简单来说，讨论linux页表就是讨论linux进程的的页表：linux页表的创建与更新都包含于进程的创建与更新中。当前的linux内核采用的是写时复制方法，在创建一个linux进程时，完全复制父进程的页表，并且将父子进程的页表均置为写保护（即写地址的时候会产生缺页异常等）。那么父子进程谁 向地址空间写数据时，产生缺页异常，分配新的页，并将两个页均置为可写，按照这种方式父子进程的地址空间渐渐变得不同。

    按照上面的分析, 只需要讨论第一个进程页表初始化，进程创建时页表的拷贝，以及缺页异常时页表的更新即可。

1.init\_task进程页表的初始化

    init\_task的地址空间是init\_mm, init\_mm在内核初始化的时候就赋值给了current->active\_mm. init\_mm的初始化页表是swapper\_pg\_dir，在mips架构中swapper\_pg\_dir初始化在函数pagetable\_init中，初始化关系是

swapper\_pg\_dir -> invalide\_pmd\_table -> invalide\_pte\_table 或

swapper\_pg\_dir -> invalide\_pte\_table.

即在init\_mm中，页表指向的全部是invalide\_pte\_table。

2.创建进程时页表的拷贝

    进程创建一般调用的是do\_fork函数，按照如下调用关系：

    do\_fork->copy\_process->copy\_mm->dup\_mm->dup\_mmap->copy\_page\_range

    找到copy\_page\_range函数，这个函数便是负责页表的拷贝，函数核心代码如下：

 874     do {

 875         next = pgd\_addr\_end(addr, end);

 876         if (pgd\_none\_or\_clear\_bad(src\_pgd))

 877             continue;

 878         if (unlikely(copy\_pud\_range(dst\_mm, src\_mm, dst\_pgd, src\_pgd,

 879                         vma, addr, next))) {

 880             ret = -ENOMEM;

 881             break;

 882         }

 883     } while (dst\_pgd++, src\_pgd++, addr = next, addr != end);

     copy\_pud\_range便是拷贝pud表，copy\_pud\_range调用copy\_pmd\_range, copy\_pmd\_range调用copy\_pte\_range，以此完成对三级页表的复制。需要注意的是在copy\_pte\_range调用的copy\_one\_pte中有如下代码：

 694     if (is\_cow\_mapping(vm\_flags)) {

 695         ptep\_set\_wrprotect(src\_mm, addr, src\_pte);

 696         pte = pte\_wrprotect(pte);

 697     }

这里便是判断如果采用的是写时复制，便将父子页均置为写保护，即会产生如下所示的缺页异常。

3.缺页异常时页表的更新

  由页表的初始化可以看到，init\_mm的页表全指向无效页表，然而普通的进程中不可能页表均指向无效项，因此肯定拥有一个不断扩充页表的机制，这个机制是通过缺页异常实现的。

 缺页异常最终会调用do\_page\_fault，do\_page\_fault调用handle\_mm\_fault，handle\_mm\_fault是公共代码，一般所有的缺页异常均会调用handle\_mm\_fault的核心代码如下：

3217     pud = pud\_alloc(mm, pgd, address);

3218     if (!pud)

3219         return VM\_FAULT\_OOM;

3220     pmd = pmd\_alloc(mm, pud, address);

3221     if (!pmd)

3222         return VM\_FAULT\_OOM;

3223     pte = pte\_alloc\_map(mm, pmd, address);

3224     if (!pte)

3225         return VM\_FAULT\_OOM;

其中pud\_alloc代码如下：

1056 static inline pud\_t \*pud\_alloc(struct mm\_struct \*mm, pgd\_t \*pgd, unsigned long address)

1057 {

1058     return (unlikely(pgd\_none(\*pgd)) && \_\_pud\_alloc(mm, pgd, address))?

1059         NULL: pud\_offset(pgd, address);

1060 }

其中pgd\_none用于判断pgd是否为invalide,如果是可调用\_\_pud\_alloc,如果不是获得其地址继续查。

pmd\_alloc函数和pte\_alloc\_map函数类似。

因此可以看出，在缺页异常中，会按照地址一次查三张页表，如果页表为invalide，比如invalide\_pmd\_table或invalide\_pte\_table，则会分配一个新的页表项取代invalide的页表项。这便是页表扩充的机制。

需要注意的是handle\_mm\_fault最终会调用handle\_pte\_fault，在handle\_pte\_fault函数中有如下代码：

3171     if (flags & FAULT\_FLAG\_WRITE) {

3172         if (!pte\_write(entry))

3173             return do\_wp\_page(mm, vma, address,

3174                     pte, pmd, ptl, entry);

3175         entry = pte\_mkdirty(entry);

3176     }

即在缺页异常中如果遇到写保护会调用do\_wp\_page，这里面会处理上面所说的写时复制中父子进程区分的问题。

如上三个部分便是linux页表的大体处理框架