

# 虚拟内存

YuanShen OS在虚拟内存上新增的特征有:

- 1. 使用vma (virtual\_memory\_area)组织进程地址空间
- 2. COW (Copy-on-Write) Fork
- 3. 为每个进程添加一个内核页表以加速系统调用。
- 4. Lazy Allocation
- 5. 文件内存映射
- 6. 修改exec系统调用以支持Demand Paging

注: 本章中所有的图片均为组员绘制，并非来源于网络。

测试代码和参考资料位于文章的末尾。

## 1.1 内存布局

YuanShen OS的进程的地址空间布局如下图所示:

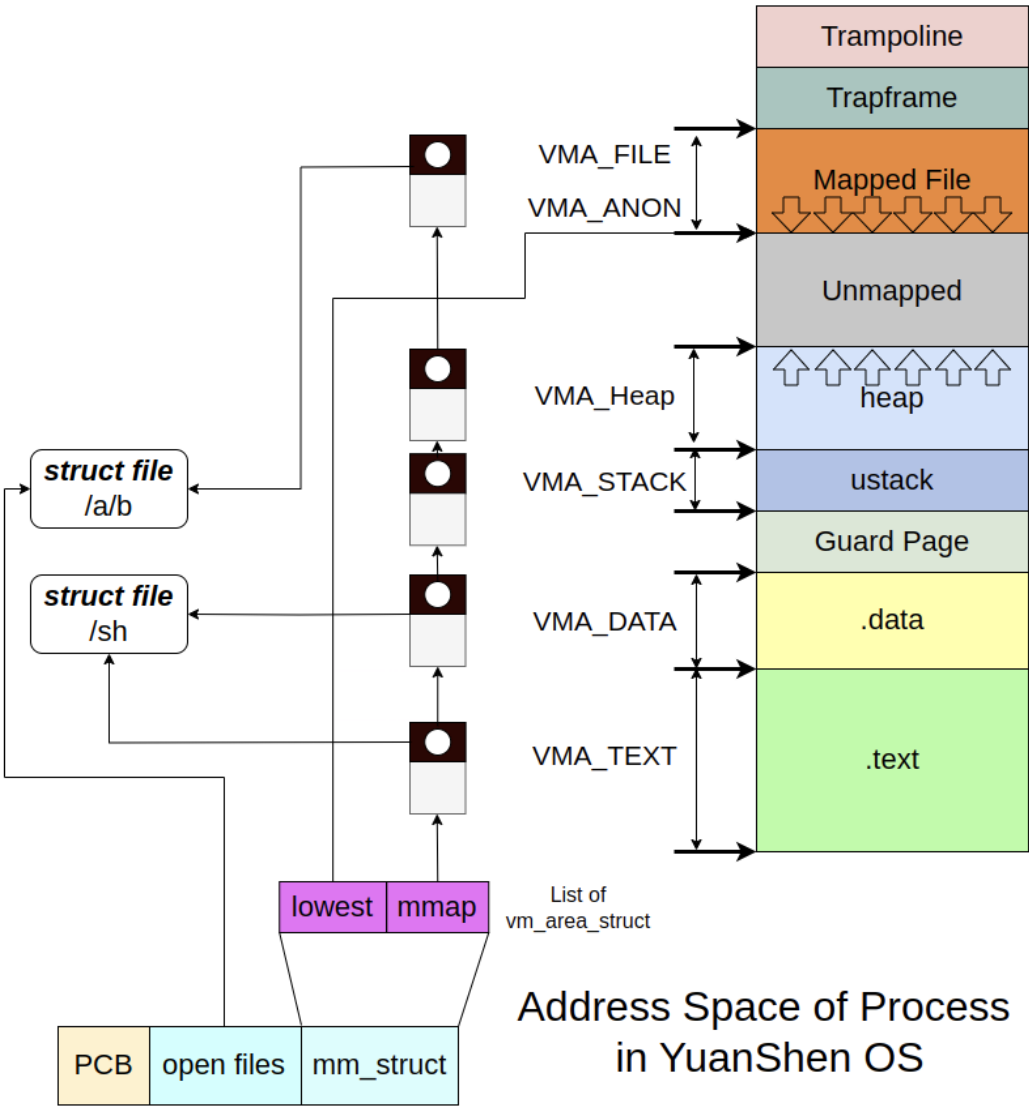


Fig1. YuanShen OS的进程地址空间管理 (原创)

总体来看，YuanShen OS的内存区域根据其被操作的模式可以分为两部分：

内核态: 在图中为Trampoline和Trapframe页，其由内核管理。

用户态: 进程的其余部分，包括进程的堆栈，数据或者代码，以及文件映射区。

用户态的内存范围由VMA进行管理，其在上图中为**链表的结点**：

1. VMA\_HEAP是用户的堆区，它的起始位置是栈区之后，也就是用户的代码段数据段的结束的位置，HEAP段的大小可以通过sbrk这个系统调用改变。
2. VMA\_TEXT和VMA\_DATA是程序的代码段和数据段，在exec系统调用载入文件时进行分配
3. VMA\_FILE和VMA\_ANON用于mmap系统调用的映射，分别对应了文件和匿名的内存区域。可以使用mmap系统调用进行创建
4. VMA\_SYACK为用户栈区。YuanShen OS的进程在默认情况下只能使用一页的栈空间。

## 1.2 VMA (Virtual Memory Area)

YuanShen OS基于vm\_area\_struct结构体管理进程的地址空间，其定义如下：

```
struct vm_area_struct {
    uint64 vm_start;
    uint64 vm_end;
    struct vm_area_struct *vm_next, *vm_prev;
    int vm_flags;
    int vm_prot;
    struct file *vm_file;
    uint64 vm_pgoff;
    uint64 vm_filesz;
    uint32 vma_type;
};
```

其中vm\_start和vm\_end控制VMA管理的内存范围。vm\_flags和vm\_prot控制页面的共享属性与保护属性。从vm\_file到vm\_filesz保存文件内存映射的元信息，而vma\_type表示VMA的类型，具有如下取值：

```
#define VMA_FILE 0x1
#define VMA_HEAP 0x2
#define VMA_TEXT 0x4
#define VMA_STACK 0x8
#define VMA_ANON 0x16
#define VMA_DATA 0x32
```

我们为每个进程配置一条VMA组成的链表，用于管理整个用户态的内存区域。

## 1.3 Generic Page Fault Handler

根据risc-v privilege spec, 关于缺页的异常号有12,14,15。我们在usertrap和kerneltrap函数里设置了处理缺页异常处理程序的入口:

```
//trap.c
if((r_scause() == 12 || r_scause() == 13 || r_scause() == 15)){
    page_fault_handler();
}
```

YuanShen OS的缺页异常的处理流程如下图所示:

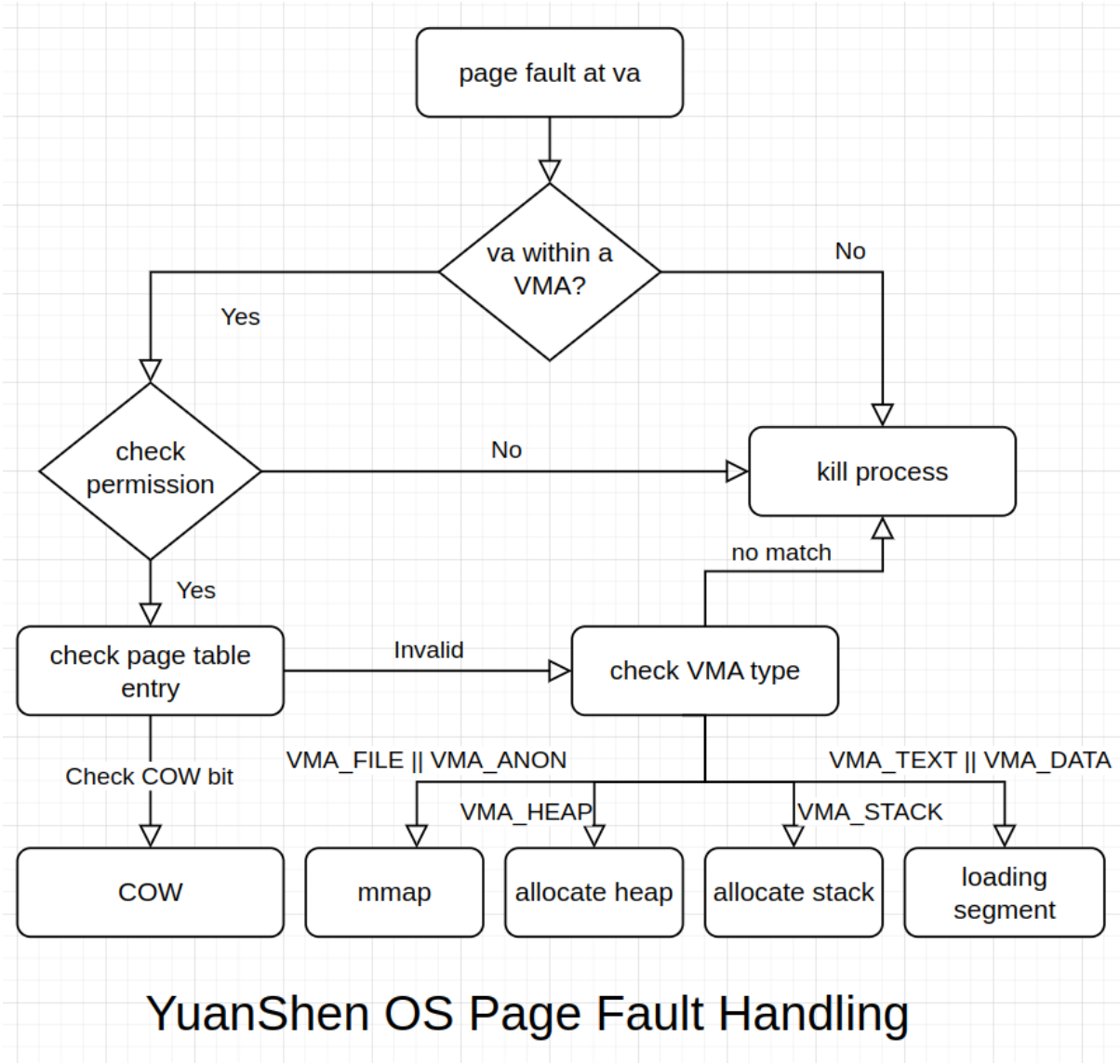


Fig2. YuanShen OS的缺页异常处理程序(原创)

首先遍历链表查找对应的VMA，如果在进程的所有vma中都没有找到该va，则杀死该进程:

```
//vm.c:page_fault_handler
struct proc *p = myproc();
uint64 va = r_stval();
if(va > MAXVA) {
    panic("page_fault_handler: maxva");
}
```

```
struct vm_area_struct *vma = find_vma(&p->mm, va);
if(vma == 0) {
    setkilled(p);
    return;
}
```

然后找到vma后，将scause中的pagefault的类型，以及vma的访问权限进行匹配。

1. 如果scause是15，则要求vma管理的区域可写。
2. 如果scause是13，则要求vma管理的区域可读。
3. 如果scause是12，则要求vma管理的区域可执行。

```
//vm.c:page_fault_handler
if((r_scause() == 13) && ((vma->vm_prot & PROT_READ) != 0))
    flag = 1;

if((r_scause() == 15) && ((vma->vm_prot & PROT_WRITE) != 0))
    flag = 1;

if(r_scause() == 12 && ((vma->vm_prot & PROT_EXEC) != 0))
    flag = 1;

if(flag == 0) {
    setkilled(p);
    return;
}
```

最后我们诊断引发缺页异常的具体原因为COW页面，惰性分配的页面和文件内存映射的页面中的一个，并采取对应的解决方案。

1. 如果是访问了COW页，则调用uvmcowcopy(va)。
2. 如果是访问了VMA\_HEAP的页面，则调用uvmlazyalloc(va)。
3. 如果访问了VMA\_FILE或VMA\_ANON的页面，则调用mmap\_(vma, va)。
4. 如果是访问了VMA\_TEXT或者VMA\_DATA或者VMA\_STACK的页面，则也调用mmap\_将程序段根据ELF Header的要求重定位到正确的位置。

```
//vm.c:page_fault_handler
pte_t *pte = walk(p->pagetable, va, 0);
if((pte == 0) || (*pte & PTE_V) == 0) {
    if(vma->vma_type == VMA_FILE || vma->vma_type == VMA_TEXT || vma->vma_type == VMA_DATA || vma->vma_type == VMA_ANON || vma->vma_type == VMA_STACK) {
        mmap_(vma, va); //mmap
    } else if(vma->vma_type == VMA_HEAP) {
        uvmlazyalloc(va); //lazy
    } else {
        setkilled(p);
    }
}
```

```
} else {  
    if(!(*pte & PTE_COW)) {  
        setkilled(p);  
    } else {  
        uvmcowcopy(va); //COW  
    }  
}
```

文档接下来的部分将着重介绍YuanShen OS在虚拟内存方面的几种优化方案以及实现原理，刚才调用的函数会在接下来的部分得到解释。

## 2.1 COW Fork

回忆在SLUB分配器的文档，我们为每个物理页分配了一个pages数组的表项，用于存储物理页面的元信息。

```
//slub.h  
struct page {  
    //COW:  
    int refcount;  
    //SLUB ALLOCATOR:  
    uint32 active_obj_count; //未回收的对象数量  
    uint64 flags;  
    void *freelist; //由header管理,其它的页不设置该字段,其指向本slab页面内部维护的  
    freelist的第一个对象节点  
    struct kmem_cache *slab;  
    struct page *header; //the beginning page of a slab  
    struct page *next; //指向下一个slab(的header页),该字段只由header管理  
};
```

其中的refcount表示有多少物理进程留有对该物理页的映射。

COW要求进程在调用fork后不立即为子进程分配新的物理页,而只是把页面标记为只读状态,等到进程改写对应内存区域的时候再由page fault handler进行物理页的分配。

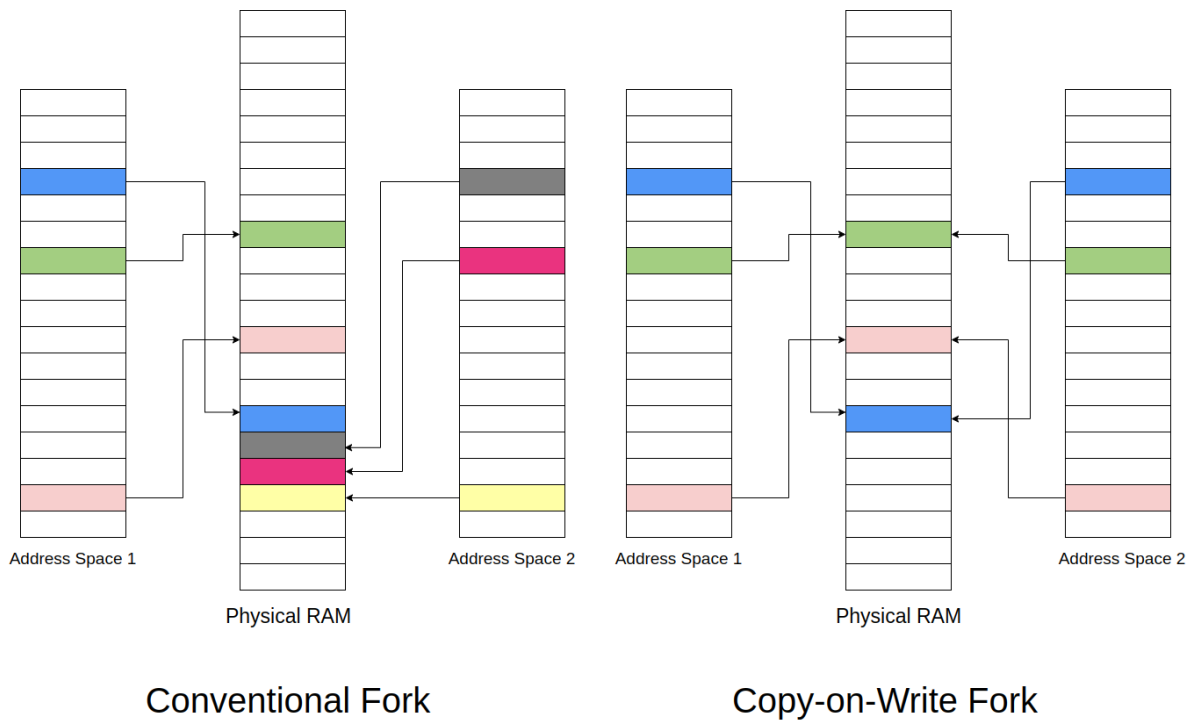


Fig3. Copy-on-Write Fork原理(原创)

当有多个进程都映射了同一个物理页的时候，refcount为这些进程的数量。如果refcount大于1,那么调用kfree时不会释放物理页，而仅仅是把物理页的引用计数减1。

## 2.2 COW的缺页异常处理程序

在usertrap和kerneltrap内我们添加了如下的检测语句,当发生缺页异常的时候，checkcowpage会被调用，检查引发异常的访问地址是否是COW页,如果是，则进一步考虑是否为用户进程分配一个独立的物理页并建立重映射：

先前在做COW的时候，我们会为对应的页表项做上COW标记(使用页表项内部的reserve bit)。现在的checkcowpage无非是检查标记以及映射是否有效。

uvmcowcopy实现的不过就是刚才所说的，为触发异常的进程分配一个新的物理页，减少原先物理页的引用计数，然后修改页表的标志位。

```
uint32
uvmcowcopy(uint64 va){
    // printf("p->name: %s, va: %lx", myproc()->name, va);
    va = PGROUNDDOWN(va);
    pte_t *pte;
    struct proc *p = myproc();
    if((pte = walk(p->pagetable, va, 0)) == 0)
        panic("uvmcowcopy: walk");
    uint64 pa = PTE2PA(*pte);
    uint64 new_pa = 0;
    if(pages[PA2PGREF_ID(pa)].refcount != 1){
        if((new_pa = (uint64)kalloc()) == 0){
            panic("uvmcowcopy: kalloc");
        }
    }
```

```

    memmove((void*)new_pa, (void*)pa, PGSIZE);
    kfree((void*)pa); // reduce the page ref count
    pa = new_pa;
}
uint64 flags = (PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW;
uvmunmap(p->pagetable, va, 1, 0);
if(mappages(p->pagetable, va, PGSIZE, pa, flags) == -1)
    panic("uvmcowcopy: mappages");
copyOnePageToKernelPgtbl(p->kpagetable, p->pagetable, va);
return 0;
}

```

### 3.1 进程内核页表

xv6系统本身不支持进程内核页表，这意味着内核地址空间和用户进程的地址空间是无交的，这为内核与用户空间间的数据搬运带来极大不便。

下面是xv6自带的copyin函数，负责把数据从用户进程的地址空间搬运到内核空间。为了得到用户的虚拟地址对应的物理地址，需要手动翻译页表，这导致各自系统调用的效率极为低下。

```

int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    uint64 n, va0, pa0;

    while(len > 0){
        va0 = PGROUNDDOWN(srcva);
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        n = PGSIZE - (srcva - va0);
        if(n > len)
            n = len;
        memmove(dst, (void*)(pa0 + (srcva - va0)), n);

        len -= n;
        dst += n;
        srcva = va0 + PGSIZE;
    }
    return 0;
}

```

进程内核页表的核心思想是为每个用户进程准备一个内核页表，然后再把用户进程的用户态地址范围的一部分甚至全部映射到内核页表中。

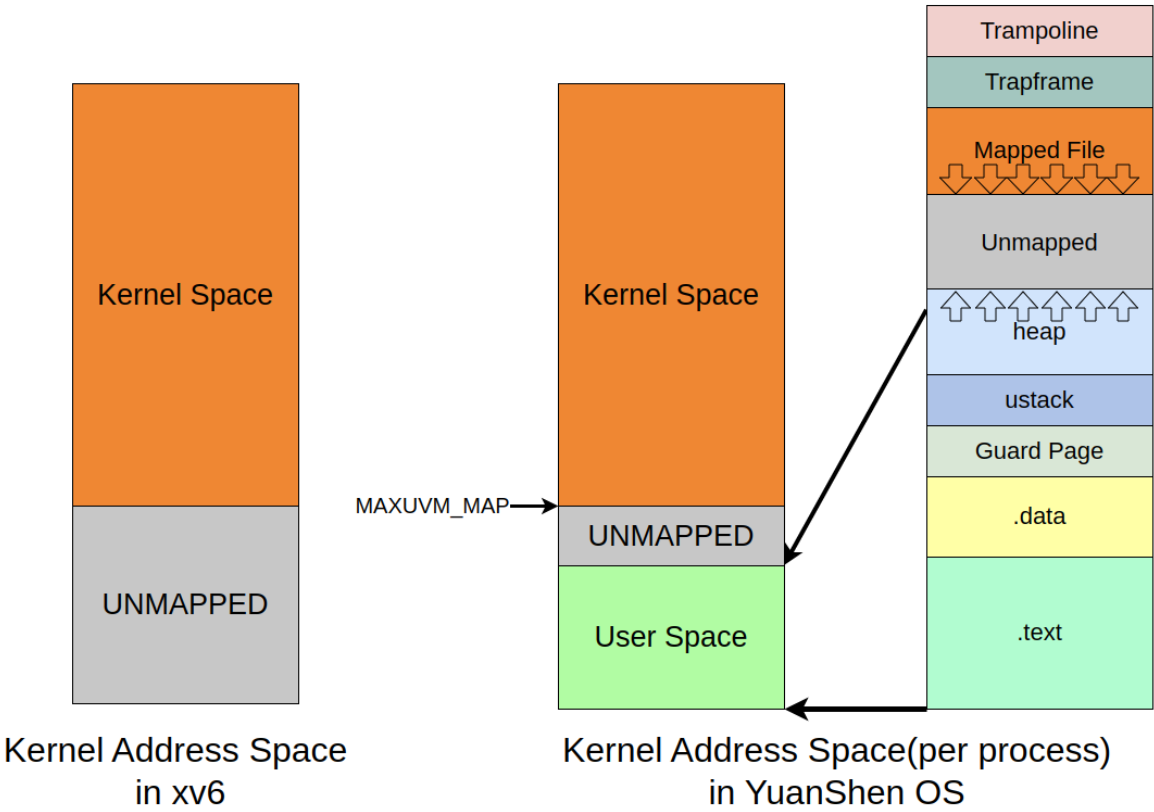


Fig4. 将用户内存映射一部分到进程内核页表(原创)

这样一来，当进程进入内核态的时候，它依旧可以直接对用户地址的解引用，只要用户进程的地址范围没有与内核的地址范围发生重叠。我们只需要把kpagetable引入进程控制块：

```
//proc.h
pagetable_t pagetable;           // User page table

pagetable_t kpagetable;          // per process kernel pagetable to support
direct dereference user va in kernel code
```

改进后的copyin函数如下：

```
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    memmove((void*)dst, (void*)srcva, len);
    return 0;
}
```

不过我们仍然需要保留旧的copyin，通过p->pagetable手动翻译用户地址到内核地址，因为我们可能仅仅是将一部分的用户空间映射到内核的地址空间。YuanShen OS会根据映射情况与访问的用户地址来决定最后使用哪个函数。



在内核页表的支持下，内核态与用户态间的数据搬迁变得极为显然。实现这一点的前提就是要保持用户进程地址空间与映射进内核的部分的一致性。

为了维护一致性，每当用户进程的地址映射发生修改的时候，内核页表的映射也要进行修改。以fork为例，在COW fork后父子进程的映射都发生了改变，于是我们必须也跟着调整它们的内核页表：

```
//int proc.c:fork
//promise to return 0, or panic.
copyPgtblToKernel(np->kpagetable, np->pagetable, p->sz, 0);
//since COW is supported, the kernel pagetable for both process are
expired
copyPgtblToKernel(p->kpagetable, p->pagetable, p->sz, 0);
```

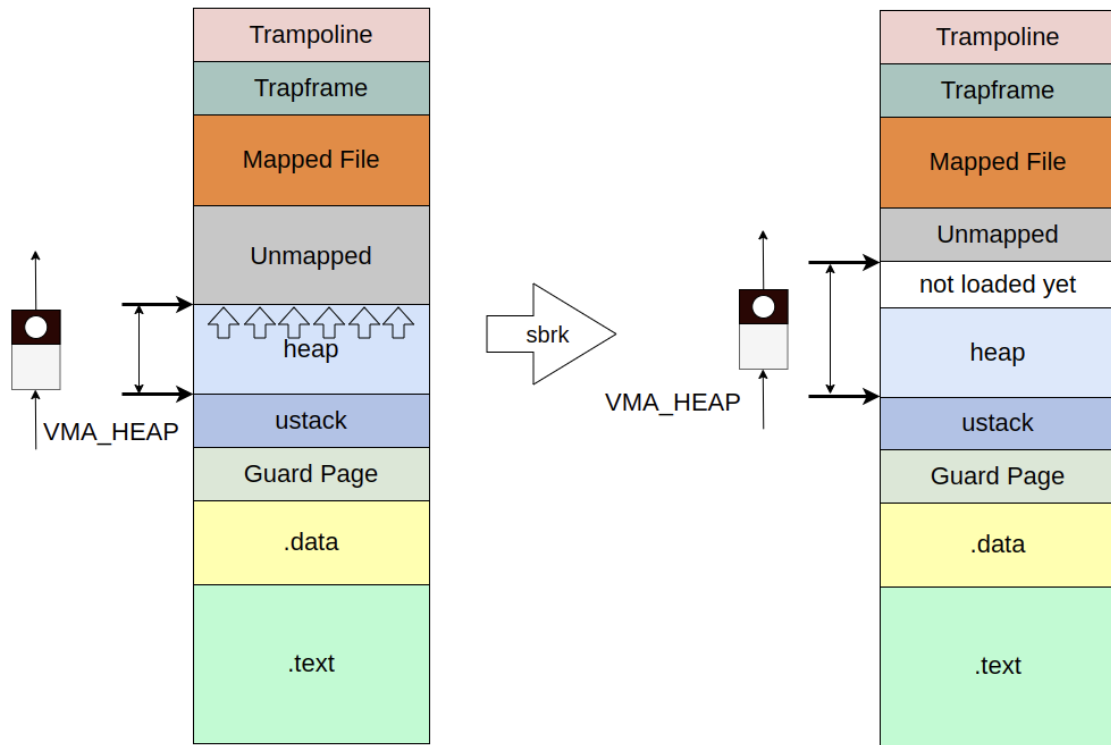
## 4.1 Lazy Allocation

惰性分配的要义在于，在进程申请heap memory的时候，操作系统并不一定就在这个时候为它分配物理页并建立映射，而把这个过程推迟到进程实际访问到这些内存的时候。显然，种种围绕虚拟内存的优化都离不开demand paging的思想，我们只在有必要的时候分配物理页。

## 4.2 Accessing Lazily Allocated Pages

在这个指导思想下，调用系统调用sbrk并不会立即让进程的地址空间变大。尽管有更多的页面可以被进程访问，但是在它真正访问它之前这些页的映射并不会建立。我们所做的不过是修改类型为VMA\_HEAP的VMA的地址范围，这样在缺页异常的时候就能找到对应类型为VMA\_HEAP的VMA：

```
//vm.c:page_fault_handler
if(vma->vma_type == VMA_HEAP) {
    uvmlazyalloc(va);
}
```



To allocate or deallocate heap memory, the YuanShen OS simply modifies the `vm_start` & `vm_end` of the vma in process's vma list

Fig5. 堆内存的惰性分配 - 只修改vma (原创)

因此如果是第一次访问惰性分配的页面，一个缺页异常会被触发。我们只需要在缺页异常发生的时候建立起对应的映射即可，在此之前我们需要判断导致缺页异常产生的地址是否是来自惰性分配的页面。

我们只需要检查当前地址是否对应惰性分配的物理页，顺带检查已有映射的flag。一旦确定该页为惰性分配，那么我们只需要为该处建立到一个新的物理页的映射即可。

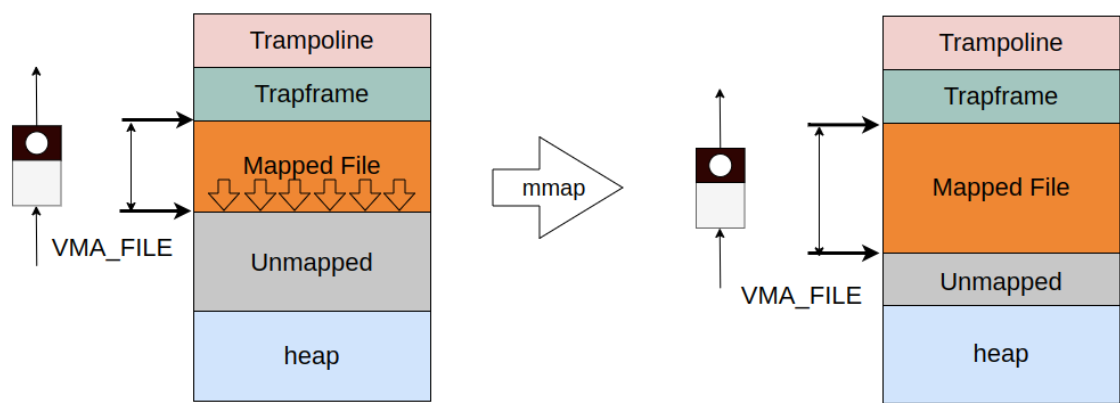
```
void
uvmlazyalloc(uint64 va) {
    struct proc *p = myproc();
    char *mem = kalloc();
    if(mem == 0) {
        p->killed = 1;
    } else {
        memset(mem, 0, PGSIZE);
        if(mappages(p->pagetable, PGROUNDOWN(va), PGSIZE, (uint64)mem,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            printf("lazy alloc: failed to map page\n");
            kfree(mem);
            p->killed = 1;
        }
        copyOnePageToKernelPgtbl(p->kpagetable, p->pagetable, va);
    }
}
```

### 5.1 文件内存映射

这一小节对应YuanShen OS的两个系统调用:

sys_mmap	41	内存映射文件
sys_munmap	42	取消文件的内存映射

内存文件映射的要义在于在地址空间的文件映射区设置起一个新的类型为VMA\_FILE的VMA，并在其中设置文件相关的各种属性，以保障当缺页异常发生的时候，缺页处理程序能把正确的文件内容写到用户地址空间内(如果是映射新的文件，那么图中是会建立一个新的VMA):



xxxxx To create a memory-mapped file, the YuanShen OS will create a new vma (not shown) in the mapped-file area.

Fig6. mmap的语义 - 建立新的vma(或延长原有的vma)(原创)

```
```c //in vm_area_struct, in vma/vma.h struct file *vm_file; uint64 vm_pgoff; uint64 vm_filesz;```
```

其中vm\_file为文件结构体指针，对其可以进行各种IO操作。vm\_pgoff表示VMA开头的位置对应到文件中的偏移量，YuanShen OS要求vma中的字段len，vm\_pgoff必须对齐到4096。

mmap实现的功能十分简单，那就是为文件建立新的VMA。不过，在此之前，需要进行一番文件权限的检查，并增加对文件结构体的引用计数，以保证在进程退出后仍能把脏页写回文件。

```
//sysfile.c:sys_mmap
struct proc* p = myproc();
struct file* f = p->ofile[fd];
if(f == 0)
    return -1;

if((!f->readable && (prot & (PROT_READ)))
    || (!f->writable && (prot & PROT_WRITE) && !(flags & MAP_PRIVATE)))
    return -1;

filedup(f);

return do_mmap(&p->mm, addr, len, offset, len, flags, f, prot, VMA_FILE);
```

do\_mmap负责将合适的VMA插入到进程维护的VMA链表中。YuanShen OS要求类型为VMA\_FILE时vm\_start不能由用户设置，而由以下过程计算而得到：

```
//vma.c:do_mmap
if(vma_type == VMA_FILE) {
    end_addr = mm->lower_bound;
    start_addr = end_addr - PGROUNDUP(len);
    mm->lower_bound = start_addr;
}
```

其中mm->lower\_bound是进程的字段mm维护的一个属性，表示文件映射区的最低地址。每次有新的文件映射它就会降低。当最低的VMA被unmap后，YuanShen OS会将其进行回调。

由于在文件内存映射的实现上我们同样采用了lazy allocation的思想，mmap并不会发生文件读写，而仅仅是添加一个新的VMA。直到进程访问到文件映射页时，由于映射未建立，因此会发生缺页中断，并将对应的数据从文件写到页面内。将数据读入页面的逻辑由mmap\_函数完成：

```
//vm.c:mmap_
int mmap_(struct vm_area_struct *vma, uint64 va) {
    struct proc *p = myproc();
    int flag = 0;
    if(vma->vm_prot & PROT_READ) {
        flag |= PTE_R;
    }
    if(vma->vm_prot & PROT_WRITE)
        flag |= PTE_W;
    flag |= PTE_U;
    void *pa;
    if((pa = kalloc()) == 0)
        panic("mmap_: no free pages");
    memset(pa, 0, PGSIZE);
    begin_op();
    ilock(vma->vm_file->ip);
    readi(vma->vm_file->ip, 0, (uint64)pa, vma->vm_pgoff + PGROUNDDOWN(va -
vma->vm_start), PGSIZE);
    iunlock(vma->vm_file->ip);
    end_op();
    mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, flag);
    return 0;
}
```

核心逻辑就是建立具有合适访问权限的内存映射，然后使用readi函数从对应的偏移量读取数据。

munmap的逻辑是类似的，如果我们要解除一部分的VMA，那么就要调整它的空间范围。YuanShen OS要求unmap的区间必须不能在某个VMA的中间，而只能对应它的开始或者结束部分：

```
//sysfile.c: sys_munmap
if(addr == vma->vm_start) {
    vma->vm_start += unmap_size;
    vma->vm_pgoff += unmap_size;
} else if((addr + len) == vma->vm_end) {
    vma->vm_end -= unmap_size;
}
```

除了缩减或者去除VMA，我们还需要根据VMA的flag确定页面是进程私有的，还是要写回到文件，这部分逻辑由unmap\_mmap\_area完成。如果有可以写回的，那么就把页面写回文件，不然我们只是单纯的把页面给free掉。

```
unmap_mmap_area(myproc()->pagetable, addr, (unmap_size >> 12), vma);
```

如果一个VMA被彻底的移除了(vm\_start == vm\_end)，那么它就应该被移除出进程的VMA链表。同时，我们可以通过fileclose把对应的文件结构体的引用计数减1:

```
//sysfile.c:munmap
if(vma->vm_start == vma->vm_end) {

    fileclose(vma->vm_file);

    if(vma->vm_next != 0)
        vma->vm_next->vm_prev = vma->vm_prev;
    if(vma->vm_prev != 0)
        vma->vm_prev->vm_next = vma->vm_next;
    else
        myproc()->mm.mmap = vma->vm_next;
    kfree((void*)vma);
}
```

当进程建立或者退出的时候，我们需要插入或者删除对应的VMA。例如在进程退出的时候会调用:

```
//vma.c
unmap_all_vma(&p->mm);
```

这个函数会把所有脏的页面根据需要写回磁盘，释放进程所有的VMA结构体，并且重置mm->lower\_bound保障下次使用进程的时候仍然是从trapframe往下建立文件映射区。

同样在进程调用exec，建立新的地址空间的时候，我们要根据elf格式建立其新的VMA:

```
//exec.c
int
exec(char *path, char **argv)
```

```

{
    ...
    struct proc *p = myproc();
    unmap_all_vma(&p->mm); // unmap all vmas

    ...//加载inode, 根据elf header的信息将程序的.text和.data segment加载入地址空间。

    p = myproc();
    uint64 oldsz = p->sz;
    sz = PGROUNDUP(sz);

    do_mmap(&p->mm, 0, sz, 0, 0, MAP_SHARED, 0, PROT_EXEC | PROT_READ |
    PROT_WRITE, VMA_TEXT);

    uint64 sz1;
    if((sz1 = uvmmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) ==
    0)
        goto bad;

    //stack area, also not lazily allocated
    do_mmap(&p->mm, sz1 - USERSTACK * PGSIZE, USERSTACK * PGSIZE, 0, 0,
    MAP_SHARED, 0, PROT_READ | PROT_WRITE, VMA_STACK);

    .....
}

```

首先, 由于exec后进程的地址空间被摧毁, 因此我们需要unmap原先所有的VMA结构体:

```

struct proc *p = myproc();
unmap_all_vma(&p->mm); // unmap all vmas

```

然后我们就可以为不同的区域设置不同的VMA:

```

//exec.c:exec
do_mmap(&p->mm, 0, sz, 0, 0, MAP_SHARED, 0, PROT_EXEC | PROT_READ |
    PROT_WRITE, VMA_TEXT);
...
do_mmap(&p->mm, sz1 - USERSTACK * PGSIZE, USERSTACK * PGSIZE, 0, 0,
    MAP_SHARED, 0, PROT_READ | PROT_WRITE, VMA_STACK);

```

## 6.1 exec

YuanShen OS的进程在执行exec系统调用后, 并不会立即将程序的.text,.data等program segment载入进程地址空间, 而仅仅将对应的VMA插入到进程维护的一个链表内,如下图所示:

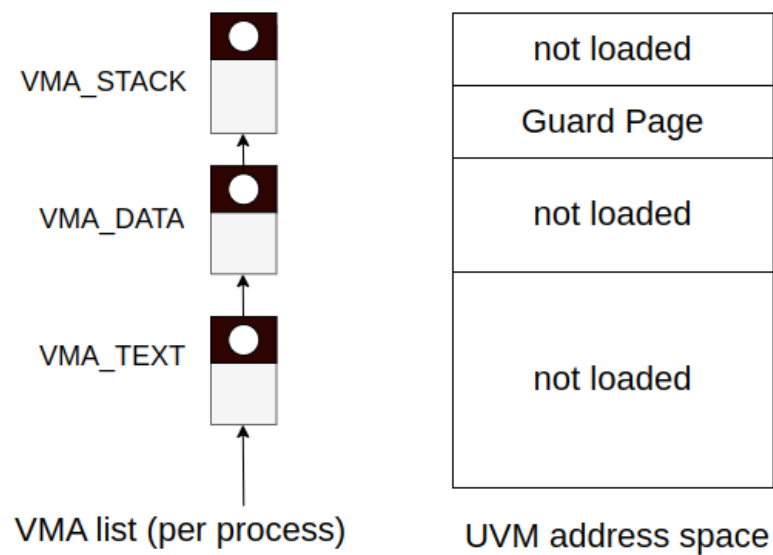


Fig7. exec - 惰性加载程序的程序段(原创)

每个VMA保存了对应地址范围的元信息，例如类型VMA\_TEXT的vma会维护一个file结构体的指针，文件偏移量以及.text段占据文件中的多少个字节。我们不妨打印出一个进程的VMA来观察一下,以sh为例，如下图所示：

```
init: starting sh
name: init, pid: 2-----
VM Area Struct:
  Start Address: 0x8000
  End Address: 0x9000
  Flags: 1
  Protection: 3
  File: 0x0000000000000000
  Page Offset: 0
  File Size: 0
  VMA Type: 0x8 (VMA_STACK)
VM Area Struct:
  Start Address: 0x2000
  End Address: 0x7000
  Flags: 2
  Protection: 3
  File: 0x000000008002a968
  Page Offset: 3000
  File Size: 10
  VMA Type: 0x32 (VMA_DATA)
VM Area Struct:
  Start Address: 0x0
  End Address: 0x2000
  Flags: 2
  Protection: 5
  File: 0x000000008002a968
  Page Offset: 1000
  File Size: 1e10
  VMA Type: 0x4 (VMA_TEXT)
page table 0x0000000080fb7000
..va 0x0000003fc0000000 : idx 255 pte 0x00000000203ee401 pa 0x0000000080fb9000
.. ..va 0x0000003fffe00000 : idx 511 pte 0x00000000203ee801 pa 0x0000000080fba000
.. .. ..va 0x0000003ffffffe000 : idx 510 pte 0x00000000203ba807 pa 0x0000000080eea000
.. .. ..va 0x0000003fffffff000 : idx 511 pte 0x0000000020002c0b pa 0x000000008000b000
-----
$
```

Fig8. 刚创生的进程的页表以及vma一览

我们可以看到程序sh在被加载时的内存布局，每个VMA维护了对应的文件指针，文件偏移量以及段的长度。只不过通过观察打印出来的页表可以注意到，在这个时候sh程序的.text和.data段并没有真的被装载入内存。

想要观察这个结果，只需要解除exec.c:exec函数最下面几行的注释:

```
//exec.c:exec
printf("name: %s, pid: %d-----\n", p->name, p->pid);
print_vma_list(p->mm.mmap);
printf("-----\n");
```

为了在exec被调用时正确地设置vma，我们查阅了ELF文件的spec[3]，如下图所示:

**Figure 2-1: Program Header**

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

**Fig9. ELF程序段格式(网图)**

其中p\_vaddr表示segment被装载入的虚拟地址，p\_filesz表示该段在文件中的长度，而p\_memsz表示该段在内存中的最小长度，YuanShen OS主动为它做了页对齐的处理。而p\_flags则最终被用于设置页面的访问权限。

据此，我们在exec函数中加入了对应的逻辑，代码如下:

```
if(flags2perm(ph.flags) & PTE_X){
    if(-1 == do_mmap(&myproc()->mm, ph.vaddr, PGROUNDUP(ph.memsz), ph.off,
ph.filesz, MAP_PRIVATE, f, PROT_EXEC | PROT_READ, VMA_TEXT)){
        panic("seg mapping fault");
    }
}
else
    if(-1 == do_mmap(&myproc()->mm, ph.vaddr, PGROUNDUP(ph.memsz), ph.off,
ph.filesz, MAP_PRIVATE, f, PROT_WRITE | PROT_READ, VMA_DATA)){
        panic("seg mapping fault");
    }
f = filedup(f);
sz1 = PGROUNDUP(ph.vaddr + ph.memsz);
...
```



看着代码有点复杂，实际上不过是检查`ph.flags`是否有可执行的权限，如果有则使用`do_mmap`创建`VMA_TEXT`类型的`vma`，不然就创建`VMA_DATA`类型的`vma`。因此，YuanShen OS的一个局限性在于暂时没有支持动态链接的`.dynamic`段等的功能。

之后我们对STACK使用一样的方法:

```
do_mmap(&p->mm, sz1 - USERSTACK * PGSIZE, USERSTACK * PGSIZE, 0, 0,
MAP_SHARED, 0, PROT_READ | PROT_WRITE, VMA_STACK);
```

对于第一个进程，由于它一开始并没有执行`exec`，我们需要手动的创建一个类型为`VMA_TEXT`的VMA:

```
//proc.c:userinit
do_mmap(&p->mm, 0, PGSIZE, 0, 0, MAP_PRIVATE, 0, PROT_EXEC | PROT_READ |
PROT_WRITE, VMA_TEXT);
```

当程序从`exec`返回后就会发生缺页异常，因为上述的各种段均没有实际地载入内存。和`mmap`的实现方法一样，缺页异常处理程序会检查VMA，然后将对应的段加载进内存，如下图所示:

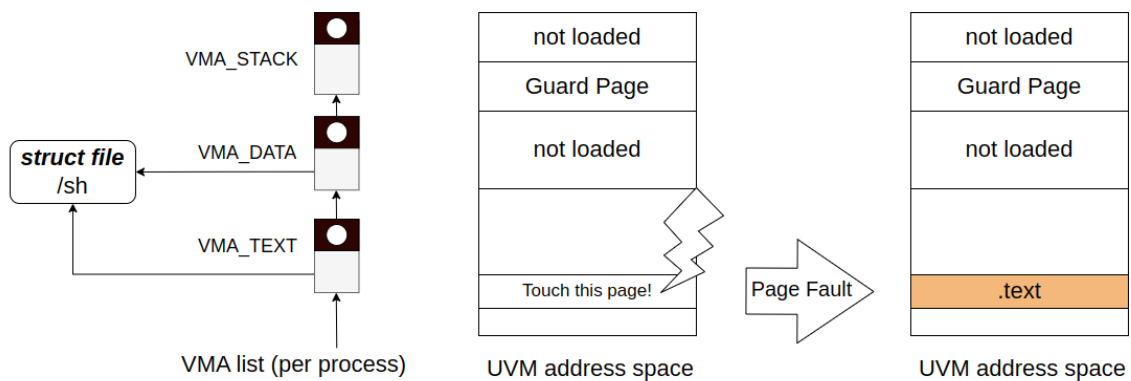


Fig10. 直到缺页异常再加载程序段(原创)

随着程序的执行，越来越多的部分会被加载进内存，最终我们会得到进程的地址空间可能的一种样貌，如下图所示:

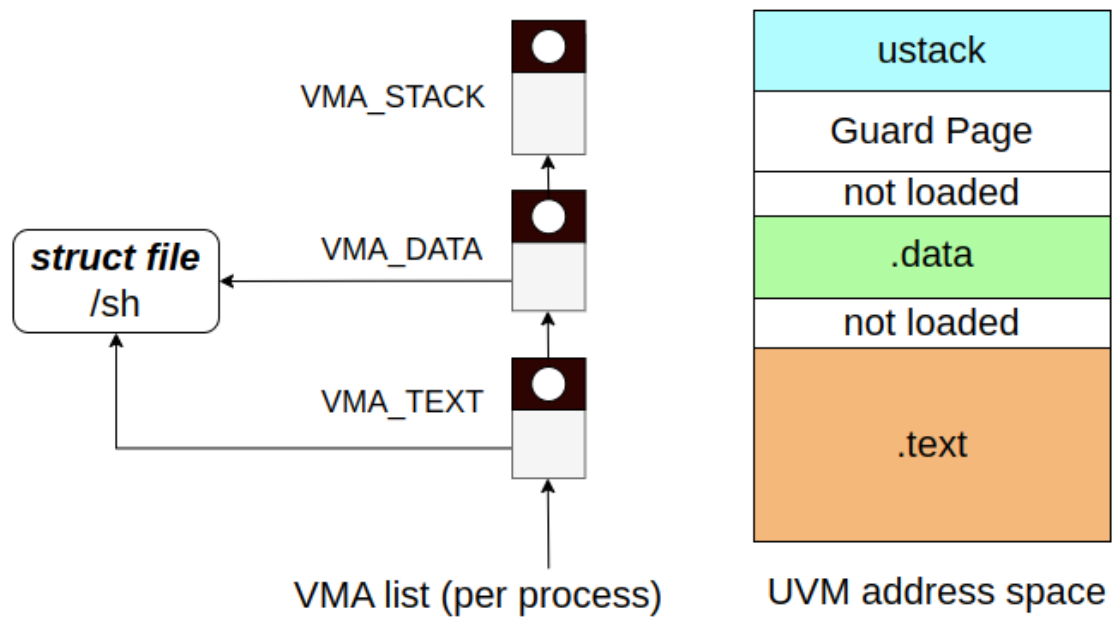


Fig11. 经过一段时间执行后的进程地址空间(原创)

事实上，对于那些从来没有使用过的页，我们也没有必要把它们加载入内存。

7.1 测试

7.1.1 开机测试

编写完代码后，通过在命令行输入make qemu进行开机。运行结果如下图所示：

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -s
false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,d
0,hostfwd=udp::26999-:2000,hostfwd=udp::31999-:2001 -object filter-dump,id=ne
tdev=net0,bus=pcie.0

xv6 kernel is booting

hart 1 starting
init: starting sh
$
```

我们可以清晰的看到YuanShen OS成功的进行了开机。

7.1.2 forktest测试

开机后输入forktest，该测试程序会进行1000次fork，父进程会等待子进程退出后才退出。我们打开了一部分的调试信息的注释，forktest的测试结果如下图所示：

```
$ forktest
COW page detected sh at 0x00000000000008f88
COW page detected sh at 0x00000000000002010
lazy allocation happened: sh at 0x00000000000009008
lazy allocation happened: sh at 0x00000000000018f48
fork test
COW page detected forktest at 0x00000000000002fa8
fork test OK
COW page detected sh at 0x00000000000008f88
$ COW page detected sh at 0x00000000000002020
ls
COW page detected sh at 0x00000000000008f88
COW page detected sh at 0x00000000000002010
lazy allocation happened: sh at 0x00000000000009008
lazy allocation happened: sh at 0x00000000000018f48
.          1 1 1024
..         1 1 1024
README    2 2 5678
cat        2 3 53256
echo       2 4 52144
forktest   2 5 16544
grep       2 6 56600
init       2 7 52576
kill       2 8 52136
ln         2 9 51944
ls         2 10 55168
mkdir      2 11 52216
rm         2 12 52192
sh         2 13 70296
```

### 7.1.3 mmaptest测试

mmap测试的内容涉及以不同的共享选项，保护权限去打开测试文件，并进行读写操作验证mmap是否能正确的将文件内容读入地址空间的合适位置。我们允许page\_fault\_hanler函数内的调试信息打印，得到运行结果为下图所示：

```

mmap
test fork: OK
test munmap prevents access
COW
mm->lower bound : 3ffffef000mmap
mmap
COW
mmap
mmap
VM Area Struct:
  Start Address: 0x3ffffed000
  End Address: 0x3ffffee000
  Flags: 1
  Protection: 3
  File: 0x000000008002a828
  Page Offset: 0
  File Size: 8192
  VMA Type: 0x1
usertrap(): unexpected scause 0xd pid=5
             sepc=0xa30 stval=0x3ffffee000
COW
mmap
mmap
VM Area Struct:
  Start Address: 0x3ffffee000
  End Address: 0x3ffffef000
  Flags: 1
  Protection: 3
  File: 0x000000008002a828
  Page Offset: 4096
  File Size: 8192
  VMA Type: 0x1
usertrap(): unexpected scause 0xd pid=6
             sepc=0xab8 stval=0x3ffffed000
COW
COW
test munmap prevents access: OK
test writes to read-only mapped memory
COW
mm->lower bound : 3ffffed000permission not correctVM Area Struct:
  Start Address: 0x3fffffeb000
  End Address: 0x3fffffed000
  Flags: 1
  Protection: 1
  File: 0x000000008002a828
  Page Offset: 0
  File Size: 8192
  VMA Type: 0x1
usertrap(): unexpected scause 0xf pid=7
             sepc=0xbfe stval=0x3fffffeb000
test writes to read-only mapped memory: OK
mmptest: all tests succeeded

```

#### 7.1.4 其他测试

为了观察其它的测试有没有受到影响，我们对它们也进行了一番测试。以testCFS为例,此测试程序涉及创建20个并行的进程以测试CFS调度器的公平性，我们借用它来进一步验证虚拟内存模块在多进程环境下的稳健性。测试结果如下图所示：

```
7, 1510000, 3655408
COW page detected testCFS at 0x00000000000008fa8
4, 1510000, 3655408
COW page detected testCFS at 0x00000000000008fa8
4, 1500000, 3631200
COW page detected testCFS at 0x00000000000008fa8
4, 1510000, 3655408
COW page detected testCFS at 0x00000000000008fa8
4, 1520000, 3679616
COW page detected testCFS at 0x00000000000008fa8
4, 1530000, 3703824
COW page detected testCFS at 0x00000000000008fa8
4, 1600000, 3873280
COW page detected testCFS at 0x00000000000008fa8
4, 1580000, 3824864
COW page detected testCFS at 0x00000000000008fa8
4, 1670000, 4042736
COW page detected testCFS at 0x00000000000008fa8
4, 1630000, 3945904
COW page detected testCFS at 0x00000000000008fa8
4, 1610000, 3897488
COW page detected testCFS at 0x00000000000008fa8
3, 1610000, 3134187
COW page detected testCFS at 0x00000000000008fa8
3, 1560000, 3036852
COW page detected testCFS at 0x00000000000008fa8
3, 1570000, 3056319
COW page detected testCFS at 0x00000000000008fa8
3, 1590000, 3095253
COW page detected testCFS at 0x00000000000008fa8
3, 1600000, 3114720
COW page detected testCFS at 0x00000000000008fa8
3, 1590000, 3095253
COW page detected testCFS at 0x00000000000008fa8
3, 1610000, 3134187
```

该程序会不断的进行fork，并在子进程内完成某个固定时长的任务，运行结束后打印自己的虚拟运行时间以及实际运行时间。我们看到testCFS测试成功的触发了COW的缺页异常处理程序并维持正常的运行。

## 参考资料

[1] 浙江大学. (2024). 操作系统实验指导五. 取自 [https://zju-sec.github.io/os24fall-stu/lab5/#\\_8](https://zju-sec.github.io/os24fall-stu/lab5/#_8)

[2] Columbia University. (n.d.). W4118: Linux memory management. 取自 <https://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l20-adv-mm.pdf>

[3] Carnegie Mellon University. (n.d.). Executable and Linkable Format (ELF). 取自 <https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf>