# 操作系统实习报告

*曹胜操 - 1500012838*

## Lab 4: Preemptive Multitasking

在这一Lab中，我们将在用户模式运行环境的基础之上实现多任务的并发。首先我们需要添加多处理器的支持，设计一个基本的调度算法，以及提供用户环境相关的系统调用接口。更进一步的内容包括类似于Unix中的`fork()`函数以及进程间通信。

### Part A: Multipr ocessor Support and Cooper ative Multitasking

我们将在JOS上支持SMP (Symmetric Multiprocessing)。在启动时，不同的处理器还是有区别的，因为首先会有一个BSP (Bootstrap Processor)来将系统初始化，并激活其它的APs (Application Processors)。我们之前已有的JOS代码都是在BSP上运行的。

每个CPU都有一个LAPIC单元，用以传递中断信号。我们将会使用LAPIC的以下功能：

- 确认当前代码运行在哪个CPU上：`cpunum()`
- 由BSP发送`STARTUP`中断给APs来启动其它CPU：`lapic_startup()`
- 触发时钟中断以支持抢占式调度：`apic_init()`

我们可以看一下`kern/lapic.c`中的这些函数。例如`cpunum()`：

```
int
cpunum(void)
{
    if (lapic)
        return lapic[ID] >> 24;
    return 0;
}
```

这里的`lapic`是如何得来的呢？处理器通过MMIO (Memory-Mapped I/O)来访问自己的LAPIC，这样要访问其中的寄存器就可以直接通过内存访问来实现。LAPIC映射到的物理内存地址从`0xFE000000`开始，但是这个地址太大以至于我们不能通过原有的加上`KERNBASE`来得到虚拟地址。我们需要自己建立从`MMIOBASE`的映射。

> Exercise 1. 实现`kern/pmap.c`中的`mmio_map_region`。可以参考`kern/lapic.c`中的`lapic_init`看看这个函数是怎么被用到的。

在`lapic_init()`中：

```
    // lapicaddr is the physical address of the LAPIC's 4K MMIO
    // region.  Map it in to virtual memory so we can access it.
    lapic = mmio_map_region(lapicaddr, 4096);
```

可见，`mmio_map_region()`返回了`lapicaddr`实际被映射到的虚拟地址，后面将用这个地址来对LAPIC进行访问。类似于`mem_init()`中，利用`boot_map_region()`我们可以写出：

```
//
// Reserve size bytes in the MMIO region and map [pa,pa+size) at this
// location.  Return the base of the reserved region.  size does *not*
// have to be multiple of PGSIZE.
//
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    // Where to start the next region.  Initially, this is the
    // beginning of the MMIO region.  Because this is static, its
    // value will be preserved between calls to mmio_map_region
    // (just like nextfree in boot_alloc).
    static uintptr_t base = MMIOBASE;

    // Reserve size bytes of virtual memory starting at base and
    // map physical pages [pa,pa+size) to virtual addresses
    // [base,base+size).  Since this is device memory and not
    // regular DRAM, you'll have to tell the CPU that it isn't
    // safe to cache access to this memory.  Luckily, the page
    // tables provide bits for this purpose; simply create the
    // mapping with PTE_PCD|PTE_PWT (cache-disable and
    // write-through) in addition to PTE_W.  (If you're interested
    // in more details on this, see section 10.5 of IA32 volume
    // 3A.)
    //
    // Be sure to round size up to a multiple of PGSIZE and to
    // handle if this reservation would overflow MMIOLIM (it's
    // okay to simply panic if this happens).
    //
    // Hint: The staff solution uses boot_map_region.
    //
    // Your code here:
    size = ROUNDUP(size, PGSIZE);
    if (base + size > MMIOLIM)
        panic("mmio_map_region: MMIO overflows!\n" );
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD | PTE_PWT | PTE_W);
    void *ret = (void*)base;
    base += size;
    return ret;
}
```

注释中已经提示了，我们需要注意页对齐、权限和边界的问题。

SMP启动的过程是这样的：

- 在APs启动之前，BSP需要调用`kern/mpconfig.c`中的`mp_init()`，通过BIOS中的内容收集必要的信息。
- APs在实模式下启动，其代码`kern/mpentry.S`就类似于BSP的启动代码`boot/boot.S`。BSP要将`kern/mpentry.S`复制到`MPENTRY_PADDR = 0x7000`的位置，以便APs启动。
- 复制完成后，`boot_aps()`通过逐个发送`STARTUP`这个中断给每个LAPIC来启动每个APs。
- 每个AP执行完`kern/mpentry.S`的代码后，即进入了带分页的保护模式，然后调用`mp_main()`，最后会设置`struct CpuInfo`中的`cpu_status`为`CPU_STARTED`，这样BSP就知道AP已经启动完毕，可以继续唤醒下一个AP了。

> Exercise 2. 阅读 `kern/init.c` 中的 `boot_aps()` `mp_main()` 以及 `kern/mpentry.S` 。理解启动过程中的控制流转换。然后修改 `kern/pmap.c` 中的 `page_init()` 避免把 `MPENTRY_PADDR` 这一页加入空闲页，这样我们才能安全地使用这一地址来运行AP的启动代码。

在 `boot_aps()` 中：

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;

    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);

    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;

        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

这里BSP把 `kern/mpentry.S` 代码复制到了 `MPENTRY_PADDR` ，然后一个个地启动AP。它通过 `lapic_startap(c->cpu_id, PADDR(code))` 发送启动的中断信号，然后忙等待直到AP启动完成。中断发送后，AP就从 `MPENTRY_PADDR` 开始执行 `kern/mpentry.S` 中的代码。在 `kern/mpentry.S` 中：

```
...
.code16
.globl mpentry_start
mpentry_start:
    cli

    xorw    %ax, %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss

    lgdt    MPBOOTPHYS(gdtdesc)
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

    ljmpl   $(PROT_MODE_CSEG), $(MPBOOTPHYS(start32))

.code32
start32:
    movw    $(PROT_MODE_DSEG), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs

    # Set up initial page table. We cannot use kern_pgdir yet because
    # we are still running at a low EIP.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Switch to the per-cpu stack allocated in boot_aps()
    movl    mpentry_kstack, %esp
    movl    $0x0, %ebp       # nuke frame pointer

    # Call mp_main().  (Exercise for the reader: why the indirect call?)
    movl    $mp_main, %eax
    call    *%eax
...
```

类似的过程，初始化一些寄存器，设置GDT，进入保护模式，打开分页，最后调用`mp_main()`。在`mp_main()`中：

```
...
    // We are in high EIP now, safe to switch to kern_pgdir
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());

    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up
...
```

在完成一些其它的初始化之后，将`thiscpu->cpu_status`设置为`CPU_STARTED`，告诉BSP启动已完成。之后BSP将继续启动下一个AP。

回到这个Exercise的正题，我们修改`page_init()`只需要加入一个特判：

```
...
    size_t t5 = PGNUM(MPENTRY_PADDR);
    size_t i;
    for (i = 0; i < npages; ++i)
    {
        if (i == t5)
        {
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
            continue;
        }
...
```

> Question 1. 比较`kern/mpentry.S`和`boot/boot.S`。前者是被编译链接到`KERNBASE`地址以上的。宏`MPBOOTPHYS`的目的是什么？为什么在前者中才需要用这个宏？

最主要的区别就是前者中需要用到地址时，都使用了`MPBOOTPHYS`宏，例如`lgdt MPBOOTPHYS(gdtdesc)`（后者中直接是`lgdt gdtdesc`）。宏的定义是：

```
#define MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)
```

可见，其目的是计算出一个绝对的物理地址。AP是启动于实模式的，只能使用这样的低地址。如果不加这个宏，这里的地址都是链接器算出来的地址，是在`KERNBASE`以上的，无法为AP所用。`boot/boot.S`就不需要了，因为它本身就在低地址上。

在多处理器OS上，区分每个CPU的私有状态是很重要的。`kern/cpu.h`中定义了大多数CPU状态。`struct CpuInfo`存储了CPU相关的变量，`cpunum()`返回当前CPU的ID，`thiscpu`宏指向当前CPU的`struct CpuInfo`。需要注意的CPU状态包括：

- 内核栈：多CPU可能同时陷入内核，因此需要各自独立的内核栈。`percpu_kstacks[NCPU][KSTKSIZE]`保留了这一空间。`inc/memlayout.h`中显示了映射的布局状态。还需要注意的是不同CPU内核栈之间还有`KSTKGAP`大小的额外空间，这样栈溢出时也不会破坏其它CPU的内核栈。
- TSS：同样需要各自独立的TSS来指定不同CPU的内核栈所在位置。TSS位于`cpus[i].cpu_ts`，TSS描述符位于`gdt[(GD_TSS0 >> 3) + i]`。之前所用的`ts`将不再有效。

- 当前环境指针：每个CPU可以并行运行不同环境，所以`curenv`被重新定义为`cpus[cpunum()].cpu_env`。
- 系统寄存器：所有的寄存器对于CPU都是私有的，包括`lcr3()` `ltr()` `lgdt()` `lidt()`等。`env_init_percpu()`和`trap_init_percpu()`会初始化这些私有的寄存器。

> Exercise 3. 修改`kern/pmap.c`中的`mem_init_mp()`来映射每个CPU各自的内核栈，其布局显示在`inc/memlayout.h`中。

通过`inc/memlayout.h`可以看到，在`KSTACKTOP`之下依次是`KSTKSIZE`大小的CPU0的内核栈，`KSTKGAP`大小的保护性无效内存，`KSTKSIZE`大小的CPU1的内核栈，`KSTKGAP`大小的保护性无效内存……总的大小不超过`PTSIZE`。于是得到`mem_init_mp()`：

```
// Modify mappings in kern_pgdir to support SMP
//   - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE, KSTACKTOP)
//
static void
mem_init_mp(void)
{
    // Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.
    //
    // For CPU i, use the physical memory that 'percpu_kstacks[i]' refers
    // to as its kernel stack. CPU i's kernel stack grows down from virtual
    // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP), and is
    // divided into two pieces, just like the single stack you set up in
    // mem_init:
    //     * [kstacktop_i - KSTKSIZE, kstacktop_i)
    //          -- backed by physical memory
    //     * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i - KSTKSIZE)
    //          -- not backed; so if the kernel overflows its stack,
    //             it will fault rather than overwrite another CPU's stack.
    //             Known as a "guard page".
    //     Permissions: kernel RW, user NONE
    //
    // LAB 4: Your code here:
    int i;
    uintptr_t kstacktop_i;
    for (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. 修改`kern/trap.c`中的`trap_init_percpu()`来初始化所有CPU的TSS及其描述符。

在Lab 3里这个函数正确地初始化了BSP的TSS及其描述符，我们只需改成找到当前CPU进行初始化就可以了：

```
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // The example code here sets up the Task State Segment (TSS) and
    // the TSS descriptor for CPU 0. But it is incorrect if we are
    // running on other CPUs because each CPU has its own kernel stack.
    // Fix the code so that it works for all CPUs.
    //
    // Hints:
    //   - The macro "thiscpu" always refers to the current CPU's
    //     struct CpuInfo;
    //   - The ID of the current CPU is given by cpunum() or
    //     thiscpu->cpu_id;
    //   - Use "thiscpu->cpu_ts" as the TSS for the current CPU,
    //     rather than the global "ts" variable;
    //   - Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS descriptor;
    //   - You mapped the per-CPU kernel stacks in mem_init_mp()
    //
    // ltr sets a 'busy' flag in the TSS selector, so if you
    // accidentally load the same TSS on more than one CPU, you'll
    // get a triple fault.  If you set up an individual CPU's TSS
    // wrong, you may not get a fault until you try to return from
    // user space on that CPU.
    //
    // LAB 4: Your code here:

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

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

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

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

我们还需要解决运行内核代码时不同CPU之间的竞争问题。最简单的办法是使用大内核锁，这是一种全局的锁，只要进入内核模式就必须获得该锁，而返回用户模式时释放该锁。这样，任何时刻只有一个CPU可以运行内核代码，其它的CPU只能等待。

`kern/spinlock.h`声明了这个大内核锁`kernel_lock`，提供了`lock_kernel()` `unlock_kernel()`来获得或释放锁。我们可以查看其代码。JOS中的锁是通过自旋锁实现的，结构体中有意义的只是一个`unsigned locked`。具体实现在`kern/spinlock.c`中：

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
spin_lock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
    if (holding(lk))
        panic("CPU %d cannot acquire %s: already holding" , cpunum(), lk->name);
#endif

    // The xchg is atomic.
    // It also serializes, so that reads after acquire are not
    // reordered before it.
    while (xchg(&lk->locked, 1) != 0)
        asm volatile ("pause");

    // Record info about lock acquisition for debugging.
#ifdef DEBUG_SPINLOCK
    lk->cpu = thiscpu;
    get_caller_pcs(lk->pcs);
#endif
}

// Release the lock.
void
spin_unlock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
    if (!holding(lk)) {
        int i;
        uint32_t pcs[10];
        // Nab the acquiring EIP chain before it gets released
        memmove(pcs, lk->pcs, sizeof pcs);
        cprintf("CPU %d cannot release %s: held by CPU %d\nAcquired at:" ,
            cpunum(), lk->name, lk->cpu->cpu_id);
        for (i = 0; i < 10 && pcs[i]; i++) {
            struct Eipdebuginfo info;
            if (debuginfo_eip(pcs[i], &info) >= 0)
                cprintf("  %08x %s:%d: %.*s+%x\n" , pcs[i],
                    info.eip_file, info.eip_line,
                    info.eip_fn_namelen, info.eip_fn_name,
                    pcs[i] - info.eip_fn_addr);
            else
                cprintf("  %08x\n", pcs[i]);
        }
        panic("spin_unlock");
    }

    lk->pcs[0] = 0;
    lk->cpu = 0;
#endif
```

```
    // The xchg instruction is atomic (i.e. uses the "lock" prefix) with
    // respect to any other instruction which references the same memory.
    // x86 CPUs will not reorder loads/stores across locked instructions
    // (vol 3, 8.2.2). Because xchg() is implemented using asm volatile,
    // gcc will not reorder C statements across the xchg.
    xchg(&lk->locked, 0);
}
```

除掉调试信息之外，对锁所做的只是通过原子的`xchg()`来修改锁中的`locked`，获得锁时如果暂时不能就暂停等待锁被释放。

我们需要在四处使用大内核锁：

- `i386_init()`中BSP唤醒APs之前获得锁
- `mp_main()`中AP初始化完之后，`sched_yield()`之前获得锁
- `trap()`中从用户模式陷入之后获得锁（但是如果之前已经在内核模式则不用）
- `env_run()`中切换到用户模式之前释放锁

> Exercise 5. 按上面所描述的使用大内核锁。

`i386_init()`中：

```
...
    // Lab 4 multitasking initialization functions
    pic_init();

    // Acquire the big kernel lock before waking up APs
    // Your code here:
    lock_kernel();

    // Starting non-boot CPUs
    boot_aps();
...
```

`mp_main()`中：

```
...
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

    // 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();
    sched_yield();
...
```

`trap()`中：

```
...
    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()`中：

```
...
    lcr3(PADDR(curenv->env_pgdir));
    unlock_kernel();
    env_pop_tf(&curenv->env_tf);
...
```

> Question 2. 似乎使用大内核锁可以保证只有一个CPU运行内核代码，那么为什么还需要各自独立的内核栈？

在实现之后，发现其实前面所说的似乎有些不准确。比如说其实从用户模式trap进入内核时，已经在内核模式下压入了`struct Trapframe`等信息，调用了`trap()`函数，直到在这里`lock_kernel()`完成后才算是获得了大内核锁，因此其实是在内核模式下进行的等待。如果只用同一个内核栈，那么`struct Trapframe`有可能会因为不同CPU同时的处理造成混乱。