

# 操作系统大作业 2——实验报告

岳毅然 18364112 2018 级 智科二班  
[yueyr3@mail2.sysu.edu.cn](mailto:yueyr3@mail2.sysu.edu.cn)

本次实验基于 Linux 操作系统 final-src-osc10e 材料和 xv6-labs-2020 系统，经过试验，我对于地址的理解更为深入，对 Linux 的分页机制的知识了解到了更多，同时通过读写代码，实践能力有所提升，并且和数据结构课程所学到的理论知识相做结合，获益匪浅。

## 1.1. 虚存管理模拟程序实验

### 1.1 Designing a Virtual Memory Manager

先进先出 FIFO 策略单纯按照出入顺序来排列置换页面，优先淘汰最早进入内存的页面，亦即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。本实验在实现 vm.c 程序中直接使用带有顺序的数组。但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。

最近最久未使用 LRU 策略利用局部性原理，根据一个 task 在执行过程中过去的页面访问历史来推测未来的行为。当需要 kill 一个页面时，程序选择在最近一段时间内最久不用的页面予以淘汰。LRU 在迭代中可能存在多次顺序交换，因此在 vm.c 文件中使用双向链表实现管理。

```
struct TLB {
    int Page;
    int Frame;
}; //TLB结构

struct TLB_Node{
    struct TLB_Node* next;
    struct TLB_Node* prior;
    int Page;
    int Frame;
}; //双向链表实现的TLB节点

struct TLB_LinkedList{
    struct TLB_Node* head;
    struct TLB_Node* rear;
}; //LRU链表

struct TLB TLB_Array[TLB_SIZE];
struct TLB_LinkedList* TLB_List;
int TLB_Index = 0;
//数组-FIFO, 双向链表-LRU
```

图 1 – TLB 数据结构定义

切换页表部分，vm.c 文件中定义了两个初始值为 FIFO 的全局变量用来表示策略：TLB\_Flag、PageReplacement\_Flag，其中 0 表示 LRU 策略，1 表示 FIFO 策略（图 2）。

```
#define FIFO 1
#define LRU 0

int TLB_Flag = FIFO;
int PageReplacement_Flag = FIFO;
```

图 2-策略 flag 定义

设置函数 FIFO/LRU\_IF\_IN\_TLB/Page\_Table、Add\_FIFO/LRU\_To\_TLB/Page\_Table 四个函数，实现 TLB 和页表置换的搜索及存在判断、插入节点活动。具体代码见附件 vm.c。

本实验 int main()承担页置换、初始化、读入文件及测试相关任务。主函数中，根据题目指引，首先调用 getopt()函数分析命令行参数（图 3），注意声明 unistd.h 头文件。

```
int opt;
while ((opt = getopt(argc, argv, "t:p:n:")) != -1){
    switch (opt) { //默认页表缓存和页面切换策略为FIFO
        case 't':
            if(strcmp(optarg, "LRU") == 0){
                TLB_Flag = LRU; //切换TLB策略至LRU
            }
            break;
        case 'p':
            if(strcmp(optarg, "LRU") == 0){
                PageReplacement_Flag = LRU; //切换页表切换策略至LRU
            }
            break;
        case 'n':
            if(strcmp(optarg, "256") == 0){
                FRAMES = 256; //切换物理帧从128到256
            }
            break;
    }
}
```

图 3-getopt 函数调用代码

其他具体详细代码请见附件 vm.c 注释，读入 address.txt 地址，以默认策略在 TLB 里查询目的页面，若命中则 TLB\_Hits 累计（图 4），未命中则使用选定的策略进入页表搜索，查询到该页面则将其加入 TLB，若未查询到则将其添加进 Page Table 中（图 5）。

打印信息部分，指定的页面确认后寻找当前页的第一个空闲帧，虚拟地址为读入的对应行的虚拟地址，物理地址为帧的大小 frame 乘以页的大小 PAGE\_SIZE 加偏移量 offset 帧的大小乘以页大小的乘积加上偏移量，Value 为 bin 中对应的量。

```
if(frame != FALSE){//命中
    TLB_Hits++;
    Physical_Add = Offset + 256 * frame;
    printf("Virtual_Address: %d, Physical_Address: %d, Value: %d\n",
        Virtual_Add, Physical_Add, Main_Mem[frame * PAGE_SIZE + Offset]);
```

图 4-命中计数+1 并打印信息

```
else{
    if(PageReplacement_Flag == FIFO){
        frame = FIFO_IF_In_Page_Table(page);
    }
    else{
        frame = In_Page_Table_LRU(page);
    }
    if (frame == FALSE){
        Page_Faults++;//Page Fault, 向页表中添加节点
        if(PageReplacement_Flag == FIFO){
            frame = Add_FIFO_To_Page_Table(page);
        }
        else{
            frame = Add_To_Page_Table_LRU(page);
        }
        memcpy(Main_Mem + frame * PAGE_SIZE, Backing + page * PAGE_SIZE, PAGE_SIZE);
        //需要复制的文件来自backing file
    }Physical_Add = Offset + 256 * frame;
    printf("Virtual_Address: %d, Physical_Address: %d, Value: %d\n",
        Virtual_Add, Physical_Add, Main_Mem[frame * PAGE_SIZE + Offset]);

    if(TLB_Flag == FIFO) Add_FIFO_To_TLB(page, frame);//FIFO
    else Add_To_TLB_LRU(page, frame);//LRU
}
```

图 5-未命中则添加进 TLB 或页表

执行 sh test.sh 详见图 6，进行地址转换测试，并输出 out.txt 文件，和

correct.txt 一致。

```
#!/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
```

图 6 -test.sh 代码

```
Compiling
Running vm
Comparing with correct.txt
1,1000c1,1000
< Virtual_Address: 16916, Physical_Address: 20, Value: 0
< Virtual_Address: 62493, Physical_Address: 285, Value: 0
< Virtual_Address: 30198, Physical_Address: 758, Value: 29
< Virtual_Address: 53683, Physical_Address: 947, Value: 108
< Virtual_Address: 40185, Physical_Address: 1273, Value: 0
< Virtual_Address: 28781, Physical_Address: 1389, Value: 0
< Virtual_Address: 24462, Physical_Address: 1678, Value: 23
< Virtual_Address: 48399, Physical_Address: 1807, Value: 67
< Virtual_Address: 64815, Physical_Address: 2095, Value: 75
< Virtual_Address: 18295, Physical_Address: 2423, Value: -35
< Virtual_Address: 12218, Physical_Address: 2746, Value: 11
< Virtual_Address: 22760, Physical_Address: 3048, Value: 0
< Virtual_Address: 57982, Physical_Address: 3198, Value: 56
< Virtual_Address: 27966, Physical_Address: 3390, Value: 27
< Virtual_Address: 54894, Physical_Address: 3694, Value: 53
< Virtual_Address: 38929, Physical_Address: 3857, Value: 0
< Virtual_Address: 32865, Physical_Address: 4193, Value: 0
```

图 7 -out 运行输出

编写 test2.sh 文件（详细见附件文件），比对不同策略的效率，打印结果如图所示，可以发现 LRU 策略更为高效。

```
Compiling
Running vm FIFO
Running vm LRU
Comparing fifo.txt with lru.txt
1,2c1,2
< Page Faults Rate= 0.021100
< TLB Hits Rate= 0.850800
---
> Page Faults Rate= 0.019500
> TLB Hits Rate= 0.860200
```

图 8 -不同策略对比

## 1.2 Designing a Virtual Memory Manager

trace 程序并无太多实际意义，仅用于测试内核页面的调用并记录运行过程，故本题目未自行编写文件代码，选用了已有的基于 mnist 的神经网络分类模型开源代码，其结构是一个具有多层函数调用的轻量级架构，源代码绝大部分拷贝自 <https://blog.csdn.net/mheartwgo/article/details/87867533>，有小部分改动，具体见附件，为 2019 年深度学习课程的小组作业项目。对于本题而言，在原有基础上主要加入了调用记录部分 ID\_Print。运行后对生成的 addresses-locality.txt 绘制直方图如图 9 所示。

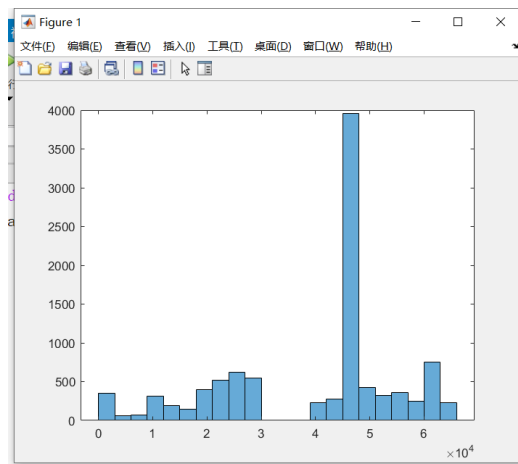


图 9 –addresses-locality.txt 直方图

以此为参数运行 vm.c，编写 test3.sh 文件，对比不同的策略，结果如图 10 所示。由此可见 LRU 在 Page Fault Rate 和 TLB Hit Rate 上有优势，但其数据结构更加复杂。

```
Compiling
Running vm FIFO
Running vm LRU
Comparing fifo.txt with lru.txt
1,2c1,d2
< Page Faults Rate= 0. 010500
< TLB Hits Rate= 0. 861100
---
> Page Faults Rate= 0. 009400
> TLB Hits Rate= 0. 872000
```

图 10 –不同策略对比

## 2. 页表实验 (Lab: Page Tables)

### 2.1 Print a Page Table 实验要求及步骤

Print a Page Table 实验要求写一个打印 Page Table 的函数，方便未来的调试 debug。在 kernel/vm.c 文件中定义一个 vmprint() 函数，接收一个 pagetable\_t 的参数，并且用给定格式打印页表。在 kernel/defs.h 中编写 vmprint() 的声明，在 exec.c 文件中的 return argc 之前插入对 vmprint() 的调用来打印第一个进程的页表，以实现启动 xv6 执行完 exec() 时打印第一个进程的页表信息。

### 2.2 代码实现

```
void prevmprint(pagetable_t pagetable, int level) // 编写的打印函数
{
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V)){
            // PTE指向低级的页表
            // 用打印...来表示树的深度
            for(int j = 0; j < level; j++){
                if(j == 0) printf("..");
                else printf(" ..");
            }
            uint64 child = PTE2PA(pte);
            // 打印page索引、PET地址、物理页地址
            printf("%d: pte %p pa %p\n", i, pte, child);
            // 查看flag位是否被设置，若被设置则为最低一层，最底层被设置了符号位
            if((pte & (PTE_R|PTE_W|PTE_X)) == 0)
                prevmprint((pagetable_t)child, level + 1);
        }
    }
}

// 定义vmprint函数
void vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    prevmprint(pagetable, 1); // 调用编写的打印函数
}
```

图 11 -vmprint 函数代码

```
void vmprint(pagetable_t); if(p->pid==1)
                        vmprint(p->pagetable);
```

图 12 -defs.h 及 exec.c 文件配置

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6f000
..0: pte 0x0000000021fdac01 pa 0x0000000087f6b000
.. ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. .. ..0: pte 0x0000000021fdb01f pa 0x0000000087f6c000
.. .. ..1: pte 0x0000000021fda40f pa 0x0000000087f69000
.. .. ..2: pte 0x0000000021fda01f pa 0x0000000087f68000
..255: pte 0x0000000021fdb801 pa 0x0000000087f6e000
.. ..511: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ |

```

图 13 -打印结果

此外，还可以通过修改 `printf.c` 文件完成任务，在这里我们添加了读写权限。代码如图 14。

```

static void flag_to_char(int flag, char *arr){
    int i = 0;
    if (flag & PTE_R){
        arr[i++] = 'R';
    }
    if (flag & PTE_W){
        arr[i++] = 'W';
    }
    if (flag & PTE_X){
        arr[i++] = 'X';
    }
    if (flag & PTE_U){
        arr[i] = 'U';
    }
}

static void traversal_pt(pagetable_t pagetable, int level){
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if(pte & PTE_V){
            vint64 child = PTE2PA(pte);
            char arr[4] = {'\0', '\0', '\0', '\0'};
            char* flag = flag_to_char(pte % 32, arr);
            if (level == 0){
                printf("..%d: pte %p (%s) pa %p\n", i, pte, arr, child);
                traversal_pt((pagetable_t)child, level: level + 1);
            }else if (level == 1){
                printf(".. ..%d: pte %p (%s) pa %p\n", i, pte, arr, child);
                traversal_pt((pagetable_t)child, level: level + 1);
            }else{
                printf(".. .. ..%d: pte %p (%s) pa %p\n", i, pte, arr, child);
            }
        }
    }
}

```

图 14 -printf.c 文件修改部分

## 2.3 问题详解

```
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
```

图 15 -打印结果中的问题

问题一：括号为空的都是不是叶子节点，即最高级的 page directory 的一条 PTE，不会指向实际的物理页，所以用户无权限，不需要类似的读写标志位。

32618 在物理内存的 0x0000000000007f6b 位置。

问题二：用户栈保护页，防止用户栈溢出。

问题三：代码页，存放代码生成的机器码，无 U 权限，此页不能被用户访问。（详见图）。

问题四：数据段页，用于存放数据段，权限均有。

问题五：用户栈页，在用户代码执行时存放栈信息（详见图），CPU 在不同进程之间切换执行，从此进程切换到其他进程时，需要将 CPU 寄存器信息保存下来，以便下次切换回此进程时，可以恢复执行。这个区域不是代码，只是存放数据，不需要执行标志位 X，可读可写但不可执行。

问题六：缓冲区 trampoline 页，存放 trampoline.S 信息，不发生写操作，不需要标志位 W。



```

// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    uint64 sz1;
    if((sz1 = uvmmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    sz = sz1;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}

```

图 16 -问题代码解释

```

// Allocate two pages at the next page boundary.
// Use the second as the user stack.
sz = PGROUNDUP(sz);
uint64 sz1;
if((sz1 = uvmmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
sz = sz1;
uvmclear(pagetable, sz-2*PGSIZE);
sp = sz;
stackbase = sp - PGSIZE;

```

图 17 -问题代码解释

### 3. 内存分配实验 (Lab: Xv6 Lazy Page Allocation)

#### 3.1 Eliminate Allocation from sbrk()子任务

为了更好地理解、完成惰性页表分配任务，本实验先进行 sbrk()实验，对 kernel/sysproc.c 文件的 sys\_sbrk()函数进行修改，只将进程的内存空间大小增加 n byte 而不进行实际的分配。修改代码，注释掉 growproc()函数，不直接分配物理内存和设置 pte，growproc()函数针对 n 的正负分别调用了 uvmmalloc 和 uvmdealloc 函数（该情况我们将在 3.3 中做进一步解释讨论和完善）。sys\_sbrk()

具体代码如下所示。

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

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    // if(growproc(n) < 0)
    //     return -1;
    myproc() ->sz += n;
    return addr;
}
```

图 18 -sys\_sbrk()函数修改

函数返回的地址是新分配的地址空间的开头，在此处为原地址空间的末尾。现增加了回收内存 myproc() -> sz，增加进程的大小并返回旧的大小，但实际却并未增加，因为在 sys\_sbrk 中并未 malloc 区域。（老版本 xv6 直接添加 proc->sz）。

make qemu 启动 xv6，运行 echo OSC，得到如图所示错误提示，只增加大小，不分配物理区域导致找不到虚拟地址对应的物理页引发报错。

```
init: starting sh
$ echo OSC
usertrap(): unexpected scause 0x000000000000000f pid=3
          sepc=0x000000000000012ac stval=0x00000000000004008
panic: uvmunmap: not mapped
```

图 19 -输出结果报错

### 3.2 Lazy Allocation 子任务

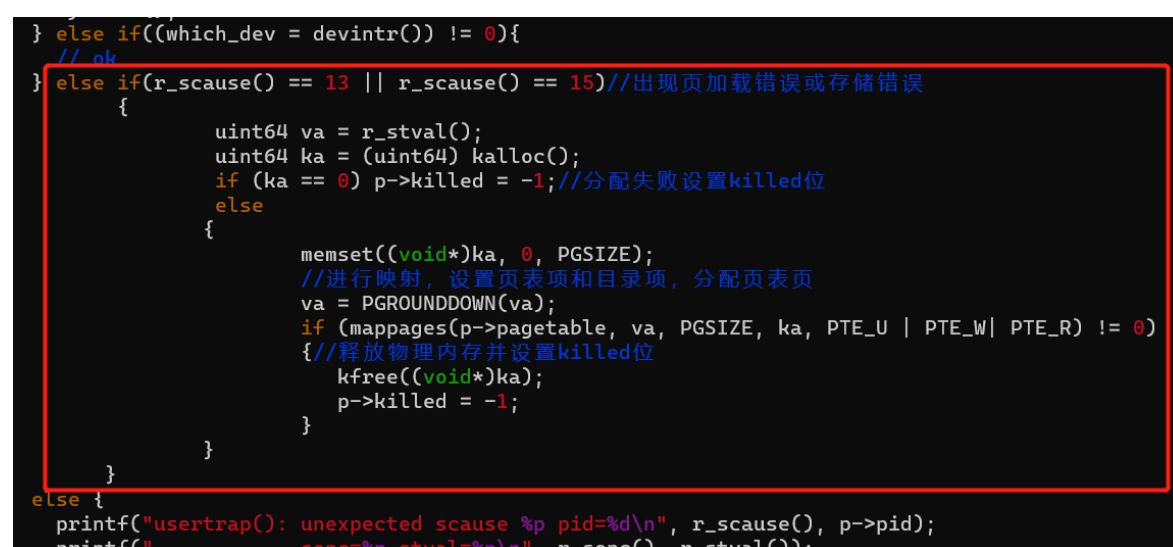
修改 trap.c 中的代码，使系统处理上述错误。trap.c 程序主要定义和实现了 asm.s 中所引用的各个硬件异常中断处理程序，子任务要求修改其中的代码，通过新分配一个物理页并映射到故障地址的方式来响应用户空间中的页故障，之后返回到用户空间来使进程继续执行。

在 trap.c 中，当发现是 page fault 错误的时候，可以按照当前进程的

proc->sz 来实际的分配内存，注意这个时候的 sz 的大小不是实际的大小，而是希望的大小值。

所以首先应该获取发生 page fault 时刻的虚地址，那个地址之后的部分就应该是本来应该分配但是实际没有分配的，而实际需要分配多少，应该根据 proc->sz 的大小来定。因为在发生 page fault 的时候的地址，是我们在 malloc 之后返回给进程的地址，而这个地址又是原来的 myproc()->sz，所以该虚地址的大小就应该是实际的内存的大小。

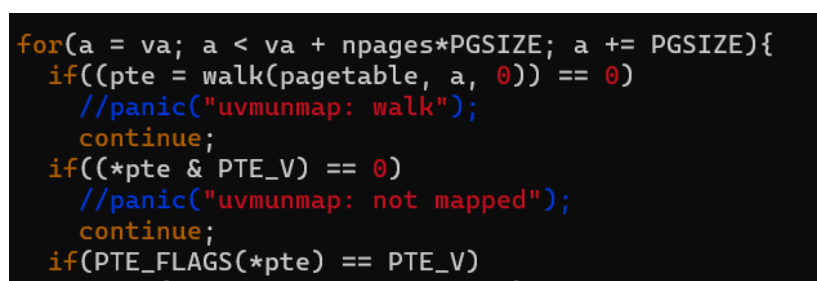
实验应当在 “pid 3 sh: trap 14”的 cprintf 调用之前（打印错误消息之前）添加代码，代码仿照 vm.c 中的 allocuvm 部分，申请内存给出现异常的虚拟地址。程序未涵盖所有极端情况和错误情况，仅满足 echo 响应的简单命令。



```
    } else if((which_dev = devintr()) != 0){
        // ok
    } else if(r_scause() == 13 || r_scause() == 15) //出现页加载错误或存储错误
    {
        uint64 va = r_stval();
        uint64 ka = (uint64) kalloc();
        if (ka == 0) p->killed = -1; //分配失败设置killed位
        else
        {
            memset((void*)ka, 0, PGSIZE);
            //进行映射，设置页表项和目录项，分配页表页
            va = PGROUNDDOWN(va);
            if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W | PTE_R) != 0)
            { //释放物理内存并设置killed位
                kfree((void*)ka);
                p->killed = -1;
            }
        }
    }
    else {
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
        printf("      sence=%p stval=%p\n", r_sense(), r_stval());
    }
}
```

图 20 -trap.c 文件 usertrap 函数修改（红框部分为插入代码部分）

此外，添加完之后仍然会报错 uvmunmap: not mapped，因为这里实际上找不到相应的物理地址，故需要 continue 忽略该部分，所以在 vm.c 文件的 uvmunmap 函数中注释掉 panic 部分。



```
for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
        //panic("uvmunmap: walk");
        continue;
    if((*pte & PTE_V) == 0)
        //panic("uvmunmap: not mapped");
        continue;
    if(PTE_FLAGS(*pte) == PTE_V)
        panic("uvmunmap: not a leaf");
}
```

图 21 -vm.c 文件 uvmunmap 函数修改

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo OSC
OSC
$ |
```

图 22 -实验 echo 输出

### 3.3 Lazytests and Usertests 子任务

根据指导书的参考思路进行修改。首先在 kernel\sysproc.c 文件中 sys\_sbrk 函数添加处理参数 n 为负数的情况，uvmdealloc 相应的内存。代码如图 23 所示。

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;
    struct proc *p = myproc();

    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    if(n < 0){//当n为负数
        if(p->sz + n < 0) return -1;
        //dealloc相应的内存n
        else uvmdealloc(p->pagetable, p->sz, p->sz + n);
    }
    p->sz += n;
    //if(growproc(n) < 0)
    //return -1;
    return addr;
}
```

图 23 -sys\_sbrk 函数修改：添加 n<0 情况

题目要求使用的地址空间不能超过 sbrk 分配的，sbrk 分配内存之后会增加 p->sz，即要求 faulting addr 不超过 p->sz。在 kernel\trap.c 文件的 usertrap 函数中，设置缺页异常情况终止进程，主要情况有读入虚拟地址比 p->sz 大、虚拟地址小于进程的用户栈、申请分配空间不够等。具体代码详见图 24。

```

} else if((which_dev != devintr()) != 0){
    // ok
} else if(r_scause() == 13 || r_scause() == 15)//出现页加载错误或存储错误
{
    uint64 va = r_stval();
    if (va < p->sz && va > PGROUNDDOWN(p->trapframe->sp)){
        //当读入虚拟地址比p->sz大或虚拟地址比进程的用户栈小
        //printf("usertrap(): fault address %p beyond heap range\n", r_stval);
        uint64 ka = (uint64) kalloc();
        if (ka == 0) {
            p->killed = -1;//分配失败设置killed位
            printf("usertrap(): fault address %p beyond heap range\n", r_stval);
        }
        else
        {
            memset((void*)ka, 0, PGSIZE);
            //进行映射, 设置页表项和目录项, 分配页表页
            va = PGROUNDDOWN(va);
            if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W | PTE_R) != 0)
            { //释放物理内存并设置killed位
                kfree((void*)ka);
                p->killed = -1;
            }
        }
    }
    else {
        p->killed = -1;
        printf("usertrap(): fault address %p beyond heap range\n", r_stval);
    }
}

else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
}

```

图 24 -usertrap 函数修改: 添加缺页情况处理

为正确处理 fork, 对 kernel/vm.c 文件 uvmcopy 函数进行修改。将父进程的内存复制到子进程, 若判断 invalid 则导致 panic, lazy allocation 不能保证此时父进程的内存均已分配, 子进程复制父进程地址空间的时候, 发现地址空间不存在时需要忽略, 继续下一页。代码如图 25 所示。

```

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){
        if((pte = walk(old, i, 0)) == 0)
            //发现地址空间不存在时需要忽略
            //panic("uvmcopy: pte should exist");
            continue;
        if((*pte & PTE_V) == 0)
            //panic("uvmcopy: page not present");
            continue;
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);

```

图 25 -usertrap 函数修改: 添加缺页情况处理

题目中要求处理以下情况：进程将有效地址从 `sbrk()` 传递给系统计算，如 `read` 或 `write`，但该地址的内存尚未分配。该情况处理进程内存的不是用户进程，而是 `read` 或 `write` 等系统调用，此时处理异常的不再是 `usertrap`。

当系统调用时，得到地址如果找不到相应的物理地址时，需要添加相应物理地址映射到 `page table` 里，这里关键就是要修改系统调用时虚拟地址转换为物理地址的代码。

内核读写用户内存，是通过 `kernel\vm.c` 中的 `copyin` 或 `copyout` 函数，主要使用 `walkaddr()` 函数找到用户内存虚拟地址对应的物理地址，然后用 `memmove` 直接通过物理地址 (`va=pa`) 传送。

如图所示，代码逻辑为：`pte` 如果为 0（未分配）或 `invalid`，引发缺页，完成同 `usertrap` 一致的工作，这里直接套用 `trap.c` 的代码；如果 `pte` 有效，那么判断该页是否能被用户访问；如果都通过则返回物理地址。

最终通过 `Lazytests` 和 `Ustests`，简要运行结果如图 27、28 所示，实验完整结果请见附录。

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

    if(va >= MAXVA)
        return 0;

    pte = walk(pagetable, va, 0);
    //if(pte == 0)
    //    return 0;
    //if((*pte & PTE_V) == 0)
    //    return 0;
    if(pte == 0 || (*pte & PTE_V) == 0)
    {
        if (va >= p->sz || va < PGROUNDUP(p->trapframe->sp))
            return 0;
        uint64 ka = (uint64)kalloc();
        if (ka == 0)
        {
            return 0;
        }
        // 因为考虑到系统调用，需要和之前额外添加PTE_X
        if (mappages(p->pagetable, PGROUNDUP(va), PGSIZE, ka, PTE_W|PTE_X|PTE_R|PTE_U) != 0)
        {
            kfree((void*)ka);
            return 0;
        }
        return ka;
    }
}

if((*pte & PTE_U) == 0)
```

图 26 -walkaddr 函数修改：回归物理地址

```
$ 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

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
```

图 27、28 -lazytests 和 usertests 运行结果

## 附录一：usertests 完整运行结果

```
$ 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: usertrap(): unexpected scause 0x000000000000000c pid=9839
                sepc=0x00000000000005580 stval=0x00000000000005580
usertrap(): unexpected scause 0x000000000000000c pid=9840
                sepc=0x00000000000005580 stval=0x00000000000005580
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
$ |
```



## 附录二： lazytests 完整运行结果

[illegible]