一、页表是如何翻译地址的？(va->pa)

软件翻译：walkaddr(pagetable,va)

何处使用？copyin()或copyout()或copyinstr()

1-kernel/vm.c：‘copyin()’或‘copyout()’或‘copyinstr()’

```
int copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len);
int copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max);
int copyout(pagetable_t pagetable, uint64 dstva/* 目标虚拟地址*/, char *src,
uint64 len)
{
  uint64 n, va0, pa0;
  while(len > 0){
    va0 = PGROUNDDOWN(dstva); // 将目标va向下取整，获得va所在的虚拟页的起始地址
    pa0 = walkaddr(pagetable, va0); // 获取该虚拟页 对应的物理页的起始地址
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
    // 下略....

  return 0;
}
```

2-kernel/vm.c->kernel/vm.c：‘walkaddr()’

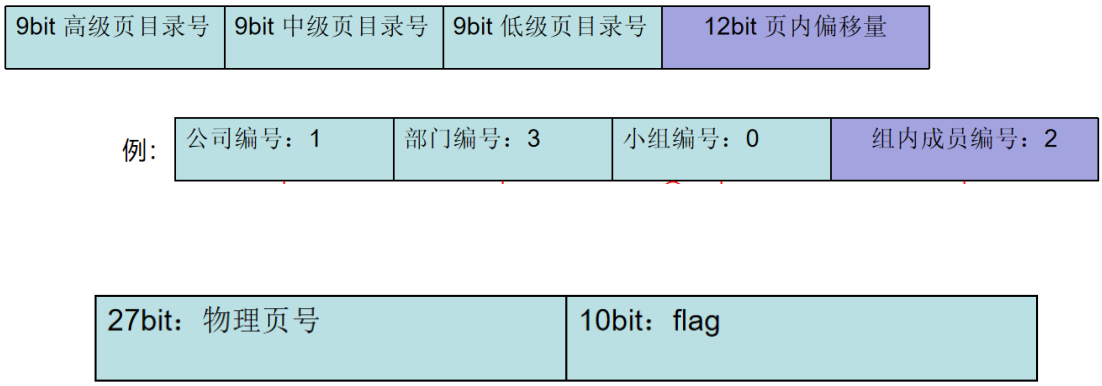
```
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;

  if(va >= MAXVA) // 检查虚拟地址是否超出可用范围
    return 0;

  pte = walk(pagetable, va, 0); // 获取va所对应的虚拟页的pte的地址(回看PPT)
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  if(pte == 0)
    return 0;
  if((*pte & PTE_V) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte); // *pte是pte条目的具体内容，根据pte的具体内容找到目标物理页面的
起始地址 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  return pa;
}
```

2-1kernel/vm.c->kernel/vm.c：‘walk()’

```c
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index. 高级页目录
//   21..29 -- 9 bits of level-1 index. 中级页目录
//   12..20 -- 9 bits of level-0 index. 低级页目录
//    0..11 -- 12 bits of byte offset within the page. 页内偏移量
pte_t *  // 获取va所对应的虚拟页的pte的地址
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA) // 先检查va是否越界
    panic("walk");

  for(int level = 2; level > 0; level--) { // 再访问三级页表的前两级，从高级页目录开始访问(level=2) (结合PPT)
    pte_t *pte = &pagetable[PX(level, va)]; // 获取下一级页目录的映射信息(pte)所在的地址 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte); // 根据pte中的信息，获取下一级页目录的所在地址，并将其作为新的页表↓↓↓↓↓↓↓
    } else { // 若va还没被映射到内存中，则现场创建映射
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)]; // 这里的pagetable是最低级的页目录
}
```

2-1-1kernel/vm.c->kernel/riscv.h：‘PX()’和‘PTE2PA()’

```c
//看下图1
#define PGSHIFT 12  // 页内偏移量是12bit
#define PXMASK          0x1FF // 由于每级页目录中有512(2^9)个条目，所以每级页目录号有9 bits,0x 1FF=1 1111 1111
#define PXSHIFT(level)  (PGSHIFT+(9*(level))) // 计算取出第level级页目录号时，需要右移多少位
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK) // 先右移，此时第level级页目录号就在最低9bit，再& 0x 1FF 操作相当于取出最低的9bit，其他bit全部清零

// 看下图2
// pte中的最低10bit全是flag位 0x3FF= 11 1111 1111
#define PTE_FLAGS(pte) ((pte) & 0x3FF) // & 0x3FF 操作相当于取出最低的10bit，其他bit全部清零
#define PA2PTE(pa) ((((uint64)pa) >> 12) << 10) // 获取一个物理地址后，先右移12bit(抹掉页内偏移量),再左移10bit 给flag留空间


```



3-kernel/vm.c->kernel/riscv.h：‘PTE2PA()’

```
// 看上图
#define PTE2PA(pte) (((pte) >> 10) << 12) // 获取一个pte后，先右移10bit把flag抹掉，得到物理页框号，再左移12bit得到该物理页的起始物理地址
```

硬件翻译：MMU

何处使用？：正常访问数据(va)时，硬件就会隐式工作，比软件翻译快很多！

二、内存布局：kernel/memlayout.h(结合PPT)

QEMU的物理内存布局

- **00001000 -- boot ROM**：QEMU提供的启动ROM，用于启动过程。
- **02000000 -- CLINT**：本地中断控制器（Core Local Interruptor），包含定时器等功能。
- **0C000000 -- PLIC**：可编程中断控制器（Platform-Level Interrupt Controller），用于管理来自外设的中断。
- **10000000 -- uart0**：UART 0设备的寄存器地址，用于串行通信。
- **10001000 -- virtio disk**：VirtIO磁盘设备的内存映射I/O接口。
- **80000000 -- kernel entry/kernel base**：QEMU的boot ROM跳转至此地址以启动内核，`-kernel`选项加载的内核镜像也放置在这里。
- **unused RAM after 80000000**：`0x80000000`之后是未使用的RAM，供内核和用户程序使用，xv6只允许后续的128MB内存空间。

xv6的物理内存使用

- **80000000 -- entry.S, then kernel text and data**：内核的入口点（`entry.S`）以及内核的代码和数据部分。
- **end -- start of kernel page allocation area**：`end`符号之后是内核开始分配页面的区域。
- **PHYSTOP -- end RAM used by the kernel**：`PHYSTOP`定义了内核使用的RAM的结束地址，通常设置为`KERNBASE + 128MB`（假设有128MB的RAM供内核使用）。
-

三、xv6是如何跑起来的?

1-kernel/entry.s（结合PPT）

```
# qemu -kernel loads the kernel at 0x80000000 and causes each CPU to jump there.
# kernel.ld causes the following code to be placed at 0x80000000.
.section .text # 这行代码指示接下来的代码应该被放置在程序的文本（代码）段中。
_entry: # 这是程序的入口点标签，QEMU或其他引导程序会将CPU的PC设置到这个地址（0x80000000），然后从这里开始执行。

    la sp, stack0 # 将stack0的地址加载到堆栈指针（sp）寄存器中。stack0是在start.c中定义的，用于存放每个CPU的堆栈基地址。
    li a0, 1024*4 # 将4096（即1024*4）加载到寄存器a0中，这里4096是每个CPU堆栈的大小（以字节为单位）。
    csrr a1, mhartid # 从CSR（控制和状态寄存器）中读取mhartid的值到寄存器a1。mhartid是硬件线程ID，用于区分多核处理器中的不同CPU核心。
    addi a1, a1, 1 # cpu的数量=mhartid+1
    mul a0, a0, a1 # 将a0（堆栈大小）和a1相乘，结果存放在a0中。这一步计算了每个CPU核心堆栈的偏移量。
    add sp, sp, a0 # 将sp（堆栈基地址）和a0（堆栈偏移量）相加，得到当前CPU核心的堆栈顶地址，并更新sp寄存器。
    # jump to start() in start.c
    call start # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

2-kernel/entry.s -> kernel/start.c：‘call start’

```
// entry.S jumps here in machine mode
__attribute__ ((aligned (16))) char stack0[4096 * NCPU];
void
start()
{
  // 将特权模式（MPP, Machine Previous Privilege）设置为监督模式（Supervisor Mode）
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // 将mepc（机器模式异常程序计数器）寄存器设置为main函数的地址。如果执行mret指令，它将跳转到main函数
  w_mepc((uint64)main); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);
  w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);

  // ask for clock interrupts.
  timerinit();
```

```
25
26    // keep each CPU's hartid in its tp register, for cpuid().
27    int id = r_mhartid();
28    w_tp(id);
29
30    // switch to supervisor mode and jump to main().
31    asm volatile("mret");
32  }
```

3-kernel/start.c -> kernel/main.c：'w_mepc((uint64)main)'

```
1   // start() jumps here in supervisor mode on all CPUs.
2   void
3   main()
4   {
5     if(cpuid() == 0){
6       consoleinit();
7       printfinit();
8       printf("\n");
9       printf("xv6 kernel is booting\n");
10      printf("\n");
11      kinit();         // 初始化空闲链表 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
12      kvminit();       // 初始化内核页表，完成直接映射 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
13      kvminithart();   // 初始化页表寄存器 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
14      procinit();      // 初始化每个进程的kstack ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
15      trapinit();      // 陷阱初始化*
16      trapinithart();  // 陷阱初始化*
17      plicinit();      // 中断初始化*
18      plicinithart();  // 中断初始化*
19      binit();         // 文件系统相关初始化*
20      iinit();         // 文件系统相关初始化*
21      fileinit();      // 文件系统相关初始化*
22      virtio_disk_init(); // 模拟磁盘初始化*
23      userinit();      // 第一个用户进程
24      __sync_synchronize();
25      started = 1;
26    } else {
27      while(started == 0)
28        ;
29      __sync_synchronize();
30      printf("hart %d starting\n", cpuid());
31      kvminithart();    // turn on paging
32      trapinithart();   // install kernel trap vector
33      plicinithart();   // ask PLIC for device interrupts
34    }
35
36    scheduler();        // 开始调度 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
37  }
38
```

3-1kernel/main.c->kernel/kalloc.c：'kinit()'

xv6的物理内存使用

- **80000000 -- entry.S, then kernel text and data**：内核的入口点（`entry.S`）以及内核的代码和数据部分。
- **end -- start of kernel page allocation area**：`end`符号之后是内核开始分配页面的区域。
- **PHYSTOP -- end RAM used by the kernel**：`PHYSTOP`定义了内核使用的RAM的结束地址，通常设置为`KERNBASE + 128MB`（假设有128MB的RAM供内核使用）。

```
void
kinit() // 初始化物理空间
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP); // 将能用的物理空间都free掉
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start); // 将start地址转换为页的起始地址，若start不是物理页的起始地址，如：0x100 ，我们应该从它的下一页开始free
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}

// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa; // 将闲置的物理(4KB)页，作为struct run 结构体，来存放空闲链表的信息

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}

```

3-2kernel/main.c->kernel/vm.c：‘kvminit()’+‘kvminithart()’

QEMU的物理内存布局

- **00001000 -- boot ROM**：QEMU提供的启动ROM，用于启动过程。
- **02000000 -- CLINT**：本地中断控制器（Core Local Interruptor），包含定时器等功能。
- **0C000000 -- PLIC**：可编程中断控制器（Platform-Level Interrupt Controller），用于管理来自外设的中断。
- **10000000 -- uart0**：UART 0设备的寄存器地址，用于串行通信。
- **10001000 -- virtio disk**：VirtIO磁盘设备的内存映射I/O接口。
- **80000000 -- kernel entry**：QEMU的boot ROM跳转至此地址以启动内核，`-kernel`选项加载的内核镜像也放置在这里。
- **unused RAM after 80000000**：`0x80000000`之后是未使用的RAM，供内核和用户程序使用。

xv6的物理内存使用

- **80000000 -- entry.S, then kernel text and data**：内核的入口点（`entry.S`）以及内核的代码和数据部分。
- **end -- start of kernel page allocation area**：`end`符号之后是内核开始分配页面的区域。
- **PHYSTOP -- end RAM used by the kernel**：`PHYSTOP`定义了内核使用的RAM的结束地址，通常设置为`KERNBASE + 128MB`（假设有128MB的RAM供内核使用）。

在操作系统中，内核页表中的空间分为两部分：直接映射区和非直接映射区

- 直接映射区的虚拟空间布局完全与物理内存布局一样，va=pa包括：
    - qemu物理布局区
    - xv6物理区（kernel base~phystop）
- 非直接映射区，则是采用将指定va映射到随机申请/指定的物理空间的方式，如：
    - trampoline、kstack

  从低到高分别是：
    - qemu物理布局区
    - xv6物理区（kernel base~phystop）
    - ktack
    - trampoline

```
void // 内核对于页面的映射是采用直接映射的方式：pa即va，此处则是进行直接映射
kvminit()
{
  // kernel_pagetable变量的地址在内核数据区，但它是个指针，它指向的地址属于自由分配区
  kernel_pagetable = (pagetable_t) kalloc();
  memset(kernel_pagetable, 0, PGSIZE);

  // 将qemu物理布局直接映射到内核页表 看ppt
  kvmmap(UART0, UART0, PGSIZE, PTE_R | PTE_W);

  kvmmap(VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  kvmmap(CLINT, CLINT, 0x10000, PTE_R | PTE_W);

  kvmmap(PLIC, PLIC, 0x400000, PTE_R | PTE_W);
```

```
16
17    kvmmap(KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
18
19    // etext就是end，所有的内核初始代码和静态变量都放在kernbase-end之间
20    kvmmap((uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);
21
22    // trampoline.S的代码，属于非直接映射区 看ppt
23    kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
24  }
25
26  void // 页表的地址存放在satp寄存器中，所以切换页表的过程就是更改satp寄存器中数据的过程
27      // 一开始satp=0,不会开启分页，也就是说不会通过页表进行地址翻译工作，此处给satp赋值，
开启分页了
28  kvminithart()
29  {
30    w_satp(MAKE_SATP(kernel_pagetable)); // 内存管理magic时刻！！！！此前，访问地址
都是物理地址，此后访问地址是虚拟地址->物理地址
31    sfence_vma(); // 刷新tlb
32  }
```

3-3kernel/main.c ->kernel/proc.c：‘procinit()’

```
1   void // 初始化每个进程的kstack 看ppt
2   procinit(void)
3   {
4     struct proc *p;
5
6     initlock(&pid_lock, "nextpid");
7     for(p = proc; p < &proc[NPROC]; p++) { // 遍历每个进程的pcb
8         initlock(&p->lock, "proc");
9
10        char *pa = kalloc(); // kstack采用的是非直接映射，所以pa由kalloc()申请而来
11        if(pa == 0)
12          panic("kalloc");
13        uint64 va = KSTACK((int) (p - proc)); // (p-proc)代表着目前是第几个进程，
此处获取第(p - proc)个进程的kstack的虚拟地址
14        kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W); // 映射kstack
15        p->kstack = va;
16    }
17    kvminithart();
18  }
```

3-4kernel/main.c->kernel/proc.c：‘userinit()’

trapframe:

这段代码定义了一个名为 `trapframe` 的结构体，它用于在 RISC-V 架构的操作系统中保存和恢复进程在执行过程中被陷阱（trap）中断时的上下文信息。这个结构体特别用于 `trampoline.S` 汇编文件中的陷阱处理代码，以及用户态和内核态之间的上下文切换。

`trapframe` 结构体包含了多个字段，每个字段都对应着进程在执行过程中需要保存或恢复的寄存器或状态信息。这些字段包括：

- `kernel_satp`：内核页表基址寄存器（SATP）的值，用于在用户态和内核态之间切换页表。
- `kernel_sp`：内核栈顶指针，指向进程内核栈的顶部。
- `kernel_trap`：陷阱处理函数的地址，当从用户态陷入内核态时，会跳转到这个函数执行。
- `epc`：异常程序计数器（EPC），保存了发生陷阱时的用户程序计数器（PC）值。
- `kernel_hartid`：内核的硬件线程ID（hartid），通常保存在线程指针（TP）寄存器中，但在用户态和内核态之间切换时可能需要保存和恢复。

接下来是用户态寄存器的保存区域，包括：

- `ra`：返回地址寄存器。
- `sp`：堆栈指针。
- `gp`：全局指针。
- `tp`：线程指针（在用户态下可能不使用，但在某些情况下需要保存）。
- `t0` 到 `t6`：临时寄存器。
- `s0` 到 `s11`：保存寄存器，用于保存跨函数调用的寄存器值。
- `a0` 到 `a7`：函数参数/返回值寄存器。

这些寄存器在用户态被陷阱中断时，其值会被保存到 `trapframe` 结构体中相应的字段里。当陷阱处理完成后，如果需要返回到用户态继续执行，这些**值会被从 `trapframe` 中恢复回相应的寄存器中。**

需要注意的是，`trapframe` 结构体**位于用户页表的某个页面**中，但在**内核页表中并不特别映射。**

```
void // 设置第一个应用进程的代码内容、指针等
userinit(void)
{
  struct proc *p;

  p = allocproc(); // 第一个应用进程 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  initproc = p;

  uvminit(p->pagetable, initcode, sizeof(initcode)); // 将initcode.S的内容映射到页表中(从地址0开始↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  p->sz = PGSIZE;

  p->trapframe->epc = 0;      // 在返回用户态时，epc的值会被复制到pc中，pc就会从该处开始执行。将epc设置为0，这样该程序在可执行后，就会执行上面的initcode代码了
  p->trapframe->sp = PGSIZE;  // user stack pointer

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

  p->state = RUNNABLE; // 允许第一个应用进程执行

  release(&p->lock);
}


// ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
// 向内核申请一个新进程，并返回该进程的pcb
static struct proc*
allocproc(void)
{
```

```c
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) { // 遍历进程队列，如果找到了unused的空闲pcb，则执行found
    acquire(&p->lock); // 获取相关的锁
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock); // 释放相关的锁
    }
  }
  return 0;

found:
  p->pid = allocpid(); // 获取pid 进程号

  if((p->trapframe = (struct trapframe *)kalloc()) == 0){ // 向内核申请页面给该进程的trapframe，但不映射到内核页表中
    release(&p->lock); // 释放相关的锁
    return 0;
  }

  p->pagetable = proc_pagetable(p); // 为该进程申请页表 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  if(p->pagetable == 0){
    freeproc(p); // 如果页表创建失败，则释放该进程
    release(&p->lock); // 释放相关的锁
    return 0;
  }
  // 如果进程从用户态进入内核态，则会把数据存到trapframe里
  // 但如果是切换进程(进程A切换到进程B)，则A要把自己的寄存器数据都放在p->context里
  memset(&p->context, 0, sizeof(p->context)); // 清空新进程的上下文数据
  p->context.ra = (uint64)forkret; // 进程被调度时，会回到上一次还没执行的代码处继续执行(ra)，而刚刚被创建的进程没有上一次的记录，所以需要给第一次执行的进程一个初始化的函数--forkret
  p->context.sp = p->kstack + PGSIZE; // 初始化内核栈顶

  return p;
}
// ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.
  pagetable = uvmcreate(); // 申请一块物理空间存放页表
  if(pagetable == 0)
    return 0;

  if(mappages(pagetable, TRAMPOLINE, PGSIZE, // 用户页表和内核页表中，trampoline都被映射在同样的va（TRAMPOLINE处）看ppt
              (uint64)trampoline, PTE_R | PTE_X) < 0){
```

```
    uvmfree(pagetable, 0);
    return 0;
  }

  if(mappages(pagetable, TRAPFRAME, PGSIZE, // 将 p->trapframe映射在用户页表中（TRAPFRAME处），不映射到内核页表 看ppt
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
  return pagetable;
}

```

3-5kernel/main.c->kernel/proc.c：‘scheduler()’

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) { // 找到一个可执行的进程
        p->state = RUNNING; // 将其改成正在执行的状态并运行它
        c->proc = p; // 将当前cpu控制的pcb改成该进程的pcb
        swtch(&c->context, &p->context); // 将目标进程之前的数据写回当前cpu的寄存器 ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;

        found = 1;
      }
      release(&p->lock);
    }
    if(found == 0) { // 若没找到可执行的进程 则进入低功耗状态
      intr_on();
      asm volatile("wfi"); // 使用asm volatile("wfi");（Wait For Interrupt）指令让CPU进入低功耗状态，等待中断发生。
    }
  }
}
```

3-5-1kernel/proc.c->kernel/swtch.S：‘swtch(&c->context, &p->context)’

```
# 将现在的上下文存到old context处，再从new context处读取下一个进程的上下文 读到寄存器中，这样就可以执行新的进程了
# 由于allocproc()中 将新进程的ra设置为了forkret()，而执行最后一行的ret后会将ra的值赋给PC，PC就能从forkret()处继续执行了
.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓


```

4-kernel/swtch.S->kernel/proc.c：‘ret’ (userinit()创造的进程——系统的第一个用户进程)

```
// 子进程在调度器（scheduler）首次调度它时的一些初始化工作。
void
forkret(void)
{
  static int first = 1;

```

```
  // Still holding p->lock from scheduler.
  release(&myproc()->lock);

  if (first) { //
    first = 0;
    fsinit(ROOTDEV);
  }
  // 所有程序被首次调度时都必须做这件事
  usertrapret(); // 从内核态返回到用户态，并且恢复用户程序执行前的状态（如寄存器值、堆栈等）。↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
}
```

5-kernel/pro.c->kernel/trap.c：‘usertrapret()’(userinit()创造的进程——系统的第一个用户进程)

```
void
usertrapret(void)
{
  struct proc *p = myproc();

  intr_off();

  // 在发生陷阱时,CPU会将STVEC寄存器的值复制到PC（Program Counter）中，此处是将trampoline.S中的uservec的地址写入
  // 如此，之后发生陷阱时，就会直接跳转到trampoline.S的uservec开始执行
  w_stvec(TRAMPOLINE + (uservec - trampoline));

  p->trapframe->kernel_satp = r_satp();         // 即将从内核态返回用户态，保存内核页表
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // kstack为内核栈的底部地址，由于栈是向下增长的，该底部地址实际为guard page的起始地址，+ PGSIZE后才是栈的顶部（起始）地址
  p->trapframe->kernel_trap = (uint64)usertrap; // 设置陷阱处理函数，在trampoline.S中使用到
  p->trapframe->kernel_hartid = r_tp();         // 保存CPU号



  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // 将ssp-bit设置为0，准备进入用户态
  x |= SSTATUS_SPIE;
  w_sstatus(x);


  w_sepc(p->trapframe->epc); // 将进入内核态之前的用户PC写入sepc，返回用户态后，系统会将sepc的值复制到pc里，用户就会返回之前的位置继续执行

  // 将用户页表放在satp变量中
  uint64 satp = MAKE_SATP(p->pagetable);

  uint64 fn = TRAMPOLINE + (userret - trampoline); // fn被设置为trampoline.S中userret的函数地址，也就是一个函数指针
```

```
  ((void (*)(uint64,uint64))fn)(TRAPFRAME, satp); // 等效于执行
userret(TRAPFRAME, satp)↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
}
```

6-kernel/trap.c->kernel/trampoline.S：‘userret(TRAPFRAME, satp)’(userinit()创造的进程——系统的第一个用户进程)

```
.globl userret
userret:
        # userret(TRAPFRAME, pagetable)
        # switch from kernel to user.
        # usertrapret() calls here.
        # a0: TRAPFRAME, in user page table.
        # a1: user page table, for satp.

        # switch to the user page table.
        csrw satp, a1 # a1是用户页表，此处是将用户页表写入satp中，实现了从内核页表->
用户页表的切换
        sfence.vma zero, zero # 刷新TLB

        ld t0, 112(a0) # 112(a0)是trapframe->a0，保存着系统调用的返回值，此时
t0=trapframe->a0
        csrw sscratch, t0 # 将trapframe->a0先写入sscratch中

        # 将trapframe中保存的数据(进入内核态之前的用户态数据)重新读回对应的寄存器中
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
        ld s5, 200(a0)
        ld s6, 208(a0)
        ld s7, 216(a0)
        ld s8, 224(a0)
        ld s9, 232(a0)
        ld s10, 240(a0)
        ld s11, 248(a0)
        ld t3, 256(a0)
```

```
        ld t4, 264(a0)
        ld t5, 272(a0)
        ld t6, 280(a0)

        # 将a0与sscratch的值互换，此时a0=trapframe->a0(即系统调研返回值),sscratch=TRAPFRAME首地址
        csrrw a0, sscratch, a0

        # sret会将sepc的值写回pc，这样用户进程就可以回到之前的工作的地方
        # 记性好的同学可能还记得，对于系统的第一个进程来说，epc在userinit()中被设定为0了，而地址0处装载着initcode.S的代码，所以第一个进程在返回用户态后就会直接执行initcode.S
        sret # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

```

7-kernel/proc.c->user/initcode.S：‘sret’ (userinit()创造的进程——系统的第一个用户进程)

```
# exec(init, argv)
.globl start
start:
        la a0, init
        la a1, argv
        li a7, SYS_exec
        ecall # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0

```

8-initcode.S->kernel/trampoline.S：‘ecall’ (userinit()创造的进程——系统的第一个用户进程)

```
# ecall执行后，会将stvec寄存器的值复制到pc中
# 记性好的同学或许记得，刚刚在usertrapret()函数中，我们已经将stvec的值设定为trampoline.S的uservec，所以开始执行uservec
uservec:
        # 记性好的同学应该也记得，刚刚在userret处，我们将sscratch的值改成了TRAPFRAME
        # 所以sscratch记录着TRAPFRAME的虚拟地址，此操作交换sscratch与a0，于是a0就记录着trapframe的地址
        # 想了解40(a0)到底是什么，可以去proc.h中查询trapframe结构体的定义，注释中有标注
        csrrw a0, sscratch, a0

        # 将之前说的寄存器都存起来
        sd ra, 40(a0)

```

```
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
        sd s4, 192(a0)
        sd s5, 200(a0)
        sd s6, 208(a0)
        sd s7, 216(a0)
        sd s8, 224(a0)
        sd s9, 232(a0)
        sd s10, 240(a0)
        sd s11, 248(a0)
        sd t3, 256(a0)
        sd t4, 264(a0)
        sd t5, 272(a0)
        sd t6, 280(a0)

    # a0的初始值存在sscratch里，现在交换sscratch和t0，再将t0(a0的初始值)存到trapframe->a0 即:112(a0)里
        csrr t0, sscratch
        sd t0, 112(a0)

        # 8(a0)是kernel_stack_pointer，由于现在要转入内核模式，所以要将sp换成 ksp
        ld sp, 8(a0)

        # 32(a0)是cpu的id，将该id存到tp寄存器中
        ld tp, 32(a0)

        # 16(a0)是kernel_trap
        # 再次，记性好的同学应该还记得，在trap.c的usertrapret()中，我们将kernel_trap的值改成了usertrap()函数地址
        # 所以此时t0即为usertrap()函数的地址
        ld t0, 16(a0)

        # 0(a0)是内核页表，由于现在要转入内核模式，所以要将页表换成内核页表
        ld t1, 0(a0)
        csrw satp, t1
        sfence.vma zero, zero

        # 之前t0中已经存储了usertrap()函数地址，现在跳转到该地址去执行usertrap()
        jr t0 # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

```

9-trampoline.S->kernel/trap.c：‘jr t0:usertrap()’

```
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;
  // status中有一个bit叫做SPP，SPP位=1代表是内核模式，=0代表是用户模式，此处要求SPP=0因为usertrap只能接受来自用户的请求
  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  w_stvec((uint64)kernelvec);

  // 由于现在进入内核态了，内核不知道之前是哪个进程调用这个函数
  // myproc()函数获取当前正在执行的进程指针，以便能够访问该进程的上下文信息。
  struct proc *p = myproc();

  // 将当前程序计数器（PC）的值（即sepc寄存器的值）保存到当前进程的trapframe中的epc字段。这对于之后恢复用户程序执行是必要的
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // 之前我们将PC的值存在epc中，该值为ecall所在地址，之后返回要返回到ecall的下一条指令中，所以+4
    p->trapframe->epc += 4;

    intr_on();

    syscall(); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret(); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
}
```

10-kernel/trap.c->kernel/syscall.c：‘exec init’

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7; // a7处存了系统调用编号
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓判断系统调用编号是否合法
    p->trapframe->a0 = syscalls[num](); // 取出对应的系统调用函数指针,执行系统调用指针
  } else { // 返回错误信息
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

11-kernel/syscall.c:exec(init)->user/init.c：‘syscalls[sys_init]()’

```
char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", CONSOLE, 0);
    open("console", O_RDWR);
  }
  dup(0);
  dup(0);

  for(;;){
    printf("init: starting sh\n");
    pid = fork(); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
    if(pid < 0){
      printf("init: fork failed\n");
      exit(1);
    }
    if(pid == 0){ // 子进程
      exec("sh", argv); // ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓子进程exec运行sh.c
      printf("init: exec sh failed\n");
      exit(1);
    }

    for(;;){ // 父进程等待子进程结束

      wpid = wait((int *) 0);
```

```
      if(wpid == pid){
        // the shell exited; restart it.
        break;
      } else if(wpid < 0){
        printf("init: wait returned an error\n");
        exit(1);
      } else {
        // it was a parentless process; do nothing.
      }
    }
  }
}

```

11-1user/init.c->kernel/proc.c：‘fork()’

```
// 创建一个子进程，父进程的返回值为子进程的pid，子进程返回值为0
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc(); // 获取当前进程(父进程)的pcb

  // Allocate process.
  if((np = allocproc()) == 0){ // 为子进程申请新的pcb
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){ // 将父进程的数据copy到子进程的内存空间中↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz; // 复制父进程的size数据

  np->parent = p; // 确定parent关系

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe); // 将父进程的trapframe内容复制给子进程

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0; // a0一般代表着系统调用的返回值，将子进程的a0设置为0 即可让子进程的系统调用返回0，用来区分 父/子

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++) // 子进程也要打开父进程使用的文件
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]); // 文件被进程打开时会有reference记录，子进程也打开这些文件所以ref值要增加
  np->cwd = idup(p->cwd);

```

```
  safestrcpy(np->name, p->name, sizeof(p->name));// 将父进程的名字复制给子进程

  pid = np->pid; // 获取子进程的pid以返回

  np->state = RUNNABLE; // 将子进程设定为可执行状态，调度后则会开始运行

  release(&np->lock);

  return pid; // 父进程返回子进程的pid，子进程返回值为0
}
```

11-1-1kernel/proc.c->kernel/vm.c：‘uvmcopy()’

```
// 将old页表中的数据copy到new页表的空间中
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){ // 每次复制一个page
    if((pte = walk(old, i, 0)) == 0) // 寻找旧页表中 va=i所对应的页表项PTE
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0) // PTE应该要是有效的
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte); // 根据PTE中的信息获取old页面的起始物理地址
    flags = PTE_FLAGS(*pte); // 根据PTE中的信息获取该页面的flags
    if((mem = kalloc()) == 0) // 申请新页面
      goto err;
    memmove(mem, (char*)pa, PGSIZE); // 将old页面的数据copy到新页面
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){ // 将新页面映射到
new页表中
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err: // 如果这个过程出现失败，则要回退之前做过的所有copy
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

12-user/init->user/sh：‘exec(“sh”,argv)’

```
int // 这就是我们看到的shell
main(void)
{
  static char buf[100];
  int fd;

```

```
  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }


  while(getcmd(buf, sizeof(buf)) >= 0){ // shell 会不停地接收指令，执行指令
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        fprintf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0) // shell 每次收到指令就fork一个子进程去运行指令，而自己则wait(0)等待子进程直接完毕才接收下一条指令
      runcmd(parsecmd(buf));
    wait(0);
  }
  exit(0);
}
```

三、任务一：

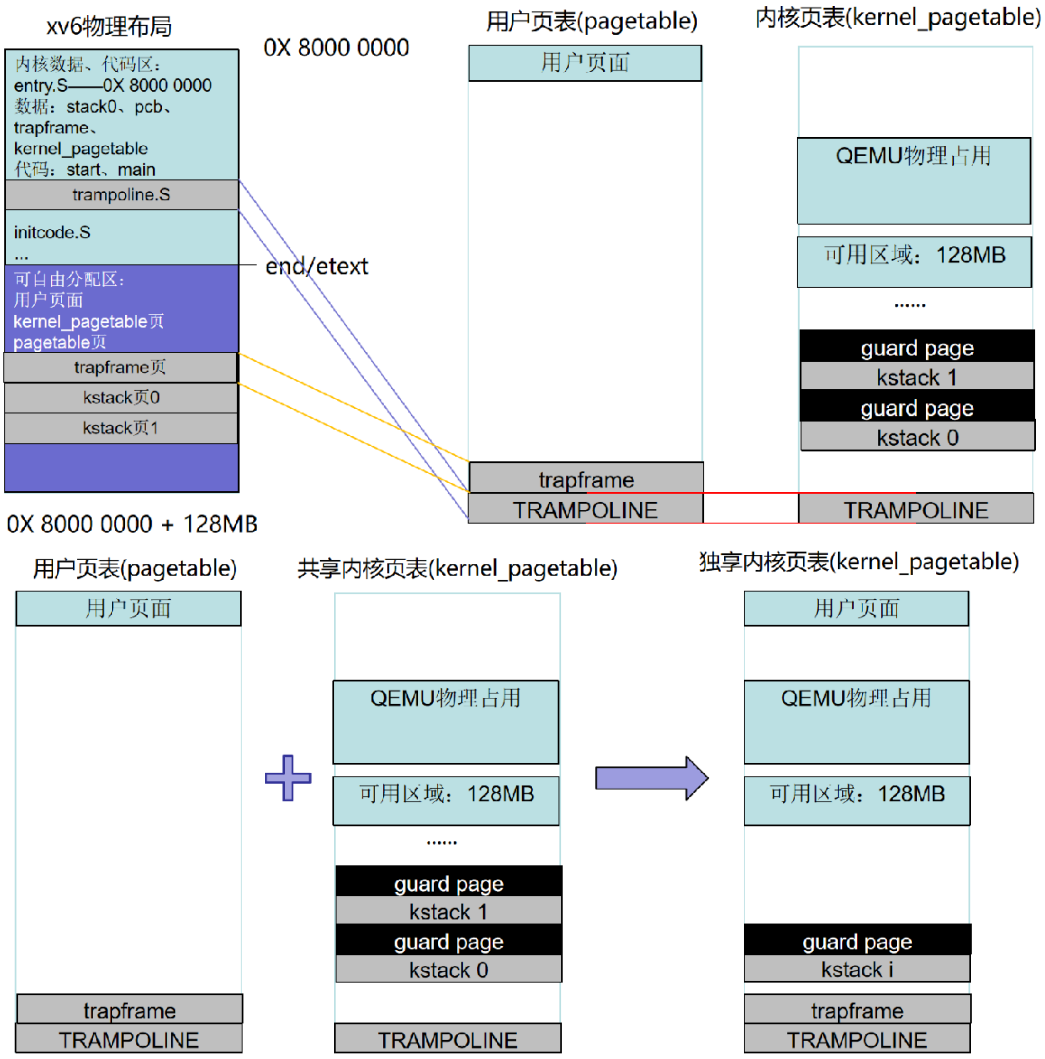
- 任务名称：打印页表
- 任务要求：
    - 定义一个vmprint（pagetable_t pagetable）函数（推荐放在vm.c文件中），可以打印出该页表中所有pte（三级页目录都要访问过去哦）
    - 若pte无效，则不打印
    - 在exec.c的exec()函数返回前增加一个条件：if(p->pid==1)，则调用vmprint（p->pagetable）
    - 输出要求如下：
        - 第一行为”page table ”+<页表地址>后续行输出 “..”* ++“： ”+“pte ”++“pa”+
        - level=1—最高级页目录，2—中级页目录，3—最低级页目录
        - index=页目录号，参照前面的ppt
        - “..”间要有空格
        - 使用深度优先遍历

```
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
```

- 提示：
    - 如何遍历页表？可以参考walk()函数
    - 如何判断pte无效？与‘PTE_V’有关

四、任务二、三：

- 任务名称：为每个用户进程新增一个独享的内核页表并应用
- 任务背景说明：
    - 由于内核页表为所有进程共享的公共资源，所以不可能保存用户进程的用户页面映射信息，这就导致在内核态使用copyin()、copyout()、copyinstr()等函数访问用户页面时，需要通过walkaddr()软件翻译用户页面的地址，速度很慢。
    - 若是为每个用户进程创建一个独享的内核页表，这个内核页表中记录了用户页面的映射信息，并在进入内核态时将satp(页表寄存器)中的页表换为独享内核页表(instead of 共享内核页表)，则内核在访问用户页面的时，可以直接访问用户页面地址（自动触发硬件翻译，很快）
- 任务要求：
    - 在proc.h中为每个pcb增加一个独享内核页表变量，也可以叫kernel_pagetable
    - 将所有的copyin()和copyinstr()替换为vmcopyin.c中的copyin_new()、copyinstr_new()
    - 以合适的方式维护该独享内核页表，如：
        - 在alloc_proc()时创建、初始化它
        - 只映射自己的kstack，别把别人的kstack映射到你的独享内核页表中（参考procinit()函数）
        - 在修改用户页表时，同步该改动，如：释放用户页表（freeproc()）时、父进程复制数据给子进程时(fork())，exec()修改页表时，growproc()时，userinit()时 等等，大家自己搜索相关的操作
        - 在内核态时都要使用内核页表，但内核态分为有程序运行状态和无程序状态，若无程序在运行，则用共享的内核页表，若有程序在运行，则使用用户程序独享的内核页表（scheduler()函数处见分晓）
        - 虚拟地址不能无限增长，不然QEMU物理内存所占用的条目就被刷新掉了(此处VA限制为PLIC)，在growproc()函数中要判断VA是否越界


xv6物理布局
0X 8000 0000
用户页表(pagetable)
内核页表(kernel_pagetable)
内核数据、代码区：
entry.S——0X 8000 0000
数据：stack0、pcb、
trapframe、
kernel_pagetable
代码：start、main
trampoline.S
initcode.S
...
end/etext
可自由分配区：
用户页面
kernel_pagetable页
pagetable页
trapframe页
kstack页0
kstack页1
0X 8000 0000 + 128MB
用户页面
QEMU物理占用
可用区域：128MB
.....
guard page
kstack 1
guard page
kstack 0
trapframe
TRAMPOLINE
TRAMPOLINE
用户页表(pagetable)
共享内核页表(kernel_pagetable)
独享内核页表(kernel_pagetable)
用户页面
QEMU物理占用
可用区域：128MB
.....
guard page
kstack 1
guard page
kstack 0
trapframe
TRAMPOLINE
TRAMPOLINE
用户页面
QEMU物理占用
可用区域：128MB
guard page
kstack i
trapframe
TRAMPOLINE