JOS2实验报告

王正1110379017

2012 年 7 月 4 日

1 准备工作

首先需要提交lab1的源代码。

```
wang@ThinkPad:~/lab$ git commit -am 'my␣solution␣to␣
   lab1'
[lab1 934bc25] my solution to lab1
 4 files changed, 82 insertions(+), 7 deletions(-)
```

然后建立新的分支。

```
wang@ThinkPad:~/lab$ git pull
Already up-to-date.
wang@ThinkPad:~/lab$ git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch lab2 from
   origin.
Switched to a new branch 'lab2'
```

合并lab1的代码。

```
wang@ThinkPad:~/lab$ git merge lab1
Merge made by recursive.
 kern/console.c |    7 ++++++-
 kern/kdebug.c  |    7 ++++++--
 kern/monitor.c |   39
    ++++++++++++++++++++++++++++++++++++++++
 lib/printfmt.c |   36
    ++++++++++++++++++++++++++++++++----
 4 files changed, 82 insertions(+), 7 deletions(-)
```

在合并的过程中发现，因为自己曾经修改过kern/init.c 增加了彩色打印的功能，

```
cprintf("color test: \033[0;32;40m hello \033[0;36;41mworld\033[0;37;40m\n");
```

但是发现无法合并，所以就恢复了init.c到初始状态下。

2 物理页面管理

lab2主要是实现内存的分页管理和虚存的实现。通常管理内存的方式有两种，分段和分页，这里jos实现的是分页管理。

> 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.

代码集中在kern/pmap.c中。主要修改boot_alloc()，mem_init()，page_init()，page_alloc()，page_free()这几个函数。可以通过check_page_free_list() 和check_page_alloc()检测分配是否正确。正确之后会有check_page_alloc() succeeded!的提示。boot_alloc()初始化页目录和页表的空间。在pmap.c中mem_init()调用

```
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);
```

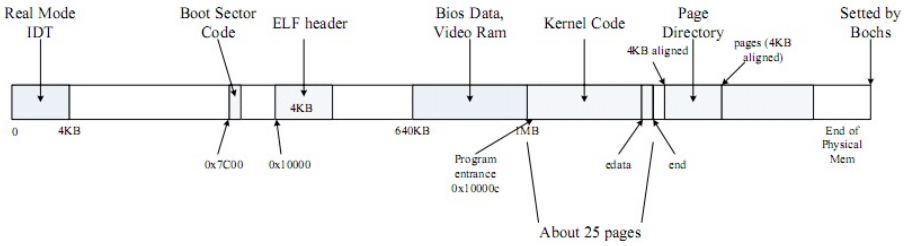
和

```
// array.  'npages' is the number of physical pages in memory.
// Your code goes here:

pages =(struct Page *) boot_alloc(npages* sizeof (struct Page));
memset(pages, 0, npages* sizeof (struct Page));
```

实现，这里页表是模仿页目录的写法而写成的。下面可以看一下页表的数据结构

```
struct Page {
struct Page *pp_link;
uint16_t pp_ref;
};
```



这个就是个简单的链表的节点，pp_link指向下个节点，pp_ref表示引用次数。空闲的节点引用次数为0. 页目录和页面链表紧接着内核空间后分配。 我们可以从这张图上看到实际上的内核层次。Page Directory 在内核代码end之后按页对齐的地方开始，在一页之后是页面的链表。所以boot_alloc()主要功能就是给这两项分配内存。

```
// 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.
result = ROUNDUP(nextfree, PGSIZE);
nextfree = result + n;
//cprintf("\nnextfree:0x%08x",nextfree);
return result;
```

所做的添加如上，这里result是分配的内存起始位置，nextfree是下一块空闲区域开始的地方。这里的地址都是物理地址。