

c 实验三 调 度

班级: 谢志鹏 姓名: 王傲 学号: 15300240004

1. [轮转调度策略]

修改 kern/sched.c 中的 sched_yield() 函数实现以下轮转调度策略:

- (1) 选择 envs 数组中最后运行的 env 之后 ENV_RUNNABLE 状态的一个, 如果是最后一个则选第一个;
- (2) 如果步骤 (1) 没有找到符合条件的 env, 而之前运行的 env 仍是 ENV_RUNNING 状态, 则继续选择它;
- (3) 调用 env_run() 函数运行步骤 (1) 或 (2) 选择的 env; 如果没有任何 env 被选定, 调用 sched_halt() 函数休眠此 CPU。

实现细节:

```
idle = thiscpu->cpu_env;
//use "idle" to record the current running process running on this CPU,
//which is the one that is going to be halted and replaced

uint32_t start = (idle != NULL) ? ENVX( idle->env_id) : 0;
//round robin organize processes in a loop
//use "start" to record where the current process is or set to 0 if there is no
process running
//so that when meet "start" again, the loop scanning is over
//"ENVX" is in inc/env.h, which is the environment index ENVX(eid) that
//equals the environment's offset in the 'envs[]' array, helping us to find the
index in envs

uint32_t i = start;
bool first = true; //set the flag to avoid stopping at first scan

for (; i != start || first; i = (i+1) % NENV, first = false)
//use mod to start at the front of the array, forming a loop
{
    if(envs[i].env_status == ENV_RUNNABLE)
    //find a process in envs[] that is runnable
    {
        env_run(&envs[i]); //start to run this process
        return ;
    }
}

if (idle && idle->env_status == ENV_RUNNING)
//if currently running process is not null and no other runnable process can be
found
{
    env_run(idle); //still run the current process
    return ;
}
// sched_halt never returns
sched_halt(); //halt the CPU if there is no runnable process
```

实验结果:

实验思路很简单, 实现 Round Robin 调度策略。每次要调度进程时, 在 envs[] 中寻找可以运行的进程替换当前进程; 若无可替换进程, 则继续运行当前进程; 否则, 将当前 CPU 挂起。Round Robin 找出在队列中离当前进程“最近”的进程进行替换。

由于在 kern/init.c 中存在以下代码:

```

#ifdef TEST
    // Don't touch -- used by grading script!
    ENV_CREATE(TEST, ENV_TYPE_USER);
#else
    // Touch all you want.
    //ENV_CREATE(user_primes, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
#endif // TEST*

```

所以输出结果如下:

```

[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.

```

2. [创建进程副本]

参考资料, 完成 kern/syscall.c 中的 sys_exofork 和 sys_env_set_status 系统调用, 实现简单的类 Unix 的用户态 fork 功能。运行 make run-dumbfork 进行测试。

实现细节:

```

sys_exofork:
    struct Env *e;
    int err;

    err = env_alloc(&e, curenv->env_id);
    //to allocate a new process, stored in "e"
    //"curenv->env_id" passed as parent_id, to set the parameter "e->env_parent_id"
    //as "e" is the child of current process

    if(err < 0)
        //env_alloc return 0 if succeed, <0 if fail
        {
            return err;
        }

    e->env_status = ENV_NOT_RUNNABLE;
    //set the status to ENV_NOT_RUNNABLE, as nothing is mapped in the user portion of
    its address space

    e->env_tf = curenv->env_tf;
    //set the trapframe, and the new environment will have
    //the same register state as the parent environment

    e->env_tf.tf_regs.reg_eax = 0;
    //set the register eax to 0 as the return value for the child process,
    //otherwise the child process will fork too, causing a non-stop duplicate

    return e->env_id;

```

```

sys_env_set_status:
    struct Env *e;

    //"envid2env" converts an envid to an env pointer stored in "e",
    //which could be used to set the status of this process

    //with parater "checkperm" set to 1(not 0), the specified environment must be
either the
    //current environment or an immediate child of the current environment, which will
    //check whether the current environment has permission to set envid's status.
    if(envid2env(envid, &e, 1) < 0)
    {
        return -E_BAD_ENV; //envid2env returns <0 / -E_BAD_ENV on error
    }

    //as is required, we can only set the status to ENV_RUNNABLE or ENV_NOT_RUNNABLE
    if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
    {
        return -E_INVALID;
    }

    e->env_status = status; //set the status
    return 0;

```

测试结果:

```

check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
0: I am the parent!
0: I am the child!
1: I am the parent!
1: I am the child!
2: I am the parent!
2: I am the child!
3: I am the parent!
3: I am the child!
4: I am the parent!
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001000] exiting gracefully
[00001000] free env 00001000
10: I am the child!
11: I am the child!
12: I am the child!
13: I am the child!
14: I am the child!
15: I am the child!
16: I am the child!
17: I am the child!
18: I am the child!
19: I am the child!
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

3. [写时复制优化]

(1) 阅读 MIT 6.828 Lab4 的练习 8~12, 对照注释和已实现的相关代码, 简要解释其作用, 包括: kern/syscall.c 中的 sys_env_set_pgfault_upcall 系统调用、kern/trap.c 中的 page_fault_handler() 函数、lib/pfentry.S 中的 _pgfault_upcall、lib/pgfault.c 中的 set_pgfault_handler()、lib/fork.c 中的 pgfault() 函数和 duppage() 函数。

(2) 参考资料, 完成 lib/fork.c 中的 fork() 函数, 实现支持写时复制的 fork 功能。

(3) 运行 make run-forktree 进行测试, 并对照 user/forktree.c 代码解释输出结果。

(1) 代码概述:

sys_env_set_pgfault_upcall:

这个函数的主要目的是当一个用户进程发生页错误 (page fault) 时, 该进程要在 JOS 的内核上记录一个 page fault handler entriypoint, 即设定 env 中 env_pgfault_upcall 参数的值。具体的, 当发生页错误时, 调用此函数, 输入进程 id, 利用 envid2env 函数将 id 转换为指向该进程的指针, 同时进行权限和异常判断。然后, 将 env_pgfault_upcall 设置为 func, 即在异常栈上记录一个错误记录, 然后分支到 func。

page_fault_handler():

当用户态下发生页错误时, 内核会让进程在异常栈下运行 pagefault handler 处理页错误。调用产生 pagefault upcall, JOS 会在用户异常栈上 (栈顶在 UXSTACKTOP, 正常栈的栈顶在 USTACKTOP) 生成一个 pagefault trapframe 记录错误信息 (UTrapFrame), 结构如下:

```
                                <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax                start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi                end of struct PushRegs
tf_err (error code)
fault_va                    <-- %esp when handler is run
```

其中 fault_va 是发生 pagefault 的虚拟地址。UTrapFrame 会被放置进异常栈。

特别注意, 如果当前进程已经运行在异常栈上的话, 说明 pagefault handler 本身出现了 page fault, 此时切换栈时栈顶直接在 trapframe 的 esp 寄存器上变换而不是 UXSTACKTOP。同时, 这是 handler 的递归调用, 要留出一个字 (4 字节) 的空间用以存储返回值。

给 UTrapFrame 在异常栈上分配空间后, 便设定相应的寄存器的值, 同时更改当前进程运行的栈到 UTrapFrame 上。最终, 销毁产生 page fault 的进程。

_pgfault_upcall:

这个函数的目的是返回发生 page fault (trap) 的地方而不用陷入内核态 (防止用户行为造成内核崩溃)。但我们不能在异常栈上直接使用 jmp 指令跳转, 因为返回发生 page fault 的位置后寄存器上的值应该为 trap 时的值; 如果直接调用 ret 指令返回, esp 寄存器的值是错误的, 仍为异常栈的栈顶。解决方法是切换到 trap 时的栈同时调用 ret 指令返回, 这会恢复 eip 寄存器的值, 返回到 trap 时的执行状态。这样的话, 寄存器的值是 trap 时的值, 栈也是进程执行时的栈。

set_pgfault_handler():

真正的处理 page fault, 设置 page fault handler。如果一个 UTrapframe 还没有 page handler 的话, 为它分配一个异常栈 (UXSTACKTOP 下的一页)。然后, 使用 sys_env_set_pgfault_upcall 函数来调用 _pgfault_upcall 进行处理。

pgfault():

注意: uvpt==user read-only virtual page table
uvpd==user read-only virtual page directory
PTE_COW==copy-on-write page table entries

这个函数的目的是因为 JOS 使用了写时复制, 只有要写一个子进程时才复制父进程相应的页面。这种情况下, 每次要写子进程, 都会产生 page fault。pgfault() 检查这是一个写错误 (FEC_WR) 并且该页的 PTE 被标志为 PTE_COW, 否则 panic。之后, pgfault 获取一个新的页面, 将它映射到一个临时的位置 (PFTEMP), 并且将相应的页的内容拷贝到里面。之后, 将该页重新映射到对应的位置, 并且设置好权限位。

duppage():

这个函数用来实现写时复制 (copy on write)，实现写时复制的映射。父进程对每个在自己 UTOP 下，且为可写或写时复制的页面，以写时复制的方式调用 sys_page_map 将其映射到子进程的地址空间中，并在自己的地址空间中将这个页面重新映射为写时复制（修改 perm 的值）。要被映射的页面编号为 pn，地址为 pn*PGSIZE，映射到编号为 envid 的进程的地址空间中。通过 (pte & PTE_W || pte & PTE_COW) 判断这个页面是可写或者写时复制的，并修改 perm 的值，增加 PTE_COW。然后，以 perm 为权限将其映射到子进程的地址空间。最后，在自己的地址空间中，用 perm 为权限值重新进行映射。

(2) 实现细节:

fork 函数:

比较复杂: 要考虑到存在二级页表

```
extern void _pgfault_upcall(void);
//because it is copy-on-write, a fork means there must be page faults

envid_t myenvid = sys_getenvid();
envid_t envid;
uint32_t i, j, pn;

set_pgfault_handler(pgfault);
//the parent installs pgfault() as the C-level page fault handler,
//using the set_pgfault_handler() function implemented above

if((envid = sys_exofork()) < 0) //create a child and check validity
    return -1;

if(envid == 0) //envid==0 means this is the child process(as the result of
sys_exofork), and we should prevent recursively creating children
{
    thisenv = &envs[ENVX(sys_getenvid())];
    return envid;
}

//copy address space to child

//Caution: Two Level page table
//First: scan Page Directory
for (i = PDX(UTEXT); i < PDX(UXSTACKTOP); i++)
//UTEXT: where user programs generally begin
//PDX(): Page Directory Index
    if(uvdp[i] & PTE_P)
        //uvpd: user read-only virtual page directory & present in the memory
        for (j = 0; j < NPTENTRIES; j++)
            //Second: scan Page Table Entry in the specific Page Directory
            //NPTENTRIES: the number of page table entries in a page directory
            {
                //get the number of the page
                pn = PGNUM(PGADDR(i, j, 0));

                //(UXSTACKTOP - PGSIZE) locates the user exception stack,
                //and user exception stack should never be marked copy-on-write
                if(pn == PGNUM(UXSTACKTOP - PGSIZE))
                    break;
                //if it is valid, then use duppage to complete COW mapping
                if(uvpt[pn] & PTE_P)
                    duppage(envid, pn);
            }

//as is said above, user exception stack should never be marked copy-on-write,
//so we must allocate a new page for the child's user exception stack
if(sys_page_alloc(envid, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_P | PTE_W) < 0)
    return -1;
```

```

        //invoke the mapping of the page (user exception stack),
        //stored in temporary 'PFTEMP'
        if(sys_page_map(envid, (void *)(UXSTACKTOP - PGSIZE), myenvid, PFTEMP, PTE_U |
PTE_P | PTE_W) < 0)
            return -1;

        //finish the mapping
        memmove((void *)(UXSTACKTOP - PGSIZE), PFTEMP, PGSIZE);

        //unmap the temporary PFTEMP
        if(sys_page_unmap(myenvid, PFTEMP) < 0)
            return -1;

        //set the pagefault upcall for the child process
        if(sys_env_set_pgfault_upcall(envid, _pgfault_upcall) < 0)
            return -1;

        //set the child process to ENV_RUNNABLE
        if(sys_env_set_status(envid, ENV_RUNNABLE) < 0)
            return -1;

        return env;

```

(3) 测试结果:

```

[00002000] exiting gracefully
[00002000] free env 00002000
2001: I am '000'
[00002001] exiting gracefully
[00002001] free env 00002001
1002: I am '1'
[00001002] new env 00003001
[00001002] new env 00003000
[00001002] exiting gracefully
[00001002] free env 00001002
3000: I am '11'
[00003000] new env 00002002
[00003000] new env 00001005
[00003000] exiting gracefully
[00003000] free env 00003000
3001: I am '10'
[00003001] new env 00004000
[00003001] new env 00001006
[00003001] exiting gracefully
[00003001] free env 00003001
4000: I am '100'
[00004000] exiting gracefully
[00004000] free env 00004000
2002: I am '110'
[00002002] exiting gracefully
[00002002] free env 00002002
1003: I am '01'
[00001003] new env 00003002
[00001003] new env 00005000
[00001003] exiting gracefully
[00001003] free env 00001003
5000: I am '011'
[00005000] exiting gracefully
[00005000] free env 00005000
3002: I am '010'
[00003002] exiting gracefully
[00003002] free env 00003002
1004: I am '001'
[00001004] exiting gracefully
[00001004] free env 00001004
1005: I am '111'
[00001005] exiting gracefully
[00001005] free env 00001005
1006: I am '101'
[00001006] exiting gracefully
[00001006] free env 00001006

```


结果解释:

user/forktree.c 中主要起作用的是 forktree 和 forkchild 两个函数, 目的是递归的通过子进程产生字符串。forktree 最开始输入一个空的字符串, 然后分支 0 和 1, 调用 forkchild; forkchild 自己 fork 了一个子进程, 通过对 id 是否为 0 进行判断, 父进程返回, 子进程递归调用 forktree 产生更长的字符串。在宏中定义了深度最大为 3, 所以字符串范围从 “” 到 “111”。

4. [抢占式多任务]

(1) 阅读 MIT 6.828 Lab4 的练习 13~14, 对照注释和已实现的相关代码, 简要解释其作用, 包括: kern/trapentry.s 和 kern/trap.c 中的中断表和中断处理函数的作用。

(2) 参考注释, 完成 kern/trap.c 中 trap_dispatch() 函数的时钟中断处理、kern/env.c 中 env_alloc() 函数内用户态中断相关配置。

(3) 运行 make run-spin 进行测试, 并对照 user/spin.c 代码解释输出结果, 如果没有时钟中断会怎样?

(4) 运行 make qemu CPUS=X 进行测试 (实际运行的是 3 个 yield 程序, 详见 kern/init.c), 尝试不同的 CPU 数量 X, 对照 user/yield.c 代码解释输出结果。

(1)

概述:

Lab4 以 user/spin.c 为例: 当进入子进程时, 进入空转, 其他进程永远无法使用这个 CPU, 说明要实现时钟中断等外部中断 (IRQ) 机制, IRQ (外部中断) 共有 16 种。这里使用了 IRQ_OFFSET (32) 作为偏移量, 防止与 CPU 引发的内部中断产生重合。设置时钟中断为 IRQ 0, 则 IDT[IRQ_OFFSET+0] 存储了内核中处理时钟中断的中断处理器的地址。以此类推。

代码概述:

kern/trapentry.s 中的中断表记录了 16 种外部中断对应的 entry points, 通过 IRQ_OFFSET 和相应序号, 以 IRQ_TIMER 为起始位置实现标识; kern/trap.c 中增加了对外部中断的中断处理函数, 方法与 CPU 内部中断 trap 类似, 外部中断标识方法与中断表相同。

(2) 实现细节:

trap_dispatch():

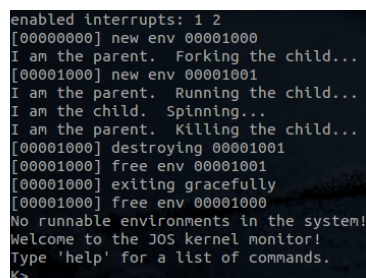
```
if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER)
{
    lapic_eoi();//used to acknowledge the interrupt
    sched_yield();//we have a interrupt now, so we schedule another process
    return;
}
```

env_alloc():

```
e->env_tf.tf_eflags |= FL_IF;

//in JOS, we make a key simplification compared to xv6 Unix.
//External device interrupts are always disabled when in the kernel
//(and, like xv6, enabled when in user space)
//External interrupts are controlled by the FL_IF flag bit of the %eflags register
//When this bit is set, external interrupts are enabled
```

(3) 测试结果:



```
enabled interrupts: 1 2
[00000000] new env 00001000
I am the parent. Forking the child...
[00001000] new env 00001001
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
[00001000] destroying 00001001
[00001000] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

结果解释:

从 user/spin.c 的代码来看, 实现比较简单。首先, 父进程 fork 一个子进程, 子进程进入了永远的空转。父进程继续执行, 通过执行几次 sys_yield() 调度子进程。由于子进程无法退出, 每次都是由于发生时钟中断后子进程退出控制, 将控制权交给了父进程继续执行。父进程最后杀掉了子进程。

如果没有时钟中断, 父进程第一次调用 sys_yield() 将控制权交给子进程后, 子进程便进入空转, 永远无法离开。

没有时钟中断的结果如图:

```
[00000000] new env 00001000
I am the parent. Forking the child...
[00001000] new env 00001001
I am the parent. Running the child...
I am the child. Spinning...
```

(4) 测试结果:

```
→ os2017 git:(lab3) X make qemu CPUS=4
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 4
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 4.
Back in environment 00001001, iteration 4.
Back in environment 00001000, iteration 4.
All done in environment 00001002.
All done in environment 00001000.
All done in environment 00001001.
[00001002] exiting gracefully
[00001002] free env 00001002
[00001001] exiting gracefully
[00001001] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
```

结果解释:

根据指令中 CPUS 参数的数量启动相应数量的 CPU (根据宏中定义, CPU 最多只有 8 个), 其中 CPU 0 为 BSP, 其余为 AP, 可以在 QEMU 的输出中看到。

在 kern/init.c 中, 存在如下的启动过程:

```
// Touch all you want.
//ENV_CREATE(user_primes, ENV_TYPE_USER);
ENV_CREATE(user_yield, ENV_TYPE_USER);
ENV_CREATE(user_yield, ENV_TYPE_USER);
ENV_CREATE(user_yield, ENV_TYPE_USER);
```

可以看出, 创建了三个进程 (1000, 1001, 1002), 运行 yield 程序 (以用户态加载 yield 的二进制文件并运行)。

yield 实现的功能也比较简单, 主要使用了 sys_yield() 系统调用实现用户态下的进程切换。总共迭代 5 次, 每一次迭代调用 sys_yield() 函数进行进程切换, 同时输出这个进程的编号和迭代轮数。在试验了不同数量的 CPU、不同的 yield 程序的进程数后, 发现一个运行 yield 程序的进程通过调用 sys_yield(), 会切换到另一个 yield 进程, 这样就会来来回回的切换, 直至 5 轮迭代终止。

5. [高级调度策略] (选做)

参考资料，替换 kern/sched.c 中的 sched_yield() 函数，实现除轮转调度外任意一种调度策略，并编写测试程序验证其正确性。必要时可以修改内核数据结构，增加统计量等。

调度策略描述：

考虑采用以执行顺序为优先级的调度策略（实际上是先来先服务，FCFS）：先执行的进程（也是时间上最久的）优先级小，每次选取最小优先级的进程执行，同时更新优先级。具体的，初始化时，由于直接获取时间比较困难，而进程的编号是按执行顺序赋予的，便将进程的编号作为它的优先级。每次要调度时，选取具有最小优先级并且可运行的进程，然后更新这个进程的优先级（变大）以防止每次都执行它；没有可运行进程的话，继续运行原来的进程；否则，将这个 CPU 挂起。这个调度策略不会发生死锁，而且可以有效避免饥饿（starvation）。

设计与实现细节：

inc/env.h: 给 struct Env 添加成员：uint32_t priority

kern/env.c: 在 env_alloc 函数中初始化优先级为进程编号：e->priority=e->env_id;

sched_yield 函数：

```
void sched_yield(void)
{
    struct Env *idle;
    int iter;
    idle = thiscpu->cpu_env;
    uint32_t priority_array[NENV];
    uint32_t max_priority = -1;
    uint32_t min_priority = 2147483647;
    int chosen_env = -1;

    for(iter = 0; iter < NENV; iter++)
        priority_array[iter] = envs[iter].priority;
        //record every process's priority

    for(iter = 0; iter < NENV; iter++)
        if(max_priority < envs[iter].priority
            && envs[iter].env_status == ENV_RUNNABLE)
            max_priority = envs[iter].priority;
            //record the maximum priority of runnable process

    for(iter = 0; iter < NENV; iter++)
        if(min_priority >= envs[iter].priority
            && envs[iter].env_status == ENV_RUNNABLE)
            chosen_env = iter;
            //find the runnable process with the minimum priority as next running process

    if(chosen_env >= 0 && chosen_env < NENV) //if the new process does exist
    {
        envs[chosen_env].priority=(max_priority+1) % 2147483647;
        //update its priority in case of fixed priority
        env_run(&envs[chosen_env]);
        return;
    }
    else if (idle && idle->env_status == ENV_RUNNING)
    {
        env_run(idle); //still run the current process if no other process can run
        return;
    }

    sched_halt(); //halt the CPU if there is no runnable process
}
```

验证测试结果：

如果调度算法正确的话，在发生调度时每次都是执行的最先被执行的（也就是时间上最久的），这样一轮下来所有进程都应该被执行。测试使用五个 yield 查看结果。

结果如下：

```
Back in environment 00001004, iteration 0.
Back in environment 00001003, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 0.
Back in environment 00001004, iteration 1.
Back in environment 00001003, iteration 1.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 1.
Back in environment 00001003, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 2.
Back in environment 00001004, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001003, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 3.
Back in environment 00001004, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001003, iteration 4.
Back in environment 00001001, iteration 4.
All done in environment 00001000.
Back in environment 00001004, iteration 4.
All done in environment 00001003.
All done in environment 00001001.
[00001000] exiting gracefully
[00001000] free env 00001000
All done in environment 00001004.
[00001003] exiting gracefully
[00001003] free env 00001003
[00001001] exiting gracefully
[00001001] free env 00001001
[00001004] exiting gracefully
[00001004] free env 00001004
Back in environment 00001002, iteration 4.
```

每一轮迭代调度的顺序是：

iteration 0: 1004 -> 1003 -> 1001 -> 1000 -> 1002

iteration 1: 1004 -> 1003 -> 1001 -> 1000 -> 1002

iteration 2: 1003 -> 1001 -> 1000 -> 1004 -> 1002

iteration 3: 1003 -> 1001 -> 1000 -> 1004 -> 1002

iteration 4: 1000 -> 1003 -> 1001 -> 1004 -> 1002

可以看出，五轮迭代，每一轮迭代中五个进程被逐次遍历，证明了调度算法的正确性。

测试的代码如下（通过迭代，调用 sys_yield，检查调度顺序是否正确）：

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    int i;

    cprintf("Hello World, I am environment %08x.\n", thisenv->env_id);
    for (i = 0; i < 5; i++)
    {
        sys_yield();
        cprintf("Switch to environment %08x, iteration %d.\n",
            thisenv->env_id, i);
    }
    cprintf("All done in environment %08x.\n", thisenv->env_id);
}
```

感想与体会

这次的实验工作量很大，然而与之前一样，搞清楚函数的功能，结合注释，还是可以得到结果的。难点之一是 fork 函数相关功能的实现：由于产生了多进程，与之前编程的线性思维不一致，有许多地方要注意，比如 fork 时要把子进程的 eax 寄存器设为 0，否则会递归产生无数子进程。这样的思维方式之前很少遇到，所以不注意的话很容易出问题。难点之二是写时复制的实现：由于相关函数很繁琐，加上有二级页表，映射关系比较混乱，实现起来比较麻烦。难点之三是自己实现新的调度策略：如果理解了 JOS 提供的 sched_yield 函数后实现新的调度策略会比较容易，但是刚开始时想利用时间确定优先级，却发现 JOS 无法提供 C 语言的<time.h>，而利用硬件获取时钟又十分复杂，加上进程间共享时间信息需要额外的通信机制，所以最终决定利用进程的执行顺序作为优先级实现调度。

这次实验工作量很大，但彻底弄清楚相关问题后，收获同样很大。