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

实验设计报告

开课学期：2023 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：10.31 地点：T2507

学生班级：8 班

学生学号：210110812

学生姓名：李春阳

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 年 10 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管理与分配的方案。在计算机系统中，内存用于存储程序和数据。当一个程序执行时，它需要将所需的指令和数据加载到内存中进行处理。然而，计算机的内存容量是有限的，无法同时容纳所有的程序和数据。因此，需要一种机制来动态地将内存空间分配给不同的程序，在需要时加载或释放。

页表机制通过将内存分割成固定大小的页（通常是 4KB），并将每个页与程序的虚拟地址关联起来，实现了虚拟内存的概念。当程序访问内存时，页表会根据程序提供的虚拟地址，将其映射到物理内存中的实际地址。如果所需的页不在内存中，操作系统会负责将其从磁盘加载到内存中。

页表机制的好处包括：允许程序使用比物理内存更大的虚拟内存空间，从而允许运行大型程序或多个程序同时运行，提高了系统的灵活性和资源利用率；可以为每个页面设置访问权限，从而保护操作系统和其他程序的内存空间不被非法访问或篡改；可以动态地分配和回收内存空间，确保内存资源的高效利用；可以通过将多个虚拟地址映射到同一物理地址，实现内存共享等等。

2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0xFFFFFFE789ABCDEF 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 标准中，低 39 位代表虚拟地址，其余高位为拓展。这 39 位被划分成了四部分：L2　L1　L0　Offset。虚拟地址 0xFFFFFFE789ABCDEF 的低 39 位转化为二进制为：110011110 001001101 010111100 110111101111。

L2 根页表索引为 110011110，即 0x19E；

L1 次级页表索引为 001001101，即 0x04D；

L0 叶子页表索引为 010111100，即 0x0BC；

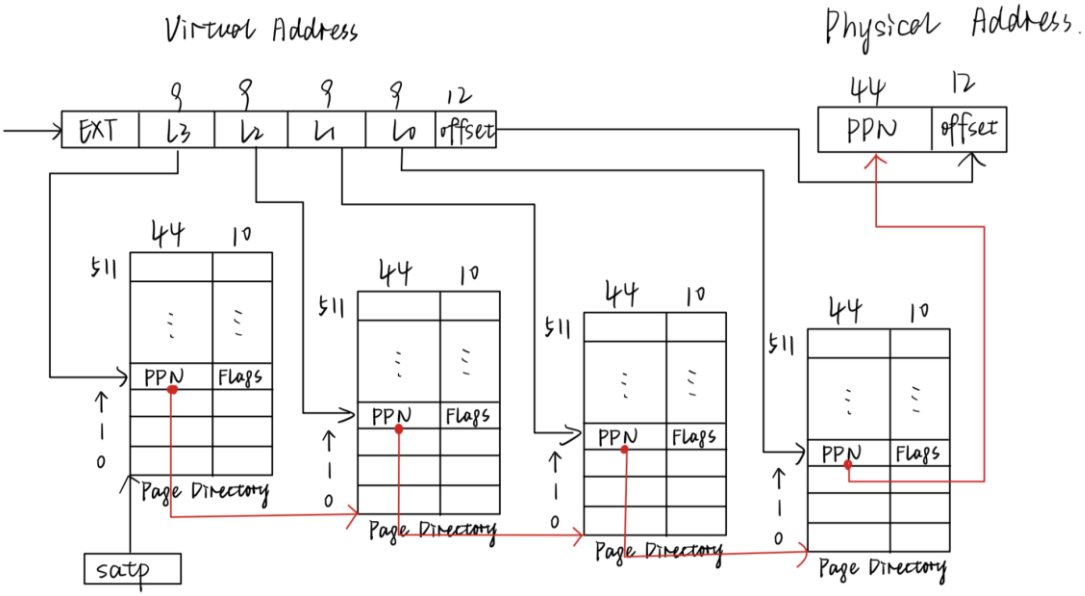
偏移量为 110111101111，即 0xDEF。

我们假设根页表基地址为 0x0000 0000 0000 0000，根据根页表基地址和根页表索引 0x19E 访问根页表的页表项 0x0000 0000 0000 019E，可以得到次级页表基地址 0x1000 0000 0000 0000（假设）。根据次级页表基地址和次级页表索引 0x04D 访问次级页表的页表项 0x1000 0000 0000 004D，可以得到叶子页表基地址 0x2000 0000 0000 0000（假设）。根据叶子页表及地址和叶子页表索引 0x0BC 访问叶子页表的页表项 0x2000 0000 0000 00 BC，可以得到物理地址基址 0x3000 0000 0000 0000（假设）。根据物理地址基址和偏移量 0xDEF，可以得到最终的物理地址 0x3000 0000 0000 0DEF。

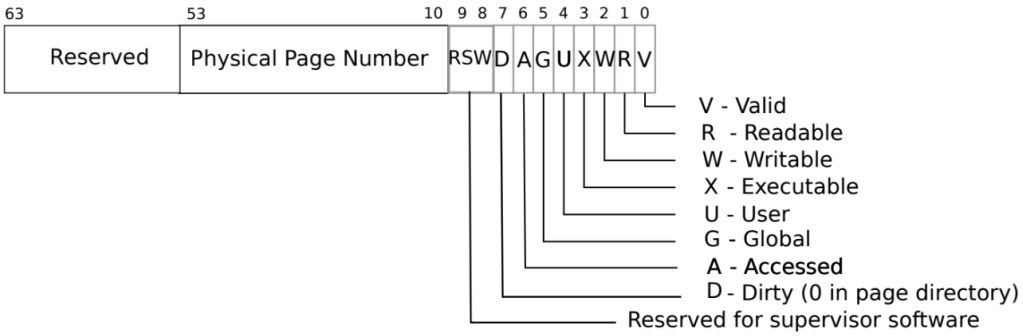
3. 我们注意到，SV39 标准下虚拟地址的 L2, L1, L0 均为 9 位。这实际上是设计中的必然结果，它们只能是 9 位，不能是 10 位或者是 8 位，你能说出其中的理由吗？（提示：一个页目录的大小必须与页的大小等大）

页表机制中，每页以及每个页目录的大小均为 4KB，每个页表项大小为 64bit=8B，因此每一个页及每个页目录中最多可以存放 512 个页表项。索引 512 个页表项共需要 9 位，因此 L0 为 9 位，同理 L1 及 L2 也均为 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）



页表项没有改变：



二、 实验详细设计

任务一：打印页表

1） 定义打印函数：在 exec.c 中的返回 argc 之前插入 vmprint()函数，以输出第一个进程或刚载入程序的页表。当 xv6 启动的时候，它自身会调用 exec() 启动第一个进程 init，这个时候我们的函数会得到输出。

```
101   if(p->pid == 1){
102     vmprint(p->pagetable);
103   }
104   return argc;  // this ends up in a0, the first argument to main(argc, argv)
```

2） 递归打印页表项

由于 RISC-V 的页表被设计成了三层，因此我们需要通过递归的方式遍历页表。但在递归之前，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，并将其放置在递归之外。

```
void vmprint(pagetable_t pgtbl){
  printf("page table %p\n", pgtbl);
  uint64 idx = 0x0000000000000000;
  vmrecursion(pgtbl, 1, idx);
}
```

然后定义子函数 vmrecursion(pagetable_t pgtbl, int depth, uint64 idx)用于递归打印其余页表。其中参数 depth 为当前页表层数（根页表为 1，次页表为 2，叶子页表为 3，递归从 depth=1 开始）。从根页表开始，遍历当前页表中 512 项页表项（pte）。

```
void vmrecursion(pagetable_t pgtbl, int depth, uint64 idx){

  for (int i = 0; i < 512; i++) {
    pte_t pte = pgtbl[i];
```

如果当前页表项有效，则首先打印出“||”。此外，由于 xv6 已给的有效位形式并非我们期待打印出来的形式，因此需要利用字符数组对其进行转换。代码如下：

```
char rwxu[4] = "----";

// 当前页表项有效
if(pte & PTE_V){
  printf("||");

  // 有效位转换打印
  if(pte & PTE_R){
    rwxu[0] = 'r';
  }
  if(pte & PTE_W){
    rwxu[1] = 'w';
  }
  if(pte & PTE_X){
    rwxu[2] = 'x';
  }
  if(pte & PTE_U){
    rwxu[3] = 'u';
  }
```

如果当前页表并非根页表，则仍需打印出 depth-1 个“||”。接着，通过 pte 与 PTE_R|PTE_W|PTE_X 按位与的结果（非叶子页表结果为 0）判断当前页表是否为叶子页表。如果并非叶子页表，则打印出该页表项在当前等级页表内的序号、页表项对应的物理地址（十六进制）及有效位情况，并递归进入下一层页表（depth 需要加 1）。如果是叶子页表，则打印出该页表项在当前等级页表内的序号、页表项的虚拟地址及对应的物理地址（十六进制）和有效位情况。

需要注意的是，由于叶子页表需要打印出虚拟地址，因此我们需要在递归函数 vmrecursion 中额外添加参数 idx 以获取页表项编号转换后得到的虚拟地址。若当前页表项并非叶子页表的目录项，则需要将 idx 加上该页表项序号并左移 9 位，并在退出递归后复位；否则将 idx 加上该页表项序号并左移 12 位，并在打印完信息后复位。

```
for(int j=1; j<depth; j++){
  printf("   ||");
}

//判断当前页表项是否有效且不包含读（PTE_R）、写（PTE_W）和执行（PTE_X）权限
if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
  // 当前页表项是一个指向较低级页表的页表项
  idx = idx + i;
  idx = idx << 9;

  printf("idx: %d: pa: %p, flags: %s\n", i, PTE2PA(pte), rwxu);
  uint64 child = PTE2PA(pte);
  vmrecursion((pagetable_t)child, depth+1, idx);
  idx = idx >> 9;
  idx = idx - i;
} else {
  idx = idx + i;
  idx = idx << 12;
  printf("idx: %d: va: %p -> pa: %p, flags: %s\n", i, idx, PTE2PA(pte), rwxu);
  idx = idx >> 12;
  idx = idx - i;
}
```

3） 最后，在 defs.h 中注册上述函数

```
void            vmprint(pagetable_t);
void            vmrecursion(pagetable_t, int, uint64);
```

任务二：独立的内核页表

在本任务中，我们需要将共享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。

1）修改 kernel/proc.h 中的 struct proc，增加两个新成员：pagetable_t k_pagetable 和 uint64 kstack_pa，分别用于给每个进程中设置一个内核独立页表和内核栈的物理地址。

```
107     pagetable_t k_pagetable;      //内核独立页表
108     uint64 kstack_pa;             //内核栈的物理地址
```

2）仿照 kvminit() 函数重新写一个创建内核页表的函数 vmcreate ()。

在 kvminit() 函数中，kvmmap 函数的作用是向共享内核页表中增加虚拟地址到物理地址的映射。而为了实现为每一个进程创建内核页表的函数 vmcreate ()，我们需要将 kvmmap 函数重写为 kvmmap_new，并传入参数 pagetable_t pagetable，用于向给定的进程独立内核页表 pagetable 中增加虚拟地址到物理地址的映射。

```
void kvmmap_new(pagetable_t k_pagetable, uint64 va, uint64 pa, uint64 sz, int perm) {
  if (mappages(k_pagetable, va, sz, pa, perm) != 0) panic("kvmmap_new");
}
```

然后实现创建独立内核页表的函数 vmcreate ()。利用 kalloc()函数为独立内核页表 k_pagetable 分配一页空间，然后调用 kvmmap_new 函数实现 UART0、VIRTIO0、PLIC、kernel text、kernel data 和 TRAMPOLINE 的虚拟地址和物理地址之间的映射（注意，不要映射 CLINT），并将其地址 k_pagetable 返回。

```
463  pagetable_t vmcreate(){
464    // 为内核页表分配内存PGSIZE
465    pagetable_t k_pagetable = (pagetable_t)kalloc();
466
467    // 设置页表的初始状态是空的
468    memset(k_pagetable, 0, PGSIZE);
469
470    // 将不同的物理地址映射到对应的虚拟地址
471    // 将UART0的物理地址映射到虚拟地址UART0上，并设置该映射为可读可写权限
472    kvmmap_new(k_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
473
474    // virtio mmio disk interface
475    kvmmap_new(k_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
476
477    // // CLINT
478    // kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);
479
480    // 将PLIC的物理地址映射到虚拟地址 PLIC 上，并设置该映射为可读可写权限
481    kvmmap_new(k_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
482
483    // map kernel text executable and read-only.
484    kvmmap_new(k_pagetable, KERNBASE, KERNBASE, (uint64)etext - KERNBASE, PTE_R | PTE_X);
485
486    // map kernel data and the physical RAM we'll make use of.
487    kvmmap_new(k_pagetable, (uint64)etext, (uint64)etext, PHYSTOP - (uint64)etext, PTE_R | PTE_W);
488
489    // map the trampoline for trap entry/exit to
490    // the highest virtual address in the kernel.
491    kvmmap_new(k_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
492
493    return k_pagetable;
494  }
```

3）修改 procinit 函数。procinit()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。我们需要把内核栈的物理地址 pa 拷贝到 PCB 新增的成员 kstack_pa 中，同时不修改 kvmmap 函数，以此保留内核栈在全局页表 kernel_pagetable 的映射。

```
25  void procinit(void) {
26    struct proc *p;
27
28    initlock(&pid_lock, "nextpid");
29    for (p = proc; p < &proc[NPROC]; p++) {
30      initlock(&p->lock, "proc");
31
32      // Allocate a page for the process's kernel stack.
33      // Map it high in memory, followed by an invalid
34      // guard page.
35      char *pa = kalloc();
36      if (pa == 0) panic("kalloc");
37      uint64 va = KSTACK((int)(p - proc));
38      kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
39      p->kstack = va;
40      p->kstack_pa = (uint64)pa;  // 内核栈的物理地址
41    }
42    kvminithart();
43  }
```

4）修改 allocproc 函数。allocproc()会在系统启动时被第一个进程和 fork 调用。在 allocproc 函数里，我们调用 vmcreate()为每个进程设置独立内核页表，并将该页表的地址保存到当前进程控制块的 k_pagetable 变量中。如果创建失败，则回收该进程并释放锁然后返回 0。如果创建成功，则调用 kvmmap_new()函数将设置的内核栈（包括内核栈虚拟地址 kstack 和物理地址 kstack_pa）映射到内核独立页表 k_pagetable 中。

```
115   // 每个进程均设置一个独立的内核页表
116   p->k_pagetable = vmcreate();
117   if (p->k_pagetable == 0) {  // 调用失败
118     freeproc(p);
119     release(&p->lock);
120     return 0;
121   }
122
123   kvmmap_new(p->k_pagetable, p->kstack, p->kstack_pa, PGSIZE, PTE_R | PTE_W);
```

5）修改调度器（scheduler），使得切换进程的时候切换内核页表。如果当前进程正在运行，那么在调用 swtch 切换至该进程的上下文环境之前，调用 w_satp 函数切换至该进程对应的独立内核页表，并将页表所在的物理地址存放在 satp 寄存器中，同时调用 sfence_vma()函数刷新 TLB。最后，在切换上下文完成后，调用 kvminithart()函数恢复至全局内核页表。

```
481           // 切换至该进程对应的独立内核页表
482           w_satp(MAKE_SATP(p->k_pagetable));
483           sfence_vma();
484
485           // 切换上下文
486           swtch(&c->context, &p->context);
487
488           // 恢复至全局内核页表
489           kvminithart();
```

如果无进程正在运行，在 scheduler()函数中需要 satp 载入全局的内核页表 kernel_pagetable。

```
#if !defined(LAB_FS)
    if (found == 0) {
      kvminithart();
      intr_on();
      asm volatile("wfi");
    }
```

6）修改 freeproc()函数来释放对应的内核页表。

设计 kpagetable_free()函数用于释放内核独立页表，从而实现释放页表但不释放叶子页表指向的共享物理页帧。如果当前页表有效且不是叶子页表（非叶子页表 pte 与 PTE_R|PTE_W|PTE_X 按位与的结果为 0），则获取下一层页表的物理地址、递归释放子页表，同时将当前页表项置为 0。如果是叶子页表，则直接将当前页表项置为 0。

```
void kpagetable_free(pagetable_t pagetable){
  for (int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i];
    if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
      uint64 child = PTE2PA(pte);
      // 递归释放子页表及其对应页面
      kpagetable_free((pagetable_t)child);
      // 释放页表项的使用
      pagetable[i] = 0;
    } else if (pte & PTE_V) {
      // 释放叶子页表
      pagetable[i] = 0;
    }
  }
  kfree((void *)pagetable);
}
```

如果当前进程存在独立内核页表，则在 freeproc()函数的尾部调用 kpagetable_free()函数释放该进程独立内核页表所占用的物理内存空间，然后将 k_pagetable 置为 0，表示该进程不再拥有一个有效的独立内核页表。

```
// 释放该进程的独立内核页表
if (p->k_pagetable) kpagetable_free(p->k_pagetable);
p->k_pagetable = 0;
```

任务三：简化软件模拟地址翻译

本任务需要在任务二的基础上，在独立内核页表加上用户地址空间的映射，同时将函数 copyin()/copyinstr() 中的软件模拟地址翻译改成直接访问，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。

1） 新增 sync_pagetable 函数，把进程的用户页表映射到内核页表中。

一种实现方法是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。通过计算，我们可知一共需要 96 个次级页表项，因此使独立内核页表中 0 号根目录所对应的 0 号次级页表中存储的 0~95 号次级页表项一一对应指向用户页表的 0 号根目录项所对应的 0 号次级页表

中存储的 0~95 号次级页表项即可。通过 PTE2PA 分别获取用户页表的 0 号次级页表项的物理地址 user_pa 和独立内核页表的 0 号次级页表项的物理地址 kernel_pa。然后遍历 0-95 号页表项，将 user_pa[i]赋给 kernel_pa[i]。

```
void sync_pagetable(pagetable_t user_pagetable, pagetable_t kernel_pagetable){

 pagetable_t user_pa = (pagetable_t)PTE2PA(user_pagetable[0]);
 pagetable_t kernel_pa = (pagetable_t)PTE2PA(kernel_pagetable[0]);

 for(int i=0; i<0x60; i++){
   // 将用户页表复制到内核页表中
   kernel_pa[i] = user_pa[i];
 }
}
```

2）用函数 copyin_new() 代替 copyin() 。直接在 copyin()里调用函数 copyin_new()，再用 copyinstr_new() 以代替 copyinstr()。但需要注意的是，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访问对应的虚拟地址。对此，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。因此我们在 kernel/riscv.h 中定义 SSTATUS_SUM：

```
#define SSTATUS_SUM (1L << 18)
```

同时，在调用 copyin_new()/copyinstr_new()之前修改 sstatus 寄存器的 SUM 位（即添加 w_sstatus(r_sstatus() | SSTATUS_SUM)代码），在调用 copyin_new()/copyinstr_new()之后去掉 sstatus 寄存器的 SUM 位（即添加 w_sstatus(r_sstatus() & ~SSTATUS_SUM)代码）。

```
int copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len) {

 int ret;
 w_sstatus(r_sstatus() | SSTATUS_SUM);
 ret = copyin_new(pagetable, dst, srcva, len);
 w_sstatus(r_sstatus() & ~SSTATUS_SUM);
 return ret;
```

```
int copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max) {

 int ret;
 w_sstatus(r_sstatus() | SSTATUS_SUM);
 ret = copyinstr_new(pagetable, dst, srcva, max);
 w_sstatus(r_sstatus() & ~SSTATUS_SUM);
 return ret;
```

3）修改对应独立内核页表的相应部分保持同步。在 fork()中调用 sync_pagetable 函数以保持将改变后的进程页表同步。

```
309     np->state = RUNNABLE;
310
311     //将改变后的进程页表同步
312     sync_pagetable(np->pagetable, np->k_pagetable);
313
314     release(&np->lock);
315
316     return pid;
```

4） 在 exec()中调用 sync_pagetable 函数以保持将改变后的进程页表同步。

```
105   //将改变后的进程页表同步
106   sync_pagetable(p->pagetable, p->k_pagetable);
107
108   return argc;  // this ends up in a0, the first argument to main(argc, argv)
109
110 bad:
111   if (pagetable) proc_freepagetable(pagetable, sz);
112   if (ip) {
113     iunlockput(ip);
114     end_op();
115   }
116   //将改变后的进程页表同步
117   sync_pagetable(p->pagetable, p->k_pagetable);
118   return -1;
119 }
```

5）在 growproc ()中调用 sync_pagetable 函数以保持将改变后的进程页表同步。

```
264   p->sz = sz;
265
266   //将改变后的进程页表同步
267   sync_pagetable(p->pagetable, p->k_pagetable);
268
269   return 0;
270 }
```

6）在 userinit()中调用 sync_pagetable 函数以保持将改变后的进程页表同步。

```
243   p->state = RUNNABLE;
244
245   //将改变后的进程页表同步
246   sync_pagetable(p->pagetable, p->k_pagetable);
247
248   release(&p->lock);
249 }
```

7）最后，修改 freeproc()函数。由于页表回收的时候需要避免重复回收，因此在回收独立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 号次级页表中 0-95 号次级页表项置零。

```
// 将内核页表的前96项置零，避免重复回收
pagetable_t pa = (pagetable_t)PTE2PA(p->k_pagetable[0]);
for (int i = 0; i < 0x60; i++) {
  pa[i] = 0;
}
```

三、 实验结果截图

任务一：打印页表

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f26000
||idx: 0: pa: 0x0000000087f22000, flags: ----
||   ||idx: 0: pa: 0x0000000087f21000, flags: ----
||   ||   ||idx: 0: va: 0x0000000000000000 -> pa: 0x0000000087f23000, flags: rwxu
||   ||   ||idx: 1: va: 0x0000000000001000 -> pa: 0x0000000087f20000, flags: rwx-
||   ||   ||idx: 2: va: 0x0000000000002000 -> pa: 0x0000000087f1f000, flags: rwxu
||idx: 255: pa: 0x0000000087f25000, flags: ----
||   ||idx: 511: pa: 0x0000000087f24000, flags: ----
||   ||   ||idx: 510: va: 0x0000003fffffe000 -> pa: 0x0000000087f76000, flags: rw--
||   ||   ||idx: 511: va: 0x0000003ffffff000 -> pa: 0x0000000080007000, flags: r-x-
init: starting sh
```

任务二：独立的内核页表（部分截图）

```
$ kvmtest
kvmtest: start
test_pagetable: 1
kvmtest: OK
$ usertest
exec usertest failed
$ 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 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：简化软件模拟地址翻译

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

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (5.1s)
== 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 (0.8s)
== Test usertests ==
$ make qemu-gdb
(174.7s)
== 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
210110812@comp4:~/xv6-oslab23-hitsz$
```