@TOC

程序地址空间

- 堆是堆，栈是栈，堆栈等于栈
- 程序地址空间--->不是内存--->是**进程虚拟地址空间**

我们来看一个现象：

```
[xifeng@VM-16-14-centos lesson10]$ ./test2
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am child ,time=0,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am child ,time=1,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am child ,time=2,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
################child更改变量##################
#############child修改变量完成#################
I am child ,time=3,g_val=200,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am child ,time=4,g_val=200,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
I am father ,g_val=100,&g_val=0x7ffca29e93c0
^C
[xifeng@VM-16-14-centos lesson10]$
```

首先我们知道，改变子进程的值不会影响父进程的值(因为有写实拷贝)，但是我们发现一个现象：**父子进程的g_val的值是不同的，可是它们的地址却是相同的----->如果打印出来的地址是物理内存的地址这一现象是不可能存在的，所以这个地址不会是物理地址，这个地址是虚拟地址**

进程地址空间概念

- 每一个进程都有一个地址空间，都认为自己在独占物理内存
- 在内核中,**地址空间就是一个数据类型**,可以用其来定义具体**进程的地址空间变量**

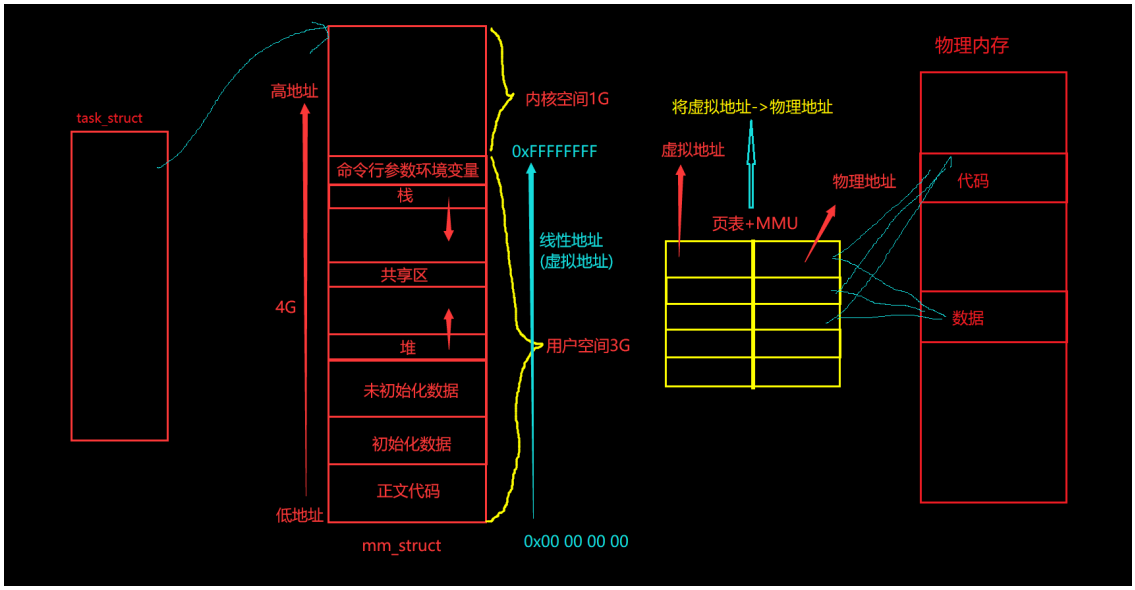
```
struct mm_struct{
//进程地址空间(32位-->4G)
    unsigned int code_start;
    unsigned int code_end;

    unsigned int init_data_start;
    unsigned int init_data_start;

    unsigned int uninit_data_start;
    unsigned int uninit_data_end;

    unsigned int heap_start;
    unsigned int heap_end;
    //......
    unsigned int stack_start;
    unsigned int stack_end;
}//虽然这里只有start和end但是每个进程都认为地址空间的划分是按照4GB空间划分的，换而言之，每个进程都认为自己拥有这4GB
```

虚拟空间划分之后，进程还是需要运行代码读取数据所以就需要**页表和MMU(MMU是硬件设备，一般集成在CPU中)**

- 虚拟地址(在Linux上等价于线性地址)：地址空间上进行区域划分时对应的线性位置(举个例子就相当于code_start到code_end这段区间都是虚拟地址)
- 区域的划分简单理解：就是有起始和终止的范围，举个例子，小时候经常听到的38线男孩和女孩通过尺子划分了两块区域，当男孩想要放置东西的时候他会对应着尺子上的范围去桌子上放置东西，但是事实上这个尺子不是一直存在的，区域划分也是类似道理，尺子就相当于是一个含有各个区域的起止和结束位置的结构体，可是事实上是没有这个结构体的，之前说的堆区栈区.....这些就是一系列的区域。

页表

- MMU(内存管理单元)：作用相当于查页表
- 页表 ： 简单理解就是这个表的左边是虚拟地址，右边是物理地址



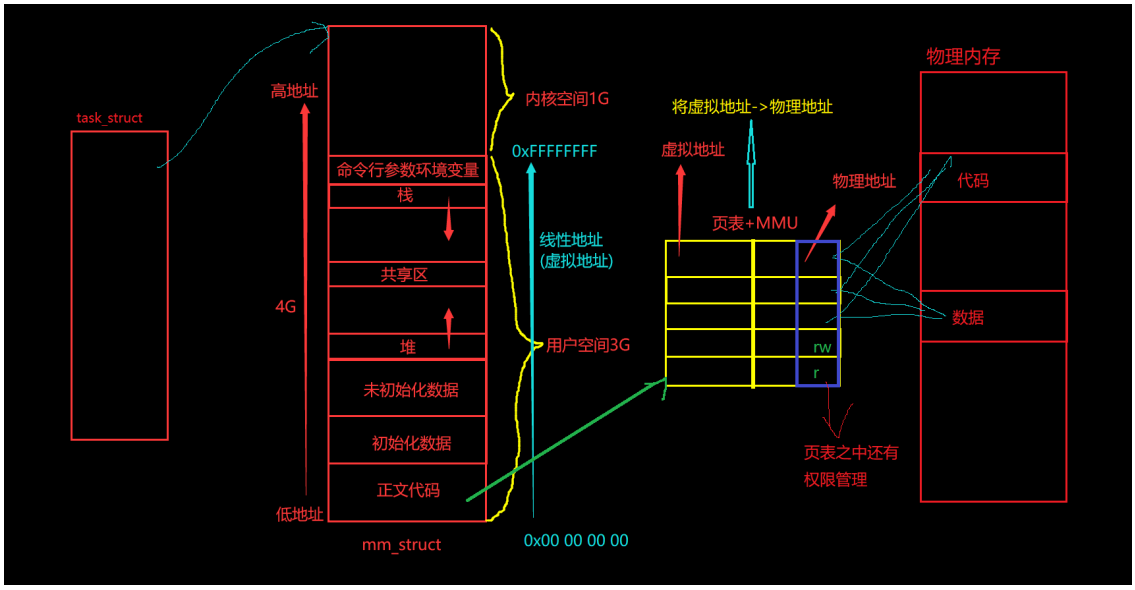
  - 核心工作：将虚拟地址转换成物理地址
  - 页表本质上就是一个映射表(哈希映射)
  - 用户在寻址时，使用的是虚拟地址，由操作系统经过页表的映射找到物理地址，进而再去访问代码和数据

为什么要有地址空间?

1. 通过添加一层**软件层**，完成有效的对进程操作内存进行**风险管理(权限管理)**，本质目的是为了**保护物理内存以及各个进程的数据安全**

   来举一个例子：

```
const char* str = "hello word!";
*str = 'H';
//我们都知道这样的方式是不被允许的，在语言层面上我们知道"hello word"是在字符常量区，所以不能被修改，可原理究竟是什么呢？
//本质是因为OS在这个区域给我们的权限只有r的权限，我设法修改的操作是不被允许的，所以会直接崩溃这个进程
```



2. **将内存申请和内存使用的概念在时间上划分清楚**，通过虚拟地址空间，来屏蔽底层申请内存的过程，达到进程读写内存和OS进行内存管理操作，进行软件上面的分离(**基于缺页中断进行物理内存申请**)

   举个例子：

   1. 我们申请了10000个字节，我们立马会使用这10000个字节吗？？--->不一定，可能会存在暂时不会全部使用，甚至暂时不使用。---->(这段话就是写时拷贝，也就是需要的时候再去给你空间)在OS角度，如果空间立马给你，就有可能意味着整个系统会有一部分空间，本来可以立马给别人用的，现在却被你闲置着，这样是不合理的(怎么判断没有被使用呢？有了空间但是从来没有过读写)------->OS的做法是：先给你在地址空间上划分一块区域，在你真的要用到的时候去物理内存去申请空间，没用到之前页表没有一个完整的映射
   2. 有了地址空间之后，就算是物理内存已经满了，也可以申请到空间，比如要运行A进程，但是物理内存已经满了，OS就会判断比如B进程所占的空间长时间没有使用，就会通过**内存管理算法**将这一部分的空间置换到磁盘上，从而让A进程申请空间成功
3. 站在CPU和应用层的角度，进程可以看做统一使用4GB空间，而且每个空间区域的相对位置，是比较确定的(**也就是程序的代码和数据可以被加载到物理内存的任意位置--->大大的减少内存管理的负担**)