

# uCore实验四报告

黄郭斌 计科1502班 201507010206

## 一. 内容

实验2/3完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过ucore OS的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用CPU来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的CPU。本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程只运行在内核态
- 用户进程会在在用户态和内核态交替运行
- 所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间
- 而用户进程需要维护各自的用户内存空间

## 二. 目的

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

## 三. 实验分析及结果

### 1. 整体实验思想分析

- 关键数据结构 -- 进程控制块

1. 首先简单说明一下内核线程与用户进程的区别：

内核线程只运行在内核态。用户进程会在在用户态和内核态交替运行所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间，而用户进程需要维护各自的用户内存空间。

2. 进程管理信息用struct proc\_struct表示，在kern/process/proc.h中定义如下：

```
1. struct proc_struct {
2.     enum proc_state state; // Process state
3.     int pid; // Process ID
4.     int runs; // the running times of Proces
5.     uintptr_t kstack; // Process kernel stack
6.     volatile bool need_resched; // need to be rescheduled to release CPU?
7.     struct proc_struct *parent; // the parent process
8.     struct mm_struct *mm; // Process's memory management field
9.     struct context context; // Switch here to run process
10.    struct trapframe *tf; // Trap frame for current interrupt
11.    uintptr_t cr3; // the base addr of Page Directroy Table(PDT)
12.    uint32_t flags; // Process flag
13.    char name[PROC_NAME_LEN + 1]; // Process name
14.    list_entry_t list_link; // Process link list
```

```

15.     list_entry_t hash_link; // Process hash list
16. };

```

- mm：内存管理的信息，包括内存映射列表、页表指针等。mm 里有个很重要的项pgdir，记录的是该进程使用的一级页表的物理地址。
- state：进程所处的状态。
- parent：用户进程的父进程（创建它的进程）。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程 idleproc。内核根据这个父子关系建立进程的树形结构，用于维护一些特殊的操作，例如确定哪些进程是否可以对另外一些进程进行什么样的操作等等。
- context：进程的上下文，用于进程切换（参见switch.S）。在ucore 中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等等）。使用context保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context 进行上下文切换的函数是switch\_to，kern/process/switch. 中定义。
- tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，ucore 内核允许嵌套中断。因此为了保证嵌套中断发生时 tf 总是能够指向当前的 trapframe，ucore 在内核栈上维护了 tf 的链，可以参考 trap.c::trap 函数 做进一步的了解。
- cr3: cr3 保存页表的物理地址，目的就是进程切换的时候方便直接使用 lcr3 实现页表切换，避免每次都根据 mm 来计算 cr3。mm 数据结构是用来实现用户空间的虚存管理的，但是内核线程没有用户空间，它执行的只是内核中的一小段代码（通常是一小段函数），所以它没有 mm 结构，也就是 NULL。当某个进程是一个普通用户态进程的时候，PCB 中的 cr3 就是 mm 中页表（pgdir）的物理地址；而当它是内核线程的时候，cr3 等于 boot\_cr3。而 boot\_cr3 指向了 ucore启动时建立好的饿内核虚拟空间的页目录表首地址。
- kstack:每个进程都有一个内核栈，并且位于内核地址空间的不同位置。对于内核线程，该栈就是运行时的程序使用的栈；而对于普通进程，该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。Ucore 在创建进程时分配了 2 个连续的物理页（参见 memlayout.h）作为内核栈的空间。这个栈很小，所以内核中的代码应该尽可能的紧凑，并且避免在栈上分配大的数据结构，以免栈溢出，导致系统崩溃。kstack 记录了分配给该进程/线程的内核栈的位置。主要作用有以下几点。首先，当内核准备从一个进程切换到另一个的时候，需要根据 kstack 的值正确的设置好 tss（可以回顾一下在 lab1 中讲述的 tss 在中断处理过程中的作用），以便在进程切换以后再次发生中断时能够使用正确的栈。其次，内核栈位于内核地址空间，并且是不共享的（每个进程/线程都拥有自己的内核栈），因此不受到 mm 的管理，当进程退出的时候，内核能够根据 kstack 的值快速定位栈的位置并进行回收。

为了管理系统中所有的进程控制块，ucore维护了如下全局变量（位于 kern/process/proc.c）：

- static struct proc \*current; //当前占用CPU，处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，目前至少需要屏蔽中断，可以参考 switch\_to 的实现，后面也会介绍到。linux 的实现很有意思，它将进程控制块放在进程内核栈的底部，这使得任何时候 current 都可以根据内核栈的位置计算出来的，而不用维护一个全局变量。这样使得一致性的维护以及多核的实现变得十分的简单和高效。感兴趣的同学可以参考 linux kernel 的代码。
- static struct proc \*initproc; //指向第一个用户态进程（proj10 以后）
- static list\_entry\_t hash\_list[HASH\_LIST\_SIZE]; //所有进程控制块的哈希表，这样proc\_struct 中的域 hash\_link 将基于 pid 链接入这个哈希表中。
- list\_entry\_t proc\_list; //所有进程控制块的双向线性列表，这样 proc\_struct 中的域list\_link将链接入这个链表中。

- 具体实现(创建并执行内核线程):

### 1. 第0个内核线程idleproc的创建

在init.c::kern\_init函数调用了proc.c::proc\_init函数。proc\_init函数启动了创建内核线程的步骤。首先当前的执行上下文（从kern\_init启动至今）就可以看成是uCore内核（也可看做是内核进程）中的一个内核线程的上下文。为此uCore通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第0个内核线程 -- idleproc。具体步骤如下：首先调用alloc\_proc函数来通过kmalloc函数获得proc\_struct结构的一块内存块，作为第0个进程控制块。并把proc进行初步初始化（即把proc\_struct中的各个成员变量清零）。但有些成员变量设置了特殊的值，比如：

```
1. proc->state = PROC_UNINIT; //设置进程为“初始”态
2.  proc->pid = -1;           //设置进程pid的未初始化值
3.  proc->cr3 = boot_cr3;     //使用内核页目录表的基址
4.  ...
```

接下来，proc\_init函数对idleproc内核线程进行进一步初始化：

```
1. idleproc->pid = 0; //第0个进程
2. idleproc->state = PROC_RUNNABLE; //准备状态，等待执行
3. idleproc->kstack = (uintptr_t)bootstack; //内核栈的起始地址
4. idleproc->need_resched = 1; //要此标志为1，马上就调用schedule函数要求调度器切换其他进程执行
5. set_proc_name(idleproc, "idle");
```

### 2. 创建第1个内核线程initproc

第0个内核线程主要工作是完成内核中各个子系统的初始化，然后通过执行cpu\_idle函数开始过退休生活了。所以uCore接下来还需创建其他进程来完成各种工作，但idleproc内核线程自己不想做，于是就是通过调用kernel\_thread函数创建了一个内核线程init\_main。下面我们来分析一下创建内核线程的函数kernel\_thread：

```
1. kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags)
2. {
3.     struct trapframe tf;
4.     memset(&tf, 0, sizeof(struct trapframe));
5.     tf.tf_cs = KERNEL_CS;
6.     tf.tf_ds = tf_struct.tf_es = tf_struct.tf_ss = KERNEL_DS;
7.     tf.tf_regs.reg_ebx = (uint32_t)fn;
8.     tf.tf_regs.reg_edx = (uint32_t)arg;
9.     tf.tf_eip = (uint32_t)kernel_thread_entry;
10.     return do_fork(clone_flags | CLONE_VM, 0, &tf);
11. }
```

注意，kernel\_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do\_fork函数，而do\_fork函数会调用copy\_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。

给中断帧分配完空间后，就需要构造新进程的中断帧，具体过程是：首先给tf进行清零初始化，并设置中断帧的代码段（tf.tf\_cs）和数据段（tf.tf\_ds/tf\_es/tf\_ss）为内核空间的段（KERNEL\_CS/KERNEL\_DS），这实际上也说明了initproc内核线程在内核空间中执行。而initproc内核线程从哪里开始执行呢？tf.tf\_eip的指出kernel\_thread\_entry（位于kern/process/entry.S中），kernel\_thread\_entry是entry.S中实现的汇编函数，它做的事情很简单：

```
1. kernel_thread_entry: # void kernel_thread(void)
2. pushl %edx # push arg
3. call %ebx # call fn
4. pushl %eax # save the return value of fn(arg)
5. call do_exit # call do_exit to terminate current thread
```

do\_fork函数主要做了以下6件事情：

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。

copy\_thread函数做的事情比较多，代码如下：

```
1. static void
2. copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
3.     //在内核堆栈的顶部设置中断帧大小的一块栈空间
4.     proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE) - 1;
5.     *(proc->tf) = *tf; //拷贝在kernel_thread函数建立的临时中断帧的初始值
6.     proc->tf->tf_regs.reg_eax = 0;
7.     //设置子进程/线程执行完do_fork后的返回值
8.     proc->tf->tf_esp = esp; //设置中断帧中的栈指针esp
9.     proc->tf->tf_eflags |= FL_IF; //使能中断
10.    proc->context.eip = (uintptr_t) forkret;
11.    proc->context.esp = (uintptr_t) (proc->tf);
12. }
```

设置好中断帧后，最后就是设置initproc的进程上下文，（process context，也称执行现场）了。只有设置好执行现场后，一旦uCore调度器选择了initproc执行，就需要根据initproc->context中保存的执行现场来恢复initproc的执行。这里设置了initproc的执行现场中主要的两个信息：上次停止执行时的下一条指令地址context.eip和上次停止执行时的堆栈地址context.esp。其实initproc还没有执行过，所以这其实就是initproc实际执行的第一条指令地址和堆栈指针。可以看出，由于initproc的中断帧占用了实际给initproc分配的栈空间的顶部，所以initproc就只能把栈顶指针context.esp设置在initproc的中断帧的起始位置。根据context.eip的赋值，可以知道initproc实际开始执行的地方在forkret函数（主要完成do\_fork函数返回的处理工作）处。至此，initproc内核线程已经做好准备执行了。

### 3. 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：

```

1. void
2. cpu_idle(void) {
3.     while (1) {
4.         if (current->need_resched) {
5.             schedule();
6.             .....

```

对于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；

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。switch\_to函数的执行流程：

```

1. globl switch_to
2. switch_to: # switch_to(from, to)
3. # 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
4. movl 4(%esp), %eax # eax points to from
5. popl 0(%eax) # esp--> return address, so save return addr in FROM's
6. context

```

```

1. # 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。
2. movl %esp, 4(%eax)
3. .....
4. movl %ebp, 28(%eax)
5. # restore to's registers
6. movl 4(%esp), %eax # not 8(%esp): popped return address already
7. # eax now points to to
8. movl 28(%eax), %ebp
9. .....
10. movl 4(%eax), %esp
11. pushl 0(%eax) # push T0's context's eip, so return addr = T0's eip
12. ret # after ret, eip= T0's eip

```

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：

```
1. .globl __trapret
2. __trapret:
3. # restore registers from stack
4. popal
5. # restore %ds and %es
6. popl %es
7. popl %ds
8. # get rid of the trap number and error code
9. addl $0x8, %esp
10. iret
11. .globl forkrets
12. forkrets:
13. # set stack to this new process's trapframe
14. movl 4(%esp), %esp //把esp指向当前进程的中断帧
15. jmp __trapret
```

## 2. 练习零

本实验依赖实验1/2/3。请把你做的实验1/2/3的代码填入本实验中代码中有“LAB1”、“LAB2”、“LAB3”的注释相应部分。

## 3. 练习一

alloc\_proc函数（位于kern/process/proc.c中）负责分配并返回一个新的struct proc\_struct结构，用于存储新建立的内核线程的管理信息。ucore需要对这个结构进行最基本的初始化，完成这个初始化过程。

【提示】在alloc\_proc函数的实现中，需要初始化的proc\_struct结构中的成员变量至少包括：state/pid/runs/kstack/need\_resched/parent/mm/context/tf/cr3/flags/name。

请说明proc\_struct中struct context context和struct trapframe \*tf成员变量含义和在本实验中的作用是啥？（提示通过看代码和编程调试可以判断出来）

### • 问题1：

代码实现如下：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        //LAB4:EXERCISE1 YOUR CODE
        /*
         * below fields in proc_struct need to be initialized
         *      enum proc_state state;           // Process state
         *      int pid;                          // Process ID
         *      int runs;                         // the running times of Proces
         *      uintptr_t kstack;                 // Process kernel stack
         *      volatile bool need_resched; // bool value: need to be rescheduled to release
```

```

CPU?
*      struct proc_struct *parent;                // the parent process
*      struct mm_struct *mm;                      // Process's memory management field
*      struct context context;                    // Switch here to run process
*      struct trapframe *tf;                      // Trap frame for current
interrupt
*      uintptr_t cr3;                            // CR3 register: the base addr of Page Directroy
Table(PDT)
*      uint32_t flags;                            // Process flag
*      char name[PROC_NAME_LEN + 1];             // Process name
*/
    proc->state = PROC_UNINIT; //设置进程为“初始”态
    proc->pid = -1; //设置进程pid的未初始化值
    proc->runs = 0; //初始化时间片
    proc->kstack = 0; //内核栈的地址
    proc->need_resched = 0; //是否需要调度
    proc->parent = NULL; //父节点为空
    proc->mm = NULL; //内存管理初始化
    memset(&(proc->context), 0, sizeof(struct context)); //进程上下文初始化
    proc->tf = NULL; //中断帧指针置为空，总是能够指向中断前的trapframe
    proc->cr3 = boot_cr3; //设置内核页目录表的基址
    proc->flags = 0; //标志位初始化
    memset(proc->name, 0, PROC_NAME_LEN); //进程名初始化
}
return proc;
}

```

## • 问题2:

请说明proc\_struct中struct context context和struct trapframe \*tf成员变量含义和在本实验中的作用是啥？

1. context：进程的上下文，用于进程切换。主要保存了前一个进程的现场（各个寄存器的状态）。在uCore中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等等）。使用context保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是在kern/process/switch.S中定义switch\_to。
2. tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，uCore内核允许嵌套中断。因此为了保证嵌套中断发生时tf总是能够指向当前的trapframe，uCore在内核栈上维护了tf的链。

## 4. 练习二

创建一个内核线程需要分配和设置好很多资源。kernel\_thread函数通过调用do\_fork函数完成具体内核线程的创建工作。do\_kernel函数会调用alloc\_proc函数来分配并初始化一个进程控制块，但alloc\_proc只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。ucore一般通过do\_fork实际创建新的内核线程。do\_fork的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。完成在kern/process/proc.c中的do\_fork函数中的处理过程。

- 问题1：

1. 思路：

- ①调用alloc\_proc，首先获得一块用户信息块。
- ②为进程分配一个内核栈。
- ③复制原进程的内存管理信息到新进程（但内核线程不必做此事）
- ④复制原进程上下文到新进程
- ⑤将新进程添加到进程列表
- ⑥唤醒新进程
- ⑦返回新进程号（设置子进程号为返回值）

2. 实现：

```
1. int
2. do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
3.     int ret = -E_NO_FREE_PROC;
4.     struct proc_struct *proc;
5.     if (nr_process >= MAX_PROCESS) {
6.         goto fork_out;
7.     }
8.     ret = -E_NO_MEM;
9.     //LAB4:EXERCISE2 YOUR CODE
10.    /*
11.     * Some Useful MACROs, Functions and DEFINES, you can use them in below
12.     * implementation.
13.     * MACROs or Functions:
14.     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
15.     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
16.     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
17.     *                   according clone_flags
18.     *                   if clone_flags & CLONE_VM, then "share" ; else "duplicate"
19.     *   copy_thread:  setup the trapframe on the process's kernel stack top and
20.     *                   setup the kernel entry point and stack of process
21.     *   hash_proc:    add proc into proc hash_list
22.     *   get_pid:      alloc a unique pid for process
23.     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
24.     * VARIABLES:
25.     *   proc_list:    the process set's list
26.     *   nr_process:   the number of process set
27.     */
28.    // 1. call alloc_proc to allocate a proc_struct
29.    // 2. call setup_kstack to allocate a kernel stack for child process
30.    // 3. call copy_mm to dup OR share mm according clone_flag
31.    // 4. call copy_thread to setup tf & context in proc_struct
32.    // 5. insert proc_struct into hash_list && proc_list
33.    // 6. call wakeup_proc to make the new child process RUNNABLE
34.    // 7. set ret vaule using child proc's pid
35.    //第一步：申请内存块，如果失败，直接返回处理
36.    if ((proc = alloc_proc()) == NULL) {
```



```

37.     }
38. //将子进程的父节点设置为当前进程
39.     proc->parent = current;
40. //第二步：为进程分配一个内核栈
41.     if (setup_kstack(proc) != 0) {
42.         goto bad_fork_cleanup_proc;
43. }
44. //第三步：复制父进程的内存信息到子进程
45.     if (copy_mm(clone_flags, proc) != 0) {
46.         goto bad_fork_cleanup_kstack;
47.     }
48.     //第四步：复制父进程相关寄存器信息（上下文）
49. copy_thread(proc, stack, tf);
50. //第五步：将新进程添加到进程列表（此过程需要加保护锁）
51.     bool intr_flag;
52.     local_intr_save(intr_flag);
53.     {
54.         proc->pid = get_pid();
55. //建立散列映射方便查找
56.         hash_proc(proc);
57. //将进程链节点加入进程列表
58.         list_add(&proc_list, &(proc->list_link));
59. //进程数加1
60.         nr_process ++;
61.     }
62.     local_intr_restore(intr_flag);
63. //第六步：一切准备就绪，可以叫醒子进程了
64.     wakeup_proc(proc);
65. //第七步：别忘了设置返回的子进程号
66.     ret = proc->pid;
67. fork_out:
68.     return ret;
69.
70. bad_fork_cleanup_kstack:
71.     put_kstack(proc);
72. bad_fork_cleanup_proc:
73.     kfree(proc);
74.     goto fork_out;
75. }

```

## • 问题2:

ucore是否做到给每个新fork的线程一个唯一的id的？

在使用 fork 或 clone 系统调用时产生的进程均会由内核分配一个新的唯一的PID值。

具体来说，就是在分配PID时，设置一个保护锁，暂时不允许中断，这样在就唯一地分配了一个PID。

## 5. 练习三

### • 问题1：

理解 proc\_run 函数和它调用的函数如何完成进程切换的。

代码实现如下：

```

// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load base addr of "proc"'s new PDT
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag); // 不允许中断保护进程
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE); // 加载内核堆栈
            lcr3(next->cr3); // 切换页目录表基址
            switch_to(&(prev->context), &(next->context)); // 根据上下文切换进程
        }
        local_intr_restore(intr_flag);
    }
}

```

## • 问题2：

在本实验的执行过程中，创建且运行了几个内核线程？

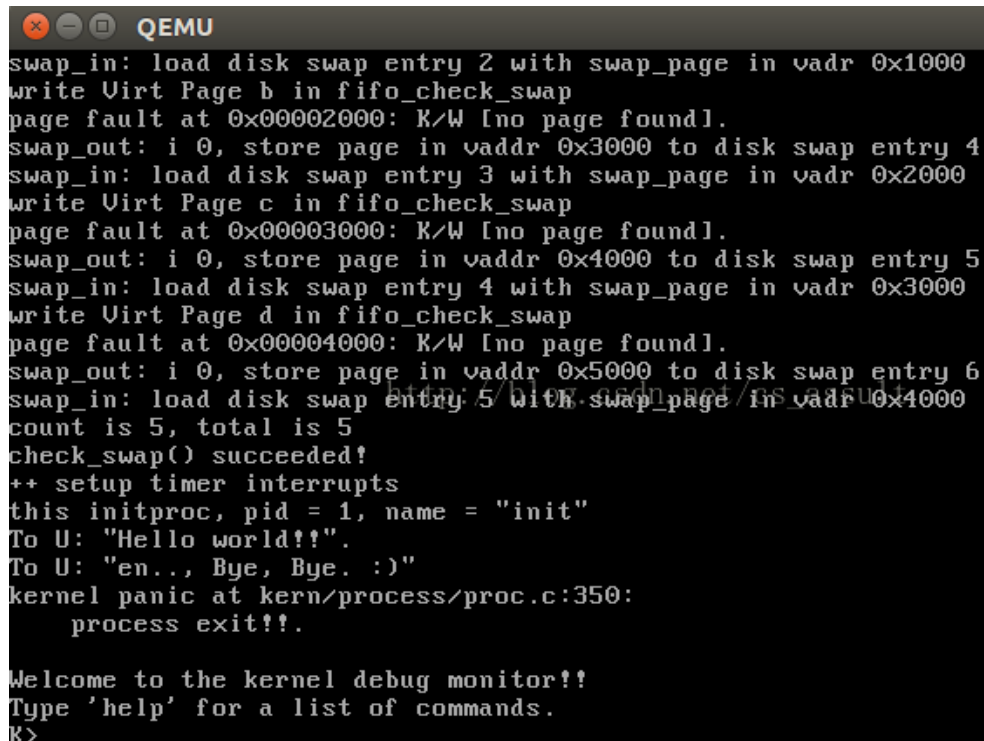
- idleproc：ucore第一个内核进程，完成内核中各个子系统的初始化，之后立即调度，执行其他进程。
- initproc：用于完成实验的功能而调度的内核进程

## • 问题3:

语句local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);在这里有何作用？

保护进程切换不会被中断，以免进程切换时其他进程再进行调度。

## • 实验结果



```

QEMU
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, total is 5
check_swap() succeeded!
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.. Bye, Bye. :)"
kernel panic at kern/process/proc.c:350:
    process exit!

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

# 四. 实验体会

---

本次实验将接触的是内核线程的管理。通过实验，我大致了解了什么是内核线程，以及内核线程与用户线程的区别。通过对内核进程的创建和相关数据结构的阅读，以及内核线程切换过程的理解，使我对用户线程调度一下的操作系统工作机理有了进一步的理解。