**练习3: 阅读分析源代码，理解进程执行 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：**

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

**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）**

fork:

首先当程序执行fork时，fork使用了系统调用SYS\_fork,而系统调用SYS\_fork则主要是由do\_fork和wakeup\_proc来完成的。主要是完成了以下工作：

1. 分配并初始化进程控制块(alloc\_proc 函数);
2. 分配并初始化内核栈(setup\_stack 函数);
3. 根据clone\_flag标志复制或共享进程内存管理结构(copy\_mm 函数);
4. 设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy\_thread 函数);
5. 把设置好的进程控制块放入hash\_list 和 proc\_list 两个全局进程链表中;
6. 自此,进程已经准备好执行了,把进程状态设置为“就绪”态;
7. 设置返回码为子进程的 id 号。

exec:

当应用程序执行的时候，会调用SYS\_exec系统调用,而当ucore收到此系统调用的时候，则会使用do\_execve()函数来实现，因此这里我们主要介绍do\_execve()函数的功能，函数主要时完成用户进程的创建工作，同时使用户进程进入执行。 主要完成了以下工作：

1. 首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。
2. 接下来是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。之后就是调用load\_icode从而使之准备好执行.

wait:

当执行wait功能的时候，会调用系统调用SYS\_wait，而该系统调用的功能则主要由do\_wait函数实现，主要工作就是父进程如何完成对子进程的最后回收工作，即：

1. 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;
2. 如果此子进程的执行状态不为PROC\_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC\_SLEEPING（睡眠），睡眠原因为WT\_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;
3. 如果此子进程的执行状态为 PROC\_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc\_list和hash\_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。

exit:

当执行exit功能的时候，会调用系统调用SYS\_exit，而该系统调用的功能主要是由do\_exit函数实现。具体过程如下：

1. 先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间;
2. 设置当前进程的中hi性状态为PROC\_ZOMBIE，然后设置当前进程的退出码为error\_code。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块）
3. 如果当前父进程已经处于等待子进程的状态，即父进程的wait\_state被置为WT\_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。
4. 如果当前进程还有子进程,则需把这些子进程的父进程指针设置为内核线程init,且各个子进程指针需要插入到init的子进程链表中。
5. 如果某个子进程的执行状态是 PROC\_ZOMBIE,则需要唤醒 init来完成对此子进程的最后回收工作。 执行schedule()调度函数，选择新的进程执行。
6. 所以说该函数的功能简单的说就是，回收当前进程所占的大部分内存资源,并通知父进程完成最后的回收工作。

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

当程序执行fork(),exec(),wait(),exit()函数时，会调用sys\_xxxx（）函数，这些函数又会调用syscall()函数，syscall()函数嵌入了内联汇编指令，int产生系统调用与中断。  
而kernel则会对这些系统调用进行统一的处理。这些函数会产生中断，如果此时current -> need\_resched ==1,则会进行调度。  
**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）。**

进程创建(fork()函数) -> 进程就绪（proc -> state == RUNNABLE）-> 进程执行（schedule()函数) -> 进程退出（do\_exit()） -> 进程结束(do\_wait()回收kstack和proc\_struct)

**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**

sched\_calss中各个函数指针的用法：

1. void (\*init)(struct run\_queue \*rq)：初始化run queue队列 2.void(\*enqueue)(struct run\_queue \*rq, struct proc\_struct \*proc)：将当前的线程入队

3.void (\*dequeue)(struct run\_queue \*rq, struct proc\_struct \*proc)：将proc线程出队

4.struct proc\_struct \*(\*pick\_next)(struct run\_queue \*rq)：在rq队列中选择一个符合条件的队列。

5.void (\*proc\_tick)(struct run\_queue \*rq, struct proc\_struct \*proc)：在产生时钟中断时将proc的时间片减少，并对其进行判断是否需要调度。

ucore的调度过程：ucore通过schedule()函数进行调度，在这个函数中，当需要进行线程的切换时，他会先将当前的线程通过sched\_class\_enqueue()函数加入队列，然后通过sched\_class\_pick\_next()函数选择下一个要执行的线程并将其从队列中删除。

**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.进程在进入待调度的队列等待时，首先进入优先级最高的Q1队列等待；

2.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；

3.对于同一个队列中的各个进程，按照时间片轮转法调度。比如Q1队列的时间片为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成；

4.在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配给新到达的作业。