# 2024 操作系统 Lab3 串讲

2024 操作系统助教组

# Lab3 主要内容

我们重点关注以下三个问题：

- 操作系统启动时为进程管理做了哪些初始化工作？

  （主要由 `env_init` 函数实现）

- 操作系统创建进程时做了什么事情？

  （主要由 `env_alloc` 函数与 `load_icode` 函数实现）

- 进程是如何运行并切换的？

  （主要由 `schedule` 函数与 `env_run` 函数实现）

# 进程管理初始化

## 进程控制块

进程控制块（Process Control Block，PCB）是系统感知进程存在的唯一标志，进程与进程控制块一一对应。在 MOS 中，进程控制块由一个 `Env` 结构体实现。在 Lab3 中需要重点关注 `Env` 结构体中的如下字段：

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    LIST_ENTRY(Env) env_link;   // Free list
    u_int env_id;               // Unique environment identifier
    u_int env_asid;             // ASID
    u_int env_status;           // Status of the environment
    Pde *env_pgdir;             // Kernel virtual address of page dir
    TAILQ_ENTRY(Env) env_sched_link;
};
```

# 进程管理初始化

## 进程控制块数组

在 Lab2 中，我们使用页控制块 `struct Page` 代表物理页面。页控制块数组 `pages` 代表所有的物理页面，以此作为物理内存管理的基本数据结构。

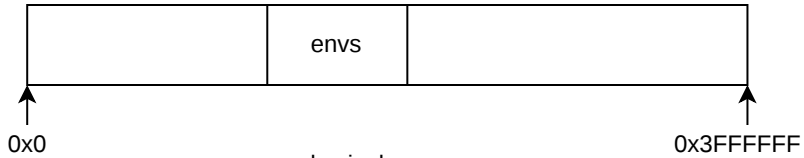
与物理内存管理类似，我们使用进程控制块数组 `struct Env envs[NENV]` 作为进程管理的基本数据结构：

```
// kern/env.c

struct Env envs[NENV] __attribute__((aligned(PAGE_SIZE))); // All environments
```

- `NENV` 宏指定了数组大小，也即 MOS 最多可以管理的进程个数。
- `__attribute__((aligned(PAGE_SIZE)))` 使用了 GCC 扩展语法让 `envs` 的起始地址页对齐到 `PAGE_SIZE`。

在内核加载到内存后，`envs` 数组就是内核数据区中的一片连续的空间：



# 进程管理初始化

env_init

在 Lab3 中，我们在 `mips_init` 函数中添加对 `env_init` 函数的调用，以初始化进程管理相关数据结构。这个函数进行了以下操作：

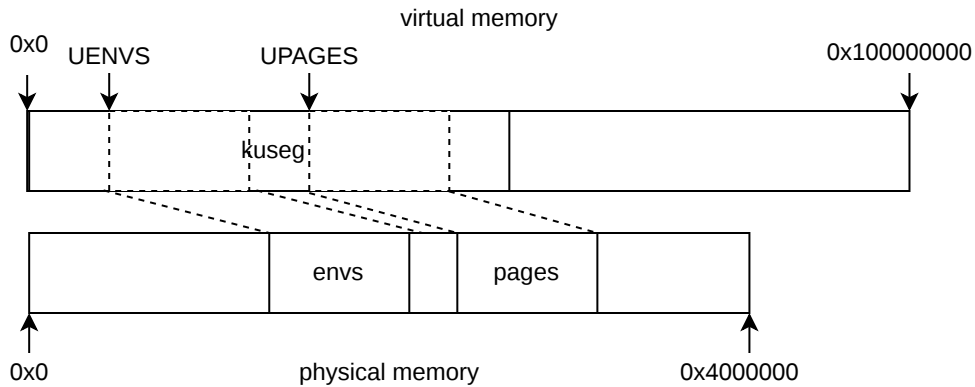
- 首先使用 `LIST_INIT` 宏初始化空闲进程控制块链表 `env_free_list` ，使用 `TAILQ_INIT` 宏初始化进程调度队列 `env_sched_list` 。
- 随后初始化每个进程控制块：将 `envs` 中的每个 `Env` 结构体中的 `env_status` 置为 `ENV_FREE` ，并将它们插入 `env_free_list` 中。
- 最后构造“模板页表” `base_pgdir` 。

与 Lab2 中使用 `page_free_list` 管理空闲物理页类似，我们使用链表 `env_free_list` 管理空闲进程控制块，使得分配空闲进程控制块的操作可以在 O(1) 的时间复杂度内完成，避免每次分配都需要从头遍历 `envs` 数组。

至于最后的“模板页表”，则和我们对用户地址空间的设计有关。

# 模板页表

在用户空间中读取内核信息




我们的 MOS 被设计成对于其中运行的每一个用户进程，都可以通过用户地址空间（kuseg）读取 `pages` 数组和 `envs` 数组的信息。这一功能将在 Lab4 中发挥作用。如图所示，为实现该功能，在创建用户进程时我们需要将 `pages` 数组和 `envs` 数组映射到用户地址空间中的 `UPAGES` 与 `UENVS` 处。

由于对 `pages` 和 `envs` 数组的映射关系在 MOS 的每一个进程的页表中都存在，所以在实现中我们选择预先创建一个页表作为“模板”，建立对 `pages` 和 `envs` 数组的映射关系。当创建新进程时，我们便可以直接以该“模板”为基础创建新的页表。这里的“模板”就是“模板页表”。（但请注意，“模板页表”只是 MOS 中的概念。）

# 模板页表

段地址映射函数

`pages` 数组和 `envs` 数组均是一段地址空间，我们需要将这两段空间映射到用户地址空间中。实现这一功能的函数为 `map_segment`。

```
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size, u_int perm);
```

`map_segment` 函数在以 `pgdir` 为页目录的内核虚拟基地址的页表中加入虚拟地址 `[va, va+size)` 到物理地址 `[pa, pa+size)` 的映射，映射空间长度为 `size` 个字节。并将该段映射的权限置为 `perm | PTE_V | PTE_C_CACHEABLE`。调用该函数时，保证 `va` ，`pa` ，`size` 都是页对齐的。

该函数实际上是对 `page_insert` 函数的封装。`page_insert` 函数将单个物理页面映射到虚拟页面，`map_segment` 则调用 `page_insert` 函数实现将多个连续的物理页面映射到多个连续的虚拟页面。

需要注意一点，`page_insert` 函数需要传入 `Page` 结构体指针代表物理页面，而 `map_segment` 函数的参数 `pa` 是物理页面的起始物理地址，因此需要使用 `pa2page` 函数进行转换。

# 模板页表

模板页表的作用

在 `env_init` 函数的最后，我们申请了一个物理页面作为“模板页表”的页目录，使用 `map_segment` 函数将 `envs` 与 `pages` 数组映射到了 `UENVS` 与 `UPAGES` 处，并让 `base_pgdir` 指向了这个页目录的内核虚拟基地址，以便创建进程时能够根据“模板页表”的内容创建自己的页表。

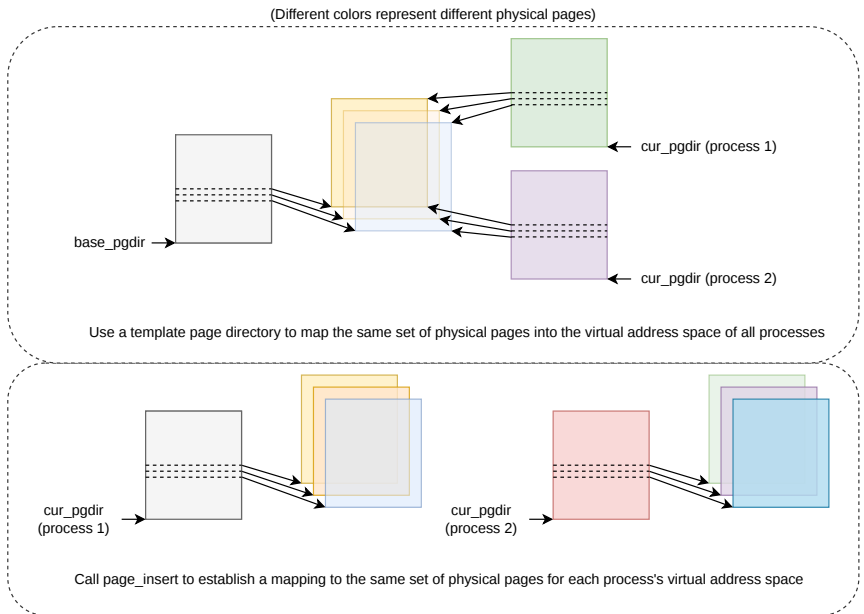
每当创建一个进程，我们都会将这个模板页表页目录中用来映射 `envs` 与 `pages` 的表项**复制**到新创建的进程的页目录中。

（注意进程是资源分配的最小单位，不同的进程各有其页目录和二级页表。）

# 模板页表

## 模板页表的作用

为什么需要建立这样一个模板页表，在创建进程时将模板页表页目录中相关内容复制到新的页表的页目录中，而不是每次创建新的进程时都使用 `map_segment` 函数在新的页表中建立 `envs` 与 `pages` 的映射？因为使用模板页表可以使用户进程共享一部分二级页表，从而节省物理页面。



# 创建进程

创建一个进程需要做什么？

- 建立进程环境（`env_alloc` 函数）
  - 分配一个空闲进程控制块（从 `env_free_list` 的头部获取一个 `Env` 结构体）
  - 建立进程虚拟地址空间（调用 `env_setup_vm` 函数）
  - 给进程分配 ASID（调用 `asid_alloc` 函数）
  - 初始化寄存器取值（`cp0_status` 寄存器与 `$sp` 寄存器）
- 加载二进制镜像（`load_icode` 函数）

# 创建进程

## 建立进程虚拟地址空间

进程是资源分配的最小单位。一个进程对应一个虚拟地址空间，而建立一个虚拟地址空间就是建立一个页表。

又因为空的页表只需要一个页目录，所以为新创建的进程建立虚拟地址空间只需要调用 `page_alloc` 函数分配一页物理页面，并将该页面的内核虚拟基地址赋给进程控制块中的 `env_pgdir` 属性。

此外还有两项工作：

- 需要将“模板页表”中用来映射 `envs` 与 `pages` 的表项复制到新创建的进程的页目录中，使得进程可以在 kuseg 段访问 `envs` 与 `pages`。
- 需要**建立页表自映射**，使得进程可以在 kuseg 段读取它自己的页表。

# 创建进程

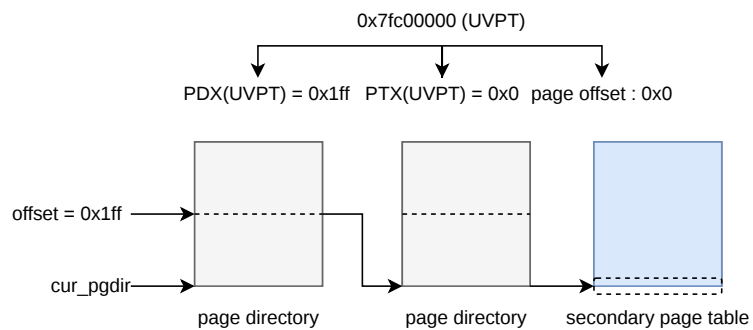
## 页表自映射

重点关注 `env_setup_vm` 函数中用于建立页表自映射的代码：

```
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
```

上面代码意为将进程页目录中虚拟地址 `UVPT` 对应的页目录项置为存放该进程的页目录的物理页面的物理页号，并将权限置为只读。

页表自映射的原理可能较为抽象，我们先举一例子。如果进程运行时访问 `UVPT` 这一虚拟地址，它会读到什么内容？下图展示了进程使用 UVPT 访存时，查询页表的过程：



# 创建进程

进程访问自身页目录

自映射就是将页目录中的某个页目录项的物理页号置为该页目录的物理基地址对应的物理页号。如果我们选择使用将页目录的物理页号设置到不同的页目录项，则页表在进程虚拟地址空间中的位置也不同。

MOS 操作系统中通过自映射将进程页表映射到了 `UVPT` 至 `ULIM` 这一段 `4 MB` 大小的空间。

当我们使用 `UVPT` 至 `ULIM` 之间的地址 `va` 进行访存，由于 `PDX(va)` 和 `PDX(UVPT)` 相同，所以进入到的二级页表即页目录本身。

随后我们通过 `PTX(va)` 计算二级页表索引，并访问页表项所对应的物理页的内容。可是由于此时二级页表即页目录本身，所以我们实际上是访问了页目录项所对应的**二级页表的内容**。

更特殊的，如果 `PTX(va)` 和 `PDX(va)` 取值相同，则此时对应的页目录项所对应的是**页目录本身**，这样我们便实现了对页目录自身内容的访问。

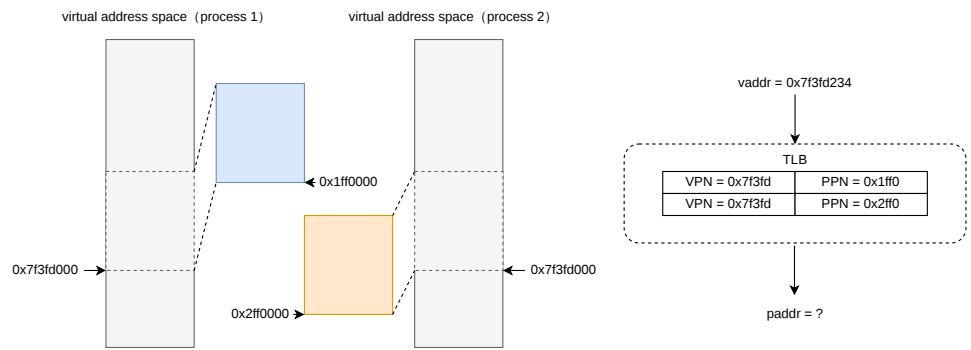
这样我们就可以明白自映射的作用了。它在用户内存空间中划分出一部分，使得用户可以通过访问这部分空间得到二级页表以及页目录中的数据。

# 创建进程

使用 TLB 进行地址转换

在一般情况下，不同进程的虚拟地址空间相互隔离。不同进程的同一个虚拟地址可能映射到不同的物理页面。

TLB 的核心机制是使用程序访存虚拟地址的虚拟页号匹配自己的所有表项中的虚拟页号域，如果有匹配的表项就将对应的物理页号与虚拟地址中的页内偏移组合形成物理地址。但是**假设**我们只使用虚拟地址作为 TLB 中的键，那么此时的“TLB”便无法区分此时的虚拟地址来自哪个进程。如果有来自两个进程的同一个虚拟地址，便只能映射到同一个物理页上，如下图所示。



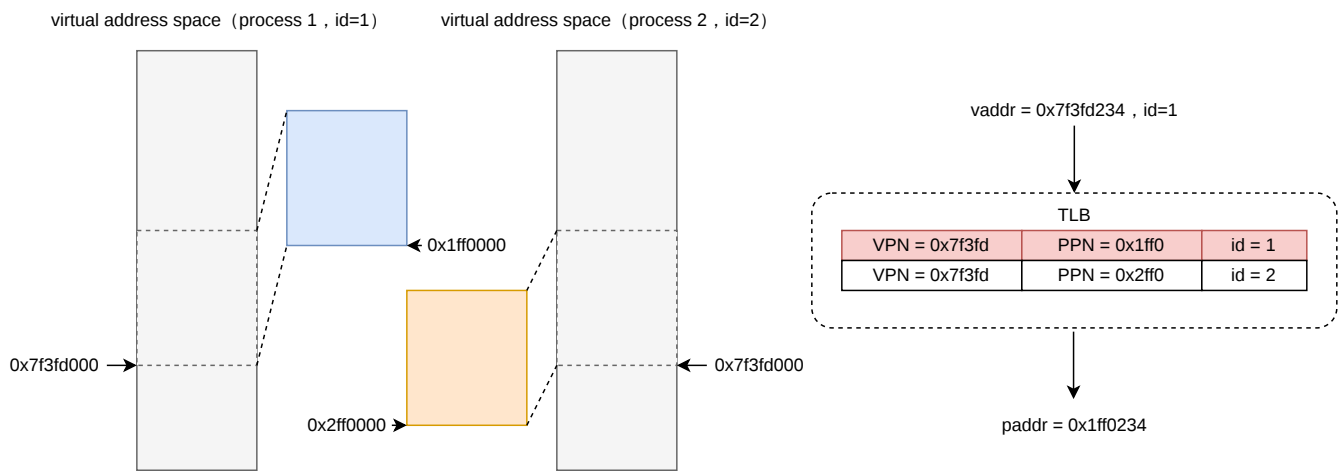

因此，在 TLB 中还需用到另一标识**区分不同的进程**，并将该标识与虚拟地址共同作为 TLB 的键。这一标识也就是 ASID（Address Space IDentifier）。

# 创建进程

## 进程地址空间的隔离

为了实现不同进程虚拟地址空间的隔离，我们在 TLB 中使用 ASID。在 TLB 中，不同的 ASID 便表示不同的虚拟地址空间。又由于进程与地址空间一一对应，所以进程同样与 ASID 一一对应。在查询 TLB 时，我们需要同时给出 ASID 和虚拟页号，只有 **ASID 与虚拟页号均匹配**且**表项有效**，才认为命中。

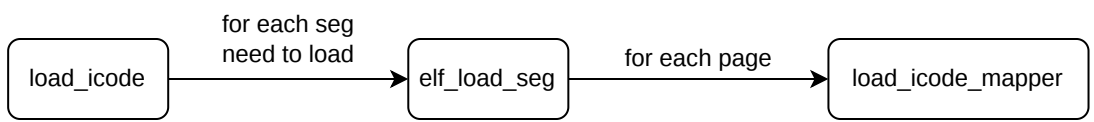
在 MOS 操作系统中，使用 `env.c` 中的 `asid_alloc` 函数给进程分配 ASID。



# 创建进程

创建一个进程需要做什么？

- 建立进程环境（`env_alloc` 函数）
- 加载二进制镜像（`load_icode` 函数）
  - 找到 ELF 文件中的所有要加载的 segment
  - 将单个 segment 加载到内存（`elf_load_seg` 函数）
    - 完成单个页面的加载过程（`load_icode_mapper` 函数）



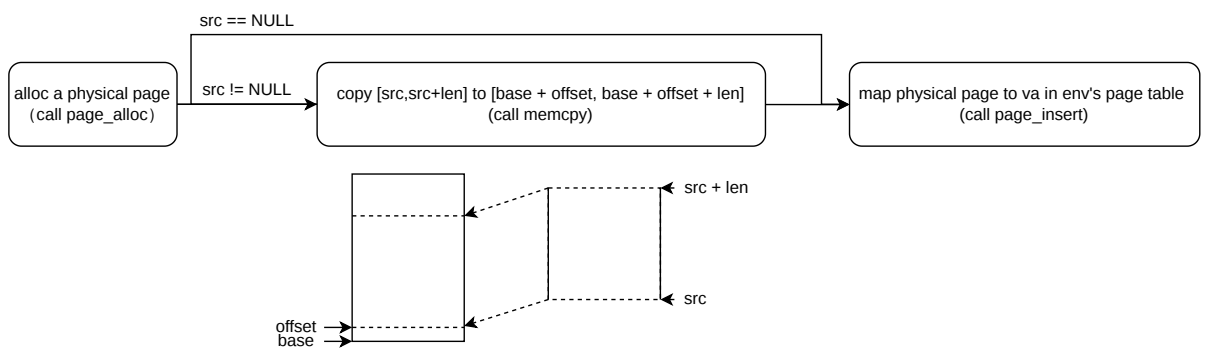
# 创建进程

load_icode_mapper

```
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src,size_t len);
```

该函数用来完成单个页面的加载，主要流程如下（注意 `data` 参数指向进程控制块结构体）：

- 首先分配一页物理页面；
- 如果 `src` 不为 `NULL`，将 `src` 开始的 `len` 字节长度的一段数据复制到新分配的物理页面中 `offset` 开始的空间；
- 最后将新分配的物理页面映射到进程虚拟地址空间中 `va` 处，并将权限设为 `perm`。



# 创建进程

## 设置寄存器取值

在进程中的程序开始执行它的第一条指令前，操作系统需要设置一些寄存器取值，包括：

- 确定进程中程序的开始执行位置：需要设置 `pc` 寄存器的初始值

  程序的 ELF 头中 `e_entry` 字段设定了程序的第一条指令的虚拟地址。在 `load_icode` 函数的最后，需要将 `e->env_tf.cp0_epc` 的值设置为 `ehdr->e_entry`。

  （为什么不是 `pc` 而是 `epc`？因为进程通过时钟中断进行切换，`epc` 是异常处理完成后的返回地址。）

- 不能假定加载的程序会自己设定栈指针：需要设置 `sp` 寄存器的初始值

  在 `env_alloc` 函数中，`e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **)`

- 在运行程序时，需要在用户模式下并且开启中断：需要设置 `cp0` 中 `status` 寄存器的值

  在 `env_alloc` 函数中，`e->env_tf.cp0_status = STATUS_IM7 | STATUS_IE | STATUS_EXL | STATUS_UM`

# 创建进程

env_create

`env_create` 函数实现了进程的创建：指定程序的二进制镜像 `binary` 以及优先级 `priority` ，创建运行了该程序的进程。

```
struct Env *env_create(const void *binary, size_t size, int priority);
```

该函数的具体流程如下：

- 调用 `env_alloc` 函数分配一个空闲进程控制块。
- 将进程控制块中 `env_status` 属性设置为 `ENV_RUNNABLE` ，并将 `env_pri` 域设置为 `priority` 。
- 调用 `load_icode` 函数将程序镜像加载到新创建进程的虚拟地址空间的特定位置。
- 使用 `TAILQ_INSERT_HEAD` 宏将进程控制块插入进程调度链表 `env_sched_list` 中。

**思考一下**： `binary` 参数所指向的数据从何而来？

# 创建进程

## binary 的来源

`binary` 参数指向了程序的二进制镜像数据，其来源可能有二：其一是通过访问文件系统从磁盘中加载，但是这种方法只有当我们在 Lab5 实现文件系统之后才可使用，因此这里不做介绍。而另一种方法就是在编译时硬编码到内核中，这也是本次实验中 `binary` 的唯一来源。

`ENV_CREATE` 宏中使用到了 `env_create` 函数。此时传入 `binary` 的为一外部数组的地址。请注意数组的特殊名称 `binary_##x##_start` 用到了宏拼接的特性。举例来说，对于 `ENV_CREATE(program)` ，进行宏替换后将产生数组名 `binary_program_start` 。

```
#define ENV_CREATE(x)                                                   \
    ({                                                                  \
        extern u_char binary_##x##_start[];                             \
        extern u_int binary_##x##_size;                                 \
        env_create(binary_##x##_start, (u_int)binary_##x##_size, 1);    \
    })
```

# 创建进程

## binary 的来源

外部数组的定义各位同学可能难以找到，因为该函数实际是由工具程序 `tools/bintoc.c` 自动生成的。

```
// tools/bintoc.c
size_t n = fread(binary, sizeof(char), size, bin);
assert(n == size);
fprintf(out,
  "unsigned int binary_%s_%s_size = %d;\n"
  "unsigned char binary_%s_%s_start[] = {",
  prefix, bin_file, size, prefix, bin_file);
for (i = 0; i < size; i++) {
  fprintf(out, "0x%x%c", binary[i], i < size - 1 ? ',' : '}');
}
```

在编译过程中，`bintoc` 工具会读取 ELF 文件，将其转化成一个包含有 `binary_##x##_start` 数组的 `.c` 文件。该文件会编译并链接到内核中，因此 `env_create` 函数才得以访问用户程序的二进制镜像数据。

# 进程切换

## 时钟中断

MOS 是分时操作系统，采用时间片轮转调度算法。当某一进程执行完分配给它的时间片后，操作系统便选择另一个进程执行下一个时间片。

这一过程并不能由进程中运行的程序自行完成，因为程序并不能确定执行到何处需要进行进程切换。所以我们需要使用外部的时间源记录已运行的时间，并在时间片用完时**中断**正在运行的程序。

4KC 中的 CP0 内置了一个可产生中断的 Timer，MOS 即使用这个内置的 Timer 产生用于时间片轮转算法的时钟中断。

CP0 中存在两个用于控制此内置 Timer 的寄存器，即 Count 寄存器与 Compare 寄存器。其中，Count 寄存器会按照某种仅与处理器流水线频率相关的频率不断自增，而 Compare 寄存器维持不变。当 Count 寄存器的值与 Compare 寄存器的值相等且非 0 时，时钟中断会被立即触发。使用 `RESET_KCLOCK` 宏将 Count 寄存器清零并将 Compare 寄存器配置为我们所期望的计时器周期数，这就初始化了时钟中断。

显然，上述初始化时钟中断的过程在每个时间片开始前都需要进行一次，对 `RESET_KCLOCK` 宏的调用在 `env_pop_tf` 函数中，该函数被 `env_run` 函数调用。（`env_run` 用于进程的切换。）

# 进程切换

## 中断的产生

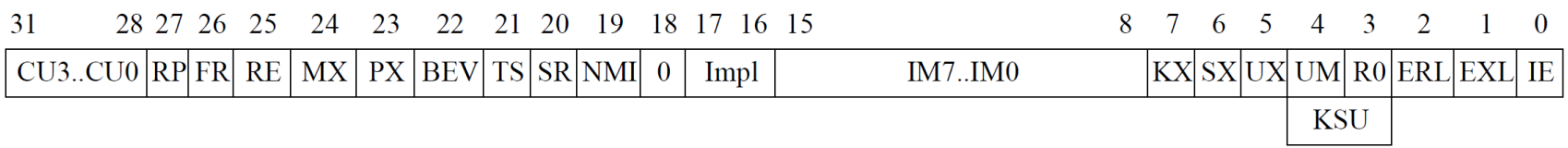
当调用 `env_run` 函数切换到某一进程并执行该进程时，时钟也在同步记录当前进程本次已运行的时间。最终在某一时刻，进程运行了一个时间片的时间，此时时钟中断产生。

中断是一种**异常**。时钟中断首先作为**异常**被 CPU 所处理。

CPU 如何处理异常？

- 设置 EPC 寄存器的值为从异常返回的地址。
- 设置 Status 寄存器（设置 EXL 位，强制 CPU 进入内核态并禁止中断）。

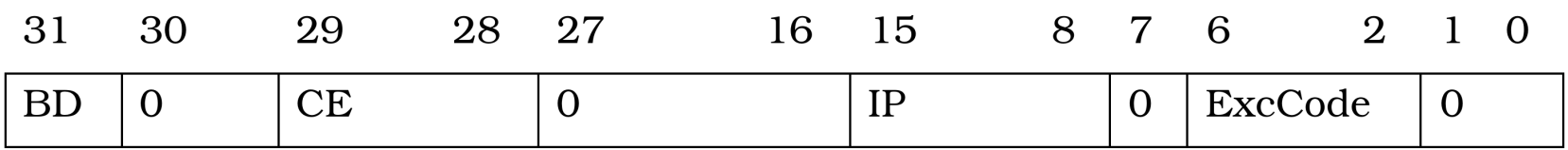
**Figure 6-12 Status Register Format**



# 进程切换

## 中断的产生

- 设置 Cause 寄存器（记录异常原因）：
  - BD（Branch Delay）：置位则表示 EPC 指向了发生异常的指令的前一条。
  - IP：记录了**中断**的信息，当 Status 寄存器中相同的位（IM，Interrupt Mask）为 1（即允许中断）时，IP 位的活动会导致中断（前提是开启了全局中断）。
  - ExcCode：异常号。在 MOS 中，**中断**是 0 号**异常**。




- 设置 PC 为**异常入口地址**，随后交给软件处理。

至此，我们成功地切换到内核程序，将异常处理的任务转交给**操作系统**。

# 进程切换

## 异常入口地址

在 MIPS 4Kc CPU 中，通用异常入口被规定在 `0x80000180`。但有一个例外：用户态的 TLB Miss 异常入口被规定在 `0x80000000`。中断是一种异常，因此其入口地址是 `0x80000180`。

在 `kern/entry.S` 中，我们将通用异常入口和 TLB Miss 异常入口分别用 `exc_gen_entry` 和 `tlb_miss_entry` 两个标签表示。通过 MIPS 伪指令 `.section`，这两个标签被放置在 `.text.exc_gen_entry` 和 `.text.tlb_miss_entry` 两个节中。

```
.section .text.tlb_miss_entry
tlb_miss_entry:
...

.section .text.exc_gen_entry
exc_gen_entry:
...
```

如何将两个标签设定到指定的入口地址？使用链接器脚本 `kernel.lds`。

# 进程切换

从入口开始执行

对于中断这种异常，操作系统从 `0x80000180` ，即 `exc_gen_entry` 开始执行。

```
exc_gen_entry:
    SAVE_ALL
    ...
```

`SAVE_ALL` 宏用于将所有的寄存器值存储到栈帧中。从而保存了异常发生时的**上下文**。

```
.macro SAVE_ALL
.set noat
.set noreorder
    mfc0    k0, CP0_STATUS
    andi    k0, STATUS_UM
    beqz    k0, 1f
    move    k0, sp
    li      sp, KSTACKTOP
1:
    subu    sp, sp, TF_SIZE
    ...
```

# 进程切换

## SAVE_ALL 与异常重入

如果**在进行异常处理的过程中，又发生了新的异常**，这时操作系统是否还能够进行正确的处理？

答案是肯定的。异常处理时，我们的上下文被保存在内核栈中，那么我们只需要在处理新的异常时不破坏内核栈中原本保存的数据即可。因此在 `SAVE_ALL` 中我们进行判断，若 Status 寄存器的 UM 位为 0，说明此次异常在内核态触发，`sp` 寄存器已经在内核异常栈中。不再将 `sp` 设置为 `KSTACKTOP`，而是使其继续增长。这样我们便能够在异常中处理新的异常，而不会破坏原本的异常处理流程。这一机制被称作**异常重入**。

```
.macro SAVE_ALL
.set noat
.set noreorder
    mfc0    k0, CP0_STATUS
    andi    k0, STATUS_UM
    beqz    k0, 1f
    move    k0, sp
    li      sp, KSTACKTOP
1:
    subu    sp, sp, TF_SIZE
    ...
```

# 进程切换

## 异常的分发

在 `SAVE_ALL` 之后，接下来的内容实现了异常的分发（对不同异常选择特定的异常处理函数）。

```
exc_gen_entry:
    SAVE_ALL
    mfc0    t0, CP0_STATUS
    and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
    mtc0    t0, CP0_STATUS

    mfc0    t0, CP0_CAUSE
    andi    t0, 0x7c
    lw      t0, exception_handlers(t0)
    jr      t0
```

- `0x7c` 取与获得 Cause 寄存器的第 2-6 位。这部分对应异常码，用于区别不同的异常。
- `exception_handlers` 数组用于定义异常对应的处理函数，实现**异常的分发**。
- `jr` 跳转到处理函数所在地址。

# 进程切换

进入中断处理函数

对于时钟中断，经过异常分发后我们现在进入了中断处理函数 `handle_int` 中。

此前，我们以**异常**的角度对时钟中断进行处理；现在，我们以**中断**的角度对时钟中断进行处理：

- 首先，我们需要确定产生了何种中断。这通过 Cause 寄存器中的中断位判断。

| 31 | 30 | 29 | 28 | 27 | 16 | 15 | 8 | 7 | 6 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BD | 0 | CE | | 0 | | IP | | 0 | ExcCode | | 0 | |

- 判断为时钟中断，跳转到 `timer_irq` 分支，进行**时钟中断**的处理。
- 最终跳转到 `schedule` 进程调度函数。（注意这里使用了 `j` 而不是 `jal`，因为 `schedule` 函数并不返回。）

# 进程切换

进程调度器

- 什么时候需要**切换**进程？

  1. 参数 `yield` 为真时：此时当前进程必须让出。
  2. `count` 减为 0 时：此时分给进程的时间片被用完，将执行权让给其他进程。
  3. 无当前进程：内核必然刚刚完成初始化，需要分配一个进程执行。
  4. 进程状态不是可运行：当前进程不能再继续执行，让给其他进程。

- 如何切换进程？

  1. 当前进程仍为就绪状态时，需要将其移到 `env_sched_list` 队列的尾部。
  2. 选中 `env_sched_list` 队列头部的进程。如果没有可用的进程，内核 panic。
  3. 设置 count 为当前进程的优先级（分配的时间片的数量）。

- 最后将 count 自减 1，调用 `env_run` 函数。

# 进程切换

进程切换的实现

进程调度过程依旧处于内核态，需要完成异常处理，返回用户态。`env_run` 实现了这一过程：

- 将**旧进程**的寄存器信息保存在**旧进程**的 `env_tf` 中。

  回忆 `SAVE_ALL` 的实现，旧进程的寄存器信息保存在内核异常栈顶部。

- 将 `curenv` 改为**新进程**的控制块指针。

- 自增 `env_runs` （在 Lab6 中用到）。

- 将 `cur_pgdir` 设置成 `curenv->env_pgdir` 。

  全局变量 `cur_pgdir` 表示当前进程的页目录地址，在 Lab2 中有使用。但当时此变量并没有被赋值。这也是在 Lab2 中页式内存管理无法使用的一个原因。

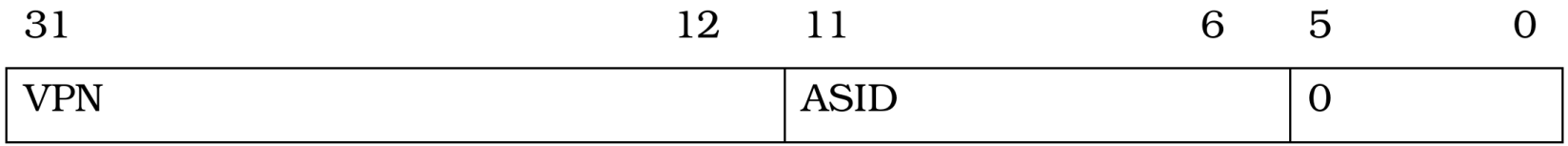
- 调用 `env_pop_tf` ，设置新进程的上下文并运行新进程。

# 进程切换

## 运行新进程

`env_pop_tf` 位于 `kern/env_asm.S`。用于设置 ASID 和重置时钟，以及最后从异常处理中返回。

- 首先，我们需要设置 `EntryHi` 寄存器的 ASID 部分。



EntryHi Register (TLB key fields)

- 随后，我们将 sp 寄存器的值设置为 `env_tf` 的**地址**，以便 `ret_from_exception` 利用。
- 之后，使用 `RESET_KCLOCK` 重置时钟。
- 最后，调用 `ret_from_exception`，**从异常返回**。

# 进程切换

## 从异常返回

`ret_from_exception` 函数位于 `kern/genex.S`。用于恢复进程的上下文并按照该上下文继续进程的运行。因为调用该函数将从异常处理中回到正常的程序执行流程，所以该函数是无返回的。

- 调用 `RESTORE_ALL` 宏进行寄存器的恢复。

  `RESTORE_ALL` 宏将 sp 之上的一个 `Trapframe` 恢复到寄存器中。

- 执行 `eret` 指令从异常处理中返回。

# 其他异常的介绍

Lab3 中主要用到了**时钟中断**这种异常。在最后我们稍微介绍一下 MOS 中实现的其他异常。

异常处理函数的声明可见于 `kern/genex.S`。在声明时用当了 `BUILD_HANDLER` 宏。

- `handle_reserved`：异常默认处理
  - 异常处理函数是 `do_reserved`。
  - 功能是输出 Trapframe 的相关信息，随后内核 panic。
- `handle_tlb`：处理 TLB 读写异常。
  - 异常处理函数是 `do_tlb_refill`。
  - 是 Lab2 中 TLB Miss 的处理函数。

# 课下习题提示

- Exercise 3.1

  注意进程控制块插入空闲链表的顺序

- Exercise 3.2

  1. `map_segment` 是对 `page_insert` 函数的封装
  2. 遍历要映射的物理和虚拟页基地址，跨度为一页的大小（`PAGE_SIZE`）

- Exercise 3.3

  1. 申请一个物理页并将其**内核虚拟地址**赋给 `env_pgdir`（使用 `page2kva`）
  2. 不要忘记更新引用计数 `pp_ref`

- Exercise 3.4

  需要赋值的进程控制块字段：`env_asid`、`env_id` 和 `env_parent_id`

# 课下习题提示

- Exercise 3.5

  `memcpy` 的目标地址是页的内核虚拟地址加上偏移量（使用 `page2kva`）

- Exercise 3.6

  将 `ehdr->e_entry` 的值赋给 `e->env_tf.cp0_epc`

- Exercise 3.7

  1. 不要忘记将进程状态设置为 `ENV_RUNNABLE`
  2. 使用 `load_icode` 将二进制程序镜像 `binary` 加载到进程的虚拟地址空间中

- Exercise 3.8

  1. 将 `curenv->env_pgdir` 的值赋给 `cur_pgdir`
  2. 传入 `env_pop_tf` 的是 `curenv` 的 Trap Frame 地址

# 课下习题提示

- Exercise 3.9

  1. 使用 `mfc0` 指令获取 `CP0_CAUSE` 寄存器的取值
  2. 使用 `addi` 提取 Cause 寄存器的第 2-6 位
  3. 使用 `lw` 获取 `exception_handlers` 数组中对应位置的函数地址
  4. 使用 `jr` 跳转到该地址

- Exercise 3.10

  回忆 Lab1 中对 `.text`、`.data` 和 `.bss` 段地址位置的确定方法，用同一方法确定 `.tlb_miss_entry` 和 `.exc_gen_entry` 的位置

- Exercise 3.11

  1. 使用 `mtc0` 将 `CP0_COUNT` 寄存器置零
  2. 将 `CP0_COMPARE` 寄存器的取值设为 `TIMER_INTERVAL`

# 课下习题提示

- Exercise 3.12

  位于调度队列 `env_sched_list` 首部的进程控制块对应的进程是当前运行的进程，**不要将其从调度队列中移出**