实　　验　　报　　告

学　　号	21030031009	姓　　名	惠欣宇	专业班级	计算机 4 班
课程名称	**操作系统**			学期	**2023 年秋季学期**
任课教师	窦金凤	完成日期	2023.12.15	上机课时间	周五 3-6
实 验 名 称	**进程与异常**				

Exercise 部分：

Exercise 3.1

> **Exercise 3.1**
> - 修改 pmap.c/mips_vm_init 函数来为 envs 数组分配空间。
> - envs 数组包含 NENV 个 Env 结构体成员，你可以参考 pmap.c 中已经写过的 **pages** 数组空间的分配方式。
> - 除了要为数组 envs 分配空间外，你还需要使用 pmap.c 中你填写过的一个内核态函数为其进行段映射，envs 数组应该被 **UENVS** 区域映射，你可以参考**./include/mmu.h**。

实现了 MIPS 架构下操作系统的虚拟内存管理的初始化过程：

分配页目录：

使用 alloc 函数分配一个页面大小的内存，用于存储页目录（页表的第一级）。

设置全局页目录指针：

将 boot_pgdir 指针指向刚刚分配的页目录，以备后续引导映射使用。

分配并映射 pages 数组：

通过 alloc 函数分配足够容纳 npage 个 struct Page 的内存，npage 表示物理页面数量。
将分配的内存通过 boot_map_segment 函数映射到虚拟地址空间中的 UPAGES 位置，同时设置了相应的权限标志（PTE_R 表示可读）。
pages 数组用于存储物理页面的信息，通过虚拟地址 UPAGES 访问时，可以得到相应的物理页面的信息。

分配并映射 envs 数组：

通过 alloc 函数分配足够容纳 NENV 个 struct Env 的内存，NENV 表示进程环境的数量。
将分配的内存通过 boot_map_segment 函数映射到虚拟地址空间中的 UENVS 位置，同时设置了相应的权限标志（PTE_R 表示可读）。
envs 数组用于存储进程环境的信息，通过虚拟地址 UENVS 访问时，可以得到相应的进程环境的信息。
mips_vm_init 函数实现如图 1 所示。

```c
void mips_vm_init()
{
    extern char end[];
    extern int mCONTEXT;
    extern struct Env *envs;

    Pde *pgdir;
    u_int n;

    /* Step 1: Allocate a page for page directory(first level page table). */
    pgdir = alloc(BY2PG, BY2PG, 1);
    printf("to memory %x for struct page directory.\n", freemem);
    mCONTEXT = (int)pgdir;

    boot_pgdir = pgdir;

    /* Step 2: Allocate proper size of physical memory for global array `pages`,
     * for physical memory management. Then, map virtual address `UPAGES` to
     * physical address `pages` allocated before. In consideration of alignment,
     * you should round up the memory size before map. */
    pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
    printf("to memory %x for struct Pages.\n", freemem);
    n = ROUND(npage * sizeof(struct Page), BY2PG);
    boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);

    /* Step 3, Allocate proper size of physical memory for global array `envs`,
     * for process management. Then map the physical address to `UENVS`. */
    envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
    n = ROUND(NENV * sizeof(struct Env), BY2PG);
    boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);

    printf("pmap.c:\t mips vm init success\n");
}
```

图 1　mips_vm_init 函数实现

Exercise 3.2

> Exercise 3.2　仔细阅读注释，填写 **env_init** 函数，注意链表插入的顺序 (函数位于 lib/env.c 中)。 ■

初始化环境（进程环境）的管理。

初始化环境空闲列表：创建一个空链表 env_free_list，用于存放空闲的环境结构。

初始化两个调度列表：创建两个空链表 env_sched_list[0] 和 env_sched_list[1]，用于存放处于不同调度状态的环境结构。

遍历 envs 数组，设置环境状态并插入到空闲列表：从数组的末尾开始，逆序遍历 envs 数组。对于每个环境结构，设置其状态为 ENV_FREE，表示空闲状态。然后，将其插入到 env_free_list 列表的头部，以保持链表中环境结构的顺序与数组中的顺序一致。

这样，完成了环境管理结构的初始化，使得空闲的环境结构都被添加到了 env_free_list 列表中，以备后续使用。

填写结果如图 2 所示。

```c
void env_init(void)
{
    int i;
    /* Step 1: Initialize env_free_list. */
    LIST_INIT(&env_free_list);
    LIST_INIT(&env_sched_list[0]);
    LIST_INIT(&env_sched_list[1]);
    /* Step 2: Traverse the elements of 'envs' array,
     *   set their status as free and insert them into the env_free_list.
     * Choose the correct loop order to finish the insertion.
     * Make sure, after the insertion, the order of envs in the list
     *   should be the same as that in the envs array. */
    for(i = NENV -1; i >= 0; i-- ){
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    }
}
```

图 2　env_init 函数实现

Exercise 3.3

> Exercise 3.3 仔细阅读注释，完成 env.c/envid2env 函数，实现通过一个 env 的 id 获取该 id 对应的进程控制块的功能。

将环境 ID（envid）映射到对应的环境结构指针的函数。主要步骤如下：

根据 envid 获取对应的环境结构指针：
如果 envid 为 0，表示获取当前环境（curenv），直接将指针指向 curenv，并返回 0 表示成功。
否则，通过计算 envs + ENVX(envid)，找到对应 envid 的环境结构。

检查环境状态和权限：
如果获取的环境状态为 ENV_FREE，或者环境 ID 不匹配，则返回错误码 -E_BAD_ENV 表示无效的环境。
如果需要检查权限（checkperm 为真），则进行权限检查：
如果当前环境不是目标环境（除非 envid 为 0 表示获取当前环境），且目标环境的父环境 ID 不等于当前环境的 ID，也返回错误码 -E_BAD_ENV。

返回环境结构指针：
将获取到的合法环境结构指针赋值给*penv，返回 0 表示成功。
代码如图 3 所示。

```c
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;
    /* Step 1: Assign value to e using envid. */
    if(envid == 0){
        *penv = curenv;
        return 0;
    }
    e = envs + ENVX(envid);
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *penv = 0;
        return -E_BAD_ENV;
    }
    /*  Step 2: Make a check according to checkperm. */
    if(checkperm){
        if(e != curenv && e->env_parent_id != curenv->env_id){
            *penv = 0;
            return -E_BAD_ENV;
        }
    }
    *penv = e;
    return 0;
}
```

图 3　envid2env 函数实现

Exercise 3.4

Exercise 3.4 仔细阅读注释，填写 env_setup_vm 函数

为给定的环境（struct Env *e）设置虚拟内存环境的函数。主要步骤如下：
分配页面用于页目录（page directory）：
调用 page_alloc 函数分配一个页面，并将其引用计数 pp_ref 加一。
将页面的虚拟地址转换为页目录的虚拟地址 pgdir。
清零页目录的 UTOP 以下的所有字段：
遍历页目录数组，将 UTOP 以下的所有项置零。
复制内核的 boot_pgdir 到 pgdir：
遍历页目录数组，将 UTOP 以上（不包括 UVPT 所在的项）的项复制自内核的 boot_pgdir。
设置环境的页目录指针 env_pgdir 指向新创建的页目录。
设置环境的页目录项（PDX(UVPT)）：
将环境的页目录物理地址 env_cr3 存储在页目录项 PDX(UVPT) 中，以便实现页表自映射。
设置环境的 env_pgdir 和 env_cr3：
将页目录指针 pgdir 存储在环境结构中的 env_pgdir。
将页目录的物理地址存储在环境结构中的 env_cr3。
设置 UVPT 页目录项（PDX(UVPT)）：
将 env_cr3 与 PTE_V 位进行或操作，然后存储在 env_pgdir 中的 PDX(UVPT) 位置。
返回成功标志：
返回 0 表示成功。

代码如图 4 所示。

```c
static int env_setup_vm(struct Env *e)
{
    int i, r;
    struct Page *p = NULL;
    Pde *pgdir;
    /* Step 1: Allocate a page for the page directory
     *   using a function you completed in the lab2 and add its pp_ref.
     *   pgdir is the page directory of Env e, assign value for it. */
    r = page_alloc(&p);
    if ( r != 0 ) {
        panic("env_setup_vm - page alloc error\n");
        return r;
    }
    p->pp_ref++;
    pgdir =(Pde *)page2kva(p);
    /* Step 2: Zero pgdir's field before UTOP. */
    for(i = 0; i<PDX(UTOP); i++){
        pgdir[i] = 0;
    }
    /* Step 3: Copy kernel's boot_pgdir to pgdir. */
    /* Hint:
     *  The VA space of all envs is identical above UTOP
     *  (except at UVPT, which we've set below).
     *  See ./include/mmu.h for layout.
     *  Can you use boot_pgdir as a template?
     */
    for(i = PDX(UTOP); i < PTE2PT; i++){
        if(i != PDX(UVPT)){
            pgdir[i] = boot_pgdir[i];
        }
    }
    e->env_pgdir = pgdir;
    e->env_cr3 = PADDR(pgdir);
    e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V;
    return 0;
}
```

图 4　env_setup_vm 函数实现

Exercise 3.5

Exercise 3.5 根据上面的提示与代码注释，填写 **env_ alloc** 函数。 ■

用于分配一个新的环境结构的函数 env_alloc。主要步骤如下：
检查是否存在空闲环境：
使用 LIST_EMPTY 宏检查空闲环境链表 env_free_list 是否为空，如果为空，表示没有可用的空闲环境结构。
如果链表为空，将 *new 设置为 0 并返回错误码 -E_NO_FREE_ENV 表示没有可用的环境。

从空闲环境链表中取出一个环境结构：
使用 LIST_FIRST 宏从空闲环境链表中取出第一个环境结构。
为该环境结构生成一个唯一的环境 ID，通过调用 mkenvid 函数。
设置父环境 ID 为传入的参数 parent_id。
设置环境状态为 ENV_RUNNABLE 表示可运行。
将环境运行次数 env_runs 置零。

初始化环境的 Trapframe（tf）：
将 cp0_status 字段设置为默认值 0x1000100c。
将寄存器 regs[29] 设置为用户栈的顶部 USTACKTOP。

从空闲环境链表中移除该环境结构：

使用 LIST_REMOVE 宏将该环境结构从空闲环境链表中移除。

将分配的环境结构指针存储在 *new 中：
将分配的环境结构指针存储在传入的 *new 中。

返回成功标志：
返回 0 表示成功。

代码如图 5 所示。

```
207 int env_alloc(struct Env **new, u_int parent_id)
208 {
209     int r;
210     struct Env *e;
211     if(LIST_EMPTY(&env_free_list)){
212         *new = 0;
213         return -E_NO_FREE_ENV;
214     }
215     e = LIST_FIRST(&env_free_list);
216     e->env_id = mkenvid(e);
217     e->env_parent_id = parent_id;
218     e->env_status = ENV_RUNNABLE;
219     e->env_runs = 0;
220 
221     e->env_tf.cp0_status = 0x1000100c;
222     e->env_tf.regs[29] = USTACKTOP;
223 
224     LIST_REMOVE(e, env_link);
225     *new = e;
226     return 0;
227 }
```

图 5　env_alloc 函数实现

Exercise 3.6

Exercise 3.6 通过上面补充的知识与注释，填充 **load_icode_mapper** 函数。 ■

用于加载程序代码和数据到用户环境的函数 load_icode_mapper。主要目的是将程序的二进制数据加载到指定的虚拟地址 va 开始的内存中。以下是代码的主要步骤：
初始化变量：
定义指向用户环境的结构体指针 env，并将 user_data 强制类型转换为 struct Env * 类型。指向页表项的指针 p。变量 i 用于迭代二进制数据。变量 offset 表示 va 到 ROUNDDOWN(va, BY2PG) 的偏移量。变量 size 表示当前需要加载的数据大小。

加载数据到内存：
首先处理不对齐的部分（offset 部分）：
如果存在偏移量 offset，则通过 page_lookup 函数查找对应虚拟地址 va+i 的页表项，并分配一个新的物理页面（如果不存在的话）。
将数据复制到该页面的偏移位置，保证对齐。
更新 i 和 offset。

循环加载数据：
在循环中，通过 page_alloc 分配新的物理页面，并将其插入到页表中，直到加载完整个程序二进制数据。
使用 bcopy 函数将二进制数据复制到物理页面中。
更新 i。

处理超出 bin_size 部分：
处理余下的部分，保证 sgsize 大小的内存空间被分配。
如果存在偏移量 offset，则处理方式类似于上面的不对齐部分。
循环分配物理页面，保证 sgsize 大小的内存被分配。
使用 bzero 函数清零余下的部分。

返回结果：
返回 0 表示加载成功。

代码如下所示。

```
static int load_icode_mapper(u_long va, u_int32_t sgsize, u_char *bin, u_int32_t bin_size, void *user_data)
{
	struct Env *env = (struct Env *)user_data;
	struct Page *p = NULL;
	u_long i = 0;
	int r;
	u_long offset = va - ROUNDDOWN(va, BY2PG);
	int size = 0;
	}
	if (offset)
	{
		p = page_lookup(env->env_pgdir, va + i, NULL);
		if(p == 0)
		{
			r = page_alloc(&p);
			if (r != 0)
			{
				return r;
			}
			page_insert(env->env_pgdir, p, va + i, PTE_R);
		}
		size = MIN(bin_size - i, BY2PG - offset);
		int sizeTemp = size;
		for(;(page2kva(p)+offset) % 4 != 0 && sizeTemp > 0; sizeTemp--,offset++,i++){
			*(char *)(page2kva(p)+offset) = *(char *)(bin + i);
		}
		bcopy((void*)bin, (void*)(page2kva(p) + offset), sizeTemp);
		i = i + sizeTemp;
	}
	/*Step 1: load all content of bin into memory. */
	while (i < bin_size)
	{
		/* Hint: You should alloc a page and increase the reference count of it. */
		size = MIN(BY2PG, bin_size - i);
		r = page_alloc(&p);
		if (r != 0)
		{
			return r;
		}
		page_insert(env->env_pgdir, p ,va + i, PTE_R);
		bcopy((void*)bin + i, (void*)(page2kva(p)), size);
		i += size;
	}
	/*Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
	* i has the value of `bin_size` now. */
	offset = va + i - ROUNDDOWN(va + i, BY2PG);
```

```
    if (offset)
    {
        p = page_lookup(env->env_pgdir, va + i, NULL);
        if(p == 0)
        {
            r = page_alloc(&p);
            if (r != 0)
            {
                return r;
            }
            page_insert(env->env_pgdir, p, va + i, PTE_R);
        }
        size = MIN(sgsize - i, BY2PG - offset);
        int sizeTemp = size;
        for(;(page2kva(p) + offset)%4 != 0 && sizeTemp > 0 ;*(char *)(page2kva(p) + offset)=0,offset++,sizeTemp--);
        bzero((void*)(page2kva(p) + offset), sizeTemp);
        i = i + size;
    }
    while (i < sgsize) {
        size = MIN(BY2PG, sgsize - i);
        r = page_alloc(&p);
        if (r != 0)
        {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        int sizeTemp = size;
        i += size;
    }
    return 0;
}
```

Exercise 3.7

> **Exercise 3.7** 通过补充的 ELF 知识与注释，填充 **load_elf** 函数和 **load_icode** 函数。 ■

函数 load_icode 是用来加载用户程序二进制数据到用户环境。为新创建的用户环境设置初始的堆栈，并通过 ELF 加载器加载用户程序的代码和数据。以下是代码的主要步骤：

分配物理页面：
使用 page_alloc 函数分配一个新的物理页面，用于加载用户程序的数据。
如果分配失败，直接返回。

设置初始堆栈：
使用 PTE_V | PTE_R 权限位创建一个新的页表项，并通过 page_insert 将物理页面插入到用户环境的页表中，设置在用户环境的栈顶 USTACKTOP - BY2PG。
如果插入失败，直接返回。

使用 ELF 加载器加载用户程序：
调用 load_elf 函数，传递用户程序的二进制数据、大小、入口点地址 entry_point，用户环境指针 (void *)e 以及加载函数 load_icode_mapper。
如果加载 ELF 失败（返回值小于 0），直接返回。

设置 CPU 的 PC 寄存器：
将用户环境的 env_tf.pc 寄存器设置为 ELF 加载器返回的入口点地址 entry_point。

代码如图 6 所示。

```
336 static void load_icode(struct Env *e, u_char *binary, u_int size)
337 {
338     /* Hint:
339      *  You must figure out which permissions you'll need
340      *  for the different mappings you create.
341      *  Remember that the binary image is an a.out format image,
342      *  which contains both text and data.
343      */
344     struct Page *p = NULL;
345     u_long entry_point;
346     u_long r;
347     u_long perm;
348     /* Step 1: alloc a page. */
349     r = page_alloc(&p);
350     if(r != 0){
351         return;
352     }
353     /* Step 2: Use appropriate perm to set initial stack for new Env. */
354     /* Hint: Should the user-stack be writable? */
355     perm = PTE_V | PTE_R;
356     r = page_insert(e->env_pgdir, p, USTACKTOP - BY2PG, perm);
357     if(r != 0){
358         return;
359     }
360     /* Step 3: load the binary using elf loader. */
361     r = load_elf(binary,size, &entry_point, (void* )e, load_icode_mapper);
362     if(r < 0){
363         return;
364     }
365     /* Step 4: Set CPU's PC register as appropriate value. */
366     e->env_tf.pc = entry_point;
367 }
```

图 6　load_icode 函数实现

Exercise 3.8

> Exercise 3.8 根据提示，完成 **env_create** 函数与 **env_create_priority** 的填写。
> ■

创建新的用户环境，并加载给定的 ELF 二进制数据。以下是它们的主要步骤：
env_create_priority 函数：
分配新的用户环境：
调用 env_alloc 函数，分配一个新的用户环境 e。
如果分配失败，直接返回。

为新环境分配优先级：
将 priority 参数赋值给新创建的用户环境的 env_pri 字段，表示其优先级。

使用 load_icode 函数加载 ELF 二进制数据：
调用 load_icode 函数，传递用户程序的二进制数据、大小和用户环境指针。
如果加载失败，直接返回。

将新环境插入到 env_sched_list[0] 中：
使用 LIST_INSERT_TAIL 将新创建的用户环境插入到就绪队列 env_sched_list[0] 的尾部。

env_create 函数：
这个函数是对 env_create_priority 函数的封装，为用户环境指定默认优先级值。
调用 env_create_priority 函数：
调用 env_create_priority 函数，传递用户程序的二进制数据和大小，同时将优先级设置为 1。

代码如图 7、8 所示。

```
406 void env_create(u_char *binary, int size)
407 {
408     /* Step 1: Use env_create_priority to alloc a new env with priority 1 */
409     env_create_priority(binary, size, 1);
410 }
```

图 7　env_create 函数实现

```
379 void env_create_priority(u_char *binary, int size, int priority)
380 {
381     struct Env *e;
382     /* Step 1: Use env_alloc to alloc a new env. */
383     int r;
384     r = env_alloc(&e,0);
385     if(r != 0){
386         return;
387     }
388     /* Step 2: assign priority to the new env. */
389     e->env_pri = priority;
390     /* Step 3: Use load_icode() to load the named elf binary,
391        and insert it into env_sched_list using LIST_INSERT_HEAD. */
392     load_icode(e, binary, size);
393     LIST_INSERT_TAIL(&env_sched_list[0],e,env_sched_link);
394 }
```

图 8　env_create_priority 函数实现

Exercise 3.9

Exercise 3.9 根据注释与理解，将上述两条进程创建命令加入 **init/init.c** 中。

加入结果如图 9 所示。

```
26 //  kclock_init();
27     ENV_CREATE_PRIORITY(user_A, 2);
28     ENV_CREATE_PRIORITY(user_B, 1);
```

图 9　加入进程创建命令

Exercise 3.10

Exercise 3.10 根据补充说明，填充完成 **env_run** 函数。

切换到一个新的用户环境，并执行该用户环境的代码。以下是该函数的主要步骤：
保存当前环境的寄存器状态：
如果当前有正在运行的环境 (curenv != NULL)，则保存当前环境的寄存器状态。
使用 TIMESTACK 找到当前环境的陷阱帧 (struct Trapframe)，将其内容复制到当前环境的 env_tf 中。
设置当前环境的程序计数器 (env_tf.pc) 为当前环境的异常程序计数器 (env_tf.cp0_epc)

将 curenv 设置为新环境：
将全局变量 curenv 指向新的用户环境 e。

增加新环境的运行计数器 (env_runs)。

使用 lcontext 切换地址空间：
调用 lcontext 函数，将当前的地址空间切换为新环境的地址空间。这样，CPU 将开始使用新环境的页表。

使用 env_pop_tf 恢复寄存器状态：
调用 env_pop_tf 函数，将新环境的寄存器状态弹出并返回到用户模式。

代码如图 10 所示。

```
491 void env_run(struct Env *e)
492 {
493     /* Step 1: save register state of curenv. */
494     /* Hint: if there is an environment running,
495      *   you should switch the context and save the registers.
496      *   You can imitate env_destroy() 's behaviors.*/
497     if(curenv != NULL){
498 
499         struct Trapframe *old;
500         old = (struct Trapframe * )(TIMESTACK - sizeof(struct Trapframe));
501         bcopy((void *)old, (void *)(&(curenv->env_tf)), sizeof(struct Trapframe));
502         curenv->env_tf.pc = curenv->env_tf.cp0_epc;
503     }
504 
505     /* Step 2: Set 'curenv' to the new environment. */
506     curenv = e;
507     curenv->env_runs++;
508 
509     /* Step 3: Use lcontext() to switch to its address space. */
510     lcontext((u_int)(curenv->env_pgdir));
511 
512     /* Step 4: Use env_pop_tf() to restore the environment's
513      *   environment   registers and return to user mode.
514      *
515      * Hint: You should use GET_ENV_ASID there. Think why?
516      *   (read <see mips run linux>, page 135-144)
517      */
518     env_pop_tf(&(curenv->env_tf), GET_ENV_ASID(curenv->env_id));
519 }
```

图 10　env_run 函数填充

Exercise 3.11

> **Exercise 3.11** 将异常分发代码填入 **boot/start.S** 合适的部分。 ■

将异常分发代码填入 start.S 中，如下所示：

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
    .set noat
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE
```

```
        la k0,exception_handlers
        andi k1,0x7c
        addu k0,k1
        lw k0,(k0)
        nop
        jr k0
        nop
END(except_vec3)
.set at
.data
                .globl mCONTEXT
mCONTEXT:
                .word 0
                .globl delay
delay:
                .word 0
                .globl tlbra
tlbra:
                .word 0
                .section .data.stk
KERNEL_STACK:
                .space 0x8000
                .text
LEAF(_start)
        .set    mips2
        .set    reorder
        /* Disable interrupts */
        mtc0    zero, CP0_STATUS
        /* Disable watch exception. */
        mtc0    zero, CP0_WATCHLO
        mtc0    zero, CP0_WATCHHI
        /* disable kernel mode cache */
        mfc0    t0, CP0_CONFIG
        and     t0, ~0x7
        ori     t0, 0x2
        mtc0    t0, CP0_CONFIG
        /* set up stack */
        li      sp, 0x80400000
        li      t0,0x80400000
        sw      t0,mCONTEXT
        /* jump to main */
        jal     main
loop:
        j       loop
        nop
END(_start)
```

Exercise 3.12

Exercise 3.12 将 lds 代码补全使得异常后可以跳到异常分发代码。 ■

补全 lds 代码之后的结果：

```
OUTPUT_ARCH(mips)
ENTRY(_start)

SECTIONS
{
```

```
    . = 0x80000080;
    .except_vec3 : {
        *(.text.exc_vec3)
    }

    . = 0x80010000;

  /*** exercise 3.13 ***/

  .text : {
        *(.text)
    }

  .bss   : {
   *(.bss)
   }

  .data : {
    *(.data)
    }

  .sdata : {
    *(.sdata)
  }

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

Exercise 3.13

Exercise 3.13 通过上面的描述，补充 **kclock_init** 函数。

启动计时器，如图 11：

```
12 void kclock_init(void)
13 {
14     // hint: use set_timer()
15     set_timer();
16 }
```

图 11　kclock_init 函数实现

Exercise 3.14

Exercise 3.14 根据注释，完成 **sched_yield** 函数的补充，并根据调度方法对 env.c 中的部分函数进行修改，使得进程能够被正确调度。

实现协作式多任务调度，即切换到另一个可运行的用户环境。以下是该函数的主要步骤：
初始化静态变量：
静态变量 count 表示当前环境还需要运行的时间片数。
静态变量 point 用于轮转调度，表示当前使用哪个环境链表。

获取当前环境 e：
使用静态变量 e 保存当前正在运行的环境 curenv。
如果 count 为 0 或当前环境不可运行，执行轮转调度。

执行轮转调度：
如果当前环境 e 存在且可运行，则将其从当前环境链表中移除，并插入到另一个环境链表中。
在另一个环境链表中查找下一个可运行的环境，并将其设置为新的运行环境。
如果没有可运行的环境，继续进行轮转调度，直到找到可运行的环境。

执行环境切换：
将 count 减少一个时间片。
调用 env_run 函数切换到新的运行环境。

代码如图 12 所示。

```
16 void sched_yield(void)
17 {
18     static int count = 0;
19     static int point = 0;
20     static struct Env *e = NULL;
21     e = curenv;
22     if(count == 0 || e == NULL || e->env_status != ENV_RUNNABLE){
23         if(e != NULL){
24             LIST_REMOVE(e, env_sched_link);
25             LIST_INSERT_TAIL(&env_sched_list[1-point],e , env_sched_link);
26         }
27         while(1){
28             while(LIST_EMPTY(&env_sched_list[point])) point = 1-point;
29             e = LIST_FIRST(&env_sched_list[point]);
30             if(e->env_status == ENV_RUNNABLE){
31                 count = e->env_pri;
32                 break;
33             }else{
34                 if(e->env_status == ENV_NOT_RUNNABLE){
35                     LIST_REMOVE(e, env_sched_link);
36                     LIST_INSERT_TAIL(&env_sched_list[1-point], e, env_sched_link);
37                 }
38             }
39         }
40     }
41     count --;
42     env_run(e);
43 }
```

图 12　sched_yield 函数实现

Thinking 部分：

Thinking 3.1

> **Thinking 3.1**　为什么我们在构造空闲进程链表时必须使用特定的插入的顺序？(顺序或者逆序) ■

解答：

构造空闲进程链表时特定的插入顺序，通常是为了维护链表的有序性，以便更高效地管理和搜索空闲进程块。这里的顺序通常指的是按照一定的规则或者特性来排序。

分配效率：有序链表可以提高分配效率。如果链表按照一定的顺序排列，例如按照地址递增的顺序，那么在分配时可以更快地找到合适大小的空闲进程块。这避免了在整个链表上搜索的开销。

合并空闲块：有序链表有助于合并相邻的空闲块。当一个进程释放内存时，如果它的空闲块与相邻的空闲块有序，那么可以更容易地进行合并操作，减少了合并的复杂性。

快速搜索：如果按照某种规则排序，可以通过一些策略更快速地搜索适合的空闲块。例如，可以使用二分查找等算法。

使用更好的内存分配算法： 有序链表为更高级的内存分配算法提供了更好的支持。某些分配算法可能依赖于链表的有序性，以便更有效地执行某些操作。

Thinking 3.2

Thinking 3.2 思考 env.c/mkenvid 函数和 envid2env 函数:

- 请你谈谈对 mkenvid 函数中生成 id 的运算的理解，为什么这么做？
- 为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？

■

解答：

mkenvid 函数生成一个唯一的环境 ID。函数使用静态变量 nextenvid 作为计数器，每次调用该函数都会生成一个新的环境 ID，并递增计数器。这样做的原因是为了确保每个环境都有一个唯一的标识符。

在函数内部，通过 nextenvid 记录下一个可用的环境 ID，并在生成新的环境 ID 时递增该计数器。静态变量确保了 nextenvid 在函数调用之间保持状态，避免了重复的环境 ID。

至于为什么在 envid2env 函数中需要判断 e->env_id != envid 的情况，原因是在找到对应环境 ID 的情况下，要确保找到的环境确实是活跃的（非空闲状态），避免返回已经被释放的环境。如果没有这步判断，可能会返回一个已经被释放或处于其他状态的环境，导致错误的行为。

Thinking 3.3

Thinking 3.3 结合 include/mmu.h 中的地址空间布局，思考 env_setup_vm 函数:

- 我们在初始化新进程的地址空间时为什么不把整个地址空间的 pgdir 都清零，而是复制内核的`boot_pgdir`作为一部分模板？(提示:mips 虚拟空间布局)
- UTOP 和 ULIM 的含义分别是什么，在 UTOP 到 ULIM 的区域与其他用户区相比有什么最大的区别？

在环境的地址空间初始化时，为什么不将整个地址空间的 pgdir 清零，而是复制内核的 boot_pgdir 作为一部分模板，主要是因为 Mips 虚拟空间布局中存在内核和用户空间的映射关系。在 Mips 中，内核空间和用户空间是共享一部分虚拟地址空间的，具体而言，高地址部分被内核使用，而低地址部分被用户程序使用。

UTOP 和 ULIM 分别是用户空间的最高地址和用户程序可访问的最高地址。UTOP 是用户虚拟地址空间的顶部，表示用户程序可以使用的最高地址。而 ULIM 表示用户程序可访问的最高物理地址，即用户程序可以操作的内存范围。

在初始化环境地址空间时，我们不需要清零整个地址空间，因为内核部分和用户部分共享一部分虚拟地址，直接复制内核的 boot_pgdir 作为模板，可以保留这个共享部分的映射关系。如果整个地址空间都清零，将破坏内核和用户空间的映射关系，可能导致系统无法正常工作。

因此，复制内核的 boot_pgdir 作为模板，然后在此基础上进行必要的修改，是为了保留内核和用户空间的共享映射关系。

- 在 step4 中我们为什么要让`pgdir[PDX(UVPT)]=env_cr3`?(提示: 结合系统自映射机制)
- 谈谈自己对进程中物理地址和虚拟地址的理解

在 env_setup_vm 函数的 Step 4 中，我们执行 pgdir[PDX(UVPT)] = env_cr3 的操作是为了建立虚拟地址空间中的页目录项，将虚拟地址 UVPT 映射到当前环境的页目录表 env_cr3 所在的物理页面。UVPT 是一个宏定义，表示用户虚拟地址空间的页目录表的起始地址。自映射机制要求页目录表的最后一项（索引为 PDX(UVPT)）指向页目录表自身的物理地址，以形成页目录表的自映射。这样，用户进程可以通过 UVPT 来访问自己的页目录表。通过将 pgdir[PDX(UVPT)] 设置为 env_cr3，我们建立了虚拟地址 UVPT 到当前环境的页目录表的映射。这是为了确保用户进程可以通过 UVPT 访问到自己的页目录表，实现自映射的要求。

关于进程中物理地址和虚拟地址的理解：
物理地址：是指实际存储器芯片上的地址，是硬件层面的地址，用于唯一标识存储器中的位置。物理地址是实际的硬件位置。
虚拟地址：是由操作系统提供的抽象地址，由程序使用。虚拟地址在程序执行时被映射到物理地址，操作系统负责管理这个映射关系。通过虚拟地址，程序可以访问内存中的数据和指令。
在操作系统中，虚拟地址空间提供了一种将程序从底层硬件的实际存储器位置中解耦的抽象机制。操作系统通过页表等机制，将虚拟地址映射到物理地址，从而为程序提供了一致的地址空间。这种抽象使得程序的开发更加简便，同时允许操作系统进行更灵活的内存管理。

Thinking 3.4

Thinking 3.4 思考 user_data 这个参数的作用。没有这个参数可不可以？为什么？(如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子) ■

解答：
在函数 load_icode_mapper 中，user_data 参数是一个指向用户提供的数据结构的指针，允许用户在加载二进制数据到内存时传递额外的信息。这种设计的作用主要有以下几点：
通过引入 user_data 参数，函数变得更加通用和灵活。用户可以将任何需要在加载过程中使用的数据结构传递给该函数。这允许用户在加载过程中携带额外的信息，根据自己的需要进行定制和扩展。
用户可以根据自己的需求实现特定的加载过程，通过 user_data 参数传递用户定义的信息。这种设计使得加载函数能够适应各种不同的应用场景。

库中的应用示例： 在一些库或框架中，加载函数可能是通用的，但加载的数据可能包含不同类型的信息，这时通过 user_data 参数传递用户定义的数据结构就显得尤为重要。例如，一个图像加载库可以使用这种机制，允许用户传递包含图像处理相关配置信息的数据结构。
如果没有 user_data 参数，加载函数将无法获取用户自定义的信息，而只能使用函数内部的局部变量或者全局变量。这样的设计会限制函数的通用性和适应性，不够灵活。引入 user_data 参数使得加载函数能够更好地适应各种用户需求，提高了函数的通用性和可扩展性。

Thinking 3.5

Thinking 3.5 结合 load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？ ■

在 load_icode_mapper 函数中，通过 binary 和 size 参数加载二进制镜像。在考虑复制的情况时，我们可能面临下面几种情况：

整页复制： 当 size 大于一页的大小时，需要进行整页复制。这是因为每个页的大小是固定的，可能会导致最后一个页没有被完全填充，所以需要考虑对整页的复制。
不足一页的复制： 如果 size 不足一页的大小，只需要复制 size 字节的数据。这种情况下，我们需要注意页表的映射是否正确，以及是否有可能跨页。
覆盖复制： 在镜像加载的过程中，可能会涉及到覆盖复制。即，镜像的一部分内容可能与已经映射的页面重叠。在这种情况下，我们需要确保不会破坏已有的映射关系。
对齐问题： 可能存在对齐问题，即 va 参数可能不是页对齐的。在复制时，需要考虑对齐问题，确保数据正确复制到目标地址。
最后一页的处理： 如果镜像的大小不是整数页的倍数，最后一页可能包含不完整的数据。需要注意最后一页的处理方式，确保不越界读取数据。
可能的非对齐复制： 如果 va 参数对应的页表项没有对应的物理页面，需要分配一个物理页面进行非对齐复制。
用户态数据的处理： user_data 参数可能包含用户自定义的数据结构，需要确保在加载过程中正确地使用和处理这些数据。

上述情况都考虑到了，可以确保 load_icode_mapper 函数能够正确处理不同复制的场景，确保数据正确加载到目标地址。

Thinking 3.6

Thinking 3.6 思考上面这一段话，并根据自己在 **lab2** 中的理解，回答：

- 我们这里出现的”指令位置”的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢？

在实际的操作系统开发中，通常使用虚拟地址来表示指令的位置，并通过页表映射将这些虚拟地址映射到物理地址。因此，在讨论加载、复制等操作时，"指令位置" 可能是针对虚拟地址空间。

- 你觉得`entry_point`其值对于每个进程是否一样？该如何理解这种统一或不同？

entry_point 是 ELF 文件中指定的程序入口地址。在标准的 ELF 文件格式中，该入口地址对于每个进程是相同的。这是因为 ELF 文件格式规定了程序的入口点，这个入口点是在编译和链接时确定的。

关键在于 ELF 文件的设计目标是为了跨平台的可执行二进制文件。因此，无论在哪个系统上运行，ELF 文件的入口点都是相同的。这使得可执行文件在不同的系统上具有相同的可移植性。

但需要注意的是，尽管程序入口点在 ELF 文件中是相同的，但由于虚拟地址空间的映射不同，该入口点在实际运行时被映射到不同的物理地址。每个进程都有自己的页表，因此虽然入口点相同，但在物理内存中的位置可能是不同的。

Thinking 3.7

Thinking 3.7 思考一下，要保存的进程上下文中的`env_tf.pc`的值应该设置为多少？为什么要这样设置？ ■

env_tf.pc 是进程上下文中的程序计数器，它保存了进程即将执行的下一条指令的地址。在保存进程上下文时，env_tf.pc 应该设置为进程被中断或陷入内核前执行的指令的下一条指令地址。

这是因为当进程被重新调度运行时，需要从上次中断或陷入内核的地方开始执行。保存的上下文信息中，env_tf.pc 所指向的地址就是进程下一次执行的指令地址。

在 xv6 操作系统中，中断处理、系统调用等操作会导致进程上下文的保存。当进程被重新调度执行时，内核会恢复保存的上下文信息，包括 env_tf.pc。这样，程序就能够从上次中断或系统调用的地方继续执行，实现了进程的断点续传。

Thinking 3.8

Thinking 3.8 思考 TIMESTACK 的含义，并找出相关语句与证明来回答以下关于 TIMESTACK 的问题：

- 请给出一个你认为合适的 TIMESTACK 的定义
- 请为你的定义在实验中找出合适的代码段作为证据 (请对代码段进行分析)
- 思考 TIMESTACK 和第 18 行的 KERNEL_SP 的含义有何不同

TIMESTACK 是一个宏定义，表示每个进程的内核栈的顶部，用于存储进程在内核态执行时的上下文信息。在 xv6 中，每个进程有一个内核栈，用于保存进程在内核态执行时的函数调用和临时变量等信息。TIMESTACK 的定义表明它位于内核栈的末尾，与进程的内核栈紧密相关。

KSTACKTOP 是内核栈的起始地址，表示内核栈的顶部。

KSTKSIZE 是每个进程的内核栈大小。

Thinking 3.9

Thinking 3.9 阅读 kclock_asm.S 文件并说出每行汇编代码的作用

.macro setup_c0_status set clr: 定义一个宏 setup_c0_status，用于设置 CP0 寄存器中的 STATUS 寄存器。
.set push: 保存当前汇编环境的设置。

mfc0 t0, CP0_STATUS: 将 CP0 寄存器 STATUS 的值加载到寄存器 t0 中。

or t0, \set|\clr: 将 t0 的值与 \set 所表示的位进行或操作，然后再与 \clr 所表示的位进行或操作。这个步骤用于设置 STATUS 寄存器的一些位，通过宏中传递的参数 \set 和 \clr 来指定设置和清除的位。

xor t0, \clr: 将 t0 的值与 \clr 所表示的位进行异或操作，用于清除 STATUS 寄存器的一些位。

mtc0 t0, CP0_STATUS: 将修改后的 t0 的值写回 CP0 寄存器 STATUS。

.set pop: 恢复之前保存的汇编环境的设置。

.text: 进入代码段。

LEAF(set_timer): 定义一个函数入口标签 set_timer。

li t0, 0xc8: 将立即数 0xc8 装载到寄存器 t0 中。

sb t0, 0xb5000100: 将 t0 的低 8 位写入物理地址 0xb5000100 处，这可能是设置定时器的寄存器地址。

sw sp, KERNEL_SP: 将当前栈指针 sp 的值写入 KERNEL_SP，用于保存当前进程的栈指针。

setup_c0_status STATUS_CU0|0x1001 0: 调用之前定义的宏 setup_c0_status，设置 STATUS 寄存器的 CU0 位和一些其他位，使得计时器开始工作。

jr ra: 从函数中返回，使用寄存器 ra 中保存的返回地址。

nop: 空操作，为了保证指令对齐。
END(set_timer): 定义函数结束标签。

Thinking 3.10

> **Thinking 3.10** 阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。

时钟周期的切换进程通常通过时钟中断来实现。时钟中断是由硬件定时器定期触发的中断，例如，在 MIPS 体系结构中，CP0 寄存器中的计时器会周期性地触发时钟中断。

大概步骤为：

时钟中断触发： 操作系统设置硬件定时器，并在每个时钟周期结束时触发一个中断，通常是通过硬件时钟计数器（如 CP0 寄存器）实现的。

中断处理程序： 当时钟中断发生时，处理器会跳转到预先设置的时钟中断处理程序。在这个处理程序中，操作系统执行与时钟相关的操作。

更新进程状态： 操作系统会根据调度算法检查当前运行的进程，决定是否进行进程切换。如果需要切换，操作系统将保存当前进程的上下文信息，例如寄存器状态，然后选择一个新的就绪进程。

切换进程： 操作系统将新选定的进程的上下文信息恢复到处理器中，这包括更新页表、加载新的寄存器状态等。这样，处理器现在开始执行新的进程。

继续执行： 处理器继续执行新选定的进程，直到下一个时钟中断发生，整个过程会在下一个时钟中断时重复。

本次实验耗时 9 小时