Lab3

思考题

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。 ■

- 在MOS中，`e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V` 实现了页目录的自映射机制：
  - UVPT 是用户空间虚拟地址的顶部，通常设置为 `0x7FC00000`，其页目录索引（`PDX(UVPT)`）正好指向页目录自身。
  - PADDR(e->env_pgdir) 获取进程页目录的物理地址，并写入到页目录的最后一个条目（即 `PDX(UVPT)` 对应的条目）。
  - PTE_V 标记该条目有效，使得用户程序可以通过 `UVPT` 访问自己的页目录，实现页表自映射。

  这样，用户程序可以通过 `UVPT` 访问自己的页表结构，而无需切换到内核模式，便于用户态管理页表（如 `fork` 时的 `duppage` 操作）。

Thinking 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？ ■

- 首先这里的回调函数是定义在 `kern/env.c` 的 `load_icode_mapper()` 函数，作用是把一段虚拟地址的内容**加载**到**某个进程管理块对应进程**的虚拟内存中（通过申请物理页面并建立页表映射）。这两个函数中的 `data` 参数相同，其来源是待加载的进程管理块指针，它用来告知 `load_icode_mapper()` 函数加载到哪个进程。所以**不可以**没有这个参数，如果没有这个参数，将无法确定加载到哪个进程的虚拟内存中。如果没有data，`load_icode_mapper` 就不能知道当前进程空间的**页目录基地址**和**asid**，所以必须要有这个参数。

Thinking 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BY2PG的binary数据写入对应页的对应地址；（第一个 `map_page`）
- 然后，依次将段内的页映射到物理空间；（第二个 `map_page`）
- 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。（第三个 `map_page`）

Thinking 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？
>
> ■

- 存储的应当为虚拟地址。因为epc存储的是发生错误时CPU所处的指令地址，那么对于CPU来说，所见的都是虚拟地址（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。

Thinking 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。 ■

- 0号异常的处理函数为handle_int，具体实现位置在kern/genex.S的NESTED(handle_int, TF_SIZE, zero)部分，而1、2、3号异常的处理具体实现在后面的BUILD_HANDLER部分。

Thinking 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。 ■

- **时钟中断关闭的时机**：
  - 异常/中断处理入口处：在`exc_gen_entry`中，通过`mfc0 t0, CP0_STATUS`和`and t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)`指令，清除了STATUS_IE位，这会全局关闭中断（包括时钟中断）。
  - 中断处理过程中：在`handle_int`中，虽然检测到了中断（包括时钟中断），但在处理过程中没有重新开启中断的指令，保持中断关闭状态。
  - 从异常返回前：在`ret_from_exception`中，虽然使用了`eret`返回，但之前的状态寄存器是通过`RESTORE_ALL`恢复的，如果之前的状态是关闭中断的，那么中断仍保持关闭。

  时钟中断开启的时机：
  - 正常执行流程中：当CPU不在异常处理状态（STATUS_EXL=0），且STATUS_IE位被设置时，时钟中断是开启的。
  - 通过`eret`指令返回后：`eret`指令会恢复异常前的状态寄存器，如果之前的状态是开启中断的，那么中断会被重新开启。
  - 在`env_pop_tf`中：虽然代码中没有直接设置STATUS寄存器，但通过`j ret_from_exception`和随后的`eret`，最终会恢复之前保存的状态，可能开启中断。

Thinking 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，我们在`kern/kclock.S`中的`kclock_init`函数完成了时钟中断的初始化，并在`genex.S`中的`enable_irq`中设置允许响应该中断

  当时钟计时归零（时间片耗尽），产生时钟中断，进入异常处理程序，并跳转到`handle_int`处理中断。

  当前我们的系统只能处理`timer_irq`一种时钟中断，所以直接进入`timer_irq`函数，恢复时钟，并执行`schedule(0);`

  在`schedule()`中，我们根据当前进程块状态进行进程块队列调度，实现进程切换或进程的舍弃，至此完成时钟中断响应

难点分析

- 首先需要理解进程管理的基本框架。实验通过进程控制块(PCB)来管理进程，核心数据结构是struct Env，它包含了进程运行所需的所有关键信息，如进程ID、状态、页表地址、调度优先级等。其中env_tf保存了进程的寄存器上下文，env_pgdir指向进程的页目录，这两个字段尤为重要。实验开始时需要完成env_init函数来初始化空闲进程链表和调度队列，注意要按照倒序插入Env控制块。
- 其次是进程创建和地址空间设置的实现。这部分需要完成env_setup_vm函数来初始化新进程的地址空间，包括分配页目录、复制内核共享的页表区域以及设置页目录自映射。env_alloc函数则负责从空闲链表中获取Env结构并初始化各个字段，特别要注意正确设置cp0_status寄存器，需要包含STATUS_IM7、STATUS_IE、STATUS_EXL和STATUS_UM这几个关键位。此外还要设置好用户栈指针的位置。
- 然后是程序加载机制的实现。通过load_icode函数将ELF格式的可执行文件加载到进程地址空间，这需要配合load_icode_mapper回调函数来完成实际的页面分配和映射工作。加载完成后要设置env_tf.cp0_epc指向程序入口地址，这个地址是虚拟地址而非物理地址。env_create函数封装了上述功能，完成从创建进程到加载程序的全过程。
- 最后是中断处理和进程调度的核心部分。实验需要实现时钟中断机制，通过配置CP0的Count和Compare寄存器来产生周期性中断。当时钟中断发生时，CPU会跳转到异常入口地址执行分发代码，最终调用handle_int处理函数。schedule函数实现了基于时间片的轮转调度算法，根据进程优先级分配时间片长度，在时钟中断触发时进行进程切换。特别要注意保存和恢复进程上下文的正确性，以及调度队列的管理逻辑。

实验体会

- 这次实验是PP调度：

  在操作系统中，**PP调度（Priority Preemptive Scheduling，优先级抢占式调度）**是一种基于优先级的进程调度算法，具有以下特点：

  1. **优先级决定执行顺序**
     - 每个进程被赋予一个优先级（Priority），优先级高的进程优先执行。
     - 在Lab3中，`env_pri` 表示进程的优先级，数值越大优先级越高（或越小优先级越高，取决于具体实现）。
  2. **抢占式调度（Preemptive）**
     - 当高优先级进程就绪时，可以抢占当前正在运行的低优先级进程。
     - 例如，在时钟中断（Timer Interrupt）发生时，调度器会检查是否有更高优先级的进程就绪，若有则切换。
  3. **时间片轮转（Round-Robin）结合优先级**
     - 相同优先级的进程采用时间片轮转（RR）方式调度。
     - 不同优先级的进程，高优先级进程先执行，直到完成或阻塞。

```
#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 clock = -1;
        clock++;

        struct Env *env;
        LIST_FOREACH (env, &env_edf_sched_list, env_edf_sched_link) {
                if (clock == env->env_period_deadline) {
                        env->env_period_deadline += env->env_edf_runtime;
                        env->env_runtime_left = env->env_edf_runtime;
                }
        }

        u_int min = 1111111;
        struct Env *m_env = NULL;
        LIST_FOREACH (env, &env_edf_sched_list, env_edf_sched_link) {
                if (env->env_runtime_left > 0 && env->env_period_deadline <= min) {
                        if (env->env_period_deadline == min && env->env_id <  m_env->env_id)
{
                                m_env = env;
                        } else if (env->env_period_deadline < min) {
                                m_env = env;
                                min = env->env_period_deadline;
                        }
                }
        }

        static struct Env* last_e = NULL;
        static int last_count = 0;
        static int count = 0; // remaining time slices of current env
        static int last = -1;
        if (m_env != NULL) {
                if (last == -1) {
                        last_e = curenv;
                        last_count = count;
                        last++;
                }
        } else {
                curenv = last_e;
                count = last_count;
                last = -1;
        }

        if (m_env != NULL) {
                env->env_runtime_left--;
```

```
                env_run(m_env);
                return;
        }

        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 != 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);
}
```