lab3实验报告

17373452 单彦博

一、思考题

1. **为什么我们在构造空闲进程链表时必须使用特定的插入的顺序？(顺序或者逆序)**

只有使用逆序插入，我们才能保证以从前往后的方式从`env_free_list`中取出PCB块，即第一次调用`env_alloc`能从中取出`envs[0]`。

2. **思考env.c/mkenvid 函数和envid2env 函数。**

   - **请你谈谈对mkenvid 函数中生成id 的运算的理解，为什么这么做?**

     先用一个静态变量表示第几次调用这个函数，然后左移11位，再或上使用的env在envs数组中的偏移。这样生成的envid可以确保是每个进程独一无二的。

   - **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?**

没有这步判断的话，会出现当一个envs[i]被使用两次后，会产生两个envid，而这两个envid的后11位是相同的，所以要进行这步判断，如果高位不相同，则表明这个envid的进程已经不存在。

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

   - **我们在初始化新进程的地址空间时为什么不把整个地址空间的pgdir 都清零，而是复制内核的boot_pgdir作为一部分模板？(提示:mips 虚拟空间布局)**

     我们MIPS的虚拟地址空间采用的是2G/2G的模式，即高2G是内核区，这个区域的虚拟地址到物理地址的映射关系都是一样的，低2G是用户区，对于所有进程都有这样的虚拟空间； 这也就意味着所有进程的页目录的一半都是一样的，所以我们可以用boot_pgdir作为一部分的模板。

   - **UTOP 和ULIM 的含义分别是什么，在UTOP 到ULIM 的区域与其他用户区相比有什么最大的区别?**

     `ULIM`是0x80000000，是操作系统分配给用户地址空间的最大值，`UTOP`是0x7f400000，是用户能够操控的地址空间的最大值。`UTOP`到`ULIM`这段空间用户不能写只能读，也应属于“内核态”，是在映射过程中留出来给用户进程查看其他进程信息的，用户在此处读取不会陷入异常。

   - **在step4 中我们为什么要让pgdir[PDX(UVPT)]=env_cr3？(提示: 结合系统自映射机制)**

     `UVPT`需要自映射到它在进程的`pgdir`中对应的页目录地址。这样当我们需要将`UVPT`这块区域的虚拟地址转换为物理地址时，就能方便地找到对应的页目录。

4. **思考user_data 这个参数的作用。没有这个参数可不可以？为什么？（如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子）**

在函数`load_icode_mapper`中，`void *user_data`被强制转换为`struct Env *env`,这也就意味着，这个用户数据指针，在函数中一直是当作一个进程指针在使用。没有进程指针，我们就不能进行映射和内存装载。

5. **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？**

要考虑到一些边角情况。首先需要注意的是va未必对齐了BY2PG，其次在拷贝的末尾需要注意va+bin_size也未必对齐了BY2PG，最后还需要注意sgsize可能大于bin_size，因此bin_size之后的部分需要填充0。

6. **思考上面这一段话，并根据自己在lab2 中的理解，回答：**

   - **我们这里出现的” 指令位置” 的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢?**

     针对虚拟空间定义的。

   - **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?**

     entry_point对于每个进程是一样的，因为elf文件都被加载到了固定位置，因此每个进程运行代码的入口点是相同的。ELF文件并不是一开始就是代码，而是要经历一系列的header和magic数据之后才可到达代码区。所以我认为，这种统一来自于ELF的格式统一。

7. **思考一下，要保存的进程上下文中的env_tf.pc的值应该设置为多少？为什么要这样设置?**

这个值应该设置为curenv->env_tf.cp0_epc，因为EPC寄存器存放的正是异常发生时，系统正在执行指令的地址，因此要保存的进程上下文中的env_tf.pc应该设置为该值。

8. **思考TIMESTACK 的含义，并找出相关语句与证明来回答以下关于TIMESTACK 的问题：**

   - **请给出一个你认为合适的TIMESTACK 的定义**

     TIMESTACK是时钟中断后存储进程状态的栈区。

   - **请为你的定义在实验中找出合适的代码段作为证据(请对代码段进行分析)**

     从mmu.h中我们可以了解到TIMESTACK的值为`0x8200 0000`，是处于内核区的。

     在我们的实验中有两处显式用到TIMESTACK：

```
//env_destory():
bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
      (void *)TIMESTACK - sizeof(struct Trapframe),
      sizeof(struct Trapframe));

//env_run():
struct Trapframe *old;
old = (struct Trapframe*)TIMESTACK - sizeof(struct Trapframe);
bcopy(old, &curenv->env_tf, sizeof(struct Trapframe));
curenv->env_tf.pc = old->cp0_epc;
```

根据使用TIMESTACK的行为，无论是在env_run()，还是在env_destory()中，我们都是把TIMESTACK之下的一段大小为sizeof(struct Trapframe)的内存作为存储进程状态的地方。在env_destory()中我们把存于KERNEL_SP的进程状态复制到TIMESTACK中；在env_run()中，我们从TIMESTACK中取出进程状态复制给当前进程。

此外，在发生时钟中断时， 我们会调用lib/genex.S里面的handle_int函数，在handle_int中，会调用include/stackframe.h中的SAVE_ALL，在SAVE_ALL中会调用get_sp，get_sp函数如下：

```
.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

.endm
```

这段代码是获取sp栈指针值，如果是中断异常，则会将sp置为0x82000000，如果是其他异常，会将sp置为KERNEL_SP，这样在发生中断时，我们就会将当前CPU状态上下文以Trapframe的形式存入TIMESTACK，之后进入调度函数sched_yield(),其调用env_run(),调用下一个就绪进程;在env_run()中,我们在把状态存入curenv->env_tf中，完成上下文的保存。

- **思考TIMESTACK 和第18 行的KERNEL_SP 的含义有何不同**

  TIMESTACK是时钟中断后进程状态的存储区，KERNEL_SP是系统调用后的进程状态的存储区。

9. **阅读kclock_asm.S 文件并说出每行汇编代码的作用。**

```
.macro  setup_c0_status set clr
  .set  push
  mfc0  t0, CP0_STATUS
  or  t0, \set|\clr
  xor t0, \clr
  mtc0  t0, CP0_STATUS
  .set  pop
.endm

  .text
LEAF(set_timer)
  li t0, 0x01
  sb t0, 0xb5000100
  sw  sp, KERNEL_SP
  setup_c0_status STATUS_CU0|0x1001 0
  jr ra
  nop
END(set_timer)
```

首先向0xb5000100 位置写入1，其中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，1 则表示1 秒钟中断1次。然后设置CP0_STATUS寄存器的状态，使系统可以响应时钟中断。最后通过jr ra返回。

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。**

一旦实时钟中断产生，就会触发MIPS 中断，从而MIPS 将PC 指向0x80000080，从而跳转到.text.exc_vec3代码段执行。对于实时钟引起的中断，通过text.exc_vec3代码段的分发，最终会调用handle_ int 函数来处理实时钟中断。在handle_ int 判断CP0_CAUSE寄存器是不是对应的4 号中断位引发的中断，如果是，则执行中断服务函数timer_ irq。在timer_ irq 里直接跳转到sched_ yield 中执行。而我们的sched_ yield函数会采用就绪进程链表调度，然后通过调用env_run函数来切换一个新的进程进行执行。

二、实验难点

1. 本次实验第一个难点是load_icode_mapper函数的填写，这个函数十分复杂，需要考虑周到，才可以填写正确。从调用关系上来看我们是在进程创建的过程中，将给定的elf文件加载到进程对应的位置，但是这个加载不上简单的一个bcopy就可以了，因为要考虑到一些边角情况。首先需要注意的是va未必对齐了BY2PG，其次在拷贝的末尾需要注意va+bin_size也未必对齐了BY2PG，最后还需要注意sgsize可能大于bin_size，因此bin_size之后的部分需要填充0。
2. 本次实验第二个难点就是sched_ yield这个调度函数的填写，今年的调度方式比往年有了升级，采用双就绪链表互相插入的方式，调度速度更快，但同时也要考虑到更多的问题。比如，进程执行的时间片数量怎么去控制，什么时候去切换另一个链表，如果两个链表中都没有进程改执行什么操作，这些都是需要我们仔细琢磨，才不至于写出bug。

三、感想与体会

1. 本次实验的过程中，我初步掌握了gxemul的调试方法，怎么去加断点，怎么单步运行，这些在以后的实验过程中是很重要的，可以帮助我们发现自己在哪里报了TOO_LOW。

2. 本次实验让我对进程的管理有了更深入的理解，对于这种新的调度方式也有了更深入的认识。我在课下花费的时间比较多，因为要自己填写很多函数，并要理解每个函数具体要做什么。而在课上测试中，我两次完成得都比较轻松，自我感觉对这方面的理解还是挺到位的。同时，我也感受到了OS_LAB各种神奇写法的巧妙之处，很值得学习。