Lab4 进程管理

小组成员： 李帅东 楚乾靖 于成俊

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

练习

练习0：填写已有实验

已填写

练习1：分配并初始化一个进程控制块（需要编程）

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:

alloc_proc初始化了某个进程所需的进程控制块（PCB）结构体。该结构体是进程管理的重要工具。

初始化的过程实际上就是对PCB结构体中的各个属性赋值。需要注意的是，proc_init函数调用了alloc_proc后，会检查这个初始化PCB的过程，在判断条件中已经包含了这些属性应该被设置的值。因此，可以根据这些判断值来设置对应属性值。此处主要说明这些数值的含义（和设置的缘由），需要初始化以下属性：

- **state**：此时未分配该PCB对应的资源，故状态为初始态。
- **pid**：与state对应，表示无法运行。
- **runs**：分配阶段故运行次数为0。
- **kstack**：内核栈暂未分配。acclo_proc之后，idleproc的内核栈即为uCore启动时设置的内核栈，但之后的其他进程需要自行分配内核栈（在do_fork函数中实现）。
- **need_resched**：不用调度其他进程、即CPU资源不分配。
- **parent**：当前无父进程。但是按照之后的代码逻辑，即将该PCB的tf属性中的a0寄存器置0（意味着它是一个子进程），这个父进程被设置为了当前运行的进程。
- **mm**：当前未分配内存。此后在do_fork函数中通过调用copy_mm函数被设置。
- **context**：上下文置零。此后在do_fork函数中通过调用copy_thread函数被设置。
- **tf**：当前无中断帧。此后在do_fork函数中通过调用copy_thread函数被设置。
- **cr3**：内核线程同属于一个内核大进程，共享内核空间，故页表相同。
- **flags**：当前暂无。
- **name**：当前暂无。

代码如下：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    // 检查idleproc初始化的条件语句已经指明了该如何设置这些值
    proc->state = PROC_UNINIT;                                  // 此时未分配该PCB对应的资源，故状态为初始态
    proc->pid = -1;                                             // 与state对应，表示无法运行
    proc->runs = 0;                                             // 分配阶段故运行次数为0
    proc->kstack = 0;                                           // 内核栈暂未分配
    proc->need_resched = 0;                                     // 不用调度其他进程、即CPU资源不分配
    proc->parent = NULL;                                        // 当前无父进程
    proc->mm = NULL;                                            // 当前未分配内存
    memset(&(proc->context), 0, sizeof(struct context));        // 上下文置零
    proc->tf = NULL;                                            // 当前无中断帧
    proc->cr3 = boot_cr3;                                       // 内核线程同属于一个内核大进程，共享内核空间，故页表相同
    proc->flags = 0;                                            // 当前暂无
    memset(&(proc->name), 0, PROC_NAME_LEN);                    // 当前暂无
    }
    return proc;
}
```

context和tf成员变量含义和在本实验中的作用：

context： 表示进程上下文信息，包括程序计数器、寄存器状态、内存管理信息等。它记录了进程执行的环境和状态，当进程被切换时，需要保存当前进程的上下文信息，并加载新进程的上下文信息。上下文切换是操作系统进行进程调度和管理的关键操作。因此，"context" 是用于描述和保存进程状态的重要数据结构。

tf：表示中断上下文信息，它是在处理中断或异常时，用于保存被中断进程的状态的数据结构。"tf" 包含了被中断进程在被中断时的寄存器状态、指令指针和其他相关信息，以便在中断处理程序执行完毕后能够恢复被中断进程的执行现场。

本实验中，二者是相互配合实现进程切换的。proc_run函数中调用的switch_to函数，使用context保存原进程上下文并恢复现进程上下文。然后，由于在初始化context时将其ra设置为forkret函数入口，所以会返回到forkret函数，它封装了forkrets函数，而该函数的参数是当前进程的tf，该函数调用了__trapret来恢复所有寄存器的值。需要注意的是，在初始化tf时将其epc设置为了kernel_thread_entry，这个函数基于s0（新进程的函数）和s1（传给函数的参数）寄存器，实现了当前进程即initproc的功能，即输出“Hello World!”。

练习2：为新创建的内核线程分配资源（需要编程）

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：

代码如下：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
//fork做了7件事:分配初始化PCB，kernelstack，根据clone_flags复制或共享内存管理结构
```

```
//              设置进程在内核正常运行和调度所需的中断帧和执行上下文
//              PCB->hashlist 和 proc_list
//              该进程进程就绪状态!

    //idleproc 并不是通过这个函数创建的
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    //    1. call alloc_proc to allocate a proc_struct
     if((proc= alloc_proc())==NULL)
     {//如果分配进程失败了，就直接退出
        goto fork_out;
     };
    //proc->parent=current;//fork出来的子进程的父进程是current
    //    2. call setup_kstack to allocate a kernel stack for child process
    if((ret=setup_kstack(proc))==-E_NO_MEM)
    {//如果没有分配下内核堆，就相当于失败了，就要清除掉之前的proc再return
        goto bad_fork_cleanup_proc;
    };
    //每一个内核线程都会专门申请一块内存区域作为自己的堆栈，而不是共用其他内核的堆栈。除了
idleproc使用的是内核堆栈
    //    3. call copy_mm to dup OR share mm according clone_flag
    //       复制原来的进程的内存管理信息到新的进程proc当中
    copy_mm(clone_flags,proc);
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc,stack,tf);
    //    5. insert proc_struct into hash_list && proc_list
   //需要关闭中断来执行以下：

        proc->pid=get_pid();
        list_add(hash_list+pid_hashfn(proc->pid),&(proc->hash_link));
        list_add(&proc_list,&proc->list_link);
        nr_process++;

    //    6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc);
    //    7. set ret vaule using child proc's pid
    ret=proc->pid;



fork_out:
    return ret;

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

- 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

  ————————

ucore通过`get_pid()`函数来给每一个新fork的进程唯一的id。

在`get_pid()`首次被调用时，就会返回last_pid=1;也就是说返回1；

第二次被调用时：

- 由于++last_pid<MAX_PID,不能进入if分支。
- 由于last_pid<next_safe(MAX_PID),因此，不进入while、不进入if分支，直接返回2；

第MAX_PID-1次调用时：

- ++last_pid=MAX_PID，因此进入if分支，last_pid=1；
- 进入inside之后，遍历整个proc链表
  - 如果找到last_pid,将last_pid+=1;
    - 如果last_pid>=next_safe：重找！！！ (last_pid>=MAXPID,置1)
  - 如果当前遍历到的proc并不是last_pid，但是比当前的last_pid还要大，当前的pid也在next_safe的范围
    - 那就将next_safe置为当前的pid,接着下一个找。、

如此一来，如果有与last_pid相同的pid，那么last_pid+1；如果没有，但是找超过了，那就重新设置next_safe.

继续调用，如果调用到next_safe-1时：

- 那就将next_safe=MAX_PID
- 继续遍历链表，如果有与last_pid相同的，就将last_pid+1，接着遍历;如果没有，就啥都不干接着遍历。

如此一来，就能保障能找到唯一的pid

练习3：编写proc_run 函数（需要编程）

请回答如下问题：

- 在本实验的执行过程中，创建且运行了几个内核线程?

答:

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;//定义中断变量
        struct proc_struct *prev = current;//指向上文
        local_intr_save(intr_flag); //屏蔽中断
        {
            current = proc;//修改当前进程为新进程
            lcr3(proc->cr3);//修改页表项,完成进程间的页表切换
```

```
            switch_to(&(prev->context), &(proc->context));//上下文切换
        }
        local_intr_restore(intr_flag); //允许中断
    }
}
```

在本实验中，创建且运行了2两个内核线程：
①idleproc：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。
②initproc：用于完成实验的功能而调度的内核进程。

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?

答:

local_intr_save(intr_flag)中调用的是__intr_save()

```
// 保存当前的中断使能状态，并将中断禁止
static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) //检查终端使能
    {
        intr_disable();//如果可以，就调用中断不能函数
        return 1;
    }
    return 0;
}
//中断不能函数，就是清除sstatus上的中断使能位
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
//清楚中断使能位的具体过程
#define clear_csr(reg, bit)
({ unsigned long __tmp; \
  asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })

//csrrc使读取、指令 控制和状态寄存器
```

local_intr_restore(intr_flag)

```
// 根据之前保存的中断状态信息，恢复中断使能状态
static inline void __intr_restore(bool flag)
{
    if (flag) {
        intr_enable();
    }
}
void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
#define set_csr(reg, bit)
({
    unsigned long __tmp; \
  asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })
//csrrs使读取、置位 控制和状态寄存器
```

```
// 这个宏实际上就是将 __intr_save() 函数封装为一个宏，用于在代码中调用 __intr_save() 函数并传递一个标志位
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
// 这个宏实际上就是将 __intr_restore() 函数封装为一个宏，用于在代码中调用 __intr_restore() 函数并传递之前保存的中断状态标志位
#define local_intr_restore(x) __intr_restore(x);
```

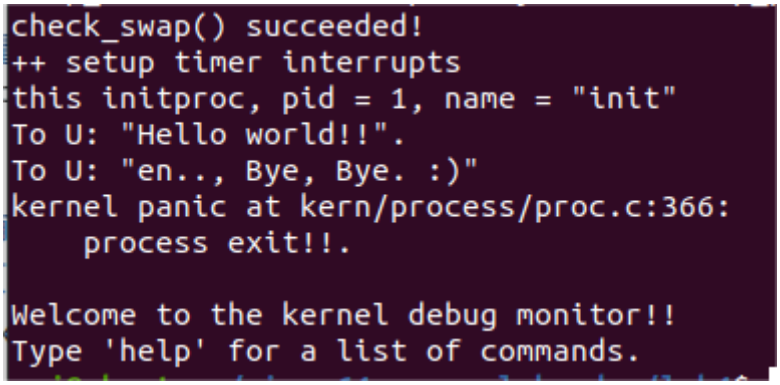
首先需要定义一个标志位（bool变量）来标识是否可以进行开关中断。在__intr_save函数中，条件语句的判断条件成立时，说明此时的中断信息已经被存储，从而可以执行后续的关中断操作。在__intr_save函数中，需要根据标志位来判断是否开中断。

另外，在定义宏 `local_intr_save(x)` 时使用了循环语句 `do { ... } while (0)`，这种写法是为了确保在宏展开时不会产生意外的行为，并且能够被安全地嵌入到其他复合语句中去。具体而言，宏展开后可能会产生一些意想不到的情况，尤其是在使用 `if`、`else` 等语句时。使用循环语句的方式可以避免这些问题，具体原因如下：

- 避免语法错误： 如果宏展开后不使用循环语句，而直接以花括号包裹，当这个宏被用在一个条件语句中时（比如 `if` 语句），可能会导致意外的语法错误。因为宏展开后的代码片段会与周围的代码上下文结合在一起，可能破坏原有的代码结构。
- 确保语句的完整性： 使用循环语句的方式能够确保宏展开后生成的代码是一个完整的语句，而不会受到外部代码块的干扰。即使在宏展开后，后面会跟着一个分号，也不会导致语法错误或逻辑错误。
- 安全的替代方式： 这种写法是 C 语言中一种常见的编程习惯，被广泛认可为一种安全的宏定义方式，能够避免许多潜在的问题和错误。

实验结果：

make qemu:



make grade:

```
c + cc kern/debug/panic.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c +
cc kern/driver/ide.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc ke
rn/driver/picirq.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc kern/trap/
trapentry.S + cc kern/mm/pmm.c + cc kern/mm/swap_fifo.c + cc kern/mm/vmm.c + cc
kern/mm/kmalloc.c + cc kern/mm/swap.c + cc kern/mm/default_pmm.c + cc kern/fs/sw
apfs.c + cc kern/process/entry.S + cc kern/process/switch.S + cc kern/process/pr
oc.c + cc kern/schedule/sched.c + cc libs/string.c + cc libs/printfmt.c + cc lib
s/hash.c + cc libs/rand.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel
 --strip-all -O binary bin/ucore.img make[1]: Leaving directory '/home/ycj/riscv
64-ucore-labcodes/lab4'
  -check alloc proc:                          OK
  -check initproc:                            OK
Total Score: 30/30
```

实验总结：

重要知识点：

- 内核线程和用户进程的区别
- 进程控制块
- 内核线程的创建
- 内核线程资源分配
- 进程(线程)切换的过程

  本实验主要是内核线程创建与切换的具体实现。在ucore中，首先创建idle_proc这个第0号内核线程，然后调用kernel_thread建立init_proc第1号内核线程，最后回到kern_init执行idle_proc线程，idle_proc总是调度到其他线程。线程具体的创建是由do_fork完成的，do_fork调用alloc_proc等函数，完成进程控制块的创建，内核栈和pid的分配，父进程上下文和中断帧的复制，还会进行一些设置，如将上下文的eip设置为fork_ret，在trapframe中将返回值设置为0等。创建完毕后返回pid，当调度器调度该线程时，调度器调用proc_run完成上下文切换后就会执行fork_ret，恢复中断帧，从而开始执行指定的程序。