MIT6.828 Lab4: Preemptive Multitasking

Zhuofan Zhang

Augest 2020

Contents

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 APIC) 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。

CPU 对 LAPIC 单元的访问使用的是一类特殊的 IO：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.

这个练习要求我们补全 mmio_map_region 函数。根据提示，这个函数用于将 MMIO 映射到上文提到的 VAS 指定区域，我们对其补全如下：

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    ...
    // Your code here:
```

```
    size = ROUNDUP(size, PGSIZE);
    if(base+size > MMIOLIM)
        panic("MMIO_MAP_REGION: out of range.");
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD|PTE_PWT|PTE_W);
    base += ROUNDUP(size, PGSIZE);
    return (void *)(base - size);
}
```

Application Processor Bootstrap

在完成对 MMIO 的映射后，可以开始对 AP 的启动。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，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 0x7000），它与 BSP 的入口程序有一定的区别。

AP 被启动时处于实模式，在经过与 BSP 相似的初始化后调用 mp_main() 进行进一步设置。

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.

init.c/i386_init() 调用 boot_aps() 用于启动 APs，这个函数再将 AP 入口加载到内存中，调用 lapic_startap() 与 APs 通讯，AP 将执行入口代码（mpentry.S），并跳转控制至 init.c/mp_main()；由于 VAS 为 MMIO 提供一段区域映射，因此在页表初始化阶段需将这段区域的页表剔除出空闲页表，因此修改 page_init() 函数：

```
void
page_init(void)
{

    size_t i;
    // for situation(1)
    pages[0].pp_ref = 1;
    // for situation(2)
    size_t pages_in_ap_entry = MPENTRY_PADDR / PGSIZE;
    for (i = 1; i < npages_basemem; i++)
    {
        // Lab 4 code: to avoid mapping at the AP entry codes.
```

```
        if(i == pages_in_ap_entry)
        {
            pages[i].pp_ref = 1;
            continue;
        }

        pages[i].pp_ref = 0;
        // At the first time, the page_free_list will be 'NULL'
        // When pp.link = NULL, it means that no more free_page after
        // this node.
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
    // for situation(3)
    ...
}
```

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?

AP 的入口程序与 BSP 的入口程序的区别主要有两点：(1)AP 入口程序不需要重复使能 A20 地址线；(2) 加载变量和跳转的地址计算模式不同，AP 入口程序使用了 MPBOOTPHYS。这个宏的目的是计算正确的物理地址（与链接地址不同，见 kernel.asm）。如果直接使用 boot.c，将加载错误的 GDT 或进行错误的跳转。

Per-CPU State and Initialization

支持多处理器的 OS 需要区分处理器私有信息与共享信息的管理。JOS 将大部分 per-CPU 信息放在 kern/cpu.h 中，其中比较重要的有：

- **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。

应当指出，每个 CPU 需要独立加载 GDT 到 GDTR(GDT Register)，但GDT 可以是共享的，JOS 采用了这种共享 GDT 的设计。在 BSP 和 APs 的启动阶段，处理器会加载各自的 Bootstrap

GDT；而在 kern/env.c 中定义了全局变量 gdt，并在 env_init_cpu() 装载到每个 CPU 的 GDTR 中。

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.

这一步是完成所有内核栈的映射，并在每个内核栈区最后保留一个不映射的 Guide Page，最终完成代码段如下：

```
static void
mem_init_mp(void)
{
    uint32_t kstacktop_i = KSTACKTOP;
    for(int i = 0; i < NCPU; ++i)
    {
        kstacktop_i = KSTACKTOP - i * (KSTKSIZE+KSTKGAP);
        boot_map_region(kern_pgdir,
                        kstacktop_i-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.

在修改前，trap_init_percpu() 只能正确初始化 BSP，要求我们将其修改为可以适用于所有 CPU 的版本。根据提示，需要注意的是：(1) 全局变量 ts 不再可使用，应从 CpuInfo 数组中找到每个 CPU 对应的 TSS；(2)GDT 中各 CPU 信息所在条目偏移量不同。修改结果如下：

```
void
trap_init_percpu(void)
{
    // LAB 4: Your code here:
    thiscpu->cpu_ts.ts_esp0 = (uintptr_t)(percpu_kstacks[cpunum()] + KSTKSIZE);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);
```

```
    gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A, (uint32_t)(&(thiscpu->cpu_ts)),
                    sizeof(struct Taskstate) - 1, 0);
    gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;

    ltr(GD_TSS0 + (cpunum() << 3));

    lidt(&idt_pd);
}
```

Locking

所有 APs 启动后将同时开始运行内核代码段，此时我们需要找到会发生竞态的代码段并解决竞态问题。JOS 使用的是名为 *big kernel lock* 的全局自旋锁，这个锁使得用户进程可以不受影响并行运行，但仅允许一个内核进程运行。

Exercise 5

In i386_init(), acquire the lock before the BSP wakes up the other CPUs.

In mp_main(), acquire the lock after initializing the AP and then call sched_yield().

In trap(), acquire the lock when trapped from user mode.

In env_run(), release the lock right before switching to user mode.

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

根据题目要求，我们对代码增加内核锁：

```
/**** i386_init() ****/
...
// Acquire the big kernel lock before waking up APs
// Your code here:
lock_kernel();
// Starting non-boot CPUs
boot_aps();
...

/**** mp_main() ****/
...
// Now that we have finished some basic setup, call sched_yield()
// to start running processes on this CPU.  But make sure that
// only one CPU can enter the scheduler at a time!
//
// Your code here:
   lock_kernel(); # Add in Lab4 Exercise 5
```

```
    sched_yield();
...

/**** trap() ****/
...
if ((tf->tf_cs & 3) == 3) {
    // Trapped from user mode.
    // Acquire the big kernel lock before doing any
    // serious kernel work.
    // LAB 4: Your code here.
    lock_kernel();
    assert(curenv);

      ...
}
...

/**** env_run() ****/
lcr3(PADDR(curenv->env_pgdir));

unlock_kernel(); # Add in Lab4 Exercise 5

env_pop_tf(&(curenv->env_tf));
```

注意到最后对 env_run() 的补全：因为 env_pop_tf() 不返回，因此锁的释放在调用该函数之前。

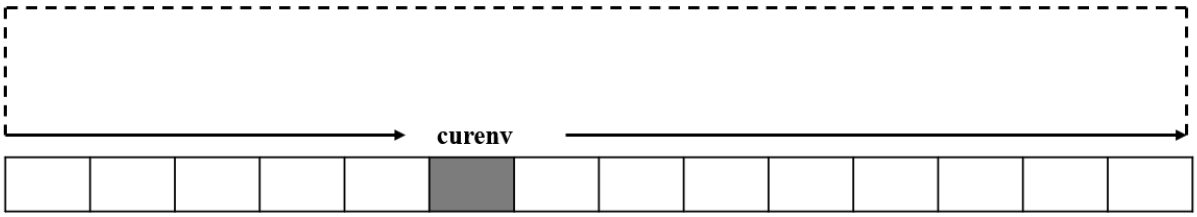
Question

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.

一个全局内核锁并不能取代多核多栈的功能。一个典型的例子是：每个 CPU 可以并行地陷入 trap，在这个过程中每个 CPU 都需要写内核栈。而 int 指令执行非原子且此时无上锁，仍会导致同时写入内核栈的错误。

1.2 Round-Robin Scheduling

JOS 在这里要求我们实现一个轮询 *(Round-Robin)* 的进程调度，这个功能的核心是 sched_yield() 函数。轮询调度检查当前运行的进程，从 envs 数组中排在其后面的进程中选取第一个状态为 RUNNABLE 的进程切换运行；若除当前进程外无可运行进程，则仍调度当前进程运行。轮询逻辑如下图所示：



Exercise 6

Implement round-robin scheduling in sched_yield() as described above.

根据本节内容，对 sched_yield() 代码进行补全：

```
void
sched_yield(void)
{
    struct Env *idle;

    // LAB 4: Your code here.
    bool findNext = false;
    if(!curenv)
    {
        // Situation 1: There is no running ENV
        idle = envs;
        for(int i = 0; i < NENV; ++i, ++idle)
            if(idle->env_status == ENV_RUNNABLE)
            {
                env_run(idle);
                findNext = true;
                break;
            }
    }
    else
    {
        // Situation 2: curenv is not NULL, do circular searching
        // Find the next ENV just behind the curenv
        int curenvIndex = ENVX(curenv->env_id);
        for(int i = 1; i < NENV; ++i)
        {
            idle = envs + (curenvIndex + i) % NENV;
            if(idle->env_status == ENV_RUNNABLE)
            {
                env_run(idle);
                findNext = true;
```

```
                break;
            }
        }
    }

    if(!findNext)
        // Situation 3: Only the curenv is runnable
        env_run(curenv);
    // sched_halt never returns
    sched_halt();
}
```

补全后我们可以对代码进行测试：根据提示，在 i386_init() 函数中增加下列语句段可建立若干进程，最后运行 make qemu CPUS=2 进行测试。

```
ENV_CREATE(user_yield, ENV_TYPE_USER);
```

Question

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?

因为参数 e 保存在内核栈中，而对于所用进程，内核段的内存映射是相同的。

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?

保存过程发生在 _alltrap 处，恢复过程发生在 env_pop_tf() 处。只有保证调度前后进程状态一直，才能确保进程运行不会出错。

1.3 System Calls for Environment Creation

完成上面部分的内容后，我们得到一个具备进程切换能力的内核，但其尚未提供用户进程接口，因此本节主要内容是构建用户态的进程创建接口。

*nix 系统提供系统调用 fork() 作为进程创建的用户接口，它复制调用它的进程的地址空间，创建新的进程。父进程与子进程的区别仅有进程 ID，且 fork() 在父进程中返回子进程 ID，在子进程中则返回 0。

JOS 实现了一组封装相对简单的系统调用，用于构建更高度封装的 fork()，主要包括：sys_exofork，sys_env_set_status，sys_page_alloc，sys_page_map 和 sys_page_unmap。

Exercise 7

Implement the system calls described above in kern/syscall.c and make sure syscall() calls them.whenever you call envid2env(), pass 1 in the checkperm parameter.

根据源码空缺处的提示，完善上述接口。

```
/* sys_exofork */
static envid_t
sys_exofork(void)
{
   // LAB 4: Your code here.
   struct Env *childEnv;
   int val = env_alloc(&childEnv, curenv->env_id);
   if(val < 0)
      return val;

   childEnv->env_status == ENV_NOT_RUNNABLE;
   childEnv->env_tf = curenv->env_tf;
   childEnv->env_tf.tf_regs.reg_eax = 0;
   return childEnv->env_id;
}

/* sys_env_set_status */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   e->env_status = status;
   return 0;
}
```

```
/* sys_page_alloc */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   e->env_status = status;
   return 0;
}

/* sys_page_map */
static int
sys_env_set_status(envid_t envid, int status)
{
   // LAB 4: Your code here.
   if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
      return -E_INVAL;

   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   e->env_status = status;
   return 0;
}

/* sys_page_unmap */
static int
sys_page_unmap(envid_t envid, void *va)
{
   // LAB 4: Your code here.
   struct Env *e;
   if(envid2env(envid, &e, 1) < 0)
      return -E_BAD_ENV;
   if(va >= (void *)UTOP || va != ROUNDDOWN(va, PGSIZE))
      return -E_INVAL;

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

Chapter 2

Copy-on-Write Fork

Chapter 3

Preemptive Multitasking and IPC