华东师范大学计算机科学与技术学院上机实践报告

课程名称：操作系统	**年级**：2022 级	**上机实践日期**：2024 年 5 月 17 日
指导教师：李东	**姓名**：岳锦鹏	**学号**：10213903403
实验名称：实验四　内核线程管理		

一、 研读理解相关代码后，回答以下问题：

1、 请简单说明 proc_struct 中各个成员变量含义以及作用。它与理论课中讲述的 PCB 结构有何不同之处?

```
struct proc_struct {
    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
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
};
```

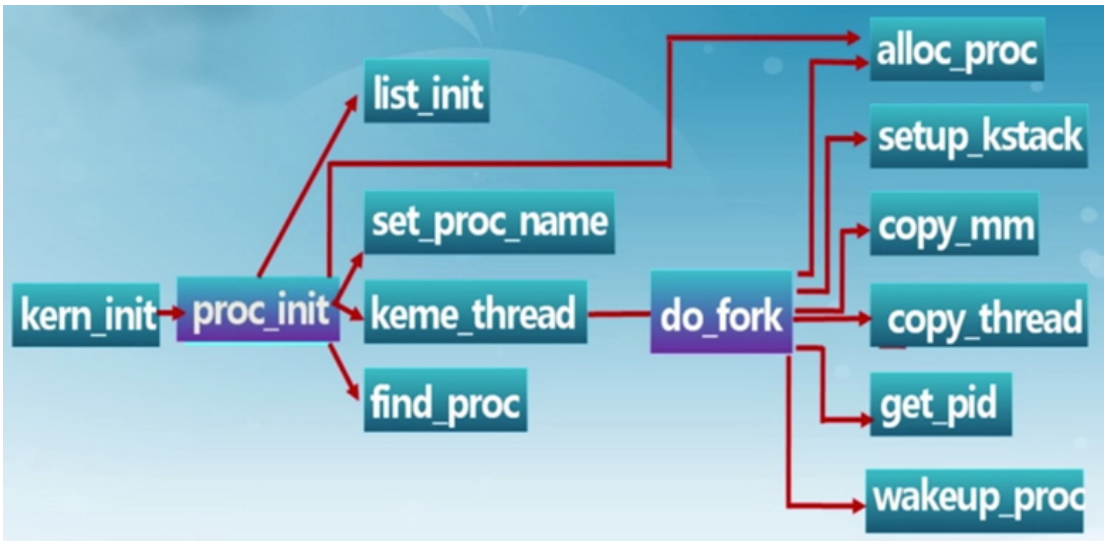
state 是进程的状态，pid 是进程的 ID，runs 是进程的被调度到的次数，kstack 是进程的堆栈，need_resched 用于非抢占式调度，表示进程是否运行结束可以把 CPU 重新调度给其他进程，parent 是进程的父进程，mm 是进程的内存管理的部分的指针，context 是进程的上下文，即

```
struct context {
    uint32_t eip;
    uint32_t esp;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t ebp;
};
```

tf 是进程的中断帧，记录了中断的相关信息，cr3 是进程的页目录表的指针，flags 是进程的各种标志位，name 是进程的名称，list_link 是所有进程的链表表项，hash_link 是具有同一个哈希值的进程的链表表项。

它与理论课中讲述的 PCB 结构的不同之处在于没有优先级、进程组、进程运行时间、进程使用的 CPU 时间、进程的子进程的 CPU 时间、文件管理（因为还没实现文件系统？）。

2、 试简单分析 uCore 中内核线程的创建过程。（要求说明处理流程，相关的主要函数、它们的功能以及调用关系。）




```
// proc_init - set up the first kernel thread idleproc "idle" by itself and
//           - create the second kernel thread init_main
void
proc_init(void) {
    int i;

    list_init(&proc_list);
    for (i = 0; i < HASH_LIST_SIZE; i ++) {
        list_init(hash_list + i);
    }

    if ((idleproc = alloc_proc()) == NULL) {
        panic("cannot alloc idleproc.\n");
    }

    idleproc->pid = 0;
    idleproc->state = PROC_RUNNABLE;
    idleproc->kstack = (uintptr_t)bootstack;
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;

    current = idleproc;

    int pid = kernel_thread(init_main, "Hello world!!", 0);
    if (pid <= 0) {
        panic("create init_main failed.\n");
    }

    initproc = find_proc(pid);
    set_proc_name(initproc, "init");

    assert(idleproc != NULL && idleproc->pid == 0);
    assert(initproc != NULL && initproc->pid == 1);
}
```

主要的内核线程创建过程在 proc_init 中，首先调用 list_init 初始化全部进程的链表和

相同哈希的进程列表，然后调用 alloc_proc 给当前进程分配一个进程控制块，并且设置相关属性，其中调用 set_proc_name 设置当前进程的名称为"idle"，之后调用 kernel_thread 创建一个 init_main 进程，并调用 find_proc，再调用 set_proc_name 将其名称设置为 init。

```
// kernel_thread - create a kernel thread using "fn" function
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    tf.tf_cs = KERNEL_CS;
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;
    tf.tf_regs.reg_ebx = (uint32_t)fn;
    tf.tf_regs.reg_edx = (uint32_t)arg;
    tf.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

kernel_thread 的功能是通过调用一个函数创建一个新的线程，由于新的线程是由当前线程创建的，这里就调用了 do_fork 函数。

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's proc->tf
 */
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm according clone_flags
     *                 if clone_flags & CLONE_VM, then "share" ; else "duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
```

```
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &(proc->list_link));
        nr_process ++;
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;
fork_out:
    return ret;

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

do_fork 函数中，先调用 alloc_proc 分配一个进程控制块，再调用 setup_kstack 分配一个内核堆栈，再调用 copy_mm 将父进程的内存空间复制或者共享给子进程，再调用 copy_thread 设置子进程的中断处理，并设置上下文的变量。之后关中断，将新的进程控制块插入到全部进程链表与相同哈希链表中，再开中断。然后把子进程设置成状态为 RUNNABLE，返回子进程的 pid。

3、 试简单分析 uCore 中内核线程的切换过程。（要求说明处理流程，相关的主要函数、它们的功能以及调用关系。）

切换线程主要使用的是 schedule 函数。

```
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
```

```
    }
    local_intr_restore(intr_flag);
}
```

在 schedule 中，我们要找到下一次调度的线程，即找到状态为 RUNNABLE 的线程，如果当前的线程为 idleproc 即 CPU 空闲的线程则从线进程链表头开始找（因为 idleproc 线程不在线程链表中），否则从当前线程的下一个线程开始找。找到需要调度的 RUNNABLE 线程后，如果新的线程和老的线程不是同一个线程，则需要调用 proc_run 进行切换。

```
// 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);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

在 proc_run 中，调用 load_esp0 设置任务状态段，调用 lcr3 加载新的进程的页表，之后调用 switch_to 保存老的进程的上下文，恢复新的进程的上下文。

```
switch_to:                      # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          # eax points to from
    popl 0(%eax)                # save eip !popl
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%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 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp

    pushl 0(%eax)               # push eip

    ret
```

在 switch_to 中，从 esp+4的位置加载 eax 指针（即&(prev->context)），并把老的线程中当前的寄存器中的值都放到这个指针所在的 context 结构体中，再从 esp+8的位置加载 eax 指针（即&(next->context)），并从这个指针的位置加载新的线程中的上下文寄存器的值，其中 eip 需要用 popl 和 pushl 来设置，不能使用 movl 来设置。