《操作系统原理》lab6实验报告

专业：计算机科学与技术

姓名：郝裕玮

学号：18329015

1. 实验目的

①理解操作系统的调度管理机制

2. 实验过程与结果

练习 0：填写已有实验

本实验依赖实验 1/2/3/4/5。请把你做的实验 2/3/4/5 的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”“LAB5”的注释相应部分。并确保编译通过。注意：为了能够正确执行 lab6 的测试应用程序，可能需对已完成的实验 1/2/3/4/5 的代码进行进一步改进。

答：需要修改的有 debug/kdebug.c，trap/trap.c，mm/default_pmm.c，mm/pmm.c，kern/process/proc.c，mm/vmm.c，swap_fifo.c 这 7 个文件，使用 meld 进行修改即可。

同时由题目可知需对已完成的实验 1/2/3/4/5 的代码进行进一步改进，具体改动如下：

①对于 kern/process/proc.c 中的 alloc_proc 函数：

由于在 proc.h 中，proc_struct 新增了 6 个成员变量：

```
    struct run_queue *rq;                       // running queue contai
ns Process
    list_entry_t run_link;                      // the entry linked in
run queue
    int time_slice;                             // time slice for occup
ying the CPU
    skew_heap_entry_t lab6_run_pool;            // FOR LAB6 ONLY: the e
ntry in the run pool
    uint32_t lab6_stride;                       // FOR LAB6 ONLY: the c
urrent stride of the process
```

```
    uint32_t lab6_priority;                     // FOR LAB6 ONLY: the p
riority of process, set by lab6_set_priority(uint32_t)
```

所以需要对 proc.c 中的 alloc_proc 函数进行如下修改（增加部分代码）：

```
        proc->rq=NULL;
        list_init(&(proc->run_link));
        proc->time_slice=0;
        proc->lab6_run_pool.left=proc->lab6_run_pool.right=proc->lab6_r
un_pool.parent=NULL;
        proc->lab6_stride=0;
        proc->lab6_priority=0;
```

②对于 kern/trap/trap.c 中的 trap_dispatch 函数：

根据注释，修改（增加）如下代码：

```
        /* LAB6 YOUR CODE */
        /* you should upate you lab5 code
         * IMPORTANT FUNCTIONS:
         * sched_class_proc_tick
         */
        ++ticks;
        sched_class_proc_tick(current);
```

③对于 kern/schedule/sched.c，需要将 sched_class_proc_tick 函数前面的 static 删除。

```
static void
sched_class_proc_tick(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->proc_tick(rq, proc);
    }
    else {
        proc->need_resched = 1;
    }
}
```

原因是 static 函数的作用域仅限于其目标文件的函数，这意味着静态函数仅在其目标文件 sched.c 中可见。

所以对应的，我们如果想要在 trap.c 中调用 sched_class_proc_tick 函数，就必须要把 static 修饰符删除，扩大其作用域。

④对于 kern/schedule/sched.h，由③易知我们在 sched.h 中补充对于 sched_class_proc_tick 函数的声明：

```
void sched_class_proc_tick(struct proc_struct *proc);
```

修改过后执行 make grade，结果如下所示（仅 priority 部分未通过测试，其他均通过，符合预期结果）：

```
priority:                 (11.3s)
  -check result:                                WRONG
   -e !! error: missing 'sched class: stride_scheduler'
   !! error: missing 'stride sched correct result: 1 2 3 4 5'

  -check output:                                OK
Total Score: 163/170
make: *** [grade] Error 1
```

在此基础上执行 make qemu：

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010c137 (phys)
  edata  0xc01addd4 (phys)
  end    0xc01b0f78 (phys)
Kernel executable memory footprint: 708KB
ebp:0xc012bf38 eip:0xc0100ae4 args:0x00010094 0x00000000 0xc012bf68 0xc01000d8
    kern/debug/kdebug.c:339: print_stackframe+21
ebp:0xc012bf48 eip:0xc0100dd3 args:0x00000000 0x00000000 0x00000000 0xc012bfb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc012bf68 eip:0xc01000d8 args:0x00000000 0xc012bf90 0xffff0000 0xc012bf94
    kern/init/init.c:59: grade_backtrace2+33
ebp:0xc012bf88 eip:0xc0100101 args:0x00000000 0xffff0000 0xc012bfb4 0x0000002a
    kern/init/init.c:64: grade_backtrace1+38
ebp:0xc012bfa8 eip:0xc010011f args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:69: grade_backtrace0+23
ebp:0xc012bfc8 eip:0xc0100144 args:0xc010c15c 0xc010c140 0x000031a4 0x00000000
    kern/init/init.c:74: grade_backtrace+34
ebp:0xc012bff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:33: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: RR_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
```

```
BEGIN check_swap: count 31849, total 31849
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
```

```
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, total is 5
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
main: fork ok,now need to wait pids.
child pid 3, acc 712000, time 1001
child pid 4, acc 720000, time 1002
child pid 5, acc 716000, time 1003
child pid 6, acc 708000, time 1003
child pid 7, acc 720000, time 1003
main: pid 3, acc 712000, time 1003
main: pid 4, acc 720000, time 1003
main: pid 5, acc 716000, time 1003
main: pid 6, acc 708000, time 1003
main: pid 7, acc 720000, time 1003
main: wait pids over
stride sched correct result: 1 1 1 1 1
```

```
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:434:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> [~/Desktop/lab6_final]
```

若不执行 make grade 直接 make qemu，则结果如下：

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010c111 (phys)
  edata  0xc01addd4 (phys)
  end    0xc01b0f78 (phys)
Kernel executable memory footprint: 708KB
ebp:0xc012bf38 eip:0xc0100ae4 args:0x00010094 0x00000000 0xc012bf68 0xc01000d8
    kern/debug/kdebug.c:339: print_stackframe+21
ebp:0xc012bf48 eip:0xc0100dd3 args:0x00000000 0x00000000 0x00000000 0xc012bfb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc012bf68 eip:0xc01000d8 args:0x00000000 0xc012bf90 0xffff0000 0xc012bf94
    kern/init/init.c:59: grade_backtrace2+33
ebp:0xc012bf88 eip:0xc0100101 args:0x00000000 0xffff0000 0xc012bfb4 0x0000002a
    kern/init/init.c:64: grade_backtrace1+38
ebp:0xc012bfa8 eip:0xc010011f args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:69: grade_backtrace0+23
ebp:0xc012bfc8 eip:0xc0100144 args:0xc010c13c 0xc010c120 0x000031a4 0x00000000
    kern/init/init.c:74: grade_backtrace+34
ebp:0xc012bff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:33: kern_init+84
memory management: default_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
```

```
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: RR_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
```

```
BEGIN check_swap: count 31849, total 31849
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
```

```
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, total is 5
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:434:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

练习 1：使用 Round Robin 调度算法（不需要编码）

完成练习 0 后，建议大家比较一下（可用 kdiff3、meld 等文件比较软件）个人完成的 lab5 和练习 0 完成后的刚修改的 lab6 之间的区别，分析了解 lab6 采用 RR 调度算法后的执行过程。

请在实验报告中完成：

请理解并分析 sched_class 中各个函数指针的用法，并结合 Round Robin 调度算法描述 ucore 的调度执行过程。

请在实验报告中简要说明如何设计实现“多级反馈队列调度算法”，给出概要设计，鼓励给出详细设计。

比较一下（可用 kdiff3、meld 等文件比较软件）个人完成的 lab5 和练习 0 完成后的刚修改的 lab6 之间的区别

答：具体区别已在练习 0 中体现，不再赘述。

请理解并分析 sched_class 中各个函数指针的用法，并结合 Round Robin 调度算法描述 ucore 的调度执行过程。

答：首先打开 kern/schedule/sched.h，并查看 sched_class 类：

```
// 调度类的引入借鉴了 Linux
// 使得核心调度器具有很强的可扩展性
// 这些类（调度程序模块）封装了调度策略
struct sched_class {
    // sched_class 的名称
    const char *name;
    // 初始化运行队列
    void (*init)(struct run_queue *rq);
    // 把 proc 放入 runqueue，这个函数必须用 rq_lock 调用
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // 将 proc 取出运行队列，并且必须使用 rq_lock 调用此函数
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // 选择下一个可运行的任务
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // dealer of the time-tick
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
    /* 用于未来的 SMP 支持
     *  load_balance
     *     void (*load_balance)(struct rq* rq);
     * 从这个 rq 中获取一些 proc，用于 load_balance
     *  返回值是得到的 proc 值
     *  int (*get_proc)(struct rq* rq, struct proc* procs_moved[]);
     */
};
```

①sched_init( )

```
//调度算法的初始化
void
sched_init(void) {
    list_init(&timer_list);

    sched_class = &default_sched_class;

    rq = &__rq;
    rq->max_time_slice = MAX_TIME_SLICE;
    sched_class->init(rq);

    cprintf("sched class: %s\n", sched_class->name);
}
```

②sched_class_enqueue( )

```
//将指定的进程状态置为 RUNNABLE 并将其加入就绪态队列
static inline void
sched_class_enqueue(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->enqueue(rq, proc);
    }
}
```

③sched_class_dequeue( )

```
//将调度算法选择的进程从就绪态队列中取出执行
static inline void
sched_class_dequeue(struct proc_struct *proc) {
    sched_class->dequeue(rq, proc);
}
```

④sched_class_pick_next( )

```
//选择要执行的下个进程
static inline struct proc_struct *
sched_class_pick_next(void) {
    return sched_class->pick_next(rq);
}
```

⑤sched_class_proc_tick( )

```
//每次产生时钟中断时执行该函数进行调度
static void
sched_class_proc_tick(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->proc_tick(rq, proc);
    }
    else {
        proc->need_resched = 1;
    }
}
```

关于 RR 调度算法：

RR 调度算法的调度思想是让所有 RUNNABLE 态的进程分时轮流使用 CPU 时间。

RR 调度器维护当前 RUNNABLE 进程的有序运行队列。当前进程的时间片用完之后，调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进

行调度。RR 调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块 proc_struct 中增加了一个成员变量 time_slice，用来记录进程当前的可运行时间片段。这是由于 RR 调度算法需要考虑执行进程的运行时间不能太长。在每个 timer 到时的时候，操作系统会递减当前执行进程的 time_slice，当 time_slice 为 0 时，就意味着这个进程运行了一段时间（这个时间片段称为进程的时间片），需要把 CPU 让给其他进程执行，于是操作系统就需要让此进程重新回到 rq 的队列尾，且重置此进程的时间片为就绪队列的成员变量最大时间片 max_time_slice 值，然后再从 rq 的队列头取出一个新的进程执行。

所以 ucore 的调度执行过程可概括为：

当线程因为 wait、exit、sleep 或时间片用完时，ucore 需要通过 schedule 函数进行调度。在 schedule 函数中，当需要进行线程切换时，我们需要调用 sched_class_enqueue 函数将其加入就绪队列，之后再调用 sched_class_pick_next 函数选择下一个要执行的线程，并同时将其从就绪队列中删除。

同时每当出现一个时钟中断时，我们需要将当前执行进程的剩余可执行时间 time_slice 减 1。当减为 0 时，则将其标记为可以被调度的，这样在中断服务例程中的后续部分就会调用 schedule 函数将该进程切换出去。

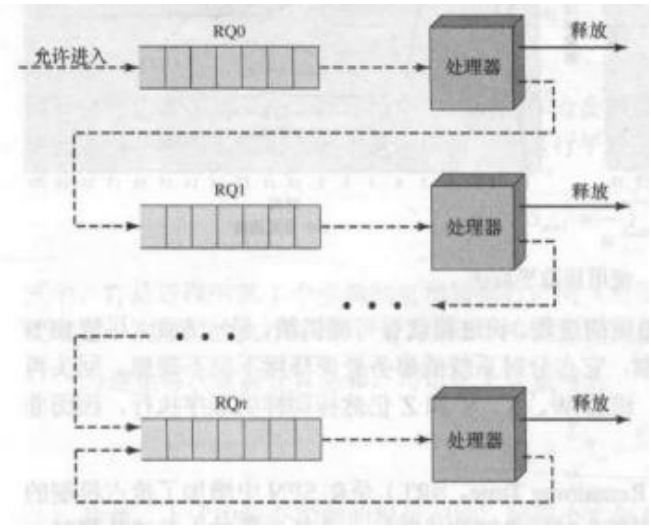
请在实验报告中简要说明如何设计实现“多级反馈队列调度算法”，给出概要设计，鼓励给出详细设计。

答：①对于不同优先级队列中的进程，我们优先调用优先级高的队列中的进程，若高优先级队列中不存在可调度的进程，则跳转至次优先级队列并从中选择进程来执行；

②对于同一队列中的进程，我们按照时间片轮转法（RR 算法）进行调度。例如，当处于 P1 队列（最高优先级队列）的作业在预定的时间片内未完成任务，则跳转至次优先级的 P2 队列中选择进程来执行（此时可把 P2 队列的时间片设置为 P1 队列时间片的 2 倍），若在预定的时间片内仍未完成任务，则跳转至再次一级优先级的 P3 队列中，以此类推，直至完成任务；

③当低优先级的队列正在运行进程时，若此时有新到达的高优先级的作业，则在该时间片结束后跳转至该新作业；

④同时为防止发生饥饿现象，我们可在算法中设置：每隔一段时间就将长时间未被选择执行的低优先级进程的优先级提高，同时也可选择将长时间占有 CPU 的的高优先级进程的优先级降低。



3. 实验感想

本次实验主要的目的是学习掌握操作系统的调度管理机制。由于老师删掉了一部分练习内容，所以工作量并不大。且调度管理机制已经在课上学习过，所以并没有感觉到很陌生。

这次实验碰到的主要困难反而是体现在练习 0 上，由于我不太清楚替换代码后应该进行怎样的修改才可以使得代码通过 lab6 的测试（尽管老师说了无需 make grade，但我本着能够进一步检测代码质量的目的，还是执行了 make grade 并在实验报告中附上了结果），所以在网上参考了一些实验报告，经过对比，我发现网上对于 trap.c 中 trap_dispatch 函数的修改主要有两种：

①使用 sched_class_proc_tick 函数：

```
        ++ticks;
        sched_class_proc_tick(current);
        break;
```

②使用 run_timer_list 函数：

```
        ticks ++;
        assert(current != NULL);
```

```
        run_timer_list();
        break;
```

我一开始没有看注释，选用了②方案，结果报错（提示我的程序中没有包含 run_timer_list 函数）：

```
obj/kern/trap/trap.o: In function `trap_dispatch':
kern/trap/trap.c:241: undefined reference to `run_timer_list'
make: *** [bin/kernel] Error 1
```

这时我才想起来去查看 sched.c 和 sched.h 确实没有这个函数的定义，然后我查看了虚拟机中的 moocos，发现里面的 sched.c 和 sched.h 比超算习堂上的对应文件多了一部分函数定义（即与 run_timer_list 函数相关的）。仔细一想，这可能也是老师有意而为之，让我们严格按照代码注释使用函数：

```
        /* LAB6 YOUR CODE */
        /* you should upate you lab5 code
         * IMPORTANT FUNCTIONS:
         * sched_class_proc_tick
         */
```

所以我改用①方案，但是发现仍然报错，提示没有 sched_class_proc_tick 函数：

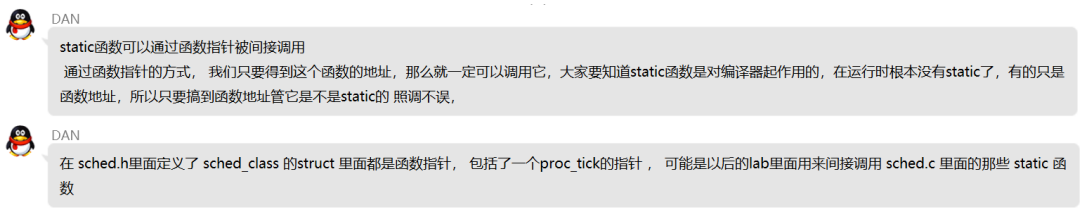
```
obj/kern/trap/trap.o: In function `trap_dispatch':
kern/trap/trap.c:237: undefined reference to `sched_class_proc_tick'
make: *** [bin/kernel] Error 1
```

但我明明在 sched.c 和 sched.h 中看到了 sched_class_proc_tick 函数的定义，为何显示 undefined reference？于是我在操统群里进行了匿名提问，经过一系列的讨论和黄聃老师的解答之后，我得到了一种解决办法（具体原因可见练习 0：主要是因为 static 函数的作用域仅限于本文件，所以导致 trap.c 的函数无法引用 sched.c 中的 static 函数），就是如练习 0 中我所解释的那样，在 sched.h 中单独声明该函数，并在 sched.c 中去掉该函数前面的 static 以扩大其作用域。修改之后再次 make，发现结果正确。

但是这个问题实际上还没有完美解决，黄聃老师也在群里说了我一开始的疑问，就是实际上 sched.h 中是有 sched_class_proc_tick 函数的函数指针的：

```
// The introduction of scheduling classes is borrrowed from Linux, and
makes the
// core scheduler quite extensible. These classes (the scheduler module
s) encapsulate
// the scheduling policies.
```

```
struct sched_class {
    // the name of sched_class
    const char *name;
    // Init the run queue
    void (*init)(struct run_queue *rq);
    // put the proc into runqueue, and this function must be called wit
h rq_lock
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // get the proc out runqueue, and this function must be called with
 rq_lock
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // choose the next runnable task
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // dealer of the time-tick
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
    /* for SMP support in the future
     *  load_balance
     *     void (*load_balance)(struct rq* rq);
     *  get some proc from this rq, used in load_balance,
     *  return value is the num of gotten proc
     *  int (*get_proc)(struct rq* rq, struct proc* procs_moved[]);
     */
};
```



按理说 static 函数应该是可以在其他文件中正常引用的，而无需像我这样在 sched.h 中增加函数声明，在 sched.c 中删除该函数前面的 static。在查阅了网上资料和对于 ucore lab6 的一些实验报告之后，我仍未找到合适的解决办法，最终只好采用了删除 static 以及添加函数声明的方法来完成这次实验。

如果助教方便的话，麻烦在这次实验报告的评价反馈里告诉我该问题的解决办法，谢谢！

除了这个问题以外，其余内容均在可接受范围内，顺利完成。