

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 7

Page Table

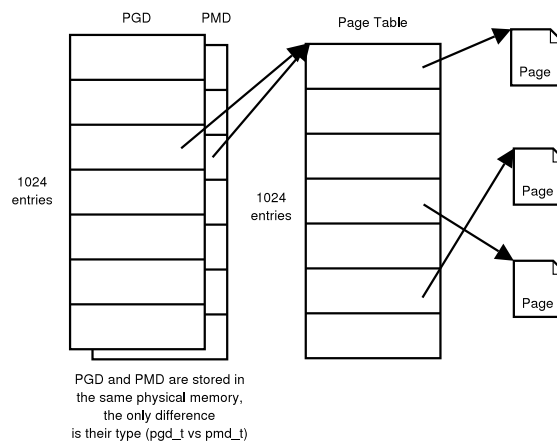
In the tutorial this week we will peek into the page table of a running Linux process. Since there is no pre-defined interface to get the page table of a process, we will write our own. This notes aims to explain the details of the Linux page table, to a depth that we can write system calls to extract them. All functions and macros mentioned here are available with `#include <linux/mm.h>`. To see the actual definitions, try `<asm/page.h>`, `<asm/pgtable.h>` and `<asm/pgtable-2level.h>`.

1. Page table levels and linear address

Conceptually, Linux uses 3-level page tables to convert virtual addresses 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.

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

Therefore, each virtual address is conceptually broken into 4 parts: an index to the PGD, an index to the PMD, an index to the page table, and the page offset within the actual memory frame. However, normally (i.e., when PAE¹ is not in use), 80x86 uses a 2-levels page table, so there is no “real” PMD. Instead, the PMD uses the same memory as the PGD. In the virtual address, 10 bits (bit 31–22) are index to this combined PGD/PMD; 10 bits are index to the page table; and 12 bits are page offset. Each page thus has `PAGE_SIZE = 4096` bytes. The following diagram summaries their relations.



The primary purpose of the page table and PGD is to **store the physical addresses of the frames** holding the pages or the next level page tables. Since frame addresses always have the last 12 bits to be 0, they need not be stored. Instead, the hardware designers decide to use them to keep information about the page. Here are the more interesting bits.

- Bit 0 (`_PAGE_PRESENT = 1`): whether the page table entry refers to a page that is **present**.

¹Starting 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. This mode is normally not activated.

If cleared, accessing the page results in a page fault. At such times, the OS is free to use the remaining bits of the page table entry to store any other information, e.g., where in the swap space hold the page.

- Bit 1 (`_PAGE_RW = 2`): whether the page is **writable**. If cleared, writing results in a page fault.
- Bit 2 (`_PAGE_USER = 4`): whether the page can be accessed from the **user mode**. If cleared, accessing the page from the user mode results in a page fault (and thus the page is kernel-only).
- Bit 5 (`_PAGE_ACCESSED = 32`): whether the page is **accessed** after the last time the bit is cleared. Everytime the CPU access a page, this bit is set to 1. The bit is used by the OS to implement certain swapping algorithms (like the 2nd chance algorithm).
- Bit 6 (`_PAGE_DIRTY = 64`): whether the page is **modified** after the last time the bit is cleared. Everytime the CPU writes a frame through a page, this bit is set to 1. The OS uses this information to implement write-back memory maps, and also to avoid writing clean pages to swap.
- Bit 7 (`_PAGE_PSE = 128`): whether the page is a **large page**. All Pentium's allow a page to be large, i.e., of 4MiB instead of 4KiB. The bit is meaningful only in the PGD. If set, there is no page table for the PGD entry, and the PGD directly points to physical frames. All the 22 bits remaining in the virtual address after the 10 bit PGD index are used to offset this large page.

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 {                               /* in <sched.h> */
    ...
    struct mm_struct *mm;
    ...
};
```

A memory descriptor stores all information of the process related to memory allocation. The address of the PGD is stored in it:

```
struct mm_struct {                                 /* in <sched.h> */
    ...
    pgd_t *pgd; // virtual address of page global directory
    ...
};
```

3. Accessing physical memory via the virtual memory system

Before we go on, it is important to understand the difference between the view of the hardware designers and that of the operating system designers on the addresses. The **hardware** uses the

physical addresses of the frames storing the various levels of page tables (PGD, PMD and page tables), before the hardware can do address translations. **Processes**, then, only access memory via **virtual** addresses. But what about the **kernel**? On one hand, it must be able to control the hardware and thus manipulate physical addresses. (E.g., the page tables and the CPU register holding the PGD frame address must be physical addresses). On the other hand, it must also be able to access memory of the users-provided buffers, which are specified as virtual addresses.

The answer is that **the kernel**, like processes, **uses virtual addresses**. It can thus access the memory of the user process easily. However, part of the virtual addresses are strongly related to physical addresses, to make it easy for the kernel to access them. In particular, the virtual addresses starting from `0xc0000000` are always mapped to the physical addresses starting from 0. For example, the kernel can read the memory at physical address `0x123bc` by reading the virtual memory address `0xc00123bc`. The macro `__pa()` will take a virtual address and subtract `0xc0000000` to get the corresponding physical address; and the macro `__va()` does the reverse. Of course, only the kernel (not the processes) can access these addresses, as marked in the page table entries for these addresses. And if you still remember, the `access_ok()` function we learnt in the last tutorial checks whether an address range has any portion in this range.

The kernel stores most addresses in virtual address form, to allow the kernel to easily access the memory. This is except the few locations, like the content of the page tables, where the hardware actually access the memory and thus must hold physical addresses. The two macros above are used to do the translations as needed. For example the `pgd` in the `mm_struct` is a virtual address (with value larger than `0xc0000000`). When the kernel performs a context switch, this value is fetched, `__pa()` is used to convert it to a physical address, and the result is put to the `cr3` register that is reserved for holding the physical address of the page global directory.

4. Reading a page table for a given linear address

Sometimes the kernel must access the virtual memory of a process when `cr3` is not storing the PGD of that process. At such time, the kernel must simulate the effect of the CPU in address translations. It requires several steps, each requiring a fair amount of bit manipulations, which are rather system dependent. Thus the kernel developers have written functions and macros to ease the process.

1. **Find the right PGD entry:** The first step to access memory is always to have a pointer, say `task`, pointing to the `task_struct` of the needed process. Knowing the virtual address `laddr`, finding the correct PGD entry can be done by calling `pgd_offset(task->mm, laddr)`, which finds the bits in `laddr` corresponding to the PGD and return a pointer to the needed `pgd_t`. It is implemented like this.

```
#define pgd_offset(mm, address) ((mm)->pgd + (((address) >> 22) & 1023))
```

2. **Check whether we can find a PMD:** Once we get to the PGD, we want to read its bits to see whether the PMD is present. This is done by calling `pgd_present(*pgd)`, where `pgd` points to the PGD entry. However, since PMD actually is the same as the PGD in a 2-level page table, it is always present. The implementation is thus trivial.

```
static inline int pgd_present(pgd_t pgd) { return 1; }
```

3. **Find the right PMD entry:** Once you verified a PGD entry, you can use the PMD bits of the virtual address `laddr` to index the PMD. This is done by calling `pmd_offset(pgd, laddr)`. Since PMD is merged into the PGD, this is trivially done by a simple casting to `pmd_t *`.

```
extern inline pmd_t * pmd_offset(pgd_t * dir, unsigned long address)
{ return (pmd_t *) dir; }
```

4. **Check whether we can find a page table:** Once we get to the PMD, we need to read its bits to see whether the page table is present. This is done by calling `pmd_present(*pmd)`, where `pmd` points to the PMD entry. This is implemented like this:

```
#define pmd_present(x) (x.pmd & _PAGE_PRESENT)
```

5. **Find the right Page table entry:** Once you verified a PMD entry, you can use the page table bits of the virtual address `laddr` to index the page table. This is done by calling `pte_offset(pmd, laddr)`, giving you a pointer to the corresponding `pte`. This is implemented as follow.

```
#define pte_offset(dir, address) \
((pte_t *) pmd_page_kernel(*(dir)) + ((address >> 12) & 1023))
```

The `pmd_page_kernel` function finds the frame storing the page table, and will be discussed shortly.

6. **Check whether we can find a frame:** Once we get to the page table, we need to read its bits to see whether the frame is present. This is done by calling `pte_present(*pte)`, where `pte` points to the page table entry. This is implemented like this:

```
#define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
```

Using these functions and macros, we can traverse from the pointer to `mm` structure down to the actual frame step by step. Note that each step you get an **page table entry**, not a **frame** that actually store the page tables or frames. To obtain them, three more macros can be used.

```
#define pgd_page(pgd) ((unsigned long) __va(pgd.pgd & ~4095))
#define pmd_page_kernel(pmd) ((unsigned long) __va(pmd.pmd & ~4095))
#define pte_page(x) (mem_map + ((unsigned long)((x).pte_low >> 12)))
```

All these macros take an actual page table entry (not a pointer to it) as argument. The first two macros, `pgd_page()` and `pmd_page_kernel()`, give you the virtual address of a PMD and a page table. They simply remove the last 12 bits from the page table entry to get the physical address, and use `__va()` to get the corresponding virtual address.

On the other hand, `pte_page()` is quite different. It shifts the page table entry right by 12 bits, thus getting a number called the **frame number** of the needed physical frame pointed to by the page table entry. The **frame table** is then indexed to find the entry correspond to that frame. The return value is actually a pointer `mmap_ptr` to this entry. The entry stores information about the frame, e.g., usage count. The virtual address of the page can be found using `page_address(mmap_ptr)`.