# 操作系统实验三

### 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 行线程管理
4. 理解操作系统的调度管理机制
5. 熟悉 ucore 的系统调度器框架，以及缺省的Round-Robin 调度算法
6. 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

1. 创建用户进程，让用户进程在用户态执行;
2. 支持ucore的系统调用为用户进程提供服务；
3. 完成对用户进程的执行过程的基本管理;

### 三、实验过程

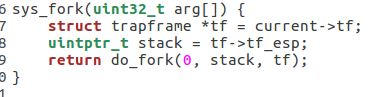
1. **Lab5-练习3**

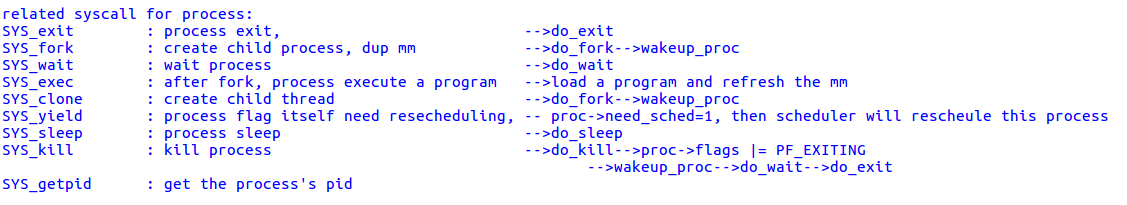
阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）请在实验报告中简要说明你对fork/exec/wait/exit函数的分析。并回答如下问题：

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？
2. 请给出ucore中一个用户态进程的执行状态生命周期图（包括执行状态，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）

**问题一回答：**

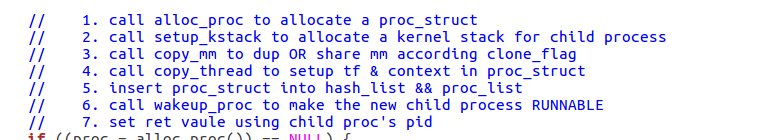
1. 对于fork函数：





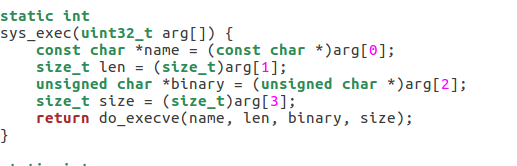
执行fork时，fork使用了系统调用SYS\_fork,而系统调用SYS\_fork则主要是由do\_fork和wakeup\_proc来完成的。

对于do\_fork函数的作用，由注释可知：

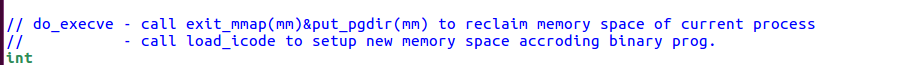


1. 分配并初始化进程控制块(alloc\_proc 函数);
2. 分配并初始化内核栈(setup\_stack 函数);
3. 根据clone\_flag标志复制或共享进程内存管理结构(copy\_mm 函数);
4. 设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy\_thread 函数);
5. 把设置好的进程控制块插入hash\_list 和 proc\_list 两个全局进程链表中;
6. 把进程状态设置为“就绪”态;
7. 设置返回码为子进程的 id 号。
8. 对于exec函数：

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，主要工作如下：







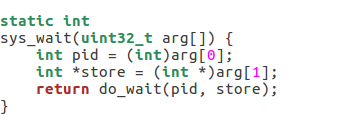
从代码中可以了解到：

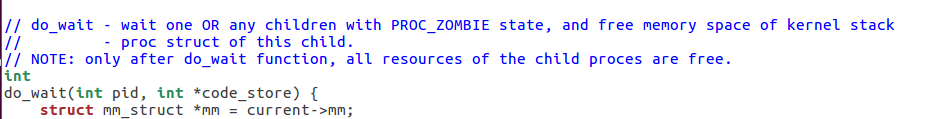
do\_execve()函数的功能，主要是完成用户进程的创建工作，同时使用户进程进入执行。

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。
3. 对于wait函数

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，具体的功能实现如下：





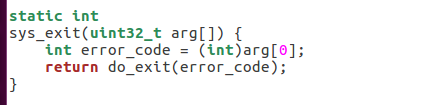


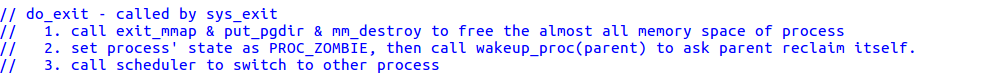
1. 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;
2. 如果此子进程的执行状态不为PROC\_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC\_SLEEPING（睡眠），睡眠原因为WT\_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;
3. 如果此子进程的执行状态为 PROC\_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc\_list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。

4. 对于exit函数

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）

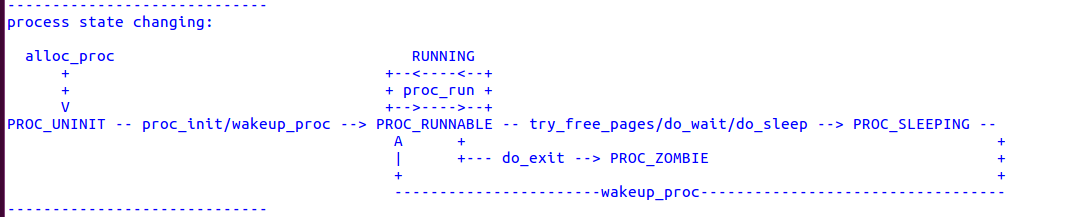
③如果当前父进程已经处于等待子进程的状态，即父进程的wait\_state被置为WT\_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。

④如果当前进程还有子进程,则需要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。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。

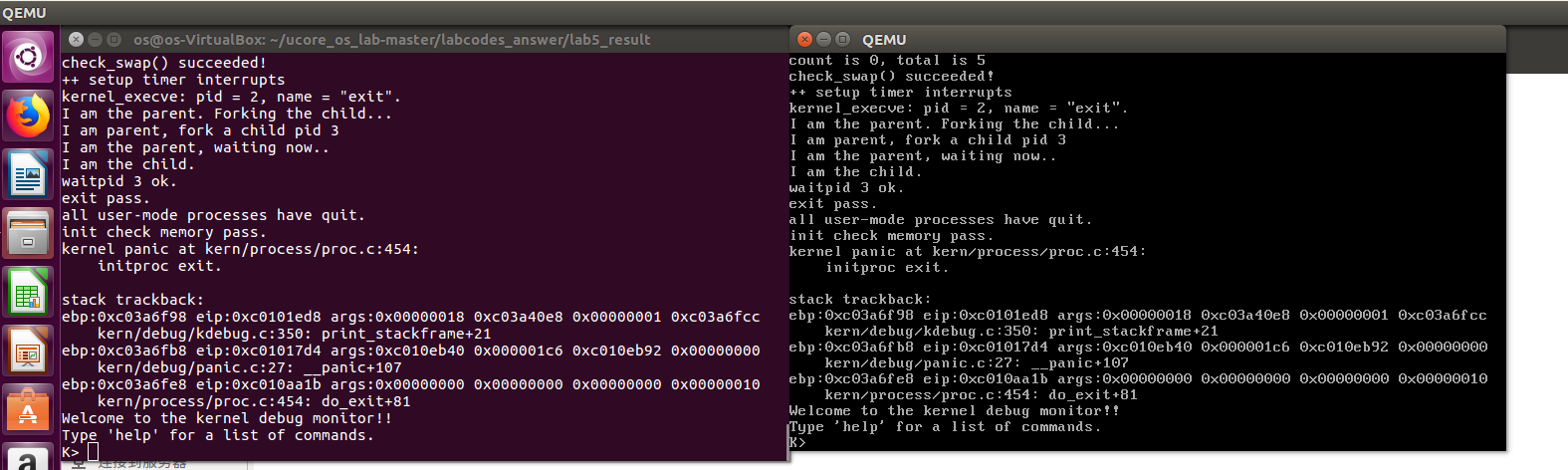
⑤执行schedule()调度函数，选择新的进程执行。

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。

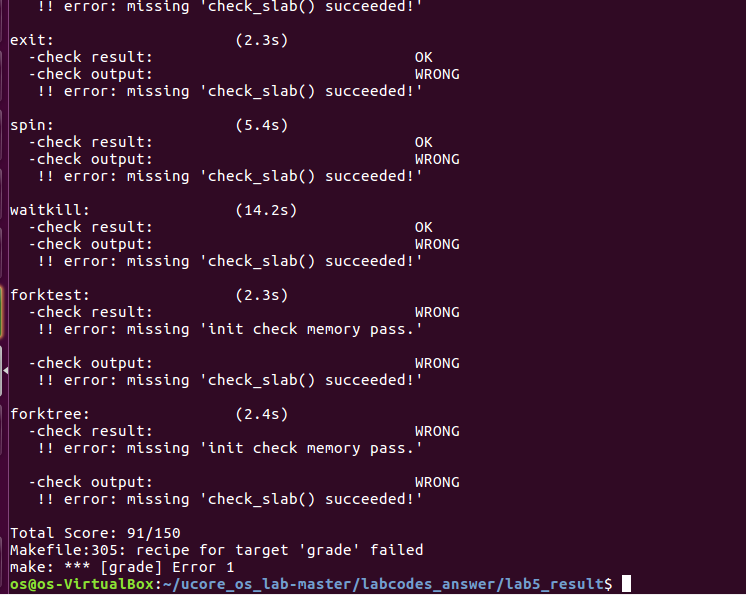
**问题二回答：**



输入make qemu执行结果如下：



输入make grade执行结果如下：



1. **Lab6-练习1**

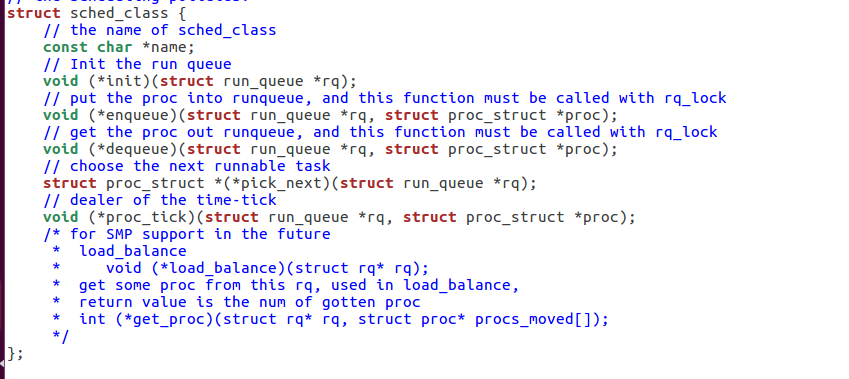
使用 Round Robin 调度算法（不需要编码）比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。

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 调度算法描述ucore的调度执行过程
2. 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

**问题一回答：**

在schedule的头文件中，可以看到sched\_class的数据结构，如下图所示：



观察sched.c文件可知：

1. init函数：完成对进程队列的初始化
2. 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；
3. 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；
4. 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；
5. 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；

**ucore调度执行过程**：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。

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，因此第一个内核线程idleproc产生。

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。

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。

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。

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。

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。

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。什么时候睡醒呢？等子进程userproc运行结束后再将其唤醒。

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。

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。

**问题二回答：**

多级反馈队列算法可以实现在不同的队列间使用不同的调度算法。

假设进程一共有4个调度优先级，分别为0、1、2、3，其中0位最高优先级，3位最低优先级。为了支持4个不同的优先级，在运行队列中开4个队列，分别命名为rq -> run\_list[0..3]。除此之外，在proc\_struct中加入priority成员表示该进程现在所处的优先级，初始化为0。

MLFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc)：判断proc进程的时间片proc -> time\_slice是否为0，如果为0，则proc -> priority += 1，否则不变。根据proc加入到对应优先级的列表中去。时间片的长度也和优先级有关，低优先级的时间片长度设置为高优先级的两倍。

static void

MLFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

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

if (proc -> time\_slice == 0 && proc -> priority != 3) {

++(proc -> priority);

}

list\_add\_before(&(rq->run\_list[proc ->priority]), &(proc->run\_link));

proc->time\_slice = (rq->max\_time\_slice << proc -> priority);

proc->rq = rq;

rq->proc\_num ++;

}

MLFQ\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc)：将proc进程从相应的优先级运行队列中删除。

static void

MLFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

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

list\_del\_init(&(proc->run\_link));

rq->proc\_num --;

}

MLFQ\_pick\_next(struct run\_queue \*rq)：为了避免优先级较低的进程出现饥饿现象，对每个优先级设置一定的选中概率，高优先级是低优先级选中概率的两倍，然后选出一个优先级，找到这个优先级中的第一个进程返回

static struct proc\_struct \*

MLFQ\_pick\_next(struct run\_queue \*rq) {

int p = rand() % (8 + 4 + 2 + 1);

int priority;

if (p >= 0 && p < 8) {

priority = 0;

} else if (p >= 8 && p < 12) {

priority = 1;

} else if (p >= 12 && p < 14) {

priority = 2;

} else priority = 3;

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

if (le != &(rq->run\_list[priority])) {

return le2proc(le, run\_link);

} else {

for (int i = 0; i < 3; ++i) {

le = list\_next(&(rq->run\_list[i]));

if (le != &(rq -> run\_list[i])) return le2proc(le, run\_link);

}

}

return NULL;

}

MLFQ\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc)：和RR算法相似。