一、 exercise1

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.

解答：按照题目要求阅读 kern/lapic.c/lapic_init()理解 mmio_map_region 调用过程，并完成 kern/pamap.c/mmio_map_region()。分析下图代码

```
	// 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);
```

由此可以知道 mmio_map_region()功能是映射 lapic 的 4k 大小 MMIO 的部分空间到指定物理地址，并将 size 对齐。所以具体代码实现如下：

```
	// Your code here:
	//panic("mmio_map_region not implemented");
	uintptr_t ret = base;
	size = ROUNDUP(size, PGSIZE);
	base = base + size;
	if (base >= MMIOLIM) {
		panic("larger than MMIOLIM");
	}
	boot_map_region(kern_pgdir, ret, size, pa, PTE_PCD | PTE_PWT | PTE_W);
	return (void *) ret;
}
```

但此时还不能完全实现 mmio_map_region()的调用，需要接着完成练习二

二、 练习二

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).

解答：题目要求通过阅读 kern/init.c/boot_aps() 和 mpmain(),kern/mpentray.S 文件，完成 page_init()，可以通过 check_page_freelist().由题目知道需要避免将 MPENTRY_PADDR 加入到空页表 free list 中所以只需要在原基础上添加!= MPENTRY_PADDR 即可。具体代码实现如下图：

```
	size_t right_i = PGNUM(PADDR(envs + NENV));
	for (i = 0; i < npages; i++) {
		if(i == 0)
			{	pages[i].pp_ref = 1;
				pages[i].pp_link = NULL;
			}
		else if(i==MPENTRY_PADDR/PGSIZE)
		{
			continue;
		}
		else if(i>=1 && i<npages_basemem && i != PGNUM(MPENTRY_PADDR))
		{
			pages[i].pp_ref = 0;
			pages[i].pp_link = page_free_list;
```

三、 问题一：

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.

解答：由于 AP 没有设置页表，而 BSP 开启了页表，所以需要自己从虚拟地址转换到物理地址，但是在 AP 的保护模式打开之前，是没有办法寻址到 3G 以上的空间的，因此用 MPBOOTPHYS 是用来计算相应的物理地址的。但是在 boot.S 中，由于尚没有启用分页机制，所以我们能够指定程序开始执行的地方以及程序加载的地址；但是，在 mpentry.S 的时候，由于主 CPU 已经处于保护模式下了，因此是不能直接指定物理地址的，给定线性地址，映射到相应的物理地址是允许的。
练习三

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().

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.)

解答：练习三要求完成 kern/pmap.c/mem_init_mp()函数，需要从 KSTACKTOP 根据 cpu 数目设置相印的内核栈，大小为大于未映射页 KSTACK bytes。完成函数后应该通过 check_kern_pgdir(),具体代吗实现如下，

```
//
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:
        //根据 CPU 数目设置相应的内核栈
        for (int i = 0; i != NCPU; ++i) {
                uintptr_t kstacktop = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
                boot_map_region(kern_pgdir, kstacktop - KSTKSIZE, KSTKSIZE, PADDR(percpu_kstacks
        }

}
//
```

调试结果如下：

练习四则要求初始化每个 CPU 的 TSS 并向 GDT 中加入相应的条目，代码如下：

```
        // user space on that CPU.
        //
        // LAB 4: Your code here:
        thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKSIZE + KSTKGAP);
        thiscpu->cpu_ts.ts_ss0 = GD_KD;
```

四、 练习五

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

解答：1、由 lock_kernel 具体代码知道其的核心是 xchg 指令，它可以原子地设置新值并返回原值，以此来判断获取锁是否成功。在唤醒其他 CPU 之前，为防止被唤醒的 CPU 立即启动进程，需要对其进行加锁，修改 kern/init/mp_main（）所以在 kern/init（）中添加如下代码：

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

2、在初始化 AP 后，在调度之前需要 lock，防止其他 CPU 干扰进程的选择，所以在进程调度函数 trap()中添加如下代码：

```
    //
    // Your code here:
    lock_kernel();
    sched_yield();
    // Remove this after you finish Exercise 4
    //for (;;);
```

3、用户态引发中断陷入内核态时，需要 lock，在 kern/trap/trap（）中添加如下代码

```
// LAB 4: Your code here.
lock_kernel();//在发生中断加锁
assert(curenv);
```

4、离开内核态之前，需要 unlock ，所以需要在 kern/env/env_run()中添加如下代码：

```
curenv->env_runs++;//curenv->env_runs++;
lcr3(PADDR(curenv->env_pgdir));
unlock_kernel();
env_pop_tf(&(curenv->env_tf));
```

在完成所有上述函数后，BPS 启动 AP 前，获取内核锁，所以 AP 会在 mp_main 执行调度之前阻塞，在启动完 AP 后，BPS 执行调度，运行第一个进程，之后释放内核锁，这样一来，其中一个 AP 就可以开始执行调度，若有的话运行进程。

六、问题一

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.

解答：如果内核栈中留下不同 CPU 之后需要的数据，可能会造成混乱

七、练习六

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.

If you use `CPUS=1` at this point, all environments should successfully run. Setting CPUS larger than 1 at this time may result in a general protection or kernel page fault once there are no more runnable environments due to unhandled timer interrupts (which we will fix below!).

解答：根据题目要求，需要完成 sched_yield()函数实现循环调度，同时修改 syscall()分派 sys_yield()，在 mp_main 中来调用 sched_yield()；修改 kern/init.c 可以创建所有应用程序 user/yield.c 多个环境。

首先，根据题目要求需要循环调度，所以基本思路就是在 envs 中，从当前进程的下一个开始（或从头开始）寻找状态为 ENV_RUNNABLE 的进程，找到则运行该进程，若找了一圈回到当前进程，则判断当前进程的状态，若为 ENV_RUNNING 则继续运行当前进程，故修改 sched.c/sched_yield()如下：

```
        // LAB 4: Your code here.
        idle = curenv;
        size_t i = idle != NULL ? ENVX(idle->env_id) + 1 : 0;
        for (size_t j = 0; j != NENV; j++, i = (i + 1) % NENV) {
                if (envs[i].env_status == ENV_RUNNABLE) {
                        env_run(envs + i);
                }
        }
        if (idle && idle->env_status == ENV_RUNNING) {
                env_run(idle);
        }
        // sched_halt never returns
        sched_halt();
}
```

为确保成功能够调用 sys_yield 需要在 syscall 中加入相应的分支，具体代码修改如下：

```
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4
uint32_t a5)
{
        // Call the function corresponding to the 'syscallno' parameter.
        // Return any appropriate return value.
        // LAB 3: Your code here.
        //panic("syscall not implemented");

        switch (syscallno) {
        case SYS_cputs:
            sys_cputs((const char *)a1, (size_t)a2);
            return 0;
        case SYS_cgetc:
            return sys_cgetc();
        case SYS_getenvid:
            return sys_getenvid();  //>=0;
        case SYS_env_destroy:
            return sys_env_destroy((envid_t)a1);
        case SYS_yield:
            sched_yield();
            return 0;
        default:
            return -E_NO_SYS;           //-E_INVAL;
        }
}
```

八、问题 3、4

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?

解答：question3：由于 e 变量是保存在内核栈上，而用户页表与内核页表在内

核区域的映射是一致的，所以可以正确地访问该地址.
Question4：这个就是进程上下文切换，保存寄存器发生在执行系统调用（例如 sys_yield）或时钟中断时，相关代码在 trapentry.S 中
九、练习 7

> **Exercise 7.** Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

解答：这部分练习需要完成 system call 的所有功能，即完善 syscall.c 文件的所有函数。具体实现如下步骤：

1、首先需要完成 fork 函数，在这里首先需要调用 evn.c/env_alloc()创建用户环境，且进程的状态为 ENV_NOT_RUNNABLE，寄存器状态为当前父进程的寄存器状态。具体做法是将 trapframe 中 eax 的值设置为 0，由于子进程的 trapframe 与父进程的相同，所以运行子进程时，会执行父进程调用 fork 时的下一条指令，而此时返回值是存在 eax 中的，这样就实现了在子进程中返回 0。代码如下：

```
static envid_t
sys_exofork(void)
{
        // Create the new environment with env_alloc(), from kern/env.c.
        // It should be left as env_alloc created it, except that
        // status is set to ENV_NOT_RUNNABLE, and the register set is copied
        // from the current environment -- but tweaked so sys_exofork
        // will appear to return 0.

        // LAB 4: Your code here.
        struct Env *e;
        int r = env_alloc(&e, curenv->env_id);
        if (r < 0) {
                return r;
        }
        e->env_status = ENV_NOT_RUNNABLE;
        e->env_tf = curenv->env_tf;
        e->env_tf.tf_regs.reg_eax = 0;
        return e->env_id;
        //panic("sys_exofork not implemented");
}
```

2、sys_env_set_status 函数。这里的 status 只有两种选择，根据要求，需要调用 kern/env.c/envid2env()把 envid 映射为 Evn 类型，并通过设置第三个参数为 1 检查是否当前环境拥有权限可以修改 envid 的 status。具体代码如下：

```
sys_env_set_status(envid_t envid, int status)
{
        // Hint: Use the 'envid2env' function from kern/env.c to translate ar
        // envid to a struct Env.
        // You should set envid2env's third argument to 1, which will
        // check whether the current environment has permission to set
        // envid's 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;
        //panic("sys_env_set_status not implemented");
}
```

3、sys_page_alloc 函数：该函数的主要功能是一个 pmap.c 中 page_alloc()和 page_insert()的包装器，注意在插入页面失败时需要释放页面，具体代码如下：

```
sys_page_alloc(envid_t envid, void *va, int perm)
{
        // Hint: This function is a wrapper around page_alloc() and
        //   page_insert() from kern/pmap.c.
        //   Most of the new code you write should be to check the
        //   parameters for correctness.
        //   If page_insert() fails, remember to free the page you
        //   allocated!
        if ((uintptr_t) va >= UTOP || (uintptr_t) va % PGSIZE || (~perm & (PTE_U |
PTE_P))) {
                return -E_INVAL;
        }
        struct Env *e;
        if (envid2env(envid, &e, 1) < 0) {
                return -E_BAD_ENV;
        }
        struct PageInfo *pp = page_alloc(ALLOC_ZERO);
        if (pp == NULL) {
                return -E_NO_MEM;
        }
        if (page_insert(e->env_pgdir, pp, va, perm) < 0) {
        page_free(pp);
        return -E_NO_MEM;
        }
```

4、sys_page_map 函数。该函数是 page_lookup()和 page_insert()的包装器，检测当前是否有权限访问页面，需要判断页面为只读而 perm 具有写权限的情况。具体代码如下：

```
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm)
{
        // Hint: This function is a wrapper around page_lookup() and
        //   page_insert() from kern/pmap.c.
        //   Again, most of the new code you write should be to check the
        //   parameters for correctness.
        //   Use the third argument to page_lookup() to
        //   check the current permissions on the page.
        if ((uintptr_t) srcva >= UTOP || (uintptr_t) srcva % PGSIZE || (uintptr_t)
dstva >= UTOP || (uintptr_t) dstva % PGSIZE || (~perm & (PTE_U | PTE_P))) {
                return -E_INVAL;
        }
        struct Env *srce, *dste;
        if (envid2env(srcenvid, &srce, 1) < 0 || envid2env(dstenvid, &dste, 1) < 0)
        {
                return -E_BAD_ENV;
        }
        pte_t *pte;
        struct PageInfo *pp = page_lookup(srce->env_pgdir, srcva, &pte);
        if (pp == NULL || ((~(*pte) & PTE_W) && (perm & PTE_W))) {
                return -E_INVAL;
        }
        if (page_insert(dste->env_pgdir, pp, dstva, perm) < 0) {
                return -E_NO_MEM;
        }
        return 0;
```

5、sys_page_unmap 函数，具体代码实现如下：

```
static int
sys_page_unmap(envid_t envid, void *va)
{
        // Hint: This function is a wrapper around page_remove().
        if ((uintptr_t) va >= UTOP || (uintptr_t) va % PGSIZE) {
                return -E_INVAL;
        }
        struct Env *e;
        if (envid2env(envid, &e, 1) < 0) {
                return -E_BAD_ENV;
        }
        page_remove(e->env_pgdir, va);
        return 0;
        // LAB 4: Your code here.
        //panic("sys_page_unmap not implemented");
}
```

6、最后，在 syscall 中添加相应的分支，具体代码如下：

```
        case SYS_page_alloc:
            return sys_page_alloc(a1, (void *) a2, a3);
        case SYS_page_map:
            return sys_page_map(a1, (void *) a2, a3, (void *) a4, a5);
        case SYS_page_unmap:
            return sys_page_unmap(a1, (void *) a2);
        case SYS_exofork:
            return sys_exofork();
        case SYS_env_set_status:
            return sys_env_set_status(a1, a2);
        default:
```

至此 part A 部分完成。

十、练习八

Exercise 8. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

根据题目要求，需要完善 sys_env_set_pgfault_upcall，并确保在查看环境 ID 时，有权限可以访问，防止出现危险访问。
根据前边的练习可以知道用户异常处理的调用关系如下：
1）用户程序调用位于 pgfault.c 的 set_pgfault_handler()，作用是设置异常处理程序，为用户异常栈分配页面，并将 env_pgfault_upcall 设置为 _pgfault_upcall
2）当在用户态出现页错误时，trap_dispatch() 调用 page_fault_handler()，然后设置用户异常栈，将 esp 指向该栈顶，eip 指向之前设置的 env_pgfault_upcall，即 _pgfault_upcall，并调用 env_run()
3）根据 eip，执行位于 pfentry.S 的 _pgfault_upcall，首先调用之前设置的异常处理程序，完成后恢复 trapframe 中的寄存器，继续执行用户态指令。
具体代码实现如下：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        struct Env *e;
        if (envid2env(envid, &e, 1) < 0) {
                return -E_BAD_ENV;
        }
        e->env_pgfault_upcall = func;
        return 0;
        // LAB 4: Your code here.
        //panic("sys_env_set_pgfault_upcall not implemented");
}
```

十一、练习九：

Exercise 9. Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

解答：根据题目，要完成在 kern / trap.c 中的 page_fault_handler，实现将页面错误分派给用户模式处理程序所需的代码。并在写入异常堆栈时，采取适当的预防措施，在用户环境在异常堆栈上的空间不足采取解决措施。
因此可以向用户异常栈中压入 UTrapframe，需要判断可能发生多次异常，这种情况下需要在之前的栈顶后先留下一个空位，再压入 UTrapframe，之后会用到这个空位，然后设置 esp 和 eip 并调用 env_run()，具体修改 kern/trap.c/page_fault_handler 代码如下：

```
        // LAB 4: Your code here.
        if (curenv->env_pgfault_upcall != NULL) {
                uintptr_t esp;
                if (tf->tf_esp > UXSTACKTOP - PGSIZE && tf->tf_esp < UXSTACKTOP) {
                        esp = tf->tf_esp - 4 - sizeof(struct UTrapframe);
                } else {
                        esp = UXSTACKTOP - sizeof(struct UTrapframe);
                }
                user_mem_assert(curenv, (void *) esp, sizeof(struct UTrapframe),
PTE_W | PTE_U | PTE_P);
                struct UTrapframe *utf = (struct UTrapframe *) (esp);
                utf->utf_fault_va = fault_va;
                utf->utf_err = tf->tf_err;
                utf->utf_regs = tf->tf_regs;
                utf->utf_eip = tf->tf_eip;
                utf->utf_eflags = tf->tf_eflags;
                utf->utf_esp = tf->tf_esp;
                tf->tf_esp = esp;
                tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;
                env_run(curenv);
        }
```

十二、练习八

Exercise 10. Implement the `_pgfault_upcall` routine in `lib/pfentry.S`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

解答：题目要求完善 lib/pfentry.S/pgfault_upcall，导致页错误返回到用户代码的原点，无需回到内核。需要同时切换堆栈重新加载 EIP。需要修改如下几个地方：

第一处：往 trap-time esp 所指的栈顶（可能是普通栈也可能是异常栈）后面的空位写入 trap-time eip 并将 trap-time esp 往下移指向该位置

第二处：跳过 fault_va 和 err，然后恢复通用寄存器

第三处：跳过 eip，然后恢复 efalgs，如果先恢复 eip 的话，指令执行的位置会改变，所以这里必须跳过

第四处：恢复 esp，如果第一处不将 trap-time esp 指向下一个位置，这里 esp 就会指向之前的栈顶

第五处：由于第一处的设置，现在 esp 指向的值为 trap-time eip，所以直接 ret 即可达到恢复上一次执行的效果

具体代码实现如下：

```
        // LAB 4: Your code here.
        movl 48(%esp), %eax
        subl $4, %eax
        movl %eax, 48(%esp)
        movl 40(%esp), %ebx
        movl %ebx, (%eax)
        // Restore the trap-time registers.  After you do this, you
        // can no longer modify any general-purpose registers.
        // LAB 4: Your code here.
        addl $8, %esp
        popal
        // Restore eflags from the stack.  After you do this, you can
        // no longer use arithmetic operations or anything else that
        // modifies eflags.
        // LAB 4: Your code here.
        addl $4, %esp
        popfl
        // Switch back to the adjusted trap-time stack.
        // LAB 4: Your code here.
        popl %esp
        // Return to re-execute the instruction that faulted.
        // LAB 4: Your code here.
        ret
```

十三、练习 11

Exercise 11. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

解答：接着在 lib/pgfault.c 中完善 set_pgfault_handler()，该函数的主要功能是为用户异常栈分配页面并设置异常处理函数。具体代码实现如下：

```
//
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
        int r;

        if (_pgfault_handler == 0) {
                // First time through!
        envid_t id = sys_getenvid();
        if ((r = sys_page_alloc(id,
        (void *) (UXSTACKTOP - PGSIZE),
        PTE_W | PTE_U | PTE_P)) < 0 ||
        (r = sys_env_set_pgfault_upcall(id, _pgfault_upcall)) < 0)
        {
                panic("sys_page_alloc: %e", r);
        }
                // LAB 4: Your code here.
                //panic("set_pgfault_handler not implemented");
        }

        // Save handler pointer for assembly to call.
        _pgfault_handler = handler;
}
```

十四、练习 12

Exercise 12. Implement `fork`, `duppage` and `pgfault` in `lib/fork.c`.

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

解答：题目需要完善 lib/fork.c 文件中的 fork,duppage 和 pgfault 函数，通过 forktree 测试代码，产生题目中的消息。

首先在 pgfault()中，需要实现下列功能：

检查 err 和 pte 是否符合条件

分配页面到地址 PFTEMP

复制内容到刚分配的页面中

将虚拟地址 addr（需要向下对齐）映射到分配的页面

取消地址 PFTEMP 的映射

具体代码实现如下：

```
        // LAB 4: Your code here.
        if (!(err & FEC_WR) || !(uvpt[PGNUM(addr)] & PTE_COW)) {
                panic("pgfault: failed!");
        }
        // Allocate a new page, map it at a temporary location (PFTEMP),
        // copy the data from the old page to the new page, then move the new
        // page to the old page's address.
        // Hint:
        //   You should make three system calls.

        // LAB 4: Your code here.
        if ((r = sys_page_alloc(0, (void *) PFTEMP, PTE_U | PTE_W | PTE_P)) < 0) {
                panic("pgfault: %e", r);
        }
        memcpy((void *) PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
        if ((r = sys_page_map(0, (void *) PFTEMP, 0, ROUNDDOWN(addr, PGSIZE), PTE_U
| PTE_W | PTE_P)) < 0) {
                panic("pgfault: %e", r);
        }
        if ((r = sys_page_unmap(0, (void *) PFTEMP)) < 0) {
                panic("pgfault: %e", r);
        }
        //panic("pgfault not implemented");
}
```

Duppage()函数，主要功能需要实现先判断权限，再建立映射，同时要注意父进

程页表也需要重新建立映射。具体代码实现如下：

```
static int
duppage(envid_t envid, unsigned pn)
{
        int r;

        // LAB 4: Your code here.
        //panic("duppage not implemented");
        int perm = PGOFF(uvpt[pn]);
        if (perm & (PTE_W | PTE_COW)) {
                perm |= PTE_COW;
                perm &= ~PTE_W;
        }
        if ((r = sys_page_map(0, (void *) (pn * PGSIZE), envid, (void *) (pn *
PGSIZE), perm)) < 0) {
                panic("duppage: %e", r);
        }
        if ((r = sys_page_map(0, (void *) (pn * PGSIZE), 0, (void *) (pn * PGSIZE),
perm)) < 0) {
                panic("duppage: %e", r);
        }
        return 0;
}
```

Fork()函数：这里需要设置异常处理函数，创建子进程，映射页面到子进程，为子进程分配用户异常栈并设置 pgfault_upcall 入口，将子进程设置为可运行的状态。具体代码实现如下：

```
fork(void)
{
        // LAB 4: Your code here.
        //panic("fork not implemented");
        int r;
        set_pgfault_handler(pgfault);
        envid_t envid = sys_exofork();
        if (envid == 0) {
                thisenv = &envs[ENVX(sys_getenvid())];
                return 0;
        }
        for (uintptr_t va = 0; va < USTACKTOP; va += PGSIZE) {
                if ((uvpd[PDX(va)] & PTE_P) && (uvpt[PGNUM(va)] & PTE_P)) {
                        duppage(envid, PGNUM(va));
                }
        }
        if ((r = sys_page_alloc(envid, (void *) (UXSTACKTOP - PGSIZE), PTE_U |
PTE_W | PTE_P)) < 0) {
                return r;
        }
        extern void _pgfault_upcall(void);
        if ((r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0) {
                return r;
        }
        sys_env_set_status(envid, ENV_RUNNABLE);
        return envid;
}
```

至此 part B 所有部分完成，make grade.

十五、练习十三

Exercise 13. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `spin`), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

解答：题目需要修改 kern/trapentry.S 和 kern/trap.c，初始化前边所述的 IDT 中的相应条目，确保用户环境是在启用中断的情况下运行。
基本思路是在 IDT 中加入相应的项，这里只要仿照之前在 lab3 中设置的代码。具体代码情况如下：

为实现确保每次用户程序在中断后启用，由于在运行 bootloader 时屏蔽了中断，这里只需要简单地设置 eflags 寄存器的 FL_IF 位就可以接收中断了，修改 env.c/env_alloc()函数如下：

```
// You will set e->env_tf.tf_eip later.

// Enable interrupts while in user mode.
// LAB 4: Your code here.
e->env_tf.tf_eflags |= FL_IF;
// Clear the page fault handler until user installs one.
e->env_pgfault_upcall = 0;
```

十六、练习 14

Exercise 14. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

解答：本次练习需要完成 trap_dispatch()，可以让 sched_yield()在发生时钟中断时调用查找并运行不同的用户环境。之后可以 user/spin 测试，父进程分离孩子 sys_yield（）几次之后在一个周期片后重新获得对 CPU 的控制，最后杀死子进程结束自己。
因此在函数中加入相应分支即可，需要先调用 lapic_eoi，作用是告诉 LAPIC 已经收到并调度中断了，于是 LAPIC 从中断请求队列中将其删除，这里可以不

需要 break，因为不会返回。具体代码如下：

```
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
        else {
                env_destroy(curenv);
=======
        switch(tf->tf_trapno) {
                //lab4 code here
            case IRQ_TIMER + IRQ_OFFSET:
                lapic_eoi();
                sched_yield();

            case (T_PGFLT):
                page_fault_handler(tf);
```

十七、练习 15

Exercise 15. Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. `user/primes` will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

解答：本次练习需要实现 kern/syscall.c 中的 sys_ipc_recv()，但是在线程中调用 envid2env 时，应当设置 checkperm 标志位为 0，以便允许用户环境向其他进程发送 IPC。然后 lib/ipc.c/ipc_recv 和 ipc_send,实现用 user/pingpong 和 user/primes 测试 IPC 通信机制。

1）sys_ipc_try_send（）函数：首先对参数进行一些检查，如果需要发送页面则建立相关映射，在设置完相关的值后，需要将目标进程 trapframe 中的 eax 设置为 0 以作为成功返回值（sys_ipc_recv 若成功由于放弃 CPU 所以不会直接返回），然后取消阻塞，设置状态为可运行的以接受调度

因为在 sys_page_map 函数中会对题目中的某些情况进行判断，这里不再判断。具体实现代码如下：

```
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        //panic("sys_ipc_try_send not implemented");
        struct Env *e;
        if (envid2env(envid, &e, 0) < 0) {
                return -E_BAD_ENV;
        }
        if (!(e->env_ipc_recving)) {
                return -E_IPC_NOT_RECV;
        }
        if (e->env_ipc_dstva && srcva && (uintptr_t) srcva < UTOP) {
                int r = sys_page_map(0, srcva, envid, e->env_ipc_dstva, perm);
                if (r < 0) {
                        return r;
                }
                e->env_ipc_perm = perm;
        } else {
                e->env_ipc_perm = 0;
        }
        e->env_ipc_value = value;
        e->env_ipc_from = curenv->env_id;

        e->env_tf.tf_regs.reg_eax = 0;
        e->env_ipc_recving = false;
        e->env_status = ENV_RUNNABLE;

        return 0;
}
```

2）sys_ipc_recv 函数：还是检查参数，然后阻塞进程，设置目标地址与进程状态，最后放弃 CPU 执行调度函数，还需要在 syscall() 函数中加入相应分支。具体代码如下：

```
sys_ipc_recv(void *dstva)
{
        // LAB 4: Your code here.
        //panic("sys_ipc_recv not implemented");
        if ((uintptr_t) dstva >= UTOP || (dstva && (uintptr_t) dstva % PGSIZE)) {
                return -E_INVAL;
        }
        curenv->env_ipc_recving = true;
        curenv->env_ipc_dstva = dstva;
        curenv->env_status = ENV_NOT_RUNNABLE;
        sched_yield();
        return 0;
}
```

3）ipc_recv 函数：根据传入的参数来设置值，具体代码如下：

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        //panic("ipc_recv not implemented");
        //return 0;
        int r;
        r = sys_ipc_recv(pg);
        if (from_env_store) {
                *from_env_store = r < 0 ? 0 : thisenv->env_ipc_from;
        }
        if (perm_store) {
                *perm_store = r < 0 ? 0 : thisenv->env_ipc_perm;
        }
        if (r < 0) {
                return r;
        }
        return thisenv->env_ipc_value;
}
```

4）ipc_send（）函数，作用是不断尝试发送消息，如果目标进程没有设置 ipc_recving 则继续发送，直至成功或者发生其他错误。具体代码如下：

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        //panic("ipc_send not implemented");
        while (1) {
                int r = sys_ipc_try_send(to_env, val, pg, perm);
                if (r < 0 && r != -E_IPC_NOT_RECV) {
                        panic("ipc_send: %e", r);
                }
                sys_yield();
                if (r == 0) {
                        break;
                }
        }
}
```

至此 lab4 全部完成，调试并 make grade 结果如下：