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：11.4 地点：T2507

学生班级：5

学生学号：200110529

学生姓名：梁爽

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？

页表确定了什么是内存地址，以及可访问哪些部分的物理内存。操作系统给每个进程提供了私有的地址空间和内存；分页使得不同的虚拟内存页可以转入同一物理页框。于此同时分页机制可以实现对每个页面的访问控制，采用多级分页的机制来实现虚拟地址，物理地址的转换，能平衡内存使用效率和地址转换效率，提供了一种间接性，映射若干个地址空间到相同的内存，使用未映射的页来保护内核栈和用户栈。

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）

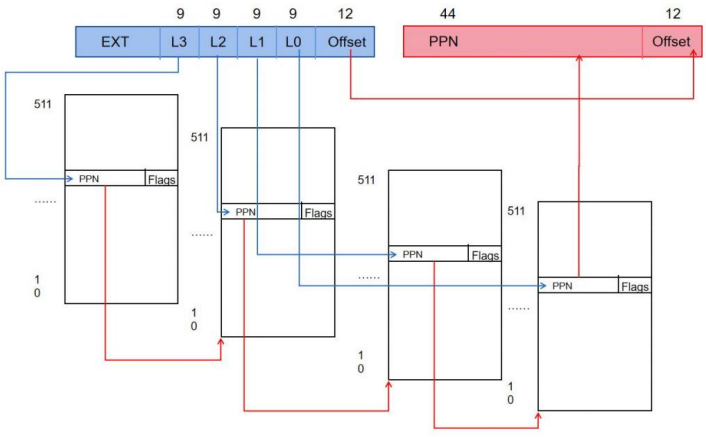
SV39 标准下只有末尾的 39 位用于寻址，其中最末尾的 12 位是一页上的偏移量，另外 27 位可以分成 3 段，每段对应其 PTE 在各级页表上的序号。

satp 寄存器中保存的是一级页表的物理首地址，加上虚拟地址 38-30 位给出的一级页表的偏移量 12'b110011110000，该地址保存的就是一级页表的页表项。每一个页表项的首 10 位无意义，中间 44 位接上 12'b0 代表下一级页表的首地址，最后 10 位是符号位。如此找到二级页表的物理首地址后加上虚拟地址 29-21 位给出的二级页表的偏移量 12'b001001101000,同理获得三级页表的首地址加上虚拟地址 20-12 位给出的三级页表的偏移量 12'b010111100000,得到最终虚拟地址所在页的物理首地址，只需要加上虚拟地址中给出的偏移量 12'b110111101111 就是最终的物理地址。

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）

页表同样用页的方式进行存储，访问时也以页的方式进行管理。所以一个页目录的大小应当与页等大，一个页有 2^12Byte,而一个页表项大小为 2^3Byte，故一个页目录只能容纳 2^9 个页表项，因此虚拟地址中的 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？（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。

实验 1：

在 defs.h 中添加头文件

在 exec.c 中返回之前添加

```
if (p->pid == 1) {
    vmprint(p->pagetable);
}
```

接着利用 vmprint 函数，递归输出信息，利用 level 的深度控制输出信息，并注意打印格式

```
void _vmprint(pagetable_t pagetable, int level) {
  for (int i = 0; i < 512; ++i) {
    pte_t pte = pagetable[i];
    if ((pte & PTE_V)) {
      for (int j = 0; j < level; ++j) {
        if (j == 0) printf("||");
        else printf(" ||");
      }
      uint64 child = PTE2PA(pte); // 通过 pte 映射下一级页表的物理地址
      printf("%d: pte %p pa %p\n", i, pte, child);
      // 查看 flag 是否被设置，若被设置，则为最低一层
      // 只有在页表的最后一级，才可进行读、写、执行
      if ((pte & (PTE_R | PTE_W | PTE_X)) == 0)
        _vmprint((pagetable_t)child, level + 1);
    }
  }
}
```

实验 2：

在 `kernel/proc.h` 中的 `struct proc` 添加新成员变量 `kernel_pagetable`，代表为每一个进程分配的内核页表

在 `kernel/vm.c` 中增加一个 `proc_kvminit()`，逻辑与 `kvminit` 基本一致。同时需要参考 `kvmmap` 实现对应的辅助函数 `ukvmmap`。

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

pagetable_t
```

```
proc_kvminit() {
  // 申请一个页表空间
  pagetable_t proc_kernel_pagetable = (pagetable_t) kalloc();
  if (proc_kernel_pagetable == 0)
    return 0;
  memset(proc_kernel_pagetable, 0, PGSIZE);
  // 与 vminint 内容上保持一致
  ukvmmap(proc_kernel_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);
  ukvmmap(proc_kernel_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  ukvmmap(proc_kernel_pagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  ukvmmap(proc_kernel_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  ukvmmap(proc_kernel_pagetable, KERNBASE, KERNBASE, (uint64)etext-KERNBASE,
PTE_R | PTE_X);
  ukvmmap(proc_kernel_pagetable, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext,
PTE_R | PTE_W);
  ukvmmap(proc_kernel_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R |
PTE_X);
  return proc_kernel_pagetable;
}
```

接着，在 `kernel/proc.c` 中的 `allocproc` 函数中添加调用 `proc_kvminit()`的代码段，以便在初始化进程空间时初始化用户内核页表。然后参考 `kernel/proc.c` 中的 `procinit()`中代码，为每个内核页表初始化内核栈。同时注释原本初始化内核栈的代码。

下一步，需要确保在切换进程是能够将对应进程的用户内核页表的地址载入 SATP 寄存器中，所以要在 kernel/proc.c 的 scheduler 函数中进行修改。修改方式参考 kernel/vm.c kvminithart 函数。根据要求，需要在每个任务执行结束后切换回 kernel_pagetable。

```
// 在切换任务前，将用户内核页表替换到 stap 寄存器中
w_satp(MAKE_SATP(p->kernel_pagetable));
// 清除快表缓存
sfence_vma();
// 调度，执行进程
swtch(&c->context, &p->context);

// Process is done running for now.
// It should have changed its p->state before coming back.
// 该进程执行结束后，将 SATP 寄存器的值设置为全局内核页表地址
kvminithart();
```

下一步我们需要考虑在销毁进程时释放对应的内核页表。在释放内核页表前需要先释放进程对应的内核栈空间。在 `kernel/proc.c` `freeproc` 中进行修改

```
// 删除内核栈
if (p->kstack) {
  // 通过页表地址， kstack虚拟地址 找到最后一级的页表项
  pte_t* pte = walk(p->kernel_pagetable, p->kstack, 0);
  if (pte == 0)
    panic("freeproc : kstack");
  // 删除页表项对应的物理地址
  kfree((void*)PTE2PA(*pte));
}
if(p->pagetable)
  proc_freepagetable(p->pagetable, p->sz);

// 删除kernel pagetable
if (p->kernel_pagetable)
  proc_freekernelpagetable(p->kernel_pagetable);
```

Proc_freekernelpagetable 函数是模仿 proc_freepagetable 完成的

```
void
proc_freekernelpagetable(pagetable_t pagetable){
  for (int i = 0; i < 512; ++i) {
    pte_t pte = pagetable[i];
    if ((pte & PTE_V)) {
      pagetable[i] = 0;
      if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
        uint64 child = PTE2PA(pte);
        proc_freekernelpagetable((pagetable_t)child);
      }
    } else if (pte & PTE_V) {
      panic("proc free kernelpagetable : leaf");
    }
  }
  kfree((void*)pagetable);
}
```

最后，修改 `kernel/vm.c` 中 `kvmpa`，将 `walk` 函数使用的全局内核页表地址换成进程自己的内核页表地址。

```
pte = walk(myproc()->kernel_pagetable, va, 0);
if(pte == 0)
  panic("kvmpa");
```

实验 3：

首先，重写 `copyin` 和 `copyinstr` 函数，用 `copyin_new` 和 `copyinstr_new` 进行替换，其相关实现已经在 `vmprint.c` 中完成。在 `kernel/defs.h` 中添加对于函数声明。

接着，我们要将用户页表的变化同步到用户进程内核页表中，则需要实现映射和缩减两个操作。模仿 uvmcopy 和 uvmdealloc 的实现，在 `kernel/vm.c` 中添加 `kvmcopymappings` 和 `kvmdealloc` 函数

根据提示，用户页表在用户内核页表中的映射范围为[0，PLIC]，但是从 xv6 book 中可以看到，全局内核页表的定义中在[0，PLIC]之间存在一个 CLINT 核心本地中断，CLINT 仅在内核启动时使用，所以用户进程内核页表中无需再存在 CLINT，所以我们将 proc_kvminit()中 CLINT 映射的部分注释掉。防止再映射用户页表时出现 remap

根据提示，需要在 `fork()`,`sbrk()`,`exec()`中进行修改，需要在这些函数改变用户进程页表后改变用户内核页表。

// Copy user memory from parent to child. （调用 kvmcopymappings，将**新进程**用户页表映射拷贝一份到新进程内核页表中）

```
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0 ||
      kvmcopymappings(np->pagetable, np->kernelpgtbl, 0, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;
```

在 `sysproc.c` 中的 `sys_sbrk()`中可以发现，执行内存相关的函数为 `growproc()`，所以我们对 `growproc()`进行修改。需要注意的是，用户页表的扩大和缩小都需要进行同步。

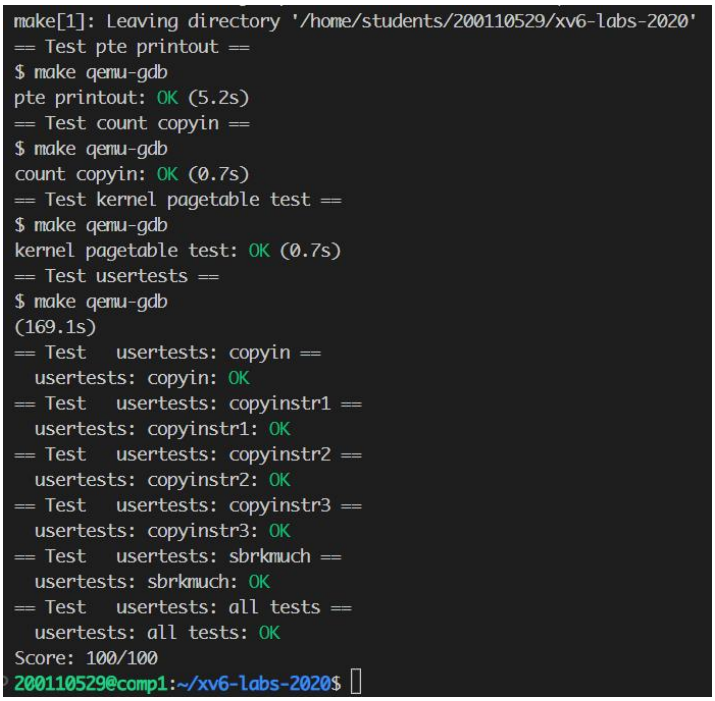
修改 `exec()`，在映射之前要先检测程序大小是否超过 PLIC，防止 remap，同时映射前要先清除[0，PLIC]中原本的内容，在将要执行的程序映射到[0，PLIC]中。

然后，根据提示，需要在 `userinit` 的内核页表中包含第一个进程的用户页表

```
kvmcopymappings(p->pagetable, p->kernelpgtbl, 0, p->sz); // 同步程序内存映射到进程内核页表中
```

三、 实验结果截图

请填写。



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。

前三个实验难度都不算大，第一个实验入门非常实用，做起来也很快，第二个实验系统调用，帮助理解了系统在内核下的工作流程和调用过程，第三个锁机制，不仅有自旋锁和睡眠锁，还要时时刻刻考虑死锁的情况发生，写起来时间就长了很多，细节很多。第四个实验做起来是非常的折磨，感觉学校给的提示模块太分散，有的在原理里面，有的在实现里面，提示的东西都是很有用的，但是实现起来依旧很麻烦，考察模仿能力和细节能力更是难上加难。大部分函数，uvminit，uvmcopy 用户态的实现，我们在实现 kvminit 内核态的时候可以参考，可以模仿原来的实现给出，但是最后的实现细节却很难复刻，某些步骤在网上参考了别人的思路才恍然大悟。操作系统实验难度感觉非常大，基本上全靠摸黑自学，有些时候走进死胡同只能看看别人怎么实现。