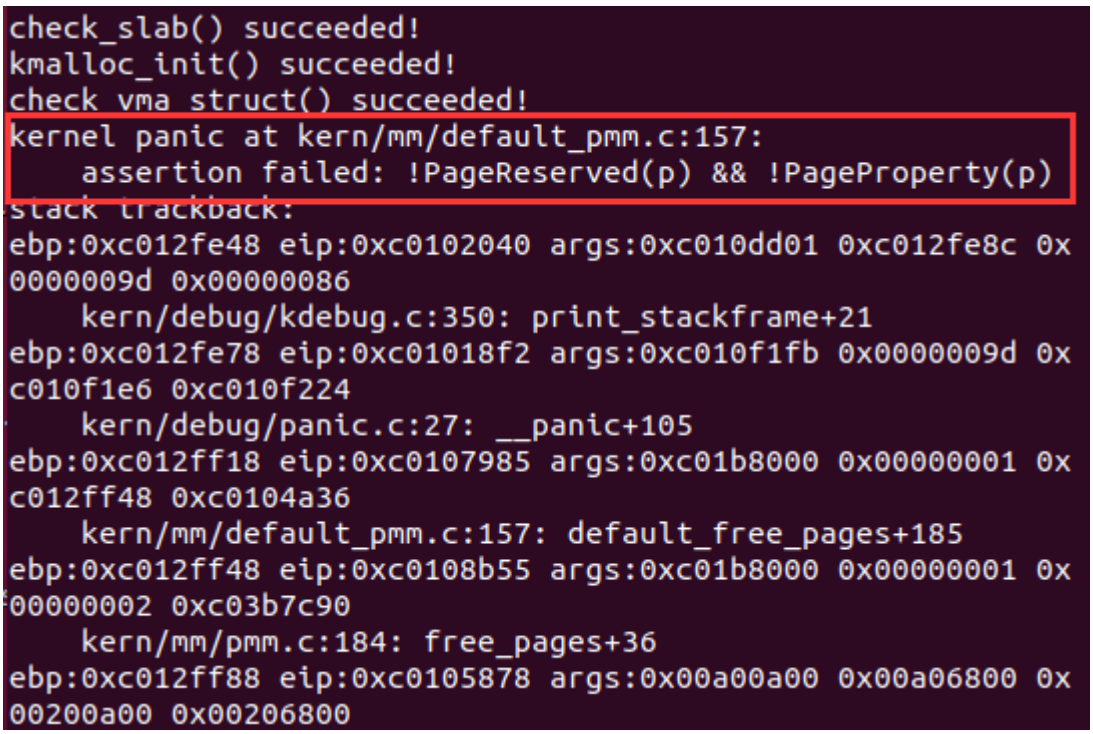
实验六: 调度器

遇到的问题

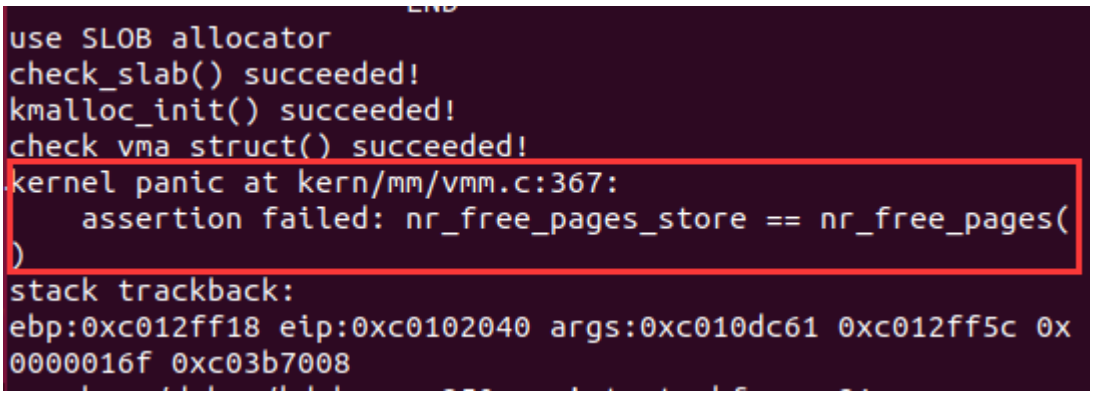
还是和之前实验一样，先在kern/mm/kmalloc.c把success改成succeeded! 在这里想要纠正自己在lab5中犯的一个错误，kern/process/proc.c的init_main(void *arg) 的assert中的那两个函数调用不是多余的，在lab5中我以为是原来的代码有错就删掉了……

本来想对照着answer边理解边写的，但是在answer里执行了一下make grade，结果出来是0分。折腾了一番，无功而返。想吐槽一下，本来就不怎么会写，结果还因为莫名的原因连answer都跑不了，不知道是不是因为qemu版本的问题，指导书使用的是 qemu-1.0.1，我的虚拟机用的是qemu-4.0.1。主要的部分还没做，在关系不大的地方上就花了很多时间。

问了好几个同学，他们在执行make qemu时都出现了这样的报错：



将报错的那一行注释掉之后，执行make qemu：



继续注释报错的那一行，执行make qemu就没什么问题了：

```
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 = "priority".
main: fork ok,now need to wait pids.
child pid 6, acc 468000, time 1003
child pid 7, acc 572000, time 1004
child pid 4, acc 244000, time 1006
child pid 5, acc 340000, time 1008
child pid 3, acc 124000, time 1009
main: pid 3, acc 124000, time 1009
main: pid 4, acc 244000, time 1010
main: pid 5, acc 340000, time 1010
main: pid 6, acc 468000, time 1010
main: pid 7, acc 572000, time 1010
main: wait pids over
stride sched correct result: 1 2 3 4 5
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:460:
    initproc exit.
```

result都是对的，output不对，但是这部分不是我写的，所以我暂时也没有去解决，我觉得我暂时可能也解决不了。

```
pcy@ubuntu:~/Desktop/labcodes/lab6$ make grade
badsegment:              (3.2s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

divzero:                 (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

softint:                 (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

faultread:               (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

faultreadkernel:         (1.4s)
  -check result:                             OK
  -check output:                             WRONG
```

在做练习一的第二问时，我怎么觉得，好像没学过多级反馈队列调度算法（这块儿是从网上找的答案）.....

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。

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。

用meld进行代码合并，修改代码如下

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        //LAB4:EXERCISE1 YOUR CODE
        proc->state = PROC_UNINIT;//设置进程为未初始化状态
        proc->pid = -1;//未初始化的进程id=-1
        proc->runs = 0;//初始化时间片
        proc->kstack = 0;//初始化内存栈的地址
        proc->need_resched = 0;//是否需要调度设为不需要
        proc->parent = NULL;//置空父节点
        proc->mm = NULL;//置空虚拟内存
        memset(&(proc->context), 0, sizeof(struct context));//初始化上下文
        proc->tf = NULL;//中断帧指针设置为空
        proc->cr3 = boot_cr3;//页目录设为内核页目录表的基址
        proc->flags = 0;//初始化标志位
        memset(proc->name, 0, PROC_NAME_LEN);//置空进程名

        //LAB5 YOUR CODE : (update LAB4 steps)
        proc->wait_state = 0;//初始化进程等待状态
        proc->cptr = proc->optr = proc->yptr = NULL;//进程相关指针初始化
        //LAB6 YOUR CODE : (update LAB5 steps)
        proc->rq = NULL;//置运行队列为空
        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

        //该进程在优先队列中的节点，仅在lab6中使用
        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
        proc->lab6_stride = 0;//初始化当前运行步数
        proc->lab6_priority = 0;//初始化优先级
    }
```

```
    return proc;
}
```

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，并结合Round Robin 调度算法描述ucore的调度执行过程

分析sched_class中各个函数指针的用法

首先，kern_init在调用vmm_init初始化完虚拟内存后，调用sched_init来初始化调度器。在sched_init函数中将sched赋值为default_sched_class（kern/schedule/sched.c）。

```
    sched_class = &default_sched_class;
```

default_sched_class的成员如下所示（kern/schedule/default_sched.c）

```
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,
};
```

RR_init初始化RUNNABLE进程链表run_list，并将RUNNABLE进程的数目proc_num设置为0

```
    list_init(&(rq->run_list));
    rq->proc_num = 0;
```

RR_enqueue将一个新进程添加到RUNNABLE进程链表run_list的末尾，并将RUNNABLE进程数目proc_num加1。并且检查该进程的时间片，如果为0，重新为它分配时间片max_time_slice。此外还要保证进程的时间片不超过max_time_slice，如果超过了，将其修正为max_time_slice。

```
    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 ++;
```

RR_dequeue从RUNNABLE进程链表run_list中删除指定进程，并将RUNNABLE进程数目proc_num减1.

```
    list_del_init(&(proc->run_link));
    rq->proc_num --;
```

RR_pick_next从RUNNABLE进程链表run_list中取出首元素，然后返回。

```
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
```

RR_proc_tick将指定进程的时间片减1，如果减1后时间片数值为0，说明该进程的时间片用完了，需要被调度出去，因此将其need_resched标志设置为1.

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

结合Round Robin调度算法描述ucore的调度执行过程

1. ucore内核初始化总入口kern_init调用sched_init来初始化调度器sched_class，接下来调用proc_init来初始化进程。
2. proc_init首先为当前正在运行的ucore程序分配一个进程控制块，并将其命名为idle，即第一个内核线程idleproc。
3. idleproc调用kernel_thread来创建一个新的内核线程initproc，kernel_thread进一步调用do_fork来完成具体的进程初始化操作，完成后调用wakeup_proc来唤醒新进程，并将内核线程initproc放在RUNNABLE队列rq的末尾。这时rq队列有了第一个进程在等待调度。
4. proc_init结束后，继续一路运行到cpu_idle，在cpu_idle中，不断判断当前进程是否需要调度，如果需要则调用schedule进行调度。由于当前进程是idleproc，其need_resched设置为1，因此进入schedule进行调度。
5. schedule首先判断当前进程是否RUNNABLE，以及是不是idleproc，如果当前进程不是idleproc而且RUNNABLE，则将其加入到rq队列的末尾。由于当前进程是idleproc，因此不会将其加入rq队列。
6. 接下来从RUNNABLE队列中取出队首的进程（此时是initproc），通过调用proc_run来运行initproc进程。这时rq队列已空。
7. initproc进程运行init_main，init_main调用kernel_thread来创建第三个进程userproc。同理，在完成userproc的初始化后，会调用wakeup_proc将其唤醒，并将其加入到rq队列的末尾。这时rq队列有一个进程userproc在等待调度。
8. initproc进程接下来调用do_wait来等待子进程结束运行，其中搜索到其子进程userproc的state不为ZOMBIE，因此调用schedule来试图调度子进程来运行。由于rq队列只有一个进程initproc在排队，因此会调用idleproc来运行。这时rq队列又空了。另外注意，由于initproc进程在调用schedule之前将自己的state设置为SLEEPING，因此在进入schedule后，不会再次将其加入到rq队列，也就是说initproc需要睡眠了，等子进程userproc运行结束后再将其唤醒。
9. userproc进程运行user_main，加载ELF文件并运行之。运行完毕，则调用do_exit，在do_exit中，将自己的state设置为Zombie，然后调用wakeup_proc来唤醒initproc，这时会将initproc加入到rq队列，因

此rq队列又有一个进程在等待了。接着调用schedule，选择刚加入的initproc来运行，rq队列再次变空。

10. initproc回收子进程userproc的资源后，打印一些字符串信息，然后退出init_main，接下来进入do_exit，do_exit调用panic，panic停留在kmonitor界面一直等待用户输入。

- 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

  多级反馈队列调度算法过程如下：

  - 多级反馈队列调度算法维护多个队列，每个新的进程加入Q_0中；
  - 每次选择进程执行的之后从Q_0开始向Q_n查找，如果某个队列非空，那么从这个队列中取出一个进程；
  - 如果来自Q_i某个进程在时间片用完之后没结束，那么将这个进程加入Q_{i+1}，时间片加倍。

数据结构

首先需要在run_queue中增加多个队列：

```
struct run_queue {
    ...
    // For Multi-Level Feedback Queue Scheduling ONLY
    list_entry_t multi_run_list[MULTI_QUEUE_NUM];
};
```

然后在proc_struct中增加进程的队列号（优先级）：

```
struct proc_struct {
    ...
    int multi_level; // FOR Multi-Level Feedback Queue Scheduling ONLY: the level
of queue
};
```

算法实现

multi_init

需要初始化每一个级别的队列。

```
static void multi_init(struct run_queue *rq) {
    for (int i = 0; i < MULTI_QUEUE_NUM; i++)
        list_init(&(rq->multi_run_list[i]));
    rq->proc_num = 0;
}
```

multi_enqueue

- 如果进程上一个时间片用完了，考虑增加level（降低优先级）；
- 加入level对应的队列；
- 设置level对应的时间片；
- 增加计数值。

```
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))
        level ++;
    proc->multi_level = level;
    list_add_before(&(rq->multi_run_list[level]), &(proc->run_link));
    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 ++;
}
```

multi_dequeue

将进程从对应的链表删除。

```
static void multi_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}
```

multi_pick_next

按照优先级顺序检查每个队列，如果队列存在进程，那么选择这个进程。

```
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;
}
```

multi_proc_tick

这和RR算法是一样的。

练习2: 实现 Stride Scheduling 调度算法（需要编码）

在default_sched.c里面直接进行修改了，将sched_init中绑定调度器的地方改为stride_scheduler。

1. 为每个runnable的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进程在调度后，stride 需要进行的累加值。
2. 每次需要调度时，从当前 runnable 态的进程中选择 stride最小的进程调度，由proc_stride_comp_f函数进行比较操作。对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。
3. 在一段固定的时间之后，回到步骤2，重新调度当前stride最小的进程

```
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;
    //注意这里是有符号数！而两个操作数都是无符号数
     if (c > 0) return 1;
     else if (c == 0) return 0;
     else return -1;
}
```

1. stride_init：初始化RUNNABLE链表run_list，初始化运行队列的运行池lab6_run_pool为空，初始化RUNNABLE进程数目为0.

```
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;
}
```

2. stride_enqueue：该函数将一个进程添加到运行队列。调用skew_heap_insert，将运行队列rq的运行进程池与新进程proc合并；将新进程的时间片初始化为最大值，关联proc->rq到rq，最后将rq的进程数目加1.

```
static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
     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;
     }
     proc->rq = rq;
     rq->proc_num ++;
}
```

3. stride_dequeue：该函数从运行队列中删除一个进程。只需调用skew_heap_remove，将进程proc从运行队列rq中删除，并取消proc->rq到rq的关联，最后将rq的进程数目减1.

```
static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
     rq->lab6_run_pool =
          skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
     rq->proc_num --;
}
```

4. stride_pick_next：选择下一个要调度的进程，其实就保存在运行队列的头部。因此只需要调用le2proc根据rq->lab6_run_pool找到对应进程的地址，然后要更新对应进程的stride。

```
static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
     if (rq->lab6_run_pool == NULL) return NULL;
     struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);
     if (p->lab6_priority == 0)
          p->lab6_stride += BIG_STRIDE;
     else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
     return p;
}
```

5. stride_proc_tick：如果进程的时间片大于0，则将其减1。减1后如果等于0，则设置need_resched为1，表示该进程需要被调度出去。

```
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;
     }
}
```

执行：make grade，check result都显示OK，但是check output会有问题，这可能是因为指导书使用的是 qemu-1.0.1，我的虚拟机用的是qemu-4.0.1，因为这个与本次实验内容无太大关系，所以没有更多地去修改。

```
pcy@ubuntu:~/Desktop/labcodes/lab6$ make grade
badsegment:              (3.2s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

divzero:                 (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

softint:                 (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

faultread:               (1.4s)
  -check result:                             OK
  -check output:                             WRONG
   !! error: missing 'page fault at 0x00000100: K/W [no page found].'

faultreadkernel:         (1.4s)
  -check result:                             OK
  -check output:                             WRONG
```

如果只是priority.c过不去，可执行 make run-priority 命令来单独调试它。