CSIS0230A Principle of Operating Systems (Class A)
Notes for Tutorial 6
Page Table

In this tutorial, you will learn how to get page table information of a process.

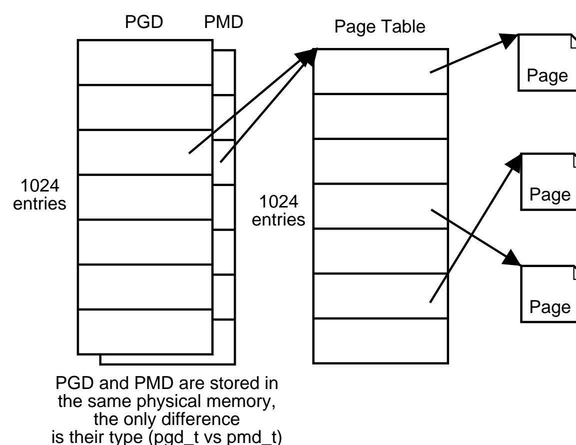
1. Page table levels and linear address

In order to support as many architectures as possible, the page directory of Linux is 3-level. Each process has a Page Global Directory (PGD), which entries are physical addresses of Page Middle Directory (PMD), each 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 unsigned long *pte_t*;
typedef unsigned long *pmd_t*;
typedef unsigned long *pgd_t*;

The number of bits of a logical address used for indexing these tables are hard-coded into the kernel with "#*define*" in the header files as constants. 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. By these boundaries, the logical address is broken into 4 parts (the last part is the offset within the page).

Normally (i.e., when PAE[1] is not in use), 80x86 (i386) uses a 2-levels page table. A linear address contains 32 bits, while we have *PGDIR_SHIFT*=*PMD_SHIFT*=22 and *PAGE_SHIFT*=12. 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 the page table. Note that *PGDIR_SHIFT*=*PMD_SHIFT*, meaning that the PMD has no bit. The PMD completely embeds in the PGD entry. Thus a **pmd_t** entry actually uses the same memory as the **pgd_t** entry. The following diagram shows their relations.



2. Finding a PGD

Every process has a PGD, which is always in memory. As we have learnt in last tutorial, if we have a process ID pid, we can find its *task_struct* structure by *find_task_by_pid*(). The structure

[1] Starting from Pentium-Pro, 80x86 support a mode of operation called Physical Address Extension (PAE), where physical address is 36-bits. This allows physical memory as large as 64GB. However, the logical address is still 32-bits, so each process sees no more than 4GB memory. The net result is that each page table entry must be at least 36-bits (actually, 64-bits are allocated). Thus a page can hold only 512 page entries, and a 3-level page directory is used.

contains a pointer *mm* to the "memory descriptor" of the process, of type **struct** *mm_struct* ∗:

```
struct task_struct {
        pid_t pid;
        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*>, and contains a field *pgd*, which is the starting address of the PGD:

```
struct mm_struct {
        ...
        pgd_t * pgd;
        ...
};
```

3. Using a page directory

Once you get a page global directory, you can use functions and macros defined in <**asm**/*page.h*> and <**asm**/*pgtable.h*> to access them. This involves three tasks: (1) given an upper-level page table entry and a logical address, find the page table entry that is used for the logical address; (2) given a page table entry, check whether the entry points to a valid page in memory; and (3) given a page table entry, find the address of the lower-level page table. These are done by the *XXX_offset*, *XXX_present* and *XXX_page* macros respectively, where *XXX* is one of *pgd*, *pmd* and *pte*.

1. The *XXX_offset*() macro. For example, if you have a memory descriptor *mm* and a logical address *laddr*, you can get a pointer to the PGD entry for finding *laddr* by simply *pgd_offset*(*mm*, *laddr*). It is defined like this:

   ```
   #define pgd_offset(mm, address) \
   ((mm)->pgd + ((address) >> PGDIR_SHIFT))
   ```

   So it does exactly what is suggested by the name of the macro: to go to the PGD offset for some address, given the memory descriptor. For all purpose, you can treat it as a function with the prototype *pgd_t* ∗ *pgd_offset*(**struct** *mm_struct* ∗ *mm*, **unsigned long** *address*).

   Similarly, if you have a pointer *pgd* to a PGD entry and want to get the PMD entry of an address *addr*, you can use *pmd_offset*(*pgd*, *laddr*). Recall that the PMD is embedded into the PGD, so this does nothing except to turn the *pgd_t* pointer to a *pmd_t* pointer. It is defined like this:

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

2. Each page table entry is a 32-bit long-type value (of type **unsigned long**). They are used for storing the starting physical address of pages. However, since pages always have bottom 12 bits 0, the page table entry can use these bits for other purpose. One particular bit is the present bit, which is set to 1 if the page table entry is actually valid. The *XXX_present* macro can be used to check this bit. For example, if you have a PMD entry *Pmd*, then

pmd_present(*Pmd*) returns non-zero if the page table is present. It is defined like this:

#define pmd_present(*x*) (*pmd_val*(*x*) & *_PAGE_PRESENT*)

We can regard it as a function: **int** *pmd_present* (*pmd_t pmd*).

3. Sometimes we want to simply dereference a page table entry rather than to offset it using a logical address. This can be done by *XXX_page*. For example, if you have a PMD entry *Pmd*, you can get the starting address of the page it points to, by *pmd_page*(*Pmd*). You can treat it as a function of prototype **unsigned long** *pmd_page*(*pmd_t pmd*). Note that the result is an unsigned long integer, and you have to cast it to a pointer if you want to dereference it. It is implemented like this:

   #define pmd_page(*pmd*) \
   ((**unsigned long**) *__va*(*pmd_val*(*pmd*) & *PAGE_MASK*))

4. Reading a page table for a given linear address

If we are given a process and one of its linear address, it requires four steps to find the page table that contains the address.

a. Find the entry in PGD table that is corresponding to the linear address.

b. Convert the PGD entry to a PMD entry.

c. Test the existence of the page table pointed by PMD entry.

d. If the page table is in memory, get its base address and read its content.

5. Modules

Please read the module part in the reading material of the last tutorial, and the sample module provided in course homepage.