

Lab2 Report

朱新宇 1120379029

CSDI

2013年4月12日

JOS Lab2 Report

Part1. 物理内存的页管理

在part1中主要是通过pages来实现对物理内存的管理，操作系统通过pages结构可以以页为单位标记物理页的状态。

```
struct Page {
    // Next page on the free list.
    struct Page *pp_link;

    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.
    uint16_t pp_ref;
};
```

连续的Page数组线性代表了连续的物理页，pmap.h 提供了page2pa, pa2page和page2kva三个函数来处理page，在page2pa中通过传入的page和page的第一个元素地址的差值计算page对应的物理地址，再通过KADDR宏就可以转成kernel virtual address。

存放pages选择放在kernel的后面，也就是通过链接器提供的end获取链接的最后面的位置，然后分配相应的空间。

在实现上则是通过boot_alloc分配npages*sizeof(struct Page)的空间

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }

    // Allocate a chunk large enough to hold 'n' bytes, then update
    // nextfree. Make sure nextfree is kept aligned
    // to a multiple of PGSIZE.
    //
    // LAB 2: Your code here.

    if( n == 0 )
```

Lab2 Report

```
        return (void *)nextfree;

    result = nextfree;
    nextfree += n;
    nextfree = ROUNDDUP(nextfree, PGSIZE);
    return result;
}
```

在分配完空间后需要调用page_init初始化pages，在初始化时需要做的是标记已经被使用的page，然后用page_free_list连接所有空闲的page，我采用从后向前初始化，由于kernel以及pages等已经占用了一部分物理页，所以需要通过boot_alloc(0)来获取kernel代码空间的最后，从这里到extended memory的开始都是已经被占用的，此外IO hole和第一个页，完成后page_free_list指向第一个空闲的page。

```
void
page_init(void)
{
    // The example code here marks all physical pages as free.
    // However this is not truly the case. What memory is free?
    // 1) Mark physical page 0 as in use.
    //     This way we preserve the real-mode IDT and BIOS structures
    //     in case we ever need them. (Currently we don't, but...)
    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
    //     is free.
    // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
    //     never be allocated.
    // 4) Then extended memory [EXTPHYSMEM, ...).
    //     Some of it is in use, some is free. Where is the kernel
    //     in physical memory? Which pages are already in use for
    //     page tables and other data structures?
    //
    // Change the code to reflect this.
    // NB: DO NOT actually touch the physical memory corresponding to
    // free pages!
    size_t i;
    int cnt;
    uint32_t add;
    size_t mile = PADDR(boot_alloc(0)) / PGSIZE;
    page_free_list = NULL;
    pages[0].pp_ref = 1;
    for(i = npages - 1; i >= mile; --i){
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    for(i = npages_basemem - 1; i >= 1; --i){
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    for(i = npages_basemem; i < mile; ++i)
        pages[i].pp_ref = 1;
}
```

这时已经完成了pages的构建，然后还需要实现page_alloc和page_free来实现对page的分配和释放。

Lab2 Report

```
struct Page *
page_alloc(int alloc_flags)
{
    // Fill this function in
    struct Page* result = NULL;
    if(page_free_list){
        result = page_free_list;
        page_free_list = (*result).pp_link;
    }else{
        return NULL;
    }
    if(alloc_flags & ALLOC_ZERO){
        void *add = page2kva(result);
        memset(add, '\0', PGSIZE);
    }
    return result;
}
```

```
void
page_free(struct Page *pp)
{
    // Fill this function in
    (*pp).pp_link = page_free_list;
    page_free_list = pp;
}
```

然后实现page_alloc_4pages和page_free_4pages，在alloc是通过连续向后取4个page，然后检查地址的差距来确定这4个page是不是连续的。

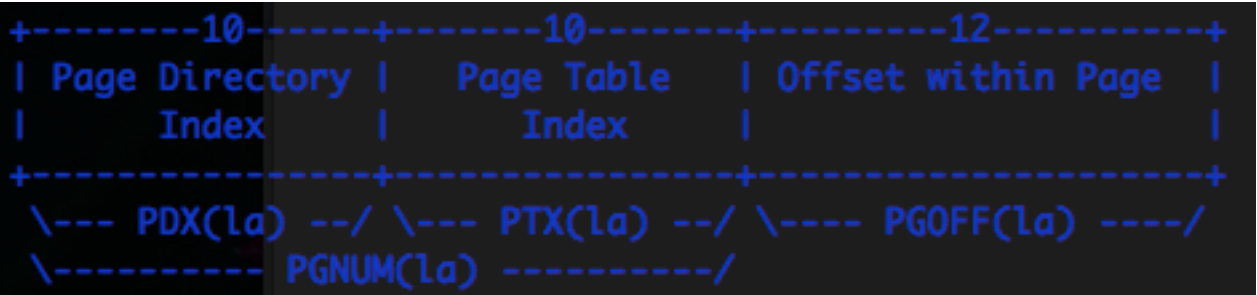
```
struct Page *
page_alloc_4pages(int alloc_flags)
{
    struct Page *head = page_free_list;
    struct Page *tail = head;
    struct Page *faketail = NULL;
    size_t i = 0;
    for(; i < 3; ++i)
        head = (*head).pp_link;
    while(head != NULL && head - tail != 3){
        head = (*head).pp_link;
        tail = (*tail).pp_link;
    }
    if(head == NULL)
        return NULL;
    faketail = tail;
    if(alloc_flags & ALLOC_ZERO){
        while(tail != head){
            void *add = page2kva(tail);
            memset(add, '\0', PGSIZE);
            tail = (*tail).pp_link;
        }
        void *add = page2kva(tail);
        memset(add, '\0', PGSIZE);
    }
    // Fill this function
    return faketail;
}
```

Part2. 虚拟地址内存

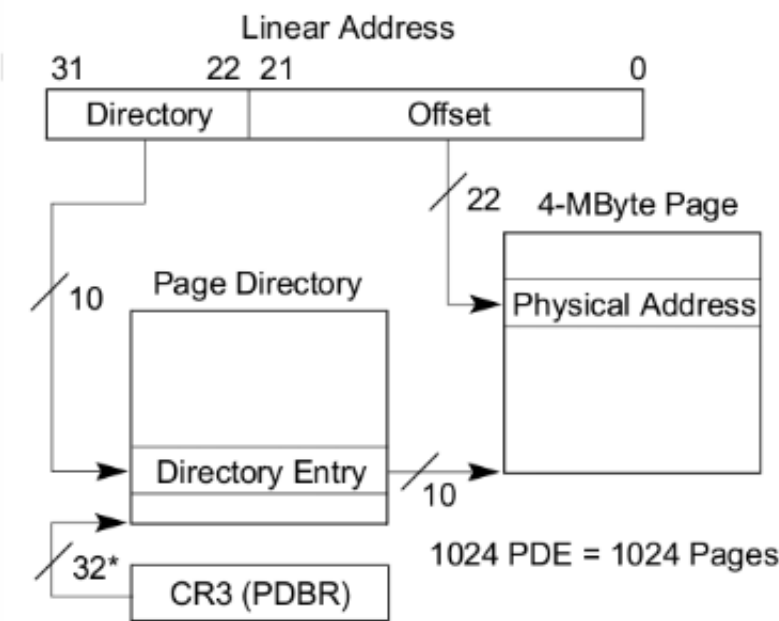
在实现虚拟内存时，必须先构建页表以实现地址的映射，由于有两级页表，第一级是page directory，第二级是page ta-

Lab2 Report

ble，分别对应虚拟地址的10bit长度，其结构可以如下图所示。



下图则表示了从一个virtual address通过二级页表翻译到实际物理内存的过程。



pdt*是指向page directory的指针，pgdir_walk作用就是把一个virtual address反映称对应的pt页表项，如果没有则根据要求创建。在pdt和和pte中存储的都是实际的物理地址。

boot_map_region则是映射一片虚拟地址到制定的物理地址。在实现的时候则是通过pgdir_walk创建对应页表项。

Lab2 Report

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    if(!pgdir)
        panic("pgdir is NULL");
    pde_t pde;
    pte_t *ptable;
    pde = pgdir[PDX(va)];
    if( pde & PTE_P ){
        ptable = (pte_t *)KADDR(PTE_ADDR(pde)); //pde is physical address of
        corresponding ptable
        return &ptable[PTX(va)];
    }
    if(!create)
        return NULL;
    struct Page *newpage = page_alloc(ALLOC_ZERO);
    if(newpage == NULL)
        return NULL;
    ++newpage->pp_ref;
    pgdir[PDX(va)] = page2pa(newpage) | PTE_P | PTE_W | PTE_U;
    return &((pte_t*)page2kva(newpage))[PTX(va)];
}
```

page_lookup则实现了从virtual address查找对应的物理内存page，同样是通过pgdir_walk找到对应的pte,然后通过pa2page转换成对应page。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte)
        return NULL;
    if(pte_store){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

page_insert则是实现新建virtual address到physical address的映射。

Lab2 Report

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    // Fill this function in
    physaddr_t add = page2pa(pp);
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte){
        pte = pgdir_walk(pgdir, va, 1);
        if(!pte){
            return -E_NO_MEM;
        }else{
            if(*pte & PTE_P)
                page_remove(pgdir, va);
        }
    }
    if(pp == page_free_list)
        page_free_list = page_free_list->pp_link;
    *pte = add | PTE_P | perm;
    pp->pp_ref++;
    return 0;
}
```

page_lookup则是查找一个virtual address对应的Page。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    pte_t *pte = pgdir_walk(pgdir, va, 0);
    if(!pte)
        return NULL;
    if(pte_store){
        *pte_store = pte;
    }
    return pa2page(PTE_ADDR(*pte));
}
```

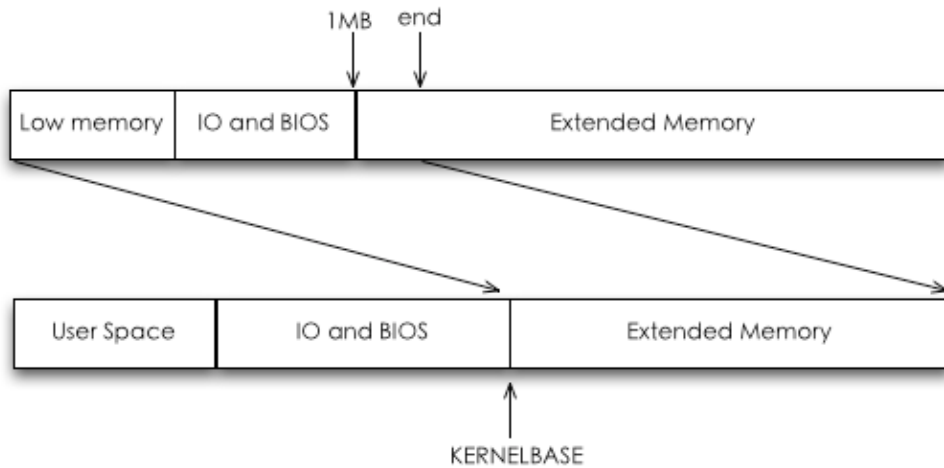
现在虚拟地址基本已经实现。

Part3. Kernel Address Space

在实现了基本的虚拟内存后，需要对内核的虚拟地址映射到实际load的物理地址，这就解决了link address和实际加载的内存地址不相符和的原因，之前entry.S中加载了entry_pgdir这个大页表，可以保证在virtual address完成前，内核空间的代码可以正常运行

下图展示了一部分映射的过程

Lab2 Report



而整个虚拟地址的分布则如下图所示

```

12      4 Gig          16          32          64          128          192          256          512          1024          2048          4096          8192          16384          32768          65536          131072          262144          524288          1048576          2097152          4194304          8388608          16777216          33554432          67108864          134217728          268435456          536870912          1073741824          2147483648          4294967296          8589934592          17179869184          34359738368          68719476736          137438953472          274877906944          549755813888          1099511627776          2199023255552          4398046511104          8796093022208          17592186044416          35184372088832          70368744177664          140737488355328          281474976710656          562949953421312          1125899906842624          2251799813685248          4503599627370496          9007199254740992          18014398509481984          36028797018963968          72057594037927936          144115188075855872          288230376151711744          576460752303423488          1152921504606846976          2305843009213693952          4611686018427387904          9223372036854775808          18446744073709551616          36893488147419103232          73786976294838206464          147573952589676412928          295147905179352825856          590295810358705651712          1180591620717411303424          2361183241434822606848          4722366482869645213696          9444732965739290427392          18889465931478580854784          37778931862957161709568          75557863725914323419136          151115727451828646838272          302231454903657293676544          604462909807314587353088          1208925819614629174706176          2417851639229258349412352          4835703278458516698824704          9671406556917033397649408          19342813113834066795298816          38685626227668133590597632          77371252455336267181195264          154742504910672534362390528          309485009821345068724781056          618970019642690137449562112          1237940039285380274899124224          2475880078570760549798248448          4951760157141521099596496896          9903520314283042199192993792          19807040628566084398385987584          39614081257132168796771975168          79228162514264337593543950336          158456325028528675187087900672          316912650057057350374175801344          633825300114114700748351602688          1267650600228229401496703205376          2535301200456458802993406410752          5070602400912917605986812821504          10141204801825835211973625643008          20282409603651670423947251286016          40564819207303340847894502572032          81129638414606681695789005144064          162259276829213363391578010288128          324518553658426726783156020576256          649037107316853453566312041152512          1298074214633706907132624082305024          2596148429267413814265248164610048          5192296858534827628530496329220096          10384593717069655257060992658440192          20769187434139310514121985316880384          41538374868278621028243970633760768          83076749736557242056487941267521536          166153499473114484112975882535043072          332306998946228968225951765070086144          664613997892457936451903530140172288          1329227995784915872903807060280344576          2658455991569831745807614120560689152          5316911983139663491615228241121378304          10633823966279326983230456482242756608          21267647932558653966460912964485513216          42535295865117307932921825928971026432          85070591730234615865843651857942052864          170141183460469231731687303715884105728          340282366920938463463374607431768211456          680564733841876926926749214863536422912          1361129467683753853853498429727072845824          2722258935367507707706996859454145691648          5444517870735015415413993718908291383296          10889035741470030830827987437816582766592          21778071482940061661655974875633165533184          43556142965880123323311949751266331066368          87112285931760246646623899502532662132736          174224571863520493293247799005065324265472          348449143727040986586495598010130648530944          696898287454081973172991196020261297061888          1393796574908163946345982392040522594123776          2787593149816327892691964784081045188247552          5575186299632655785383929568162090376495104          11150372599265311570767859136324180752990208          22300745198530623141535718272648361505980416          44601490397061246283071436545296723011960832          89202980794122492566142873090593446023921664          178405961588244985132285746181186892047843328          356811923176489970264571492362373784095686656          713623846352979940529142984724747568191373312          1427247692705959881058285969449495136382746624          2854495385411919762116571938898990272765493248          5708990770823839524233143877797980545530986496          11417981541647679048466287755595961091061972992          22835963083295358096932575511191922182123945984          45671926166590716193865151022383844364247891968          91343852333181432387730302044767688728495783936          182687704666
```

```
boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages * sizeof(s-
truct Page), PGSIZE), PADDR(pages), PTE_U | PTE_P);
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PAD-
DR(bootstack), PTE_W);
boot_map_region(kern_pgdir, KERNBASE, ROUNDUP(0xffffffff-
KERNBASE, PGSIZE), 0, PTE_W);
```

第一句话映射pages结构到UPAGES，第二句话映射了KernelStack，最后一句话则映射KernelBase到最后的所有内存。

Question 3:

Pde和pte有U/S和R/W位可以保护内核内存地址的访问，防止来自用户程序的访问。

Question 4:

Lab2 Report

由于是32位，所以支持4G。

Question 5

page directory占用了4KB的空间，pages数组占用了4MB，page table占用了4MB

Question 6:

```
move $relocated, %eax
```

```
jmp %eax
```

这两句话把EIP转移到KERNELBASE以上，在此之前如在此报告上文中所述，从0开始的一个大页表（4MB）已经映射到了KERNELBASE开始的空间

```
movl    $(RELOC(entry_pgdir)), %eax
```

```
movl    %eax, %cr3
```

```
# Turn on paging.
```