OS-lab3

thinking

3.1

> Thinking 3.1 请结合 MOS 中的页目录自映射应用解释代码中 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V` 的含义。 ■

本段代码完成的某个进程 e 所对应的页目录的自映射，选定的自映射位置是 PDX(UVPT)，其中 UVPT 是**用户页表起始处的内核虚拟地址**，也就是我们让 pgdir 的第 UVPT 个表项作为自映射表项，相应地在这个表项中存入页目录自身转化为物理地址的结果，并且加上 PTE_V 的权限位

3.2

> Thinking 3.2 `elf_load_seg` 以函数指针的形式，接受外部自定义的回调函数 `map_page`。请你找到与之相关的 `data` 这一参数在此处的来源，并思考它的作用。没有这个参数可不可以？为什么？ ■

data 是传入的进程控制块指针，在 load_icode_mapper 函数中被调用，虽然我们以 void* 类型传入，但是立刻会强转回 Env* 类型，这个参数显然是必须要有的，因为我们要做的就是要为进程分配物理页，把有关信息加载到 data 所对应的进程中，其中还要在这个进程有关的 pgdir 下完成地址映射，那么记录了进程信息的 data 显然是必须要有的

3.3

> Thinking 3.3 结合 `elf_load_seg` 的参数和实现，考虑该函数需要处理哪些页面加载的情况。 ■

首先，函数判断虚拟地址 va 是否页对齐，如果不对齐，需要将多余的地址记为 offset，并且 offset 所在的剩下的 PAGE_SIZE 的 binary 数据写入对应页的对应地址；然后，依次将段内的页映射到物理空间；最后，若发现其在内存的大小大于在文件中的大小，则需要把多余的空间用 0 填充满

3.4

> Thinking 3.4 思考上面这一段话，并根据自己在 **Lab2** 中的理解，回答：
>
> - 你认为这里的 `env_tf.cp0_epc` 存储的是物理地址还是虚拟地址?
>
> ■

存储的是虚拟地址

因为 epc 存储的是发生错误时 CPU 当前所执行到的那一条指令的地址，而对 CPU 而言其所操作的地址都是虚拟地址（在程序中引用访存的都应该是虚拟地址），因此 env_tf.cp0_epc 存储的是虚拟地址

3.5

> Thinking 3.5 试找出 0、1、2、3 号异常处理函数的具体实现位置。8 号异常（系统调用）涉及的 `do_syscall()` 函数将在 Lab4 中实现。 ■

handle_int 在 genex.S 文件中

handle_mod、handle_tlb、handle_sys 则是通过 genex.S 文件中的宏函数 BUILD_HANDLER 实现的

3.6

> Thinking 3.6 阅读 entry.S、genex.S 和 env_asm.S 这几个文件，并尝试说出时钟中断在哪些时候开启，在哪些时候关闭。 ■

开启时钟中断：eret 返回时（根据恢复的 STATUS 寄存器）、用户态执行或内核主动开启
关闭时钟中断：异常/中断入口、中断处理期间、上下文恢复前

3.7

> Thinking 3.7 阅读相关代码，思考操作系统是怎么根据时钟中断切换进程的。 ■

首先，模拟器通过 kclock_init 函数完成时钟中断的初始化，设置时钟中断发生的频率；接着，调用 enable_irq 函数开启中断；在进程运行过程中，若时钟中断产生，则会触发 MIPS 中断，系统将 PC 指向 0x800000080，即跳转到 .text.exc_gen_entry 代码段，进行异常分发：由于是中断，判断为 0 号异常，则跳转到中断处理函数 handle_init；进而判断属于中断中的 IM4(时钟中断)，进而跳转到 timer_irq 函数处理；timer_irq 函数调用 schedule 函数开始进行进程调度；在 schedule 函数中，将当前运行的进程控制块 curenv 取出来；对当前进程的可用时间片数量减一；当满足以下四种条件之一：

1. 尚未调度过任何进程（curenv == NULL）
2. 当前进程已经用完了时间片（count == 0）
3. 当前进程不再就绪，如：被阻塞或退出（e->env_status != ENV_RUNNABLE）
4. yield 参数指定必须发生切换（yield != 0）

则进行进程切换：判断若 curenv 的状态为 ENV_RUNNABLE，则把 curenv再次插入调度队列的尾部，并且接着从调度队列头取出一个进程；将剩余时间片数量设置为新进程的优先级；调用 env_run 函数切换选中进程，并且进行运行

地址布局

```
o     4G ----------->  +----------------------------+------------0x100000000
 o                     |       ...                  |  kseg2
 o      KSEG2    -----> +----------------------------+------------0xc000
0000
 o                     |          Devices           |  kseg1
 o      KSEG1    -----> +----------------------------+------------0xa000
0000
 o                     |      Invalid Memory        |   /|\
 o                     +----------------------------+----|-------Physical
Memory Max
 o                     |       ...                  |  kseg0
 o      KSTACKTOP-----> +----------------------------+----|-------0x8040
0000--------end
 o                     |       Kernel Stack         |    | KSTKSIZE
   /|\
 o                     +----------------------------+----|------
     |
 o                     |       Kernel Text          |    |
   PDMAP
 o      KERNBASE -----> +----------------------------+----|-------0x8002
0000    |
 o                     |      Exception Entry       |   \|/
   \|/
```

```
 o      ULIM      -----> +----------------------------+------------0x8000
0000-------
 o                       |         User VPT           |     PDMAP
   /|\
 o      UVPT      -----> +----------------------------+------------0x7fc0
0000    |
 o                       |           pages            |     PDMAP
    |
 o      UPAGES    -----> +----------------------------+------------0x7f80
0000    |
 o                       |           envs             |     PDMAP
    |
 o  UTOP,UENVS    -----> +----------------------------+------------0x7f40
0000    |
 o  UXSTACKTOP -/        |     user exception stack   |     PTMAP
    |
 o                       +----------------------------+------------0x7f3f
f000    |
 o                       |                            |     PTMAP
    |
 o      USTACKTOP ----> +----------------------------+------------0x7f3f
e000    |
 o                       |     normal user stack      |     PTMAP
    |
 o                       +----------------------------+------------0x7f3f
d000    |
 a                       |                            |
    |
 a                       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    |
 a                       .                            .
    |
 a                       .                            .
  kuseg
 a                       .                            .
    |
 a                       |~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
    |
 a                       |                            |
    |
 o       UTEXT     -----> +----------------------------+------------0x0040
0000    |
 o                       |      reserved for COW      |     PTMAP
    |
 o       UCOW      -----> +----------------------------+------------0x003f
f000    |
 o                       |   reversed for temporary   |     PTMAP
    |
 o       UTEMP     -----> +----------------------------+------------0x003f
e000    |
 o                       |       invalid memory       |
   \|/
 a     0 ------------>  +----------------------------+ ----------------------
-------
 o
```

kseg2：内核段2

1. 一般用于映射 IO 设备，高级内核模块等
2. 不是直接映射物理内存，需要通过页表转换
3. 所有地址必须通过页表才有效

kseg1：内核段1

1. 一般用于访问 IO 设备和内存控制器
2. 不通过页表转换，直接把地址映射到物理内存
3. 非缓存区
4. 一般在 MIPS 中用作范根内存映射寄存器

kseg0：内核段0

1. 映射物理内存的前 512 MB
2. 不需要页表
3. 可缓存
4. 内核代码、数据、栈空间存放的位置

KSTACKTOP：内核栈顶

1. 内核线程的栈空间地址的起始位置
2. 栈向下生长

KERNBASE：内核代码和数据段开始地址

1. 内核起始位置（内核代码，全局数据等）
2. 一般与物理地址 0020000 映射

ULIM：用户访问限制地址

1. 用户进程不能访问此地址以上的内存
2. 内核通过这个边界保护自身空间不被访问

UVPT：用户可访问的页表虚拟地址

1. 用户可读但不可写
2. 用户进程用来查看自己的页表项
3. 映射整个两级页表，包括页目录自映射

UPAGE：物理页面数组信息（页结构体）

1. 系统记录物理页使用情况的结构体
2. 通常为一个 struct Page 数组

UTOP，UENVS：用户虚拟地址空间顶部，环境结构体区域

1. 包含所有用户进程 env 信息
2. 系统使用的 struct Env 数组存放于此

USTACKTOP：用户栈栈顶

1. 一般用户程序使用的正常栈
2. 栈向下生长

注意：其实这里我们把物理地址中的 pages 和 envs 映射了两份，一份是通过内核程序的页表映射到 KERNBASE 之上的地址，内核程序可以通过这些地址读写 pages 和 envs 空间；一份是通过进程页表映射到 UPAGE 和 UENVS，用户进程只能通过页表只读地访问这部分区域

进程(环境)控制块 struct Env，这是一个典型的进程描述结构体，用来记录某个环境(也可以理解成是用户态程序/进程)的执行上下文，内存状态，调度信息，IPC状态等

```c
// Control block of an environment (process).
struct Env {
        // CPU 上下文
        struct Trapframe env_tf;          // 保存寄存器上下文(进程调度或陷入内核时存储上下文)

        // 链表结构
        LIST_ENTRY(Env) env_link;         // 加入空闲链表的指针
        TAILQ_ENTRY(Env) env_sched_link; // 加入调度队列的指针

        // 身份和状态标识
        u_int env_id;                     // 全局唯一ID(高位为计数器，低位为索引)
        u_int env_asid;                   // 地址空间标识符(用于TLB)
        u_int env_parent_id;              // 创建该进程的父环境ID
        u_int env_status;                 // 当前环境的状态(可以看成是若干标志位集合)
        Pde *env_pgdir;                   // 当前进程对应的[页目录]的[内核][虚拟]地址

        // IPC(进程间通信)
        u_int env_ipc_value;   // 接收到的值
        u_int env_ipc_from;    // 发送方的 envid
        u_int env_ipc_recving; // 当前是否在等待接收
        u_int env_ipc_dstva;   // 接收共享页的目标虚地址
        u_int env_ipc_perm;    // 映射该页时使用的权限

        // TLB 异常处理
        u_int env_user_tlb_mod_entry; // 用户态自定义的 TLB 修改异常处理函数入口

        // 调度相关
        u_int env_runs; // 被调度运行的次数
        u_int env_pri;  // 调度优先级
};
```

其中 env_status 只有三种取值：

1. ENV_FREE：表示该进程控制块处于空闲状态，没有被任何进程所使用，即该进程控制块处在进程空闲链表中
2. ENV_NOT_RUNNABLE：表示该进程处于阻塞状态，下一步会等待相应条件的到来变为就绪状态，其中阻塞
3. ENV_RUNNABLE：表示该进程处于执行状态或就绪状态，既可能正在执行，有可能在等待被调度，对于这种状态的进程，应该把他放入调度队列 env_sche_list 中

保存进程 CPU 上下文的结构：struct Trapframe

```c
struct Trapframe {
        /* 32 个通用寄存器 */
        unsigned long regs[32];

        /* 特殊寄存器 */
        unsigned long cp0_status;     // CPU 的全局状态
        unsigned long hi;             // 高位寄存器
        unsigned long lo;             // 低位寄存器
        unsigned long cp0_badvaddr;   // 错误地址寄存器
        unsigned long cp0_cause;      // 中断异常原因记录寄存器
        unsigned long cp0_epc;        // 异常发生时的 PC
};
```

有关宏、变量、系统实现好的函数：

```
---------系统宏、函数、变量命名习惯---------

envs ： 存放进程控制块的物理内存，在系统启动时就完成了分配；

env_sched_list ： 调度队列(deque 底层实现)，管理已经被分配的进程控制块和对进程的调度，进程创建时为该进程分配进程控制块，并加入 env_sched_list，进程释放时把该进程对应的进程控制块从 env_sched_list 中移除，同时只有标志位为 ENV_RUNNABLE 的进程可以存放在该队列中；

env_free_list ： 空闲进程控制块队列(list 底层实现)，把所有空闲的 Env 控制块以链表形式串联；

base_pgdir ： 操作系统模版页目录的基地址，内核会预先设定好一个页目录，它通常映射了操作系统内核的空间，它通常作为模版，用于在创建新进程和新页表的时候复制使用，比如我们创建一个新进程对应的页表 new_page 的时候，我们可能会 memcpy(new_pgdir, base_pgdir, sizeof(base_pgdir));这样新进程就自动拥有了和内核一样的页表结构，不需要重新构建这些页表；

UTOP ： 用户虚拟地址空间的上限，即用户程序可用的最高虚拟地址，内核在此地址以上分配和映射内核自身空间，以下则是给用户程序使用的(va)；

UVPT ： 一个特殊的虚拟地址，用来只读映射当前进程自己的页表，允许用户程序查看自己的页表结构，用户通过访问 UVPT + i * PGSIZE 的地址，即可访问第 i 个页表项(va)；

const Elf32_Ehdr *elf_from(const void *binary, size_t size) ： 从原始的二进制数据中解析并返回 ELF 文件头(Elf32_Ehdr)指针，入参含义为 binary 指向 ELF 二进制文件的内存地址，size 文件大小；

int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data) ： 根据 ELF 程序头 ph 的描述，从 bin 中读取该段数据，并且通过 map_page 回调函数将其映射到内存中，具体来讲就是把一个可执行段(比如 .text / .data)加载到用户虚拟内存中，入参含义：ph 为指向 ELF 的程序头，描述该段的虚拟地址、大小、权限等，map_page 为一个函数指针，用于把指定数据映射到对应的虚拟地址，bin 为原始的 ELF 二进制数据，data 为传给 map_page 的上下文参数，通常是某个进程；

---------待填写的代码---------

static int asid_alloc(u_int *asid) ： 分配一个新的进程地址标识符，存入 asid 的引用中；

static void asid_free(u_int i) ： 释放 i 对应的进程地址标识符；

static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size, u_int perm) ： 在页目录 pgdir 和进程地址标识符 asid 环境下，把虚拟地址 [va, va + size) 空间映射到物理地址 [pa, pa + size) 空间;※

u_int mkenvid(struct Env *e) ： 根据当前进程 e 的地址创建一个 envid 编号；

int envid2env(u_int envid, struct Env **penv, int checkperm) ： 根据 envid 取出一个 Env 进程结构体，如果 envid 为 0，就取当前运行中的 curenv 进程，反之就从对应 envs 数组位置中取；

void env_init(void) ： 初始化所有 envs 数组，标记为空闲，并且插入到内存空闲链表中，完成 base_pgdir 下UPAGES 和 UENVS 段的映射(会调用 map_segment 函数);※
```

```
static int env_setup_vm(struct Env *e) : 初始化进程 e 所对应的地址空间，主要分配一个物理页，将其转化为内核虚拟地址后作为进程页目录基地址，把模版页目录以 UTOP 开始，UTOP-UVPT 这段长度的内容拷贝到当前进程的页目录下，并且设置页目录在 UVPT 处的自映射；※

int env_alloc(struct Env **new, u_int parent_id) : 分配并初始化一个 Env 结构体，并且给出其父进程的 id;※

static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src, size_t len) :  data 本质是一个 Env，src 本质是一个 ELF 数据源，函数作用是把 src 中的数据加载到 data 这个进程中，具体操作就是为进程分配一个物理页，在这个物理页对应内核虚拟地址偏移量为 offset 的位置处，把 src 数据源中长度为 len 的数据加载进去，并且在当前进程的页目录和 asid 标识符下，建立物理页和虚拟地址 pa 的映射关系;※

static void load_icode(struct Env *e, const void *binary, size_t size) : 把 binary 所指向的 ELF 文件内容加载到进程 e 的执行环境中;

struct Env *env_create(const void *binary, size_t size, int priority) : 创建一个新进程，并且把 binary 所指向的 ELF 文件和 priority 优先级加载进去(调用 load_icode);※

void env_free(struct Env *e) : 把 e 进程对应的环境释放，同时更新有关页目录，页表，TLB 的相关内存信息;

void env_destroy(struct Env *e) : 销毁 e 进程，释放 e 进程对应的环境(调用 env_free)，如果释放的是正在运行的进程，需要重新调度一个新的进程继续运行;

void env_run(struct Env *e) : 把当前进程 curenv 切换成 e 进程，分别保存和恢复他们各自的运行环境; ※

void env_pop_tf(struct Trapframe *tf, u_int asid) : 从 tf 中恢复进程执行需要的上下文，CPU 从内核态切换回用户态，继续用户进程的执行;

---------链表、队列宏操作---------

TAILQ_INIT(&env_sched_list) : 初始化调度队列;

TAILQ_REMOVE(&env_sched_list, e, env_sched_link) : 删除队头元素;

TAILQ_INSERT_HEAD(&env_sched_list, e, env_sched_link) : 向队头插入 e;

TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link) : 向队尾插入 e;

TAILQ_FIRST(&env_sched_list) : 获取队头元素;

TAILQ_LAST(&env_sched_list) : 获取队尾元素;

TAILQ_NEXT(e1) : 获取 e1 的后一个元素;

TAILQ_PRE(e1) : 获取 e1 的前一个元素;

TAILQ_EMPTY(&e_sched_list) : 判断队列是否为空;

TAILQ_INSERT_AFTER(&e1, e, env_sched_link) : 在 e1 之后插入 e;

TAILQ_INSERT_BEFORE(&e1, e, env_sched_link) : 在 e1 之前插入 e;

TAILQ_FOREACH(e, &env_sched_list, env_sched_link) : 遍历队列，每次取到的元素存在 e 中;
```

三类地址的区别：物理地址，内核虚拟地址，用户进程虚拟地址

```
物理地址(pa)：内存中实际的地址，无法直接被访问，只能通过虚拟地址+映射的机制被访问到；
内核虚拟地址(kva)：操作系统的内核代码/内核操作想要访问物理地址的时候，必须通过内核虚拟地址来间接访问，内核虚拟地址和物理地址之间的映射关系通常是确定的(因为只有一段内核代码)；
用户进程虚拟地址(va)：用户进程的代码所看到的地址空间，通过页表机制和物理地址建立映射关系，由于页表的特性，通常只能由用户虚拟地址快速访问到物理地址，但是由某个物理地址反向索引虚拟地址则需要遍历页表
```

我们所写的代码都是操作系统内核的代码，并不涉及到用户层面具体执行的代码，因此我们操作的虚拟地址都是内核虚拟地址，如果我们想访问某个物理地址，都是先把他转化成**内核虚拟地址**再进行访问；而对于用户进程的虚拟地址而言，我们会做的只是给定一个 va，建立它和某个物理地址 pa 的页表映射关系，不会在内核代码本体中通过它来访问

模版页目录

进程地址空间的分析：每个进程的虚拟空间总体上都可以分为**内核空间**和**用户空间**这两部分，而我们的操作系统在初始化的时候，内核代码（CPU 执行的代码）操作的也是虚拟地址，因此就需要一张页表来实现内核虚拟地址到物理地址的映射，这就是 base_pgdir 的由来，同时它会完成 pages 和 envs 从物理地址到虚拟地址的映射，但是这部分映射不仅内核代码是这么做的，其他进程的映射也是这样的，因此我们就考虑以后如果再有新的进程，其中 envs 和 pages 到 UENVS 和 UPAGES 的映射就不需要重新建立，只要拷贝 base_pgdir 即可，因此我们就有了下面的设计

```
map_segment(base_pgdir, 0, PADDR(pages), UPAGES, ROUND(npage * sizeof(struct Page), PAGE_SIZE), PTE_G);
map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct Env), PAGE_SIZE),PTE_G);
```

具体来说，我们预处理的映射关系就是以下两组

1. 物理内存中的物理页面集合 pages 和虚拟地址 UPAGE 之间的映射
2. 物理内存中的进程集合 envs 和虚拟地址 UENVS 之间的映射

我们把这两组映射关系存在 base_pgdir 中，以后再给新的进程 env 分配地址空间的时候，就可以直接拷贝

```
memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP), sizeof(Pde) * (PDX(UVPT) - PDX(UTOP)));
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
```

对于进程虚拟空间而言，UTOP 之下的空间因进程而异，我们不需要初始化；从 UTOP 到 UVPT 之间的空间则是我们之前映射好的模版，直接拷贝即可；而 UVPT 乃至 ULIM 再往上的空间用户进程无权访问，我们也不需要考虑

第二条代码的含义是，进程页目录中 PDX(UVPT) 位置完成了页目录的自映射，也就是 env_pgdir 的第 PDX(UVPT) 中存的是 env_pgdir 本身所转化成的物理地址

内核暴露机制

每个进程的虚拟地址空间都会进行如下的划分：

```
---------------
|    kseg3    |  -> 内核态，需要 TLB 转换
---------------
|    kseg2    |  -> 特权访问
---------------
|    kseg1    |  -> 内核管理，不需要 TLB 转换，线性映射(所有进程共享)
---------------
|    kseg0    |  -> 内核管理，不需要 TLB 转换，线性映射(所有进程共享)
---------------  -> ULIM
|    kuseg    |  -> 用户空间，访问需要 TLB 转换
---------------
```

而对 kuseg 又有如下的划分：

```
---------------  -> ULIM(系统分配给用户虚拟空间的最高地址)
|   进程页表    |  -> 存储进程自己页表的空间
---------------  -> UVPT
|  共享只读区   |  -> 所有进程只读共享的内核区域(这段区域直接拷贝了内核页表 base_pgdir)
---------------  -> UTOP
|   用户空间    |  -> 每个进程各自加载的信息
---------------
```

其中 kuseg 称为用户虚拟地址空间，kseg0 kseg1 kseg2 kseg3 统称为内核虚拟地址空间，其中我们先前映射的两个特殊地址，也就是 UPAGES 和 UENVS 都处在 UTOP 和 UVPT 之间，也就是说我们每次拷贝 UTOP 和 UVPT 之间的这段区域，对应地就得到了所有进程都共享的 pages 和 envs 的映射

ELF 的加载

加载一个 ELF 文件到内存 <=> 把 ELF 文件中所有需要加载的**程序段**都加载到对应的虚拟地址上

从整体上来看，我们通过 load_icode 这个函数实现 ELF 文件的加载

```
// 函数本体
static void load_icode(struct Env *e, const void *binary, size_t size);
// 具体实现
load_icode(e, binary, size);
```

通过执行这段代码，我们就可以把 binary 所指向 ELF 文件的二进制镜像加载到 e 这个进程上运行，具体来讲该函数内部又分为以下几步来执行

① 首先我们解析 ELF 文件的文件头：

```
// 函数本体
const Elf32_Ehdr *elf_from(const void *binary, size_t size);
// 具体应用
const Elf32_Ehdr *ehdr = elf_from(binary, size);
```

通过执行这段代码，我们就可以把 binary 所对应的 ELF 文件文件头解析到 ehdr 指针中

② 在解析出 ELF 文件头以后，我们遍历每个程序段 segment 的段头 ph，以及其在内存中的起始位置 bin，调用 elf_load_seg 函数把指定程序段加载到进程的地址空间中

```
// 函数本体
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data);
// 具体应用
elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e)
```

这里 elf_load_seg 中又调用了 load_icode_mapper 作为回调函数，这个函数的作用就是为给定的进程 data 分配一个物理页，加载 src 对应的数据源到该物理页对应的**内核虚拟地址**偏移 offset 的位置，并且在这个进程的 pgdir 前提下，建立这个物理页和用户虚拟地址 va 之间的映射

```
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src, size_t len);
if ((r = map_page(data, va + i, 0, perm, bin + i, MIN(bin_size - i, PAGE_SIZE))) != 0) {
      return r;
} //这里的 map_page 就是回调函数 load_icode_mapper
```

通过执行这段代码，我们就可以通过每一个程序段 ph 头的指引，把二进制文件 binary 中的指定信息，借助回调函数加载到 e 这个进程中

③ 最后由 env_tf.cp0_epc 字段指示进程恢复运行时 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，当我们运行进程时，CPU 将自动 从 PC 所指的位置开始执行二进制码

可以认为函数关系如下，其中 load_icode 完成整个 ELF 文件的加载，elf_seg_map 完成一个程序段的加载，load_icode_mapper 则完成一个程序段中单个页面的加载，三者自底向上依次实现

进程执行有关

进程创建，通过 env_create 函数来实现

```
// 函数本体
struct Env *env_create(const void *binary, size_t size, int priority)
```

通过给定的二进制镜像，我们可以返回一个 PCB，用于表示创建好的进程

进程调度，通过 schedule 函数来实现

```
// 函数本体
void schedule(int yield);
```

用来模拟运行中进程时间片递减的过程，每次都把当前进程的时间片递减，如果进程需要切换，就给出一个新的待运行进程

进程切换，通过 env_run 函数来实现

```
// 函数本体
void env_run(struct Env *e);
```

运行中进程 curenv 切换成传入的进程 e，并且相应保存和恢复对应的进程上下文，这里存保存进程运行上下文是通过内核栈 KSTACK 来实现的，恢复进程运行上下文是通过 enf_pop_tf 来实现的

进程创建流程

```
kernel init : 初始化内核
|  env_init() : 初始化所有 envs 数组，完成模版页目录中 pages 和 envs 的映射
|  |   map_segment() : 实现某段物理地址向虚拟地址的映射
|  env_create() : 创建进程 e
|  |   env_alloc() : 为进程 e 分配一个 Env 结构体，并且设置很多相关标志位
|  |   |   env_setup_vm() : 为进程 e 分配一段页目录空间，并且完成 pages 和 envs 的映射拷贝与自映射
|  |   |     mkenvid() : 为进程 e 分配一个进程 id
|  |   load_icode() : 完成 ELF 文件的加载到进程 e 和虚拟地址的映射
|  |   |   elf_load_seg() : 完成一个程序段 segment 的加载和映射
|  |   |     load_icode_mapper() : 完成一个程序段 segment 中的一个页的加载和映射
|  TAILQ_INSERT_HEAD : 把创建好的进程压入调度队列
```

中断异常

我们只考虑**异步异常**这一种情况，CPU 处理异常需要完成以下四步工作：

1. 设置 EPC 指向从异常返回的地址
2. 设置 SR 位，强制 CPU 进入内核态，并且关中断
3. 设置 Cache 寄存器，用于记录异常发生的原因
4. CPU 开始从异常入口位置取指，之后交由软件处理

异常与对应编号的关系：

```
0 号异常 的处理函数为 handle_int：表示中断，由时钟中断、控制台中断等中断造成

1 号异常 的处理函数为 handle_mod：表示存储异常，进行存储操作时该页被标记为只读

2 号异常 的处理函数为 handle_tlb：表示 TLB load 异常

3 号异常 的处理函数为 handle_tlb：表示 TLB store 异常

8 号异常 的处理函数为 handle_sys：表示系统调用，用户进程通过执行 syscall 指令陷入内核
```

entry.S

中断分发函数

```
#include <asm/asm.h>
#include <stackframe.h>
/*
   当 EXL 位被设置或者 UM 位没有被设置的时候，处理器处于内核模式
   当 EXL 位被设置的时候，发生新异常的时候 EPC 的值不会被更新
   为了让处理器保持在内核模式且支持异常可重入，我们清楚 UM 和 EXL 位，并且清除 IE 位使得全局禁用中断
*/

// TLB 异常入口
.section .text.tlb_miss_entry
tlb_miss_entry:
        j       exc_gen_entry
// 通用异常入口
.section .text.exc_gen_entry
exc_gen_entry:
```

```
        SAVE_ALL // 宏，用于将当前所有通用寄存器的值保存到内核栈中,确保异常处理后能恢复现场

        // 修改 CP0 状态寄存器
        /*
          先把 CP0 的状态寄存器 STATUS 读到 t0 通用寄存器，接着清除以下位：
            1. STATUS_UM：强制进入内核模式（清除用户模式位）
            2. STATUS_EXL：清除异常级别位，允许新异常触发时更新 EPC
            3. STATUS_IE：全局禁用中断（防止中断嵌套）
          再把修改后的 t0 写回 STATUS 寄存器
        */
        mfc0    t0, CP0_STATUS
        and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
        mtc0    t0, CP0_STATUS

        // 异常处理分发：根据异常原因跳转到对应的处理函数
        /*
          先读取 CAUSE 寄存器的值到 t0，该寄存器记录了异常原因，接着仅保留 t0 的 2-6 位，也就是 ExcCode 字段，得到异常类型编号
          接着以 t0 位偏移，从 exception_handlers 数组中加载对应的异常处理函数地址到 t0
          最后跳转到对应的异常处理函数
        */

/* Exercise 3.9: Your code here. */
        mfc0 t0, CP0_CAUSE
        andi t0, 0x7c
        lw t0, exception_handlers(t0)
        jr t0
```

exercise 解析

```
#include <asm/cp0regdef.h>
#include <elf.h>
#include <env.h>
#include <mmu.h>
#include <pmap.h>
#include <printk.h>
#include <sched.h>

// 定义所有的进程数组，共有 NENV 个，采用 PAGE_SIZE 对齐，便于页级映射
struct Env envs[NENV] __attribute__((aligned(PAGE_SIZE)));

// 当前这在运行的进程指针，初始设为 NULL
struct Env *curenv = NULL;

// 空闲环进程链表(LIST_HEAD 定义)，存放所有状态为 ENV_FREE 的 Env 结构体
static struct Env_list env_free_list;

// 可运行(RUNNABLE)进程的调度链表(双向队列)，只有 env->env_status == ENV_RUNNABLE 的进程才在这个列表中
struct Env_sched_list env_sched_list;

// 模版页目录
static Pde *base_pgdir;

```

```c
// 位图，表示某个 ASID 是否被分配过，每个 bit 表示一个 ASID 是否被占用，其中 NASID 表示系统所支持的最大 ASID 的数量
static uint32_t asid_bitmap[NASID / 32] = {0};

/* Overview:
  函数功能：分配一个未使用的 ASID
  后置条件：如果成功分配，就返回 0，并且将分配的 ASID 存入 *asid 中；如果没有可用的 ASID，就返回 -E_NO_FREE_ENV
  入参含义：传入一个无符号指针，用于记录分配到的 ASID(类比 C++ 中的引用传值)
 */
static int asid_alloc(u_int *asid) {
// 遍历所有的可能的 ASID，i 是 ASID 的编号
        for (u_int i = 0; i < NASID; ++i) {
// 分别得到第 i 个 AISD 在位图的第几个 int 中，又位于这个 int 整型的第几位中
                int index = i >> 5;// i / 32
                int inner = i & 31;// i % 32
// 检查当前 ASID 是否被使用，其中 index 是第 i 个 ASID 所处的那个 32 位的整数，i << inner 得到一个掩码，也就是看 bitset 中的第 index 个整数的低 inner 位是否为 0，如果为 0，则表示这一位对应的 ASID 没有使用；为 1 则表示被使用
                if ((asid_bitmap[index] & (1 << inner)) == 0) {
// 调用完之后把这一位标记为使用，同时记录我们找到的是第 i 个 ASID，写回传入的 *asid 指针中，成功找到 ASID 就返回 0
                        asid_bitmap[index] |= 1 << inner;
                        *asid = i;
                        return 0;
                }
        }
// 遍历结束都找不到合法的 ASID，就返回错误码
        return -E_NO_FREE_ENV;
}

/* Overview:
  函数功能：释放一个 ASID
  前置条件：这个 ASID必须是之前通过 asid_alloc 分配得到的
  后置条件：该 ASID 被成功释放，且在之后可以被重新分配使用
  入参含义：当前要释放的 ASID 的编号
 */
static void asid_free(u_int i) {
// 具体操作就是找到第 i 个 ASID 对应位图中第几个整型 int 的第几位，然后把这个整型的对应位从 1 变成 0 即可
        int index = i >> 5;
        int inner = i & 31;
        asid_bitmap[index] &= ~(1 << inner);
}

/* Overview:
  函数功能：把虚拟地址空间 [va, va + size) 映射到物理地址区间 [pa, pa + size)，并且写入页目录，对每个页表项使用权限位 perm | PTE_V
  前置条件：物理地址 pa，虚拟地址 va，以及偏移量 size 的大小必须都是页大小 PAGE_SZIE 对齐的
  入参含义：指向页目录的指针，地址空间标识符，页对齐的起始物理地址，页对齐的起始虚拟地址，映射到的地址空间大小(以字节为单位，必须是页大小的整数倍)，页表项的权限标志位(不包含有效位)
 */
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size, u_int perm) {
// 首先给出断言，pa，va，size 都是页大小对齐
        assert(pa % PAGE_SIZE == 0);
        assert(va % PAGE_SIZE == 0);
```

```
72          assert(size % PAGE_SIZE == 0);
73
74  // 按照页大小一次建立每个页的映射
75          for (int i = 0; i < size; i += PAGE_SIZE) {
76  // 核心代码：把虚拟地址 va + i 映射到物理地址 pa + i，核心功能就是 page_insert 函数的实现，该函数用来实现 va + i 这个虚拟地址到 pa + i 这个物理地址对应的物理页 page 的映射，因为这里使用了页表映射，而不是 KADDR 函数，因此是用户虚拟地址而非内核虚拟地址
77                  /* Exercise 3.2: Your code here. */
78                  panic_on(page_insert(pgdir, asid, pa2page(pa + i), va + i, perm));
79          }
80  }
81
82  /* Overview:
83    函数功能：为每个环境 env 创建一个唯一的 ID(envid)
84    前置条件：参数 e 必须是有效的环境指针
85    后置条件：成功的时候返回该环境对应的 envid
86    入参含义：给出一个表示环境(进程)的结构体指针
87   */
88  u_int mkenvid(struct Env *e) {
89  // 定义一个静态局部变量 i，它只会全局初始化一次，作为自增计数器使用，最终得到的编号分为两部分，高位是 i << (1 + LOG2NENV)，它表示一个唯一的编号，本质就是左移 11 位，低位是 e - envs，其中 envs 是 e 所在的结构体数组的起始地址，二者相减得到 e 的数组下标，也就是它所在位置的编号，这个编号用来填充 envid 的低位
90          static u_int i = 0;
91          return ((++i) << (1 + LOG2NENV)) | (e - envs);
92  // 这样的设计可以称为：高位表示唯一性计数器，低位表示环境索引，能够实现唯一性保证(某个环境被销毁后再次创建，i自增可以保证envid不同)与索引能力(可以通过低位保留的数组索引，快速通过envid找回环境结构体)
93  }
94
95  /* Overview:
96    函数功能：根据给定的 envid 找到对应的进程 e，如果 envid 是 0，返回当前正在执行的进程；另外，如果 checkperm 非零，则要求指定的 env 必须是当前环境(curenv)或其直接子环境
97   */
98  int envid2env(u_int envid, struct Env **penv, int checkperm) {
99          struct Env *e;
100
101 // 如果传入的 envid == 0，则表示调用者想获取当前环境，直接设置 *penv = curenv，并且返回成功，0 这个特殊 ID 表示我自己，可以减少一次查找 curenv->envid 的操作
102         /* Exercise 4.3: Your code here. (1/2) */
103         if (envid == 0) {
104                 *penv = curenv;
105                 return 0;
106         }
107 // 根据 envid 查找进程结构体指针，用 ENVX(envid) 提取 envid 的低位索引部分，从 envs 数组中取出对应的进程结构体
108         e = &envs[ENVX(envid)];
109
110 // 检查环境是否合法，即未被释放且 ID 匹配，即使数组索引对了(ENVX)，也要验证 env_id 是否匹配，避免访问错误/过期的环境结构，这是经典的 ID 冗余验证机制
111         if (e->env_status == ENV_FREE || e->env_id != envid) {
112                 return -E_BAD_ENV;
113         }
114
```

```
// 根据 checkperm 判断是否有权限操作这个环境，如果设置了 checkperm 标志，则说明需要检查调用者是否有权限操作目标环境，如果目标环境既不是自己(e != curenv)，也不是自己的直接子环境(e->env_parent_id != curenv->env_id)，则说明权限不足，这是因为为了安全起见，防止一个环境非法修改/终止其他无关进程，只允许操作自己或自己 fork 出来的子环境
        /* Exercise 4.3: Your code here. (2/2) */
        if (checkperm && e != curenv && e->env_parent_id != curenv->env_id) {
                return -E_BAD_ENV;
        }

// 把找到的合法环境返回给调用者
        *penv = e;
        return 0;
}

/* Overview:
  函数功能：将所有 envs 数组中的进程标记为空闲，并且插入到 env_free_list 链表中，插入顺序为逆序插入，这样第一次调用 env_alloc 的时候会返回 envs[0]，同时调用 map_segment 函数，把内核中的 Page 数据结构的物理地址映射到用户地址的 UPAGES 的虚拟地址，把内核中的 Env 数据结构的物理地址映射到用户地址的 UENVS 的虚拟地址
  入参含义：无
 */
void env_init(void) {
        int i;
// env_free_list : 用于存放空闲环境的链表(单向)
// env_sched_list : 调度用队列(双向尾队列)，用于调度可运行环境
        /* Exercise 3.1: Your code here. (1/2) */

        LIST_INIT((&env_free_list));// 初始化空点环境链表表头
        TAILQ_INIT(&env_sched_list);// 初始化调度队列

// 倒序遍历 envs 数组中的所有元素，把他们的状态设置成 ENV_FREE，并且把他们插入到 env_free_list(空闲环境链表)中，确保插入后他们的顺序和在 envs 数组中的顺序一致，所以我们是倒序插入实现的

        /* Exercise 3.1: Your code here. (2/2) */

        for (i = NENV - 1; i >= 0; --i) {
                envs[i].env_status = ENV_FREE;// 每个环境都标记为空闲状态
                LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);// 使用头插法，把环境插入到 env_free_list 的头部
        }

// 把 UPAGES 和 UENVS 映射到每个用户空间，并使用 PTE_G 权限，而不是用 PTE_D 权限，这样用户程序可以读取但不能写入内核数据结构 pages 和 envs，其中 UPAGES 和 UENVS：这是内核中定义的虚拟地址，分别用于表示页表(pages)和环境(envs)的物理内存，UPAGES 代表页面数据结构的虚拟地址，UENVS 代表环境数据结构的虚拟地址

// 首先用 page_alloc 分配一个空闲页，准备作为 base_pgdir，分配的同时表示该页已经被引用了一次
        struct Page *p;
        panic_on(page_alloc(&p));
        p->pp_ref++;

// 把当前物理页使用 page2kva 转化为内核虚拟地址，作为初始页目录，其中 base_pgdir 是一个内核级的模版页目录，用户空间页表会拷贝它
        base_pgdir = (Pde *)page2kva(p);
```

```
// 注释中说的 UPAGES 和 UENVS 在我们的程序中就是 pages 和 envs，我们通过 map_segment 函数分别把他们映射到用户空间的 UPAGES 和 UENVS 区域
        map_segment(base_pgdir, 0, PADDR(pages), UPAGES,
                    ROUND(npage * sizeof(struct Page), PAGE_SIZE), PTE_G);
        map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct Env), PAGE_SIZE),
                    PTE_G);
}

/* Overview
  函数功能：初始化给定进程 e 的用户地址空间，这里的地址空间主要指的是进程页目录的地址空间
  入参含义：给定一个存放环境信息的结构体 e
 */
static int env_setup_vm(struct Env *e) {
// 使用 page_alloc 为进程 e 所对应的页目录分配一页物理内存页，增加该物理页的引用计数 pp_ref，并且将该物理页转化为内核地址后分配给 e->env_pgdir
        struct Page *p;
        try(page_alloc(&p));
        /* Exercise 3.3: Your code here. */

        p->pp_ref++;
        e->env_pgdir = (Pde *)page2kva(p);

// 复制内核空间的映射，把模版页目录 base_pgdir 复制到 e->env_pgdir
        memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP),
               sizeof(Pde) * (PDX(UVPT) - PDX(UTOP)));

// 在 UVPT 处把环境自己的页表映射为只读权限，这样用户程序就可以通过 UVPT 读取自己的页表，具体实现上来看，就是把进程 e 对应的页目录 env_pgdir 的第 PDX(UVPT) 项中所存的物理地址，设置为 env_pgdir 自己本身的物理地址，并且仅设置有效位但不设置可写位，也就是构建页目录在 PDX(UVPT) 位置处的自映射
        e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
        return 0;
}

/* Ovewview
  函数功能：分配并初始化一个新的 Env 结构体，如果分配成功，新创建的 Env 会通过指针 *new 返回，具体来讲就是 env 结构体申请，地址空间分配，结构体内若干标志位的设置这几步
  前置条件：如果新的 Env 没有父进程，则 parent_id 应该为 0；必须在调用本函数之前先调用过 env_init
  后置条件：分配成功时返回 0，Env 的基本字段保证已经被初始化；失败时返回负值，可能是没有可用的 Env，没有可用的 ASID，或调用 env_setup_vm 失败
  入参含义：引用新环境的指针，父环境 ID
 */
int env_alloc(struct Env **new, u_int parent_id) {
        int r;
        struct Env *e;

// 从 env_free_list 中获得一个空闲的 Env
        /* Exercise 3.4: Your code here. (1/4) */

        e = LIST_FIRST(&env_free_list);// 取出链表第一个元素的宏
        if (e == NULL) {
                return -E_NO_FREE_ENV;
        }

// 设置地址空间，调用 env_setup_vm 初始化该环境的地址空间(分配页目录等)
        /* Exercise 3.4: Your code here. (2/4) */
```

```
206
207         if ((r = env_setup_vm(e)) != 0) {
208                 return r;
209         }
210
211 // 初始化以下四个字段
212         e->env_user_tlb_mod_entry = 0; // for lab4，用于 TLB 异常处理
213         e->env_runs = 0;                // for lab6，当前运行次数初始化为 0
214         /* Exercise 3.4: Your code here. (3/4) */
215
216 // 分配一个可用的 ASID
217         if ((r = asid_alloc(&e->env_asid)) != 0) {
218                 return r;
219         }
220 // mkenvid 函数为当前 Env 分配唯一的环境 ID，env_parent_id 用来记录其父环境的 ID
221         e->env_id = mkenvid(e);
222         e->env_parent_id = parent_id;
223
224 /* 初始化用户栈指针和 CP0 状态寄存器，CP0 寄存器中的几个标志位分别表示：
225     1.STATUS_IM7：开启中断 7(时钟中断)
226     2.TATUS_IE：开启中断使能
227     2.STATUS_EXL：异常级别，表示当前仍处于内核态(防止 ERET 立即切回用户态)
228     4.STATUS_UM：用户模式位设置为 1(在 ERET 后恢复用户态)
229   EXL 为 0 且 UM 为 1 时表示处理器处于用户模式，其余情况均处于内核模式
230 */
231         e->env_tf.cp0_status = STATUS_IM7 | STATUS_IE | STATUS_EXL |
    STATUS_UM;
232 // 设置栈指针($sp = 寄存器 29)，保留空间用于 argc 和 argv，这里的栈是用户栈而并非内核
    栈
233         e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);
234
235 // 从空闲链表中移除该 Env
236         /* Exercise 3.4: Your code here. (4/4) */
237
238         LIST_REMOVE(e, env_link);
239 // 把分配得到的 Env 用 *new 指针记录返回
240         *new = e;
241         return 0;
242 }
243
244 /* Overview
245   函数功能：为进程 data 分配一个物理页面，并且把 src 中长度为 len 的信息，加载到这个物
    理页面对应的内核虚拟地址中偏移 offset 的位置上，同时完成这个物理页面和虚拟地址 va 之间的
    映射
246   前置条件：offset + len 不能大于一页的大小 PAGE_SZIE
247   入参含义：当前正在被加载的进程(void *data 实际上应该是 struct Env *data)，该页应该
    被映射到的用户虚拟地址空间的位置，拷贝数据在页内的偏移，页的访问权限，数据源，要拷贝的字节
    数
248  */
249 static int load_icode_mapper(void *data, u_long va, size_t offset, u_int
    perm, const void *src, size_t len) {
250         struct Env *env = (struct Env *)data;// 先类型强转，变回 Env
251         struct Page *p;// 指向将来要分配的新物理页
252         int r;
253
254 // 分配一页内存，调用 page_alloc 进行分配，分配好的物理页存在 p 指针中
255         /* Exercise 3.5: Your code here. (1/2) */
256
```

```
257         if ((r = page_alloc(&p)) != 0) {
258                 return r;
259         }
260
261 // 从 src 拷贝内容到新页的 offset 位置，先把物理页面转化为内核虚拟地址，再加上偏移量就是拷贝的起始位置，拷贝数据的长度为 len
262         if (src != NULL) {
263                 /* Exercise 3.5: Your code here. (2/2) */
264                 memcpy((void *)page2kva(p) + offset, src, len);
265         }
266
267 // 使用 page_insert 函数把页面 p 插入到 env 的页目录中
268         return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
269 }
270
271 /* Overview
272   函数功能：把 binary 所指向的 ELF 文件(可执行文件)的内容加载到环境 e 对应的用户空间中，binary 是指向一个 ELF 可执行文件的指针，它的大小是 size 字节，这个文件中包含了代码段(text segment)和数据段(data segment)
273   入参含义：
274     Env *e：目标进程(环境)的指针，表示当前正在创建或初始化的进程
275     void *binary：指向 ELF 格式的二进制文件内容的内存映像
276     size：ELF 文件的总字节数(binary 文件的大小)
277  */
278 static void load_icode(struct Env *e, const void *binary, size_t size) {
279 // 使用 elf_from 函数，从 binary 中解析出 ELF 头指针 Elf32_Ehdr，存入 ehdr 指针中
280         const Elf32_Ehdr *ehdr = elf_from(binary, size);
281         if (!ehdr) {
282                 panic("bad elf at %x", binary);
283         }
284
285 // 加载程序段：定义变量 ph_off，用于存储每个程序头(Elf32_Ehdr)的偏移地址，宏 ELF_FOREACH_PHDR_OFF 用于遍历 ELF 文件中的每个程序头，每次迭代设置 ph_off 为下一个程序头的偏移
286         size_t ph_off;
287         ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
288         // 通过偏移值 ph_off 从 binary 中得到一个程序头指针 ph，可以认为 binary 是起始地址，ph_off 是偏移量，二者相加就得到当前的程序头指针 ph
289                 Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
290         /* 检查这个段是否为 PT_LOAD 类型，只有这种段才需要映射进用户地址空间，也就是调用 elf_load_seg 函数，把该段从 ELF 文件加载到用户空间，其中
291             ph 为段头结构体
292             binary+ph->p_offset 为该段在文件中的起始地址
293             load_icode_mapper 是指示如何映射页的函数指针
294             e 是目标环境
295           其中 elf_load_seg 会根据段头信息，分配内存页，把 ELF 文件中的数据复制到内存，并且完成虚拟地址映射
296         */
297                 if (ph->p_type == PT_LOAD) {
298                         panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper, e));
299                 }
300         }
301
302 // 设置进程的用户态入口地址(执行的起点)，CP0_epc 是 MIPS 中返回用户态时候的 PC 寄存器，e_entry 是 ELF 文件中的入口地址字段，当调度器最终运行该环境(调用 env_run(e))时，会根据 cp0_epc 设置 PC，使得 CPU 从入口地址开始执行用户程序
303         /* Exercise 3.6: Your code here. */
```

```
304
305         e->env_tf.cp0_epc = ehdr->e_entry;
306
307 }
308
309 /* Overview
310   函数功能：创建一个带有指定 binary(程序)和优先级(priority)的新环境(进程)，这个函数仅用于系统初始化阶段从内核创建早期环境，在第一个环境被调度执行之前，其中 binary 是内存中的 ELF 可执行文件镜像
311   入参含义：
312     binary：指向 ELF 文件内容的内存指针
313     size：ELF 文件的大小
314     priority：新环境的调度优先级
315   返回值：新环境(进程)的结构体指针
316 */
317 struct Env *env_create(const void *binary, size_t size, int priority) {
318         struct Env *e;
319 // 定义一个指向 struct Env 类型的指针变量 e，用来存储新创建的环境结构体地址，调用 env_alloc 分配一个新环境，并把结果赋值到 e，其中第二个参数 0 代表父进程 ID，因为是初始化，所以此时没有父进程
320         /* Exercise 3.7: Your code here. (1/3) */
321
322         panic_on(env_alloc(&e, 0));
323
324 // 设置优先级和状态，分别把 priority 赋值给 e->env_pri，把 ENV_RUNNABLE 赋值给 e->env_status，只有该状态的进程才能被调度器选中并运行
325         /* Exercise 3.7: Your code here. (2/3) */
326
327         e->env_pri = priority;
328         e->env_status = ENV_RUNNABLE;
329
330 // 加载程序并插入调度队列，调用 load_icode 函数把 ELF 格式的程序加载到该环境 e 的地址空间，其中 binary 是程序镜像的起始地址，size 是大小，同时调用 BSD 宏，把新创建的环境插入调度队列 env_sched_list 的队头
331         /* Exercise 3.7: Your code here. (3/3) */
332
333         load_icode(e, binary, size);
334         TAILQ_INSERT_HEAD(&env_sched_list, e, env_sched_link);
335
336         return e;
337 }
338
339 /* Overview
340   函数功能：释放环境 e 以及其使用的所有内存
341  */
342 void env_free(struct Env *e) {
343         Pte *pt;
344         u_int pdeno, pteno, pa;
345
346         /* 记录环境(进程)的终结信息 */
347         printk("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e->env_id);
348
349         /* 清除用户地址空间中所有已经映射的页 */
350         for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
351                 /* 只检查那些被映射的页表 */
352                 if (!(e->env_pgdir[pdeno] & PTE_V)) {
353                         continue;
```

```
354                 }
355                 /* 找到页表对应的物理地址和虚拟地址 */
356                 pa = PTE_ADDR(e->env_pgdir[pdeno]);
357                 pt = (Pte *)KADDR(pa);
358                 /* 取消该页表中所有页表项(PTE)的映射 */
359                 for (pteno = 0; pteno <= PTX(~0); pteno++) {
360                         if (pt[pteno] & PTE_V) {
361                                 page_remove(e->env_pgdir, e->env_asid,
362                                             (pdeno << PDSHIFT) | (pteno <<
PGSHIFT));
363                         }
364                 }
365                 /* 释放页表本身所占用的物理页 */
366                 e->env_pgdir[pdeno] = 0;
367                 page_decref(pa2page(pa));
368                 /* 从 TLB 中失效该页表的映射 */
369                 tlb_invalidate(e->env_asid, UVPT + (pdeno << PGSHIFT));
370         }
371         /* 释放页目录 */
372         page_decref(pa2page(PADDR(e->env_pgdir)));
373         /* 释放地址空间标识符 */
374         asid_free(e->env_asid);
375         /* 从 TLB 中使页目录失效 */
376         tlb_invalidate(e->env_asid, UVPT + (PDX(UVPT) << PGSHIFT));
377         /* 将该环境返回到空闲环境链表中 */
378         e->env_status = ENV_FREE;
379         LIST_INSERT_HEAD((&env_free_list), (e), env_link);
380         _REMOVE(&env_sched_list, (e), env_sched_link);
381 }
382
383 /* Ovrview
384   函数功能：释放或销毁环境(进程) e，如果 e 是当前运行的进程，则需要调度一个新的进程来运行
385   入参含义：即将被销毁的进程 e
386  */
387 void env_destroy(struct Env *e) {
388 // 直接调用 env_free 函数进行进程销毁即可
389         env_free(e);
390
391 // 如果销毁的进程是当前正在执行的进程，则首先把当前进程指向 NULL，再通过调度器 schedule
调度一个新的进程运行，1 通常作为一个标志参数(如当前环境被杀死，或者强制切换等)
392         if (curenv == e) {
393                 curenv = NULL;
394                 printk("i am killed ... \n");
395                 schedule(1);
396         }
397 }
398
399 // WARNING BEGIN: DO NOT MODIFY FOLLOWING LINES!
400 #ifdef MOS_PRE_ENV_RUN
401 #include <generated/pre_env_run.h>
402 #endif
403 // WARNING END
404
405 extern void env_pop_tf(struct Trapframe *tf, u_int asid)
__attribute__((noreturn));
406
407 /* Overview:
408   函数功能：实现运行中进程的切换
```

```
  后置条件：把 e 设置为当前正在运行的环境 curenv(当前正在运行的环境，全局唯一)
  入参含义：进程上下文要切换到的目标环境(进程)的结构体指针
 */
void env_run(struct Env *e) {
// 首先断言 e 是可运行状态，才能进行切换
        assert(e->env_status == ENV_RUNNABLE);
//-------------------------
// 这部分是系统用来调试或统计信息的宏，不需要修改
#ifdef MOS_PRE_ENV_RUN
        MOS_PRE_ENV_RUN_STMT
#endif
//-------------------------

// 如果当前有正在运行的进程，就保存这个进程有关的上下文信息，具体来讲就是把用户态 CPU 寄存器中的值先压入内核栈，再由内核栈存入当前进程 curenv 的 env_tf 中，因为陷入内核是对内核栈当下的最后一次操作，因此有关信息就存在内核栈顶
        if (curenv) {
                curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
        }

// 切换当前环境变量(进程)为 e， 环境的运行计数器 env_runs 自增 1，用来表示该进程被调度运行的次数
        curenv = e;
        curenv->env_runs++;

// 切换地址空间，系统要运行用户程序，必须切换到用户自己的地址空间，这里把全局页目录变量 cur->pgdir 设置为当前进程的页表 env->pgdir
        /* Exercise 3.8: Your code here. (1/2) */

        cur_pgdir = curenv->env_pgdir;

// 通过 env_pop_tf 函数，把当前进程 curenv 的 env_tf 数组中存储的进程上下文信息恢复到用户态 CPU 的寄存器中
        /* Exercise 3.8: Your code here. (2/2) */

        env_pop_tf(&curenv->env_tf, curenv->env_asid);

}
```

```
void schedule(int yield) {
        static int count = 0; // 剩余时间片
        struct Env *e = curenv;

/*
我们总是将 count 减 1
如果设置了 'yield'(表示主动让出 CPU)，或者 count 已经减到 0，或者 e(也就是之前运行的 curenv)是 NULL，或者 e 的状态不是 RUNNABLE(不可运行)，那么我们需要从 env_sched_list(表示所有可运行环境的队列)中取出一个新的环境（进程）来运行，并将 count 设置为它的优先级，然后用 env_run 来调度它，如果队列为空就 panic
注意，如果 e 仍然是一个可运行的进程，我们应该把它从队列头移动到队列尾，这样我们才能真正地轮转调度进程，不然总是调度队列头那个进程，否则，我们就继续调度当前的 e 即可
 */

*/
        /* Exercise 3.12: Your code here. */
        if (yield || count == 0 || e == NULL || e->env_status !=
ENV_RUNNABLE) {
```

```
                if (e != NULL && e->env_status == ENV_RUNNABLE) {
                        TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
                        TAILQ_INSERT_TAIL(&env_sched_list, e,
env_sched_link);
                }
                e = TAILQ_FIRST(&env_sched_list);
                if (e == NULL) {
                        panic("schedule: no runnable envs\n");
                }
                count = e->env_pri;
        }
        count--;
        env_run(e);
}
```

```
// 全局时钟计数器
static u_int global_time = 0;

// 用于比较两个环境的截止时间
static int compare_deadline(struct Env *a, struct Env *b) {
    return a->env_deadline - b->env_deadline;
}

// 更新任务的截止时间
static void update_deadline(struct Env *e) {
    if (e->env_period > 0) {
        // 计算下一个周期的截止时间
        e->env_deadline +=  e->env_period;
        // 重置剩余执行时间
        e->env_remaining_time = e->env_execution_time;
    }
}

// 从调度队列中找到最早截止时间的任务
static struct Env* find_earliest_deadline() {
    struct Env *e;
    struct Env *earliest = NULL;

    TAILQ_FOREACH(e, &env_sched_list, env_sched_link) {
        if (e->env_status == ENV_RUNNABLE) {
            if (earliest == NULL || e->env_deadline < earliest-
>env_deadline) {
                earliest = e;
            }
        }
    }
    return earliest;
}

void schedule(int yield) {
    struct Env *next_env = NULL;

    // 更新全局时间
    global_time++;

    // 如果当前进程存在且运行完一个时间片
    if (curenv) {
```

```
        if (curenv->env_remaining_time > 0) {
            curenv->env_remaining_time--;
        }

        // 检查是否错过截止时间
        if (global_time > curenv->env_deadline) {
            printk("Process %08x missed deadline at time %d\n",
                   curenv->env_id, global_time);
        }

        // 如果当前进程完成了本次执行
        if (curenv->env_remaining_time == 0) {
            update_deadline(curenv);
            // 将进程重新插入调度队列
            TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);
            TAILQ_INSERT_TAIL(&env_sched_list, curenv, env_sched_link);
        }
    }

    // 选择下一个要执行的进程
    if (yield || !curenv || curenv->env_remaining_time == 0) {
        next_env = find_earliest_deadline();

        if (!next_env) {
            panic("schedule: no runnable env");
        }

        // 如果是新的周期开始，更新相关参数
        if (next_env->env_last_schedule == 0 ||
            global_time >= next_env->env_deadline) {
            update_deadline(next_env);
        }

        next_env->env_last_schedule = global_time;

        // 从队列中移除并重新插入到尾部
        TAILQ_REMOVE(&env_sched_list, next_env, env_sched_link);
        TAILQ_INSERT_TAIL(&env_sched_list, next_env, env_sched_link);

        env_run(next_env);
    } else {
        // 继续执行当前进程
        env_run(curenv);
    }
}
```

课上 exam

传统时间片调度

就是我们课下实现的 schedule 函数的本体，后面两个函数都是在这个函数的基础上进行修改的

```
// 基础时间片轮转
void schedule(int yield) {
  static int count = 0;
  struct Env *e = curenv;
  if (yield || count == 0 || e != NULL || e->env_status != ENV_RUNNABLE) {
```

```
    if (e != NULL && e->env_status == ENV_RUNNBALE) {
      TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
      TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
    }
    e = TAILQ_FIRST(&env_sched_list, e, env_sched_link);
    if (e == NULL) {
      panic("schedule: no runnable envs\n");
    }
    count = e->env_pri;
  }
  count--;
  env_run(e);
}
```

多用户调度

题目要求：要求改进 schedule 函数，实现了一个**基于多用户时间公平性**的调度器，它在传统的时间片轮转（Round-Robin）算法基础上引入了用户分组的概念，每个用户（共 5 个，用 0~4 表示）被公平分配时间片，优先调度**所占用时间片最少的用户**中的一个进程运行（其中每个用户所占的时间片是所有属于这个用户的进程的时间片累加之和，可以多次计算，如果某个用户的进程多次运行，那么每次都要累加相应的时间片）

具体代码：

```
void schedule(int yield) {
  static int count = 0;// 时间片
  static int user_time[5] = {0};// 记录每个用户所占有的总时间片
  static int able[5] = {0};// 记录每个用户是否在调度队列中有合法进程
  struct Env *e = curenv;

  for (int i = 0; i < 5; i++) {
    able[i] = 0;
  }

  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);
      user_time[e->env_user] += e->env_pri;// 进程每次运行，都要给对应用户的
times 加上一次它的时间片
    }
    if (TAILQ_EMPTY(&env_sched_list)) {
      panic("schedule: no runnable envs\n");
    }

    // 正常时间片轮转得到的进程
    e = TAILQ_FIRST(&env_sched_list);
    count = e->env_pri;

    // 遍历调度队列，设置 able 数组
    struct Env *en = NULL;
    TAILQ_FOREACH(en, &env_sched_list, env_sched_link) {
      able[en->env_user] = 1;
    }

    // 寻找总时间片最少的用户
    int user = -1;
```

```
    u_int times = 0x3f3f3f3f;
    for (int j = 0; j < 5; j++) {
      if (user_time[j] < times && able[j]) {
        times = user_time[j];
        user = j;
      }
    }

    // 遍历调度队列，找到 user 对应的进程
    TAILQ_FOEACH(en, &env_sched_list, env_sched_link) {
      if (en->env_user == user) {
        e = en;
        count = e->env_pri;
        break;
      }
    }
  }
  count--;
  env_run(e);
}
```

多队列调度

调度规则：在全局有三个调度队列，记为 env_sched_list[3] ，每个队列中的进程单次运行的时间片数量为进程优先级乘以不同的权重，具体的：

env_sched_list[0] 中进程单次运行时间片数 = 进程优先级数 * **1**；
env_sched_list[1] 中进程单次运行时间片数 = 进程优先级数 * **2**；
env_sched_list[2] 中进程单次运行时间片数 = 进程优先级数 * **4**；

进程创建时全部插入到第一个调度队列（即 env_sched_list[0] ）的**队首**进程时间片用完后，根据自身优先级数值加入到另外两个调度队列**队尾**，若自身优先级为**奇数**，则顺次加入到**下一个调度队列**队尾，若为**偶数**，则加入到**下下个调度队列**队尾。当然，进程不再处于原调度队列中。具体地：

env_sched_list[0] 中的进程时间片用尽，若优先级为奇数，加入到 env_sched_list[1] 队尾；若为偶数，加入到env_sched_list[2] 队尾；env_sched_list[1] 中的进程时间片用尽，若优先级为奇数，加入到 env_sched_list[2] 队尾；若为偶数，加入到 env_sched_list[0] 队尾；env_sched_list[2] 中的进程时间片用尽，若优先级为奇数，加入到 env_sched_list[0] 队尾；若为偶数，加入到 env_sched_list[1] 队尾；

schedule 函数首先从 env_sched_list[0] 队列开始调度，之后依次按照 0 , 1 , 2 , 0 , 1 , ...的顺序切换队列，且**仅在当前队列中没有可运行进程时切换到下一个队列**

代码实现

```

void schedule(int yield) {
    static int current_queue = 0;       // 当前调度的队列索引（0, 1, 2）
    static int count = 0;               // 当前进程剩余的时间片
    struct Env *e = curenv;

    // 如果主动 yield、时间片用完、当前进程不可运行或无当前进程，则重新调度
    if (yield || count == 0 || e == NULL || e->env_status != ENV_RUNNABLE) {
        // 如果当前进程仍可运行，则根据规则将其移动到下一个队列
        if (e != NULL && e->env_status == ENV_RUNNABLE) {
            // 从当前队列中移除
            TAILQ_REMOVE(&env_sched_list[current_queue], e, env_sched_link);
```

```
13
14              // 时间片用尽，进行跨队列调度
15             if (count == 0) {
16              // 根据优先级奇偶性选择目标队列
17                int next_queue;
18                if (current_queue == 0) {
19                    next_queue = (e->env_pri % 2 == 1) ? 1 : 2;
20                } else if (current_queue == 1) {
21                    next_queue = (e->env_pri % 2 == 1) ? 2 : 0;
22                } else { // current_queue == 2
23                    next_queue = (e->env_pri % 2 == 1) ? 0 : 1;
24                }
25              } else {
26                // 时间片未用尽回到原队列队尾
27                TAILQ_INSERT_TAIL(&env_sched_list[current_queue], e,
env_sched_link);
28              }
29
30              // 插入到目标队列的队尾
31              TAILQ_INSERT_TAIL(&env_sched_list[next_queue], e,
env_sched_link);
32          }
33
34          // 按顺序查找下一个可调度的队列（0 -> 1 -> 2 -> 0 -> ...）
35          int tried_queues = 0;
36          do {
37              e = TAILQ_FIRST(&env_sched_list[current_queue]);
38              if (e != NULL) {
39                  break; // 找到可调度的进程
40              }
41              current_queue = (current_queue + 1) % 3;
42              tried_queues++;
43          } while (tried_queues < 3);
44
45          // 如果没有可运行进程，则 panic（其实系统保证一定有可运行进程，不会找不到）
46          if (e == NULL) {
47              panic("schedule: no runnable envs\n");
48          }
49
50          // 设置时间片 = 优先级 * 权重（权重为 1, 2, 4）
51          count = e->env_pri * (1 << current_queue); // 1<<0=1, 1<<1=2, 1<<2=4
52      }
53
54      count--;
55      env_run(e);
56  }
```

EDF 调度

```
1  typedef struct Env {
2      int env_id;
3    int env_deadline;
4    int env_start_time;
5    int env_period;
6    int env_execution_time;
7    int env_remaining_time;
8  }Env;
```

```
9
10  Env* a;
11  Env *b;
12  vector<Env*>q;
13  Env* curenv = NULL;
14
15  // 全局时钟计数器
16  static int global_time = 0;
17
18  // 更新任务的时间信息
19  static void update_deadline(struct Env* e) {
20      if (e->env_period > 0) {
21          // 重置起始时间
22          e->env_start_time = e->env_deadline;
23          // 计算下一个周期的截止时间
24          e->env_deadline += e->env_period;
25          // 重置剩余执行时间
26          e->env_remaining_time = e->env_execution_time;
27      }
28  }
29
30  // 从调度队列中找到最早截止时间且可以执行的任务
31  static struct Env* find_earliest_deadline() {
32      struct Env* e;
33      struct Env* earliest = NULL;
34
35      for (auto e : q) {
36          // 起始时间不能晚于全局时间，否则这样的任务无法开始
37          if (e->env_start_time <= global_time) {
38              if (earliest == NULL || e->env_deadline < earliest-
    >env_deadline) {
39                  earliest = e;
40              }
41          }
42      }
43      return earliest;
44  }
45
46  void schedule() {
47      struct Env* e = curenv;
48      struct Env* next_e = find_earliest_deadline();
49
50      // 进行抢占替换，被替换者写回队列
51      if (!e || next_e->env_deadline < e->env_deadline) {
52          if (e) {
53              for (int i = 0; i < q.size(); i++) {
54                  if (q[i]->env_id == e->env_id) {
55                      q[i] = e;
56                      break;
57                  }
58              }
59          }
60          e = next_e;
61      }
62
63      // 更新全局时间
64      global_time++;
65      if (e == a)printf("%d : a running\n", global_time);
```

```
    if (e == b)printf("%d : b running\n", global_time);

    // 如果当前进程存在，就让他执行一个时间片
    if (e != NULL) {
        // 剩余时间--
        if (e->env_remaining_time > 0) {
            e->env_remaining_time--;
        }

        // 检查是否错过截止时间
        if (global_time > e->env_deadline) {
            printf("missing the deadline!\n");
        }

        // 如果当前进程完成了本次执行
        if (e->env_remaining_time == 0) {
            // 更新进程有关信息
            update_deadline(e);
            for (int i = 0; i < q.size(); i++) {
                if (q[i]->env_id == e->env_id) {
                    q[i] = e;
                    break;
                }
            }
            // 无条件替换
            next_e = find_earliest_deadline();
            e = next_e;
        }
    }

    if (!e) {
        printf("curenv is NULL!\n");
    }

    // run...
    curenv = e;
}

signed main()
{
    // 模拟初始化
    a = (Env*)malloc(sizeof(Env)), b = (Env*)malloc(sizeof(Env));
    a->env_id = 1, b->env_id = 2;
    a->env_start_time = 0, b->env_start_time = 0;
    a->env_deadline = 10, b->env_deadline = 25;
    a->env_execution_time = a->env_remaining_time = 5, b-
>env_execution_time = b->env_remaining_time = 12;
    a->env_period = 10, b->env_period = 25;
    // 压入调度队列
    q.push_back(a), q.push_back(b);
    // 模拟全局时钟
    for (; global_time < 50;) {
        schedule();
    }
    return 0;
}
```

感想

本次 exam 的难度稍有增加，和理论课进行了一定程度的衔接，因此以后要持续注重理论的学习；此外需要注重不同 lab 之间的连贯性，比如在 lab2 中的页表机制以及 lab1 中的总体内存分配在 lab3 中都有很重要的应用，因此对每个 lab 都不能心存侥幸，即便本单元所涉及不多，也要严肃对待，防止出现知识体系漏洞百出的情况