

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

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

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);
}

```