Lab3 实验梳理

从逻辑上，Lab3主要包括三个模块：进程创建，中断处理，进程调度。

一，进程创建

1.env_init()

内核调用这个函数，完成进程创建前的准备工作——准备好PCB。之后在创建进程时，直接获取一个准备好的PCB即可。

PCB的准备工作有如下两个部分：

（1）PCB初始化

首先初始化空闲PCB链表与就绪队列；

然后将envs数组（此时储存着所有PCB块）中的PCB块状态改为ENV_FREE，插入空闲链表中备用。

（2）模板目录页创建

pages,envs数组这些数据本来是内核数据，但由于用户进程经常会进行访问，所以我们将其映射到用户进程可访问的地址空间中会更加方便。

作为所有进程的共享数据，这里的处理机制十分类似与进程之间的共享内存。

- 对于每个进程，将地址空间中的固定位置（UPAGES，UENVS）划分出来，并将pages，envs映射到这个位置

  （每个进程均通过相同的va访问这些资源的物理地址）

- 权限为设为只读

  （用户进程不能写只能读）

我们申请一个页作为页目录，并完成以上映射。以后新建进程时都进行拷贝，这样所有的进程实现了相同的访问功能。

（额外地说一下映射函数）

2.map_segment()

在指定的页目录中，将某一段虚拟空间地址映射到指定的物理空间地址，通过调用page_insert()实现。

由lab2中可知，本质还是修改pgdir中的页目录项（及其控制的页表项）。

3.env_create()

在init.c中，env_init()执行完成后，使用了宏ENV_CREATE_PRIORITY创建了两个进程。该宏正是调用了env_create()函数实现其核心功能。而这个函数主要做了三件事情

- 从PCB空闲链中申请一个属于自己的PCB（调用env_alloc()）
- 设置进程的优先级与状态，这些量**只与进程调度有关**，对于每个进程都不一样，所以不放在env_alloc()中执行
- 将进程要执行的程序装入内存（调用load_icode()）

详细看一下1，3点。

4.env_alloc()

我们在这里申请一个空闲PCB块；

但申请到的空白PCB还不能直接用于运行，所以我们还要进行一些进程状态量的设置。这些设置**只与进程执行有关**，所以放在env_alloc()中完成。

另外，还要建立进程的地址空间（使用env_setup_vm()完成）

5.env_setup_vm()

程序执行时要用到的虚拟地址暂不用管，进程创建时有两点需要立即处理：

- 为访问pages,envs等公共资源划分的地址空间（UTOP-UVPT），其中的内容直接从env_init()中创建的模板页目录上拷贝而来
- 每个进程存储自身页表的地址空间（UVPT-ULIM）,我们需要现在此处完成页目录自映射

这样处理后，页目录就完成了加工流程，可以设为e的pgdir了。

6.load_icode()

逻辑并不复杂，装入过程按照三步执行：

- 检查传入的是否binary是否为ELF文件头
- 遍历ELF文件的程序头表，对于每个需要装入的程序(ph->p_type == PT_LOAD),调用elf_load_seg()进行装入
- 装入完成，在trap frame中设置epc

7.elf_load_seg()

load_icode()通过elf_load_seg()实现内核功能。该函数与其参数中的map_page函数一同完成了内存映射。map_page作为回调函数，只接受地址并进行映射；而对于地址没有对齐、程序中有.bss段等处理逻辑则在elf_load_seg()中实现。处理的逻辑如下：

- 虚拟地址未页对齐：则首先“剪切”未对齐的部分，先进行映射

- 再将剩余数据正常按页映射
- 如果有.bss(段大小大于数据大小，这两个信息都在程序头表中)，为多出的部分申请空间，不进行填充

  （不填充体现：上面两个过程传入binary对应位置指针`binary + i`,这个过程传入NULL）

一言以蔽之，elf_load_seg()实现了不同（地址，.bss）情况下的内存映射。我们将不同情况的处理逻辑保留在该函数中，而将每个地方都用到的映射函数进行封装(map_page())。

lab3中，我们的map_page()函数仅用到了load_icode_mapper()

8.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)
```

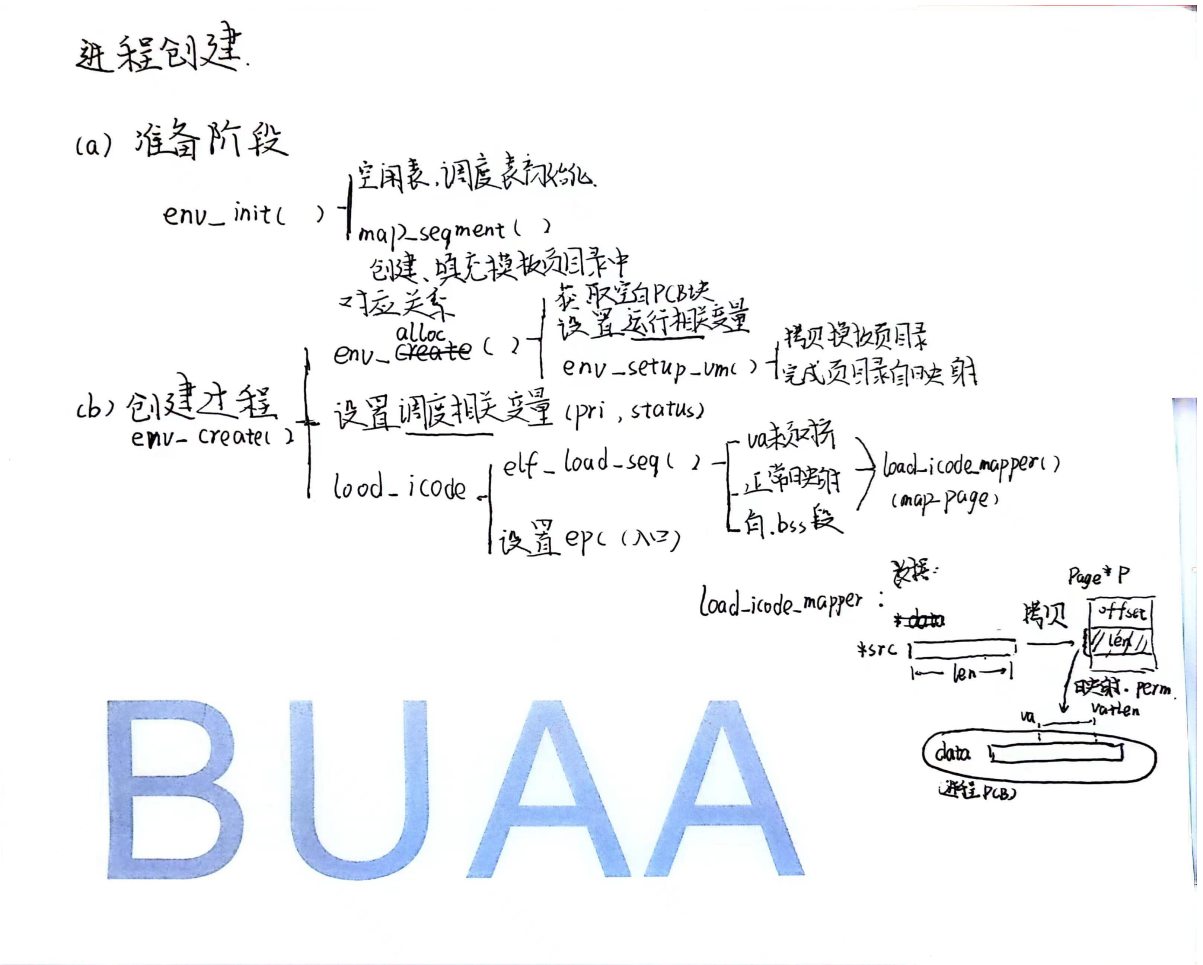
函数通过它接受到的参数，接到一个这样的任务：

- 现有一段待映射的数据，始址为src(为NULL则无待映射内容)，其长度为len;
- 我需要将这段数据映射到特定进程（对应于控制块data）地址空间内的va处，tlb中的权限位应为perm
- 此外应注意，映射到物理页时，物理页开头部分应空出offset大小的空间，以模拟内存不对齐的情况

合理调用memcpy()函数,满足这些要求即可。函数中的实际流程如下：

- 将data还原为PCB控制块
- 用page_alloc申请一个物理页
- 用memcpy()进行映射，三个参数`page2kva(p) + offset`,src和len分别对应了1、3点要求
- 最后建立进程地址空间和这段物理内存的联系(调用page_insert())函数

简要回顾一下，进程创建部分的流程与函数间关系如下：



二、中断处理

其实在lab3中，进程调度是时钟中断处理的最后一步（因为最后我们要换一个进程执行）。但这两个部分依然可以分别分析。

进程的整个生命周期中内容丰富，即使只关注异常处理也是如此。在lab3中我们只关心中断产生后的处理过程。

1.exc_gen_entry

时钟中断由硬件产生，产生过程、CP0中值的保存均由硬件完成。我们直接着眼于处理。

中断发生以后，程序会跳转至内存中的固定位置执行异常处理函数。（还记不记得0x4180啊23333333~）实验中，中断跳转至exc_gen_extry函数，tlb_miss跳转至tlb_miss_entry函数。

exc_gen_entry函数在执行异常处理时，要进行以下步骤：

- 通过SAVE_ALL保存当前上下文
- 修改cp0寄存器t0的值，进入内核态（关中断，允许嵌套异常）
- 检查cp0寄存器中cause保存的异常码，决定调用哪个异常处理函数

为了保证异常发生时的正常跳转，我们在kernel.lds中将exc_gen_entry和tlb_miss_entry放在约定的位置处。

2.handle_int

1号异常与2，3号的异常处理函数均不是本次实验的重点。我们关注0号异常处理函数。

这个异常处理函数的步骤如下：

- 判断是否为时钟中断
- 如果是，跳转至时钟中断处理程序timer_irq

lab3中，我们的中断异常只有时钟中断一种

3.timer_irq

我们处理时钟中断的方法就是重新调度程序。

步骤如下：

- //不懂li a0 0是干嘛用的，我是飞舞
- 跳转至schedule

注意，处理时钟中断调用的一系列函数均不返回。最后我们直接换了一个进程。

在原本的handler程序中，异常处理函数执行后会返回，并跳转到ret_from_exception函数，这里我们保存现场并重新执行。但在handler_int中，我们无需如此，而是通过另一种方式进行i跳转。

三、进程调度

使用进程调度有两个场景：刚执行完env_init(),将要调度一个进程去执行；以及有一个进程刚刚时间中断，需要重新调度。

（值的一提的是，在mips_init中，去年是将内核程序的中断初始化后开始while(1),等待中断后调度其他用户进程；今年我们直接调用schedule函数，这里略有不同）

1.schedule()

有两个步骤：

- 如果因为各种原因，我们必须要更换一个进程，那就进行更换
- 更新进程的时间片，然后执行env_run()

注意，函数中的count被赋值为e->pri,代表了进程一次可以执行多少个时间片。这也是进程优先级的体现，即优先级越高执行机会越多。

2.env_run():

我们马上就要执行一个进程了。如果这个进程是第一个执行的，那么curenv应该是null;否则说明我们正在换掉被中断的进程，换上新进程。

这个函数执行的步骤如下：

- 如果curenv不为null,保存（原）进程中的上下文
- 更换curenv,cur_pgdir（此时新进程就算是正式上位了）
- 调用env_pop_tf函数，进行新进程执行前的准备工作，最后开始执行

新进程的执行工作主要包括时钟控制硬件的配置，将当前进程所需上下文换到CPU中。

从此我们也能发现时钟中断异常的处理与其他异常处理的显著不同：其他异常处理并不切换进程，所以CPU的上下文也不需切换；而时钟中断需要切换进程，所以我们又要保存原进程的上下文，又要换入新进程的上下文。这也是handle_int如此特殊的原因之一。

3.env_pop_tf()

步骤如下：

- 我们用a1传入了进程的asid，此时我们把这个值存在entryHi寄存器中
- 把栈指针设为当前进程的trap frame处，后续调用ret_from_exception时就将按新进程的tf恢复上下文
- 通过宏RESET_KCLOK配置时钟
- 跳转至ret_from_exception

好的，绕了这么一圈，最后还是殊途同归了。

4.RESET_KCLOK

逻辑很简单，就是配置两个寄存器：

- 将count寄存器的值归零
- 将compare寄存器的值设为我们预先设定的值

之后时钟的计数由硬件自行完成，count=compare时将触发中断

5.ret_from_exception

- 调用宏RESTORE_ALL，恢复CPU上下文
- 执行eret，程序将恢复正常执行。

简单回顾，二、三部分的叙述可概括为如下流程：

中断处理 & 进程调度