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

实验设计报告

开课学期：2024 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 月 14 日 地点：T2507

学生班级：计算机 4 班

学生学号：220110430

学生姓名：吴梓滔

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2024 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？

页表机制发明是为了解决以下问题：

1. 进程的内存保护：通过虚拟地址映射，实现进程间的内存隔离，提高系统安全性。
2. 为编程提供便利：程序员只需要关注虚拟地址，不必考虑实际的物理内存分配。
3. 使内存管理灵活：操作系统可以更灵活地管理物理内存，减少内存碎片，提高内存利用率。

页表提供一个翻译地址的功能，使编程时只关注虚拟地址而非物理地址，各进程有自己的虚拟地址空间，有上述三点好处。

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 标准下，虚拟地址实际用到低 39 位进行地址映射。这个虚拟地址的低 39 位按照“L2 页号-L1 页号-L0 页号-偏移量”的格式拆解如下：

L2 页号：0x19E
L1 页号：0x4D
L0 页号：0xBC
偏移量：0xDEF

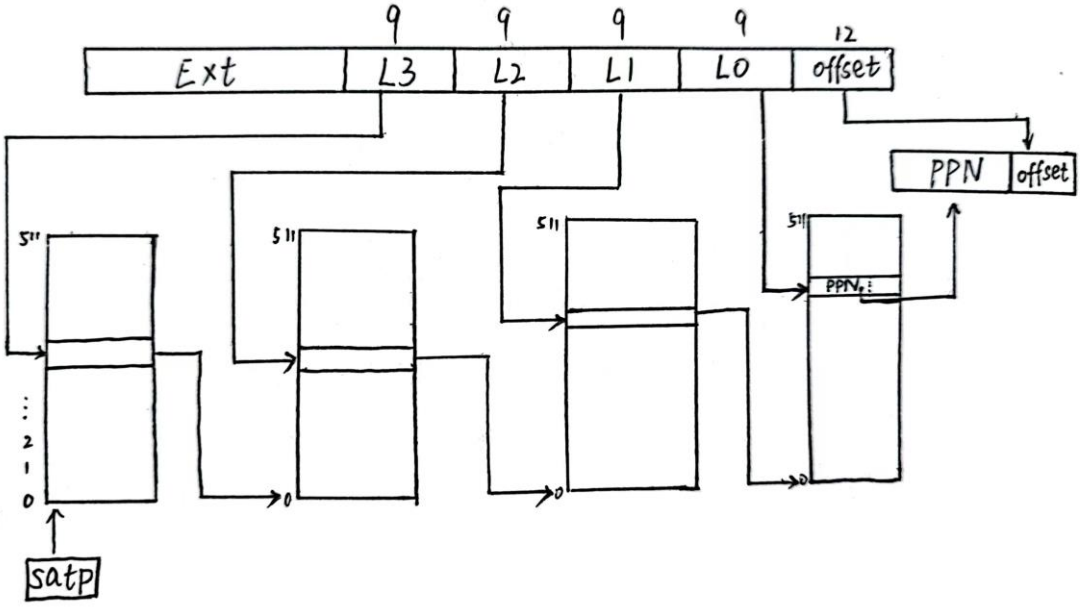
页表翻译过程：

1) 先根据 SATP 寄存器，找到根页表（L2 页表）的物理地址 PA2，用 PA2 为基址，加上 L2 页号的偏移量（以 8 字节为单位），得到物理地址 PA2[0x19E]，找到 L2 页表上的 PTE，取出该 PTE 的内容，得到对应的 L1 页表的基址 PA1。
2) 以 PA1 为基址，加上 L1 页号的偏移量，即物理地址 PA1[0x4D]，找到 L1 页表上的 PTE，取出该 PTE 的内容，得到对应的 L0 页表的基址 PA0。
3) 以 PA0 为基址，加上 L0 页号的偏移量，即物理地址 PA0[0xBC]，找到 L0 页表上的 PTE，取出该 PTE 的内容，即目标页帧的物理地址基址 PAbase。
4) 将 PAbase 加上虚拟地址中的偏移量 0xDEF，即得到最终的物理地址 PA。

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）

能。在页表设计中，一个页表项即 PTE 的大小是 8 字节，这是由于 xv6 中地址为 64 位决定的。因为已经规定一个页的大小是 4kB，一个 PTE 的大小是 8B，所以一个页表中能够容纳的 PTE 的个数是 4kB/8B=512 个，即 2^9 个，因此一个页表中的页号用 9bit 来编址。所以三级页表的页号都设置为 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。

任务 1 vmprint 的设计

下面对于三级页表的称呼，采用 xv6 的设计，称 L2 页表、L1 页表、L0 页表，数据页称为页帧。L2 页表即根页表。L0 页表即叶子页表。

vmprint 读入一个页表，按格式打印相关信息。由于要打印三级页表中所有有效的页表以及数据页框，总体采用仿照 freewalk 的方式进行页表项的读取和有效判断。但由于输出时要根据页表深度输出相应数量的“||”，且页表和页帧的输出格式不同，页帧还需要知道输出虚拟地址，需要记录每一级页表的偏移量来构成虚拟地址。这需要函数时刻知道处在第几级页表或页帧，记录相关数据，因此可能不是很适合递归打印。考虑到页表一共三级，可以直接用三层嵌套的 for 循环来完成页表的遍历，而不是使用递归结构。

由于不适合递归打印，而该函数输入参数 pagetable 一定是一个 L2 页表，可以直接逐层遍历，直至页帧，在此过程中，三层 for 循环的 i，j，k 参数都是一路记

录下来的，在输出页帧时也刚好可以还原出虚拟地址，因此很适合用三层 for 循环完成页表遍历。

按照格式，输出 L2 页表地址，用%p 格式化输出。之后直接用 for 循环遍历 L2 页表，由于有 512 个页表项，i 从 0 到 511。每获取一个页表项 pte 后，仿照 freewalk 中的判断条件

```
if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0)
```

来判断该 pte 是否有效。若有效，则需要输出并继续遍历该页表项所指的 L1 页表。通过宏定义 PTE2PA 获取该 pte 所指的 L1 页表，并用类似的 for 循环继续遍历 L1 页表。输出的语句为

```
printf("||idx: %d: pa: %p, flags: ----\n", i, l1);
```

由于是 L1 页表，输出语句前面直接加上“||”，后续 L2 页表则加“|| ||”，直接硬编码相应个数的杠号。由于是页表，flags 硬编码成“----”。

当遍历 L0 页表时，判断页表项是否有效的方法不同于 L2 页表和 L1 页表，判断条件只考虑有效位。每次取得的页表项直接指向一个页帧。判断因此要根据输出格式要求制作标志位字符串和虚拟地址。虚拟地址可以直接使用三层 for 循环的 i，j，k 来还原出 L2，L1，L0 页号。值得注意的是，i，j，k 定义时是 int 型，应该强转成 uint64 型，才能通过移位成为虚拟地址。标志位字符串则先声明字符串，对页表项 pte 依次判断 4 个标志位并作相应设置。

```
char flags[4] = "----";
if (pte & PTE_R) flags[0] = 'r';
if (pte & PTE_W) flags[1] = 'w';
if (pte & PTE_X) flags[2] = 'x';
if (pte & PTE_U) flags[3] = 'u';
```

采用以下语句做出虚拟地址：

```
uint64 va = ((uint64)i << 30) | ((uint64)j << 21) | ((uint64)k << 12);
```

之后即可用格式化输出打印页帧的相关数据。前面硬编码 3 个双杠符号，用%s 格式化输出标识位字符串。

```
printf("|| || ||idx: %d: va: %p -> pa: %p, flags: %s\n", k, va, PTE2PA(pte), flags);
```

任务 2　独立内核页表的设计

为了给每个进程增加独立页表，类似于 PCB 中的独立用户页表 pagetable 一样，在 PCB 的定义即结构体 proc 中增加 pagetable_t k_pagetable 字段，作为独立内核页表；增加 uint64 kstack_pa，储存内核栈物理地址。储存内核栈物理地址是为了方便在 procinit 中申请内核栈后先储存物理地址，直到 allocproc 中再将其映射到页表。

重新设计创建内核页表的函数。原本的创建内核页表的函数 kvminit，进行内核页表映射的函数 kvmmap 都是直接对全局内核页表进行操作。设置每个内核页表后，创建内核页表的函数需要能够申请内核页表并返回，设置页表映射的函数需要能够对指定的页表设置映射，因此对这两个函数进行改装，设计新的函数 kvminlt 以及 kvmmap_2。

kvmmap_2 的函数体如下：

```
void kvmmap_2(uint64 va, uint64 pa, uint64 sz, int perm, pagetable_t
pagetable) {
   if (mappages(pagetable, va, sz, pa, perm) != 0) panic("kvmmap_2");
}
```

该函数之比 kvmmap 函数增加了一个形参 pagetable，并把 mappages 映射的页表改为传入的 pagetable。从而实现可以对指定的页表进行映射。

基于新的函数 kvmmap_2,可以实现新的页表创建函数 kvmin1t。

```
pagetable_t kvmin1t() {
   pagetable_t k_pagetable = (pagetable_t)kalloc();
   memset(k_pagetable, 0, PGSIZE);
   // 下面映射各个内存部分，除了 CLINT
   // uart registers
   kvmmap_2(UART0, UART0, PGSIZE, PTE_R | PTE_W, k_pagetable);

   // virtio mmio disk interface
   kvmmap_2(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W, k_pagetable);

   // PLIC
   kvmmap_2(PLIC, PLIC, 0x400000, PTE_R | PTE_W, k_pagetable);

   // map kernel text executable and read-only.
   kvmmap_2(KERNBASE, KERNBASE, (uint64)etext - KERNBASE, PTE_R | PTE_X,
k_pagetable);

   // map kernel data and the physical RAM we'll make use of.
   kvmmap_2((uint64)etext, (uint64)etext, PHYSTOP - (uint64)etext,
PTE_R | PTE_W, k_pagetable);

   // map the trampoline for trap entry/exit to
   // the highest virtual address in the kernel.
   kvmmap_2(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X,
k_pagetable);

   return k_pagetable;
}
```

kvmin1t 和 kvminit 按照几乎相同的方式进行了地址映射，但是没有映射 CLINT 地址，并且 kvminit 中使用的 kvmmap 函数都换成 kvmmap_2。最后返回新创建的 k_pagetable。

内核栈映射方式的修改

原本的 procinit 函数会为每个进程申请内核栈，并映射到全局内核页表中。新增独立内核页表后，只增加一个语句，将新申请的内核栈物理地址 pa，存入该进程 PCB 中内核栈物理地址的字段 kstack_pa 中。

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

为了将内核栈映射到独立页表中，在 allocproc 中，在进程确定找到并且要执行后，再完成内核栈在独立内核页表中的映射。此时内核栈虚拟地址和物理地址均在 PCB 中存储好，直接调用 kvmmap_2 方法，传入 PCB 中的两个地址实现映射。

```
kvmmap_2(p->kstack, p->kstack_pa, PGSIZE, PTE_R | PTE_W, p->k_pagetable);
```

调度器的修改

内核页表的切换是在调度器切换进程的时候完成的。在 schedular()函数中，调用 swtch 函数时是切换到进程的上下文环境,satp 寄存器中页表的切换应该在 swtch 之前完成。仿造 kvminithart 函数，将 satp 寄存器设置为当前进程的独立内核页表并刷新快表。

```
w_satp(MAKE_SATP(p->k_pagetable));
sfence_vma();     // 刷新 TLB
```

进程运行结束后，会回到 swtch 函数的下一行。此时，再次将页表切换回全局内核页表。直接调用 kvminithart 函数即可。

内核页表的释放

进程结束后会在 freeproc 中释放进程相关的页表、栈等，为了给新定义的独立内核页表设置相应的释放功能，应实现一个释放独立内核页表的函数。该函数与已有的 freewalk 不同，freewalk 用于释放进程的用户页表。freewalk 与 uvmunmap 是配合使用完成释放的，先由 uvmunmap 释放所有的数据页帧，再运行 freewalk 释放全部三级页表，freewalk 确保所有映射到的数据页帧已经释放，否则会报错。

对于内核页表而言，释放页表时是不希望释放映射到的页帧的，因为这些页帧其实是由多个进程的独立内核页表共享的。仿造 freewalk 的逻辑，设计 freewalk 函数，用于释放页表，但不要求页帧已经释放。为了实现该功能，简单的想法是记录页表等级，在递归调用的函数中作为参数传递。因此，freewalk 函数除读入也表外，还读入整数 level，对于 level 为 2 和 1 的 L2 页表和 L1 页表，需要递归释放所有有效的次级页表（传入参数 level-1），再将自身释放；对于 level 为 0 的 L0 页表，直接将自身释放，而无需遍历自身的 PTE。这样的设计能够不遍历 L0 页表而是直接删除，在时间上也比 freewalk 更有效率。

```
void freewalk(pagetable_t pagetable, int level) {
  if(level < 0 || level > 2) panic("freewalk: level");
  if(level == 0){} else
  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);
      freewalk((pagetable_t)child, level - 1); // 递归释放下一级
    }
  }
  kfree((void *)pagetable);
}
```

freewalk 函数每次递归调用自身时为 freewalk((pagetable_t)child, level-1)由于参数 level 每次调用会减少 1，调用深度最多为 3。规定了调用 freewalk 函数时，第二个参数必须传入 2。在函数内部，还会检查 level 是否为 0，1，2 以外的值，如果 level 取了别的值将报错。

在 freeproc 函数中，调用 freewalk 来释放独立内核页表。在 freeproc 函数体内增加如下语句：

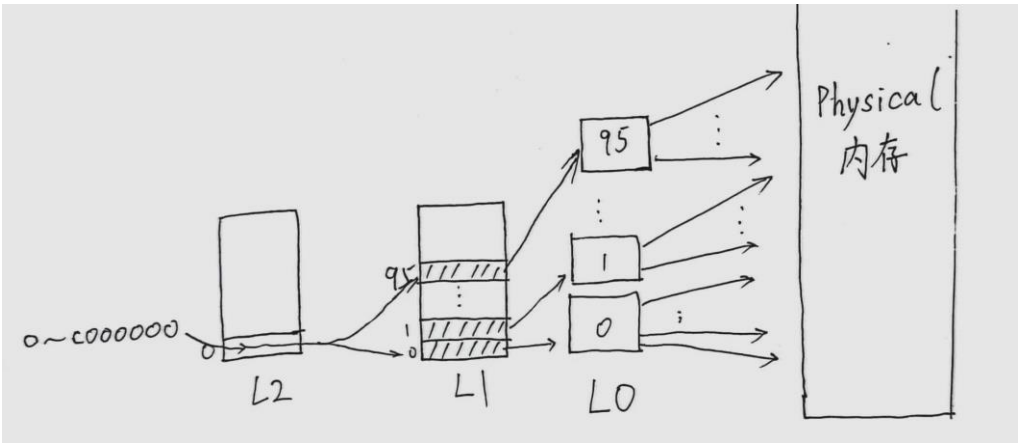
```
freewalk(p->k_pagetable, 2);
p->k_pagetable = 0;
```

任务 3　简化软件模拟地址翻译

这一任务的实质是把进程的用户页表中用户地址空间添加到内核页表中，并启用 copyin_new 来进行数据传输。

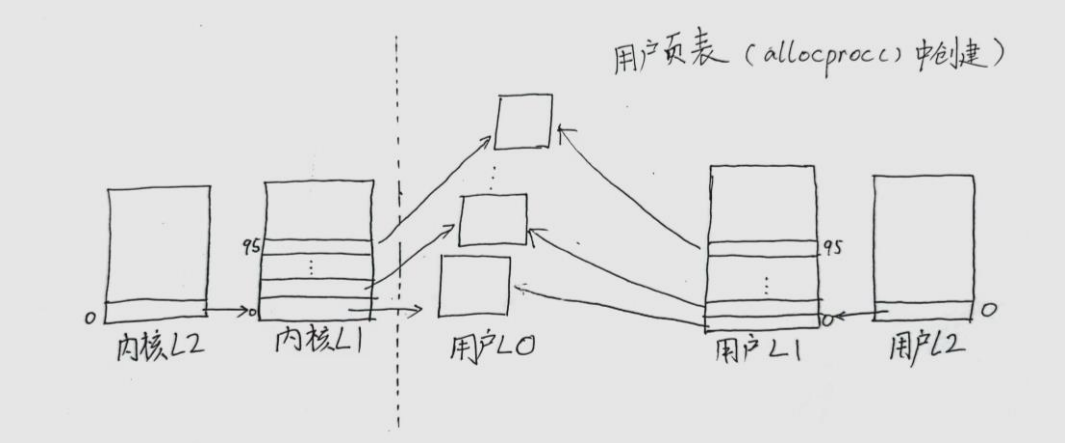
用户地址空间的映射

用户地址空间的范围为 0x0-0xC000000，最高地址转换成十进制是 12×224，由于一个页帧大小是 4kB，用户地址空间的页数=（12×224）/4k = 12×212 页，而一个 L0 页表能够映射 512 个页帧，所以用户地址空间需要的 L0 页表数=（12×212）/512 = 96 个，即 96 个 L0 页表就覆盖了用户地址空间。相当于 L2 页表中偏移为 0 的 PTE 所映射的 L1 页表的前 96 个 PTE，便覆盖了用户地址空间。



用户地址空间示意图，只会用到最低的 96 个 L1 PTE

基于这个计算，用户地址空间可以认为是前 96 个 L1 页表项。采用内核页表和用户页表共享 L0 页表的方式，只需要将内核页表中对应的 96 个 L1 页表项设置为和用户页表中对应页表项相同，那么内核 L1 页表也就指向了用户页表的 96 个 L0 页表，从而用户地址空间的虚拟地址通过内核页表也可以完成正确的翻译。



内核页表中用户地址空间映射示意图

基于这一思路，将用户页表同步到内核页表中，其实只需要将用户页表的第一个 L1 页表的前 96 个页表项，赋值给内核页表中对应的 L1 页表项即可。这样的实现方式，避免了数量庞大的 L0 页表的复制。

在要进行页表同步的时候，还需确定内核页表中，在 L2 页表中偏移为 0 的页表项对应的 L1 页表（称作第一个 L1 页表）已经存在。可以断定这个 L1 页表一定存在，因为创建内核页表的时候，会映射 PLIC 地址，由于第一个 L1 页表覆盖的范围 512*512*4kB=230B，PLIC 的地址值显然是小于这一个地址范围，所以在映射 PLIC 的时候一定创建了第一个 L1 页表。因此，在同步用户页表到内核页表的时候，可以直接用索引为 0 来访问这一个 L1 页表。

基于以上对页表项个数、页表是否已申请的说明，可以设计一个将用户页表同步到内核页表的函数 sync_pagetable。

```
void sync_pagetable(pagetable_t u_pagetable, pagetable_t k_pagetable){
   pagetable_t u_l1 = (pagetable_t)PTE2PA(u_pagetable[0]);
   pagetable_t k_l1 = (pagetable_t)PTE2PA(k_pagetable[0]);
   for(int i = 0; i < 96; i++)          k_l1[i] = u_l1[i];
}
```

这一函数读入用户页表 u_pagetable 和内核页表 k_pagetable,直接使用索引值 0 和宏定义 PTE2PA 获取到用户页表和内核页表的第一个 L1 页表，直接对前 96 个页表项进行赋值，就完成了整个用户地址空间的同步。

替换 copyin 和 copyinstr 函数

用 copyin_new 替换 copyin 函数，用 copyinstr_new 替换 copyinstr 函数。在新的函数内，会通过内核页表直接硬件翻译的方式完成数据访问。但是，在上面同步页表的时候，内核页表中获取的用户页表项是直接赋值，并没有设置 User 位。在用户页表中，User 位为 1，直接赋值到了内核页表中。对内核而言，User 位为 1 是不允许访问的。为了使内存能够访问目标地址，除了修改对应页表项的标识位外，还可以通过修改 sstatus 寄存器的 SUM 位来实现。本设计采用修改 SUM 位的方法。

RISC-V 架构中，sstatus 寄存器的第 18 位是 SUM 位，用于临时允许内核态访问用户态内存，而无需修改页表项的 User 位。当 SUM 位为 1 时，内核是可以直接访问 User 为 1 的页表项的。在每次进行 copyin_new 之前，把 SUM 位改为 1,在 copyin_new 之后立刻改回 0，即可实现允许内核态的访问。

具体的操作过程，需先宏定义 SUM 位。在 riscv.h 中增加一句宏定义

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

此后将改写 sstatus 寄存器的语句加到 copyin_new 前后。修改后，copyin 的函数体为：

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

对于 copyinstr 做相同的改动。

增加同步页表的语句

现在可以将此前设计的同步页表的函数在需要同步页表的时候调用。每一次用户页表被修改了映射的同时，都应该立即和内核页表做一次同步。一共需要在 fork、exec、growproc、userinit 四个函数内增加同步的语句。

在 userinit 中，初始进程初始化用户页表后，应该进行同步。将 sync_pagetable 的调用放在 uvminit 语句后。

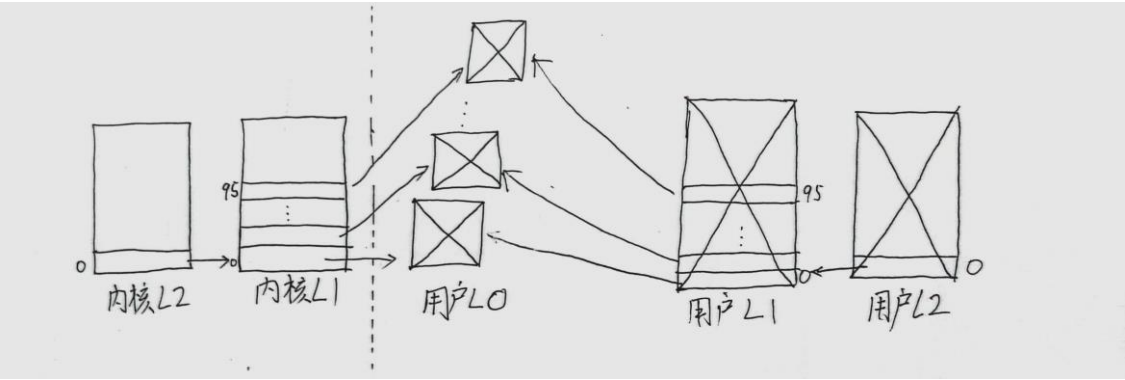
```
uvminit(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
// 这里需要同步页表
sync_pagetable(p->pagetable, p->k_pagetable);
```

在 fork 中，子进程完成了用户页表的继承后，就可以调用页表同步函数，将用户页表同步到内核页表。sync_pagetable 的调用出现在原来 uvmcopy 的后面。

在 exec 中，p->pagetable = pagetable 语句完成了进程用户页表的替换。在这一句之后，就可以调用 sync_pagetable(p->pagetable, p->k_pagetable)完成页表的同步。

在 growproc 中，原本会使用 uvmalloc 或者 uvmdealloc 更改用户页表。在这一分支语句后，调用 sync_pagetable 函数完成页表的同步。

页表释放的修改



页表删除示意图

由于内核页表和用户页表共享了叶子节点，在 freeproc 中，先执行 proc_freepagetable(p->pagetable, p->sz);释放了用户页表后，三级用户页表全部删除（用×表示）。此时对于独立内核页表中用户地址空间的映射而言，L0 页表是不存在的，但是相应的 L1 页表项不会因为用户页表的删除而更改，此时 L1 页表

项与删除之前是一样的。此后若直接运行删除内核页表的 freewalk 函数，仍然会尝试释放已经释放掉的 L0 页表，从而导致错误。

为了避免重复释放的错误，需要在释放了用户页表之后，将独立内核页表中对应的页表项也设置为 0，以防止 freewalk 函数释放独立内核页表时重复释放。

```
if (p->pagetable) proc_freepagetable(p->pagetable, p->sz);
pagetable_t l1 = (pagetable_t)PTE2PA(p->k_pagetable[0]);
for (int i = 0; i < 96; i++)      l1[i] = 0;
```

上面代码在释放用户页表后，获取独立内核页表的第一个 L1 页表，用循环语句将 96 个页表项逐一清 0，这样就完成了完整的用户页表释放，后续不会发生重复释放的问题。

三、 实验结果截图

请给出任务一、任务二、任务三和 make grade 的运行结果截图。

```
220110430@comp0:~/xv6-oslab24-hitsz$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virti

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
$
```

```
$ kvmtest
kvmtest: start
test_pagetable: 1
kvmtest: OK
```

```
$ stats stats
copyin: 58
copyinstr: 12
```

```
$ 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 p
            sepc=0x0000000000005476 stval=0x0000000000005476
usertrap(): unexpected scause 0x000000000000000c pid=3237
            sepc=0x0000000000005476 stval=0x0000000000005476
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
```

```
usertrap(): unexpected scause 0x000000000000000d pid=62
            sepc=0x0000000000002048 stval=0x0000000801
OK
test sbrkfail: usertrap(): unexpected scause 0x0000000
            sepc=0x0000000000003ec2 stval=0x0000000000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x0000000
            sepc=0x00000000000021b6 stval=0x0000000000
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: 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 (4.2s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.0s)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (1.0s)
== Test usertests ==
$ make qemu-gdb
(174.4s)
== 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
220110430@comp0:~/xv6-oslab24-hitsz$
```

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。

4 个实验的收获，实验 1 是帮助学生对 xv6 系统的初步上手，实验 1 帮我熟悉了 ssh、linux 的使用，以及后续实验中的操作方式。

实验 2 是系统调用和内核调度进程有关的，通过实验 2 我清晰地理解了操作系统内核的角色，感受到内核和用户程序是相对隔离的，它们是通过硬件来建立一个联系，这个感悟是在理论课无法收获到的。这一次实验的收获对于理解“操作系统是什么”是不可或缺的。

实验 3 我认为是做并发优化，在物理内存分配和磁盘缓冲区这两个背景下提升并发性，自主开发的偏多一点。这次实验中我收获了如何设计程序来提升并发性的思想，这不只是在操作系统中有用。同时也对并发进程和为什么要有互斥锁有了清

晰的理解，在多核的时候，一个进程有短暂的空窗不持有锁，那么它访问的资源内容是否和此前一样是不能保证的，这也是在亲自编程中才获得的感悟。

实验 4 是页表的设计，我认为这个实验的收获不如实验 2 和实验 3，实验 4 是为我们提供了一个页表机制的案例，通过这个案例，能够感受到进程的虚拟地址是独立的，页表的翻译是依靠硬件的，页表是帮程序员屏蔽物理地址的。别的收获就是积累了一些调试经验和编程教训。

我通过实验 4 加深了页表做地址翻译的机制，但是收获有限，对于理论课不能算补充，只能算是巩固了一下理解。实验 4 遇到的困难主要是了解 xv6 设计页表的机制，难度在于它个性化的部分，我在试验中遇到的挑战主要在于 bug 的调试，也有一部分困难来自于指导书讲的不大清楚，总之感觉实验 4 的质量和收获都不如 2 和 3。

总的来说，实验 2 和实验 3 带给我超出理论课的收获。实验 4 有点迷茫，不能算补充理论课，只能算是巩固了相关知识，感觉和理论课配合的不是很好，指导书和实验任务配合的不是很好。

做完实验，感觉麻省的学生也没多厉害。

我感觉 xv6 的实验本质上是代码的阅读理解，而不是设计或开发。主要是根据实验任务要求理解相关代码，之后用很少的改动完成需要的功能。我的浅显理解是，Xv6 实验的灵魂在于指导书恰到好处的引导，要想从这实验中有所收获，好的指导书是最关键的部分。指导书给的东西少了，做实验一头雾水不知道到底想干什么，导致花了大量没有意义的时间终于跑通了实验，还不知道有什么收获。指导书的提示给多了，会糊里糊涂做完了却不知道为什么能成功，这两种情况实验的效果都不好。

我认为这个实验的效果来自于指导书、代码框架、实验任务的良好配合，实验的难度也是来源于这三者配合的不好。Xv6 的代码本身是不难的，实验任务要做的代码工作本身也是很简单的，难度都是来源于第一次接触代码框架时，对 xv6 系统实现方式的不了解。做实验 4 的时候尤其觉得实验指导书可能需要优化一下，我做的时候就感到迷茫。