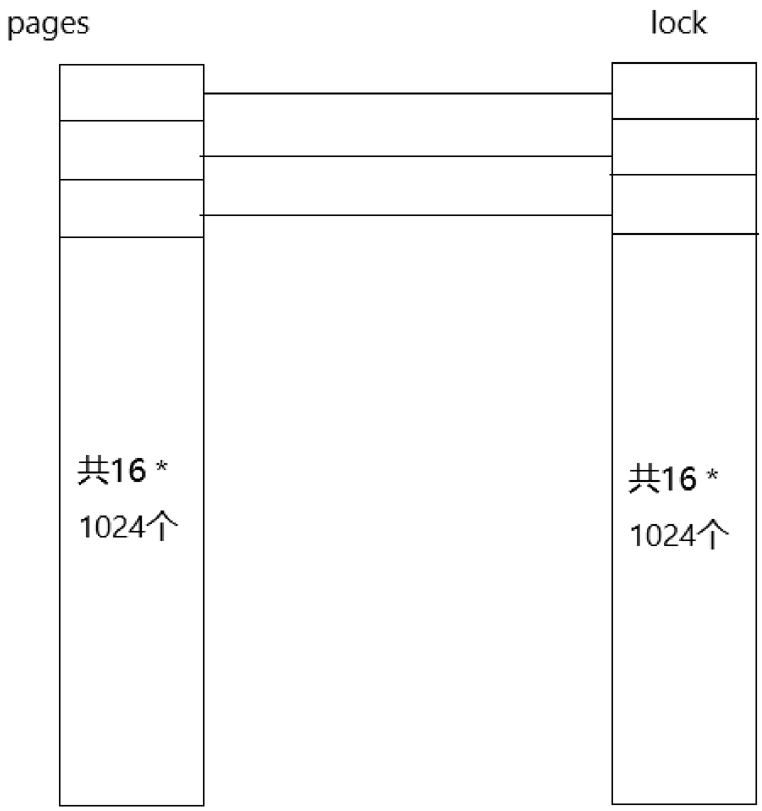
lab2-challenge 内存管理实验报告

part1

简要思路



通过相同的索引值建立数组lock和结构体数组pages的一一对应关系，通过4个函数控制对应lock的值从而实现对页面的上锁与解锁。（详细介绍请见下文）

增加的数据结构

- `int lock[16 * 1024]`，是一个起到锁作用的数组，1代表上锁，0代表未上锁
- `int futureLock`，是一个标记是否需要锁住未来页面的标志位，1代表之后分配的页面都需要上锁，0代表不需要

实现的函数

- mlock()

```
int mlock(u_long addr,size_t len) {
    //判断长度是否是BY2PG的整数倍，不是的话返回-1
    if (len % BY2PG != 0 ){
        printf("the len is not a multiple of BY2PG in mlock()");
        return -1;
    }
    u_long va = ROUND(addr,BY2PG);// 虚拟地址对齐
    int i;
```

```
    // 对从addr开始的len/BY2PG个页面上锁
    for (i = 0;i < len / BY2PG;i++){
        //找到虚拟地址对应的物理地址
        u_long pa = va2pa(boot_pgdir,va + i * BY2PG);
        //通过物理地址找到对应的page结构体，若其pp_ref大于1，说明页面已经被映射超过1次（共享），不可以上锁
        if (pa2page(pa)->pp_ref > 1) {
            printf("this page whose address is %08x has been refered one more time\n",va + i * BY2PG);
            continue;
        }
        lock[pa / BY2PG] = 1;// 上锁，索引值和page结构体对应
    }
    return 0;
}
```

- munlock()

```
//实现思路和mlock基本一致，逆过程
int munlock(u_long addr,size_t len) {
    if (len % BY2PG != 0 ){
        printf("the len is not a multiple of BY2PG in mlock()");
        return -1;
    }
    u_long va = ROUND(addr,BY2PG);
    int i;
    for (i = 0;i < len/ BY2PG;i++){
        u_long pa = va2pa(boot_pgdir,va + i * BY2PG);
        lock[pa / BY2PG] = 0;
    }
    return 0;
}
```

- mlockall()

```
int mlockall(int flags) {
    if (flags == MCL_CURRENT){ //对当前所有被映射的页面上锁
        int i;
        for (i = 0;i < npage;i++) {
            if (pages[i].pp_ref == 1 && lock[i] == 0) {
                lock[i] = 1;
            }
        }
    }
    if (flags == MCL_FUTURE){//对之后被映射的页面上锁
        futureLock = 1;
    }
    return 0;
}
```

- munlokcall()

```c
int munlockall(void) {
    int i;
    for (i = 0;i < 16 * 1024;i++) { //去掉所有的锁
        lock[i] = 0;
    }
    futureLock = 0;            //将未来锁置0
    return 0;
}
```

修改的函数

- page_init

```c
int i;
 // 进程初始化(mips_init)后的已经被映射的物理页面默认被锁定C
for (i=0;i<PADDR(freemem)/BY2PG;i++){
    pages[i].pp_ref = 1;
    lock[i] = 1;
}
```

- page_free()

```c
//添加位置位于函数开头，若判断被释放的页面是否已经上锁，若已经上锁，则不能被释放，直接返回
if (lock[pp - pages] == 1) {
    printf("lock[%d] = 1.this page has been locked.\n",pp-pages);
    return;
}
```

- page_insert()

```c
//添加位置位于函数开头，判断插入的页面是否已经被上锁，若已经上锁，则说明已经被插入一次，不能被共享
if (lock[pp - pages] == 1) {
    printf("this page has been locked. It can't insert.\n");
    return -1;
}
```

```c
//添加位置位于函数返回之前，此时页面已被插入，判断未来锁标志位是否为1，若是1则将页面上锁
if (futureLock == 1) {
    lock[va2pa(pgdir,va) / BY2PG] = 1;
}
```

- page_remove()

```c
 //添加位置位于函数开头，若判断被移除的页面是否已经上锁，若已经上锁，则不能被释放，直接返回
if (lock[va2pa(pgdir,va) / BY2PG]==1) {
    printf("this page has been locked. It can't be removed\n");
    return;
}
```

测试函数

```
void lock_check(){

    printf("lock check starts\n");

    struct Page *p1,*p2,*p3,*p0;
    p0 = p1 = p2 = p3 = 0;

    //分配三个页面
    assert(page_alloc(&p0) == 0);
    assert(page_alloc(&p1) == 0);
    assert(page_alloc(&p2) == 0);

    assert(p0);
    assert(p1 && p1 != p0);
    assert(p2 && p2 != p1 && p2 != p0);


    // free pp0 and try again: pp0 should be used for page table
    page_free(p0);
    assert(page_insert(boot_pgdir, p1, 0x0, 0) == 0);
    assert(PTE_ADDR(boot_pgdir[0]) == page2pa(p0));

    assert(va2pa(boot_pgdir, 0x0) == page2pa(p1));
    assert(p1->pp_ref == 1);


    assert(page_insert(boot_pgdir,p1,BY2PG*5,0) == 0);// 共享p1
    assert(va2pa(boot_pgdir,BY2PG *5) == page2pa(p1));
    assert(p1->pp_ref == 2);
    mlock(0x0,BY2PG);                               // 尝试对p1上锁（已经被共享，不会上锁）
    // 输出“this page whose address is %08x has been refered one more time”
    page_remove(boot_pgdir,BY2PG*5);                // 取消映射（解除共享）
    mlock(0x0,BY2PG);                              // 对p1上锁
    page_free(p1);                                 // 尝试释放p1
    // 输出“lock[%d] = 1.this page has been locked.page_free()”
    page_remove(boot_pgdir,0x0);                    //尝试移除p1
    //输出“this page has been locked. It can't be removed”
    assert(p1 != LIST_FIRST(&page_free_list));       //移除失败，p1不等于空闲链表的第一个
    assert(va2pa(boot_pgdir,0x0) == page2pa(p1));
    printf("lock p1 successfully\n");

    assert(page_insert(boot_pgdir,p1,BY2PG * 2,0) <0);// 尝试共享p1
    // 输出“this page has been locked. It can't insert.”
    assert(page_insert(boot_pgdir,p2,BY2PG,0) == 0);  // 映射p2
    mlock(BY2PG,BY2PG); // lock p2                  // 对p2上锁
    page_free(p2);                                 //尝试释放p2
    //输出“lock[%d] = 1.this page has been locked.”
    page_remove(boot_pgdir,BY2PG);                  //尝试移除
    //输出“this page has been locked. It can't be removed”
    assert(va2pa(boot_pgdir,BY2PG) == page2pa(p2));
    printf("lock p2 successfully\n");

    munlock(0,BY2PG * 2);                           // 对p1和p2解锁
    page_remove(boot_pgdir,0);                       //解除p1映射
```

```c
    assert(p1 == LIST_FIRST(&page_free_list));        // p1等于空闲链表的第一个元素
    page_remove(boot_pgdir,BY2PG);                   //解除p2映射
    assert(p2 == LIST_FIRST(&page_free_list));        // p2等于空闲链表的第一个元素
    assert(va2pa(boot_pgdir,0x0) != page2pa(p1));
    assert(va2pa(boot_pgdir,BY2PG) != page2pa(p2));
    printf("unlock test successfully\n");

    p0 = p1 = p2 = p3 = 0;
    assert(page_alloc(&p0) == 0);
    assert(page_alloc(&p1) == 0);
    assert(page_alloc(&p2) == 0);
    assert(page_alloc(&p3) == 0);
    struct Page *p4,*p5;
    assert(page_alloc(&p4) == 0);
    assert(page_alloc(&p5) == 0);
    assert(p0);
    assert(p1 && p1 != p0);
    assert(p2 && p2 != p1 && p2 != p0);
    assert(p3 && p3 != p1 && p3 != p2 && p3 != p0);

    page_free(p0);

    assert(page_insert(boot_pgdir,p1,BY2PG,0) == 0);    // 映射p1
    assert(page_insert(boot_pgdir,p2,BY2PG * 3,0) == 0);// 映射p2
    mlockall(MCL_CURRENT);                          // 对现存的所有页面上锁
    page_remove(boot_pgdir,BY2PG);                  // 尝试移除p1
    // 输出“this page has been locked. It can't be removed”
    page_free(p1);                                  // 尝试释放p1
     // 输出“lock[%d] = 1.this page has been locked.”
    page_remove(boot_pgdir,BY2PG * 3);              // 尝试移除p2
    // 输出“this page has been locked. It can't be removed”
    page_free(p2);                                  // 尝试释放p2
    // 输出“lock[%d] = 1.this page has been locked.page_free()”
    mlockall(MCL_FUTURE);                           // 对之后映射的页面上锁
    assert(page_insert(boot_pgdir,p3,BY2PG * 5,0) == 0); // 映射p3
    page_remove(boot_pgdir,BY2PG * 5);              // 尝试移除p3
    // 输出“this page has been locked. It can't be removed”
    page_free(p3);                                  // 尝试释放
    // 输出“lock[%d] = 1.this page has been locked.page_free()”

    assert(va2pa(boot_pgdir,BY2PG) == page2pa(p1));
    assert(va2pa(boot_pgdir,BY2PG * 3) == page2pa(p2));
    assert(va2pa(boot_pgdir,BY2PG * 5) == page2pa(p3));
    printf("mlockall test successfully\n");

    munlockall();                                   // 对所有页面解锁
    page_remove(boot_pgdir,BY2PG);                  // 正常移除3个页面
    assert(p1 == LIST_FIRST(&page_free_list));
    page_remove(boot_pgdir,BY2PG*3);
    assert(p2 == LIST_FIRST(&page_free_list));
    page_remove(boot_pgdir,BY2PG*5);
    assert(p3 == LIST_FIRST(&page_free_list));

    assert(page_insert(boot_pgdir,p4,BY2PG*7,0) == 0);  // 映射
    page_remove(boot_pgdir,BY2PG*7);                  // 正常移除
    assert(p4 == LIST_FIRST(&page_free_list));
```

```

    printf("lock check successfully\n");

}
```

测试结果

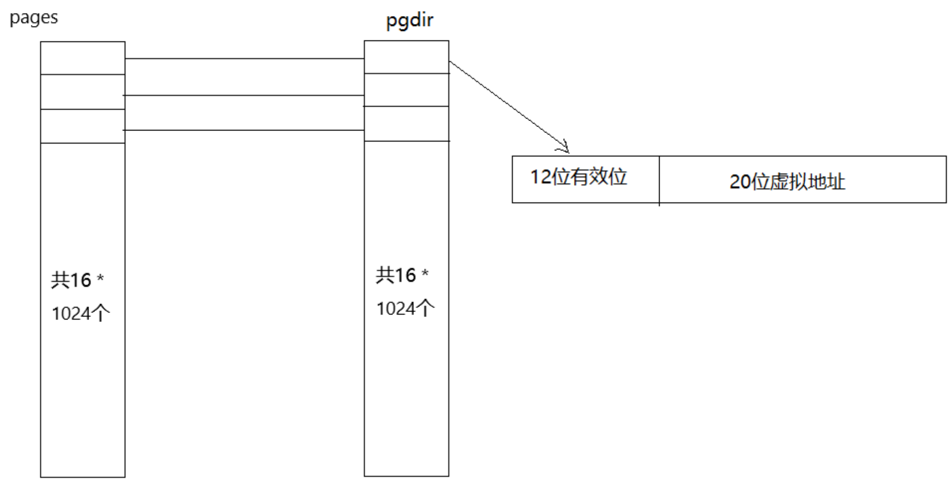
```
jovyan@1b42c1ef811a:~/18373489-lab$ bash test.sh
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.128.254
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
-------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

lock check starts
```

```

pmap.c:  mips vm init success

lock check starts

this page whose address is 00000000 has been refered one more time

lock[16318] = 1.this page has been locked.

this page has been locked. It can't be removed

lock p1 successfully

this page has been locked. It can't insert.

lock[16317] = 1.this page has been locked.

this page has been locked. It can't be removed

lock p2 successfully

unlock test successfully

this page has been locked. It can't be removed

lock[16318] = 1.this page has been locked.

this page has been locked. It can't be removed

lock[16316] = 1.this page has been locked.

this page has been locked. It can't be removed

lock[16315] = 1.this page has been locked.

mlockall test successfully

lock check successfully

panic at init.c:30: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

GXemul> quit
jovyan@1b42c1ef811a:~/18373489-lab$
```

part2

在Rpmap.c文件中

简要思路



页表pgdir共16 * 1024个页表项，通过相同的索引值与pages一一对应，通过相对应的函数实现对页面的分配、查询、移除和释放等功能。

增加的数据结构及函数

- hashcode()

```
// 31 * 31 * 31 = 29791 和取其他位数相比离PAGESUM较近，且个位0-9的立方对应个位0-9
int hashcode(int addr)
{
    addr = addr >> 12;
    addr = addr & 0x1f;                 // 取虚拟地址的13-17位
    int res = addr * addr * addr;        // 立方作为哈希值
    while (res > PAGESUM) {
        res -= PAGESUM;
    }
    return res;
}
```

修改的函数及数据结构

- 将`boot_pgdir`作为反置页表
- Rboot_map_segment()

```
// 为一段虚拟地址分配空间
void Rboot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
{
    int i;
    Pte *pgtable_entry;
    u_long va_temp = va, pa_temp = pa / BY2PG;
    u_long num = size / BY2PG;virtual address `va`. */
    for (i=0;i<num;i++){
        int index = hashcode(va_temp);          // 计算哈希值
        int count = 0;                          // 计数器，下同
```

```
        while(((*(pgdir + index)) & VALID) != 0) {// 寻找无效的页表项
            index++;
            index %= PAGESUM;
            count++;
            // 计数器大于PAGESUM，说明已经将反置页表都找了一遍，直接退出
            if (count > PAGESUM) {
                panic("no more space\n");
            }
        }
        // 修改页表项，低20位为虚拟地址高20位，高12位为有效位VALID(0xfff)
        *(pgdir + index) = (va_temp >> 12) | VALID;
        va_temp+=BY2PG;
        pa_temp++;
    }
}
```

- Rpage_alloc()

```
int Rpage_alloc(Pde* pgdir,u_long va)
{
    int index = hashcode(va);
    int count = 0;
    // 查找可分配的页面，若发生哈希冲突则继续向下寻找
    while((*(pgdir + index) & VALID) != 0) {
        // 如果找到一个页面，其对应的虚拟地址和va相同，说明这个虚拟地址也存在映射关系，返回-1
        if(((*(pgdir + index)) == ((va >> 12) | VALID ))) {
            printf("the address has been mapped\n");
            return -1;
        }
        index++;
        if (index > PAGESUM)
            index -= PAGESUM;
        count++;
        if (count > PAGESUM) {
            return -E_NO_MEM;
        }
    }
    bzero((void *)page2kva(&pages[index]),BY2PG);// 置0
    *(pgdir + index) = (va >> 12) | VALID;      // 设置页表项
    return index;                               // 分配成功则返回索引值
}
```

- Rpage_free()

```
// 找到对应的页面，并将页表项清除
void Rpage_free(Pde* pgdir,u_long va)
{
    int count = 0;
    int index = hashcode(va);
    while(!((*(pgdir + index) & VALID) != 0 && ((*(pgdir + index)) & 0x000fffff) == (va>>12))) {
        index++;
        index %= PAGESUM;
        count++;
        if (count > PAGESUM) {
```

```
            printf("Can't find the physic page\n");
            return;
        }
    }
    *(pgdir + index) = 0;
}
```

- Rpage_lookup()

```
// 找到对应的页面，如果成功则返回索引值
int Rpage_lookup(Pde *pgdir, u_long va)
{
    int count = 0;
    int index = hashcode(va);
    while(!((*(pgdir + index)) == ((va >> 12) | VALID )))
    {
        index++;
        count++;
        index %= PAGESUM;
        if (count > PAGESUM) {
            printf("this physic page does not exist\n");
            return -1;
        }
    }
    return index;
}
```

测试函数

```
void invertPage_check()
{
    printf("invertPage_check() starts\n");
    // 在Rmips_vm_init(和mips_vm_init函数体相同)中已经映射过了
    printf("boot_pgdir:%08x\n",*(boot_pgdir+hashcode(UPAGES)) & 0x000fffff);
    printf("UPAGES:%08x\n",UPAGES>>12);
    assert((*(boot_pgdir+hashcode(UPAGES))& 0x000fffff) == (UPAGES>>12));

    int index = Rpage_alloc(boot_pgdir,BY2PG * 2);      // 分配页面
    printf("index:%d\n",index);                        // 输出索引值
    assert(index >= 0);
    // 对比页表项
    assert((*(boot_pgdir + index) & 0x000fffff) == (BY2PG * 2) >> 12);

    assert(Rpage_alloc(boot_pgdir,BY2PG * 2) < 0) ;      // 相同虚拟地址再次分配，失败返回-1

    assert(Rpage_lookup(boot_pgdir,BY2PG * 3)<0 );       // 未映射相应虚拟地址查找失败，返回-1
    assert(Rpage_lookup(boot_pgdir,BY2PG *2) == index);  // 查找返回值等于index

    Rpage_free(boot_pgdir,BY2PG*2);                      // 解除映射，释放页面
    assert((*(boot_pgdir + index)) != (BY2PG * 2) >> 12); //页表项应该置0
    assert((*(boot_pgdir + index)) == 0);

    int i = 0;
```

```c
    int temp[11];
    for (i =0;i < 10;i++) {                          //分配10个页面，输出索引值
        assert((temp[i]=Rpage_alloc(boot_pgdir,BY2PG * (i + 3))) >= 0);
        printf("temp[%d]:%d\n",i,temp[i]);
    }

    for (i = 0;i < 10;i++) {                         // 查找上述10个页面，输出查找索引值
    printf("Rpage_lookup[%d]:%d\n",i,Rpage_lookup(boot_pgdir,BY2PG * (i+3)));
        assert(Rpage_lookup(boot_pgdir,BY2PG*(i + 3)) == temp[i]);
    }

    for (i = 0;i < 10;i++) {                         // 将上述10个页面释放，并再次查找，输出结果都应该为-1
        Rpage_free(boot_pgdir,BY2PG *(i+3));
        assert(Rpage_lookup(boot_pgdir,BY2PG*(i+3))<0);
    }
    printf("invertPage_check() succeeded\n");
}
```

测试结果

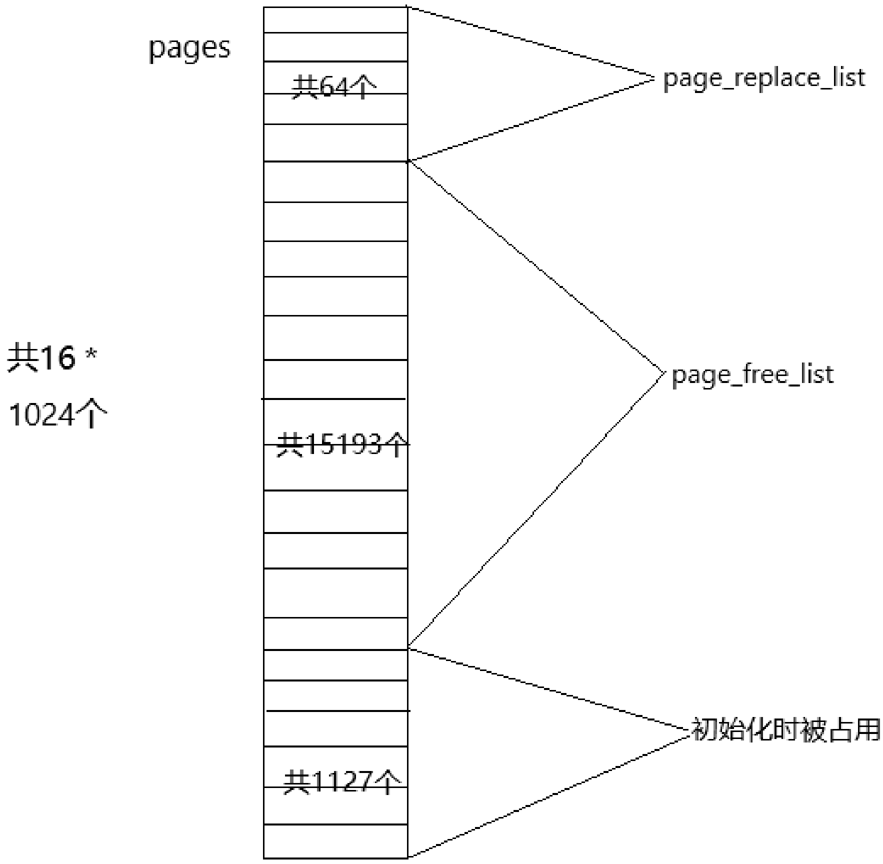
```
jovyan@1b42c1ef811a:~/18373489-lab$ bash test.sh
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.128.254
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
-------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80404000 for struct page directory.

to memory 80434000 for struct Pages.

pmap.c:  mips vm init success

invertPage_check() starts

boot_pgdir:0007f800

UPAGES:0007f800

index:9

the address has been mapped

this physic page does not exist

temp[0]:28

temp[1]:65
```

```
temp[2]:126
temp[3]:217
temp[4]:344
temp[5]:513
temp[6]:730
temp[7]:1001
temp[8]:1332
temp[9]:1729
Rpage_lookup[0]:28
Rpage_lookup[1]:65
Rpage_lookup[2]:126
Rpage_lookup[3]:217
Rpage_lookup[4]:344
Rpage_lookup[5]:513
Rpage_lookup[6]:730
Rpage_lookup[7]:1001
Rpage_lookup[8]:1332
Rpage_lookup[9]:1729
this physic page does not exist
this physic page does not exist
this physic page does not exist
```

```
Rpage_lookup[2]:126
Rpage_lookup[3]:217
Rpage_lookup[4]:344
Rpage_lookup[5]:513
Rpage_lookup[6]:730
Rpage_lookup[7]:1001
Rpage_lookup[8]:1332
Rpage_lookup[9]:1729
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
this physic page does not exist
invertPage_check() succeeded
panic at init.c:30: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
GXemul> quit
jovyan@1b42c1ef811a:~/18373489-lab$
```

part3

简要思路



从pages结构体数组中分出64个页面作为置换页面，初始化时放在page_replace_list链表中，使用后放入page_used_list中，通过相应的函数实现页面分配、置换、移除、释放以及页表项、页目录项、页面内容的恢复。当page_free_list链表为空时，根据FIFO算法找到一个置换页，然后在page_replace_list链表中找到一个空白页，将置换页中的内容复制到空白页中并记录相应的虚拟地址、页目录项和页表项，将置换页内容清零加入到page_free_list链表中，得到一个空白页，完成模拟页面置换操作

增加的数据结构

- `static struct Page_list page_replace_list`，作为未使用的置换页的表头
- `static struct Page_list page_used_list`，作为已被使用的置换页的表头
- `int replaceIndex`，置换页面索引值
- `u_long page2va[16 * 1024]`，记录页面映射的虚拟地址
- `Pde re2pde[REPLACENUM]`，记录被置换页的页目录项
- `Pte re2pte[REPLACENUM]`，记录被置换页的页表项
- `u_long re2va[REPLACENUM]`，记录被置换页原来所对应的虚拟地址

增加的函数

- replace2recover()

```
// 根据虚拟地址在被置换页中找到对应的页面，恢复内容、页表项和页目录项
int replace2recover(Pde* pgdir,u_long va,struct Page *page)
{
    struct Page *pp;
    struct Page *temp;
    Pde *pgdir_entry_temp;
    Pte *pagetable_entry_temp;
    // 已被使用的置换页的链表为空，说明没有页面被置换，返回-1
    if (LIST_FIRST(&page_used_list) == NULL)
        return -1;
```

```
    // 对被使用的置换页的链表进行遍历，根据虚拟地址找到对应的被置换页
    for (temp = LIST_FIRST(&page_used_list);temp != NULL;temp = LIST_NEXT(temp,pp_link)) {
        //printf("temp:%d  va:0x%08x\n",temp - pages,page2va[temp - pages]);
        if (re2va[temp-pages] == va) {
            pgdir_entry_temp = re2pde[temp - pages];
            pagetable_entry_temp = re2pte[temp - pages];
            break;
        }
    }
    Pde* pgdirEntry = boot_pgdir + PDX(va);
    Pte* pgtableEntry = KADDR(PTE_ADDR(*pgdirEntry)) + PTX(va);
            C
    pgtableEntry = re2pte[&pages[replaceIndex] - pages] ;// 恢复页表项
    pgdirEntry = re2pde[&pages[replaceIndex] - pages];  // 恢复页目录项

    pp = LIST_FIRST(&page_replace_list);                // 恢复内容
    bcopy(page2kva(temp),page2kva(pp),BY2PG);
    bcopy(page2kva(page),page2kva(temp),BY2PG);
    bcopy(page2kva(pp),page2kva(page),BY2PG);
    bzero(page2kva(pp),BY2PG);
    return 0;
}
```

修改的函数

- page_init()

```
//将原函数的页面初始化方式修改如下
int i;
    for (i=0;i<PADDR(freemem)/BY2PG;i++){
    pages[i].pp_ref = 1;
    lock[i] = 1;
    }
    //printf("PADDR(freeemem) / BY2PG = %d\n",PADDR(freemem)/BY2PG);//result = 1127
// 64 个 pages 作为置换页面 REPLACENUM是一个宏，值为64
   for (;i<npage - REPLACENUM;i++){
        pages[i].pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list,&pages[i],pp_link);
   }
   // 将64个置换页面加入page_replace_list中
   for (;i<npage;i++) {
        pages[i].pp_ref = 0;
        LIST_INSERT_HEAD(&page_replace_list,&pages[i],pp_link);
   }
    replaceIndex = npage - REPLACENUM - 1;// 设置置换页索引值
```

- page_alloc()

```
// 将原函数的if (LIST_FIRST(&page_free_list) == NULL){函数体} 修改如下
// page_free_list中没有空闲的页面，可能需要进行页面置换
if (LIST_FIRST(&page_free_list) == NULL) {
        // 若没有可用于置换的页，说明空间已满
        if (LIST_FIRST(&page_replace_list) == NULL) {
            printf("no more physic memory\n");
```

```c
            return -E_NO_MEM;
        }

        replaceIndex--; // 索引值自减1
        int count = 0;  // 同理计数器
        // 找到未上锁且被映射的页面
        // 采用FIFO的页面置换算法（从pages索引值大方向开始分配，所以从相应位置开始寻找置换页）
        while(!(lock[replaceIndex]==0 && pages[replaceIndex].pp_ref >0)) {

            replaceIndex--;
            if (replaceIndex < 0)
                replaceIndex = npage - REPLACENUM - 1;
            count++;
            if (count > npage) {
                printf("no page\n");
                return -E_NO_MEM;
            }

        }
        // 输出提示，找到对应的置换页，开始置换
        printf("the physic memory is full. Page replacement starts\n");
        repage = LIST_FIRST(&page_replace_list);// 取出一个置换页
        LIST_REMOVE(repage,pp_link);             //移除
        LIST_INSERT_HEAD(&page_used_list,repage,pp_link);//插入page_used_list中
        u_long va = page2va[&pages[replaceIndex] - pages]; // 找到映射的虚拟地址
        Pde* pgdirEntry = boot_pgdir + PDX(va);
        Pte* pgtableEntry = KADDR(PTE_ADDR(*pgdirEntry)) + PTX(va);

        re2pte[&pages[replaceIndex] - pages] = pgtableEntry;// 记录页表项
        re2pde[&pages[replaceIndex] - pages] = pgdirEntry;//记录页目录项

        re2va[repage - pages] = va;                     // 记录映射的虚拟地址

        bcopy(page2kva(&pages[replaceIndex]),page2kva(repage),BY2PG);
        // 输出提示
        printf("replaced page's index is %d,mapped va is 0x%08x\nx",replaceIndex,va);

        page_remove(boot_pgdir,va); // 将被置换的页面移除映射，重新加入空闲链表中
    }
```

- page_insert()

```c
    page2va[pp-pages] = va; // 添加，记录页面映射的虚拟地址
```

测试函数

- pageReplacement_check1()C

```c
// 测试置换页面的数量
void pageReplacement_check1()
{
    printf("pageReplacement_check1() starts\n");
```

```
    struct Page *pp[16 * 1024],*p0;
    int i;
    // 共16 * 1024 = 16384个页面，在page_init中已分配出去1127个页面，64个页面用于页面置换，剩余15193个可用页面。19193中留出一个页面作为页表
    for (i = 0;i < 15192;i++) { // 分配15192个页面
        // leave a page for page_table in page_insert();
        assert(page_alloc(&pp[i]) == 0);
    }

    for (i = 0;i <= 63;i++) // 映射64个页面
        assert(page_insert(boot_pgdir,pp[i],BY2PG*i,0) == 0);

    // 除去64个置换页面，15193个可用页面已经全部分配
    for (i = 15194;i < 15194 + 64;i++) {// 再次分配64个页面，全部需要页面置换，64组输出提示
        assert(page_alloc(&pp[i]) == 0);
        assert(page_insert(boot_pgdir,pp[i],BY2PG * (i - 15194 + 67),0) == 0);
    }C

    assert(page_alloc(&p0) < 0);// 无可用页面，分配失败
    printf("pageReplacement_check1() succeeded\n");
}

```

- pageReplacement_check2()

```
void pageReplacement_check2()
{
    printf("pageReplacement_check2() starts\n");
    struct Page *pp[16 * 1024], *p0,*p1;
    int i,j;
    for (i = 0;i < 15192;i++) {
        assert(page_alloc(&pp[i]) == 0); //分配15192个页面
    }
    for (i = 0; i <= 64 ;i++) {          // 映射最初分配的64个页面
        assert(page_insert(boot_pgdir,pp[i],BY2PG * (i + 1), 0) ==0);
    }

    // test lock
    mlock(BY2PG,BY2PG);                  // 将第一个页面上锁
    mlock(BY2PG * 3, BY2PG);              // 将第三个页面上锁
    assert(page_alloc(&p0) == 0);        // 会置换第二个页面，对应虚拟地址BY2PG * 2

    munlock(BY2PG * 3,BY2PG);            // 解锁第三个页面
    assert(page_alloc(&p0) == 0);        // 会置换第三个页面

    // test replace
    int* temp = (int*)page2kva(&pages[16316]);
    *temp = 100;                         // 在第四个页面上写入100

    assert(page_alloc(&p0) == 0);        // 第四个页面被置换，其中的写入的值变为0
    assert(*temp == 0);
    //page_used_list 对应的第一个页面记录被替换页的内容，值为100
    assert(*(int*)page2kva(LIST_FIRST(&page_used_list)) == 100);

```

```c
    mlock(BY2PG * 5,BY2PG);                   // 对第五个页面上锁
    temp = (int *)page2kva(&pages[16315]);
    *temp = 1000;                             // 在第五个页面上写入1000
    assert(page_alloc(&p0) ==0);      // 第五个页面不会被置换，其值不变，第六个页面被置换
    assert(*temp == 1000);

    temp = (int*)page2kva(&pages[16313]);
    *temp = 12345;                            // 在第七个页面写入12345
    assert(page_alloc(&p0) == 0);             // 第七个页面被置换
    assert(page_insert(boot_pgdir,p0,BY2PG * 66 ,0) == 0);   // 映射虚拟地址
    assert(*temp == 0);                       // 其值变为0

    assert(replace2recover(boot_pgdir,BY2PG*7,p0) == 0);// 页面恢复
    assert(*temp == 12345);                       // 第七个页面的值恢复为12345

    printf("pageReplacement_check2() succeeded\n");
}
```

测试结果

```
jovyan@1b42c1ef811a:~/18373489-lab$ bash test.sh
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.128.254
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
-------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

pageReplacement_check1() starts

the physic memory is full. Page replacement starts

repage=16383 replaced page's index is 16318,mapped va is 0x00001000

xthe physic memory is full. Page replacement starts

repage=16382 replaced page's index is 16317,mapped va is 0x00002000

xthe physic memory is full. Page replacement starts

repage=16381 replaced page's index is 16316,mapped va is 0x00003000

xthe physic memory is full. Page replacement starts
```

```
repage=16328 replaced page's index is 16263,mapped va is 0x00038000
xthe physic memory is full. Page replacement starts
repage=16327 replaced page's index is 16262,mapped va is 0x00039000
xthe physic memory is full. Page replacement starts
repage=16326 replaced page's index is 16261,mapped va is 0x0003a000
xthe physic memory is full. Page replacement starts
repage=16325 replaced page's index is 16260,mapped va is 0x0003b000
xthe physic memory is full. Page replacement starts
repage=16324 replaced page's index is 16259,mapped va is 0x0003c000
xthe physic memory is full. Page replacement starts
repage=16323 replaced page's index is 16258,mapped va is 0x0003d000
xthe physic memory is full. Page replacement starts
repage=16322 replaced page's index is 16257,mapped va is 0x0003e000
xthe physic memory is full. Page replacement starts
repage=16321 replaced page's index is 16256,mapped va is 0x0003f000
xthe physic memory is full. Page replacement starts
repage=16320 replaced page's index is 1127,mapped va is 0x00000000
xno more physic memory
pageReplacement_check1() succeeded
panic at init.c:30: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

GXemul> quit
jovyan@1b42c1ef811a:~/18373489-lab$
```

```
        starting cpu0 at 0x80010000
-------------------------------------------------------------------------------

main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
pageReplacement_check2() starts
the physic memory is full. Page replacement starts
repage=16383 replaced page's index is 16318,mapped va is 0x00002000
xthe physic memory is full. Page replacement starts
repage=16382 replaced page's index is 16317,mapped va is 0x00003000
xthe physic memory is full. Page replacement starts
repage=16381 replaced page's index is 16316,mapped va is 0x00004000
xthe physic memory is full. Page replacement starts
repage=16380 replaced page's index is 16314,mapped va is 0x00006000
xthe physic memory is full. Page replacement starts
repage=16379 replaced page's index is 16313,mapped va is 0x00007000
xpageReplacement_check2() succeeded
panic at init.c:30: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

GXemul> quit
jovyan@1b42c1ef811a:~/18373489-lab$
```