Lab4

练习0：填写已有实验

> 本实验依赖实验2/3。请把你做的实验2/3的代码填入本实验中代码中有“LAB2”，“LAB3”的注释相应部分。

除了do_pgfault需要复制进来，其他的基本不需要复制Lab2和Lab3的代码。

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）

`alloc_proc` 函数的作用是实现初始化空闲进程的函数，其调用了 `kmalloc` 函数获得一个空的进程控制块作为空闲进程的进程控制块，然后如果获取成功会对进程控制块这个结构体中的成员进行初始化，会将进程状态设置成 `PROC_UNINIT` 代表进程初始态，pid设为-1代表进程还未分配，cr3设为uCore内核页表的基址，剩余的成员几乎都是进行清零和置空处理，具体代码如下：

```
// alloc_proc - 分配一个proc_struct并初始化proc_struct的所有字段
static struct proc_struct *
alloc_proc(void)
{
    // 使用kmalloc分配内存空间给新的proc_struct
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // LAB4:EXERCISE1 你的代码
        /*
         * 需要初始化的proc_struct中的字段
         *       enum proc_state state;                      // 进程状态
         *       int pid;                                    // 进程ID
         *       int runs;                                   // 进程的运行次数
         *       uintptr_t kstack;                           // 进程的内核栈地址
```

```
16          *       volatile bool need_resched;               // 布尔值：是否需要重新调度以释放CPU?
17          *       struct proc_struct *parent;               // 父进程
18          *       struct mm_struct *mm;                     // 进程的内存管理字段
19          *       struct context context;                   // 切换到此处运行进程
20          *       struct trapframe *tf;                     // 当前中断的trapframe
21          *       uintptr_t cr3;                            // CR3寄存器：页目录表(PDT)的基地址
22          *       uint32_t flags;                           // 进程标志
23          *       char name[PROC_NAME_LEN + 1];             // 进程名
24          */
25         proc->state = PROC_UNINIT;                         // 设置进程状态为未初始化
26         proc->pid = -1;                                    // 设置进程ID为-1（还未分配）
27         proc->cr3 = boot_cr3;                              // 设置CR3寄存器的值（页目录基址）
28         proc->runs = 0;                                    // 设置进程运行次数为0
29         proc->kstack = 0;                                  // 设置内核栈地址为0（还未分配）
30         proc->need_resched = 0;                            // 设置不需要重新调度
31         proc->parent = NULL;                               // 设置父进程为空
32         proc->mm = NULL;                                   // 设置内存管理字段为空
33         memset(&(proc->context), 0, sizeof(struct context)); // 初始化上下文信息为0
34         proc->tf = NULL;                                   // 设置trapframe为空
35         proc->flags = 0;                                   // 设置进程标志为0
36         memset(proc->name, 0, PROC_NAME_LEN);              // 初始化进程名为0
37     }
38     return proc; // 返回新分配的进程控制块
39 }
```

在操作系统中，`proc_struct` 数据结构用于存储有关进程的各种信息。`struct context` 和 `struct trapframe` 是 `proc_struct` 中的成员，它们的**成员变量含义**如下：

1. `struct context context` 用于保存进程上下文，即进程被中断或切换出CPU时需要保存的几个关键的寄存器，如程序计数器（PC）、堆栈指针（SP）和其他寄存器。当操作系统决定恢复一个进程的执行时，会从这个结构体中恢复寄存器的状态，从而继续执行进程。在 `ucore` 中，`context` 结构通常在上下文切换函数 `switch_to` 中被使用。

2. `struct trapframe *tf` 指针指向一个 `trapframe` 结构，该结构包含了当进程进入内核模式时（比如因为系统调用或硬件中断）需要保存的信息。`trapframe` 保存了中断发生时的CPU状态，包括所有的寄存器值和程序计数器（epc）。这使得操作系统可以准确地了解中断发生时进程的状态，并且可以在处理完中断后恢复到之前的状态继续执行。在系统调用或中断处理的代码中经常会用到这个结构。

在本实验中，这两个结构在进程调度和中断处理的**作用**是：

- `context` 在不同进程之间进行上下文切换时保存和恢复进程状态。当调度器选择一个新的进程运行时，它会使用保存在 `context` 中的信息来设置CPU的状态，从而开始执行新进程。
- `tf` 在处理系统调用、异常或中断时使用。当进程从用户模式切换到内核模式时，用户模式下的状态（如寄存器）会被保存在 `trapframe` 中。内核完成处理后，可以使用这些信息来恢复进程的状态并继续用户模式下的执行。

这两个结构对于内核能够管理进程的执行，特别是在处理中断和系统调用、实现多任务处理方面至关重要。在设计和实现 `alloc_proc` 函数时，确保这些成员变量正确初始化是非常重要的，因为任何错误的状态都可能导致进程无法正确运行或系统崩溃。

练习2：为新创建的内核线程分配资源（需要编码）

> 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。因此，我们**实际需要"fork"的东西就是stack和trapframe**。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。

设计实现过程

`do_fork` 函数是创建新进程的核心函数。在UNIX-like系统中，`fork` 用来创建一个与当前进程几乎完全相同的子进程，实现的代码如下：

```
/* do_fork - 通过父进程来创建一个新的子进程
 * @clone_flags: 用于指导如何克隆子进程
 * @stack:       父进程的用户栈指针。如果stack为0，则意味着要fork一个内核线程。
 * @tf:          trapframe信息，它将被复制到子进程的proc->tf中
 */
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
{
    int ret = -E_NO_FREE_PROC; // 默认返回值，表示没有空闲的进程结构体
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) // 如果当前进程数已达到系统限制，则无法再创建新进程
    {
        goto fork_out;
    }
    ret = -E_NO_MEM; // 如果内存不足，则返回内存错误码

    /* LAB4:EXERCISE2 YOUR CODE
     * 你可以在以下实现中使用一些有用的宏、函数和定义。
     * 宏或函数:
     *   alloc_proc:   创建一个proc_struct结构体并初始化字段（lab4:exercise1）
     *   setup_kstack: 为进程分配大小为KSTACKPAGE的页面作为内核栈
     *   copy_mm:      根据clone_flags决定是复制还是共享当前进程的mm结构体
     *                 如果clone_flags设置了CLONE_VM，则"共享"；否则"复制"
     *   copy_thread:  在进程的内核栈顶部设置trapframe，并设置进程的内核入口点和栈
     *   hash_proc:    将进程添加到proc hash_list中
     *   get_pid:      为进程分配一个唯一的pid
     *   wakeup_proc:  将proc->state设置为PROC_RUNNABLE
     * 变量:
     *   proc_list:    进程集合的列表
     *   nr_process:   进程集合的数量
     */

    // 1. 调用alloc_proc来分配一个proc_struct
    if ((proc = alloc_proc()) == NULL)
        goto fork_out;
    // 2. 调用setup_kstack为子进程分配内核栈
    proc->parent = current; // 设置子进程的父进程为当前进程
    if (setup_kstack(proc))
        goto bad_fork_cleanup_kstack;
    // 3. 调用copy_mm根据clone_flag来复制或共享内存
    if (copy_mm(clone_flags, proc))
        goto bad_fork_cleanup_proc;
    // 4. 调用copy_thread来设置子进程的tf和context
    copy_thread(proc, stack, tf);
```

```
    // 5. 将新进程添加到进程列表和哈希表中
    bool intr_flag;
    local_intr_save(intr_flag); // 禁用中断
    {
        proc->pid = get_pid();                    // 为子进程分配一个唯一的进程ID
        hash_proc(proc);                          // 将新进程添加到哈希表中
        list_add(&proc_list, &(proc->list_link)); // 将新进程添加到进程列表中
    }
    local_intr_restore(intr_flag); // 恢复中断
    // 6. 调用wakeup_proc使新的子进程变为可运行状态
    wakeup_proc(proc);
    // 7. 使用子进程的pid作为返回值
    ret = proc->pid;

fork_out:
    return ret; // 返回子进程的PID或者错误码

bad_fork_cleanup_kstack:
    put_kstack(proc); // 如果内核栈设置失败，清理分配的内核栈
bad_fork_cleanup_proc:
    kfree(proc); // 清理分配的proc_struct
    goto fork_out;
}
```

`do_fork` 函数负责克隆当前进程的状态并生成一个新进程。该函数的执行始于调用 `alloc_proc` ，它构建了一个新的进程控制块作为线程的数据核心。我随后为新线程分配了内核栈，确保它有自己的运行空间。由于我们创建的是内核线程，不涉及用户空间，因此我们跳过了内存管理信息的复制。

接下来的步骤是复制执行上下文，这通过 `copy_thread` 完成，它在新线程的内核栈中设置了必要的初始状态。然后，新线程被加入到全局的进程列表中，这一步是在中断禁用的情况下进行的，以避免并发操作的冲突。通过调用 `wakeup_proc` ，新线程被标记为可运行状态。

最后，新线程的唯一进程标识符（PID）被分配并返回。这个PID是通过 `get_pid` 函数获取的，它保证了每个线程的PID的唯一性，因为 `get_pid` 会遍历现有的进程列表以避免ID冲突。

关于 `ucore` 是否为每个新fork的线程分配一个唯一的PID

`ucore` 确实能够为每个新fork的线程分配一个唯一的PID。通过 `get_pid` 函数的逻辑，它遍历当前所有进程，以确保新分配的PID不与任何现有进程的PID冲突。在 `get_pid` 函数中，"安全的PID"（ `next_safe` ）是指在 `get_pid` 函数中用来追踪可以安全分配而不会与现有进程的PID冲突的PID值，也就是说维护了一个性质：当前分配了 `last_pid` 这一PID后，下一次 `last_pid+1` 到 `next_safe-1` 的PID都是可以分配的。这个机制主要用于优化搜索性能，避免每次分配PID时都需要遍历整个进程列表来检查PID是否已被占用。

也就是说：

- `last_pid`：记录最后一次成功分配的PID。每次`get_pid`被调用时，它会从`last_pid + 1`开始寻找下一个可用的PID。
- `next_safe`：记录下一个没有被任何活动进程使用的PID。当`last_pid`达到或超过`next_safe`时，`get_pid`函数知道它需要重新扫描活动进程列表来更新`next_safe`的值。

这种方法可以减少每次寻找新PID时遍历进程列表的次数。例如，在一个PID范围大而活动进程相对少的系统中，这种优化可以显著提高PID分配的效率。如果`last_pid`加一后还没有到达`next_safe`，`get_pid`就可以安全地分配`last_pid`而无需遍历进程列表。只有在`last_pid`达到`next_safe`时，才需要检查进程列表来更新`next_safe`。

这个策略假定分配的PID会比较平均地分散在整个可用范围内，而且系统中的进程创建和销毁也是比较均匀的。在这些假设下，“安全的PID”（`next_safe`）充当了一个缓存的角色，指向一个确定没有被占用的PID，直到下一次必须重新扫描进程列表来更新这个信息为止。

基于`get_pid`函数的实现，只要`MAX_PID`大于`MAX_PROCESS`并且系统中的其他部分也正确地管理着PID的分配和回收，`ucore`就能为每个新fork的线程提供一个唯一的PID。这个设计假定系统不会在极短时间内创建超过`MAX_PID`数量的进程，这在实践中是合理的。

此外，`local_intr_save(intr_flag);`和`local_intr_restore(intr_flag);`在分配PID的过程中禁用和恢复中断，这保证了`get_pid`函数在寻找和分配新PID时不会被中断，从而避免了并发执行的问题。且在`ucore`的实验设置中，操作系统运行在单核环境中，因此，`do_fork`函数在分配PID时不会遇到来自其他CPU核心的并发执行问题，也就是说保证了`get_pid`的原子性。

练习3：编写proc_run 函数（需要编码）

> proc_run用于将指定的进程切换到CPU上运行。它的大致执行步骤包括：
>
> - 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。
> - 禁用中断。你可以使用`/kern/sync/sync.h`中定义好的宏`local_intr_save(x)`和`local_intr_restore(x)`来实现关、开中断。
> - 切换当前进程为要运行的进程。
> - 切换页表，以便使用新进程的地址空间。`/libs/riscv.h`中提供了`lcr3(unsigned int cr3)`函数，可实现修改CR3寄存器值的功能。
> - 实现上下文切换。`/kern/process`中已经预先编写好了`switch.S`，其中定义了`switch_to()`函数。可实现两个进程的context切换。
> - 允许中断。
>
> 请回答如下问题：
>
> - 在本实验的执行过程中，创建且运行了几个内核线程？
>
> 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。

`proc_run` 的实现如下：

```
// proc_run - 使得进程 "proc" 在CPU上运行
// 注意：在调用switch_to之前，应当加载"proc"的新页目录表(PDT)的基地址
void proc_run(struct proc_struct *proc)
{
    // 只有当proc不是当前进程时才需要进行上下文切换
    if (proc != current)
    {
        // LAB4:EXERCISE3 你的代码
        /*
         * 以下是一些有用的宏、函数和定义，你可以在下面的实现中使用它们。
         * 宏或函数:
         *   local_intr_save():         禁用中断
         *   local_intr_restore():      启用中断
         *   lcr3():                    修改CR3寄存器的值，用于切换当前使用的页目录表
         *   switch_to():               在两个进程之间进行上下文切换
         */
        // 定义用于保存中断状态的变量
        bool intr_flag;
        // 记录当前进程和即将运行的进程
        struct proc_struct *prev = current, *next = proc;

        // 禁用中断以保护上下文切换过程
        local_intr_save(intr_flag);
        {
            // 将当前进程更新为proc
            current = proc;
            // 加载新进程的页目录表到CR3寄存器并切换地址空间
            lcr3(next->cr3);
            // 执行上下文切换，切换到新进程
            switch_to(&(prev->context), &(next->context));
        }
        // 恢复之前的中断状态
        local_intr_restore(intr_flag);
    }
}
```

在本实验中，一共创建了两个内核线程，一个为 `idle` 另外一个为执行 `init_main` 的 `init` 线程。这是因为在 `proc_init` 函数中创建了两个内核线程：

1. idleproc: 这是第一个内核线程，称为idle线程，它的作用是在没有其他线程可运行时占据CPU。这个线程通常会运行一个简单的循环，等待其他线程变为可运行状态。

2. initproc: 这是第二个内核线程，由 `kernel_thread(init_main, "Hello world!!", 0)` 创建，这个线程开始执行 `init_main` 函数，也就是打印消息：

```
1 static int
2 init_main(void *arg)
3 {
4     cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid, get_prc
5     cprintf("To U: \"%s\".\n", (const char *)arg);
6     cprintf("To U: \"en.., Bye, Bye. :)\"\n");
7     return 0;
8 }
```

注意，这次实验 `init_main` 函数中是没有启动user_main内核线程的。

然后，在 `cpu_idle` 函数中会调用 `schedule` 调度函数，使可用的内核线程运行。

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?

在操作系统中，特别是在内核级别的代码中，经常需要临时禁用中断，以保护代码执行期间的临界区不被中断，从而避免竞态条件和保持数据一致性；就是说，起到了一个形成临界区的作用，`local_intr_save(intr_flag);` 和 `local_intr_restore(intr_flag);` 就是这样的一对宏（内联函数），用于实现这个目的。它们的工作原理为：

1. `local_intr_save(intr_flag);` 这个语句保存当前的中断状态，并禁用当前CPU的本地中断。 `intr_flag` 是一个局部布尔变量，用于保存旧的中断状态，以便稍后可以恢复。在RISC-V或者其他体系结构上，这通常涉及读取中断控制寄存器的当前值（如状态寄存器），并将该值存储在变量中。然后，先判断一下S态的中断使能位是否被设置了，如果设置了，则调用 `intr_disable` 函数设置中断控制寄存器的值以禁用中断。在RISC-V中，这可能涉及修改状态寄存器（ `sstatus` ）中的全局中断使能位（SIE）。

2. 禁用中断后，执行的代码块（即 `local_intr_save` 和 `local_intr_restore` 之间的代码）可以安全地执行，不会被中断打断。 `local_intr_restore(intr_flag);` 这个语句恢复之前通过 `local_intr_save` 保存的中断状态。它读取 `intr_flag` 变量，将中断状态寄存器设置回之前保存的值。如果先前的状态是允许中断的，这将重新启用中断。这样做保证了如果在进入临界区之前中断是启用的，那么离开临界区后中断仍然是启用的。
   `local_intr_save(intr_flag);` 会保存当前的中断状态到 `intr_flag` 并禁用中断（如果它们当前是使能的）。 `local_intr_restore(intr_flag);` 会根据 `intr_flag` 的值恢复之前的中断状态。如果 `intr_flag` 为1（true），中断会被重新使能。