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。

> 【提示】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：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。

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）

初始化struct proc_struct

在本实验中，进程管理信息用struct proc_struct表示，首先查看proc_struct的定义：

```
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
};
```

里面几个比较重要的成员变量及其含义为：

- `mm`：保存内存管理的信息，包括内存映射，虚存管理等内容。
- `state`：进程所处的状态。共有四种状态，`PROC_UNINIT`（未初始化状态）、`PROC_SLEEPING`（休眠状态）、`PROC_RUNNABLE`（可运行状态（可能正在运行））和`PROC_ZOMBIE`（几乎已终止，等待父进程回收其资源）。
- `parent`：保存进程的父进程的指针。
- `context`：保存进程执行的上下文（几个关键的寄存器的值）。
- `tf`：保存进程的中断帧。当进程从用户空间跳进内核空间的时候，进程的执行状态被保存在了中断帧中。
- `cr3`：保存页表所在的基址。
- `kstack`：分配给该进程/线程的内核栈的位置。

下面在alloc_proc函数对struct proc_struct结构进行初始化：

```
    proc->state = PROC_UNINIT;// 设置进程为初始化状态
    proc->pid = -1;//还未分配进程ID 设置pid为-1
    proc->runs = 0; //刚初始化的进程 设置运行次数为0
    proc->kstack = 0;//进程未被执行 也未被重定位 设置内核栈地址为0（默认地址）
    proc->need_resched = 0;// 设置不需要重新调度
    proc->parent = NULL;// 设置父进程为空
    proc->mm = NULL;// 设置内存管理字段(虚拟内存)为空
    memset(&(proc->context), 0, sizeof(struct context));// 初始化上下文信息为0
    proc->tf = NULL;// 设置中断帧指针trapframe为空
    proc->cr3 = boot_cr3;// 设置CR3寄存器的值为内核页目录基址
    proc->flags = 0;// 设置进程标志位为0
    memset(proc->name, 0, PROC_NAME_LEN);// 初始化进程名为0
```

根据实验指导书，下面几个成员变量需要设置特殊的值：

- `proc->state`：设置进程状态为`PROC_UNINIT`，表示进程初始态。
- `proc->pid`：由于进程还未分配，设置其`pid`为-1。
- `proc->cr3`：`cr3`寄存器保存页表所在的基址，初始化时设置为内核页目录基址。

除这几个成员变量外，其余成员变量均清零。

成员变量`context`和`tf`的含义

1. `context`：`context`中保存了进程执行的上下文信息，其中包含了`ra`，`sp`，`s0~s11`共14个寄存器，这些寄存器的值用于在进程切换中还原之前进程的运行状态，例如，`context.ra`保存的是返回地址，`context.sp`保存的是栈指针等信息，以确保进程能够从被中断的位置继续执行。
2. `tf`：`tf`中保存了进程的中断帧，在进程因处理系统调用、硬件中断或异常从用户空间跳进内核空间时，当前进程的状态（如程序计数器、堆栈指针等）需要被保存，以便在中断处理完成后恢复进程的执行。`tf`存储了这些信息，它包括了与中断相关的寄存器值，例如`epc`（异常程序计数器），`status`（中断控制状态）等

成员变量`context`和`tf`的作用

1. `context`：在本实验中，`context`用于保存当前进程在切换时的寄存器状态。具体来说，当一个进程被挂起或者一个新的进程被调度时，操作系统会保存当前进程的`context`，然后加载下一个进程的`context`，使得新进程能够从正确的位置恢复执行。在上下文切换的关键函数`switch_to`中，就是通过`context`结构体来实现从一个进程的上下文切换到另一个进程的上下文。具体的切换过程就是保存当前进程的`context`，然后加载下一个进程的`context`。
2. `tf`：在本实验中，`tf`主要用于保存当前进程在系统调用或中断时的现场信息。在进程从用户空间跳到内核空间被中断时，`tf`保存了必要的寄存器值，以便在中断返回时能够恢复到正确的执行点。具体而言，在`proc_init`函数的执行过程中，通过调用`kernel_thread`函数，为线程`initproc`的中断帧成功分配了相应的内存空间，并进行了初始化操作。具体包括：把函数`fn`赋值给`s0`寄存器，将`fn`函数所需的参数赋值给`s1`寄存器，以及将入口点（`epc`）设定为`kernel_thread_entry`函数。完成上述初始化步骤之后，随即调用`do_fork`函数，利用已创建的中断帧`tf`来创建线程`initproc`，与此同时，把上下文中的`ra`设置为`forkret`函数的入口地址，以此确保线程`initproc`在创建完成后能够成功切换，按照预期的流程顺利执行后续操作。

练习2：为新创建的内核线程分配资源

> kern/process/proc.c 中 do_fork函数：

```
// 调用 alloc_proc ，首先获得一块用户信息块。
proc = alloc_proc();
proc->parent = current;
// 为进程分配一个内核栈。
setup_kstack(proc);
// 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）
copy_mm(clone_flags, proc);
// 复制原进程上下文到新进程
copy_thread(proc, stack, tf);
// 将新进程添加到进程列表
int pid = get_pid();
proc->pid = pid;
hash_proc(proc);
list_add(&proc_list, &(proc->list_link));
nr_process++;
// 唤醒新进程
proc->state = PROC_RUNNABLE;
// 返回新进程号
ret = proc->pid;
```

1. 分配进程控制块（`alloc_proc`）

- `alloc_proc()` 是用来分配一个新的进程控制块（PCB）。这个控制块是内核用来管理进程的一个重要数据结构，保存了关于进程的各种信息，例如进程状态、父进程、进程ID等。

2. 设置进程的父进程（`proc->parent = current;`）

- 通过 `proc->parent = current;` 设定新进程的父进程。`current` 是指当前正在执行的进程（即调用 `do_fork` 函数的进程）。

3. 设置内核栈（`setup_kstack(proc);`）

- 内核栈是进程执行期间用于存储内核态栈帧、局部变量、函数参数等的内存区域。通过 `setup_kstack(proc)` 为新进程分配和初始化内核栈，以确保进程在内核态时能够正确地存储调用信息。

4. 复制内存管理信息（`copy_mm(clone_flags, proc);`）

- `copy_mm()` 负责复制当前进程的内存管理信息到新进程。这包括虚拟内存的映射、页表等。该操作是必须的，因为每个进程都有自己的虚拟内存空间和资源。参数 `clone_flags` 用来标识哪些资源需要共享（例如，父子进程可以共享某些内存区域）。
- 对于内核线程（如代码中的情况），通常不需要复制用户空间的内存管理信息，因为内核线程不需要用户空间的虚拟内存。

5. 复制进程上下文（`copy_thread(proc, stack, tf);`）

- `copy_thread()` 负责复制当前进程的上下文（即进程的 CPU 寄存器值、程序计数器等），并保存到新进程的进程控制块中。

- `stack` 和 `tf` 分别是新的进程堆栈地址和异常帧。此时，新进程的执行上下文已经准备好，可以在后续调度时正确恢复。

6. 分配进程 ID（`int pid = get_pid();`）

- 使用 `get_pid()` 获取一个新的唯一进程 ID。进程 ID 是操作系统用来标识进程的一个重要字段。新进程的 PID 被设置为 `proc->pid = pid;`。

7. 将新进程添加到进程列表（`hash_proc(proc)` 和 `list_add(&proc_list, &(proc->list_link));`）

- `hash_proc()` 将新进程加入到进程哈希表中，用于快速查找。
- `list_add(&proc_list, &(proc->list_link));` 将新进程添加到进程链表 `proc_list` 中，管理所有进程。

8. 更新进程数量（`nr_process++`）

- 增加当前进程的数量 `nr_process`，以便内核能够追踪系统中当前运行的进程数。

9. 唤醒新进程（`proc->state = PROC_RUNNABLE;`）

- 设置新进程的状态为 `PROC_RUNNABLE`，表示该进程已准备好被调度执行。

10. 返回新进程的 PID（`ret = proc->pid;`）

- 最后，返回新创建的进程 ID `proc->pid`。这是 `do_fork()` 的返回值，通常会在父进程中接收到新子进程的 PID，而在子进程中返回 `0`。

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。

`ucore`能做到给每个新`fork`的线程一个唯一的`id`。

在 `ucore` 中，确实会为每个新创建的进程或线程分配一个唯一的进程 ID (PID)，这通过调用 `get_pid()` 函数实现。根据代码中提供的信息，新进程的 PID 是由 `get_pid()` 动态分配的，并且通过以下代码行设置：

```
cCopy codeint pid = get_pid();
proc->pid = pid;
```

`get_pid()` **函数的作用**：

- `get_pid()` 函数负责生成一个唯一的进程 ID (PID)，它会维护 PID 的分配机制，确保每次分配的 PID 不与当前系统中存在的进程 PID 重复。
- 它的原理是对于一个可能分配出去的`last_id`，遍历线程链表，判断是否有`id`与之相等的线程，如果有，则将`last_id`自增1，且保证自增之后不会与当前查询过的线程`id`冲突，并且其不会超过最大的线程数，重新从头开始遍历链表。如果没有，则更新下一个可能冲突的线程`id`。

练习3：编写proc_run 函数

> **问题一**：proc_run 用于将指定的进程切换到 CPU 上运行，请实现该函数：

```
void proc_run(struct proc_struct *proc)
{
    // 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        // 禁用中断
        local_intr_save(intr_flag);
        {
            // 切换当前进程为要运行的进程
            current = proc;
            // 将CR3寄存器设置为目标进程的页目录表基址，以确保虚拟地址空间正确映射。
            lcr3(next->cr3);
            // 使用switch_to保存当前线程的上下文到prev->context，
            // 并将next->context加载到对应的寄存器中，实现上下文切换
            switch_to(&(prev->context), &(next->context));
        }
        // 允许中断
        local_intr_restore(intr_flag);
    }
}
```

代码的基本流程：

- 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换
- 使用`local_intr_save(x)`禁用中断。
- 切换当前进程为要运行的进程。
- 将 CR3 寄存器设置为目标进程的页目录表基址，以确保虚拟地址空间正确映射。
- 使用`switch_to`保存当前线程的上下文到`prev->context`，并将`next->context`加载到对应的寄存器中，实现上下文切换
- 使用`ocal_intr_restore(x)`允许中断。

> **问题二**：在本实验的执行过程中，创建且运行了几个内核线程？

在本实验中，一共创建了两个内核线程：

- **线程 0：idleproc**：

  ucore 在`kern_init`中调用了`proc_init`，对 idleproc 进行了初始化，下面对初始化过程进行分析：

  - **内存分配**：调用`alloc_proc`函数，首先通过`kmalloc`函数分配了一个进程控制块的内存空间，然后对进程控制块进行了一些初始化，如练习一所示。
  - **检测进程控制块**：通过一系列条件判断语句检测进程控制块是否如我们预想地那样被分配和初始化了。
  - **初始化 idleproc**：将线程 id 设为 0，将状态设为`PROC_RUNNABLE`，将内核栈设为 ucore 启动时设置的内核栈，设置为需要调度，将名称设置为`idle`，增加线程数`nr_process`。
  - **设置当前线程**：将当前线程指针`current`指向`idleproc`。

  idleproc 与其他线程不同，仅仅算是“继承了”ucore 的运行，因为当前的执行上下文就可以看成是 uCore 内核中的一个内核线程的上下文。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，以便对其进行管理。

- **线程 1：initproc**：

  ucore 在`proc_init`中初始化`proc_init`之后，又在其中完成了`initproc`的创建和初始化。

  - **线程初始化**：`ucore`通过调用`kernel_thread`创建`proc_init`线程。在函数内部，会首先初始化一个该函数的`trapframe`，然后进一步调用`do_fork`完成线程地创建，并返回线程具体分析见之前实验的相关部分。
  - **查找线程**：初始化之后，会调用`find_proc`通过返回的 pid 查找该线程。`find_proc`首先通过`pid`的哈希值找到`hash_list`对应的位置，并在其中遍历根据`pid`相等找到该线程。使用`hash_list`主要是为了加快查找速度，基本可以实现 O(1)的查找。
  - **设置线程名称**：设置线程的`name`属性为`init`。
  - **线程调度**：最后在`cpu_idle`函数中，`idle`线程会通过`schedule`函数进行调用，找到第一个`PROC_RUNNABLE`的线程，即`init`线程，并通过上述的`proc_run`函数运行，具体见前面对`proc_run`函数的实现。

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?

我们可以在kern/sync/sync.c中找到`local_intr_save`和`local_intr_restore`这两个函数的相关代码：

```
#ifndef __KERN_SYNC_SYNC_H__
#define __KERN_SYNC_SYNC_H__

#include <defs.h>
#include <intr.h>
#include <riscv.h>

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();
    }
}

#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
#define local_intr_restore(x) __intr_restore(x);

#endif /* !__KERN_SYNC_SYNC_H__ */
```

1. `local_intr_save(intr_flag)`：这个宏的作用是保存当前中断的状态并禁用中断。它调用了 `__intr_save()` 函数。
    - 在 `__intr_save()` 函数内部通过读取 `sstatus` 寄存器来判断当前是否允许中断（`SSTATUS_SIE` 标志位）。
        - 如果 `SSTATUS_SIE` 位为 `1`，表示当前允许中断，函数会调用 `intr_disable()` 禁用中断，并返回 `1`，表示中断被禁用。
        - 如果 `SSTATUS_SIE` 位为 `0`，表示当前禁止中断，函数直接返回 `0`。
    - `local_intr_save(x)` 宏会将 `__intr_save()` 的返回值（`1` 或 `0`）保存在 `x` 中。这个返回值表示中断的原始状态（是否被启用）。如果中断之前是启用的，它将被禁用，同时将这个信息保存在 `x` 中。
2. `local_intr_restore(intr_flag)`：这个宏的作用是恢复中断的状态，它调用了 `__intr_restore(flag)` 函数。
    - `__intr_restore(bool flag)` 函数会根据传入的 `flag` 值来决定是否重新启用中断。如果 `flag` 为 `1`，表示之前禁用了中断，那么调用 `intr_enable()` 恢复中断。

总结一下，`local_intr_save(intr_flag)` 保存当前中断状态（启用还是禁用），以便之后恢复，然后禁用中断。`local_intr_restore(intr_flag)` 根据之前保存的状态恢复中断的启用状态。这两个宏定义函数结合起来就可以实现在一个进程发生切换前禁用中断，切换后重新启用中断，以实现开关中断。

实验中的知识点

进程与线程

我们平时编写的源代码，经过编译器编译就变成了可执行文件，我们管这一类文件叫做**程序。**而当一个程序被用户或操作系统启动，分配资源，装载进内存开始执行后，它就成为了一个**进程。**进程与程序之间最大的不同在于进程是一个“正在运行”的实体，而程序只是一个不动的文件。进程包含程序的内容，也就是它的静态的代码部分，也包括一些在运行时在可以体现出来的信息，比如堆栈，寄存器等数据，这些组成了进程“正在运行”的特性。

如果我们只关注于那些“正在运行”的部分，我们就从进程当中剥离出来了**线程。**一个进程可以对应一个线程，也可以对应很多线程。这些线程之间往往具有相同的代码，共享一块内存，但是却有不同的CPU执行状态。相比于线程，进程更多的作为一个资源管理的实体（因为操作系统分配网络等资源时往往是基于进程的），这样线程就作为可以被调度的最小单元，给了调度器更多的调度可能。

进程控制块（PCB）

进程控制块（`PCB`） 是操作系统用于管理进程的一个数据结构。它保存了进程在系统中的所有关键信息，操作系统通过 `PCB` 来进行进程调度、管理以及控制。每个进程在系统中都有一个对应的 `PCB`，操作系统通过这个结构体来记录进程的状态以及与进程相关的所有资源信息。

`PCB` 的基本内容：进程标识符（`pid`）、进程状态（`state`）、父进程指针（`parent`）、`CPU` 上下文（`context`）、内存管理信息（`mm`）、中断帧（`trapframe`）、内核栈地址（`kstack`）、进程标志（`flags`）、进程名字（`name`）、进程链表指针（`list_link`、`hash_link`）、运行次数（`runs`）。

`PCB` 的作用：进程调度和切换、进程管理、资源管理、进程间通信和同步、调试和监控、进程的状态管理。