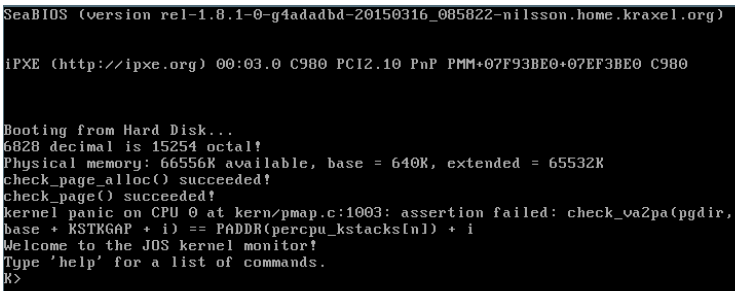
实验 2 同步/互斥

班级：___谢志鹏___ 姓名：___王傲___ 学号：___15300240004___

请于 2017 年 10 月 31 日实验课后当天内，上传到 ftp://10.141.251.211 的 labupload 文件夹的 lab2 中
（用户名：stu，密码：os）。

Q1. 阅读材料，实现在 kern/pmap.c 文件下的 mmio_map_region 函数。该函数通过实现内存映射为支持多处理器启动做准备。其中用到存储管理方面的内容参考阅读资料。

运行结果（使用命令 make qemu）：



实现代码段：

```
    void * record = (void *)base;
    //since 'base' is static, use 'record'
    //to record the value of 'base' and serve as a return value

    size = ROUNDUP(size, PGSIZE);
    //according to the instruction, 'size' has to be round up to
    //a multiple of PGSIZE to avoid fault

    if (base + size > MMIOLIM || base + size < base)
    {
        panic("mmio_map_region : wrong allocation\n");
    }
    //according to the instruction, try to panic
    //if this reservation would overflow MMIOLIM

    boot_map_region(kern_pgdir,base,size,pa,(PTE_W|PTE_PCD|PTE_PWT));
    //reserve 'size' bytes of virtual memory starting at 'base' and
    //map physical pages [pa,pa+size) in the page table rooted at
'pgdir'
    //to virtual addresses [base,base+size)
    base += size;//change 'base' after the allocation

    return record;
```

实现思路及对应代码解释：

根据 mmio_map_region 函数中相应的指导，实现思路为预留从 base 开始的 size 大小的 MMIO 的虚拟地址空间并将其映射到从 pa 开始的 size 大小的物理地址空间。

具体的，因为 base 是静态的，因此首先使用变量 record 记录 base，将 record 作为返回值。然后，根据要求，将 size 变量 roundup 为 size 和 PGSIZE 的公倍数，用于内存对齐，其中 PGSIZE 为页大小。之后，对是否溢出 MMIOLIM 进行判断，溢出则调用 panic。之后，调用 boot_map_region 函数，利用页表 kern_pgdir、物理起始地址 pa、MMIO 起始地址 base 和调整大小后的 size 进行映射。特别注意，对于权限问题，即 boot_map_region 函数的参数 perm 的设置，由于这不是一般的 DRAM，所以 CPU 需要知道随意 Cache 访问这个内存单元是不安全的，因此，将权限设置为 PTE_W|PTE_PCD|PTE_PWT，对应为 Writeable,Cache-Disable 和 Write-Through。

Q2.修改 kern/pmap.c 中的 mem_init_mp（）函数，实现对每个处理器的内核栈的初始化。

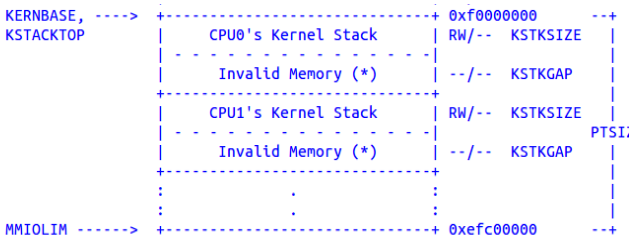
实现代码段：

```
int i = 0;
uintptr_t kstacktop_i;

for (i = 0; i < NCPU; i++)
{
    kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir,
                    kstacktop_i - KSTKSIZE,
                    ROUNDUP(KSTKSIZE, PGSIZE),
                    PADDR(&percpu_kstacks[i]),
                    PTE_W | PTE_P);
}
//the explanation of this chunk of code is sophisticated,
//thus detailed explanation is in the report
```

实现思路及对应代码解释：

根据指导和 inc/memlayout.h 中的信息可以获得实现思路。如图：



根据设计，percpu_kstacks[NCPU][KSTKSIZE]这个二维数组给 NCPU 个 CPU 保留了内核栈的空间。我们需要做的，就是映射每个 CPU 的内核栈（多个）到内核栈的虚拟地址（一

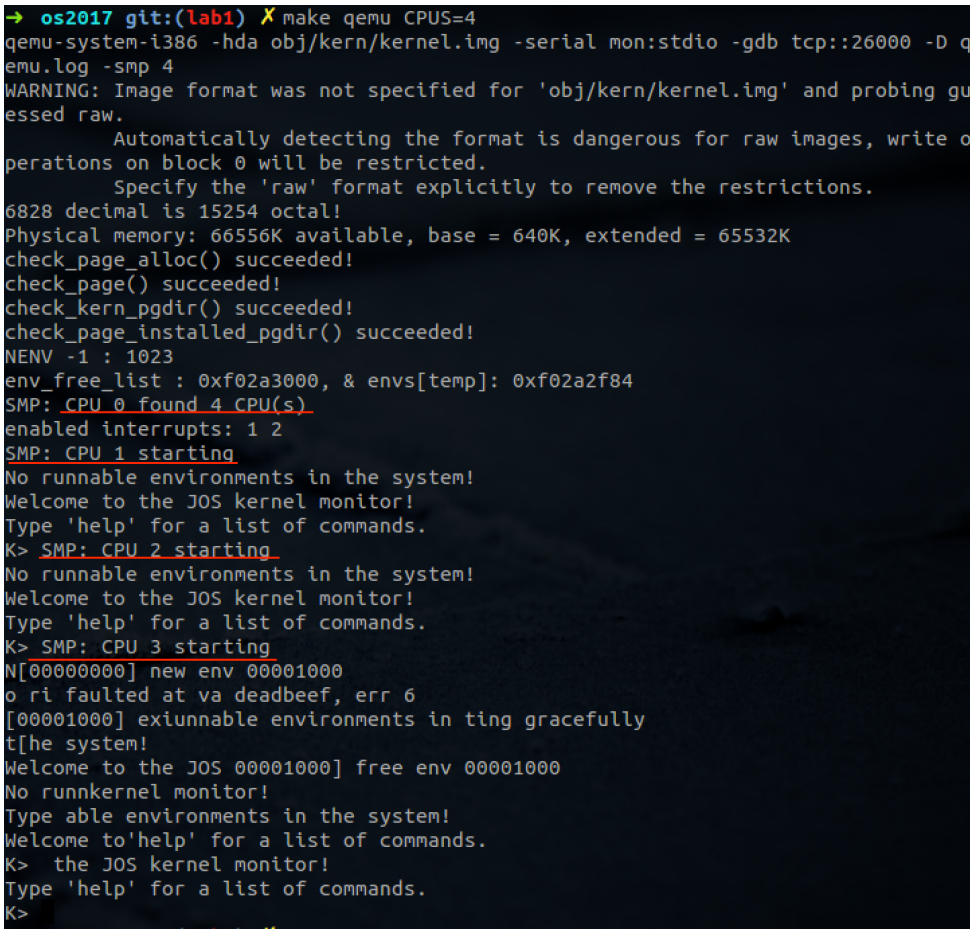
个），并且每个栈之间相隔一个 guard pages 作为缓冲区 buffer。CPU0 的栈将从 KSTACKTOP（设为 KERNBASE）向下增长，CPU 1 的栈将在 CPU 0 的栈增长方向的底部之后的 KSTKGAP 字节开始。每个虚拟内核栈的大小为 KSTKSIZE，每个缓冲区的大小为 KSTKGAP。

因此，实验的思想并不复杂，就是通过一个循环，从 KSTACKTOP 开始为每一个 CPU 分配内核栈的空间并完成映射。其中，kstacktop_i 是第 i 个 CPU 的内核栈的起始地址，计算方法为 KSTACKTOP - i * (KSTKSIZE + KSTKGAP)。另外，[kstacktop_i - KSTKSIZE, kstacktop_i]被用于作为内核栈的映射。缓冲区用于防止某个 CPU 的内核栈数据溢出后对其他 CPU 内核栈上存储的数据造成影响。

注意，使用 boot_map_region 进行映射时，映射的起始地址为 kstacktop_i - KSTKSIZE，大小为 ROUNDUP(KSTKSIZE,PGSIZE)(用于内存对齐)，实际物理地址为 &percpu_kstacks[i]，并利用 PADDR 函数获得真正的物理（Physical）地址。至于权限的设置，要求中明确说明允许内核读写，不允许用户操作，所以设置为 PTE_W | PTE_P 而没有 PTE_U。

Q3.修改 trap_init_percpu(文件 kern/trap.c 中)，初始化 BSP 的 TSS 和 TSS 描述符

运行结果（使用 make qemu CPUS=4）：



实现代码段：

```
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKSIZE +
KSTKGAP);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

    //Initialize the TSS slot of the gdt/initialize the TSS
descriptor
    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;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0 + sizeof(struct Segdesc) * cpunum());//set the
register

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

实现思路及对应代码解释：

更改前的代码如图：

```
ts.ts_esp0 = KSTACKTOP;
ts.ts_ss0 = GD_KD;
ts.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;

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

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

可以看出，原来代码存在的问题是设置 TSS 和 TSS descriptor 时仅仅设置了 CPU0 的 ts，然而在其他 CPU 上初始化 trap，调用 trap_init_percpu 函数时会产生错误，因为每个 CPU 有自己的内核栈，TSS 和 TSS discriptor 也不同。考虑到 thiscpu 永远指向当前正在运行的 CPU 的 struct CpuInfo，因此初始化 esp0 和 ss0 应该修改 thiscpu 的 cpu_ts 属性，即当前正在运行的 CPU 的 ts，而不是全局变量 ts。同时，相应的地址也会发生变化。由于 cpunum()反应了当前 CPU 的编号，所以 esp0，即当前 CPU 内核栈的栈顶不再是 KSTACK，而是 KSTACKTOP - cpunum() * (KSTKSIZE + KSTKGAP)，即第 cpunum()个 CPU 的栈顶；gdt[(GD_TSS0 >> 3) + cpunum()]反映了第 cpunum()个 CPU 的 TSS descriptor。

特别的，对于 ltr 函数，ltr 在 TSS descriptor 中设置了一个 busy 的标记，在多个 CPU 上重复调用同一个 TSS 的话，会触发一个 triple fault。所以，对不同的 CPU 使用 ltr 对相应的 TSS descriptor 进行设置。sizeof (struct Segdesc)为 TSS descriptor 的大小。用 ltr 设置 TSS descriptor 时按线性顺序增加。

Q4. 阅读材料，在 i386_init(),mp_main(),trap()这三个函数中的适当位置调用 lock_kernel()函数加锁，实现多处理器之间的互斥机制。

实现代码段：

i386_init():

```
    // Acquire the big kernel lock before waking up APs
    // lab2 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!
    // Lab2 Your code here:
    lock_kernel();
    sched_yield();
```

trap():

```
    // Trapped from user mode.
    // Acquire the big kernel lock before doing any
    // serious kernel work.
    // LAB 2: Your code here.
    lock_kernel();
    assert(curenv);
```

env_run():

```
    // LAB 2: Your code here.
    unlock_kernel();
    env_pop_tf(&(e->env_tf));
```

实现思路及对应代码解释：

实验思路比较清晰：在 i386_init()函数中，BSP 先获得大内核锁然后再调用 boot_aps()函数启动其余的 CPU；在 mp_main()函数中，在初始化 AP 后获得大内核锁，然后调用 sched_yield()开始在这个 AP 上运行用户环境；在 trap()函数中，从用户态陷入到内核态必须获得大内核锁，通过检查 tf_cs 的低位确定这个陷入发生在用户态还是在内核态；在 env_run()函数中，在切换到用户态之前释放大内核锁。

Q5. 阅读材料，在 kern/syscall.c 文件中，实现 sys_ipc_recv（），sys_ipc_try_send()函

数，理解进程间通信的同步机制。

实现代码段：

sys_ipc_try_send()：

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 2: Your code here.
    //try to send 'value' and page 'srcva' to the target env 'envid'
    //if srcva is not null, then it is shared
    struct Env *e; //refer to the receiver
    struct PageInfo *page;
    pte_t *pte;

    //-E_BAD_ENV if environment envid doesn't currently exist
    if(envid2env(envid, &e, 0) != 0)
        return -E_BAD_ENV;

    //the send fails with a return value of -E_IPC_NOT_RECV if the
    //target is not blocked to wait for an IPC
    if(e->env_ipc_recving == 0)
        return -E_IPC_NOT_RECV;

    //set the parameters of the receiver env(e)
    e->env_ipc_recving = 0;//env_ipc_recving is set to 0 to block future
sends
    e->env_ipc_from = curenv->env_id;//env_ipc_from is set to the sending
envid to record the sender
    e->env_ipc_value = value;//env_ipc_value is set to the 'value' parameter
to finish the value passing of the IPC

    //if srcva < UTOP, then also send shared page currently mapped at
'srcva'
    //'< UTOP' means the page is in the user-available space
    if((uint32_t) srcva < UTOP)
    {
        //-E_INVAL if srcva < UTOP but srcva is not page-aligned
        if(((uint32_t)srcva % PGSIZE) != 0)
            return -E_INVAL;

        //-E_INVAL if srcva < UTOP and perm is inappropriate
        if((perm & PTE_U) == 0 ||
           (perm & PTE_P) == 0 ||
```

```
            (perm & ~PTE_SYSCALL) != 0)
            return -E_INVAL;

        //-E_INVAL if srcva < UTOP but srcva is not mapped in the caller's
        //address space
        if((page = page_lookup(curenv->env_pgdir,srcva,&pte))== NULL)
            return -E_INVAL;

        //-E_INVAL if (perm & PTE_W), but srcva is read-only in the
        //current environment's address space
        if((perm & PTE_W) != 0 && (*pte & PTE_W) == 0)
            return -E_INVAL;

        //-E_NO_MEM if there's not enough memory to map srcva in envid's
        //address space
        if((page_insert(e->env_pgdir,
                        page,
                        e->env_ipc_dstva,
                        perm))!= 0)
            return -E_NO_MEM;

        cprintf("sys_ipc_try_send: from 0x%x\n", e->env_ipc_from);
        e->env_ipc_perm = perm;//env_ipc_perm is set to 'perm' if a page was
transferred, 0 otherwise
    }
    else
        e->env_ipc_perm = 0;

    //the target environment is marked runnable again, returning 0
    //from the paused sys_ipc_recv system call
    e->env_status = ENV_RUNNABLE;//free from stuck to wait for the message
    return 0;
    //panic("sys_ipc_try_send not implemented");
}
```

sys_ipc_recv ():

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 2: Your code here.
    curenv->env_ipc_recving = 1;//blocked and is ready to receive message

    //if 'dstva' is < UTOP, then is ready to receive a page of data
```

```
    if((uint32_t) dstva < UTOP)
    {
        //-E_INVAL if srcva < UTOP but srcva is not page-aligned
        if((uint32_t) dstva % PGSIZE != 0)
            return -E_INVAL;//return < 0 on error

        //'dstva' is the virtual address at which the sent page should be
mapped
        curenv->env_ipc_dstva = dstva;//after mapping the page, it's in a
shared form
        cprintf("sys_ipc_recv: va 0x%x\n", (uint32_t)dstva);
    }

    curenv->env_tf.tf_regs.reg_eax = 0;//set the return value on register
eax
    curenv->env_status = ENV_NOT_RUNNABLE;//set the status of current env

    //this function only returns on error, but the system call will
eventually
    //return 0 on success
    return 0;
}
```

实现思路及对应代码解释：

主要思路是实现进程间通信 IPC。进程使用 JOS 的 IPC 机制所发送的信息包括两部分：单个 32 位的值（uint32_t value）和可选的单页映射（void *srcva）。

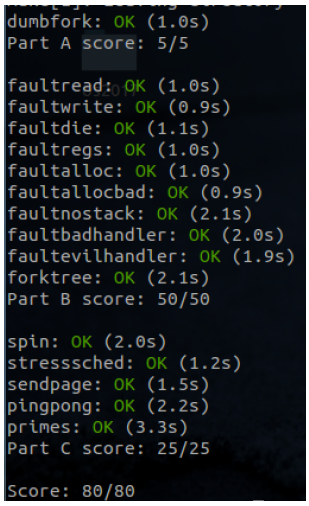
发送和接受消息时，接收进程调用 sys_ipc_recv 函数接收消息。当接收进程等待接收消息时，任何其他进程都可以向其发送消息，而不仅仅是特定的进程（体现在 sys_ipc_recv 函数没有 envid_t envid 参数，即不指定特定发送进程）。JOS 以接收消息的进程的 id 以及待发送的值为参数调用 sys_ipc_try_send 函数发送一个值。如果接收消息的进程正在等待接收消息（该进程调用 sys_ipc_recv 系统调用，但还没有接收到值），sys_ipc_try_send 系统调用传送消息并返回 0，否则返回-E_IPC_NOT_RECV 表示目标进程当前不希望接收到一个值。

当进程使用有效的 dstva 参数（低于 UTOP）调用 sys_ipc_recv 时，表示它愿意接收页面映射。如果发送者发送一个 page，那么这个 page 应该在接收者的地址空间中的 dstva 映射。如果接收者已经在 dstva 上映射了一个页面，那么之前的页映射被取消。当进程以有效的 srcva（在 UTOP 下面）以及权限 perm 为参数调用 sys_ipc_try_send 时，这意味着发送者想要将当前映射到 srcva 的页面发送给接收者。在成功的 IPC 之后，发送方在其地址空间中的 srcva 保持页面的原始映射，但是接收方在其地址空间最初指定的 dstva 处获得了与发送者同一物理页的映射。因此，该页面在发送者和接收者之间共享。如果发送者或接收者没有指示要传送的页面（srcva 为空），那么没有页面被传送。在任何 IPC 之后，内核将接

收者的 Env 结构中的新字段 env_ipc_perm 设置为接收到的页面的权限，如果没有接收到页面，则为零。

具体的代码实现上，由于整体的思路在上面已经给出，详细的错误判断和处理等又十分冗长，因此不再展开，详细思路见代码注释。

make grade 结果：



实验感想及难点：

这次的实验，对于每个单独的问题，如果理解了要达到的目的和需要完成的函数内的注释，实现起来并不是十分复杂。实验最难的地方在于查找各个函数实现的功能，将所有的过程联系起来，了解从开启到运行的整个过程，因为这需要理解各个文件和函数实现的功能，同时又要接触大量我们并不熟悉的汇编代码，所以比较难。考虑整体运行流程时，从 kern/init.c 开始，逐个看启动时涉及的函数，以及可能需要用到的函数，会有助于理解。

本次实验的主要目的是实现对多处理器的支持，考虑并解决存在多个 CPU 时可能出现的问题，同时实现进程间通信。JOS 对多 CPU（Multi CPU）进行了抽象，用 struct CPUInfo 记录每个 CPU 的状态，用 struct mp 对多处理器进行抽象，而对多 CPU 的启动和执行等则由 kern/mpconfig.c 实现。具体的，与第一次实验仅有一个 CPU 不同，在 i386_init()中增加了 mp_init()和 lapic_init()用于启动多处理器 mp 并完成 LAPIC 的初始化，准备传递系统中断，并用 boot_aps()启动除 BSP 外的其他 AP。

与单处理器不同的是，不同的 CPU 可能同时由于系统调用陷入内核态，切换到内核栈，因此在内核栈中应为每个 CPU 分配空间，防止互相干扰，同时留出 buffer，防止溢出时改写内核栈上的其他数据。在内核栈上每个 CPU 的内核栈线性向下增长，具体排布见 inc/memlayout.c。同时，不再用全局的 TSS 记录中断信息和数据，每个 CPU 保有自己的 TSS 和 TSS descriptor。

由于多 CPU 的出现，我们需要实现相应的互斥机制。JOS 用原语实现了一个大内核锁。这样的实现比较简单，但同时很粗糙，因为多 CPU 情况下可以并行跑用户程序，然而要陷入内核态的话必须获得大内核锁，而大内核锁只有一个，也就是说一次只有一个 CPU 可以陷入内核态。之后，对于每个 CPU，在进入内核态临界区之前要申请获得锁，否则等待；从内核态退回到用户程序，则要释放锁。

特别说一下，由于实现了大内核锁，一次只能有一个 CPU 陷入内核态，因此每个 CPU 各自有一个内核栈看似多余。实际上，不同的内核栈上有不同的信息，一个 CPU 退出内核态后内核栈上可能还有有用的信息，如果只有一个内核栈的话，这些信息会被下一个陷入内核态的 CPU 的信息覆盖掉，所以每个 CPU 要有自己的内核栈。

另一个比较重要的方面，JOS 用 sched_yield 函数实现了 Round_Robin 抢占式调度算法，为每个 CPU 划分时间片，在一个 CPU 的时间片结束后由另一个可用的 CPU 抢占。此外，JOS 还实现了 sys_exofork、sys_env_set_status、sys_page_alloc、sys_page_map、sys_page_unmap 函数,结合 user/dumbfork.c 实现了一个很基本的 fork 功能,生成子进程。由于这些部分不是由我们实现的，因此不再详述。

本次实验另一个重要的部分是实现进程间通信，最为重要的是实现 sys_ipc_recv（），sys_ipc_try_send()两个函数，而这两个函数是 ipc_recv 和 ipc_send 函数的基础。其中，sys_ipc_try_send()函数尝试将一个 32 位的值发送给接收者，并且将一个页映射到接收者（之后这个页二者共享）。最主要的实现就是 e->env_ipc_value=value;，并在各种判断条件中完成对页的传递。sys_ipc_recv（）函数将当前进程阻塞，直至接收到消息；完成接收后，传递的页面将是共享的。

操作系统内核的实现确实十分复杂，但是理解了实现机制后，也会有很大的收获。