0913
进程地址空间与进程控制

回答一个问题：

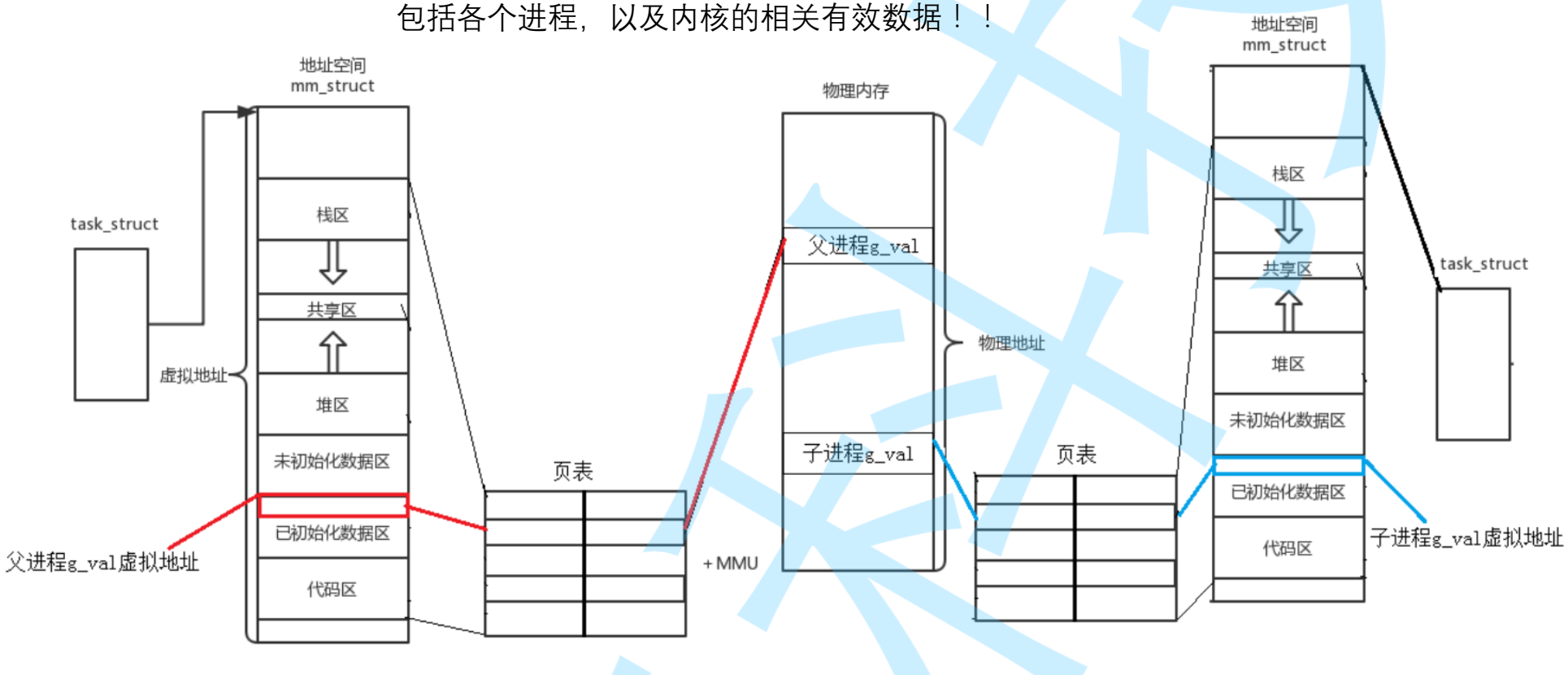
为什么要有地址空间？

1. 凡事非法的访问或者映射，OS都会识别到，并终止你这个进程！！

有效的保护了物理内存！！

所有的进程崩溃，
难道不就是进程退出吗？？
OS杀掉了这个进程

因为地址空间和页表是OS创建并维护的！是不是
也就也为这凡是想使用地址空间和页表进行映射，
也一定要在OS的监管之下来进行访问
至此，也便保护了物理内存中的所有的合法数据
包括各个进程，以及内核的相关有效数据！！



上面是为什么要地址空间的第一个理由
现在是为什么要有地址空间的第二个理由

2. 因为有地址空间的存在，因为有页表的映射的存在，我们的物理内存中，是不是可以对未来的数据进行任意位置的加载？
当然可以！！
物理内存的分配就可以和进程的管理之间，没有关系！！

物理内存的分配 -> 内核中的内存管理模块
进程管理 -> 内核中的进程管理模块

内存管理模块 vs 进程管理模块
这两部分就完成了解耦合！

所以，我们在C/C++语言上new malloc空间的时候，本质是在哪里申请的呢？
1. 物理内存 **2. 虚拟地址空间**

紧接着一个问题：
如果我们直接申请了物理空间，但是我不立马使用，是不是空间的浪费呢？
是的！！

所以！
本质上，有地址空间的存在，所以上层申请空间，其实是在地址空间上申请的，物理内存可以甚至一个字节都不给你！！
而你真正进行对物理地址空间访问的时候，才执行内存的相关管理算法，帮你申请内存，构建页表映射关系，然后，再让你进行内存的访问！

<真正对物理地址空间访问的时候，执行内存相关管理算法>这里这个动作
是OS自动完成，用户，包括进程，完全0感知！
这个叫做——延迟分配的策略！

那么，OS是如何知道，一些内存空间虽然在虚拟上给了，但是物理上还没给呢？
这里有个技术叫做 —— 缺页中断！（后面我们再完善这个概念）

为什么要有地址空间的第三个理由

因为物理内存中理论上可以任意位置加载，那么是不是物理内存中的几乎所有的数据和代码在内存是乱序的？

但是，因为页表的存在，它可以进行映射

那么是不是在进程视角所有的内存分布，都可以是有序的？

是的！

地址空间+页表的存在可以将内存的分布有序化！

地址空间是OS给进程画的大饼

->

结合第2条理由：进程要访问的物理内存中的数据和代码，可能目前并没有在物理内存中，同样，也可以让不同的进程映射到不同的物理内存！那么这样是不是就很容易做到，进程独立性的实现

->

进程的独立性，可以通过，地址空间+页表的方式实现！

得出的结论：

因为有进程的存在，每一个进程都认为自己拥有4GB空间（32），并且各个区域是有序的，进而可以通过页表映射到不同的区域，来实现进程的独立性！！

每一个进程不知道，也不需要知道其他进程的存在！！

重新理解 什么是挂起

加载本质就是创建进程，
那么是不是必须非得立马把所有的程序的代码和数据加载到内存中，并创建内核数据结构，建立映射关系呢？
答案是 不是！

在最极端的情况下，甚至只有内核结构被创建出来了
比如 task_struct, mm_struct, 页表
这些，其他的（映射关系这些）甚至都不需要！
此时操作系统很忙啊，说：我暂时先不想运行你！
此时这种状态，给他一个名字：

新建状态！

理论上，可以实现对程序的分批加载！

既然可以分批加载（换入），可以分批换出吗？

当然可以咯！

甚至这个进程段时间内不会再被执行了，比如阻塞！
我们玩一个游戏，我们的开机界面，是不是进入了主界面之后，开机界面这些代码和数据，就短时间内不会再被执行了？
此时，所以我们此时就可以把这些代码和数据换出内存！

一旦被换出，此时就叫挂起！

其实，页表映射的时候，可不仅仅映射的是内存磁盘中的位置，也可以映射哦！！

其实挂起也不是交换，因为我们在磁盘中也有映射，因此，挂起的时候其实数据和代码在内存中直接抹掉就行了！
直接通过页表对磁盘的映射把数据放到磁盘的swap中，
这个就是一个基本的挂起！

进程控制

本节重点：

- 学习进程创建,fork/vfork
- 学习到进程等待
- 学习到进程程序替换, 微型shell，重新认识shell运行原理
- 学习到进程终止,认识$?

面试题：请你描述一下，fork()创建子进程，操作系统都做了什么？？

fork创建子进程，是不是系统里多了一个进程？是的！

进程 = 内核数据结构 + 进程代码和数据
（OS） （一般从磁盘中来，也就是.c/.cpp -> .exe这些东西）

进程调用fork，当控制转移到内核中的fork代码后，内核做：

- 分配新的内存块和内核数据结构给子进程
- 将父进程部分数据结构内容拷贝至子进程
- 添加子进程到系统进程列表当中
- fork返回，开始调度器调度

创建子进程，给子进程分配对应的内核结构，必须子进程自己独有了，因为进程具有独立性！理论上，子进程也要有自己的代码和数据！
可是一般而言，**子进程时候没有加载的过程的**，因为子进程就是从父进程中来的，不是加载而来的，也就是说，**子进程没有自己的代码和数据！！！**
所以，子进程只能“使用”父进程的代码和数据！！**那么进程独立性如何保证？**
从代码层面：都是不可以被写的，只能读取，所以父子共享，没有问题，独立性得以保证
从数据层面：是可以修改的，所以必须分离
对于数据而言：1. 创建进程的时候，就直接拷贝分离 – 问题：可能拷贝了子进程根本就不会用到的数据空间，即使用到了，也可能只是读取
2. （下一页ppt继续）

对于数据而言：
1. 创建进程的时候，就直接拷贝分离 – 问题：可能拷贝了子进程根本就不会用到的数据空间，即使用到了，也可能只是读取

因此，创建子进程不会将不会被访问的，或者只会被读取的数据拷贝一份

但是，我们还是要拷贝的，那么什么数据值得拷贝呢？

→ 将来会被父或子进程写入的数据

一般而言，即便是OS，也无法提前知道那些空间无法会被写入！
即便是提前拷贝了，你会立马使用吗？不会

所以，OS选择了，写时拷贝技术，来进行将父子进程的数据进行分离

```c
int main()
{
    //我们曾经学习C语言的时候，肯定写过这样一份代码
    const char *str1 = "aaa";
    const char *str2 = "aaa";
    printf("%p\n",str1);
    printf("%p\n",str2);
}
```

```
[yufc@VM-12-12-centos 0913]$ ./myproc
0x400610
0x400610
[yufc@VM-12-12-centos 0913]$
```

这个例子和我们要阐述的观点没有直接关系，但是他告诉我们，编译器编译的时候都知道节省空间，更不用说OS了

总结：OS为何要选择写时拷贝的技术，对父子进程进行分离
1. 用的时候，再给你分配，是搞笑使用内存的一种表现
2. OS无法在代码执行前与之哪些空间会被访问

问题：
fork之后
是fork之后的代码共享
还是fork之后和之前的代码都共享？
所有的！！

我们先了解一些共识：

1. 我们的代码汇编之后，会有很多行代码，而且每行代码加载到内存之后，都有对应的地址

2. 因为进程随时可能被中断（可能并没有执行完），下次回来，还必须从之前的位置继续运行（不是最开始），就要求cpu必须随时继续下，当前进程执行的位置，所以，cpu内有对应的寄存器数据，用来记录当前进程的执行位置

3. 其实计算机表现出来的智能，其实不是硬件一个人做到的，硬件其实很无脑。比如内存，只负责存取，cpu其实也不是很聪明，其实他也不知道自己一直在干啥，它一直在做：取指令，分析指令，执行指令等这些事情而已。

我们曾经说过：

寄存器在cpu内，只有一份，寄存器内的数据，是可以有多份的！

我们曾说：寄存器的数据，是进程上下文的数据！

既然他是数据，那么eip这个寄存器在子进程创建的时候，要不要交给子进程？

虽然父子进程各自调度，各自会修改自己的eip
但是已经不重要了，因为子进程已经认为自己的eip起始值，就是fork之后的代码！！

所以子进程也从fork之后执行不是看不见fork之前的代码，而是因为有eip寄存器！