武汉大学国家网络安全学院实验报告

课程名称	操作系统设计与实践			成　　绩		教师签名	
实验名称	综合装配 & 安全分析			实验序号	final	实验日期	2020.11.26
姓　　名	唐浩淼	学号	2019302141024	专　　业	信息安全	年级班级	19 级 8 班

一、 个人工作概述

在本次实验中，我主要负责实现 PartA 中的多进程管理部分。实验目标是实现一个多级反馈队列调度算法，并用其尝试调度 5-8 个任务，输出性能评价信息。

二、 实验设计

多级反馈队列的实现是基于第六章 r 文件夹下的代码。多级反馈队列的算法描述如下：

- 进程在进入待调度的队列等待时，首先进入优先级最高的 Q_1 等待。
- 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q_1、Q_2、Q_3 三个队列，当且仅当在 Q_1 中没有进程等待时才去调度 Q_2，同理，只有 Q_1、Q_2 都为空时才会去调度 Q_3。
- 对于同一个队列中的各个进程，按照 FCFS 分配时间片调度。比如 Q_1 队列的时间片为 N，那么 Q_1 中的作业在经历了 N 个时间片后若还没有完成，则进入 Q_2 队列等待，若 Q_2 的时间片用完后作业还不能完成，一直进入下一级队列末尾，直至完成。
- 在最后一个队列 Q_n 中的各个进程，按照时间片轮转分配时间片调度。
- 在低优先级的队列中的进程在运行时，又有新到达的作业，此时须立即把正在运行的进程放回当前队列的队尾，然后把处理机分给高优先级进程。换而言之，任何时刻，只有当第 1 到 i-1 队列全部为空时，才会去执行第 i 队列的进程（抢占式）。特别说明，当再度运行到当前队列的该进程时，仅分配上次还未完成的时间片，不再分配该队列对应的完整时间片。

基于此，我们的设计思路是：

- 在进程表中加入所在队列以及所在队列的位置两个字段，这样就可以表示进程所在队列的位置。
- 一个进程不仅需要有一个总时间片了，还应该在进程表中加入在当前队列的剩余时间片。这样当在该队列剩余时间片为 0 的时候，就会被转移至下一个队列（除了在最后一个队列放在队尾）
- 利用 **clocker_handler()** 和 **schedule()** 函数配合实现多级反馈调度算法
  - 每次时钟中断检查当前进程是否还有总时间片，或者是否在当前队列还有时间片。如果有，那么总时间片和当前队列的时间片都需要减一；

– 如果没有，就需要调用 **schedule()** 函数处理该进程，把它放入下一个队列（或者当前队列队尾），并且选择下一个进程。

评价该算法性能的小程序需要打印响应时间、周转时间和等待时间等等信息。比较可惜的是，因为我们基于第六章实现的，在这里操作系统所有进程还是我们手工准备的，而不能进行 fork 中途加入进程。所以我们多级反馈队列相应也没有实现抢占式算法。这里我们的思路是可以再往进程表中加入提交时间、响应时间、等待时间、周转时间等性能字段。由于是手工准备的进程，那么提交时间都是 0 时刻；响应时间就是第一次处理该进程的时间，我们可以通过判断进程表的响应字段是否被初始化从而获得响应时间；周转时间是进程结束时刻减去提交时刻，结束时刻只需要在时钟中断中，判断到总时间片是否用完获得结束时刻；等待时间就是周转时间减去进程刚开始初始化的需要运行的时间片。

三、 实现过程

首先我们需要给我们进程表添加如下字段，queue 和 pos 表示该进程在第几个队列的第几个位置。ft、st 和 tt 分别表示在第一、二和三队列剩余的时间片。in_ticks、out_ticks 和 resp_ticks 是为了输出性能评价，分别表示提交时刻、结束时刻和响应时刻。

```
typedef struct s_proc {
  STACK_FRAME regs;      /* process registers saved in stack frame */

  u16 ldt_sel;           /* gdt selector giving ldt base and limit */
  DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */

       int ticks;              /* remained ticks */
       int priority;
    int queue;
    int pos;

    int ft;
    int st;
    int tt;

       int in_ticks;
       int out_ticks;
       int resp_ticks;

  u32 pid;               /* process id passed in from MM */
  char p_name[16];       /* name of the process */
}PROCESS;
```

我们还需要在 global.h 中定义如下字段。其中 first_num、second_num 和 third_num 表示每个队列当前进程的个数，定义这些字段可以更好对队列进行操作。first_ticks、second_ticks 和 third_ticks 指的是在每个队列的时间片。剩下两个是为了更好调试以及更好输出性能评价信息，再之后用到的时候在进行解释。

```
EXTERN int    first_num;
EXTERN int    second_num;
EXTERN int    third_num;
```

```
EXTERN int    first_ticks;
EXTERN int    second_ticks;
EXTERN int    third_ticks;

EXTERN int    dbg_disp_time;
EXTERN int    finish_proc_num;
```

上面的常数和进程的一些新的字段都需要在下面进行初始化。42-48 行初始化进程都在第一队列，并且在各个队列的剩余时间片都是满的，也就是 10、20 和 30（在 68-70 行初始化了每个队列有多少时间片）。并且定义了任务提交时间都是 0 时刻，也就是 os 刚启动的时刻。并且初始化响应时间为-1，这样如果后面处理了任务，但是发现响应时间还是-1，那就可以把当前时刻赋予给响应时间字段。56-62 行定义了一个任务的总时间片，64-66 初始化了每个队列当前有多少任务。

```
/*======================================================================*
                           kernel_main
 *======================================================================*/
PUBLIC int kernel_main()
{
  disp_str("-----\"kernel_main\" begins-----\n");

  TASK*  p_task   = task_table;
  PROCESS* p_proc  = proc_table;
  char*  p_task_stack = task_stack + STACK_SIZE_TOTAL;
  u16  selector_ldt = SELECTOR_LDT_FIRST;
  int i;
  for (i = 0; i < NR_TASKS; i++) {
    strcpy(p_proc->p_name, p_task->name); // name of the process
    p_proc->pid = i;   // pid

    p_proc->ldt_sel = selector_ldt;

    memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
          sizeof(DESCRIPTOR));
    p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
          sizeof(DESCRIPTOR));
    p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
    p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
      | SA_TIL | RPL_TASK;
    p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
      | SA_TIL | RPL_TASK;
    p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
      | SA_TIL | RPL_TASK;
    p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
      | SA_TIL | RPL_TASK;
    p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
      | SA_TIL | RPL_TASK;
    p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
      | RPL_TASK;
```

```
  p_proc->regs.eip = (u32)p_task->initial_eip;
  p_proc->regs.esp = (u32)p_task_stack;
  p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */

  p_proc->queue = 1;
  p_proc->pos = i;
  p_proc->ft = 10;
  p_proc->st = 20;
  p_proc->tt = 30;
      p_proc->in_ticks = 0;
      p_proc->resp_ticks = -1;

  p_task_stack -= p_task->stacksize;
  p_proc++;
  p_task++;
  selector_ldt += 1 << 3;
}

proc_table[0].ticks = proc_table[0].priority = 150;
proc_table[1].ticks = proc_table[1].priority = 120;
proc_table[2].ticks = proc_table[2].priority = 130;
proc_table[3].ticks = proc_table[3].priority = 110;
proc_table[4].ticks = proc_table[4].priority = 100;
proc_table[5].ticks = proc_table[5].priority = 80;
proc_table[6].ticks = proc_table[6].priority = 160;

first_num = 7;
second_num = 0;
third_num = 0;

first_ticks = 10;
second_ticks = 20;
third_ticks = 30;

k_reenter = 0;
ticks = 0;

p_proc_ready = proc_table;


  dbg_disp_time = 0;
  finish_proc_num = 0;

  disp_pos = 0;
  for (i = 0; i < 80 * 25; i++) {
      disp_str(" ");
  }
  disp_pos = 0;
  disp_queue();
```

```
      /* 初始化 8253 PIT */
      out_byte(TIMER_MODE, RATE_GENERATOR);
      out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
      out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));

      put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序 */
      enable_irq(CLOCK_IRQ);               /* 让8259A可以接收时钟中断 */

  restart();

 while(1){}
}
```

所有准备工作做完后，我们开始利用 cloker_handler 和 schedule 两个函数来完成多级反馈队列。首先每次产生时钟中断，都要首先看看当前进程的响应时间是否被赋值，如果没有就把当前时刻设为响应时间。然后 ticks++，k_reenter 解决中断重入问题。随后

- 判断当前进程在当前它所处的队列是否还有时间片，如果没有那么就要调用 schedule 函数进行调度。
- 否则再判断一下当前进程的总时间片是否用完了，如果用完了也要用 schudle 函数进行调度。
- 如果不满足上面两个情况，那么就让这个进程在当前队列的剩余时间片和总时间片都减 1。

```
PUBLIC void clock_handler(int irq)
{
   if (p_proc_ready->resp_ticks == -1) p_proc_ready->resp_ticks = ticks;
 ticks++;

 if (k_reenter != 0) {
  return;
 }

 if (p_proc_ready->ft > 0 && p_proc_ready->queue == 1) {
  if (p_proc_ready->ticks > 0) {
    p_proc_ready->ft--;
    p_proc_ready->ticks--;
    return;
  }
 }

 if (p_proc_ready->st > 0 && p_proc_ready->queue == 2) {
  if (p_proc_ready->ticks > 0) {
    p_proc_ready->st--;
    p_proc_ready->ticks--;
    return;
  }
 }
```

```
  if (p_proc_ready->tt > 0 && p_proc_ready->queue == 3) {
    if (p_proc_ready->ticks > 0) {
      p_proc_ready->tt--;
      p_proc_ready->ticks--;
      return;
    }
  }

  schedule();
}
```

调用了 schedule 函数，就说明总时间片或者在当前的队列的时间片用完了。于是我们先要处理这种情况，如果总时间片用完了，那么这个进程就需要被删除，同时记录该进程结束时刻到 out_ticks 字段中（finish_pro_num 是为了更好打印性能评价信息，记录了当前有几个进程结束了，如果所有进程结束了，那么他就会按顺序打印性能评价信息，在 7-25 行）。如果不是总时间片用完了，那么就是在当前队列的时间片用完了。在 1 和 2 队列的进程处理方式一致，都是先删除，然后我们再把进程放入下一个队列的末尾，放入末尾后，那个队列的进程数量也要加一。在第三个队列处理方式不同，它要把进程删除后重新放入第 3 个队列，并且还要重新把当前进程在第 3 队列的剩余时间片恢复。

处理完该进程后，我们要选择下一个运行的进程，方式就是从 1 队列到 3 队列、从头往后找，找到第一个进程，并且把当前运行进程 p_proc_ready 指向它。

```
PUBLIC void schedule()
{
  if (p_proc_ready->ticks == 0) {
    delete_proc();
    p_proc_ready->out_ticks = get_ticks();
    finish_proc_num++;
    if (finish_proc_num == NR_TASKS) {
      disp_str(" PROC_NAME | submit | finish | response | waiting |\n");
      int i = 1;
      for (PROCESS* p = proc_table; p < proc_table + NR_TASKS; p++) {
        disp_str("PROCESS ");
        disp_int(i);
        i++;
        disp_str("|  ");
        disp_int(p->in_ticks);
        disp_str(" |   ");
        disp_int(p->out_ticks);
        disp_str(" |   ");
        disp_int(p->resp_ticks);
        if (i == 2) disp_str(" ");
        disp_str(" |   ");
        disp_int(p->out_ticks - p->priority);
        disp_str(" |\n");
      }
    }
    // add_proc();
  } else if (p_proc_ready->queue == 1) {
    delete_proc();
```

```
    p_proc_ready->queue = 2;
    p_proc_ready->pos = second_num;
    second_num++;
  } else if (p_proc_ready->queue == 2) {
    delete_proc();
    p_proc_ready->queue = 3;
    p_proc_ready->pos = third_num;
    third_num++;
  } else { // if (p_proc_ready->queue == 3)
    delete_proc();
    p_proc_ready->queue = 3;
    p_proc_ready->pos = third_num;
    third_num++;
    p_proc_ready->tt = third_ticks;
  }

    if (finish_proc_num < NR_TASKS) {
    disp_queue();
    }

  // find next
  PROCESS* p;
  for (p = proc_table; p < proc_table + NR_TASKS; p++) {
    if (p->pos == 0 && p->queue == 1) {
      p_proc_ready = p;
      return;
    }
  }

  for (p = proc_table; p < proc_table + NR_TASKS; p++) {
    if (p->pos == 0 && p->queue == 2) {
      p_proc_ready = p;
      return;
    }
  }

  for (p = proc_table; p < proc_table + NR_TASKS; p++) {
    if (p->pos == 0 && p->queue == 3) {
      p_proc_ready = p;
      return;
    }
  }
}
```

上面多次提到了删除进程的函数，其具体实现如下所示。首先当前进程所在队列的进程个数要减 1，然后把当前进程的队列字段改为 0。随后我们还需要遍历所有进程，把和删除进程所在相同队列的进程的位置都减 1，也就是往前挪 1 位。

```
PRIVATE void delete_proc() {
  int queue = p_proc_ready->queue;
```

```
  if (queue == 1) {
    first_num--;
  } else if (queue == 2) {
    second_num--;
  } else { // if (queue == 3)
    third_num--;
  }
  p_proc_ready->queue = 0;

  PROCESS* p;
  for (p = proc_table; p < proc_table + NR_TASKS; p++) {
    if (p->queue == queue) p->pos--;
  }
}
```

四、 实验心得总结

这部分任务的要点在于在 clock() 和 schedule() 中实现多集反馈队列调度算法，并适当添加或修改进程表对应的字段。在我们的设计中，多级队列并没有对应的内存实体，而是通过进程表中的字段标示出进程在多级队列中的位置，以此来“虚拟”表示多级队列，因此无需添加新的数据结构或模块。总体而言，这部分任务的思路是比较清晰的，但是有很多细节需要处理，例如每当为进程表添加新的字段，就需要添加对应的初始化操作来与之配合。另外，为了能够打印性能评价细信息，我们也需要向进程表中添加提交时间、响应时间、等待时间、周转时间等性能字段，这些字段的想要顺利与程序配合和也要经过细致的调整。实验的过程也是对多级反馈队列调度算法的逐步回顾，加深了我对于进程调度过程的理解。