MIT6.828 Lab4: Preemptive Multitasking

Zhuofan Zhang

April 2020

Contents

1 Multiprocessor Support and Cooperative Multitasking **1**

1.1 Multiprocessor Support 1

1.2 Application Processor Bootstrap 2

1.3 Per-CPU State and Initialization 4

1.4 Round-Robin Scheduling 8

Chapter 1

Multiprocessor Support and Cooperative Multitasking

1.1 Multiprocessor Support

本次 Lab 的内容是对 JOS 进行补充以提供多处理器支持，并实现任务调度功能。

JOS 实现的多核支持属于对称多核支持 *(Symmertric Multiprocessing, SMP)*，即所有处理器对资源（内存、IO 总线等）的访问是平等的。SMP 的概念是在系统初始化完成后建立的，在 Boot 阶段，仍然需要选择一个 CPU 作为启动处理器 *(Bootstrap Processor, BSP)*，用来初始化资源并启动 OS，最后启动其他的 CPU（称为应用处理器 *(Application Processor, AP)*）。对 BSP 的选择是由 BIOS 完成的，在当前实验阶段，我们相当于完成了 BSP 的启动内容。

在 SMP 体系中，每个 CPU 都拥有一个称为 LAPIC(local Advanced-Programmable Interrupt Controller) 的可编程中断单元，用于在系统中传递中断信息，并提供 CPU 的唯一识别信息。因此，与多个 CPU 的通讯依赖于 LAPIC 单元。

在本次实验中我们使用了 LAPIC 的如下功能：

- 代码可以利用 APIC-id 确认自己运行于哪一个 cpu 上（见 cpunum() 的实现）
- 利用 BSP 来启动其他的 AP（见 lapic_startap() 的实现）
- 使用 LAPIC 中的时钟中断来实现抢断式调度（PartC，apic_init()）

CPU 对 LAPIC 单元的访问使用的是一段映射到 PAS 特定位置的区域：memory-mapped I/O (MMIO)。MMIO 是一段硬连接到部分 IO 设备寄存器的物理地址区域，常用于访问设备的寄存器。JOS 的 VAS 在 MMIOBASE 处留有 4MB 的空间用于映射这段物理地址区域。

Exercise 1

Implement mmio_map_region in kern/pmap.c. To see how this is used, look at the beginning of lapic_init in kern/lapic.c. You'll have to do the next exercise, too, before the tests for mmio_map_region will run.

这一个 Exercise 要求我们实现 mmio_map_region()。我们查看该函数的注释可知，它就是用来实现上文提到的 MMIO 在 VAS 中映射的函数。JOS 的 VAS 中预留了 [MMIOBASE, MMIOLIM) 可供使用。根据提示使用 boot_map_region() 并补上禁用缓存的 PWT-flag，实现函数如下：

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    static uintptr_t base = MMIOBASE;

    size = ROUNDUP(size, PGSIZE);
    if(base + size > MMIOLIM)
        panic("mmio_map_region: MMIOLIM overflow.\n");
    boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);
    base += size;
    return (void *)(base - size);

}
```

1.2 Application Processor Bootstrap

在启动 APs 前，先启动的 BSP 必须能够获取 APs 的信息，如 CPU 数量、APIC IDs 等。这些信息被保留在 BIOS 中的 MP config table 中，kern/mpconfig.c/mp_init() 将其读出。

kern/init.c/boot_aps() 是整个流程的核心函数，它将 AP 的入口程序加载到内存，并发送启动信号。AP 的入口程序被加载到任意可用的低位 640k 地址（JOS 将其加载到 MPENTRY_PADDR），它与 BSP 的入口程序有一定的区别。

boot_aps() 函数通过给每个 AP 的 LAPIC 发送 STARTUP 信号的方式唤醒它们，同时设置它们的 entry 位置，即为 AP 准备的入口程序（位于 MPENTRY_PADDR），执行完入口程序后 AP 将运行 mp_main()；与此同时，BSP 唤醒每一个 AP 时等待它的 CPU 状态被设置为 CPU_STARTED——即该 AP 成功启动后，再开始唤醒下一个 AP。

Exercise 2

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).

这一个 Exercise 的编码任务比较简单：修改我们在 Lab2 实现的 page_init 函数，将现在存放有 AP 初始代码的那个物理页从空闲列表中拿出去：

```
void
page_init(void)
{
    // LAB 4:
    // Change your code to mark the physical page at MPENTRY_PADDR
    // as in use

    ...

    // 2) Base-memory
    size_t i;
    for (i = 1; i < npages_basemem; i++) {
        // Add for Lab4
        if(i * PGSIZE == MPENTRY_PADDR)
        {
            pages[i].pp_ref = 1;
            continue;
        }

        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }

    ...

}
```

代码之外的任务是要求我们阅读 boot_aps() 及 mp_main() 源码以及 APs 的 entry 汇编代码，捋清执行流。boot_aps() 做的事情就是将 mpentry.S 搬运到前文提到的 MPENTRY_PADDR 物理页上，再循环唤醒 APs（使用 lapic_startaps()）；APs 被唤醒后从 mpentry.S 开始执行，再跳转到 mp_main() 上。

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? Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

AP 的入口程序与 BSP 的入口程序差异主要表现在：

- 使用地址时需要使用 MPBOOTPHYS-macro 进行一步转换；
- 无需进行 A20 地址线的使能操作；
- 直接在入口处打开了分页功能，并使用已经被 BSP 设置好的内核页表目录

使用 MPBOOTPHYS 是因为 mpentry.S 被链接到了内核的高地址处（above KERNBASE）, 需要进行转换。

1.3 Per-CPU State and Initialization

每个 CPU 具有自己独立的状态和信息，JOS 将各个 CPU 独立的信息定义在 kern/cpu.h 中。其中比较关键的内容有：

- **CPU 内核栈**：支持多 CPU 同时陷入内核；
- **TSS 及 TSS 描述符**：每个 CPU 拥有独立的 TSS 指明内核栈位置；
- **当前运行 Env 指针**：用于记录每个 CPU 的当前运行进程；
- **系统寄存器**：每个 CPU 需独立加载页表、GDT、LDT 等内容

本节实验的内容是对这类 per-CPU 状态进行初始化。

Exercise 3

Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in 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 Exercise 要求我们对各个内核的内核栈进行映射。BSP 在初始化 VAS 时，完成对各个 cpu 的内核栈的映射（JOS 默认支持 8 核，且不管每次实际启动的 CPU 数量，为 8 个内核栈都分配空间）。根据提示，我们知道分配给各个 cpu 内核栈的物理空间由 percpu_kstacks 数组描述（定义于 mpconfig.c 中），我们需要将 VAS 中的各个内核栈位置映射到这个数组描述的物理地址

处。同时也注意到，JOS 的 VAS 中每个内核栈也预留了一个 Guard-page，我们需要跨过这段区域进行映射。

实现内核栈映射的方法与之前 Lab2 中对 BSP 的内核栈映射方式相同，同时注意到：此处进行映射时也替换了原本 BSP 中的内核栈位置（从 bootstack 切换到了 &percpu_kstacks[0]）。

```
static void
mem_init_mp(void)
{
    ...
    // LAB 4: Your code here:
    for(int i = 0; i < NCPU; ++i)
    {
        // Map the kstk_i
        boot_map_region(
            kern_pgdir,
            KSTACKTOP - i*(KSTKSIZE + KSTKGAP) - KSTKSIZE,
            KSTKSIZE,
            PADDR(percpu_kstacks[i]),
            PTE_W
        );
    }
}
```

Exercise 4

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.)

这个 Exercise 要求我们修改 trap 初始化的部分，主要就是修改 TSS 及其 descriptor：TSS 内主要要修改的是对应的内核栈（每个 CPU 使用各自独立的内核栈），同时修改 GDT 表项（如前文所述，每个 CPU 拥有自己独立的 TSS-descriptor）。根据实验提示我们可以完成下列代码：

```
void
trap_init_percpu(void)
{
    ...
    // LAB 4: Your code here:

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    // ts.ts_esp0 = KSTACKTOP;
    // ts.ts_ss0 = GD_KD;
    // ts.ts_iomb = sizeof(struct Taskstate);
    uint8_t nowCpuId = thiscpu->cpu_id;
    struct Taskstate* nowTs = &(thiscpu->cpu_ts);

    nowTs->ts_esp0 = KSTACKTOP - nowCpuId*(KSTKSIZE + KSTKGAP);
    nowTs->ts_ss0 = GD_KD;
    nowTs->ts_iomb = sizeof(struct Taskstate);

    // Initialize the TSS slot of the gdt.
    // gdt[(GD_TSS0 >> 3)] = SEG16(STS_T32A, (uint32_t) (&ts),
    //            sizeof(struct Taskstate) - 1, 0);
    // gdt[(GD_TSS0 >> 3)].sd_s = 0;
    gdt[(GD_TSS0 >> 3) + nowCpuId] = SEG16(STS_T32A, (uint32_t) (nowTs),
                  sizeof(struct Taskstate) - 1, 0);
    gdt[(GD_TSS0 >> 3) + nowCpuId].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    // ltr(GD_TSS0);
    ltr(GD_TSS0 + (nowCpuId << 3));

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

Locking

目前我们的 mp_main() 仍处于空转状态，但在有效使用 APs 来并行运算代码前，我们必须解决并发所带来的竞争问题。JOS 解决并发访问冲突的方法是使用 big-kernel-lock（也即早期 Linux 使用的方法），任意时刻仅能容纳 1 个 user-mode 进程陷入内核。JOS 已经提前为我们实现了两个 API：lock_kernel() 和 unlock_kernel()，其内部实现是一个全局自旋锁。

根据 Lab 页面的提示，我们需要在以下部分上锁/释放锁：

- **i386_init()**： BSP 在启动 APs 前需先持有锁（该锁在后续执行流的 sched_yield() 中释放），防止与 APs 同时调度进程时出现 race-condition；
- **mp_main()**： APs 与 BSP 同理，为防止进程调度时产生 race-condition，调度前需先持锁；
- **trap()**：如前文所述，当 user-mode 进程陷入内核时需持锁，保证同一时刻仅一个进程陷入内核中；
- **env_run()**：env_run 即从内核态返回 user-mode，此时释放内核锁，让其他等待陷入的进程可以进入内核

> **Exercise 5**
>
> Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

这个练习就是要求我们实现上述内容，我们直接在对应函数中相应区域填入代码即可：

```
/**** i386_init() ****/
...
lock_kernel();
boot_aps();
...
/**** mp_main() ****/
...
    lock_kernel(); # Add in Lab4 Exercise 5
    sched_yield();
...
/**** trap() ****/
...
if ((tf->tf_cs & 3) == 3) {
    ...
    lock_kernel();
    assert(curenv);
...
/**** env_run() ****/
...
lcr3(PADDR(curenv->env_pgdir));
unlock_kernel(); # Add in Lab4 Exercise 5
env_pop_tf(&(curenv->env_tf));
```

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.

在进入内核前，实际上也有对内核栈的操作（trap 将寄存器信息压入内核栈中），此时栈处于无锁状态，有可能发生 race-condition。

1.4 Round-Robin Scheduling

本节要求我们实现最简单的 Round robin 进程调度。sched_yield() 函数负责从 envs 中挑选返回 user-mode 后运行的新进程。调度算法即 Round-robin：在循环队列中寻找当前进程（curenv）位置后方可执行的（ENV_RUNNABLE）第一个进程。

Exercise 6

Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().

Make sure to invoke sched_yield() in mp_main.

Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.

Run make qemu. You should see the environments switch back and forth between each other five times before terminating, like below.

Test also with several CPUS: *make qemu CPUS=2*.

```
...
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

After the yield programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

我们首先定位到 sched_yield() 函数。根据注释的提示，我们可以将函数实现如下（需要考虑三种调度的情况）：

```
void
sched_yield(void)
{
    struct Env *idle = NULL;
    ...
    // situation 1: no running env
    if(!curenv)
    {
        for(int i = 0; i < NENV; ++i)
        {
            if(envs[i].env_status == ENV_RUNNABLE)
            {
                idle = &envs[i];
                break;
            }
        }
    }
    // situation 2: search after not-null curenv
    else
    {
        int currentIdx = curenv - envs;
        for(int i = 1; i < NENV; ++i)
        {
            int idx = (currentIdx + i) % NENV;
            if(envs[idx].env_status == ENV_RUNNABLE)
            {
                idle = &envs[idx];
                break;
            }
        }
    }
    // situation 3: only the curenv can run
    if(!idle && curenv && curenv->env_status == ENV_RUNNING)
        idle = curenv;

    // find next-run success
    if(idle)
        env_run(idle);  // the env_run will unlock_kernel

    // sched_halt never returns
    sched_halt();
}
```

之后，我们实现提供给用户态的 sys_yield() 接口（此处从略）。为了测试结果，我们根据提示在 i386_init() 中添加创建用户进程的代码：

```
/**** i386_init() ****/
...
#else
   // Touch all you want.
   // ENV_CREATE(user_primes, ENV_TYPE_USER);
#endif // TEST*

   for(int i = 0; i < 3; ++i)
   {
      ENV_CREATE(user_yield, ENV_TYPE_USER);
   }

   // Schedule and run the first user environment!
   sched_yield();
```

在 *make qemu* 命令启用单个 CPU 运行，得到运行结果如下：

```
VNC server running on `127.0.0.1:5900'
Physical memory: 131072KB available, base = 640KB, extended = 130432KB
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
```

若以 *make qemu CPUS=2* 命令启用双核系统，得到的运行结果如下：

```
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 2 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001002, iteration 3.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
Back in environment 00001002, iteration 4.
All done in environment 00001001.
All done in environment 00001002.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001002] exiting gracefully
[00001002] free env 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
```

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?

4. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

第一个问题的答案很简单：所有的进程都映射了完整的 VAS 的内核部分，而 envs 的映射也位于内核部分中，故所有进程对于 envs 数组任一进程的地址翻译都是相同的。

第二个问题：进程的现场保存发生在 _alltrap 过程中，现场恢复发生在 env_pop_tf() 时。