

## Lecture 5: March 16

*Lecturer: Alessandro Pellegrini**Scribe: Anzhelo Xhebraj*

In this lecture we will look into the organization and initialization of memory of the linux i386 kernel v. = 2.4.22.

## 5.1 Early Paging

As we anticipated in Lecture 3 (Sec 3.4) paging is enabled in `startup_32` through the following instructions

```
> arch/i386/kernel/head.S
    movl $swapper_pg_dir-__PAGE_OFFSET, %eax
    movl %eax, %cr3
    movl %cr0, %eax
    orl $0x80000000, %eax
    movl %eax, %cr0
```

```
ENTRY(swapper_pg_dir)
    .long 0x00102007 // pg0
    .long 0x00103007 // pg1
    .fill
    BOOT_USER_PGD_PTRS-2,4,0
    /* default: 766 entries */
    .long 0x00102007
    .long 0x00103007
    /* default: 254 entries */
    .fill
    BOOT_KERNEL_PGD_PTRS-2,4,0
```

where `swapper_pg_dir` is a label corresponding to the virtual address of the first level page table and `__PAGE_OFFSET` is `0xc0000000` (3GB, which is the virtual address of the kernel in i386). `%cr3` is not directly set to `swapper_pg_dir`, but the value shown above which is the physical address of `swapper_pg_dir` (the difference between virtual and physical addresses of the kernel is just an offset). In `startup_32` the kernel initializes its page tables to span only the first 8MB of the kernel. This is done by initializing two Page Tables (last level) found at label `pg0` and creating 4 entries in the first level page table (pointed by `swapper_pg_dir`): entry 0 and `0x300` (768) contain in the address field the physical address of `pg0` while entry 1 and `0x301` (769) contain the physical address of `pg1`. When paging is enabled, given the configuration of the page table we have that both the virtual addresses `0x0` to `0x007fffff` (from 0 to 8MB) and `0xc0000000` to `0xc07fffff` (from 3GB to 3GB + 8MB) will map to the physical addresses `0x0` to `0x007fffff`. The former is called *identity map* since maps the first 8MB of virtual addresses to the first 8MB of physical addresses while the latter is called *kernel map* since it maps the virtual addresses of the kernel to its physical addresses.

## 5.2 Bootmem Allocator

The transition from 8MB to 896MB of virtual memory is performed in the `start_kernel()` function in `init/main.c`. This function calls `setup_arch()` defined in `arch/i386/kernel/setup.c` (architecture dependent code) which initializes various data structures with among them the Bootmem allocator. The Bootmem allocator is a data structure and a set of functions that is used by the kernel to allocate memory (at the granularity of page sizes, 4KB) before the kernel Main Memory subsystem is setup. This set of APIs is available only at early setup of memory therefore it has `__init` in its signature.

The bootmem allocator initialization is performed in `setup_memory()` implemented in `arch/i386/kernel/setup.c` through the `init_bootmem()` function (`mm/bootmem.c`) initializing a bitmap where each bit refers to a page frame in the range of physical addresses

`_end` (after the last section of the decompressed kernel image) up to 896MB. First all the bits are set to 1 meaning that no page can be used for allocation. After that it is up to the function `register_bootmem_low_pages()` to query the E820 table and set the bits of free page frames to 0.

### 5.3 Paging in Linux

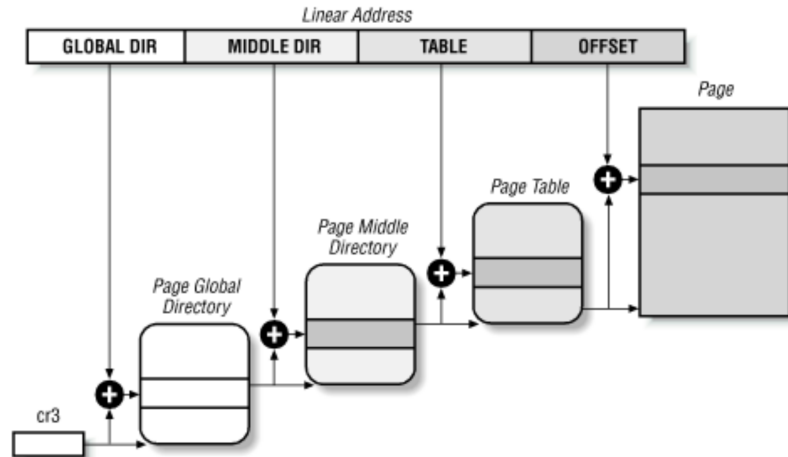
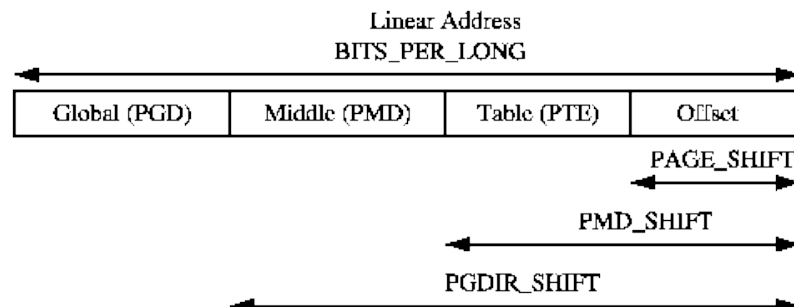


Figure 5.1: Linux Page Tables (< v.2.10)

Linux adopts a common paging model that fits both 32-bit and 64-bit architectures consisting of three-level paging up to kernel v.2.10 and four-level paging from kernel 2.11 introducing the Page Upper Directory as second level before Page Middle Directory (refer to Figure 5.1). Such scheme allows the kernel to be highly architecture independent reducing the amount of code needed to write for specific architectures. Various macros are then used to map the paging scheme that Linux uses to the hardware specific paging scheme.



$$x\_SIZE = 2^{x\_SHIFT}$$

Also  $\text{PTRS\_PER\_x}$  are defined to determine the number of entries in each level of the page table

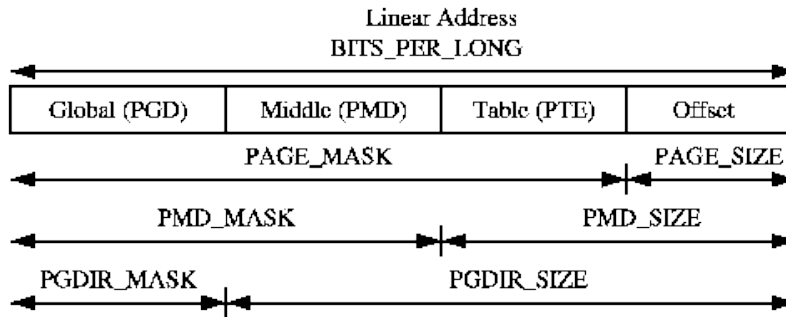


Figure 5.2: Macros for paging mapping

All the page tables entry types are defined through structs like `typedef struct {unsigned long pte_low; } pte_t` to ensure typechecking when manipulating table entries.

Various masks are defined in order to perform easy checks on page table entries such as `_PAGE_PRESENT` to be used as follows:

```
pte_t x;

x = ...;

if ((x.pte_low) & _PAGE_PRESENT) {
    /* executed if true */
}
```

Also multiple types of page entry flags are defined for the most common types of combinations of them such as `PAGE_SHARED`, `PAGE_KERNEL`, `PAGE_READONLY` etc.

## 5.4 Kernel Page Table Initialization

When carrying out the setup of architecture specific data structures in `setup_arch()`, also the transition from 8MB to 896MB is performed in `paging_init()` found in `arch/i386/mm/init.c` of which main subroutine is the following.

```
> arch/i386/mm/init.c
static void __init pagetable_init (void) {
    end = (unsigned long)__va(max_low_pfn*PAGE_SIZE);

    pgd_base = swapper_pg_dir;
    i = __pgd_offset(PAGE_OFFSET);
    pgd = pgd_base + i;

    for (; i < PTRS_PER_PGD; pgd++, i++) {
        vaddr = i*PGDIR_SIZE;
        if (end && (vaddr >= end)) break;
        pmd = (pmd_t *)pgd;
        ...
        for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
```

```

        vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
        if (end && (vaddr >= end))
            break;
        ...
        pte_base = pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
            if (end && (vaddr >= end)) break;
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
        }
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
        ...
    }
}... }

```

In the snippet shown above `end` is `0xf8000000` (128 MB short 4GB) and `i` is `0xc0000000` meaning that those are the virtual addresses used by the kernel. The routine uses the bootmem allocator to allocate Page Tables (last level) therefore the page tables might not be stored contiguously in memory. The function maps linearly the virtual addresses `0xc0000000` to `0xf8000000` to the first 896MB of page frames of physical memory.

Once the paging is setup we must ensure that no old entry is cached in the TLB. This is done in `paging_init()` by calling `load_cr3(swapper_pg_dir)` and `__flush_tlb_all()`.

```

> include/asm-i386/processor.h
    #define load_cr3(pgdir) \
        asm volatile( "movl %0, %%cr3": : "r" (__pa(pgdir)) );

```

As shown above, in `%cr3` is written `swapper_pg_dir`. This is done because the page tables used up to now might not be the ones in `swapper_pg_dir` (following the execution path from `arch/i386/kernel/head.S`) but the ones setup by the boot process which might have launched the kernel from a different entry point. Most of the `i386` architectures ensure that the TLB is flushed if the `%cr3` register is written but there can still be some architectures that need further instructions to ensure that the TLB is indeed flushed and in that case this is done through `__flush_tlb_all()`.

In newer versions of the kernel (v. 4.16) `load_cr3()` is reimplemented as follows:

```

> arch/x86/include/asm/special_insns.h
    ...
    static inline void native_write_cr3(unsigned long val) {
        asm volatile( "movl %0, %%cr3": : "r" (val), "m" (__force_order)) );
    }
    ...
    static inline void load_cr3(pgd_t *pgdir) {
        native_write_cr3(__pa(pgdir));
    }

```

The motive of `__force_order` is nicely explained in a comment within the file in which the functions are implemented.

```

> arch/x86/include/asm/special_insns.h

```

```

/*
 * Volatile isn't enough to prevent the compiler from reordering the
 * read/write functions for the control registers and messing everything up.
 * A memory clobber would solve the problem, but would prevent reordering of
 * all loads stores around it, which can hurt performance. Solution is to
 * use a variable and mimic reads and writes to it to enforce serialization
 */

```

### 5.4.1 TLB APIs

The types of TLB events can be classified across two main characteristics: scale and typology. When some event affects virtual addresses accessible by every CPU/core in real-time-concurrency its scale is said to be *global*, instead if it affects only virtual addresses accessible in time-sharing concurrency it is said to be *local*. The typology classification of events describes whether it is a virtual to physical address remapping or virtual address access rule modification.

Other considerations needed to be done when dealing with TLB flushing is its costs in terms of performance. Costs can be split into two sets: direct costs and indirect costs. The former are the latency of the firmware to perform the invalidation of the entries in the TLB plus the cost for cross-CPU coordination. The latter is about the costs of TLB renewal latency by the MMU firmware upon misses in the translation process.

The linux kernel provides various APIs for dealing with flushes of the TLB that are then mapped to architecture dependent instructions. While the APIs provide the possibility to perform selective flushing<sup>1</sup>, the real effect that it will have on the TLB depends on the instructions provided by the firmware. Nevertheless it is highly recommended to use the most specific API which is effective for the task and that doesn't make the software too complex. Follow the interfaces provided by the linux kernel nicely described under `Documentation/cachetlb.txt`.

`flush_tlb_all(void)` Flushes the entire TLB on *all* processors running in the system and it is usually invoked when the kernel page tables change since they are global by nature. Its implementation is based on a function that allows to execute a portion of code on all processors based on IPIs. Such portion of code is `__flush_tlb_all`.

`flush_tlb_mm(struct mm_struct *mm)` Flushes all the TLB entries related to a user address space. This is invoked usually when it is needed to invalidate all entries associated to a process for example when performing a `fork()` to make the address space not writable for COW (Copy on Write). On some architectures (MIPS) this is required for all cores instead of affecting only the local processor.

`flush_tlb_range(struct mm_struct *mm, unsigned long start, unsigned long end)` Similarly to the function above it is used to flush a range of (user) virtual addresses translations from the TLB. Primarily, this is used for `munmap()/mremap()` or `mprotect()`. The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized translations from the TLB instead of having the kernel call `flush_tlb_page` for each entry which may be modified.

`flush_tlb_page(struct vm_area_struct *vma, unsigned long page)` Flushes a single page from the TLB. Mainly used for flushing the TLB entry of some page after it

---

<sup>1</sup> flushing just a subset of the entries in the TLB.

has been paged out or faulted in meaning that the access to that page caused a fault (for example COW).

`flush_tlb_pgtables(struct mm_struct *mm, unsigned long start, unsigned long end)` Used when page tables of some software are being torn down. Some platforms cache the lowest level of the page tables in a linear virtually mapped array, to make TLB miss processing more efficient. In these cases the TLB needs to be flushed when parts of the page tables tree are unlinked/freed.

`update_mmu_cache(struct vm_area_struct *vma, unsigned long address, pte_t pte)` Used to inform the CPU that there exists a translation for the virtual address `address` corresponding to the entry `pte`. Such information can be used in many ways by the CPU such as deciding whether to flush its data cache or preload TLB translations.

## 5.5 NUMA

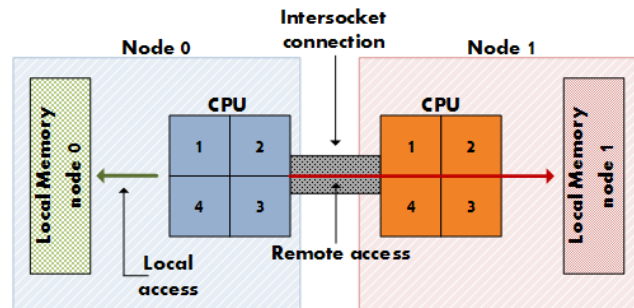


Figure 5.3: NUMA systems

With the ever growing disparity between the performance of processors and memory, memory accesses started to become a bottleneck in multi processor systems. This issue is addressed by arranging memory into banks and assigning each bank to one or a set of cores. We denote by the term *node* the set of cpus and banks coupled together. All cores can access all the memory but depending on their distance from the banks the cost to access memory is different. Such architectures are defined as Non-Uniform Memory Access (NUMA).

## References

- [lin(2004)] Linux i386 boot code howto, Jan 2004. URL <https://www.tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/>.
- [Bovet and Cesati(2006)] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. OReilly, 2006.
- [Gorman(2004)] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.