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

```
    /* Lab4 Challenge */
    enum EnvPriority env_priority;
```

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

```
/* Lab4 Challenge */
// priority for lab4 challenge
enum EnvPriority {
    ENV_PRIORITY_LOW = 0,
    ENV_PRIORITY_HIGH
};
```

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。

```
/* Lab4 Challenge */
envid_t highest_priority_env_id = -1;

if (curenv) {
    env_id = ENVX(curenv->env_id);
    for (i = (env_id + 1) % NENV; i != env_id; i = (i + 1) % NENV) {
        if (envs[i].env_type != ENV_TYPE_IDLE &&
            envs[i].env_status == ENV_RUNNABLE) {
            /* Lab4 Challenge */
            if (highest_priority_env_id == -1 ||
                envs[i].env_priority > envs[highest_priority_env_id].env_priority) {
                highest_priority_env_id = i;
            }
        }
    }

    /* Lab4 Challenge */
    if (highest_priority_env_id != -1) {
        if (curenv->env_type != ENV_TYPE_IDLE && curenv->env_status == ENV_RUNNING) {
            if (curenv->env_priority > envs[highest_priority_env_id].env_priority) {
                env_run(curenv);
            }
        }
        env_run(&envs[highest_priority_env_id]);
    }

    /* If no envs are runnable,  running on this CPU is still ENV_RUNNING,
    * it's okay to choose */
    if (curenv->env_type != ENV_TYPE_IDLE && curenv->env_status == ENV_RUNNING) {
        env_run(curenv);
    }

} else {
    for (i = 0; i < NENV; i++) {
        if (envs[i].env_type != ENV_TYPE_IDLE &&
            envs[i].env_status == ENV_RUNNABLE) {
            /* Lab4 Challenge */
            if (highest_priority_env_id == -1 ||
                envs[i].env_priority > envs[highest_priority_env_id].env_priority) {
                highest_priority_env_id = i;
            }
        }
    }
```

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。

```c
/* Lab4 Challenge */
static void
sys_env_set_priority(envid_t env_id, int32_t priority)
{
    int res;
    struct Env *e = NULL;
    enum EnvPriority env_priority;

    res = envid2env(env_id, &e, 0);
    if (res < 0) {
        return;
    }

    if (priority == 0) {
        env_priority = ENV_PRIORITY_LOW;
    } else if (priority == 1) {
        env_priority = ENV_PRIORITY_HIGH;
    } else {
        return;
    }

    e->env_priority = env_priority;
}
```

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。

```c
/* Lab4 Challenge */
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    struct Env *e;
    envid_t who;

    int parent_priority = 1;
    int child_priority = 0;

    /* set priority to parent, 0 = LOW, 1 = HIGH */
    sys_env_set_priority(0, parent_priority);
    cprintf("l'm parent %x\n", sys_getenvid());

    if ((who = fork()) != 0) {
        /* set priority to child */
        sys_env_set_priority(who, child_priority);
        cprintf("l'm child %x\n", who);

        if ((who = fork()) != 0) {
            sys_env_set_priority(who, child_priority);
            cprintf("l'm child %x\n", who);
        }
    }

    /* check the schedule order */
    e = (struct Env *)envs + ENVX(sys_getenvid());
    int i;
    for (i = 0; i < 5; i++) {
        cprintf("l'm %x\n", sys_getenvid());
        cprintf("my priority is %d\n", e->env_priority);
        sys_yield();
    }
}
```

OS Lab

先让父进程优先级为 1，子进程为 0

```
[00000000] new env 00001008
l'm parent 1008
[00001008] new env 00001009
l'm child 1009
[00001008] new env 0000100a
l'm child 100a
l'm 1008
my priority is 1
l'm 1008
my priority is 1
l'm 1008
my priority is 1
l'm 1008
my priority is 1
l'm 1008
my priority is 1
[00001008] exiting gracefully
[00001008] free env 00001008
l'm 1009
my priority is 0
l'm 100a
my priority is 0
l'm 1009
my priority is 0
l'm 100a
my priority is 0
l'm 1009
my priority is 0
l'm 100a
my priority is 0
l'm 1009
l'm 100a
my priority is 0
my priority is 0
l'm 100a
my priority is 0
l'm 1009
my priority is 0
[0000100a] exiting gracefully
[0000100a] free env 0000100a
[00001009] exiting gracefully
[00001009] free env 00001009
```

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

```
l'm parent 1008
[00001008] new env 00001009
l'm child 1009
[00001008] new env 0000100a
l'm child 100a
l'm 1008
my priority is 0
l'm 1009
my priority is 1
l'm 100a
my priority is 1
l'm 1009
my priority is 1
l'm 100a
my priority is 1
l'm 1009
my priority is 1
l'm 100a
my priority is 1
l'm 1009
my priority is 1
l'm 100a
my priority is 1
l'm 1009
my priority is 1
l'm 100a
my priority is 1
[00001009] exiting gracefully
[00001009] free env 00001009
[0000100a] exiting gracefully
[0000100a] free env 0000100a
l'm 1008
my priority is 0
l'm 1008
my priority is 0
l'm 1008
my priority is 0
l'm 1008
my priority is 0
[00001008] exiting gracefully
[00001008] free env 00001008
```

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。