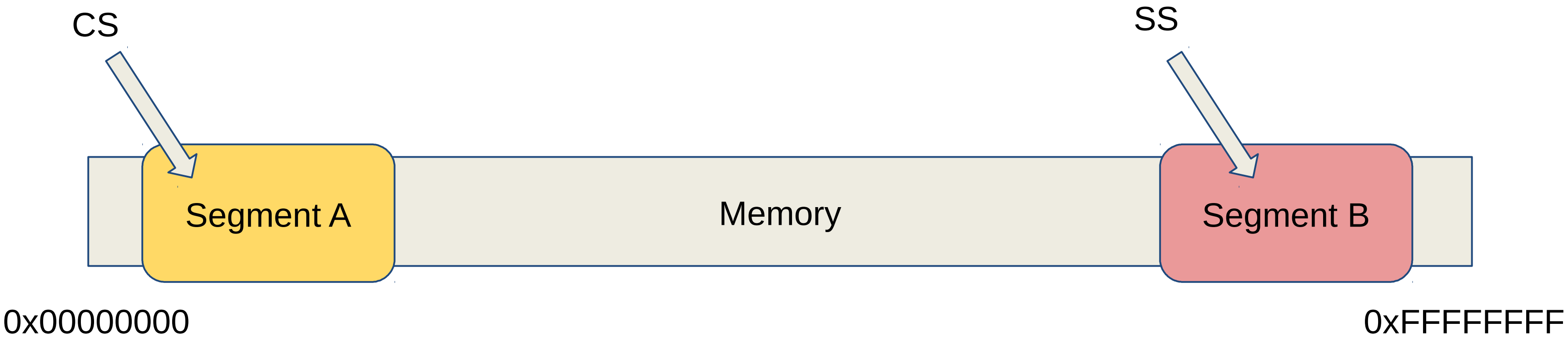
x86 Memory management

What was it you were talking about?

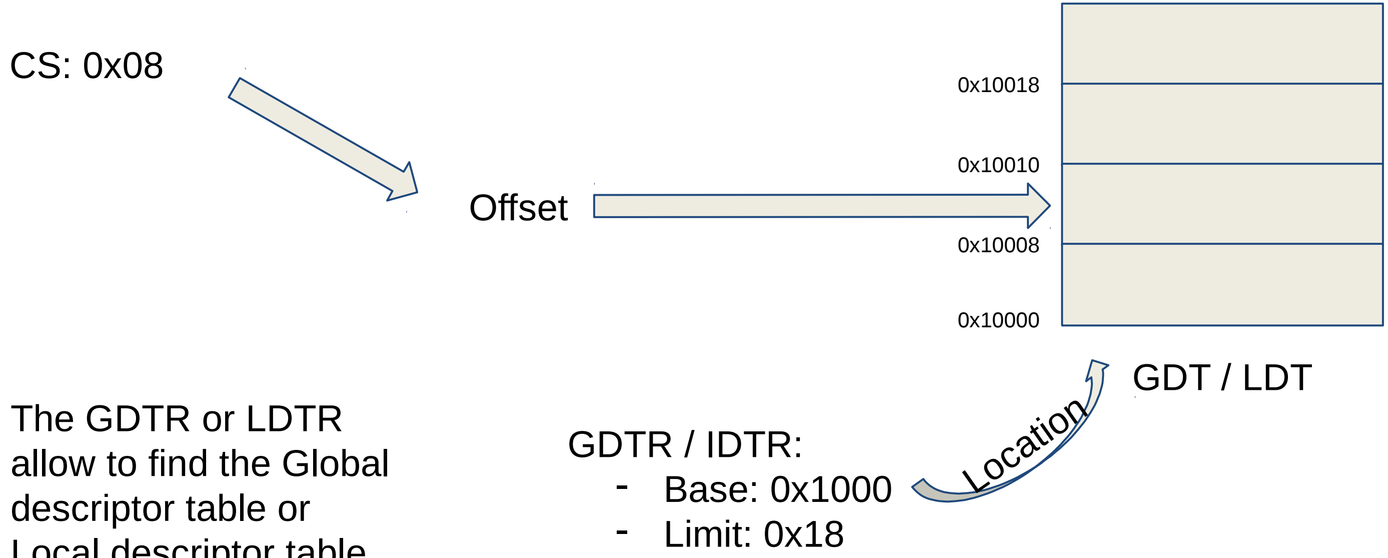
So, exactly what is the difference between segmentation and virtual memory?

What is segmentation?



A segment defines a **zone of memory**. A segment register then points to that zone allowing access to it.

Segmentation setup x86



The GDTR or LDTR allow to find the Global descriptor table or Local descriptor table on the memory.

On Linux

traps.c

```
#ifdef CONFIG_X86_32
	set_system_intr_gate(IA32_SYSCALL_VECTOR, entry_INT80_32);
	set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif

	/*
	 * Set the IDT descriptor to a fixed read-only location, so that the
	 * "sidt" instruction will not leak the location of the kernel, and
	 * to defend the IDT against arbit
	 * It will be reloaded in cpu_init
	__set_fixmap(FIX_RO_IDT, __pa_symb
	idt_descr.address = fix_to_virt(FI

	/*
	 * Should be a barrier for any ext
	 */
	cpu_init();
```

```
void cpu_init(void)
{
	int cpu = smp_processor_id();
	struct task_struct *curr = current;
	struct tss_struct *t = &per_cpu(cpu_tss, cpu);
	struct thread_struct *thread = &curr->thread;

	wait_for_master_cpu(cpu);

	/*
	 * Initialize the CR4 shadow before doing anything that could
	 * try to read it.
	 */
	cr4_init_shadow();

	show_ucode_info_early();

	pr_info("Initializing CPU#%d\n", cpu);

	if (cpu_feature_enabled(X86_FEATURE_VME) ||
	    boot_cpu_has(X86_FEATURE_TSC) ||
	    boot_cpu_has(X86_FEATURE_DE))
		cr4_clear_bits(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);

	load_current_idt();
	switch_to_new_gdt(cpu);

```

common.c

On Linux

common.c

```
void switch_to_new_gdt(int cpu)
{
	/* Load the original GDT */
	load_direct_gdt(cpu);
	/* Reload the per-cpu base */
	load_percpu_segment(cpu);
}
```

```
void load_direct_gdt(int cpu)
{
	struct desc_ptr gdt_descr;

	gdt_descr.address = (long)get_cpu_gdt_rw(cpu);
	gdt_descr.size = GDT_SIZE - 1;
	load_gdt(&gdt_descr);
}
```

desc.h

```
struct gdt_page {
	struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));

DECLARE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page);

/* Provide the original GDT */
static inline struct desc_struct *get_cpu_gdt_rw(unsigned int cpu)
{
	return per_cpu(gdt_page, cpu).gdt;
}
```

```
#define load_gdt(dtr)				native_load_gdt(dtr)
```

```
static inline void native_load_gdt(const
{
	asm volatile("lgdt %0"::"m" (*dt
}
```

On Linux

Bolded bits shows the DPL (descriptor privilege level).
0 = Kernel mode, 3 = User mode.
Definite proof that Linux only uses 2 of the 4 rings available on x86!

common.c

```
DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
#ifdef CONFIG_X86_64
	/*
	 * We need valid kernel segments for data and code in long mode too
	 * IRET will check the segment types  kkeil 2000/10/28
	 * Also sysret mandates a special GDT layout
	 *
	 * TLS descriptors are currently at a different place compared to i386.
	 * Hopefully nobody expects them at a fixed place (Wine?)
	 */
	[GDT_ENTRY_KERNEL32_CS]		= GDT_ENTRY_INIT(0xc09b, 0, 0xfffff),
	[GDT_ENTRY_KERNEL_CS]		= GDT_ENTRY_INIT(0xa09b, 0, 0xfffff),
	[GDT_ENTRY_KERNEL_DS]		= GDT_ENTRY_INIT(0xc093, 0, 0xfffff),
	[GDT_ENTRY_DEFAULT_USER32_CS]	= GDT_ENTRY_INIT(0xc0fb, 0, 0xfffff),
	[GDT_ENTRY_DEFAULT_USER_DS]	= GDT_ENTRY_INIT(0xc0f3, 0, 0xfffff),
	[GDT_ENTRY_DEFAULT_USER_CS]	= GDT_ENTRY_INIT(0xa0fb, 0, 0xfffff),
#else
	[GDT_ENTRY_KERNEL_CS]		= GDT_ENTRY_INIT(0xc09a, 0, 0xffff
	[GDT_ENTRY_KERNEL_DS]		= GDT_ENTRY_INIT(0xc092, 0, 0xff
	[GDT_ENTRY_DEFAULT_USER_CS]	= GDT_ENTRY_INIT(0xc0fa, 0, 0xff
	[GDT_ENTRY_DEFAULT_USER_DS]	= GDT_ENTRY_INIT(0xc0f2, 0, 0xff
```

Looking at the flags for the 32 bits version, you should see the following binary pattern for Kernel mode code:

0b 1100 0000 1**00**1 1010

Looking at the flags for the 32 bits version, you should see the following binary pattern for User mode code:

0b 1100 0000 1**11**1 1010

```
/*
 * FIXME: Accessing the desc_struct through its fields is more elegant,
 * and should be the one valid thing to do. However, a lot of open code
 * still touches the a and b accessors, and doing this allow us to do it
 * incrementally. We keep the signature as a struct, rather than a union,
 * so we can get rid of it transparently in the future -- glommer
 */
/* 8 byte segment descriptor */
struct desc_struct {
	union {
		struct {
			unsigned int a;
			unsigned int b;
		};
		struct {
			u16 limit0;
			u16 base0;
			unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
			unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
		};
	};
} __attribute__((packed));

#define GDT_ENTRY_INIT(flags, base, limit) { { { \
		.a = ((limit) & 0xffff) | (((base) & 0xffff) << 16), \
		.b = (((base) & 0xff0000) >> 16) | (((flags) & 0xf0ff) << 8) | \
			((limit) & 0xf0000) | ((base) & 0xff000000), \
	} } }
```

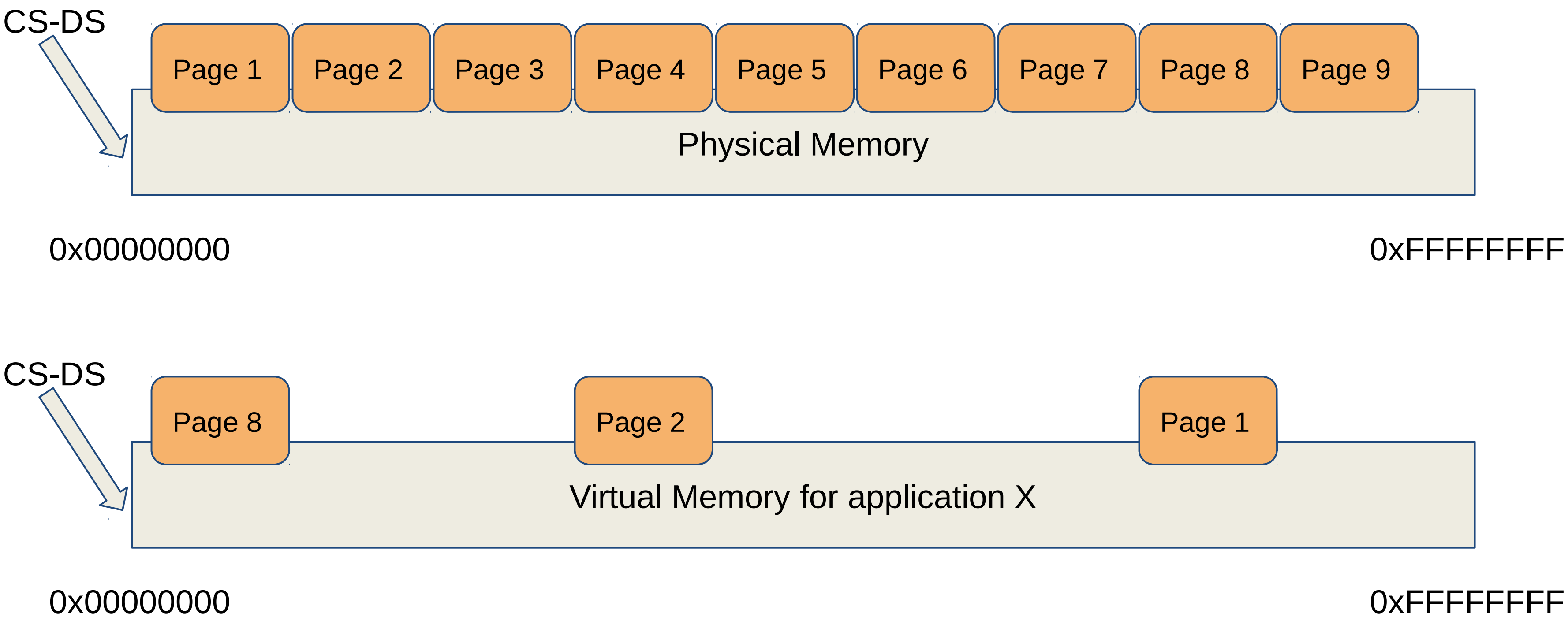
desc_defs.h

Alright, let's write an OS…
Just kidding, let's talk about virtual memory...

On x86, granularity in memory management is made possible by the use of virtual memory.

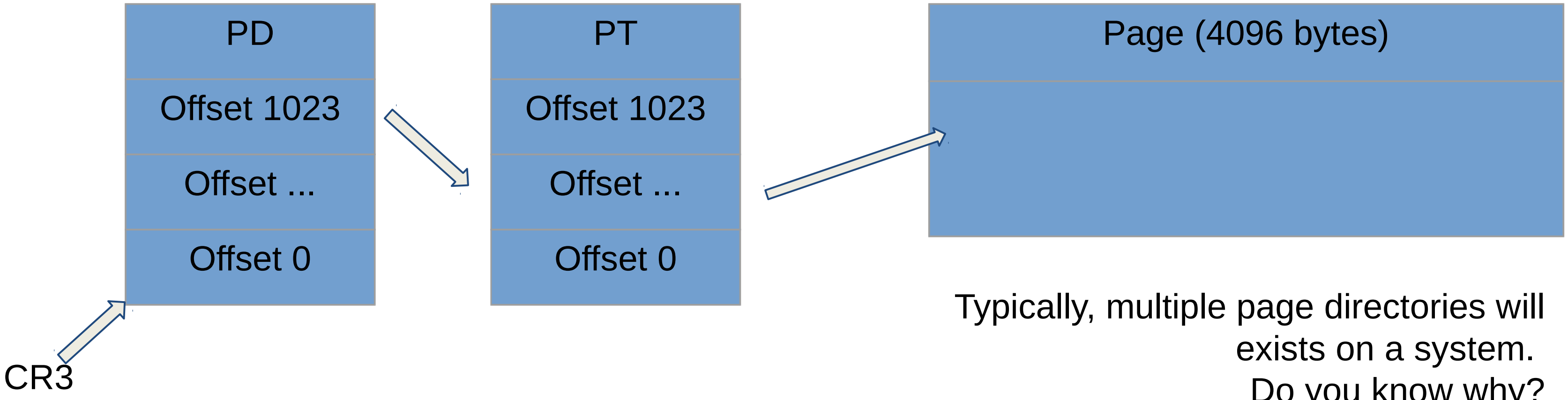
Be careful, Intel calls that “paging”. Not to be confused with paging/swapping.

Virtual memory



Virtual Address

31 - 22	21 - 12	11 - 0
Page Directory Offset	Page Table Offset	Page Offset



Typically, multiple page directories will exists on a system.
Do you know why?

On Linux

head_32.s

```
128 #endif
129
130         /* Create early pagetables. */
131         call  mk_early_pgtbl_32
132
```

```
290 /*
291  * Enable paging
292  */
293         movl $pa(initial_page_table), %eax
294         movl %eax,%cr3          /* set the page table pointer.. */
295         movl $CR0_STATE,%eax
296         movl %eax,%cr0          /* ..and set paging (PG) bit */
297         ljmp $__BOOT_CS,$1f     /* Clear prefetch and normalize %eip */
298 1:
299         /* Shift the stack pointer to a virtual address */
300         addl $__PAGE_OFFSET, %esp
```

Before line 296, the system is running using physical addresses. Once x86 paging is activated, the system is running on virtual address.
The jump and esp modifications are there to make sure pointers shows virtual addresses.