Jos Lab 4: Preemptive Multitasking Lab Report

王勇杰 5120379068

2014.12.7

注：**lab** 简介以及 **questions** 都在此文档中，可通过目录直接跳转。

Contexts

1. Multiprocessor Support and Cooperative Multitasking

在这一部分介绍了如何实现对多处理器的支持。

Exercise 1

> Read boot_aps() and mp_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page_init() in kern/pmap.c to avoid adding the page at MPENTRY_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated check_page_free_list() test (but might fail the updated check_kern_pgdir() test, which we will fix soon).

第一个练习主要是在初始化 freepage 链表的时候，如果物理地址在 MPENTRY_PADDR 就不能把它加入 freepage 链表中，具体只需在 page_init 时对插入的空页做一下判断，具体代码如下：

```
333     page_free_list = NULL;
334     for (i = 1; i < npages_basemem; i++)
335     {
336         pages[i].pp_ref = 0;
337         if(i == PGNUM(MPENTRY_PADDR)) continue;
338         pages[i].pp_link = page_free_list;
339         page_free_list = &pages[i];
340     }
341 
342     for (i = IOPHYSMEM/PGSIZE; i < EXTPHYSMEM/PGSIZE; ++i)
343         pages[i].pp_ref = 1;
344 
345     for (; i < PADDR(boot_alloc(0))/PGSIZE; ++i)
346         pages[i].pp_ref = 1;
347     for (; i < npages; ++i)
348     {
349         pages[i].pp_ref = 0;
350         if(i == PGNUM(MPENTRY_PADDR)) continue;
351         pages[i].pp_link = page_free_list;
352         page_free_list = &pages[i];
353     }
```

即在上面代码的 337 行与 350 行对要插入的页地址做一下判断即可。

Question 1

> Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?

MPBOOTPHYS 这个宏的作用就是将内核的 Linear Address 转化为 Physical Address。而起必须在 mpentry.S 中使用是因为 BSP 启动 AP 时候，不同于 BSP 启动时候的 KERNBASE 上 4MB 空间映射，内存地址的计算方式并不相同。

Exercise 2

> Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in our revised inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages. Your code should pass the new check in check_kern_pgdir()

这个部分完成对每个 CPU 的映射环境，根据注释计算内存地址后调用 boot_map_regin()来进行映射，具体代码如下：

```
    // LAB 4: Your code here:
    size_t i;
    for(i = 0; i < NCPU; ++i)
    {
        uint32_t kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE,
                        PADDR(percpu_kstacks[i]), PTE_W);
    }
}
```

Exercise 3

> The code in trap_init_percpu() (kern/trap.c) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs.Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.)

```
    // Get CPU ID
    int cpuID = thiscpu->cpu_id;

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - (KSTKSIZE + KSTKGAP)*cpuID;
    thiscpu->cpu_ts.ts_ss0 = GD_KD;

    // Initialize the TSS slot of the gdt.
    int gdtOffset = GD_TSS0 + (cpuID<<3);
    int gdtIndex = gdtOffset>>3;
    gdt[gdtIndex] = SEG16(STS_T32A, (uint32_t) &thiscpu->cpu_ts, sizeof
    gdt[gdtIndex].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(gdtOffset);

    // Load the IDT
    lidt(&idt_pd);
```

这个练习是为了每个 CPU 初始化 TSS 与 TSS 描述符，具体代码如上。

Exercise 4

Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

这个练习要求在一些位置应用全局锁，具体需加锁解锁的位置在 lab 介绍中也说明得很明确。

Exercise 4.1

Implement ticket spinlock in kern/spinlock.c. You can define a macro USE_TICKET_SPIN_LOCK at the beginning of kern/spinlock.h to make it work. After you correctly implement ticket spinlock and define the macro, your code should pass spinlock_test(). Don`t use ticket spinlock before you implement all the rest exercises. Ticket spinlock may cause stresssched and primes fail because of its poor performance. (Optional) Consider why ticket spinlock is slow here.

这个练习需要完成排队自旋锁，具体代码可见 spinlock.c，运行测试输出如下：

```
Pass check_page_free_list
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
spinlock_test() succeeded on CPU 0!
```

Question 2

It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

在中断发生后拿 kernel 锁之前，原来 environment 的数据会被 push 到栈上，这时候如果共享 kernel stack，就会出现保存现场状态出错。因此需要为每个 CPU 分配不同的内存栈。

Exercise 5

Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().
Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.
After the yield programs exit, when only idle environments are runnable, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

这个练习是实现 round-robin 的进程调度，具体代码如下：

```
32     if(curenv == NULL) i = 0;
33     else i = ENVX(curenv->env_id) + 1;
34 
35     int times = 0;
36     while(times < NENV){
37         int tem = (i + times) % NENV;
38         if(envs[tem].env_type != ENV_TYPE_IDLE &&
39             envs[tem].env_status == ENV_RUNNABLE){
40             env_run(envs + tem);
41         }
42         times++;
43     }
44 
45     if(curenv && curenv->env_status == ENV_RUNNING)
46         env_run(curenv);
```

首先在 32-33 行获取当前在运行的线程 id，而后在 36-43 行做 NENV 次循环，找到接下来需要运行的线程。

Question 3

In your implementation of env_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

e 指向的是内核地址空间，页表切换时不切换 kernel 的寻址空间的地址。

Exercise 6

Implement the system calls described above in kern/syscall.c. You will need to use various functions in kern/pmap.c and kern/env.c, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

这个练习需要完成一个 fork 的功能，具体实现如下：

1. sys_exofork()：

```
    struct Env* env;
    if(env_alloc(&env, thiscpu->cpu_env->env_id))
        return -E_NO_FREE_ENV;

    env->env_status = ENV_NOT_RUNNABLE;
    env->env_tf = thiscpu->cpu_env->env_tf;
    env->env_tf.tf_regs.reg_eax = 0;

    return env->env_id;
```

这个函数的作用就是复制当前的环境，返回新的环境 id。

2. sys_env_set_status()：

```
    struct Env* env;

    if(status != ENV_RUNNABLE &&
        status != ENV_NOT_RUNNABLE)
        return -E_INVAL;

    if(envid2env(envid, &env, 1))
        return -E_BAD_ENV;

    env->env_status = status;
    return 0;
```

这个函数用户设置环境状态，根据目前的 status 进行设置。

3. sys_page_alloc()：

```
    // LAB 4: Your code here.
    if((uint32_t)va>=UTOP || ROUNDUP(va, PGSIZE)!=va)
        return -E_INVAL;

    if(!(perm&PTE_U) || !(perm&PTE_P) || (perm&~PTE_SYSCALL))
        return -E_INVAL;

    struct Env* env;
    if(envid2env(envid, &env, 1)<0) return -E_BAD_ENV;

    struct Page* page;
    page = page_alloc(ALLOC_ZERO);
    if(!page) return -E_NO_MEM;

    if(page_insert(env->env_pgdir, page, va, perm)){
        page_free(page);
        return -E_NO_MEM;
    }

    return 0;
```

这个函数是为 envid 的环境申请新的物理页，并做虚拟地址的映射。

4. sys_page_map()：

```
	if((uint32_t)srcva>=UTOP ||
		ROUNDUP(srcva, PGSIZE)!=srcva ||
		(uint32_t)dstva>=UTOP ||
		ROUNDUP(dstva, PGSIZE)!=dstva){
		return -E_INVAL;
	}

	if(!(perm&PTE_U) || !(perm&PTE_P) || (perm&~PTE_SYSCALL))
		return -E_INVAL;

	struct Env* env_s, * env_d;
	if(envid2env(srcenvid, &env_s, 1)||
		envid2env(dstenvid, &env_d, 1))
		return -E_BAD_ENV;

	pte_t* pte;
	struct Page* page = page_lookup(env_s->env_pgdir, srcva, &pte);
	if(!page)
		return -E_INVAL;

	if((perm & PTE_W) && !(*pte & PTE_W))
		return -E_INVAL;

	if(page_insert(env_d->env_pgdir, page, dstva, perm))
		return -E_NO_MEM;

	return 0;
```

5. sys_page_unmap()：

```
	// LAB 4: Your code here.
	if((uint32_t)va>=UTOP ||
		ROUNDUP(va, PGSIZE)!=va)
		return -E_INVAL;

	struct Env* env;
	if(envid2env(envid, &env, 1))
		return -E_BAD_ENV;

	page_remove(env->env_pgdir, va);
	return 0;
```

2. Copy-on-Write Fork

Exercise 7

Implement the sys_env_set_pgfault_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
	// LAB 4: Your code here.
	struct Env* env;
	if(envid2env(envid, &env, 1))
		return -E_BAD_ENV;

	env->env_pgfault_upcall = func;
	return 0;

	//panic("sys_env_set_pgfault_upcall not implemented");
}
```

这个函数用于设置当发生 page fault 时用哪个函数来处理。

Exercise 8

Implement the code in page_fault_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

```
    if(curenv->env_pgfault_upcall){

        uint32_t stktop = (tf->tf_esp >= UXSTACKTOP - PGSIZE &&
                           tf->tf_esp < UXSTACKTOP) ?
                           tf->tf_esp - sizeof(struct UTrapframe) - 4 :
                           UXSTACKTOP - sizeof(struct UTrapframe);

        user_mem_assert(curenv, (void*) stktop, sizeof(struct UTrapframe), PTE_U|PTE_W);

        struct UTrapframe* utf = (struct UTrapframe*) stktop;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_eip = tf->tf_eip;
        utf->utf_err = tf->tf_err;
        utf->utf_esp = tf->tf_esp;
        utf->utf_fault_va = fault_va;
        utf->utf_regs = tf->tf_regs;

        tf->tf_eip = (uint32_t) curenv->env_pgfault_upcall;
        tf->tf_esp = stktop;

        env_run(curenv);
```

这个函数为当发生用户态下的 page fault 时，如果该进程的环境设置了 upcall，那么就将 eip 指向 upcall，并在调用页处理函数前将当前状态压入用户异常栈。

Exercise 9

Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

这个练习需要实现页错误处理函数的入口，具体实现见 lib/pfentry.S。

Exercise 10

Finish set_pgfault_handler() in lib/pgfault.c

```
    // LAB 4: Your code here.
    if(sys_page_alloc(0, (void*) (UXSTACKTOP - PGSIZE), PTE_W|PTE_U|PTE_P)){
        cprintf("lib/pgfault:set_pgfault_handler: out of memory.");
        return;
    }

    sys_env_set_pgfault_upcall(0, _pgfault_upcall);
```

Exercise 11

Implement fork, duppage and pgfault in lib/fork.c

这里需要实现 fork 的一些主要函数，其中 duppage 主要将页 pn 映射到 envid 中，pgfault 为用户页错误处理函数，fork 函数则在 exofork 的基础上做拓展，当发生 page fault 时可以由用户定义函数来处理。具体代码实现见 lib/fork.c。

3. Preemptive Multitasking and Inter-Process communication (IPC)

Exercise 12

Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.

这个练习需要初始化 IRQ，详见具体代码。

Exercise 13

Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.

这个练习要求在发生时钟中断时调用 sched_yield，具体代码如下

```
    if(tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER){
        lapic_eoi();
        sched_yield();
        return;
    }
```

Exercise 14

Implement sys_ipc_recv and sys_ipc_try_send in kern/syscall.c.

这个练习需要实现进程之间的通信，具体实现见代码部分。

4. Challenge

Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

加入了调整程序自己优先级的 system call，每次 sched_yield()中，会优先搜索最高优先级的进程运行，然后默认，然后低优先级。测试程序通过修改 user/hello.c 的代码实现。测试结果如下

```
[00001008] High Priority
[00001008] High Priority
[00001008] High Priority
[00001008] High Priority
[00001008] exiting gracefully
[00001008] free env 00001008
[00001009] new env 00002008
[0000100a] Default Priority
[0000100a] Default Priority
[0000100a] Default Priority
[0000100a] Default Priority
[0000100a] Default Priority
[0000100a] exiting gracefully
[0000100a] free env 0000100a
[00001009] Default Priority
[00001009] Default Priority
[00001009] Default Priority
[00001009] Default Priority
[00001009] Default Priority
[00001009] exiting gracefully
[00001009] free env 00001009
[00002008] Low Priority
[00002008] Low Priority
```