杜佳男 21210110

操作系统lab3实验报告

1、实验思考题

Thinking 3.1

思考 envid2env 函数: 为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况?

答：首先我们可以先看 `envid` 是怎么生成的：

```
u_int mkenvid(struct Env *e) {
    u_int idx = e - envs;
    u_int asid = asid_alloc();
    return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx;
}
```

再来看一些 `env.h` 中相关的定义：

```
#define LOG2NENV    10
#define NENV        (1<<LOG2NENV)
#define ENVX(envid) ((envid) & (NENV - 1))
#define GET_ENV_ASID(envid) (((envid)>> 11)<<6)
```

因此可以看出，`envid` 由 `asid` 左移11位构成高5位，第11位为1，低10位为进程在相对数组 `envs` 基地址的偏移。

而在 `envid2env` 中，在 `envid` 不为0时对 `envid` 查找时也是采用偏移低10位偏移进行的，其中 `ENVX` 的定义已在前面给出。

```
    if(envid == 0){
        *penv = curenv;
        return 0;
    }
    else e = &envs[ENVX(envid)];
```

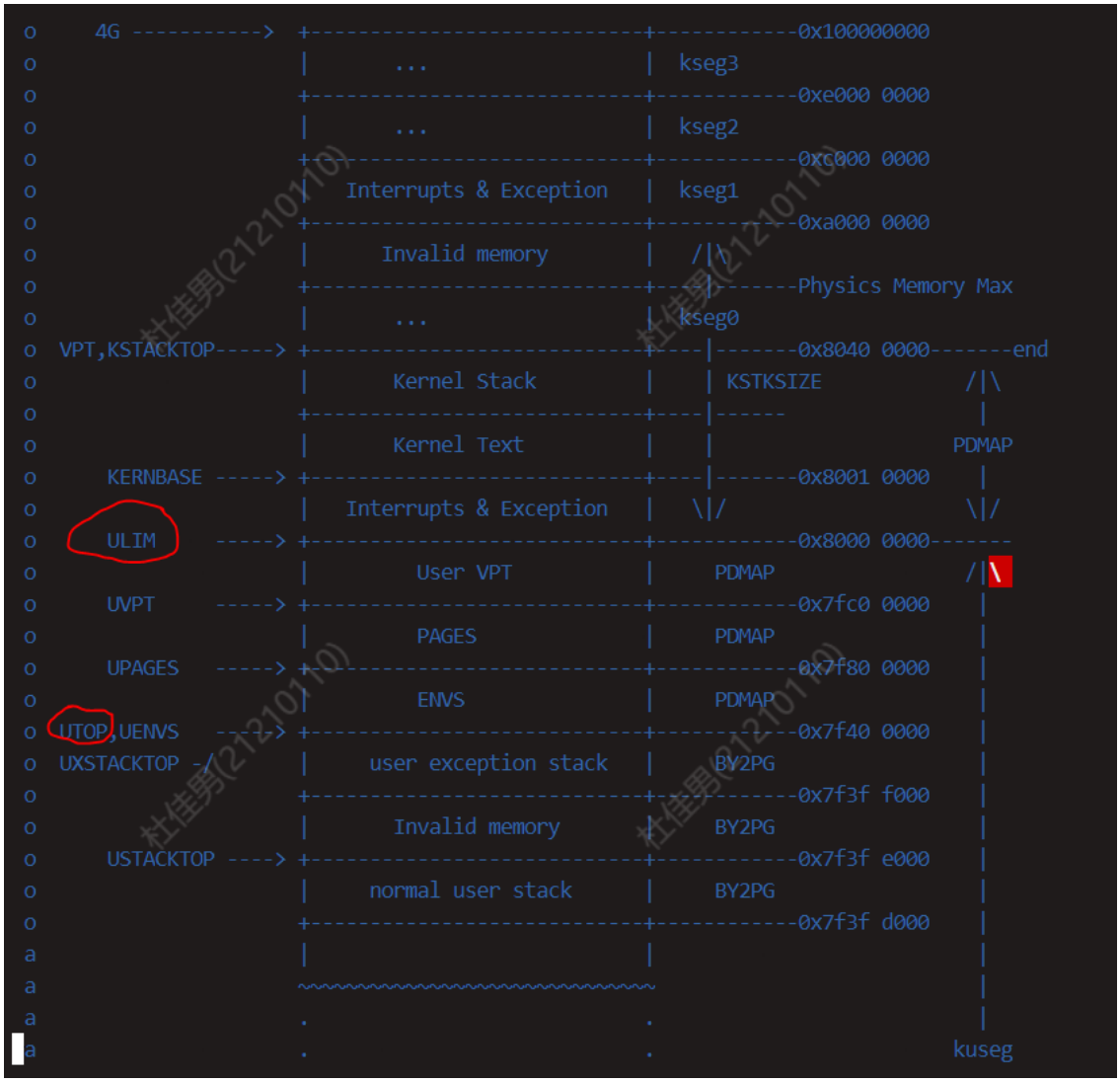
我们在之前的 `asid` 必要性中已经提到，多个进程有可能运行在同一个虚拟地址，通过 `asid` 映射到不同物理地址，对这些进程来说，物理地址不同，虚拟地址相同，`envid` 因 `asid` 不同因此高5位不同，但因虚拟地址相同具有相同偏移所以低10位相同，使得通过 `e = &envs[ENVX(envid)]` 访问到的进程不一定是所需要的进程，还有可能是其他地址空间相同虚拟地址上运行的进程，因此需要通过 `e->env_id != envid` 来校验。

Thinking 3.2

结合 include/mmu.h 中的地址空间布局，思考 env_setup_vm 函数：

- **UTOP 和 ULIM 的含义分别是什么，UTOP 和 ULIM 之间的区域与 UTOP 以下的区域相比有什么区别?**

  答：首先来看这张熟悉的内存分布图（include/mmu.h）：



  可以找到`UTOP = 0x7f400000`，在该地址以下为用户所能操纵的地址空间，包括`noral user stack`（普通用户栈）、`Invalid memory`（部分不可用内存）和`user exception stack`（用户异常栈），因此可以推断出`UTOP`的含义为用户所能操纵的地址空间的最大值；`ULIM = 0x80000000`，在`UTOP`和`ULIM`间为`ENVS`（进程空间）、`PAGES`（页空间）、`User VPT`（用户虚拟页表空间），均为系统分配给用户的地址空间，因此`ULIM`的含义为操作系统分配给用户地址空间的最大值。相较于`UTOP`以下区域用户的可操作性，`UTOP`和`ULIM`间的空间被定义为一个只读片段，属于内核态，用户访问此区域的内容需要进行系统调用，在此处进行读取不会陷入异常

- **请结合系统自映射机制解释代码中pgdir[PDX(UVPT)] = env_cr3的含义。**

  答：`UVPT`的含义为用户虚拟页表地址，其值为`0x7fc00000`，`PDX`作用为求一级页表号，也即页目录号，因此`pgdir[PDX(UVPT)]`的作用为寻找页目录中`UVPT`所在的页表项；`env_cr3`是页目录本身所在的物理地址，此代码的含义即为将这一物理地址填充到上述页表项。

- **谈谈自己对进程中物理地址和虚拟地址的理解。**

  答：进程只能操作虚拟地址，而实现虚拟地址和物理地址之间的映射由操作系统完成，每个进程有着各自独立的地址空间，进程切换时不同进程对相同的虚拟地址空间进行访问时互不影响。

Thinking 3.3

找到 user_data 这一参数的来源，思考它的作用。没有这个参数可不可以？为什么？（可以尝试说明实际的应用场景，举一个实际的库中的例子）

答：在函数`load_icode_mapper`中，被传入的`user_data`被强制转换为进程类型：

```
struct Env *env = (struct Env *)user_data;
```

因此可以猜测`user_data`实际上在函数中的真正含义就是指向被操作进程的一个指针。

进一步溯源，函数`load_icode`调用了`load_icode_mapper`，将其作为参数传入函数`load_elf`。`load_elf`的调用如下：

```
load_elf(binary, size, &entry_point, (void*)e, load_icode_mapper);
```

而在`load_elf`中，在使用传入的函数，也即使用`load_icode_mapper`时，传给`load_icode_mapper`中的`user_data`参数即为`load_elf`的`user_data`参数。

`load_elf`的定义：

```
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
             int (*map)(u_long va, u_int32_t sgsize,
                        u_char *bin, u_int32_t bin_size, void *user_data));
```

传参给输入函数：

```
 r = map(phdr->p_vaddr, phdr->p_memsz, binary + phdr->p_offset, phdr->p_filesz,
user_data);
```

回看`load_elf`的输入参数，在`user_data`位置的输入为`(void*)e`，即`load_icode`的输入参数`struct Env *e`，为待加载二进制镜像的进程，猜测得证。

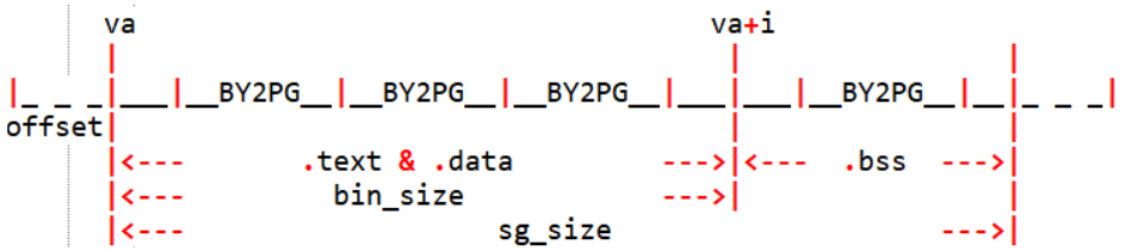
没有这一参数显然不可以，因此这一参数的作用是方便向更内层传参。库函数举例：

- 内存分配函数`malloc`其返回类型同样是`void*`型，用户在使用这个指针的时候，要进行强制类型转换。
- C语言标准库函数中的`qsort`函数，`void* base`参数同样可以传入任何类型的指针，最后一个参数传入的参数也是一个函数。

Thinking 3.4

结合 load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？

答：放一张指导书中复制到内存的图。



可能发生的情况有以下内容的排列组合：

- `va` 是否对齐（包含对齐和未对齐两种情况）
- `va + bin_size` 是否对齐（包含对齐和未对齐两种情况）
- `va` 和 `va + bin_size` 是否处于同一个页面（包含处于同一页面和不处于同一页面两种情况）
- `va + sg_size` 是否对齐（包含对齐和未对齐两种情况）
- `va + bin_size` 和 `va + sg_size` 是否处于同一个页面（包含处于同一页面和不处于同一页面两种情况）

一些可能的排列组合样例如下：

- `va`、`va + bin_size` 和 `va + sg_size` 均不对齐且三者处于同一个页面；
- `va`、`va + bin_size` 均不对齐，但处于同一页，`va + sg_size` 也不对齐，在另外的页；
- `va` 和 `va + sg_size` 对齐，`va + bin_size` 不对齐，三者处于不同页；
- ……

Thinking 3.5

e->env_tf.pc指示了进程要恢复运行时 pc 应恢复到的位置。冯诺依曼体系结构的一大特点就是：程序预存储，计算机自动执行。我们要运行的进程的代码段预先被载入到了 entry_point 为起点的内存中，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。

思考上面一段话，根据自己在 lab2 中的理解，回答：

- **你认为这里的 env_tf.pc 存储的是物理地址还是虚拟地址?**

  答：应该是虚拟地址。实际存储中大多存储指令所在的物理页面可能是分散的，一个进程程序段可能不在一段连续的物理空间上，但在计算机对指令的执行过程中，常用操作为PC = PC + 4，而不会在分散的地址间来回跳转，因此可以分析出env_tf.pc 存储的是虚拟地址，虚拟地址是连续的。

- **你觉得entry_point其值对于每个进程是否一样？该如何理解这种统一或不同?**

  答：entry_point对于每个进程来说是相同的，都是从elf文件中读取的，但他们储存的物理地址是不同的。

Thinking 3.6

请查阅相关资料解释，上面提到的 epc 是什么？为什么要将env_tf.pc 设置为 epc 呢?

答：`EPC` 寄存器是用来存放异常中断发生时进程正在执行的指令的地址的，在进入异常处理时，进程上下文中的 `env_tf.pc` 应该设置为 `epc`，以在处理完中断之后返回到之前的位置继续执行。

Thinking 3.7

关于 TIMESTACK，请思考以下问题：

- **操作系统在何时将什么内容存到了 TIMESTACK 区域**

  答：首先打开include/mmu.h可以看到 `TIMESTACK` 区域的地址：

  ```
  #define UTOP UENVS
  #define UXSTACKTOP (UTOP)
  #define TIMESTACK 0x82000000
  ```

  之后寻找对这一区域的调用，在include/stackframe.h可以找到如下代码段，分析如下：

  ```
  .macro get_sp
      mfc0    k1, CP0_CAUSE /*读取协处理器0的cause寄存器保存到k1中*/
      andi    k1, 0x107C /*取得k1的2-6位（即exeCode）和第13位*/
      xori    k1, 0x1000
      bnez    k1, 1f /*判断是否为时钟中断*/
      nop
      li  sp, 0x82000000 /*是的话把sp设为TIMESTACK用于保存上下文*/
      j   2f
      nop
  1: /*不是时钟中断跳转到这*/
      bltz    sp, 2f /*sp寄存器小于0则跳到2*/
      nop
      lw  sp, KERNEL_SP /*将sp寄存器的值写入KERNEL_SP*/
      nop
  2: /*是时钟中断，设置完sp后跳转到这*/
      nop
  .endm
  ```

  因此可以推测出 `TIMESTACK` 为发生时钟中断时存放寄存器状态的栈顶地址。在发生时钟中断进行进程切换时，我们会将寄存器的值保存在 `TIMESTACK` 对应的页面。

- **TIMESTACK 和 env_asm.S 中所定义的 KERNEL_SP 的含义有何不同**

  答：从上述 `get_sp` 的部分可以发现，在发生时钟中断时，向sp寄存器存入 `TIMESTACK` 的地址，将进程的寄存器状态保存到 `TIMESTACK` 中，若不为时钟中断，则向sp寄存器存入 `KERNEL_SP` 指向的地址，将进程的寄存器状态保存到 `KERNEL_SP` 中。

Thinking 3.8

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，TLB 异常，TLB 中没有和程序地址匹配的有效入口

3 号异常 的处理函数为 handle_tlb，TLB 异常，TLB 失效，且未处于异常模式（用于提高处理效率）

8 号异常 的处理函数为 handle_sys，系统调用，陷入内核，执行了 syscall 指令

试找出上述 5 个异常处理函数的具体实现位置。

答：`handle_int`、`handle_mod`和`handle_tlb`的具体实现位置在lib/genex.S中，handle_sys的具体实现位置在lib/syscall.S中。具体结果如下（部分截取）：

handle_int:

```
NESTED(handle_int, TF_SIZE, sp)
.set    noat

//1: j 1b
nop

SAVE_ALL
CLI
.set    at
mfc0    t0, CP0_CAUSE
mfc0    t2, CP0_STATUS
and t0, t2

andi    t1, t0, STATUSF_IP4
bnez    t1, timer_irq
nop
END(handle_int)
```

handle_tlb和handle_mod：

```
BUILD_HANDLER reserved do_reserved cli
BUILD_HANDLER tlb   do_refill   cli
BUILD_HANDLER mod   page_fault_handler cli
```

```
NESTED(do_refill,0 , sp)
            //li    k1, '?'
            //sb    k1, 0x90000000
            .extern mCONTEXT
//this "1" is important
1:          //j 1b
            nop
            lw      k1,mCONTEXT
            and     k1,0xfffff000
                mfc0        k0,CP0_BADVADDR
                srl     k0,20
                and     k0,0xfffffffc
            addu        k0,k1

            lw      k1,0(k0)
            nop
                    move        t0,k1
                    and     t0,0x0200
                    beqz        t0,NOPAGE
            nop
```

handle_sys:

```
NESTED(handle_sys,TF_SIZE, sp)

SAVE_ALL
CLI

//1: j 1b
nop
.set at
lw t1, TF_EPC(sp)
sw      t1, TF_EPC(sp)
la      t1, sys_call_table
lw      t2, (t1)
lw      t0,TF_REG29(sp)

lw      t1, (t0)
lw      t3, 4(t0)
lw      t4, 8(t0)
lw      t5, 12(t0)
lw      t6, 16(t0)
```

Thinking 3.9

阅读 kclock_asm.S 和 genex.S 两个文件，并尝试说出 set_timer 和 timer_irq 函数中每行汇编代码的作用。

答：在lib/kclock_asm.S中，set_timer的实现及解释：

```
LEAF(set_timer)
    li t0, 0xc8 /*将立即数0xc8存到寄存器t0，0xc8代表时钟频率，表示1秒钟中断200次*/
    sb t0, 0xb5000100 /*将t0寄存器保存的值写入地址为0xb5000100的内存，t0存的是上一行的时钟频率*/
    /*0xb5000000是模拟器(gxemul) 映射实时钟的位置。偏移量为0x100表示来设置实时钟中断的频率*/
    sw  sp, KERNEL_SP /*将sp寄存器的值存入KERNEL_SP*/
    setup_c0_status STATUS_CU0|0x10010 /*调用宏函数setup_c0_status设置CP0_STATUS的值*/
    /*设置CP0_STATUS第1位和第4、5位为10，第1位为EXL，表示是否处于异常级，异常发生时置1，第4、5位为KSU，代表CPU特权级，10代表用户级，紧跟在异常后无论此域为何值CPU都自动处于核心态*/
    jr ra /*返回*/
    nop
END(set_timer)
```

在lib/genex.S中，timer_irq的实现及解释：

```
timer_irq:
    sb zero, 0xb5000110 /*关闭实时钟*/
1:  j   sched_yield /*跳转到调度函数*/
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception /*从异常处理程序中返回*/
    nop
```

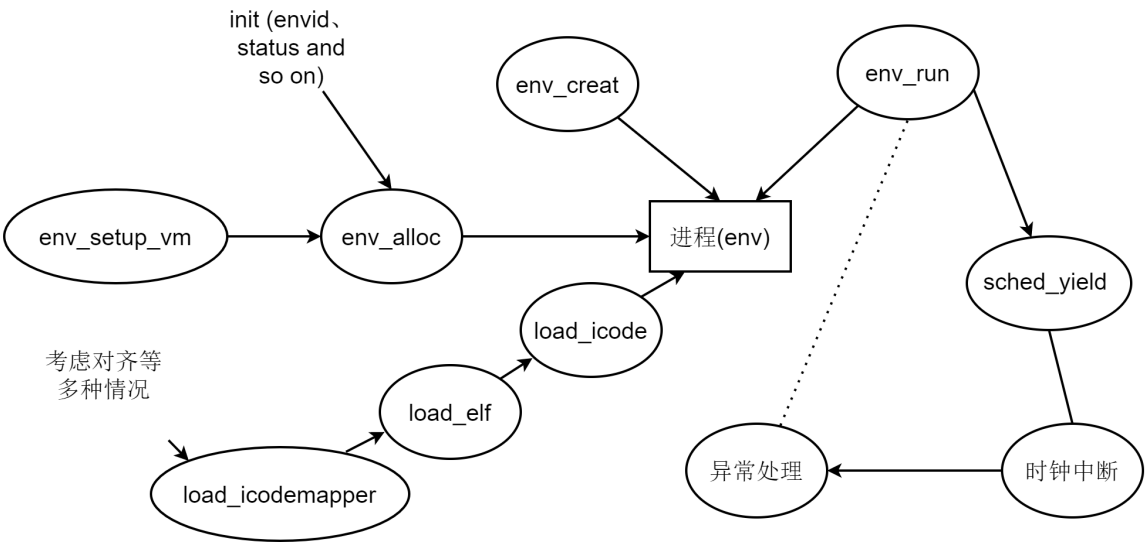
（没查到0x110的偏移量表示什么）

Thinking 3.10

阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。

答：采用时间时间片轮转算法，每个进程会被分到一个时间片，用作计时，设置一个进程就绪队列，每当一个进程的时间片用完，则代表该进程需要进行时钟中断，进行进程切换，若该进程未执行完，则被移动到就绪队列尾部，时间片复原，然后再从就绪队列首部取进程运行直到进程结束或时间片用完，按此规律循环。在本次实验中设置了两个进程列表，且列表中进程不一定是就绪进程，需要在系统取进程时进行判别，将进程的优先级视为该进程执行的时间片，并在系统取进程时将此优先级赋值给count，代表一个时间片，每运行一次count减一，当count变为0时代表该时间片用完，将其塞到另一个进程列表尾部，若此进程列表没有进程则转到另一个进程列表去取进程直到另一个进程列表为空。

2、实验难点图示



3、体会与感受

难度评价较高，实验花费时间约三天，实验报告约5h，实验量较大，整体来讲有好多内容，一头雾水，而且debug时间也久。需要去理解的内容很多，去查去理解的时候甚至可以直接崩溃，总体问题和上次实验相似，都是上手难，一旦上手后续处理则会简单许多，如二进制镜像文件的加载，一旦明白二进制镜像是怎么加载的（用于写代码）和加载过程可能遇到的情况（用于debug），就可以把这部分完成（当然还要知道bcopy和bzero，没意识到有这俩函数就可能像我一样直接抓瞎），再如进程调度，只要明白了时间片轮转算法，整个函数也就是一个队列操作的过程，也就没什么难点了（当然，不排除可能存在bug）。最最最后，一定要提防 “TOO LOW”，只能说看到这一行就心肺骤停……

4、指导书反馈

- 在一些细节的实现上与实验代码提示有些不太一致，比如进程调度时，进程时间片用完后，指导书写的插入头部，而代码则提示插入尾部；
- 有一小部分提示内容在文档版的指导书中有而网站中没有；
- 函数envid2env代码Overview提示中的*penv = envs[ENVX(envid)]不能直接用容易误导（不知道为啥直接这么写会出问题），用penv = &envs[ENVX(envid)]就可以；
- 加载二进制内核文件可以提示一下使用bcopy和bzero；
- 可以写一下相对gxemul映射实时钟的位置0xb5000000偏移量为0x110的值代表什么（在timer_irq将其置零，不确定是不是关闭实时钟，只知道偏移量0x110是设置时钟频率）。

5、残留难点

- 相对gxemul映射实时钟的位置0xb5000000偏移量为0x110的值代表什么
- 关于异常处理和时钟中断的诸多细节实现
- 加载二进制镜像的细节