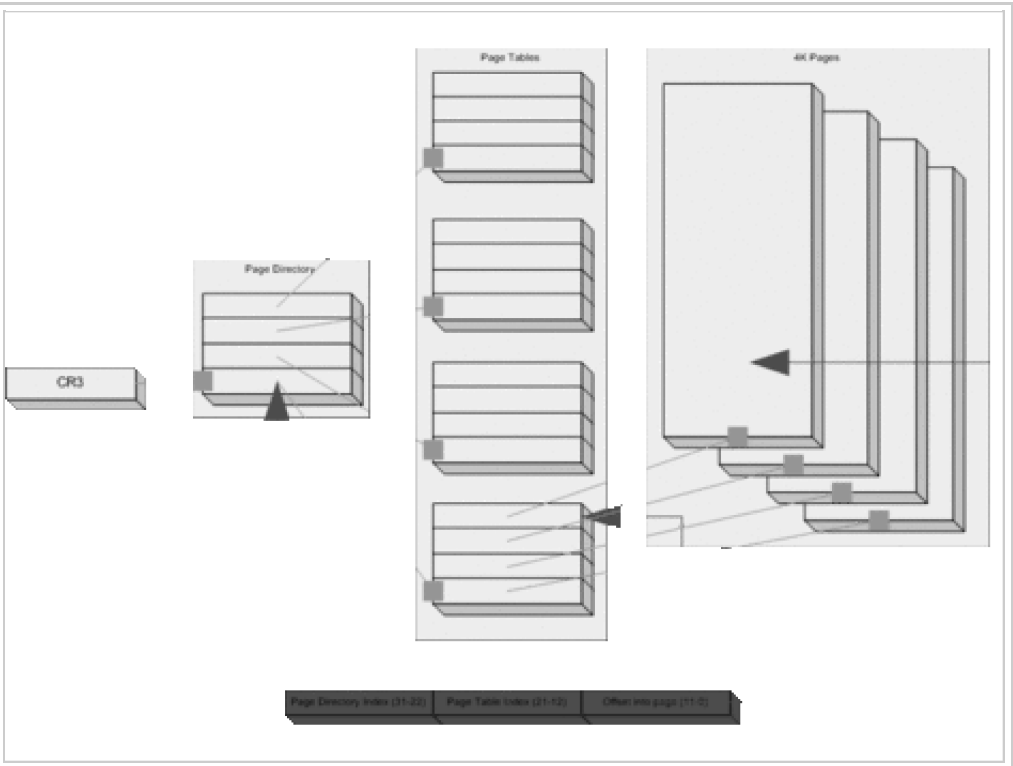
Paging

From OSDev Wiki



x86 Paging Structure

Contents

Overview

32-bit x86 processors support a 4GiB virtual address space and current 64 bit processors support a 256TiB virtual address space (with a theoretical maximum of 16EiB). Paging is a system which allows each process to see the full virtual address space, without actually requiring the full amount of physical RAM to be physically installed. In fact, current implementations of x86-64 has a current physical RAM limit of 1TiB and a theoretical limit of 4PiB of physical RAM.

In addition to this, paging introduces the benefit of page-level protection. In this system, user-level processes can only see and modify data which is paged in to their own address space, providing hardware isolation. System pages are also protected from user processes. On the x86-64 architecture, page-level protection now completely supersedes Segmentation as the memory protection mechanism. On the IA32 architecture, both paging and segmentation exist, but segmentation is now considered 'legacy'.

Once an Operating System has paging, it can also make use of other benefits and workarounds, such as linear framebuffer simulation for memory-mapped IO and paging out to disk, where disk storage space is used to free up physical RAM.

MMU

Paging is achieved through the use of the MMU (temporary: article 1, article 2). The MMU is a unit that transforms virtual addresses into physical addresses based on the current page table.This section focuses on the x86 MMU.

Overview

On the x86, the MMU maps memory through a series of tables, two to be exact. They are the paging directory, and the paging table.

Both tables contain 1024 4byte entries, making them each 4kb. In the page directory, each entry points to a page table. In the page table, each entry points to a physical address that is then mapped to the virtual address found by calculating the offset within the directory and the offset within the table. This can be done as the entire table system represents a linear 4gb virtual memory map.

Page Directory

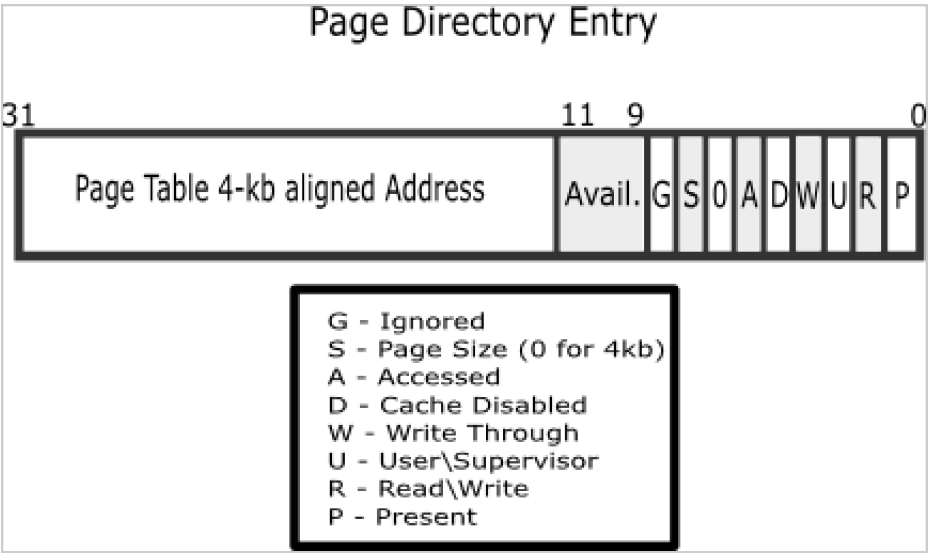
The topmost paging structure is the page directory. It is essentially an array of page directory entries that take the following form.

Note: With 4mb pages, bits 21 through 12 are Reserved!

The page table address field represents the physical address of the page table that manages the four megabytes at that point. Please note that it is very important that this address be 4 KiB aligned. This is needed, due to the fact that the last bits of the 32-bit value are overwritten by access bits and such.

- S, or 'Page **S**ize' stores the page size for that specific entry. If the bit is set, then pages are 4 MiB in size. Otherwise, they are 4 KiB. Please note that for 4 MiB pages PSE have to be enabled.
- A, or '**A**ccessed' is used to discover whether a page has been read or written to. If it has, then the bit is set, otherwise, it is not. Note that, this bit will not be cleared by the CPU, so that burden falls on the OS (if it needs this bit at all).

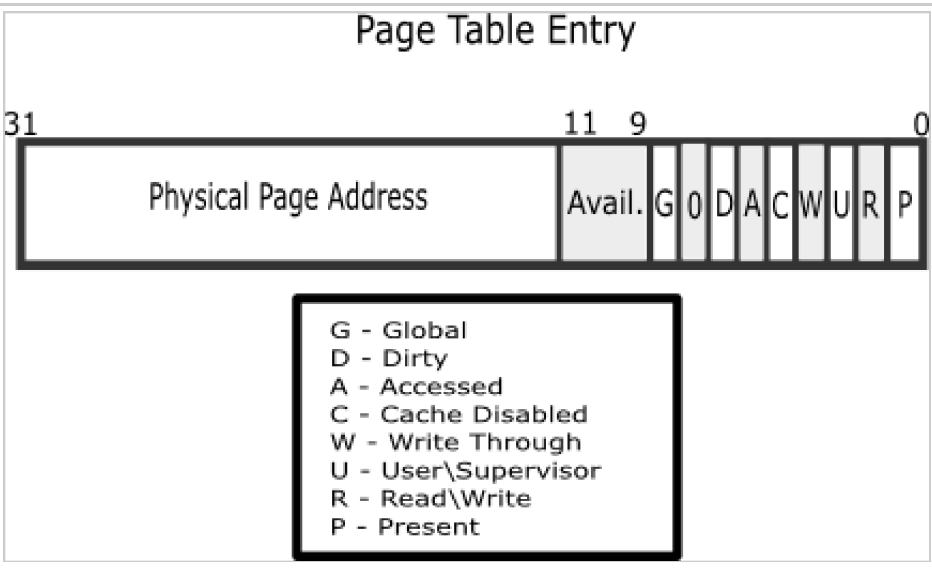
- D, is the 'Cache **D**isable' bit. If the bit is set, the page will not be cached. Otherwise, it will be.
- W, the controls '**W**rite-Through' abilities of the page. If the bit is set, write-through caching is enabled. If not, then write-back is enabled instead.
- U, the '**U**ser\Supervisor' bit, controls access to the page based on privilege level. If the bit is set, then the page may be accessed by all; if the bit is not set, however, only the supervisor can access it. For a page directory entry, the user bit controls access to all the pages referenced by the page directory entry. Therefore if you wish to make a page a user page, you must set the user bit in the relevant page directory entry as well as the page table entry.
- R, the '**R**ead/Write' permissions flag. If the bit is set, the page is read/write. Otherwise when it is not set, the page is read-only. The WP bit in CR0 determines if this is only applied to userland, always giving the kernel write access (the default) or both userland and the kernel (see Intel Manuals 3A 2-20).
- P, or '**P**resent'. If the bit is set, the page is actually in physical memory at the moment. For example, when a page is swapped out, it is not in physical memory and therefore not 'Present'. If a page is called, but not present, a page fault will occur, and the OS should handle it. (See below.)



A Page Directory Entry

Page Table

In each page table, as it is, there are also 1024 entries. These are called page table entries, and are **very** similar to page directory entries.



A Page Table Entry

Note: Only explanations of the bits unique to the page table are below.

The first item, is once again, a 4kb aligned physical address. Unlike previously, however, the address is not that of a page table, but instead a 4kb block of physical memory that is then mapped to that location in the page table and directory.

The Global, or 'G' above, flag, if set, prevents the TLB from updating the address in it's cache if CR3 is reset. Note, that the page global enable bit in CR4 must be set to enable this feature.

If the Dirty flag ('D') is set, then the page has been written to. This flag is not updated by the CPU, and once set will not unset itself.

The 'C' bit is 'D' bit above.

INVLPG

INVLPG is an instruction available since the 486 that invalidates a single page table entry in the TLB. Intel notes that this instruction may be implemented differently on future processes, but that this alternate behavior must be explicitly enabled. INVLPG modifies no flags.

NASM example:

```
      invlpg [0]
```

Inline asm in GCC (from Linux kernel source):

```
static inline void __native_flush_tlb_single(unsigned long addr)
{
   asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

Example

Say I loaded my kernel to 0x100000. However, I want it mapped to 0xc0000000. After loading my kernel, I initiate paging, and set up the appropriate tables. (See Higher Half Kernel) After Identity Paging the first megabyte, I start to create my second table (ie. at entry #768 in my directory.) to map 0x100000 to 0xc0000000. My code could be like:

```
mov eax, 0x0
mov ebx, 0x100000
.fill_table:
     mov ecx, ebx
     or ecx, 3
     mov [table_768+eax*4], ecx
     add ebx, 4096
     inc eax
     cmp eax, 1024
     je .end
     jmp .fill_table
.end:
```

Enabling

Enabling paging is actually very simple. All that is needed is to load CR3 with the address of the page directory and to set the paging bit of CR0.

```
mov eax, [page_directory]
mov cr3, eax

mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

To enable PSE (4 MiB pages) the following code is required.

```
mov eax, cr4
or eax, 0x00000010
mov cr4, eax
```

Physical Address Extension

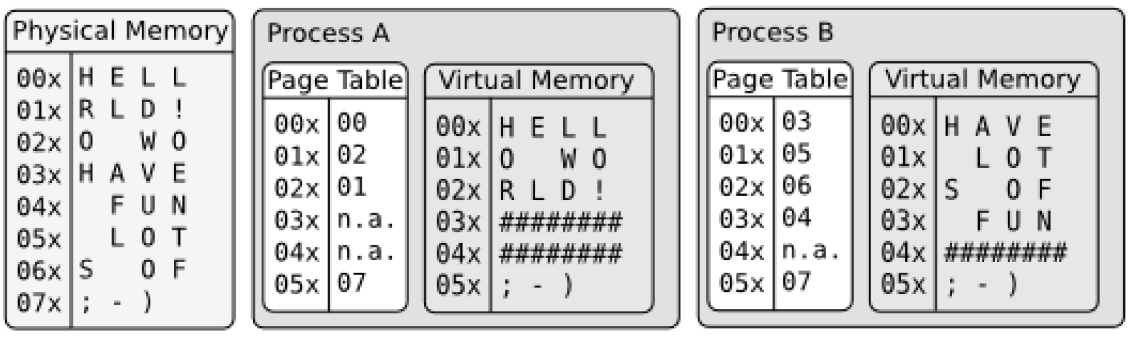
All Intel processors since Pentium Pro (with exception of the Pentium M at 400 Mhz) and all AMD since the Athlon series implement the Physical Address Extension (PAE). This feature allows you to access up to 64 GB (2^36) of RAM. You can check for this feature using CPUID. Once checked, you can activate this feature by setting bit 5 in CR4. Once active, the CR3 register points to a table of 4 64bit entries, each one pointing to a page directory made of 4096 bytes (like in normal paging), divided into 512 64bit entries, each pointing to a 4096 byte page table, divided into 512 64bit page entries.

Usage

Due to the simplicity in the design of paging, it has many uses.

Virtual Address Spaces

In a paged system, each process may execute in its own 4gb area of memory, without any chance of effecting any other process's memory, or the kernel's.



paging illustrated: two process with different views of the same physical memory

Virtual Memory

Because paging allows for the dynamic handling of unallocated page tables, an OS can swap entire pages, not in current use, to the hard drive where they can wait until they are called. In the mean time, however, the physical memory that they were using can be used elsewhere. In this way, the OS can manipulate the system so that programs actually seem to have more RAM than there actually is.

More...

Manipulation

The CR3 value, that is, the value containing the address of the page directory, is in physical form. Once, then, the computer is in paging mode, only recognizing those virtual addresses mapped into the paging tables, how can the tables be edited and dynamically changed?

Many prefer to map the last PDE to itself. The page directory will look like a page table to the system. To get the physical address of any virtual address in the range 0x00000000-0xFFFFF000 is then just a matter of:

```
void * get_physaddr(void * virtualaddr)
{
    unsigned long pdindex = (unsigned long)virtualaddr >> 22;
    unsigned long ptindex = (unsigned long)virtualaddr >> 12 & 0x03FF;

    unsigned long * pd = (unsigned long *)0xFFFFF000;
    // Here you need to check whether the PD entry is present.

    unsigned long * pt = ((unsigned long *)0xFFC00000) + (0x400 * pdinde›
    // Here you need to check whether the PT entry is present.

    return (void *)((pt[ptindex] & ~0xFFF) + ((unsigned long)virtualaddr
}
```

To map a virtual address to a physical address can be done as follows:

```
void map_page(void * physaddr, void * virtualaddr, unsigned int flags)
{
    // Make sure that both addresses are page-aligned.

    unsigned long pdindex = (unsigned long)virtualaddr >> 22;
    unsigned long ptindex = (unsigned long)virtualaddr >> 12 & 0x03FF;

    unsigned long * pd = (unsigned long *)0xFFFFF000;
    // Here you need to check whether the PD entry is present.
    // When it is not present, you need to create a new empty PT and
    // adjust the PDE accordingly.

    unsigned long * pt = ((unsigned long *)0xFFC00000) + (0x400 * pdinde›
    // Here you need to check whether the PT entry is present.
    // When it is, then there is already a mapping present. What do you c
```

```
    pt[ptindex] = ((unsigned long)physaddr) | (flags & 0xFFF) | 0x01; //

    // Now you need to flush the entry in the TLB
    // or you might not notice the change.
}
```

Unmapping an entry is essentially the same as above, but instead of assigning the `pt[ptindex]` a value, you set it to 0x00000000 (i.e. not present). When the entire page table is empty, you may want to remove it and mark the page directory entry 'not present'. Of course you don't need the 'flags' or 'physaddr' for unmapping.

Page Faults

A page fault exception is caused when a process is seeking to access an area of virtual memory that is not mapped to any physical memory, when a write is attempted on a read-only page, when accessing a PTE or PDE with the reserved bit or when permissions are inadequate.

Handling

The CPU pushes an error code on the stack before firing a page fault exception. The error code must be analyzed by the exception handler to determine how to handle the exception. The bottom 3 bits of the exception code are the only ones used, bits 3-31 are reserved.

```
Bit 0 (P) is the Present flag.
Bit 1 (R/W) is the Read/Write flag.
Bit 2 (U/S) is the User/Supervisor flag.
```

The combination of these flags specify the details of the page fault and indicate what action to take:

```
US RW  P - Description
0  0  0 - Supervisory process tried to read a non-present page entry
0  0  1 - Supervisory process tried to read a page and caused a protection fault
0  1  0 - Supervisory process tried to write to a non-present page entry
0  1  1 - Supervisory process tried to write a page and caused a protection fault
1  0  0 - User process tried to read a non-present page entry
1  0  1 - User process tried to read a page and caused a protection fault
1  1  0 - User process tried to write to a non-present page entry
1  1  1 - User process tried to write a page and caused a protection fault
```

When the CPU fires a page-not-present exception the CR2 register is populated with the linear address that caused the exception. The upper 10 bits specify the page directory entry (PDE) and the middle 10 bits specify the page table entry (PTE). First check the PDE and see if it's present bit is set, if not setup a page table and point the PDE to the base address of the page table, set the present bit and iretd. If the PDE is present then the present bit of the PTE will be cleared. You'll need to map some physical memory to the page table, set the present bit and then iretd to continue processing.

Paging Tricks

The processor always fires a page fault exception when the present bit is cleared in the PDE or PTE regardless of the address. This means the contents of the PTE or PDE can be used to indicate a location of the page saved on mass storage and to quickly load it. When a page gets swapped to disk, use these entries to identify the location in the paging file where they can be quickly loaded from then set the present bit to 0.

See Also

Articles

- Identity Paging
- Page Frame Allocation
- Setting Up Paging
- Page Tables

External Links

- INVLPG thread (http://forum.osdev.org/viewtopic.php?f=1&t=18222)