Page Protection in Multithreaded Systems

Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, Università di Pisa
via G. Caruso 16, 56122 Pisa, Italy
Email: l.lopriore@iet.unipi.it

***Abstract*—With reference to a classical address translation scheme supporting the notion of a paged virtual address space, and a program execution environment in which programs are allowed to have multiple concurrent threads of execution, we present a low-cost addition to the usual hardware inside the memory management unit aimed at supporting page protection at the thread level. The resulting protection system makes it possible to define several distinct protection domains within the boundaries of the same virtual space. Different threads of the same process can have different domains, and the access rights of a given thread can change dynamically as a consequence of actions of amplification and reduction of access privileges, so that at any given time the running thread is given the smallest set of access rights that is necessary for that thread at that time to carry out its job.**

Rather than protecting applications from software attacks of malicious code, our protection environment is aimed at limiting the consequences of programming errors, for instance, when an otherwise secure program is extended by the addition of unverified foreign code, e.g. a plugin that is prone to corrupt and even crash the main process, or a device driver executed in the same virtual space as the operating system kernel. We do not force the user to adhere to a specific protection model. Instead, our protection system features a set of hardware/software mechanisms that makes it possible to implement different protection paradigms at little effort.

***Index Terms*—access right, memory protection, page, process, protection domain, thread**

I. Introduction

Let us refer to a program execution environment in which programs are allowed to have multiple concurrent execution threads. Interactions between the threads of the same father process are common, whereas interactions between the threads of different processes are comparatively rare, and consequently, efficiency in these interactions is not a stringent requirement. In a classical address translation scheme supporting the notion of a paged virtual space, all the threads generated by a given process share a common pool of virtual pages, which is private to that process. This makes it easy to implement forms of cooperation between threads. On the other hand, a thread of a given process cannot even reference a page in the virtual space of a different process, and this guarantees protection against page references crossing process boundaries.

The salient advantage of multithreaded programming is the ease of program structuring into independent execution flows, which is obtained without incurring the run-time overhead connected with context switches between processes [17]. However, heavy protection problems follow from a single address space which is shared by all the threads. Let us consider an otherwise secure application program that supports extensibility through the addition of plugins implementing specific functionalities. This is only an example of a wide class of software systems that support the notion of dynamic extensibility [3]. If the main application and the plugins share a common address space, we are in the presence of a security hole. The main application and the plugins may well be programmed by independent software developers. In fact, plugins are unverified foreign code that is prone to corrupt the main process and even to crash the application [15], [19]. Similar problems are connected with device drivers. These software components are usually executed in the same virtual space as the operating system kernel. This means that, rather than being enforced by the protection system, protection of the kernel space is based on programming conventions [16], [18]. This problem is exacerbated by the fact that drivers are often developed by device manufacturers that have inadequate knowledge of the kernel structure. As a result, drivers have been recognized as a main source of operating system failure [4], [8]. This paper aims at proposing a solution to these protection problems. Essentially, we limit the extent of the memory area which can be addressed by the threads of the same given process so that each thread can access only a subset of the pages of the virtual space of its father process.

In our protection paradigm, a *protection domain* is a collection of access rights to the virtual pages. Two actions are possible on a virtual page, to read the page contents and to modify these contents. These actions are made possible by the access rights *read* and *write*, respectively. At any given time a protection domain is associated with the thread that is running at that time. This domain, called the *active* protection domain, identifies the pages of the virtual space of the father process that the thread can access and the actions that the thread can perform on these pages. As a consequence of address space separation between processes, a thread has no access right to the pages of the virtual space of any process except its father process. When the running thread performs an access attempt to a given page for read or write, the protection system ascertains whether the active domain con-

doi:10.4304/jcp.5.9.1297-1304

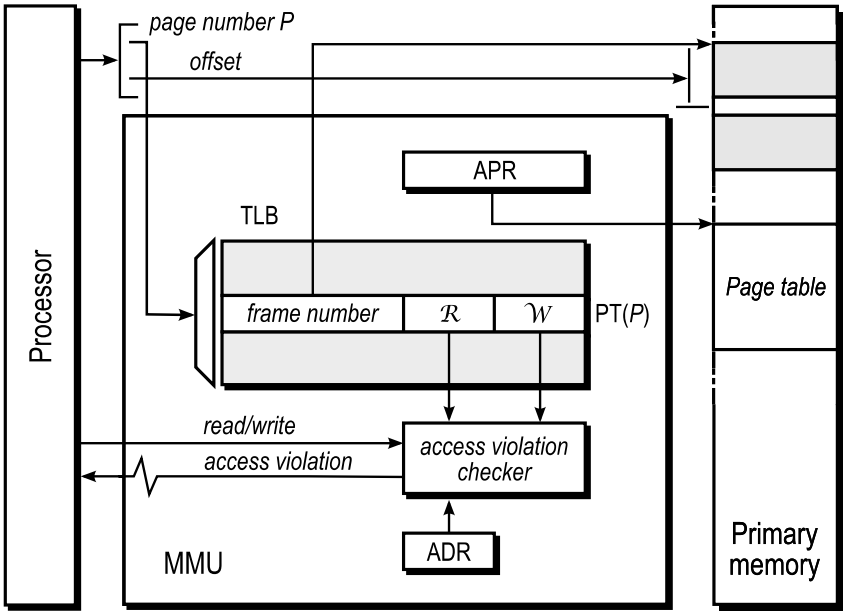

Figure 1. System configuration featuring a memory management unit (MMU) interposed between the processor and the primary memory.

tains the access right to this page that is required to accomplish the access successfully. If this is not the case, an exception of protection violation is raised to the processor and the access terminates with failure.

We shall introduce a low-cost modification of the usual hardware inside the memory management unit aimed at supporting the page protection model outlined so far. The resulting protection system makes it possible to define several distinct protection domains within the boundaries of the same virtual space, so that different threads can run in different domains. Rather than protecting applications from software attacks of malicious code, our protection environment is aimed at limiting the consequences of programming errors. This means that we are concerned with safety rather than security [10].

Essentially, our system is aimed at giving run-time support to the implementation of the *principle of least privileges* [12], so that at any given time the running thread is given the smallest set of access rights that is necessary for that thread at that time to carry out its job. This not only means that different threads of the same process may be given different sets of access rights, but also that the access privileges of a given thread are prone to change dynamically, as a consequence of actions of amplification and reduction of access rights that follow from the changes to the extent of the data areas on which the thread is called to operate.

Thus, a main application and its extensions may well share a common virtual space; however, the extensions run in a protection domain smaller than that of the application. If protection domains are carefully designed, an approach of this type is able to prevent extensions from corrupting the memory space of the main application. Of course, a similar result could be well obtained by assigning extensions a separate address space, at the high time cost of the repeated actions of context switch. In fact, our protection mechanisms support a form of protected procedure calls without incurring the time overhead connected with virtual space changes.

At the hardware level, our design effort has been guided by a main objective, to keep the cost of the protection hardware low. Rather than introducing access right caching within the memory management unit in the form of such a complex circuitry as a protection cache [14] or a protection lookaside buffer [20], we extend the page table to contain protection information that can be handled by a traditional architecture of a translation lookaside buffer.

The rest of this paper is organized as follows. Section II introduces a low-cost addition to the hardware inside the memory management unit aimed at supporting page protection, and the software primitives to control the protection hardware. Our protection environment does not force the user to adhere to a specific protection model. Instead, a set of hardware/software mechanisms makes it possible to implement different protection paradigms at little effort. This issue is considered in depth in Section III. Finally, Section IV summarizes the most important features of the approach to page protection presented in the previous sections. The focus is on the low hardware cost of the proposed protection hardware and the significant flexibility of the resulting protection environment to support different strategies of error confinement.

II. The Protection System

Let us refer to a system configuration featuring a memory management unit (MMU) interposed between the central processor and the primary memory (Fig. 1). The processor references a virtual space that is partitioned into fixed-size *pages*. The physical memory space is logically divided into primary memory *frames* whose

size is equal to the page size. An address generated by the processor is partitioned into a *page number* and an *offset*. The page number is used to access a *page table* in the primary memory and select the page table entry corresponding to the referenced page. This entry contains the number of the primary memory frame storing the referenced page. The frame number is paired with the offset to obtain the physical address of the referenced information item in the primary memory.

The page table is process-specific. At any given time, a register of the memory management unit, the *active process register* (APR), contains the primary memory address of the page table of the process that is running at that time. Inside the memory management unit, a cache of the page table, the *translation lookaside buffer* (TLB), supports fast address translation [1]. On the occurrence of a memory access, if the memory mapping information corresponding to the page referenced by the processor is contained in the TLB, virtual to physical address translation can be accomplished within the MMU without accessing the page table, whereas in the case of a TLB miss the page table entry corresponding to the referenced page will be fetched from the primary memory to the TLB. When a process releases the processor and a new process is assigned the processor, an address space change takes place that invalidates all the memory mapping information contained in the TLB. The primary memory address of the page table of the new process is loaded into APR. The usual caching mechanisms will reconstruct the contents of the TLB for the new process at the cost of the time necessary to access the page table of the new process in the primary memory. This is not the case for a context switch between threads of the same process, as the address space does not change.

A. Hardware Support

We enforce separation of address privileges between the different threads of the same process by extending the page table and the TLB to contain protection information, as follows. Let PT_P be the page table entry corresponding to a given page P. Besides the name of the frame storing this page in the primary memory, PT_P contains two protection fields, called *read protection* ($\mathcal{R}$) and *write protection* ($\mathcal{W}$). The size of the protection fields is fixed and is equal to the number n of the *basic domains* BD_0, BD_1, …, BD_{n-1} that the protection system defines for each virtual address space. Let $\mathcal{R}_i$ and $\mathcal{W}_i$ denote the i-th bit of $\mathcal{R}$ and $\mathcal{W}$, respectively. If asserted, $\mathcal{R}_i$ specifies that basic domain BD_i includes the read access right for page P, and similarly for $\mathcal{W}_i$ and the write access right.

At any given time, a register of the memory management unit, called the *active domain register* (ADR), specifies the extent of the domain that is active at that time. The size of ADR is n bits, one bit for each basic domain. Let ADR_i denote the i-th bit of ADR. If ADR_i is asserted, then the active domain includes all the access rights that are part of basic domain BD_i. It follows that the active domain is the result of the logical union of the basic domains corresponding to the bits of ADR which are asserted. If, for instance, only one bit of ADR is asserted, say ADR_i, then the active domain is formed by a single basic domain, BD_i, and contains only the access rights in BD_i. If ADR contains the all-one bit pattern, then the active domain is the result of the logic union of all basic domains, and contains all the access rights in these domains.

When the processor references page P, entry PT_P is selected. If the page access is for read, the bitwise AND of the contents of ADR and the read protection field $\mathcal{R}$ of PT_P is evaluated. If the result is 0, then no basic domain in the active domain contains the read access right for page P; an exception of access violation is generated to the processor and the access fails. If this is not the case, the access is accomplished successfully. If the access is for write, the contents of the write protection field $\mathcal{W}$ are involved in the check.

Thus, at any given time the register pair {APR, ADR} univocally identifies the active process and the protection domain of the thread that is running at that time. From the point of view of the physical memory, these two registers together identify the primary memory frames that the processor can access successfully. When the processor is switched from thread *T'* to thread *T"* of the same process, ADR is accessed as part of the actions involved in the switch. The contents of this register are replaced with the bit configuration corresponding to the domain of the new thread, *T"*. On the other hand, as both threads share a common address space, the contents of APR do not change and validity of the contents of TLB is preserved. In this way, we obtain a form of protection domain separation between the threads of the same process while keeping the execution time cost connected with thread switch as low as the loading of a new bit configuration into ADR.

B. Protection Primitives

The software interface of the protection system consists of a set of primitives, the protection primitives, that allows us to carry out the page protection activity delineated so far. By using these primitives, we shall be able to access and modify the active domain register ADR and the protection fields $\mathcal{R}$ and $\mathcal{W}$ of the page table entries. These primitives and the actions caused by each of them are summarized in Table I. We shall now discuss these actions in detail.

Operation *makeActive*() makes a given domain active; this domain is expressed in terms of the basic domains that compound it. This operation has the form

makeActive(*d*)

Quantity d is an n-bit configuration featuring one bit for each basic domain (i.e., the size of d is equal to the ADR size). For each bit in d, say the i-th bit, if this bit is asserted, then the active domain will include basic domain BD_i. Execution of this operation loads quantity d into ADR. It follows that after execution of this instruction the active domain is the result of the logical union of the ba-

TABLE I
PROTECTION PRIMITIVES

Primitive	Action
makeActive(*d*)	Activates a domain including the basic domains corresponding to the bits of *d* that are asserted.
copyAR(*P*, AR, *i*, *j*)	Grants BD_j the subset included in BD_i of the access privilege for page *P* that is specified by AR.
clearAR(*P*, AR, *i*, *j*)	Deprives BD_j of the subset included in BD_i of the access privilege for page *P* that is specified by AR.

sic domains corresponding to the bits of *d* which are asserted.

Operation *copyAR*() grants an access privilege for a given page to a basic domain. This operation is as follows:

copyAR(*P*, AR, *i*, *j*)

Quantity AR specifies an access privilege for page *P* in terms of the read access right, the write access right, or both. This operation grants access privilege AR to basic domain BD_j. If basic domain BD_i includes only a subset of AR for page *P*, this subset is granted. Execution accesses the entry PT(*P*) reserved for *P* in the page table. If AR specifies the read access right and the *i*-th bit $\mathcal{R}_i$ of the protection field $\mathcal{R}$ of PT(*P*) is asserted, then bit $\mathcal{R}_j$ is set, and similarly for the write access right, bit $\mathcal{W}_i$ and bit $\mathcal{W}_j$.

Finally, operation *clearAR*() clears an access privilege for a given page from a basic domain. This operation is as follows:

clearAR(*P*, AR, *i*, *j*)

Quantity AR specifies an access privilege for page *P*. This operation clears access privilege AR from basic domain BD_j. If basic domain BD_i includes only a subset of AR for page *P*, this subset is cleared. Execution accesses the entry PT(*P*) reserved for *P* in the page table. If AR specifies the read access right and the *i*-th bit $\mathcal{R}_i$ of the protection field $\mathcal{R}$ of PT(*P*) is asserted, then bit $\mathcal{R}_j$ is cleared, and similarly for the write access right, bit $\mathcal{W}_i$ and bit $\mathcal{W}_j$.

III. PROTECTION MODELS

We shall now show how the protection mechanisms introduced in the previous sections can be used in several examples of practical application. In fact, our protection environment does not force the user to adhere to a specific protection model. Instead, the protection system features a set of hardware/software mechanisms that makes it possible to implement different protection paradigms at little effort.

A. Disjoint Domains

In a protection model using disjoint protection domains, each thread of a multithreaded process runs in its own domain. In a well-structured design, according to the principle of least privilege, this will be the smallest domain that allows the thread to carry out its work successfully. In our system, a result of this type can be obtained by reserving a basic domain for each thread. The basic domain reserved for a given thread will include the read and/or write access rights for each page that the thread should access, and no access right for every other page.

Let us refer to the classical problem of a bounded buffer, for instance. In a solution using three threads, a *Producer* thread assembles information items to be sent to the buffer, a *BoundedBuffer* thread manages the buffer space and a *Consumer* thread uses information items taken from the buffer. A basic domain will be reserved for each of these threads. Let BD_0 be the domain reserved for *BoundedBuffer*. This domain will include both the read and the write access rights for the memory pages reserved to store the buffered items. No access right for these pages will be included in the basic domains, say BD_1 and BD_2, reserved for *Producer* and *Consumer*. This means that in the page table entries reserved for these pages only bits $\mathcal{R}_0$ and $\mathcal{W}_0$ corresponding to basic domain BD_0 are asserted; all the other bits are cleared. When execution of an operation of *BoundedBuffer* is started up, ADR is set to specify basic domain BD_0. To this aim, we issue protection primitive *makeActive*(*d*), where quantity *d* is a bit configuration featuring bit 0 asserted.

B. Domain Unions

We shall now consider the case of an active domain that is the result of the logical union of a number of basic domains. This corresponds to a situation, for instance, in which a thread is called to operate on two data sets at the same time. If each data set corresponds to a basic domain, the thread must run in the domain resulting from the union of these basic domains.

Let us refer to the example illustrated in Fig. 2a. Basic domain BD_0 contains the read access right for pages *P'* and *P"*, and basic domain BD_1 contains the write access right for *P'*. The logical union of these two domains produces a domain D that includes both the read and the write access rights for *P'* and the read access right for *P"*. With reference to our protection system, Fig. 2b shows the corresponding configuration for the page table and ADR. For each given page table entry, the figure only shows the page protection fields of this entry, whereas the frame number field, which is not related to protection, is not shown. In the page table entry PT(*P'*) reserved for page *P'*, bits $\mathcal{R}_0$ (corresponding to domain BD_0) and $\mathcal{W}_1$ (corresponding to domain BD_1) are asserted, and all the other bits are cleared. In the page table entry PT(*P"*) reserved for page *P"*, only bit $\mathcal{R}_0$ (corresponding to domain BD_0) is asserted. We shall make domain D active by issuing protection primitive *makeActive*(*d*), where quantity *d* is a bit configuration featuring bits 0 and 1 asserted. So doing, we grant the access rights in both BD_0 and BD_1 to the running thread.

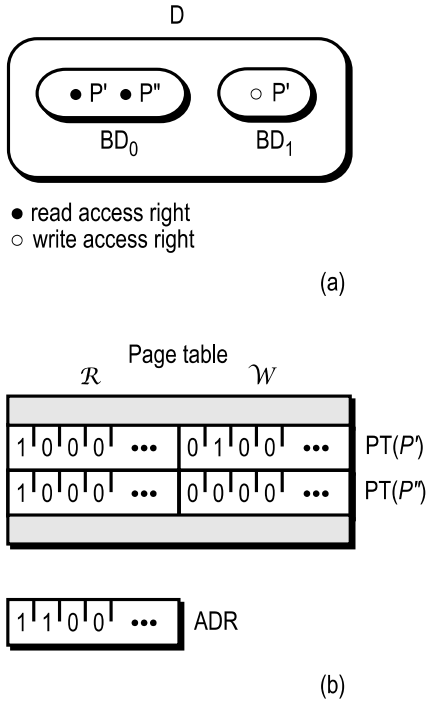

Figure 2. Domain union: (a) configuration of two basic domains BD_0 and BD_1 and their logical union D; (b) corresponding configurations of the page table and ADR.

C. Hierarchical Domains

Protection domains can be organized into a hierarchy where a parent domain at the directly higher level to one or more child domains imports all the access rights of the child domains. A salient feature of a hierarchical organization of protection domains is that revocation of an access right from a child domain produces revocation of this access right from the father domain and, recursively, from all the domains at the higher levels in the hierarchy that import this access rights.

Let us refer, for instance, to a two-level hierarchy in which a given domain D_k is at a directly higher level to child domains D_0, D_1, …, D_{k-1}. In our protection system, we shall reserve a basic domain BD_i to each child domain D_i, $i = 0, 1, \ldots, k-1$. For each read or write access right in D_i we set the i-th bit in the $\mathcal{R}$ or $\mathcal{W}$ field of the corresponding page table entry. Moreover, we reserve a basic domain BD_k for father domain D_k. For each read or write access right in D_k, we set the k-th bit in the $\mathcal{R}$ or $\mathcal{W}$ protection field of the corresponding page table entry (it should be noted that no action is required for the access rights that D_k imports from its child domains). In the ADR configuration for child domain D_i we set only the i-th bit corresponding to basic domain BD_i; all the other bits will be cleared. For the father domain D_k, we set bits $0, 1, \ldots, k$. So doing, an access right of a child domain will be part of the father domain too, and if this access right is deleted from the child domain it will be deleted from the father domain as well. Of course, the configuration outlined above can be easily extended to cope with deeper domain hierarchies.

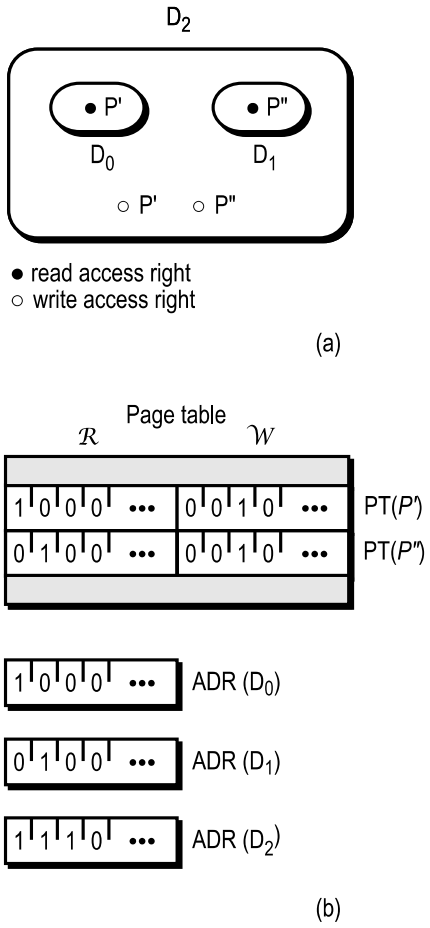

Figure 3. Hierarchical protection domains: (a) configuration of a father domain D_2 and two child domains D_0 and D_1; (b) corresponding configurations of the page table and ADR.

An example of a hierarchical organization of protection domains is shown in Fig. 3a. Domain D_2 is at a directly higher level to domains D_0 and D_1. Domain D_0 includes the read access right for page *P'*, and similarly for domain D_1 and page *P"*. Domain D_2 includes the write access right for both *P'* and *P"*, and imports the access rights in D_0 and D_1. Fig. 3b shows the corresponding configurations for the page table and ADR. Three basic domains, namely BD_0, BD_1 and BD_2, are reserved for domains D_0, D_1 and D_2, respectively. In the page table entry PT(*P'*) reserved for page *P'*, bits $\mathcal{R}_0$ (corresponding to domain BD_0) and $\mathcal{W}_2$ (corresponding to domain BD_2) are asserted, and similarly for page table entry PT(*P"*), bit $\mathcal{R}_1$ and bit $\mathcal{W}_2$. In the ADR configuration for child domain D_0, only bit ADR_0 is asserted, and similarly for D_1 and bit ADR_1. In the ADR configuration for father domain D_2, bits ADR_0, ADR_1 and ADR_2 are asserted and all the other bits are cleared. In this way, a thread running in D_2 will be granted full access rights for both *P'* and *P"*. In the configuration proposed, revocation of a given access right from D_0 or D_1 produces revocation of this access right from D_2, too.

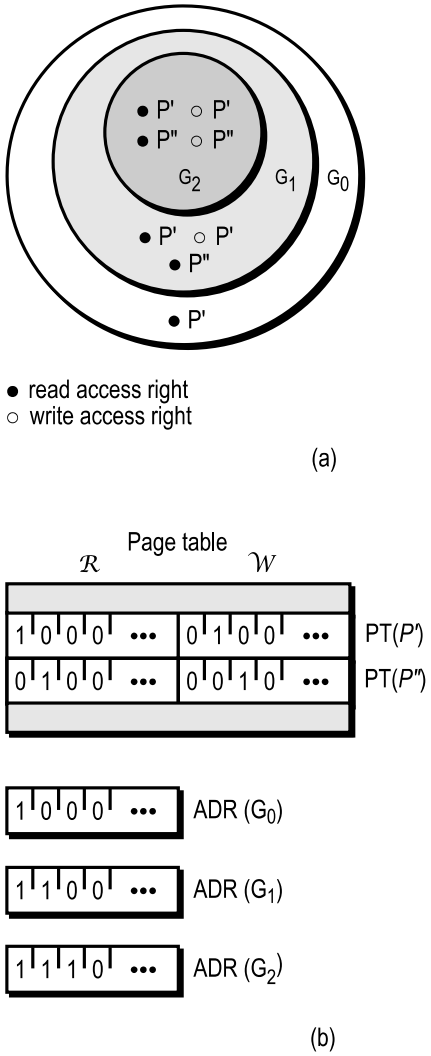

Figure 4. Protection rings: (a) configuration of three protection rings; (b) corresponding configurations of the page table and ADR.

D. Protection Rings

A protection ring is a hierarchical organization of protection domains in which each father domain has a single child [13]. Protections rings are usually depicted as concentric rings where domain G_i corresponding to an inner ring is at a higher level in the hierarchy than the domains G_0, G_1, …, G_{i-1} corresponding to outer rings (this is in sharp contrast with the usual scheme used to represent hierarchical domains as shown in Fig. 3, where outer domains have higher strength than inner domains). In our protection system, we shall implement the protection rings by using the basic domains as follows: basic domain BD_i contains those access rights that are exclusively owned by ring G_i (whereas BD_i will *not* contain the access rights that G_i shares with the lower level rings G_0, G_1, …, G_{i-1}). In the ADR configuration for ring G_i, bits ADR_0, ADR_1, …, ADR_i are asserted and the other bits are cleared.

Fig. 4a shows a protection system featuring three rings, and Fig. 4b shows the corresponding implementation in our protection system. Basic domain BD_0 corresponds to the outer (least privileged) ring G_0 that includes the read access right for page *P'*, basic domain BD_1 corresponds to the middle-level ring that includes the write access right for *P'* and the read access right for *P"*, and finally, basic domain BD_2 corresponds to the inner (most privileged) ring that includes the write access right for *P"*. In the page table entry PT(*P'*) reserved for page *P'*, only bits $\mathcal{R}_0$ and $\mathcal{W}_1$ are asserted; in PT(*P"*), only bits $\mathcal{R}_1$ and $\mathcal{W}_2$ are asserted. In the ADR configuration for ring G_0, bit ADR_0 is asserted and all the other bits are cleared; in the configuration for G_1, bits ADR_0 and ADR_1 are asserted and the other bits are cleared; and finally, in the configuration for G_2, bits ADR_0, ADR_1 and ADR_2 are asserted and the other bits are cleared.

E. User/Supervisor Protection Domains

A particular case of a two-level protection ring organization is the traditional pair of protection domains, *user* and *supervisor*. The user domain limits the extent of the memory areas that can be accessed by the running thread to a subset of all pages, whereas no such limit exists for a thread running in the supervisor domain. In our protection system, a result of this type can be obtained by adding a bit to ADR, say bit ADR_n, which, if asserted, inhibits the generation of exceptions of access violation to the processor.

It should be noted that if we insert the all-one bit configuration into ADR, we obtain a domain that includes the access rights for all the pages which are part of at least one of the basic domains of the running process. Of course, this is not the same as the supervisor domain, as the resulting active domain includes no access right for a page that is not part of at least one of these basic domains. An effective, alternative solution is to disable generation of exceptions of access violation to the processor in the presence of the all-ones bit configuration in ADR; however this solution prevents us from activating the weaker domain that results from the logical union of all the basic domains of the running process.

On the other hand, it has been widely demonstrated that the supervisor domain is inadequate to cope with protection of the inner kernel layers [17]. A better alternative is that even device drivers comply with the principle of least privilege. This means that the partitioning of access privileges into domains should be maintained even for the kernel components. So doing we facilitate in-line expansion of system calls [10], and avoid the run-time overhead that is usually connected with the supervisor state owing to the need to save and then restore the state of the running process at each system call [2].

F. Capability Lists

In the capability-based protection paradigm, a *capability* is a pair {object name, access rights} [9], [11]. A thread that holds a capability for a given object is allowed to access this object and carry out the actions that are made possible by the access rights in the capability. Pro-

tection domains take the form of lists of capabilities. A thread running in a given domain holds all the capabilities in the list associated with this domain. A salient feature of capability based protection is the ease of implementing small protection domains and small objects to facilitate error confinement and fault detection [5], [6].

Our page-oriented protection environment supports a form of capability protection as follows. A capability consists of the read and/or the write access rights for a memory page. The capability list associated with basic domain BD_i includes a capability for a given page P if the page table entry PT(P) corresponding to this page specifies the read and/or the write access rights for BD_i, that is, the $\mathcal{R}_i$ and/or the $\mathcal{W}_i$ bits of PT(P) are asserted. The protection fields of the page table entries can be seen as forming a two-dimensional array featuring one row for each page and two columns for each domain (i.e., for BD_i, the two columns formed by the bits in the i-th position of the $\mathcal{R}$ and the $\mathcal{W}$ fields of all the page table entries). These two columns considered as a whole form the capability list of BD_i.

G. Access Control Lists

An access control list is a collection of pairs {domain name, access rights} that is associated with a given object [12]. When a thread running in a given domain performs an access attempt to a protected object, the access control list of this object is inspected to ascertain whether that domain holds the access right permitting successful accomplishment of the access. By keeping all the access permissions for a given protected object grouped in a single list, access control lists are especially well suited to access right review and revocation.

In our protection environment, the access control list of a given page P is implemented by the $\mathcal{R}$ and $\mathcal{W}$ fields of the page table entry PT(P) corresponding to this page. If a bit, say bit $\mathcal{R}_i$, of PT(P) is asserted, then the access control list of P contains the pair {BD_i, read}.

H. Amplification and Revocation of Access Privileges

An amplification of privileges is the action of increasing the strength of the active domain by adding new access rights. This is necessary, for instance, when execution of the running thread reaches a point at which one or more components of the internal representation of a data object M must be accessed, and this data object is not part of the active domain. In a situation of this type, the active domain should be amplified to contain the access rights to the pages containing the components of M.

We shall reserve a basic domain, say domain BD_0, to the pages of the components of M. For each page P, we set bit $\mathcal{R}_0$ and/or bit $\mathcal{W}_0$ of the page table entry PT(P) reserved for this page. Let ADR* denote a bit configuration of the same size as the ADR size, featuring bit 0 asserted and all the other bits cleared. The desired amplification of the active domain will be obtained by using the *makeActive*() protection primitive to replace the actual contents of ADR with the result of the bitwise OR of the previous contents and quantity ADR*.

A revocation of access privileges is the action of reducing the strength of the active domain by eliminating a subset of the access rights it contains [7]. Let us refer again to data object M. When the running thread terminates its access to this data object, the access rights for the pages storing the internal representation of the object are no longer necessary for the running thread to carry on its job. According to the principle of least privilege, these access rights should be revoked from the active domain. Let ADR** denote a bit configuration of the same size as the ADR size, featuring bit 0 cleared and all the other bits asserted. The desired revocation of access rights will be obtained by using the *makeActive*() protection primitive to replace the actual contents of ADR with the result of the bitwise AND of the previous contents and quantity ADR**.

IV. Concluding Remarks

With reference to a multithreaded environment and a paged memory system, we have approached the problem of supporting protection domain separation between the threads of the same process. Essentially, we have extended the page table to contain protection information together with the usual virtual to physical address mapping information that is necessary to translate virtual page numbers into physical frame numbers. So doing, we can take advantage of the translation lookaside buffer to cache the page protection information inside the memory management unit. The resulting cost of the protection circuitry is low, and is much lower than that of, for instance, such a complex hardware circuitry as an *ad-hoc* protection cache inside the memory management unit.

We have demonstrated that our protection environment makes it possible to support several distinct protection models, including capability lists and access control lists, as well as the amplification and the revocation of access privileges. Rather than imposing a specific view of page protection, we introduce a set of protection mechanisms at the hardware level that can be controlled by a small set of protection primitives. These primitives can be used to implement the desired protection paradigm at little effort.

On the other hand, no mechanism prevents malicious software from using the protection primitives and control domain definition to subvert the protection boundaries. Rather than precluding software attack from producing disruptive effects, our protection environment is aimed at limiting the consequences of programming errors. This is the case for a software extension that is not intended to produce harmful effects but is prone to programming errors that expose the main application to severe risks of corruption. By limiting the extent of thread actions in memory, our proposal is aimed at limiting the consequences of situations of this type.

References

[1] M. Cekleov, M. Dubois, “Virtual-address caches. Part 1:

problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5 (September/October 1997), pp. 64–71.

[2] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, C. Yoon, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 148–157.

[3] T. Chiueh, G. Venkitachalam, P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Charleston, South Carolina, USA, December 1999, pp. 140–153.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler, "An empirical study of operating systems errors," *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, October 2001, pp. 73–88.

[5] P. Corsini, L. Lopriore, "An implementation of storage management in capability environments," *Software — Practice and Experience*, vol. 25, no. 5 (May 1995), pp. 501–520.

[6] E. F. Gehringer, *Capability Architectures and Small Objects*, UMI Research Press, Ann Arbor, Mich., 1982.

[7] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6 (November 1979), pp. 575–586.

[8] B. Leslie, N. FitzRoy-Dale, G. Heiser, "Encapsulated user-level device drivers in the Mungi operating system," *Proceedings of the Workshop on Object Systems and Software Architectures*, Victor Harbor, South Australia, Australia, January 2004, pp. 16–30.

[9] H. M. Levy, *Capability-Based Computer Systems*, Digital Press, 1984.

[10] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, W. Schröder-Preikschat, "Configurable memory protection by aspects," *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, Stevenson, Washington, USA, October 2007.

[11] L. Lopriore, "Access control mechanisms in a distributed, persistent memory system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10 (October 2002), pp. 1066–1083.

[12] J. H. Saltzer, M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9 (September 1975), pp. 1278–1308.

[13] M. D. Schroeder, J. H. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3 (March 1972), pp. 157–170.

[14] J. Shen, G. Venkataramani, M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, San Jose, California, USA, October 2006, pp. 52–57.

[15] M. M. Swift, B. N. Bershad, H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Transactions on Computer Systems*, vol. 23, no. 1 (February 2005), pp. 77–110.

[16] A. S. Tanenbaum, J. N. Herder, H. Bos, "Can we make operating systems reliable and secure?," *Computer*, vol. 39, no. 5 (May 2006), pp. 44–51.

[17] M. Wilkes, "Operating systems in a changing world," *SIGOPS Operating Systems Review*, vol. 28, no. 2 (April 1994), pp. 9–21.

[18] E. Witchel, K. Asanović, "Hardware works, software doesn't: enforcing modularity with Mondriaan memory protection," *Proceedings of the 9th Conference on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003, pp. 24–29.

[19] E. Witchel, J. Cates, K. Asanović, "Mondrian memory protection," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2002, pp. 304–316.

[20] E. Witchel, J. Rhee, K. Asanović, "Mondrix: memory isolation for Linux using Mondriaan memory protection," *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005, pp. 31–44.