# 2023/09/11~2023/09/15

## XV6第三章

### 分页硬件

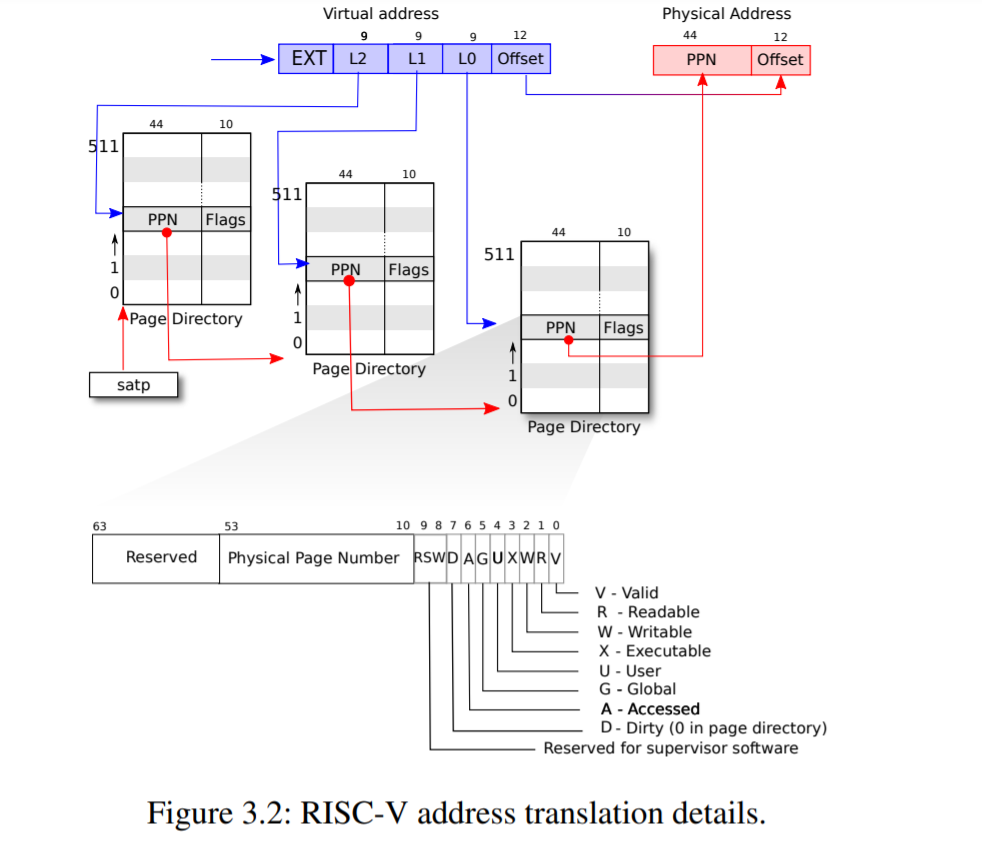
原来很熟悉的页表形式只是一个最简单最理想的的页表形式。

首先，页表并不是只有一张的，对每一个进程都有一个单独的页表，这个页表的首地址存放在CPU中的SATP寄存器中。

当CPU接收到一个虚拟地址时，它将把VA和SATP寄存器的值一同交给MMU，MMU将使用SATP访问位于内存中的页表，将VA翻译成对应的PA。

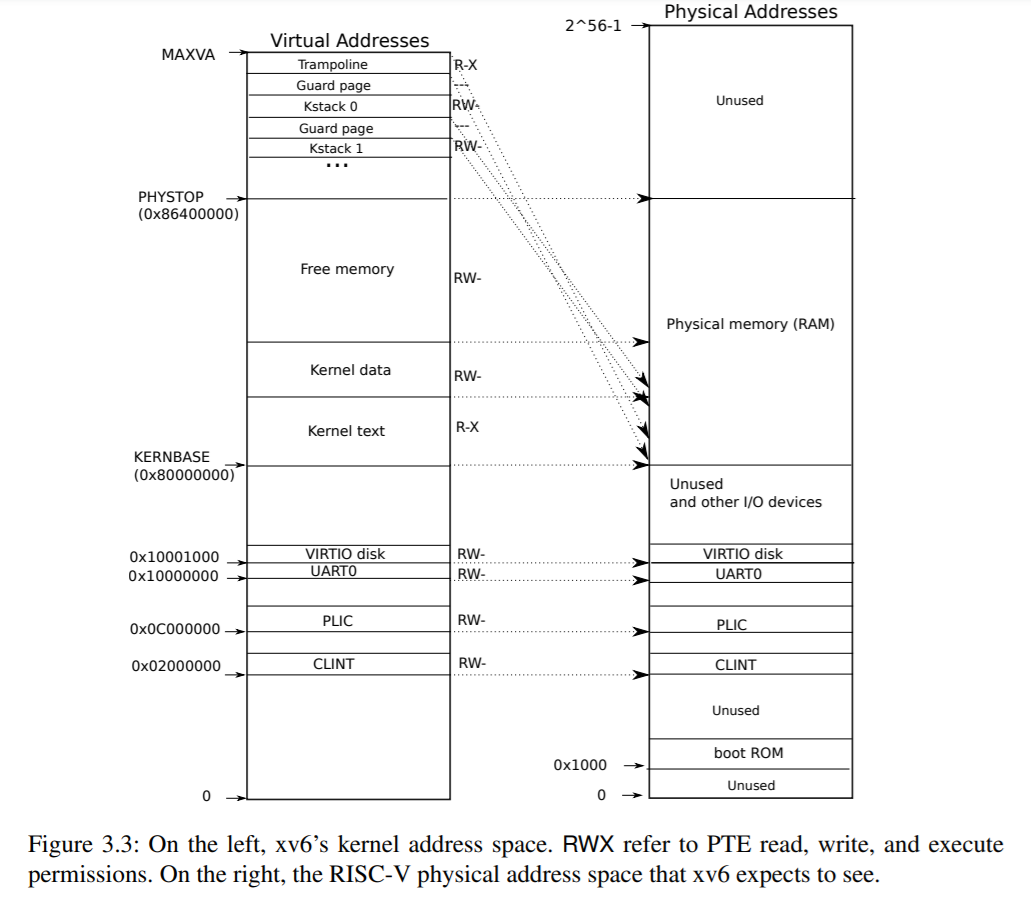
然而，翻译过程也并非是最原始的PTE->PPN->PA的流程。这样的流程有个显而易见的问题，对于27位的PTE而言，它将对应一个大小为2^27\*54(约0.8GB)的页表，这对于宝贵的内存空间来说实在是太多了！并且显而易见地对于每一个进程我们用不到这么多的PTE项！于是就有了页表分级。通过把PTE除了offset位之外的27位拆分成9位一组，第一组的每一个PNN都对应着一个二级页表的首地址，第二组对第三组也是如此，而第三组的每一个PNN则直接在三级页表中查找PA。

这样做的好处在于，总共能够建立的VA->PA总数在长期内和2^27的方式是一样的，但是你并不用一次性全部写入它们！那些使用不到(不存在)的大范围的VA->PA映射所构成的页表将不会被一级页表索引，也就实际上不存在了。



### 内核地址空间

每个进程都包含用户地址空间和一个内核地址空间。内核地址空间布局如下图:



特别的是，内核并不直接具有访问用户空间的物理内存的能力。它需要借用用

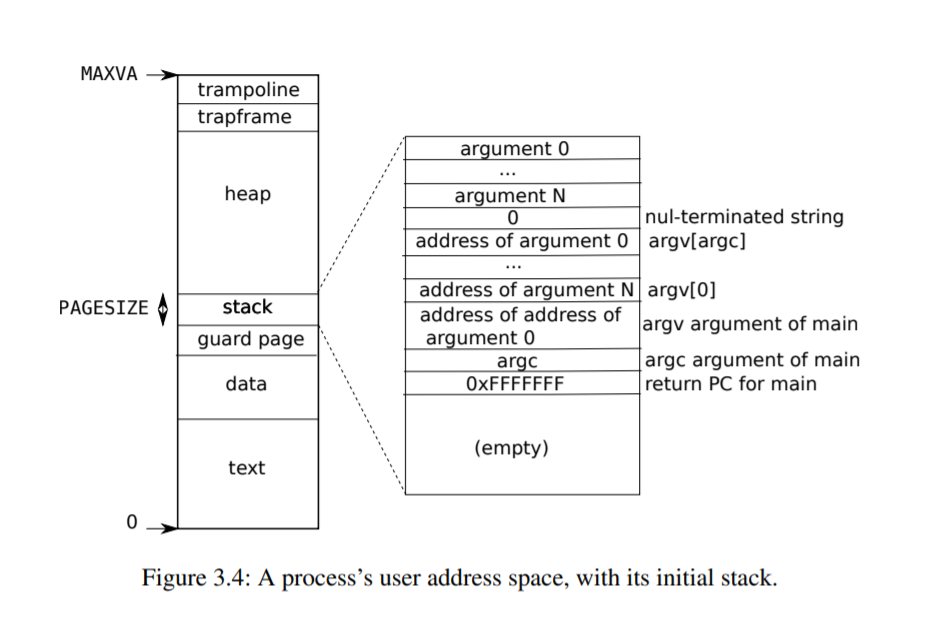
户内存和内核内存的映射+用户内存和物理内存的映射来访问用户的物理内存。

此外，内核栈和trampoline不直接映射。内核栈被映射到高地址处，并在每个内核栈页前添加了守卫页来防止内核栈溢出。trampoline被映射到虚拟地址空间顶端。

### 进程地址空间

进程地址空间即用户地址空间，原则上该地址空间允许进程寻址256GB的内存。

进程要求更多用户内存时，xv6将调用kalloc来分配物理页，并把物理页的PTE添加到进程页表中，并且设置W,X,R,U,V标志位。

进程地址空间布局如下：

### 相关代码详解见文档

内容涵盖vm.c：地址空间的建立（页表建立），kalloc.c：物理地址分配，系统调用sbrk：进程收缩或增长内存，系统调用exec.c：创建用户地址空间的进程调用。

## Lab3 Page tables

### Print a page table

模仿freewalk，宽度优先遍历多级页表并输出信息即可。

void

vmprint(pagetable\_t p,int level){

    if(level==0)printf("page table %p\n",p);

    for(int i = 0; i < 512; i++){

      pte\_t pte = p[i];

      if(pte & PTE\_V){

        uint64 child = PTE2PA(pte);

        if(level==0){

          printf("..%d: pte %p pa %p\n",i,pte,child);

        }

        else if(level==1){

          printf(".. ..%d: pte %p pa %p\n",i,pte,child);

        }

        else{

          printf(".. .. ..%d: pte %p pa %p\n",i,pte,child);

          continue;

        }

        vmprint((pagetable\_t)child,level+1);

      }

  }

}

### A kernel page table each process

缺乏对进程生命周期的了解实在是寸步难行…一边看网上别人的笔记一边自己写……

对于一个进程而言，它的一生大概过程如下(这里是页表实验内容，不深究进程生命周期X细节)：

1. 操作系统初始化时，调用proc.c:***void procinit(void)***初始化内核进程列表，把所有进程标记为未使用，并为其分配va。内核主页表建立时会将每一个进程物理地址映射到对应的va。
2. 建立进程时，调用proc.c:***struct static struct proc\* allocproc(void)***初始化进程，将进程列表中第一个UNUSED进程标记为USED，为其分配pid，trapframe，pagetable,context等PCB内容(PCB的具体形式为proc结构体)。
3. 调用进程时，调用proc.c:***void scheduler(void)***，将进程列表中第一个RUNNABLE进程标记为RUNNING，将CPU当前进程指针指向该进程，并切换上下文。
4. 进程进入阻塞态时,将CPU当前进程指针置空。scheduler重新寻找RUNNABLE进程。
5. 进程销毁时, 调用vm.c:***static void freeproc(struct proc\* p)***通过PCB释放进程占用的资源。（部分资源的释放我暂时没有找到相关函数）。对于占用了对应物理空间的资源，先通过页表释放掉占用的物理内存，再释放页表。

对照上述步骤，可以依次编写为每个进程分配页表所需的代码。

首先是和PCB中的用户地址空间的pagetable一样，为每个进程添加一个单独的kpagetable。在proc.h中修改proc结构体，添加行：

pagetable\_t kpagetable;          // Kernel page table

并且在vm.c定义进程内核页表初始化函数,proc\_kvmmap多了页表参数，原有kvmmap只对主内核页表操作，不再映射CLIENT，CLIENT的存在占用下一实验空间，并且CLIENT仅在系统初始化时使用。

pagetable\_t

proc\_kvminit(void)

{

  pagetable\_t p = (pagetable\_t) kalloc();

  memset(p, 0, PGSIZE);

  // uart registers

  proc\_kvmmap(p,UART0, UART0, PGSIZE, PTE\_R | PTE\_W);

  // virtio mmio disk interface

  proc\_kvmmap(p,VIRTIO0, VIRTIO0, PGSIZE, PTE\_R | PTE\_W);

  // PLIC

  proc\_kvmmap(p,PLIC, PLIC, 0x400000, PTE\_R | PTE\_W);

  // map kernel text executable and read-only.

  proc\_kvmmap(p,KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE\_R | PTE\_X);

  // map kernel data and the physical RAM we'll make use of.

  proc\_kvmmap(p,(uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE\_R | PTE\_W);

  // map the trampoline for trap entry/exit to

  // the highest virtual address in the kernel.

  proc\_kvmmap(p,TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE\_R | PTE\_X);

  return p;

}

在proc.c中将procinit中分配进程内核栈的代码段移动到allocproc，之后每个进程内核页表仅仅持有该进程的内核栈，而主进程页表不再持有任何内核栈。

//An initialized kernel page table for this porccess

  p->kpagetable=proc\_kpagetable();

  if(p->kpagetable == 0){

    freeproc(p);

    release(&p->lock);

    return 0;

  }

  //Alloc a kernel stack for this proccess

  char \*pa = kalloc();

  if(pa == 0)panic("kalloc");

  uint64 va = KSTACK((int) (p - proc));

  kvmmap(p->kpagetable, va, (uint64)pa, PGSIZE, PTE\_R | PTE\_W);

  p->kstack=va;

到此进程的建立过程就结束了，接下来是进程的调度，现在在scheduler函数中，切换进程时需要切换satp指向的页表，当进程执行结束时，调用kvminithart切换主内核页表。

      if(p->state == RUNNABLE) {

        // Switch to chosen process.  It is the process's job

        // to release its lock and then reacquire it

        // before jumping back to us.

        p->state = RUNNING;

        c->proc = p;

        //Switch kernel page table

        sfence\_vma();

        w\_satp(MAKE\_SATP(p->kpagetable));

        sfence\_vma();

        swtch(&c->context, &p->context);

        // Process is done running for now.

        // It should have changed its p->state before coming back.

        kvminithart();

        c->proc = 0;

      }

现在需要考虑进程的销毁，实验提示告诉我们需要实现一个不释放叶子页表的freewalk，实际上这么做是所有进程的内核页表和主内核页表都共同映射的几乎是同一片物理地址空间，对于每个进程物理空间，销毁进程需要释放的仅仅是其内核栈而已。所以需要先用uvmunmap函数释放kstack的物理地址空间及其映射，再调用proc\_freekpagetable来释放进程内核页表。

在freeproc中调用：

  if(p->kstack)

    uvmunmap(p->kpagetable,p->kstack,1,1);

  p->kstack=0;

  if(p->kpagetable)

    proc\_freekpagetable(p->kpagetable);

proc\_freekpagetable的定义如下：

void

proc\_freekpagetable(pagetable\_t pagetable){

  // there are 2^9 = 512 PTEs in a page table.

  for(int i = 0; i < 512; i++){

    pte\_t pte = pagetable[i];

    if((pte & PTE\_V) && (pte & (PTE\_R|PTE\_W|PTE\_X)) == 0){

      // this PTE points to a lower-level page table.

      uint64 child = PTE2PA(pte);

      proc\_freekpagetable((pagetable\_t)child);

      pagetable[i] = 0;

    } else if(pte & PTE\_V){

      pagetable[i]=0;

    }

  }

  kfree((void\*)pagetable);

}

至此，现在每个进程都能够携带自己的内核页表进入内核了。

### Simplify copyin/copyinstr

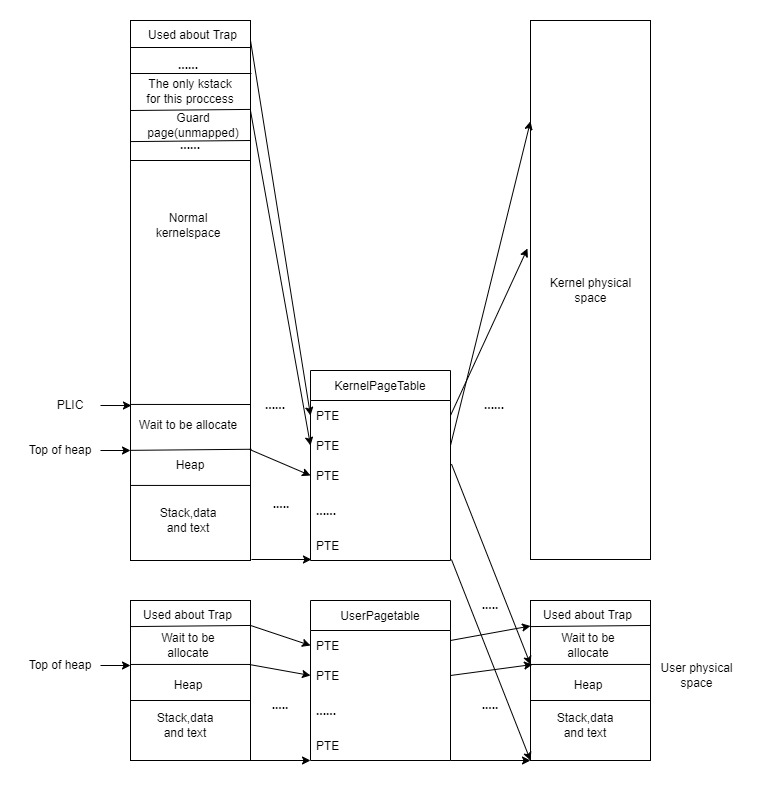
最初没有理解题目什么意思，后来看了英文原文理解了。题目提出的背景是这样的：对于每一个从用户空间传来的指针一类的指向用户物理地址的变量，内核是没有页表来直接分析它的，而是需要借助用户页表来查找对应的pa地址。但实际上在此题目背景下，对于内核而言，它可以直接解引用存在于内核页表中的虚拟地址，从而将其转换为物理地址。

为了达成这一目标，我需要将用户物理地址在内核虚拟地址中进行映射，位置可以选用为内核虚拟地址从0到PLIC的空间。

这一映射在内核虚拟地址和用户虚拟地址中都是从0开始的，所以对于这一映射，二者的页表项是相同的。

所以每次在用户页表改变之后，都对应地改变内核页表即可。需要注意的是，内核页表改变时不对物理地址空间进行操作，因为这一操作在用户页表改变的时候已经进行了。

为此我还绘制了一张图片便于理解：



于是，首先自定义proc\_uvmcopy函数，它和uvmcopy功能类似，但不对物理地址进行操作，同时还增加了oldsz参数，以便于对页表进行增量操作。

uint64

proc\_uvmcopy(pagetable\_t old, pagetable\_t new, uint64 oldsz,uint64 newsz)

{

  pte\_t \*pte;

  uint64 pa, i;

  uint flags;

  for(i = PGROUNDUP(oldsz); i < newsz; i += PGSIZE){

    if((pte = walk(old, i, 0)) == 0)

      panic("proc\_uvmcopy: pte should exist");

    if((\*pte & PTE\_V) == 0)

      panic("proc\_uvmcopy: page not present");

    pa = PTE2PA(\*pte);

    flags = PTE\_FLAGS(\*pte)&(~PTE\_U);

    if(mappages(new, i, PGSIZE, pa, flags) != 0){

      goto err;

    }

  }

  return 0;

 err:

  uvmunmap(new, 0, i / PGSIZE, 0);

  return -1;

}

对应的，定义proc\_uvmunmap函数，和uvmdealloc类似，对页表进行减量操作，同样地，不对物理地址空间进行操作：

uint64

proc\_uvmunmap(pagetable\_t pagetable, uint64 oldsz, uint64 newsz)

{

  if(newsz >= oldsz)

    return oldsz;

  if(PGROUNDUP(newsz) < PGROUNDUP(oldsz)){

    int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;

    uvmunmap(pagetable, PGROUNDUP(newsz), npages, 0);

  }

  return newsz;

}

接下来按照提示在exec，fork，growproc函数中，在每个对用户页表操作的时候都对应地修改进程内核页表：

fork（）中对应复制kpagetable，不同的是，该复制不以p->kpagetable为原本，因为可能在执行这一条时，父进程已经被释放了。

if(proc\_uvmcopy(np->pagetable, np->kpagetable,0, p->sz) < 0){

    freeproc(np);

    release(&np->lock);

    return -1;

  }

growproc（）中增加增量操作后是否越过PLIC的检查，需要时刻注意sz和p->sz的变化。

int

growproc(int n)

{

  uint sz;

  struct proc \*p = myproc();

  sz = p->sz;

  if(sz+n>PLIC){

    return -1;

  }

  if(n > 0){

    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {

      return -1;

    }

    proc\_uvmcopy(p->pagetable,p->kpagetable,p->sz,sz);

  } else if(n < 0){

    sz = uvmdealloc(p->pagetable, sz, sz + n);

    proc\_uvmunmap(p->kpagetable,p->sz,sz);

  }

  p->sz = sz;

  return 0;

}

exec（）中同样增加增量越界检查，在exec交付之前进行进程内核页表拷贝：

越界检查点1：

 // Load program into memory.

  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){

……

    if(ph.vaddr + ph.memsz < ph.vaddr)

      goto bad;

    uint64 sz1;

    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)

      goto bad;

    if(sz1 >= PLIC)

      goto bad;

    sz = sz1;

……

  }

越界检查点2：

  // Allocate two pages at the next page boundary.

  // Use the second as the user stack.

  sz = PGROUNDUP(sz);

  uint64 sz1;

  if((sz1 = uvmalloc(pagetable, sz, sz + 2\*PGSIZE)) == 0)

    goto bad;

  uvmclear(pagetable,sz);

  if(sz1 >= PLIC)

    goto bad;

  sz = sz1;

  sp = sz;

  stackbase = sp - PGSIZE;

交付前拷贝：

  // Save program name for debugging.

  for(last=s=path; \*s; s++)

    if(\*s == '/')

      last = s+1;

  safestrcpy(p->name, last, sizeof(p->name));

  proc\_uvmunmap(p->kpagetable,p->sz,0);

  proc\_uvmcopy(pagetable,p->kpagetable,0,sz);

  // Commit to the user image.

  oldpagetable = p->pagetable;

  p->pagetable = pagetable;

至此所有自定义操作完成，替换原有的copyin和copyinstr，运行测试即可。

## 总结：

如同MIT授课老师所说，在pagetable这一节，细节非常重要，关于内存和页表的操作，每一行代码都要注意检查。

因为对页表进行操作时存在诸多限制，比如：不允许复写，不允许重复置0等。在我的认知中这些限制没有必要，后续在网上找到的的很多答案也手动修改了原页表操作中的panic机制。但既然这些panic其实可以由严谨的页表操作避免，还是尽量保留。

在做lab3之前就听闻lab3是整个xv6最难的部分，名不虚传。

有一个sbrkbasic测试的bug我de了很久，其实我当时隐约意识到是和va的上下取整有关的，奈何没有找到究竟是哪个值不对。最后参考网上的代码才发现是proc\_uvmcopy中的oldsz没有ROUNDUP出的问题。

问题是这样的：对于上下取整而言，影响的都仅仅只是va的offset部分。然而在walk寻址时，使用的是中间27位，并不包括offset。如果oldsz不进行ROUNDUP,walk+PTE2PA会得到ROUNDOWN(oldsz)对应的PA（walk只与中间27位有关,PTE2PA用于移除PTE低10位后再左移12位，得到的PA的offset始终为0）。

这样一来，如果对ROUNDOWN(oldsz)进行操作，很明显会覆盖掉oldsz向下取整所舍去的部分，这也是为什么sbrkbasic测试一直panic：remap。

在fork（）中拷贝用户页表的时候，并不能将已经拷贝好用户页表的父进程内核页表拷贝给子进程内核页表，而是再次将子进程的用户页表拷贝到子进程的内核页表中。如果不做到这一点将无法通过部分fork测试。这个bug也是de了许久都没有结果，最后在网上参考他人代码后解决。理由是父进程也许在这一步拷贝之前就已经退出销毁了。（对fork的一知半解导致我完全无法发现这个问题，我现在也不理解什么条件下能出现这种情况）。

此外，我发现有趣的现象是，内核进程页表似乎并不访问拷贝进来的用户栈。因为无论我将拷贝后的用户栈的GuardPage的PTE\_U设为1还是0，都不影响用户栈越界测试。另外，通过对用户空间创建的代码阅读，我发现用户态和内核态的trap空间也是完全不同的。

我想知道我对上述两个现象的结论推导是否正确，以及这样设计背后的动机。

我还困惑于va的offset段在何时使用，虽然在kvmpa里有对于offset的详细调用，但是我仅仅在virto disk中发现了kvmpa的调用，但显然不止于此，否则offset段的设计实在是过于多余。相同的事情还发生在我调试的过程中，我的调试并没有进入到某一函数，却能够捕捉到该函数的panic信息，我依然无法完整地理解操作系统中代码执行的过程。

做到lab3才知道每个lab都有独立的git分支……以前都是在网上扒下来缺的代码直接用，这一个lab就不行了。并且MIT官方仓库里2020lab似乎分支不开源了，只得去国内找镜像源。途中还遇到了因为qemu版本过高导致xv6初始化阻塞的情况,折腾得够呛。