Exercise 3.1

完成env_init 函数。

实现Env 控制块的空闲队列和调度队列的初始化功能。请注意，你需要按倒序将所有控制块插入到空闲链表的头部，使得编号更小的进程控制块被优先分配。

```c
void env_init(void) {
    int i;
    /* Step 1: Initialize 'env_free_list' with 'LIST_INIT' and 'env_sched_list' with
     * 'TAILQ_INIT'. */
    /* Exercise 3.1: Your code here. (1/2) */
    LIST_INIT(&env_free_list);
    TAILQ_INIT(&env_sched_list);

    /* Step 2: Traverse the elements of 'envs' array, set their status to 'ENV_FREE' and insert
     * them into the 'env_free_list'. Make sure, after the insertion, the order of envs in the
     * list should be the same as they are in the 'envs' array. */

    /* Exercise 3.1: Your code here. (2/2) */
    for(i=NENV-1; i>=0; i--) {
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    }

    /*
     * We want to map 'UPAGES' and 'UENVS' to *every* user space with PTE_G permission (without
     * PTE_D), then user programs can read (but cannot write) kernel data structures 'pages' and
     * 'envs'.
     *
     * Here we first map them into the *template* page directory 'base_pgdir'.
     * Later in 'env_setup_vm', we will copy them into each 'env_pgdir'.
     */
    struct Page *p;
    panic_on(page_alloc(&p));
    p->pp_ref++;

    base_pgdir = (Pde *)page2kva(p);    //转内核虚拟地址。
    map_segment(base_pgdir, 0, PADDR(pages), UPAGES, ROUND(npage * sizeof(struct Page), BY2PG),
            PTE_G);
    map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct Env), BY2PG),
            PTE_G);  //在该页表中将内核数组pages和envs映射到了用户空间的UPAGES 和UENVS处。
}
```

Exercise 3.2

请你结合env_init 中的使用方式，完成map_segment 函数.

> 段地址映射函数void map_segment(Pde *pgdir, u_long pa, u_long va, u_long size,u_int perm)，功能是在一级页表基地址pgdir 对应的两级页表结构中做段地址映射，将虚拟地址段[va,va+size) 映射到物理地址段[pa,pa+size)，因为是按页映射，要求size 必须是页面大小的整数倍。同时为相关页表项的权限为设置为perm。

```c
/* Overview:
 *   Map [va, va+size) of virtual address space to physical [pa, pa+size) in the 'pgdir'. Use
 *   permission bits 'perm | PTE_V' for the entries.
 *
 * Pre-Condition:
 *   'pa', 'va' and 'size' are aligned to 'BY2PG'.
 */
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size, u_int perm) {

    assert(pa % BY2PG == 0);
    assert(va % BY2PG == 0);
    assert(size % BY2PG == 0);

    /* Step 1: Map virtual address space to physical address space. */
    for (int i = 0; i < size; i += BY2PG) {
        /*
         * Hint:
         *  Map the virtual page 'va + i' to the physical page 'pa + i' using 'page_insert'.
         *  Use 'pa2page' to get the 'struct Page *' of the physical address.
         */
        /* Exercise 3.2: Your code here. */
        page_insert(pgdir, asid, pa2page(pa+i), va+i, perm|PTE_V); //pa2page是将物理地址转page虚拟地址.pa是真实物理页框地址段
    }
}
```

Exercise 3.3

完成env_setup_vm 函数。
仔细阅读前文的提示理解一个进程虚拟地址空间的分布，根据注释完成函数，实现初始化一个新进程地址空间的功能。

```c
/* Overview:
 *   Initialize the user address space for 'e'.
 */
```

```c
static int env_setup_vm(struct Env *e) {
    /* Step 1:
     *   Allocate a page for the page directory with 'page_alloc'.
     *   Increase its 'pp_ref' and assign its kernel address to 'e->env_pgdir'.
     *
     * Hint:
     *   You can get the kernel address of a specified physical page using
'page2kva'.
     */
    struct Page *p;
    try(page_alloc(&p));
    /* Exercise 3.3: Your code here. */
    p->pp_ref++;
    e->env_pgdir = (Pde*)page2kva(p);    //env_pgdir : 这个字段保存了该进程页目录的内
核虚拟地址。用page2kva

    /* Step 2: Copy the template page directory 'base_pgdir' to 'e->env_pgdir'.
*/
    /* Hint:
     *   As a result, the address space of all envs is identical in [UTOP,
UVPT).
     *   See include/mmu.h for layout.
     */
    memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP),
           sizeof(Pde) * (PDX(UVPT) - PDX(UTOP)));

    /* Step 3: Map its own page table at 'UVPT' with readonly permission.
     * As a result, user programs can read its page table through 'UVPT' */
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
    return 0;
}
```

Exercise 3.4

完成env_alloc 函数。

env_alloc 函数实现了申请并初始化一个进程控制块的功能。这里给出如下提示：

1. 回忆Lab2 中的链表宏LIST_FIRST、LIST_REMOVE，实现在env_free_list 中申请空闲进程控制块。
2. 用env_setup_vm 初始化新进程的地址空间。
3. 仔细阅读前文中对与Lab3 相关的域的介绍，思考相关域的恰当赋值。

```c
/* Overview:
 *   Allocate and initialize a new env.
 *   On success, the new env is stored at '*new'.
 *
 * Pre-Condition:
 *   If the new env doesn't have parent, 'parent_id' should be zero.
 *   'env_init' has been called before this function.
 *
 * Post-Condition:
 *   return 0 on success, and basic fields of the new Env are set up.
 *   return < 0 on error, if no free env, no free asid, or 'env_setup_vm' failed.
 *
 * Hints:
 *   You may need to use these functions or macros:
 *     'LIST_FIRST', 'LIST_REMOVE', 'mkenvid', 'asid_alloc', 'env_setup_vm'
 *   Following fields of Env should be set up:
 *     'env_id', 'env_asid', 'env_parent_id', 'env_tf.regs[29]', 'env_tf.cp0_status',
 *     'env_user_tlb_mod_entry', 'env_runs'
 */
int env_alloc(struct Env **new, u_int parent_id) {
    int r;
    struct Env *e;

    /* Step 1: Get a free Env from 'env_free_list' */
    /* Exercise 3.4: Your code here. (1/4) */
    if(LIST_EMPTY(&env_free_list)) {
        return -E_NO_FREE_ENV;
    }
    e = LIST_FIRST(&env_free_list);   //从链表头部取一个

    /* Step 2: Call a 'env_setup_vm' to initialize the user address space for this new Env. */
    /* Exercise 3.4: Your code here. (2/4) */
    try(env_setup_vm(e));

    /* Step 3: Initialize these fields for the new Env with appropriate values:
     *   'env_user_tlb_mod_entry' (lab4), 'env_runs' (lab6), 'env_id' (lab3), 'env_asid' (lab3),
     *   'env_parent_id' (lab3)
     * Hint:
     *   Use 'asid_alloc' to allocate a free asid.
     *   Use 'mkenvid' to allocate a free envid.
     */
    e->env_user_tlb_mod_entry = 0; // for lab4
    e->env_runs = 0;           // for lab6
    /* Exercise 3.4: Your code here. (3/4) */
    e->env_id = mkenvid(e);       //制作进程新ID
    e->env_parent_id = parent_id;
    r = asid_alloc(&e->env_asid);   //判断asid是否成功分配。r是函数返回值，函数正常返回为0.
    if(r != 0){
        return r;
    }
    /* Step 4: Initialize the sp and 'cp0_status' in 'e->env_tf'. */
    // Timer interrupt (STATUS_IM4) will be enabled.
    e->env_tf.cp0_status = STATUS_IM4 | STATUS_KUp | STATUS_IEp;
    // Keep space for 'argc' and 'argv'.
    e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);  //设置栈指针。栈寄存器是29号

    /* Step 5: Remove the new Env from env_free_list. */
    /* Exercise 3.4: Your code here. (4/4) */
    LIST_REMOVE(e, env_link);

    *new = e;
    return 0;
}
```

Exercise 3.5

完成kern/env.c 中的load_icode_mapper 函数。

提示：可能使用到的函数有page_alloc，page_insert，memcpy.

```c
/* Overview:
 *   Load a page into the user address space of an env with permission 'perm'.
 *   If 'src' is not NULL, copy the 'len' bytes from 'src' into 'offset' at this page.
 *
 * Pre-Condition:
 *   'offset + len' is not larger than 'BY2PG'.
 *
 * Hint:
 *   The address of env structure is passed through 'data' from 'elf_load_seg', where this function
 *   works as a callback.
 *
 */
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src,
                 size_t len) {
    struct Env *env = (struct Env *)data;
    struct Page *p;
    int r;

    /* Step 1: Allocate a page with 'page_alloc'. */
    /* Exercise 3.5: Your code here. (1/2) */
    try(page_alloc(&p));    //给p分配空间，地址是page（虚拟地址）

    /* Step 2: If 'src' is not NULL, copy the 'len' bytes started at 'src' into 'offset' at this
     * page. */
    // Hint: You may want to use 'memcpy'.
    if (src != NULL) {
        /* Exercise 3.5: Your code here. (2/2) */
        memcpy(page2kva(p) + offset, src, len);  //page2kva(p)用于获取与页面p的物理地址对应的内核虚拟地址(KVA)

        /*在物理内存中，相邻的页面不一定是连续的。如果页面在物理内存中不是连续的，直接使用页面指针 p 加上 offset 得到的地址可能不是这个页面在虚拟地址空间中所对应的正确的内核虚拟地址，也可能导致访问到错误的物理内存页面。使用 page2kva(p) 可以保证得到的是正确的内核虚拟地址，以便正确访问页面所对应的物理内存中的数据。物理地址和虚拟地址之间的内存映射可能不是一对一的。换句话说，一个物理页面在不同的上下文或不同的时间可能映射到不同的虚拟地址。通过使用page2kva()，代码确保在当前上下文中使用正确的虚拟地址来访问页面。 */
    }

    /* Step 3: Insert 'p' into 'env->env_pgdir' at 'va' with 'perm'. */
    return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
}
```

Exercise 3.6

根据注释的提示，完成kern/env.c 中的load_icode 函数。

> 这个函数通过调用elf_load_seg 函数来将ELF 文件真正加载到内存中，将load_icode_mapper这个函数作为参数传入。

```
/* Overview:
 *   Load program segments from 'binary' into user space of the env 'e'.
 *   'binary' points to an ELF executable image of 'size' bytes, which contains both text and data
 *   segments.
 */
static void load_icode(struct Env *e, const void *binary, size_t size) {
    /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
    const Elf32_Ehdr *ehdr = elf_from(binary, size);
    if (!ehdr) {
        panic("bad elf at %x", binary);
    }

    /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and 'elf_load_seg'.
     * As a loader, we just care about loadable segments, so parse only program headers here.
     */
    size_t ph_off;
    ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
        Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
        if (ph->p_type == PT_LOAD) {
            // 'elf_load_seg' is defined in lib/elfloader.c
            // 'load_icode_mapper' defines the way in which a page in this segment
            // should be mapped.
            panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e));
        }
    }

    /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */ /*设置'e->env_tf.cp0_epc'的内容为'ehdr->e_entry'*/
    /* Exercise 3.6: Your code here. */
    e->env_tf.cp0_epc = ehdr->e_entry;  //这里env_tf.cp0_epc字段指示了进程恢复运行时PC 应恢复到的位置。我们要运行的进
                                        //程的代码段预先被载入到了内存中，且程序入口为e_entry，
}
```

Exercise 3.7

完成env_create 函数。

根据提示，理解并恰当使用前面实现的函数，完成kern/env.c 中env_create 函数的填写，实现创建一个新进程的功能。

```
/* Overview:
 *   Create a new env with specified 'binary' and 'priority'.
 *   This is only used to create early envs from kernel during initialization, before the
 *   first created env is scheduled.
 *
 * Hint:
 *   'binary' is an ELF executable image in memory.
 */
struct Env *env_create(const void *binary, size_t size, int priority) {
    struct Env *e;
    /* Step 1: Use 'env_alloc' to alloc a new env, with 0 as 'parent_id'. */
    /* Exercise 3.7: Your code here. (1/3) */
    env_alloc(&e, 0);    //分配一个新的ENV结构体，注意函数参数设置。

    /* Step 2: Assign the 'priority' to 'e' and mark its 'env_status' as runnable. */
    /* Exercise 3.7: Your code here. (2/3) */
    e->env_pri = priority;
    e->env_status = ENV_RUNNABLE;

    /* Step 3: Use 'load_icode' to load the image from 'binary', and insert 'e' into
     * 'env_sched_list' using 'TAILQ_INSERT_HEAD'. */
    /* Exercise 3.7: Your code here. (3/3) */
    load_icode(e, binary, size);
    TAILQ_INSERT_HEAD(&env_sched_list, e, env_sched_link);  //注意参数若要求指针，对于数要传入地址。

    return e;
}
```

Exercise 3.8

完成env_run。 仔细阅读前文讲解，并根据注释填写kern/env.c 中的env_run 函数。

```c
/* Overview:
 *   Switch CPU context to the specified env 'e'.
 *
 * Post-Condition:
 *   Set 'e' as the current running env 'curenv'.
 *
 * Hints:
 *   You may use these functions: 'env_pop_tf'.
 */
void env_run(struct Env *e) {
    assert(e->env_status == ENV_RUNNABLE);
    pre_env_run(e); // WARNING: DO NOT MODIFY THIS LINE!

    /* Step 1:
     *   If 'curenv' is NULL, this is the first time through.
     *   If not, we may be switching from a previous env, so save its context into
     *   'curenv->env_tf' first.
     */
    if (curenv) {
        curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);  //curenv->env_tf是当前进程的上下文所存放的区域。我们将把KSTACKTOP 之下的Trapframe 拷贝到当前进程的env_tf中，以达到保存进程上下文目的。
    }

    /* Step 2: Change 'curenv' to 'e'. */
    curenv = e;
    curenv->env_runs++; // lab6

    /* Step 3: Change 'cur_pgdir' to 'curenv->env_pgdir', switching to its address space. */
    /* Exercise 3.8: Your code here. (1/2) */
    cur_pgdir=curenv->env_pgdir;    //切换当前进程页目录地址。

    /* Step 4: Use 'env_pop_tf' to restore the curenv's saved context (registers) and return/go
     * to user mode.
     *
     * Hint:
     *  - You should use 'curenv->env_asid' here.
     *  - 'env_pop_tf' is a 'noreturn' function: it restores PC from 'cp0_epc' thus not
     *    returning to the kernel caller, making 'env_run' a 'noreturn' function as well.
     */
    /* Exercise 3.8: Your code here. (2/2) */
    env_pop_tf(&curenv->env_tf, curenv->env_asid);  //调用env_pop_tf函数，恢复现场、异常返回
}
```

Exercise 3.9

补充kern/entry.S。 理解异常分发代码，并将异常分发代码填至kern/entry.S 恰当的部分。

```
#include <asm/asm.h>
#include <stackframe.h>

.section .text.tlb_miss_entry
tlb_miss_entry:
    j       exc_gen_entry

.section .text.exc_gen_entry
exc_gen_entry:
    SAVE_ALL     #SAVE_ALL 宏将当前上下文保存到内核的异常栈中。
/* Exercise 3.9: Your code here. */
    mfc0 t0, CP0_CAUSE       #将Cause 寄存器的 内容拷贝到t0 寄存器中
    andi t0, 0x7c            #取得Cause 寄存器中的2~6 位，也就是对应的异常码，
    lw   t0, exception_handlers(t0)    #以得到的异常码作为索引在exception_handlers 数组中找到对应的中断处理函数
    jr   t0      #跳转到对应的中断处理函数中，从而响应了异常,并将异常交给了对应的异常处理函数去处理。
```

Exercise 3.10

补全kernel.lds。根据前文讲解将kernel.lds 代码补全使得异常发生后可以跳到异常分发代码.

```
/*
 * Set the architecture to mips.
 */
OUTPUT_ARCH(mips)
/*
 * Set the ENTRY point of the program to _start.
 */
ENTRY(_start)

SECTIONS {
    /* Exercise 3.10: Your code here. */
    . = 0x80000000;
    .tlb_miss_entry : {
        *(.text.tlb_miss_entry)
    }

    . = 0x80000080;
    .exc_gen_entry : {
        *(.text.exc_gen_entry)
    }
/*.text.exc_gen_entry 段和.text.tlb_miss_entry 段需要被链接器放到特定的位置。在R3000中，这两个段分别要求放到地址0x80000080 和0x80000000 处，它们是异常处理程序的入口地址。在我们的系统中，CPU 发生异常（除了用户态地址的TLB Miss 异常）后，就会自动跳转到地址0x80000080处；发生用户态地址的TLB Miss 异常时，会自动跳转到地址0x80000000 处*/

    /* fill in the correct address of the key sections: text, data, bss. */
    /* Exercise 1.2: Your code here. */
    . = 0x80010000;
    .text : { *(.text) }
    .data : { *(.data) }
    .bss : { *(.bss) }

    bss_end = .;
    . = 0x80400000;
    end = . ;
}
```

Exercise 3.11

补充kclock_init 函数。 通过上面的描述，补充kern/kclock.S 中的kclock_init 函数。

```
#include <asm/asm.h>
#include <drivers/dev_rtc.h>
#include <kclock.h>
#include <mmu.h>

LEAF(kclock_init)
    li      t0, 200 // the timer interrupt frequency in Hz

    /* Write 't0' into the timer (RTC) frequency register.
     *
     * Hint:
     *   You may want to use 'sw' instruction and constants 'DEV_RTC_ADDRESS' and
     *   'DEV_RTC_HZ' defined in include/drivers/dev_rtc.h.
     *   To access device through mmio, a physical address must be converted to a
     *   kseg1 address.
     *
     * Reference: http://gavare.se/gxemul/gxemul-stable/doc/experiments.html#expdevices
     */
    /* Exercise 3.11: Your code here. */
    sw      t0, (KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_HZ)  //kern/kclock.S 中的kclock_init 函数完成了时钟中断的初始化，该函数向KSEG1 | DEV_RTC_ADDRESS| DEV_RTC_HZ 位置写入200，其中KSEG1 | DEV_RTC_ADDRESS 是模拟器（GXemul）映射实时钟的位置。偏移量为DEV_RTC_HZ 表示设置实时钟中断的频率。

    jr      ra
END(kclock_init)
```

Exercise 3.12

完成schedule 函数。 根据注释，填写kern/sched.c 中的schedule 函数实现切换进程的功能，使得进程能够被正确调度。此外，这里也给出如下提示：

1. 使用静态变量来存储当前进程剩余执行次数；
2. 调度队列中存在且只存在所有就绪（状态为ENV_RUNNABLE）的进程。

```c
#include <env.h>
#include <pmap.h>
#include <printk.h>

/* Overview:
 *   Implement a round-robin scheduling to select a runnable env and schedule it using 'env_run'.
 *
 * Post-Condition:
 *   If 'yield' is set (non-zero), 'curenv' should not be scheduled again unless it is the only
 *   runnable env.
 *
 * Hints:
 *   1. The variable 'count' used for counting slices should be defined as 'static'.
 *   2. Use variable 'env_sched_list', which contains and only contains all runnable envs.
 *   3. You shouldn't use any 'return' statement because this function is 'noreturn'.
 */
void schedule(int yield) {
    static int count = 0; // remaining time slices of current env
    struct Env *e = curenv;

    /* We always decrease the 'count' by 1.
     *
     * If 'yield' is set, or 'count' has been decreased to 0, or 'e' (previous 'curenv') is
     * 'NULL', or 'e' is not runnable, then we pick up a new env from 'env_sched_list' (list of
     * all runnable envs), set 'count' to its priority, and schedule it with 'env_run'. **Panic
     * if that list is empty**.
     *
     * (Note that if 'e' is still a runnable env, we should move it to the tail of
     * 'env_sched_list' before picking up another env from its head, or we will schedule the
     * head env repeatedly.)
     *
     * Otherwise, we simply schedule 'e' again.
     *
     * You may want to use macros below:
     *   'TAILQ_FIRST', 'TAILQ_REMOVE', 'TAILQ_INSERT_TAIL'
     */
    /* Exercise 3.12: Your code here. */
    if(yield || (count == 0) || (e == NULL) || (e->env_status != ENV_RUNNABLE))
{
        if(e && e->env_status == ENV_RUNNABLE) {
            TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
            TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
        }
        if(TAILQ_EMPTY(&env_sched_list)) {
            panic("no runnable envs");
        }
        e = TAILQ_FIRST(&env_sched_list);
        count = e->env_pri;
    }
    count--;
    env_run(e);

}
```