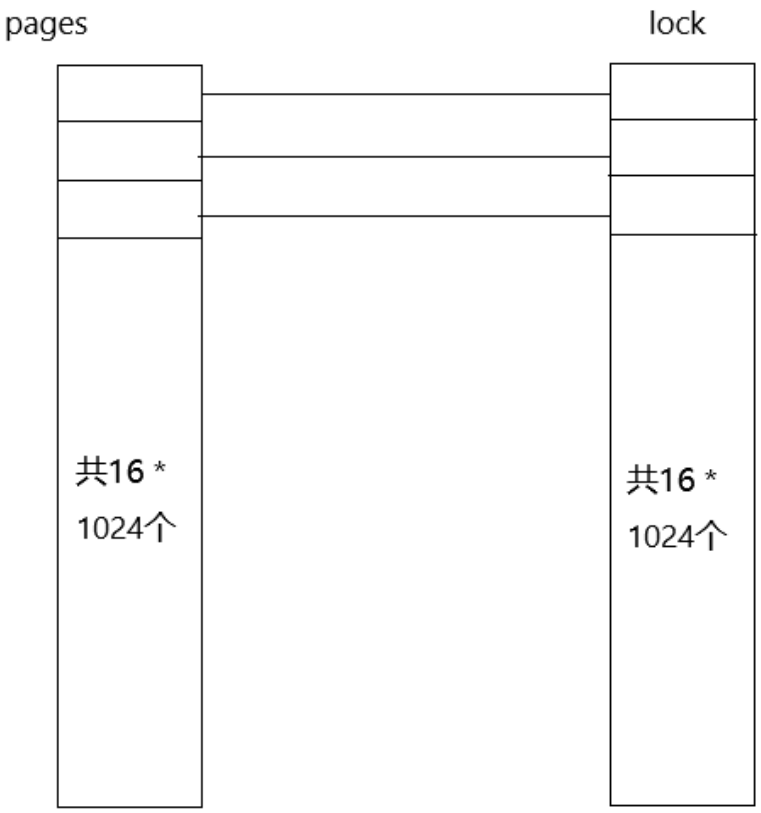
申优答辩 lab2-challenge

part1

简要思路



增加的数据结构

- `int lock[16 * 1024]`
- `int futureLock`

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()

```
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

```
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
for (i=0;i<PADDR(freemem)/BY2PG;i++){
    pages[i].pp_ref = 1;
    lock[i] = 1;
}
```

- page_free()

```
//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()

```
//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;
}
```

```
//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()

```
 //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

实验报告有具体说明

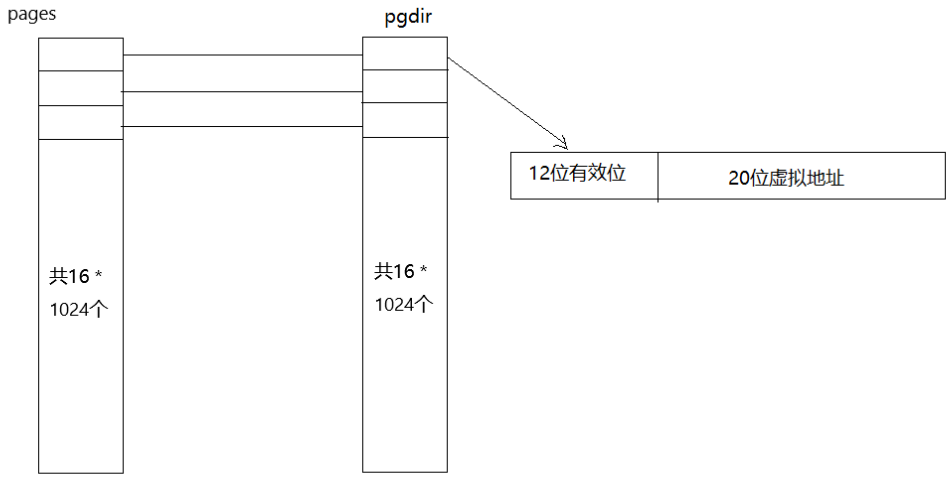
测试结果

实验报告有截图

part2

在Rpmap.c文件中

简要思路



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_mep_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()

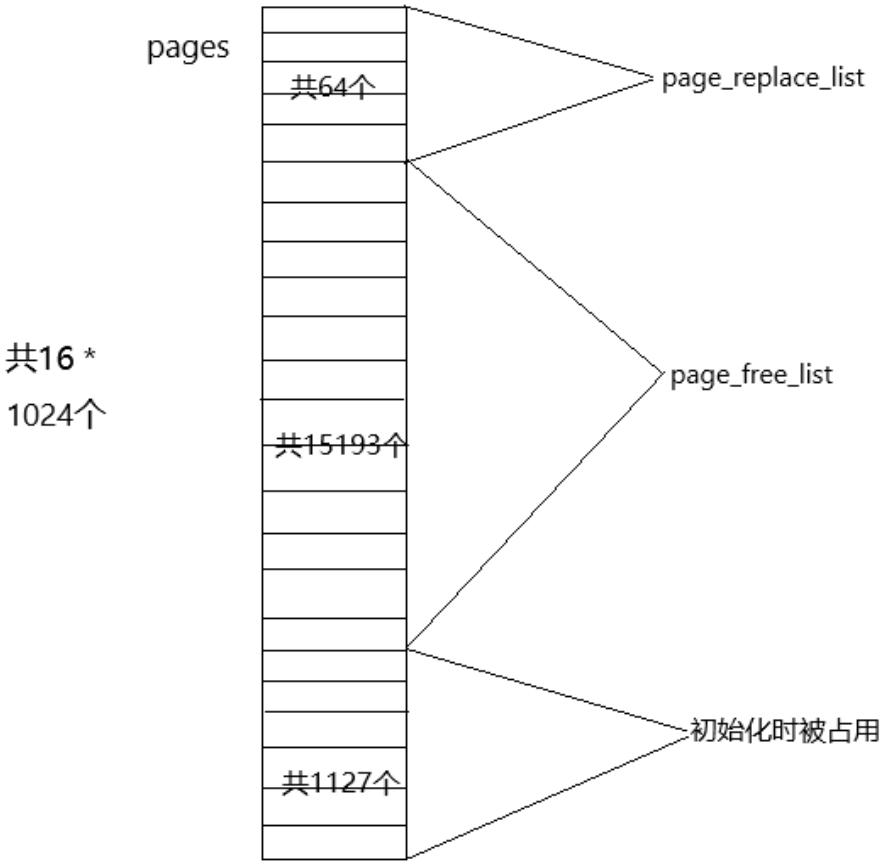
```
// 找到对应的页面，如果成功则返回索引值
int Rpage_lookup(Pde *pgdir, u_long va)
{
    int count = 0;
    int index = hashcode(va);
    while(!((*(pgdir + index)) == ((va >> 12) | VALID )))
    {
        index++;
        count++;
        index %= PAGESUM;
        if (count > PAGESUM) {
            printf("this physic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

测试函数

测试结果

part3

简要思路



增加的数据结构

- `static struct Page_list page_replace_list`
- `static struct Page_list page_used_list`
- `int replaceIndex`
- `u_long page2va[16 * 1024]`
- `Pde re2pde[REPLACENUM]`
- `Pte re2pte[REPLACENUM]`
- `u_long re2va[REPLACENUM]`

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");
            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()

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

测试函数

测试结果

（以上所有部分在实验报告中都有详细说明）