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

实验设计报告

开课学期：_____

课程名称：_____

实验名称：_____

实验性质：课内实验

实验时间：_____地点：_____

学生班级：_____

学生学号：_____

学生姓名：_____

评阅教师：_____

报告成绩：_____

实验与创新实践教育中心印制

2018 年 12 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？

答：页表机制的发明是为了实现操作系统对内存空间的控制。它的好处在于使操作系统让不同进程各自的地址空间映射到相同的物理内存上，还能够为不同进程的内存提供保护。除此之外，页表还能把不同地址空间的多段内存映射到同一段物理内存（内核部分），在同一地址空间中多次映射同一段物理内存（用户部分的每一页都会映射到内核部分），以及通过一个没有映射的页保护用户栈。

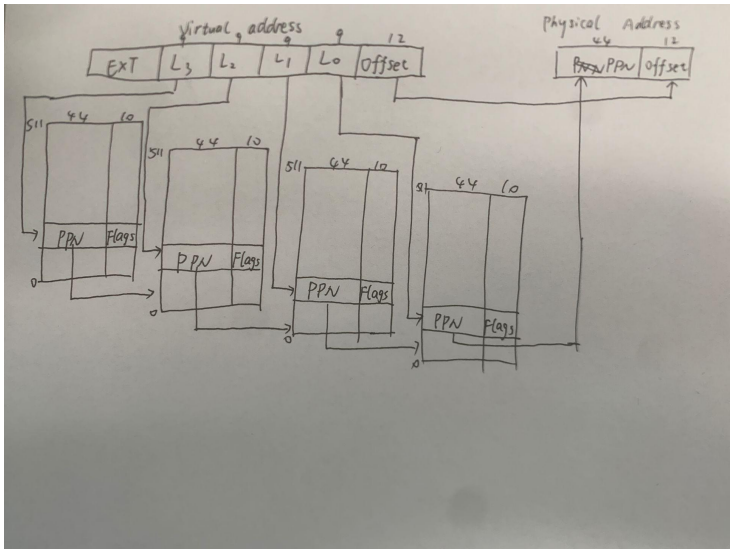
2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0x123456789ABCDEF 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

答：低 12 位（Offset）为 110111101111，
L0 为 010111100，L1 为 001001101，L2 为 110011110.
先看 L2，根据 L2 的索引找到该虚拟地址在根页表中的目录项，再根据该目录项的 PPN 找到对应的次页表；再看 L1，根据 L1 的索引找到该虚拟地址在次页表中的目录项，再根据该目录项的 PPN 找到对应的叶子页表；接着看 L0，根据 L0 的索引找到该虚拟地址在叶子页表中的目录项，该目录项的 PPN 即最终物理地址的 PPN；最后，原虚拟地址的 Offset 即该物理地址的 Offset，二者结合就可以得到最终的物理地址。

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）

答:由于每个页目录项是 8 个字节，每个页面的大小为 4k，因此每一页都是 512 个页目录项，512=2^9 个页目录项需要 9 位的索引，而一个页目录的大小和页的大小等大，由此可以推出 L2,L1,L0 三者的位数必须相等。因此虚拟地址的 L2,L1,L0 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？答：大致如下：



二、 实验详细设计

任务一：打印页表

即参考 freewalk()中遍历页表的过程，实现 vmprint()函数。本任务采用了递归的实现方法，遍历到叶子页表的时候即递归出口。递归的函数以及题目要求的 vmprint()

具体代码大致如下：

```
void pte_and_pa_print(pagetable_t pagetable, int level)
{
  for(int i = 0; i < 512; i++)
  {
    pte_t pte = pagetable[i];
    if(pte & PTE_V)
    {
      uint64 pa = PTE2PA(pte);
      for (int j = 0; j < level; j++) printf("|| ");
      printf("||%d: pte %p pa %p\n", i, pte, pa);
      if (level < 2)
        pte_and_pa_print((pagetable_t)pa, level + 1);
    }
  }
}
void vmprint(pagetable_t pagetable)
{
  printf("page table %p\n", pagetable);
  pte_and_pa_print(pagetable, 0);
}
```

任务二：独立内核页表

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。首先需要给每个进程创建独立内核页表，那么就参考之前初始化共享内核页表的函数 kvminit()，仿照着写一个 proc_k_pagetable()用于给进程创建它的独立内核页表，另外需要添加 proc.h 中结构体 proc 的成员 k_pagetable.

```
pagetable_t proc_k_pagetable()
{
  pagetable_t p = (pagetable_t) kalloc();
  memset(p, 0, PGSIZE);
  if(mappages(p, UART0, PGSIZE, UART0, PTE_R | PTE_W) != 0) return 0;
  if(mappages(p, VIRTIO0, PGSIZE, VIRTIO0, PTE_R | PTE_W) != 0) return
0;
  if(mappages(p, PLIC, 0x400000, PLIC, PTE_R | PTE_W) != 0) return 0;
  if(mappages(p, KERNBASE, (uint64)etext-KERNBASE, KERNBASE, PTE_R |
PTE_X) != 0) return 0;
```

```
  if(mappages(p, (uint64)etext, PHYSTOP-(uint64)etext,
(uint64)etext, PTE_R | PTE_W) != 0) return 0;
  if(mappages(p, TRAMPOLINE, PGSIZE, (uint64)trampoline, PTE_R |
PTE_X) != 0) return 0;
  return p;
}
```

然后我们需要修改 procinit()函数，即在进程初始化的时候需要保留下进程内核栈的物理地址，以便在 allocproc()中把映射到进程的内核页表。procinit()只需要添加一行代码（在申请分配物理地址空间之后），但同样需要事先在 proc 结构体中添加代表内核栈物理地址的成员 pakstack.

```
p->pakstack = (uint64)pa;
```

接着修改 allocproc()，参考其中对于用户页表的创建方式创建内核页表，再调用 mappages()函数完成映射，其中入口参数的 p->pakstack 和 p->kstack 都是结构体成员变量，pakstack 在前面的 procinit 已经初始化为这部分做好了铺垫。映射部分代码：

```
if (mappages(p->k_pagetable, (uint64)p->kstack, PGSIZE,
(uint64)p->pakstack, PTE_R | PTE_W) != 0)
  {
      freeproc(p);
      release(&p->lock);
      return 0;
  }
```

再然后需要修改调度器 scheduler()，使得切换进程的时候切换内核页表。仿照 kvminithart()的页表载入方式，在 swtch()函数的调用之前添加两行代码：

```
        w_satp(MAKE_SATP(p->k_pagetable));
         sfence_vma();
```

另外需要在切换进程部分代码之后，即没有进程运行的情况下才会来到的区域，添加一行载入全局页表的代码，即调用一次 kvminithart()

最后需要修改 freeproc()函数，用于释放进程独立内核页表，释放时参考释放用户页表的写法，但是具体的释放有所不同。因为释放独立内核页表需要满足释放页表但不释放叶子页表指向的物理页帧。这跟 freewalk()的功能如出一辙，只不过可以不需要处理叶子页表。仿照 freewalk()，构建一个自己的释放独立内核页表的函数：

```
void proc_k_freepagetable(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);
      proc_k_freepagetable((pagetable_t)child);
      pagetable[i] = 0;
    }
  }
  kfree((void*)pagetable);
```

```
}
```

值得一提的是，它和 freewalk()唯一的不同就是把 freewalk 里 else 的遍历到叶子页表部分删去了。至此，独立内核页表的创建以及各个功能的正常衔接已经完成。

任务三：简化软件模拟地址翻译

需要写一个函数把进程的用户页表映射到内核页表中。这里采用的是复制页表项的做法，其中复制以后需要把内核页表所有的 User 位都置为 0。复制的过程采用 walk()函数对页表进行遍历：

```
void copypagetable(pagetable_t upagetable, pagetable_t kpagetable,
uint64 sz0, uint64 sz1)
{
  sz0 = PGROUNDUP(sz0);//向上取整
  for (uint64 i = sz0; i < sz1; i += PGSIZE)
  {
    pte_t *pte_u = walk(upagetable, i, 0);
    pte_t *pte_k = walk(kpagetable, i, 1);
    if (pte_u == 0) panic("copypagetable: fail to copy!");
    if (pte_k == 0) panic("copypagetable: fail to paste!");
    uint64 pa = PTE2PA(*pte_u);
    uint flag = (PTE_FLAGS(*pte_u)) & (~PTE_U);
    *pte_k = PA2PTE(pa) | flag;
  }
}
```

其中第一行对齐操作尤为重要，可以避免映射到某个已经映射过的页表中间的位置，发生错误。

然后是按照指导书将 copyin 和 copyinstr 函数分别改成返回 copyin_new 和 copyinstr_new

接着需要修改 fork()、sbrk()和 exec()，其中对 sbrk()的修改只需要修改 growproc().

fork()部分修改只需要在子进程的 size 继承以后，使用刚刚编写的函数将子进程的用户页表映射到内核页表。

growproc()部分修改较为复杂，n>0 时首先需要保证用户空间的大小在 PLIC 部分之下，即需要添加一句条件语句确保 sz+n>PLIC 的情况不发生，然后在 uvmalloc()分配新的内存后，将新增的用户地址空间复制到内核页表；n<0 时需要在 uvmdealloc()之后连通将内核页表的这部分空间解除映射，调用 uvmunmap 函数完成这一操作。

exec()部分修改，需要使用 uvmunmap()清除映射，然后使用第一步编写的函数进行页表的复制。

最后需要修改 userinit()，因为第一个用户进程也需要将用户页表映射到内核页表中。和 fork()类似，也是在进程的 size 确认之后，使用第一步编写的函数将第一个进程的用户页表映射到内核页表。

三、 实验结果截图

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (6.1s)
    (Old xv6.out.pteprint failure log removed)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.8s)
    (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
(195.8s)
== 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
190111005@OSLabExecNode0:~/xv6-labs-2020$
```