My Paging Notes

<Monsieur Ashish Shukla alias Wah Java !!>

February 22, 2006

Abstract

This article explains paging feature available in IA-32 processors since 80386 till the latest Pentium 4 processors. And will cover each and every aspect of paging (both at OS and processor level). This article was based on my research on paging. Most of the stuff covered in this document is general, i.e. also applies to other CPU architectures. Sorry for my english. :-(

1 Intro

Paging is one of the great features which are provided by IA-32 processors. Generally, paging is used to emulate virtual memory. In paging, the address space is divided into pages, and each process accesses memory through virtual addresses, which are translated into physical addresses by processor with the help of page directories, and page tables provided by the Operating System.

2 Support in IA32

The size of page in IA32 architecture is 4K, 4M and 2M. The 4M pages are available only when `PSE` flag in register CR4 is set. The 2M pages are available only when `PAE` flag in register CR4 is set[1] is used. Each page begins at a page aligned address. An address is page aligned or aligned to page boundary if it is divisible by the page size, i.e.

`iif PageAddress MOD PageSize = 0` ⇒ `PageAddress is page aligned.`

e.g.
`3A4000h` is aligned to `4K` page boundary.
`100000h` is not aligned to `4M` page boundary, but aligned to `4K` page boundary.

Some formulae (usually implemented as macros) useful for aligning addresses to page boundary:

1. **Truncation**. This formula aligns address to the previous page boundary.

   ```
   AlignedAddress = (Address AND (NOT (PageSize - 1)))
   ```

[1] The 36-bit address is not discussed in this article

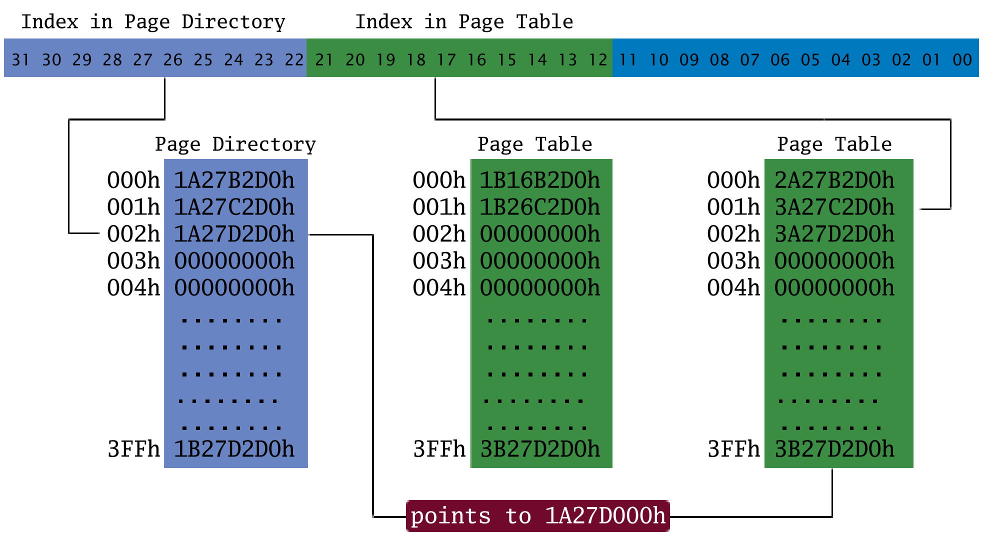

Figure 1: How page translation occurs ?

2. **Round Off**. This formula aligns address to the next page boundary.

```
AlignedAddress = ((Address + PageSize - 1) AND (NOT (PageSize - 1)))
```

Page Directory, and Page Tables are 4096 bytes wide buffers consisting of 1024 entries, each of width 4 bytes. The CR3 register points to the page directory. It is also known as *Page Directory Base Register* (PDBR). Each PDE (page directory entry) points to physical address of a page table or a 4M wide page [2]. Each PTE (page table entry) points to a 4K page. The page directory and page tables are aligned on 4K page boundary.
Higher 10 bits of the address are used as an index in page directory to lookup a page table. The bits 12-21 are used as an index into that page table to lookup the physical address of the page. If PDE points to a 4M page, then no page table lookup is needed. This is shown in Figure 1 on Page 2.
These *PDE*s and *PTE*s have some attributes which controls caching, access rights (read/write), user/supervisor, presence of page etc. The detailed structures of *PDE* and *PTE* are available in [IA32SDM3].
In this article, we need to get aware of two attributes of *PDE* and *PTE*, which are access right bit, user/supervisor bit. The access right bit can be 0 for read-only, 1 for read-write access. The user/supervisor bit can be 0 for *supervisor* and 1 for *user*. The pages which are marked *supervisor* can not be accessed by the user in any way. The pages which are marked *user* and also marked *readonly*, will be read-only for user, and any attempt to write on them by user, will generate a #PF[3]. For *supervisor*, all pages are readable and writable.
There is a WP flag in register CR0 which controls *supervisor* mode sensitivity to *user* mode pages. If WP flag is clear then, *supervisor* can read and write all

[2] if PSE bit of CR4 is set
[3] Page Fault

pages. But when `WP` flag is set, then *supervisor* is not allowed to write on *user* mode *readonly* pages. This feature is used in *COW* [4] protection, explained later.

3 How is it implemented in OS ?

You might have heard that in Windows and Linux each process runs in its own separate virtual address space. This is implemented with the help of paging. Each *task* has its own Page Directory and its own set of page tables. The page tables don't contain pages belonging to other *task*. Although other tasks are also loaded in the memory. But they're not visible in virtual address space. So, in this way all *tasks* executes loads at same virtual address space without overwriting each other's memory. When *context-switching* occurs, the page directory of next *task* is restored, and now pages belong to that *task* are visible. So, in this way, each *task* run in its own separate virtual address space. If any *task* wants access to other *task* pages, it can request it from OS via syscalls. Now, it is upto OS to grant or deny access. If it grants access, then it maps the requested pages into the requesting *task*'s virtual address space.

4 COW

COW stands for *Copy-on-write*. Imagine you're running multiple instances of single application simultaneously. So, this means for each new instance, OS has to reload the application executable in memory, and also needs to allocate memory for all its dependencies. And since, application executable is same for each instance, i.e. its code is same for each instance. So loading the same code, multiple times will lead to the wastage of memory, and time (the time spent in reading the executable from storage). So, in order to avoid this wastage of memory and time, *COW* is used.
In *COW*, the *sharable* (usually code) pages are shared (i.e. mapped *readonly* into their page directories) among the multiple tasks. So when first instance of application is loaded, the executable is loaded in the memory from storage. And when next instance of same application is loaded, then instead of reloading executable from storage, OS maps the *shareable* physical address space of the executable into the virtual address space (i.e. Page maps) of the new process. So, in this way, for each new task, only new mappings have to be set up. If any process attempts to write into the shared page, the processor generates a #PF exception. The OS handles this exception by making a private copy of that shared page, unmapping *shared* page, and then remapping private copy of that page into the task's page maps at the same location. The new *private copy* is mapped with *read/write* access. So any modifications to page by one *task* won't alter other *task*'s page. Usually, OS loads executables with code pages marked *readonly*. So, in order to write into the code pages, task has to explicitly request OS to grant *COW* access to the pages via syscalls. When OS grants *COW* access, it doesn't alter pagemaps (i.e. it doesn't marks page *read/write* in pagemaps instead it marks *read/write* in its internally maintained datastructures). So, when #PF occurs it checks the fault address, that whether

[4] Copy-on-write

that address is marked *read/write* in its own internal data structures or not. If it is not marked *read/write*, it is an *Access Violation* [5] error, but if it is marked *read/write* then clearly it is the case of *COW*. OS automatically marks pages used by DLLs, Shared Objects, Shared libraries for *COW* protection but they're still *readonly* for processor.
Note: *COW* protection is not provided by processor, instead it is emulated by OS. So, and also it is possible to implement paging in OS without adding support for *COW*. But, having *COW* is an advantage.

5 What if *supervisor* modifies *COW* pages ?

As it is mentioned previously that *supervisor* code can modify any page without caring about the access, this means *supervisor* code won't cause #PF and hence *COW* won't happen at all. To change this behavior, `WP` flag in register `CR0` needs to be set, which prevents *supervisor* from writing on to *usermode* pages marked *readonly*, and hence causing #PF, which then causes *COW* to happen.

6 TLB

TLB stands for *Translation Lookaside Buffer*. For address translation in paging mode, pagemap entries are needed. As, we know that pagemaps are stored in RAM, so access time is more. To overcome this problem, *TLB*s are invented. *TLBs* are the on-chips caches used by processor to store recently accessed *PDE*s and *PTE*s. When any pagemap entry is accessed for first time, it is cached in the *TLB*, and after that any subsequent accesses for that pagemap entry will be using *TLB*. Any pagemap entry not present in *TLB* will be fetched into *TLB*. If any pagemap entry is modified in memory, which was also present in *TLB*, should be invalidated by OS, otherwise the memory accesses for that entry are still using old physical address, and attributes. There are two bits in pagemap entries which are useful in *TLB* invalidation logic for OS.

1. **Accessed Bit**. Processor sets this bit in pagemap entry, whenever it accesses that pagemap entry.
2. **Dirty Bit**. Processor sets this bit in *PTE*, when *PTE* is used for store operations. This bit is present in *PTE*.

These bits are *sticky*, means if they're once set, they're not implicitly cleared by the processor. The *accessed* bit can be used to check whether the pagemap entry has been used by the processor, which means that the entry might be in *TLB*, so it is time to invalidate the *TLB* entry. The *dirty* bit is useful for paging in and paging out stuff[6].

7 Invalidating TLB

*TLB*s can be invalidated in following ways:

[5] SIGSEGV signal in Linux
[6] Might be useful in *TLB* stuff. As I'll know, I'll update this document

1. By reloading page directory whole *TLBs* can be invalidated. The page directory can be loaded by a following `MOV` instruction:
   `MOV CR3, EAX`
   or at task switch[7] when `CR3` register is reloaded. Also, see section 8 for Global Pages.

2. Specific *TLB* entries can be invalidated using `INVLPG` instruction as shown below:
   `INVLPG [EAX]`
   In the above example, the *TLB* entry for the page containing the memory address specified in EAX register is flushed[8].

3. Any changes to `PG`, `PE` flags in `CR0` register will flush all[9] *TLBs*.

Note: The *TLBs* can be invalidated in privilege level 0 only.

8 Global Pages

Suppose we want some part of memory to be shared (not really shared but mapped) among all programs at common addresses. To share those pages, we mapped them in the pagemaps of all processes. But each time context switch happens (or page directory is reloaded), all page map entries flushed automatically from *TLBs*. To avoid this, the concept of Global Pages is introduced in IA-32 processors. When `PGE` flag in `CR4` is set, global pages are enabled . The pagemap entries for global pages are explicitly hinted to avoid their automatic flushing from *TLBs*. This is achieved by a bit in pagemap entries which specifies whether the page corresponding to pagemap entry is a global page or not. The cached *TLB* entries, corresponding to global pages are prevented from automatically invalidating on task switch and reloading of `CR3` register. To flush *TLB* entries for specific global page use `INVLPG` instruction. To flush all pages (including global pages) clear the `PGE` flag in `CR4` register and then reload `CR3` register.

9 Locking Pages

Page locking is a feature provided by OS, not processor. In page locking memory pages are locked in memory. The pages are not swapped out (or paged out) to the disk. Page locking is done for several reasons like:

Speed up. Page faults are prevented since the page is always present in memory, hence no page faults. Since, there are no page fault, all process's memory is present in RAM, therefore application speeds up.

Security. For some security reasons, pages are locked. e.g. You've stored password (or confidential information) in RAM, then if that page is paged out, to the page file (or swap file), then your password and confidential information also goes to disk which means anybody can see your password.

[7]A *context switching* mechanism provided by the processor
[8]or invalidated
[9]Global entries too

So, in order to prevent this the page containing your password should be locked in memory.

10 Mapping to memory

Mapping to the memory is the fastest way to access a file (or devices). In memory mapping, a file is mapped into memory. i.e. In the internal data structures of the OS kernel, that portion of memory is marked as "mapped to a file", and marked as *not-present* in *PTEs*. When initial memory access is made, a page fault occurs, and then OS kernel loads the of required page of file in the memory and marks its corresponding *PTE present*. So, no explicit read from file occurs. And since, memory accesses are faster than disk accesses.

11 Page Execution

Page execution controls whether page is executable or not or whether code can be executed from those pages or not. On IA-32, page execution is not supported[10]. In some OSs, esp. those which run on different processor architectures, expose their APIs in accordance with hardware abstraction layer. On some processor architectures, page execution is supported, so in order to provide a similar interface across all architures, these OSs support execution bit, which is not effective on IA32 architecture.
Some OSs emulate the page execution by some other means, e.g. code segment resizing[11]. In code segment resizing, the limits of the code segment descriptor is changed. e.g. To disable execution on code segment's last 2 pages, invoke the OSs page protection routine(s) with execution bit clear. The OS will then decrease the upper limit of the code segment by 2 pages. So if program tries to execute or tries to read via code segment selector, from those last 2 pages, the segment limit check fails, and a exception is raised. But what if we wanted to disable execution of some pages which are located in the middle of the code segment, then segment resizing won't work because there are only two limits[12] of the segment descriptor. Similarly making the last page executable, means making your whole address space executable.
There are some workarounds for this page based execution out of which one i.e. code segment resizing is mentioned above. For those, who are interested in researching on this feature, there is a project named **PAX**[13], whose objective is to research various defense mechanism againsthe exploitation of software bugs that give an attacker arbitrary read/write access to the attacked task's address space going on.

12 NXE Bit

The NXE bit (also known as Execute Disable bit) is available in recent IA-32 processors. The execute disable bit is available when PAE mode is active (i.e.

[10]On some recent processors, page execution bit is available, see section 12
[11]Linux does this
[12]Upper limit or lower limit
[13]See, http://pax.grsecurity.net/

```
00898000-008ad000 r-xp 00000000 03:09 3260429    /lib/ld-2.3.3.so
008ad000-008ae000 r--p 00014000 03:09 3260429    /lib/ld-2.3.3.so
008ae000-008af000 rw-p 00015000 03:09 3260429    /lib/ld-2.3.3.so
008b5000-009d6000 r-xp 00000000 03:09 2932849    /lib/tls/libc-2.3.3.so
009d6000-009d8000 r--p 00120000 03:09 2932849    /lib/tls/libc-2.3.3.so
009d8000-009da000 rw-p 00122000 03:09 2932849    /lib/tls/libc-2.3.3.so
009da000-009dc000 rw-p 009da000 00:00 0
08048000-0804c000 r-xp 00000000 03:09 2031672    /bin/cat
0804c000-0804d000 rw-p 00003000 03:09 2031672    /bin/cat
08d5c000-08d7d000 rw-p 08d5c000 00:00 0
f6dda000-f6fda000 r--p 00000000 03:09 479119     /usr/lib/locale/locale-archive
f6fda000-f6fdb000 rw-p f6fda000 00:00 0
fee3a000-ff000000 rw-p fee3a000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

Figure 2: A dump of /proc/self/map for /bin/cat

`PAE` bit in `CR4` register)

13 OS API Routines

The OS exposes some page manipulation routines through syscall interface. In some OS, the syscall[14] interface is publically exposed, whereas in some OS it is private. e.g. Unix etc. OSs exposes syscall interface, whereas Windows don't exposes. In Windows, you can access kernel routines indirectly via system DLLs (e.g. ntdll.dll). The routines in these system DLLs make syscalls to the kernel. In this section, I'll briefly document library routines[15] for page manipulation on Windows and Unix[16]. For detailed documentation refer to [PSDK] for Windows routines and [UNIXMAN] for Unix routines.

1. **Page Protection**. Page protection routines are used to change protection of pages. `VirtualProtect()` and `mprotect()` routines are used for page protection in Windows and Unix respectively. Page protection can be queried using `VirtualQuery()` routine in Windows. In Unix, the `/proc` filesystem is used for querying page protection. It contains a file named `maps` in process directories. The maps for current process is listed in `/proc/curproc/maps` (in FreeBSD) or `/proc/self/maps` (in GNU/Linux) directory depending on the Unix you're running. A dump of `/proc/self/map` is shown in Figure 2. First column contains address range. Second column contains type of protection (`r=read`, `w=write`, `x=execute`, `p=copy-on-write`, `s=shared`). Third column contains offset into the file, i.e. from where the file is mapped. Fourth column contains the device. Fifth column contains the i-node. The last column contains the path of the file mapped.

[14]System call i.e. call to kernel running in privileged mode
[15]For syscalls interface, check developer documentation of the OS
[16]See [IEEE1003.1]

2. **Page Locking/Unlocking**. Pages are locked using `VirtualLock()` and `mlock()` routines in Windows and Unix respectively. Pages are unlocked using `VirtualUnlock()` and `munlock()` routines in Windows and Unix respectively. There are `mlockall()` and `munlockall()` routines in Unix which are used to lock and unlock all pages (code, data, stack etc.) mapped in the address space of the process, respectively. The memory locks do not stack, in other words locking multiple times can be unlocked by a single call to the unlock routine.

3. **Memory Mapping/Unmapping**. Files can be mapped to memory using `mmap()` and `MapViewOfFileEx()` routines in Unix and Windows respectively. Their parameters are file handle or file descriptor, offset in the file, length of mapping, page protection flags, type of mapping to create, base address where file has to be mapped (used as hint). In Windows, mapping stuff is slightly different from Unix, `CreateFileMapping()` or `OpenFileMapping()` are used to initialize a mapping object. The mapping opject is then passed to `MapViewOfFileEx()` routine. To unmap files from memory, `munmap()` and `UnmapViewOfFile()` routines are used, in Unix and Windows respectively.

14 Units used

`K` stands for Kilobyte. `1K` = `1024` bytes
`M` stands for Megabyte. `1M` = `1024K` = `1048576` bytes

15 Operators used

`MOD` - Remainder Function
`AND` - Boolean AND
`NOT` - Boolean NOT
$\Rightarrow$ - Implies that

16 Terminology

Context Switch. Context switching refers to the switching from one process to another process. In Multitasking systems, context switch occurs frequently.

User Code. Code running in privilege 3 usually user apps.

Supervisor Code. Code running in privilege 0, 1 and 2 usually OS kernel, drivers.

#PF. Page Fault, Interrupt 14.

PE Flag. Protection Enable, Bit 0 of CR0. When set switches processor into protected mode.

PG Flag. Paging Enable, Bit 31 of CR0. When set paging is enabled.

WP Flag. Write Protect, Bit 16 of CR0. When set enables supervisor write protection over usermode pages.

PSE flag. Page Size Extension, Bit 4 of CR4. When set enables 4M pages.

PAE flag. Page Address Extension, Bit 5 of CR4. When set enables 2M pages and 36-bit addressing.

CR0, CR1, CR2, CR3, CR4. Control Registers in IA32.

Pagemaps. A term used for describing page directories and page tables.

COW. Copy-on-write protection. See, section 4, page no. 3.

TLB. Translation Lookaside Buffer. See, section 6, page no. 4.

OS. Operating System e.g. Windows, Linux, FreeBSD, MS-DOS etc. In this document OS refers to protected mode OS such as Windows, Linux, FreeBSD etc.

17 Dedications

Dedicated to Gautam Renjen a.k.a. AIDS

References

[IA32SDM3] IA-32 Intel Architecture Software Developer's Manual, Volume 3:System Programming Guide (Order No. 253668).

[IEEE1003.1] The Open Group Base Specifications Issue 6, IEEE Std 1003.1-2001

[PSDK] Platform SDK Documentation, Available from http://msdn.microsoft.com/msdownload/platformsdk/sdkupdate/

[UNIXMAN] Unix man pages

[GLIBCINF] GNU C Library Reference Manual, Edition 0.10, for version 2.3.x of the GNU C library