An Alternative Implementation for Accelerating Some Functions of Operating System

János Végh[1], Ádám Kicsák[2], Zsolt Bagoly[2] and Péter Molnár[2]
[1]*Faculty of Informatics, University of Debrecen, Kassai Str 26, Debrecen, Hungary*
[2]*PhD School of Informatics, University of Debrecen, Kassai Str 26, Debrecen, Hungary*

Keywords: Operating System, System Call, Hardware Acceleration, Reconfigurable Module, Semaphore, Communication between Processes.

Abstract: Processes running under an operating system are independent and autonomous entities. However, they need to share resources, communicate, use OS services, etc. The operating system's services can be reached through system calls, which contribute – sometimes excessive – overhead activity. In some cases the payload activity, used in the system call, is much shorter than that needed for implementing the Exceptional Control Flow, implementing the system call frame. In certain cases, the OS service in question can be implemented in an alternative way, practically without overhead. The paper presents such a case, using an easy to understand simple example, an alternative implementation of a simple binary semaphore. The semaphore has been implemented and tested in a prototyping environment, using an operating system running on a soft processor equipped with custom instruction. For implementing the semaphores, a reconfigurable device was used.

1 INTRODUCTION

Shortly after inventing the electronic computers, making software for computers became a science on its own right. Most of the software developers are working with a closed hardware (HW) with finished development, and their new ideas and needs might be included (if ever) only in the next generation of hardware. This is why sometimes the goals could only be reached through abandoning obstacles (or simply not foreseen needs) in the architecture of the HW, so the power of the software could not keep pace with the development of the hardware (Ousterhout, 1990). This unfortunate status quo seems to change with the more and more widespread usage of reconfigurable devices (Xilinx, 2012; Altera, 2013; Adapteva, 2014), the end users can develop and test user-defined hardware architectures and softwares running on them.

At the beginning HW accelerators were developed for some well-defined user task. In the past few years several, reconfigurable device based developments have been published for a wide range of tasks in connection with operating systems (OS), see for example (Akesson, 2001; So, 2007; Ferreira and Oliveira, 2009) and references within. Although they reach their intended goal and prove the advantages of acceleration, they only reach partial sucesses. The main reason is that *they simply base their approach on the the traditional methods of computer science, but implemented in hardware. It also means that they are also abandoning the same (in their case virtual) obstacles in the architecture* .

In this paper we re-think the goal of the implementation of some OS services, and show that in some cases a much more effective alternative implementation is possible. We show the advantages of using the inherent features of reconfigurable (RC) components, on the example of implementing semaphores which are extensively used for synchronizing processes of operating systems and which are of utmost importance in real-time embedded systems.

2 THE SOFTWARE SEMAPHORE

The computers with Neumann architecture were originally able to run one single process only. Consequently, that single process could use all available resources of the computer, without restriction.

2.1 Cooperating Processes

The first real challenge appeared with inventing the processor interrupt, and following that, immediately

494 Végh J., Kicsák Á., Bagoly Z. and Molnár P..
An Alternative Implementation for Accelerating Some Functions of Operating System.
DOI: 10.5220/0005104704940499
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 494-499
ISBN: 978-989-758-036-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

appeared the first classic issues. The program with interrupt handling facility can be considered in such a way, that *two processes* are started at the beginning. The main program (the ”core process”) works as usual, the other process (the interrupt service routine) is blocked until the triggering signal from the interrupting device appears. Following that the signal arrives, the interrupt servicing process takes over (gets unblocked) and runs until it terminates. After that, the control returns to the core process, exactly to the place and state, where and how the interrupt occurred.

The two processes need to solve their joint task together, i.e. they need to communicate (to some degree) with each other. This can be realized through *making changes in the state of their shared resources*, typically making changes in the memory. Since both of these processes use the same resources, special care must be taken to *avoid changes exceeding this intended communication*. This means, that both the hardware and the interrupt servicing process must save and properly restore the temporary changes they caused, in order to make unnoticeable for the core process when the control is returned to the main process after servicing the interrupt, that an interrupt occurred.

The operating principle of the computer assures only that the single machine instructions are completed, but in case of operations requiring more than one machine instructions special care must be taken to make sure that the execution of a closely related group of instructions will not be interrupted. *In case of the interrupt service routine the hardware assures that the control is not returned to the core process until the servicing process terminates*. This means, that making non-atomic operations within the interrupt control routine needs no special care or action. On the opposite, *in the core task one has to protect atomic operations* (for example, in an easy but not elegant way) through disabling/enabling interrupts. Notice the asymmetry of the roles of the two tasks.

2.2 The Problem of Communication

In multitasking operating systems there are several ”core” processes and all those processes can be interrupted by the scheduling clock, so resources shared by two (or more) such processes can be safely used only through accepting a kind of agreement between tasks, since (in contrast with the case in previous section) *there is no hardware support for protecting atomic operations*. Namely, the first process reaching the state in which it wants to use a resource, for the period of executing the so called critical section, monopolizes its usage. It does so through changing some area, visible also for other tasks, of the memory, designating that right now it is using the corresponding resource. After finishing the critical activity, it makes the resource free again through resetting the changed memory location. For the other processes needing the same resource, the resource will only be available after the first process makes the resource free again, even if the scheduler would run some other process meanwhile.

Of course, *this mechanism is viable only if both processes keep the agreement* (they communicate with each other before they begin the action), i.e. before using the resource, the process checks whether it is available, reserves the resource (changes the respective memory) and uses the resource only if the resource is free (otherwise waits for the resource), and after using it, releases the resource (resets the memory).

2.3 Communication in Operating Systems

Implementing this kind of communication of processes under an operating system is not really simple. One of the basic tasks of the operating system is to protect the processes from each other, including the memory they use extensively. This also means, that *the changed memory belonging to one task surely must not even be ”seen” by any other task*. If independent processes want to communicate with each other, they need the help of a ”reliable third party”.

The obvious available solution is the OS itself, mainly because the processes of course must be able to communicate with OS in other businesses as well. Those processes that need to change (in cooperation with the OS) the memory content in a region belonging to the OS, and of course in cooperation with the OS they can also have information on the content of that memory. Technically it means that *before and after every single usage of the semaphores the processes must contact the OS, changing operation mode and passing parameters there and back. This happens using software interrupts, which means complete context switching there and back, and means executing up to hundreds of machine instructions* (according to (Bryant and O’Hallaron, 2014), the complete context switching might take up to 20,000 clock cycles!). The so called multitasking operating systems use this kind of synchronization extensively (sometimes even recursively (C. A. Thekkath and H. M. Levy, 1994)), which means a considerable administrative load for the processor, especially if the implemented payload functionality is as simple as in case of handling a binary semaphore.

Such a mechanism, the so called semaphores (and their relatives) are extensively applied in OSs for synchronizing processes. It is worth then to check whether this mechanism could be implemented using a quicker, simpler technology, which is functionally equivalent with the original one. The basic problem here is, that in the traditional computer the only place for storing information is the memory, which in multitasking environment is under special protection. Notice that *the OS has only administrative role here: it is not checked whether the process is legible to use the resource, and similarly, freeing the resource is also not supervised.* After noticing that, one can attempt to invent a more simple and quick mechanism for replacing the current one. The key is to provide another facility for storing the information shared between tasks: semaphore array, implemented using non-Neumann method of operation.

3 THE ALTERNATIVE SEMAPHORE

Note that the reconfigurable module described in this section is strictly needed for making the prototype only. The final solution can either be a configurable component (in order to provide more flexibility for the end-user, as in our prototype system) or non-configurable HW component (in order to provide more comfort), but anyhow something outside the address space of the process(es).

3.1 Implementing the Semaphore

The task of the module is to handle a certain (configurable) number of semaphores. The implementation of the module has a COUNT parameter, which defines the capacity of the module, i.e. how many semaphores can be used simultaneously. This parameter allows to accomodate to the needs of the operating system.

Every single semaphore must be created, be acquired (P operation), released (V operation), and destroyed. Also, the module must be able to give information on possible error situations. Creating and destroying the semaphore are the equivalents of allocating and freeing up memory in the traditional software implementation. In our hardware solution these operations are implemented through setting/resetting a single flag bit within the new HW module. All four operations are executed in one clock cycle, and in this period the eventuell error information is also displayed. The operations (state transitions) and the states of the semaphore are the well-known ones. For the terminology, many different notations and naming conventions exist. The present paper uses the one found in (Li and Yao, 2003), see Fig. 1.

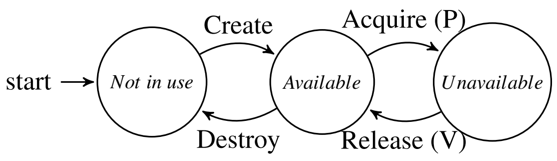

Figure 1: State diagram of a simple binary semaphore.

The state of the reconfigurable semaphores, similarly to the software semaphores, cannot be accessed imediately. Outside of the module only the module interface (see Fig 2) is visible, and only the four mentioned semaphore operations can be used to step from one state to another one. That is, the semaphore represents a completely closed system for other modules.

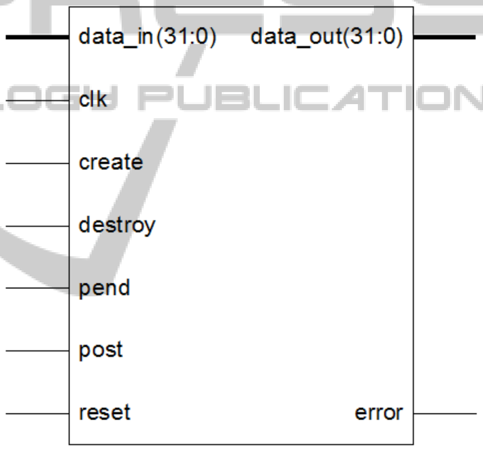

Figure 2: Interface of the semaphore module.

3.2 The Interface of the Module

For operating the module, a clock signal (`clk`) and a reset input line (`rst`) are surely needed. Four control lines (`create`, `destroy`, `pend` and `post`) are used to implement the four operations. For data input and output the 32-bit data buses `data_in` and `data_out` are used. The eventuell errors are displayed in the status line `error`, which shows the status of the last operation. When this line is active, the operation could not be carried out. In such a case the value on the data lines provide information on the nature of the error.

3.3 Testing and Simulation

The correct operation of the module is proved by a carefully assembled series of test cases. In this series all semaphore operations are executed, and all error conditions produced. The test simulates the operation of the module, checks the output signals of the module as well as the correctness of the internal states. The following test cases are included, in the order of the listing:

1. Resetting the module
2. Initializing all semaphores
3. Initializing one more semaphore
4. Putting an arbitrarily selected semaphore out of use
5. Allocating, freeing up and putting out of use the semaphore above
6. Initializing a semaphore
7. Allocating the initialized semaphore
8. Allocating the semaphore again
9. Freeing up the allocated semaphore
10. Freeing up the semaphore again
11. Setting control lines to invalid state
12. Using invalid semaphore ID
13. Resetting the module

4 USING THE MODULE

The module described above is a closed hardware unit, with functionally equivalent with the traditional software-implemented semaphores.

4.1 Linking the Module to the CPU

After having implemented the module, we had to decide how to link it to the CPU. It could be linked using a kind of ”HW/SW API” (So, 2007) , a kind of ”co-processor” (Ferreira and Oliveira, 2009), etc. Those solutions (as well as practically all kinds of hardware accelerators) use their modules as a kind of I/O peripheral. This would provide a simple and easy solution, but handling I/O devices under OSs is only possible through using OS system calls (i.e. adding considerable overhead due to the mandatory use of the context switching). This way of implementation is only worth if the functionality implemented in HW is complex and lengthy, so the overhead needed for using the module from OS can be neglected. For our simple bit testing and setting it is not the case.

Because of its simplicity, the another reason that the overwhelming majority uses that way of linking is, that for a closed CPU unit it would be the only reasonable way for implementing some CPU-related functionality. Fortunately, when using so called ”soft processors”, some other ways are also available for adding extra functionality to the CPU.

4.2 The ”Soft Processor”

The soft processors are CPUs implemented in a reconfigurable device by the end users. When using reconfigurable technology for prototyping, a high degree of customization is available for the end-user. For this implementation, the Nios II soft processor (Altera, 2011) was used. From our point of view, its very advantageous feature is that it has so called ”custom instruction”s. These are ”empty” CPU instructions, the implementation of the functionality of which is left for the user. The implemented semaphore module – using a proper interface – can be linked to the CPU, and so the semaphore functionality can be reached through using a CPU instruction (apparently, as a C language function). (*Recall here, that the semaphore handling does not need any functionality from the OS, just the safe storage place; so in this way the handling can be carried out directly from the user space, without the overhead of changing context to and from the kernel.*)

4.3 The ”Soft OS”

For testing the operation of the semaphore module, implemented as a CPU instruction, an operating system running on that CPU is needed. Fortunately, the highly configurable $\mu COSII$ (Micrium, 2012) operating system is available for Nios II. Since it is available also in form of modifiable source, it makes easy to implement the semaphore wrap functions in different forms. Actually, three versions of the wrap functions has been prepared. In addition to the one delivered with $\mu COSII$, wrappers for the HW semaphore module linked to the CPU as I /O device as well as with custom instruction has been implemented.

5 BENCHMARKING THE MODULE

5.1 Principles

The different semaphore wrappers can be built into different versions of the ”soft OS”. Bechmarking OS functionality is a sensitive area, so maximum care must be excercised. When running a SW in an OS, resource sharing (including CPU time) takes place, so the normal benchmarking makes sense only, if the possible non-payload activity (including running other tasks and system activity) is negligible. When measuring small (below the msec range) time, the usual (OS based) timing methods cannot be used. Even when counting the processor clock cycles, a correction must be made for the possible exception handling, interleaving the payload activity.

In the case of applications, one can start the cycle counter before starting the application, or within the application code. When making measurements on the OS itself, there is no such an obvious possibility. One needs a special measuring framework, which is able to build into the OS and carry out the measuring functionality there. The effect of scheduling can be accounted for, switched off or even the interrupts disabled for the time of measuring. The subtleties of measuring time in a computer system are excellently discussed in the former edition of (Bryant and O'Hallaron, 2014). In our case an extra difficulty is that both Neumann-style architecture elements and simple combinational circuits are included in the test.

5.2 The Setup

The benchmarking setup consists of a NIOS-II soft processor (equipped with custom instructions implementing the semaphore operations) built with Altera SoPC builder, running a $\mu COSII$ operating system, so actually a benchmarking SoPC configuration is used. (In all cases the pre-configured reference project settings were used.) The available OS profiling facilities can be used to measure execution times for both the unmodified semaphore, implemented in traditional SW, and – using alternative function names – the semaphore implemented in HW. A further test is to measure the semaphore implemented in HW through I/O interface.

The timing facilities of the operating systems are prepared for measuring "macroscopic" times (typically several milliseconds). The $\mu COSII$ platform is equipped with excellent benchmarking functionality, but they are prepared for system and process level timing. On platform $\mu COSII$, depending on the setting of the base time, the timer interrupt will occur at every 10-200 ms; this period will also be used for scheduling. As we learned from the measurements made by (Bryant and O'Hallaron, 2014), this allows for untolerable precision and accuracy when making measurements in the range of just a couple of clock periods. Even in case of measuring the execution time of complete processes, special care must be exercised. This timing facility could be (and will be) used in a latter phase, when testing the accelerating effect of the different semaphore implementations on the execution time of complete, synchronizing applications.

So, we had to look for another timing facilities, closer to the HW. Fortunately, our Altera HW platform supports this kind of activity with two standard modules, the Altera Performance Counter and Altera Interval Timer. The Altera Performance Counter IP (Intellectual Property) core allows to support benchmarking/profiling at HW level. The Performance Counter counts – according to its documentation – the clock periods in a 64-bit counter. The C macro controlled counter uses just a few instructions for operating the counter, much less than the other micro-scale timing codes. In this way one can measure the time through counting clock periods. The Altera Interval Timer – according to its documentation – counts 10 μs intervals, so it looks like better suited for measuring single, system-level activities.

5.3 Calibration and Validation

For our initial goals, only the Altera Performance Counter looks to be suitable. At the beginning, we wanted to find out the time needed to handle the 64-bit counter in the Altera Performance Counter. In principle, the difference of two consecutive readings to that 64-bit counter, with no instruction between, should result the "cost" of reading the counter; a correction factor needed to measure a period with precision of clock periods. Those measurements results that reading the counter needs ≈ 200 clock periods, with a few periods variation. For the first look, it seems pretty expensive for a simple counter reading, and also the variation of the length of that period also implies the presence of some unknown factors.

Anyhow, at the moment this is the best facility for measuring the execution time related to the semaphore. The method provides consistent results, but with unsatifying precision. This is why not yet engineering-style results are presented. Development work to carry out our own timing solution is in course.

5.4 Preliminary Results

The alternative implementation of semaphores provides a timing task for testing the implementation itself, as well as measuring the effect of using the alternative implementation versus the traditional implementation. This exactly corresponds to the microscopic and macroscopic time scales, mentioned in (Bryant and O'Hallaron, 2014). These two time scales require different methods and different timing facilities.

On microscopic scale, we wanted to measure the time needed to execute simple semaphore operations.

We used a single run method, and a loop run method. When measuring the semaphore operation in a single run, we measured about ≈ 150 clock cycles. To make a reasonable measurement, in a loop acquiring and releasing a semaphore (10,000 times) was run, and the measured total time was divided by the number of the loop cycles. The result was about ≈ 100 clock cycles. The two results are consistent, considering the results of calibration and validation. (Of course, one has to take care of the number of interrupts, the number of context switches, etc.)

When evaluating these results, one must recall that a simple counter reading takes ≈ 200 clock cycles. Although the module itself needs 1 clock cycle only, the offset of implementing it as custom instruction should have some offset, in the order of magnitude of reading a counter. In order to make more detailed and accurate measurements, further studies and more information on the timing facilities needed. (We guess that the Performance Counter was designed for much more complex functionality, and is implemented in a form too complex for our goals.)

The same measurements have been carried out also on semaphores implemented using the traditional SW method. The measurements consequently show that the HW implementation is ≈ 30 times quicker than the traditional one. We guess that using a better support from the processor, this ratio can be reasonably higher.

Note that there are several attempts (like (Li et al., 2011; Akesson, 2001; So, 2007; Ferreira and Oliveira, 2009)) to implement more or less functionality of the (mainly real-time) operating systems in HW. Their functionality is either more complex or connected with some other functionality, or use a different method of linking to the CPU, or do not provide timing data, so a direct comparison with them is not possible.

6 CONCLUSIONS

Processes running under operating systems are using a lot of operations, which are needed for the safe and/or comfortable operation rather than for the payload work. The extensive use of such operations in multitasking operating systems reduces the payload computing power. In case of using an external hardware module properly linked to the CPU and running parallel with it provides the possibility to implement such operations on different principle, in a considerably more effective, but functionally equivalent way. The article presents how communication of processes in one clock period only can be implemented, possibly replacing hundreds of machine instructions. The semaphore implemented on this principle has been built (in reconfigurable device) and tested. Initial benchmarking results are also presented.

The present prototyping implementation – as a proof of concept – shows that it is possible to considerably increase some OS-related functionality through implementing it as a HW module, running parallel with the CPU. Based on this, more complex OS-related functionality could and should also be implemented. Following the same idea – considering the original intention rather than simply re-implementing the traditional SW solution in HW – the speed of the execution of other tasks in OSs can be considerably enhanced.

REFERENCES

Adapteva (2014). Parallella Board. http://www.parallella.org/.

Akesson, J. F. (2001). *Interprocess Communication Utilising Special Purpose Hardware*. PhD thesis, Uppsala University.

Altera (2011). Nios II Processor Reference. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

Altera (2013). DE2 board. https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=30.

Bryant, R. E. and O'Hallaron, D. R. (2014). *Computer Systems; A Programmer's Perspective*. Pearson.

C. A. Thekkath and H. M. Levy (1994). Hardware and software support for efficient exception handling. In *ASPLOS-VI Proceedings*. DOI: 10.1145/195470.195515.

Ferreira, C. M. and Oliveira, A. S. R. (2009). *RTOS Hardware Coprocessor Implementation in VHDL*, page 6. Intellectual Property / Embedded Systems Conference (IP/ESC).

Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. CMP Books.

Li, Y., Gu, P.-P., and Wang, X.-X. (2011). The implementation of Semaphore Management in Hardware Real Time Operating System. *Information Technology Journal*, 10(1):158–163.

Micrium (2012). μC/OS-II. http://micrium.com/rtos/ucosii/overview/.

Ousterhout, J. K. (1990). Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conference*.

So, H. K.-H. (2007). *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, University of California, Berkeley.

Xilinx (2012). Zedboard. http://zedboard.org/.