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

实验设计报告

开课学期：2021 年秋季学期

课程名称：操作系统

实验名称：Lab4:页表

实验性质：课内实验

实验时间：11 月 4 日 地点：T2608

学生班级：计算机 4 班

学生学号：190110419

学生姓名：李怡凯

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？

为了充分使用内存资源，最大化利用空间以及进行内存保护，提出页表机制来实现内存管理需求；

1) 将内存分为大小相等的页，为每个进程提供自己的内存页，实现内存分配和内存保护；
2) 通过分页机制减少内存内部碎片出现，提高内存利用率

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）

二进制地址：110 0111 1000 1001 1010 1011 1100 1101 1110 1111

L2：110 0111 10 0x19E

L1：00 1001 101 0x04D

L0：0 1011 1100 0x0BC

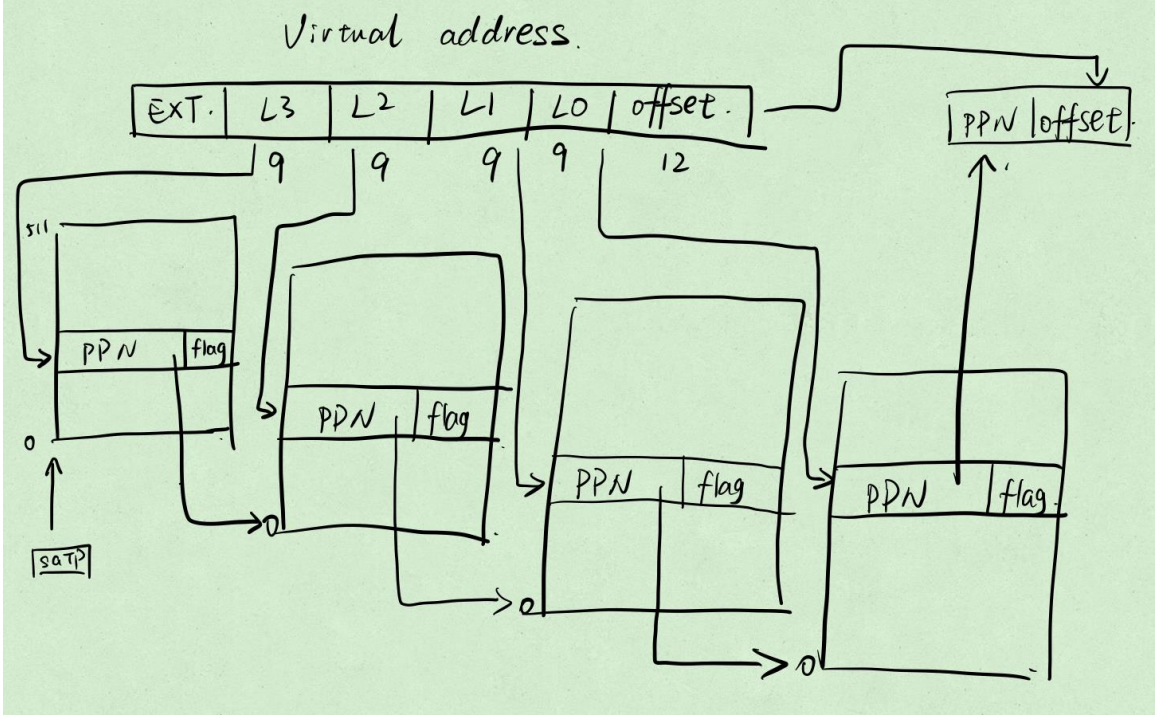
Offset：1101 1110 1111 0xDEF

通过根页表索引和根页表基址（假设为 0x0000 0000 0000 0000）访问根页表，得到次页表的基地址，假设为 0x1000 0000 0000 0000；结合次页表索引访问 0x1000 0000 0000 004D，得到叶子页表基地址，假设为 0x2000 0000 0000 0000，结合叶子页表索引得到页基址，假设为 0x3000 0000 0000 0000，结合页内偏移，得到地址为 0x3000 0000 0000 0DEF，完成寻址。

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）

页目录本质上也是页表。按字节寻址，4K 大小的页表大小对应 4KB 空间，一个页表项为 8B，页表内有 512 个页表项，对应 9 位偏移量。

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. 打印页表

任务 1 要求实现一个打印页表程序协助调试，可参考 freewalk 的实现来遍历整个页表，流程图如图 1 所示。

2. 独立内核页表

任务 2 要求将 xv6 原来的共享内核页表改为独立内核页表，新的进程的独立内核页表要能够映射到原来内核页表的地址空间，使其能够正常使用，具体设计如下。

首先在 proc 结构体中添加一个成员变量内核页表，然后在调用 allocproc 设置进程的时候的时候初始化该内核页表，建立相关映射，如图 2、图 3、图 4 所示。

然后修改 scheduler 的部分逻辑，使其在选定一个进程运行的时候调入该进程的内核页表，在进程执行完成返回后，重新启用全局的内核页表，具体设计如图 5 所示。

最后在释放进程的时候将内核页表的内核栈部分解除映射并释放内存空

间，其他部分解除映射，如图 6，图 7，图 8 所示。

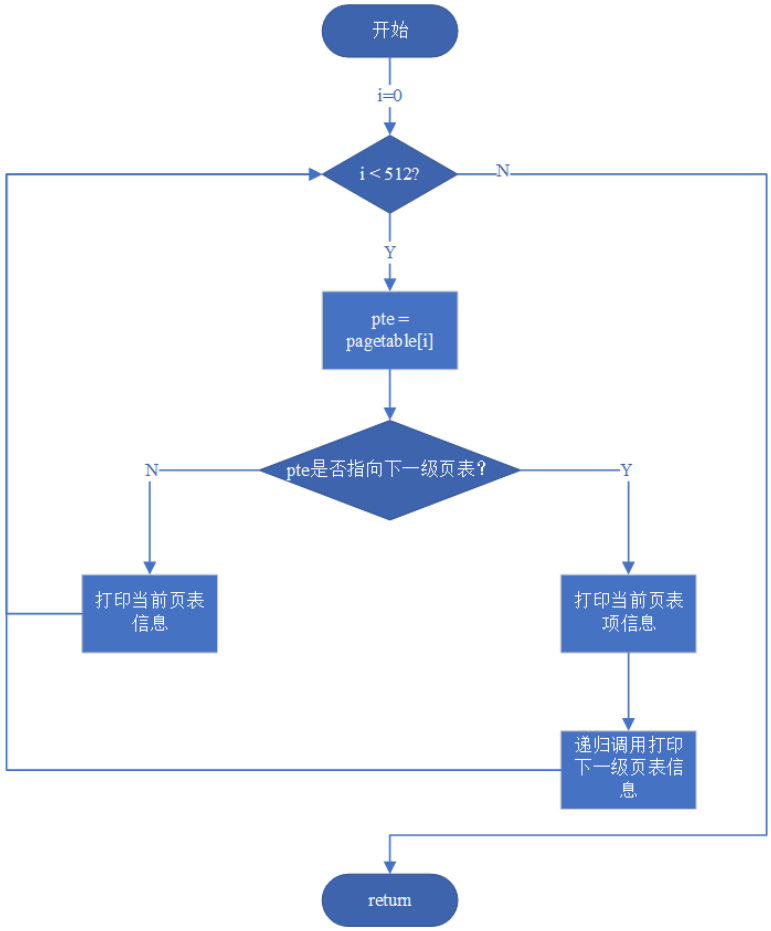


图 1 vmprint 流程图

图 2 在 proc 结构体中添加成员变量内核页表

```
86  struct proc {
87    struct spinlock lock;
88
89    // p->lock must be held when using these:
90    enum procstate state;        // Process state
91    struct proc *parent;         // Parent process
92    void *chan;                  // If non-zero, sleeping on chan
93    int killed;                  // If non-zero, have been killed
94    int xstate;                  // Exit status to be returned to parent's wait
95    int pid;                     // Process ID
96
97    // these are private to the process, so p->lock need not be held.
98    uint64 kstack;               // Virtual address of kernel stack
99    uint64 kstack_pa;
100   uint64 sz;                   // Size of process memory (bytes)
101   pagetable_t pagetable;       // User page table
102   pagetable_t kernel_pagetable;  // kernel page table
```

```
119   p->kernel_pagetable = proc_kvminit();
120   if(p->kernel_pagetable == 0){
121     freeproc(p);
122     release(&p->lock);
123     return 0;
124   }
125   // 建立栈映射
126   char *pa = kalloc();
127   if(pa == 0)
128     panic("kalloc");
129   uint64 va = MAXVA - 4*PGSIZE;
130   mappages(p->kernel_pagetable, va, PGSIZE, (uint64)pa, PTE_R | PTE_W);
131   p->kstack = va;
```

图 3 allocproc 中创建内核页表并建立内核栈映射

```
58
59    memset(kernel_pagetable, 0, PGSIZE);
60
61    // uart registers
62    // kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);
63    mappages(kernel_pagetable, UART0, PGSIZE, UART0, PTE_R | PTE_W);
64
65
66    // virtio mmio disk interface
67    // kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
68    mappages(kernel_pagetable, VIRTIO0, PGSIZE, VIRTIO0, PTE_R | PTE_W);
69
70
71    // CLINT
72    // kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);
73
74    // PLIC
75    // kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);
76    mappages(kernel_pagetable, PLIC, 0x400000, PLIC, PTE_R | PTE_W);
77
78    // map kernel text executable and read-only.
79    // kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
80    mappages(kernel_pagetable, KERNBASE, (uint64)etext-KERNBASE, KERNBASE, PTE_R | PTE_X);
81
82    // map kernel data and the physical RAM we'll make use of.
83    // kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
84    mappages(kernel_pagetable, (uint64)etext, PHYSTOP-(uint64)etext, (uint64)etext, PTE_R |
85
86    // map the trampoline for trap entry/exit to
87    // the highest virtual address in the kernel.
88    // kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
89    mappages(kernel_pagetable, TRAMPOLINE, PGSIZE, (uint64)trampoline, PTE_R | PTE_X);
90
91    return kernel_pagetable;
92  }
93
```

图 4 创建内核页表并建立基础映射的过程

```
531     for(p = proc; p < &proc[NPROC]; p++) {
532       acquire(&p->lock);
533       if(p->state == RUNNABLE) {
534         // Switch to chosen process.  It is the process's job
535         // to release its lock and then reacquire it
536         // before jumping back to us.
537         p->state = RUNNING;
538         // 调入进程对应的内核页表
539         w_satp(MAKE_SATP(p->kernel_pagetable));
540         sfence_vma();
541
542         c->proc = p;
543         swtch(&c->context, &p->context);
544
545         // Process is done running for now.
546         // 进程运行完成需要切换回内核页表
547         kvminithart();
548         // It should have changed its p->state before coming back.
549         c->proc = 0; // cpu dosen't run any process now
550
551         found = 1;
```

图 5 经过修改的 scheduler 逻辑

```
153   static void
154   freeproc(struct proc *p)
155   {
156     if(p->trapframe)
157       kfree((void*)p->trapframe);
158     p->trapframe = 0;
159
160     // 释放内核页表
161     if(p->kernel_pagetable)
162       proc_free_kernelpagetable(p->kernel_pagetable, p->kstack, p->sz);
163     if(p->pagetable)
164     {
165       proc_freepagetable(p->pagetable, p->sz);
166
167     }
168     p->kstack = 0;
169     p->kernel_pagetable = 0;
170     p->pagetable = 0;
171     p->sz = 0;
172     p->pid = 0;
173     p->parent = 0;
174     p->name[0] = 0;
175     p->chan = 0;
176     p->killed = 0;
177     p->xstate = 0;
```

图 6 经过修改的 freeproc 逻辑

```
556  void proc_free_kernelpagetable(pagetable_t kernel_pagetable, uint64 kstack, uint64 size)
557  {
558    if(kstack)
559    {
560      pte_t *pte = walk(kernel_pagetable, kstack, 0);
561      kfree((void *)PTE2PA(*pte));
562    }
563
564    freewalk_new(kernel_pagetable);
565  }
```

图 7 释放内核页表过程

```
339  void freewalk_new(pagetable_t pagetable)
340  {
341    for(int i = 0; i < 512; i++){
342      pte_t pte = pagetable[i];
343      if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
344        // this PTE points to a lower-level page table.
345        uint64 child = PTE2PA(pte);
346        freewalk_new((pagetable_t)child);
347        pagetable[i] = 0;
348      } else if(pte & PTE_V){
349        pagetable[i] = 0;
350      }
351    }
352    kfree((void*)pagetable);
353  }
```

图 8 freewalk_new 流程

3. 简化软件模拟地址翻译

任务三要求在内核页表中添加对用户页表的映射，从而在内核态下需要访问用户页表时可以省略切换页表的过程，具体设计如下。

首先在用户页表被修改的 fork()，exec()，sbrk()三个函数中添加内核页表对用户页表修改的同步。其中页表的映射采用复制叶子页表的方式实现，注意需要将原页表项中的 User 标志位置 0.

fork 函数会创建一个子进程，子进程继承父进程的页表，所以需要在 fork 函数中，在子进程完成对父进程的页表的复制后，添加对用户页表的映射，如图 9 所示。

```
351    // 同时也将子进程的页表复制到内核页表上TODO:开注释
352    for(uint64 va = 0; va < np->sz; va+=PGSIZE)
353    {
354      pte_t *pte = walk(np->pagetable, va, 0);
355      pte_t *kernel_pte = walk(np->kernel_pagetable, va, 1);
356      *kernel_pte = (*pte)&(~PTE_U);
357    }
```

图 9 fork 函数中的修改

exec 函数会创建一个新的页表并抛弃旧的页表，所以在内核页表中同

样需要释放对旧页表的映射，并建立对新页表的映射，如图 10 所示。

```
122   // 释放原来的用户页表对应的内核页表部分，重新映射新的用户页表TODO:开注释
123   uvmunmap(p->kernel_pagetable, 0, PGROUNDUP(oldsz)/PGSIZE, 0);
124   proc_freepagetable(oldpagetable, oldsz);
125   // 建立新的用户页表和内核页表的映射TODO:开注释
126   for(uint64 va = 0; va < (p->sz); va+=PGSIZE)
127   {
128     pte_t *pte = walk(p->pagetable, va, 0);
129     pte_t *kernel_pte = walk(p->kernel_pagetable, va, 1);
130     *kernel_pte = (*pte)&(~PTE_U);
131   }
```

图 10 exec 函数的修改

sbrk 函数用于扩展内存空间，本质上是调用 growproc 的过程，所以在 growproc 函数中进行修改，首先判断内存空间拓展后是否会发生越界，如果不越界再根据新增或减少的内存增加或减少内核页表中的映射，如图 11 所示。

```
276   uint sz;
277   struct proc *p = myproc();
278
279   sz = p->sz;
280   // 用户内存不得越界
281   if(sz + n >= PLIC)
282   {
283     printf("growproc out of range\n");
284     return -1;
285   }
286   if(n > 0){
287     if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
288       return -1;
289     }
290     //TODO:开注释
291     for(uint64 va = sz-n; va < sz; va+=PGSIZE)
292     {
293       pte_t *pte = walk(p->pagetable, va, 0);
294       pte_t *kernel_pte = walk(p->kernel_pagetable, va, 1);
295       *kernel_pte = (*pte)&(~PTE_U);
296     }
297   } else if(n < 0){
298     sz = uvmdealloc(p->pagetable, sz, sz + n);
299     // TODO:开注释
300     if(PGROUNDUP(sz) < PGROUNDUP(sz-n)){
301       int npages = (PGROUNDUP(sz-n) - PGROUNDUP(sz)) / PGSIZE;
302       // printf("Unmap user pgtbl in kernel pgtbl\n");
303       uvmunmap(p->kernel_pagetable, PGROUNDUP(sz), npages, 0);
304     }
305   }
306   p->sz = sz;
307   return 0;
308 }
309
310 // Create a new process, copying the parent.
```

图 11 growproc 函数的修改

除此之外，在 userinit 中需要对建立的首个用户页表进行映射，如图 12 所

示。

```
// 将初始进程的唯一一个页表项复制到内核页表 TODO:开注释
pte_t *pte = walk(p->pagetable, 0, 0);
pte_t *kernel_pte = walk(p->kernel_pagetable, 0, 1);
*kernel_pte = (*pte)&(~PTE_U);
```

图 12 userinit 函数修改

三、 实验结果截图

请填写

运行结果如图 13 所示。

```
$ make qemu-gdb
pte printout: OK (8.5s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.9s)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (1.8s)
== Test usertests ==
$ make qemu-gdb
(183.9s)
== 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
```

图 13 程序运行结果