Lab4实验报告

练习1：加载应用程序并执行（需要编码）

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，请在实验报告中简要说明你的设计实现过程。

设置在alloc_proc函数中设置trapframe的初始信息如下：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;  // 初始状态为PROC_UNINIT
        proc->pid = -1;  // -1表示未设置正确pid
        proc->runs = 0;  // 被运行次数为0
        proc->kstack = 0;  // 内核栈位置为0
        proc->need_resched = 0;  // 初始为不需要被调度
        proc->parent = NULL;  // 父进程信息为空
        proc->mm = NULL;  // 程内存管理结构体信息为空
        memset(&(proc->context), 0, sizeof(struct context));  // 将进程上下切换信息开辟出需要的空间，初始化为0
        proc->tf = NULL;  // 终端帧为空
        proc->cr3 = boot_cr3;   // 页目录表为内核页目录表
        proc->flags = 0;  // 标志位初始化为0
        memset(proc->name, 0, PROC_NAME_LEN + 1);  // 进程名开辟为指定大小
    }
    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？（提示通过看代码和编程调试可以判断出来）

- `struct context context`：用于保存进程的上下文切换信息，是几个“被调用寄存器”的信息，用于进程切换时还原之前进程的运行状态。如在通过`proc_run`将指定的进程切换到CPU上运行时，需要调用`switch_to`将原进程的寄存器状态保存，以便下次切换回去时读出，保持之前的状态，实现进程的正确切换。
- `struct trapframe *tf`：用于保存进程中断或异常处理的帧，是32个通用寄存器以及异常相关寄存器的信息。它是当进程进入内核模式时（如由于系统调用或硬件中断）需要保存信息，内核中断的处理完成后，会根据tf中的信息恢复进程状态。本次实验中利用其中的`s0、s1`寄存器来传递线程执行的函数指针与函数参数；利用其中`epc`寄存器来指定恢复到的位置，执行了`init_main`；还在创建子进程时，将子进程`tf.a0`寄存器置为0，作为一个子进程的标识。

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。因此，我们实际需要"fork"的东西就是stack和trapframe。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：

- 调用alloc_proc，首先获得一块用户信息块。
- 为进程分配一个内核栈。
- 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）
- 复制原进程上下文到新进程
- 将新进程添加到进程列表
- 唤醒新进程
- 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：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;

    // 检查当前进程数 nr_process 是否超过了系统的最大进程数 MAX_PROCESS
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }

    // 为新进程分配一个 proc_struct 结构
    proc = alloc_proc();
    if (!proc) {
        goto fork_out;
    }

    proc->parent = current;//将子进程的父节点设置为当前进程

     // 为新进程分配内核栈
    if (!setup_kstack(proc)) {
        goto bad_fork_cleanup_proc;
    }

    // 调用copy_mm()函数复制父进程的内存信息到子进程
    proc->mm = copy_mm(clone_flags, proc);
    if (!proc->mm) {
        goto bad_fork_cleanup_kstack;
    }

    // 调用copy_thread()函数复制父进程的中断帧和上下文信息
```

```
    copy_thread(proc, stack, tf);

    //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，intr_flag置为1

    proc->pid = get_pid();//获取当前进程PID
    hash_proc(proc); //建立hash映射
    list_add(&proc_list, &(proc->list_link));//加入进程链表
    nr_process ++;//进程数加一

    local_intr_restore(intr_flag);//恢复中断

    // 设置新进程的状态为 PROC_RUNNABLE，表示它可以被调度执行
    wakeup_proc(proc);

    // 成功创建进程后，返回新进程的 PID
    ret = proc->pid;

fork_out:
    return ret;

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

可以保证给每个新fork的线程一个唯一的id.

函数get_pid中,定义了`next_safe`和`last_pid`静态整型变量，用于追踪下一个安全的PID（即尚未使用的最小PID）和最后一个分配的PID。由于`next_safe`总是更新为遇到的第一个大于`last_pid`的进程PID（如果存在的话），或者重置为`MAX_PID`（如果没有找到这样的PID），并且`last_pid`总是从`next_safe`开始递增查找，直到找到一个未使用的PID，因此可以确保找到的PID是唯一的。通过重置`last_pid`和在必要时重新计算`next_safe`，代码能够处理PID的循环使用，同时保持唯一性。

```
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);//确保PID的范围足够大，可以容纳所有可能的进程
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    //last_pid 从 MAX_PID 开始递减使用
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    //last_pid 达到或超过 next_safe,寻找下一个安全的PID
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
```

```
        //过遍历 proc_list 链表，检查每个进程的PID
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

练习3：编写proc_run 函数

proc_run用于将指定的进程切换到CPU上运行。它的大致执行步骤包括：

- 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换。
- 禁用中断。你可以使用`/kern/sync/sync.h`中定义好的宏`local_intr_save(x)`和`local_intr_restore(x)`来实现关、开中断。
- 切换当前进程为要运行的进程。
- 切换页表，以便使用新进程的地址空间。`/libs/riscv.h`中提供了`lcr3(unsigned int cr3)`函数，可实现修改CR3寄存器值的功能。
- 实现上下文切换。`/kern/process`中已经预先编写好了`switch.S`，其中定义了`switch_to()`函数。可实现两个进程的context切换。
- 允许中断。

proc_run函数实现如下

```
void proc_run(struct proc_struct *proc) {
    //检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换
    if (proc != current)
    {
        bool intr_flag; // 定义用于保存中断状态的变量
        struct proc_struct *prev = current, *next = proc; // 记录当前进程和即将运行的
进程
        local_intr_save(intr_flag); // 禁用中断以保护上下文切换过程
        {
            // 将当前进程更新为要运行的进程
            current = proc;
            // 切换页表 加载新进程的页目录表到CR3寄存器并切换地址空间
            lcr3(next->cr3);
            // 执行上下文切换，切换到新进程
            switch_to(&(prev->context), &(next->context));
```

```
        }
        local_intr_restore(intr_flag); // 恢复之前的中断状态
    }
}
```

在本实验的执行过程中，创建且运行了几个内核线程?

在本实验 中一共创建了两个内核线程：

1. idleproc: 这是一个空闲进程，在操作系统中，空闲进程是一个特殊的进程，它的主要目的是在系统没有其他任务需要执行时，占用 CPU 时间，同时便于进程调度的统一化。
2. initproc: 该内核线程通过调用kernel_thread函数创建了一个内核线程init_main，在实验四中，这个子内核线程通过调用init_main函数打印一些字符串，就完成了它的工作然后返回。

   init_main函数如下：

```
static int init_main(void *arg) {
    cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid,
get_proc_name(current));
    cprintf("To U: \"%s\".\n", (const char *)arg);
    cprintf("To U: \"en.., Bye, Bye. :)\"\n");
    return 0;
}
```

在cpu_idle函数中调用schedule调度函数，使可用的内核线程运行。

扩展练习 Challenge：

说明语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`是如何实现开关中断的?

```
static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

#define local_intr_save(x)      do { x = __intr_save(); } while (0)
#define local_intr_restore(x)   __intr_restore(x);
```

工作原理为：

1、 local_intr_save(intr_flag); 这个语句保存当前的中断状态，并禁用当前CPU的本地中断。 intr_flag是一个局部布尔变量，用于保存旧的中断状态，以便稍后可以恢复。读取`sstatus`寄存器，判断`SIE`中断使能位的值，如果该位为1，则说明中断是能进行的，这时需要调用`intr_disable`函数设置中断控制寄存器的值以禁用中断，并返回1，将`intr_flag`赋值为1。如果`SIE`中断使能位为0，则说明中断此时已经不能进行，则返回0，将`intr_flag`赋值为0。以此保证之后的代码执行时不会发生中断。

2、local_intr_restore(intr_flag);这个语句恢复之前 local_intr_save 保存的中断状态。它读取 intr_flag 变量，将中断状态寄存器设 置回之前保存的值。如果先前的状态是允许中断的， intr_flag 为1（true），这将重新启用中断。

禁用中断后，执行的代码块（即 local_intr_save 和 local_intr_restore 之间的代码） 可以安全地执行，不会被中断打断。

do { x = __intr_save(); } while (0)：这个结构是为了确保宏可以安全地用在任何上下文中，尤其是当宏包含多条语句时。它将宏的内容包裹在一个单独的复合语句中，防止宏在条件语句或其他上下文中导致语法错误。