Virtual Memory and MMU Concepts

Andres Krapf
{Andres.Krapf}@sophia.inria.fr

INRIA, Sophia Antipolis - Méditerranée

30th October 2007

Abstract

This document reviews the concept of Virtual Memory. We summarize basic concepts, such as page faults and Memory Management in general. Finally we introduce MMU virtualization and a mechanism to address it.

1 Virtual Memory Details

First, we should introduce the concept of virtual address space. As the term implies, the virtual address space is the program's address space - how much memory the program would require if it needed all the memory at once. But there is an important distinction; the word "virtual" means that this is the total number of uniquely-addressable memory locations required by the application, and not the amount of physical memory that must be dedicated to the application at any given time.

In order to implement virtual memory, it is necessary for the computer system to have special memory management hardware. This hardware is often known as an MMU (Memory Management Unit). Without an MMU, when the CPU accesses RAM, the actual RAM locations never change - memory address 123 is always the same physical location within RAM.

However, with an MMU, memory addresses go through a translation step prior to each memory access. This means that memory address 123 might be directed to physical address 82043 at one time, and physical address 20468 another time. As it turns out, the overhead of individually tracking the virtual to physical translations for billions of bytes of memory would be too great. Instead, the MMU divides RAM into pages - contiguous sections of memory of a set size that are handled by the MMU as single entities.

Keeping track of these pages and their address translations might sound like an unnecessary and confusing additional step, but it is, in fact, crucial

to implementing virtual memory. For the reason why, consider the following point.

Assume that an application's first instruction accesses data stored at address 12374. However, also assume that our computer only has 12288 bytes of physical RAM. What happens when the CPU attempts to access address 12374?

What happens is known as a page fault. Next, let us see what happens during a page fault.

1.1 Page Faults

First, the CPU presents the desired address (12374) to the MMU. However, the MMU has no translation for this address. So, it interrupts the CPU and causes software, known as a page fault handler, to be executed. The page fault handler then determines what must be done to resolve this page fault. It can:

- Find where the desired page resides on disk and read it in (this is normally the case if the page fault is for a page of code)
- Determine that the desired page is already in RAM (but not allocated to the current process) and direct the MMU to point to it
- Point to a special page containing nothing but zeros and later allocate a page only if the page is ever written to (this is called a copy on write page, and is often used for pages containing zero-initialized data)
- Get it from somewhere else (which is discussed in more detail later)

While the first three actions are relatively straightforward, the last one is not. For that, we need to cover some additional topics.

1.2 The Working Set

The group of physical memory pages currently dedicated to a specific process is known as the working set for that process. The number of pages in the working set can grow and shrink, depending on the overall availability of pages on a system-wide basis.

The working set grows as a process page faults. The working set shrinks as fewer and fewer free pages exist. In order to keep from running out of memory completely, pages must be removed from process's working sets and turned into free pages, available for later use. The operating system shrinks processes' working sets by:

- Writing modified pages to a dedicated area on a mass storage device (usually known as swapping or paging space)

- Marking unmodified pages as being free (there is no need to write these pages out to disk as they have not changed)

In order to determine appropriate working sets for all processes, the operating system must track usage information for all pages. In this way, the operating system can determine which pages are being actively used (and must remain memory resident) and which pages are not (and therefore, can be removed from memory). In most cases, some sort of least-recently used algorithm determines which pages are eligible for removal from process working sets.

1.3 Swapping

While swapping (writing modified pages out to the system swap space) is a normal part of a system's operation, it is possible to experience too much swapping. The reason to be wary of excessive swapping is that the following situation can easily occur, over and over again:

- Pages from a process are swapped
- The process becomes runnable and attempts to access a swapped page
- The page is faulted back into memory (most likely forcing some other processes' pages to be swapped out)
- A short time later, the page is swapped out again

If this sequence of events is widespread, it is known as thrashing and is indicative of insufficient RAM for the present workload. Thrashing is extremely detrimental to system performance, as the CPU and I/O loads that can be generated in such a situation can quickly outweigh the load imposed by a system's real work. In extreme cases, the system may actually do no useful work, spending all its resources moving pages to and from memory.

2 MMU functions

```
mmu_map(unsigned int mem, unsigned int log, unsigned int phy,
        unsigned int size);
```

The first specifies the translation to be set up. The second and third parameters specify the logical and physical addresses of the translation. The final parameter specifies the size of the region.

```
mmu_map_del(unsigned int mem);
```

2.1 Para-virtualisation of the MMU

Typical "full virtualisation" uses a method known as "shadow page tables", whereby two sets of pagetables are maintained: the guest domain's set, which aren't visible to the hardware via cr3, and page tables visible to the hardware which are maintained by the hypervisor. As only the hypervisor can control the page tables the hardware uses to resolve TLB misses, it can maintain the virtualisation of the address space by copying and validating any changes the guest domain makes to its copies into the "real" page tables.

All these duplicates pages come at a cost of course. A para-virtualisation approach (that is, one where the guest domain is aware of the virtualisation and complicit in operating within the hypervisor) can take a different tack. In Xen, the guest domain is made aware of a two-level address system. The domain is presented with a linear set of "pseudo-physical" addresses comprising the physical memory allocated to the domain, as well as the "machine" addresses for each corresponding page. The machine address for a page is what's used in the page tables (that is, it's the real hardware address). Two tables are used to map between pseudo-physical and machine addresses. Allowing the guest domain to see the real machine address for a page provides a number of benefits, but slightly complicates things, as we'll see.

2.1.1 Xen - an example of paravirtualization

Unfortunately, x86 does not have a software-managed TLB; instead TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the current address space should be present in the hardware-accessible page table. Moreover, because the TLB is not tagged, address space switches typically require a complete TLB flush. Given these limitations, Xen made two decisions: (i) guest OSes are responsible for allocating and managing the hardware page tables, with minimal involvement from Xen to ensure safety and isolation; and (ii) Xen exists in a 64MB section at the top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor.

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables. Guest OSes may batch update requests to amortize the overhead of entering the hypervisor. The top 64MB region of each address space, which is reserved for Xen, is not accessible or

remappable by guest OSes. This address region is not used by any of the common x86 ABIs however, so this restriction does not break application compatibility. Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have lower privilege than Xen, and (ii) they may not allow any access to the Xen- reserved portion of the address space.

3 Privilege levels

Most modern processors include the capability to change privilege levels for different tasks. Lower privilege levels are denied the ability to directly access some or all hardware resources. This way, for example, applications cannot directly read from the hard drive. This is very important when the operating system must guarantee security, since an application that could read from the disk without the kernel becoming involved could bypass access control to files on that disk.

The exact mechanisms provided to restrict access to hardware differ between processors. There are questions of exactly how privilege is assigned, as well as how the transition from one privilege level to another happens. (Obviously, the transition from lower to higher privilege must also include a guarantee that trusted code is run afterward.) To insulate themselves from hardware-specific implementations of privilege, many operating systems treat the processor as having two modes: kernel mode and user mode. This is pretty likely to be implementable on top of any hardware protection scheme. Other variations are possible that take advantage more of specific processor features.

Ultimately, the purpose of distinct operating modes for the CPU is to provide hardware protection against accidental or deliberate corruption of the system environment (and corresponding breaches of system security) by software. Only "trusted" portions of system software are allowed to execute in the unrestricted environment of kernel mode, and only then when absolutely necessary. All other software executes in one or more user modes. If a processor generates a fault or exception condition in a user mode, in most cases system stability is unaffected; if a processor generates a fault or exception condition in kernel mode, most operating systems will halt the system with an unrecoverable error. When a hierarchy of modes exists (ring-base security), faults and exceptions at one level of privilege may destabilize higher levels of privilege, but not lower levels of privilege. Thus, a fault in Ring 0 (the kernel mode with the highest privilege) will crash the entire system, but a fault in Ring 2 will only affect rings 3 and beyond and Ring 2 itself, at most.

Transitions between modes are at the discretion of the executing thread

when the transition is from a level of high privilege to one of low privilege (as from kernel to user modes), but transitions from lower to higher levels of privilege can take place only through secure, hardware-controlled "gates" that are traversed by executing special instructions or when external interrupts are received.

In Arm platforms permissions are checked using *MMU Domains* where a page table entry specifies which domain (of 16) it belongs to, and a processor control register specifies the permission currently in force for the sixteen different domains.