

# 操作系统实验报告

18364067 罗淦元

## 一、实验要求

在github上创建os-assignment2项目，提供

- (1) 虚存管理模拟程序源代码及结果（存成文本文件）；
- (2) 实验报告（word/pdf），包含所有实验的基本过程描述。

## 二、实验过程

### 1.虚存管理模拟程序

#### 1.1 Chapter 10. Programming Projects: Designing a Virtual Memory Manager (OSC 10th ed.)，30分。

- (1) 保存为vm.c，使用如下测试脚本test.sh，进行地址转换测试，并和correct.txt比较

```
#!/bin/bash -e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm BACKING_STORE.bin addresses.txt > out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

注：本小题不要求实现Page Replacement，TLB分别实现FIFO和LRU两种策略。

实验思路首先是通过一个掩码从读取的虚拟地址中提取页码和偏移，然后从TLB或页表中获取对应的帧码，并通过偏移读取到物理内存对应位置。当页表中找不到对应的页码则从后备存储中载入到内存里。从address.txt文件中获取的是一个16位的数字，其前8位是页码，后八位是偏移。因此可以通过位操作来提取出两者。如下。

```
/*页码和偏移的掩码，用于从虚拟地址地区页号和偏移*/
uint8_t pageMask(uint32_t integer){
    uint32_t mask = 255;           //和255进行与操作能够提取出后8位
    integer = integer >> 8;         //如果要提取前八位，可以先右移
    uint8_t page = integer & mask;
    return page;
}

uint8_t biasMask(uint32_t integer){
    uint32_t mask = 255;
    uint8_t bias = integer & mask;
    return bias;
}
```

首先是FIFO算法，通过定义一个TLB结构和page结构来形成TLB和数组。并通过一个16和256大小的数组，以队列的方式组织。每次在TLB中查找目标页后，如果找到则返回对应帧号。若找不到则在页表中重复查询操作。更新TLB时，则通过或运算直接覆盖TLB中最后一条条目的下一条，即将最久以前加入的条目直接覆盖为新条目。由于FIFO的实现非常简单，在这里不再过多叙述。相关完整源代码已经提交到github上vm.c文件中。

```
typedef struct {
    int page;
    int frame;
} TLBelem;

/*用FIFO时使用的page结构*/
typedef struct {
    int page;
    int frame;
} page;

int searchTLB(uint8_t page){
    for(int i = 0; i < TLB_SIZE; i++){
        if(TLB[i].page == page){
            //printf("TLB hit! on %d\n", i);
            return TLB[i].frame;
        }
    }
    //printf("TLB miss!\n");
    return -1;
}

void TLB_update_FIFO(int page, int frame){
    //printf("TLB_update_FIFO\n");
    TLB[TLB_HEAD].page = page;
    TLB[TLB_HEAD].frame = frame;
    TLB_HEAD = (TLB_HEAD+1)%TLB_SIZE;
}

int searchPageTable(uint8_t page){
    for(int i = 0; i < PAGETABLE_SIZE; i++){
        if(pageTable[i].page == page){
            //printf("page hit!\n");
            return pageTable[i].frame;
        }
    }
    //printf("page miss!\n");
    return -1;
}

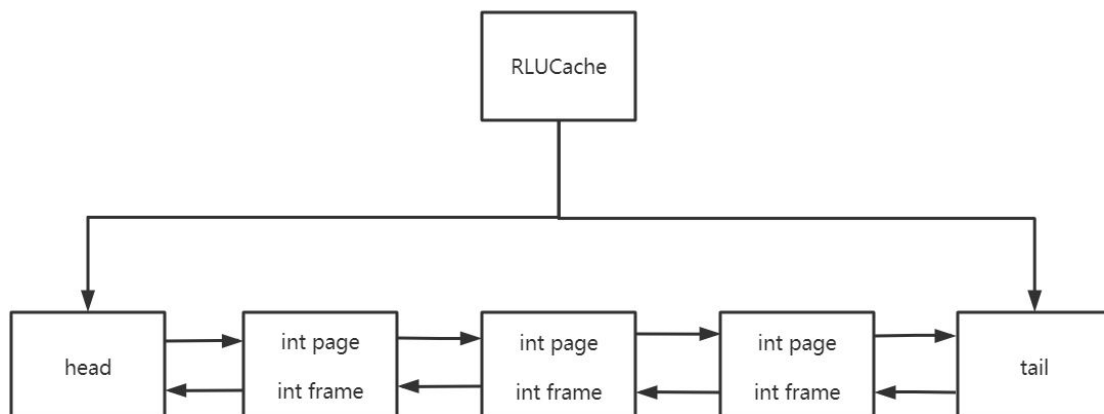
void pageTable_update_FIFO(int page, int frame){
    int swapOut;
    pageTable[PAGE_HEAD].page = page;
    pageTable[PAGE_HEAD].frame = frame;
    PAGE_HEAD = (PAGE_HEAD+1)%PAGETABLE_SIZE;
}
```

在test.sh中运行结果如下。

```
osc@ubuntu:~/final-src-osc10e/ch10$ sh ./test.sh
Compiling
Running vm
Comparing with correct.txt
```

而LRU算法相对要复杂一些。如果使用一个整数来记录每个TLB或页表中节点的最长未访问时间，需要遍历整个TLB或页表来查找该节点。而同时和FIFO类似，如果采用遍历的方式来查找需要的页条目，两个操作（删除最久未访问节点和查找条目）都将会达到 $O(N)$ 的复杂度。其访问效率将非常低，且随着页表和TLB大小的增大访问时间变得不可接受。

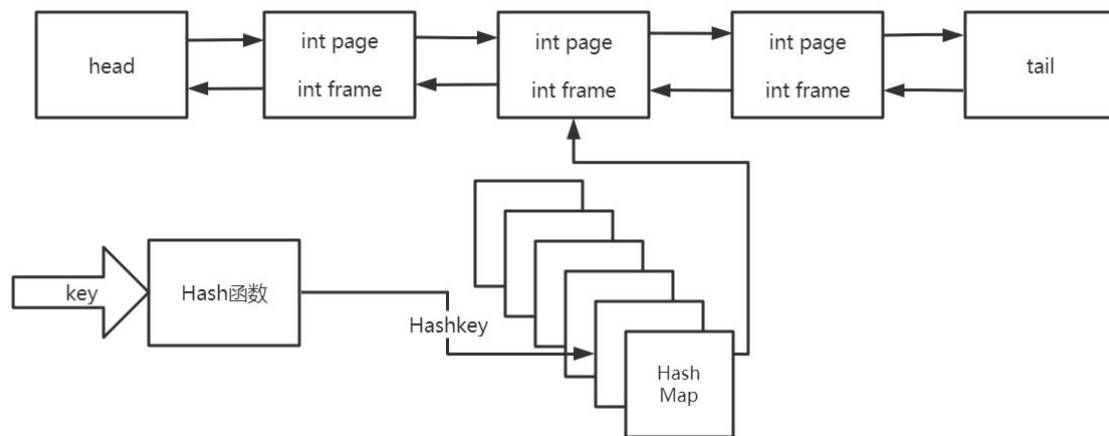
考虑到这一问题，本次实验决定尝试使用双向链表来维护LRU缓存结构。LRU缓存中带有指向一个双向链表首尾的指针，每次增加条目时，只要没有超出容积，就在表头添加一个节点，从而使得新添加的节点总在更靠近表头的位置。



```
/*双向链表结构*/
typedef struct DLinkedNode{
    int key;
    int frame;
    struct DLinkedNode* prev;
    struct DLinkedNode* next;
} DLinkedNode;
/*双向链表结构定义结束*/
```

而每次访问一个条目时，将会把该条目移动至表头，从而使得表尾总是最长时间未访问条目。通过这种方法，删除最久未访问节点的时间复杂度将被降为 $O(1)$ （通过删除`tail->prev`直接删除该节点），从而大大提高了效率。

除此之外，本实验使用哈希表来完成key值到节点的映射。这种做法的缺点很明显，例如要实现一个16条目的LRU缓存，则最少需要一块额外的最少16个哈希表条目大小的空间用于存放哈希表。但是相应的，在最理想情况下，我们能够实现对于条目的查找复杂度为 $O(1)$ ，而最糟糕的情况下也不会超过遍历的查找方式 $O(N)$ 。



```
//哈希表元素，包含一个键值、一个数据指针和一个状态
//其中，数据指针指向LRU缓存维护的双向链表中地元素，本身不存储值
//stat在Empty状态表示未使用，valid为已使用并分配内存，unvalid表示已删除并分配内存
typedef struct{
    int key;
    DLinkedNode* node;
    HashStat stat;
}HashElem;
```

为了对内存实施清晰有效的管理，我们设定所有的内存完全交给双向链表管理，哈希表相关函数仅仅负责接收指针和key(在本题中即page)来建立key到指针的映射。同时使用求余运算作为哈希函数，这是可以在未来进行优化的。

```
//哈希函数:key对哈希表大小求余
size_t HashFunction(int key){
    return key%HashMaxSize;
}
```

最后，基于实现的HashMap和DLinkedNode结构，最终建立出了LRUCache结构用于进行LRU算法。

```
/*LRU缓存定义*/
/*LRU缓存由一张哈希表和一个双向链表组成*/
/*哈希表主要负责查找，链表维护了一个以节点最近*/
/*访问顺序排列的队列，并管理空间分配*/
typedef struct LRUCache{
    HashMap cache;
    int size;
    int capacity;
    DLinkedNode head;
    DLinkedNode tail;
}LRUCache;
```

在源代码vm.c中，本实验已经将哈希表、双向链表和LRU缓存结构的相关函数进行了模块化的封装，最后关键的函数在于put和get函数。

```
/*put函数负责向LRU中加入一个条目*/
/*如果key存在于LRU缓存中，则将其移到表头*/
/*否则，如果缓存未满，加入表头，如果已满，删除表尾并加入表头*/
```

```

void put(LRUCache* lc, int key, int value){
    int key_;
    if(HashMapFindKey(&(lc->cache), key, &key_)){           //如果哈希表有key元素
        LRURemoveNode(lc, (lc->cache.data[key_].node));      //删除key元素，并稍
        后在表头重加。由于其未从哈希表删除，我们可以通过下标访问被remove但未delete的节点
    }
    if(lc->size == lc->capacity) {                             //如果已满，删除尾部元素
        HashRemove(&(lc->cache), lc->tail->prev->key);
        LRURemoveTail(lc);
    }
    DLinkedNode* newNode = (DLinkedNode*)malloc(sizeof(DLinkedNode)); //建立新节
    点
    newNode->key = key; newNode->frame = value;                //节点的键
    值分别是虚拟内存页和帧页码
    LRUaddNode(lc, newNode);                                   //将节点添
    加到LRU缓存
    HashMapInsert(&(lc->cache), key, newNode);
}

/*get函数负责从哈希表中找到LRU链表节点，*/
int get(LRUCache* lc, int key){
    int key_;
    if(HashMapFindKey(&(lc->cache), key, &key_)==0) return -1; //在哈希表中找到
    key对应哈希表下标
    DLinkedNode* tempNode = lc->cache.data[key_].node;         //备份该节点
    LRURemoveNode(lc, tempNode);                               //删除该节点
    LRUaddNode(lc, tempNode);                                  //然后将其提到最前面
    return tempNode->frame;                                     //返回其值
}

```

其中put函数将接受一个新键值作为参数，首先检查确认链表中不存在这一键值，然后当LRU缓存已满，则删除链表中尾部元素，同时将为新节点分配内存并加入到链表头，接着更新哈希表。get函数则从哈希表中寻找键值对应的条目，如果找不到返回-1表示获取失败。否则将节点从链表中移除重加，即移动到头部，然后返回对应的值。基于这两个接口，我们可以很容易实现LRU算法并实现TLB和页表的更新。在test.sh中运行结果如下。

```

osc@ubuntu:~/final-src-osc10e/ch10$ sh ./test.sh
Compiling
Running vm
Comparing with correct.txt

```

由于vm.c在第二小题中需要进行主函数的更改，本实验将第一题封装为函数

```

int question1(int argc, char* argv[]);

```

只需要在主函数中将argc和argv直接传入到函数，并注意添加BACKING\_STORE.bin和addresses.txt参数即可。“-r”参数是可选的，参数可以使用“FIFO”和“LRU”选择使用的算法。当此项为空时将默认使用LRU算法进行测试。

(2) 实现基于LRU的Page Replacement; 使用FIFO和LRU分别运行vm (TLB和页置换统一策略)，打印比较Page-fault rate和TLB hit rate，给出运行的截屏。提示：通过getopt函数，程序运行时通过命令行指定参数。

在第一题中，本实现已经考虑了页表小于物理内存的情况，因此可以直接在源代码中改变页表大小常量。

```
#define PAGETABLE_SIZE 128 //页表大小
```

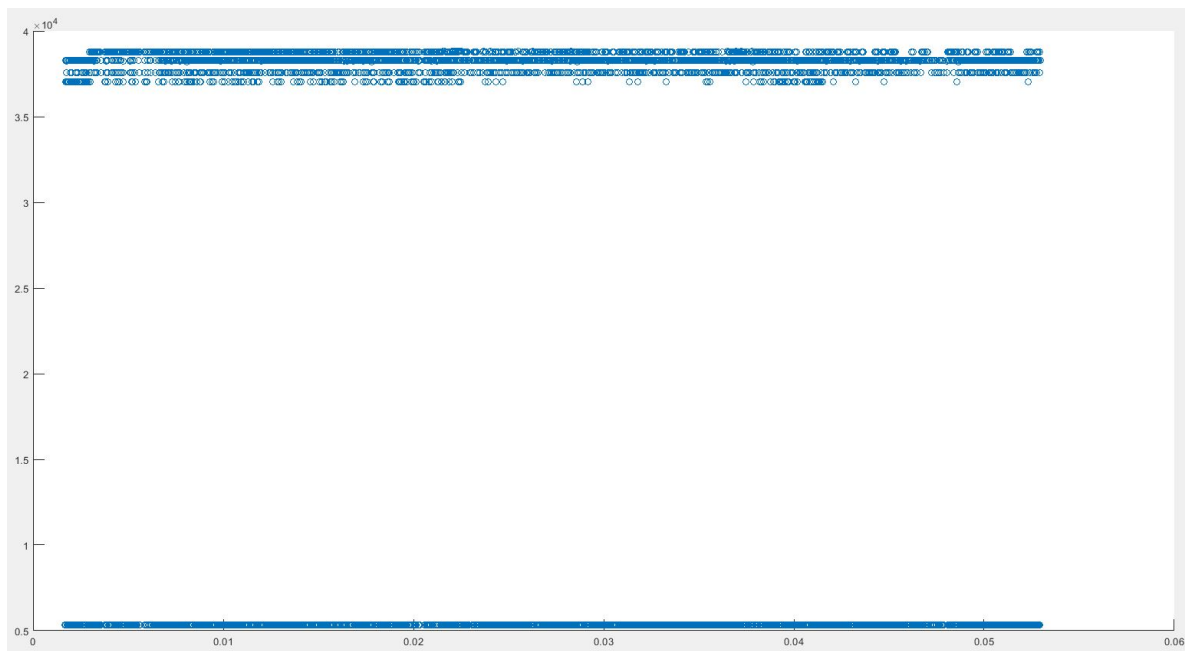
程序即能够正常进行页表Page Replacement。运行结果如图。

```
osc@ubuntu:~/final-src-osc10e/ch10$ ./vm BACKING_STORE.bin addresses.txt -rFIFO
TLBHitRate: 5.400000%
pageFaultRate: 53.800000%
osc@ubuntu:~/final-src-osc10e/ch10$ ./vm BACKING_STORE.bin addresses.txt -rLRU
TLBHitRate: 5.500000%
pageFaultRate: 53.700000%
```

## 1.2 编写一个简单trace生成器程序

可以用任意语言，报告里面作为附件提供。运行生成自己的addresses-locality.txt，包含10000条访问记录，体现内存访问的局部性（参考Figure 10.21, OSC 10th ed.），绘制类似图表（数据点太密的话可以采样后绘图），表现内存页的局部性访问轨迹。然后以该文件为参数运行vm，比较FIFO和LRU策略下的性能指标，最好用图对比。给出结果及分析，10分。

使用python的id函数获取对象虚拟地址，然后提取页号得到。采用的基本代码是21点强化学习使用的代码，相关代码见附件trace文件夹。得到的输出如图所示。



可以看到，内存访问地址具有很强的时间和空间局部性，这对于代码算法的优化能给我们很大的启发。

## 2.xv6-lab-2020页表实验 (Lab:page tables) , 20分

完成Print a page table任务。要求按图1格式打印页表内容；其中括号内表示页表项权限，R表示可读，W表示可写，X表示可执行，U表示用户可访问。物理页后的数字 (pa 32618) 表示第几个物理页帧。要求在报告中提供实现所需的源代码和运行截屏，代码要求有充分注释。然后，回答接下来的6个问题（分别对应代码注释行中的标签）。

```

page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 () pa 32618(th pages) //问题1
.. ..0: pte 0x0000000021fda401 () pa 32617(th pages)
.. .. ..0: pte 0x0000000021fdac1f (RWXU) pa 32619(th pages) //问题2
.. .. ..1: pte 0x0000000021fda00f (RWX) pa 32616(th pages) //问题3
.. .. ..2: pte 0x0000000021fd9c1f (RWXU) pa 32615(th pages) //问题4
..255: pte 0x0000000021fdb401 () pa 32621(th pages)
.. ..511: pte 0x0000000021fdb001 () pa 32620(th pages)
.. .. ..510: pte 0x0000000021fdd807 (RW) pa 32630(th pages) //问题5
.. .. ..511: pte 0x0000000020001c0b (RX) pa 7(th pages) //问题6

```

实现代码如下图，放置在vm.c文件中。为了格式化输出，我们可以使用一个格外的参数depth记录页表访问深度或者在vmprint中使用一个独立的子函数完成。这里选择前者实现，当调用外界vmprint时，depth初值应为0。

```

void vmprint(pagetable_t pagetable, int depth){
    if(depth == 0) printf("page table %p\n", pagetable); //最高层页表，输出页表地址
    for(int i = 0; i < 512; i++){ //遍历页表项
        pte_t pte = pagetable[i]; //获取一个条目
        if((pte & PTE_V)){ //如果条目有效
            printf("..");
            uint64 pa = PTE2PA(pte); //从页表项中获取物理地址
            for(int j = 0; j<depth; j++) printf(" .."); //根据当前深度格式化输出
            printf("%d: pte %p ", i, pte);
            printf("(");
            if(pte & (PTE_R)) printf("R"); //查看是否有读权限
            if(pte & (PTE_W)) printf("W");
            if(pte & (PTE_X)) printf("X");
            if(pte & (PTE_U)) printf("U");
            printf(") ");
            //pa中第12位-55位是PPN，再减去一个起始地址得到页号
            printf("pa %d(th pages)\n", ((pa>>12)-0x80000));
            //如果pte不可读不可写不可执行，说明这页页表的子项应该是下一层页表的地址，则深度+1并输出下一层页表
            if((pte & (PTE_R|PTE_W|PTE_X)) == 0) vmprint((pagetable_t)pa,
depth+1);
        }
    }
}

```

其中PTE2PA () 函数在riscv.h中被定义为，即去掉RSW、D、A等位后左移12个offset位得到。

```
#define PTE2PA(pte) (((pte) >> 10) << 12)
```

而PTE\_V、PTE\_R等权限位掩码在riscv.h中被定义为

```

#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access

```

其运行结果如图所示。







```
//如果pa不是一页倍数、物理地址比栈
if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

memset(pa, 1, PGSIZE);           //内存初始化

r = (struct run*)pa;

acquire(&kmem.lock);
r->next = kmem.freelist;         //把物理地址pa指向的空间组织为链表
kmem.freelist = r;               //头结点指向空余内存中地址最大的
release(&kmem.lock);
}
```

使得地址大的空余内存存在freelist中更靠近表头。而调用kalloc分配物理内存时，就可以通过表头直接取得一页大小的内存空间。而获得地址时，kalloc函数直接从freelist表头获取地址，导致先分配的内存地址更高。

## 问题2：这是什么页？装载的什么内容？结合源代码内容进行解释。

这是一段 program segment，负责存储程序代码。它在exec.c中被分配

```
if((sz1 = uvmmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
```

其中，ph是一个Program section header结构，exec从initcode中被调用，通过readi函数从ip指向的文件中读取init的二进制代码到ph结构。

```
if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
```

然后通过uvmmalloc为init进程增长一次空间。因为xv6使用的是三层页表结果，在uvmmalloc及其中的mappage的过程中，创建出了打印结果中前三行的三层页表。然后loadseg将一段 program segment 载入到虚拟地址为ph.vaddr所指的位置中

```
if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

## 问题3：这是什么页，有何功能？为什么没有U标志位？

这一页在exec.c中的

```
if((sz1 = uvmmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
```

被分配，然后在其后

```
uvmclear(pagetable, sz-2*PGSIZE);
```

被马上取消了用户模式访问权限。exec在栈页下面会放置这样一个不可访问页guard page，这样程序试图使用超过一页的空间时就会因为访问到这一区域而报错，同时这一页能够帮助exec处理太大的参数arguments，使得copyout函数试图将变量复制到用户栈时，会发现目标页不可访问而返回-1。为了实现这一目的，其没有U标志位。

#### 问题4：这是什么页？装载的什么内容？指出源代码初始化该页的位置。

这是用户栈，和上一页一起在

```
if((sz1 = uvmmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
```

中被分配，并由uvmmalloc中的

```
memset(mem, 0, PGSIZE);
```

初始化为0后建立页表映射，并在exec.c的下一段中将argument和其他推入到用户栈中

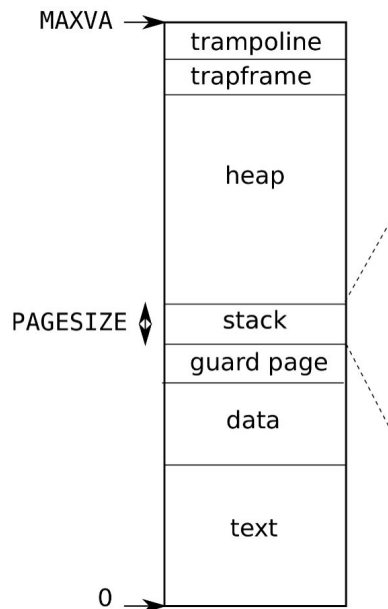
```
// 装载argument
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16;
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0) //推入一个
argument
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;
```

然后压入了指向这些argument的指针和一个空指针。

```
// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;
```

#### 问题5：这是什么页，为何没有X标志位？

这是trapframe。根据xv6 book 其位于虚拟地址的高位，保存了一些关键信息。



这些信息包括帮助trap函数判断是什么引起了中断的标志、系统调用号、内核栈顶地址等等。显然其不是可执行的数据，同时也不允许用户态的更改。

## 问题6：这是什么页，为何没有W标志位？装载的内容是什么？为何这里的物理页号处于低地址区域（第7页）？结合源代码对应的操作进行解释。

这是trampoline。这段代码不允许进行更改，并装载了trampoline.s中的代码。其虚拟页号总是位于整个虚拟地址的最高位（511），保证在不同的进程中都可以正确进入trampoline中。

由于其由系统内核创建。在kvm\_init中的kvm\_make中，按照先后顺序，内核将trampoline分配在了物理地址第7页。

```
pagetable_t
kvmmake(void)
{
    pagetable_t kpgtbl;

    kpgtbl = (pagetable_t) kalloc();           //分配一页用于放置根页表
    memset(kpgtbl, 0, PGSIZE);

    kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

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

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

    kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

    kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
    PTE_W);

    //将trampoline存储到物理内存，并添加页表映射
    kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X); //物
    理地址第7页

    // 映射内核栈
    proc_mapstacks(kpgtbl);
}
```

```

    return kpgtbl;
}

```

它完全由trampoline.s中载入，并只能读取和执行，同时只能属于内核，只能由内核对代码进行访问。每个进程虚拟地址中的trampoline都指向物理地址中这一位置

### 3.xv6-lab-2020内存分配实验 (Lab: xv6 lazy page allocation), 40分

#### 3.1 完成Lazy allocation子任务，要求echo hi正常运行，报告中可以描述自己的尝试过程，以及一些中间变量。

首先删除sys\_sbrk中的grow\_proc操作，仅仅更改进程的size。

```

uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;           //更改进程大小,但是暂未分配空间
    myproc()->sz += n;
    return addr;
}

```

考虑到n有可能为负数，其中如果n绝对值大于sz且为负数的情况，需要异常处理。

```

uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;

    struct proc* p = myproc();
    addr = p->sz;           //addr是p原大小,同时也是分配内存的起始位置
    if(n < 0){              //新大小比进程原大小小
        if(p->sz+n < 0) return -1;    //不能使程序大小为负
        uvmdealloc(p->pagetable, addr, addr+n);    //释放部分物理内存
    }
    p->sz += n;             //更新进程大小
    return addr;
}

```

而单纯这样处理，会导致usertrap的panic。原因是usertrap无法判断这一次trap为什么发生，以及uvmunmap函数发现有空间未分配。

```
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
          sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
```

为了解决这一问题，我们自然可以把目光放到这两个函数上。对于uvmunmap函数，有两个判断语句会对lazy allocation产生影响。

```
if((pte = walk(pagetable, a, 0)) == 0)
    panic("uvmunmap: walk");
if((*pte & PTE_V) == 0)
    panic("uvmunmap: not mapped");
```

前者通过将walk函数的alloc参数为0，将walk函数以一种巧妙的方式重新运用，使walk函数从上到下查询页表，并且当任意一级页表PTE\_V位无效时产生一次panic。如果正常返回，将得到最低一层的物理地址。第二条将判断这一地址是否有效。我们可以简单地跳过这两条判断，将使得uvmunmap对于这种未分配的情况视为一种正常情况忽略，当做该页成功释放。

```
if((pte = walk(pagetable, a, 0)) == 0)
    continue;
if((*pte & PTE_V) == 0)
    continue;
```

而在实验中，一开始发现在注释掉panic语句后会发生kfree panic错误，经过仔细检查发现，如果不在if语句中增加continue语句，会导致这些未被分配的内存存在其后

```
if(do_free){                                //&& (*pte & PTE_V) != 0
    uint64 pa = PTE2PA(*pte);
    kfree((void*)pa);
}
```

产生尝试释放未分配内存的错误，continue是不可以去除的。

对于usertrap,我们根据提示可以知道当r\_scause()寄存器为13或15时，说明当次陷入由页读取或也写入造成，即产生了一次缺页错误。那么对于这种情况，我们需要做的就是为缺失的页分配一块内存，然后返回到读写页的部分。

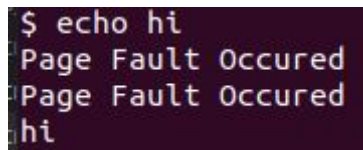
```
else if (r_scause() == 13 || r_scause() == 15){                //如果发生缺页错误
    printf("Page Fault Occured\n");
    uint64 va = r_stval();                                       //r_stval()得到产生缺页错误的
    位置
    if(va < p->sz && va > PGROUNDDOWN(p->trapframe->sp)){      //如果位置有效
        uint64 ka = (uint64) kalloc();                          //尝试分配一块地址
        if (ka == 0) p->killed = 1;                              //分配失败，杀死进程
        else{
            memset((void*)ka, 0, PGSIZE);                       //分配成功，初始化
            va = PGROUNDDOWN(va);                                //地址取整
            //尝试映射空间到页表的va位置
            if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W | PTE_R) != 0){
                //如果失败，释放空间并杀死进程
                kfree((void*)ka);
                p->killed = 1;
            }
        }
    }
}
```

```

//曾经遇到问题的位置
else{
    p->killed = 1;
}
}
} 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;
}
}

```

到此位置，echo hi指令产生缺页错误时，就能够被正确分配空间。



```

$ echo hi
Page Fault Occured
Page Fault Occured
hi

```

在实验中碰到的一个问题出现在上述标注的位置。因为这一部分和紧接着的else语句即为相似，很容易被忽略掉。这会导致程序因为if语句进入到缺页错误处理区块，而当分配的内存非法时，无法进入p->killed = 1;语句导致进程不被杀死进入死循环。

**3.2 完成Lazytests and Usertests子任务。对于Lazytests，要求屏幕输出如下图所示；对于usertests任务，要求通过所有除sbrkarg之外的测试。给出运行截屏。在阅读报告中提供代码修改片段，说明针对哪些文件，哪些函数进行了修改，新代码加上充分注释；可以写一些体会。**

lazytests和usertests主要需要解决的问题有

- 处理sbrk参数为负数的情况（已完成，前一题sysproc.c的sys\_sbrk()函数中）
- 页错误发生在一个超过程序大小的虚拟地址的情况（已完成，前一题trap.c的usertrap()函数中）
- fork () 函数中，父进程到子进程的内存拷贝问题
- 系统调用时，发生了地址的内存未分配的情况
- kalloc未能成功分配内存的情况（已完成，前一题trap.c的usertrap()函数中）
- 在用户栈低位的无效页情况（已完成，前一题trap.c的usertrap()函数中）

那么需要解决的主要问题主要集中在fork函数和系统调用上。

对于fork函数，我们首先追踪到造成错误的位置

```

if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    //复制父进程的用户内存到子进程
    freeproc(np);
    //创建子进程失败，释放子进程内存
    release(&np->lock);
    return -1;
}

```

这一位置调用的uvmcopy函数使用了和uvmunmap()类似的做法，通过walk和PTE\_V来对页表页的有效性进行判断。



```

if((pte = walk(old, i, 0)) == 0)
    panic("uvmcopy: pte should exist");
if((*pte & PTE_V) == 0)
    panic("uvmcopy: page not present");

```

同样的解决方案也是可行的。

```

if((pte = walk(old, i, 0)) == 0)
    continue;
if((*pte & PTE_V) == 0)
    continue;

```

而对于系统调用问题，主要就是要找到系统调用中通过什么函数把虚拟地址映射到物理地址中并添加相应页表。通过查询可以看到，copyin/copyout/copyinstr/loadseg等函数均使用walkaddr函数来转换虚拟和物理内存地址。那么我们可以在walkaddr中处理缺页问题，保证获得一个有效虚拟地址时能够返回一个有效的物理地址。

具体的实现和前面的usertrap相似。

```

// Look up a virtual address, return the physical address,
// or 0 if not mapped.
// Can only be used to look up user pages.
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
    pte_t *pte;
    uint64 pa;

    if(va >= MAXVA)
        return 0;

    pte = walk(pagetable, va, 0);                //获取物理地址

    struct proc* p = myproc();                  //当前进程
    if(pte == 0 || (*pte & PTE_V) == 0)          //如果物理地址无效，尝试分配
    {
        if (va >= p->sz || va < PGROUNDUP(p->trapframe->sp)) //超出进程大小或超出用户栈
            return 0;                                       //转换失败
        uint64 ka = (uint64)kalloc();                //尝试分配内存
        if (ka == 0) return 0;                        //如果kalloc分配失败，返回

        //如果映射物理地址到虚拟地址失败，释放空间并返回
        if (mappages(p->pagetable, PGROUNDUP(va), PGSIZE, ka,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            kfree((void*)ka);
            return 0;
        }
        //分配成功，返回物理地址
        return ka;
    }

    /*
    if(pte == 0)
        return 0;
    if((*pte & PTE_V) == 0)
        return 0;
    */

```

```

*/

if((*pte & PTE_U) == 0)
    return 0;
pa = PTE2PA(*pte);
return pa;
}

```

到此就完成了简单的lazy allocation机制。其运行结果如下。由于lazyytests中的fault address和mappages failed过长，故和测试本身分开输出截图。

**usertests:**

```

init: starting sh
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
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: 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 exectest: 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 validateptest: 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

```

**lazyytests:**



```
usertrap(): fault address 0x0000000023584000 beyond heap range
usertrap(): fault address 0x00000000235c4000 beyond heap range
usertrap(): fault address 0x0000000023604000 beyond heap range
usertrap(): fault address 0x0000000023644000 beyond heap range
usertrap(): fault address 0x0000000023684000 beyond heap range
usertrap(): fault address 0x00000000236c4000 beyond heap range
usertrap(): fault address 0x0000000023704000 beyond heap range
usertrap(): fault address 0x0000000023744000 beyond heap range
usertrap(): fault address 0x0000000023784000 beyond heap range
usertrap(): fault address 0x00000000237c4000 beyond heap range
usertrap(): fault address 0x0000000023804000 beyond heap range
usertrap(): fault address 0x0000000023844000 beyond heap range
usertrap(): fault address 0x0000000023884000 beyond heap range
usertrap(): fault address 0x00000000238c4000 beyond heap range
usertrap(): fault address 0x0000000023904000 beyond heap range
usertrap(): fault address 0x0000000023944000 beyond heap range
usertrap(): fault address 0x0000000023984000 beyond heap range
usertrap(): fault address 0x00000000239c4000 beyond heap range
usertrap(): fault address 0x0000000023a04000 beyond heap range
usertrap(): fault address 0x0000000023a44000 beyond heap range
usertrap(): fault address 0x0000000023a84000 beyond heap range
usertrap(): fault address 0x0000000023ac4000 beyond heap range
usertrap(): fault address 0x0000000023b04000 beyond heap range
usertrap(): fault address 0x0000000023b44000 beyond heap range
usertrap(): fault address 0x0000000023b84000 beyond heap range
```

running test lazy unmap

```
mappages failed, va=0x0000000000004000
mappages failed, va=0x000000000001004000
mappages failed, va=0x000000000002004000
mappages failed, va=0x000000000003004000
mappages failed, va=0x000000000004004000
mappages failed, va=0x000000000005004000
mappages failed, va=0x000000000006004000
mappages failed, va=0x000000000007004000
mappages failed, va=0x000000000008004000
mappages failed, va=0x000000000009004000
mappages failed, va=0x00000000000a004000
mappages failed, va=0x00000000000b004000
mappages failed, va=0x00000000000c004000
mappages failed, va=0x00000000000d004000
mappages failed, va=0x00000000000e004000
mappages failed, va=0x00000000000f004000
mappages failed, va=0x000000000010004000
mappages failed, va=0x000000000011004000
mappages failed, va=0x000000000012004000
mappages failed, va=0x000000000013004000
mappages failed, va=0x000000000014004000
mappages failed, va=0x000000000015004000
mappages failed, va=0x000000000016004000
mappages failed, va=0x000000000017004000
mappages failed, va=0x000000000018004000
mappages failed, va=0x000000000019004000
```

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
```