练习0 填写已有实验

修改proc.c的alloc_proc

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
    /*
     * below fields in proc_struct need to be initialized
     *       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
     */
        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;
        // lab 6
        proc->rq = NULL;
        list_init(&(proc->run_link));
        proc->time_slice = 0;
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc-
>lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;
        proc->lab6_priority = 0;
    }
```

```
    return proc;
}
```

增加相关初始化函数

修改trap.c 的中断处理函数

```
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:
#if 0
    LAB3 : If some page replacement algorithm(such as CLOCK PRA) need tick to
change the priority of pages,
    then you can add code here.
#endif
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a
global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton,
such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        /* LAB5 YOUR CODE */
        /* you should upate you lab1 code (just add ONE or TWO lines of code):
         *    Every TICK_NUM cycle, you should set current process's current-
>need_resched = 1
         */
        ticks ++;
        assert(current != NULL);
        sched_class_proc_tick(current);
        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;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    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);
        }
        // in kernel, it must be a mistake
        panic("unexpected trap in kernel.\n");

    }
}
```

增加sched_class_proc_tick(current);在中断中运行调度处理函数

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。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。

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，并接合Round Robin 调度算法描ucore的调度执行过程

```
// The introduction of scheduling classes is borrrowed from Linux, and makes the
// core scheduler quite extensible. These classes (the scheduler modules)
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 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[]);
     */
};
```

name: 调度算法名称

init: 调度初始化函数

enqueue: 将新的进程加入调度队列

dequeue: 将旧的进程移出调度队列

pick_next: 获取下一个被调度的进程

proc_tick: 每次调用减少当前运行进程的剩余时间

运行流程：

首先，uCore调用sched_init函数用于初始化相关的就绪队列。

之后在proc_init函数中，建立第一个内核进程，并将其添加至就绪队列中。

当所有的初始化完成后，uCore执行cpu_idle函数，并在其内部的schedule函数中，调用sched_class_enqueue将当前进程添加进就绪队列中（因为当前进程要被切换出CPU了）

然后，调用sched_class_pick_next获取就绪队列中可被轮换至CPU的进程。如果存在可用的进程，则调用sched_class_dequeue函数，将该进程移出就绪队列，并在之后执行proc_run函数进行进程上下文切换。

需要注意的是，每次时间中断都会调用函数sched_class_proc_tick。该函数会减少当前运行进程的剩余时间片。如果时间片减小为0，则设置need_resched为1，并在时间中断例程完成后，在trap函数的剩余代码中进行进程切换。

- 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

1. When a process starts executing then it first enters queue 1.
2. In queue 1 process executes for 4 units and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit then the priority of this process does not change and if it again comes in the ready queue then it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 units then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 units.In a general case if a process does not complete in a time quantum then it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in an FCFS manner.
6. A process in a lower priority queue can only execute only when higher priority queues are empty.

7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

根据多级反馈队列的这几条规则设计如下：

- 该算法需要设置多个`run_queue`，而这些`run_queue`的`max_time_slice`需要按照优先级依次递减。
- 在`sched_init`函数中，程序先初始化这些`run_queue`，并依次从大到小设置`max_time_slice`。
- 而执行`sched_class_enqueue`时，先判断当前进程是否是新建立的进程。如果是，则将其添加至最高优先级（即时间片最大）的队列。如果当前进程是旧进程（即已经使用过一次或多次CPU，但进程仍然未结束），则将其添加至下一个优先级的队列，因为该进程可能是IO密集型的进程，CPU消耗相对较小。
- `sched_class_pick_next`要做的事情稍微有点多。首先要确认下一次执行的该是哪条队列里的哪个进程。为便于编码，我们可以直接指定切换至队列中的**第一个**进程（该进程是**等待执行时间**最久的进程）。

  但队列的选择不能那么简单，因为如果只是简单的选择执行**第一个队列**中的进程，则大概率会产生**饥饿**，即低优先级的进程长时间得不到CPU资源。所以，我们可以设置每条队列占用**固定时间/固定百分比**的CPU。例如在每个队列中添加一个`max_list_time_slice`属性并初始化，当该队列中的进程**总运行时间**超过当前进程所在队列的`max_list_time_slice`（即**最大运行时间片**），则CPU切换至下一个队列中的进程。

练习2 实现 Stride Scheduling 调度算法（需要编码）

首先需要换掉RR调度器的实现，即用default_sched_stride_c覆盖default_sched.c。然后根据此文件和后续文档对Stride度器的相关描述，完成Stride调度算法的实现。

后面的实验文档部分给出了Stride调度算法的大体描述。这里给出Stride调度算法的一些相关的资料（目前网上中文的资料比较欠缺）。

设计初始化函数如下：

```
/*
 * 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 proj13.1/libs/list.h for routines of the list structures.
 */
static void
stride_init(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
     list_init(&(rq->run_list));
     rq->proc_num = 0;
}
```

初始化 run_list和proc_num

设计入队函数如下：

```
/*
 * 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 date in ``rq'' structure.
 *
 * proc->time_slice denotes the time slices allocation for the
 * process, which should set to rq->max_time_slice.
 *
 * hint: see proj13.1/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 */
     assert(list_empty(&(proc->run_link)));
     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->rq = rq;
     rq->proc_num ++;
}
```

设计出队函数如下：

```
/*
 * 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 proj13.1/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 */
     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
     list_del_init(&(proc->run_link));
     rq->proc_num --;
}
```

设计tick函数如下

```
/*
 * 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 */
     if (proc->time_slice > 0) {
          proc->time_slice --;
     }
     if (proc->time_slice == 0) {
          proc->need_resched = 1;
     }
}
```

都和RR调度算法一样

只有pick函数不一样

```
/*
 * 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 proj13.1/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 proj13.1/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 */
     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);
     }
     if (p->lab6_priority == 0)
          p->lab6_stride += BIG_STRIDE;
     else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
     return p;
}
```

实现思路就是选择一个当前步长最小的运行

然后增加它的步长

```
-> % make grade
badsegment:              (s)
  -check result:                           OK
  -check output:                           OK
divzero:                 (s)
  -check result:                           OK
  -check output:                           OK
softint:                 (s)
  -check result:                           OK
  -check output:                           OK
faultread:               (s)
  -check result:                           OK
  -check output:                           OK
faultreadkernel:         (s)
  -check result:                           OK
  -check output:                           OK
hello:                   (s)
  -check result:                           OK
  -check output:                           OK
testbss:                 (s)
  -check result:                           OK
  -check output:                           OK
pgdir:                   (s)
  -check result:                           OK
  -check output:                           OK
yield:                   (s)
  -check result:                           OK
  -check output:                           OK
badarg:                  (s)
  -check result:                           OK
  -check output:                           OK
exit:                    (s)
  -check result:                           OK
  -check output:                           OK
spin:                    (s)
  -check result:                           OK
  -check output:                           OK
waitkill:                (s)
  -check result:                           OK
  -check output:                           OK
forktest:                (s)
  -check result:                           OK
  -check output:                           OK
forktree:                (s)
  -check result:                           OK
  -check output:                           OK
matrix:                  (s)
  -check result:                           OK
  -check output:                           OK
priority:                (s)
  -check result:                           OK
  -check output:                           OK
Total Score: 170/170
```

challenge 实现 Linux 的 CFS 调度算法

实现 Linux 的 CFS 调度算法

- CFS调度算法原理

CFS 算法的基本思路就是尽量使得每个进程的运行时间相同，所以需要记录每个进程已经运行的时间：

```
struct proc_struct {
    ...
    int fair_run_time;                  // FOR CFS ONLY: run time
};
```

每次调度的时候，选择已经运行时间最少的进程。所以，也就需要一个数据结构来快速获得最少运行时间的进程， CFS 算法选择的是红黑树，但是项目中的斜堆也可以实现，只是性能不及红黑树。CFS是对于**优先级**的实现方法就是让优先级低的进程的时间过得很快。

(1) 数据结构

首先需要在 run_queue 增加一个斜堆：

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
    // For LAB6 ONLY
    skew_heap_entry_t *lab6_run_pool;
    //CFS
    skew_heap_entry_t *fair_run_pool;
};
```

在 proc_struct 中增加三个成员：

- 虚拟运行时间：fair_run_time
- 优先级系数：fair_priority，从 1 开始，数值越大，时间过得越快
- 斜堆：fair_run_pool

```
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)
    //CFS
    int fair_run_time;
    int fair_priority;
    skew_heap_entry_t fair_run_pool;
};
```

(2) 算法实现

- **proc_fair_comp_f函数**

比较函数，用于比较两个任务的运行时间，确保红黑树可以排序得到`fair_run_time`最小的节点。

首先需要一个比较函数，同样根据 $MAX_RUNTIME - MIN_RUNTIE < MAX_PRIORITY$ 完全不需要考虑虚拟运行时溢出的问题。

```
static int proc_cfs_comp_f(void *a, void *b) {
     struct proc_struct *p = le2proc(a, fair_run_pool);
     struct proc_struct *q = le2proc(b, fair_run_pool);
     int32_t c = p->fair_run_time - q->fair_run_time;
     if (c > 0)
         return 1;
     else if (c == 0)
         return 0;
     else
         return -1;
}
```

- **fair_init**

初始化，初始化堆为空，proc_num进程数为 0

```
static void cfs_init(struct run_queue *rq) {
    rq->fair_run_pool = NULL;
    rq->proc_num = 0;
}
```

- **fair_enqueue**

入堆，在将指定进程加入就绪队列的时候，需要调用斜堆的插入函数将其插入到斜堆中，然后对时间片等信息进行更新

```
static void cfs_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    rq->fair_run_pool = skew_heap_insert(rq->fair_run_pool, &(proc->fair_run_pool), proc_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;
    rq->proc_num++;
}
```

- **fair_dequeue**

出队列，将指定进程从就绪队列中删除，只需要将该进程从斜堆中删除掉即可。

```
static void cfs_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    rq->fair_run_pool = skew_heap_remove(rq->fair_run_pool, &(proc->fair_run_pool), proc_cfs_comp_f);
    rq->proc_num--;
}
```

- **fair_pick_next**

选择调度函数，选择下一个要执行的进程，选择虚拟运行时`fair_run_time`最小的节点。

```
static struct proc_struct *cfs_pick_next(struct run_queue *rq) {
    if (rq->fair_run_pool == NULL)
        return NULL;
    skew_heap_entry_t *le = rq->fair_run_pool;
    struct proc_struct * p = le2proc(le, fair_run_pool);
    return p;
}
```

- **fair_proc_tick**

时间片函数，需要更新虚拟运行时，增加的量为优先级系数。

```
static void
fair_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {          //到达时间片
        proc->time_slice --;             //执行进程的时间片 time_slice 减一
        proc->fair_run_time += proc->fair_priority; //优先级系数：fair_priority，从1 开始，数值越大，时间过得越快
    }
    if (proc->time_slice == 0) {         //时间片为 0，设置此进程成员变量 need_resched 标识为 1，进程需要调度
        proc->need_resched = 1;
    }
}
```

兼容调整

为了保证测试可以通过，需要将 Stride Scheduling 的优先级对应到 CFS 的优先级：

```
void lab6_set_priority(uint32_t priority)
{
    ...
    // FOR CFS ONLY
    current->fair_priority = 60 / current->lab6_priority + 1;   //
    if (current->fair_priority < 1)
        current->fair_priority = 1;     //设置此进程成员变量 need_resched 标识为 1，进程需要调度
}
```

由于调度器需要通过虚拟运行时间确定下一个进程，如果虚拟运行时间最小的进程需要 yield ( )，那么必须增加虚拟运行时间

例如可以增加一个时间片的运行时。

```
int do_yield(void) {
    ...
    // FOR CFS ONLY
    current->fair_run_time += current->rq->max_time_slice * current->fair_priority;
    return 0;
}
```

我遇到的问题——数据结构

> 为什么 CFS 调度算法使用红黑树而不使用堆来获取最小运行时进程？

查阅了网上的资料以及自己分析，得到如下结论：

- 堆基于数组，但是对于调度器来说进程数量不确定，无法使用定长数组实现的堆；
- ucore 中的 Stride Scheduling 调度算法使用了斜堆，但是斜堆没有维护平衡的要求，可能导致斜堆退化成为有序链表，影响性能。

综上所示，红黑树因为平衡性以及非连续所以是CFS算法最佳选择。

```
-> % make grade
badsegment:              (s)
  -check result:                              OK
  -check output:                              OK
divzero:                 (s)
  -check result:                              OK
  -check output:                              OK
softint:                 (s)
  -check result:                              OK
  -check output:                              OK
faultread:               (s)
  -check result:                              OK
  -check output:                              OK
faultreadkernel:         (s)
  -check result:                              OK
  -check output:                              OK
```

```
hello:                    (s)
  -check result:                    OK
  -check output:                    OK
testbss:                  (s)
  -check result:                    OK
  -check output:                    OK
pgdir:                    (s)
  -check result:                    OK
  -check output:                    OK
yield:                    (s)
  -check result:                    OK
  -check output:                    OK
badarg:                   (s)
  -check result:                    OK
  -check output:                    OK
exit:                     (s)
  -check result:                    OK
  -check output:                    OK
spin:                     (s)
  -check result:                    OK
  -check output:                    OK
waitkill:                 (s)
  -check result:                    OK
  -check output:                    OK
forktest:                 (s)
  -check result:                    OK
  -check output:                    OK
forktree:                 (s)
  -check result:                    OK
  -check output:                    OK
matrix:                   (s)
  -check result:                    OK
  -check output:                    OK
priority:                 (s)
  -check result:                    OK
  -check output:                    OK
Total Score: 170/170
```