**实验四**

练习**3:** 阅读分析源代码理解进程**fork/exec/wait/exit** 的实现，以及系统调用的实现（不需要编码）

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

1. 请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的？

答:

fork将创建新的子线程，将子线程的状态由UNINIT态变为RUNNABLE态，不改变父进程的状态  
  
exec完成用户进程的创建工作，同时使用户进程进入执行，不改变进程状态  
  
wait完成子进程资源回收，如果有已经结束的子进程或者没有子进程，那么调用会立刻结束，不影响进程状态；否则，进程需要等待子进程结束，进程从RUNNIG态变为SLEEPING态。  
  
exit完成对资源的回收，进程从RUNNIG态变为ZOMBIE态。

2.请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。（使用的是qemu- 1.0.1）

答：

创建进程 +---wait()--------------------------RUNNING------------------------+

| | ^ | |

| | | | |

alloc\_page() | proc\_run() exit()

| | | | |

V | | v v

UNINIT ------wakeup\_proc()--> RUNNABLE --exit()--> ZOMBIE--父进程调用wait()-->

| ^ ^

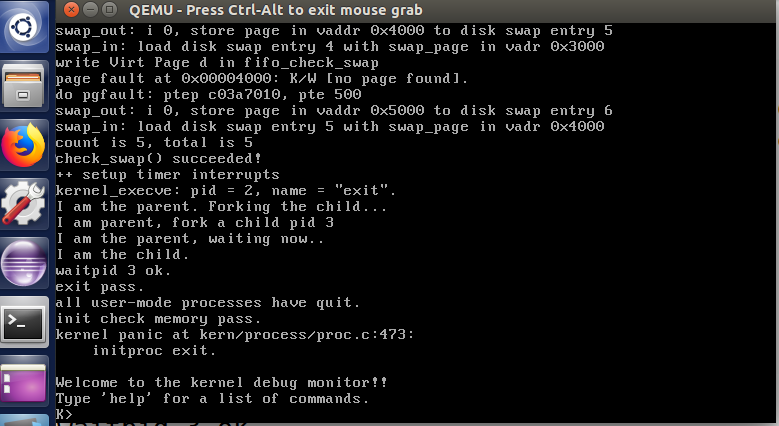
| | |

| 子进程调用exit() exit()

| | |

| | |

+---wait()------------->SLEEPING---------------------------+



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：

1.请理解并分析sched\_calss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

答：

RR算法的核心是设置一个时间片，执行过程中，如果进程只需要小于时间片的CPU区间，则进程完成后释放CPU。否则定时器中断并产生系统中断，进行上下文切换，将进程加入到就绪队列的尾部，然后从其头部取出进程进行调度。

struct sched\_class {

//调度器的名字

const char \*name;

//初始化运行队列

void (\*init)(struct run\_queue \*rq);

// 将进程p插入到运行队列q中

void (\*enqueue)(struct run\_queue \*rq, struct proc\_struct \*proc);

// 将进程p从运行队列q中删除

void (\*dequeue)(struct run\_queue \*rq, struct proc\_struct \*proc);

// 返回运行队列中下一个可执行的过程

struct proc\_struct \*(\*pick\_next)(struct run\_queue \*rq);

// timetick的处理函数

void (\*proc\_tick)(struct run\_queue \*rq, struct proc\_struct \*proc);

};

ucore通过schedule()函数进行调度，我们细看一下schedule函数的框架：

Void schedule(void) {

bool intr\_flag;

struct proc\_struct \*next;

local\_intr\_save(intr\_flag);

{

current->need\_resched = 0;

//插入运行队列

if (current->state == PROC\_RUNNABLE) {

sched\_class\_enqueue(current);

}

//选取下一个运行的进程并将其从队列中删去

if ((next = sched\_class\_pick\_next()) != NULL) {

sched\_class\_dequeue(next);

}

if (next == NULL) {

next = idleproc;

}

next->runs ++;

if (next != current) {

proc\_run(next);

}

}

local\_intr\_restore(intr\_flag);

}

在内核初始化时调用sched\_init函数选择调度器并对调度器初始化。每当进程被唤醒则调用enqueue函数将其加入调度器等待调度的进程队列。每当发生进程切换时，首先调用enqueue将当前正在运行的进程加入等待调度的进程队列，然后调用pick\_next函数获取接下来将执行的进程，并调用dequeue将被选中即将执行的进程从调度队列中移除。每当产生时钟中断时，需要调用proc\_tick函数来更新调度器的时钟信息。

1. 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

答：

---维护三个优先级队列，可以将之前的run\_list用run\_list[3]来维护，其中run\_list[0]表示最高优先级队列，run\_list[2]表示最低优先级队列。将进程的初始态也划分为3个优先级，初始化加入队列时加入相应的优先级，通常进程可以直接加入最高优先级，随着降级逐渐降至低优先级。

---初始化时，将每个队列置空即可。并将每个队列对应的proc\_num置0。

---enqueue时，判断进程的时间片是否为0，如果是0，说明该进程应当降级，将其priority+1，否则的话不需要改变。然后将该进程根据其优先级加入到对应的队列中。

---dequeue时，将进程从相应的优先级队列中删去即可，没有太大变化。

---pick\_next时，通过算法选出下一个要执行的进程。在这里为了避免出现低优先级进程饥饿现象，可以设置为高优先级每处理100个单位时间后，低优先级处理30个单位时间 类似的思想。用这个方法来避免低优先级进程饥饿现象。

---proc\_tick时钟中断所使用,每次时钟中断，减少当前进程的时间片。若为0，则将进程标记为需要调度。

---具体实现时，将run\_list[0]队列采用时间片为8ms的RR算法(因为绝大多数进程所需的CPU时间是小于8ms的)，将run\_list[1]队列采用时间片为16ms的RR算法，将run\_list[2]队列采用FCFS算法。

---每次选取执行的队列时，可以用概率或者时分的方法将优先级进行区分。