Prototype of Modular Operating System for Embedded Applications

Martin Vojtko
Faculty of Informatics and
Information Technologies,
Slovak University of Technology,
Ilkovičova 3,
842 16 Bratislava, Slovakia
Email: martin.vojtko@fiit.stuba.sk

Tibor Krajčovič
Faculty of Informatics and
Information Technologies,
Slovak University of Technology,
Ilkovičova 3,
842 16 Bratislava, Slovakia
Email: tkraj@fiit.stuba.sk

***Abstract*—Future complexity and heterogeneousness of embedded systems will result into problems with compatibility and portability of an application software and also of an operating systems. There is need to change a standard architecture of operating systems to reduce these problems. This paper summarises the proposal and implementation results of a Modular Operating System which presents a novel concept of an embedded operating system with emphasis on portability and modularity. New point of view leads into reduced customisation time of the operating system and minimal changes to user applications.**

***Index Terms*—embedded systems, embedded operating systems, hardware abstraction, kernel, portability, modularity**

I. Introduction

According to Moore's law each 24 or 36 months the number of transistors doubles [1]. This causes growing complexity and heterogeneousness of embedded systems. Growing of these factors increases the time needed for integration of software into new embedded systems. The code reusing is harder and software adaptation is more time consuming. The problem of code reusing is reduced when an operating system is used as abstraction layer between hardware and software, but the operating system must be adapted to the new platform. We can affirm that the architecture of contemporary embedded operating systems must be revised.

The standard operating system reduces complexity of the processor and its peripheries [2]. The center of each operating system is the kernel. The kernel of embedded OS manages the tasks, system memory and I/O devices. There are also special types of kernels, named micro-kernels and nano-kernels, which are mostly used in embedded systems [3]. In this work we propose a revision of the kernel concept, because if we analyse the kernel, we can find there platform-dependent and platform-independent parts of code. This fact results into organization of kernel into two layers (see Figure 1). The division of the kernel reduces code changes on platform-dependent parts only, which simplifies the transport from one platform to another. Also it is not necessary to change the platform-independent code, which results into no or minimal changes in the application software.

Besides the portability of operating systems there are other attributes to take into consideration in order to simplify the transfer to other platforms. At the first, there is a need for a modular architecture of operating systems. The layered and modular architecture can be realised together as shown on Figure 1.

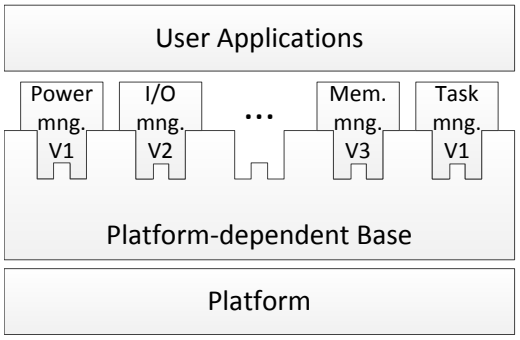

Fig. 1. The layered and modular architecture of embedded operating system. Each module type connects to base trough specific base connector. Also each module can have more variations.

The modules can vary in energy effectiveness, performance, memory footprint etc. As is shown in Figure 1 the OS should be a simple frame, which is filled up by chosen modules which best fit the platform of concrete embedded system. So we propose that OS should not be only one implementation, but it should be a system of services and packages, from which the consumer can choose.

In this paper we present concepts of the Modular Operating System and we provide a brief summary of its architecture. This paper is presenting some related embedded operating systems. We also present results from the testing of the MOS.

II. Related Work

The release of a new generation of embedded operating systems is near so this part of research is lucrative for many researchers. In this paper we include a brief summary of two embedded operating systems from the research.

The TinyOS offers an interesting concept of operating system. This operating system is build on the concept of three hardware abstraction layers. The bottom layer (Hardware presentation layer, HPL) presents

the services of hardware to the middle layer. It consists of modules which encapsulates hardware devices. This layer is platform-dependent. The middle layer (Hardware adaptation layer, HAL) adapts the services of HPL to the top layer. It consist of modules which encapsulates modules from HPL. Modules from the HAL apart from HPL modules can store state of devices. The HAL is platform-dependent too. The top layer (Hardware interface layer, HIL) interfaces the bottom layers into the unified interface. Modules of the HIL encapsulates modules from the HPL. Above the HIL is a layer of platform-independent user applications, which do not change from platform to platform. [4].

The FreeRTOS is a popular embedded operating system. The concept of this system is based on many developed ports of this system on many platforms. The developer of the embedded system can choose the platform port and set-up the basic configuration and then the test system is prepared for use. FreeRTOS community supports new platform port development by very good system documentation [5]. Also the concept of prepared ports and demo applications minimises developing time to a minimum.

III. Architecture of MOS

As we mentioned, the kernel of the Modular Operating System (in short MOS) consists of two layers (Figure 1). The first one is platform-dependent base layer (in short PDBL) and consists of pieces of assembly code which encapsulates hardware into a higher abstraction layer. The second one is platform-independent modular layer (in short PIML) and encapsulates the PDBL into a presentation layer (see Figure 1). PIBL consists of modules which are connected to PDBL by customisable connectors. Other MOS modules and user applications can be found above the kernel (see Figure 2) [6].

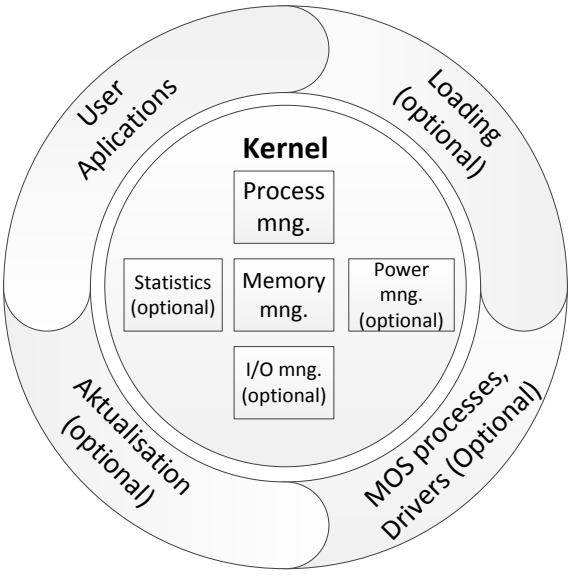

Fig. 2. Architecture of MOS [6].

Modules which are presented in Figure 2 can be implemented in more variations. One variation can be optimized for best energy savings and another can be optimized for best reaction times. Each module is connected to the PDBL by connector.

Compulsory modules of MOS are Process Management and Memory Management. These two modules are the core of MOS and they provide the basic functions of the whole system. It is also possible to connect optional modules which do not have to be used (in Figure 2).

The strength of modular architecture lies in simple change management and development. User can choose from many suggested modules that were implemented by the developer of the OS. We propose to view the OS as a group of many modules in many variations from which the user can combine his own system.

A. Process Management

The task of Process Management module is to manage tasks running in the system. In order to ensure better change management and portability, the process management module is divided into the Task Management sub-module, the Inter-process Communication (IPC) sub-module and the Scheduler sub-module.

Task Management covers all programs and tasks needs. These needs can be divided into task switch handling, stored programs managing and running tasks managing (creating, destroying, scheduling and inter-task communication).

Each task has its own header (Task Control Block TCB) which contains information about the task (see Figure 3). The TCB is encapsulated in the item structure for scheduling purposes. Task-switch which provides extraction of new task context and storing of old task context to its task header or optionally to task stack is in the PDBL.

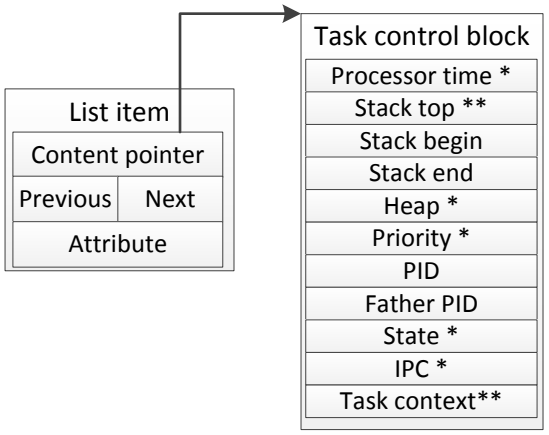

Fig. 3. Task header (task control block). * included in header if configured. ** user can choose if the context is stored in the header or in the stack. *** if priority is configured it gets value of this priority [6].

Programs are stored in program memory. Each program has its own program header called Program Control Block (PCB). PCB stores information about program memory localization, data size etc. (see Figure 4).The PCB is encapsulated in a item of a list structure for managing purpose.

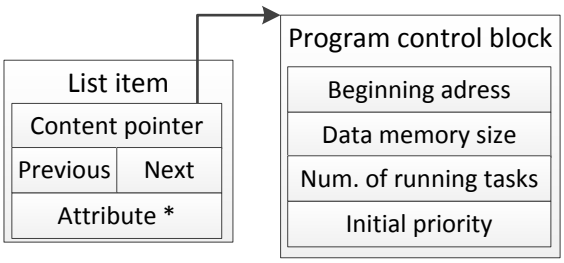

Fig. 4. Structure of the Program Control Block and its encapsulating item. * Attribute set up as program id [6].

The scheduler orders tasks into a waiting list. We prepared three types of the schedulers which differ in concept of the tasks ordering:

- **Round-Robin scheduler**, which pushes switched tasks to the list.
- **Priority scheduler**, which inserts tasks into list in an order based on the priority criterion.
- **Multi-list scheduler**, which pushes tasks based on priority into the concrete priority list.

IPC module manages the creation of the communication channels between two tasks. We use a Sender-Receiver model for task communication. The sender must register for communication with the chosen receiver. Registration is stored in the list of waiting requests. The receiver pops the requests from list and decides if communication will be created. Initialized communication is provided by queue data structure.

B. Memory Management

Memory Management is the second compulsory module. It manages program and data memory of the whole operating system. We decided not to implement a file system for storing data and programs, because it is not important for our research.

Memory can be divided into three partitions. In the first partition the whole program base is stored. The base consists of a MOS code and a user applications code. MOS is in a state were no changes in program memory can be done until the system is running. We plan to implement a functionality which allows an update or an upload of user applications. For actual research it is not important.

Data created by MOS is stored in the second partition. This partition contains the kernel heap, where the program and task headers are stored. This memory is allocated by the kernel's memory allocation call. This partition also contains the kernel stack where context of kernel procedures is stored.

The third partition is the task heap, which stores data of running tasks. Place for data is allocated by kernel call when the tasks are created and it is freed when the tasks are destroyed. The task heap or kernel heap is organized as a list of memory chunks. At the beginning, the heap contains only one chunk, which has the size of a whole heap (see Figure 5).

Task data is divided into the heap and the stack. The stack contains data pushed by procedure call. The user can manage the heap by memory allocation

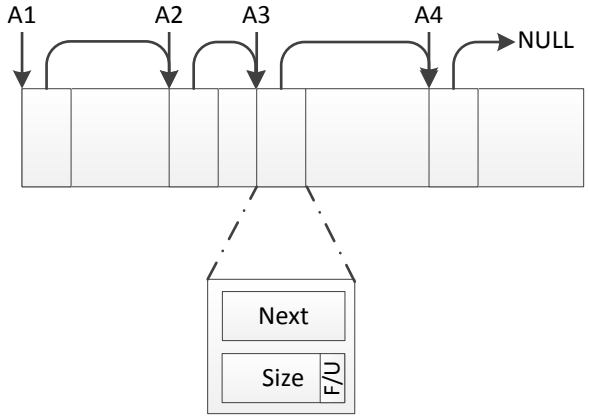

Fig. 5. The heap fragmentation process and structure of the chunk in the heap [6].

call. We decided to implement First Fit algorithm for memory allocation. The memory allocation can be simply changed by implementing another algorithm.

C. Optional modules

MOS can be set-up as a two-layered micro-kernel but apart from the compulsory modules also other modules can be included in MOS.

We decided to implement a Statistical module for testing purposes. This module collects information from the compulsory modules. This information consists of whole system and individual task running time, program memory usage, kernel memory usage and task memory usage. Statistical module only collects and sends data through the serial port to the host PC where they are processed.

IV. Results

We implemented port of MOS on Atmel's processor at91sam7s256 [7] from family arm7tdmi [8]. This processor was embedded in development kit sam7p256 [9]. On this platform we provide analysis of the MOS implementation.

For testing purposes, we prepared three programs which were running at the platform together with MOS Root task in two types of task context storage settings and two types of scheduling algorithms. After system start-up and initialisation the Root is started with priority 14. The Root runs task $T1$ with priority 12 and then task $T3$ with same priority as $T1$. Task $T1$ creates task $T2$ with priority 12 (see Figure 6).

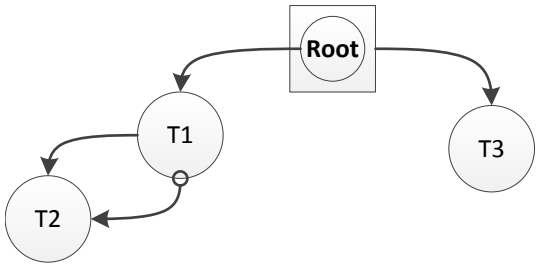

Fig. 6. Graph of starting and communication dependency. Arrow with circle shows communication from $T1$ to $T2$ [6].

TABLE I
MEASURED MEMORY USAGE AND PROCESSOR TIME DISTRIBUTION [6].

Task	Stack memory used with task context in (B)		Heap memory used with task context in (B)		Processor time in (s)	
	in stack	in TCB	in stack	in TCB	Round-Robin	Priority
Root	120	56	0	0	$60,0025$	$90,0176$
T1	88	24	$76+24$	$76+24$	$59,9071$	$44,9283$
T2	120	56	$40+16$	$40+16$	$0,0824$	$0,0629$
T3	80	16	0	0	$60,0080$	$44,9912$
Kernel	36	36	$624+160$	$880+160$	-	-
Total	444	188	940	1196	$180,0$	$180,0$

The Root task collects data for measuring after running tasks $T1$ and $T3$. $T2$ is waiting for data from $T1$. Tasks $T1$, $T2$ and $T3$ are doing simple counting. We were monitoring the work with the system memory and the processing time of tasks (see Table I).

User can chose where will be the task context stored. If it is stored in the stack, the area of task memory is used more. If the task context is stored in Task Control Block, the area of kernel memory is used more (see columns 2 and 3 at Table I).

Results of two different scheduling techniques (see column 4 at Table I) shows that the $T2$ runs on processor small fragment of time used for the others. It is caused by $T2$s waiting for $T1$.

The MOS uses $5.5kB$ of program memory at minimal configuration and $9.5kB$ at maximal configuration. This size is comparable to other embedded systems.

Data memory usage is divided into modules (see Table II). As can be seen data memory usage of MOS is more economical.

TABLE II
DATA MEMORY USAGE OF SYSTEM IN B [6].

Name of user	Used data in B	FreeRTOS usage in B [5]
Scheduler	$17+16$ per task	236
Task minimal	$30+16$ item	64
Task maximal	$116+16$ item	64
Queue	$25+12$ per item	76
List	$25+16$ per item	-

At clock rate $48MHz$ and task switch clock rate $10kHz$, task switch takes $14,3\mu s$ which is $14,3\%$ overhead. This time was measured only with usage of Round-Robin scheduler. Other scheduling techniques are dependent on priority and it means that the tasks must be ordered. This process is not deterministic.

V. FUTURE WORK

Research on MOS is still continuing. There are many fields where the new OS can be oriented. We plan to improve power efficiency of MOS. This improvement begins with exact analysis of MOS power consumption. Consumption can be measured or modelled in many ways. Analysis of these techniques will be done. The OS can influence power consumption by observing and adjusting the system performance.

We also orient our research on MOS in the sphere of multiprocessor and distributed systems. This decision is promoted by spreading of distributed and multiprocessor embedded systems at world trade.

Our plan is to achieve a complete software platform where user can simply set up Modular Operating System according to his needs, without complicated coding and developing. The result of system set up would be binary code prepared for porting on chosen hardware platform.

VI. CONCLUSIONS

In this paper we presented the result of the research on embedded operating systems sphere which we named Modular Operating System. Main goals of research were to implement modular, flexible, portable. Experimental OS is still under research but the first results show that the method which we have chosen was right.

ACKNOWLEDGEMENT

This work was supported by the Grant number VEGA 1/1105/11 of the Slovak VEGA Grant Agency.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *Solid-State Circuits Newsletter, IEEE*, vol. 11, no. 5, pp. 33 –35, sept. 2006.

[2] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, 2006.

[3] J. J. Labrosse, J. Ganssle, and e. a. Robert Oshana, *Embedded Software: Know It All (Newnes Know It All)*. Newnes, 2007.

[4] TinyOS-community, *TinyOS*, 2011, http://www.tinyos.net.

[5] FreeRTOS-community, *FreeRTOS™*, 2011, http://www.freertos.org/.

[6] M. Vojtko, "Modulárny operačný systém pre vnorené systémy (in Slovak)," Master's thesis, Faculty of Informatics and Information Technologies, 2012.

[7] Atmel-corporation, *Datasheet, AT91SAM7S256*, 2010.

[8] Arm-corporation, *Technical Reference Manual, ARM7TDMI (Rev 3)*, 2001.

[9] Olimex-Ltd, *Datasheet, SAM7-S256 development board, Users Manual*, 2008.