实验四：页表

《操作系统》课程实验

目录

实验目的

- 了解页表的实现原理。
- 修改页表，使内核更方便的进行用户虚拟地址翻译。

实验任务

- 实验开始前，请切换到pgtbl分支：
  - 同步上游仓库：https://hitsz-lab.gitee.io/os-labs-2021/tools/#31
- 本实验有三个任务：
  - **打印页表**
  - **独立内核页表**
  - **简化软件模拟地址翻译**

实验分数说明

1. 实验报告

- 回答实验中的问题 40%
- 给出实验设计思路 60% 1、任务一：30% 2、任务二：20% 3、任务三：10%
- 若对应任务未给出实验设计，那么对应任务代码分记0分。

2. 实验代码

- 1、任务一：50% 2、任务二：30% 3、任务三：20%
- 若对应任务代码无法通过测试，那么实验报告中对应任务的设计分最高只能得到满分的50%。

实验任务一

打印页表 *void vmprint(pagetable_t pgtbl)*

- 该函数将获取一个根页表指针作为参数，然后打印对应的页表数据。
- 该函数一定要插入在exec()的末尾，来打印刚刚载入程序的页表数据。
  - 第一行打印的是vmprint获得的页表参数，之后打印的是页表项，打印的格式为： *$index: pte $pte_bits pa $physical_address*

```
page table 0x0000000087f6e000
||0: pte 0x0000000021fda801 pa 0x0000000087f6a000
|| ||0: pte 0x0000000021fda401 pa 0x0000000087f69000
|| || ||0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
|| || ||1: pte 0x0000000021fda00f pa 0x0000000087f68000
|| || ||2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
||255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
|| ||511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
|| || ||510: pte 0x0000000021fdd807 pa 0x0000000087f76000
|| || ||511: pte 0x0000000020001c0b pa 0x0000000080007000
```

独立内核页表

- **将共享内核页表改成独立内核页表**，使得每个进程拥有自己独立的内核页表。
- 虚实地址相同的映射应该要保留，**先不需要**加上用户页表的内容，在**任务三**中再加上加上用户页表的内容。
- 首先在xv6上运行测试程序kvmtest，然后再运行usertests，确保所有测试通过。

```
$ kvmtest
kvmtest: start
kvmtest: OK
$
```

实验任务三

简化软件模拟地址翻译

- 在独立内核页表**加上用户地址空间**的映射，同时将函数 copyin()/copyinstr()替换成copyin_new()/copyinstr_new()，即将软件模拟地址翻译改成直接访问。
- 首先在xv6上运行测试程序stats stats，然后再运行usertests，确保所有测试通过。

```
$ stats stats
copyin: 62
copyinstr: 12
$
```

实验任务三

- 当完成上述三个实验后，在命令行输入make grade进行测试

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (5.5s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.1s)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (1.0s)
== Test usertests ==
$ make qemu-gdb
(178.8s)
== 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
```

实验原理 | Xv6内核地址空间

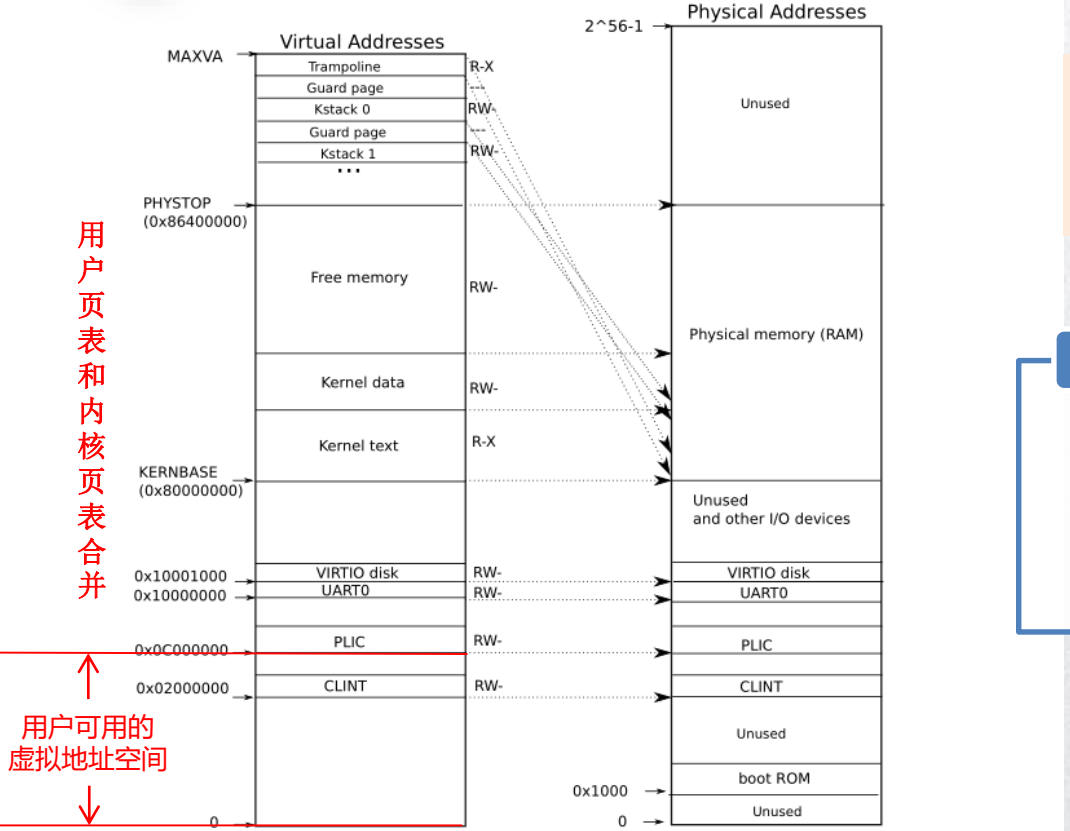


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

来自《xv6 book》

CLINT：Core Local Interruptor本地中断控制器
PLIC：Platform-Level Interrupt controller中断控制器
UART0：Universal Asynchronous Receiver/Transmitter串口
VIRTIO disk：VIRTIO磁盘

用户的可用虚拟地址空间：0~0xC000000

- xv6只有在内核初始化时用到CLINT，因此为用户进程生成内核页表时，可以不必映射这段地址。
- 用户页表是从虚拟地址0开始，用多少建多少，但最高不能超过内核的起始地址（即PLIC地址）

实验原理 | 页表结构

- 页表就是存放虚实地址映射的表格，里面装满了虚实地址的映射。
- xv6采用的页表标准为SV39标准，也就是虚拟地址最多为39位。

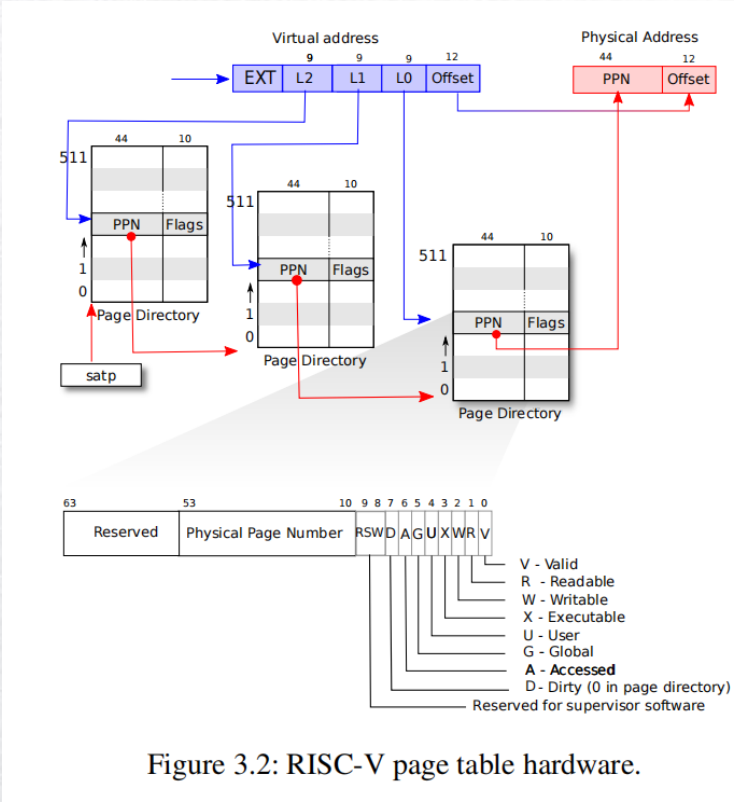


Figure 3.2: RISC-V page table hardware.

实验原理 | 页表结构

- L2：根页表的索引（index），可以得到次页表的基地址。
- L1：次页表的索引，可以得到叶子页表的基地址。
- L0：叶子页表的索引，可以得到64位的页表项。
- Offset：虚实地址的Offset（低12位）完全相同，刚好覆盖1个page中的每个字节。

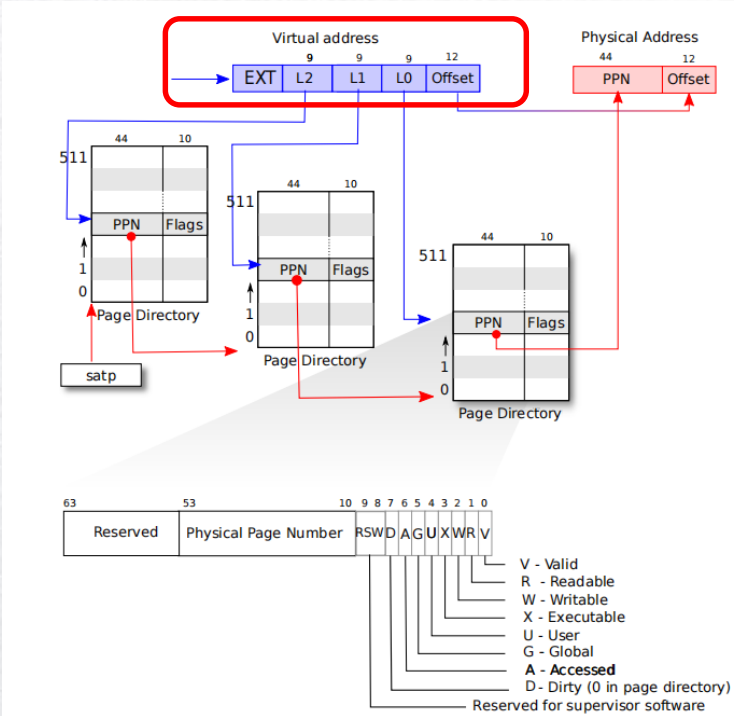


Figure 3.2: RISC-V page table hardware.

实验原理 | 页表结构

- SATP：页表首地址
- PTE：页表项
  - PPN: 物理页帧号
  - Valid: 有效位
  - Readable/ Writable/ Executable：可读/可写/可执行
  - User：该页表项指向的物理页能在用户态访问

根页表是个4K的页，包含了512个PTE，每个PTE都对应了第二级的1个页，每个页都包含了第三级的512个PTE。

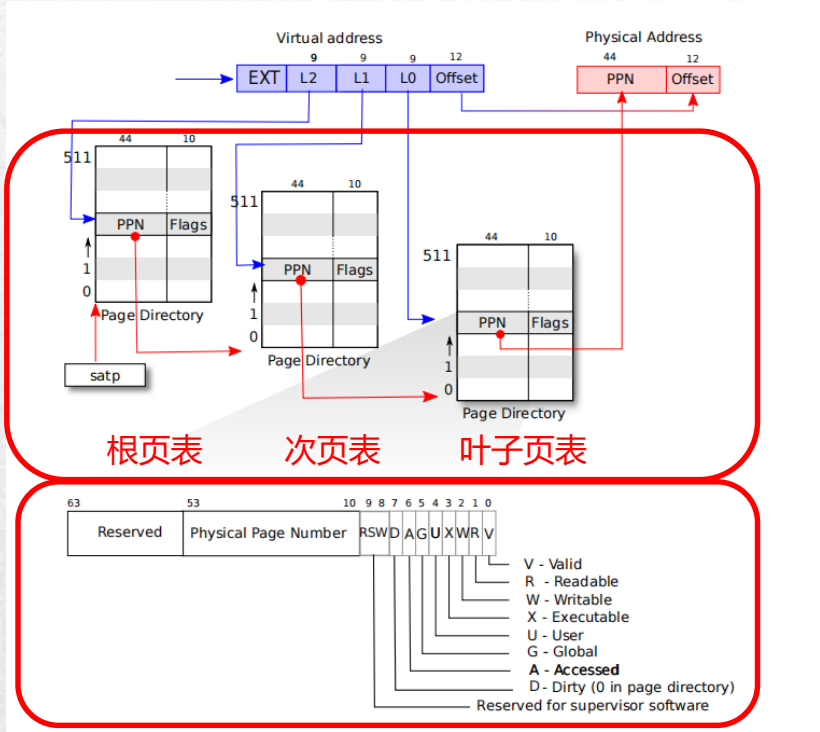


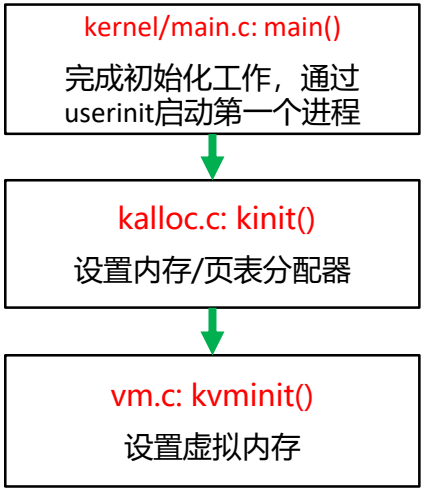
Figure 3.2: RISC-V page table hardware.

实验原理 | 内核页表

- kvminit()/24行: kernel_pagetable **全局的内核根页表的指针**
- Kvmmap() 将物理地址映射到相同的虚拟地址(因为前两个参数一致)
- Kvmmap() ->mappages()对每个要映射的虚拟地址，调用walk()去找到虚拟地址对应的PTE，然后初始化PTE来保存相关物理页号、所需权限Flag等

```
18  /*
19   * create a direct-map page table for the kernel.
20   */
21  void
22  kvminit()
23  {
24    kernel_pagetable = (pagetable_t) kalloc();
25    memset(kernel_pagetable, 0, PGSIZE);
26
27    // uart registers
28    kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);
29
30    // virtio mmio disk interface
31    kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
32
33    // PLIC
34    kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);
35
36    // map kernel text executable and read-only.
37    kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
38
39    // map kernel data and the physical RAM we'll make use of.
40    kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);
41
42    // map the trampoline for trap entry/exit to
43    // the highest virtual address in the kernel.
44    kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
45  }
```

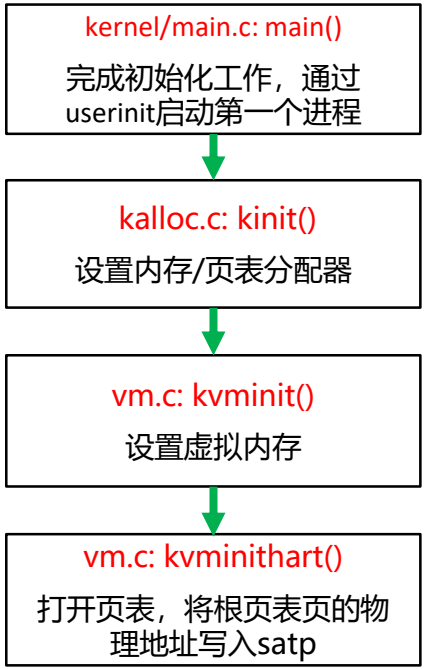
```
121 // Create PTEs for virtual addresses starting at va that refer to
122 // physical addresses starting at pa. va and size might not
123 // be page-aligned. Returns 0 on success, -1 if walk() couldn't
124 // allocate a needed page-table page.
125 int
126 mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
127 {
128   uint64 a, last;
129   pte_t *pte;
130
131   a = PGROUNDDOWN(va);
132   last = PGROUNDDOWN(va + size - 1);
133   for(;;){
134     if((pte = walk(pagetable, a, 1)) == 0)
135       return -1;
136     if(*pte & PTE_V)
137       panic("remap");
138     *pte = PA2PTE(pa) | perm | PTE_V;
139     if(a == last)
140       break;
141     a += PGSIZE;
142     pa += PGSIZE;
143   }
144   return 0;
145 }
```



实验原理 | 内核页表

- 52行: 将kernel_pagetable写入satp
- 53行: 执行sfence.vma指令flush清空TLB(页表缓存)

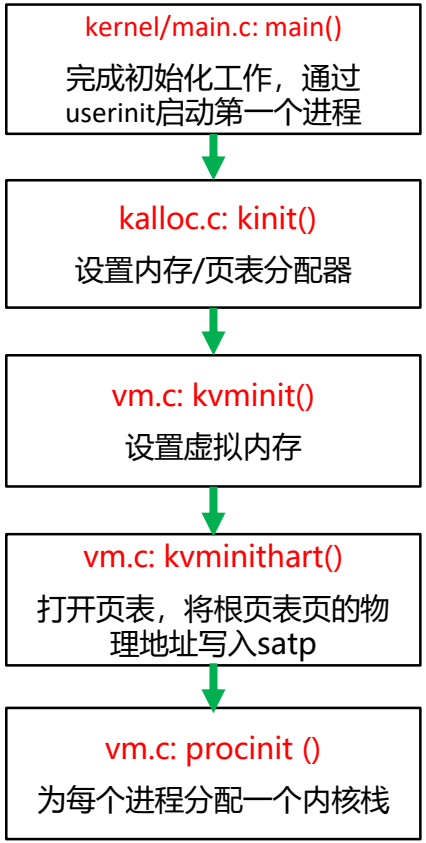
```
47  // Switch h/w page table register to the kernel's page table,
48  // and enable paging.
49  void
50  kvminithart()
51  {
52    w_satp(MAKE_SATP(kernel_pagetable));
53    sfence_vma();
54  }
```



实验原理 | 内核页表

- 41行：在内核页表建立内核栈的映射
- 42行：将内核栈的虚拟地址存储进程控制块PCB
- 44行：kvminithart()重新加载内核页表到satp，让硬件知道新的PTE

```
24  // initialize the proc table at boot time.
25  void
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        kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
42        p->kstack = va;
43    }
44    kvminithart();
45  }
```



实验实现 | 任务1：打印页表

① *void vmprint(*pagetable_t pgtbl*)*

- 把 vmprint() 放在 kernel/vm.c
- vm.c/freewalk() 能帮助你理解遍历页表的过程
- 可以使用 kernel/riscv.h 中尾部的宏定义
- 在 kernel/defs.h 中定义 vmprint() 的接口
- 实现 vmprint()，并在 exec() 函数中插入语句 if(p->pid==1) vmprint(p->pagetable)，这条语句插在 exec.c 中 return argc 代码之前。

```
249  // Recursively free page-table pages.
250  // All leaf mappings must already have been removed.
251  void
252  freewalk(pagetable_t pagetable)
253  {
254    // there are 2^9 = 512 PTEs in a page table.
255    for(int i = 0; i < 512; i++){
256      pte_t pte = pagetable[i];
257      if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
258        // this PTE points to a lower-level page table.
259        uint64 child = PTE2PA(pte);
260        freewalk((pagetable_t)child);
261        pagetable[i] = 0;
262      } else if(pte & PTE_V){
263        panic("freewalk: leaf");
264      }
265    }
266    kfree((void*)pagetable);
267  }
```

vm.c: freewalk()
功能：释放页表
入参pagetable：根页表页的指针

↓

255行
遍历一个页表页的PTE表项

↓

256行
获取第i条PTE

↓

257~261行
判断PTE的Flag位，如果还有下一级页表，则递归调用freewalk释放页表项，并将对应的PTE清零

↓

262~264行
如果是叶子页表（valid为1）的虚拟地址还有映射到物理地址，报错panic。（注：调用freewalk前应该会先uvmunmap释放物理内存）

实验实现 | 任务2：独立内核页表

- 将共享内核页表改成独立内核页表，使得每个进程拥有自己独立的内核页表
- ***参考步骤:***
  - Step1：仿照kvminit函数写一个创建内核页表的函数

    （注：实现的时候不要映射CLINT，否则会发送地址重合问题）
  - Step2：修改kernel/proc.h中的 struct proc，增加新成员

    pagetable_t k_pagetable；在每个进程中保存一个内核独立页表

实验实现 | 任务2：独立内核页表

- Step3：修改procinit函数。procinit()是在系统引导时，用于给进程分配内核栈的物理页并在页表建立映射。

- ***参考优化方法***：把procinit()中内核栈的物理地址存储到PCB，然后在allocproc()中再把它映射到进程的内核页表里。

要保留内核栈在全局页表kernel_pagetable的映射

```
24  // initialize the proc table at boot time.
25  void
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        kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
42        p->kstack = va;
43    }
44    kvminithart();
45  }
```

实验实现 | 任务2：独立内核页表

- Step4：修改allocproc函数。allocproc()会在系统启动时被第一个进程和fork调用。
- ***allocproc函数功能***：在进程表中查找空闲PCB，如果找到，初始化在内核中运行所需的状态，并保持p->lock返回。如果没有空闲PCB，或者内存分配失败，则返回0。
  - 在allocproc创建内核页表以及内核栈

```
88  // Look in the process table for an UNUSED proc.
89  // If found, initialize state required to run in the kernel,
90  // and return with p->lock held.
91  // If there are no free procs, or a memory allocation fails, return 0.
92  static struct proc*
93  allocproc(void)
94  {
95    struct proc *p;
96
97    for(p = proc; p < &proc[NPROC]; p++) {
98      acquire(&p->lock);
99      if(p->state == UNUSED) {
100       goto found;
101     } else {
102       release(&p->lock);
103     }
104   }
105   return 0;
106
107 found:
108   p->pid = allocpid();
109
110   // Allocate a trapframe page.
111   if((p->trapframe = (struct trapframe *)kalloc()) == 0){
112     release(&p->lock);
113     return 0;
114   }
115
116   // An empty user page table.
117   p->pagetable = proc_pagetable(p);
118   if(p->pagetable == 0){
119     freeproc(p);
120     release(&p->lock);
121     return 0;
122   }
```

实验实现 | 任务2：独立内核页表

- Step5：修改调度器中的scheduler()函数，使得切换进程的时候切换内核页表。
- ***参考方法***：在进程切换的同时也要切换页表将其放入寄存器 satp中（一定要借鉴 kvminithart()的页表载入方式）

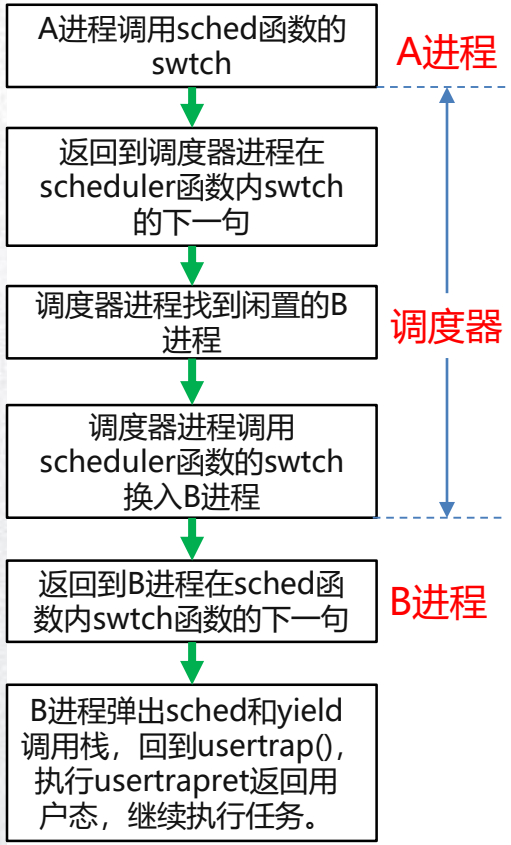
```
458  void
459  scheduler(void)
460  {
461    struct proc *p;
462    struct cpu *c = mycpu();
463
464    c->proc = 0;
465    for(;;){
466      // Avoid deadlock by ensuring that devices can interrupt.
467      intr_on();
468
469      int found = 0;
470      for(p = proc; p < &proc[NPROC]; p++) {
471        acquire(&p->lock);
472        if(p->state == RUNNABLE) {
473          // Switch to chosen proces
474          // to release its lock and
475          // before jumping back to
476          p->state = RUNNING;
477          c->proc = p;
478          swtch(&c->context, &p->context);
479
480          // Process is done running
481          // It should have changed
482          c->proc = 0;
483
484          found = 1;
485        }
486        release(&p->lock);
487      }
488      if(found == 0) {
489        intr_on();
490        asm volatile("wfi");
491      }
492    }
493  }
```

swtch前插入
切换进程内核页表

swtch后插入
全局内核页表

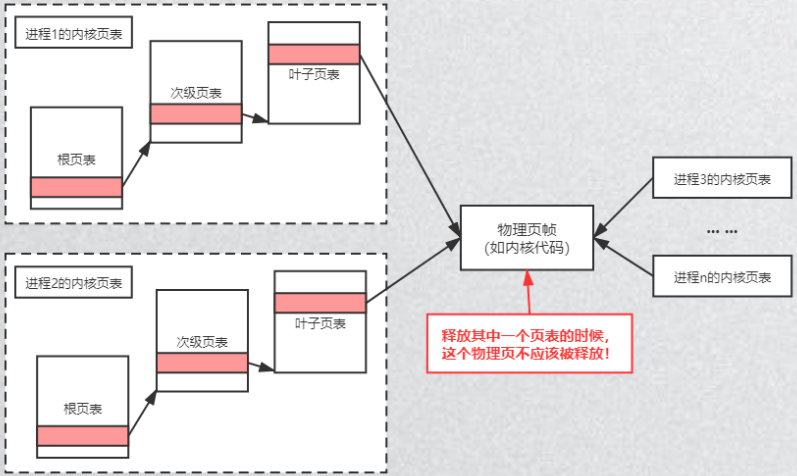
当目前没有进程运行的时候，scheduler() 应该要satp载入全局的内核页表 kernel_pagetable (kernel/vm.c)

两进程切换执行流程



实验实现 | 任务2：独立内核页表

- Step6：修改freeproc()函数，释放对应的内核页表。
- ***参考方法***：你需要找到 **释放页表但不释放叶子页表指向的物理页帧** 的方法。你可参考kernel/vm.c中的freewalk，其用于释放整个页表，但要求叶子页表的页项已经被清空。




```c
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);
  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;
}
```

```c
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}
```

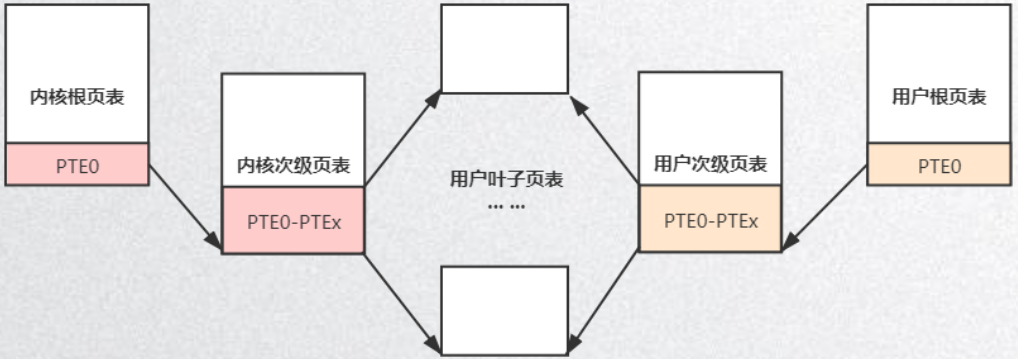
```c
void
uvmfree(pagetable_t pagetable, uint64 sz)
{
  if(sz > 0)
    uvmunmap(pagetable, 0, PGROUNDUP(sz)/PGSIZE, 1);
  freewalk(pagetable);
}
```

```c
void
freewalk(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  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);
      pagetable[i] = 0;
    } else if(pte & PTE_V){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

实验实现 | 任务3：简化软件模拟地址翻译

- 在独立内核页表加上用户页表的映射，同时替换 copyin()/copyinstr()为 copyin_new()/copyinstr_new()
- ***参考步骤:***
  - Step1：写一个XXX函数把进程的用户页表映射到内核页表中，同时在defs.h中声明。
  - 推荐一种较为***优雅的实现方法***：内核页表直接**共享**用户页表的叶子页表，即内核页表中次页表的部分目录**直接指向**用户页表的叶子页表。

提示：前面已经提到用户地址空间的范围为0x0-0xC000000，试计算，多少个次级页表项就能涵盖整个用户地址空间?



实验实现 | 任务3：简化软件模拟地址翻译

- Step2：用函数 copyin_new() （在 kernel/vmcopyin.c中定义）代替 copyin()（在 kernel/vm.c中定义）。确保程序能运行之后再用 copyinstr_new() 以代替 copyinstr()。
- Step3：注意独立内核页表的用户页表的映射的标志位的选择。

  （标志位User一旦被设置，内核就不能访问该虚拟地址了）

  - ***推荐方案：*** 在调用copyin_new()/ copyinstr_new()之前修改sstatus寄存器的SUM位 *w_sstatus(r_sstatus() | SSTATUS_SUM);*

```
356 int
357 copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
358 {
359   uint64 n, va0, pa0;
360
361   while(len > 0){
362     va0 = PGROUNDDOWN(srcva);
363     pa0 = walkaddr(pagetable, va0);
364     if(pa0 == 0)
365       return -1;
366     n = PGSIZE - (srcva - va0);
367     if(n > len)
368       n = len;
369     memmove(dst, (void *)(pa0 + (srcva - va0)), n);
370
371     len -= n;
372     dst += n;
373     srcva = va0 + PGSIZE;
374   }
375   return 0;
376 }
```

```
29  int
30  copyin_new(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
31  {
32    struct proc *p = myproc();
33
34    if (srcva >= p->sz || srcva+len >= p->sz || srcva+len < srcva)
35      return -1;
36    memmove((void *) dst, (void *)srcva, len);
37    stats.ncopyin++;   // XXX lock
38    return 0;
39  }
```

实验实现 | 任务3：简化软件模拟地址翻译

- Step4：在xv6中，涉及到进程页表改变的只有三个地方：fork(), exec(), sbrk()，要将改变后的进程页表同步到内核页表中。
- Step5：注意：第一个用户进程也需要将用户页表映射到内核页表中（kernel/proc.c: userinit()）

实验要点 | 注意事项

- 好好利用 vmprint() 来帮助debug。
- 如果内核缺失了地址映射造成了页缺失（page fault），通常会打印个sepc=0x00000000XXXXXXXX，这代表的是出错时pc的值，你可以查kernel/kernel.asm看看对应地址的代码的含义。
- 请认真阅读实验指导书 http://hitsz-lab.gitee.io/os-labs-2021/lab4/part1/

实验要点 | 作业提交

- 请务必提前自测grade-lab-pgtbl；
- 截止下一次实验课提交，请参考实验二的提交方式；
- **注意查看作业提交截止时间！！！**

THANKS

同学们，
请开始实验吧！