Lab4

练习1：分配并初始化一个进程控制块（需要编码）

如下是初始化进程控制块的代码，下面将使用注释的形式解释。

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // LAB4:EXERCISE1 2313815_段俊宇_2313485_陈展_2310591_李相儒
        /*
         * below fields in proc_struct need to be initialized
         *       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 pgdir;                            // the base addr of
Page Directroy Table(PDT)
         *       uint32_t flags;                             // Process flag
         *       char name[PROC_NAME_LEN + 1];               // Process name
         */
        proc->state = PROC_UNINIT;                          // 进程状态初始化为未启
动状态
        proc->pid = -1;                                     // 进程ID未分配，因此
是-1
        proc->runs = 0;                                     // 进程运行次数初始化为
0
        proc->kstack = 0;                                   // 内核栈指针初始化为0
        proc->need_resched = 0;                             // 是否需要重新调度设置
为0，也就是不需要
        proc->parent = NULL;                                // 父进程指针初始化为
NULL
        proc->mm = NULL;                                    // 内存管理结构初始化为
NULL
        memset(&(proc->context), 0, sizeof(struct context)); // 上下文结构体使用
memset函数清零
```

```
        proc->tf = NULL;                                    // 中断帧指针初始化为
NULL
        proc->pgdir = boot_pgdir_pa;                        // 页目录基址初始化为引
导页目录物理地址
        proc->flags = 0;                                    // 进程标志清零
        memset(proc->name, 0, PROC_NAME_LEN+1);             // 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？（提示通过看代码和编程调试可以判断出来）

1. **struct context context**
   - **含义：**这个结构体中保存了一系列寄存器的状态，因此$context$保存了进程切换的上下文信息，也就是关键寄存器的状态。
   - **作用：**在进程切换时保存关键寄存器的状态，便于在重新调度时恢复
2. **struct trapframe *tf**
   - **含义：**这个结构体保存了中断帧相关的信息，当进程从用户空间转向内核空间时，该结构体保存了执行状态。
   - **作用：**在进程从用户空间进入内核空间时保存进程的执行状态，便于在返回到用户空间时恢复之前的执行状态。

练习2 为新创建的内核线程分配资源（需要编码）

如下是我们编写的$do_fork()$函数，下面将使用注释的形式解释。

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork
a kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's
proc->tf
 */
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 2313815_段俊宇_2313485_陈展_2310591_李相儒
    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
```

```
     *                 if clone_flags & CLONE_VM, then "share" ; else
"duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top
and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid

    // 1.调用alloc_proc，首先获得一块用户信息块
    proc = alloc_proc();
    if(proc==NULL)
        goto fork_out;
    // 2.使用setup_kstack()函数为进程分配一个内核栈
    if((setup_kstack(proc)!=0))
        goto bad_fork_cleanup_proc;
    // 3.使用copy_mm()函数复制原进程的内存管理信息到新进程
    if(copy_mm(clone_flags, proc)!=0)
        goto bad_fork_cleanup_kstack;
    // 4.使用copy_thread()函数复制原进程上下文到新进程
    copy_thread(proc, stack, tf);

    // 初始化其他信息
    proc->pid = get_pid();  // 获取唯一进程号
    proc->state = PROC_UNINIT;
    proc->parent = current; // 设置当前进程为新进程的父进程
    proc->runs = 0;

    // 将新进程添加到进程列表和哈希列表
    hash_proc(proc);
    list_add(&proc_list, &proc->list_link);

    // 使用wakeup_proc()函数唤醒新进程，将进程状态设置为PROC_RUNNABLE
    wakeup_proc(proc);

    // 返回新进程号
    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。

可以做到，$get_pid()$函数可以为每一个进程分配唯一的进程id。get_pid算法通过维护 $last\_pid$ 和 $next_safe$两个状态变量来高效分配进程ID。它优先尝试简单递增 $last\_pid$，如果该值在安全范围内且未被占用则直接分配；一旦发现潜在的PID冲突或超出安全范围，算法便会扫描整个进程列表，在冲突时递增$last_pid$ 并重新扫描，同时记录下一个已占用PID作为新的安全上限，以此确保总是能找到并分配当前最小的可用PID，并在PID耗尽时从1开始回绕复用。

练习3：编写proc_run 函数（需要编码）

如下是我们编写的$proc_run()$函数，下面将使用注释的形式解释。

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        // LAB4:EXERCISE3 2313815_段俊宇_2313485_陈展_2310591_李相儒
        /*
         * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
         * MACROs or Functions:
         *   local_intr_save():        Disable interrupts
         *   local_intr_restore():     Enable Interrupts
         *   lsatp():                   Modify the value of satp register
         *   switch_to():              Context switching between two processes
         */
        // 记录进程标志
        unsigned long intrflag;
        struct proc_struct *prev = current;
        // 禁用中断
        local_intr_save(intrflag);
        // 将当前进程状态变成需要转换的进程状态
        current = proc;
        // 切换页表
        lsatp(proc->pgdir);
        // 是否需要调度设置为0
        proc->need_resched = 0;
        proc->runs++;
        // 切换进程，上下文切换
        switch_to(&(prev->context), &(proc->context));
        // 恢复中断机制
        local_intr_restore(intrflag);
    }
}
```

在本实验的执行过程中，创建且运行了几个内核进程?

创建并运行了两个内核进程，一个是$idleproc$，它表示空闲进程，主要是为了在没有系统进程调度时占用CPU以便后续统一调度；另一个是$initproc$，它$fork$了$idleproc$进程的信息，并且输出了$Hello\ world!$这一内容。

下面是使用$make\ \ qemu$命令运行的结果，成功输出了一系列信息，说明我们成功完成了进程初始化、$fork$创建新进程以及进程切换的任务！

扩展练习 Challenge

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

首先，我们分析每一行代码，了解其功能：

代码行	说明
`local_intr_save(intrflag);`	保存当前中断状态，并关闭本地CPU中断，防止在关键区间被打断。

代码行	说明
`unsigned long intrflag;`	定义变量 `intrflag` 用于存放原先CPU中断使能标志。
`struct proc_struct *prev = current;`	保存当前正在运行的进程指针，用于后续恢复。
`current = proc;`	切换全局变量 `current` 为目标进程 `proc`，表示CPU现在要执行的新进程。
`lsatp(proc->pgdir);`	修改RISC-V的 `satp` 寄存器，切换到新进程的页表。
`proc->need_resched = 0;`	清除"需要重新调度"标志，表示当前进程已获得CPU。
`proc->runs++;`	增加该进程的运行次数，用于统计调度次数。
`switch_to(&(prev->context), &(proc->context));`	执行上下文切换，保存旧进程寄存器状态并恢复新进程寄存器状态。
`local_intr_restore(intrflag);`	恢复原中断状态，如果之前开中断则重新打开，保证切换完成后系统可响应中断。

根据代码，我们能知道位于 $sync.c$ 的两个宏定义是如何实现控制开关的中断的。

宏定义	功能说明
`local_intr_save(x)`	读取 `sstatus` 寄存器保存中断标志位（`SSTATUS_SIE`），然后清除该位关闭中断。
`local_intr_restore(x)`	将之前保存的标志位写回 `sstatus` 寄存器，恢复中断状态。

2.深入理解不同分页模式的工作原理（思考题）

- **首先是第一个问题，为什么 `get_pte()` 中有两段高度相似的代码?**

我们先来了解一下 RISC-V 的分页机制（$Sv32$ / $Sv39$ / $Sv48$）。对于这三种层级的分页模式，$RISC-V$ 都采用 多级页表结构。虚拟地址在翻译时，会被拆分成若干段 VPN（$Virtual\ Page\ Number$），每一段对应页表的一级索引。不同分页模式的唯一区别在于页表的层数不同：$Sv32$ 有两级页表、$Sv39$ 有三级页表、$Sv48$ 有四级页表，这里的数字表示虚拟地址的总位数，其中三种级别页表的页内偏移都是 12 位，$Sv39$ 与 $Sv48$ 中的每一段 VPN 占 9 位，而在 $Sv32$ 中占 10 位。

虽然页表层数不同，但每一级的操作方式完全相同，都是对一个页表页进行索引、检查其是否有效、必要时分配一个新的页表页，然后继续向下一层遍历。

基于这一点，在 $uCore$ 当前使用的三级页表结构中，只有上两级页表需要按需创建页表页，而最底层的页表页（用于存放最终 PTE）在第二段代码中已经被分配。因此，$get_pte()$在访问虚拟地址时需要执行两次"相同模式"的操作：第一次处理高一级页目录，第二次处理下一级页表，而最终一级不再需要创建下一层。这就是出现两段结构高度相似代码的原因。

- **接着是第二个问题，页表项的查找与分配放在一个函数里是否合理？是否需要拆分?**

把"查找页表项"和"必要时创建中间页表页"合并在一个函数里，对于 $Sv32$ / $Sv39$ / $Sv48$ 这类典型多级页表模式来说非常合理，这样使得调用方只需要一次调用即可保证对应层级存在，可以减少代码复杂度与函数调用开销。我们可以看本函数中的实现，其中第一段处理上一级页目录，第二段处理下一级页表 ，都是先查找、若无效则分配，然后继续向下，如此调用者无需关心每一级页表是否存在，只需一

次调用即可得到最终的 PTE，减少调用复杂度与错误率。

如果将逻辑拆成 $lookup_pte()$与 $alloc_pte()$两个函数，调用者就必须在外层重复这两段代码所实现的逻辑：先检查一级是否有效，再根据情况分配页表页，然后再进入下一级。这意味着调用者必须手动处理 PTE_V检查、$alloc\_page()$、$memset()$等的写入顺序，这会使得代码相对冗长、更容易出错。

因此，从代码结构上看，$get\_pte()$ 将“查找 + 分配”合并在一起，完全贴合多级页表递归下降的访问模式，同时在 uCore 这种轻量级的内核中，这种写法更易维护，结构更统一。只有在更复杂的内存管理需要（如 $lazy\ allocation$ 或 VMA调试）时，才有必要将查找与分配分离。