

# Lab3.内存管理

在进行本实验之前，你需要进行分支管理，请在工程目录下输入以下命令：

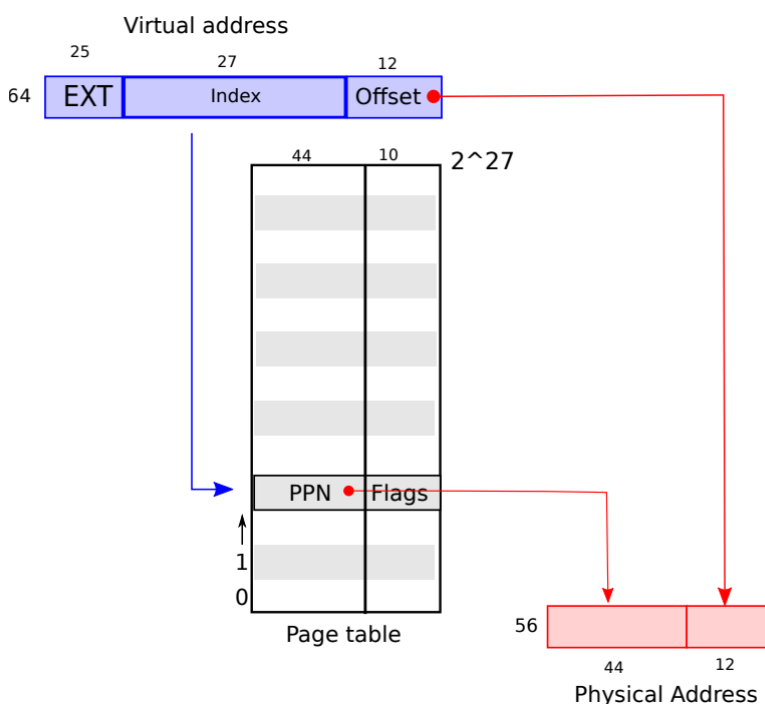
```
git fetch
git checkout lab3
make clean
```

对于内存管理，我想大家对相关概念也比较熟悉，从逻辑地址到物理地址之间的地址变换，到动态内存的分配与回收，再到虚拟内存中的分段式、分页式内存管理，大家在操作系统理论课中也学习过。在本次实验中，我们会主要聚焦于虚拟内存，具体来说，我们会聚焦于xv6中页表相关的内容。

## SV39的硬件机制

相信在理论课的学习中你们对页表有了足够的了解，接下来我们会根据xv6的页表来详细探讨。xv6是基于SV39页表运行的，这种页表只使用64位虚拟地址的低39位。那么高25位有什么作用？高25位保留下来可以供未来RISC-V定义更多级别的转换，这也是RISC-V的灵活之处。此外，SV39 分页模式规定 64 位虚拟地址的 [63:39]这 25 位必须和第 38 位相同，否则 MMU 会直接认定它是一个不合法的虚拟地址。

我们先来看看SV39页表进行虚拟地址和物理地址转换的过程。

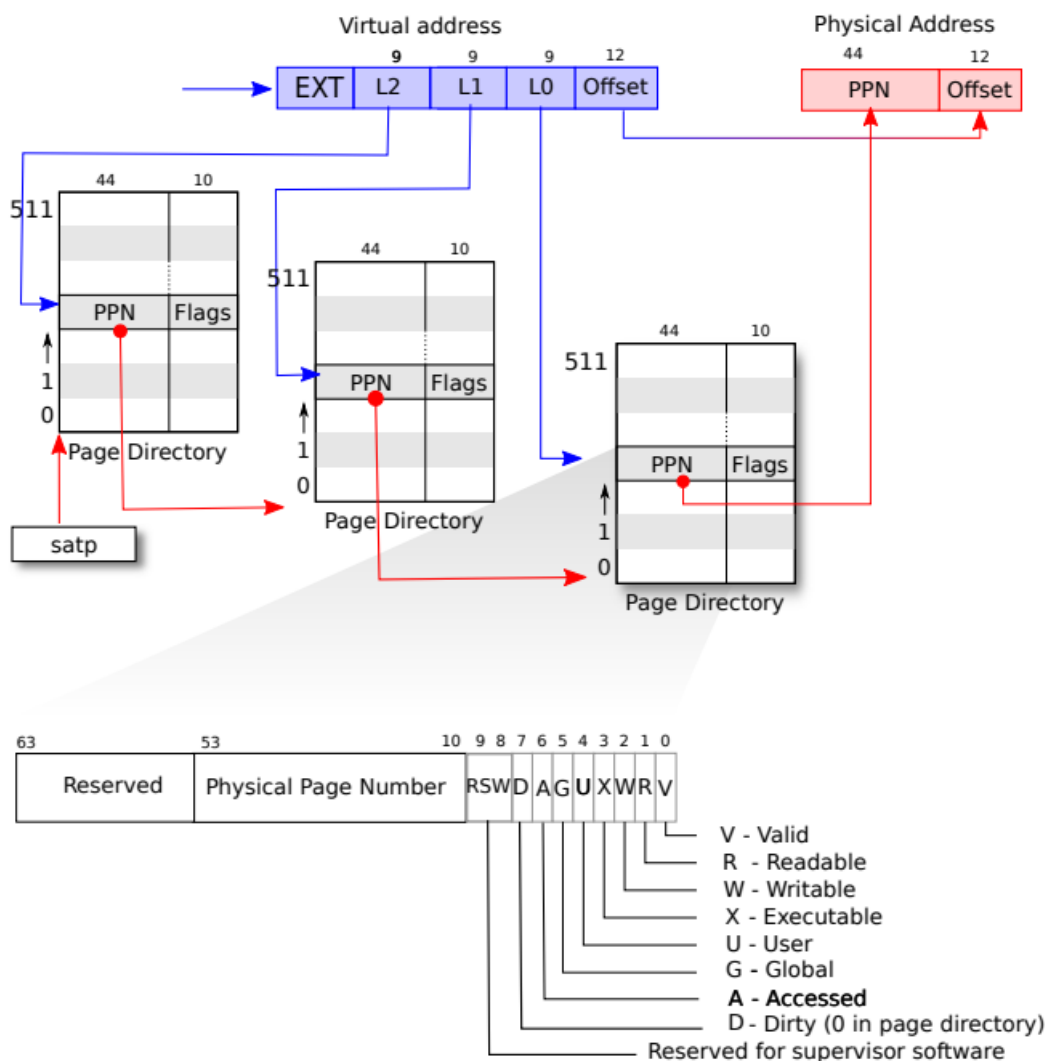


如图所示，页表在逻辑上是由一个由 $2^{27}$ 个PTE(Page Table Entries)组成的数组。每个PTE包含一个44位的PPN和一些标志位。在转换过程中，虚拟地址39位中的前27位索引页表，找到一个PTE，然后生成一个56位的物理地址，其中前44位来自PTE的PPN，后12位来自虚拟地址的偏移（Offset），这个偏移量必须是12位，因为在RISC-V中一个页是4KB，也就是4096字节。

## 三级页表

事实上，实际的页表被设计为三级页表。

之前所说的27位的索引，实际上分为了三个等长的索引，分别是的L2、L1、L0，这三个等长的索引各9位，所以每一级页表有 $2^9 = 512$ 个PTE。分页硬件使用27位中的前9位（L2）在根页表页面中选择PTE，中间9位（L1）在下一级页表页面中选择PTE，最后9位（L0）选择最终的PTE。此外，最高一级页表的PPN用于指向中间级页表，中间级页表的PPN用于指向最后一级页表，而最后一级页表的PPN就是物理地址中的PPN。下面是三级页表和PTE的具体内容的图示：



我们可以看到PTE中有10位被设置为标志位。

- **PTE\_V**：指示PTE是否存在。

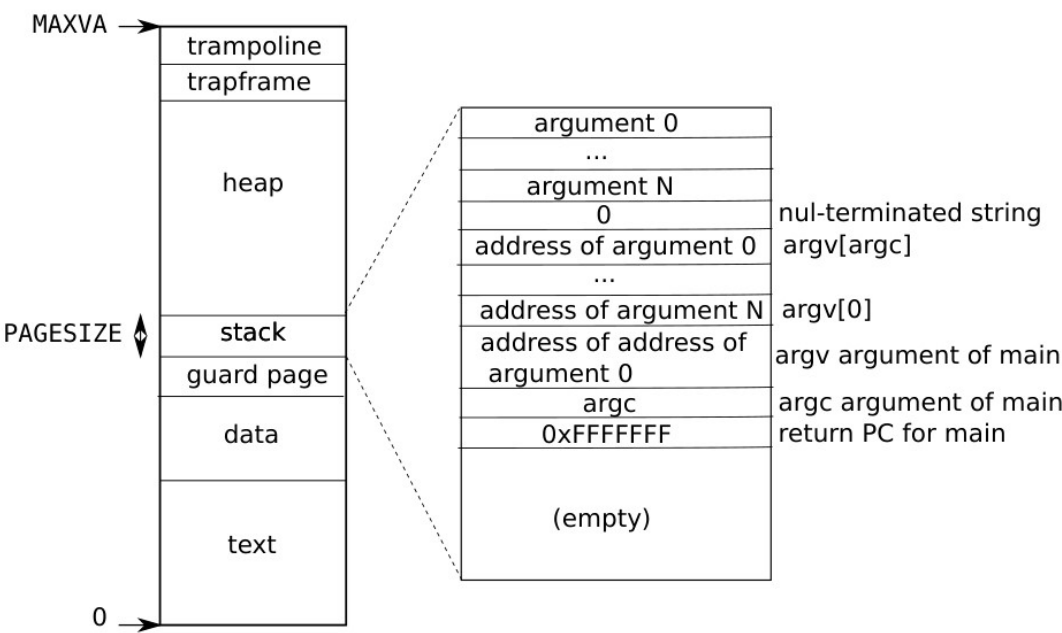
- `PTE_R`：控制是否允许指令读取到页面。
- `PTE_W`：控制是否允许指令写入到页面。
- `PTE_X`：控制CPU是否可以将页面内容解释为指令并执行它们。
- `PTE_U`：控制用户模式下的指令是否被允许访问页面。

在 `kernel/riscv.h` 中我们可以看到上述相关结构的定义。

# 地址空间

## 进程地址空间

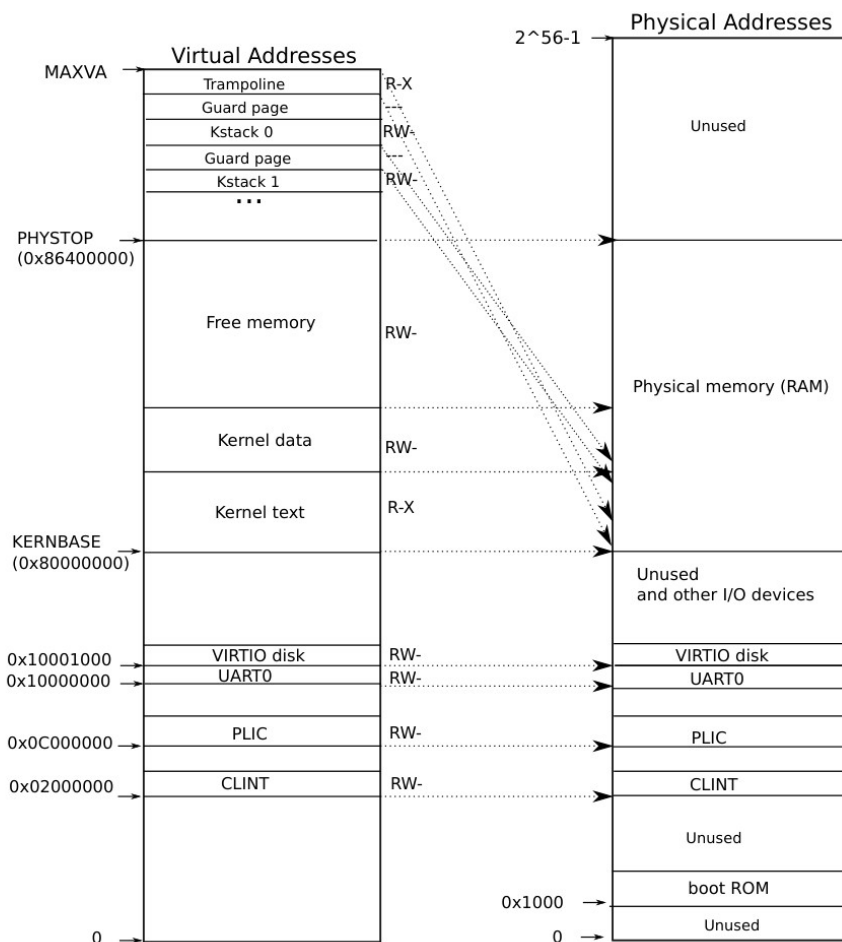
每个进程都会有一个属于自己的单独页表，进行进程切换时，也会切换页表。在xv6中，一个进程的用户内存从虚拟地址0开始，到MAXVA。当进程向xv6请求更多的用户内存时，xv6会使用 `kernel/kalloc.c` 中的 `kalloc` 来分配物理页面，然后将PTE加到进程页表中。下面是进程地址空间的布局图：



其中除了我们熟悉的 `data`，`text`，`stack`，`heap` 段外，还有其他一些段，`guard page` 的作用是针对用户栈检测是否溢出，`trampoline` 和 `trapframe` 的作用在此先不讨论。

## 内核地址空间

在xv6中，只有一个内核空间的页表，描述了内核虚拟地址访问物理内存和各种硬件资源。`kernel/memlayout.h` 声明了xv6内核内存布局的常量，阅读代码会发现相关信息和下图中的描述一模一样。



事实上，上图右半部分的硬件资源结构完全由硬件设计者决定，正如你们之前的操作中发现，操作系统从地址 `0x80000000` 开始运行，这个地址其实也是由硬件设计者决定的。在这里，我们是使用了 QEMU 来模拟了这样一台计算机。从物理地址 `0x80000000` 开始并至少到 `0x86400000` 结束的 RAM（物理内存），xv6 称结束地址为 `PHYSTOP`。QEMU 模拟还包括 I/O 设备，如磁盘接口。QEMU 将设备接口作为内存映射控制寄存器暴露给软件，这些寄存器位于物理地址空间 `0x80000000` 以下。内核可以通过读取/写入这些特殊的物理地址与设备交互，这种读取和写入与设备硬件而不是 RAM 通信。

内核使用直接映射的方式获取内存和内存映射设备寄存器。也就是说，将资源映射到等于物理地址的虚拟地址。例如，内核本身在虚拟地址空间和物理内存中都位于 `KERNBASE=0x80000000`。直接映射简化了读取或写入物理内存的内核代码。

当然，如图所示，也有部分的内核虚拟地址不是通过直接映射，这里暂时不展开讨论。

回到 xv6 的代码中，操作系统如何创建一个地址空间？

“源码面前，了无秘密”，接下来需要你仔细阅读代码，搞清楚 xv6 的内存管理机制。

### 💡 注意！

在开始下面的练习之前，请保证你已经浏览了下面的文件并对其中的代码有了基本的认识：

- `kernel/memlayout.h`: 它捕获了内存的布局。
- `kernel/vm.c`: 包含大多数虚拟内存相关的代码。
- `kernel/kalloc.c`: 包含分配和释放物理内存的代码。

## 打印页表

你需要实现一个简单的工具，帮你了解三级页表的构造，以及有助于你后续进行页表相关的调试。

在 `kernel/vm.c` 中实现 `vmprint()`，它应当接收一个 `pagetable_t` 作为参数，其中 `kernel/riscv.h` 末尾处相关的宏定义会对你有帮助。如果你不知道怎么实现该函数，可以参考 `freewalk()`。

打印页表格式如下：

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.....0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

第一行显示字符“page table”和整个页表的内存始址。之后的每行对应一个 PTE，包含树中指向页表页的 PTE。每个 PTE 行都有一些“..”的缩进表明它在树中的深度。每个 PTE 行显示其在页表页中的 PTE 索引、PTE 比特位以及从 PTE 提取的物理地址。**不要打印无效的 PTE。**在上面的示例中，顶级页表页具有条目 0 和 255 的映射。条目 0 的下一级只映射了索引 0，该索引 0 的下一级映射了条目 0、1 和 2。

具体来说，第二行，“..”表示 L2 级页目录，后面的 0 表示一个有效页表项在 L2 级页目录中的索引号；“pte”后面的 `0x0000000021fda801` 表示的是该 PTE 的 bits 值；而“pa”后面的 `0x0000000087f6a000`，表示的是该页表项所对应的物理地址。

第三行，“... ”表示的是 L1 级页目录，其他的和上面的表述相同。

你的代码可能会发出与上面显示的不同的物理地址。条目数和虚拟地址应相同。

在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，可以打印第一个进程的页表，启动 xv6 时可以验证你的结果是否正确。

使用 `python3 grade-lab-3 pte` 可以测试你的程序。

完成了打印页表，相信你对xv6的三级页表机制有了一定的了解，接下来我们需要使用 `vmprint()` 作为工具，探索xv6中创建内核地址空间的细节，看看内核虚拟地址是如何直接映射到物理内存的。

### 调试 `kvm_init()`，验证映射关系

使用gdb在 `kvm_init()` 处设置一个断点，选择其中一个 `kvmmap` 函数映射的I/O设备，在其后面用 `vmprint` 打印它的页表，完成之后，根据三级页表的构造，通过物理地址验证打印页表中的PTE序号，这一步可以利用gdb的打印功能完成。

## 缺页异常

现在新的问题产生了，我们在上面讨论了页表，而我们知道，如果我们需要的页不在物理内存中，就会发生缺页异常（page fault）。正常来说，操作系统对于不同的异常会有异常处理程序，而xv6是一个极其简单的操作系统，遭遇异常时，它的处理方式也非常简单：如果用户空间中发生异常，内核将终止故障进程。如果内核中发生异常，则内核会崩溃。

在现代计算机中，遭遇缺页异常可以使用lazy allocation，copy-on-write fork，demand paging，memory mapped files等方式来处理。接下来我们会讨论并在xv6中实现lazy allocation，这是一种较为简单的处理方式。

在讨论lazy allocation之前，思考一下，我们要处理缺页异常，什么信息是重要的？显然，我们需要知道出错的虚拟地址，缺页异常的类型，以及触发缺页异常的指令的地址。

## Lazy allocation

首先看看一个重要的系统调用：`sbrk`。`sbrk`是xv6提供的系统调用，它使得用户应用程序能扩大自己的heap。当一个应用程序启动的时候，`sbrk`指向的是heap的最底端，同时也是stack的最顶端。

在xv6中 `sbrk` 的主要功能是由 `kernel/proc.c` 中的函数 `growproc` 实现的，`growproc` 根据 `n` 是正的还是负的调用 `uvma11oc` 或 `uvmdeallc`。`uvma11oc` 用 `ka11oc` 分配物理内存，并用 `mappages` 将PTE添加到用户页表中。`uvmdeallc` 调用 `uvmunmap`，`uvmunmap` 使用 `walk` 来查找对应的PTE，并使用 `kfree` 来释放PTE引用的物理内存。具体的细节可以阅读源码进行思考。

所以，当 `sbrk` 实际发生或者被调用的时候，内核会分配一些物理内存，并将这些内存映射到用户应用程序的地址空间，然后将内存内容初始化为0，再返回 `sbrk` 系统调用。这样，应用程序可以通过多次 `sbrk` 系统调用来增加它所需要的内存。类似的，应用程序还可以通过给 `sbrk` 传入负数作为参数，来减少或者压缩它的地址空

间。

但是，目前的xv6中一旦调用了 `sbrk`，内核会立即分配应用程序所需要的物理内存。实际上，对于应用程序来说很难预测自己需要多少内存，所以通常来说，应用程序倾向于申请多于自己所需要的内存。这意味着，进程的内存消耗会增加许多，但是有部分内存永远也不会被应用程序所使用到。

而lazy allocation要解决的问题，就是在调用 `sbrk` 时，仅仅是记录增长，但不分配物理内存，当之后应用程序使用到了新申请的那部分内存，因为我们还没有分配内存，所以会触发缺页异常，于是这时候我们就可以分配内存，使得程序重新执行下去。

### 取消 `sbrk` 的内存分配

修改 `kernel/sysproc.c` 中的函数 `sys_sbrk()`，使得系统调用 `sbrk` 只增加进程内存的大小，但不分配内存。

修改完成后，在xv6中输入：`echo hi`，你会发现出现了缺页异常。

注意观察此时缺页异常的打印结果，输出了 `scause` 寄存器内容，我们可以看到它的值是15，查阅RISC-V手册可以发现这是一个Store page fault。事实上，在RISC-V中，缺页异常只有Load page fault和Store page fault两种，分别对应的 `scause` 寄存器的值为13和15。

那么要如何解决这个缺页异常？观察缺页异常的打印，可以看到首先出现异常的是 `usertrap()`，没有正确处理 `scause` 寄存器值为15的问题，于是我们定位到函数 `usertrap()`，可以看到，这里有 `scause` 寄存器值为8时针对系统调用的处理，所以我们需要在此添加缺页异常的处理：

```
else if(r_scause() == 13 || r_scause() == 15){
    // page fault
    uint64 va = r_stval();
    printf("page fault %p\n", va);
    uint64 ka = (uint64)kalloc();
    if(ka == 0){
        p->killed = 1;
    }
    else{
        memset((void *)ka, 0, PGSIZE);
        va = PGROUNDDOWN(va);
        if(mappages(p->pagetable, va, PGSIZE, ka,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            kfree((void *)ka);
            p->killed = 1;
        }
    }
}
```

```
}
```

首先我们会打印一些调试信息，在哪个虚拟地址发生了缺页异常。然后如果没有物理内存可以分配了，我们会杀死进程；如果有物理内存，首先会将内存内容设置为0，之后将物理页指向用户地址空间中合适的虚拟内存地址。具体来说，我们首先将虚拟地址向下取整，再将物理内存地址跟取整之后的虚拟内存地址的关系加到页表中，对应的PTE需要设置常用的权限标志位。

再次在xv6中运行 `echo hi`，你会发现并没有正常工作。这里出现了两个缺页异常，且与上一步一样的是，`uvmunmap` 在报错。

### 解决这个panic

定位到代码中的这个panic，思考一下这里没有映射的内存是什么内存。事实上，这个panic的意思是我们删除了并不存在映射关系的页面，想想lazy allocation的机制，想办法解决掉这个panic。

解决掉这个panic之后，两个缺页异常仍然存在，但是 `echo hi` 已经可以正常工作了。

到这里，已经实现了一个简单版本的lazy allocation，然而，这里并没有检查触发缺页错误的虚拟地址小于 `sbrk()` 分配的内存大小时的情况，也没有考虑如果 `sbrk()` 申请的内存是负数的情况。想想看，申请的内存是负数意味着什么？

### 完善lazy allocation

首先基于上述的工作，处理 `sbrk()` 申请的内存是负数的情况，以及如果某个进程在高于 `sbrk()` 分配的任何虚拟内存地址上出现缺页异常，则中止该进程，同时用户栈下面的无效页面上发生的异常也是中止进程，注意阅读 `kernel/proc.h` 文件，对这一步很有帮助。

处理所有因为页不存在而发生的panic，解决方法和上面的panic是一样的。注意，在 `fork()` 中父到子内存拷贝中也涉及该问题。

处理这种情形：进程从 `sbrk()` 向系统调用（如 `read` 或 `write`）传递有效地址，但尚未分配该地址的内存。阅读内核中 `read` 和 `write` 的代码，观察其使用了什么函数进行用户态和内核态页表的拷贝，在相应的函数中添加对缺页异常进行处理的机制。

使用 `python3 grade-lab-3 lazytests` 和 `python3 grade-lab-3 usertests` 分别进行测试，如果两个测试都通过了，证明你的解决方案是正确的。

### 扩展阅读：Copy On Write Fork

事实上，常见操作系统都会使用COW Fork这个机制。

COW Fork中方法是让父子最初共享所有物理页面，但将它们映射为只读。因此，当子级或父级执行存储指令时，CPU引发缺页异常。为了响应此异常，内核复制了包含异常地址的页面。它在子级的地址空间中映射一个权限为读/写的副本，在父级的地址空间中映射另一个权限为读/写的副本。更新页表后，内核会在导致故障的指令处恢复故障进程的执行。由于内核已经更新了相关的PTE以允许写入，所以错误指令现在将正确执行。

当释放页面时，我们需要对于每一个物理内存页面的引用进行计数，当我们释放虚拟页面时，我们将物理内存页面的引用数减1，如果引用数等于0，那么我们就释放该页面。

有兴趣的同学，可以在xv6中实现COW Fork的功能，这是一个很有挑战性且很有趣的任务。