OS Lab4 文档

515030910211姜子悦

PartA:

Exercise1

将mpentry映射的一页从freelist里拿掉，标记为已分配。比较简单

Exercise2

在pmap中，根据memlayout将iobase和多cpu的stack的位置映射。

坑点在于之前打开大页bit的操作只在mem init中作了，而没在percpu中做，所以会大页读取会挂掉。需要在每个cpu中打开大页。

Exercise3

将trap percpu中的ts改成每个cpu自己的ts，将对应的gate信息修改为自己stack所在的信息。

坑点是之前的sysente的wmrs在trap中做了，需要放到per cpu中做，并修改相应stack位置。

Exercise4

在相应的几个lab4 todo下边加lock，unlock就行了。已经为我们实现好了一个可以用的spin\_lock

Exercise4.1

实现ticket\_spin\_lock，和评分无关后面也重新注释掉了的一个独立练习，实现了一个十分慢的spin\_lock，让我们感受一下如何写简单的锁机制。

记得写完需要注释掉，不然最后一个exercise会timeout，直接run primes是可以抛出正确结果的，纯粹是因为太慢了会挂掉grade。

坑点是while循环中，需要给lk->own原子读或者volatile一下，似乎不是因为原子性，而是这个变量被编译器放进寄存器里以后会被优化掉，其他线程的修改寄存器不会再重新读一次值，导致值永远不会变。

Exercise5

完成yield函数，实现一个最简单的rr调度。选择当前env后一个runnable env来执行，如果没有runnable了，就继续跑自己，否则跑idle。

然后在syscall加上新的case和sys\_yield中调用sched\_yield

这个调度算法本身不算难写，但要记得判断当前env是不是running，因为其实当前env yield之前是可以变到not\_runnable的。

Exercise6

完成一系列操作内存的syscall，为之后的fork和ipc做准备。

比较简单的部分是加上新case，调用这些函数

实现syscall有点难，需要想清楚每个syscall的错误都要怎么处理。不过注释写得很详细，照着翻译可以解决一部分问题。

坑点在于有些syscall有5参数（之前实现的sysenter支持4参数），以及exofork并不走sysenter而是int 30。所以需要修改sysenter使他能传一个指向参数列表的指针，从参数列表取出必要参数。以及增🏠int 30 syscall所需要的gate、trapentry、handler一系列上次做过类似的函数。

一开始非常简单粗暴的直接干掉了sysenter，采用int 30来做syscall，很舒服很偷懒，什么bug都fix了。

最后做完整个lab之后回去考虑了一下sysenter如何修改，做出了相应修改。

至此part a完成，拿到了15分，有10分是送的。

不过为后面的part打好了基础，实现了很多useful的机制和函数。

PartB:

Exercise7

实现sys\_env\_set\_pgfault\_upcall

它可以为user pgfault注册一个自己想要的handler。

这个exercise还没有到user pgfault的核心部分，比较简单。

Exercise8

实现user page\_fault\_handler的trap部分，在上次写过的处理kernel pgfault的代码后面，写处理user pf。

如果是第一次处理，在uxstacktop处建一个utf结构，设置好各种值，指向pgfault\_upcall设置的handler

否则在esp-4的位置建一个utf结构，直到超出一个pgsize，就直接爆掉。

空出一个32bit是为了ret

Exercise9

实现\_pgfault\_upcall的汇编部分，因为传参和构造返回值以及跳转到handler这些东西c写比较麻烦，直接用汇编完成。最后能达到从kernel handler到uxstack上，ret到用户自定义handler的功能

根据注释写也还是难，汇编看得头疼。

Exercise10

完成pgfault\_handler最后部分，统合之前写的函数。

在uxstack上开一个pg为之后的utf做准备，然后调用\_pgfault\_upcall来完成pgfault处理

Exercise11

实现fork功能，和unix的cow fock类似，fock之后子进程有一个自己独立的env，但是va的映射还是原来的位置，并将子进程和父进程的这些va都置cow位。当要写的时候，再开一个新的page复制cow的内容过去，将权限改为w，这样就完成了cow机制的fork

PartC:

Exercise12

这个练习做打开中断，因为我们之前是默认没开中断，一旦有一个恶意进程无限spin会导致永远回不到kernel中，因为我们没有打断它的方法。通过硬件时钟中断可以做到这一点。

和上次lab异常、syscall的做法一样，先设置gate和写handler

然后在env\_alloc中初始打开user env的中断，这样我们就可以通过timer中断去yield一些运行很久的user env。

坑点在于gate的第二个参数istrap

上次看注释，说0 for interrupt， 1 for exception，于是都用1来调用，而这次的irp全用0调用。这样过不了grade。原来这个标志位为0代表trap的时候自动关中断，1代表手动关中断，而我们实现中trap异常和irq都是同一个函数来handle的，并且这个函数会assert是否已经关闭中断。

故所有的gate都应该以istrap=0来调用grade才能通过。

Exercise13

在dispatch里处理timer irq，使其做yield操作，来使我们的cpu调度更合理。每过一个timer中断，就会yield一次，完成rr调度。

Exercise14

这部分开始实现ipc了，做一个简单但完整的进程通讯过程。采取的是share page以及直接传value的方式。每次send&receive，接受者可以拿到一个value以及一个page映射，前者可以用来做一些简单的通讯，而后者可以用来传递大量信息。

注释十分完整，基本上逐条翻译即可。值得注意的是pg null不能直接传进去，因为0这个va其实是合法的映射地址。需要给pg为null的情况赋一个错值来表明这是一个null pg，只需要传递value的情况。

Challenge

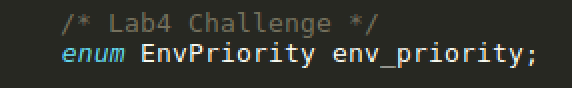
实现了priority的schedule算法

大体是实现了一个可以给env设置不同优先级，然后高优先级的env会被优先schedule的功能。

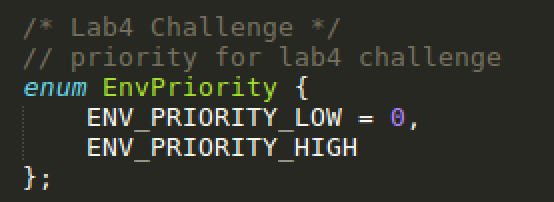
通过全局搜索Lab4 Challenge可以看到所有这个challenge修改的内容。

主要部分如下

在Env结构中增加一个priority来记录这个env的优先级



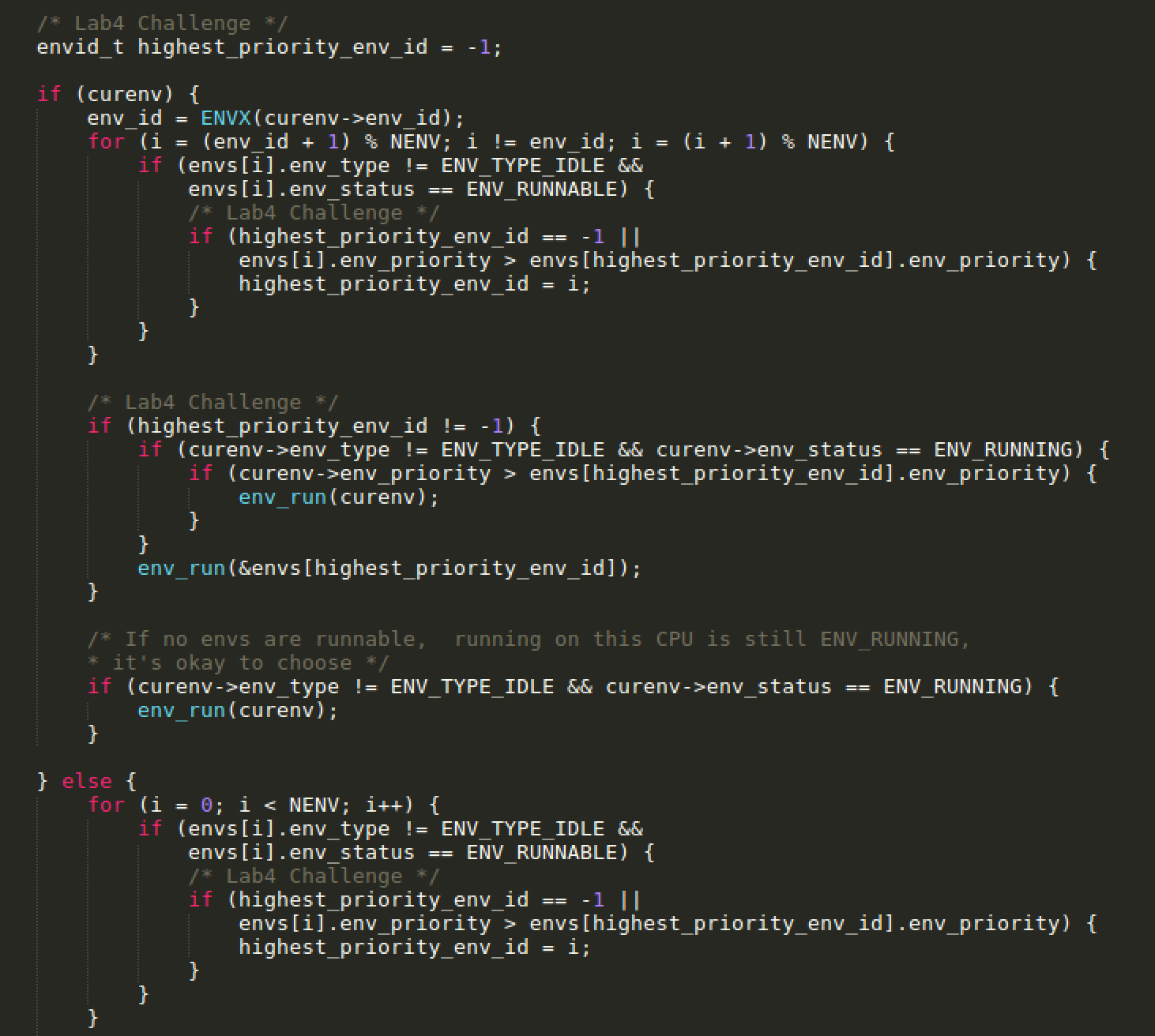
新的枚举集合表示不同优先级，目前只写了两个来实现功能，low是低优先级，high是高优先级。以后需要扩展十分容易



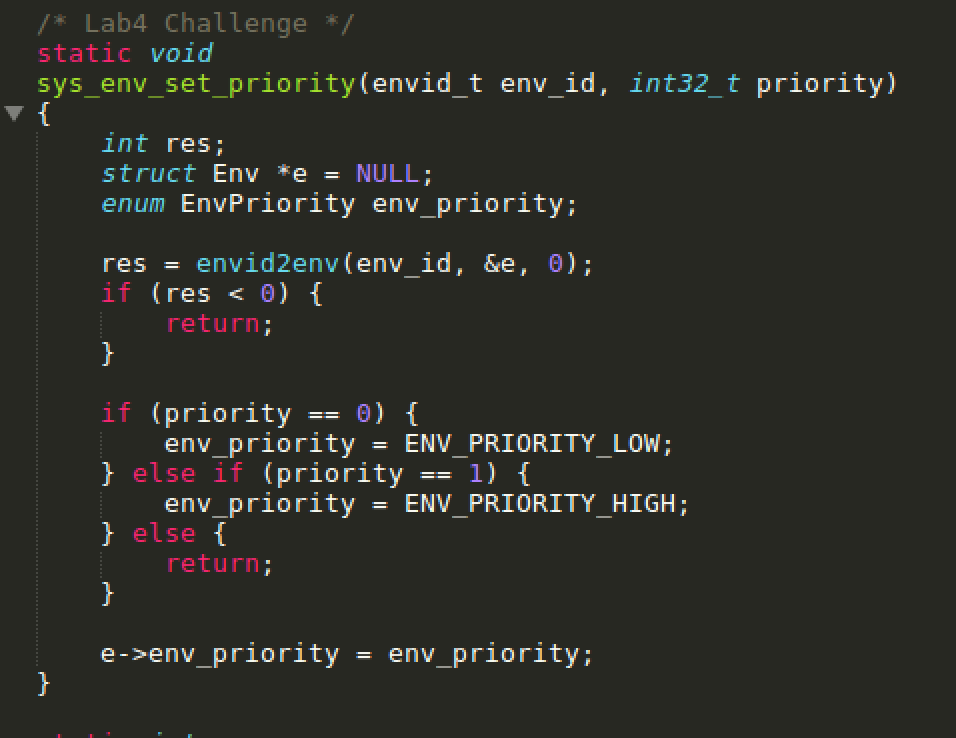
修改了原本exercise实现的调度算法，新的调度算法会优先调度优先级高的、能执行的env来run(包括自己)，同优先级会轮流执行一定时间片。

新的调度算法同样能过所有grade，拥有原来算法具有的所有功能。

不过由于优先级判断需要遍历env，所以调度会比之前的算法要慢一些。

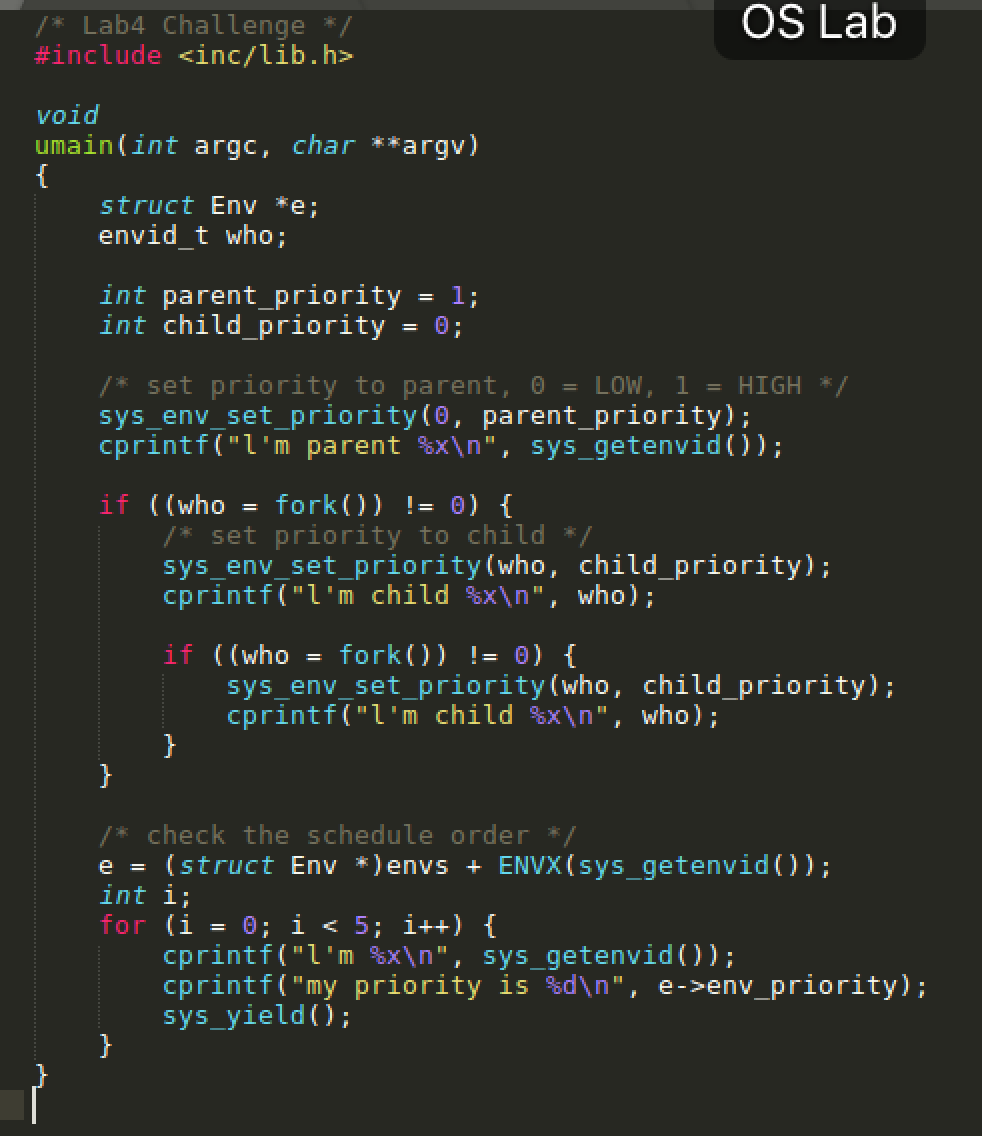


增加了一个syscall，可以调整优先级。并修改一个syscall需要的所有相应过程。

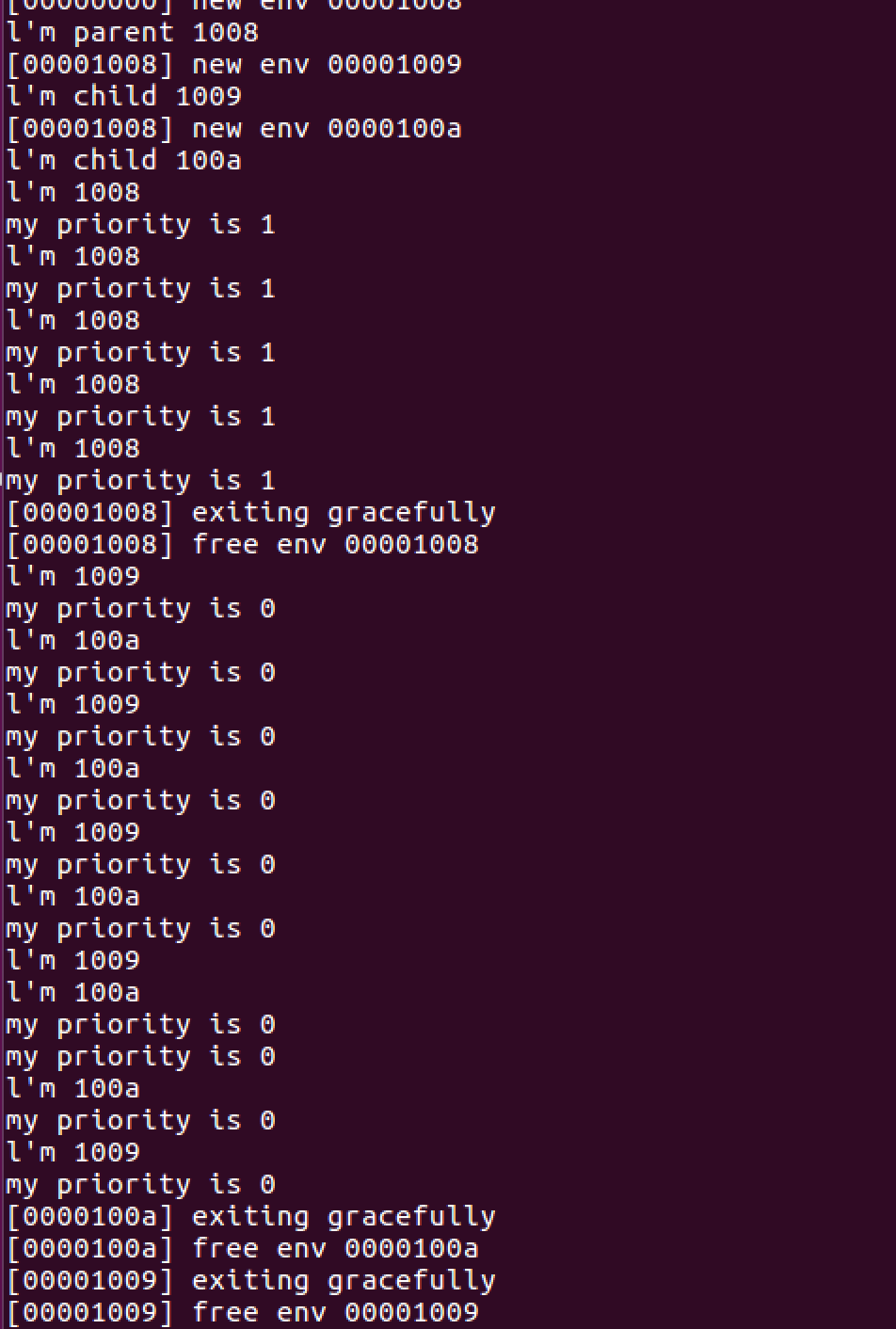


最后实现了一个user example来验证调度算法的正确性

可以通过修改parent和child的优先级来得到不同的结果，看看符不符合预期。希望达到的状态是优先级为1的先被执行完，然后再执行优先级为0的，并且优先级相同的env应该交替执行而不是独占。



先让父进程优先级为1，子进程为0



可以看到高优先级的父进程先被执行完了，然后两个同优先级的子进程交替执行。

然后交换父子优先级



同样是符合预期的，有一个0跑在1前面是因为在子进程env还没有完成set时，父进程已经向下执行了一个时间片。当子进程完成set后，直到子进程运行完成前父进程便不会再执行了。