

1、

"进程有多个，但是 kernel 只有一份" 这句话本身是正确的，但它需要结合操作系统的层次和概念来理解。以下是更详细的解释：

### 1. 内核的唯一性

操作系统的 **kernel (内核)** 是整个系统的核心部分，它为用户进程提供底层的资源管理和硬件抽象层。通常来说，内核是 **唯一的**，无论系统中有多少个进程，所有进程共享同一个内核代码和数据结构。因此，内核是单一的，所有进程在执行系统调用时，都会进入同一个内核态代码。

### 2. 多个进程与独立的内核栈

虽然内核只有一份，但每个进程在切换到内核态时，都会有自己专属的 **内核栈**。每个进程的内核栈是为这个进程的内核态操作准备的私有栈空间，当进程进入内核态时，使用的是属于该进程的内核栈。因此，虽然内核代码是唯一的，但进程的执行上下文是隔离的（包括内核栈）。

### 3. 内核的共享

所有进程共享内核的代码和全局数据结构，例如文件系统、设备驱动、内存管理系统等。当多个进程执行相同的系统调用时，它们实际上是在调用相同的内核代码，只不过在不同的进程上下文中运行。

#### 理解这句话的背景：

- **内核一份**：指操作系统内核的代码和资源管理功能是唯一的，所有进程共享这个内核，内核代码不会为每个进程复制一份。
- **进程多个**：每个进程都有自己独立的用户态资源和内核态资源（如内核栈），虽然内核代码是唯一的，但每个进程在内核态执行时都有独立的执行上下文（包括内核栈和进程控制块等）。

就像线程：线程全局区、堆区共享。但是有时，人们又想有独立的自己的空间：local\_thread 关键字：堆上的数据 每个进程自己各自有一份独立的

2、walk 模拟硬件 MMU的功能：返回页表内容PTE 或写入页表内容

#### • walk():

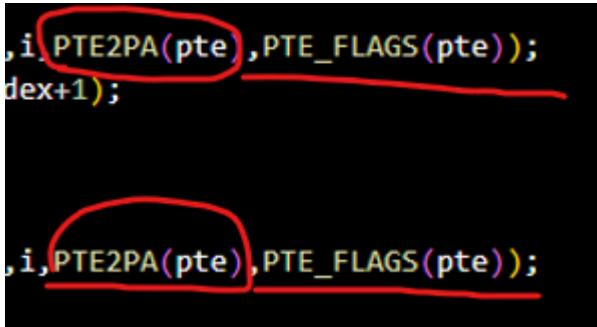
```
pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)
```

入参：【根页表地址pagetable】、【虚拟地址va】、【如果对应页表未分配，是否要进行分配的flag位 alloc: 1分配, 0不分配】。

返回参数：【叶子页表中的页表项pte\_t的地址，即&pte\_t】。

进行映射

Note: walk (pagetable, va, 0) 函数 返回的是 pte (如果 第三个参数是 1 则是写入 pte)，walkaddr (pagetable, va) 函数返回的是 pa



```
, i, PTE2PA(pte), PTE_FLAGS(pte));
dex+1);

, i, PTE2PA(pte), PTE_FLAGS(pte));
```

3、qemu的 gdb调试：内部有一个 gdb输出端口，端口号 20666。。，然后在写好 gdb 配置文件，remote localhost 连上端口即可

如果代码出现了问题，在gdb中看到的地址，你可以直接在kernel.asm找到具体的行，分析问题的原因，然后再向相应的地址设置断点。

在gdb中输入layout asm，可以在tui窗口看到所有的汇编指令。再输入layout reg可以看到所有的寄存器信息。

kernel.asm 汇编文件

#### 4、页表 Lab

普通用户进程的页表也是存在内核空间的。这很容易理解，毕竟页表没有VMA来对应。在应用进程创建的时候，task\_struct->m\_struct描述内存信息，mm->gpd指定页表基地址。页表的分配是通过调用内核伙伴算法接口分配到物理内存，内核在启动阶段已经创建了内核页表，

用户进程的页表可以分为两个部分，其指定了不仅有0-3GB（传统32位系统）的用户空间映射（也可能还没建立映射），还有3-4GB的内核空间页。而3-4GB的内核空间页表是内核之前就已经创建好的，用户进程页表会复制内核页表，所以我们说，所有的用户进程共享内核页表。

用户进程在通过中断或者系统调用进入内核之后，MMU页表基地址依旧是当前进程的页表基地址，只不过在内核态可以访问所有的地址空间（0-4GB），当然这时候在内核态如果访问低地址（0-3GB）内存的话，就是访问到了当前进程的用户地址空间。因此使用copy\_from\_user的时候，用户空间地址参数在内核是可以访问的，因为此时内核可以访问该进程的用户空间页表。copy\_to\_user也一样。

将 kernel页表复制一份

copy\_in copy\_out

### Simplify copyin / copyinstr (hard)

内核的 copyin 函数读取用户指针指向的内存。它通过将用户指针转换为内核可以直接解引用的物理地址来实现这一点。这个转换是通过在软件中遍历进程页表来执行的。在本部分的实验中，您的工作是将用户空间的映射添加到每个进程的内核页表（上一节中创建），以允许 copyin（和相关的字符串函数 copyinstr）直接解引用用户指针。

本来是要通过地址（对用户地址进行翻译）翻译之后，才能继续进行操作

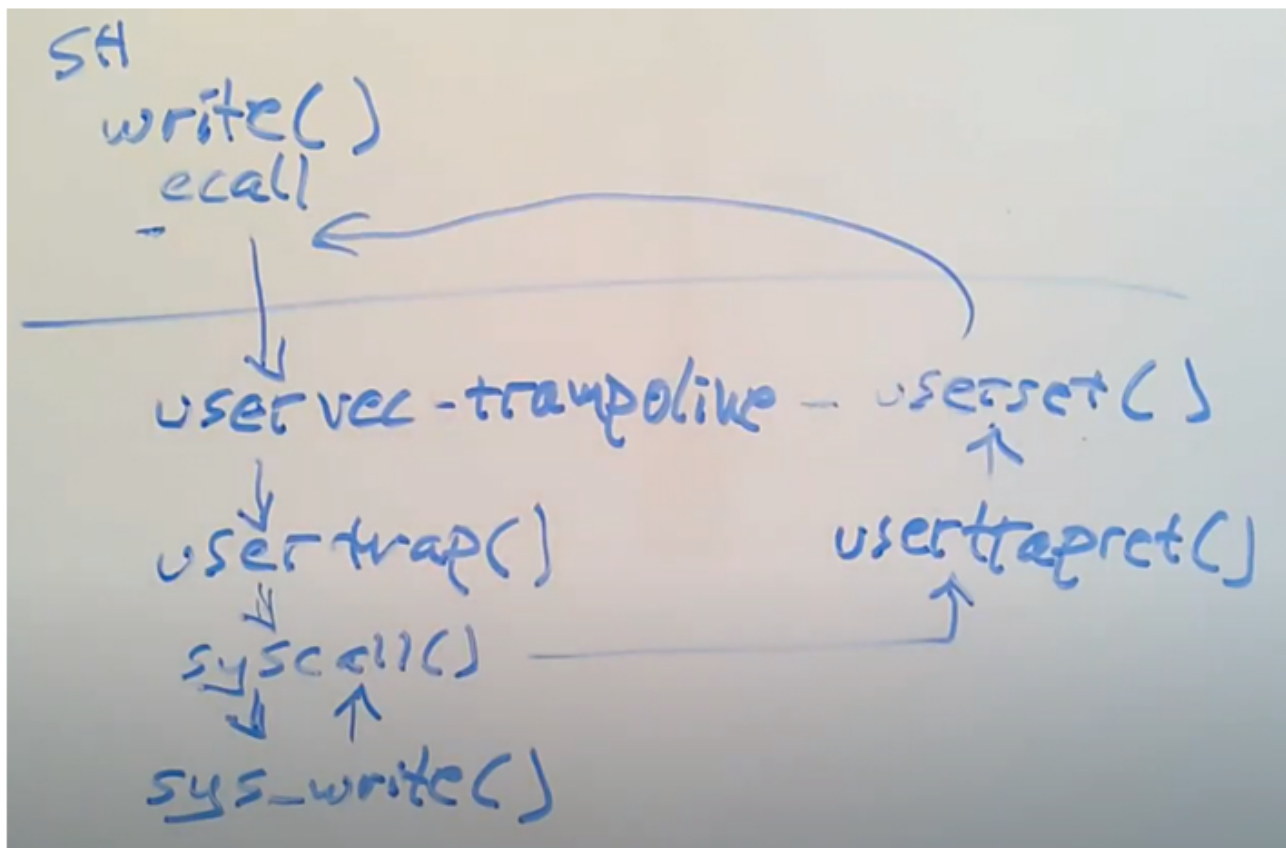
Question: fork 中, 页表复制用到的是 np (新进程, 而不是 p (父进程)) (无论是直接写复制也好, 还是调用 vm.c 中写的复制页表函数传参也好, 都是 np) 原因: 在于父进程和子进程可以不是同一时间结束的!!! 所以父进程先结束, 子进程若没有自己的页表, 则地址查询失败!!! 这也是为什么, 子进程和父进程的三级映射是不同的, 但是线程确可以是相同的, 因为线程结束时, 是都结束了 (除非是线程分离, 而且从这里也能看出, 如果某线程分离出去, 则它一定拷贝了一份页表出去!!!!)

## 5、几个重要的寄存器

### 几个比较重要的寄存器:

- 1、堆栈寄存器 sp
- 2、程序技术器 pc
- 3、mode 标志位 user/supervisor
- 4、CPU 控制寄存器: 如 SATP (存放 high pagetable 的地址)
- 5、STVEC 寄存器: 指向内核中处理 tarp 指令的起始地址
- 6、SEPC 寄存器: 在 trap 的过程保存程序计数器的值
- 7、SSRATCH 寄存器 (交换保存 a0 寄存器内容, 之后 kernel 可随意使用 a0)

## 6、跳床 trampoline



uservec userret

7、ecall: mode改为 supervisor 保存现在的程序计数器 (SEPC) PC设为 trap.c所在的地址位置 STVEC

8、懒分配 我们先让 进程 sbrk调用成功，让进程 认为自己的 va已经有了对应的 pa；但是 实际上并没有，并且，下次它 执行 有关 va的操作时，产生 page fault，再对此 va进行 pa分配以及pagetable 进行映射

9、va后三位不为0，则va一定不是一个页表的起始地址，则 解除映射函数一定返回 false

所以 页表的 uvmunmap 解除映射操作时：都要求 va 是 aliend 和页表对齐（都是一张表 一张表的解除，然后 for 循环中 a 的起始位置 是 va，所以 va 必须是某张表的开始地址，也就是说 va 的后三位均为 0，所以  $va \% PGSIZE == 0$ ）

```
// Remove npages of mappings starting from va. va must be
// page-aligned. The mappings must exist.
// Optionally free the physical memory.
```

```
if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");
```

10、

再提一下：锁的提出是为了，提升 CPU 的并发能力；但为了锁正常使用，又要限制 CPU 的并发能力

11、ext3 多事务 日志处理方式

begin\_op end\_op（提交事务）

在最开始首先会对log的outstanding字段减1，因为一个transaction正在结束；其次检查committing状态，当前不可能在committing状态，所以如果是的话会触发panic；如果当前操作是整个并发操作的最后一个的话（log.outstanding == 0），接下来立刻就会执行commit；如果当前操作不是整个并发操作的最后一个的话，我们需要唤醒在begin\_op中sleep的操作，让它们检查是不是能运行。

（注，这里的outstanding有点迷，它表示的是当前正在并发执行的文件系统操作的个数，MAXOPBLOCKS定义了一个操作最大可能涉及的block数量。在begin\_op中，只要log空间还足够，就可以一直增加并发执行的文件系统操作。所以XV6是通过设定了MAXOPBLOCKS，再间接的限定支持的并发文件系统操作的个数）

最初write请求会被发到block cache。block cache就是磁盘中block在内存中的拷贝，所以最初对于文件block或者inode的更新走到了block cache。

write 首先作用到 block cache 上；最后是否会作用到磁盘上，还看最后 write 写磁盘操作是否成功