

OS实验2: 页表 实验报告

吕佳鸿 10235501436

vmprint

这个任务是编写一个打印页表内容的函数。通过提示我们可以阅读freewalk.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_RIPTE_WIPTE_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);
}
```

首先是遍历512个页表的条目，用pte保存当前条目的值。

然后是判断该页表项是否有效及是否有权限

PTE_V、**PTE_R**、**PTE_W**、**PTE_X**：这些是PTE中的标志位（flag）含义如下：

- PTE_V：有效位，表示该条目有效且在使用中。
- PTE_R：读取权限，如果设置，则允许对该页进行读取操作。
- PTE_W：写入权限，如果设置，则允许对该页进行写入操作。
- PTE_X：执行权限，如果设置，则允许从该页执行代码。

然后递归调用，将PTE清零

仿照这个思路 可以也用递归的思路完成vmprint函数

```
void recursive_vmprint(pagetable_t pagetable, int level) {
    for (int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i]; // 获取当前页表项 (PTE)
        if (pte & PTE_V) { // 检查 PTE 是否有效 (PTE_V)
            for (int j = 0; j < level; j++) {
                if (j)
                    printf(" ");
                printf("..");
            }
            uint64 child = PTE2PA(pte);
            printf("%d: pte %p pa %p\n", i, pte, child); // 打印当前 PTE 的
            // 索引、PTE 的值以及物理地址
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
                recursive_vmprint((pagetable_t)child, level + 1); // 递归打印子页
                // 表，层级加1
            }
        }
    }
}
```

```
void vmprint(pagetable_t pagetable) {
    printf("page table %p\n", pagetable);
    recursive_vmprint(pagetable, 1); // 调用递归函数打印页表层级信息，初始level为1
}
```

还是先遍历512个条目，若PTE有效将当前层页表全部打印，当到下一级页表的时候再递归调用，将页表全部打印出来

A kernel page table per process

本实验主要是让每个进程都有自己的内核页表，这样在内核中执行时使用它自己的内核页表的副本。

首先给*kernel/proc.h*里面的 `struct proc` 加上内核页表的字段。

```
pagetable_t kernel_page; // set up a kernel page for each progress
```

接着在*vm.c*里添加*kernel_page_init*用于在*allocproc* 中初始化进程的内核页表。这个函数还需要一个辅助函数 `uvmmmap`，该函数和 `kvmmmap` 方法几乎一致，不同的是 `kvmmmap` 是对Xv6的内核页表进行映射，而 `uvmmmap` 将用于进程的内核页表进行映射。然后在*allocproc*里调用。

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

// Create a kernel page table for the process
pagetable_t
kernel_page_init(){
    pagetable_t kernel_page = uvmcreate();
    if (kernel_page == 0) return 0;
```

```

    uvmmmap(kernel_page, UART0, UART0, PGSIZE, PTE_R | PTE_W);
    uvmmmap(kernel_page, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
    uvmmmap(kernel_page, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
    uvmmmap(kernel_page, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
    uvmmmap(kernel_page, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R |
PTE_X);
    uvmmmap(kernel_page, (uint64)etext, (uint64)etext, PHYSTOP-
(uint64)etext, PTE_R | PTE_W);
    uvmmmap(kernel_page, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R |
PTE_X);
    return kernel_page;
}

```

每一个进程的内核页表都关于该进程的内核栈有一个映射。我们需要将procinit方法中相关的代码迁移到allocproc方法中。

```

// Allocate a page for the process's kernel stack.
// Map it high in memory, followed by an invalid
// guard page.
char *pa = kalloc();
if(pa == 0)
    panic("kalloc");
uint64 va = KSTACK((int) (p - proc));
uvmmmap(p->kernelpt, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
p->kstack = va;

```

即这段代码，调用kalloc函数从内核内存分配一页内存，用于存储当前进程的内核栈，然后调用uvmmmap函数将内核栈的物理地址pa映射到虚拟地址va。

这里要注意把原有的procinit里的代码删掉，否则会映射两遍而导致virtio_disk报错

然后修改scheduler函数，将内核页表加载到satp寄存器中，使用sfence_mac刷新TLB，确保新的页表生效。

这里注意要调用kvminithart(), 再次初始化当前硬件线程的内核页表. 否则还是会报错。

在实现完kernel_page后, 接下来该释放, 否则会有panic : kvmmap的报错

在freeproc中释放页表的内核栈

```
// 释放内核栈
uvmunmap(p->kernelpt, p->kstack, 1, 1);
p->kstack = 0;
```

然后释放进程的内核页表, 先在kernel/proc.c里面添加free_kernel_page。如下, 历遍整个内核页表, 然后将所有有效的页表项清空为零。如果这个页表项不在最后一层的页表上, 继续进行递归。

```
void
free_kernel_page(pagetable_t kernel_page)
{
    // similar to the freewalk method
    // there are 2^9 = 512 PTEs in a page table.
    for(int i = 0; i < 512; i++){
        pte_t pte = kernel_page[i];
        if(pte & PTE_V){
            kernel_page[i] = 0;
            if ((pte & (PTE_RIPTE_WIPTE_X)) == 0){
                uint64 child = PTE2PA(pte);
                free_kernel_page((pagetable_t)child);
            }
        }
    }
    kfree((void*)kernel_page);
}
```

并在freeproc函数中调用,来释放内核页表

修改 vm.c 中的kvmpa, 将原先kernel_pagetable改成myproc()->kernel_page, 使用进程的内核页表

最后将所有新添加的函数加到def.h

踩的几个坑:

1. 没有将原有页表清除, 导致virtio_disk_intr status错误
2. 没有调用free_kernel_page 导致panic:kvmpa
3. 函数没有全加到def.h, 导致编译时找不到函数
4. 在scheduler.c 中 在转换到内核页表后没有调用kvminithart再次映射, 导致make qemu后卡住

Simplify copyin/copyinstr

这个题目的任务就是要将用户页表的映射添加进进程的内核页表中

并注意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

首先添加复制函数。需要注意的是, 在内核模式下, 无法访问设置了PTE_U的页面, 所以我们要将其移除。

```
void kvmcopy(pagetable_t pagetable, pagetable_t kernel_page, uint64
oldsz, uint64 newsz) {
    pte_t *pte_from, *pte_to;

    oldsz = PGROUNDUP(oldsz);
    for (uint64 i = oldsz; i < newsz; i += PGSIZE) { // 遍历从 oldsz 到 newsz
的每一页
```

```

    if ((pte_from = walk(pagetable, i, 0)) == 0) // 查找用户页表 pagetable
中虚拟地址 i 对应的页表项 (PTE)。
        panic("kvmcopy: src pte does not exist"); // 如果找不到 PTE, 则报错并
终止。
    if ((pte_to = walk(kernel_page, i, 1)) == 0) // 查找或创建内核页表
kernel_page 中虚拟地址 i 对应的页表项。
        panic("kvmcopy: pte walk failed"); // 如果无法创建 PTE, 则报错并终止。
    uint64 pa = PTE2PA(*pte_from);
    uint flags = (PTE_FLAGS(*pte_from)) & (~PTE_U); 移除用户访问权限 (PTE_U)
    *pte_to = PA2PTE(pa) | flags;
}
}

```

接下来在fork(), exec(), growproc()中添加复制函数

注意在growproc函数中, 为了防止用户进程增长到超过PLIC的地址, 要加一个限制

```

int
growproc(int n)
{
    uint sz;
    struct proc *p = myproc();

    sz = p->sz;
    if(n > 0){
        if (PGROUNDUP(sz + n) >= PLIC){
            return -1;
        }
        if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
            return -1;
        }
        // 复制到内核页表
        kvmcopy(p->pagetable, p->kernel_page, sz - n, sz);
    }
}

```

```

    } else if(n < 0){
        sz = uvmdealloc(p->pagetable, sz, sz + n);
    }
    p->sz = sz;
    return 0;
}

```

然后替换掉原有的copyin()和copyinstr()


并将copying_new和copyinstr_new添加到内核函数中

几个问题的理解：

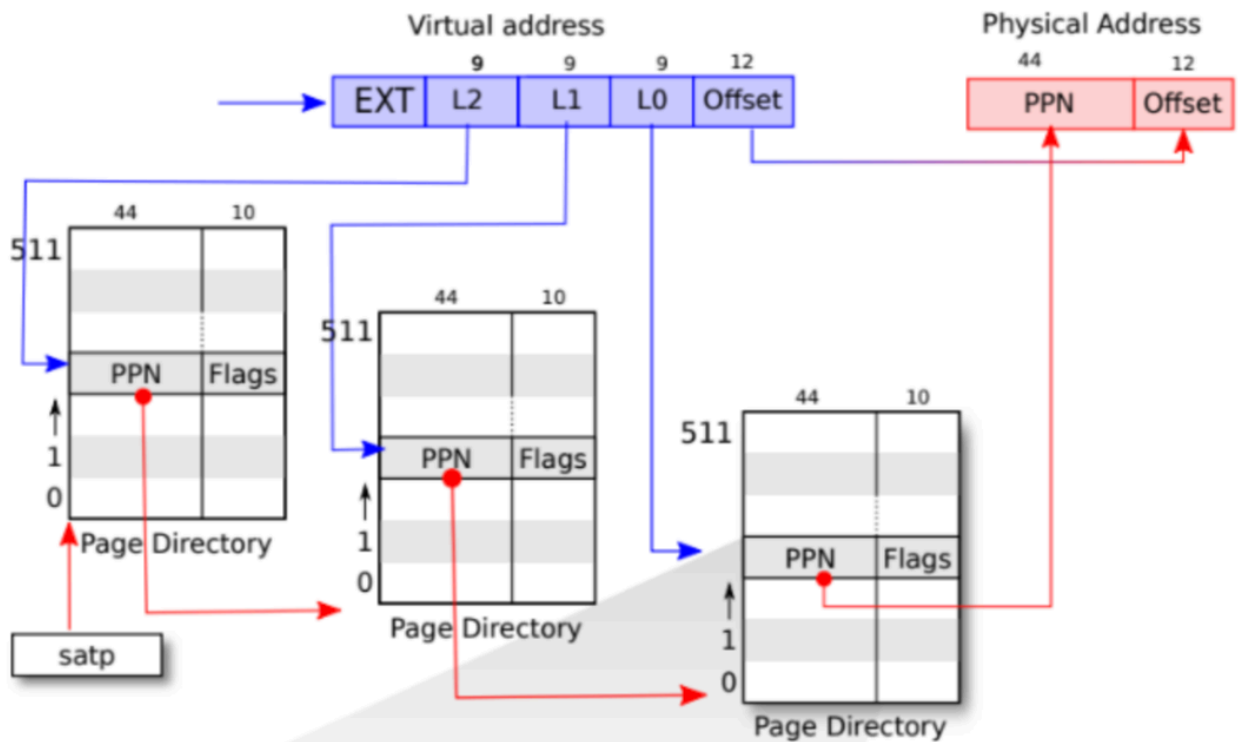
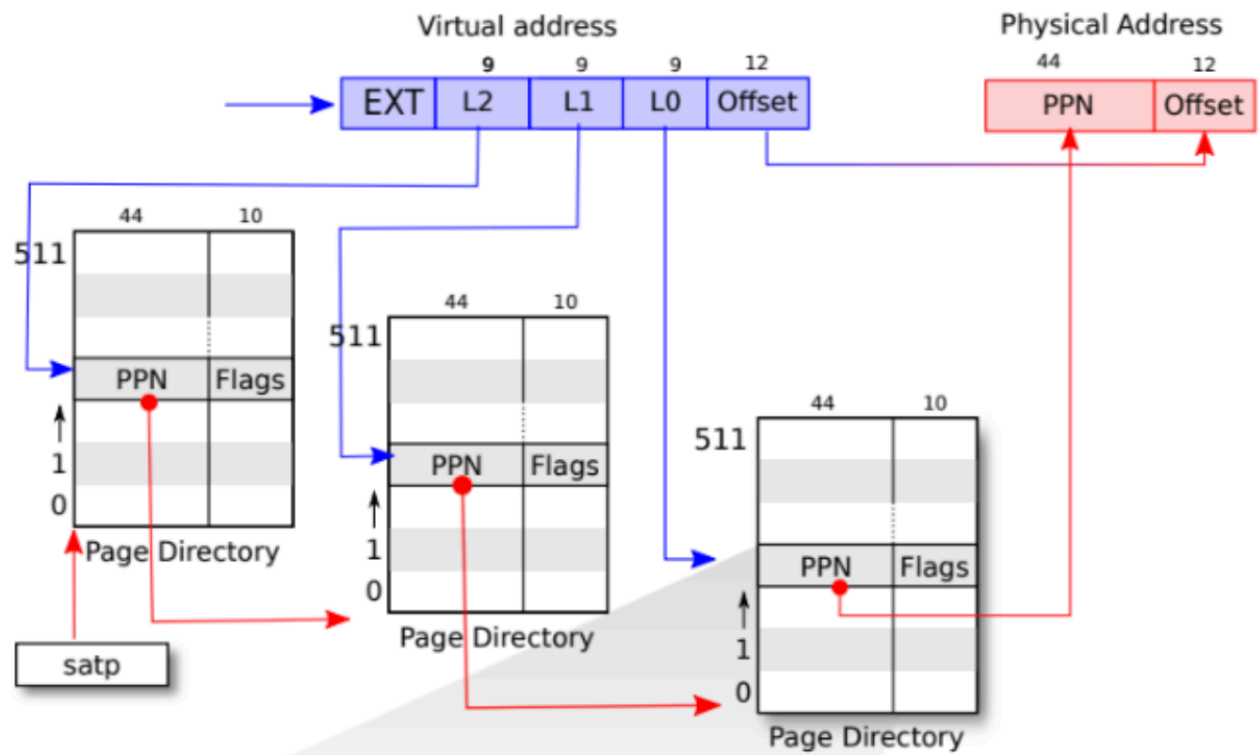
PTE2PA()和PA2PTE()中的左右移操作？

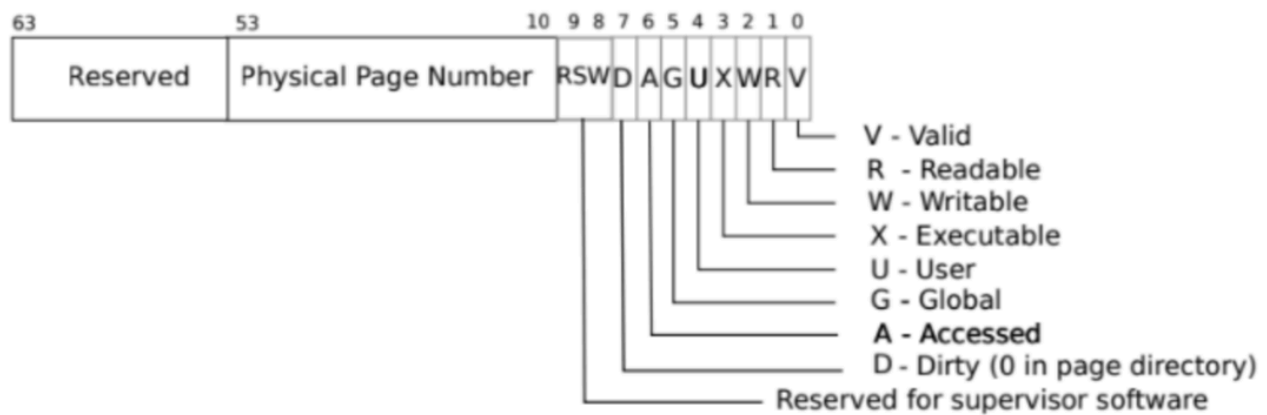
先看一下这两个宏是怎么定义的

```

#define PTE2PA(pte) ((pte) >> 10 << 12)
#define PA2PTE(pa) ((pa) >> 12 << 10)!
(/Users/kerwinlv/Library/Application Support/typora-user-images/image-
20241020135318320.png)

```



PTE2PA是将pte地址转化为pa地址右移10位去除flag，左移12位为OFFSET留下空间

PA2PTE是将物理地址pa转化为pte地址，右移12位是去除OFFSET，左移10位是为FLAG留下空间

PTE中的各个flag代表什么含义，分别在什么时候被设置，用户的kernel_pagetable一定不能有哪一个flag?

PTE_R（读标志，Readable）：

表示该页是否可读,并在创建可读页时设置，例如在加载代码段、数据段时设置

PTE_W（写标志，Writable）：

表示该页是否可写。当映射内存页并且希望该页可写时设置

PTE_X（执行标志，Executable）：

表示该页是否可以执行（即可以将其内容当作指令来执行）。在加载程序的代码段时设置，使得该段可以被执行

PTE_U（用户态标志，User-accessible）：

表示该页是否可被用户态访问。用户程序的页需要设置该标志，以允许在用户态下访问。

PS: `kernel_pagetable` 中的页表项一定不能有这个标志，因为这是内核的页表，用户态程序不应直接访问内核态内存区域，否则会造成安全风险。

PTE_V (有效标志, Valid) :

表示该页表项是否有效。在建立页表项时设置，如果页表项无效 (PTE_V为0)，则尝试访问该页会引发页表异常。

pc相关的寄存器，如sepc、mepc、epc、ra?

sepc (Supervisor Exception Program Counter) 当CPU在用户态执行时发生异常)，硬件会将导致异常的指令地址保存到sepc，然后跳转到异常处理程序。

mepc (Machine Exception Program Counter) 当发生trap时，会跳转到异常控制流处执行，而mepc负责保存异常控制流函数的返回地址

epc (Exception Program Counter) 通常是指sepc和mepc的泛指，用于存储发生异常时的pc值

ra (Return Address) 返回地址寄存器，在调用函数时，ra寄存器会保存当前pc的值，以便函数返回时使用。

copyin_new()中的if中各个条件的含义，它们为什么不合法?

```
if (srcva >= p->sz || srcva+len >= p->sz || srcva+len < srcva)
    return -1;
```

第一个 `srcva >= p->sz` 用来源地址srcva是否在用户进程的地址空间范围内，若超过了地址范围，即 `p->sz`，则不合法

第二个 `srcva+len >= p->sz` 用来检查从srcva开始的长度为len的地址范围是否超出用户进程的地址空间，len是要复制的长度，srcva是起始地址，如果`srcva + len`超过了`p->sz`，则说明要复制的整个内存范围部分超出了用户地址空间的合法边界。这会导致内核尝试读取未分配的虚拟地址，进而可能触发页错误或内存访问违规。

第三个 `srcva+len < srcva` 检查是否发生地址溢出

COW

在`kernel/riscv.h`中选取PTE中的保留位定义标记一个页面是否为COW Fork页面的标志位

```
#define PTE_F (1L << 8) // record the page uses fork()
```

定义引用并初始化计数的全局变量 `ref`，其中包含了一个自旋锁和一个引用计数数组，由于 `ref` 是全局变量，会被自动初始化为全0。

这里使用自旋锁是考虑到这种情况：进程P1和P2共用内存M，M引用计数为2，此时CPU1要执行 `fork` 产生P1的子进程，CPU2要终止P2，那么假设两个CPU同时读取引用计数为2，执行完成后CPU1中保存的引用计数为3，CPU2保存的计数为1，那么后赋值的语句会覆盖掉先赋值的语句，从而产生错误

```
struct ref_stru {  
    struct spinlock lock;  
    int cnt[PHYSTOP / PGSIZE];  
} ref;
```

在 `kalloc` 中初始化内存引用计数为1，在 `kfree` 函数中对内存引用计数减1，如果引用计数为0时才真正删除

添加函数 `cowpage` 来判断一个页面是否是COW页面，实现如下：

```

int cowpage(pagetable_t pagetable, uint64 va) {
    if(va >= MAXVA)
        return -1;
    pte_t* pte = walk(pagetable, va, 0);
    if(pte == 0)
        return -1;
    if((*pte & PTE_V) == 0)
        return -1;
    return (*pte & PTE_F ? 0 : -1);
}

```

添加函数 `cowpagealloc`，确保当一个进程对页面进行写操作时，不会影响到其他进程

```

void* cowpagealloc(pagetable_t pagetable, uint64 va) {
    if(va % PGSIZE != 0)
        return 0;

    // 获取虚拟地址va对应的物理地址。
    uint64 pa = walkaddr(pagetable, va); // walkaddr函数返回va对应的物理地址。
    if(pa == 0)
        return 0; // 如果pa为0，表示该虚拟地址无效或未映射。

    uint64 pa = walkaddr(pagetable, va); // walk函数返回指向该地址的PTE指针。

    if(krefcnt((char*)pa) == 1) {
        // 引用计数为1，表示没有其他进程共享该页面，可以直接修改权限为可写。
        *pte |= PTE_W; // 设置PTE_W标志，允许写操作。
        *pte &= ~PTE_F; // 清除PTE_F标志，表示不再是COW页面。
        return (void*)pa; // 返回原物理页面地址，因为此时不需要分配新页面。
    } else {
        // 如果引用计数大于1，说明有多个进程共享此物理页面。
        // 因此需要为当前进程分配一个新的物理页面来存储这个虚拟地址的数据。
        // 分配一个新的物理内存页，用于存储该虚拟地址的内容。
    }
}

```

```

char* mem = kalloc();
if(mem == 0)
    return 0;

// 将原物理页面的内容复制到新分配的页面中，确保新页面和旧页面内容一致。
memmove(mem, (char*)pa, PGSIZE);

*pte &= ~PTE_V;    // 清除PTE_V

将新页面与虚拟地址进行映射。
// 将新页面与虚拟地址进行映射,并将新物理地址mem与虚拟地址va关联，设置PTE_W（可
写），同时清除PTE_F
if(mappages(pagetable, va, PGSIZE, (uint64)mem, (PTE_FLAGS(*pte) |
PTE_W) & ~PTE_F) != 0) {
    kfree(mem);
    *pte |= PTE_V;
    return 0;
}

// 成功完成映射后，释放原物理页面的引用计数（减1），因为当前进程不再使用它。
kfree((char*)PGROUNDDOWN(pa));

return mem;
}
}

```

添加 `refcnt` 和 `addrefcnt` ,分别用于用于查询某个物理页的当前引用计数和将该物理页的引用计数增加一

```

int refcnt(void* pa) {
    return ref.cnt[(uint64)pa / PGSIZE];
}

int addrefcnt(void* pa) {
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >=
PHYSTOP)
        return -1;
    acquire(&ref.lock);
    ++ref.cnt[(uint64)pa / PGSIZE];
    release(&ref.lock);
    return 0;
}

```

修改 `uvmcopy`，不为子进程分配内存，而是使父子进程共享内存，但禁用 `PTE_W`，同时标记 `PTE_F`

```

// 仅对可写页面设置COW标记
if(flags & PTE_W) {
    // 禁用写并设置COW Fork标记
    flags = (flags | PTE_F) & ~PTE_W;
    *pte = PA2PTE(pa) | flags;
}

if(mappages(new, i, PGSIZE, pa, flags) != 0) {
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}
// 增加内存的引用计数
kaddrefcnt((char*)pa);

```

在 `copyout` 中修改，如果是COW页面，需要更换 `pa0` 指向的物理地址

```
if(cowpage(pagetable, va0) == 0) {  
    // 更换目标物理地址  
    pa0 = (uint64)cowpagealloc(pagetable, va0);  
}
```

几个问题的理解：

使用COW与不使用COW的优缺点

优点：COW技术最显著的优势是减少内存使用。当多个进程需要共享同一块数据（如父子进程复制页表时），它们可以共享同一块物理内存，而不需要立即为每个进程都分配一份独立的数据副本。只有当某个进程尝试写入这块数据时，才会创建一个新的副本。

缺点：虽然COW可以推迟内存复制，但当写操作发生时，需要进行内存页的复制操作。这可能会带来显著的性能开销，尤其是在写入操作频繁发生的场景中。复制操作涉及到物理页的分配和内容的复制，会导致写操作的延迟增加。

系统是如何知道有进程在写共享数据的

在内存页被多个进程共享时，操作系统会将这些共享页面的保护权限设置为只读。这样，如果某个进程试图写入该共享页面，硬件（例如 x86 架构中的MMU）会触发一个**页错误（page fault）**。

页错误处理：

当进程对只读页面执行写操作时，会触发页错误异常，操作系统会捕捉到这个页错误。操作系统的页错误处理程序会检查该页面是否属于写时复制的页面：如果是写时复制页面，操作系统就会为当前进程分配一个新的物理页，将原来的内容复制到这个新页面中，然后将该页面的权限设置为可读写，并更新页表。如果该页不是写时复制页面，则操作系统可能会终止该进程并报告错误。

对共享页面的写操作，是否只会发生在用户态？

不是，如果内核需要在一个共享内存区域上进行某些内存操作，它可能会直接对这个区域进行写入。这些写操作是在内核态中进行的，而不是用户态

总结

页表的实验花了很长时间来理解，中间踩了无数次的坑，经历了无数次的报错和make qemu卡住,在网上查了很多资料后才通过，但也对页表的理解更深了，相比之下COW用的时间较为匆忙，感觉还是需要再看看再理解一下

```
make[1]: Leaving directory '/home/parallels/xv6-labs-2020'
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (2.8s)
(Old xv6.out.pteprint failure log removed)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.0s)
(Old xv6.out.count failure log removed)
== Test usertests ==
$ make qemu-gdb
(154.3s)
== 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
== Test time ==
time: OK
Score: 66/66
(base) parallels@ubuntu-linux-22-04-02-desktop:~/xv6-labs-2020$
```

xv6 kernel is booting

hart 1 starting

hart 2 starting

init: starting sh

\$ cowtest

simple: ok

simple: ok

three: ok

three: ok

three: ok

file: ok

ALL COW TESTS PASSED

test copyinstr2: OK

test copyinstr3: OK

test rwsbrk: OK

test truncate1: OK

test truncate2: OK

test truncate3: OK

test reparent2: OK

test pgbug: OK

test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3248

sepc=0x00000000000055d2 stval=0x00000000000055d2

usertrap(): unexpected scause 0x000000000000000c pid=3249

sepc=0x00000000000055d2 stval=0x00000000000055d2

OK

test badarg: OK

test reparent: OK

test twochildren: OK

test forkfork: OK

test forkforkfork: OK

test argptest: OK

test createdelete: OK

test linkunlink: OK

test linktest: OK

test unlinkread: OK

test concreate: OK

test subdir: OK

test fourfiles: OK

test sharedfd: OK

test dirttest: OK

test exectest: OK

test bigargtest: OK

```
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validate: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```