BUAA-OS-lab3

发表于 2023-04-18 | 更新于 2023-06-19 | 养德（学习）
| 字数总计: 5.5k | 阅读时长: 20分钟 | 阅读量: 930

Lab3实验报告

思考题

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。

- 解：
  - 左边：

    `UVPT`： **用户页表的起始处**的**内核虚拟地址**

    `PDX(UVPT)`： *UVPT*所处的**页目录号**（即 *UVPT* 处于第 *PDX(UVPT)* 个页目录项所映射的4MB空间；联系 *UVPT* 的含义，因此**页目录**也被第PDX(UVPT)映射）
  - 右边：

    `e->env_pgdir`： 进程 e 的**页目录**的**内核虚拟地址**

`PADDR(e->env_pgdir)`：进程 e 的**页目录**的**物理地址**

`PADDR(e->env_pgdir) | PTE_V`：页目录的物理基地址，加上权限位

- 页表基地址：*UVPT*

  页目录基地址：*UVPT+UVPT>>10*

  映射到页目录的页目录项的基地址：*UVPT+UVPT>>10+UVPT>>20*

  该页表项处于第几个页目录项：*(UVPT>>20) >> 2 = UVPT>>22 = PDX(UVPT)*

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？

- 解：
  - *data* 是传入的进程控制块指针，在 `load_icode_mapper` 和 `load_icode` 函数中被调用，在 `load_icode` 函数中 *data* 被赋予进程控制块的指针 *e*：

```
1   static int load_icode_mapper(void *data, u_long va,
2
3   static void load_icode(struct Env *e, const void *b
4           //...
5       panic_on(elf_load_seg(ph, binary + ph->p_offset
6       //...
7   }
```

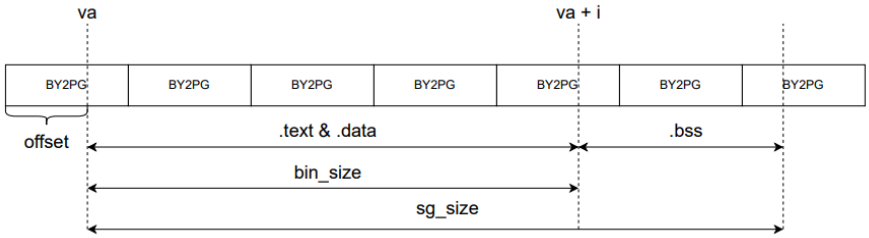
  - 作用：在增加虚拟地址到物理地址映射的时候提供env_pgdir和env_asid；

    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

> 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。

- 解：



- 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

> 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?

- 解：
  - 存储的是虚拟地址；
  - 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

> 试找出上述 5 个异常处理函数的具体实现位置。

- 解：
  - **handle_int**在**genex.S**文件中

    handle_mod和**handle_tlb**，**handle_sys**藏得比较深，三者都是通过**genex.S**文件中的宏函数**BUILD_HANDLER**实现的

Thinking 3.6

> 阅读 $init.c$、$kclock.S$、$env_asm.S$ 和 $genex.S$ 这几个文件，并尝试说出 `enable_irq` 和 `timer_irq` 中每行汇编代码的作用。

- 解：
  - enable_irq 函数：

```
LEAF(enable_irq)
        li      t0, (STATUS_CU0 | STATUS_IM4 | STAT
        //将t0赋为打开时钟中断的值：IM4（即时钟中断）
        mtc0    t0, CP0_STATUS
        //将t0的值赋给SR寄存器，使之可以相应时钟中断
        jr      ra
        //返回
END(enable_irq)
```

  - timer_irq 函数：

```
timer_irq:
        sw      zero, (KSEG1 | DEV_RTC_ADDRESS | D
        //写 KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_INT
        //KSEG1 | DEV_RTC_ADDRESS 是模拟器（GXemul)
        //偏移量为 DEV_RTC_INTERRUPT_ACK 表示发生时
        li      a0, 0
        //传参 让 a0 为 0
        j       schedule
        //跳转到 schedule 调度函数，进行进程调度
END(handle_int)
```

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。

- 解：
  - 首先，模拟器通过 kclock_init 函数完成时钟中断的初始化，设置了时钟中断发生的频率；

    然后，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，

  1. 将当前运行的进程控制块 curenv 取出来；
  2. 对当前进程的可用时间片数量减一（即 count--; ）；
  3. 当满足以下四种条件之一：
     - 尚未调度过任何进程（ curenv == NULL ）
     - 当前进程已经用完了时间片（ count == 0 ）
     - 当前进程不再就绪，如：被阻塞或退出（ e->env_status != ENV_RUNNABLE ）
     - yield 参数指定必须发生切换（ yield != 0 ）

     则进行进程切换：
     1. 判断若 curven 的状态为 ENV_RUNNABLE ，则把 curven 再次插入调度队列的尾部；
     2. 从调度队列头取出一个进程；
     3. 将剩余时间片数量设置为新进程的优先级；
  4. env_run 运行当前选中进程

实验体会

进程

进程控制块 PCB

```
1   struct Env {
2           struct Trapframe env_tf; // 进程上下文（“进程调度”或
3           LIST_ENTRY(Env) env_link; // 空闲队列env_free_list
4       u_int env_id; // 进程的唯一标识符
5           u_int env_parent_id; // 父进程的唯一标识符
6           u_int env_status; // PCB状态 或 进程状态
7           Pde *env_pgdir; // 进程【页目录】的【内核】【虚拟地址
8           TAILQ_ENTRY(Env) env_sched_link; // 调度队列env_sc
9           u_int env_pri; // 进程优先级
10  };
```

- *env_status*：
  - *ENV_FREE*：该**PCB进程控制块**没有被使用，处于*env_free_list* ；
  - *ENV_NOT_RUNNABLE*：进程处于**阻塞状态**；
  - *ENV_RUNNABLE*：进程处于**执行状态**或**就绪状态**。

- *envs* 数组里面存了所有进程控制块，我们用 *env_free_list* 和 *env_sched_list* 来使用进程块们

  env_free_list 用 `LIST_宏` 操作；*env_sched_list* 用 `TAILQ_宏` 操作。

段地址映射

- *base_pgdir* 是模板页表，**全局**变量，存有**内核虚拟地址**，由 `page_alloc` 函数申请而来，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** 。

```
1  //env.c : void env_init(void){}
2          //......
3          struct Page *p;
4          panic_on(page_alloc(&p));
5          p->pp_ref++;
6
7          base_pgdir = (Pde *)page2kva(p);
8          map_segment(base_pgdir, 0, PADDR(pages), UPAG
9          map_segment(base_pgdir, 0, PADDR(envs), UENVS
```

- `void map_segment(Pde *pgdir, u_long pa, u_long va, u_long size, u_int perm)`
  - 功能：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)* ，并将相关页表项的权限为设置为 perm。（因为是按页映射，要求 *size* 必须是页面大小的整数倍）
  - 使用例子：

```
1  map_segment(base_pgdir, 0, PADDR(pages), UPAGES, RO
```

  - 实现：在判断前提条件成立的条件下，遍历 *size* 每一页，用 `page_insert` 函数进行映射。

进程的标识

- *env_id* 不等于 *env_asid*，*env_id* 用 `mkenvid` 函数得到，*env_asid* 用 `asid_alloc` 函数得到。
- `u_int mkenvid(struct Env *e)`
  - 功能：生成一个新的 *env_id*

- 使用例子：

```
1   e->env_id = mkenvid(e); //env.c : int env_alloc(str
```

- 由 *TLB* 结构的 *Key Fields*（也就是 *EntryHi* 寄存器）得到 *ASID* 部分只占据了 *6-11* 共 6 个 bit，故 *ASID* 一共有 *NASID = 64* 个。实验采用了**位图法**管理 *ASID* 的 *asid_bitmap[ ]* 数组，当“**进程被销毁**”或 “***TLB* 被清空**” 时，才可以把其 *ASID* 分配给其他进程。
- `static int asid_alloc(u_int *asid)`
  - 功能：申请一个空闲的的 *asid*
  - 使用例子：

```
1   if (asid_alloc(&(e->env_asid)) == -E_NO_FREE_ENV) {
2           //...
3   } //env.c : int env_alloc(struct Env **new, u_int p
```

设置PCB进程控制块

- `e->env_tf.cp0_status` = R3000 的 *SR* 寄存器
  - 第12位的 *IM4*： 4 号中断可以被响应
  - *SR* 寄存器的低六位是一个二重栈的结构：*KUo* + *IEo* ， *KUp* + *IEp* ，*KUc* + *IEc*
    - *KU* 位表示是否位于用户模式，*IE* 位表示中断是否开启
    - *KUc* 和 *IEc* 表示 CPU 当前实际的运行状态
    - 异常发生时：把 p 复制到 o，把 c 复制到 p，把 c 置位0

      `rfe` 指令时：把 o 复制到 p，把 p 复制到 c **【每个进程在每一次被调度时**都会执行 `rfe` 指令】
- `e->env_tf.regs[29]` = 第 29 号寄存器 = **用户栈**寄存器（非内核栈）
- `int env_alloc(struct Env **new, u_int parent_id)`
  - 功能：申请创建一个进程
  - 使用例子：

```
1   if (env_alloc(&e, 0) == -E_NO_FREE_ENV) {
2                   //...
3   } //env.c : struct Env *env_create(const void *bina
```

  - 实现：

1. 从 *env_free_list* 中申请一个空闲的 PCB（当 `LIST_EMPTY` 不为空时，用 `LIST_FIRST` 得到）
2. 用 `env_setup_vm` 函数为进程初始化页目录，配置地址空间
3. 手动初始化 PCB：

   env_id, *env_asid* , *env_parent_id*, ***env_tf.cp0_status* , *env_tf.regs[29]*** (lab3)

   env_user_tlb_mod_entry (lab4)

   env_runs (lab6)
4. 从 *env_free_list* 中摘出 PCB （使用宏 `LIST_REMOVE` ）

- 虚拟地址 *ULIM* 是 kseg0 与 kuseg 的分界线，是系统给**用户进程**分配的**最高地址**；

  ULIM 以上的地方，kseg0 和 kseg1 两部分内存的访问不经过 TLB，这部分内存由内核管理、所有进程共享；

  在 MOS 操作系统特意将一些内核的数据**暴露**到用户空间，使得进程**不需要切换到内核态就能访问**，在 Lab4 和 Lab6 中将用到此机制。

  - 暴露内容： *UTOP* 往上到 *UVPT* 之间**所有进程共享**的**只读**空间；
  - 暴露方法：

    把 *UTOP* ~ *UVPT* 的内存对应的**内核页表 *base_pgdir* 拷贝到进程页表中。（*UVPT* 往上到 *ULIM* 之间则是进程自己**的页表）
- `static int env_setup_vm(struct Env *e)`
  - 功能：初始化新进程的地址空间（初始化页目录）
  - 使用例子：

    ```
    1   if ( env_setup_vm(e) == -E_NO_MEM) {
    2                   //...
    3   } //env.c : int env_alloc(struct Env **new, u_int p
    ```

  - 实现：

    1. 用 `page_alloc` 申请一个页表，把页表的**内核虚拟地址（强制转化为 (Pde*)类型)**存储到 *env_pgdir* ；
    2. 用 `memcpy` 把 *UTOP* ~ *UVPT* 的内存对应的**内核页表 *base_pgdir* **拷贝到本进程页表中；
    3. 实现自映射，具体见**Thinking 3.1 **。

加载二进制镜像

- 加载一个 ELF 文件到内存 = 将 ELF 文件中**所有需要加载的程序段（program segment）加载到对应的虚拟地址**上
- `const Elf32_Ehdr *elf_from(const void *binary, size_t size)`
  - 功能：解析 ELF 文件头的部分
  - 使用例子：

```
1   const Elf32_Ehdr *ehdr = elf_from(binary, size); //
```

- `int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data)`
  - 功能：遍历将 ELF 文件的一个 segment 加载到内存（回调函数 `load_icode_mapper`）
  - 使用例子：

```
1   panic_on(elf_load_seg(ph, binary + ph->p_offset, lo
2   //env.c : static void load_icode(struct Env *e, cor
```

  - 实现：指导书**P89** + **P90**的segment加载地址布局
    1. 加载该段的所有数据（bin）中的所有内容到内存（va）；
    2. 如果该段在文件中的内容的大小达不到为填入这段内容新分配的页面大小，即分配 了新的页面但没能填满（如 .bss 区域），那么余下的部分用 0 来填充。
- `static void load_icode(struct Env *e, const void *binary, size_t size)`
  - 功能：加载可执行文件 *binary* 到进程 e 的内存中
  - 使用例子：

```
1   load_icode(e, binary, size); //env.c : struct Env *
```

  - 实现：
    1. 用 `elf_from` 函数从 ELF 文件中解析文件头 ehdr；
    2. 遍历每个 segment 的段头 ph，以及其数据在内存中的 起始位置 bin，用 `elf_load_seg` 函数将参数指定的程序段（program segment）加载到进程的地址空间中；

3. 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，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。

- static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src, size_t len);
  - 功能：完成单个 segment 页面的加载过程
  - 使用例子：

```
1   if ((r = map_page(data, va + i, 0, perm, bin + i, M
2                             return r;
3   } //这里的map_page就是回调函数load_icode_mapper
4   //elfloader.c : int elf_load_seg(Elf32_Phdr *ph, cc
```

  - 实现：

    1. 用 page_alloc 函数申请一个物理页面；
    2. 若 *src* 非空，用 memcpy 函数将该处的 ELF 数据拷贝到物理页面中；
    3. 用 page_insert 函数建立映射。

创建进程

- struct Env *env_create(const void *binary, size_t size, int priority)
  - 功能：创建一个新进程
  - 使用例子：【没有好例子，下面这个不太好？】

```
1   env_create(binary_##x##_start, (u_int)binary_##x##_
```

  - 实现：

    1. 用 env_alloc 分配一个新的 Env 结构体;
    2. 设置进程控制块（ e->env.pri 和 e->env_status ）;
    3. 用 load_icode 将程序载入到该进程的地址空间;
    4. 用 TAILQ_INSERT_HEAD 将进程控制块加入到 env_sched_list 调度队列。

进程运行与切换

- 进程上下文

  进程上下文就是**进程执行时所有寄存器的状态**，具体来说就是代码中的**Trapframe**，包括*通用寄存器、HI、 LO 和 CP0 中的 SR，EPC，Cause 和 BadVAddr 寄存器*。

  进程控制块除了 env_tf 其他的字段在 进程切换后还保留在原本的进程控制块中，并不会改变。

  lab3中，寄存器状态保存的地方是 ***KSTACKTOP*** 以下的一个 **`sizeof(TrapFrame)` 大小的区域**中。

- `void env_run(struct Env *e)`
  - 功能：运行一个新进程（**往往意味着是进程切换，而不是单纯的进程运行**）
  - 使用例子：

    ```
    1   env_run(e); //sched.c : void schedule(int yield){}
    ```

  - 实现：
    1. 保存当前进程上下文 (如果当前没有运行的进程就跳过这一步)；
    2. 切换 *curenv* 为即将运行的进程；
    3. 设置**全局变量** *cur_pgdir* 为**当前进程页目录地址**（在 **TLB 重填**时将用到该全局变量）；
    4. . 用 `env_pop_tf` 函数，恢复现场、异常返回。
- `extern void env_pop_tf(struct Trapframe *tf, u_int asid) __attribute__((noreturn))`
  - 功能： `env_pop_tf` 是定义在 `kern/env_asm.S` 中的一个**汇编函数**。该函数呼应 “进程每次被调度运行前一定会**执行 `rfe` 汇编指令**”。
  - 使用例子：

    ```
    1   extern void env_pop_tf(struct Trapframe *tf, u_int
    2   env_pop_tf(&(curenv->env_tf), curenv->env_asid); //
    ```

中断与异常

- 我们实验里认为中断是异常的一种，并且是仅有的一种**异步异常**。

MIPS CPU 处理一个异常时大致要完成 *四项工作*：

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（行使更高级的特权）并禁止中断。
3. 设置 Cause 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，此后一切交给软件处理。

【SR 寄存器 和 Cause 寄存器具体结构件指导书**P94**】

异常的分发

- 异常分发代码 `exc_gen_entry`

```
#entry.S
.section .text.exc_gen_entry
exc_gen_entry:
        SAVE_ALL
        #使用 SAVE_ALL 宏将当前上下文保存到内核的异常栈
        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
        lw              t0, exception_handlers(t0)
        #以得到的异常码作为索引在 exception_handlers 数
        jr              t0
        #5. 跳转到对应的中断处理函数中，从而响应了异常，
```

- 宏 `SAVE_ALL`
  - 功能：将当前的 CPU 现场（上下文）保存到内核的异常栈中

异常向量组 `exception_handlers`

- **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

时钟中断

- 时间片轮调度

  时间片轮转调度是一种**进程调度算法**。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则该进程将挂起，切换到另一个进程运行。

- `kclock_init`
  - 功能：完成时钟中断的初始化
  - 实现：向 `KSEG1 | DEV_RTC_ADDRESS | DEV_RTC_HZ` 位置写入 200

```
1   LEAF(kclock_init)
2           li      t0, 200 // the timer interrupt frec
3           /* Exercise 3.11: Your code here. */
4           sw      t0, (KSEG1 | DEV_RTC_ADDRESS | DEV_
5           jr      ra
6   END(kclock_init)
```

    `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；

    200 表示 1 秒钟中断 200 次；如果写入 0，表示关闭时钟中断。

进程调度

- `e->env_pri` ： 进程每次运行的时间片数量；

  `curenv` **全局变量**：当前正在运行的进程（在第一 个进程被调度前为 *NULL*）；

  `count` **静态变量**：当前正在运行进程的**剩余时间片数量** = 当前进程**剩余的执行次数**；

  `env_sched_list` 调度队列：**存在且只存在**所有就绪（状态为 ENV_RUNNABLE）的进程，其中也需要**包括正在运行**的进程；

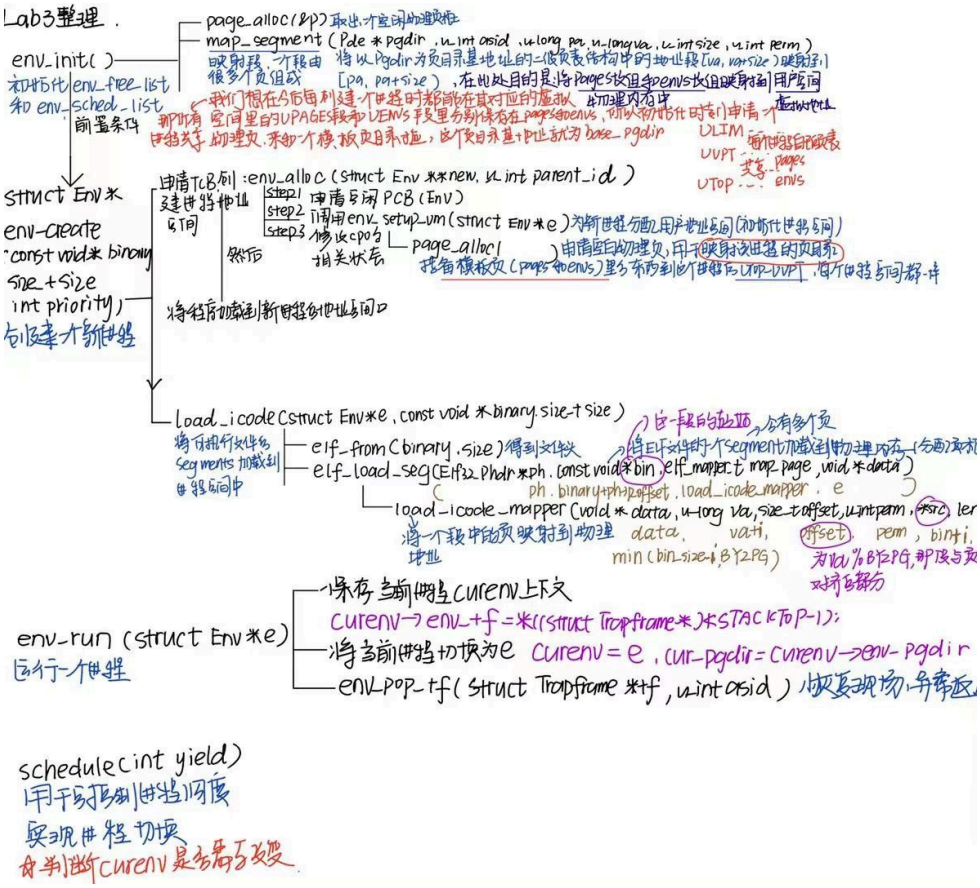
- 当满足下列条件中的**任意一个**时，需要**进程切换**：
  - 尚未调度过任何进程（ `curenv == NULL` ）
  - 当前进程已经用完了时间片（ `count == 0` ）
  - 当前进程不再就绪，如：被阻塞或退出（ `e->env_status != ENV_RUNNABLE` ）
  - yield 参数指定必须发生切换（ `yield != 0` ）
- 无需进程切换时，将剩余时间片数量 count 减去 1，然后调用 env_run 函数；

需要进程切换时，判断当前进程是否仍然就绪，如果是则将其移动到调度链表的尾部。之后，我们选中调度链表首部的进程来调度运行，将剩余时间片数量设置为其优先级。

- 调度函数 schedule 以及其中逐级调用的 env_run、env_pop_tf 和 ret_from_exception 函数都是**不返回**（no return）的函数，被调用后会从内核跳转到被调度进程的用户程序中执行。在 MIPS 中通常使用 `j` 指令而非 `jal` 调用不返回的函数，因为它们不会再返回到其调用者。

Lab3 在 MOS 中的概况

！！指导书**P100~103**！！ *下图引用段宇涵小姐姐的细心整理：*



难点分析

进程创建

- 每个进程都是一个独立的主体，有着自己的地址空间，每一个进程都有一个进程控制块（PCB），系统利用它来控制和管理进程。在我们的实验中，我们用结构体*Env*来作为进程控制块。
- 用和*Page*一样的方式构造链表*env_free_list*,同时再用*TAILQ*宏定义构造另一个调度链表*env_sched_list*。

- *envs*数组是存放进程控制块的物理内存，在系统启动后就已经分配好。首先，通过 env_init 函数进行进程控制块系统的初始化，其中调用 page_alloc 函数为进程的页目录分配一个物理页框，调用 map_segment 函数将处于内核中的*pages*数组和*envs*数组映射到用户地址。以便用户之后访问。之后会调用 env_setup_vm 将用户空间的这部分模板页表复制到每一个进程空间的相应位置。
- 接下来进行进程的创建(env_create 函数)。首先利用 env_alloc 函数来初始化新进程的地址空间，再调用 load_icode 函数将可执行ELF文件加载到进程空间中。

中断异常及进程的调度

详见思考题**Thinking 3.7**

课上测试

lab3-exam

简单的调度方法编写 *kern/sched.c* ，但是我课下的时候有一个bug没有测出来，导致课上测试的时候耽误了很久：

- 错误之处：

  不是在 `if (e && e->env_status == ENV_RUNNABLE)` 外面把调度队列队头元素删去，而应该是在 `if` 判断条件成立的前提下，才删去队头，移到队尾。

- 正确代码：

```
//kern/sched.c
void schedule(int yield) {
        static int count = 0; // remaining time slic
        struct Env *e = curenv;
        count--;
        if ((yield != 0) || (count == 0) || (e == NU
                if (e && e->env_status == ENV_RUNNAB
                        TAILQ_REMOVE(&env_sched_list
                        TAILQ_INSERT_TAIL(&env_sched
                }
        //错误 TAILQ_REMOVE(&env_sched_list, e, env_s
                panic_on(TAILQ_EMPTY(&env_sched_list
                e = TAILQ_FIRST(&env_sched_list);
                count = e->env_pri;
                env_run(e);
        } else {
                env_run(e);
```

```
18          }
19  }
```

lab3-extra

lab3-extra实现的12号 Ov 异常，具体题目见文章Lab3-Extra-Ov题干。

重点在于理解如何由 `tf->cp0_epc` 地址获得对应的异常指令内容。

- 王雨帆小姐姐的答案（稍微改动了一点）：

```
1   void do_ov(struct Trapframe *tf) {
2       curenv->env_ov_cnt++;
3
4       u_long va = tf->cp0_epc; //va是异常指令的虚拟地址
5       Pte *pte;
6       page_lookup(curenv->env_pgdir, va, &pte);//通过查
7       u_long pa = PTE_ADDR(*pte) | (va & 0xfff);//由页表
8       u_long kva = KADDR(pa);
9       //将物理地址转化至"kseg0 区间中对应的虚拟地址"（内核
10
11      int *instr = (int *)kva;//内核虚拟地址可以直接访存
12      int code = (*instr)>>26;
13      int subcode = (*instr)&(0xf);
14
15      if (code == 0) {
16          if (subcode == 0) {
17              printk("add ov handled\n");
18          } else if (subcode == 2) {
19              printk("sub ov handled\n");
20          }
21          (*instr) = (*instr)|(0x1); //把指令换成addu或
22      } else {
23          tf->cp0_epc += 4;
24          printk("addi ov handled\n");
25          int reg_s = ((*instr)>>21) & (0x1f);
26          int reg_t = ((*instr)>>16) & (0x1f);
27          u_int imm = (*instr) & (0xffff);
28          tf->regs[reg_t] = tf->regs[reg_s]/(u_int)2 +
29      }
30      return;
31  }
```

体会与感想

这部分的难点主要在调试，跳板机上多进程的调试让人很头大，也不知道什么时候会切换进程，所以debug是要虚空调试了【善用 `printk` 吧~】。

这部分的Exercise中断异常的地方甚至只需要复制代码就行，难度不大。但是，在进程创建和调度两个地方，还是遇到了很多问题，对PCB的初始化设置究竟需要考虑到哪些参数是没有明说的，以及进程调度函数中的代码也需要大量的思考逻辑。