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

实验设计报告

开课学期：大三上

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：2021.11.5 地点：T2210

学生班级：1901105

学生学号：190110509

学生姓名：王铭

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？

为了避免在分配内存后释放部分内容时会出现碎片从而造成资源浪费的情况，引入了页表机制。
分页存储管理将内存分割成大小相同且固定的页，以页作为内存分配的基本单位，解决了外部碎片的问题，虽然在需要的大小不足一页时会出现内部碎片的问题，但相比外部碎片造成的内存浪费是完全可以接受的。同时，分页存储管理在分配内存时能够更加灵活，能支持虚拟地址，只需要建立一个类似目录的数据结构页表，通过虚拟地址查询页表得到页框号，再与偏移量组合即可得到物理地址。

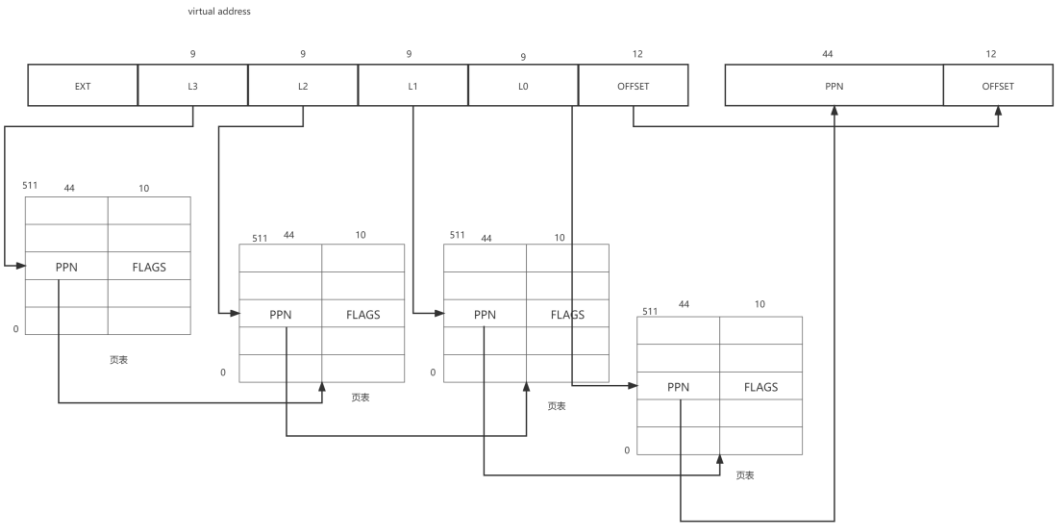
2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0x123456789ABCDEF 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

0x123456789ABCDEF 中的 6789ABC 即 110011110 001001101 010111100
第三级页表索引即为：11 * 16 + 12 = 188
第二级页表索引即为：64 + 13 = 77
第一级页表索引为：256+144+14=414
先查询第三级页表中第 188 个页表项，可以得到第二级页表的物理地址如 0x00 0000 0020 0000，去该物理地址对应的页中查找第 77 个页表项，可以得到第一级页表的物理地址如 0x00 0000 0030 0000，去该物理地址对应的页中查找第 414 个页表项，得到真正的物理页号如 0x00 0000 0040 0000，该物理页的地址与偏移量 0xDEF 组合，即可得到最终的物理地址 0x00 0000 0040 0DEF。

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？

一个页目录的大小为一个页的大小 PGSIZE=4096Bytes，一个页表项大小为 8Bytes，则要索引这些页表项需要 log（4096/8）=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？



二、 实验详细设计

1.vmprint

每个页目录有 512 个索引项，则遍历这 512 个索引项，若其有效位（PTE_V）为 1，则说明该索引项有用，需要打印出来。同时，若该页目录还有下一级目录，则递归打印即可。

```
void
vmprint(pagetable_t pagetable,int level)
{
  printf("page table %p\n",pagetable);
  digui(pagetable,level);
}
```

```
534  void
535  digui(pagetable_t pagetable,int level)
536  {
537      for(int i = 0;i < 512;i++){
538        pte_t* pte = &pagetable[i];
539        if((*pte) & PTE_V){ // 只打印有效的PTE
540          if(level == 1)
541            printf("|| ||");
542          else if(level == 2)
543            printf("||");
544          else
545            printf("|| || ||");
546          uint64 child = PTE2PA((*pte));
547          printf("%d: pte %p pa %p\n",i,pagetable[i],child);
548          if(level > 0)
549            digui((pagetable_t)child,level - 1);
550      }
551    }
552  }
```

2.创建独立内核页表

首先将在 PCB 中添加成员 kernel_pagetable。

在 procinit 函数中申请内核栈，由于此时未创建内核页表，故在 PCB 中加入成员 kk 来记录 kernel_stack 的物理地址，方便在申请内核页表时将内核栈映射到内核页表里。

```
26  procinit(void)
27  {
28    struct proc *p;
29
30    initlock(&pid_lock, "nextpid");
31    for(p = proc; p < &proc[NPROC]; p++) {
32        initlock(&p->lock, "proc");
33
34        // Allocate a page for the process's kernel stack.
35        // Map it high in memory, followed by an invalid
36        // guard page.
37        char *pa = kalloc();
38        if(pa == 0)
39          panic("kalloc");
40        uint64 va = KSTACK((int) (p - proc));
41
42        // 先保留内核栈的虚拟地址
43        p->kstack = va;
44        p->kk = (uint64)pa;
45    }
46    kvminithart();
```

在 allocproc 中申请内核页表，将内核栈映射到内核页表里去：

```
109  found:
110    p->pid = allocpid();
111      // 将内核栈的映射加入到该进程的内核页表中
112    p->kernel_pagetable = mykvminit();
113    mykvmmap(p->kernel_pagetable,p->kstack,p->kk, PGSIZE, PTE_R | PTE_W);
114    // Allocate a trapframe page.
115    if((p->trapframe = (struct trapframe *)kalloc()) == 0){
116      release(&p->lock);
117      return 0;
118    }
```

其中，mykvminit 用于创建并返回一个内核页表，mykvmmap 用于将虚拟地址与物理地

址的映射添加到页表中。定义如下：

```
pagetable_t
mykvminit(){
  pagetable_t pt = (pagetable_t)kalloc();
  memset(pt, 0, PGSIZE);

  // uart registers
  mykvmmap(pt,UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  mykvmmap(pt,VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  mykvmmap(pt,PLIC, PLIC, 0x400000, PTE_R | PTE_W);

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

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

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

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

在 scheduler 进行上下文切换之前，将进程的内核页表载入 satp 寄存器，当 cpu 没有运行任何进程时，调用 kvminithart 函数仍使用原先的内核页表。

```
486         w_satp(MAKE_SATP(p->kernel_pagetable));
487         sfence_vma();
488         swtch(&c->context, &p->context);
489         // Process is done running for now.
490         // It should have changed its p->state before coming back.
491         c->proc = 0; // cpu dosen't run any process now
492         kvminithart();
```

修改 kvmpa 函数，为该函数新添加一个页表参数，用于查找虚拟地址对应的物理地址。

```
uint64
kvmpa(pagetable_t pt,uint64 va)
{
  uint64 off = va % PGSIZE;
  pte_t *pte;
  uint64 pa;
  pte = walk(pt, va, 0);
  if(pte == 0){
    panic("kvmpa");
  }

  if((*pte & PTE_V) == 0){
    panic("kvmpa");
  }
  pa = PTE2PA(*pte);
  return pa+off;
}
```

3.将用户页表的内容添加到内核页表

定义一个拷贝页表索引的函数 mymap，其中 pagetable 是源页表（用户页表），pt 是目的页表（内核页表），va 是要拷贝的起始虚拟地址，size 为结束地址。

通过 walk 函数获取到 va 对应的用户页表的页表项 pte，和在内核页表的相应页表项 pte2，将 pte 的标记位中标明用户使用的标记位置 0，以便内核访问。

```
int
mymap(pagetable_t pagetable,pagetable_t pt,uint64 va,uint64 size){
  for(uint64 i = va;i < size;i += PGSIZE){
    if(i >= MAXVA){
      printf("mymap:no\n");
    }
    pte_t *pte = walk(pagetable,i,0);
    if(((*pte) & PTE_V) == 0) return -1;
    pte_t *pte2 = walk(pt,i,1);
    *pte2 = (*pte) & (~PTE_U);
  }
  return 1;
}
```

利用该函数，在相应添加映射即可

①userinit 中添加映射

```c
void
userinit(void)
{
  struct proc *p;
  p = allocproc();
  initproc = p;

  // allocate one user page and copy init's instructions
  // and data into it.
  uvminit(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;
  mymap(p->pagetable,p->kernel_pagetable,0,PGSIZE);
  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```

②exec

由于 exec 是更换了用户页表，故要先解除原先所有用户页表的映射，然后将新页表的内容拷贝到内核页表中。

```c
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  // 取消之前的映射
  uvmunmap(p->kernel_pagetable,0,PGROUNDDOWN(oldsz)/PGSIZE,0);
  // 添加新的映射
  mymap(p->pagetable,p->kernel_pagetable,0,p->sz);
  proc_freepagetable(oldpagetable, oldsz);

```

③growproc

在扩展/收缩用户页表时，内核页表也要做相应的操作。扩展时，调用 mymap 函数，将新添加的内容拷贝到内核页表。收缩时，调用 uvmunmap 函数取消映射，传入参数 0 表示不释放物理内存。

```c
int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();
  sz = p->sz;
  if(PGROUNDUP(sz + n) >= PLIC) return -1;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
      return -1;
    }
    mymap(p->pagetable,p->kernel_pagetable,p->sz,p->sz + n);
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
    int npages = (PGROUNDUP(p->sz) - PGROUNDUP(p->sz+n)) / PGSIZE;
    uvmunmap(p->kernel_pagetable, PGROUNDUP(p->sz+n), npages, 0);
  }
  p->sz = sz;
  return 0;
```

④fork

将父进程的页表拷贝到内核页表即可。

```
281   // Copy user memory from parent to child.
282   if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
283     freeproc(np);
284     release(&np->lock);
285     return -1;
286   }
287   np->sz = p->sz;
288   np->parent = p;
289   mymap(np->pagetable,np->kernel_pagetable,0,np->sz);
290   // copy saved user registers.
291   *(np->trapframe) = *(p->trapframe);
```

4.释放内核页表

在释放进程时，要释放掉内核页表，首先取消对内核栈的映射，但不释放（因为申请时是在 procinit 里申请的，由 PCB 中的 kk 变量保存，故不能释放）。接着调用释放内核页表的函数 proc_freekerneltable 释放。

```
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->pagetable){
    proc_freepagetable(p->pagetable, p->sz);
  }
  if(p->kernel_pagetable){
    uvmunmap(p->kernel_pagetable,p->kstack,1,0);
    proc_freekerneltable(p->kernel_pagetable,p->sz);
  }
  p->kernel_pagetable = 0;
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

其中 proc_freekerneltable 定义如下，取消内核页表的映射，同时因为是拷贝而不是共用了用户页表，故要调用 freewalk 函数。

```
void
proc_freekerneltable(pagetable_t pt,uint64 sz){
  uvmunmap(pt, TRAMPOLINE, 1, 0);
  uvmunmap(pt,UART0, 1,0);
  uvmunmap(pt,VIRTIO0, 1,0);

  uvmunmap(pt,PLIC, PGROUNDUP(0x400000)/PGSIZE,0);
  uvmunmap(pt,KERNBASE, PGROUNDUP((uint64)etext-KERNBASE)/PGSIZE,0);
  uvmunmap(pt,(uint64)etext, PGROUNDUP(PHYSTOP-(uint64)etext)/PGSIZE,0);
  if(sz > 0)
    uvmunmap(pt, 0, PGROUNDUP(sz)/PGSIZE, 0);
  freewalk(pt);
}
```

三、 实验结果截图

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (6.3s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (0.5s)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (0.5s)
== Test usertests ==
$ make qemu-gdb
(271.5s)
== 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: 100/100
[ming@localhost xv6-labs-2020]$
```