JOS 2016 Challenge

陈一茹 1400012976

December 7, 2016

Contents

1 Introduction

这个JOS的所有challenge的合并。总共做了6个challenge。其中lab4 Fine-gained lock是LevelA。

2 lab 1 Console color printing

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret ANSI escape sequences embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

- 之前提到，控制台打印字符的时候高 8 位为颜色,低 8 位为字符，为了实现不同的颜色， if (!(c & 0xFF)) c |= COLOR; 我们希望能够在格式字符串 fmt 中指定不同的字符串打印不同 的颜色,所以这就需要改进一下 fmt 的格式。

我上网查了查，C 语言有这样的 \033[字背景颜色;字体颜色 字符串的用法，我就模仿一个，写一下

主要就是改一下fmt格式，我定义了一个#include <inc/color.h>的头文件

```
1     int FG_COLOR;
2     int BG_COLOR;
3     int COLOR;
```

只有三行，用来定义一下背景色和字体色和合起来的颜色。

有了这些变量， 我接着在printfmt.c里面添加了对于颜色的识别和赋值语句，如下

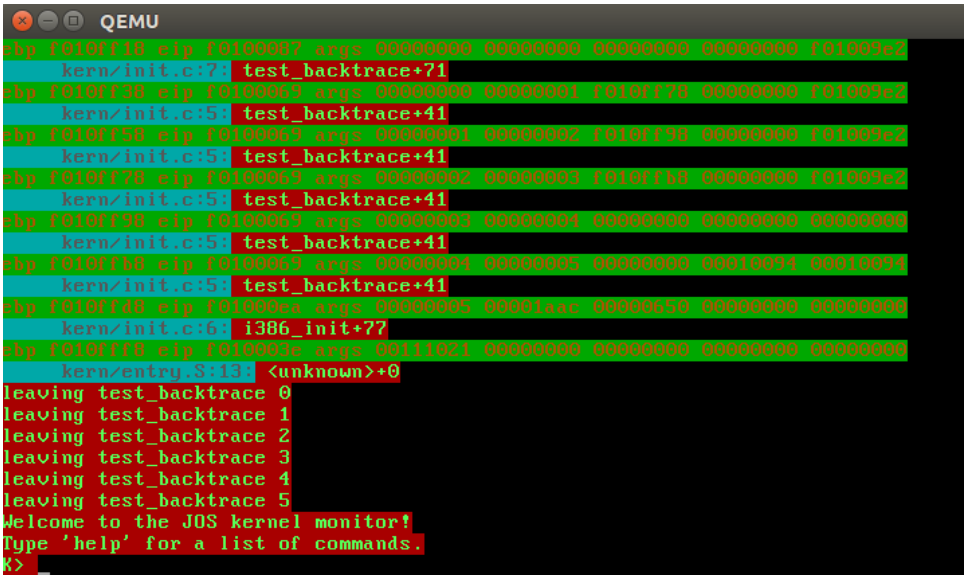
```
1     while (1) {
2     while ((ch = *(unsigned char *) fmt++) != '%') {
3     if (ch == '\0')
4     return;
5     else if(ch == '\033'){
6     if((ch = *(unsigned char *) fmt++) != '[') {
7     putch(ch, putdat);
8     continue;
9     }
10    BG_COLOR = *(unsigned char *) fmt++;
11    FG_COLOR = *(unsigned char *) fmt++;
12
13    if(BG_COLOR >= '0' && BG_COLOR <= '9')
14    BG_COLOR -= '0';
15    else if(BG_COLOR >= 'a' && BG_COLOR <= 'f')
16    BG_COLOR = BG_COLOR - 'a' + 10;
17    else if(BG_COLOR >= 'A' && BG_COLOR <= 'F')
18    BG_COLOR = BG_COLOR - 'A' + 10;
```

```
19      else BG_COLOR = 0;
20
21      if(FG_COLOR >= '0' && FG_COLOR <= '9')
22      FG_COLOR -= '0';
23      else if(FG_COLOR >= 'a' && FG_COLOR <= 'f')
24      FG_COLOR = FG_COLOR - 'a' + 10;
25      else if(FG_COLOR >= 'A' && FG_COLOR <= 'F')
26      FG_COLOR = FG_COLOR - 'A' + 10;
27      else BG_COLOR = 7;
28
29      COLOR = (BG_COLOR << 12) | (FG_COLOR << 8);
30      continue;
31      }
32      putch(ch, putdat);
33      }
```

如果有 \033[字背景颜色;字体颜色, 就会显示用户自定义的颜色，没有写的话，紧跟上文的颜色。

比如说，我对mon_backtrace()附这样的值，效果是这样的

```
1       cprintf("\033[26ebp %08x eip %08x args %08x %08x %08x %08x
            %08x\n",ebp,eip,ary[0],ary[1],ary[2],ary[3],ary[4]);
2       cprintf("\033[38  %s:%d:", eip_info.eip_file, eip_info.
            eip_line);
3       cprintf("\033[4a %.*s+%d\n", eip_info.eip_fn_namelen,
            eip_info.eip_fn_name, eip - eip_info.eip_fn_addr);
```



这样就完成了challenge啦!

3 lab 2 Monitor debugging commands

先做一个想做的challenge，其他的看时间吧，这个challenge能帮我调试，而且对前面一个question解答也能起到帮助。

Challenge! Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter `'showmappings 0x3000 0x5000'` to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

这个challeng 有三个任务：

1．实现了 pgmap 指令，用于查看虚拟地址到物理地址的映射

2．实现了 pgperm 指令，用于修改页表的权限，可添加、删除、修改

3. 实现了 memdump 指令，用于观察内存中的内容，可接受虚拟地址或物理地址

不管其他，先给monitor.c 加个头文件 #include ¡kern/pmap.h¿

```
1     int
2     mon_pgmap(int argc, char **argv, struct Trapframe *tf)
3     {
4     uintptr_t va1, va2, va;
5     struct PageInfo *pg;
6     pte_t *pte;
7     if (argc != 3) {
8     cprintf("Usage: pgmap va1 va2\n Display physical memory
          mapping from virtual memory va1 to va2\nva1 and va2 are
           hex\n");
9     return 0;
10    }
11    else {
12    for (va1 = strtol(argv[1], 0, 16), va2 = strtol(argv[2], 0,
           16); va1 < va2; va1 += PGSIZE) {
13    va = va1 & ~0xfff;
14    pg = page_lookup(kern_pgdir, (void*)va, 0);
15    pte = pgdir_walk(kern_pgdir, (void* )va,0);
16    if (pg){
```

```
17      cprintf("[%x, %x) ---> [%x, %x)     ", va, va + PGSIZE,
            page2pa(pg), page2pa(pg) + PGSIZE);
18      if(*pte & PTE_U)
19      cprintf("user: ");
20      else
21      cprintf("kernel: ");
22
23      if(*pte &PTE_W)
24      cprintf("read/write ");
25      else
26      cprintf("read only ");
27      }else
28      cprintf("[%x, %x) ---> NULL     ", va, va + PGSIZE);
29
30      cprintf("\n");
31      }
32      }
33      return 0;
34      }
```

```
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    kernel: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
K> pgmap 0xefff8000 0xf0000000
[efff8000, efff9000) ---> [10e000, 10f000)    kernel: read/write
[efff9000, efffa000) ---> [10f000, 110000)    kernel: read/write
[efffa000, efffb000) ---> [110000, 111000)    kernel: read/write
[efffb000, efffc000) ---> [111000, 112000)    kernel: read/write
[efffc000, efffd000) ---> [112000, 113000)    kernel: read/write
[efffd000, efffe000) ---> [113000, 114000)    kernel: read/write
[efffe000, effff000) ---> [114000, 115000)    kernel: read/write
[effff000, f0000000) ---> [115000, 116000)    kernel: read/write
K> pgmap 0xef000000 0xef008000
[ef000000, ef001000) ---> [11a000, 11b000)    user: read only
[ef001000, ef002000) ---> [11b000, 11c000)    user: read only
[ef002000, ef003000) ---> [11c000, 11d000)    user: read only
[ef003000, ef004000) ---> [11d000, 11e000)    user: read only
[ef004000, ef005000) ---> [11e000, 11f000)    user: read only
[ef005000, ef006000) ---> [11f000, 120000)    user: read only
[ef006000, ef007000) ---> [120000, 121000)    user: read only
[ef007000, ef008000) ---> [121000, 122000)    user: read only
```

实现出来的效果是这样的，这几个地址是前面问题中的，先打印出来，验证一下前面的答案。

```
1       int
2       mon_pgperm(int argc, char **argv, struct Trapframe *tf)
3       {
4       uintptr_t va, perm;
5       if (argc != 4) {
```

```
6       cprintf("Usage: pgperm +/-/= perm va\nset perm of page
            which contains va, va is hex\n");
7       return 0;
8       }
9       else {
10      va = strtol(argv[3], 0, 16);
11      perm = strtol(argv[2], 0, 16);
12      pte_t *pte = pgdir_walk(kern_pgdir, (void*)va, 0);
13      if (!pte) {
14      cprintf("0x%x is not mapped\n", va);
15      }
16      else {
17      if (argv[1][0] == '+') *pte |= perm;
18      if (argv[1][0] == '0') *pte &= ~perm;
19      if (argv[1][0] == '=') *pte = PTE_ADDR(*pte) | perm;
20      }
21      }
22      return 0;
23      }
```

实现的过程很简单，下面展示一下实现过后的效果图。

```
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    kernel: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
K> pgperm + 0x004 0xf0002222
K> pgmap 0xf0000000 0xf0004000
[f0000000, f0001000) ---> [0, 1000)    kernel: read/write
[f0001000, f0002000) ---> [1000, 2000)    kernel: read/write
[f0002000, f0003000) ---> [2000, 3000)    user: read/write
[f0003000, f0004000) ---> [3000, 4000)    kernel: read/write
```

用于观察内存中的内容，可接受虚拟地址区间或物理地址区间

```
1
2       int
3       mon_memdump(int argc, char **argv, struct Trapframe *tf)
4       {
5       uintptr_t a1, a2, a;
6       struct PageInfo *pg;
7       if (argc != 4) {
8       cprintf("Usage: memdump p/v a1 a2\n Dump memory content via
            virtual or physical address\na1 and a2 are hex\n");
9       return 0;
10      }
11      else {
12      a1 = strtol(argv[2], 0, 16), a2 = strtol(argv[3], 0, 16);
13      if (argv[1][0] == 'p') a1 = (int)KADDR(a1), a2 = (int)KADDR
            (a2);
14      for (a = a1; a < a2 && a >= KERNBASE; a += 4) {
```

```
15      if (!((a - a1) & 0xf)) cprintf("\n%x:\t", a);
16      cprintf(" %x", *(int*)(a));
17      }
18      cprintf("\n");
19      }
20      return 0;
21      }
```

```
memdump - Dump the contents of a range of memory
K> memdump v 0xf0003000 0xf0003070

f0003000:             97979797 97979797 97979797 97979797
f0003010:             97979797 97979797 97979797 97979797
f0003020:             97979797 97979797 97979797 97979797
f0003030:             97979797 97979797 97979797 97979797
f0003040:             97979797 97979797 97979797 97979797
f0003050:             97979797 97979797 97979797 97979797
f0003060:             97979797 97979797 97979797 97979797
K> memdump p 0x00003000 0x00003070

f0003000:             97979797 97979797 97979797 97979797
f0003010:             97979797 97979797 97979797 97979797
f0003020:             97979797 97979797 97979797 97979797
f0003030:             97979797 97979797 97979797 97979797
f0003040:             97979797 97979797 97979797 97979797
f0003050:             97979797 97979797 97979797 97979797
f0003060:             97979797 97979797 97979797 97979797
```

这是效果图，非常的完美啊。

4 lab 3 Clean up similar code

Challenge! You probably have a lot of very similar code right now, between the lists of `TRAPHANDLER` in `trapentry.S` and their installations in `trap.c`. Clean this up. Change the macros in `trapentry.S` to automatically generate a table for `trap.c` to use. Note that you can switch between laying down code and data in the assembler by using the directives `.text` and `.data`.

```
1       #define TRAPHANDLER(name, num)                  \
2       .text;                  \
3       .globl name;    /* define global symbol for 'name' */ \
4       .type name, @function;  /* symbol type is function */   \
5       .align 2;   /* align function definition */   \
6       name:     /* function starts here */    \
7       .if !(num == 8 || num >= 10 && num <= 14 || num == 17); \
```

```
8       pushl $0; \
9       .endif; \
10      pushl $(num);             \
11      jmp _alltraps;              \
12      .data;                \
13      .long name;
```

这里将TRAPHANDLER_NOEC和TRAPHANDLER合并起来，判断一下他是否有错误码。

然后将之后的统一起来：

```
1       .data
2       .global vectors
3       vectors:
4       TRAPHANDLER(vector0, 0)
5       TRAPHANDLER(vector1, 1)
6       TRAPHANDLER(vector2, 2)
7       TRAPHANDLER(vector3, 3)
8       TRAPHANDLER(vector4, 4)
9       TRAPHANDLER(vector5, 5)
10      TRAPHANDLER(vector6, 6)
11      TRAPHANDLER(vector7, 7)
12      TRAPHANDLER(vector8, 8)
13      TRAPHANDLER(vector9, 9)
14      TRAPHANDLER(vector10, 10)
15      TRAPHANDLER(vector11, 11)
16      TRAPHANDLER(vector12, 12)
17      TRAPHANDLER(vector13, 13)
18      TRAPHANDLER(vector14, 14)
19      TRAPHANDLER(vector15, 15)
20      TRAPHANDLER(vector16, 16)
21      TRAPHANDLER(vector17, 17)
22      TRAPHANDLER(vector18, 18)
23      TRAPHANDLER(vector19, 19)
24      TRAPHANDLER(vector20, 20)
25      TRAPHANDLER(vector21, 21)
26      TRAPHANDLER(vector22, 22)
27      TRAPHANDLER(vector23, 23)
28      TRAPHANDLER(vector24, 24)
29      TRAPHANDLER(vector25, 25)
30      TRAPHANDLER(vector26, 26)
31      TRAPHANDLER(vector27, 27)
32      TRAPHANDLER(vector28, 28)
33      TRAPHANDLER(vector29, 29)
34      TRAPHANDLER(vector30, 30)
35      TRAPHANDLER(vector31, 31)
36      TRAPHANDLER(vector32, 32)
37      TRAPHANDLER(vector33, 33)
38      TRAPHANDLER(vector34, 34)
39      TRAPHANDLER(vector35, 35)
40      TRAPHANDLER(vector36, 36)
41      TRAPHANDLER(vector37, 37)
42      TRAPHANDLER(vector38, 38)
```

```
43      TRAPHANDLER(vector39, 39)
44      TRAPHANDLER(vector40, 40)
45      TRAPHANDLER(vector41, 41)
46      TRAPHANDLER(vector42, 42)
47      TRAPHANDLER(vector43, 43)
48      TRAPHANDLER(vector44, 44)
49      TRAPHANDLER(vector45, 45)
50      TRAPHANDLER(vector46, 46)
51      TRAPHANDLER(vector47, 47)
52      TRAPHANDLER(vector48, 48)
```

这个研究了一下，C语言中并不能循环写。 其余的不变，trap.c这样写

```
1       // LAB 3: Your code here.
2       for (i = 0; i <= 0x30; ++i) {
3       switch (i) {
4       case T_BRKPT:
5       case T_SYSCALL:
6       SETGATE(idt[i], 0, GD_KT, vectors[i], 3);
7       break;
8       default:
9       SETGATE(idt[i], 0, GD_KT, vectors[i], 0);
10      }
11      }
```

5 lab3 Breakpoint and disassembler

Challenge! Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the `int3`, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the EFLAGS register in order to implement single-stepping.

这个通过修改tf中的eflags的陷入位FL_TF实现stepi和continue功能，使用env_pop_tf(tf)返回程序继续运行。

代码如下

kern/monitor.c

```
1       int
2       mon_continue(int argc, char **argv, struct Trapframe *tf)
3       {
4       if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
5       tf->tf_eflags &= ~FL_TF;
6       env_pop_tf(tf);
7       }
8       return 0;
9       }
10
11      int
```

```
12      mon_stepins(int argc, char **argv, struct Trapframe *tf)
13      {
14      if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
15      tf->tf_eflags |= FL_TF;
16      env_pop_tf(tf);
17      }
18      return 0;
19      }
```

添加commands[]

```
1       { "si", "single-step one instruction at a time",
            mon_stepins },
2       { "c", "continue", mon_continue },
```

kern/trap.c/trap_dispatch()中也加上：

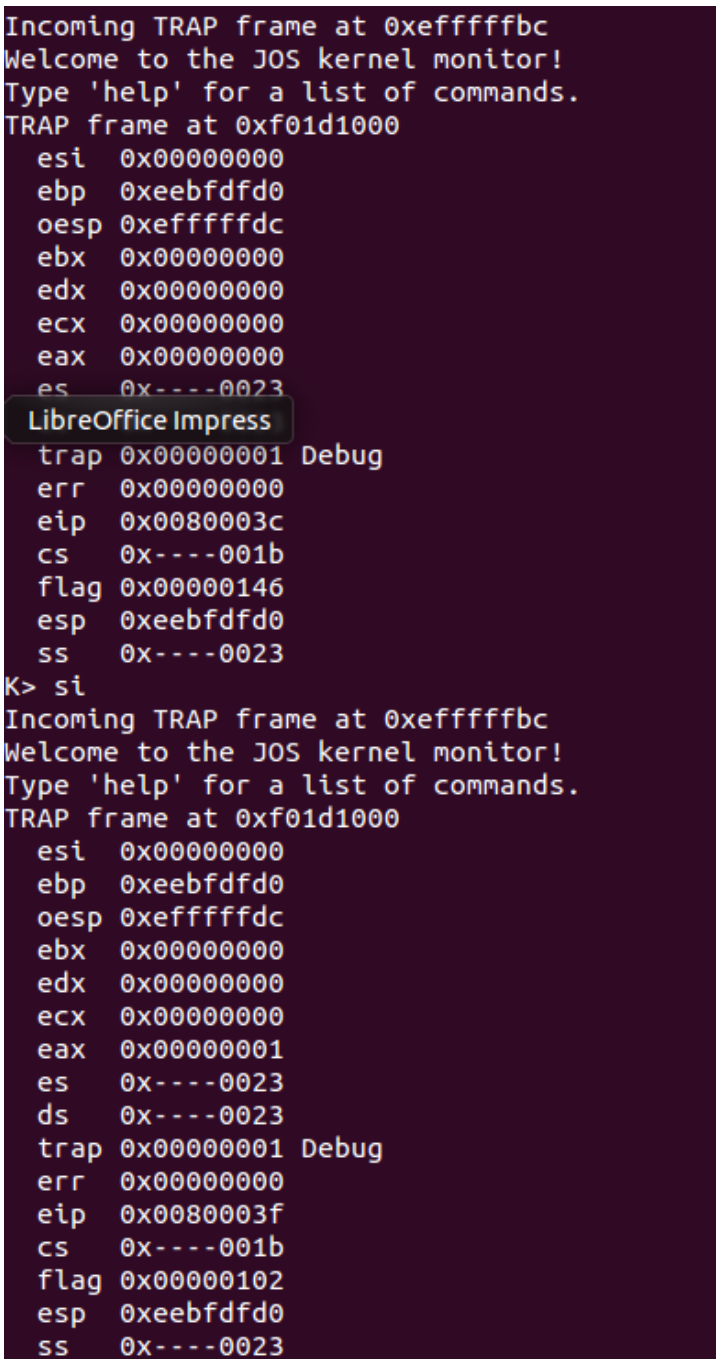
```
1       case T_DEBUG:
2       case T_BRKPT:
3       monitor(tf);
4       break;
```

这样就好了，测试一下：

在user/breakpoint.c进行测试，实测每次%eax会增加1，并且continue功能也正常。

```
1       asm volatile("movl $0, %eax");
2       asm volatile("addl $1, %eax");
3       asm volatile("addl $1, %eax");
4       asm volatile("addl $1, %eax");
5       asm volatile("addl $1, %eax");
6       asm volatile("addl $1, %eax");
```

这是成果图：



6 lab4 Fine-gained lock(LevelA)

Challenge! The big kernel lock is simple and easy to use. Nevertheless, it eliminates all concurrency in kernel mode. Most modern operating systems use different locks to protect different parts of their shared state, an approach called *fine-grained locking*. Fine-grained locking can increase performance significantly, but is more difficult to implement and error-prone. If you are brave enough, drop the big kernel lock and embrace concurrency in JOS!

It is up to you to decide the locking granularity (the amount of data that a lock protects). As a hint, you may consider using spin locks to ensure exclusive access to these shared components in the JOS kernel:

- The page allocator.
- The console driver.
- The scheduler.
- The inter-process communication (IPC) state that you will implement in the part C.

这里我实现了5类锁:

- page_lock 对pmap.c文件中所有函数实现互斥的锁。
- console_lock 同一时刻只能有一个进程调用读字符和 输出行。
- env_lock 锁struct Env *env_free_list
- Env结构体中的锁， struct spinlock lock;每Env结构体中的每一条信息在同一时刻只能有一个进程能访问
- monitor_lock 只能有一个进程运行到monitor 中。

新定义的变量和函数： kern/spinlock.h

```
1     //Challenge
2 extern struct spinlock page_lock, console_lock, env_lock,
      monitor_lock;
3     static inline void lock(struct spinlock* spl)
4 {
5     spin_lock(spl);
6 }
7
8 static inline void unlock(struct spinlock* spl)
9 {
10    spin_unlock(spl);
11    asm volatile("pause");
12 }
```

inc/env.h新定义一个自旋锁 和bool变量env_in_kernel

```
1 struct Env {
2         struct Trapframe env_tf;        // Saved registers
3         struct Env *env_link;           // Next free Env
```

```
4         envid_t env_id;                 // Unique environment
              identifier
5         envid_t env_parent_id;          // env_id of this env's
               parent
6         enum EnvType env_type;          // Indicates special
              system environments
7         unsigned env_status;            // Status of the
              environment
8         uint32_t env_runs;              // Number of times
              environment has run
9         int env_cpunum;                 // The CPU that the env
               is running on
10
11        // Address space
12        pde_t *env_pgdir;               // Kernel virtual
              address of page dir
13
14        // Exception handling
15        void *env_pgfault_upcall;       // Page fault upcall
              entry point
16
17        // Lab 4 IPC
18        bool env_ipc_recving;           // Env is blocked
              receiving
19        void *env_ipc_dstva;            // VA at which to map
              received page
20        uint32_t env_ipc_value;         // Data value sent to
              us
21        envid_t env_ipc_from;           // envid of the sender
22        int env_ipc_perm;               // Perm of page mapping
               received
23
24        struct spinlock lock;    // challenge
25        bool env_in_kernel;     // challenge
26 };
```

struct Env中的env_in_kernel记录了进程是否陷入了内核中

初始化锁

在kern/init.c:i386_init()初始化锁

```
1     //Challenge
2     spin_initlock(&page_lock);
3     spin_initlock(&console_lock);
4     spin_initlock(&env_lock);
5     spin_initlock(&monitor_lock);
```

在kern/env.c中初始化锁

```
1     void
2 env_init(void)
3 {
4     // Set up envs array
5     // LAB 3: Your code here.
```

```
6     int i;
7     env_free_list = NULL;
8     for (i = NENV - 1; i >= 0; i--) {
9         envs[i].env_id = 0;
10        envs[i].env_link = env_free_list;
11        spin_initlock(&envs[i].lock);       //initiate the lock
              Challenge
12        env_free_list = &(envs[i]);
13    }
14
15    // Per-CPU part of the initialization
16    env_init_percpu();
17 }
```

跟page_lock有关的锁都很简单， 就是成对出现的，在pmap.c中的函数出现之前会加锁，用完之后会解锁。因为有很多地方用到，就不一一贴代码了。

console_lock是关于输入输出流的锁，一共有两个地方用了这个锁：

kern/printf.c

```
1 int
2 vcprintf(const char *fmt, va_list ap)
3 {
4         int cnt = 0;
5
6         lock(&console_lock);
7         vprintfmt((void*)putch, &cnt, fmt, ap);
8         unlock(&console_lock);
9         return cnt;
10 }
```

kern/syscall.c

```
1 // Read a character from the system console without blocking.
2 // Returns the character, or 0 if there is no input waiting.
3     static int
4 sys_cgetc(void)
5 {
6     lock(&console_lock);
7     int c = cons_getc();
8     unlock(&console_lock);
9     return c;
10 }
```

env_lock 锁的是struct Env *env_free_list, 在alloc进程和free进程的时候加锁：

env_alloc()：

```
1    lock(&env_lock);
2     if (!(e = env_free_list)) {
3         unlock(&env_lock);
4         return -E_NO_FREE_ENV;
```

```
5     }
6     lock(&e->lock);
7   ......
8     // commit the allocation
9     env_free_list = e->env_link;
10    *newenv_store = e;
11
12    unlock(&env_lock);
```

env_free() :

```
1
2     // return the environment to the free list
3     e->env_status = ENV_FREE;
4     lock(&env_lock);
5     e->env_link = env_free_list;
6     env_free_list = e;
7     unlock(&env_lock);
```

monitor_lock: 在进入monitor()之前加锁，在出monitor()或从中env_pop_tf()前解锁

```
1     void
2 monitor(struct Trapframe *tf)
3 {
4     char *buf;
5
6     lock(&monitor_lock);
7
8     cprintf("Welcome to the JOS kernel monitor!\n");
9     cprintf("Type 'help' for a list of commands.\n");
10
11    if (tf != NULL)
12        print_trapframe(tf);
13
14    while (1) {
15        buf = readline("K> ");
16        if (buf != NULL)
17            if (runcmd(buf, tf) < 0)
18                break;
19    }
20    unlock(&monitor_lock);
21 }
```

```
1
2 extern void env_pop_tf(struct Trapframe *tf);
3     int
4 mon_continue(int argc, char **argv, struct Trapframe *tf)
5 {
6     if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
7         tf->tf_eflags &= ~FL_TF;
8         unlock(&monitor_lock);
```

```
9           env_pop_tf(tf);
10      }
11      return 0;
12  }
```

```
1       int
2   mon_stepins(int argc, char **argv, struct Trapframe *tf)
3   {
4       if (tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG) {
5           tf->tf_eflags |= FL_TF;
6           unlock(&monitor_lock);
7           env_pop_tf(tf);
8       }
9       return 0;
10  }
```

最后也是这个challenge 最复杂的， env 结构体的加锁和解锁：

我制定一个规则：调用者为调用的函数涉及的变量加锁， 一般也是调用者解锁，但是特殊的函数会在env_pop_tf 或者env_destroy解锁。

还有一个例外就是envid2env的时候函数返回的结构体是已经加锁的。

以sched_yield()为例，在读取某个进程信息前加锁，在结束读取后解锁，但进入env_run()时处于加锁状态：

```
1
2       void
3   sched_yield(void)
4   {
5       struct Env *idle;
6
7       int i;
8       idle = (curenv ? curenv + 1 : envs);
9
10      for (i = 0; i < NENV; i += 1, idle += 1)
11      {
12          if (idle >= envs + NENV) idle = envs;
13          lock(&idle->lock);
14          if (!idle->env_in_kernel && idle->env_status ==
                ENV_RUNNABLE)
15              env_run(idle);
16          unlock(&idle->lock);
17      }
18      if (curenv != NULL) {
19        lock(&curenv->lock);
20          if(curenv->env_status==ENV_RUNNING)
21              env_run(curenv);
22      }
23      // sched_halt never returns
24      sched_halt();
25  }
```

这里加一层判断，!idle-¿env_in_kernel只有这个时候才会去调度这个进程，经试验我发现这是有必要的： 不妨考虑一下情况：

CPU 1 上运行一个进程执行recv，然后 CPU2 运行send,在 recv 将进程设成NOT_RUNNABLE, 然后就被调度下CPU，系统调用并没有完成，还没有保存 %eax， 这时候，send 将该进程改成RUNNABLE， 进行下一次调度的时候，这个进程就有可能被调度到其他的CPU上，执行env_pop_tf(),将得到错误的%eax码，这就产生了问题。所以我们不得不再加一个env的状态号，记录这个进程是不是在内核中，如果是的话，就不允许其他CPU 调度这个进程.

env_run()，注意避免重复加锁，在env_pop_tf()中解锁：

```
1
2     void
3 env_run(struct Env *e)
4 {
5     if (curenv && e != curenv) {
6         lock(&curenv->lock);
7         if (curenv->env_status == ENV_RUNNING)
8             curenv->env_status = ENV_RUNNABLE;
9         curenv->env_in_kernel = 0;
10        unlock(&curenv->lock);
11    }
12    curenv = e;
13    curenv->env_status = ENV_RUNNING;
14    curenv->env_runs++;
15    lcr3(PADDR(e->env_pgdir));
16
17    env_pop_tf(&e->env_tf);
18 }
19   void
20 env_pop_tf(struct Trapframe *tf)
21 {
22    // Record the CPU we are running on for user-space
          debugging
23    curenv->env_cpunum = cpunum();
24    if ((curenv->env_tf.tf_cs & 3) == 3) {
25        curenv->env_in_kernel = 0;
26    }
27    unlock(&curenv->lock);
28
29    asm volatile(
30            "\tmovl %0,%%esp\n"
31            "\tpopal\n"
32            "\tpopl %%es\n"
33            "\tpopl %%ds\n"
34            "\taddl $0x8,%%esp\n" /* skip tf_trapno and
                  tf_errcode */
35            "\tiret\n"
36            : : "g" (tf) : "memory");
37    panic("iret failed");  /* mostly to placate the compiler */
38 }
```

env_destroy中解锁：

```
1
2     void
3 env_destroy(struct Env *e)
4 {
5     // If e is currently running on other CPUs, we change its
          state to
6     // ENV_DYING. A zombie environment will be freed the next
          time
7     // it traps to the kernel.
8     if (e->env_status == ENV_RUNNING && curenv != e) {
9         e->env_status = ENV_DYING;
10        return;
11    }
12
13    env_free(e);
14
15    if (curenv == e) {
16        curenv = NULL;
17        unlock(&e->lock);  // challenge
18        sched_yield();
19    }
20 }
```

envid2env中加了锁之后，在返回前不会去解锁

```
1     int
2 envid2env(envid_t envid, struct Env **env_store, bool checkperm
      )
3 {
4     struct Env *e;
5
6     if (envid == 0) {
7         *env_store = curenv;
8         return 0;
9     }
10    e = &envs[ENVX(envid)];
11
12    lock(&e->lock);
13    if (e->env_status == ENV_FREE || e->env_id != envid) {
14        *env_store = 0;
15        unlock(&e->lock);
16        return -E_BAD_ENV;
17    }
18
19    if (checkperm && e != curenv && e->env_parent_id != curenv
          ->env_id) {
20        *env_store = 0;
21        unlock(&e->lock);
22        return -E_BAD_ENV;
23    }
24
25    *env_store = e;
```

```
26      return 0;
27  }
```

7 lab5 Unix-style exec

Challenge! Implement Unix-style exec.

这里要我来实现一个Unix style的exec 。

为什么spawn 会比 Unix-style exec 要容易实现?

注意到执行 spawn 的时候，实际上当前进程 fork 了一个子进程，然后将需要执行的 elf 文 件导入至子进程。而 Unix style exec 是 replaces the current process image with a new process image. 是不通过 fork 的。这就是两者的区别，Unix-style exec 是比较困难实现的，因为很可能 需要执行的elf文件需要存放的虚拟内存位置会和当前进程的代码和数据冲突，而导致崩溃。

那么真实的操作系统是如何做到的呢?通过动态链接，这样用户库的 exec 就不会被读入的静态数据和代码而覆盖掉。

JOS里可以采用一个小trick，而不用去写动态链接。 lab4 中 pgfault 里面 copy-on-write 的时候通过临时页表进行保存。因此我们可以先把 elf 文件都存放在别的页面， 等陷入 kernel AS里面再移送到正确的位置。于是我很邪恶地将某一块地址 0x80000000 开始的地方 作为临时的缓冲区域。

大致流程就是 exec 先将 elf 文件以及新的栈内容拷贝到临时内存中作为缓冲，再引发一个 system call 来实现将临时内存的内容拷贝至真是的地址上。因为在系统调用中处于内核，因此 不用担心新的代码和数据将原用户代码和数据覆盖掉。

在 spawn.c 中设置 exec 和 execl 函数, 跟之前的spawn和spawnl是类似的，稍作修改行了。注意这里面不用新建一个进程。

然后还有就是写一个系统调用，实现把该进程从0x80000000的位置map到正确的位置，再执行。

exec 和spawn对应，稍微修改一下，将其映射到0x80000000上。

```
1   int
2 exec(const char *prog, const char **argv) {
3     unsigned char elf_buf[512];
4     uintptr_t tf_esp;
5     int fd, i, r;
6     struct Elf *elf;
7     struct Proghdr *ph;
8     int perm;
9     if ((r = open(prog, O_RDONLY)) < 0)
10        return r;
11    fd = r;
12    // Read elf header
```

```
13      elf = (struct Elf*) elf_buf;
14      if (readn(fd, elf_buf, sizeof(elf_buf)) != sizeof(elf_buf)
15              || elf->e_magic != ELF_MAGIC) {
16          close(fd);
17          cprintf("elf magic %08x want %08x\n", elf->e_magic,
                ELF_MAGIC);
18          return -E_NOT_EXEC;
19      }
20      //this use the address 0x80000000
21      // Set up program segments as defined in ELF header.
22      ph = (struct Proghdr*) (elf_buf + elf->e_phoff);
23      for (i = 0; i < elf->e_phnum; i++, ph++) {
24          if (ph->p_type != ELF_PROG_LOAD)
25              continue;
26          perm = PTE_P | PTE_U;
27          if (ph->p_flags & ELF_PROG_FLAG_WRITE)
28              perm |= PTE_W;
29          if ((r = map_segment(0,ph->p_va+0x80000000, ph->p_memsz
                ,
30                               fd, ph->p_filesz, ph->p_offset, perm))
                    < 0)
31              goto error;
32      }
33      close(fd);
34      fd = -1;
35      // Set up Stack
36      if ((r = init_stack(0, argv, &tf_esp)) < 0)
37          return r;
38      if (sys_exec(elf->e_entry, tf_esp, (void *)(elf_buf + elf->
            e_phoff), elf-> e_phnum) < 0)
39          goto error;
40      return 0;
41  error:
42      sys_env_destroy(0);
43      close(fd);
44      return r;
45  }
```

几乎和spawnl一样

```
1    // exec, taking command-line arguments array directly on the
         stack.
2   // NOTE: Must have a sentinal of NULL at the end of the args
3   // (none of the args may be NULL).
4       int
5   execl(const char *prog, const char *arg0, ...)
6   {
7       // We calculate argc by advancing the args until we hit
            NULL.
8       // The contract of the function guarantees that the last
9       // argument will always be NULL, and that none of the other
10      // arguments will be NULL.
11      int argc=0;
12      va_list vl;
```

```
13      va_start(vl, arg0);
14      while(va_arg(vl, void *) != NULL)
15          argc++;
16      va_end(vl);
17
18      // Now that we have the size of the args, do a second pass
19      // and store the values in a VLA, which has the format of
            argv
20      const char *argv[argc+2];
21      argv[0] = arg0;
22      argv[argc+1] = NULL;
23
24      va_start(vl, arg0);
25      unsigned i;
26      for(i=0;i<argc;i++)
27          argv[i+1] = va_arg(vl, const char *);
28      va_end(vl);
29      return exec(prog, argv);
30  }
```

将可执行文件map到正确的位置上，然后运行。

```
1    //lab5 challange!! Unix-like exec
2  static int
3  sys_exec(uint32_t eip, uint32_t esp, void * v_ph, uint32_t
       phnum) {
4      // set new eip and esp
5      memset((void *)(&curenv->env_tf.tf_regs), 0, sizeof(struct
           PushRegs));
6      curenv->env_tf.tf_eip = eip;
7      curenv->env_tf.tf_esp = esp;
8      int perm, i;
9
10     uint32_t now_addr = 0x80000000;
11     uint32_t now_stack = now_addr - PGSIZE;
12     uint32_t va, end_addr;
13     struct PageInfo* pg;
14
15     // Elf
16     struct Proghdr *ph = (struct Proghdr*) v_ph;
17     for (i = 0; i < phnum; i++, ph++) {
18         if (ph->p_type != ELF_PROG_LOAD)
19             continue;
20         perm = PTE_P | PTE_U;
21         if (ph->p_flags & ELF_PROG_FLAG_WRITE)
22             perm |= PTE_W;
23         // Move to real virtual address
24         end_addr = ROUNDUP(ph->p_va + ph->p_memsz, PGSIZE);
25         for (va = ROUNDDOWN(ph->p_va, PGSIZE); va != end_addr;
               va += PGSIZE)
26         {
27             if ((pg = page_lookup(curenv->env_pgdir, (void *)(
                   now_addr+va), NULL)) == NULL)
28                 return -E_NO_MEM; // no page
```

```
29              if (page_insert(curenv->env_pgdir, pg, (void *)va,
                    perm) < 0) return -E_NO_MEM;
30                  page_remove(curenv->env_pgdir, (void *)(
                        now_addr+va));
31          }
32      }
33
34      // New Stack
35      if ((pg = page_lookup(curenv->env_pgdir, (void *)now_stack,
             NULL)) == NULL) return -E_NO_MEM;
36   if (page_insert(curenv->env_pgdir, pg, (void *)(USTACKTOP -
         PGSIZE), PTE_P| PTE_U|PTE_W) < 0)
37          return -E_NO_MEM;
38      page_remove(curenv->env_pgdir, (void *)now_stack);
39      env_run(curenv); // never return
40      return 0;
41  }
```

这里其他的修改细节就不展示了。

展示一下成果：

```
[00000000] new env 00001000
[00000000] new env 00001001
FS is running
FS can do I/O
Device 1 presence: 1
i am parent environment 00001001
block cache is good
superblock is good
bitmap is good
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
file rewrite is good
hello, world
i am environment 00001001
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
```

这是spawnhello.c的程序，我们可以看到，这里的进程号没有发生改变。

This completes the challenge!