Lab 5

练习3

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。do\_fork()完成的工作在lab4的时候已经做过详细介绍，这里再简单说一下，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。

而wakeup\_proc函数主要是将进程的状态设置为等待，即proc->wait\_state = 0，此处不赘述。

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。

主要工作如下：

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，完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间。

  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。

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;（具体的回收过程不作详细说明）

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。

关于系统调用

ucore所有的系统调用

与用户态的函数库调用执行过程相比，系统调用执行过程的有四点主要的不同：

不是通过“CALL”指令而是通过“INT”指令发起调用；

不是通过“RET”指令，而是通过“IRET”指令完成调用返回；

当到达内核态后，操作系统需要严格检查系统调用传递的参数，确保不破坏整个系统的安全性；

执行系统调用可导致进程等待某事件发生，从而可引起进程切换；

根据之前的分析，应用程序调用的 exit/fork/wait/getpid 等库函数最终都会调用 syscall 函数,只是调用的参数不同而已（分别是 SYS\_exit / SYS\_fork / SYS\_wait / SYS\_getid ）

当调用系统函数时，一般执行INT T\_SYSCALL指令后，CPU 根据操作系统建立的系统调用中断描述符，转入内核态，然后开始了操作系统系统调用的执行过程，在执行之前，会保留系统调用前的执行现场，然后保存当前进程的trapframe中，之后操作系统就可以开始完成具体的系统调用服务，完成服务后，调用IRET，CPU根据内核栈的情况恢复到用户态，并把EIP指向tf\_eip的值。这样整个系统调用就执行完毕了。

Lab6

练习1 使用Round Robin调度算法

Round Robin调度算法的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。Round Robin 调度器维护当前 runnable进程的有序运行队列。当前进程的时间片用完之后,调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。

在这个理解的基础上，我们来分析算法的具体实现。

这里Round Robin调度算法的主要实现在default\_sched.c之中，源码如下：

static void

RR\_init(struct run\_queue \*rq) {

list\_init(&(rq->run\_list));

rq->proc\_num = 0;

}

static void

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

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

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

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

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

}

proc->rq = rq;

rq->proc\_num ++;

}

static void

RR\_dequeue(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 --;

}

static struct proc\_struct \*

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

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

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

return le2proc(le, run\_link);

}

return NULL;

}

static void

RR\_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;

}

}

struct sched\_class default\_sched\_class = {

.name = "RR\_scheduler",

.init = RR\_init,

.enqueue = RR\_enqueue,

.dequeue = RR\_dequeue,

.pick\_next = RR\_pick\_next,

.proc\_tick = RR\_proc\_tick,

};

现在我们来逐个函数的分析，从而了解Round Robin调度算法的原理。

首先是RR\_init函数，函数比较简单，不再罗列，完成了对进程队列的初始化。

然后是RR\_enqueue函数，

static void RR\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

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

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

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

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

}

proc->rq = rq;

rq->proc\_num ++;

}

看代码，首先，它把进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为max\_time\_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加一。

然后是RR\_dequeue函数

static void

RR\_dequeue(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 --;

}

即简单的把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的proc\_num减一。

接下来是RR\_pick\_next函数。

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

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

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

return le2proc(le, run\_link);

}

return NULL;

}

选取函数，即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。

最后是

static void RR\_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;

}

}