实验五：用户进程管理

一、实验目的

1、了解第一个用户进程创建过程

2、了解系统调用框架的实现机制

3、了解ucore如何实现系统调用sys\_fork/sys\_exec/sys\_exit/sys\_wait来进行进程管理

二、实验内容

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）

fork：在执行了fork系统调用之后，会执行正常的中断处理流程，最终将控制权转移给syscall，之后根据系统调用号执行sys\_fork函数，进一步执行了上文中的do\_fork函数，完成新的进程的进程控制块的初始化、设置、以及将父进程内存中的内容到子进程的内存的复制工作，然后将新创建的进程放入可执行队列（runnable），这样的话在之后就有可能由调度器将子进程运行起来了；

exec：在执行了exec系统调用之后，会执行正常的中断处理流程，最终将控制权转移给syscall，之后根据系统调用号执行sys\_exec函数，进一步执行了上文中的do\_execve函数，在该函数中，会对内存空间进行清空，然后将新的要执行的程序加载到内存中，然后设置好中断帧，使得最终中断返回之后可以跳转到指定的应用程序的入口处，就可以正确执行了；

wait：在执行了wait系统调用之后，会执行正常的中断处理流程，最终将控制权转移给syscall，之后根据系统调用号执行sys\_wait函数，进一步执行了的do\_wait函数，在这个函数中，将搜索是否指定进程存在着处于ZOMBIE态的子进程，如果有的话直接将其占用的资源释放掉即可；如果找不到这种子进程，则将当前进程的状态改成SLEEPING态，并且标记为等待ZOMBIE态的子进程，然后调用schedule函数将其当前线程从CPU占用中切换出去， exit：在执行了exit系统调用之后，会执行正常的中断处理流程，最终将控制权转移给syscall，之后根据系统调用号执行sys\_exit函数，进一步执行了的do\_exit函数，首先将释放当前进程的大多数资源，然后将其标记为ZOMBIE态，然后调用wakeup\_proc函数将其父进程唤醒（如果父进程执行了wait进入SLEEPING态的话），然后调用schedule函数，让出CPU资源，等待父进程进一步完成其所有资源的回收；

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？

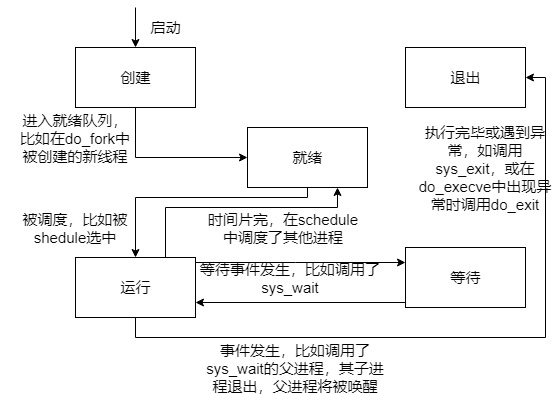
fork不会影响当前进程的执行状态，但是会将子进程的状态标记为RUNNALB，使得可以在后续的调度中运行起来；

exec不会影响当前进程的执行状态，但是会修改当前进程中执行的程序；

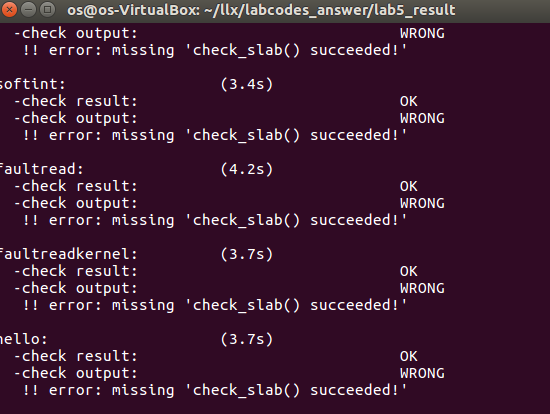
wait系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为SLEEPING态，等待执行了exit的子进程将其唤醒；

exit会将当前进程的状态修改为ZOMBIE态，并且会将父进程唤醒（修改为RUNNABLE），然后主动让出CPU使用权；

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）



实验六: 调度器

一、实验目的

1、理解操作系统的调度管理机制

2、熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法

3、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

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：

init函数：这个函数被封装为sched\_init函数，用于调度算法的初始化，使用grep命令可以知道，该函数仅在ucore入口的init.c里面被调用进行初始化；

enqueue函数：该函数的功能为将指定的进程的状态置成RUNNABLE，并且放入调用算法中的可执行队列中，被封装成sched\_class\_enqueue函数，可以发现这个函数仅在wakeup\_proc和schedule函数中被调用，前者为将某个不是RUNNABLE的进程加入可执行队列，而后者是将正在执行的进程换出到可执行队列中去；

dequeue函数：该函数的功能为将某个在队列中的进程取出，其封装函数sched\_class\_dequeue仅在schedule中被调用，表示将调度算法选择的进程从等待的可执行的进程队列中取出进行执行；

pick\_next函数：该函数的封装函数同样仅在schedule中被调用，功能为选择要执行的下个进程；

proc\_tick函数：该函数表示每次时钟中断的时候应当调用的调度算法的功能，仅在进行时间中断的ISR中调用；

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

· 在ucore中调用调度器的主体函数（不包括init，proc\_tick）的代码仅存在在wakeup\_proc和schedule，前者的作用在于将某一个指定进程放入可执行进程队列中，后者在于将当前执行的进程放入可执行队列中，然后将队列中选择的下一个执行的进程取出执行；

· 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；这就是具体的Round-Robin enqueue函数的实现；

· 当需要将某一个进程从就绪队列中取出的时候，只需要将其直接删除即可；

· 当需要取出执行的下一个进程的时候，只需要将就绪队列的队头取出即可；

· 每当出现一个时钟中断，则会将当前执行的进程的剩余可执行时间减1，一旦减到了0，则将其标记为可以被调度的，这样在ISR中的后续部分就会调用schedule函数将这个进程切换出去；

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

· 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，为其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；

· 处理调度算法初始化的时候需要同时对N个队列进行初始化；

· 在处理将进程加入到就绪进程集合的时候，观察这个进程的时间片有没有使用完，如果使用完了，就将所在队列的优先级调低，加入到优先级低1级的队列中去，如果没有使用完时间片，则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；

· 在选择下一个执行的进程的时候，有限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；（有可能导致饥饿）

· 从就绪进程集合中删除某一个进程就只需要在对应队列中删除即可；

· 处理时间中断的函数不需要改变；

· 至此完成了多级反馈队列调度算法的具体设计；

练习2: 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。

后面的实验文档部分给出了Stride调度算法的大体描述。这里给出Stride调度算法的一些相关

的资料（目前网上中文的资料比较欠缺）。

strid-shed paper location1

strid-shed paper location2  
也可GOOGLE “Stride Scheduling” 来查找相关资料

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。如果只是priority.c

过不去，可执行 make run-priority 命令来单独调试它。大致执行结果可看附录。（ 使用的是

qemu-1.0.1 ）。

请在实验报告中简要说明你的设计实现过程。

Stride算法的原理是对于每一个进程，有一个stride值和一个pass值。每次进行调度时，选择stride最小的进程运行，并将这个进程的stride加上pass。pass越小那么被调度的次数就会越多。在实验中，pass依托优先级实现。优先级越大，pass即用一个大常数除以优先级得到的值就越小，也就意味着被调度的次数越多。具体实现如下

* 首先，需要设置一个大整数，将来的所有pass值都由这个大整数除以进程的优先级得到：

#define BIG\_STRIDE 0x7FFFFFFF

* 初始化时，需要将列表、斜堆清空，并置运行列表中的进程数为0。

static voidstride\_init(struct run\_queue \*rq) {// (1) init the ready process list: rq->run\_list

list\_init(&(rq -> run\_list));// (2) init the run pool: rq->lab6\_run\_pool

rq -> lab6\_run\_pool = NULL;// (3) set number of process: rq->proc\_num to 0

rq -> proc\_num = 0;}

* 将进程加入运行队列时，插入到斜堆中，并将运行队列的进程计数加一，同时需要在进程的数据结构中关联运行队列。

static voidstride\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {#if USE\_SKEW\_HEAP// (1) insert the proc into rq correctly

rq -> lab6\_run\_pool = skew\_heap\_insert(rq -> lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);#else

assert(list\_empty(&(proc->run\_link)));

list\_add\_before(&(rq->run\_list), &(proc->run\_link));#endif// (2) recalculate proc->time\_slice

if (proc->time\_slice == 0 || proc->time\_slice > rq->max\_time\_slice) {

proc->time\_slice = rq->max\_time\_slice;

}// (3) set proc->rq pointer to rq

proc -> rq = rq;// (4) increase rq->proc\_num

rq -> proc\_num ++;}

* 将进程从运行队列移走时，需要将进程从斜堆中删除，并将运行队列的进程计数减一。

static voidstride\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc) {#if USE\_SKEW\_HEAP// (1) remove the proc from rq correctly

rq -> lab6\_run\_pool = skew\_heap\_remove(rq -> lab6\_run\_pool, &(proc -> lab6\_run\_pool), proc\_stride\_comp\_f);#else

assert(!list\_empty(&(proc->run\_link)) && proc->rq == rq);

list\_del\_init(&(proc->run\_link));#endif// (2) decrease rq -> proc\_num by 1

rq -> proc\_num --;}

* 选择接下来调度的进程时，如果使用斜堆只需要取出堆顶，随后要更新进程的stride值。

static struct proc\_struct \*stride\_pick\_next(struct run\_queue \*rq) {

struct proc\_struct \*proc;#if USE\_SKEW\_HEAP

if (rq -> lab6\_run\_pool == NULL) {

return NULL;

}// (1) get a proc\_struct pointer p with the minimum value of stride

proc = le2proc(rq -> lab6\_run\_pool, lab6\_run\_pool);#else

list\_entry\_t \*le = list\_next(&(rq->run\_list));

if (le == &(rq->run\_list)) {

return NULL;

}

proc = le2proc(le, run\_link);

while (le != &(rq -> run\_list)) {

struct proc\_struct \*temp = le2proc(le, run\_link);

if (proc\_stride\_comp\_f(temp, proc) < 0) {

proc = temp;

}

le = list\_next(le);

}#endif// (2) update p;s stride value: p->lab6\_stride

if (proc -> lab6\_priority == 0) {

proc -> lab6\_stride += BIG\_STRIDE;

} else {

proc -> lab6\_stride += BIG\_STRIDE / proc -> lab6\_priority;

}

return proc;}

* 处理时钟和RR算法一样，如果time slice大于0，则将值减一。否则以为着时间片用完，需要调度。

static voidstride\_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;

}}