CSDI Lab 2

1120379050 王灏 wh.sjtu@gmail.com

April 17, 2013

Exercise 1:

In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).

```
boot_alloc()
mem_init() (only up to the call to check_page_free_list(1))
page_init()
page_alloc()
page_free()
```

`check_page_free_list()` and `check_page_alloc()` test your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

根据 `pmap.c` 函数说明，`boot_alloc()` 函数用于allocate物理内存，在JOS建立起虚拟内存管理系统之前。同时，`boot_alloc()` 在参数 `n > 0` 时，将检查是否有n个byte的空闲内存空间，如果有，则返回该空间的 `kernel virtual address`，没有就调用 `panic`。如果 `n == 0` 则返回下一个空闲页地址。

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree;
    char *result;

    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }

    // LAB 2: Your code here.
    //对齐page
    result = ROUNDUP(nextfree, PGSIZE);
    //需要开辟的内存
    uint32_t alloc_space = (uint32_t) result - KERNBASE + n;
    //由i386_detect_memory函数获得空闲物理页数npages
    //计算出空闲空间
    uint32_t total_space = (uint32_t) npages * PGSIZE;
    if(alloc_space > total_space){
    //空间不够，报panic
        panic("[boot_alloc] out of physical memory\n");
    }
    //nextfree指向下一个空闲空间的虚拟地址
    nextfree = result + n;
    return result;
}
```

在 mem_init() 函数中根据提示，要实现一个 array 来跟踪每一个物理页，要给这样一个 array 开辟空间。在 memelayout.h 里可以看到 struct Page 的定义，实际上要开辟的空间，就是 struct Page 的大小乘以页数 npages：

```
// the size of space to store all the Page structs
size_t pages_sz = npages * sizeof(struct Page);
//申请空闲页，获得始地址
pages = (struct Page *) boot_alloc(pages_sz);
//初始化空间，全赋0表示已经占用
memset(pages, 0, pages_sz);
```

page_init() 负责建立一个单项链表 page_free_list 。根据提示在物理地址上需要避开 IO hole 地址段：

```
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;
    page_free_list = NULL;
    //获得空闲空间起始虚拟地址，并转成物理地址
    size_t EXTPHYSMEM_END = PADDR(boot_alloc(0)) / PGSIZE;
    //ROUNDDOWN in case of occupying address of IO hole
    size_t IOPHYSMEM_START = ROUNDDOWN(IOPHYSMEM, PGSIZE) / PGSIZE;

    for (i = npages-1; i >= 0; i--) {
        pages[i].pp_ref = 0;
        if(i == 0) {
        //首节点
            break;
        }
        if(IOPHYSMEM_START <= i && i < EXTPHYSMEM_END) {
        //在IO hole的范围内
            continue;
        }
        //每一个新page都放在page_free_list单向链表的首位
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

page_alloc() 是在 page_free_list 建立以后，真正的内存分配器。 page_alloc() 只分配一页:

```
struct Page *
page_alloc(int alloc_flags)
{
    // Fill this function in
    struct Page * page_ptr;
    //判断page_free_list是否为空
    if(page_free_list != NULL){
        page_ptr = page_free_list;
        //更新单项链表的首节点
        page_free_list = (struct Page *) page_free_list->pp_link;
        if(alloc_flags & ALLOC_ZERO) {
            memset(page2kva(page_ptr), '\0' , PGSIZE);
        }
        return page_ptr;
    } else {
        return NULL;
    }
}
```

空闲页回收在函数 page_free() 中进行，即操作 page_free_list 单向链表:

```
void
page_free(struct Page *pp)
{
    // Fill this function in
    //put the pp to the head of page_free_list
    pp->pp_link = page_free_list;
    page_free_list = pp;
    return;
}
```

到这里，物理内存的allocator可以说完成了。 boot_alloc 临时分配器在物理内存建立了页数组，然后 page_alloc 根据页数组进行内存分配。

Exercise 2:

In the file kern/pmap.c , you must implement code for the following functions.

```
page_alloc_4pages()
page_free_4pages()
```

check_four_pages() , called from mem_init() , tests your page table management routines. Youshould make sure it reports success before proceeding.

根据 check_continuous() 可知 page_alloc_4pages() 函数需要分配出物理地址相邻的4页。需要用到 page2pa() 函数。(由于在 page_init() 函数中建立的 page_free_list 是从

首到尾)

```
struct Page *
page_alloc_4pages(int alloc_flags) {

    struct Page *result;
    int found = 0;
    int cnt=0;
    if (!page_free_list) {
        panic("page_alloc_4pages: out of page_free_list");
        return NULL;
    }
    result = page_free_list;
    while(result->pp_link) {
        //检查是否相邻
        if(check_continuous(result)) {
            found = 1;
            break;
        }
        //找到相邻的4页
        result = result->pp_link;
    }
    if (found) {
        //分配空间
        page_free_list = result->pp_link->pp_link->pp_link->pp_link;
        if (alloc_flags & ALLOC_ZERO) {
            memset(page2kva(result), 0, PGSIZE * 4);
        }
        return result;
    } else {
        panic("page_alloc_4pages: out of memory1");
        return NULL;
    }
}
```

page_free_4pages() 将连续的4页加入到 page_free_list 里面，首先对4页的连续性做验证，然后再添加到chunk_list里：

```
int
page_free_4pages(struct Page *pp)
{
    // Fill this function
    int i;
    if(check_continuous(pp)) {
        page_free_list = pp->pp_link->pp_link->pp_link->pp_link;
        chunck_list = pp;
    } else {
        return -1;
    }
    return 0;
}
```

Question 1:

1.Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t` ?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

x 变量类型应该为 `unitptr_t` ， `value` 被一个指针赋值，而 x 则是被 `value` 这一指针的指针值赋值，而C程序中指针指向的是虚拟地址，因此 x 为 `unitptr_r` 。

Exercise 5:

In the file kern/pmap.c, you must implement code for the following functions.

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

`check_page()` , called from `mem_init()` , tests your page table management routines. You should make sure it reports success before proceeding.

`page_walk` 函数，接收一个指向 `page directory` 的指针，以及对应的线性地址 `va` ，返回指向 `page table entry` 的指针.相当于需要访问 `Page Directory` 和 `Page Table` 两级结构。

同时，根据guide说明，如果对应的页表项的页不存在，则申请分配这样一个页，并定义相应的权限：

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    // Fill this function in
    pte_t *pgdir_entry = &pgdir[PDX(va)];

    //if the pgdir_entry exists
    if(*pgdir_entry & PTE_P) {
        pte_t *pt_va = (pte_t *) KADDR(PTE_ADDR(*pgdir_entry));
        return pt_va + PTX(va);
    } else {
        //if the page table does not exist
        if(!create) {
            return NULL;
        } else {
            //ask for a new physical page
            struct Page *new_pgt = page_alloc(1);
            if(new_pgt != NULL) {
            //分配成功
                new_pgt->pp_ref = 1;
                //转成物理地址
                physaddr_t new_pgt_pa = page2pa(new_pgt);
                //设置属性
                *pgdir_entry = new_pgt_pa | PTE_W | PTE_U | PTE_P;
                return (pte_t *)KADDR(new_pgt_pa) + PTX(va);
            } else {
                //no more space
                return NULL;
            }
        }
    }
    return NULL;
}
```

函数 boot_map_region() 将一段虚拟地址空间映射到一段物理地址空间。输入是指向 page directory 的指针，虚拟起始地址，空间大小，物理起始地址，以及权限标志位。

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int
perm)
{
    // Fill this function in
    uintptr_t unit;
    pte_t *pgt_entry;
    //循环对每一页分配的页进行设置
    for(unit = 0; unit < size; unit += PGSIZE) {
        //get the page table entry for every page
        pgt_entry = pgdir_walk(pgdir, (void *)va, 1);
        //set the permission and
        *pgt_entry = pa | perm | PTE_P;
        pa += PGSIZE;
        va += PGSIZE;
    }
    return;
}
```

查找页如果实现在 page_lookup() 中，即通过 pgdir 指针和线性地址 va 获得指向对应页的指针。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    // Fill this function in
    //获得对应页表项的指针
    pte_t *pgt_entry = pgdir_walk(pgdir, (void *)va, 0);

    if(pgt_entry == NULL || (*pgt_entry & PTE_P) == 0) {
        //页表项指针为空，或者页表项不存在
        return NULL;
    }
    if(pte_store != 0) {
        //如果pte_store不为0， 则存放页表项地址
        *pte_store = pgt_entry;
    }
    return pa2page(PTE_ADDR(*pgt_entry));
}
```

page_remove() 用来删除物理页和虚拟页地址的映射关系。要删除一个页的映射，首先要找到这个页，然后对这个页进行 page_decref() ，同时 tlb_invalidate() 。

```
void
page_remove(pde_t *pgdir, void *va)
{
    // Fill this function in
    pte_t *pgt_entry;
    struct Page *pg_rm;
    pg_rm = page_lookup(pgdir, va, &pgt_entry);

    if(pg_rm != NULL) {
        page_decref(pg_rm);
        *pgt_entry = 0;
        tlb_invalidate(pgdir, va);
    }
    return;
}
```

page_insert() 用来建立物理地址 pp 到虚拟地址 va 的映射关系。根据guide，如果已经有物理页被映射到 va ，则将该页删除。新建页应该分配到对应的 pgdir 里， page 的引用应该被增加。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pgt_entry = pgdir_walk(pgdir, (void *)va, 1);

    if(pgt_entry == NULL) {
        //页表分配失败
        return -E_NO_MEM;
    } else {
        if(*pgt_entry & PTE_P) {
            if(PTE_ADDR(*pgt_entry) == page2pa(pp)) {
                //如果pp和va已经存在映射关系，先去除
                pp->pp_ref--;
                tlb_invalidate(pgdir, va);
            } else {
                //如果va和其他页有映射关系，去除该页
                page_remove(pgdir, va);
            }
        }
    }
    *pgt_entry = page2pa(pp) | perm | PTE_P;
    pp->pp_ref++;
    return 0;
}
```

至此，页表的管理函数大致已经完成了。通过 make qemu-nox 可以看到 check_page()

```
successed!
```

Exercise 6:

Fill in the missing code in `mem_init()` after the call to `check_page()`.

在 `mem_init()` 函数中，`check_four_pages()` 之后，有三个地方需要填写，根据guide，是需要实现虚拟内存地址到相应物理地址的映射，和权限分配。首先是将 pages 这一页链表的物理地址映射到UPAGES开始的虚拟地址。

```
size_t size = ROUNDUP(npages * sizeof(struct Page), PGSIZE);
boot_map_region(kern_pgdir, UPAGES, size, PADDR(pages), PTE_W|PTE_P);
```

然后则是 [KSTACKTOP-PTSIZE, KSTACKTOP) 的映射，其中 [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) 这一段是invalid memory，所以映射如下：

```
boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE,
                PADDR(bootstack), PTE_W|PTE_P);
```

接下来是整个从KERNBASE以来的虚拟地址，和256MB的物理地址的全部映射：

```
boot_map_region(kern_pgdir, KERNBASE, 256*1024*1024, 0, PTE_W|PTE_P);
```

这样便完成了相关映射了。

Question 2:

1. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

   In the QEMU monitor, we could get the page-table by `info pg`:

   ```
   (qemu) info pg
   PDE(001) ef000000-ef400000 00400000 urw
    |-- PTE(000021) ef000000-ef021000 00021000 -rw
   PDE(001) ef400000-ef800000 00400000 ur-
    |-- PTE(000001) ef7bc000-ef7bd000 00001000 urw
    |-- PTE(000001) ef7bd000-ef7be000 00001000 ur-
    |-- PTE(000001) ef7be000-ef7bf000 00001000 urw
    |-- PTE(000040) ef7c0000-ef800000 00040000 urw
    |-- PTE(000008) efbf8000-efc00000 00008000 -rw
   PDE(040) f0000000-00000000 10000000 urw
    |-- PTE(010000) f0000000-00000000 10000000 -rw
   ```

But with page information above it's hard to infer the information to fill the table.

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
1022	?	?
…	…	…
960	0xf0000000	KERNBASE
959	0xefc00000	VPT, KSTACKTOP
958	0xef800000	ULIM
957	0x00800000	UVPT
956	0x00800000	UPAGES
955	0x00800000	UTOP, UENVS, UXSTACKTOP
…	…	…
2	0x00800000	UTEXT
1	0x00400000	UTEMP
0	0x00000000	IDT/BIOS

2. (From Lecture 3) We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

   Because the kernel should be protected from being modified by user applications which access the virtual address beyond the ULIM, or the system will break down. To protect the kernel memory, there are U/S, R/W permission flags to be set to PDE and PTE, which control the accesses to the corresponding memory space.

3. What is the maximum amount of physical memory that this operating system can support? Why?

   As the system allocate a PTSIZE (0x00400000) to `pages`, and the size of `struct Page` is 12 bytes. So in a `pages` there will be 1/3 M `Page`s, and one `Page` in `pages` maps to a 4KB physical page, which means the max size of memory supported could be 1/3M * 4KB = 1.33GB.

4. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

   JOS adopts 2-level Page memory management system. One page directory, 4KB; 1024 * page entries, and 4KB per page entry, but as the following code in `kern/pmap.c` puts out, the `kern_pgdir` recursively insert PD in itself as a page table, to form a virtual page table at virtual address UVPT.

```
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

so altogether, the memory space for management will be `(1+1024-1)*4KB = 4MB`.

To reduce the overhead, we could increase the page size, which results in less page entries.

5. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?

   In entry.S:

```
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov $relocated, %eax
jmp *%eax
```

   After paging is turned on, the EIP is still low, but code `mov $relocated, %eax` and `imp %eax` works normally, we can find the reason explained in `kern/entrypgdir.c`:

```
// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0,
4MB)).
// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.  We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
```

   The virtual addresses [0, 4MB) are also mapped to physical addresses [0, 4MB). So those two instructions could work well.