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：11.2 地点：T2608

学生班级：20 级 8 班

学生学号：200210231

学生姓名：王木一

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？

页表机制为了解决内存分配中的碎片问题。

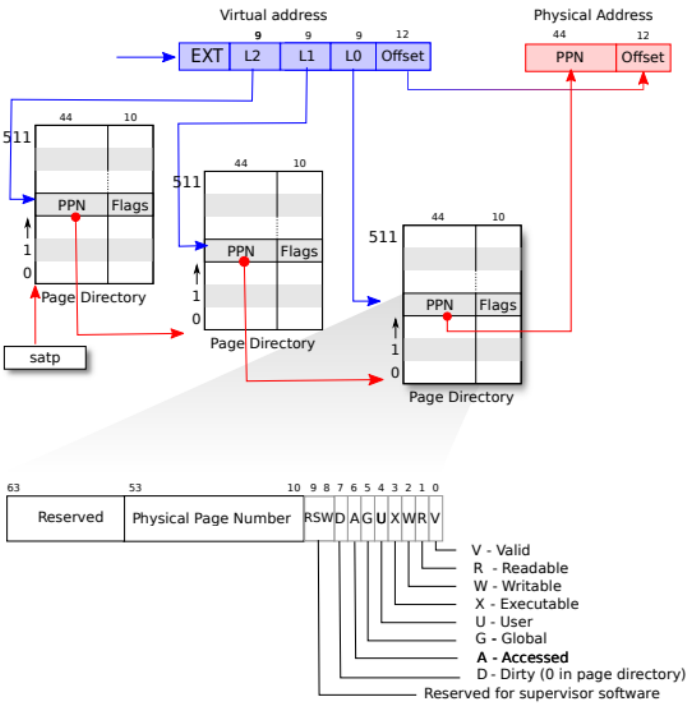
好处：

① 解决外部碎片问题，同时内部碎片的大小也可以接受。

② 使得每个进程独立，让每个进程都有 0-maxva 的虚拟地址空间。（内存管理对用户透明）

③ 实现用户进程之间、用户与内核之间的保护。避免一个进程访问其他进程的代码和数据。还可以通过设置标志位，控制访问权限。

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）



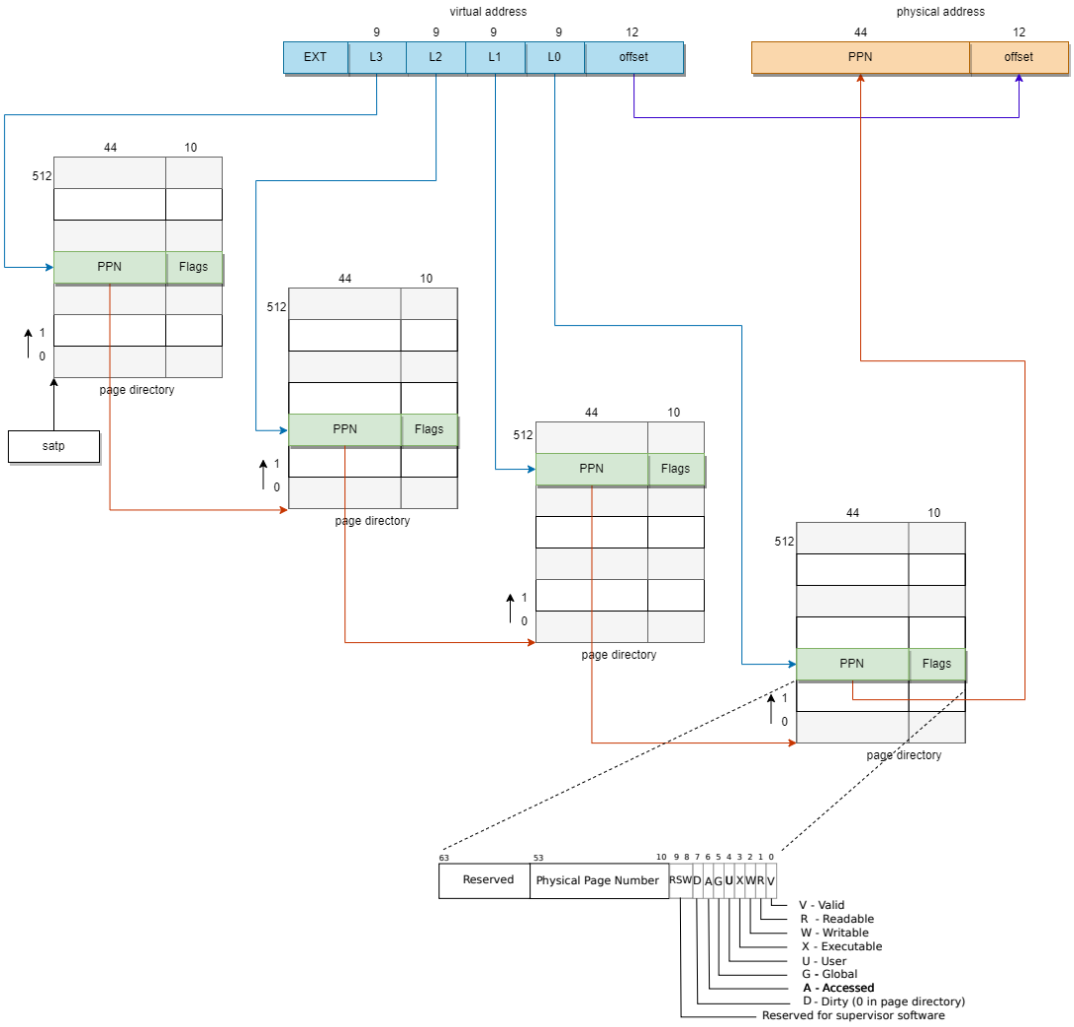
1. Satp 保存根页表的物理页帧号，将其转化为物理地址，找到根页表基址。通过虚拟地址的 L2 索引((unsigned)0x123456789ABCDEF >> 30) & 0x1FF = 0x19E = 414,找到根页表的目录项，其保存有次级页表的物理页帧号，将物理页帧号转化为次级页表的物理地址 pa_L1
2. 根据 pa_L1，通过虚拟地址的 L1 索引((unsigned)0x123456789ABCDEF >> 21) & 0x1FF = 0x4D = 77,找到次级页表的目录项，其保存有叶子页表的物理页帧号，将物理页帧号转化为叶子页表的物理地址 pa_L0
3. 根据 pa_L0，通过虚拟地址的 L0 索引((unsigned)0x123456789ABCDEF >> 12) & 0x1FF = 0xBC = 188,找到叶子页表的目录项，其保存有物理地址的物理页帧号，将物理页帧号转化为物理地址 pa

4. offset=0xDEF， pa+offset 即为虚拟地址 0x123456789ABCDEF 的对应的物理地址。

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）

Xv6 中一个物理页的大小为 4KB.通常拿一个物理页来装页表本身，同时页表项大小为 8B，那么一个页表中有 4KB/8B = 512 = 2^9 个 PTE，故虚拟地址 L2,L1,L0 都为 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。

1. 任务 1：打印页表

要求设计一个 `void vmprint(pagetable_t pgtbl)`函数，来打印页表的有效页表项。
实现：设计一个递归调用函数 `void _vmprint(pagetable_t pgtbl, int level)`
level 表示当前递归的深度，便于打印对应数量的的竖线“||”。
从页表索引 0-511 遍历，若有效则打印其内容。同时若不是叶子页表项，就递归调用自身，打印下一级页表。

```
//vmprint base
void
_vmprint(pagetable_t pgtbl, int level)
{
  pte_t *pte;
  int index;
  uint64 pa;

  for(pte = pgtbl, index = 0; pte < &pgtbl[PGSIZE/8]; pte++, index++){
    if(*pte & PTE_V){
      pa = PTE2PA(*pte);
      switch(level){
        case 1 :
          printf("||%d: pte %p pa %p\n", index, *pte, pa);
          break;
        case 2 :
          printf("|| ||%d: pte %p pa %p\n", index, *pte, pa);
          break;
        case 3 :
          printf("|| || ||%d: pte %p pa %p\n", index, *pte, pa);
      }
      if(level != 3){
        _vmprint((pagetable_t)(pa), level+1);
      }
    }
  }
}
```

2. 任务 2：内核独立页表

要求：实现为每个进程设置一个独立的内核页表，完成这个内核独立页表整个生命周期（准备、创建、使用、释放）的代码。（添加用户内存的映射、简化 copy 将在任务 3 完成）

1. 在 PCB 中增加两个新成员

- pagetable_t k_pagetable 此进程内核独立页表的物理地址
- uint64 kstack_pa 此进程内核栈的物理地址

2. 创建内核独立页表的函数

模仿 kvminit()函数，写一个 kuvinit()函数，但不映射 CLINT，同时需要返回创建好的内核独立页表。模仿 kvmmap()写一个建立内核独立页表映射的函数 kuvmmap()

```
// add a mapping to the private (aka process(user)-specific)
kernel page table.
// only used when booting.
// does not flush TLB or enable paging.
void
kuvmmap(uint64 va, uint64 pa, uint64 sz, int perm, pagetable_t
k_pagetable)
{
  if(mappages(k_pagetable, va, sz, pa, perm) != 0)
    panic("kuvmmap");
}

// create a process(user)-specific direct-map kernel pagetable
pagetable_t
kuvminit(void)
{
  pagetable_t k_pagetable;
  if((k_pagetable = (pagetable_t) kalloc()) == 0){
    // return k_pagetable
    panic("kuvminit");
  }
  memset(k_pagetable, 0, PGSIZE);

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

...
其他映射在这里省略
```

```
...
  return k_pagetable;
}
```

3. **修改 procinit()**

   当只有一个全局内核页表时，IO 设备和 trampline 在 kvminit()中完成了映射。用户的内核栈虚实地址映射在 procinit()中添加到全局内核页表中。

   当每个进程都有自己独立的内核页表时，IO 设备和 trampline 在 kuvminit()中完成了映射。而用户的内核栈虚实地址映射在 procinit()中**不能添加**到内核独立页表中，因为此时**进程的独立内核页表还没创建**，我们转而将用户内核栈物理地址保存到 kstack_pa 中，待独立内核页表创建时再建立内核栈虚实地址映射。

   实现：这里只需添加一行保存内核栈物理地址的代码。

4. **修改 allocproc()**

   每个进程创建前都需要调用 allocproc()以获得一个 PCB，我们在此时创建此进程对应的内核独立页表。

   实现：当找到一个空闲 PCB 后，调用 kuvminit()创建内核独立页表，再添加此进程内核栈在此页表中的映射。

```
// create proc's private k/u page table
  if((p->k_pagetable = kuvminit()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  kuvmmap(p->kstack, p->kstack_pa, PGSIZE, PTE_R | PTE_W,
p->k_pagetable);
```

5. **修改 scheduler()**

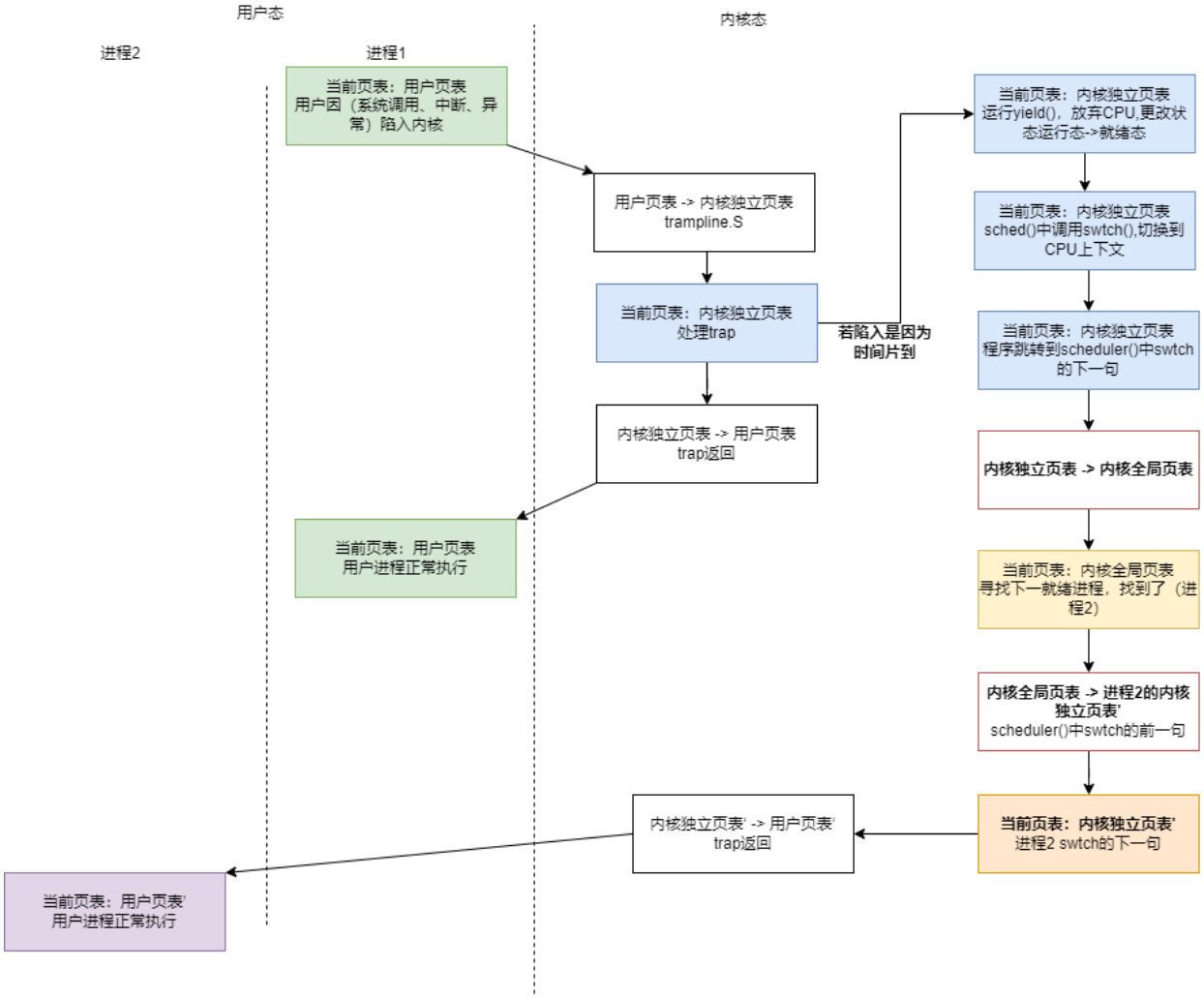
   Scheduler 是 xv6 中的进程调度器，要让我们的独立内核页表起作用，就要在这里切换。为何在这里切换页表，就需要看进程陷入内核再返回的过程。

   Xv6 页表切换：

   1. 进程处于用户态时使用自己的用户页表
   2. 进程从用户态陷入内核态时（系统调用、中断、异常），在 trampline.S 中将切换为内核页表，这一步是靠将 p->trampline->kernel_satp 中内核页表保存在 satp 寄存器中实现.
      (在本实验前这个内核页表就是全局内核页表，此实验中的内核页表就是此进程的内核独立页表)
   3. 从内核态返回用户态时，在 usertrapret()中将当前 satp 寄存器值保存到 p->trampline->kernel_satp 中。再在 trampline.S 中切换回用户页表。

4. 当因为时间片到陷入内核，放弃 CPU 后（CPU 此时**没运行任何进程**），CPU 运行 sheduler 函数选择下一个就绪的进程，**此时的内核应使用全局内核页表**
5. 当 sheduler 选择好下一个要执行的进程后，就要切换为此**进程对应的内核独立页表**

将上面的流程总结为下面的一张图（不同颜色表示不同页表）：



要做的**只有上图红框中的切换**，其他地方的页表切换 xv6 已经帮我们做好了。**只需在 sheduler()的 swtch 上下切换即可**

```
// switch to k_pagetable
w_satp(MAKE_SATP(p->k_pagetable));
sfence_vma();

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

// switch to global kernel_pagetable
w_satp(MAKE_SATP(kernel_pagetable));
sfence_vma();
```

6. **释放独立内核页表**

因为各个内核独立页表都指向同一物理内核代码，删除时只用清除 PTE 和删除内核独立页表本身。**模仿 freewalk 即可。**

```
// Recursively free page-table pages.(also delete page table itself)
void
freewalkelse(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){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalkelse((pagetable_t)child);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      pagetable[i] = 0;
    }
  }
  kfree((void*)pagetable);
}
```

3. **任务 3：简化软件模拟地址翻译**

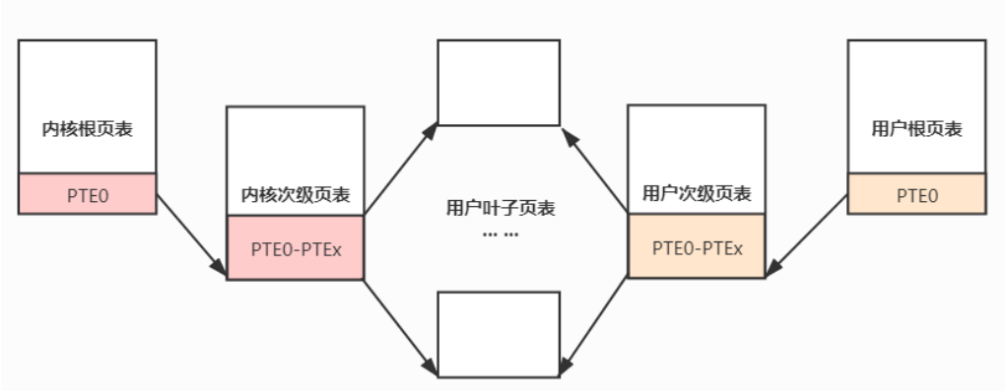
要求：在内核独立页表中添加用户页表的映射，这样在 copyin()和 copyinstr()中就可以不用进行模拟地址翻译。

1. **设计添加用户页表映射的函数**

需要解决两个问题：

1. 是否需要完全复制用户页表内容到内核独立页表？

指导书给出的方案为不复制，内核独立页表和用户页表**共享叶子页表（同时注意在回收内核页表时，要解除内核次级页表和用户叶子页表之间的联系，避免重复回收）**



2. 如何遍历用户虚拟地址空间？

已知用户地址空间的范围为 0x0-0xC000000，一种实现方式是从 0 开始扫描，步长为 PGSIZE，（即一页一页开始扫描）判断此页是否有效（有没有被分配给用户进程），若有则将此映射也添加到内核独立页表中。（然而这种方式太耗时，我最开始就是用这种傻方法）。其实 PCB 中保存有当前进程已分配空间的大小（p->sz），我们只需遍历 0~p->sz，且这些虚拟地址都是有效的。

实现：

实现与用户页表共享叶子页表，对于某用户有效的虚拟地址 va，其在叶子页表中有映射，即叶子页表的 pte 是 valid，也说明在**用户次级页表**中保存有用户叶子页表的地址，即 va 对应的**用户次级页表 PTE** 也是 valid，那么我们将**用户次级页表 PTE** 复制到内核次级页表对应 PTE 中即可。

• 如何找到 va 对应的次级页表 PTE，以便共享叶子页表？

已知 walk()函数可以找到 va 对应的叶子页表 PTE 的物理地址。我们稍微**修改一下 walk()函数循环的次数**，就能找到 va 对应的次级页表 PTE 的物理地址。这个函数我们叫它 walkone()

```
pte_t *
walkone(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walkone");

  for(int level = 2; level > 1; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(1, va)];
}
```

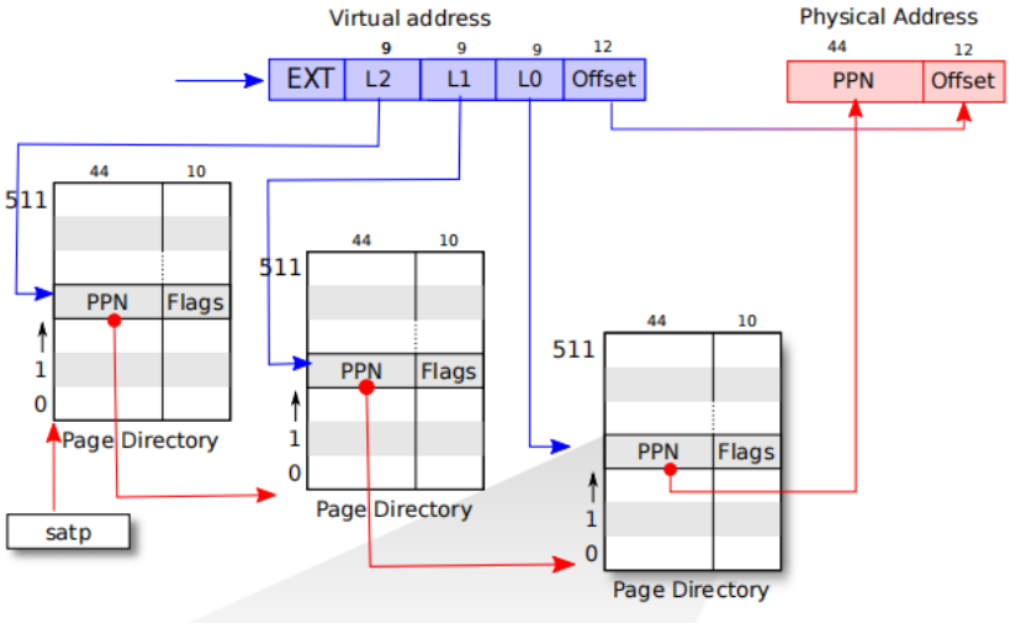
• 从 0-p->sz 的步长怎么选？

如果仍按 PGSIZE 步长一页一页复制，当调用 walkone()时会返回重复的地址，因为有些 va 对应的叶子页表 PTE 是在同一张叶子页表中，它们次级页表的 PTE 是相同的。这样无疑做了无用功。

根据虚拟地址的划分，我们可以发现已知两个虚拟地址 va1，va2。若两者 L2 9 位相同，即它们根页表页号相同，它们对应的次级页表 PTE 在同一个次级页表中；若 L2-L1 18 位相同，即他们根页表页号相同和次级页表页号相同，它们叶子页表 PTE 在同一个叶子页表中（共享同一叶子页表）

例如 va1= 000000000 000000000 000000001 offset$_1$

va2 = 000000000 000000000 000000002 offset$_2$，它们仅 L0 不同（不考虑 offset），这两页对应的叶子页表 PTE 是在同一叶子页表中（它们对应的次级页表 PTE 相同）



那么为了提高效率步长就应该扩大，为(1L << 21),这样每次 walkone 返回的次级页表 PTE 的物理地址都不同。

设计一个宏定义来 FRAMEONEROUNDUP（va）计算 va 的上界（类似 PGROUNDUP），例如 va = 000000000 000000002 000001111 111111111111

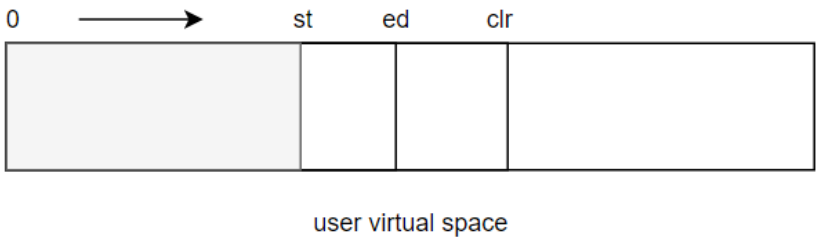
PGROUNDUP(va) = 000000000 000000002 000001112 000000000000

FRAMEONEROUNDUP = 000000000 000000003 000000000 000000000000

• 如何处理用户空间变小？

当用户进程分配的内存变小的情况下，我们也要在内核页表中解除多余空间的映射。用户空间变小的情况有两个 sbrk(n) n<0（growproc()）；exec()切换新程序所占空间小于原来程序的空间。

我们设计的复制映射的函数就输入三个参数，st， ed， clr



复制 st-ed 之间的映射，删除 ed-clr 之间的映射。

```
#define FRAMEONEROUNDUP(sz) (((sz)+(1L << 21)-1) & ~((1L << 21)-1))
int
makesharedmapping(pagetable_t k_pagetable, pagetable_t u_pagetable,
uint64 st, uint64 ed, uint64 clr)
{
```

```
  pte_t *k_pte, *u_pte;
  uint64 a;
    //更新 st~ed 之间的映射
    for(a = FRAMEONEROUNDUP(st); a < ed; a += (1L << 21)){
      if((u_pte = walkone(u_pagetable, a, 0)) == 0)
        panic("makesharedmapping: user walkone");
      if(!(*u_pte & PTE_V))
        panic("makesharedmapping: pte should valid");
      if((k_pte = walkone(k_pagetable, a, 1)) == 0)
        panic("makesharedmapping: kernel walkone");
      *k_pte = *u_pte; //复制 PTE
    }
    //删除 ed~clr 之间的映射
    for(a = FRAMEONEROUNDUP(ed); a < clr; a += (1L << 21)){
      if((k_pte = walkone(k_pagetable, a, 0)) == 0)
        panic("makesharedmapping: kernel walkone");
      *k_pte = 0;//清空 PTE
    }

  return 0;
}
```

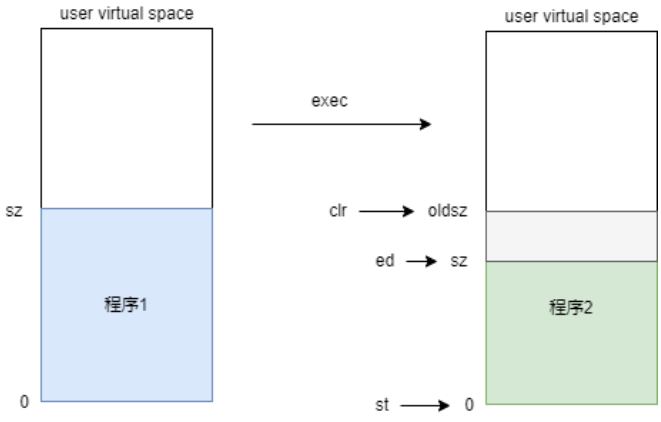
在回收独立页表时，我们要**先解除与用户叶子页表的联系，再清空各级 PTE，删除页表自身。**（这里也从 0-p->sz 遍历，用 walkone()，不再赘述）

2. **更换 copyin()和 copyinstr()**
   由于共享用户叶子页表，使用独立内核页表访问得到的叶子页表 PTE 的 PTE_U 标志位并未改变。这里**修改 sstatus 寄存器中 SUM 位**的值即可。

```
  w_sstatus(r_sstatus() | SSTATUS_SUM);
  int flag = copyinstr_new(pagetable, dst, srcva, max);
  w_sstatus(r_sstatus() & ~SSTATUS_SUM);
  // return 0;
  return flag;
```

3. **同步用户页表映射**
   用户页表被修改了映射时，就**用 makesharedmapping()更新一下内核独立页表**，这里需要注意 st，ed，clr 的取值。
   - sbrk()(即 growproc())
   - fork()
   - exec()
   - userinit()

   这里用 exec()来解释如何设置 st，ed，clr.当 exec 执行的程序大小小于之前的程序。



三、 实验结果截图

请填写。

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (5.1s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.7s)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (0.6s)
== Test usertests ==
$ make qemu-gdb
(169.3s)
== 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
200210231@comp6:~/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。

Xv6 实验回顾

Lab：util

难度：☆☆☆
收获：初步了解一个操作系统工作原理，知道各种系统调用的功能和如何使用。实验任务不难（2 星）。此外因为面临新的实验环境，对虚拟机操作、Linux 操作、gdb 调试、git 操作、Makefile 有了更深刻认识（学习这些内容需要深刻理解，1 星）

Lab：syscall

难度：☆☆
收获：对 xv6 系统调用的运行过程，“陷阱”机制有了理解。实验任务较简单，3 星难度

Lab：lock

难度：☆☆☆☆
收获：深刻理解了为什么需要锁机制，如何去优化锁机制。本实验主要是数据结构+锁。任务一较为简单，而任务二是双向链表和会涉及同时获得两个锁，很容易因为链表处理不当（自己就因为在这里踩了坑，卡了半天）或锁处理不当造成死锁。而一旦死锁系统不会有任何提示，只会无限期卡住，增大了 debug 难度，所以 4 星难度。

Lab：pgtbl

难度：☆☆☆☆☆
收获：对页表机制有了深（n 个深）刻的理解，特别是虚实地址映射、用户内存空间。同时也加深了用户态内核态切换等问题了理解。任务一简单（2 星），任务二跟随指导书思路即可（3.5 星），任务三设计同步用户映射的函数很难，这里我尝试了 2 种方法，一开始还走了弯路（遍历全部 0-C0000000）（上面只阐述了指导书提供的共享叶子的方法 makesharedmapping()函数，我一开始实现的是不共享完全复制的方法 makemapping()函数，即复制 va 对应的叶子页表 PTE）其实我觉得共享叶子的方法更难。完成这个需要对用户进程空间的分配回收和页表映射有深刻的理解，值得 5 星。

实验建议

1. 虽然每个实验都侧重于 OS 的不同方面，但做完 4 个实验，对操作系统全貌还是了解不深，特别是一开始对整个系统框架，流程完全不清楚，看了 xv6 book 和老师的 B 站视频才明白。所以可以增加更多这方面的内容。
2. 虽然学了 GDB 调试，但还是不怎么会用，可以增加这方面内容，以便后续出问题好解决。
3. 每次实验的解答题可以像之前 lab1 抽查 Linux 一样，这样预习效率更高，写报告也简单点（小小建议）