Virtual memory management

# Theory

Although virtual memory is very useful, it is wholly dependent on hardware support. It cannot be emulated by software. Luckily, the x86 has just such a thing. It's called the MMU (memory management unit), and it handles all memory mappings due to segmentation and paging, forming a layer between the CPU and memory.

Virtual memory is an abstract principle. As such it requires concretion through some system/algorithm. Both segmentation and paging are valid methods for implementing virtual memory. However, segmentation is becoming obsolete. Paging is the newer, better alternative for the x86 architecture.

Paging works by splitting the virtual address space into blocks called pages, which are usually 4KB in size. Pages can then be mapped on to frames: equally sized blocks of physical memory. As you can see here we need physical memory management of some sorth, because we need to be able to get an unallocated physical page that we can map to a specified virtual address.

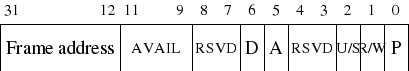
A very important function that the virtual memory manager should have is the ability to map a random page to specified virtual address at any time. This is accomplished by the map function. But a problem occurs when we try to map a virtual page when we’re out of pages and the page tables are full. Because then we have no virtual place to write a page entry to. Pro-Type solves this problem by allocating a 4MB region at 0x1A00000. There you can find every page entry, these entries are ordered like they are in the virtual address space. This memory is identity mapped so that we can directly pass the pointer to the page directory.

Right before we enable paging we need to map the directory as the last pageentry of the 1022nd table so that we can change a physical addresses of the pagetables.

# Practical

# Page entries

Each process normally has a different set of page mappings, so that virtual memory spaces are independent of each other. In the x86 architecture (32-bit) pages are fixed at 4KB in size. Each page has a corresponding descriptor word, which tells the processor which frame it is mapped to. Note that because pages and frames must be aligned on 4KB boundaries (4KB being 0x1000 bytes), the least significant 12 bits of the 32-bit word are always zero. The architecture takes advantage of this by using them to store information about the page, such as whether it is present, whether it is kernel-mode or user-mode etc. The layout of this word is in the picture underneath.



The fields in that picture are pretty simple, so let's quickly go through them.

* P:

Set if the page is present in memory.

* R/W:  
  If set, that page is writeable. If unset, the page is read-only. This does not apply when code is running in kernel-mode (unless a flag in CR0 is set).
* U/S:   
  If set, this is a user-mode page. Else it is a supervisor (kernel)-mode page. User-mode code cannot write to or read from kernel-mode pages.
* Reserved:   
  These are used by the CPU internally and cannot be trampled.
* A:   
  Set if the page has been accessed (Gets set by the CPU).
* D:

Set if the page has been written to (dirty).

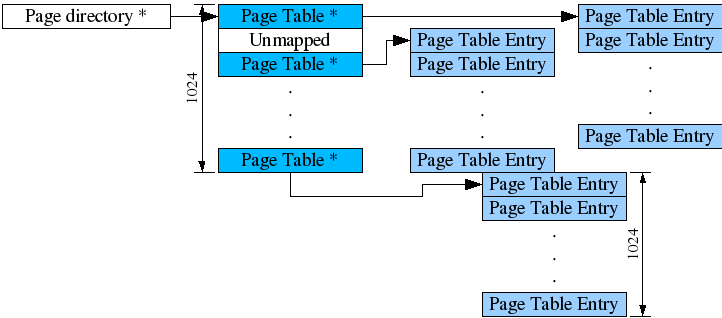
* AVAIL:  
  These 3 bits are unused and available for kernel-use.
* Page frame address:

The high 20 bits of the frame address in physical memory.

## Page directories/tables

Possibly you've been tapping on your calculator and have worked out that to generate a table mapping each 4KB page to one 32-bit descriptor over a 4GB address space requires 4MB of memory. 4MB may seem like a large overhead, and to be fair, it is. If you have 4GB of physical RAM, it's not much. However, if you are working on a machine that has 16MB of RAM, you've just lost a quarter of your available memory! What we want is something progressive, that will take up an amount of space proportionate to the amount of RAM you have.

Well, we don't have that. But Intel did come up with something similar - they use a 2-tier system. The CPU gets told about a \*page directory\*, which is a 4KB large table, each entry of which points to a page table. The page table is, again, 4KB large and each entry is a \*page table entry\*, described beneath.



This way, The entire 4GB address space can be covered with the advantage that if a page table has no entries, it can be freed and its present flag unset in the page directory.

* Note: A pagedirectory entry or pagetable is just a collection of 1024 page entrie pointers. And a Page directory is the collection of 1024 page table pointers and describes 4GB virtual address space.

## Enable paging

Enabling paging is extremely easy:

* Copy the location of your page directory into the CR3 register. This must, of course, be the physical address.
* Set the PG bit in the CR0 register. You can do this by OR-ing with 0x80000000.

## Page Faults

When a process does something the memory-management unit doesn't like, a page fault interrupt is thrown. Situations that can cause this are (not complete):

\* Reading from or writing to an area of memory that is not mapped (page entry/table's 'present' flag is not set)

\* The process is in user-mode and tries to write to a read-only page.

\* The process is in user-mode and tries to access a kernel-only page.

\* The page table entry is corrupted - the reserved bits have been overwritten.

\* The page fault interrupt is number 14, and looking at chapter 3 we can see that this throws an error code. This error code gives us quite a bit of information about what happened.

* Bit 0:

If set, the fault was not because the page wasn't present. If unset, the page wasn't present.

* Bit 1:

If set, the operation that caused the fault was a write, else it was a read.

* Bit 2:

If set, the processor was running in user-mode when it was interrupted. Else, it was running in kernel-mode.

* Bit 3:

If set, the fault was caused by reserved bits being overwritten.

* Bit 4:

If set, the fault occurred during an instruction fetch.

The processor also gives us another piece of information - the address that caused the fault. This is located in the CR2 register. Beware that if your page fault handler itself causes another page fault exception this register will be overwritten - so save it early!