哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2022 年秋季

课程名称：操作系统

实验名称：页表

实验性质：课内实验

实验时间：10.4 地点：T2507

学生班级：5 班

学生学号：200110513

学生姓名：宗晴

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2022 年 9 月

一、 回答问题

1. 查阅资料，简要阐述页表机制为什么会被发明，它有什么好处？

页表机制是一种内存分配与管理的方案。

在页表机制前，还有连续内存分配（包括固定分区、可变分区等）和分段内存管理等方式。连续内存分配中的固定分区方案会产生大量内部碎片，而可变分区方案会产生大量外部碎片，均会大大影响内存的利用率。而分段内存管理，将程序按照含义分段，分别存储，虽然对程序员友好，但仍然存在内部碎片和外部碎片，导致空间低效。

由此，页表机制便被发明，通过按需供给内存的方式，增大空间利用率。具体来说，将逻辑地址空间和内存分别分割为大小相等的片（页和页框），在分配内存时每次分配一页，按需分配，同时通过页表建立从虚拟地址到物理地址的映射，便于在内存中查找真正用于存储的物理块。

页表机制的好处是它没有外部碎片，并且内部碎片有上界，有效地提高了空间利用率。

2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0x123456789ABCDEF 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

在 SV39 标准下，使用三级页表进行存储。虚拟地址最多为 39 位，其余为保留位。

在虚拟地址的 39 位 0x6789ABCDEF 中，高 9 位 0x19E 为根页表索引，假设根页表的基地址为 0x0000 0000 87f2 6000，可以据此查询到 0x19E 对应的根页表项，假设为 0x0000000021fda801，则第 10-53 位共 44 位表示次页表的物理页帧 0x87f6a，则次页表的基地址为 0x0000000087f6a000；

然后的 9 位 0x04D 为次页表的索引，根据上述次页表的基地址，可以查询到 0x04D 对应的次页表项，假设为 0x0000000021fda401，则第 10-53 位共 44 位表示叶子页表的物理页帧 0x87f69，则叶子页表的基地址为 0x0000000087f69000；

接下来的 9 位 0x0BC 为叶子页表项的索引，根据上述叶子页表的基地址，可以查询到 0x0BC 对应的叶子页表项，假设为 0x0000000021fdac1f，则 10-53 位共 44 位表示虚拟地址所对应物理地址的物理页帧为 0x87f6b，则物理块的起始地址为 0x0000000087f6b000；

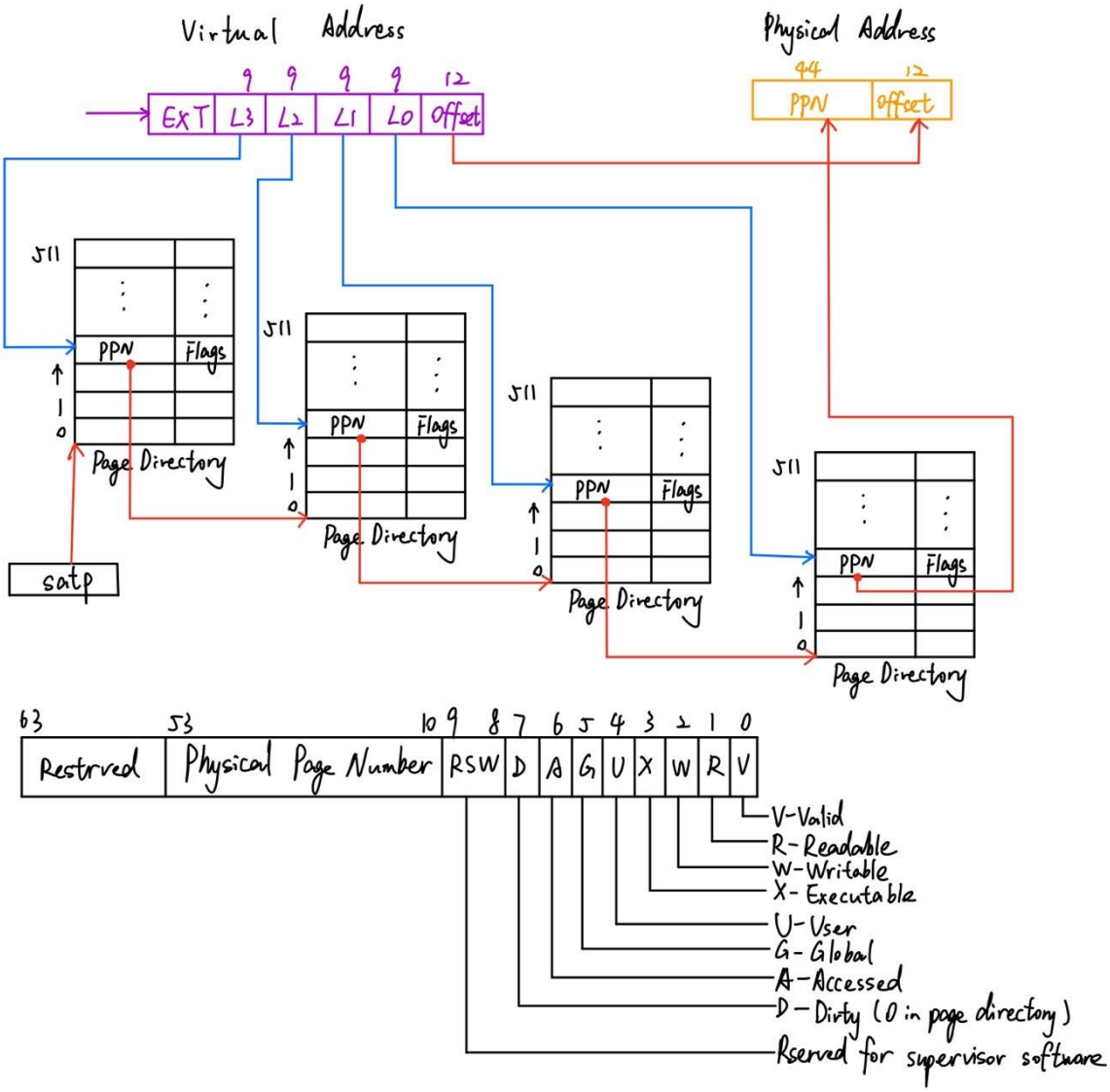
最后 12 位 0xDEF 为页内偏移，根据上述物理块的起始地址，可以用于得到最终的物理地址为 0x0000000087f6bDEF。

3. 我们注意到，虚拟地址的 L2，L1，L0 均为 9 位。这实际上是设计中的必然结果，它们只能是 9 位，不能是 10 位或者是 8 位，你能说出其中的理由吗？（提示：一个页目录的大小必须与页的大小等大）

因为每一页以及每一个页目录的大小均为 4KB，又由于 xv6 是 64 位操作系统，即地址为 64 位，则每个页表项的大小为 8B，所以每一页能存放 4K/8=512 个页表项，这 512 个页表项需要 9 位索引，所以 L0 为 9 位，同理每个次页表和根页表也分别能存放 512 个次页表项和根页表项，均需 9 位索引，所以 L1 和 L2 也是 9 位。

4. 在“实验原理”部分，我们知道 SV39 中的 39 是什么意思。但是其实还有一种标准，叫做 SV48，采用了四级页表而非三级页表，你能模仿“实验原理”部分示意图，画出 SV48 页表的数据结构和翻译的模式图示吗？（SV39 原图请参考指导书）

SV48 页表的数据结构和翻译的模式图示如下所示：



二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写。

1、任务一：打印页表

在此任务中，我们需要实现页表打印的功能。

首先，我们在 kernel/vm.c 中实现 vmprint(pagetable_t pgtbl)函数用于打印所有的有效页表，输入参数 pgtbl 为根页表的物理地址。

由于在此实验中使用了 SV39 标准，即采用了三级页表，因此我们主要需要设计遍历页表的方式。参考函数 freewalk(pagetable_t pagetable)，我们使用递归遍历来打印页表。

由于根页表的输出方式与其它不同，所以我们先打印出根页表的物理地址（十六进制）：

```
printf("page table %p\n", pgtbl);
```

然后定义子函数 vmreprint(pagetable_t pgtbl, int level)用于递归打印其余页表：

```
// 打印页表信息子函数（用于递归调用）
void
vmreprint(pagetable_t pgtbl, int level)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pgtbl[i];
    if(pte & PTE_V){  // 该页表项有效
      printf("||");
      for(int j = 1; j < level; j++) printf(" ||");
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0){  // 不是叶节点
        uint64 child = PTE2PA(pte);
        vmreprint((pagetable_t)child, level + 1);  // 递归打印页表
      }
    }
  }
}
```

如上图，输入参数 pgtbl 为当前待打印的页表的物理地址，level 为当前页表的层数（根页表为 1，次页表为 2，叶子页表为 3）。然后遍历该页表中的全部 512 个页表项 pte，若该页表项有效（通过 pte 与 PTE_V 按位与求得 value 位，该位为 1 表示有效），那么先打印出 level 个“||”，然后打印出该页表项在当前等级页表内的序号（取值范围：0-511）、页表项的内容（十六进制）和这个页表项对应的物理地址（十六进制）。然后判断该页表项是否为叶节点（对于根页表和次页表的目录项，Readable、Writable、Executable 这三位均为 0，通过 pte 与 PTE_R|PTE_W|PTE_X 按位与判断这三位是否都为 0），若不是叶节点，则将该页表项转化为对应的物理地址，然后调用 vmreprint 函数递归打印该页表中的内容，同时传入的参数 level 要增加 1。

然后在刚刚的主打印函数 vmprint(pagetable_t pgtbl)中调用 vmreprint(pgtbl, 1)来递归打印根页表、次页表和叶子页表的目录项，初始时输入参数 level 设置为 1。vmprint(pagetable_t pgtbl)如下：

```
// 打印页表信息
void
vmprint(pagetable_t pgtbl)
{
  printf("page table %p\n", pgtbl);
  vmreprint(pgtbl, 1);
}
```

然后我们在 kernel/defs.h 中定义该页表打印函数的接口：

```
void            vmprint(pagetable_t);  // 添加页表打印函数的声明
```

最后在 kernel/exec.c 的 exec()函数中插入如下语句：

```
// 打印页表信息
if(p->pid==1) vmprint(p->pagetable);
```

插入在 return argc 之前，用于在第一个进程启动时打印页表信息。

2、任务二：独立内核页表

在此实验中，我们需要将共享的内核页表改成独立内核页表，使得每个进程拥有自己独立的内核页表（便于任务三利用独立页表去除软件模拟翻译），但是仍需保留原有的 kvminit()以及 kernel/vm.c 中的全局页表 kernel_pagetable，因为有些时候 CPU 可能并未执行用户进程。

首先，我们在 kernel/proc.h 中的进程控制块数据结构 proc 中，添加 pagetable_t k_pagetable 表示每个进程的独立内核页表，以及 uint64 kstack_pa 表示每个进程的内核栈的物理地址：

```
pagetable_t k_pagetable;      // 每个进程的内核独立页表
uint64 kstack_pa;             // 每个进程的内核栈的物理地址
```

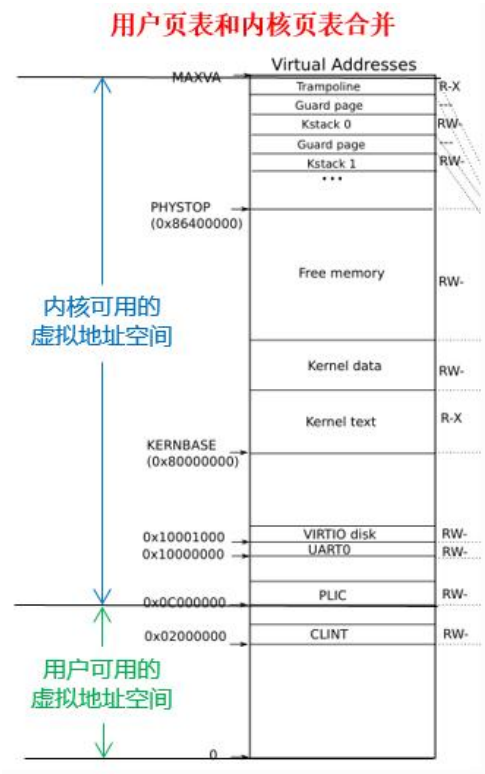
然后，我们参考 kvminit()函数写一个创建独立内核页表的函数 kvminit_new()。为此我们需要参考 kvmmap(uint64 va, uint64 pa, uint64 sz, int perm)函数实现一个新的 kvmmap_new(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm)函数，用于向给定的进程独立内核页表 pagetable 中增加一个虚拟地址到物理地址的映射，与原 kvmmap 函数的不同之处仅在于向 mappages 函数中传入的是给定的进程独立内核页表 pagetable：

```
// add a mapping to the kernel page table for each proc.
// only used when booting.
// does not flush TLB or enable paging.
void
kvmmap_new(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(pagetable, va, sz, pa, perm) != 0)
    panic("kvmmap");
}
```

然后我们实现创建独立内核页表的函数 kvminit_new()。在原有的 kvminit()函数中，实现了 UART0、VIRTIO0、CLINT、PLIC、kernel text、kernel data 和 TRAMPOLINE 的虚拟地址和物理地址在全局内核页表中的映射。新的 kvminit_new()函数中，我们先调用 kalloc()函数给独立内核页表 k_pagetable 分配一页空间，然后调用刚刚新定义的 kvmmap_new 函数将这些映射添加到 k_pagetable 中。最后返回该独立内核页表 k_pagetable 的地址：

```c
/*
 * create a  new direct-map page table for the kernel for each proc.
 */
pagetable_t
kvminit_new()
{
  pagetable_t k_pagetable = (pagetable_t) kalloc();
  memset(k_pagetable, 0, PGSIZE);

  // uart registers
  kvmmap_new(k_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap_new(k_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // 此处不映射CLINT

  // PLIC
  kvmmap_new(k_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap_new(k_pagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmmap_new(k_pagetable, (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.
  kvmmap_new(k_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  return k_pagetable;  // 返回该内核独立页表的地址
}
```

在此函数中，我们不映射 CLINT，否则会在任务三中发生地址重合问题。任务三中要实现的是用户页表和内核页表的合并，虚拟地址空间的 0~0x0C000000 为用户可用，0x0C000000~MAXVA 为内核可用。但是如下图，在 0~0x0C000000 的虚拟地址空间中存储了 CLINT（Core Local Interruptor 本地中断控制器），会产生地址重合，从而产生冲突。事实上，CLINT 只会在内核初始化的时候使用，所以为用户进程生成独立内核页表的时候，可以不必映射这段地址，只需在全局内核页表中保留这段映射即可。



然后需要修改 kernel/proc.c 中的 procinit()函数。该函数是在系统引导时，用于给进程分配内核栈的物理页并在页表建立映射。在此函数中，我们需要把新创建的内核栈的物理地址 pa 拷贝到 PCB 的新增成员 kstack_pa 中，同时通过原有的 kvmmap()函数保留内核栈在全局页表 kernel_pagetable 中的映射：

```
// Allocate a page for the process's kernel stack.
// Map it high in memory, followed by an invalid
// guard page.
char *pa = kalloc();
if(pa == 0)
  panic("kalloc");
uint64 va = KSTACK((int) (p - proc));
kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);  // 保留内核栈在全局页表kernel_pagetable的映射
p->kstack = va;
p->kstack_pa = (uint64)pa;  // 内核栈的物理地址
```

然后在 allocproc()函数中分配内核栈的物理页并在独立内核页表中建立映射，xv6 本会在 procinit()函数中实现该功能，但因为执行 procinit()的时候进程的内核页表还未被创建，所以可以在 procinit()中只保留内存的分配，但在 allocproc()中完成映射。

修改 allocproc()函数，该函数会在系统启动时被第一个进程和 fork 调用。在该函数中，在最后返回 PCB 指针前，我们调用上述新写的 kvminit_new()函数，创建每个进程的独立内核页表，并将该页表的地址保存到 PCB 的 k_pagetable 变量中。若创建失败（没有空闲 PCB，或者内存分配失败），则回收该进程并释放该进程的锁，然后返回 0。否则调用上述新写的 kvmmap_new()函数，将内核栈的虚拟地址 kstack 到物理地址 kstack_pa 的映射保存到独立内核页表中：

```
// 创建每个进程的内核独立页表
p->k_pagetable = kvminit_new();
if(p->k_pagetable == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
// 将kstack到kstack_pa的映射保存到内核独立页表中
kvmmap_new(p->k_pagetable, (uint64)p->kstack, p->kstack_pa, PGSIZE, PTE_R | PTE_W);

return p;
```

然后修改 kernel/proc.c 中的调度器 scheduler()函数，使得切换进程的时候切换内核页表。在找到 RUNNABLE 状态的进程后，在调用 swtch()函数切换至该进程之前，调用 w_satp()函数切换每个进程的独立内核页表，将其放入寄存器 satp 中，然后调用 sfence_vma()函数清空 TLB。在该进程执行完返回到 scheduler()函数中，立即调用 kvminithart()函数切换回全局内核页表：

```
int found = 0;
for(p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  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;

    w_satp(MAKE_SATP(p->k_pagetable));  // 切换每个进程的内核独立页表，将其放入寄存器satp中
    sfence_vma(); // 清空TLB

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

    // 进程切换回调度器后立即切换回全局内核页表
    kvminithart();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0; // cpu dosen't run any process now

    found = 1;
  }
  release(&p->lock);
}
```

同时，若当目前没有进程运行（即 found==0），scheduler() 应该要调用 kvminithart()函数载入全局的内核页表：

```
#if !defined (LAB_FS)
    if(found == 0) {
      kvminithart();  // 当目前没有进程运行时，应satp载入全局的内核页表
      intr_on();
      asm volatile("wfi");
    }
```

然后修改 kernel/proc.c 中的 freeproc()函数释放对应的独立内核页表（用户页表已经由 proc_freepagetable()函数释放），需要释放各级页表但不释放叶子页表指向的物理页帧，因为物理页帧中存储的内核的代码和数据都还是唯一的，各个内核的叶子页表项指向了这些共享的物理页。

为此，参考 proc_freepagetable(pagetable_t pagetable, uint64 sz)函数和 freewalk(pagetable_t pagetable)函数，实现 proc_free_k_pagetable(pagetable_t k_pagetable)函数，用于释放各个进程对应的独立内核页表：

```
// Free a process's kernel page table
void
proc_free_k_pagetable(pagetable_t k_pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = k_pagetable[i];
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
    // 页表项有效，且不是叶子页表，则递归释放其页表项指向的页表
      uint64 child = PTE2PA(pte);
      proc_free_k_pagetable((pagetable_t)child);
      k_pagetable[i] = 0;
    } else if(pte & PTE_V){
    // 页表项有效，且是叶子页表，则不用释放其页表项指向的物理页帧
      k_pagetable[i] = 0;
    }
  }
  kfree((void*)k_pagetable);
}
```

如上图，遍历该页表中的全部 512 个页表项 pte，若该页表项有效（通过 pte 与 PTE_V 按位与求得 value 位，该位为 1 表示有效）且不是叶子页表（对于根页表和次页表的目录项，Readable、Writable、Executable 这三位均为 0，通过 pte 与 PTE_R|PTE_W|PTE_X 按位与判断这三位是否都为 0），那么通过该页表项获取其指向的下一级页表的物理地址，调用 proc_free_k_pagetable()函数对该页表进行递归释放，然后将该页表项置零。若该页表项有效且是叶子页表，则将该页表项置零。遍历结束后，释放该页表的内存。

在 freeproc(struct proc *p)函数的最后，调用该 proc_free_k_pagetable(pagetable_t k_pagetable)函数，释放各个进程对应的独立内核页表，然后将 PCB 中的 k_pagetable 变量置零，需注意 PCB 中的 kstack 变量和 kstack_pa 变量不能置零，因为它们都是在 procinit()中初始化的，而该函数只会在最初执行一次：

```
// 回收内核页表
if(p->k_pagetable)
  proc_free_k_pagetable(p->k_pagetable);
p->k_pagetable = 0;

// kstack和kstack_pa不能释放，因为它们都是在procinit()中被初始化的，只在启动的时候执行
```

最后在 kernel/defs.h 中定义这些新增函数的接口：

+3 kernel/defs.h

```
@@ -92,6 +92,7 @@ int             fork(void);
92  92  int             growproc(int);
93  93  pagetable_t     proc_pagetable(struct proc *);
94  94  void            proc_freepagetable(pagetable_t, uint64);
    95  + void          proc_free_k_pagetable(pagetable_t);
95  96  int             kill(int);
96  97  struct cpu*     mycpu(void);
97  98  struct cpu*     getmycpu(void);
@@ -159,9 +160,11 @@ int             uartgetc(void);
159 160
160 161  // vm.c
161 162  void            kvminit(void);
    163 + pagetable_t    kvminit_new();  // 初始化内核独立页表
162 164  void            kvminithart(void);
163 165  uint64          kvmpa(uint64);
164 166  void            kvmmap(uint64, uint64, uint64, int);
    167 + void          kvmmap_new(pagetable_t, uint64, uint64, uint64, int);  // 为内核独立页表pagetable_t增加映射
165 168  int             mappages(pagetable_t, uint64, uint64, uint64, int);
166 169  pagetable_t     uvmcreate(void);
167 170  void            uvminit(pagetable_t, uchar *, uint);
```

3、任务三：简化软件模拟地址翻译

在该实验中，我们需要在独立内核页表加上用户页表的映射，同时替换 copyin()/copyinstr()为 copyin_new()/copyinstr_new()，使得内核能够不必花费大量时间用软件模拟的方法一步一步遍历页表，而是直接利用硬件像处理内核虚拟地址一样处理用户虚拟地址，提升性能。

为了在独立内核页表加上用户页表的映射，我们让独立内核页表直接共享用户页表的叶子页表，即内核页表中次页表的部分目录项直接指向用户页表的叶子页表。由于用户地址空间的范围为 0x0~0xC000000，且一页大小为 4KB，所以共需要 12*(16^6)/(2^12)=12*(2^12)个叶子页表项就能涵盖整个用户地址空间。又因为一页可以存储 512 个页表项，所以需要 12*(2^12)/512=96 页叶子页表，即需要 96 个次级页表项。所以只需将用户页表的 0 号根目录项所对应的 0 号次级页表中存储的 0~95 号次级页表项复制到独立内核页表中的相同位置即可。

因此，我们在 kernel/vm.c 中实现 vmshare(pagetable_t u_pgtbl, pagetable_t k_pgtbl)函数，用于完成上述复制工作。输入参数分别为用户根页表的基地址 u_pgtbl 和独立内核根页表的基地址 k_pgtbl。首先，我们分别获取用户页表的 0 号根目录项所对应的 0 号次级页表项的物理地址 u_pgtbl_pa 和独立内核页表的 0 号根目录项所对应的 0 号次级页表项的物理地址 k_pgtbl_pa。然后遍历这两个页表中的 0~95 号次级页表项，对每个页表项，首先获取该号内核页表项的地址，然后将该号用户页表项中的内容复制入该地址中：

```c
// 实现内核页表直接共享用户页表的叶子页表
// 经计算，96个次级页表项就能涵盖整个用户地址空间
// (计算过程见实验报告)
// 所以只需将用户页表的96个次级页表项复制到内核页表中
void vmshare(pagetable_t u_pgtbl, pagetable_t k_pgtbl) {
  // 获取二级页表的起始物理地址
  uint64 u_pgtbl_pa = PTE2PA(u_pgtbl[0]);
  uint64 k_pgtbl_pa = PTE2PA(k_pgtbl[0]);

  // 将用户页表的96个次级页表项复制到内核页表中
  pte_t *p;
  for (int i = 0; i < 96; i ++ ) {
    p = &(((pagetable_t)k_pgtbl_pa)[i]);
    *p = ((pagetable_t)u_pgtbl_pa)[i];
  }
}
```

然后用 kernel/vmcopyin.c 中定义的函数 copyin_new()代替函数 copyin()，用函数 copyinstr_new()代替函数 copyinstr()。为此，我们直接在函数 copyin()和函数 copyinstr()内部修改，使它们分别直接转发到函数 copyin_new()和函数 copyinstr_new()。

但如果直接把用户页表的内容复制到内核页表，即其中页表项的 User 位置 1，那么内核依旧无法直接利用硬件访问对应的虚拟地址。为此，我们需要借助 RISC-V 的 sstatus 寄存器，如果该寄存器的 SUM 位（第 18 位）置为 1，那么内核也可以直接访问上述的虚拟地址。所以在调用 copyin_new()/copyinstr_new()之前，需要通过 w_sstatus(r_sstatus() | SSTATUS_SUM)修改 sstatus 寄存器的 SUM 位。由于大多情况下，该位需要置 0。所以在调用 copyin_new()/copyinstr_new()之后，需要通过 w_sstatus(r_sstatus() & ~SSTATUS_SUM)去掉 sstatus 寄存器的 SUM 位。

我们先在 kernel/riscv.h 中定义 SSTATUS_SUM 辅助实现该功能：

```c
#define SSTATUS_SUM (1L << 18) // sstatus寄存器的SUM位
```

然后修改 kernel/vm.c 中的 copyin()函数和 copyinstr()函数：

```c
// Copy from user to kernel.
// Copy len bytes to dst from virtual address srcva in a given page table.
// Return 0 on success, -1 on error.
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  w_sstatus(r_sstatus() | SSTATUS_SUM);
  int r = copyin_new(pagetable, dst, srcva, len);
  w_sstatus(r_sstatus() & ~SSTATUS_SUM);
  return r;
```

```
// Copy a null-terminated string from user to kernel.
// Copy bytes to dst from virtual address srcva in a given page table,
// until a '\0', or max.
// Return 0 on success, -1 on error.
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  w_sstatus(r_sstatus() | SSTATUS_SUM);
  int r = copyinstr_new(pagetable, dst, srcva, max);
  w_sstatus(r_sstatus() & ~SSTATUS_SUM);
  return r;
```

如上所述，先将 sstatus 寄存器的 SUM 位置一，然后各自调用函数 copyin_new()和函数 copyinstr_new()后，将 sstatus 寄存器的 SUM 位置零。

在独立内核页表加上用户页表的映射的时候，每一次用户页表被修改了映射的同时，都要修改对应独立内核页表的相应部分保持同步。所以，需要在 fork()、exec()和 growproc()这三个函数里将改变后的进程页表同步到内核页表中。

首先修改 kernel/proc.c 中的 fork()函数，在释放子进程的锁之前，调用上述新增的 vmshare()函数，在子进程的独立内核页表中加上用户页表的映射：

```
  np->state = RUNNABLE;

  // 内核页表共享用户页表
  vmshare(np->pagetable, np->k_pagetable);

  release(&np->lock);

  return pid;
```

然后修改 kernel/exec.c 中的 exec()函数，在将程序加载进内存时，增加判断防止用户虚拟地址空间进入内核虚拟地址空间的 PLIC 部分：

```
// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  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) // 防止用户虚拟地址空间进入内核虚拟地址空间的PLIC部分
    goto bad;
  sz = sz1;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
```

在打印任务一的页表信息之前，调用上述新增的 vmshare()函数，在该进程的独立内核页表中加上用户页表的映射：

```
// 内核页表共享用户页表
vmshare(p->pagetable, p->k_pagetable);

// 打印页表信息
if(p->pid==1) vmprint(p->pagetable);

return argc; // this ends up in a0, the first argument to main(argc, argv)
```

然后修改 kernel/proc.c 中的 growproc()函数，在返回前调用上述新增的 vmshare()函数，在该进程的独立内核页表中同步用户页表的映射：

```
p->sz = sz;

// 内核页表共享用户页表
vmshare(p->pagetable, p->k_pagetable);

return 0;
```

此外，第一个进程也需要将用户页表映射到内核页表中，因此修改 kernel/proc.c 中的 userinit()函数，在释放锁前，在该进程的独立内核页表中增加用户页表的映射：

```
p->state = RUNNABLE;

// 内核页表共享用户页表
vmshare(p->pagetable, p->k_pagetable);

release(&p->lock);
```

同时，在页表回收的时候需要避免重复回收。为此，需要修改 kernel/proc.c 中的 freeproc()函数，在任务二中增加的回收内核页表的操作之前，先将内核页表中指向用户的叶子页表的次级页表项置零，也即将 0 号根目录项所对应的 0 号次级页表中存储的 0~95 号次级页表项置零，避免重复回收：

```
p->killed = 0;
p->xstate = 0;
p->state = UNUSED;

// 将内核页表的前96项置零，避免重复回收
uint64 k_pgtbl_pa = PTE2PA(p->k_pagetable[0]);
for (int i = 0; i < 96; i++) {
  ((pagetable_t)k_pgtbl_pa)[i] = 0;
}

// 回收内核页表
if(p->k_pagetable)
  proc_free_k_pagetable(p->k_pagetable);
p->k_pagetable = 0;
```

最后，在 kernel/defs.h 中定义这些新增函数的接口：

+5 kernel/defs.h

```
@@ -183,6 +183,11 @@ int             copyin(pagetable_t, char *, uint64, uint64);
183 183  int             copyinstr(pagetable_t, char *, uint64, uint64);
184 184  int             test_pagetable();
185 185  void            vmprint(pagetable_t);  // 添加页表打印函数的声明
    186 + void          vmshare(pagetable_t, pagetable_t); // 内核页表共享用户页表
    187 +
    188 + // vmcopyin.c
    189 + int           copyin_new(pagetable_t, char *, uint64, uint64);
    190 + int           copyinstr_new(pagetable_t, char *, uint64, uint64);
186 191
187 192  // plic.c
188 193  void            plicinit(void);
```

三、 实验结果截图

请填写。

任务一：

```
200110513@comp2:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M
ice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f26000
||0: pte 0x0000000021fc8801 pa 0x0000000087f22000
|| ||0: pte 0x0000000021fc8401 pa 0x0000000087f21000
|| || ||0: pte 0x0000000021fc8c1f pa 0x0000000087f23000
|| || ||1: pte 0x0000000021fc800f pa 0x0000000087f20000
|| || ||2: pte 0x0000000021fc7c1f pa 0x0000000087f1f000
||255: pte 0x0000000021fc9401 pa 0x0000000087f25000
|| ||511: pte 0x0000000021fc9001 pa 0x0000000087f24000
|| || ||510: pte 0x0000000021fdd807 pa 0x0000000087f76000
|| || ||511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

任务二：

kvmtest:

```
$ kvmtest
kvmtest: start
kvmtest: OK
```

usertests:

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3235
            sepc=0x00000000000053fc stval=0x00000000000053fc
usertrap(): unexpected scause 0x000000000000000c pid=3236
            sepc=0x00000000000053fc stval=0x00000000000053fc
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6215
            sepc=0x000000000000201a stval=0x0000000080000000
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6216
            sepc=0x000000000000201a stval=0x0000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6217
            sepc=0x000000000000201a stval=0x0000000800186a0
usertrap(): unexpected scause 0x000000000000000d pid=6218
            sepc=0x000000000000201a stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6219
            sepc=0x000000000000201a stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6220
            sepc=0x000000000000201a stval=0x0000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6221
            sepc=0x000000000000201a stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6222
            sepc=0x000000000000201a stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6223
            sepc=0x000000000000201a stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6224
            sepc=0x000000000000201a stval=0x0000000008006ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6225
            sepc=0x000000000000201a stval=0x0000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6226
            sepc=0x000000000000201a stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6227
            sepc=0x000000000000201a stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000201a stval=0x0000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000201a stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000201a stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000201a stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000201a stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000201a stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000201a stval=0x00000000800e7ef0
usertrap(): unexpected scause 0x000000000000000d pid=6235
            sepc=0x000000000000201a stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6236
            sepc=0x000000000000201a stval=0x0000000080100590
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6237
            sepc=0x000000000000201a stval=0x000000008010c8e0
usertrap(): unexpected scause 0x000000000000000d pid=6238
            sepc=0x000000000000201a stval=0x0000000080118c30
usertrap(): unexpected scause 0x000000000000000d pid=6239
            sepc=0x000000000000201a stval=0x0000000080124f80
usertrap(): unexpected scause 0x000000000000000d pid=6240
            sepc=0x000000000000201a stval=0x00000000801312d0
usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x000000000000201a stval=0x000000008013d620
usertrap(): unexpected scause 0x000000000000000d pid=6242
            sepc=0x000000000000201a stval=0x0000000080149970
usertrap(): unexpected scause 0x000000000000000d pid=6243
            sepc=0x000000000000201a stval=0x0000000080155cc0
usertrap(): unexpected scause 0x000000000000000d pid=6244
            sepc=0x000000000000201a stval=0x0000000080162010
usertrap(): unexpected scause 0x000000000000000d pid=6245
            sepc=0x000000000000201a stval=0x000000008016e360
usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x000000000000201a stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x000000000000201a stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6248
            sepc=0x000000000000201a stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6249
            sepc=0x000000000000201a stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6250
            sepc=0x000000000000201a stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6251
            sepc=0x000000000000201a stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6252
            sepc=0x000000000000201a stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x000000000000201a stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000201a stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6266
            sepc=0x0000000000003e76 stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
```

```
            sepc=0x0000000000003e76 stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6270
            sepc=0x0000000000002188 stval=0x000000000000fbc0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

任务三：

stats stats:

```
$ stats stats
copyin: 27
copyinstr: 10
```

make grade:

```
make[1]: Leaving directory '/home/students/200110513/xv6-labs-2020'
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (4.6s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.7s)
    (Old xv6.out.count failure log removed)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (0.7s)
== Test usertests ==
$ make qemu-gdb
(171.0s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
Score: 100/100
200110513@comp2:~/xv6-labs-2020$
```

四、 实验总结

请总结 xv6 4 个实验的收获，给出对 xv6 实验内容的建议。
注：本节为酌情加分项。

在理论课上我可能会对操作系统的某些函数为什么要这样设计感到很疑惑，也会奇怪为什么某些函数能实现这样的效果，还会想当然地认为某些函数或者数据结构就应该是这么设计的。但通过实验课，我才真正弄明白它们事实上是怎么实现的，以及一些想当然的设计会产生很多问题。xv6 的这 4 个实验让我对理论课所学的知识有了更全面的认识，让我注意到了一些在理论课上一带而过的细节，从而多次对操作系统的一些函数和流程恍然大悟。

第一个实验主要是让我们熟悉 xv6，并且自己编写一些简单的用户程序。在这次实验中，我对管道有了全新的认识，我发现自己在理论课上对于管道的理解只是浮于表面。比如在我最初对于管道的认知中，我以为关闭了管道的读端或写端就阻止了所有进程向这个管道中读或写，但通过实验我发现，事实上我关闭的是某个进程对于这个管道的读端或者写端，其它进程对于这个管道的读写端并没有受到影响。同时，由于 xv6 的文件描述符和进程数量有限，而且管道是一段固定大小的缓冲区，

因此操作系统有时并不能处理非常多进程并发或者大量数据通过管道传输的情况，这在 primes 实验中也有体现。

第二个实验则开始进入了 xv6 的内核，主要是为 xv6 添加一些系统调用。通过这次实验，我终于了解了 xv6 实现系统调用的全过程，以及程序在内核态和用户态之间的切换过程，比如在用户态将系统调用编号存在 a7 寄存器中，将参数存在 a0、a1 等寄存器中，通过 ecall 指令触发软中断陷入内核态，比如通过 syscall 函数分发来自用户的系统调用,比如切换用户页表后通过 sret 指令返回用户态,再通过 ret 指令返回调用函数的位置执行下一条语句等等。还有 copyin 和 copyout 函数让我意识到用户空间与内核空间的分离。

第三个实验是锁机制的应用，主要是重新设计内存分配器和磁盘缓存，从而减少锁征用、提高并行性。在这次实验中，主要的困难是极易碰到死锁的情况。有时通过了 usertests，make grade 也成功了，但再仔细读一遍代码，会发现还是有死锁的可能，又或是这两个测试以较小的概率间歇性失败，这给完成整个实验带来了很大的困难。当然，在考虑了各种死锁情况并多遍完善代码之后，我对死锁的产生条件以及进程之间的各种切换方式所带来的错误有了非常深刻的认识，相信以后遇到死锁的情况应该能比较快速地识别出来。

第四个实验是页表，通过这个实验，我对虚拟地址和物理地址之间的映射关系有了更深刻的体会，也从代码层面了解到了虚拟地址的翻译过程，同时也对程序的各个部分在虚拟地址空间中的分布有了具体的认知。在完成独立内核页表时，指导书上介绍的在独立内核页表中添加映射的方式让我觉得 xv6 的设计非常巧妙，尤其是考虑到 CLINT 只会在内核初始化的时候使用，所以可以不必将这段地址映射到独立内核页表中，这让我认识到 xv6 的所有设计都是经过仔细构思的。

总的来说，xv6 的实验真的让我一遍又一遍地深入理解了操作系统的运作过程，多次对理论课上一知半解的知识恍然大悟。

事实上，目前为止的两次课堂验收给了我很大的压力（戴手环以来它第一次显示我压力偏高...），主要是在准备第一次验收时我没有注意细节，对于这些 linux 指令的理解也都浮于表面，没怎么关注它们的参数或者是返回值的具体含义，也没有对这些指令进行具体的实操，所以导致我在实验课上临时抱佛脚，验收时回答的也是磕磕绊绊勉强才过关。在第一次验收比较糟糕的体验之后，我在准备第二次验收的时候就认真了很多，过了好几遍指导书，也熟练了每次实验的逻辑，考虑了每个实验的边界情况。尤其是实验三的任务二，在回看的时候，我多次完善了代码，考虑了哈希桶和时间戳两种实现方法共用时，会产生的各种死锁情况，以及重复载入相同缓存块的情况，在此过程中，我觉得我对各个进程的调度情况以及相应的产生各种死锁产生的情况有了非常深刻的认识，对于解决死锁问题也更加熟练了。相应的在第二次验收时也比较顺利。在每一遍回看指导书和实验代码的过程中，我都能新领悟到一些东西，所以我觉得课堂验收对于我理解知识起到了非常大的推动力（虽然过程挺折磨的...）。

对于 xv6 实验内容的建议，主要是觉得后面这两次实验还是比较困难的，花的时间也比前两次要多得多，主要原因可能是对于提供的 xv6 的代码框架理解的还不够深入，对于某些文件的作用并不是很清楚。所以，如果可能的话，希望在第一次或第二次实验时，可以带同学们大致了解一下整个代码框架中各个文件的作用以及程序在各个文件之间的跳转过程，或者是可以在原代码框架中多增加一些注释，告知同学们各个文件的功能，再或者只是在每次实验时在指导书上集中地列出相关的代码文件及其功能，这样应该对同学们理解 xv6 的代码框架和完成后续的实验有比

较大的帮助。目前的几个实验只是让我们熟悉了与它们相关的代码内容，但对其它的部分还是理解的不够透彻。当然，这也可能是因为实验课时的限制，让我们无法对 xv6 的每个细节进行详细的展开。