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

实验设计报告

开课学期：2021 秋季学期

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：2021/11/4 地点：T2210

学生班级：计科 2 班

学生学号：1190303311

学生姓名：王志军

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

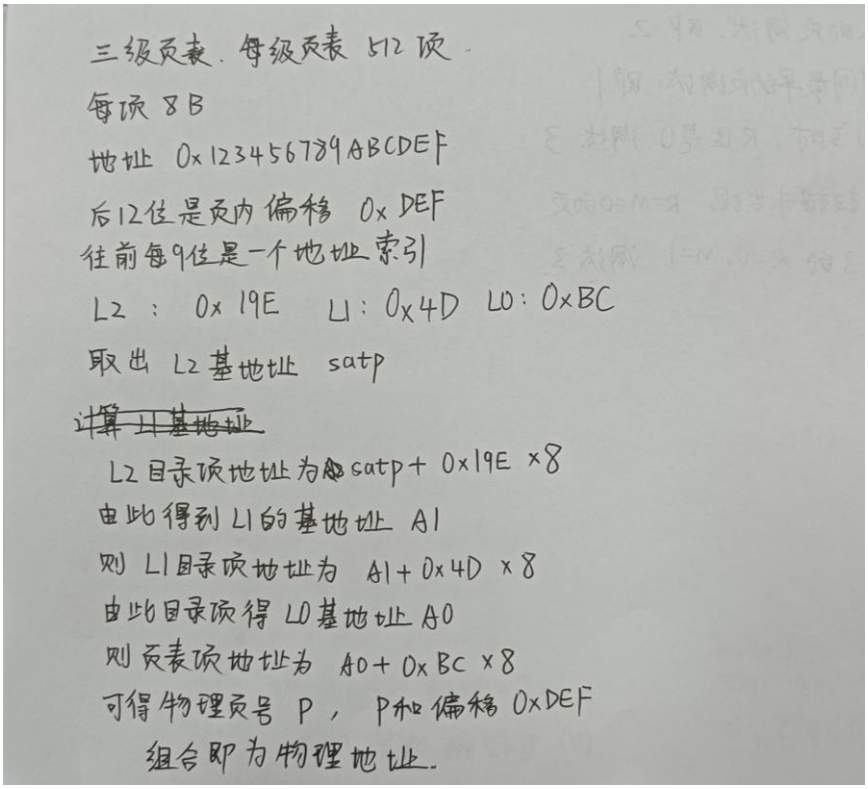
1. 查阅资料，简要阐述页表机制为什么会被发明，它有什么好处？

没有页表机制的情况下，程序编写时需要考虑实际运行在内存中的那些地址，是硬编码，程序之间很可能互相影响，很麻烦也危险；

然后出现了段表，将程序按属性功能分段，用段表来维护物理地址和线性地址之间的映射，对程序员非常友好，也隔离开了物理地址空间，但是程序的堆栈段大小是未知可变的，因此分段会产生很多碎片；

这时页表机制就出现了，将内存分成固定大小的页，既隔离开了物理地址空间和线性地址空间，还减少了内部碎片的产生，需要内存就申请分配页，页表维护虚拟地址到物理地址的映射，也可以让程序员只用关心虚拟地址空间。

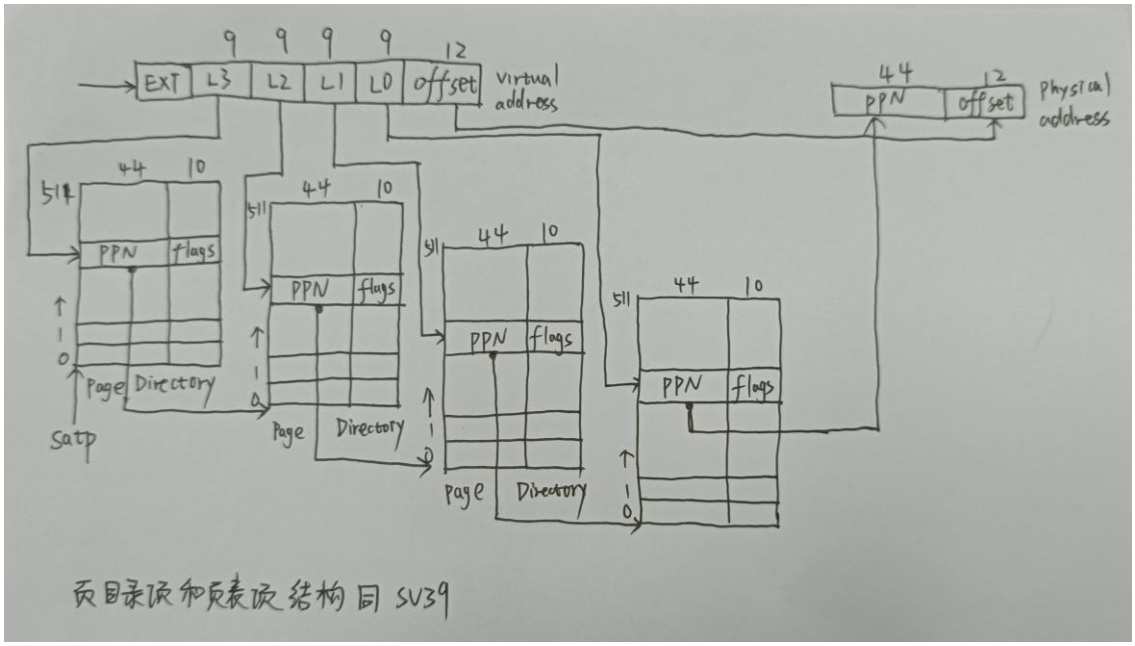
2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0x123456789ABCDEF 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）



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）

一个页的大小为 4K，页表也是要存储在内存中的，因此页表的大小也应该是 4K，而页目录项和页表项的大小都是 8 个字节，因此一个页表只能有 4K/8=512 个条目，也就需要 9 位虚拟地址来索引，因此 L2, L1, L0 都是 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）



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

使用一个中间递归函数 vmprintf，vmprint 调用它打印信息，vmprint 先打印页表的信息，然后调用 vmprintf 函数，这个函数接受页表参数和深度参数，用于打印，判断当前页表项有效时，接着判断深度，根据深度打印出对应个数的 “||“，最后对于叶子打印三个。

```
void vmprint(pagetable_t pgtbl)
{
  printf("page table %p\n", pgtbl);
  vmprintf(pgtbl, 1);
}
```

```
void vmprintf(pagetable_t pgtbl, int deepth)
{
  // 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) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      if(deepth == 1)
        printf("||");
      if(deepth == 2)
        printf("|| ||");
      printf("%d: pte %p pa %p\n", i, pte, child);
      vmprintf((pagetable_t)child, deepth+1);
    } else if(pte & PTE_V){
      printf("|| || ||");
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
    }
  }
}
```

2、任务二：独立内核页表

首先仿照 Kvminit 写一个 Pvminit，函数为每一个进程创建一个内核页表，因此需要 kalloc 一个页表，映射完内核空间之后返回这个页表，此处不能映射 CLINT，防止地址重合：

```
pagetable_t
Pvminit()
{
  pagetable_t pgtbl = (pagetable_t) kalloc();
  memset(pgtbl, 0, PGSIZE);

  // uart registers
  Pvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W, pgtbl);

  // virtio mmio disk interface
  Pvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W, pgtbl);

  // PLIC
  Pvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W, pgtbl);

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

  // map kernel data and the physical RAM we'll make use of.
  Pvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W, pgtbl);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  Pvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X, pgtbl);
  return pgtbl;
}
```

用到了不一样的映射函数 Pvmmap，因为 Kvminit 中用的 Kvmmap 是对全局变量内核页表的映射操作，所以这里写了一个对传入页表进行地址映射的函数 Pvmmap：

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

Procinit，boot 的时候运行，初始化进程的内核栈，并映射到内核页表，此处由于进程独立的内核页表还没有被创建，因此先将这些内核栈的物理地址保存在对应进程的 PCB 当中：

```
      char *pa = kalloc();
      if(pa == 0)
        panic("kalloc");
      p->pa = pa;
```

PCB 中添加了这一字段，还有内核页表的字段：

```
  pagetable_t pagetable;       // User page table
  pagetable_t kpgtbl;          // kernel page table
  char *pa;                    // kernel stack physical address
```

Allcoproc，选择一个为使用的进程在内核中运行，在这个程序中实现独立内核页表的创建，还有内核栈到这个页表的映射，首先通过 Pvminit 创建一个映射好的内核页表，赋值给这个进程的内核页表字段，然后用 Pvmmap 将之前保存在 PCB 中的内核栈物理地址映射到这个页表中：

```
// An empty kernel page table
p->kpgtbl = (pagetable_t)Pvminit();
if(p->kpgtbl == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
uint64 va = KSTACK((int) (p - proc));
Pvmmap(va, (uint64)(p->pa), PGSIZE, PTE_R | PTE_W, p->kpgtbl);


// An empty user page table.
p->pagetable = proc_pagetable(p);
if(p->pagetable == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
```

然后是调度器 scheduler，swtch 是程序阻塞的地方，也就是在这一步进入到实际的用户进程去运行，进程结束后返回，因此在 swtch 之前把进程的内核页表基地址保存在 satp 中，返回之后再恢复为原先内核全局的页表基地址：

```
c->proc = p;
Pvminithart(p->kpgtbl);
swtch(&c->context, &p->context);

// Process is done running for now.
// It should have changed its p->state before coming back.
kvminithart();
```

这里的 Pvminithart 仿照 Kvminithart，接受一个页表，将其地址写入 satp 寄存器：

```
void
Pvminithart(pagetable_t pgtbl)
{
  w_satp(MAKE_SATP(pgtbl));
  sfence_vma();
}
```

最后是 freeproc， 将进程的内核独立页表释放：

```
if(p->kpgtbl)
  Pfreewalk(p->kpgtbl);
p->kpgtbl = 0;
```

Pfreewalk 仿照 freewalk，但是不会把页表项指向的物理地址释放，这里的 if 语句涵义是，当前 pte 有效且不可读写、执行，也就是说这个 pte 是个有下级页表的页目录项，因此递归它的 child，else if 的涵义就是当前 pte 有效但是读写、执行位不全为 0，也就是页表项，此时不能再递归其 child，直接释放这个 pte 即可：

```
// Recursively free page-table pages. lab4 task2
void
Pfreewalk(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);
      Pfreewalk((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      pagetable[i] = 0;
    }
  }
  kfree((void*)pagetable);
}
```

3、*任务三：简化软件模拟地址翻译*

任务三未实现

大致思路：任务三最主要的任务应该是把用户地址空间的页目录项复制到内核页表，函数的结构应该和 uvmcopy 类似，uvmcopy 是用于子进程复制父进程的页表的，做法是先 walk 找到旧页表的一个 pte，得到物理地址后把这这页内容复制到一个新分配的地址，然后再映射到子进程的页表；

这里做法应该可以找到用户进程页表的 pte 之后，得到物理地址，然后直接将其映射到内核页表中，然后在 fork、exec、sbrk 函数中加入这个函数，也就是进行进程的页表项赋值给内核页表，尝试了几次之后，copyin_new 都没过，未能成功；

至于 ppt 里说的优化方法，将内核页表的次级页表直接指向用户页表的 pte，感觉需要修改 mappages 等函数，能力有限。

```c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
```

三、 实验结果截图

请填写

```
make[1]: Leaving directory '/home/guests/1190303311/xv6-labs-2020'
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (7.7s)
== Test count copyin ==
$ make qemu-gdb
count copyin: FAIL (3.1s)
    got:
      0
    expected:
      28
    QEMU output saved to xv6.out.count
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (2.6s)
== Test usertests ==
$ make qemu-gdb
(181.5s)
== 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: 90/100
make: *** [Makefile:317: grade] Error 1
1190303311@OSLabExecNode0:~/xv6-labs-2020$
```