Microprocessors and Microsystems 38 (2014) 226–232

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Hardware support for memory protection in sensor nodes

Lanfranco Lopriore

Dipartimento di Ingegneria dell'Informazione, Università di Pisa, via G. Caruso 16, 56126 Pisa, Italy

ARTICLE INFO

Article history:
Available online 22 February 2014

Keywords:
Access right
Privileged mode
Memory protection
Sensor node

ABSTRACT

With reference to the typical hardware configuration of a sensor node, we present the architecture of a memory protection unit (MPU) designed as a low-complexity addition to the microcontroller. The MPU is aimed at supporting memory protection and the privileged execution mode. It is connected to the system buses, and is seen by the processor as a memory-mapped input/output device. The contents of the internal MPU registers specify the composition of the protection contexts of the running program in terms of access rights for the memory pages. The MPU generates a hardware interrupt to the processor when it detects a protection violation. The proposed MPU architecture is evaluated from a number of salient viewpoints, which include the distribution, review and revocation of access permissions, and the support for important memory protection paradigms, including hierarchical contexts and protection rings.

1. Introduction

In the architecture of a sensor node, stringent limitations exist in terms of hardware complexity [1,17]. These include the absence of a memory management unit for virtual to physical address translation, and the lack of processor support for the two traditional execution modes, a kernel, privileged mode and a user mode with memory access limitations [6,9,13]. This means that the operating system kernel and the user programs share a single address space [2]. Each program has unlimited access to the whole primary memory, as well as to all hardware resources, including input/output devices, timers, sensors and actuators.

Owing to the lack of memory protection, program errors and illegitimate program behavior have a potential for corrupting system integrity [7]. The kernel is especially vulnerable. Every program is in a position to access the information items strictly reserved for the kernel, e.g. the cryptographic keys. Consequently, an erroneous or deliberately harmful program can disrupt the whole node [12,21]. In a sensor network, programs are not prevented from using the cryptographic keys of the network protocols [23]; damages may ensue that cross the node boundaries and span a possibly large network fraction. The problem of programming errors is exacerbated by the fact that the writing of application software for sensor nodes is a quite complex task, owing to the requirements for real-time response and support for concurrency, the necessity to comply with different classes of sensors and actuators, and the stringent limitations in terms of memory space, processing power and energy consumption [4,16].

Partial solutions to these problems have been proposed in the past. In [22] software execution in isolated compartments is obtained by extending the microcontroller with ad hoc hardware in the form of a memory protection unit (MPU) that partitions the memory into up to 128 segments of variable size. The MPU is interposed between the processor and the memory modules; it generates an interrupt to the processor in case an access violation is detected. The variable segment size increases MPU hardware complexity. No privileged mode of execution is supported. In [5] a form of protection against control flow attacks is proposed that is based on utilization of a separate stack, called the *return stack*, contained in a memory location different from the normal execution stack. The return stack is reserved for storage of the return addresses. Protection of the return stack against accidental or deliberately harmful alterations relies on dedicated hardware, and implies a modification of the call and return machine instructions. In [7] the problem of preventing memory corruption from erroneous applications is dealt with in a hardware/software co-design approach to memory protection that relies on processor core enhancements, e.g. in the implementation of the store, call and return instructions. A memory map checker is interposed between the processor and the data memory, and is aimed at validating the memory accesses at the hardware level. Protection domains are supported, but no mechanism prevents an erroneous module from corrupting its own state, which resides completely within a single protection domain.

In contrast, with reference to the typical hardware configuration of a sensor node, this paper presents the architecture of a memory protection unit that has been designed by taking the following security and architectural requirements into account: (i)

E-mail address: l.lopriore@iet.unipi.it

http://dx.doi.org/10.1016/j.micpro.2014.01.004
0141-9331/

the complexity of the MPU architecture should be low, so that the cost of the additional hardware is kept to a minimum (e.g. much lower than that of a traditional memory management unit with the address translation portion removed); (ii) the inclusion of the MPU into the existing system should imply no modification of the architecture of the microcontroller, e.g. the instruction set; (iii) a privileged mode should be provided to encapsulate the memory area reserved for the kernel and make this area inaccessible to user programs; and (iv) a separation of the memory access privileges of different portions of the same software module should be supported.

In our protection model, the primary memory is partitioned into fixed-size pages. Memory protection is based on protection domains specifying collections of access rights for the memory pages. In the privileged mode, reserved for the kernel, memory protection is disabled. This means that the kernel routines are allowed to access the whole primary memory.

The rest of this paper is organized as follows. Section 2 presents the architecture of the MPU. A set of system commands is introduced, the *MPU commands*, which allow the running program to modify the composition of the protection domains in a strictly controlled fashion. Special attention is paid to the transition to the privileged mode that ensues when an interrupt handling routine is executed. Section 3 discusses the proposed MPU architecture from a number of salient viewpoints, including the distribution, review and revocation of access permissions, and the support for important memory protection paradigms, including hierarchical contexts and protection rings. Finally, Section 4 gives concluding remarks.

2. MPU architecture

As anticipated in Section 1, in our protection model, fixed-size pages are the basic unit of memory protection. A protection domain is expressed in terms of a collection of access rights for the memory pages; the possible access rights are READ and WRITE. When a program is running, it is assigned a *protection context* defined in terms of a set of protection domains. The protection context limits the extent of the program activities in the primary memory to the access rights in the component domains. The MPU supports two protection contexts for each program, a *global* context and a *local* context. The global context states the memory pages that can be accessed by the program as a whole. The program is free to limit the extent of the global context to form a local context for each program component, e.g. a subroutine. The local context is defined in terms of a subset of the domains in the global context. Thus, the global context enforces protection at program level; a program cannot violate the boundaries of its own global context to access the private memory pages of the other programs or the system kernel. On the other hand, the local context makes it possible to restrict the activity of the given program component to a fraction of the pages in the global context. Local contexts will be defined by taking the *principle of least privilege* [19] into consideration, according to which each software component should be assigned the most restricted set of access rights that allows this software component to carry out its job successfully.

2.1. Protection domains

We shall refer to a typical microcontroller configuration featuring a simple processor and a few kilobyte primary memory (Fig. 1). The primary memory space is logically partitioned into p fixed-size pages. The MPU is connected to the system buses. Each time the processor generates a memory address, the MPU captures this address and the specification of the memory access mode (for read or write). If the MPU detects that the running program does not hold an access privilege congruent with the intended access, it generates a hardware interrupt to the processor. Thus, an interrupt from the MPU corresponds to an *exception of violated protection*. MPU interrupts are non-maskable, and have the highest priority.

The MPU hardware supports d protection domains by associating a *page register* with each memory page. Fig. 2 shows the configuration of the page registers in a system featuring a three-page primary memory and four domains. Page register PR_i associated with page i is partitioned into two protection fields, corresponding to the two access rights, READ and WRITE. In the read protection field, the ith bit, if asserted, specifies that access right READ is included in the ith protection domain. This is similar to the write protection field for access right WRITE. It follows that the composition of the ith domain is expressed by the contents of the ith bit of the two page protection fields of all page registers.

In the configuration of Fig. 2, domain D_0 includes access right READ for page 0. Domain D_1 contains both access rights READ and WRITE for page 1. Domain D_2 contains both access rights READ and WRITE for page 2. Domain D_3 contains access right READ for the three pages.

A protection context is a set of one or more protection domains. At any given time, an MPU register, the *global context register* (GCR), specifies the composition of the global context of the

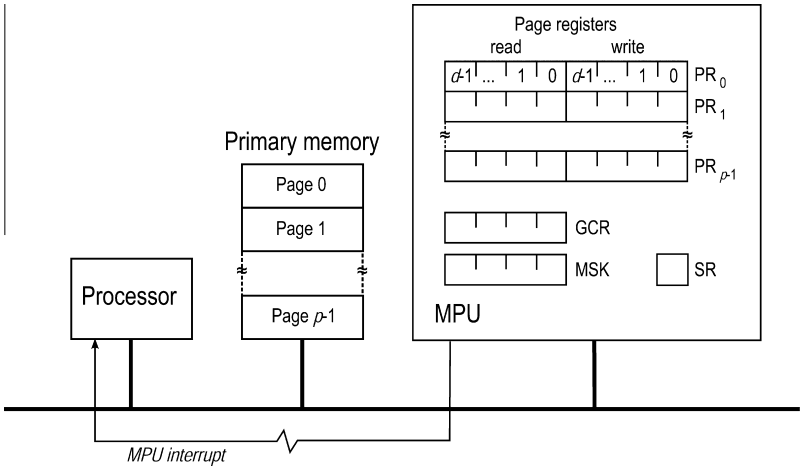

Fig. 1. Hardware configuration featuring a processor, a primary memory, and a memory protection unit that is connected to the system buses and can generate hardware interrupts to the processor.