

LAB6：调度器

白欣雨 PB18051183

LAB6：调度器

实验目的

实验内容

实验过程

练习0：填写已有实验

练习1：使用 Round Robin 调度算法

代码分析

练习2：实现Stride Scheduling调度算法

算法简述

代码实现

Challenge 1：实现Linux的CFS调度算法

算法分析

代码实现

CFS验证

运行结果

感想与总结

相关知识点

参考资料

实验目的

- 理解操作系统的调度管理机制；
- 熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法；
- 基于调度器框架实现一个(Stride Scheduling)调度算法来替换缺省的调度算法。

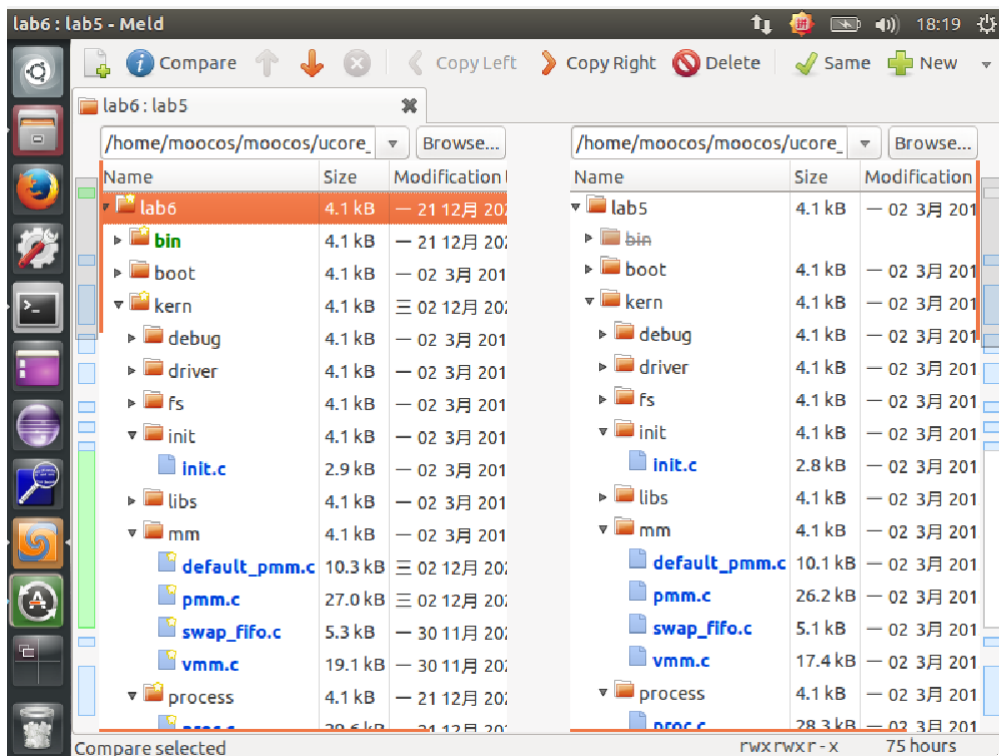
实验内容

实验五完成了用户进程的管理，可在用户态运行多个进程。但到目前为止，采用的调度策略是很简单的FIFO调度策略。本次实验，主要是熟悉ucore的系统调度器框架，以及基于此框架的Round-Robin (RR)调度算法。然后参考RR调度算法的实现，完成Stride Scheduling调度算法。

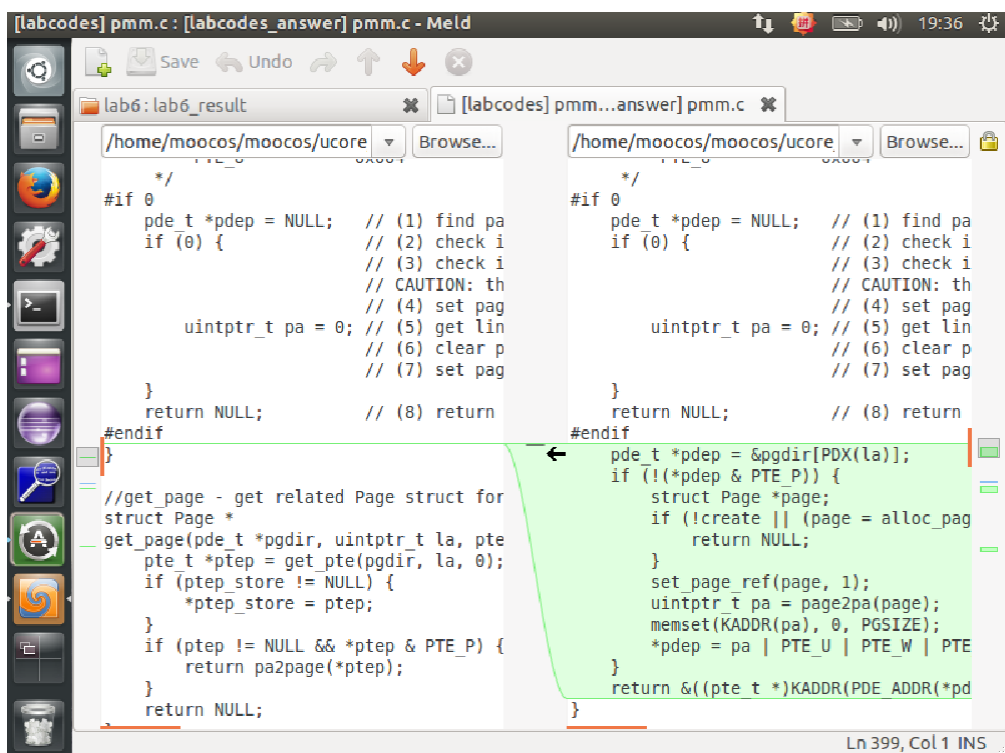
实验过程

练习0：填写已有实验

此练习需要把LAB6中的LAB1-LAB5缺省处补齐，本练习继续使用meld图形化方式进行合并，选择两个目录进行比较：



然后根据之前实验的内容，需要修改的文件有：proc.c、default_pmm.c、pmm.c、swap_fifo.c、vmm.c、trap.c。以其中一个文件为例进行合并：



如图所示，点击箭头即可将绿色的代码部分添加到相应部分。对其余文件作同样处理，确保编译可以通过。

除此以外，根据实验要求，我们还需要对部分的代码进行改进：

检查代码知，在/kern/process/proc.h中定义的结构体：

```
struct proc_struct {
    enum proc_state state;           // Process state
    int pid;                         // Process ID
    int runs;                        // the running times of Proces
    uintptr_t kstack;               // Process kernel stack
```

```

    volatile bool need_resched;           // bool value: need to be
rescheduled to release CPU?
    struct proc_struct *parent;           // the parent process
    struct mm_struct *mm;                 // Process's memory management
field
    struct context context;               // Switch here to run process
    struct trapframe *tf;                 // Trap frame for current
interrupt
    uintptr_t cr3;                        // CR3 register: the base addr
of Page Directroy Table(PDT)
    uint32_t flags;                       // Process flag
    char name[PROC_NAME_LEN + 1];         // Process name
    list_entry_t list_link;               // Process link list
    list_entry_t hash_link;              // Process hash list
    int exit_code;                        // exit code (be sent to parent
proc)
    uint32_t wait_state;                  // waiting state
    struct proc_struct *cptr, *yptr, *optr; // relations between processes
    struct run_queue *rq;                 // running queue contains
Process
    list_entry_t run_link;                 // the entry linked in run queue
    int time_slice;                       // time slice for occupying the
CPU
    skew_heap_entry_t lab6_run_pool;      // FOR LAB6 ONLY: the entry in
the run pool
    uint32_t lab6_stride;                 // FOR LAB6 ONLY: the current
stride of the process
    uint32_t lab6_priority;               // FOR LAB6 ONLY: the priority
of process, set by lab6_set_priority(uint32_t)
};

```

其中新增加了9行定义，根据注释我们了解其作用：

```

    int exit_code;                        // 退出码（发送到父进程）
    uint32_t wait_state;                  // 等待状态
    struct proc_struct *cptr, *yptr, *optr; // 进程间的关系
    struct run_queue *rq;                 // 运行队列中包含进程
    list_entry_t run_link;                 // 进程的调度链表结构
    int time_slice;                       // 进程剩余的时间片
    skew_heap_entry_t lab6_run_pool;      // 进程在优先队列中的节点
    uint32_t lab6_stride;                 // 进程的调度步进值
    uint32_t lab6_priority;               // 进程的调度优先级
};

```

因此我们找到其初始化的函数alloc_proc，对其增加一些初始化（也有注释提示）：

```

static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
    }
}

```

```

proc->mm = NULL;
memset(&(proc->context), 0, sizeof(struct context));
proc->tf = NULL;
proc->cr3 = boot_cr3;
proc->flags = 0;
memset(proc->name, 0, PROC_NAME_LEN);
proc->wait_state = 0;
proc->cptr = proc->optr = proc->yptr = NULL;

/*以下为新增部分*/
proc->rq = NULL; //初始化运行队列为空
list_init(&(proc->run_link)); //初始化运行队列的指针
proc->time_slice = 0; //初始化时间片
proc->lab6_run_pool.left = proc->lab6_run_pool.right proc-
>lab6_run_pool.parent = NULL; //初始化各类指针为空，包括父进程等待
proc->lab6_stride = 0; //步数初始化
proc->lab6_priority = 0; //初始化优先级
}
return proc;
}

```

以及trap_dispatch函数，根据注释的提示得到：

```

static void
trap_dispatch(struct trapframe *tf) {
    char c;
    int ret=0;
    switch (tf->tf_trapno) {
    case T_PGFLT: //page fault
        if ((ret = pgfault_handler(tf)) != 0) {
            print_trapframe(tf);
            if (current == NULL) {
                panic("handle pgfault failed. ret=%d\n", ret);
            }
            else {
                if (trap_in_kernel(tf)) {
                    panic("handle pgfault failed in kernel mode. ret=%d\n",
ret);
                }
                cprintf("killed by kernel.\n");
                panic("handle user mode pgfault failed. ret=%d\n", ret);
                do_exit(-E_KILLED);
            }
        }
        break;
    case T_SYSCALL:
        syscall();
        break;
    case IRQ_OFFSET + IRQ_TIMER:

        //修改部分
        ticks ++;
        assert(current != NULL);
        run_timer_list(); //更新定时器，并根据参数调用调度算法
        break;

    case IRQ_OFFSET + IRQ_COM1:

```

```

        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        print_trapframe(tf);
        if (current != NULL) {
            cprintf("unhandled trap.\n");
            do_exit(-E_KILLED);
        }
        panic("unexpected trap in kernel.\n");
    }
}

```

编译通过

```

[~/moocos/ucore_lab/labcodes/lab6]
moocos-> make
+ cc kern/trap/trap.c
kern/trap/trap.c:22:13: warning: 'print_ticks' defined but not used [-Wunused-function]
static void print_ticks() {
        ^
+ ld bin/kernel
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0406105 s, 126 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000814614 s, 629 kB/s
1447+1 records in
1447+1 records out
741147 bytes (741 kB) copied, 0.00354847 s, 209 MB/s

```

练习1：使用 Round Robin 调度算法

Round Robin 调度算法的调度思想是让所有就绪态的进程分时轮流使用 CPU 时间。Round Robin 调度器维护当前就绪状态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。

代码分析

在分析各个函数之前, 首先分析函数中出现的变量含义以及结构体定义, 以便更好地了解程序运行的过程, 例如RR_init函数中的run_queue结构体:

```

struct run_queue {
    list_entry_t run_list;           //哨兵，可以看作是队列的头与尾
    unsigned int proc_num;           //队列内程序数
    int max_time_slice;              //每个进程被分配的时间片
    // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool;
};

```

而skew_heap_entry_t也是一个结构体，其为一个进程池，查看其定义知其数据结构为树，是一个树形进程池：

```

struct skew_heap_entry{
    struct skew_heap_entry *parent,*left,*right;
};
typedef struct skew_heap_entry skew_heap_entry_t

```

具体分析算法见代码的注释：

```

static void
RR_init(struct run_queue *rq) {           //队列初始化
    list_init(&(rq->run_list));
    rq->proc_num = 0;                      //进程数量
}

/*RR_enqueue: 设置进程状态，并且放入调用算法中的可执行队列中。先把进程的进程控制块指针放入到rq
队列末尾，若进程控制块的时间片为0，则需把它重置为max_time_slice。这表示如果进程在当前的执行时
间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加
一。*/
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link)); //进程控制块指针放到rq队尾
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) { //判断进
程控制块的时间片为0或者进程时间片大于最大时间片
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq; //添加到进程队列
    rq->proc_num ++; //就绪态进程数+1
}

/*RR_dequeue 函数把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的
proc_num减1。即取出队列中的一个进程开始执行。 */
static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link)); //将进程控制块指针从队列中删除
    rq->proc_num --; //就绪进程数-1
}

/*RR_pick_next函数主要作用选择要执行的下一个程序，是用于选取就绪进程队列rq中的队头队列元素，并
把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。*/
static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list)); //选取就绪态队列队首元素
    if (le != &(rq->run_list)) { //获取其进程，成功则返回进程控制块指针
        return le2proc(le, run_link);
    }
}

```

```

    }
    return NULL;
}

/*RR_proc_tick函数 此函数为时钟中断时需要调用的调度算法*/

static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --; //执行进程的时间片时间-1
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1; //若时间片为0, 则设置need_resched为1说明需要调度
    }
}

/*定义一个C语言类, 提供调度算法的切换接口*/
struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};

```

- 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;

sched_class类的定义位于kern/schedule/sched.h中, 代码如下:

```

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 with
    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[]);
    */
};

```

根据注释易得:

```

struct sched_class {
    const char *name;           //调度器名字
    void (*init)(struct run_queue *rq);    //初始化运行队列
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    //将进程插入运行队列中
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    //将进程从运行队列中删除
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    //返回运行队列下一个可执行进程
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
    //timetick处理
};

```

结合具体算法，简单描述调度过程：

- 在ucore中调用调度器的主体函数的有wakeup_proc函数和schedule函数。wakeup_proc的作用时将一个指定进程放入可执行进程队列中；schedule函数的作用为将当前执行的进程放入可执行队列中，然后将队列中下一个可执行的进程取出并执行。
- 将某一个进程加入就绪进程队列中，需要将这个进程的能够使用的时间片进行初始化，然后将其插入到使用链表组织的队列的对尾；而将某一个进程从就绪队列中取出的时候，将其直接删除即可。
- 当需要取出执行的下一个进程的时候，将就绪队列的队头取出，插入到运行队列中。
- 每当出现一个时钟中断，将当前执行的进程的剩余可执行时间-1，当其减到了0时，将其标记为可以被调度的，这样就会调用schedule函数将这个进程切换出去。

- 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。

概要设计思路如下：

- 在proc_struct中添加总共N个多级反馈队列的入口，每个队列都有着各自的优先级，编号越大的队列优先级越低，并且优先级越低的队列上时间片的长度越大，为其上一个优先级队列的两倍；并且在PCB中记录当前进程所处的队列的优先级；
- 处理调度算法初始化的时候需要同时对N个队列进行初始化；
- 在处理将进程加入到就绪进程集合的时候，观察这个进程的时间片有没有使用完，如果使用完了，就将所在队列的优先级调低，加入到优先级低1级的队列中去，如果没有使用完时间片，则加入到当前优先级的队列中去；
- 在同一个优先级的队列内使用时间片轮转算法；
- 在选择下一个执行的进程的时候，有限考虑高优先级的队列中是否存在任务，如果不存在才转而寻找较低优先级的队列；
- 从就绪进程集合中删除某一个进程就只需要在对应队列中删除即可；处理时间中断的函数不需要改变；

练习2：实现Stride Scheduling调度算法

算法简述

本实验要求完成实现Stride Scheduling调度算法，查阅指导书给出的链接以及相关书籍资料，实验开始之前先简述一下Stride Scheduling调度算法：

- 维护两个变量stride与pass，程序为每个进程设置一个stride状态，pass代表进程调度以后，需要增加的值（步长），即每次进程被调度的时候： $stride = stride + pass$ 。其中，pass的值与优先级有关， $pass = BIG_STRIDE / \text{优先级}$ ，因此我们通过设置优先级，可以给该进程设置好对应的步长，也实现了优先级越高越容易被调度的思想。
- 每次调度时，从进程中选取出当前stride最小的进程，作为获得CPU的对象，并将其对应的stride值增加pass。

代码实现

根据指导书的要求，我们要完成kern/schedule/default_sched_stride.c文件：

```
#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

#define USE_SKEW_HEAP 1

/* You should define the BigStride constant here*/
/* LAB6: YOUR CODE */
#define BIG_STRIDE    /* you should give a value, and is ??? */

/* The compare function for two skew_heap_node_t's and the
 * corresponding procs*/
static int
proc_stride_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool);
    struct proc_struct *q = le2proc(b, lab6_run_pool);
    int32_t c = p->lab6_stride - q->lab6_stride;
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}

/*
 * stride_init initializes the run-queue rq with correct assignment for
 * member variables, including:
 *
 * - run_list: should be a empty list after initialization.
 * - lab6_run_pool: NULL
 * - proc_num: 0
 * - max_time_slice: no need here, the variable would be assigned by the
 * caller.
 *
 * hint: see libs/list.h for routines of the list structures.
 */
static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE
     * (1) init the ready process list: rq->run_list
     * (2) init the run pool: rq->lab6_run_pool
     * (3) set number of process: rq->proc_num to 0
     */
}

/*
 * stride_enqueue inserts the process ``proc'' into the run-queue
 * ``rq''. The procedure should verify/initialize the relevant members
 * of ``proc'', and then put the ``lab6_run_pool'' node into the
 * queue(since we use priority queue here). The procedure should also
 * update the meta data in ``rq'' structure.
 */
```

```

* proc->time_slice denotes the time slices allocation for the
* process, which should set to rq->max_time_slice.
*
* hint: see libs/skew_heap.h for routines of the priority
* queue structures.
*/
static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE
    * (1) insert the proc into rq correctly
    * NOTICE: you can use skew_heap or list. Important functions
    *         skew_heap_insert: insert a entry into skew_heap
    *         list_add_before: insert a entry into the last of list
    * (2) recalculate proc->time_slice
    * (3) set proc->rq pointer to rq
    * (4) increase rq->proc_num
    */
}

/*
* stride_dequeue removes the process ``proc'' from the run-queue
* ``rq'', the operation would be finished by the skew_heap_remove
* operations. Remember to update the ``rq'' structure.
*
* hint: see libs/skew_heap.h for routines of the priority
* queue structures.
*/
static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE
    * (1) remove the proc from rq correctly
    * NOTICE: you can use skew_heap or list. Important functions
    *         skew_heap_remove: remove a entry from skew_heap
    *         list_del_init: remove a entry from the list
    */
}

/*
* stride_pick_next pick the element from the ``run-queue'', with the
* minimum value of stride, and returns the corresponding process
* pointer. The process pointer would be calculated by macro le2proc,
* see kern/process/proc.h for definition. Return NULL if
* there is no process in the queue.
*
* When one proc structure is selected, remember to update the stride
* property of the proc. (stride += BIG_STRIDE / priority)
*
* hint: see libs/skew_heap.h for routines of the priority
* queue structures.
*/
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: YOUR CODE
    * (1) get a proc_struct pointer p with the minimum value of stride
    *     (1.1) If using skew_heap, we can use le2proc get the p from rq-
    >lab6_run_poll
    *     (1.2) If using list, we have to search list to find the p with
    minimum stride value
    * (2) update p;s stride value: p->lab6_stride

```

```

        * (3) return p
        */
    }

/*
 * stride_proc_tick works with the tick event of current process. You
 * should check whether the time slices for current process is
 * exhausted and update the proc struct ``proc''. proc->time_slice
 * denotes the time slices left for current
 * process. proc->need_resched is the flag variable for process
 * switching.
 */
static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
}

struct sched_class default_sched_class = {
    .name = "stride_scheduler",
    .init = stride_init,
    .enqueue = stride_enqueue,
    .dequeue = stride_dequeue,
    .pick_next = stride_pick_next,
    .proc_tick = stride_proc_tick,
};

```

下面根据注释，逐一介绍各个函数的功能与实现思路：

1.proc_stride_comp_f

根据函数名和注释，易知其功能为比较两个进程的优先级，比较的方法是两个进程的权值相减，根据正负情况判断大小。

```

static int
proc_stride_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool); //获取一个进程
    struct proc_struct *q = le2proc(b, lab6_run_pool); //获取一个进程
    int32_t c = p->lab6_stride - q->lab6_stride;          //stride相减
    if (c > 0) return 1;                                   //判断不同的情况
    else if (c == 0) return 0;
    else return -1;
}

```

2.stride_init

根据注释，易知其为初始化函数，主要要初始化run_list、lab6_run_pool、proc_num三个变量，因此很容易补全代码实现：

```

static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
    list_init(&(rq->run_list)); //初始化调度器类
    rq->lab6_run_pool = NULL;   //初始化优先级队列
    rq->proc_num = 0;           //队列目前为空
}

```

3.stride_enqueue

根据函数名称与注释，易知其作用是将进程加入到运行队列，初始化刚进入运行队列的进程的stride属性，然后比较队首进程与当前进程的步数大小，选择步数最小的运行，即将其插入放入运行队列中去，然后初始化时间片，并将运行队列进程数目加1。

```
static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);
    #else
        assert(list_empty(&(proc->run_link)));
        list_add_before(&(rq->run_list), &(proc->run_link));
    #endif
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;          //更新进程的就绪队列
    rq->proc_num ++;        //进程数+1
}
```

4.stride_dequeue

根据注释，此函数为出队函数stride_dequeue，将一个进程从队列中移除，然后运行队列数目减1。实现本函数时，调用了skew_heap_remove函数进行移除指定进程的操作。

```
static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool =
            skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
    #else
        assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
        list_del_init(&(proc->run_link));
    #endif
    rq->proc_num --; //队列进程数-1
}
```

5.stride_pick_next

根据注释与函数名称，易知其主要作用是选择下一个要执行的进程，过程大概为：先遍历整个运行队列，返回其中stride值最小的对应进程，然后更新该进程的stride值：如果优先级为0，则设置步长为最大值与stride值相加；如果优先级不为0，则设置步长值为然后将步长设置为步长最大值除以优先级，然后相加。

```
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
```

```

    /* LAB6: YOUR CODE */
#ifdef USE_SKEW_HEAP
    if (rq->lab6_run_pool == NULL) return NULL;
    //选择权值最小的进程:
    struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);
#else
    list_entry_t *le = list_next(&(rq->run_list));

    if (le == &rq->run_list)
        return NULL;

    struct proc_struct *p = le2proc(le, run_link);
    le = list_next(le);
    while (le != &rq->run_list)
    {
        struct proc_struct *q = le2proc(le, run_link);
        if ((int32_t)(p->lab6_stride - q->lab6_stride) > 0)
            p = q;
        le = list_next(le);
    }
#endif
    //分情况更新进程的stride值
    if (p->lab6_priority == 0)
        p->lab6_stride += BIG_STRIDE;
    else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}

```

6.stride_proc_tick

根据注释，此函数为时间片函数stride_proc_tick，其主要作用是检查当前进程的时间片是否已用完，即变为0。如果时间片用完，应该设置相应的标记，表示进程需要切换。

```

static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    if (proc->time_slice > 0) { //如果还有剩余时间片
        proc->time_slice--;
    }
    if (proc->time_slice == 0) { //如果进程时间片为0
        proc->need_resched = 1; //需要被调度
    }
}

```

7.sched_class类

该部分为一个C语言类，主要是提供调度的接口，与Round Robin算法实现类似。

```

struct sched_class default_sched_class = {
    .name = "stride_scheduler",
    .init = stride_init,
    .enqueue = stride_enqueue,
    .dequeue = stride_dequeue,
    .pick_next = stride_pick_next,
    .proc_tick = stride_proc_tick,
};

```

综上所述，可以得到完整代码：

```
#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

#define USE_SKEW_HEAP 1
#define BIG_STRIDE 0x7FFFFFFF

static int
proc_stride_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool);
    struct proc_struct *q = le2proc(b, lab6_run_pool);
    int32_t c = p->lab6_stride - q->lab6_stride;
    if (c > 0) return 1;
    else if (c == 0) return 0;
    else return -1;
}

static void
stride_init(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
    list_init(&(rq->run_list));
    rq->lab6_run_pool = NULL;
    rq->proc_num = 0;
}

static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);
    #else
        assert(list_empty(&(proc->run_link)));
        list_add_before(&(rq->run_list), &(proc->run_link));
    #endif
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}

static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    #if USE_SKEW_HEAP
        rq->lab6_run_pool =
            skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
    #else
        assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
        list_del_init(&(proc->run_link));
    #endif
}
```

```

#endif
    rq->proc_num --;
}

static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
    /* LAB6: YOUR CODE */
#ifdef USE_SKEW_HEAP
    if (rq->lab6_run_pool == NULL) return NULL;
    struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);
#else
    list_entry_t *le = list_next(&(rq->run_list));

    if (le == &rq->run_list)
        return NULL;

    struct proc_struct *p = le2proc(le, run_link);
    le = list_next(le);
    while (le != &rq->run_list)
    {
        struct proc_struct *q = le2proc(le, run_link);
        if ((int32_t)(p->lab6_stride - q->lab6_stride) > 0)
            p = q;
        le = list_next(le);
    }
#endif
    //更新对应进程的stride值
    if (p->lab6_priority == 0)
        p->lab6_stride += BIG_STRIDE;
    else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    return p;
}

static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    /* LAB6: YOUR CODE */
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}

struct sched_class default_sched_class = {
    .name = "stride_scheduler",
    .init = stride_init,
    .enqueue = stride_enqueue,
    .dequeue = stride_dequeue,
    .pick_next = stride_pick_next,
    .proc_tick = stride_proc_tick,
};

```

完成代码时出现了一些错误，与答案对照后发现是初始化的问题，对一些函数的理解不是很到位，但是大部分实现的思路与标准答案没什么区别，按照注释给出的思路与提示完成就行了。

至此LAB6的主要实验内容已经完成，下面是实现拓展部分。

Challenge 1: 实现Linux的CFS调度算法

算法分析

在实现算法之前，我们先了解一下CFS（Completely Fair Scheduler），CFS背后的主要想法是维护为任务提供处理器时间方面的平衡以保证公平性。这意味着应给每个进程分配适当次数的CPU。分给某个任务的时间失去平衡时，应给失去平衡的任务分配运行时间。

要实现平衡，cfs定义了一种新的模型，它给cfs_rq（cfs的run queue）中的每一个进程安排一个虚拟时钟，即vruntime。如果一个进程得以执行，随着时间的增长，其vruntime将不断增大。没有得到执行的进程vruntime不变。而调度器总是选择vruntime跑得最慢的那个进程来执行。这就是所谓的“完全公平”。为了区别不同的进程，优先级高的进程vruntime增长得慢，以至于它可能得到更多的运行机会。

代码实现

按照算法分析，首先应该记录进程的运行时间、优先级以及堆，因此在定义进程的结构体中应该加入记录相关内容的变量：

```
struct proc_struct {
    enum proc_state state;           // Process state
    int pid;                         // Process ID
    int runs;                        // the running times of Proces
    uintptr_t kstack;                // Process kernel stack
    volatile bool need_resched;
    struct proc_struct *parent;      // the parent process
    struct mm_struct *mm;            // Switch here to run process
    struct context context;
    struct trapframe *tf;
    uintptr_t cr3;
    uint32_t flags;
    char name[PROC_NAME_LEN + 1];    // Process name
    list_entry_t list_link;
    list_entry_t hash_link;
    int exit_code;
    uint32_t wait_state;
    struct proc_struct *cptr, *yptr, *optr;
    struct run_queue *rq;
    list_entry_t run_link;
    int time_slice;
    skew_heap_entry_t lab6_run_pool;
    uint32_t lab6_stride;
    uint32_t lab6_priority;

    int CFS_run_time;
    int CFS_priority;
    skew_heap_entry_t CFS_run_pool;
};
```

按照查找的资料来看，我们应该选择使用红黑树这个数据结构，但是由于LAB6的代码已经定义好了一个堆，为方便起见，我们使用已经定义好的堆来记录每个进程的运行时间。


```

struct run_queue {
    list_entry_t run_list;           //哨兵，可以看作是队列的头与尾
    unsigned int proc_num;           //队列内程序数
    int max_time_slice;              //每个进程被分配的时间片
    // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool;
    skew_heap_entry_t *CFS_run_pool;
};

```

然后我们需要比较当前进程与其他进程运行时间的比较函数以及初始化堆的函数，仿照LAB6实验内容定义的函数：

CFS_comp_f

```

static int
CFS_comp_f(void *a, void *b)
{
    struct proc_struct *p = le2proc(a, lab6_run_pool); //获取一个进程
    struct proc_struct *q = le2proc(b, lab6_run_pool); //获取一个进程
    int32_t c = p->CFS_run_time - q->CFS_run_time;      //运行时间相减
    if (c > 0) return 1;                                //判断不同的情况
    else if (c == 0) return 0;
    else return -1;
}

```

CFS_init

```

static void
CFS_init(struct run_queue *rq) {
    rq->CFS_run_pool = NULL;           //初始化队列
    rq->proc_num = 0;                  //队列目前为空
}

```

对于将进程加入与移除运行队列的函数，由于不需要比较stride，因此同样在原函数上进行修改即可：

CFS_enqueue

```

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

    rq->CFS_run_pool = skew_heap_insert(rq->CFS_run_pool, &(amp;proc->CFS_run_pool), CFS_comp_f);
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;                      //更新进程的就绪队列
    rq->proc_num ++;                    //进程数+1
}

```

CFS_dequeue

```
static void
CFS_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    rq->CFS_run_pool =
        skew_heap_remove(rq->CFS_run_pool, &(proc->CFS_run_pool), CFS_comp_f);
    rq->proc_num --;
}
```

在遍历整个堆的时候，选择的目标也不再是stride值，而是运行时间，因此修改选择函数：

CFS_pick_next

```
static struct proc_struct *
CFS_pick_next(struct run_queue *rq) {

    if (rq->CFS_run_pool == NULL) return NULL;

    struct proc_struct *p = le2proc(rq->CFS_run_pool, CFS_run_pool);
    return p;
}
```

对于检查时间片函数，根据算法原理的分析，本算法的时间片不是一成不变的，而是要根据优先级系数进行变化，可以得到：

CFS_proc_tick

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

    if (proc->time_slice > 0) {
        proc->time_slice --;
        proc->CFS_run_time = proc->CFS_run_time + proc->CFS_priority;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

主要函数已经完成。但为了测试运行情况，还应当对优先级的定义进行处理：

```
void lab6_set_priority(uint32_t priority){
    if (priority == 0)
        current->lab6_priority = 1;
    else {
        current->CFS_priority = 60 / current->CFS_priority + 1;
        if(current->CFS_priority < 1)
            current->CFS_priority = 1;
    }
}
```

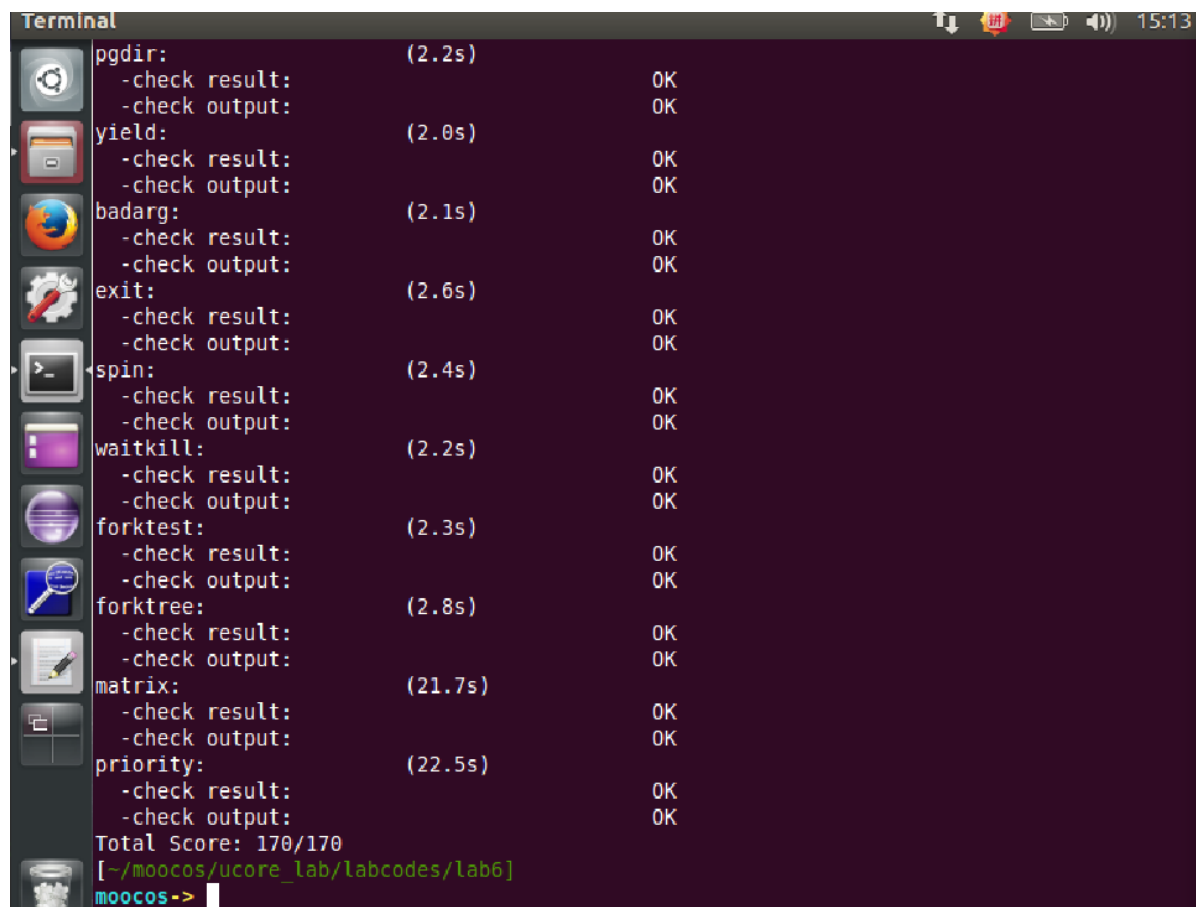
```
int do_yield(void){
    current->need_resched = 1;
    current->CFS_run_time = current->rq->max_time_slice * current->CFS_priority;
    return 0;
}
```

最后更改一下接口定义，由于是在lab6基础上进行修改，所以.name保留：

```
struct sched_class default_sched_class = {
    .name = "stride_scheduler",
    .init = CFS_init,
    .enqueue = CFS_enqueue,
    .dequeue = CFS_dequeue,
    .pick_next = CFS_pick_next,
    .proc_tick = CFS_proc_tick,
};
```

CFS验证

编译后使用 make grade命令：



```
Terminal
pgdir: (2.2s)
  -check result: OK
  -check output: OK
yield: (2.0s)
  -check result: OK
  -check output: OK
badarg: (2.1s)
  -check result: OK
  -check output: OK
exit: (2.6s)
  -check result: OK
  -check output: OK
spin: (2.4s)
  -check result: OK
  -check output: OK
waitkill: (2.2s)
  -check result: OK
  -check output: OK
forktest: (2.3s)
  -check result: OK
  -check output: OK
forktree: (2.8s)
  -check result: OK
  -check output: OK
matrix: (21.7s)
  -check result: OK
  -check output: OK
priority: (22.5s)
  -check result: OK
  -check output: OK
Total Score: 170/170
[~/moocos/ucore_lab/labcodes/lab6]
moocos->
```

然后是执行部分：

(THU.CST) os is loading ...

Special kernel symbols:

```
entry 0xc010002a (phys)
etext 0xc010c59d (phys)
edata 0xc01aedd4 (phys)
end 0xc01b1f78 (phys)
```

Kernel executable memory footprint: 712KB

memory management: default_pmm_manager

e820map:

```
memory: 0009fc00, [00000000, 0009fbff], type = 1.
memory: 00000400, [0009fc00, 0009ffff], type = 2.
```

```

memory: 00010000, [000f0000, 000ffffff], type = 2.
memory: 07efe000, [00100000, 07ffdf000], type = 1.
memory: 00002000, [07ffe000, 07ffffff], type = 2.
memory: 00040000, [ffffc000, 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: stride_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:      262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 31848, total 31848
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].
do pgfault: ptep c0396004, pte 200
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].
do pgfault: ptep c0396008, pte 300
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].
do pgfault: ptep c039600c, pte 400
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].

```

```

do pgfault: ptep c0396010, pte 500
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:460:
    initproc exit.

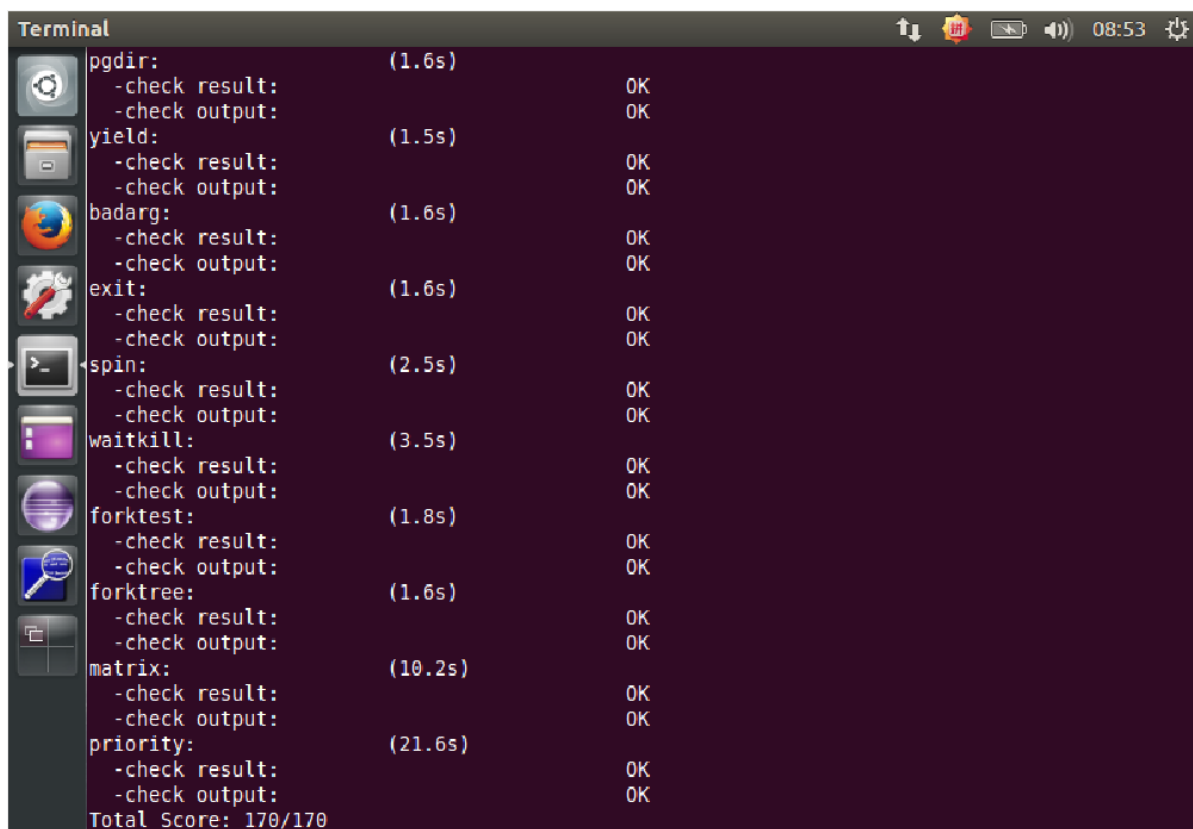
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

运行结果

下面对LAB6的代码进行验证（非扩展实验）。

首先是make grade命令下分数：



```

Terminal
pgdir: (1.6s)
- check result: OK
- check output: OK
yield: (1.5s)
- check result: OK
- check output: OK
badarg: (1.6s)
- check result: OK
- check output: OK
exit: (1.6s)
- check result: OK
- check output: OK
spin: (2.5s)
- check result: OK
- check output: OK
waitkill: (3.5s)
- check result: OK
- check output: OK
forktest: (1.8s)
- check result: OK
- check output: OK
forktree: (1.6s)
- check result: OK
- check output: OK
matrix: (10.2s)
- check result: OK
- check output: OK
priority: (21.6s)
- check result: OK
- check output: OK
Total Score: 170/170

```

然后是执行部分：

(THU.CST) os is loading ...

```

Special kernel symbols:
entry 0xc010002a (phys)

```

```

    etext 0xc010db64 (phys)
    edata 0xc01b1dd4 (phys)
    end    0xc01b4f78 (phys)
Kernel executable memory footprint: 724KB
ebp:0xc012ff38 eip:0xc0101f67 args:0x00010094 0x00000000 0xc012ff68 0xc01000d8
    kern/debug/kdebug.c:350: print_stackframe+21
ebp:0xc012ff48 eip:0xc0102256 args:0x00000000 0x00000000 0x00000000 0xc012ffb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc012ff68 eip:0xc01000d8 args:0x00000000 0xc012ff90 0xffff0000 0xc012ff94
    kern/init/init.c:58: grade_backtrace2+33
ebp:0xc012ff88 eip:0xc0100101 args:0x00000000 0xffff0000 0xc012ffb4 0x0000002a
    kern/init/init.c:63: grade_backtrace1+38
ebp:0xc012ffa8 eip:0xc010011f args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:68: grade_backtrace0+23
ebp:0xc012ffc8 eip:0xc0100144 args:0xc010db9c 0xc010db80 0x000031a4 0x00000000
    kern/init/init.c:73: grade_backtrace+34
ebp:0xc012fff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:32: 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, 07ffdfdf], type = 1.
    memory: 00002000, [07ffe000, 07ffffff], type = 2.
    memory: 00040000, [fffc0000, ffffffff], type = 2.
check_all_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
check_slab() success
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/w [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
ide 0: 10000(sectors), 'QEMU HARDDISK'.
ide 1: 262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 31813, total 31813
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].
do pgfault: ptep c03b9004, pte 200
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].
do pgfault: ptep c03b9008, pte 300
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].
do pgfault: ptep c03b900c, pte 400
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].
do pgfault: ptep c03b9010, pte 500
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:460:
    initproc exit.

welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

与指导书一致。

感想与总结

本次实验了解了Round Robin算法的实现过程，对其实现的思想有了更透彻的了解，并且自己完成实现了Stride Scheduling调度算法，对调度的主体思想以及实现有了更进一步的认识。在实验过程中遇到了很多问题，通过查阅相关资料一步步解决，了解了很多调度的其他方面知识，使用Linux各种命令辅助完成代码，让自己更熟练使用Linux系统，收获颇多。

相关知识点

本实验中相关的知识点

- Round Robin调度算法
- Stride Scheduling调度算法
- CFS调度算法

本实验未涉及

- 其余的单处理器调度算法
- 多处理器调度

参考资料

1. https://blog.csdn.net/qg_31098037/article/details/78125893
2. <https://www.cnblogs.com/tianguiyu/articles/6091378.html>
3. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.3502&rank=1>