OSLAB-3实验报告

`OSLAB讲义`

> 南京大学匡亚明学院 刘志刚
> 学号：141242022
> 邮箱：njuallen@foxmail.com

实验进度

总体进度：完成了JOS的lab4，并以此为基础将我原来的游戏迁移了上去，并为JOS添加了若干系统调用，以满足我们实验的要求。

- [x] 进程的组织和调度
  - [x] 进程数据结构 （Env)
  - [x] 进程数据结构组织（一个数组而已）
  - [x] 进程管理的函数（create，yield，destroy）
  - [x] 进程调度（最简单的时间片轮转）
- [x] Fork（copy on write fork）
- [x] Exit
- [x] Sleep

具体说明：

- 游戏的运行
  在控制台输入make run-gobang即可，同时上下左右以及enter分别被替换成了kjhl(vim的上下左右）以及i。
- 进程内核栈
  在JOS中，所有进程共用一个内核栈。同时内核被设计为不可重入的，进入内核必须要先获得锁。同时在内核状态时是全程关中断的。通过这种方式来简化实现，避免可能出现的竞争问题。

- 进程管理的函数有：

```
// 初始化env数组
void    env_init(void);
// 初始化每个cpu的gdt, idt
void    env_init_percpu(void);
// 新分配一个空env（被env_create调用）
int env_alloc(struct Env **e, envid_t parent_id);
// 回收一个env的内存空间（被env_destroy调用）
void    env_free(struct Env *e);
// 创建一个env
void    env_create(uint8_t *binary, enum EnvType type);
// 销毁一个env并回收其占有的所有资源
void    env_destroy(struct Env *e);
// 由envid得到env结构体
int envid2env(envid_t envid, struct Env **env_store, bool checkperm);
// 由env结构体得到envid
int env2envid(struct Env *env);
// 运行一个env
void    env_run(struct Env *e) __attribute__((noreturn));
// 从内核栈上pop出一个trapframe（由env_run调用）
void    env_pop_tf(struct Trapframe *tf) __attribute__((noreturn));
```

- 进程调度
  在env结构体中我们已经留下了env_runs来记录当前进程的运行次数，但是当前我们尚未使用它。我们当前的调度方式是最简单的时间片轮转。只要一有时钟中断，我们就进行调度。但是由于我们现在的进程数目很少，所以这样子也没有什么问题。
- Fork
  完整实现了JOS的copy on write fork，可以通过它的所有test（make grade）。建议在控制台输入make run-forktree-nox测试一下效果。
- Sleep
  我在user文件中创建了一个sleep.c文件，用来测试sleep的实现。运行make run-sleep-nox可以看到每2s左右，程序会输出一行信息。
- 队列
  我在实现中没有使用任何队列，只是将进程的状态信息记录在env->env_status中，每次执行调度或唤醒只要一个循环将所有结构体查一遍即可。

问题及解决方法

我在tips文件中完整记录了我在完成JOS各个阶段过程中遇到的各种坑，以及自己的体会。JOS的设计非常简洁，但功能却又非常强大。里面的技巧很多，也很巧妙。有很多非常tricky的东西，具体请见tips文件，这里我只简要说一说我对lab4中一些问题的理解。

copy on write

在copy on write fork的实现中，我想到了一个问题。每当一个copy on write的页被写触发了一个page fault后，我们采取的是重新分配一个页，并把原来的内容拷贝进去的方法，那最开始的那一页最后是由谁释放的？后来我想了想，发现那个物理页是在引用计数变为0之后自动释放的，就和智能指针的释放差不多。

内核关中断

JOS是怎样实现内核关中断的？答案是将IDT的所有表项都设置为中断门，而不是陷阱门。利用x86中断门自动关中断，iret恢复eflags时自动开中断的特性来实现这个。

lapic eoi

PIC在发出中断信号给CPU之后会等待CPU返回一个中断结束信号（eoi），这样子后续的中断才能由PIC发出并由CPU处理。PIC返回中断结束信号的模式有好几种，我们一般见到的教学用操作系统基本都采取的是自动发送中断结束的信号给控制器的方式，不用我们手动发送eoi信号。但是JOS中采取的是处理完了之后，手动发送eoi信号的方式。在中断处理函数中如果忘了加上这个，就会导致中断只来一次的窘况。

键盘输入

JOS的键盘输入是分为两个来源的：串口输入（qemu的标准输入）以及键盘输入（从qemu弹出的那个新窗口中输入）。但是JOS在早期的lab中并没有开中断，为了在中断不开的情况下能正确处理串口输入，它的输入采用的是轮询的方式。如果有输入就返回非0值，如果没有输入就返回0。对键盘中断的处理是要到lab5中才加入的。所以现在我的游戏读取键盘是通过系统调用，而那个系统调用是采取轮询串口的方式读取键盘输入的。
在运行游戏时，我们不能切换到qemu弹出的窗口上输入，否则会触发一个键盘中断，导致游戏挂掉。我们只能在qemu的控制台上输入。

sleep的一些问题

在JOS中，如果当前没有进程可运行的话，就会陷入kernel monitor。而kernel monitor是内

核态的，运行时是关中断的。所以在使用sleep系统调用，让程序睡眠之后，会导致当前没有进程可运行，而且由于关了中断，它永远也不可能醒过来。为了解决这个问题，我在gobang游戏中fork了一个子进程，它始终处于就绪态。只要父进程一sleep，它就可以执行。这样子就防止了陷入kernel monitor。