调度并执行内核线程 **initproc**

在uCore执行完proc_init函数后，就创建好了两个内核线程：idleproc和initproc，这时uCore当前的执行现场就是idleproc，等到执行到init函数的最后一个函数cpu_idle之前，uCore的所有初始化工作就结束了，idleproc将通过执行cpu_idle函数让出CPU，给其它内核线程执行，具体过程如下：

```
void
cpu_idle(void) {
        while (1) {
                if (current->need_resched) {
                        schedule();
                        ……
```

首先，判断当前内核线程idleproc的need_resched是否不为0，回顾前面“创建第一个内核线程idleproc”中的描述，proc_init函数在初始化idleproc中，就把idleproc->need_resched置为1了，所以会马上调用schedule函数找其他处于“就绪”态的进程执行。

uCore在实验四中只实现了一个最简单的FIFO调度器，其核心就是schedule函数。它的执行逻辑很简单：

1．设置当前内核线程current->need_resched为0； 2．在proc_list队列中查找下一个处于“就绪”态的线程或进程next； 3．找到这样的进程后，就调用proc_run函数，保存当前进程current的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换。

至此，新的进程next就开始执行了。由于在proc10中只有两个内核线程，且idleproc要让出CPU给initproc执行，我们可以看到schedule函数通过查找proc_list进程队列，只能找到一个处于“就绪”态的initproc内核线程。并通过proc_run和进一步的switch_to函数完成两个执行现场的切换，具体流程如下：

1. 让current指向next内核线程initproc；
2. 设置任务状态段ts中特权态0下的栈顶指针esp0为next内核线程initproc的内核栈的栈顶，即next->kstack + KSTACKSIZE ；
3. 设置CR3寄存器的值为next内核线程initproc的页目录表起始地址next->cr3，这实际上是完成进程间的页表切换；
4. 由switch_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当switch_to函数执行完“ret”指令后，就切换到initproc执行了。

注意，在第二步设置任务状态段ts中特权态0下的栈顶指针esp0的目的是建立好内核线程或将来用户线程在执行特权态切换（从特权态0<-->特权态3，或从特权态3<-->特权态3）时能够正确定位处于特权态0时进程的内核栈的栈顶，而这个栈顶其实放了一个trapframe结构的内存空间。如果是在特权态3发生了中断/异常/系统调用，则CPU会从特权态3-->特权态0，且CPU从此栈顶（当前被打断进程的内核栈顶）开始压栈来保存被中断/异常/系统调用打断的用户态执行现场；如果是在特权态0发生了中断/异常/系统调用，则CPU会从从当前内核栈指针esp所指的位置开始压栈保存被中断/异常/系统调用打断的内核态执行现场。反之，当执行完对中断/异常/系统调用打断的处理后，最后会执行一个“iret”指令。在执行此指令之前，CPU的当前栈指针esp一定指向上次产生中断/异常/系统调用时CPU保存的被打断的指令地址CS和EIP，“iret”指令会根据ESP所指的保存的址CS和EIP恢复到上次被打断的地方继续执行。

在页表设置方面，由于idleproc和initproc都是共用一个内核页表boot_cr3，所以此时第三步其实没用，但考虑到以后的进程有各自的页表，其起始地址各不相同，只有完成页表切换，才能确保新的进程能够正常执行。

第四步proc_run函数调用switch_to函数，参数是前一个进程和后一个进程的执行现场：process context。在上一节“设计进程控制块”中，描述了context结构包含的要保存和恢复的寄存器。我们再看看switch.S中的switch_to函数的执行流程：

```
.globl switch_to
switch_to: # switch_to(from, to)
# save from's registers
movl 4(%esp), %eax # eax points to from
popl 0(%eax) # esp--> return address, so save return addr in FROM’s
context
movl %esp, 4(%eax)
……
movl %ebp, 28(%eax)
# restore to's registers
movl 4(%esp), %eax # not 8(%esp): popped return address already
# eax now points to to
movl 28(%eax), %ebp
……
movl 4(%eax), %esp
pushl 0(%eax) # push TO’s context’s eip, so return addr = TO’s eip
ret # after ret, eip= TO’s eip
```

首先，保存前一个进程的执行现场，前两条汇编指令（如下所示）保存了进程在返回switch_to函数后的指令地址到context.eip中

```
movl 4(%esp), %eax # eax points to from
popl 0(%eax) # esp--> return address, so save return addr in FROM’s
context
```

在接下来的7条汇编指令完成了保存前一个进程的其他7个寄存器到context中的相应成员变量中。至此前一个进程的执行现场保存完毕。再往后是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从context的高地址的成员变量ebp开始，逐一把相关成员变量的值赋值给对应的寄存器，倒数第二条汇编指令“pushl 0(%eax)”其实把context中保存的下一个进程要执行的指令地址context.eip放到了堆栈顶，这样接下来执行最后一条指令“ret”时，会把栈顶的内容赋值给EIP寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程了。uCore会执行进程切换，让initproc执行。在对initproc进行初始化时，设置了initproc->context.eip = (uintptr_t)forkret，这样，当执行switch_to函数并返回后，initproc将执行其实际上的执行入口地址forkret。而forkret会调用位于kern/trap/trapentry.S中的forkrets函数执行，具体代码如下：

```
.globl __trapret
 __trapret:
 # restore registers from stack
 popal
 # restore %ds and %es
 popl %es
 popl %ds
 # get rid of the trap number and error code
 addl $0x8, %esp
```

```
    iret
    .globl forkrets
    forkrets:
    # set stack to this new process's trapframe
    movl 4(%esp), %esp //把esp指向当前进程的中断帧
    jmp __trapret
```

可以看出，forkrets函数首先把esp指向当前进程的中断帧，从_trapret开始执行到iret前，esp指向了current->tf.tf_eip，而如果此时执行的是initproc，则current->tf.tf_eip=kernel_thread_entry，initproc->tf.tf_cs = KERNEL_CS，所以当执行完iret后，就开始在内核中执行kernel_thread_entry函数了，而initproc->tf.tf_regs.reg_ebx = init_main，所以在kernl_thread_entry中执行“call %ebx”后，就开始执行initproc的主体了。Initprocde的主体函数很简单就是输出一段字符串，然后就返回到kernel_tread_entry函数，并进一步调用do_exit执行退出操作了。本来do_exit应该完成一些资源回收工作等，但这些不是实验四涉及的，而是由后续的实验来完成。至此，实验四中的主要工作描述完毕。