

Protection by Hiding: Eliminating the Need for Kernel Mode

David L. Cohn, Vsevolod V. Panteleenko, Alan G. Yoder, Larry Barchett, Frederic Barrat

Department of Computer Science and Engineering

University of Notre Dame

Notre Dame, Indiana, USA

{dlc,vvp,agy,ldb,fbarrat}@cse.nd.edu

Abstract

Kernel mode has been assumed necessary to give the operating system exclusive rights to certain important system functions. For example, it has allowed the creation of user processes which are confined to separate virtual address spaces. The resulting “hardware” protection is valuable, but the required context changes incur a significant overhead cost. Even systems that use software protection to isolate processes rely on hardware protection to guard the kernel. The kernel classically uses its privilege to manage the virtual memory hardware which establishes these separate address spaces. An alternative scheme, which could be used for processors with 64-bit addresses but which makes most sense with a 128-bit address space, is to use probabilistic protection, or *protection by hiding*. The approach eliminates the need for a privileged kernel mode while retaining highly reliable protection. Rather than separate address spaces, it hides address regions within a huge global address space. With hardware mechanisms proposed in this paper, protection by hiding eliminates the need for kernel mode. Context switches essentially disappear and the privileges associated with the kernel can be safely distributed in various ways. The new structure also deals with some of the problems of super-large address spaces and offers an interesting approach to massively parallel system design.

1. The Problem

Protecting address spaces from each other is a fundamental responsibility of modern operating systems. The classic mechanism used is paged virtual memory, aided by virtual memory hardware. Physical memory is partitioned into *pages*, which usually correspond to some convenient disk block size. For resident pages, a *virtual address*, which locates the page in the large virtual address space, is translated by the virtual memory hardware into a *physical address*, which specifies a location in main memory. This address is sent to memory, and contents of the page at the requested offset are then loaded. For non-resident pages, a page fault occurs and the requesting process blocks. Performance considerations dictate the use of a special cache, usually called the *Translation Lookaside Buffer* (TLB), to speed up the translation process. The TLB holds the results of recent virtual/physical page number mappings and enables near-instantaneous virtual/physical address translation over 90% of the time in modern processors.

So long as a given address space is in use, the TLB mitigates the cost of virtual/physical address translation. When an address-space context switch must be made, however, there is a large cost. The TLB must be flushed to prevent the process in the new address space from accessing memory belonging to the prior context. The flush itself takes several processor cycles, and each TLB line must be filled by a lookup to the page table. For large-memory-space machines, such as the 64- and 128-bit address machines considered here, this must be an *inverted page table*. Such a table holds one entry for each resident page, instead of an entry for each page of virtual memory. Address translation using an inverted page table is slow, even with the use of hash tables, and takes typically some dozens of processor cycles. There are other things that also have to be done during a context switch. The end result is that a context switch causes hundreds of lost pro-

cessor cycles in most modern processors. The Pentium Pro, for example, has a quoted context switch time of over 200 cycles, and may not even include the cycles necessary to reload the TLB.

2. A Solution Using Probabilistic Protection

2.1 Protection by hiding

Protection by hiding relies on probabilistic, rather than absolute, protection guarantees. It does not assume separate virtual address spaces for each process, but rather a single global address space. Using a system quite similar to Anonymous RPC as originally proposed by Yarvin, Bukowski and Anderson [1], contiguous memory elements are randomly placed in the space. Any process can physically generate any address, but the sparseness of the space makes it essentially impossible to accidentally or maliciously find an arbitrary region.

For example, consider a 128-bit address space supporting a thousand processes each with a thousand contiguous memory regions as big as a gigabyte. Thus, 10^{15} locations of the 2^{128} or 3.4×10^{38} in the address space are being used. We will assume that the contiguous memory regions are placed randomly throughout the space. Then, if an errant or malicious process tried to find one, each attempt would have only a one in 3.4×10^{23} chance of success. Further suppose that this process made one such attempt every microsecond, that this went on for 5,000 years (the length of recorded human history) and that no one noticed. The process would have made 1.6×10^{17} tries, giving it a success probability of less than one in a million. (Of course, if it finds one location, it would have no trouble finding contiguous locations.) Thus, while probabilistic protection is not perfect, it works fairly well in 128-bit address spaces.

2.2 Addresses as Capabilities

With probabilistic protection, a global address is a capability; only someone who knows the global address of a memory region can access content in that region. As we shall see, there is no need for kernel mode, nor any hardware protection based on user or process ID. With some changes to traditional hardware notions, sufficient “privilege” accrues to the first code to run that it can act as the “kernel”. This code gains its power by setting the global addresses of certain memory areas including those which contain the vector interrupt table, any memory-mapped I/O areas, and special hardware we call *tag tables*. This code may share its privilege by simply giving out some of these global address.

2.3 Addressing Memory

Unlike traditional computers, we do not propose that virtual addresses be translated to physical addresses in a CPU and that physical addresses be transmitted on an address bus. Rather, we assume that the global addresses themselves are placed on an address bus, and that physical devices in the memory space do the “translating”. Each memory-mapped device would have one or more *tags* that define where that device is currently located in the global address space. The device would “snoop” the address bus waiting for an address that falls into a range defined by one of its tags.

We assume that the system hardware is capable of detecting one of three possible outcomes of each memory reference:

- Exactly one device responds - the normal case.
- No devices respond - a “page miss” that triggers a memory manager interrupt.
- Multiple devices respond - a collision that triggers a different memory manager interrupt.

We assume that the “kernel” code has set up the vector interrupt table so that these interrupts are indeed serviced by code that can manage memory.

Memory elements are be divided into “page-like” *memory units*, each of which has its own tag. Each tag indicates where the corresponding memory unit is mapped in the global address space. The tags them-

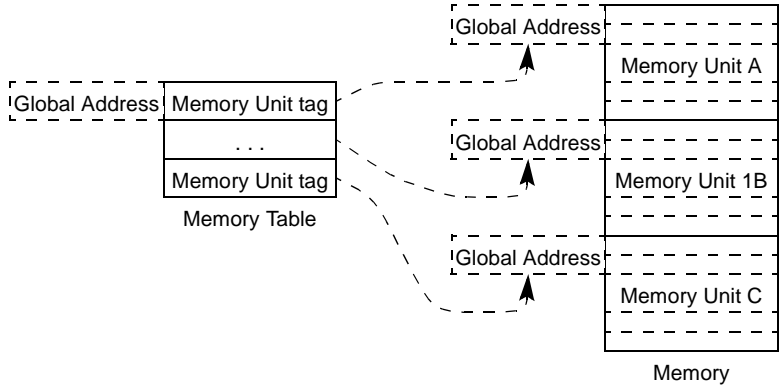


Figure 1 Tagged Memory Device

selves would be memory mapped and collected into a *memory table*. As shown in Figure 1, the memory table is also memory-mapped and has its own tag. Thus, to manage memory, code must know the global address of the corresponding memory table

The notion is recursive. The memory table tags are also memory-mapped. For simplicity, we assume that there is only one level of recursion and that memory table tags are mapped into a global tag table. As shown in Figure 2, the global tag table also includes the tags for the vector table, for memory-mapped I/O and for itself. At power-up, the tag table is at a known location, perhaps location zero, so the tag table tag is easily found. The “kernel” moves the tag table by writing a new address to the tag table tag. It is important that the system hardware treat the writing of tags as atomic operations.

Effectively hiding things in memory involves generating random tags and then storing them in the tag table. Randomness ensures that exhaustive searching is the only available technique to locate hidden memory units. The tags must be random at least to the level of cryptographic security, so that knowing both the algorithm and any sequence of previous tags yields no useful knowledge of what the next tag will be. Available techniques are somewhat expensive (costing on the order of hundreds of operations), but need be

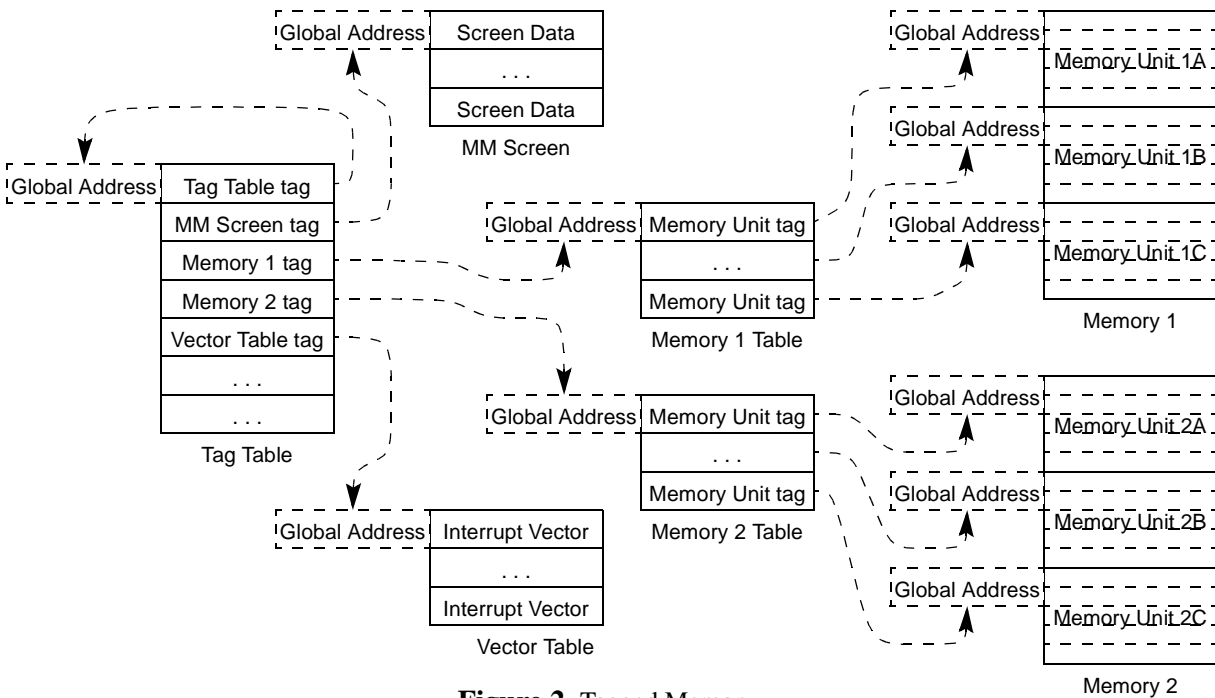


Figure 2 Tagged Memory

used only when a new memory unit is allocated or remapped. Alternatively, a hardware generator could be build from a set of free-running flip-flops. If the flip-flops have different periods which are short relative to the rate at which tags are generated, the tags would be essentially random.

2.4 Multiply-Tagged Memory

It is possible for a given memory unit to have multiple tags. For example, it could have a read tag, a write tag and an execute tag. Reads from the memory unit would have to be from the address specified by the read tag; writes would be to the write tag address, and execution at the execution tag address. Of course, this means that the memory would have to be able to distinguish between a normal data read and an instruction fetch, but this is common with modern processors. As we shall see, safe procedure calls and “system calls” depend on being able to provide execute-only access to regions of memory.

2.5 Process Creation and System Calls

Although there is no kernel in the classic sense, code that has “privilege” must perform kernel functions. For example, here are the steps the “kernel” (i.e. code that can manipulate tag tables) would use to launch and interact with an non-privileged “user” process:

- Allocate and tag memory for the “user” program text and load the memory from non-volatile memory. Linking would entail placing tags in the proper places in the code.
- Allocate and tag memory for stack, heap and PTE or the equivalent.
- Write appropriate tags into registers so the “user” process can use the allocated memory.
- Set the processor’s program counter with the execute-only global address tag of the entry point to the program text.

After this is done, control is transferred to the program text of the new process.

With no kernel mode, there is no “system call” in the usual sense. All processes run at “user” level. However, it is necessary for non-privileged code to request services from privileged code without compromising the privileged code. For example, suppose that “user” process A wants to read a file. To do so, it will run the file reading library code in its own “context,” with neither A nor the library code able to discover the location of the other. Preserving anonymity on both sides requires the assistance of an intermediary code with execute-only access. The level of indirection provided by execute-only access allows the library entry point to be effectively specified by the caller, without the caller actually knowing what it is. The intermediary code is trusted and has the “privilege” necessary to read and write these locations and see that they contain correct values.

The intermediary code is very small and carefully tested to be sure that entering somewhere besides the entry point will do no damage. Its role is to isolate the caller from the callee, and a procedure call operates as follows:

- The caller does the following:
 - Allocates a new memory area, to be used as a stack by the callee
 - Reserves space for a return area and pushes any arguments onto this stack.
 - Saves and clears its registers.
 - Places the address of the new stack into its stack pointer.
 - Calls the execute-only entry point of the intermediary code.
- The intermediary code then:
 - Pops the return address off the stack and saves it.
 - Calls the callee’s entry point which places an execute-only return address on the stack. The return address will be unique to this call, allowing the intermediary code to identify the thread of execution when the callee completes.
- The callee then:
 - Runs to completion, using the provided stack for arguments and return area.

- Cleans up anything important left on the stack.
- Does a return to the intermediary code.
- The intermediary code then pushes the caller's return address and does a return.
- The caller retrieves the return data from the memory area which was used as a stack by the callee

This is similar in some respects to the method proposed by Yarvin, Bukowski and Anderson, particularly in the use of execute-only memory to hide the callee's actual location from the caller.

3. Hardware Implications

As indicated, our solution assumes modifications to traditional hardware design. In addition to a very wide physical address bus, we assume that the CPU (or CPUs) export global addresses rather than translating them into some sort of physical addresses. We also assume that normal memory is somewhat intelligent. It "snoops" the address bus, comparing these global addresses to information in its tag registers. The notion we've called *memory units* could correspond to classic fixed-size pages in which case tags would be the most significant portion of the global address. Memory units could also have variable size, implying tags which specify base and size information.

3.1 Design of a Memory Unit

A memory unit could be implemented as a small memory macro augmented with extra circuitry which:

- Stores the global addresses to be answered to for read, write and execute accesses¹.
- Acts like a small content-addressable memory, comparing the registers to addresses on the address bus, and responding appropriately when there is a match:
 - Signal that a match has been detected.
 - If the bus indicates multiple matches, abort the transaction.
 - For a read or execute match, it places the corresponding contents on the memory bus.
 - For a write match, it copies the memory bus into the corresponding location.

3.2 Contiguous memory areas

Requested memory regions may exceed the size of the available memory units. A contiguous region of the global address space can be made up by using multiple memory units and tagging them with successive locations. This does not substantially alter the protection characteristics except by giving an attacker a larger target to find. However, the global address space is so vast that finding even very large regions is extremely difficult.

3.3 Implications

Protection by hiding avoids two of the major costs incurred by modern operating systems. First, it does away with protection-level switch costs. The processor need not incorporate logic for protection levels or for instruction privilege. Second, because address translation is effectively done by the memory, no TLB is required. The hardware gain is minimal, but the software gain is large—no TLB refreshes are necessary after a "context switch."

Protection by hiding also allows radical distribution of memory and processing work. Unlike traditional virtual addresses, global addresses have fixed meaning independent of any context. This has several interesting implications. Consider, for example, the handling of a "page fault". (We use quotes because we are not tied to the normal notion of a "page".) Traditionally, fault resolution requires that the context of the faulting process be saved and that the fault resolution subsystem know which process caused the fault. Here, there is no context to save, and fault resolution need only know the faulting address. Thus, resolution can be handled by a different processor, perhaps one optimized for fault resolution. Meantime, the original

1. Other types of access, such as read/write, are also possible.

processor can restart the faulting process whenever it chooses. If the fault has not yet been resolved, the process will fault again and have to wait some more.

Similar schemes are possible for bulk I/O transfers. If the global address of a memory buffer is given to a DMA device, the device can complete the transfer whether or not the requesting process is “resident”. If the buffer is prematurely released from resident memory, the DMA transfer will fault, and the fault resolution processor can retrieve the buffer.

This addressing freedom seems particularly well suited to massively parallel systems. Most processors would be assigned to running useful processes, and a few processors would be given the overhead of the operating system and I/O transactions. Since addresses are not context sensitive, the need for processor coordination is substantially reduced. This has important operating system design implications for Processors-in-Memory technology in which CPU macros are embedded directly into memory chips in order to overcome the traditional processor-memory bottleneck.

4. References

- [1] Yarvin, Bukowski and Anderson. Anonymous RPC: Low-Latency Protection in a 64-bit Address Space. *Proc. Summer USENIX Conference*, 1993, pp. 175-186.