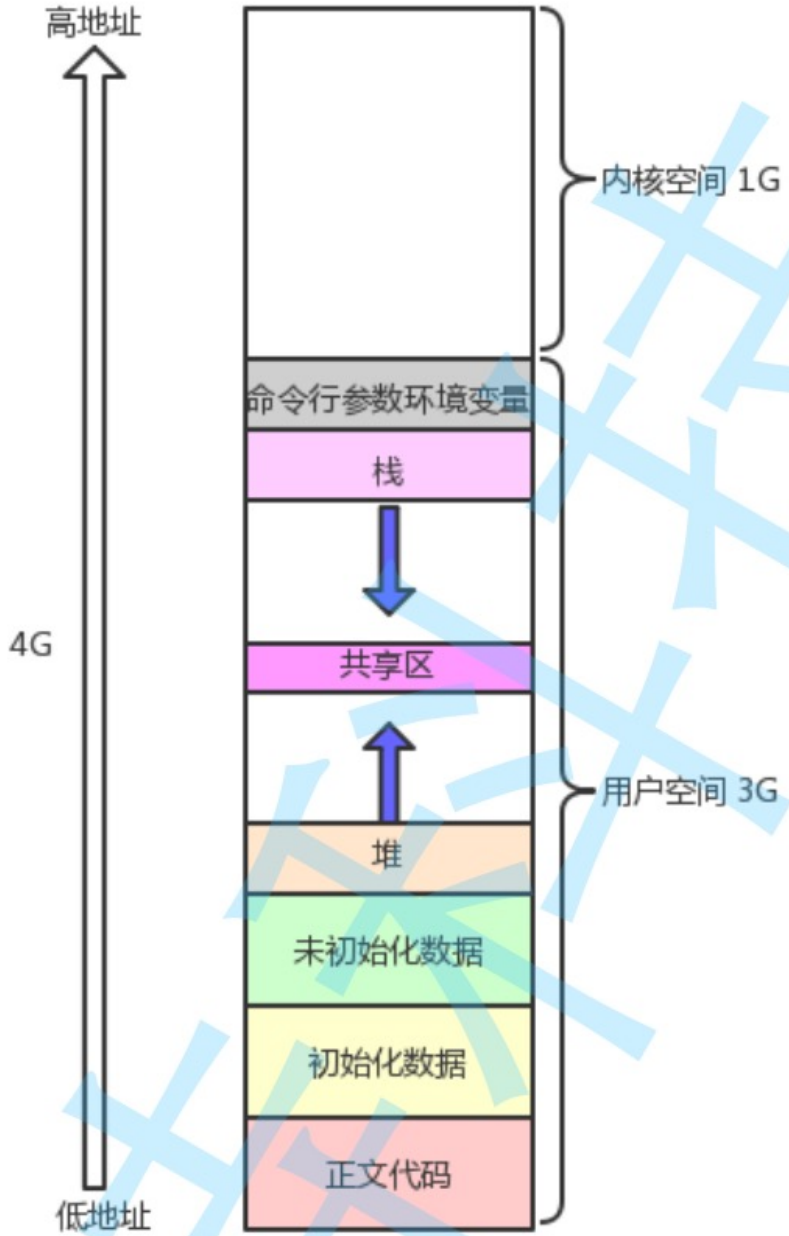
0909 地址空间

1. 什么是地址空间？
2. 地址空间是如何设计的？
3. 为什么要有地址空间？



我们用代码去证明一下这个结构！

打印各种地址 – 本质是进程打印！

```
1 hello:hello.c
2     gcc -o $@ $^
3 .PHONY:clean
4 clean:
5     rm -f hello
```

以后我们写Makefile都这样写

$@ 表示hello
$^ 表示：右边所有的文件

```
~
~
hello
```

```c
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

int g_unval;//未初始化全局变量
int g_val =  100;//初始化全局变量

int main(int argc, char* argv[], char* env[])
{
    printf("code addr: %p\n",main);
    printf("init global addr: %p\n",&g_val);
    printf("uninit global addr: %p\n",&g_unval);

    char *heap_mem  = (char*)malloc(10);//堆上的空间
    printf("heap addr: %p\n",heap_mem);

    printf("stack addr: %p\n",&heap_mem);//栈区的地址

    for(int i = 0;i < argc; i++)
    {
        printf("argv[%d]: %p\n",i,argv[i]);
    }
    for(int i = 0; env[i] ; i++)
    {
        printf("env[%d]: %p\n",i,env[i]);
    }
    return 0;
}
```

```c
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

int g_unval;//未初始化全局变量
int g_val =  100;//初始化全局变量

int main(int argc, char* argv[], char* env[])
{
    printf("code addr: %p\n",main);
    printf("init global addr: %p\n",&g_val);
    printf("uninit global addr: %p\n",&g_unval);

    char *heap_mem  = (char*)malloc(10);//堆上的空间
    printf("heap addr: %p\n",heap_mem);

    printf("stack addr: %p\n",&heap_mem);//栈区的地址

    for(int i = 0;i < argc; i++)
    {
        printf("argv[%d]: %p\n",i,argv[i]);
    }
    for(int i = 0; env[i] ; i++)
    {
        printf("env[%d]: %p\n",i,env[i]);
    }
    return 0;
}
```

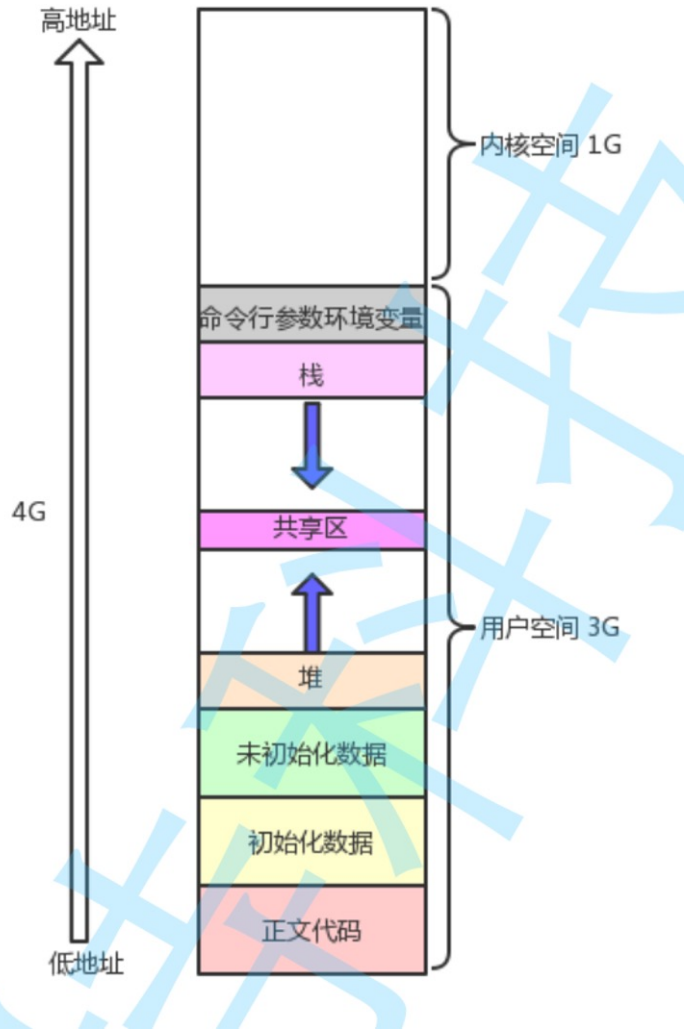
通过这种形式，我们就验证了各种类型变量的地址

我们还发现 – 堆区和栈区的地址中间有一大块镂空
具体用法我们以后再讲

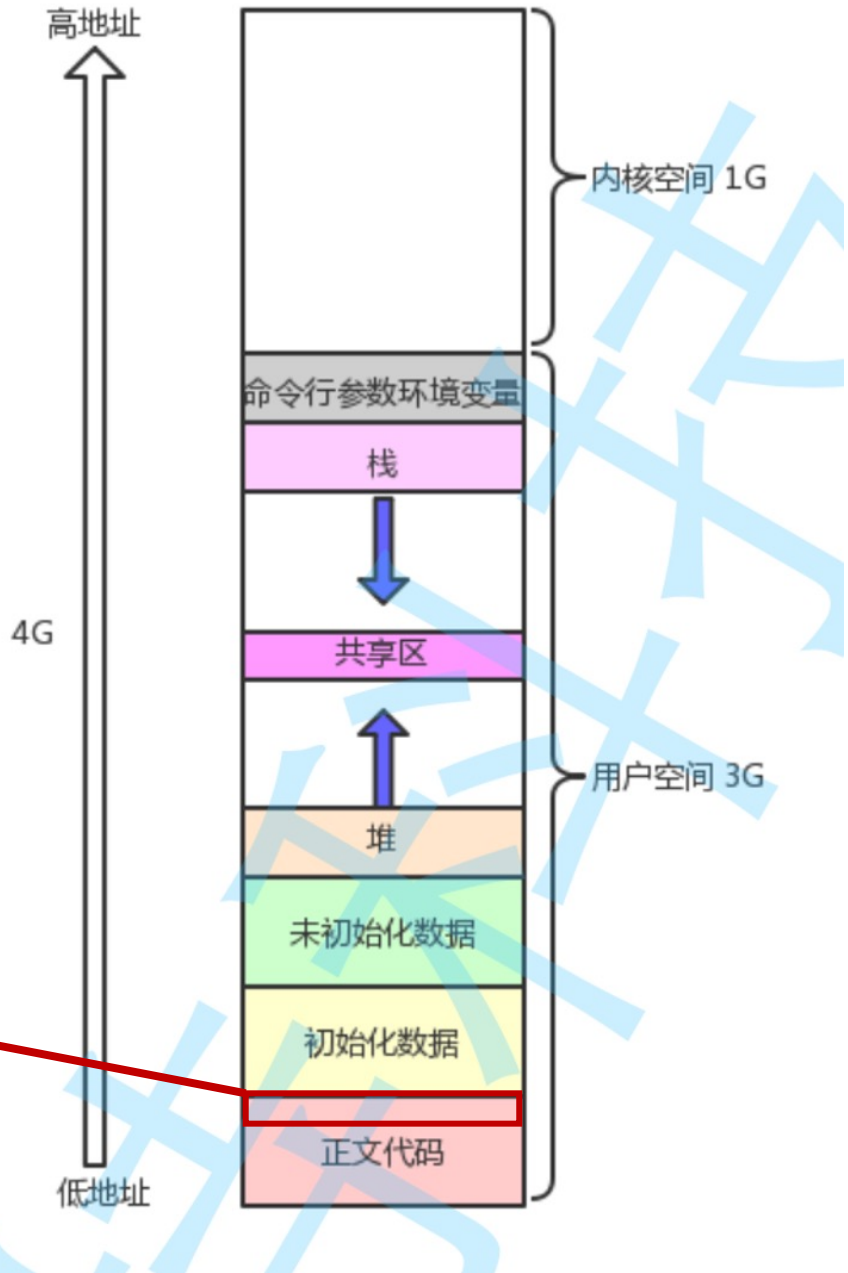
我们还可以验证，栈区是向下增长的
堆区是向上增长的

```
[yufc@learningmechine 0909]$ ./hello -a -b -c
code addr: 0x400710
init global addr: 0x42002c
uninit global addr: 0x420034
heap addr: 0x1b291010
stack addr: 0x7fdc181b40
argv[0]: 0x7fdc182380
argv[1]: 0x7fdc182388
argv[2]: 0x7fdc18238b
argv[3]: 0x7fdc18238e
env[0]: 0x7fdc182391
env[1]: 0x7fdc1823a2
env[2]: 0x7fdc1823b6
env[3]: 0x7fdc1823cf
env[4]: 0x7fdc1823e3
env[5]: 0x7fdc1823f3
env[6]: 0x7fdc182401
env[7]: 0x7fdc182423
env[8]: 0x7fdc18243f
env[9]: 0x7fdc182449
env[10]: 0x7fdc182b01
env[11]: 0x7fdc182bd8
env[12]: 0x7fdc182bf2
env[13]: 0x7fdc182c0a
env[14]: 0x7fdc182c1b
env[15]: 0x7fdc182c3a
env[16]: 0x7fdc182c51
```

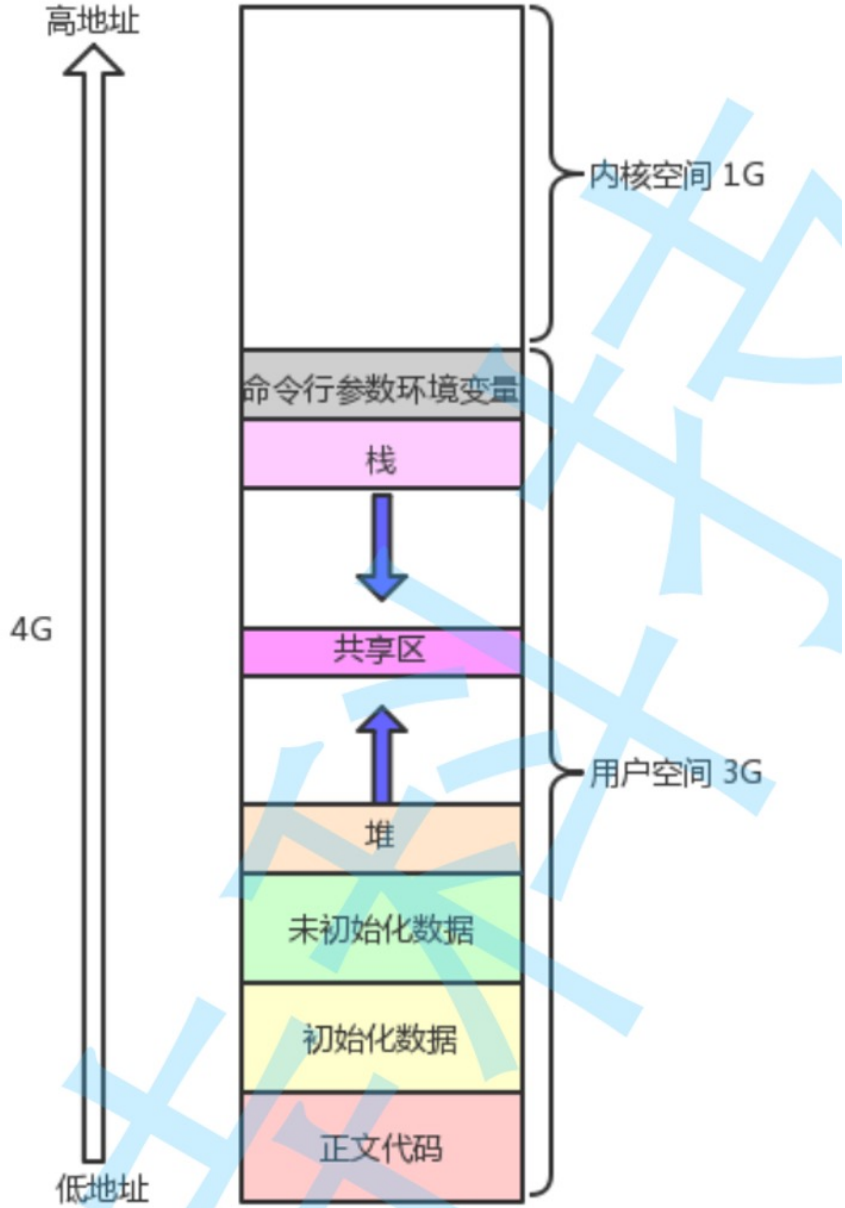
带上一些命令行参数
方便查看



static修饰局部变量的本质
将变量开辟在全局区域



这里面其实还有一个字符常量区
存这些：
const char* str = “abcdefg”;



1. 内核空间 vs 用户空间

2. Linux vs Windows

在32位下，一个进程的地址空间，取值范围是0x0000~0xffff

[0,3GB] – 用户空间
[3GB,4GB] – 内核空间 – 会讲一些整体认识

上一张ppt的结论 – 默认Linux下有效

如果在windows下跑，可能会不一样的结果

因为windows做了很多其他的设计

1. 什么是地址空间？

2. 地址空间是如何设计的？

3. 为什么要有地址空间？

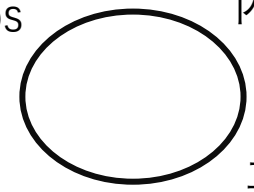
给各个进程画饼！

画饼：先描述再组织！！

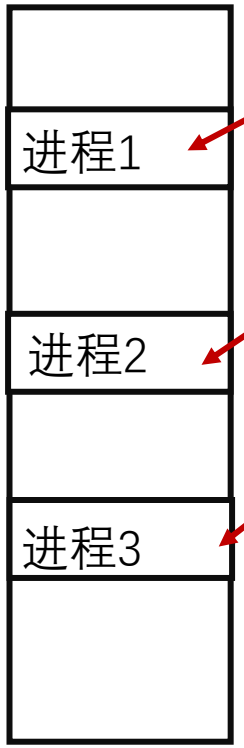
在内核中的 地址空间 本质也是一种数据结构！！
将来一定要和一进程关联起来！

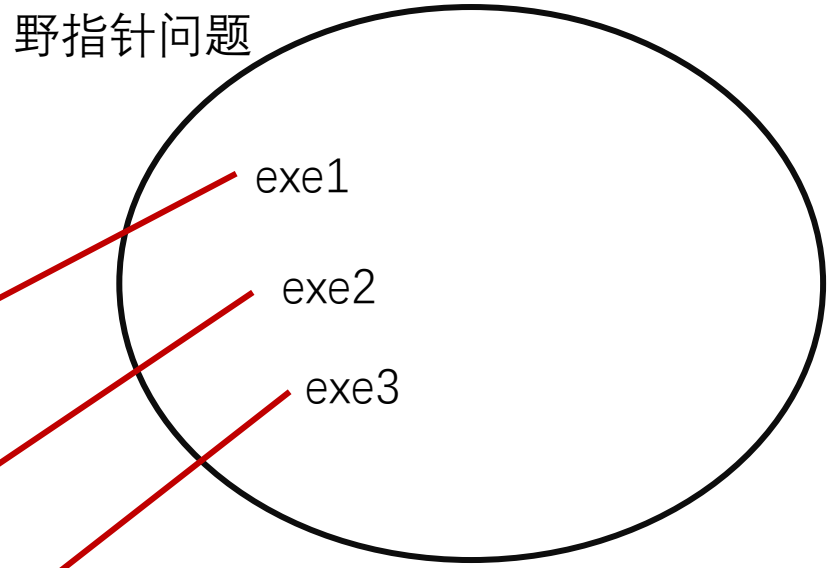
历史：直接访问物理内存

内存的特变：随时可以被读写



内存本身不关心
读写是否安全
它只负责读写
其他的是操作系统
做的事情！





如果我们直接使用物理内存 会出现什么状况？

假如进程1我们产生了一个野指针，这个指针我写错了

我们对这个指针进行修改，可能就把进程3中的数据给改了

与此同时，如果我们通过计算内存的位置，是不是就可以直接拿到别的用户的数据了？

所以直接使用物理内存的一个问题：特别不安全！！！

现代计算机，提出了以下几种方式

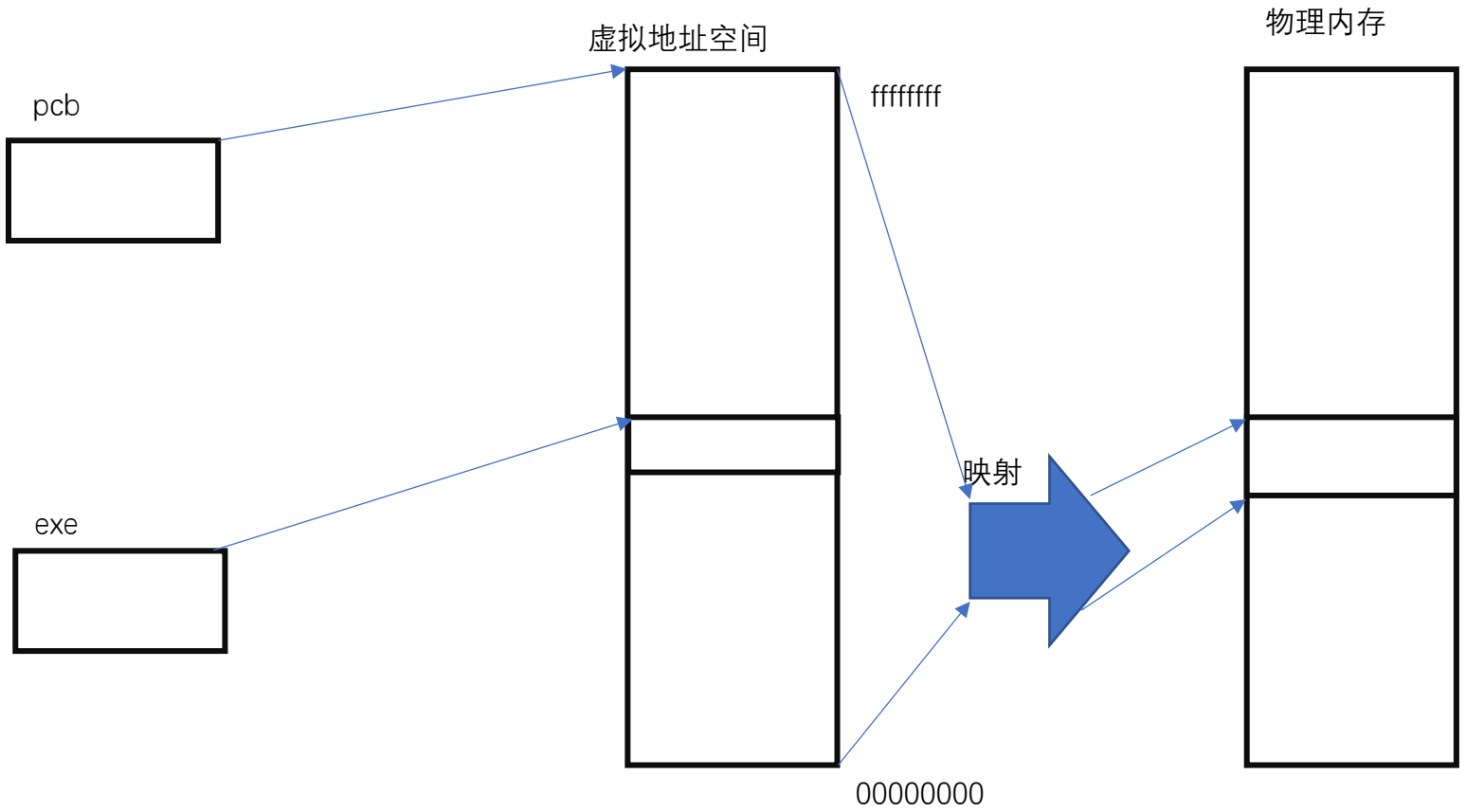
1. 使用task_struct

2. 使用虚拟地址空间

问题：

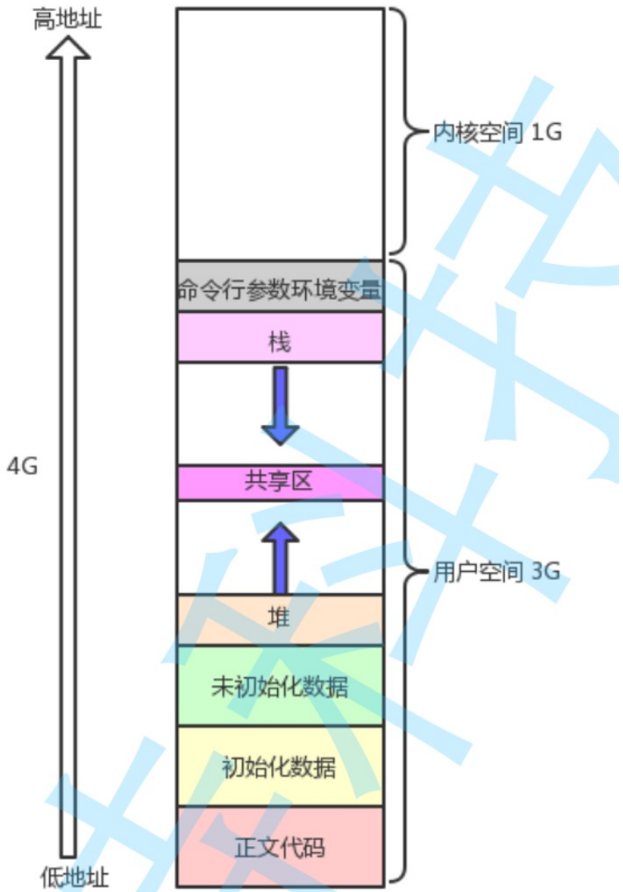
1. 映射机制？
2. 这种方式最终还是会访问物理地址啊？
   这样还是会出现不安全问题啊？
   玩意我的虚拟地址是一个非法地址呢？

可以禁止
这种行为
形成保护！



1. 虚拟地址空间究竟是什么？
2. 映射关系的维护是谁做的？

如何理解这个区域划分？



小学的时候男女同学桌面区域划分！
画出 “38线”
用C语言描述：

```
struct desktop
{
    int start;
    int end;
}

struct desktop one = {1,5}
struct desktop two = {5,10}
```

这，就是划分！

结合之前我们谈过的例子

我们得出的结论：

地址空间是一种内核数据结构
它里面至少要有：各个区域的划分！！

```
struct addr_room
{
    int code_start;
    int code_end;

    int init_start;
    int init_end;

    int uninit_start;
    int uninit_end;

    int heap_start;
    int heap_end;
    //...其他属性

}
```

但是，我们知道
这些区域是会变的哦！

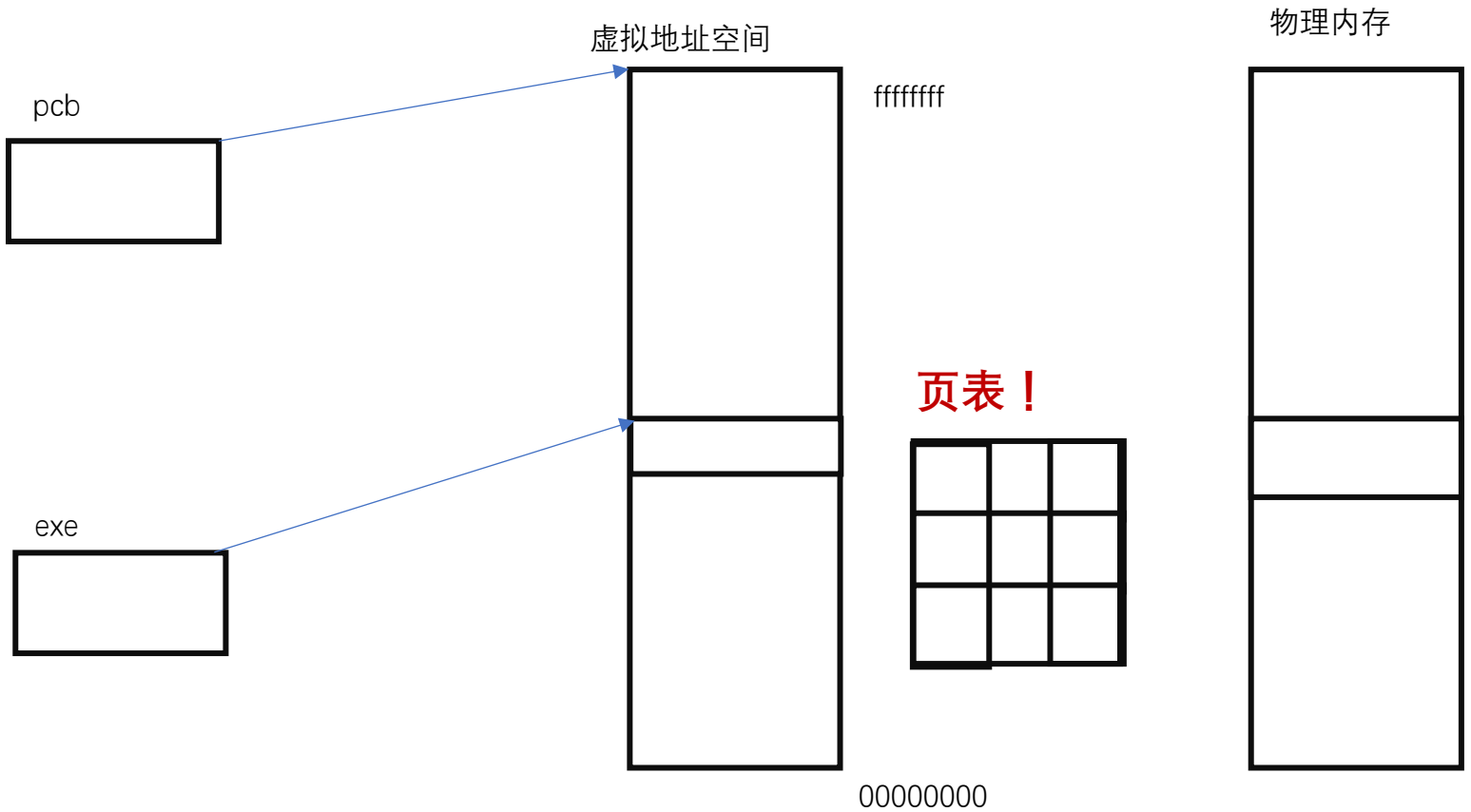
比如，堆和栈的增长

本质就是对start，end
进行+-对操作！

现在我们又知道多了一个task_struct里面的字段了！

叫做 mm_struct* mm !!

地址空间和页表（用户级）是每一个进程都私有一份的
只要保证，每一个进程的页表，映射的是物理内存的不同区域
就能做到，进程之间不会互相干扰进程的独立性！

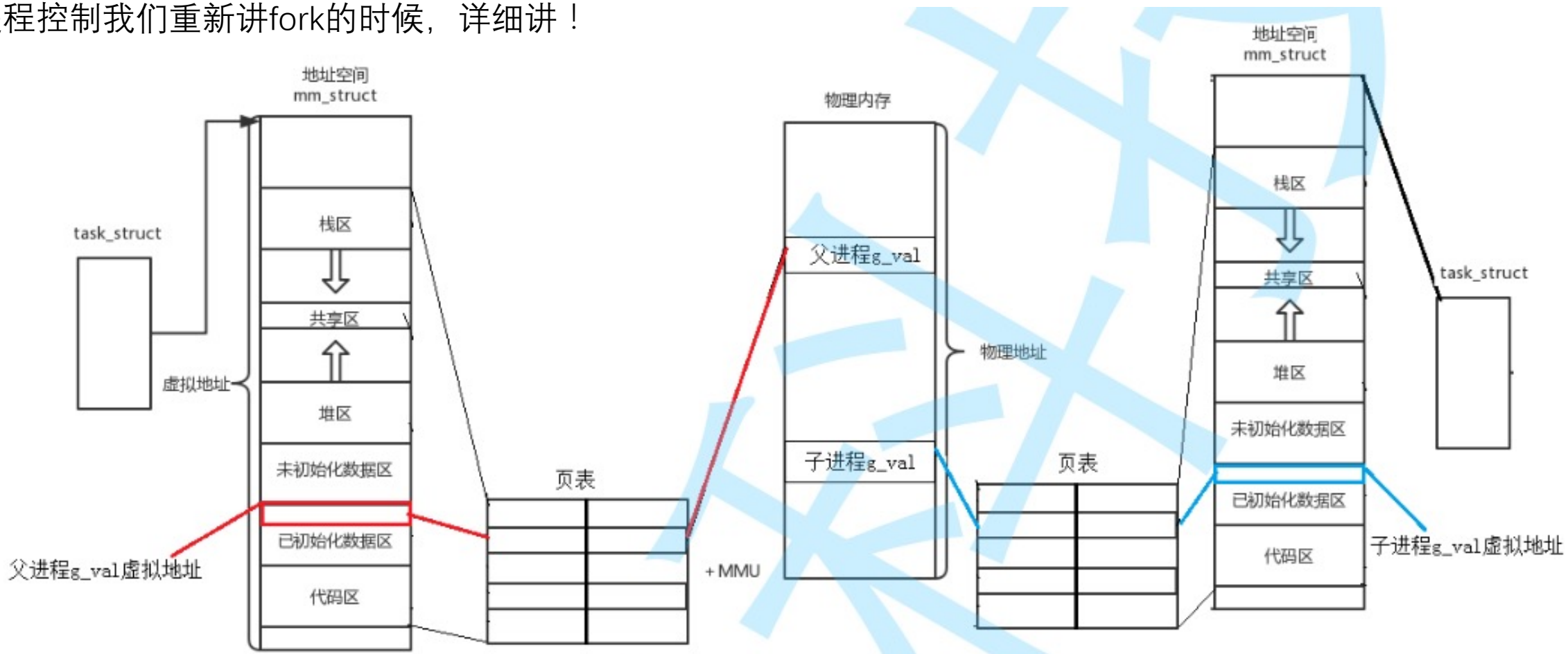


今天我们先把页表这个概念弱化一下

详细的我们会在线程里面去讲！

回答上一节课的问题：
为什么那个程序，地址一样，但是里面的值不一样？

虚拟地址是可以一样的！只要我和两个一样的虚拟地址，映射到物理内存上的不同地方即可！
我们刚刚代码中，g_val修改之前，指向同一个位置，但是修改之后，让它们指向了物理内存的不同位置，这种策略叫做写时拷贝！！
进程控制我们重新讲fork的时候，详细讲！



我们还要回答一个遗留问题：

一个变量怎么可能会保存不同的值？

Return会被执行两次
Return的本质不就是对值进行写入吗？ -- 此时发生了写时拷贝！
所以两个进程各自其实在物理内存中，有属于自己的变量空间！
只不过是在用户层面用同一个变量（虚拟地址！）来标识了！

扩展内容

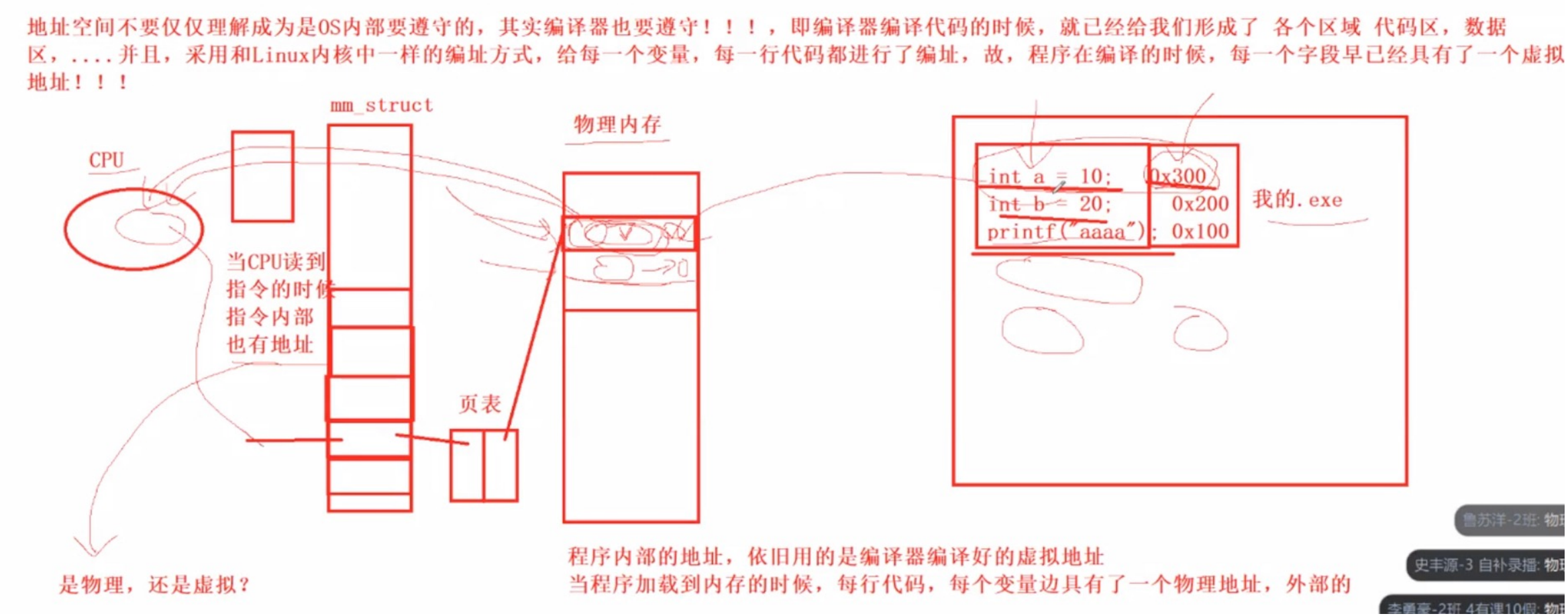
当我们的程序，在编译的时候，形成可执行程序，但没有被加载到内存中的时候，请问：
我们程序内部，有地址吗？？

其实，内部已经有地址了！

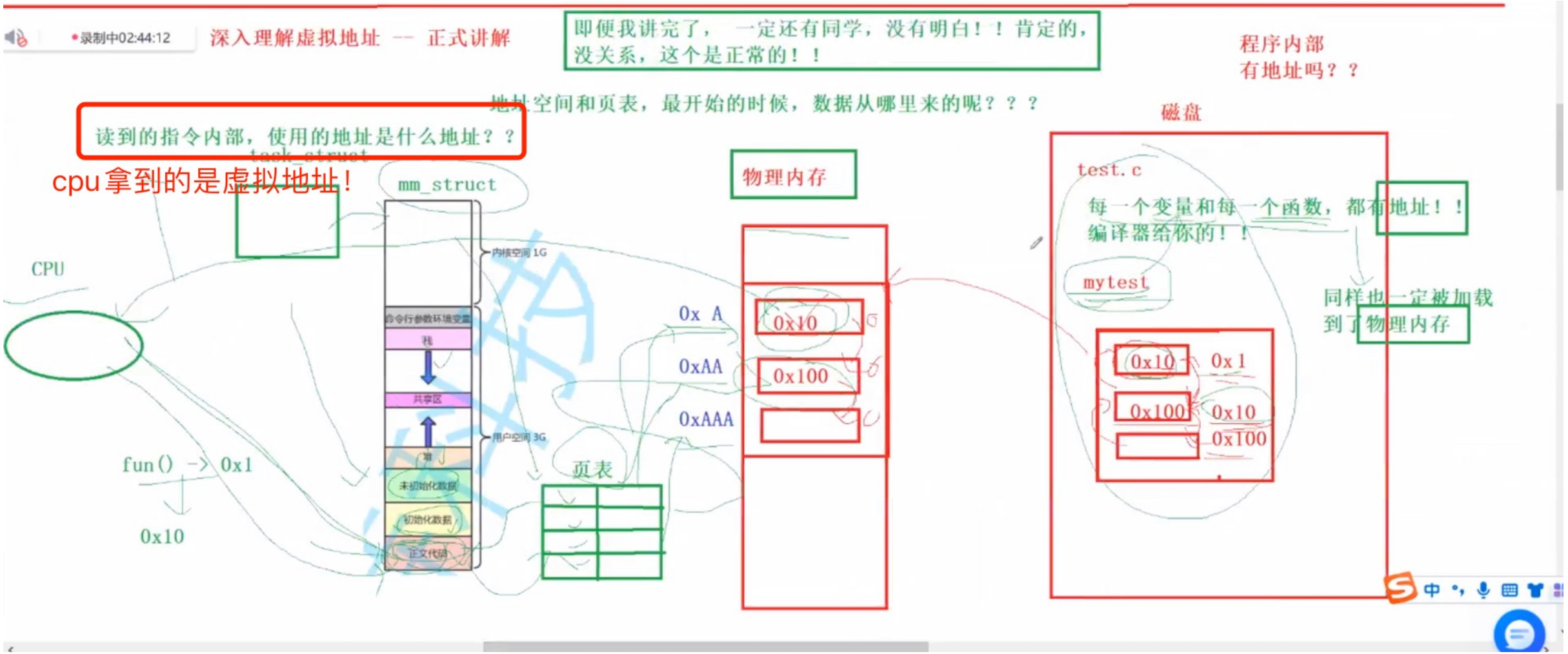
```
[yufc@VM-12-12-centos 0909]$ ls
hello.c  Makefile
[yufc@VM-12-12-centos 0909]$ make
g++ -o hello hello.c
[yufc@VM-12-12-centos 0909]$ ls
hello  hello.c  Makefile
[yufc@VM-12-12-centos 0909]$ objdump -afh hello

hello:     file format elf64-x86-64
hello
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004004b0

                virtual memery address
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000400274  0000000000400274  00000274  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.hash     0000001c  0000000000400298  0000000000400298  00000298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

重要结论：
地址空间不要仅仅理解成为是OS内部要遵守的，其实编译器也要遵守！
即编译器编译代码的时候，就一斤给我们的形成了各个区域（代码区，数据区）形成了地址！
并且，采用和Linux内核中一眼更多编址方式，给每一个变量，每一行代码都进行了编址！
故，程序在编译的时候，每一个字段早已经具有了一个虚拟地址！！

答案是：虚拟地址！CPU拿到的一直都是虚拟地址！！
这部分比较难理解！在0909视频的2:43:34处



重新再理解！


录制中02:44:12
深入理解虚拟地址 — 正式讲解
即便我讲完了，一定还有同学，没有明白！！肯定的，
没关系，这个是正常的！！
程序内部
有地址吗？？
地址空间和页表，最开始的时候，数据从哪里来的呢？？？
读到的指令内部，使用的地址是什么地址？？
cpu拿到的是虚拟地址！
task_struct
mm_struct
CPU
内核空间 1G
命令行参数环境变量
栈
共享区
堆
用户空间 3G
未初始化数据
初始化数据
正文代码
fun() -> 0x1
0x10
页表
物理内存
0x A
0xAA
0xAAA
0x10
0x100
磁盘
test.c
每一个变量和每一个函数，都有地址！！
编译器给你的！！
mytest
同样也一定被加载
到了物理内存
0x10
0x1
0x100
0x10
0x100