Lab2实验报告

1611251 刘逸伦

目录：

-[Exercise1](#Exercise1)

-[问题1](#问题1)

-[Exercise4](#Exercise4)

-[Exercise5](#Exercise5)

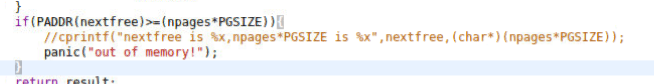
-[问题2-5](#问题2)

-[Make Grade](#Makegrade)

-[Challenge](#Challenge)

1. JOS监视器命令：展示内存范围内的映射关系、权限位和内容，并修改和重置权限位
2. JOS监视器：实现按向上箭头读取上一条历史命令，按下箭头回退一条命令
3. 连续分配多个页面的malloc()，实现思路

Exercise1



1. boot\_alloc 判断内存是否用尽，一开始总是报内存用尽。后来发现原因是nextfree是内核地址，忘记在判断时把它转换为物理地址了，导致nextfree 是f开头的数，必大于总内存。
2. Issue:实验说明书的此处不应该要求学生使用check\_page\_alloc来检查代码，因为check\_page\_alloc会调用此刻尚未编写的page\_insert()函数



**问题 1.**

假设以下 JOS 内核代码是正确的，变量 x 应该是什么类型？uintptr\_t 还是 physaddr\_t？

mystery\_t x;

char \* value = return\_a\_pointer（）;

\* value = 10;

x =（mystery\_t）value;

Answering: x应该是uintptr\_t，因为它是由内核分配的地址。

Exercise4:

1. 三种地址没有搞清楚。线性地址、物理地址和内核映射地址。内核映射地址=物理地址+f0000000;

PDE和PTE应储存为页的物理地址，而在从PDE读取PT时， 应该把读到的物理地址先取前二十位，再转变为内核地址，否则就会出现超限错误。。

1. pgdir\_walk这个函数要求返回的是PTE!!!!不是PT!!!PTE是page table 还要遍历一次得到的指针！！！
2. 关于page\_insert（）重复插入相同页的问题。

在page\_insert(pgdir,pp,va)中，首先要进行page\_remove操作，移除地址va上原有的页，然后再把pp放到va里。当连续的两次插入相同页pp时，第二次运行时会把pp先remove，再插入一次pp。在remove时，由于pp的ref已经变成了0，所以pp会被放到page\_free\_list中，而插入pp时，在page\_insert中并没有执行分配页面的操作，而只是使pp\_ref加1，所以会导致pp错误地存在于page\_free\_list中。

解决办法一（是错误的解决办法）：

修改page\_alloc，使得当读取到的页面ref值不为0时，跳过这个页面，继续向前读取一个，直到有ref值为0的页面为止或直到页面为NULL。

这个方法的问题在于若连续的三次插入相同页pp时，第三次的remove会把pp再一次放到page\_free\_list上，并且让pp指向下一个元素，也就是它本身，自己指向了自己：

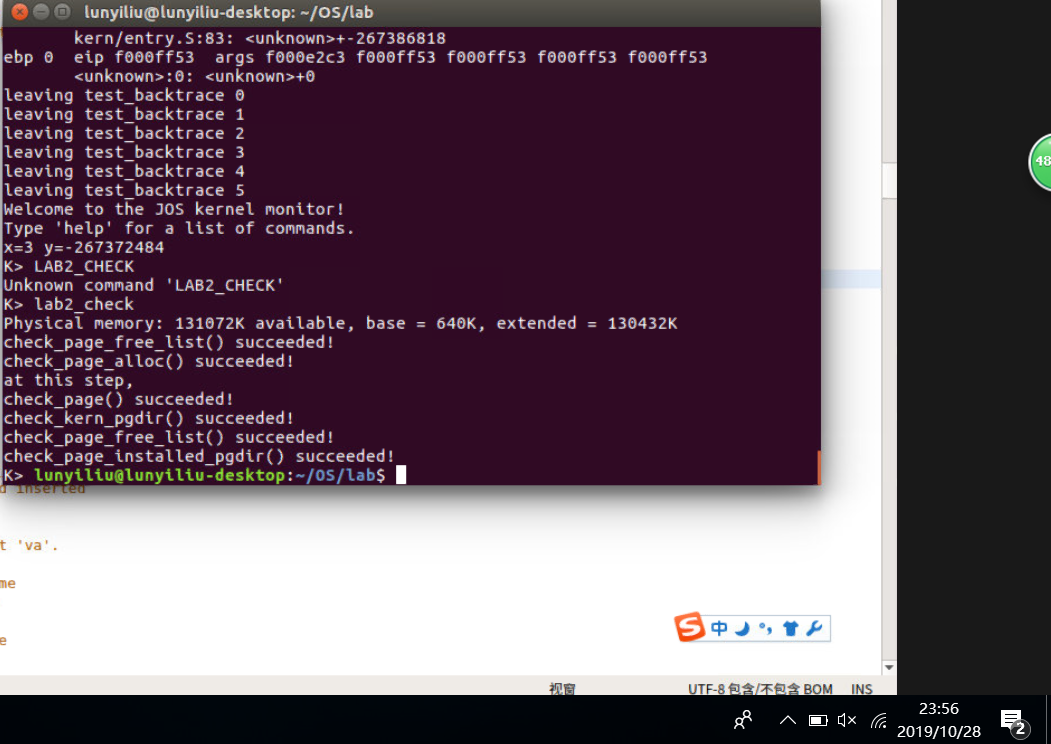
Page\_free\_list: <-pp<-pp

按照解决办法一的话，会出现死循环！！

解决办法二：

在remove之前就进行pp\_ref加一的操作。这样即使插入了相同的pp，remove的时候也不会把pp放到page\_free\_list上了。

Exercise5:



**问题 2.**

假设下图描述的是系统的页目录表，哪些条目（行）已经被填充了？它们是怎么样进行地址映射的？它们所指向的位置在哪里？请尽可能完善这张表的内容。

| **Entry** | **Base Virtual Address** | **Points to (logically)** |
| --- | --- | --- |
| 1023 | 4G limitation | Page table for top 4MB of phys memory |
| 960 | KernBase | 960-1023 Page tables for kern memory |
| 959 | KernelStack | Page table for kernel Stack memory |
| 958 | MMIOBASE | Memory-mapped I/O |
| 957 | UVPT | Page table for page directory storage |
| 956 | Upages | Page table for physical PageInfo array |

**问题**

1. 我们已经将内核和用户环境放在同一地址空间内。为什么用户的程序不能读取内核的内存？有什么具体的机制保护内核空间吗？

Answering: 因为用户的程序一旦出错可能会造成内核数据的破坏，导致计算机崩溃，而且用户有可能盗取、篡改内核数据。 在页目录和页表中，每个元素的后12位都被设置为权限位，以此划分出只有内核能访问的内核空间。

1. JOS 操作系统可以支持的最大物理内存是多少？为什么？

Answering :3.73G，内核占用了68个页表，也就是272MB空间。

1. 如果我们的硬件配置了可以支持的最大的物理内存，那么管理内存空间的开销是多少？这一开销是怎样划分的？

Answering:: 管理内存空间的开销是4100KB， 也就是1025页，其中1页用于存放page directory， 1024页用于存放page table。

1. 再次分析 kern/entry.S 和 kern/entrypgdir.c 的页表设置的过程，在打开分页之后，EIP 依然是一个数字（稍微超过 1MB）。在什么时刻我们才开始在 KERNBASE 上运行 EIP 的？当我们启动分页并在 KERNBASE 上开始运行 EIP 之时，我们能否以低地址的 EIP 继续执行？这个过渡为什么是必须要的？

Answering:

在这两个文件的分页设置过程中，[0-4MB]的页面被手动分配到了一个初始页表entry\_page\_table中，然后再把这个页表映射到了[KERNBASE,KERNBASE+4MB] 和 [0-4MB]的虚拟位置。

在这一句EIP跳到了高地址：

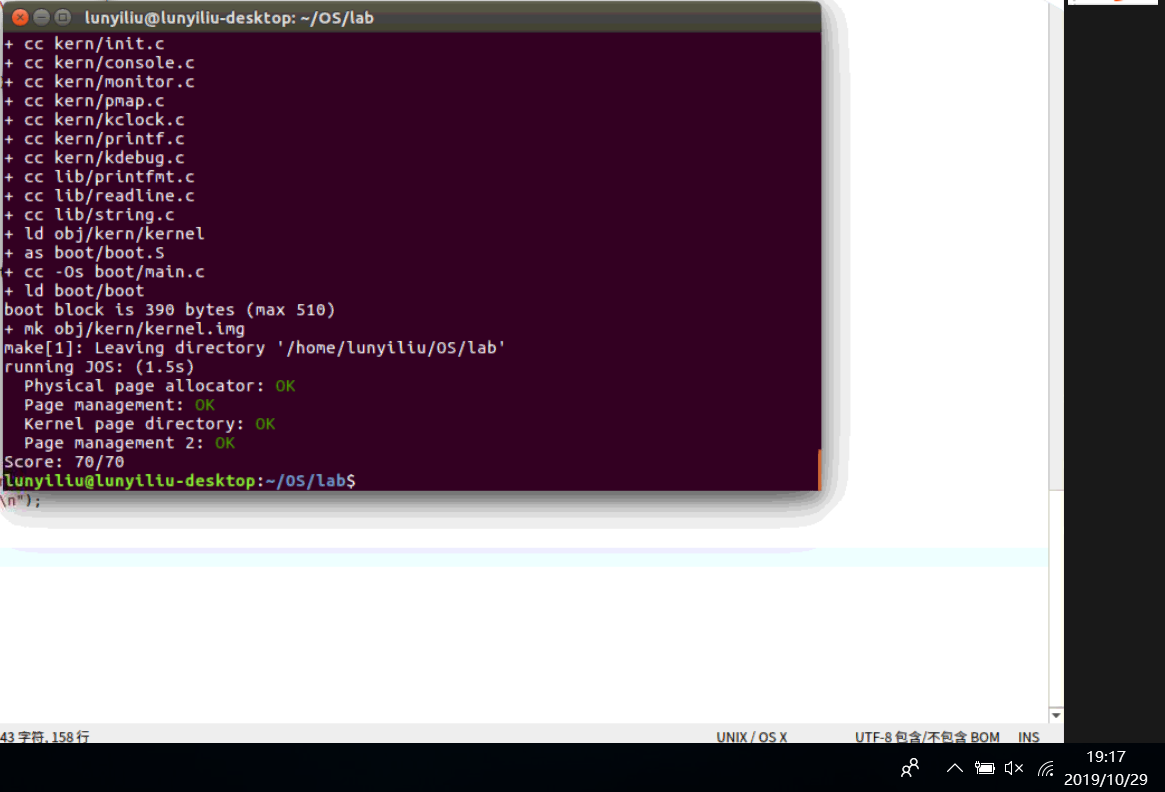
mov $relocated, %eax

jmp \*%eax

这时依然是可以以低地址的EIP执行的，因为虚拟地址的[0-4MB]也被映射到了实际的地址中。

在本次实验中，写boot\_alloc时，我们直接就把pgdir代入memset中了， 而此时pgdir 是虚拟地址。之所以可以这么做就是因为在Entry.S中对页目录和页表进行了部分分配

Make Grade:截图



Challenge:

1. JOS监视器命令：展示内存范围内的映射关系、权限位和内容，并修改和重置权限位

实现：

1. 调通监视器命令参数传递的过程。
2. 在monitor.c中写新函数。利用pgdir\_walk()函数拿到某个虚拟地址对应的pte，然后读取该pte前二十位作为物理地址，把后十二位作为权限位，都展示出来。
3. 用新权限位替代原来pte对应的权限位。
4. JOS监视器：实现按向上箭头读取上一条历史命令，按下箭头回退一条命令

实现：

1. 修改readline.c函数，识别输入流中出现的向上箭头和向下箭头。在内核监视器中，输入流用“[A”标识向上箭头，“[B”标识向下箭头，因此要在输入流中监听连续字符。
2. 在monitor.c中维护存放历史命令的结构。这里采用二维数组顺序储存。注意每次获取到readline()返回的char\* buf之后，不能直接赋给历史命令数组，而必须采用strcpy()。
3. 在readline.c中，每次监听到向上箭头，则读取上一条历史命令，首先清除K>后面的所有内容，然后把上一条历史命令打印到控制台，接着手动把这条历史命令填到buf中

代码：

Monitor.c-monitor():

char\* cmd\_history[MAX\_CMD\_SIZE];

for(int i=0;i<MAX\_CMD\_SIZE;i++){

cmd\_history[i]=(char\*)(i\*100+KERNBASE);

}

int current\_cmd=0;

char \*buf;

cprintf("Welcome to the JOS kernel monitor!\n");

cprintf("Type 'help' for a list of commands.\n");

cprintf("x=%d y=%d", 3);

cprintf("\n");

check\_lab2();

while (1) {

buf = readline("K> ",cmd\_history,current\_cmd);

strcpy(cmd\_history[current\_cmd],buf);

current\_cmd++;

if (buf != NULL)

if (runcmd(buf, tf) < 0)

break;

Readline.c-readline():部分代码

while (1) {

c = getchar();

if (c < 0) {

cprintf("read error: %e\n", c);

return NULL;

} else if ((c == '\b' || c == '\x7f') && i > 0) {

if (echoing)

cputchar('\b');

i--;

} else if (c >= ' ' && i < BUFLEN-1) {

if(c=='['&&flag==0){flag=1;continue;}

if(c=='A'&&flag==1){

flag=0;

if(count<=0)continue;

if(count<current\_cmd){

int back\_len=strlen(cmd\_history[count]);

for(int j=0;j<back\_len;j++){

cputchar('\b');

i--;

}

}

char\* cmd\_string=cmd\_history[--count];

cprintf("%s",cmd\_string);

for(int j=0;j<strlen(cmd\_string);j++){

buf[i++] = cmd\_string[j];

}

continue;

}

if(c=='B'&&flag==1){

flag=0;

if(count<0)continue;

if(count>=current\_cmd)continue;

int back\_len=strlen(cmd\_history[count]);

for(int j=0;j<back\_len;j++){

cputchar('\b');

i--;

}

char\* cmd\_string=cmd\_history[++count];

cprintf("%s",cmd\_string);

for(int j=0;j<strlen(cmd\_string);j++){

buf[i++] = cmd\_string[j];

}

continue;

}

if(flag==1){flag=0;}

if (echoing)

cputchar(c);

buf[i++] = c;

} else if (c == '\n' || c == '\r') {

if (echoing)

cputchar('\n');

buf[i] = 0;

return buf;

}

}

1. 连续分配多个页面的malloc()，实现思路

——在struct pageInfo中新增元素 pp\_size，用来表示和该页一起被绑定分配的页面的个数。

——在分配时，首先算出需要分配的页面数量，假设是3页。然后分配连续的3页给该地址，其中每页的pp\_size都设为3。只把头尾页的pp\_link设为NULL，并把头页返回给用户。

——释放时，只需要知道头页地址和pp\_size，就得到了连续页面的所有地址，然后依次释放页面。