resort to the Arduino library from the official Arduino AVR core. The code was split into different components to achieve better organization. Each of these components will be explained in the following sections. For debugging purposes, the PlatformIO was briefly used as well.

A. Multiple stacks

The proposed kernel has full preemption, meaning a higher priority task can interrupt a lower priority one whenever it becomes ready (the only exception is when the lower priority task is holding a mutex, which will be discussed in section III-E). The tasks used are modelled similarly to POSIX tasks, meaning that a single function is responsible for initializing all variables and running the task in a while loop. To accommodate this task model, a single stack for all tasks is not suitable, each task must have a separate stack. Two implementations were considered: each TCB containing the allocated memory for the stack or allocating a global stack, or each TCB containing pointers to the beginning of the allocated space for that specific task. The second option was chosen because it enables the size of each stack to be different without using dynamically allocated memory. The implemented method consists of allocating a big global array which will contain all the tasks' stacks and keeping track in an auxiliary variable of how much space in that stack has already been reserved. When adding a new task, the programmer must specify the stack size to allocate for that specific task. The microkernel then reserves that space in the global stack by incrementing the auxiliary variable.

B. Tasks

To store the characteristics of a given task, the group defined a structure with the following parameters:

- stackPointer (uint8_t) – stack pointer of the current task
- bottomOfStack (uint8_t) – pointer to the bottom of the task's stack in the global stack
- period (int) – task execution period
- delay (int) – initial task delay and then used to control task activation
- deadline (int) – task deadline
- func (void*) – function pointer to the task's function
- exec (int) – boolean indicating if the task is ready to execute
- isIdleTask (int) – boolean indicating whether this task is the idle task
- inheritedDeadline (int) – used in PIP for inheriting the priority of a task with higher priority when a mutex is currently locked
- id (int) – unique task identifier
- blocked (int) – indicates if the task is blocked waiting for a mutex
- numRuns(int) – stores the number of times the task has already run to calculate the absolute deadline
- initialDelay (int) – stores initial delay

As previously stated, it is the programmer's responsibility to specify the required stack size for a specific task. If that size is lower than the actual required size, the program will show unexpected behaviour and probably crash. Besides the stack size, the deadline, initial delay, and period must all be specified when creating a new task. The tasks follow a POSIX-like structure, meaning that a single function is responsible for initializing all variables and running the task in a while loop. At the end of the while loop, the task should yield to allow the scheduler to start the execution of the next task.

C. Context switching

As previously stated, each stack has a separate stack mapped on a global stack. Whenever a higher priority task than the currently running one becomes ready, a context switch is required. The first step in this process is saving the state of the currently running task. This step has to be written in platform-specific Assembly. Saving the current state of execution involves saving all general-purpose registers (R0-R31), the status register, the stack pointer, and the program counter. Each of these values is pushed sequentially to the currently running task's stack. The second step is restoring the context of the task which is scheduled to run next. This is just the reverse process of the first step. All values are popped sequentially from the desired task's stack into the CPU registers.

The first time a task becomes ready is an edge case because there was never a state saved to that task's stack, so there is no state to restore. There are two solutions for this problem: the use of a boolean which is set based on if that task's state was already initialized, or initializing all tasks states with a dummy state. This implementation uses the latter because it makes the scheduler code simpler.

D. Scheduler

The kernel scheduler is set up so that it runs every timer interrupt. Every time this interrupt occurs, the current state is saved to the current task's stack. After that, the scheduler loops through all the tasks and decrements the delay attribute of all tasks. If any reaches 0, that task becomes ready and the exec attribute is set to 1. The next step is dispatching the correct task to resume execution. The scheduling algorithm used in the implementation is Earliest Deadline First (EDF), which means the next task to run is the one with the lowest absolute deadline. The absolute deadline of a task can be calculated with the formula:

$$AbsDeadline = initDelay + nRuns * period + deadline$$

To assert which task should run next, the C *qsort* function is used to sort the tasks array based on the absolute deadline. Finally, the scheduler sets the current task to the first one of the array and restores its context to the CPU. If no task is ready to run, the system fallbacks to running the idle task. This is a special task that doesn't do anything (empty *while* loop).

E. Mutexes

To control access to critical regions, the kernel implements mutexes with the Priority Inheritance Protocol. In this protocol, if a task of lower priority is holding a mutex and a higher priority task tries to lock it, the current mutex holder inherits the higher priority task's priority. This way, the duration of the blocking time is limited because tasks with intermediate priority can't interrupt the lower priority task. A mutex has the following structure:

- isLocked (int) – boolean indicating if the mutex is locked
- holder (task_t*) – pointer to the task currently holding the mutex

All the mutexes are kept in a global array. The programmer is responsible for using the appropriate mutexes in the respective critical regions.

The mutexes provide a simple API with only two functions, *lock* and *unlock*. Whenever a task needs to access a critical region, it attempts to lock the relevant mutex. In the locking process, two things can happen: the mutex is free, in which case the mutex is just locked normally, or the mutex is locked. For the latter, this means that a lower priority task than the one requesting the lock is currently running. When this happens, the mutex holder inherits the priority of the requesting task, thus making it a higher priority, and the requesting task is marked as blocked. After this, the scheduler dispatch is called to resume the execution of the mutex's holder task, performing a context switch. This process occurs inside a *while* loop so that when the holder task finishes, the execution of the higher priority task is resumed, retrying the attempt to lock the mutex. The unlocking operator just releases the mutex if it is locked, and does nothing otherwise.

IV. Validation

To validate the correct working of the microkernel, three different task sets were used. To represent the execution of each task, an LED turns on when it starts running and turns off at the end of its execution.

The first task set developed allowed the validation of the EDF scheduling. The Gantt chart describing this task set, as well as the specification of each task, can be found in Fig. 1.

Task set

τ_i	T_i	C_i
1	3	1
2	4	1
3	6	2.1

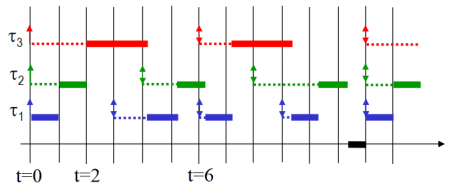

Fig. 1. EDF validation tasks

The second task set is used to validate the simple functioning of mutexes. For this, two tasks are used, the first is a recurring task that checks for a button press. If the button is pressed, it tries to lock a mutex and toggles an LED. The second task is also recurring, but with a longer period that

locks the mutex and holds it for a long time. The idea of this test is that by holding the button, the LED of the first task keeps blinking until the LED of the second task turns on. At this moment, the mutex is locked, and the first task can't lock it to toggle the LED, so it stops blinking.

Finally, the third set of tasks is used to validate the Priority Inheritance Protocol. The Gantt chart can be found in Fig. 2.

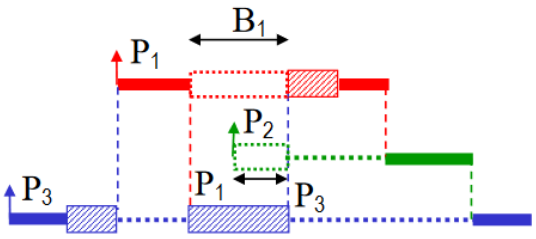

Fig. 2. PIP validation tasks

All of these tests can be seen in the demonstration video at https://www.youtube.com/watch?v=qF9XWb5MAr4.

V. Measurements

TABLE I
TIME MEASUREMENTS

#Tasks	Dispatch	Schedule
1	356	16
2	289.6	16
3	266.4	16
4	251.2	20
5	335.2	24
6	344.8	24
7	373.6	24

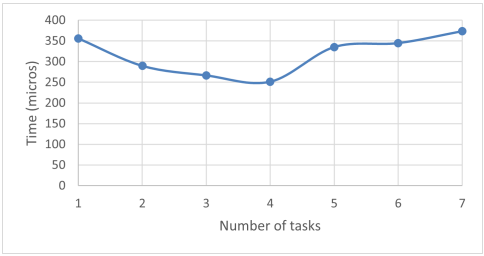

Fig. 3. Dispatcher measurements

To measure the performance of our implementation, we opted on using a software-based approach, due to the time constraints that we faced in the last phases of development. To measure the execution of a single segment of code, we surround it with two calls to store the timestamp in the global timestamp array. In addition to this, we also created a task that periodically checks if the number of defined measurements has already been reached. When this happens, the tasks are charged with sending the values that were measured through the serial port. For this project, we measured the dispatcher as well as the scheduler (section III-D), both with an increasing

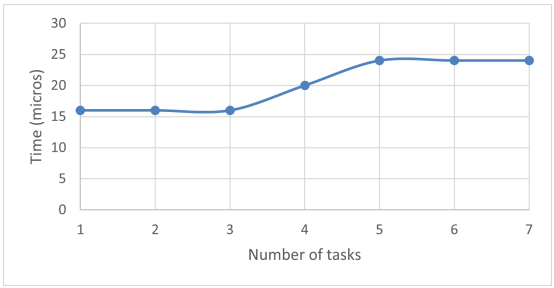

Fig. 4. Scheduler measurements

number of tasks. After receiving every recorded timestamp through the serial port, we make use of a simple python script to calculate the averages of the measurements for each iteration. The results are displayed in table I.

VI. Conclusion

Context switching involves tricking the compiler into thinking everything is running as a single task. This makes it a very hard process to implement correctly. This was one of the hardest parts of the project. FreeRTOS [1] was used as an implementation reference. The software-based measuring also posed a difficulty, mostly due to a lack of experience from the group.

In this project, the group set out to implement a tickless microkernel. The kernel has an EDF scheduler, mutexes with Priority Inheritance Protocol and is multi-stack (context switching). All tasks were successfully completed. Some possibilities for future work include implementing the Priority Ceiling Protocol for mutexes, implementing stack overrun detection using stack canaries and dynamically allocating the stack size for each task.

References

[1] FreeRTOS - Market leading RTOS for embedded systems with Internet of Things extensions.