OSlab4--by：周延霖

实验2/3完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过ucore OS的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用CPU来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的CPU。

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 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，而用户进程需要维护各自的用户内存空间

练习0：经典合并代码

把Lab1-3的代码并入（注意Lab3里面本来在pmm里面写好的kmalloc和kfree函数要先注释掉，不然会和Lab4新增的kmalloc.h和kmalloc.c在函数定义和使用上出现冲突）

练习1：分配并初始化一个进程控制块（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。
>
> 在alloc_proc函数的实现中，需要初始化的proc_struct结构中的成员变量至少包括以下内容：
>
> state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name
>
> 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?

实现内核线程的第一步是给线程创建进程（ucore中的线程相当于一个不拥有资源的轻量级进程）控制块。

在kern/process/proc.c的alloc_proc函数中，给要创建的进程控制块指针（struct proc_struct *proc）分配了内存空间，设置如下变量：

```
    /*       enum proc_state state;                      // Process state
     *       int pid;                                    // Process ID
     *       int runs;                                   // the running
times of Process
     *       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
     */
```

各个变量的详细解释如下：

- state：进程状态，proc.h中定义了四种状态：创建（UNINIT）、睡眠（SLEEPING）、就绪（RUNNABLE）、退出（ZOMBIE，等待父进程回收其资源）
- pid：进程ID，调用本函数时尚未指定，默认值设为-1
- runs：线程运行总数，默认值0
- need_resched：标志位，表示该进程是否需要重新参与调度以释放CPU，初值0（false，表示不需要）
- parent：父进程控制块指针，初值NULL
- mm：用户进程虚拟内存管理单元指针，由于系统进程没有虚存，其值为NULL
- context：进程上下文，默认值全零
- tf：中断帧指针，默认值NULL
- cr3：该进程页目录表的基址寄存器，初值为ucore启动时建立好的内核虚拟空间的页目录表首地址boot_cr3（在kern/mm/pmm.c的pmm_init函数中初始化）
- flags：进程标志位，默认值0
- name：进程名数组

可以写出初始化代码：

```
proc->state = PROC_UNINIT;
proc->pid = -1;
proc->runs = 0;
proc->kstack = 0;
proc->need_resched = 0;
proc->parent = NULL;
proc->mm = NULL;
memset(&(proc->context), 0, sizeof(struct context));
proc->tf = NULL;
proc->cr3 = boot_cr3;
proc->flags = 0;
memset(proc->name, 0, PROC_NAME_LEN);
```

回答问题

> 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?

context指进程上下文，这部分空间用于保存创建进程时父进程的部分寄存器值：eip, esp, ebx, ecx, edx, esi, edi, ebp；其他寄存器在切换进程时值不变，故不需要保存。

tf是中断帧的指针，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。

练习2：为新创建的内核线程分配资源（需要编程）

> 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
>
> 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

根据注释中的步骤我们可以很容易编写如下的代码

```
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;
    //上面的是原有部分，上面的代码尝试新建一个进程，如果进程数大于等于MAX_PROCESS则失败
    //下面的是新增部分
    if((proc = alloc_proc()) == NULL)goto fork_out;//尝试分配内存，失败就退出
    proc->parent = current;//把此进程的父进程设置为current
    if(setup_kstack(proc)==-E_NO_MEM)goto bad_fork_cleanup_proc;//尝试分配内核栈
    if(copy_mm(clone_flags,proc)!= 0)goto bad_fork_cleanup_kstack;//尝试复制父进程内存
    copy_thread(proc,stack,tf);//复制中断帧和上下文
    bool flag;
    local_intr_save(flag);//屏蔽中断
    proc->pid=get_pid();//获取PID
    hash_proc(proc);//建立hash映射
    list_add_after(&proc_list,&(proc->list_link));//添加进链表
    /*用list_add_before或者list_add也完全ok*/
```

```
    nr_process++;//进程数++
    local_intr_restore(flag);//恢复中断
    wakeup_proc(proc);//唤醒进程
    return proc->pid;//返回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

对于这个问题，分配ID的代码是get_pid()，其内容如下

```
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;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;//当第一次调用时lastpid=1
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list)
        {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid)
            {//如果当前进程pid等于lastpid，lastpid++
                if (++ last_pid >= next_safe)
                {//如果lastpid大于nextsafe
                    if (last_pid >= MAX_PID)
                    {//如果lastpid大于MAX_PID,lastpid=1
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;//把next_safe设置为MAXPID，跳回repeat
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
            {//如果proc->pid在lastpid到safe之间，就把safe设置为proc->pid
                next_safe = proc->pid;
            }
```

```
            }
        }
        return last_pid;
    }
```

这段代码会不断的在链表中遍历，直到找到一个合适的last_pid才会返回，这个last_pid满足两个条件(1)不大于MAX_PID(2)未被分配过，因此ucore为每个新fork的线程分配了一个唯一的id。

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）

> 请在实验报告中简要说明你对proc_run函数的分析。并回答如下问题：
>
> - 在本实验的执行过程中，创建且运行了几个内核线程?
> - 语句 local_intr_save(intr_flag);....local_intr_restore(intr_flag); 在这里有何作用?请 说明理由
>
> 完成代码编写后，编译并运行代码：make qemu
>
> 如果可以得到如 附录A所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

1. proc_run:

proc_run 用于使一个线程在 CPU 中运行

```
void
proc_run(struct proc_struct *proc) {
    // 首先判断要切换到的进程是不是当前进程，若是则不需进行任何处理。
    // 调用local_intr_save和local_intr_restore函数去使能中断，避免在进程切换过程中出现中断。
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            // 将当前进程设为传入的进程
            current = proc;
            // 修改 esp 指针的值
            // 设置任务状态段tss中的特权级0下的esp0指针为next内核线程的内核栈的栈顶
            load_esp0(next->kstack + KSTACKSIZE);
            // 修改页表项
            // 重新加载 cr3 寄存器(页目录表基址) 进行进程间的页表切换
            lcr3(next->cr3);
            // 使用 switch_to 进行上下文切换。
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

2. switch_to:

首先把当前寄存器的值送到原线程的 context 中保存，再将新线程的 context 赋予各寄存器。

```
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
```

Q&A：

Q1：在本实验的执行过程中，创建且运行了几个内核线程?

有两个内核线程: **1） 创建第0个内核线程idleproc**。在 init.c::kern_init 函数调用了 proc.c::proc_init 函数。proc_init 函数启动了创建内核线程的步骤。首先当前的执行上下文（从 kern_init 启动至今）就可以看成是 uCore 内核（也可看做是内核进程）中的一个内核线程的上下文。为此，uCore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第 0 个内核线程 – idleproc。

2） 创建第 1 个内核线程 initproc。第 0 个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行 cpu_idle 函数开始过退休生活了。所以 uCore 接下来还需创建其他进程来完成各种工作，但 idleproc 内核子线程自己不想做，于是就通过调用 kernel_thread 函数创建了一个内核线程 init_main。在Lab4中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看 init_main 函数）。但在后续的实验中，init_main 的工作就是创建特定的其他内核线程或用户进程。

```
// init_main - the second kernel thread used to create user_main kernel
threads
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;
}
```

Q2：语句 local_intr_save(intr_flag); ...local_intr_restore(intr_flag); 在这里有何作用?请说明理由。

关闭中断和打开中断。

这两个函数实现的意义就是**避免在进程切换过程中处理中断**。因为有些过程是互斥的，只允许一个线程进入，因此需要关闭中断来处理临界区；如果此时在切换过程中又一次中断的话，那么该进程保存的值就很可能出bug并且丢失难寻回了。

Challenge：实现支持任意大小的内存分配算法

> 这不是本实验的内容，其实是上一次实验内存的扩展，但考虑到现在的slab算法比较复杂，有 必要实现一个比较简单的任意大小内存分配算法。可参考本实验中的slab如何调用基于页的内 存分配算法（注意，不是要你关注slab的具体实现）来实现first-fit/best-fit/worst-fit/buddy等支 持任意大小的内存分配算法。