# Lab3 实验报告

陈永恒 151220012

## 实验要求

1. 实现分页机制
2. 实现PCB结构体管理进程
3. 实现进程调度schedule()
4. Yield()
5. Fork()
6. Getpid()
7. Exit()
8. Sleep()

## 实验进度，遇到的问题以及解决方案

所有要求已经实现，具体如下：

1. 实现分页机制

```
>> 32 void mm_alloc(pde_t *pgdir, uint32_t va, size_t len)
   33 {
   34     struct PageInfo *p;
>> 35     uint32_t va_start = ROUNDDOWN(va, PGSIZE);
>> 36     uint32_t va_end = ROUNDUP(va+len, PGSIZE);
   37     int i;
   38 
>> 39     for (i = va_start; i < va_end; i += PGSIZE) {
>> 40         p = page_alloc(0);
>> 41         page_insert(pgdir, p, (void*)i, PTE_W | PTE_P | PTE_U);
   42         //printf("Page alloc:%x\n",i);
   43     }
   44 }
```

```
   75     void
   76 page_init(void)
   77 {
   78     // The example code here marks all physical pages as free.
   79     // However this is not truly the case.  What memory is free?
   80     //
   81     // NB: DO NOT actually touch the physical memory corresponding to
   82     // free pages!
   83     //
   84 
   85     init_kern_pgdir();
   86 
   87     unsigned long i;
   88     //int base = (EXTPHYSMEM + 4096) / PGSIZE;
   89     int base = 0x400;
   90     for (i = 0; i < base; i++) {
   91         pages[i].pp_ref = 1;
   92     }
   93     for (i = npages-1; i >= base; i--) {
   94         pages[i].pp_ref = 0;
   95         pages[i].pp_link = page_free_list;
   96         page_free_list = &pages[i];
   97     }
   98     //kern_pgdir = entry_pgdir;
   99 
  100     //boot_map_region(kern_pgdir, KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), (PTE_W | PTE_P));
  101 
  102     boot_map_region(kern_pgdir, (uintptr_t)VMEM_ADDR, ROUNDUP(SCR_SIZE, PGSIZE), (physaddr_t)VMEM_ADDR, (PTE_W | PTE_P));
  103 }
```

kernel中初始化页表，进程动态分配页表

```
   27 int main(){
   28     init_page();
   29     init_segment();
   30     init_serial();
   31     init_timer();
   32     init_idt();
   33     init_intr();
   34     set_keyboard_intr_handler(nothing2);
   35     pcb_pool_init();
```

```
142 void pcb_load(PCB* pcb, uint32_t offset){
143     struct ProgramHeader *ph, *eph;
144     unsigned char* pa, *i;
145     lcr3(PADDR(pcb -> pgdir));
146 
147     mm_alloc(pcb->pgdir, ELFADDR, PGSIZE);
148     readseg((unsigned char*)elf, 8*SECTSIZE, offset);
149 
150     ph = (struct ProgramHeader*)((char *)elf + elf->phoff);
151     eph = ph + elf->phnum;
152 
153     for(; ph < eph; ph ++) {
154         pa = (unsigned char*)ph->paddr;
155         mm_alloc(pcb->pgdir, ph->vaddr, ph->memsz);
156         readseg(pa, ph->filesz, offset+ph->off);
157         for (i = pa + ph->filesz; i < pa + ph->memsz; *i ++ = 0);
158     }
159     entry = elf->entry;
160     mm_alloc(pcb->pgdir, USTACKTOP-STACKSIZ, STACKSIZ);
161     pcb_init(pcb, USTACKTOP-0x1FF, entry, 3);
162     lcr3(PADDR(kern_pgdir));
163 }
```

2.实现PCB结构体管理进程

这次的实验我采用了链表的方式管理PCB结构体，进程有RUNNING,BLOCKED,READY三种状态

```
 9 typedef enum{
10     YIELD,
11     READY,
12     RUNNING,
13     BLOCKED,
14 }PROCESS_STATE;
15 
16 typedef struct PCB {
17     struct{
18         PROCESS_STATE ps;
19         uint32_t inuse;
20         uint32_t pid;
21         uint32_t ppid;
22         uint32_t time_lapse;
23         struct TrapFrame *tf;
24         pde_t *pgdir;
25     };
26     struct PCB *next;
27     uint8_t kern_stack[STACKSIZ];
28     uint8_t kern_stacktop[16];
29     uint8_t user_stack[USTACKSIZE];
30     uint8_t user_stacktop[16];
31     //uint8_t kstackprotect[0x10];
32 } PCB;
33 
```

```
38 void pcb_pool_init();
39 void pcb_init(PCB *p, uint32_t ustack, uint32_t entry, uint8_t privilege);
40 PCB* pcb_create();
41 void pcb_enter(PCB*);
42 void pcb_ready(PCB*);
```

```
34 extern PCB *cur_pcb;
35 extern PCB* ready_l;
36 extern PCB* block_l;
```

PS:在实现链表的时候，对于链表的操作函数如delete()和insert()我一开始选择选择的参数类型是PCB*,即指向PCB的指针．但是发现在运行的时候总是出错，后来想到是因为我用的是一个PCB*来指向PCB链表的头，如果我选择参数类型是PCB*，那么由于参数是值传递的，虽然参数指向的也是PCB链表的头，但是一些全局变量如PCB* ready_l就不会相应地进行改变，这样在再次使用的时候就会出错．解决方案是把参数类型改成PCB**

3. 实现进程调度

我采用的进程调度方案是轮回方案，每次顺序调用ready_l列表中的第一个进程．由于有init()进程：

```
21 void init(){
22   while(1){
>> 23     system_yield();
24   }
25 }
```

其保证了任何时候都能有进程可以执行(当初没有实现init的时候就会在测试进程都结束之后系统崩溃),而在有其他更有用的进程需要执行是，init的不断调用yield()可以第一时间让别的进程运行.

下面来具体看一下schedule

```
172 void schedule(){
173     while(1){
174         if(cur_pcb==NULL){
175             cur_pcb=pcb_pop(&ready_l);
176             cur_pcb->time_lapse=0;
177             cur_pcb->ps=RUNNING;
178             printf("switch to pid %d\n",cur_pcb->pid);
179             scheduler_switch(cur_pcb);
180             break;
181         }else if(cur_pcb->ps==BLOCKED){
182             pcb_enqeque(&block_l,cur_pcb);
183             cur_pcb=NULL;
184         }else if(cur_pcb->time_lapse>400||cur_pcb->ps==YIELD){
185             cur_pcb->ps=READY;
186             pcb_enqeque(&ready_l,cur_pcb);
187             cur_pcb=NULL;
188         }else{
189             break;
190         }
191     }
192 }
```

三个条件分支:

a. 如果当前没有进程执行，则在就绪队列头中取出一个进程来执行.

b. 如果当前进程被阻塞，则把它添加到阻塞队列中，把当前的进程置空，再执行调度

c. 如果当前的进程执行了yield()或执行时间超过400个时钟，就把它添加到就绪队列的对尾，把当前的进程置空，再执行调度

由于调度的存在，对时钟信号的处理也要稍作修改，每隔50个时钟信号执行一次进程调度，且被阻塞的进程在一定的时间间隔后重新回到就绪状态:

```
 9 void do_timer(){
10     ticks++;
11     cur_pcb->time_lapse++;
12     if(ticks%50==0){
13         schedule();
14     }
15
16     PCB* ptr=block_l;
17     while(ptr!=NULL){
18         if((--ptr->time_lapse)<=0){
19             ptr->ps=READY;
20             pcb_del(&block_l,ptr);
21             pcb_enqeque(&ready_l,ptr);
22         }
23         ptr=ptr->next;
24     }
25 }
26
```

4. yield,getpid,sleep,exit

   这几个函数相对简单，在这里就不详细讲述

5. fork

   Fork的作用是创建一个和父进程一模一样的子进程．一开始我直接把PCB的结构体用memcpy复制一下，但发现系统在调度的时候就发生崩溃．

   后来思考了一段时间，再次查看PCB的定义的时候，发现了一个问题:

   PCB中的成员Trapframe以及页表基地址pgdir都只是一个指针变量．

   这说明了如果单纯地用memcpy，页表根本就没有重新分配，而Trapframe及pgdir均指向的是父进程的栈以及页表．

   于是我实现了一个fork_pgdir来复制页表:

```
362 void fork_pgdir(pde_t *dest,pde_t* src){
363     int pfirst,psecond;
364     pte_t* table_src,*table_dest;
365     struct PageInfo* ptr;
366     for(pfirst=0;pfirst<NPDENTRIES;pfirst++){
367         if(src[pfirst]&PTE_P){
368             if(dest[pfirst]&PTE_P)continue;
369             table_src=KADDR(PTE_ADDR(src[pfirst]));
370             ptr=page_alloc(ALLOC_ZERO);
371             ++ptr->pp_ref;
372             dest[pfirst]=page2pa(ptr)|PTE_ATTR(src[pfirst]);
373 
374             table_dest=KADDR(PTE_ADDR(dest[pfirst]));
375             for(psecond=0;psecond<NPTENTRIES;psecond++){
376                 if(table_src[psecond]&PTE_P){
377                     ptr=page_alloc(0);
378                     ++ptr->pp_ref;
379                     table_dest[psecond]=page2pa(ptr)|PTE_ATTR(table_src[psecond]);
380                     memcpy(page2kva(ptr),KADDR(PTE_ADDR(table_src[psecond])),PGSIZE);
381                 }
382             }
383         }
384     }
385 }
NORMAL > SPELL > +0 ~0 -0 lab1 > 1:pmap.c
1:pmap.c
```

并把Trapframe指针指向子进程的栈中正确的偏移位置，有了以下的fork():

```
194 int fork(){
195     PCB* fork_pcb=pcb_create();
196     if(fork_pcb==NULL){
197         return -1;
198     }
199     int old_pid=fork_pcb->pid;
200     pte_t* old_pte=fork_pcb->pgdir;
201     cur_pcb->tf->eax=old_pid;
202     memcpy((void*)fork_pcb,(void*)cur_pcb,sizeof(PCB));
203     fork_pcb->pgdir=old_pte;
204     fork_pgdir(fork_pcb->pgdir,cur_pcb->pgdir);
205     fork_pcb->ppid=cur_pcb->pid;
206     fork_pcb->pid=old_pid;
207     fork_pcb->tf=(struct TrapFrame*)((uint32_t)fork_pcb->kern_stack+((uint32_t)cur_pcb->tf-(uint32_t)cur_pcb->kern_stack));
208     fork_pcb->tf->eax=0;
209     pcb_ready(fork_pcb);
210     return 0;
211 }
```

可以看到通过正确的设置Trapframe中的eax来达到fork的两次返回值不一样.

以上便是这次实现的大概流程.

# 实验总结

这次实验中个人感觉最难实现的就是分页机制，虽然参考了框架代码，但是在与进程管理融合的时候还是出现了各种各样的bug.主要是:

1. 在没有进程的概念的时候，我完成了分页，系统正常运行.但是在加入了进程后，在游戏以一个新的进程加载完跳进去的时候，出现了0x13号保护错误异常
2. 我一开始以为我在分页的时候页表权限设置错误，便强行把游戏设置成内核态，结果它就能正常运行了.
3. 然而在我加入进程调度后，发现在调度时现场根本不能保护好，每一次调度都是重新运行.
4. 感到绝望
5. 后来仔细思考后，想起每次中断发生时，系统进入内核态，然而我创建的进程若是用户态，则其Trapframe还是指向用户态的栈.这样不仅特权级不正确，还会破坏进程的现场
6. 于是我在每次中断发生的时候把当前进程的Trapframe指针指向了内核的Trapframe,everything solved!!!