lab4

知识准备

1.进程与线程定义与区别

进程是指一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程，通常我们可以将其视为资源的拥有者。线程是进程的一部分，描述指令流的执行状态，是进程中指令执行流的最小单元，进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位。

内核线程与用户进程的区别：

内核线程只运行在内核态，用户进程会在在用户态和内核态交替运行

所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间；用户进程需要维护各自的用户内存空间。

各线程可以共享地址空间和文件等资源，但是寄存器（保存上下文信息）。堆栈（保存局部变量以及使用后的返回地址）不共享。

表示进程状态的数据结构

```
enum proc_state {
    PROC_UNINIT = 0,  //未初始状态
    PROC_SLEEPING,    // 分配了物理页，此时处于睡眠状态或等待状态
    PROC_RUNNABLE,    // 运行与就绪状态
    PROC_ZOMBIE,      // 死亡状态
};
```

需要强调的是进程创建后进入为初始状态，分配物理页后进程进入睡眠状态。进程处于PROC_RUNNABLE状态不一定在运行。当程序指令执行完毕，由操作系统回收进程所占用的资源时，进程进入了“死亡”状态。

进程管理信息proc_struct结构体

```
//进程管理信息proc_struct
struct proc_struct {
    enum proc_state state;                  // 进程状态
    int pid;                                // 进程 ID
    int runs;                               // 运行时间
```

```
    uintptr_t kstack;                       // 内核栈位置
    volatile bool need_resched;             // 是否需要重新调度释放CPU
    struct proc_struct *parent;             // 父进程控制块
    struct mm_struct *mm;                   // 进程内存描述符
    struct context context;                 // 进程上下文
    struct trapframe *tf;                   // 当前中断帧的指针
    uintptr_t cr3;                          // 页表地址
    uint32_t flags;                         // 反应进程标志
                                    // 用于内核识别进程，以备下一步操作
    char name[PROC_NAME_LEN + 1];           //进程名字
    list_entry_t list_link;                 // 进程的链表
    list_entry_t hash_link;                 // Process hash list
};
```

一些重要变量的说明

mm：内存管理的信息，包括内存映射列表、页表指针等。内核线程常驻内存，不需要考虑swap page问题，因此在lab4中mm对于内核线程就没有用了，这样内核线程的proc_struct的成员变量mm=0是合理的，mm里有个很重要的项pgdir，记录的是该进程使用的一级页表的物理地址。

state：表示进程所处的状态

parent：用户进程的父进程（创建它的进程）。内核根据这个父子关系建立一个树形结构，用于维护一些特殊的操作，例如确定某个进程是否可以对另外一个进程进行某种操作等等。

context：进程的上下文，用于进程切换。在 uCore中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等）。使用 context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。

tf：当前中断帧的指针。当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。

cr3：cr3 保存页表的物理地址，目的就是进程切换的时候方便直接使用 lcr3实现页表切换，避免每次都根据 mm 来计算 cr3。 **当某个进程是一个普通用户态进程的时候，PCB 中的 cr3 就是 mm 中页表（pgdir）的物理地址；而当它是内核线程的时候，cr3 等于 boot_cr3。** 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

kstack: kstack记录分配给该进程/线程的内核栈的位置。每个线程都有一个内核栈，并且位于内核地址空间的不同位置。对于内核线程，该栈就是运行时的程序使用的栈；而对于普通进程，该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。

switch.s中

switch_to函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进

程的上下文信息保存到对于寄存器中。

练习一 分配并初始化一个进程控制块

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
代码

```
// alloc_proc -负责创建并初始化一个新的proc_struct结构存储内核线程信息
static struct proc_struct *
alloc_proc(void)
{
    //为创建的线程申请空间
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
    //LAB4:EXERCISE1 YOUR CODE
    //因为没有分配物理页，故将线程状态初始为初始状态
     proc->state=PROC_UNINIT;
     proc->pid=-1; //id初始化为-1
     proc->runs=0; //运行时间为0
     proc->kstack=0;
     proc->need_resched=0; //不需要释放CPU，因为还没有分配
     proc->parent=NULL;  //当前没有父进程，初始为null
     proc->mm=NULL;    //当前未分配内存，初始为null
     //用memset非常方便将context变量中的所有成员变量置为0
     //避免了一一赋值的麻烦。。
     memset(&(proc -> context), 0, sizeof(struct context));
     proc->tf=NULL;          //当前没有中断帧,初始为null
     proc->cr3=boot_cr3;   //内核线程，cr3 等于boot_cr3
     proc->flags=0;
     memset(proc -> name, 0, PROC_NAME_LEN);
    }
    return proc;
}
```

state：进程所处的状态。由于分配进程控制块时，进程还处于创建阶段，因此设置其状态的PROC_UNINIT，表示尚未完成初始化。
pid：先设置pid为无效值-1，用户调完alloc_proc函数后再根据实际情况设置pid。
cr3：设置为前面已经创建好的页目录表boot_pgdir的物理地址。注意是物理地址，实际编码时应写成PADDR(boot_pgdir)。便于进程的切换时，直接使用cr3实现页表的切换。
need_resched：标记是否需要调度其他进程。初始化为0，表示不需调度其他进程。
kstack：内核栈地址，先初始化为0，后续根据需要来设置
tf：中断帧指针，先初始化为NULL，后续根据需要来设置

问题：struct context context和struct trapframe *tf的含义及作用是什么

context的结构：

```
//主要用于上下文切换保存寄存器状态
struct context {
    uint32_t eip;  //存储CPU要读取指令的地址
    uint32_t esp;  //栈指针寄存器，指向栈顶
    uint32_t ebp;  //基址指针寄存器，指向栈底
    uint32_t ebx;  //数据寄存器
    uint32_t ecx;   //计数寄存器
    uint32_t edx;   //数据寄存器
    uint32_t esi;  //变址寄存器，主要用于存放存储单元在段内的偏移量
    uint32_t edi;  //变址寄存器，主要用于存放存储单元在段内的偏移量

};
//需要强调的是不需要保存所有的段寄存器，因为这些都是跨内核上下文的常量
//保存的是上下文切换时前一个进程的状态现场。
//保存上下文的函数在switch.S中
```

context是进程上下文，即进程执行时各寄存器的取值。用于进程切换时保存进程上下文比如本实验中，当idle进程被CPU切换出去时，可以将idle进程上下文保存在其proc_struct结构体的context成员中，这样当CPU运行完init进程，再次运行idle进程时，能够恢复现场，继续执行。利用context进行上下文切换的函数是在kern/process/switch.S中定义

switch_to。
trapframe定义如下：

```
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;

    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;

    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，uCore内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的trapframe，uCore 在内核栈上维护了tf的链。

```
/*
kernel_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do_fork函数，而do_fork函数会调用copy_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间
*/
int kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
```

```
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    //kernel_cs和kernel_ds表示内核线程的代码段和数据段在内核中
    tf.tf_cs = KERNEL_CS;
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS;
    //fn指实际的线程入口地址
    tf.tf_regs.reg_ebx = (uint32_t)fn;
    tf.tf_regs.reg_edx = (uint32_t)arg;
    //kernel_thread_entry用于完成一些初始化工作
    tf.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
{
    //将tf进行初始化
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    //设置tf的esp，表示中断栈的信息
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;
    //对context进行设置
    //forkret主要对返回的中断处理，基本可以认为是一个中断处理并恢复
    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

trap_frame与context的区别

从内容上看，trap_frame包含了context的信息，除此之外，trap_frame还保存有段寄存器、中断号、错误码err和状态寄存器eflags等信息。

从作用时机来看，context主要用于进程切换时保存进程上下文，trap_frame主要用于发生中断或异常时保存进程状态。

当进程进行系统调用或发生中断时，会发生特权级转换，这时也会切换栈，因此需要保存栈信息（包括ss和esp）到trap_frame，但不需要更新context。

trap_frame与context在创建进程时所起的作用
当创建一个新进程时，我们先分配一个进程控制块proc，并设置好其中的tf及context变量；然后，当调度器schedule调度到该进程时，首先进行上下文切换，这里关键的两个上下文信息是context.eip和context.esp，前者提供新进程的起始入口，后者保存新进程的trap_frame地址。
上下文切换完毕后，CPU会跳转到新进程的起始入口。在新进程的起始入口中，根据trap_frame信息设置通用寄存器和段寄存器的值，并执行真正的处理函数。可见，tf与context共同用于进程的状态保存与恢复。
综上，由上下文切换到执行新进程的处理函数fn，中间经历了多次函数调用：

forkret() -> forkrets(current->tf) ->__trapret -> kernel_thread_entry->init_main.

练习2：为新创建的内核线程分配资源

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：
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

代码：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC; //尝试为进程分配内存
    struct proc_struct *proc; //定义新进程
    if (nr_process >= MAX_PROCESS) { //分配进程数大于4096,返回
```

```
        goto fork_out;  //返回
    }
    ret = -E_NO_MEM;  //因内存不足而分配失败
if ((proc = alloc_proc()) == NULL) { //分配内存失败
        goto fork_out; //返回
    }
    proc->parent = current; //设置父进程名字
    if (setup_kstack(proc) != 0) {//分配内核栈
        goto bad_fork_cleanup_proc; //返回
    }
    if (copy_mm(clone_flags, proc) != 0) { //复制父进程内存信息
        goto bad_fork_cleanup_kstack; //返回
    }
    copy_thread(proc, stack, tf); //复制中断帧和上下文信息
    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
    wakeup_proc(proc); //唤醒新进程
    ret = proc->pid; //返回当前进程的PID
fork_out:  //已分配进程数大于4096
    return ret;
bad_fork_cleanup_kstack: //分配内核栈失败
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

上述调用函数的功能：

```
//创建一个proc并初始化所有成员变量
```

```
void alloc_proc(void)
//为一个内核线程分配物理页
static int setup_kstack(struct proc_struct *proc)
//暂时未看出其用处，可能是之后的lab会用到
static int copy_mm(uint32_t clone_flags, struct proc_struct *proc)
//复制原进程上下文到新进程
static void copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
//返回一个pid
static int get_pid(void)
//将proc加入到hash_list
static void hash_proc(struct proc_struct *proc)
// 唤醒该线程，即将该线程的状态设置为可以运行
void wakeup_proc(struct proc_struct *proc);
```

do_fork()函数做的事情有：
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
上述代码调用的函数：

```
static int //内核栈复制函数
setup_kstack(struct proc_struct *proc) {
    struct Page *page = alloc_pages(KSTACKPAGE);
    if (page != NULL) {
        proc->kstack = (uintptr_t)page2kva(page);
        return 0;
    }
    return -E_NO_MEM;
}

static int  //该函数在本次实验并没有实现
```

```
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    assert(current->mm == NULL); //判断当前函数的虚拟内存非空
    return 0;
}

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;

    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
void
intr_disable(void) { //禁止中断函数
    cli(); //禁止中断
}

static inline bool
__intr_save(void) {
    if (read_eflags() & FL_IF){ //如果允许屏蔽中断，即IF=1.则中断
        intr_disable(); //禁止中断
        return 1;
    }
    return 0;
}

static inline void
__intr_restore(bool flag) { //如果中断被屏蔽，则恢复中断
    if (flag) {
        intr_enable(); //恢复中断
    }
}
```

问题 ucore是否做到给每个新fork的线程一个唯一的id

首先，本实验不提供线程释放的功能，意味着pid只分配不回收。当fork的线程总数小于MAX_PID时，每个线程的pid是唯一的。当fork的线程总数大于MAX_PID时，后面fork的线程的pid可能与前面的线程重复（暂不确定）。

为了回答这个问题，必须从给fork id的函数入手，也就是get_pid()函数，查看get_pid()的代码：

```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    //实际上，之前定义了MAX_PID=2*MAX_PROCESS，意味着ID的总数目是大于
PROCESS的总数目的
    //因此不会出现部分PROCESS无ID可分的情况
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    //next_safe和last_pid两个变量，这里需要注意！ 它们是static全局变量！！！
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    //++last_pid>-MAX_PID,说明pid以及分到尽头，需要从头再来
    if (++ last_pid >= MAX_PID)
    {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe)
    {
    inside:
        next_safe = MAX_PID;
    repeat:
        //le等于线程的链表头
        le = list;
        //遍历一遍链表
        //循环扫描每一个当前进程：当一个现有的进程号和last_pid相等时，则将
last_pid+1；
            //当现有的进程号大于last_pid时，这意味着在已经扫描的进程中
```

```
        //[last_pid,min(next_safe, proc->pid)] 这段进程号尚未被占用，继续扫描。
        while ((le = list_next(le)) != list)
        {
            proc = le2proc(le, list_link);
            //如果proc的pid与last_pid相等，则将last_pid加1
            //当然，如果last_pid>=MAX_PID,then 将其变为1
            //确保了没有一个进程的pid与last_pid重合
            if (proc->pid == last_pid)
            {
                if (++ last_pid >= next_safe)
                {
                    if (last_pid >= MAX_PID)
                    {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            //last_pid<pid<next_safe，确保最后能够找到这么一个满足条件的区间，获得合法的pid；
            else if (proc->pid > last_pid && next_safe > proc->pid)
            {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

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）

在本实验的执行过程中，创建且运行了几个内核线程?

语句 local_intr_save(intr_flag);….local_intr_restore(intr_flag); 在这里有何作用?请说明理由。

函数功能
1 屏蔽中断
2 修改esp0，页表项和进行上下文切换
3 允许中断

switch函数

```
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
    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

    # 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
    pushl 0(%eax)               #将context的eip压入栈中
    ret
```

所以switch_to函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。
schedule函数

```
/* 宏定义:
  #define le2proc(le, member)         \
   to_struct((le), struct proc_struct, member)*/
void
schedule(void) {
    bool intr_flag; //定义中断变量
    list_entry_t *le, *last; //当前list，下一list
    struct proc_struct *next = NULL; //下一进程
    local_intr_save(intr_flag); //中断禁止函数
    {
        current->need_resched = 0; //设置当前进程不需要调度
     //last是否是idle进程(第一个创建的进程),如果是，则从表头开始搜索
     //否则获取下一链表
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do { //一直循环，直到找到可以调度的进程
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);//获取下一进程
                if (next->state == PROC_RUNNABLE) {
                    break; //找到一个可以调度的进程，break
                }
            }
        } while (le != last); //循环查找整个链表
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc; //未找到可以调度的进程
        }
        next->runs ++; //运行次数加一
        if (next != current) {
            proc_run(next); //运行新进程,调用proc_run函数
        }
    }
    local_intr_restore(intr_flag); //允许中断
}
```

schedule函数的执行逻辑：1．设置当前内核线程current->need_resched为0； 2．在proc_list队列中查找下一个处于“就绪”态的线程或进程next； 3．找到这样的进程后，就调

用proc_run函数，保存当前进程current的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换

proc_run函数

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag; //定义中断变量
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag); //屏蔽中断
        {
            current = proc; //修改当前进程为新进程
            load_esp0(next->kstack + KSTACKSIZE); //修改esp
            lcr3(next->cr3); //修改页表项
            //上下文切换
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag); //允许中断
    }
}
```

可以看到ucore实现的是FIFO调度算法：
1 调度开始时，先屏蔽中断。
2 在进程链表中，查找第一个可以被调度的程序
3 运行新进程，允许中断
回答问题1：本实验创建且运行了几个内核线程
创建且运行了两个内核线程，分别是idle和init线程。通过kernel_thread函数、proc_init函数以及具体的实现结果可知，本次实验共建立了两个内核线程。首先是idleproc内核线程，该线程是最初的内核线程，完成内核中各个子线程的创建以及初始化。之后循环执行调度，执行其他进程。还有一个是initproc内核线程，该线程主要是为了显示实验的完成而打印出字符串"hello world"的内核线程。
回答问题2：local_intr_save和local_intr_restore的作用
避免在进程切换过程中处理中断。
该语句应该是关闭中断以及恢复中断。在操作系统理论课上"进程同步"中讲过类似的操作–原子操作。在此处应该是可以把这一过程视为原子操作的，主要目的是避免在运行过程中被其他中断打断，或是被其他线程修改了相关变量的值导致程序结果与预期不符。

实验吐槽

练习二的代码根据注释写的还可以，就是练习一关于get_pid函数的理解问题上出现了重大问题，代码干了啥，似乎知道，可是这是为什么呢。另外，上一次实验的时候，看了后边的参考文献基本知道干了啥。可是，这一次，不知道时没有带脑子，还是啥，居然又重新看了几遍。我好难啊……难且坚强快乐的活着。