Lab3实验报告

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?

Ans：在判断前，实际上已经有了e = &envs[ENVX(envid)];这表明e->env_id一定与envid在低10位上相同，因为e的产生就是通过低10位的序号来访问struct Env结构体的。但在ASID上却并不一定相同。因此如果没有这个判断，就有可能查询到一个不存在的进程。

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?

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

• 谈谈自己对进程中物理地址和虚拟地址的理解。

Ans：UTOP=0x7f400000，指的是用户所能操纵地址空间的最大值，ULIM=0x80000000，是操作系统分配给用户地址空间的最大值，这两者之间的区域是一个对于用户进程来说只读的片段，保存页表、Page、Env，UTOP以下的区域可以被用户读写。

pgdir[PDX(UVPT)]对应着页目录的起始地址，end_cr3是该进程页目录的物理地址，所以可以据此通过页表项的虚拟地址找到物理地址。

虚拟地址对于每个进程来说是独立的，但是不同进程的虚拟地址映射在了不同的物理地址上，进程间共享的部分除外，他们指向同一片物理地址。

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）

Ans：是进程的指针，根据这个指针可以找到该进程对应的页目录，从而根据页目录加载二进制代码到该进程的虚拟地址对应的物理地址上。存在user_data的目的是因为load_elf不仅仅只是将elf文件加载成进程，还可能是在链接过程中进行加载，此时加载的位置出现的变化，例如printf的实现过程之中，其对应的elf文件是被加载到了对应的进程之中，而不是单独创建进程。

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?

Ans：va不页对齐，va+bin_size不页对齐，va+sg_size不页对齐。加载.text .data .bss段时首先判断首是否页对齐，在判断尾是否页对齐。

3.5

这里的e->env_tf.pc是什么呢？这个字段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.pc 存储的是物理地址还是虚拟地址?

• 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?

Ans：虚拟地址。对每个进程都一样，由于*entry_point = ehdr->e_entry，但由于进程的不同对应的实际物理地址不同，需要统一的是虚拟地址的入口。

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?

Ans：epc是CP0寄存器组中的一个寄存器，用来存放异常发生时正在执行的指令地址，在切换进程时，相当于有一个异常，此时会把pc的值保存在epc中，下次到这个进程运行时便会从这里开始执行。

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

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

Ans：发生时钟中断时把当前进程上下文存在了TIMESTACK中，此外，每次异常中断时会调用汇编的SAVE_ALL函数重新存一边TIMESTACK

TIMESTACK是用于时钟中断时使用，而KERNEL_SP是在系统调用时使用

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。

Ans：handle_int函数在genex.S文件中，handle_sys函数在syscall.S文件中。另外三个函数handle_reserved、handle_tlb、handle_mod都在genex.S文件中，没有直接明确的函数名，是靠拼接而成，具体声明位于最后，但定义在最开始。

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

Ans：

```
LEAF(set_timer)
     li t0, 0xc8 /*设置t0为200*/
     sb t0, 0xb5000100 /*将t0存入0xb5000100，从而设置中断频率*/
     sw sp, KERNEL_SP /*设置时间中断对应的栈区*/
    setup_c0_status STATUS_CU0|0x1001 0 /*设置状态寄存器的相应位*/
     jr ra
     nop
END(set_timer)
```

```
timer_irq:
    sb zero, 0xb5000110 /*响应时钟中断*/
1: j sched_yield /*跳转到调度函数*/
```

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。

Ans：进程装在两个队列中，一次运行一个进程。定时器周期性产生中断，使得当前进程被迫停止，通过执行sched_yield函数，来进行进程的调度，若该进程时间片还未用完，则可用时间片数量 - 1，否则会切换到下一个进程，保存上下文。并将原来的进程送到另一个队列的末尾，若进程不处于RUNNABLE状态，则会进行其他处理。

实验难点图示

load_icode_mapper函数

首先判断首地址是否页对齐，把不是页对齐的部分单独放到一页里，然后把后续到bin_size段的部分加载进来，需要判断最后是否有没对齐的尾，有的话就只把这部分尾加入到单独的一页，最后把sgsize段加载进来即可

sched_yield函数

整个判断流程应该是：

1.判断是否需要调度（count==0？ curenv==null？ e->end_status==ENV_NOT_RUNNABLE）

2.不需调度，直接count--，需要调度则遍历当前的sched_list首

3.遍历过程中，如果可以运行则直接切换到此进程运行，不能则把他加到另一个sched_list中，如果这个sched_list遍历完了仍然没有符合要求的就换另一个sched_list

4.找到可进行调度的进程后切换count，最后count--并且env_run(e)

体会与感想

lab3中实验1每个函数的作用理解起来相对较容易，但是实现会难很多，尤其是加载二进制镜像的mapper函数，写了无数个版本，从一开始的自测斗不过，到自测过了测评没过，再到lab3的测评过了lab4中系统调用时不过，经历了无数次debug，但也对这部分有了更好的理解，核心思想还是在考虑页对齐等因素的情况下把程序放在虚拟地址上，其中又分了几种没对齐的情况。

此外就是进程调度那里，也写了无数个版本，由于lab3的测评数据不是很强，不判断notrunnable甚至都能过，但是到lab4中却过不了，因此也是反复写了好几版调度函数，最后才捋清思路。

整个lab3前半部分写的时间较长，整个lab3debug的时间也很长。。但是感觉写完之后对进程的理解更深刻，尤其是进程调度的实现原理（结合lab4中系统调用过程的断异常）。

指导书反馈

对env的部分有点疑惑，在lab3中env_setup_vm中写到

事情，请先阅读以下提示:

在我们的实验中，虚拟地址 ULIM 以上的地方，kseg0 和 kseg1 两部分内存的访问不经过 TLB，它们不归属于某一个进程，而是由内核管理的。在这里，操作系统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，这也是 MOS 的微内核设计。我们将在 lab4 和 lab6 中用到此机制。而这里我们要暴露的空间是 UTOP 以上 ULIM 以下的部分，也就是把这部分内存对应的内核页表拷贝到进程页表中。

Exercise 3.4 仔细阅读注释，填写 env_setup_vm 函数 ■

而后续进程间通信ipc中又提到

想要让两个完全独立的地址空间之间发生联系，最好的方式是什么呢？我们要去找一找它们是否存在共享的部分。虽然地址空间本身独立，但是有些地址也许被映射到了同一物理内存上。如果你之前详细地看过进程的页表建立的部分的话，想必你已经找到线索了。线索就在 env_setup_vm 这个函数里面。

```
static int
env_setup_vm(struct Env *e)
{
    //略去的无关代码

    for (i = PDX(UTOP); i <= PDX(~0); i++) {
        pgdir[i] = boot_pgdir[i];
    }
    e->env_pgdir = pgdir;
    e->env_cr3   = PADDR(pgdir);

    //略去的无关代码
}
```

感觉指导书说的稍微有点迷惑，个人理解是UTOP到ULIM中的内容可以在用户态下进行访问，ULIM中的内容只能在内核态访问，所以UTOP以上的内容对于进程来说都应该共享，只是其中不同部分有不同权限。

残留难点

由于中断和异常部分没写汇编代码，对这个过程的理解不是很透彻，导致lab4-2-extra没过（原来根据epc直接改就行）后来根据lab4系统调用的过程对这部分有了更好的理解，也可能中断与异常主要就由系统调用产生所以了解系统调用才会更好的理解中断与异常?