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： http://hitsz-cslab.gitee.io/os-labs/tools/#331
- 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，来打印第一个进程或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
/* 根页表物理地址：0x0000000087f6e000 */
||0: pte 0x0000000021fda801 pa 0x0000000087f6a000
 /* 根页目录项 0，PTE内容为0x0000000021fda801，表示下一级（第二级）页表的PPN物理页帧号为0x87f6a，Flags为0x001，PTE_V有效。该物理页的起始物理地址为0x0000000087f6a000。*/
|| ||0: pte 0x0000000021fda401 pa 0x0000000087f69000
    /*第二级目录项 0， PTE的PPN物理页帧号：0x87f69，Flags为0x001*/
|| || ||0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
       /*第三级目录项 0，其PPN物理页帧号：0x87f6b，Flags为0x01f，物理块起始地址：0x0000000087f6b000 */
|| || ||1: pte 0x0000000021fda00f pa 0x0000000087f68000
       /*第三级目录项 1，其PPN物理页帧号：0x87f68，Flags为0x00f，物理块起始地址：0x0000000087f68000 */
|| || ||2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
       /*第三级目录项 2，其PPN物理页帧号：0x87f67，Flags为0x01f，物理块起始地址：0x0000000087f67000 */
||255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
 /* 根页目录项 255，下一级（第二级）页表的PPN物理页帧号为0x87f6d，Flags为0x001，PTE_V有效。该物理页的起始物理地址为0x0000000087f6d000 */
|| ||511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
    /*第二级目录项511， PPN物理页帧号为0x87f6c，Flags为0x001 */
|| || ||510: pte 0x0000000021fdd807 pa 0x0000000087f76000
       /*第三级目录项 510，其PPN物理页帧号：0x87f76，Flags为0x007，物理块起始地址：0x0000000087f76000 */
|| || ||511: pte 0x0000000020001c0b pa 0x0000000080007000
       /*第三级目录项 511，其PPN物理页帧号：0x80007，Flags为0x00b，物理块起始地址：0x0000000080007000 */
```

实验任务二

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 | 页表结构

页表是什么？

- 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。

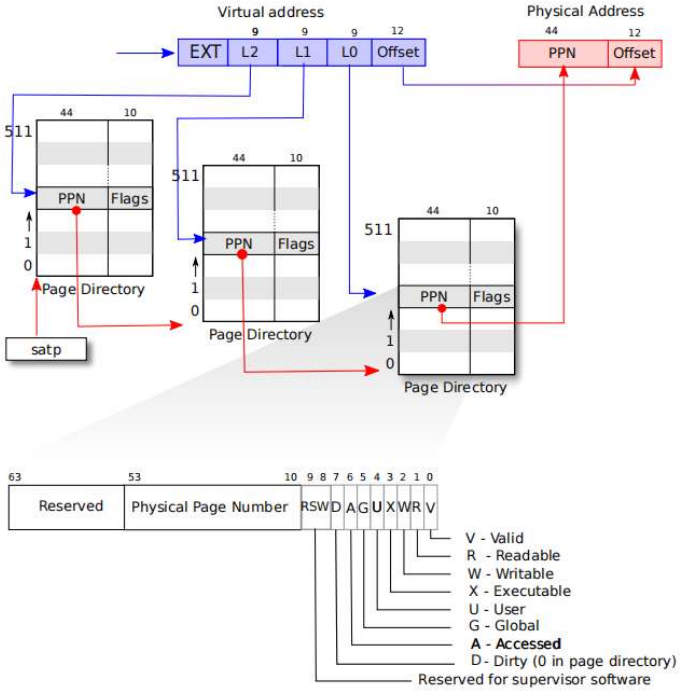


Figure 3.2: RISC-V page table hardware.

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。

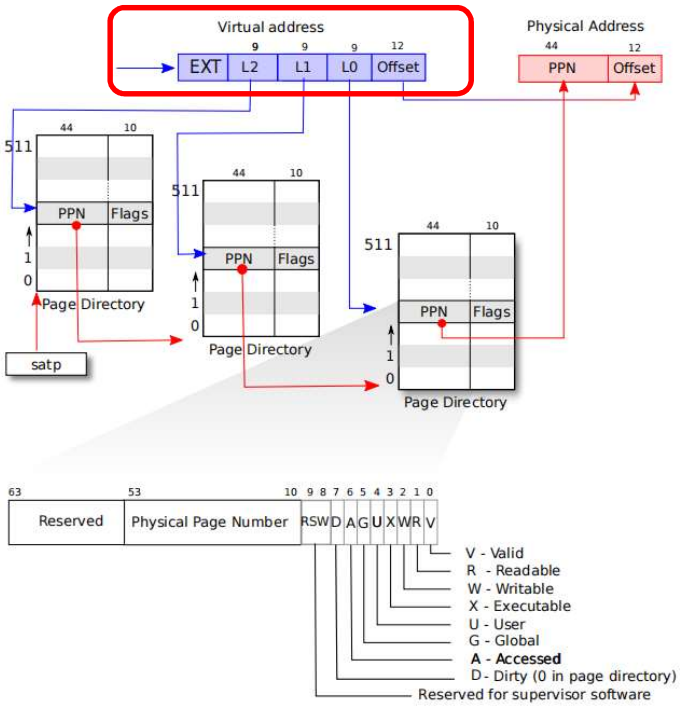


Figure 3.2: RISC-V page table hardware.

实验原理 | 页表结构

- SATP：页表首地址
- PTE：页表项
  - PPN: 物理页帧号
  - Flags标志位
    - 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

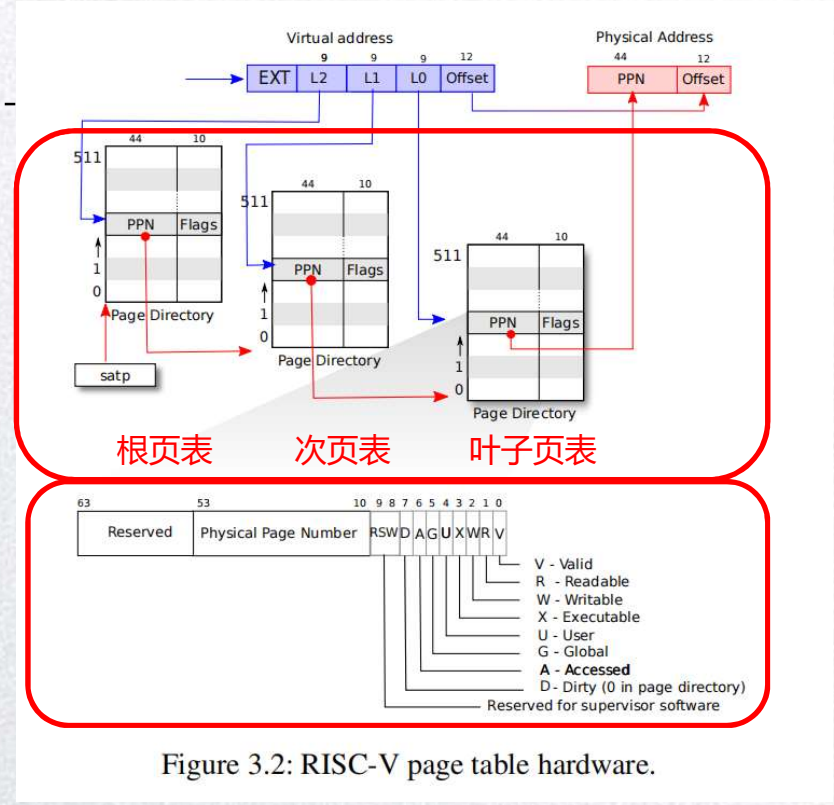


Figure 3.2: RISC-V page table hardware.

在RISC-V指令集中，当需要开启页表服务时，操作系统要将预先配置好的页表首地址放入 satp 寄存器中。从此之后， **计算机硬件** 将把访存的地址 **均视为虚拟地址** ， 都需要通过硬件查询页表，将其 **翻译成为物理地址** ， 然后将其作为地址发送给内存进行访存。

实验原理 | Xv6用户虚拟地址空间

- 范围：从0到MAXVA（256GB）

仅包含自身的代码和数据的虚实地址映射，内核代码和数据不包含在内。

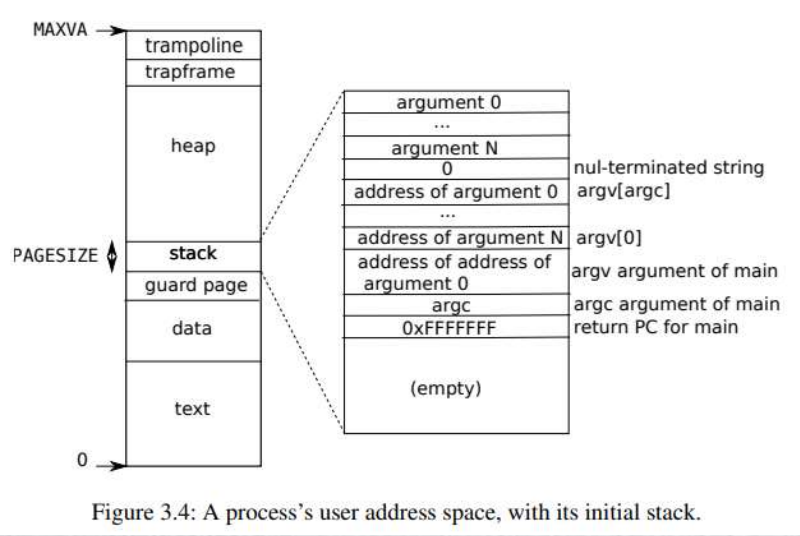


Figure 3.4: A process's user address space, with its initial stack.

- 其代码实现在kernel/exec.c。
- exec使用proc_pagetable分配了TRAMPOLINE和TRAPFRAME的页表映射，然后用uvmalloc来为每个ELF段分配内存及页表映射，并用loadseg把每个ELF段载入内存。

- **trampoline**：用户态-内核态跳板
- **trapframe**：用来存放每个进程的用户寄存器的内存空间
- **heap**：堆
- **stack**：用户栈
- **guard page**：守护页
- **data**：用户程序的数据段
- **text**：用户程序的代码段

实验原理 | Xv6内核虚拟地址空间

- 范围：从0到MAXVA（256GB）

仅包含内核的代码和数据的虚实地址映射，用户程序的代码和数据不包含在内。

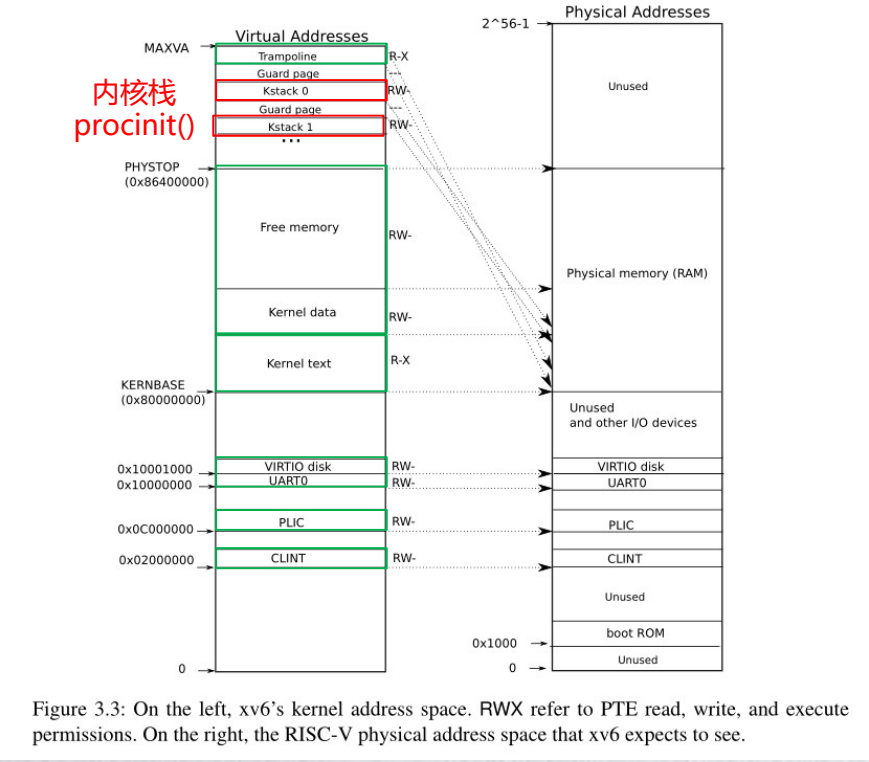


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

- 其代码实现在kernel/vm.c的kvminit()。
- 在kvminit()函数内，除了内核栈外，完成了UART0、VIRTIO0、CLINT、PLIC、kernel text、kernel data和TRAMPOLINE的虚拟地址和物理地址的映射。

什么是软件模拟翻译？

- 由于用户地址的映射并未存储于内核页表，如果我们需要处理用户程序传来的虚拟地址（比如系统调用传入的指针），我们需要先找到用户页表，逐个页表项地找到能够翻译对应虚拟地址的页表项后，才可以获取实际的物理地址并进行访问

实验原理 | 用户页表和内核页表合并

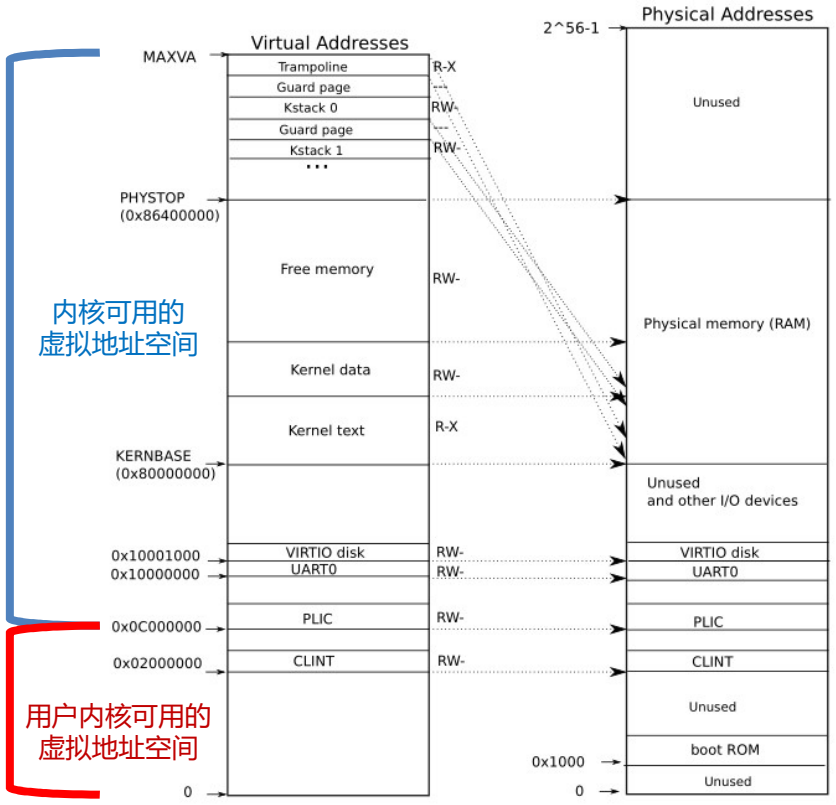


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

来自《xv6 book》

前提：地址不重合

- 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）。
- **内核的可用虚拟地址空间：**
  0xC000000~MAXVA
- **用户的可用虚拟地址空间：**
  0~0xC000000

实验实现 | 任务1：打印页表

*void vmprint(*pagetable_t pgtbl*)*

- vm.c/freewalk()释放页表函数 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。

```
// Recursively free page-table pages.
// All leaf mappings must already have been removed.
275 void
276 freewalk(pagetable_t pagetable)
277 {
278  // there are 2^9 = 512 PTEs in a page table.
  // 遍历一个页表页的PTE表项
279  for(int i = 0; i < 512; i++){
280    pte_t pte = pagetable[i]; //获取第i条PTE

    /*判断PTE的Flag位，如果还有下一级页表(即当前是根页表或次页表)，
    则递归调用freewalk释放页表项，并将对应的PTE清零*/
281    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
282      // this PTE points to a lower-level page table.
283      uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
284      freewalk((pagetable_t)child); // 递归调用freewalk
285      pagetable[i] = 0; // 清零
286    } else if(pte & PTE_V){
    /*如果叶子页表的虚拟地址还有映射到物理地址，报错panic。
    因为调用freewalk之前应该会先uvmunmap释放物理内存*/
287      panic("freewalk: leaf");
288    }
289  }
290  kfree((void*)pagetable); // 释放pagetable指向的物理页
291 }
```

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：修改kernel/proc.h中的 struct proc， 增加两个新成员：
    内核独立页表k_pagetable; 内核栈的物理地址kstack_pa;，
  - Step2：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，否则会发生地址重合问题。

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pa拷贝到PCB新增的成员kstack_pa中，同时还需要保留内核栈在全局页表kernel_pagetable的映射，然后在allocproc()中再把它映射到进程的内核页表里。

```
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  }
```

关于内核栈映射

注意，要保证在每一个进程的内核页表中映射该进程的内核栈。xv6本会在 procinit() 中分配内核栈的物理页并在页表建立映射。但是现在，应该在allocproc()中实现该功能，因为执行procinit()的时候进程的内核页表还未被创建。你可以在procinit()中只保留内存的分配，但在allocproc()中完成映射。

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创建内核页表以及映射内核栈
  - 调用Step2创建的内核页表映射函数
  - 调用kvmmap映射PCB新增的成员kstack_pa

```
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

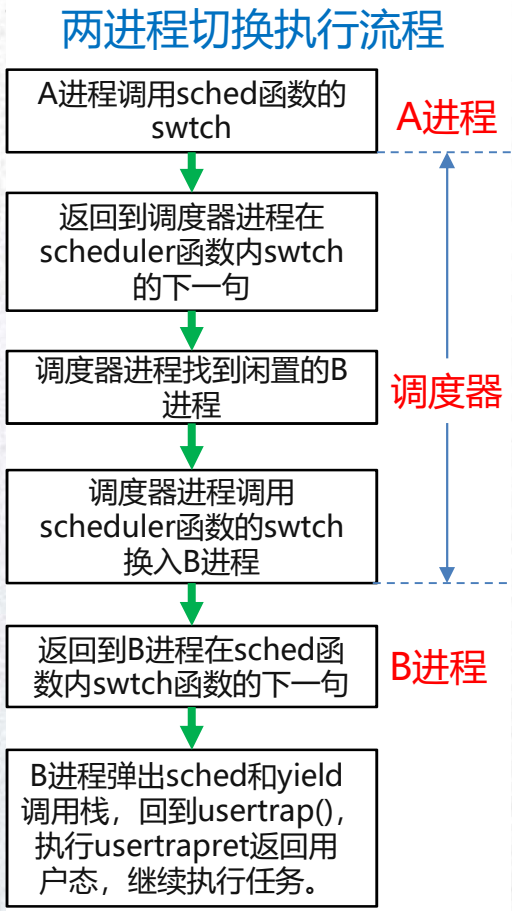
- Step5：修改调度器中的scheduler()函数，使得切换进程的时候切换内核页表。
- ***参考方法***：在进程切换的同时也要切换页表将其放入寄存器 satp中（一定要借鉴 kvminithart()的页表载入方式）

```
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函数前插入切换进程内核页表

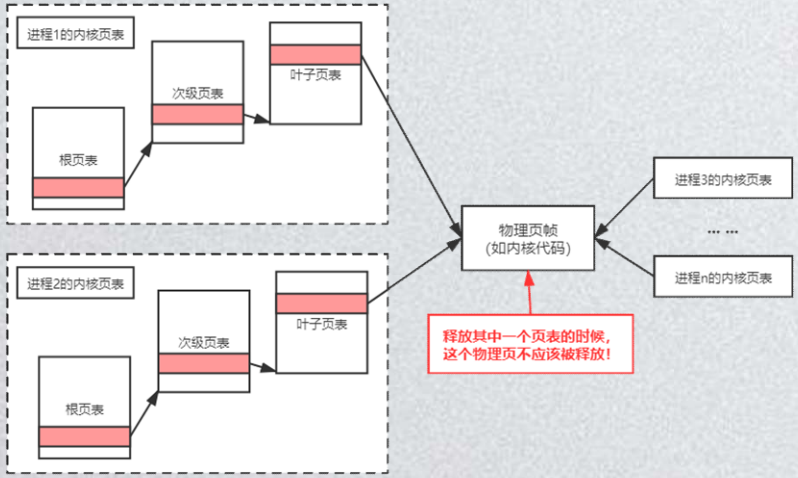
Swtch函数后插入全局内核页表

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)



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***：在各个内核页表的叶子页表中，页表项指向了共享的物理页。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。



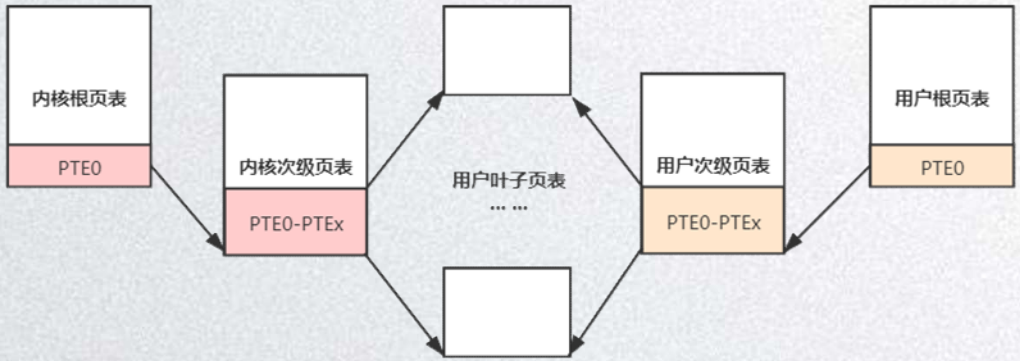

```
136  static void
137  freeproc(struct proc *p)
138  {
139    if(p->trapframe)
140      kfree((void*)p->trapframe);
141    p->trapframe = 0;
142    if(p->pagetable)
143      proc_freepagetable(p->pagetable, p->sz);
144    p->pagetable = 0;
145    p->sz = 0;
146    p->pid = 0;
147    p->parent = 0;
148    p->name[0] = 0;
149    p->chan = 0;
150    p->killed = 0;
151    p->xstate = 0;
152    p->state = UNUSED;
153  }
```

```
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  }
```

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);*
    之后去掉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-cslab.gitee.io/os-labs/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！