ucoreOS操作系统实验——Lab6

问题发现与改进

- 其实练习二并不一定要像实验指导书说的一样用stride覆盖RR，可以使用前几个lab做Challenge使用的方法，将class的init改写即可
- trap中对 `schedule` 的调用体现了用户进程的可抢占性以及内核进程的不可抢占性

```
if (!in_kernel) {
  //只有用户进程执行用户代码发生中断，且当前进程控制块成员变量need_resched为1才执行shedule函数
    ……

    if (current->need_resched) {
        schedule();
    }
}
```

- 进程切换后执行的是进程上一次在内核调用schedule函数返回后的下一行代码

练习0

> 同Lab5要做修改

- 需要继续完善 `proc.c` 中的对进程初始化的部分,因为proc.h中对PCB的定义又多了几个Lab6 only

```
static struct proc_struct *alloc_proc(void) {
    proc->rq = NULL;//初始化进程所属就绪队列
    list_init(&(proc->run_link));//初始化进程队列指针
    proc->time_slice = 0;//初始化进程时间片
}
```

- 以及 `trap.c` 中Lab5添加的部分改成

```
static void trap_dispatch(struct trapframe *tf) {
  // if (++ticks % TICK_NUM == 0) {
  // assert(current != NULL);
  // current->need_resched = 1;
  // }
    ++ticks;
    sched_class_proc_tick(current);// 这个标记现在已被调度程序所使用,就不需要自己控制了
}
```

拿MacOS上的FileMerge看了一下,区别基本上就是:

- PCB中增加了三个与stride调度算法相关的成员变量，以及增加了对应的初始化过程
- 新增了斜堆数据结构的实现

- 新增了默认的调度算法Round Robin的实现，在调用sched_class_*等一系列函数之后，进一步调用调度器sched_class中封装的函数，默认该调度器为Round Robin调度器，这是在default_sched.*中定义的
- 新增了set_priority，get_time的系统调用

练习一

- 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

1.

- sched_class结构体：

```
struct sched_class {
    const char *name;  // 调度器名称
    void (*init)(struct run_queue *rq);// 队列数据结构
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // 进程入队 就绪队列
    // 进程出队 将进程从就绪队列中删去
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // 挑出下一个进度器选择的进程
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // 更新调度器的时钟信息
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

```c
// 初始化算法所数据结构
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}
// 进程入队
// 将进程加入就绪队列(不同的就绪队列的时间片不同，有不同优先级的就绪队列)
// 当进程时间片为0或应某种情况被阻塞则将其加入到就绪队列并将其时间片进行重置
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));
    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 ++;
}
// 进程出队 将进程从就绪队列中删去
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));
    rq->proc_num --;
}
// 挑选出下一个进程
// 在 RR调度中 直接按照就绪队列的顺序轮询
chyyuu, 7 years ago | 1 author (chyyuu)
static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
// 时钟中断时 调用此函数 在 RR调度中 每次调用都会减少当前进程时间片      You, a few
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

- Round Robin 调度执行过程：
  - RR的实现主要是当需要将某一个进程加入就绪进程队列中，则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，具体执行流程：
    - 设置当前进程的剩余时间片
    - 每发生时间中断，都将当前进程剩余时间片–1
    - 当剩余时间片为0时，当前进程的 `need_resched` 被置1，表示该进程需要被调度
    - ISR中断服务例程一旦发现当前进程需要被调度就调用 `schedule()` 函数调度它
    - 并将当前进程放入就绪队列中，同时把时间片设为当前队列的最大时间片
    - 调用 `pick_next()` 选择下一个需要运行处理的进程，若选择成功将其从就绪队列删除
    - 找不到就用idle，该懒进程会死循环无穷无尽的查找可被调度的进程
    - 调用 `proc_run()` 进行进程切换

2.

- MLFQ：

- 时间片大小随优先级级别增加而增加
- 进程在当前时间片没有完成则降到下一优先级

- 实现：
  - sched_init：
    - 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约低，并且优先级越低的队列上时间片的长度越大，是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
    - 初始化的时候需要同时对N个队列初始化

```
struct run_queue {//增加多个队列
  ...
    // For MLFQ ONLY
    list_entry_t multi_run_list[MULTI_QUEUE_NUM];
};
struct proc_struct {//增加队列优先级号
  ...// FOR MLFQ ONLY
    int multi_level;
};
static void multi_init(struct run_queue *rq) {//对每一个队列初始化
    for (int i = 0; i < MULTI_QUEUE_NUM; i++)
        list_init(&(rq->multi_run_list[i]));
    rq->proc_num = 0;
}
```

  - enquene：
    - 若该进程的时间片使用完，考虑降低其优先度，加入到优先级低1级的队列中去
    - 若该进程时间片没有用完，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

```
static void multi_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    int level = proc->multi_level;
    if (proc->time_slice == 0 && level < (MULTI_QUEUE_NUM-1))//时间片用完
        level ++;
    proc->multi_level = level;//没用完不变
    list_add_before(&(rq->multi_run_list[level]), &(proc->run_link));//把进程加入相应队列
    if (proc->time_slice == 0 || proc->time_slice > (rq->max_time_slice << level))
        proc->time_slice = (rq->max_time_slice << level);
    proc->rq = rq;
    rq->proc_num ++;
}
```

  - dequene：
    - 从就绪进程集合中删除某一个进程只需在对应队列中删除即可

```
static void multi_dequeue(struct run_queue *rq, struct proc_struct *proc) {//不变
  assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
  list_del_init(&(proc->run_link));
  rq->proc_num --;
}
```

  - pick_next：

- 优先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

```
static struct proc_struct *multi_pick_next(struct run_queue *rq) {
    for (int i = 0; i < MULTI_QUEUE_NUM; i++)
        if (!list_empty(&(rq->multi_run_list[i])))
            return le2proc(list_next(&(rq->multi_run_list[i])), run_link);
    return NULL;
}
```

- proc_tick:
  - 无需更改

练习二

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

- 首先在PCB的初始化中加入以下内容

```
//     Stride Only
    proc->lab6_run_pool.parent = proc->lab6_run_pool.left = proc->lab6_run_poo
    // 优先级（和步进成反比）
    proc->lab6_priority = 0;
    // 步进值
    proc->lab6_stride = 0;      You, a few seconds ago • Uncommitted changes
```

- 并把 `BIG_STRIDE` 赋成32位下的int最大值 `#define BIG_STRIDE (((uint32_t)-1) / 2)`
- `stride_init`：初始化，在本stride调度算法的实现中使用了斜堆来实现优先队列，因此需要对相应的成员变量进行初始化

```
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
      */
     list_init(&rq->run_list);
     rq->lab6_run_pool = NULL; // 对斜堆进行初始化，表示有限队列为空
     rq->proc_num = 0;
}
```

- `stride_enqeue`：调用斜堆的插入函数将进程插入优先队列，更新进程剩余时间片，设置进程队列指针，增加进程计数值

```c
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
      */
     // 将新的进程插入到表示就绪队列的斜堆中，该函数的返回结果是斜堆的新的根
     rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &proc->lab6_run_pool, proc_stride_comp_f);
     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
          proc->time_slice = rq->max_time_slice; // 将该进程剩余时间置为时间片大小
     }
     proc->rq = rq; // 更新进程的就绪队列
     rq->proc_num ++; // 维护就绪队列中进程的数量
}
```

- `stride_dequeue`： 将进程从优先队列中移除，将进程计数值减一

```c
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
      */
     rq->lab6_run_pool = skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);
     rq->proc_num --;
}
```

- `stride_pick_next`：队列为空返回空指针，从优先队列获得一个进程(选择stride最小的，即斜堆根节点)，更新stride值

```c
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
      */
     if (rq->lab6_run_pool == NULL)
        return NULL;
        // 斜堆的顶就是 stride 值最小的进程
    skew_heap_entry_t *le = rq->lab6_run_pool;
    struct proc_struct * p = le2proc(le, lab6_run_pool);
    p->lab6_stride += BIG_STRIDE / p->lab6_priority;// 更新该进程的stride值
    return p;
}
```

- `stride_proc_tick`： 每次时钟中断需要调用的函数，与RR算法一致

```c
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;
     }
}
```

这里还有两个提问

1. 如何证明 `STRIDE_MAX – STRIDE_MIN <= PASS_MAX` ?

- 反证法。假设不成立，说明就绪队列在上一次找出用于执行的进程的时候，假设选择进程P，即存在另外一个就绪的进程P'，且有P'的stride比P严格地小，这就说明上一次调度出现了问题，这和stride算法的设计是相违背的，所以原命题成立

2. 为什么 `BIG_STRIDE` 是最终的取值?

- Stride调度算法的思想是每次都找stride步进值最小的进程
- 每个进程每次执行完以后都要stride步进 += pass步长，又因为**步长**和**优先级**成**反比**的，所以步长可以反映出进程的优先级
- 但是随着调度次数增多，步长不断增加，就有可能溢出
- 所以有一个步长最大上限，即使溢出了也可以继续比较
- ucore里面BIG_STRIDE和stride都是无符号32位整数，所以最大值只能是$(2^{32}-1)$
- 但是一个进程的stride已经是$(2^{32}-1)$再加上pass步长就没有比较的意义
- 又因为 `pass = BIG_STRIDE / priority <= BIG_STRIDE` → `pass_max <= BIG_STRIDE`
- 所以最大步长–最小步长≤步长 `max_stride – min_stride <= pass_max`
- 所以 `max_stride – min_stride <= BIG_STRIDE`
- 又因为最大是$(2^{32}-1)$，最小是0，所以只要保证任意两个进程stride的差值在32位有符号数能够表示的范围之内即可
- `BIG_STRIDE` 为 $\frac{2^{32}-1}{2}$

实验结果

```
do pgfault: ptep c0000004, 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
count is 1, 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.

stack trackback:
ebp:0xc03bbf98 eip:0xc0101edd args:0x00000018 0x00000000 0x00000000 0xc03bbfcc
    kern/debug/kdebug.c:350: print_stackframe+21
ebp:0xc03bbfb8 eip:0xc01017d9 args:0xc010eeb4 0x000001cc 0xc010ef06 0x00000000
    kern/debug/panic.c:27: __panic+107
ebp:0xc03bbfe8 eip:0xc010a96b args:0x00000000 0x00000000 0x00000000 0x00000010
    kern/process/proc.c:460: do_exit+81
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

```
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
main: fork ok,now need to wait pids.
child pid 7, acc 984000, time 1001
child pid 6, acc 808000, time 1002
child pid 4, acc 424000, time 1005
child pid 5, acc 632000, time 1007
child pid 3, acc 224000, time 1007
main: pid 3, acc 224000, time 1008
main: pid 4, acc 424000, time 1009
main: pid 5, acc 632000, time 1009
main: pid 6, acc 808000, time 1009
main: pid 7, acc 984000, time 1009
main: wait pids over
stride sched correct result: 1 2 3 4 4
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:460:
    initproc exit.
```

```
badsegment:              (1.7s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (1.6s)
  -check result:                             OK
  -check output:                             OK
softint:                 (1.6s)
  -check result:                             OK
  -check output:                             OK
faultread:               (1.6s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:         (1.6s)
  -check result:                             OK
  -check output:                             OK
hello:                   (1.6s)
  -check result:                             OK
  -check output:                             OK
testbss:                 (1.7s)
  -check result:                             OK
  -check output:                             OK
pgdir:                   (1.6s)
  -check result:                             OK
  -check output:                             OK
yield:                   (1.6s)
  -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:                    (1.8s)
  -check result:                             OK
  -check output:                             OK
waitkill:                (2.1s)
  -check result:                             OK
  -check output:                             OK
forktest:                (1.7s)
  -check result:                             OK
  -check output:                             OK
forktree:                (1.6s)
  -check result:                             OK
  -check output:                             OK
matrix:                  (15.9s)
  -check result:                             OK
  -check output:                             OK
priority:                (11.5s)
  -check result:                             OK
  -check output:                             OK
Total Score: 170/170
```

Challenge

> 有空再做 TODO

没空