Lab 4

练习1：分配并初始化一个进程控制块

补充完整后alloc_proc函数如下：

```
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        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->pgdir = boot_pgdir_pa;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN + 1);
    }
    return proc;
}
```

代码解释：

- state：设置进程为初始态，我们需要将其设置为PROC_UNINIT
- pid：未初始化的进程号我们需要设置为-1
- runs：初始化时间片，刚刚初始化的进程，运行时间设置为0
- kstack：内核栈地址,该进程分配的地址为0，因为还没有执行，也没有被重定位，因为默认地址都是从0开始的
- need_resched：是一个用于判断当前进程是否需要被调度的bool类型变量，为1则需要进行调度。初始化的过程中我们不需要对其进行调度，因此设置为0
- parent：父进程为空，设置为NULL
- mm：虚拟内存为空，设置为NULL
- memset(&proc->context,0,sizeof(struct context))：初始化上下文，将上下文结构体context初始化为0
- tf：中断帧指针为空，设置为NULL
- pgdir：它保存了进程的页目录表的物理基地址。我们将其设为内核页目录表的物理地址boot_pgdir_pa
- flags：标志位flags设置为0
- memset(&proc->name, 0, PROC_NAME_LEN)：进程名name初始化为0

设计实现过程：
我们定位到proc_struct结构体的位置（proc.h），结构信息如下所示：

```
struct proc_struct {
    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 cr3;                              // CR3 register: the base addr
of Page Directroy Table(PDT)
    uint32_t flags;                             // Process flag
    char name[PROC_NAME_LEN + 1];               // Process name
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
};
```

我们需要做的就是将这些变量进行初始化即可。根据指导书上的提示，可以很轻松地完成代码补充。具体代码和解释参见上文，这里不再赘述。

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结构体如下：

```
struct context
{
    uintptr_t ra;
    uintptr_t sp;
    uintptr_t s0;
    uintptr_t s1;
    uintptr_t s2;
    uintptr_t s3;
    uintptr_t s4;
    uintptr_t s5;
    uintptr_t s6;
    uintptr_t s7;
    uintptr_t s8;
    uintptr_t s9;
    uintptr_t s10;
    uintptr_t s11;
};
```

context 结构体用于保存进程的内核执行上下文。它主要存储了一组非易失性的寄存器，即那些在函数调用返回后必须保持不变的寄存器（如保存的寄存器 s0-s11、堆栈指针 sp 和返回地址 ra 等）。

在实验中作用：
context 用于内核线程之间主动的切换（即调度器引起的切换）。

保存内核状态：当一个进程（内核线程）通过 schedule() 调用 switch_to() 让出 CPU 时，它将自身的内核寄存器状态保存到 current->context 中。

恢复执行点：当调度器再次选中该进程时，switch_to() 会从 proc->context 中恢复这些寄存器，从而使进程从它上次离开调度器的地方继续执行。

trapframe结构体如下：

```
struct trapframe
{
    struct pushregs gpr;
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};
```

tf是一个指针，指向存储在进程内核栈顶部的 trapframe 结构。
它用于保存进程的完整处理器状态。陷阱帧记录了所有通用寄存器（gpr）、状态寄存器（status）程序计数器（epc）、导致异常的虚拟地址（badvaddr）以及异常原因代码（cause）等。

在实验中作用：
tf 用于处理异常、中断或系统调用（即 CPU 被迫进入内核）。

异常/中断保存：当进程从用户态进入内核态时，硬件和内核会将被中断时的所有状态信息保存在当前进程内核栈顶部的 trapframe 结构中，proc->tf 则指向这个结构。

返回用户态：当系统调用或中断处理完毕后，内核会从 proc->tf 中恢复完整的处理器状态，使 CPU 能够精确地返回到中断/异常发生前的执行点。

练习2：为新创建的内核线程分配资源

下面是补充完整的 `do_fork` 函数，在 `kern/process/proc.c` 中。

```
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 2313983
    /*
     * 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. 调用alloc_proc分配一个进程控制块
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    //    2. 调用setup_kstack为子进程分配内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    //    3. 调用copy_mm根据clone_flags复制或共享内存管理信息
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    //    4. 调用copy_thread设置进程的中断帧和上下文
    copy_thread(proc, stack, tf);

    //    5. 将进程控制块插入哈希表和进程链表
    bool intr_flag;
    local_intr_save(intr_flag);  // 禁用中断，保证操作原子性
    {
        proc->pid = get_pid();            // 为进程分配唯一pid
        hash_proc(proc);                  // 加入哈希表
        list_add(&proc_list, &(proc->list_link));  // 加入进程链表
        nr_process++;                     // 增加进程计数
    }
    local_intr_restore(intr_flag);        // 恢复中断

    //    6. 调用wakeup_proc将新进程设置为可运行状态
    wakeup_proc(proc);

    //    7. 将返回值设置为子进程的pid
    ret = proc->pid;

fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);    // 清理内核栈
bad_fork_cleanup_proc:
    kfree(proc);         // 释放进程控制块
    goto fork_out;
}
```

设计实现过程

首先分配进程控制块，用于存储进程管理信息；然后分配内核栈，为进程执行提供栈空间；接着处理内存空间，根据标志决定复制或共享；最后设置执行上下文，使新进程能够开始执行。

ucore是否做到给每个新fork的线程一个唯一的id?

是可以的！理由如下：

1. **pid分配算法**：
    - 在`get_pid()`函数中实现了专门的pid分配逻辑
    - 使用`last_pid`和`next_safe`两个变量来管理pid分配范围
2. **唯一性保证机制**：

- 分配pid时会遍历所有现有进程，检查pid是否已被使用
- 如果发现冲突，会递增pid并重新检查，直到找到可用的pid
- 当pid达到最大值时，会从1开始重新查找

3. **原子性保护**：
   - 在`do_fork`中分配pid和加入列表的操作是在禁用中断的情况下进行的
   - 这防止了在分配过程中被其他进程打断，导致pid重复分配
4. **数据结构支持**：
   - 使用哈希表存储进程，便于快速检查pid是否已存在
   - 进程链表也提供了遍历所有进程的能力

练习3：编写 proc_run 函数

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current)
    {
        // LAB4:EXERCISE3 YOUR CODE
        /*
         * 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
         */

        bool intr_flag = 0;
        struct proc_struct *prev = current;

        local_intr_save(intr_flag);

        current = proc;

        lsatp(proc->pgdir);

        switch_to(&prev->context, &proc->context);

        local_intr_restore(intr_flag);
    }
}
```

该函数实现了：

1. 检查是否为当前进程。
2. 若切换进程，则先禁用中断。
3. 将当前地址空间切换到切换到新进程的页目录地址。
4. 实现了新旧进程的上下文切换。
5. 最后将中断允许。

在本次实验的过程中，创建了两个进程，分别是 idleproc 和 initproc 。只创建并运行了一个进程 initproc。

运行截图：

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?

相关代码如下：

```
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();
    }
}
```

当调用local_intr_save()时，会读取sstatus寄存器，根据SIE位的值判断进入前中断是否开启。如果该位为1，则说明中断已开启，执行intr_disable()，将SIE位设置为0，关闭中断。函数返回1并将此值保存在 intr_flag 中；如果该位为0，则说明中断已关闭，函数直接返回 0，并将此值保存在 intr_flag 中。

当需要恢复中断时，调用local_intr_restore()，需要判断intr_flag的值。如果其值为1，则执行 intr_enable()，将SIE位设置为1，重新开启中断；如果其值为0，则不执行任何操作，保持中断关闭。

2.深入理解不同分页模式的工作原理（思考题）

(1) get_pte()函数中有两段形式类似的代码， 结合sv32，sv39，sv48的异同，解释这两段代码为什么如此相像。
get_pte() 中出现两段几乎相同的代码，是因为 RISC-V 多级页表的每一级（SV32 两级、SV39 三级、SV48 四级）都需要执行完全相同的步骤：根据线性地址索引查找下一级页表项，并检查其有效性（PTE_V）。如果下一级页表不存在，并且允许创建 (create=true)，则分配一个新的物理页并将其作为下一级页表，然后更新当前层的页表项使其指向这个新分配的页。因此每一层页表的处理逻辑本质一致，只是层数不同，使得代码自然呈现重复结构。

(2)目前get_pte()函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？

将页表项的查找和分配合并在一个函数中，这在 ucore 等教学内核中是为了代码的简洁性，在处理缺页异常时效率较高；但在更复杂的内核中，为了遵循单一职责原则、简化并发控制（特别是读写锁机制）以及提高模块的独立性，通常有必要将查找操作和资源分配/修改操作拆分开。