CS334 Assignment3

EX0 CPU Scheduling

Time	HRRN	FIFO/FCFS	RR	SJF	Priority
1	A	A	A	A	A
2	A	A	A	A	B
3	A	A	B	A	A
4	A	A	A	A	D
5	B	B	D	B	D
6	D	D	A	D	C
7	D	D	C	D	C
8	C	C	D	C	C
9	C	C	C	C	A
10	C	C	C	C	A
Avg. Turn-around Time	4.5	4.5	4.75	4.5	4.25

EX1

设计思想主要是层层递进找到最基本的应该添加方法的地方，添加 `set_priority` 方法实现功能，有点面向头文件编程的意思，逐步找到最底层，然后从底向上编程。

以下是修改代码的截图：

最底层是在 `proc.c` 当中，添加 `set_proc_priority()` 函数

```
// set_proc_priority - set the priority of proc
int set_proc_priority(int pri){
    current->labschedule_priority = pri;
    cprintf("set priority to %d\n", pri);
    return pri;
}
```

在这之后，找到需要调用此函数的位置，是在 `kern/syscall/syscall.c` 当中，模仿上下函数的模式，实现自己的函数，截图如下。

```c
static int
sys_set_priority(uint64_t arg[]){
    int pri = (int)arg[0];
    return set_proc_priority(pri);
}
```

之后，发现前后的函数都定义在一张函数表`static int (*syscalls[])(uint64_t arg[]`当中，于是添加表项`SYS_labschedule_set_priority`

```c
static int (*syscalls[])(uint64_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_getpid]            sys_getpid,
    [SYS_putc]              sys_putc,
    [SYS_gettime]           sys_gettime,
    [SYS_labschedule_set_priority] sys_set_priority,
};
```

之后，找到调用 `syscalls`的地方，是同一文件当中的`syscall`，继续追踪，发现是需要从 `libs/syscall.c`调用`syscall`来使用`sys_set_priority`方法，于是在里面再次封装一层调用

```c
int
sys_syscall_set_priority(int64_t pri){

    return syscall(SYS_labschedule_set_priority, pri);
}
```

最后一层是`ulibs`中的调用，直接提供给 `ex1.c`使用

```c
int
set_priority(int priority){
    return sys_syscall_set_priority(priority);
}
```

最终运行截图如下，和 PDF截图一致。

```
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08800000, [0x80200000, 0x885fffff].
sched class: RR_scheduler
SWAP: manager = fifo swap manager
setup timer interrupts
The next proc is pid:1
The next proc is pid:2
kernel_execve: pid = 2, name = "ex1".
Breakpoint

-------ex1---start------
set priority to 5
-------ex1----end-------

The next proc is pid:1
all user-mode processes have quit.
The end of init_main
kernel panic at kern/process/proc.c:426:
    initproc exit.

lmq@lmq-virtual-machine:~/Desktop/Assignment3$ 
```

EX2

设计思想：先找到与调度相关的函数 `default_sched.c`，在里面找到 `RR` 算法对应的 `enqueue` 方法：`RR_enqueue`，之后修改入队时的进程的时间片信息，修改完之后，对每个进程，都打印相关的时间片信息。

修改的代码截图如下：

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {


    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice * proc -> labschedule_priority;
    }
    cprintf("pid:%d 's time slice is %d\n", proc -> pid, proc -> time_slice);
    proc->rq = rq;
    rq->proc_num ++;
}
```

运行的截图如下：

```
pid:6 's time slice is 25
The next proc is pid:7
pid:7 's time slice is 10
The next proc is pid:3
pid:3 's time slice is 15
The next proc is pid:4
pid:4 's time slice is 5
The next proc is pid:5
pid:5 's time slice is 20
The next proc is pid:6
pid:6 's time slice is 25
The next proc is pid:7
pid:7 's time slice is 10
The next proc is pid:3
pid:3 's time slice is 15
The next proc is pid:4
pid:4 's time slice is 5
The next proc is pid:5
pid:5 's time slice is 20
The next proc is pid:6
pid:6 's time slice is 25
The next proc is pid:7
```

通过检查，进程运行结束的顺序是 6,5,3,7,4；符合`ex2`当中的运行优先级。

EX3

设计思路以及代码截图：

首先在`kern/init/init.c`中禁用时钟中断

```c
int
kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    cons_init();                // init the console

    const char *message = "OS is loading ...";
    cprintf("%s\n\n", message);

    pmm_init();                 // init physical memory management

    idt_init();                 // init interrupt descriptor table

    vmm_init();                 // init virtual memory management
    sched_init();
    proc_init();                // init process table

    ide_init();                 // init ide devices
    swap_init();                // init swap

    // clock_init();               // init clock interrupt
    intr_enable();              // enable irq interrupt

    cpu_idle();                 // run idle process
}
```

之后，添加一个`syscall`定义

```c
/*only for labschedule*/
#define SYS_labschedule_set_priority 255
#define SYS_labschedule_set_good 256
```

之后，在`syscalls`当中添加一个表项

```
[SYS_labschedule_set_priority] sys_set_priority,
[SYS_labschedule_set_good] sys_set_good,
```

之后实现 `sys_set_good`，和第一题类似，只不过需要封装的层数没有那么多

```
static int
sys_set_good(uint64_t arg[]){
    int good = (int)arg[0];
    return set_proc_good(good);
}
```

之后，在`proc.c`当中实现 `set_proc_good()`方法

```
int set_proc_good(int good){
    current -> labschedule_good = good;
    cprintf("set pid: %d 's good to %d\n", current -> pid, good);
    schedule();
    return 0; // represents success.
}
```

在实现了设置 `good`的`syscall`之后，接下来描述根据 `good`的抢占式进程调度的实现

这个抢占式实现的主要思路是根据每一个进程`proc`的`good`值不断交换他们在`RR_queue`当中的顺序。

在这里理解 `list_add_before`比较重要，因为 `RR`算法是基于双向循环链表实现的，所以在这里将新的进程插入`rq`的前面，就相当于插入在了链表的最后方。

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {


    list_add_before(&(rq->run_list), &(proc->run_link));
    list_entry_t *tmp = &(proc -> run_link);
    // propagation until the head of the running queue.
    while(tmp -> prev != &(rq -> run_list)){
        // To get the good value of the process, need to convert the run_list to process struct.
        struct proc_struct *prev =  le2proc(tmp -> prev, run_link);
        struct proc_struct *curr = le2proc(tmp, run_link);
        //when the good value of the previous proc < that of curr, their orders should be exchange.
        if(prev -> labschedule_good < curr -> labschedule_good){
            list_entry_t * tmp_prev = tmp -> prev;
            tmp_prev -> prev -> next = tmp;
            tmp -> prev = tmp_prev -> prev;
            tmp -> next -> prev = tmp_prev;
            tmp_prev -> next = tmp -> next;
            tmp -> next = tmp_prev;
            tmp_prev -> prev = tmp;
        }

    }
```

运行的结果截图如下所示：

```
main: fork ok,now need to wait pids.
The next proc is pid:3
Set good to be 3.
The next proc is pid:4
Set good to be 1.
The next proc is pid:5
Set good to be 4.
The next proc is pid:6
Set good to be 5.
The next proc is pid:7
Set good to be 2.
The next proc is pid:6
child pid 6, acc 4000001
The next proc is pid:2
The next proc is pid:5
Set good to be 4.
child pid 5, acc 4000001
The next proc is pid:2
The next proc is pid:3
Set good to be 3.
child pid 3, acc 4000001
The next proc is pid:2
The next proc is pid:7
child pid 7, acc 4000001
The next proc is pid:2
The next proc is pid:4
child pid 4, acc 4000001
The next proc is pid:2
main: wait pids over
The next proc is pid:1
all user-mode processes have quit.
The end of init_main
kernel panic at kern/process/proc.c:426:
    initproc exit.
```