

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 9

Page Table

In the tutorial this week we will try to peek into the page table of a running Linux process. Since there is no pre-defined interface to get page table information of a process, we will write our own. This notes aims to explain the details of the Linux page table, to a depth that we can write system calls to extract entries from it.

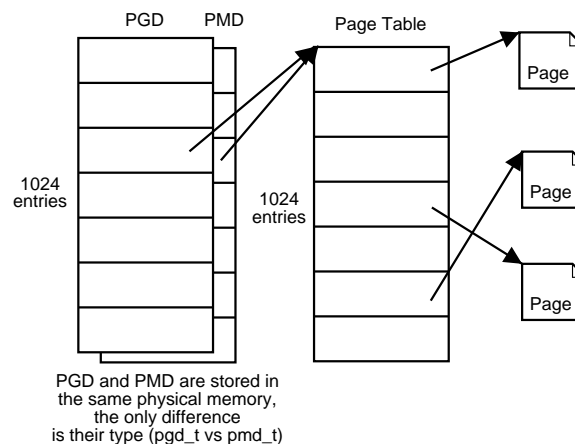
1. Page table levels and linear address

Conceptually, Linux uses 3-level page tables to convert virtual addresses¹ to physical frame addresses. Each process has a Page Global Directory (PGD), which entries are physical addresses of Page Middle Directory (PMD), each entry of which in turn stores physical addresses of Page Tables. A PGD, PMD and page table entry is of type *pgd_t*, *pmd_t* and *pte_t* respectively, defined in `include/asm/page.h`.

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
```

The number of bits of a virtual address used for indexing page tables are hard-coded into the kernel with “*#define*” (see `include/asm/pgtable-2level.h`). In particular, *PGDIR_SHIFT*, *PMD_SHIFT* and *PAGE_SHIFT* are set to the starting bit that is used for indexing the PGD, PMD and page table respectively. Using these as bit boundaries, the virtual address is broken into 4 parts (the last part is the offset within the frame).

Normally (i.e., when PAE² is not in use), 80x86 uses a 2-levels page table, so the page middle directory is not used. We have *PGDIR_SHIFT*=*PMD_SHIFT*=22 and *PAGE_SHIFT*=12, where the virtual address has 32 bits. Thus the top 10 bits (31–22) are for indexing the PGD, the next 10 bits (21–12) are for the page table, and the last 12 bits are for offset within the frame. The PMD completely embeds in the PGD entry, i.e., a *pmd_t* entry actually uses the same memory as the *pgd_t* entry. The following diagram shows their relations.



¹As Linux does not use segmentation, virtual address is the same as logical address.

²Starting from Pentium-Pro, 80x86 support a paging mode called Physical Address Extension (PAE), where physical address is 36-bits (i.e., at most 64GiB physical memory). Each page table entry is thus enlarged to 64 bits. A page frame can hold only 512 page entries, so a 3-level page directory is needed.

The content of the page table and PGD is supposed to be the physical address of the frame holding the page or the next level page table. Note that frame addresses always has the last 12 bits to be 0. It is wasteful to store those 12 bits of known 0s, so the hardware designers decide to use those 12 bits to keep information about the page: whether the page exists (“present”), is writable, can be used only by the kernel, is modified, is accessed, etc.

2. Finding a PGD

Every process has a PGD, which is always in memory. Recall that given the *pid* of a process, we can find its *task_struct* structure by *find_task_by_pid()*. The structure contains a pointer *mm* to the “memory descriptor” of the process, of type **struct mm_struct** *:

```
struct task_struct {  
    ...  
    struct mm_struct *mm;  
    ...  
};
```

A memory descriptor stores all information of the process related to memory allocation. The structure is defined in `<linux/sched.h>`. The starting address of the PGD is stored in it:

```
struct mm_struct {  
    ...  
    pgd_t *pgd; // virtual address of page global directory  
    ...  
};
```

During a context switch, *mm->pgd - 0xc0000000* is loaded into `CR3`, the hardware register used for paging. It might not be immediately clear why we need to subtract the magic number *0xc0000000* from *pgd*. Recall that the first 1GiB of the physical memory installed (in most cases, this means all memory of the computer) is mapped to the virtual address starting from *0xc0000000* to allow the kernel to easily access them. The page table is marked to be kernel-only, i.e., they cannot be used in user-mode. Since the *pgd* field stores the virtual address, *0xc0000000* needs to be subtracted from the linear address to get the physical address. This is done by a macro called `__pa()`, like `__pa(mm->pgd)`. There is a macro called `__va()` which does the reverse, adding *0xc0000000* to a physical address to get a (kernel-only) virtual address.

3. Reading a page table for a given linear address

Suppose we are given a process and a linear address, and want to know what is the address of the corresponding physical memory. It requires five steps:

1. Find the *mm_struct* pointer of the process.
2. Find the corresponding *pgd_t* (PGD entry). Check that the PMD is in memory.
3. Find the corresponding *pmd_t* (PMD entry). Check that the PTE is in memory.
4. Find the corresponding *pte_t* (PTE entry). Check that the page is in memory.
5. Get the address of the page.

To do these, we need to know whether a level of the page table is not used by the platform (e.g.,

the PMD is not used in 2-level mode). We also need to know which bit reveals whether a page is present. Even if we know all these information, we need to do rather complicated bit-manipulations to do those tasks. Luckily, kernel developers don't like such platform dependence and such complicated bit-manipulations either, so they write macros and functions to do all those tasks.

4. Using a page directory

The functions and macros are called *XXX_offset()*, *XXX_present* and *XXX_page*, where *XXX* is one of *pgd*, *pmd* and *pte*. We will describe them one by one. These kernel function is designed in a way that you need only to have variables holding the virtual address of page table **entries**, never the page table themselves.

1. The simplest of these functions are those checking whether the next level page table is present. You can imagine them to be functions of prototype **int** *pmd_present(pmd_t pmd)* (given the content of a PMD entry, check whether the page table is present), etc. The implementation is also simple:

```
static inline int pgd_present(pgd_t pgd) { return 1; }
#define pmd_present(x) (pmd_val(x) & _PAGE_PRESENT)
#define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
```

2. The *XXX_page* functions are also easy to understand. They return the virtual address of the next level page table (i.e., extract the bits corresponding to the content of an entry, and turn them to virtual address). You can imagine them to be functions of prototype **int** *pmd_page(pmd_t)* (given the content of a PMD entry, find the virtual address of the next level page table), etc. The implementation is like:

```
#define pgd_page(pgd) ((unsigned long) _va(pgd_val(pgd) & PAGE_MASK))
#define pmd_page(pmd) ((unsigned long) _va(pmd_val(pmd) & PAGE_MASK))
#define pte_page(x) (mem_map + ((unsigned long)((x).pte_low >> PAGE_SHIFT)))
```

The *pte_page()* function is special. It doesn't return you the address of the frame. Instead, it returns a pointer to the **struct** *page* structure that holds information about the page, e.g., usage count. The virtual address of the page can be found using *page_address(pte_page(x))*.

3. Given a page table entry of the level just above *XXX* and a linear address, *XXX_offset()* find the corresponding page table entry of the level *XXX*. E.g., if you have a memory descriptor *mm* and a virtual address *laddr*, you get a pointer to the PGD entry for *laddr* by *pgd_offset(mm, laddr)*. For all purpose, you can treat it as a function with the prototype *pgd_t *pgd_offset(struct mm_struct *mm, unsigned long address)*.

The implementation is different for all the three levels. They are similar to the following:

```
#define pgd_offset(mm, address) \
((mm)->pgd + ((address) >> PGDIR_SHIFT))
extern inline pmd_t *pmd_offset(pgd_t *dir, unsigned long address) {
    return (pmd_t *) dir;
}
#define pte_offset(dir, address) ((pte_t *) pmd_page(*(dir)) + \
((address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1)))
```