《操作系统原理》lab2实验报告

专业：管理学院会计学（辅修）

姓名：莫书琪

学号：17318086

1. 实验目的

- 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

2. 实验过程

练习1：分配并初始化一个进程控制块

编码

`alloc_proc`函数主要作用是对进程控制块(PCB)初始化，具体来说是对PCB结构体属性初始化。从`proc.h`中可以了解PCB结构体的定义。

```
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
};
```

完成`alloc_proc`函数的初始化过程编码：

```
static struct proc_struct * alloc_proc(void) {
    // 为进程控制块分配内存
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    // 初始化进程控制块的参数
    if (proc != NULL) {
    // LAB4:EXERCISE1 YOUR CODE
        proc->state = PROC_UNINIT;  // 当前进程的状态：未初始化
        proc->pid = -1;             // 进程ID
```

```
        proc->runs = 0;                 // 当前进程被调度的次数
        proc->kstack = 0;               // 内核栈
        proc->need_resched = 0;         // 是否需要被调度
        proc->parent = NULL;            // 父进程
        proc->mm = NULL;                // 当前进程管理的虚拟内存页
        memset(&(proc->context), 0, sizeof(struct context));     // 初始化上下文
        proc->tf = NULL;                // 当前的中断帧
        proc->cr3 = boot_cr3;           // 页目录表为内核页目录表的基址
        proc->flags = 0;                // 当前进程的相关标志
        memset(proc->name, 0, PROC_NAME_LEN);    // 进程名称清零
    }
    return proc;
}
```

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?

- `struc context context`的含义是线程的当前上下文，主要作用是存储进程当前的状态（各个寄存器的值）以进行进程切换
  - `context`的结构体定义(`proc.h`文件)

    ```
    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;
    };
    ```

  - 比如在`proc_run`函数中，通过调用`swith_to`方法实现从`prev->context`保存的进程切换到`next->context`保存的进程

    ```
    void switch_to(struct context *from, struct context *to);

    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);
                // 进程切换
                switch_to(&(prev->context), &(next->context));
            }
            local_intr_restore(intr_flag);
        }
    }
    ```

- `struct trapframe *tf` 的含义是中断帧指针指向的内核栈中的某个位置，主要作用是存储进程当前的状态以进行用户态到内核态的切换
  - `trapframe` 的结构体定义(`trap.h` 文件)

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

  - 比如在 `copy_thread` 函数中，当某个 `fork` 得到的新进程获取到控制流后，首先会执行 `proc->context.eip` 中的指令，但是此时新进程仍处于内核态，所以需要从内核态切换回用户态，即中断返回
  - 中断返回需要两个步骤：先利用 `forkret` 执行中断返回，然后利用 `proc->tf` 把 `trapframe` 中保存的信息恢复到各个寄存器中

```
.globl forkrets
forkrets:
    # set stack to this new process's trapframe
    movl 4(%esp), %esp
    jmp __trapret
```

```
void forkrets(struct trapframe *tf);

// forkret -- the first kernel entry point of a new thread/process
// NOTE: the addr of forkret is setted in copy_thread function
// after switch_to, the current proc will execute here.
static void
forkret(void) {
    forkrets(current->tf);
}

static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe
*tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
```

```
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    // 从内核态切换回用户态：中断返回
    // 1.执行进程中断返回，其中eip存储下一条指令的地址
    proc->context.eip = (uintptr_t)forkret;
    // 2.中断返回时进程恢复现场并开始执行用户代码，其中esp指向栈顶
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

练习2

编码

完成`do_fork`函数的编码：

```
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
    // 1.分配并初始化进程控制块PCB
    if ((proc = alloc_proc()) == NULL)
        goto fork_out;  // 错误处理
    proc->parent = current; //设置父进程
    // 2.分配并初始化内核栈
    if (setup_kstack(proc) != 0)
        goto bad_fork_cleanup_proc; // 错误处理
    // 3.根据clone_flag标志复制或共享进程内存管理结构
    if (copy_mm(clone_flags, proc) != 0)
        goto bad_fork_cleanup_kstack;   // 错误处理
    // 4.复制线程的状态，包括寄存器上下文等等
    copy_thread(proc, stack, tf);
    // 5.将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（此过程需要加保护
锁）
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);    // 建立哈希表方便查找
        list_add(&proc_list, &(proc->list_link));   // 将进程链节点加入进程列表
        nr_process++;   // 进程数加1
    }
    local_intr_restore(intr_flag);
    // 6.把子进程状态设置为就绪态
    wakeup_proc(proc);
    // 7.设置返回码为子进程的id号
    ret = proc->pid;

fork_out:
    return ret;
bad_fork_cleanup_kstack:
    put_kstack(proc);
```

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

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。

ucore可以做到给每个新fork的线程一个唯一的id，这一过程通过关中断和`get_pid()`函数实现。

- 关中断：进程号分配时需要查看进程列表中全部进程以避免发生冲突。如果进程号已分配而进程尚未添加进进程列表时被中断，该进程号可能会被重复分配，所以进程号分配与进程添加应为原子操作，进行上述操作时需加保护锁关闭中断
- `get_pid`函数：分配新的进程号`last_pid`，并遍历进程列表中的已有进程确保该进程号唯一
  - 思路：在`[0,MAXPID)`的区间内循环检测可获取的进程号，如果`last_pid`小于`MAX_PID`和`next_safe`，当前分配的`last_pid`一定是安全的
  - `next_safe`的用处：`next_safe`表示大于`last_pid`且值最小的已占用的进程号，不对`last_pid`小于`next_safe`的情况遍历可以减少循环检测的次数，优化了程序性能

```
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    // 如果last_pid大于MAX_PID，需要重新设置last_pid和next_safe
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        // 遍历proc_list
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            // 如果last_pid被占用
            if (proc->pid == last_pid) {
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            // 如果proc->pid大于last_pid，更新next_safe
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
```

```
}
```

练习3：理解 proc_run 函数和它调用的函数如何完成进程切换的

在本实验的执行过程中，创建且运行了几个内核线程?

创建且运行了`idleproc`和`initproc`两个内核线程。具体流程可以在`proc_init`函数中查看。

- `idleproc`：操作系统的第0个内核进程，主要作用是完成内核中各个子系统的初始化，并最后用于调度其他进程
- `initproc`：第一个内核进程是未来所有新进程的父进程

```
void
proc_init(void) {
    int i;

    list_init(&proc_list);
    for (i = 0; i < HASH_LIST_SIZE; i ++) {
        list_init(hash_list + i);
    }
    // idleproc（空闲进程）的创建与初始化
    // 1.分配一个proc_struct结构
    if ((idleproc = alloc_proc()) == NULL) {
        panic("cannot alloc idleproc.\n");
    }
    // 2.设置pid为0
    idleproc->pid = 0;
    // 3.设置进程状态为可运行
    idleproc->state = PROC_RUNNABLE;
    // 4.设置内核栈
    idleproc->kstack = (uintptr_t)bootstack;
    // 5.设置进程为可调度状态
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;
    // 6.设置当前运行的进程为idleproc
    current = idleproc;

    // initproc的创建
    // 1.通过kernel_thread创建init的主线程
    int pid = kernel_thread(init_main, "Hello world!!", 0);
    if (pid <= 0) {
        panic("create init_main failed.\n");
    }
    // 2.通过pid查找proc_struct
    initproc = find_proc(pid);
    // 3.设置进程名字
    set_proc_name(initproc, "init");

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

进程切换的流程

1. `cpu_idle`函数：初始化完成后，ucore会调用`cpu_idle`函数，进入一个死循环，当发现当前内核线程让出了CPU时，调用`schedule`函数进行进程调度

```
// cpu_idle - at the end of kern_init, the first kernel thread idleproc will do below works
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

2. `schedule`函数：找出处于就绪态的进程，执行`proc_run`函数

```
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        // 1.清空调度标志
        current->need_resched = 0;
        // 2.从当前进程在链表中的位置开始，遍历进程控制块
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                // 3.找到就绪态进程
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        // 4.对该进程执行proc_run函数
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

3. `proc_run`函数：完成内核栈地址设置，加载页目录表地址等前置操作，然后执行上下文切换

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
```

```
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        // 1.关闭中断
        local_intr_save(intr_flag);
        {
            // 2.将当前进程设置为要切换到的进程
            current = proc;
            // 3.设置任务状态段tss中特权级0下的esp0指针为要切换到的内核线程的内核栈的栈
顶
            load_esp0(next->kstack + KSTACKSIZE);
            // 4.设置CR3寄存器的值为要切换到的内核线程的页目录表起始地址，完成进程间的页
表切换
            lcr3(next->cr3);
            // 5.调用switch_to进行上下文的保存与切换
            switch_to(&(prev->context), &(next->context));
        }
        // 6.接触中断关闭
        local_intr_restore(intr_flag);
    }
}
```

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

- 作用：关闭中断和解除中断关闭
- 原因：在准备切换进程的时候，需要重新设置栈和页表，并且需要切换进程上下文，如果不关中断，进程信息设置到一半的时候很可能会被中断打断，导致寄存器状态处于一个不一致的状态，造成程序运行出错。

3. 实验感想

进程与线程是操作系统中十分核心的概念，理论课上虽然也有重点学这方面的内容，但在概念上还是会感觉有一些混乱。这次实验中真实的体会了内核线程/进程在操作系统中是如何被创建出来的，以及操作系统是如何对其进行一些初步的管理，感觉对进程和线程的概念清晰了很多。最后想在感想部分贴一段实验指导书中让我醍醐灌顶的对进程与线程的描述：

> 一个进程拥有一个存放程序和数据的的虚拟地址空间以及其他资源。一个进程基于程序的指令流执行，其执行过程可能与其它进程的执行过程交替进行。因此，一个具有执行状态（运行态、就绪态等）的进程是一个被操作系统分配资源（比如分配内存）并调度（比如分时使用CPU）的单位。在大多数操作系统中，这两个特点是进程的主要本质特征。但这两个特征相对独立，操作系统可以把这两个特征分别进行管理。
>
> 这样可以把拥有资源所有权的单位通常仍称作进程，对资源的管理成为进程管理；把指令执行流的单位称为线程，对线程的管理就是线程调度和线程分派。对属于同一进程的所有线程而言，这些线程共享进程的虚拟地址空间和其他资源，但每个线程都有一个独立的栈，还有独立的线程运行上下文，用于包含表示线程执行现场的寄存器值等信息。