Virtual Memory

# Overview

Virtual memory is a form of low level memory management that attempts to isolate processes into a separate address spaces and attempts to make memory fragmentation due to multiple processes running in separate address spaces transparent. Although there are many different forms of memory virtual memory, we are interested only in the x86 model.

The x86 model of virtual memory uses a two level table structure. The top most level table is called the page directory. The page directory contains 1024 entries each of which point to a single page table. Each page table contains 1024 entries each of which point to a memory address. Each entry in both tables is exactly 32 bits large making each table exactly 4096 bytes large or 4KiB large.

One of the ultimate goals in the x86 virtual memory model is to give each process its very own address space. This allows each process to be address independent from every other process thereby allowing applications to be hard coded to certain addresses. Address hard coded applications generally run faster than address independent code and are also easier to load and execute. This also allows processes to be ignorant about every other process running. Because each process gets its very own address space, each process can use the entire allotted address space as it sees fit. This eliminates the need for processes to be weary of using memory that another process is currently using. Imagine a process that is running amidst several other processes that are each taking up a few hundred megabytes. As each process runs, its stack and heap grow larger and larger. The chance that every process’s memory will be contiguous is very low and impractical. It is therefore imminent that each processes’ memory will have to become fragmented in order for the heap space requirements of each process to be met.

Virtual memory works around each of the previously mentioned issues. Virtual memory maps a virtual address to a physical address. The virtual address is, for the most part, a made up address. The physical address must be a valid memory address that exists in physical memory (i.e. RAM). Any virtual address can be mapped to any valid physical address. Multiple virtual addresses may be mapped to the same physical address. However, due to the design of virtual memory in the x86 architecture, a single virtual address may only be mapped to a single physical address. This creates a one to one relationship between a virtual address and a physical address but a many to one relationship between a physical address and a virtual address.

|  |  |  |
| --- | --- | --- |
| 0x0000 |  | 0x0000 |
| 0x1000 |  | 0x1000 |
| 0x2000 |  | 0x2000 |
| 0x3000 |  | 0x3000 |
| 0x4000 |  | 0x4000 |
| 0x5000 |  | 0x5000 |
| 0x6000 |  | 0x6000 |

# Scope

The scope of the problem exists when trying to dynamically update a page directory entry or a page table entry. This operation is common when allocating and deallocating memory. During these operations page table entries, and possibly page directory entries, have to be updated.

# Problem

When virtual memory is activated and being used by the CPU, all physical addresses must be accessed through the use of virtual addresses. Direct physical memory access is impossible. So in order to access a specific physical memory address, one needs to know what virtual address points to that physical address. In order to dynamically update a page directory entry or page table entry, one must access the physical address that those entries reside at. However, this task becomes much more complicated by the fact that virtual memory is being used by the CPU and we do not have direct physical memory access. Somehow, one must map virtual addresses to the physical addresses that these table entries reside at.