实验目的

- 创建一个进程并成功运行
- 实现时钟中断，通过时钟中断内核可以再次获得执行权
- 实现进程调度，创建两个进程，并且通过时钟中断切换进程执行

具体而言，就是运行一个**用户模式的进程。**需要使用数据结构进程控制块**Env**来跟踪用户进程，并建立一个简单的用户进程，**加载一个程序镜像**到指定的内存空间，然后让它运行起来。同时，MIPS内核拥有**处理异常**的能力。

实验难点

易漏变量

本次实验的数据结构核心毫无疑问是系统专门设置用来管理进程的进程控制块（PCB），该结构体的信息已经表述得十分清楚，不过在实验中我偶尔会忘记几个重要的域，导致实验过程中出现一些小困扰：

- `env_pgdir`：保存了该进程页目录的内核虚拟地址
- `env_cr3`：　保存了该进程页目录的物理地址
- `struct Trapframe env_tf`：保存了一些寄存器。这是我最常忘记的域，毕竟它的使用还关联到另外一个陷阱帧结构体，以下代码中标明了需要特别注意又时常忘记的一些寄存器。
    - 陷阱帧是指中断、自陷、异常进入内核后，在堆栈上形成的一种数据结构，它存储在出现异常时保存的寄存器集合，因此使用陷阱帧可以返回并继续执行指令（在处理异常或 irq 时），具体应用放到 *中断和异常* 一节中分析。

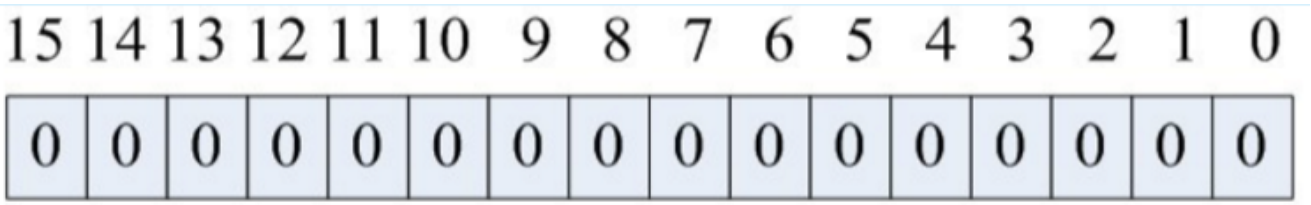
```
struct Trapframe {
    /* 保存主要的处理器寄存器，特别注意regs[29] */
    unsigned long regs[32];

    /* 保存特殊寄存器 */
    unsigned long cp0_status;   /* Attention */
    unsigned long hi;
    unsigned long lo;
    unsigned long cp0_badvaddr;
    unsigned long cp0_cause;    /* Attention */
    unsigned long cp0_epc;      /* Attention */
    unsigned long pc;
};
```

位图法的运用：分配ASID

虽然这部分没有需要我们编写的代码，但是我认为用位图管理大数据存在性的思想还是值得学习。上一次见到这个名词是在内存管理单元，本单元的代码则让我对它的认识更加清晰。

位图（bitmap）法可以解决海量数据的存在性问题，又不占用很多内存。所谓bitmap，就是用每一个比特位来存放某种状态，适用于大规模数据，但数据状态又不是很多的情况。通常是用来判断某个数据存不存在的。其数据结构为 `unsigned int bit[N];`，在这个数组里面，可以存储 `N * sizeof(int) * 8` 个数据（因为 1 个 int 是 4 Bytes），但是最大的数只能是 `N * sizeof(int) * 8 - 1`。假如，我们要存储的数据范围为 0 - 15，则我们只需要使得 N = 1，这样就可以把数据存进去。



体现在本实验中，则是对 `asid_bitmap` 数组的处理。由上述原理我们知道这个 `asid_bitmap` 只要分 2 个 int 就可以满足 64 个 ASID 的判断了。

当然，难点当然不在概念的理解，而是在处理位图数据的位运算上（对位运算掌握不甚了了的笔者而言）。

```
for (int i = 0; i < 64; i++) {
    index = i >> 5;
    inner = i & 31;
    if ((asid_bitmap[index] & (1 << inner)) == 0) {
        asid_bitmap[index] |= 1 << inner;
        return i;
    }
}
```

这里是为了找到一个未被分配的 ASID 分配给新创建的进程。首先需要通过 `index` 定位当前数在位图数组的哪一个元素，然后利用 `inner` 定位当前数处于哪一个比特位中。如果后续有需要修改位图的具体应用，也需要掌握这种计算方法。

地址空间初始化：mmu.h的理解复习

说到底，这部分内容掌握不精还是因为没有读透 `include/mmu.h` 的代码，对地址空间结构的理解不够，直到面对实验中一大串意味不明的宏时，才会边痛苦边悔恨欠下的债始终是要还的。因此在这里做一些简单的梳理。

```
/*
 o     4G ----------->  +----------------------------+------------0x100000000
 o                      |       ...                  |  kseg3
 o                      +----------------------------+------------0xe000 0000
 o                      |       ...                  |  kseg2
 o                      +----------------------------+------------0xc000 0000
 o                      |   Interrupts & Exception   |  kseg1
 o                      +----------------------------+------------0xa000 0000
 o                      |      Invalid memory        |   /|\
 o                      +----------------------------+----|-------Physics Memory Max
 o                      |       ...                  |  kseg0
 o  VPT,KSTACKTOP-----> +----------------------------+----|-------0x8040 0000-------end
 o                      |       Kernel Stack         |    | KSTKSIZE            /|\
 o                      +----------------------------+----|------                |
 o                      |       Kernel Text          |    |                    PDMAP
 o      KERNBASE -----> +----------------------------+----|-------0x8001 0000    |
 o                      |   Interrupts & Exception   |   \|/                    \|/
 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   |     BY2PG                |
 o                      +----------------------------+------------0x7f3f f000    |
 o                      |       Invalid memory       |     BY2PG                |
 o      USTACKTOP ----> +----------------------------+------------0x7f3f e000    |
 o                      |     normal user stack      |     BY2PG                |
 o                      +----------------------------+------------0x7f3f d000    |
 a                      |                            |                          |
 a                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                           |
 a                      .                            .                          |
 a                      .                            .                        kuseg
 a                      .                            .                          |
 a                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~|                          |
 a                      |                            |                          |
 o       UTEXT   -----> +----------------------------+                          |
 o                      |                            |     2 * PDMAP             \|/
 a     0 ------------>  +----------------------------+ ----------------------------
 o
*/
```

其实理解了缩写之后内涵应该都比较清楚：

- USTACKTOP - User STACK TOP

  UXSTACKTOP - User eXception STACK TOP (这两个我在lab4都还经常记混)
- UTOP - User TOP：不是用户地址空间的顶端，是用户进程能自由读写区域的顶端

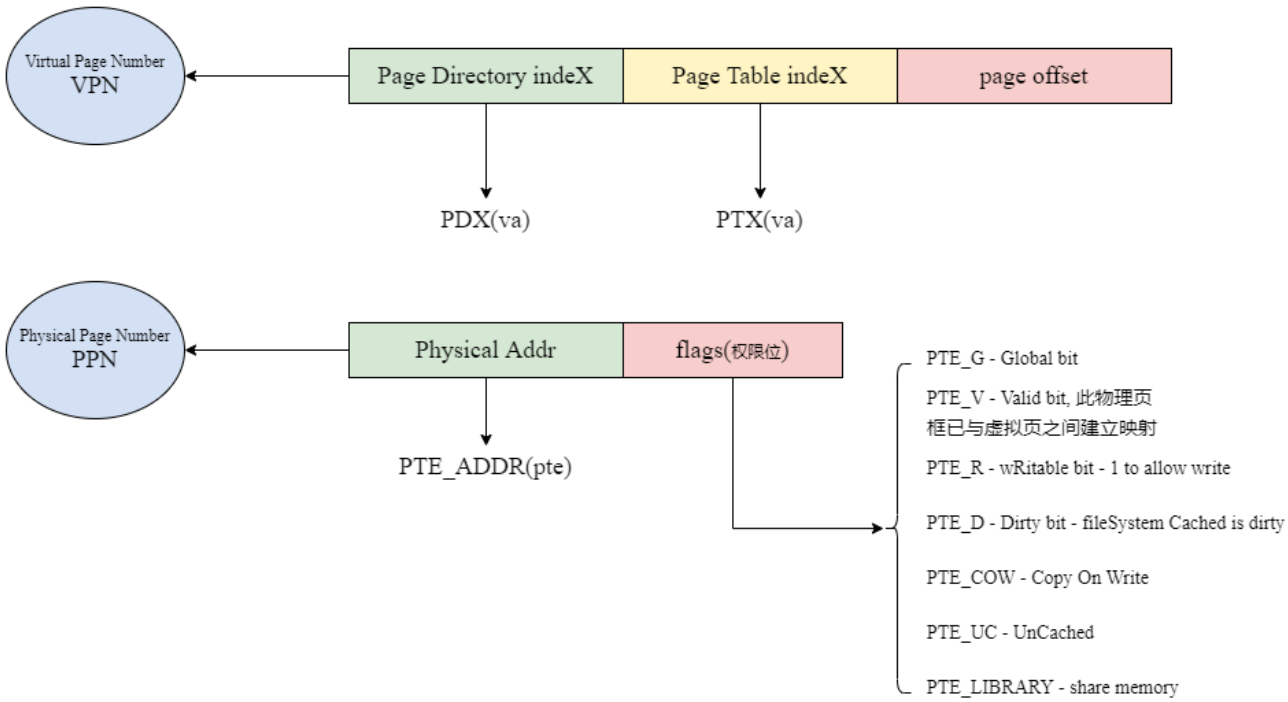
UENVS - User ENVironmentS：一个用来映射分配给进程管理使用的Env结构体的空间的内核页表的起始虚拟地址，也就是说，envs数组映射在这一段区域里

- UPAGES - User PAGES：用来存放用户的页表信息的空间的起始虚拟地址
- UVPT - User Virtual Page Table：用户页表的自映射项
- ULIM - User LIMit：用户地址空间的上界，kseg0和kuseg的分界线，UTOP到ULIM的区域只可以读不可以写，存放用户的进程信息与页表信息

```
#define ULIM  0x80000000
#define PDMAP (4*1024*1024)     // bytes mapped by a page directory entry
```

- 再往上的空间就只是将 用户 替换为 **内核（Kernel）**而已了，不再赘述

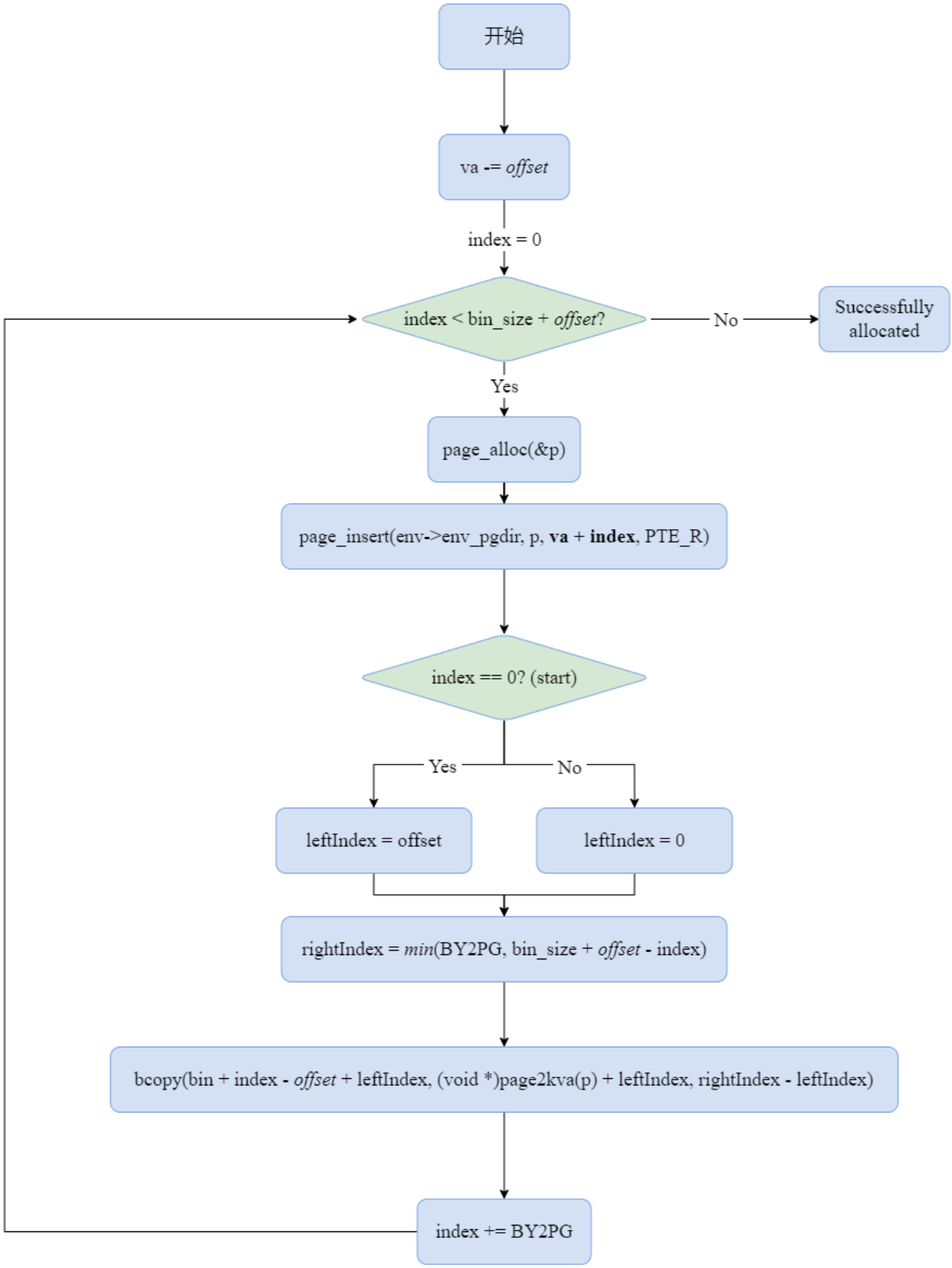
当然，搞明白了这个，还有一些常用的宏需要记住，总结如下图所示。



页面对齐

> 这个函数是至关重要的，它很可能成为未来大量 TOO LOW 的元凶。因此建议你静下心来，认真考虑对齐的问题，保证在应有位置完成 ELF 加载后既不会破坏原有页面也不会引入新的多余内容。

此处直接使用**最糟糕**的情况进行分析，我对对齐问题的处理流程如下图所示：

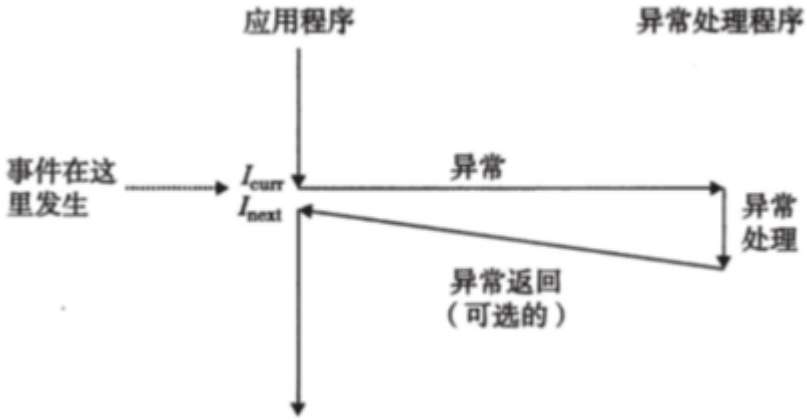


也就是说，首先清理开头，然后判断 bin_size 开头，如果页对齐则可以直接写（当然此处均按最坏情况算，都算不对齐），若不对齐则判断页面是否已经被申请，若没有则申请；对于 bin_size 结尾，不多写一个字节，尽量避免造成覆盖，这里用一对首尾 index 来指示位置。对于 sg_size 同理。

异常处理函数

本章第二部分的主体是（ OS 管理的那部分）中断与异常。上一次见到这个词还是在计组，因此花费了一些时间来理解这个部分。

对于 OS 掌握的部分，核心应当是异常分发程序与异常处理函数，也就是软件化的一些过程。当发生异常时，处理器会进入一个用于分发异常的异常分发程序，它将检测发生了哪种异常，并调用相应的异常处理函数，一般来说它被放在一个固定的物理地址上，以保证处理器在检测到异常时能够正确地跳转到那里。



而对于异常处理函数，处理的过程其实也是大同小异的（从观察汇编代码可得）。首先保护现场，然后屏蔽中断，这些过程通过设置 SR 寄存器完成；接下来的过程则因异常原因而异了。调用完异常处理函数后，恢复上下文，最后返回即完成了整个异常处理过程。主要是要知道各种过程是在什么地方实现的，调用了哪些函数，修改了哪些寄存器。

思考题

Thinking 3.1

首先应该搞清楚 `envid2env` 究竟是干什么的，才能明白它为什么限制了不活动的进程和进程 id 不对应的进程不能被获取，或者说，为什么进程 id 不能被对应。

它的用处是通过一个 env 的 id 获取该 id 对应的进程控制块，把找到的进程控制块返回到 `penv` 参数中去，至于找的过程则是简单粗暴地利用 id 为**数组索引**，到进程控制块列表 envs 中直接获取对应值。也就是说，此 id 索引对应的数组元素不对应真实 env_id 是完全有可能的，比如说数组元素被替换等等，因为简单的数组里并没有建立一个一一对应的关系。因此，如果不对 `e->env_id != envid` 的特殊情况做出判断，就有可能会导致找到的进程块是错误的——毕竟这个数组索引并不对 env_id 的一一对应性负责到底。

Thinking 3.2

实验难点中刚好总结了此题内容。

- UTOP 不是用户地址空间的顶端，是用户进程能自由读写区域的顶端；ULIM 是用户地址空间的上界，kseg0 和 kuseg 的分界线；UTOP 到 ULIM 的区域只可以读不可以写，存放用户的进程信息与页表信息，而 UTOP 以下的区域可供用户进程自由读写。
- 我们知道 UVPT 是用户页表的自映射项，而 env_cr3 则保存了进程页目录的物理地址，因此通过 `pgdir[PDX(UVPT)] = env_cr3` 可以将页目录子映射机制建立起来，使得页目录的第 `PDX(UVPT)` 项映射到进程页目录自身。
- 显然我们在进程中使用的是虚拟地址，通过查询进程页表可以将它们映射到相应的物理地址上。在我们的实验中，不同进程对于内核的 2G 地址空间是相同的，对于用户的 2G 有各自**独立**的地址空间，不同的进程使用同样

的虚拟地址也可以访问到不同的物理地址，不会发生冲突。

Thinking 3.3

在 `lib/kernel_elfloader.c` 中可以查看到 `load_elf` 函数的定义以及其调用的 `user_data` 函数参数。需要这个函数参数的根源在于外层函数需要向内层函数传值，本实验中用于传递进程控制块，因此不可以缺少。

案例：`qsort()` 函数，相当于 `qsort(user_data,int *map(user_data))`

Thinking 3.4

其实就是头尾对齐或不对齐的几种排列组合情况，最坏的情况是头尾都不对齐。书写代码时按照最坏情况考虑就不会遗漏场景（因为对齐的时候就相当于不对齐的 *offset* = 0，特判即可）。

Thinking 3.5

- 存储虚拟地址，也即程序运行地址
- entry_point 对于每个进程应该是一样的，因为它们都从 elf 文件的入口复制而来。这样可以使得进程每次执行都从一个固定的虚拟地址开始，对CPU是友好的，此时由于虚拟地址相同但是进程PCB不同，可以让不同进程的地址空间映射到各自不同的物理地址——这种统一本质上反映了不同进程所见的内存空间的同一性。

Thinking 3.6

epc 是指程序完成异常处理后返回的地址。因为此时对本进程而言，切换到其他进程的过程相当于一个异常，需要将 pc 置为异常结束之后应当跳转到的地址，才能保证其正确执行。

Thinking 3.7

- TIMESTACK 是时钟中断时存放 CPU 寄存器状态的栈的栈顶地址，用于保存现场a恢复时把这个区域的值写回寄存器，因此 OS 是在发生时钟中断时将内核寄存器的值写入到 TIMESTACK 区域中。
- TIMESTACK 是发生时钟中断时固定的栈顶地址，KERNEL_SP是发生其他中断时存放寄存器状态的栈的栈顶地址，并且前者写入的是内核寄存器的值，后者则是用户寄存器。

```
.macro get_sp
    mfc0    k1, CP0_CAUSE
    andi    k1, 0x107C
    xori    k1, 0x1000
    bnez    k1, 1f
    nop
    li  sp, 0x82000000
    j   2f
    nop
1:
    bltz    sp, 2f
    nop
    lw  sp, KERNEL_SP
    nop

2:  nop
```

Thinking 3.8

- handle_int: lib/genex.S
- handle_mod: lib/genex.S
- handle_tlb: lib/genex.S
- handle_sys: lib/syscall.S

Thinking 3.9

```
LEAF(set_timer)

    li t0, 0xc8          #设置一秒钟中断200次，也即0xc8
    sb t0, 0xb5000100    #向0xb5000100的低16位写入0xc8，表示此时一秒钟中断200次，0xb5000000是模拟器映射实时钟的位置
    sw  sp, KERNEL_SP    #把sp寄存器的值写到KERNEL_SP
setup_c0_status STATUS_CU0|0x1001 0      #把CP0_STATUS变成0x10001001，低两位为1，表示当前处于用户态且中断开启；第12位是1，表示响应4号中断；第28位是1，表示允许在用户模式下
    jr ra                #返回

    nop
END(set_timer)
```

```
timer_irq:

    sb zero, 0xb5000110     #禁用时钟中断
1:  j   sched_yield         #跳转到调度函数，决定下一个时间片运行哪个进程
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception  #跳转至epc，从异常状态返回
    nop
```

Thinking 3.10

我们设置了存放就绪进程的数据结构，CPU每秒的中断次数，并设置每个进程能占用的时间块。当一个进程的状态转变为就绪态（ENV_RUNNABLE），我们就将其插入第一个链表；调用调度函数时，先判断该进程的时间片是否用完，当一个进程的时间片用完了就会暂停，且被插入另一个就绪状态进程链表的结尾，让CPU去处理别的内容，实现进程切换。如果当前就绪状态进程链表为空，则指针会被切换到另一个就绪状态进程链表检查上述情况。

实验体会

完成这部分课下内容的过程其实比 lab2 内存管理的部分轻松一些，主要是因为通过仔细阅读指导书了解了函数功能和一些重要的数据结构（比如env结构体的具体项目，包括这些项目的用途、出现位置等等），这样在书写代码时就能知道需要设置哪些结构体成员、调用哪些函数完成要求的过程，当然，细致的指导书与代码注释也起到了非常关键的作用。如果仅仅是为了完成代码任务，光看注释甚至都已经绰绰有余。在本单元中，我感觉宏观掌握函数之间调用关系的思路更加清晰了。

中断和异常部分，上一次接触是上学期的计组（当然是从硬件层面了解的），中间的软件实现过程在阅读过汇编代码的细节之后就不再那么扑朔迷离，许多过程是自然的（此处仍需要感谢指导书），主要想吐槽的就是此处汇编的语法与之前学习的毕竟有些许区别，在阅读与修改时难免还是有点阻滞。

此外，虽然详细的注释在我编写代码时帮了我不小的忙，但部分函数体中的注释内容依然让人有些不好下手，最后我选择了用其他的方式实现。因此或许可以精简一些提示，或者说将提示内容写得更灵活，从思路入手，而非直接提示需要使用的函数这样，让同学们在完成的过程中能够充分发挥自己的主观能动性。