《操作系统原理》lab2实验报告

专业：计算机科学与技术

姓名：郝裕玮

学号：18329015

1. 实验目的

①了解内核线程创建/执行的管理过程

②了解内核线程的切换和基本调度过程

2. 实验过程与结果

练习 1：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，你需要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。

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

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）

答：需要补充的结构体成员已在代码注释和题目中给出：

```
//LAB4:EXERCISE1 YOUR CODE
/*
 * below fields in proc_struct need to be initialized
 *       enum proc_state state;                      // 进程状态
 *       int pid;                                    // 进程 ID
 *       int runs;                                   // 进程运行时间
 *       uintptr_t kstack;                           // 进程内核栈
 *       volatile bool need_resched;                 // bool值:是否需要重新调度以释放CPU?
 *       struct proc_struct *parent;                 // 父进程
 *       struct mm_struct *mm;                       // 进程的内存管理区域
 *       struct context context;                     // 进程的上下文，用于进程的切换
 *       struct trapframe *tf;                       // 当前中断帧的指针
 *       uintptr_t cr3;                              // CR3寄存器:页目录表(PDT)的基本地址
 *       uint32_t flags;                             // 进程标志符
 *       char name[PROC_NAME_LEN + 1];               // 进程名
 */
```

绝大多数成员的初始化为赋值 0 或 NULL 或分配一定大小的空间，具体初始化代码如下所示（各成员的初始值解释也已放在代码注释中）：

```
    proc->state=PROC_UNINIT;//设置进程为未初始化状态（因为新创建的进程还未被分配物理页）
    proc->pid=-1;//若进程未初始化，则其pid为-1
    proc->runs=0;//初始化时间片,对于刚初始化的进程来说，运行时间为0
    proc->kstack=0;//内核栈地址,该进程分配的地址为0，因为还没有执行，也没有被重定位。
    proc->need_resched=0;//刚刚分配出来的进程，尚未进入CPU，所以无需重新调度释放CPU
    proc->parent=NULL;//父进程为空
    proc->mm=NULL;//虚拟内存为空
    memset(&(proc->context),0,sizeof(struct context));//初始化上下文（进程的上下文用于进程切换）
    //为context结构体申请空间
    proc->tf=NULL;//中断帧指针为空
    proc->cr3=boot_cr3; //页目录表地址设为内核页目录表基址
    //cr3中含有页目录表物理内存基地址，而boot_cr3指向了ucore启动时建立好的内核虚拟空间的页目录表首地址，所以我们
将cr3指向boot_cr3
    proc->flags=0;//标志位为0
    memset(proc->name,0,PROC_NAME_LEN+1);//为进程名申请空间，建立大小为（PROC_NAME_LEN+1）的char数组
```

对于 struct context context：

（1）含义：结构体具体内容可见 proc.h，如下所示（补充的代码注释也已放在图中）：

```
// 为内核上下文切换保存寄存器。
// 不需要保存所有的%fs等段寄存器，
// 因为它们在内核上下文中是常量。
// 保存所有常规寄存器，这样我们就不需要关心了
// 它们是调用者save，而不是返回寄存器%eax。
// （不保存%eax只是简化了切换代码。）
// context的布局必须与switch.S中的代码匹配。
struct context {
    uint32_t eip;//eip寄存器里存储的是CPU下次要执行的指令的地址。
    uint32_t esp;//esp寄存器里存储的是在调用函数fun()之后，栈的栈顶，并且始终指向栈顶。
    uint32_t ebx;//基底寄存器，在内存寻址时存放基地址。
    uint32_t ecx;//计数寄存器，是重复(rep)前缀指令和loop指令的内定计数器。
    uint32_t edx;//资料寄存器，用来放整数除法产生的余数。
    uint32_t esi;//源索引寄存器，在很多字符串操作指令中，DS:ESI指向源串。
    uint32_t edi;//目标索引寄存器，在很多字符串操作指令中，DS:EDI指向目标串，
    uint32_t ebp;//ebp寄存器里存储的是是栈的栈底指针，通常叫栈基址，这个是一开始进行fun()函数调用之前，由esp传递
给ebp的。（在函数调用前可以理解为：esp存储的是栈顶地址，也是栈底地址）
};
```

（2）作用：①进程的上下文，用于进程切换。

②context 保存前一个进程各个寄存器的状态，用于上下文切换。

③当 idle 进程被 CPU 切换为 init 进程时，将 idle 进程的上下文保存在 idleproc->context 中，即保存在相应的寄存器中即可。

④当进程切换为 idle 进程时，就需要根据 idleproc->context 来恢复现场从而继续执行。

⑤对于 init 进程：本次实验的内核线程，被创建用于打印"Hello World"。

⑥对于 idle 进程：最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他线程。

对于 struct trapframe *tf：

（1）含义：中断帧的指针，总是指向内核栈的某个位置。

（2）作用：①当进程从用户空间跳入内核空间时，中断帧记录该进程被中断前的状态；

②当该进程从用户空间跳回内核空间时，需要调整中断帧来恢复对应的寄存器值，从而使得该进程继续执行；

③由于中断帧一般发生在系统调用或中断时，所以会发生特权级转换。那么此时就需要用 tf 来保存前一个被中断或异常打断线程的状态，记录段信息、通用寄存器信息等发生中断以及特权级转换时硬件和软件保存的信息，以便后续还原现场并继续执行进程。

练习 2：为新创建的内核线程分配资源（需要编码）

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。你需要完成在 kern/process/proc.c 中的 do_fork 函数中的处理过程。它的大致执行步骤包括：

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

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

请说明 ucore 是否做到给每个新 fork 的线程一个唯一的 id？请说明你的分析和理由。

答：由题目中提示的 7 个执行步骤可完成此部分的代码补充，补充后的 do_fork 函数见下（绝大部分分析已放在代码注释中）：

```
/* do_fork -      一个新的子进程的父进程
 * @clone_flags: 用于指导如何克隆子进程
 * @stack:       父节点的用户堆栈指针。如果 stack==0，意味着分叉一个内核线程。
 * @tf:          trapframe 信息，它将被复制到子进程的 proc->tf
 */
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;//尝试创造一个新的进程
    struct proc_struct *proc;//生成新进程 proc
    if (nr_process >= MAX_PROCESS) {//若生成进程数已超过上限
        goto fork_out;//若分配失败则直接返回处理
    //fork_out 内部执行的是 return ret，且 ret 初始值为-E_NO_MEM（内存不足导致请求失败）
    }
    ret = -E_NO_MEM;//内存不足导致请求失败
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * 一些有用的宏、函数和定义，您可以在下面的实现中使用它们。
     * 宏和函数:
     *   alloc_proc:   创建一个 proc 结构体和 init 字段(lab4:exercise1)
     *   setup_kstack: 分配大小为 KSTACKPAGE 的页面作为进程内核堆栈
     *   copy_mm:      进程 proc 还是共享当前进程 current，由 clone_flags 决定，如果 clone_flags&CLONE_VM 为真，则共享，否则复制
     *   copy_thread:  在进程的内核堆栈顶部设置 trapframe，并设置进程的内核入口点和堆栈
```

```
     *   hash_proc:     将 proc 添加到 proc hash_list（进程哈希列表）中
     *   get_pid:       为进程分配一个唯一的 pid
     *   wakup_proc:    设置 proc->state=PROC_RUNNABLE，唤醒进程
     * 变量:
     *   proc_list:     进程集合的列表
     *   nr_process:    进程集合的数量
     */

    //    1. 调用 alloc_proc 分配 proc_struct（获得一块用户信息块），即分配并初始化进程控制块
    //    2. 调用 setup_kstack 为子进程分配并初始化内核堆栈
    //    3. 调用 copy_mm 并根据 clone_flag 来复制或共享进程内存管理结构
    //    4. 调用 copy_thread 在 proc_struct 中设置 tf 和 context（中断帧和进程上下文）
    //    5. 把 proc_struct 插入到 hash_list && proc_list 这两个全局进程链表中
    //    6. 调用 wakup_proc 使新的子进程可运行（设置为就绪态）
    //    7. 使用子程序的 pid 设置 ret 值

    if((proc=alloc_proc())==NULL){//1.调用 alloc_proc()函数申请内存块
        goto fork_out;//若分配失败则直接返回处理
    //fork_out 内部执行的是 return ret，且 ret 初始值为-E_NO_MEM（内存不足导致请求失败）
    }
    proc->parent=current;//将子进程的父节点设置为当前进程
    if(setup_kstack(proc)!=0){//2.调用 setup_stack()函数为进程分配一个内核栈
        goto bad_fork_cleanup_proc;//若分配失败则调用 kfree 释放 kmalloc 申请的内存空间
    //并跳转至 fork_out 直接返回处理
    }
    if(copy_mm(clone_flags,proc)!=0){//3.调用 copy_mm()函数复制父进程的内存信息到子进程
        goto bad_fork_cleanup_kstack;//复制失败则释放进程内核栈的内存空间
    }
    copy_thread(proc,stack,tf);//4.调用 copy_thread()函数复制父进程的中断帧 tf 和进程上下文 context
    //5.将新进程添加到进程的 hash 列表中
    bool intr_flag;
    local_intr_save(intr_flag);//屏蔽中断，并将 intr_flag 置为 1
    {
        proc->pid=get_pid();//获取当前进程的 pid
        hash_proc(proc); //建立 hash 映射
        list_add(&proc_list,&(proc->list_link));//加入进程链表
```

```
        nr_process++;//进程数加 1
    }
    local_intr_restore(intr_flag);//恢复中断
    wakeup_proc(proc);//6.唤醒子进程
    ret=proc->pid;//7.返回子进程的唯一 pid
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

同时补充一些在该函数中调用的宏和函数定义及其分析：

①对于 E_NO_FREE_PROC 和 E_NO_MEM（位于 error.h）：

```
#define E_NO_MEM            4   // 内存不足导致请求失败
#define E_NO_FREE_PROC      5   // 尝试创造一个新的过程
```

②对于 put_kstack（位于 proc.c）：

```
// put_kstack - 释放进程内核栈的内存空间
static void
put_kstack(struct proc_struct *proc) {
    free_pages(kva2page((void *)(proc->kstack)), KSTACKPAGE);
}
```

对于每个新 fork 的线程的 id 是否唯一：

我们可查看 get_pid 的代码来分析（该问题的分析已放在代码注释中）：

补充：注释中提到的安全区间为：[last_pid , min(next_safe,MAX_PID)]，该区间始终保证内部的所有 pid 均未被使用过。

```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);//静态断言，用于发现运行期间的错误
    //判定 MAX_PID 是否大于 MAX_PROCESS，防止程序报错
    ////实际上，宏定义里已经定义了 MAX_PID=2*MAX_PROCESS，所以 PID 的总数目必定大于 PROCESS 的总数目
    struct proc_struct *proc;//创立新进程 proc
    list_entry_t *list = &proc_list, *le;//list 指向进程列表 proc_list 的头
```

```
    static int next_safe = MAX_PID, last_pid = MAX_PID;//注意是静态变量
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;//每次 last_pid 超过 MAX_PID 时就重置为 1，从头开始寻找
没用过的 pid
        goto inside;//跳转至 inside 部分
    }
    if (last_pid >= next_safe) {//若 last_pid 超过安全区间上限
    inside:
        next_safe = MAX_PID;//将安全区间上限重置为 MAX_PID
    repeat:
        le = list;//le 赋值为进程的链表头
        while ((le = list_next(le)) != list) {//遍历链表
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {//当一个已经存在的进程的 pid 和
last_pid 相等时，则在下一步判定时将 last_pid+1
            //由于 last_pid 是静态变量，所以在下一次循环时会保留这次的+1
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;//若超出安全区间且大于 MAX_PID，则将
last_pid 重置为 1，重新遍历寻找
                    }
                    next_safe = MAX_PID;//因为当前区间内有已经用过的 pid，
所以重置 next_safe 为 MAX_PID，重新遍历确定安全区间
                    goto repeat;//重新遍历
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid) {//
若进入该条件语句则意味着没有与 last_pid 相等的 proc->pid
                next_safe = proc->pid;//缩小安全区间
            }
        }
    }
    return last_pid;//为新进程分配唯一可用的 pid 号
}
```

所以综合上述代码及其代码注释分析可知，ucore 可以做到给每个新 fork 的线程一个唯一的 id。

练习 3：练习 3：阅读代码，理解 proc_run 函数和它调用的函数如何完成进程切换的。（无编码工作）

请在实验报告中简要说明你对 proc_run 函数的分析。并回答如下问题：

（1）在本实验的执行过程中，创建且运行了几个内核线程？

（2）语句 local_intr_save(intr_flag);....local_intr_restore(intr_flag); 在这里有何作用?

请说明理由。

完成代码编写后，编译并运行代码：make qemu

如果可以得到如附录 A 所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

答：对于 proc_run 函数和它调用的函数如何完成进程切换：

①cpu_idle 函数：

```
// cpu_idle - 在 kern_init 的末尾，第一个内核线程 idleproc 将执行以下工作
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

该函数的作用为：内核初始化完毕后（kern_init 末尾），不断为通过调用 schedule()来为当前进程 current 进行调度。

②schedule 函数（对该函数的所有分析均已放在代码注释中）（位于 sched.c）：

```
void
schedule(void) {
    bool intr_flag;//中断变量
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);//屏蔽中断
    {
        current->need_resched = 0;//设置当前进程 current 不需要调度
        last = (current == idleproc) ? &proc_list : &(current->list_lin
k);
        //判断 last 是否是 idleproc(第一个创建的进程),如果是，则从表头开始搜
索，否则则获取当前进程的下一链表。
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
```

```
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);//该 while 循环的作用就是遍历该进程的 hash 队列，在 proc_list 中寻找处于就绪态（PROC_RUNNABLE）的线程
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
            //若没找到处于就绪态的线程就将 next 重新指向 idleproc（闲置线程）
        }
        next->runs ++;//运行次数+1
        if (next != current) {
            proc_run(next);//调用 proc_run 函数
            //保存当前进程 current 的执行现场(进程上下文 context)，恢复新进程 next 的执行现场，完成进程切换
        }
    }
    local_intr_restore(intr_flag);//恢复中断
}
```

③switch_to 函数（对该函数的所有分析均已放在代码注释中）（位于 switch.s）：

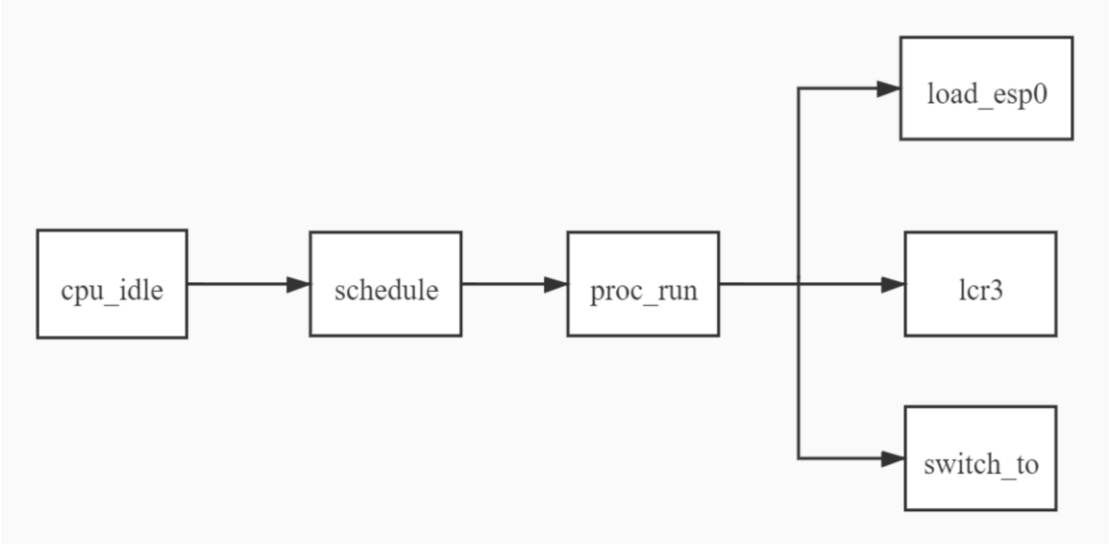
```
.text
.globl switch_to
switch_to:                      //switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          //保存 from 的首地址
    popl 0(%eax)                //将返回值保存到 context 的 eip
    //以上上 2 条指令用于保存前一个进程 from 的执行现场
    movl %esp, 4(%eax)          //保存 esp 的值到 context 的 esp
    movl %ebx, 8(%eax)          //保存 ebx 的值到 context 的 ebx
    movl %ecx, 12(%eax)         //保存 ecx 的值到 context 的 ecx
    movl %edx, 16(%eax)         //保存 edx 的值到 context 的 edx
    movl %esi, 20(%eax)         //保存 esi 的值到 context 的 esi
    movl %edi, 24(%eax)         //保存 edi 的值到 context 的 edi
    movl %ebp, 28(%eax)         //保存 ebp 的值到 context 的 ebp
    //以上 7 条指令保存前一个进程的其他 7 个寄存器到进程上下文 context 中的相应位置。

    # restore to's registers
    movl 4(%esp), %eax          //保存 to 的首地址到 eax
```

```
    movl 28(%eax), %ebp          //保存 context 的 ebp 到 ebp 寄存器
    movl 24(%eax), %edi          //保存 context 的 ebp 到 ebp 寄存器
    movl 20(%eax), %esi          //保存 context 的 esi 到 esi 寄存器
    movl 16(%eax), %edx          //保存 context 的 edx 到 edx 寄存器
    movl 12(%eax), %ecx          //保存 context 的 ecx 到 ecx 寄存器
    movl 8(%eax), %ebx           //保存 context 的 ebx 到 ebx 寄存器
    movl 4(%eax), %esp           //保存 context 的 esp 到 esp 寄存器
    //以上 7 条指令的作用为：从进程上下文 context 的高位地址 ebp 开始，按顺序把相应位置的 7 个值赋给对应的寄存器（相当于之前 7 条指令的逆操作）。

    pushl 0(%eax)                //将 context 的 eip 压入栈中
    //实际上就是把下一个进程要执行的指令地址放到栈顶

    ret                          //把栈顶的内容赋值给 eip 寄存器，至此已完成进程切换。
```

④proc_run 函数（对该函数的所有分析均已放在代码注释中）：

```
// 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) {//判断 proc 是否为当前正在执行的进程
        bool intr_flag;//中断变量
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);//屏蔽中断，避免在切换新线程时发生嵌套中断
        {
            current = proc;//设置当前进程为待调度的进程
            load_esp0(next->kstack + KSTACKSIZE);//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
            //因为 ucore 中的每个线程都需要有自己的内核栈。所以在进行线程调度切换时，也需要及时的修改 esp0 的值，使之指向新线程的内核栈顶。
            lcr3(next->cr3);//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
            //从而完成进程间的页表切换
            switch_to(&(prev->context), &(next->context));//完成具体的两个线程的执行现场切换
        }
        local_intr_restore(intr_flag);//恢复中断
    }
}
```

所以可用流程图来表示进程切换的函数调用过程：



（1）在本实验的执行过程中，创建且运行了几个内核线程？

答：创建了 2 个内核线程：

①idleproc：最初的内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他线程。

②initproc：本次实验的内核线程，被创建用于打印"Hello World"。

（2）语句 local_intr_save(intr_flag);....local_intr_restore(intr_flag); 在这里有何作用?

答：local_intr_save(intr_flag)：屏蔽中断；

local_intr_restore(intr_flag)：恢复中断。

作用：①保护进程切换不被中断，防止避免进程切换时被其他进程进行调度，即避免发生嵌套中断；

②该保护机制可确保进程执行不被打乱。

编译并运行代码：make qemu，结果如下图所示：

```
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010b220 (phys)
  edata  0xc0128a90 (phys)
  end    0xc012bc18 (phys)
Kernel executable memory footprint: 175KB
ebp:0xc0127f38 eip:0xc0101e6a args:0x00010094 0x00000000 0xc0127f68 0xc01000d3
    kern/debug/kdebug.c:308: print_stackframe+21
ebp:0xc0127f48 eip:0xc0102159 args:0x00000000 0x00000000 0x00000000 0xc0127fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0127f68 eip:0xc01000d3 args:0x00000000 0xc0127f90 0xffff0000 0xc0127f94
    kern/init/init.c:57: grade_backtrace2+33
ebp:0xc0127f88 eip:0xc01000fc args:0x00000000 0xffff0000 0xc0127fb4 0x0000002a
    kern/init/init.c:62: grade_backtrace1+38
ebp:0xc0127fa8 eip:0xc010011a args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:67: grade_backtrace0+23
ebp:0xc0127fc8 eip:0xc010013f args:0xc010b23c 0xc010b220 0x00003188 0x00000000
    kern/init/init.c:72: grade_backtrace+34
ebp:0xc0127ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:32: kern_init+84
memory management: default_pmm_manager
e820map:
```

```
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
use SLOB allocator
check_slab() success
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
```

```
SWAP: manager = fifo swap manager
BEGIN check_swap: count 31950, total 31950
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
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 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:359:
    process exit!!.

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

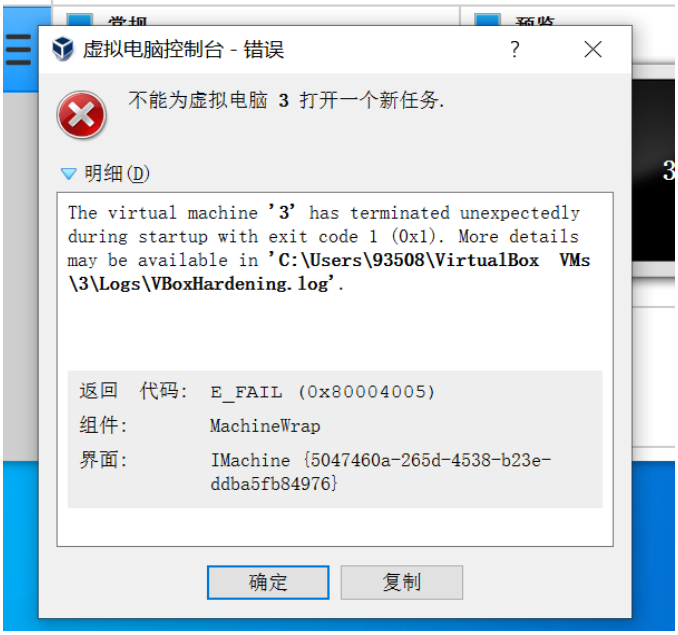
经对比可知结果大体一致，所以运行结果正确。

3. 实验感想

本次实验相较于 Lab1 其实工程量小了很多，在进行实验与书写报告的过程中也感觉到轻松了不少。

在总结这次实验之前我想说先下我的准备工作。正如 Lab1 中的实验感想所说，为了不再出现因为 ubuntu 版本，gcc 版本和 ucore 代码不匹配导致编译运行结果出现偏差的现象，我决定从第二次实验开始使用虚拟机完成实验。

然而在我按照教程将 vdi 文件导入 Virtual Box 时，vb 却报了这样的错误：



在咨询过 TA 过后我得知可能是因为杀毒软件或者证书过期而导致的。在查询过网上相关资料之后，发现我这种报错的情况较少，且解决办法并不统一，于是最后我决定使用 VMWare 来代替 Virtual Box（在这之前已尝试过更新 VB 版本和卸载重装），且在安装了 VMWare Tools 之后，我可自由复制文本和文件到主机桌面上，这显然比 Virtual Box 更加便捷。

重新回到这次实验：这次实验的代码量并不大，前两问的代码补充只要按着代码注释中的提示即可完成大半，同时在我参考了网上的几篇实验报告之后，我独立完成了全部的代码补充。并在补充的代码中写下了十分详细的分析注释，我相信这足以证明我真正明白了我所写代码的意义与作用。

这次实验主要讲的是内核线程的建立/执行/切换，由于在课上已经学习过相关理论知识，所以在实验中需要额外学习的知识并不多，主要需要查询的知识都集中在各种宏定义和函数的用法上。

总而言之，此次实验让我对内核线程的创建/执行/切换等基本步骤有了更深刻的理解，同时也明白了进程调度的基本原理。让自己在理论课上学到的知识得到了进一步的巩固，希望下次实验也可以和理论相辅相成，互相促进两方面的知识学习。