Lab4 进程管理

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。

小组成员：张高2213219 张铭2211289 黄贝杰2210873

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）*

答：alloc_proc初始化了某个进程所需的进程控制块（PCB）结构体。该结构体是进程管理的重要工具。

初始化的过程实际上就是对PCB结构体中的各个属性赋值。需要注意的是，proc_init函数调用了

alloc_proc后，会检查这个初始化PCB的过程.在这里主要说明这些数值的含义（和设置的缘由），需要初

始化以下属性：

1. `**state**：此时未分配该PCB对应的资源，故状态为初始态。`````
2. **pid**：与state对应，表示无法运行。
3. **runs**：分配阶段故运行次数为0。
4. **kstack**：内核栈暂未分配。acclo_proc之后，idleproc的内核栈即为uCore启动时设置的内核栈，
5. 但之后的其他进程需要自行分配内核栈（在do_fork函数中实现）。
6. **need_resched**：不用调度其他进程、即CPU资源不分配。
7. **parent**：当前无父进程。但是按照之后的代码逻辑，即将该PCB的tf属性中的a0寄存器置0（意味

   着它是一个子进程），这个父进程被设置为了当前运行的进程。 一般来说`parent`指针在`alloc_proc`中通常会初始化为NULL，或者指向创建该进程的父进程。
8. **mm**：当前未分配内存。此后在do_fork函数中通过调用copy_mm函数被设置。
9. **context**：上下文置零。此后在do_fork函数中通过调用copy_thread函数被设置。
10. **tf**：当前无中断帧。此后在do_fork函数中通过调用copy_thread函数被设置。

11. **cr3**：内核线程同属于一个内核大进程，共享内核空间，故页表相同。

12. **flags**：当前暂无。

13. **name**：当前暂无。`

```
static struct proc_struct *

alloc_proc(void) {

  struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));

  if (proc != NULL) {

  //LAB4:EXERCISE1 YOUR CODE

  /*

   \* 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 cr3;                // CR3 register: 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; // 此时未分配该PCB对应的资源，故状态为初始态

  proc->pid = -1; // 与state对应，表示无法运行
```

```
    proc->runs = 0; // 分配阶段故运行次数为0
    proc->kstack = 0; // 内核栈暂未分配
    proc->need_resched = 0; // 不调度其他进程、即CPU资源不分配
    proc->parent = NULL; // 当前无父进程
    proc->mm = NULL; // 当前未分配内存
    memset(&(proc->context), 0, sizeof(struct context)); // 上下文置零
    proc->tf = NULL; // 当前无中断帧
    proc->cr3 = boot_cr3; // 内核线程同属于一个内核大进程，共享内核空间，故页表相同
    proc->flags = 0; // 当前暂无
    memset(&(proc->name), 0, PROC_NAME_LEN); // 当前暂无
    }
    return proc;
}
```

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）

答：

context： 表示进程上下文信息，包括程序计数器、寄存器状态、内存管理信息等。它记录了进程执行

的环境和状态，当进程被切换时，需要保存当前进程的上下文信息，并加载新进程的上下文信息。`struct context`用于保存和恢复进程的CPU寄存器状态，以便在进程切换时能够恢复执行。

tf：表示中断上下文信息，它是在处理中断或异常时，用于保存被中断进程的状态的数据结构。`struct trapframe *tf`用于保存和恢复进程的中断状态，以便在处理完中断后能够恢复进程的执行。

本实验中，可以说二者是相互配合实现进程切换的。proc_run函数中调用的switch_to函数，使用context保存

原进程上下文并恢复现进程上下文。然后，由于在初始化context时将其ra设置为forkret函数入口，所

以会返回到forkret函数，它封装了forkrets函数，而该函数的参数是当前进程的tf，该函数调用了

__trapret来恢复所有寄存器的值。

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。**

设计思路

do_fork函数为当前进程创建一个新进程，基本功能类似于 UNIX 或 Linux 系统中的 fork 系统调用。 该函数会分配一个新的进程结构体 (proc_struct)，为新进程分配内核栈， 复制内存管理信息，复制进程的上下文，最终将新进程添加到进程列表中并唤醒它。

代码实现

```
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:EXERCISE2:2211289 张铭
    /*
     * 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
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    //    2. call setup_kstack to allocate a kernel stack for child process
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    //    3. call copy_mm to dup OR share mm according clone_flag
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc, stack, tf);
    //    5. insert proc_struct into hash_list && proc_list
    int intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &proc->list_link);
        nr_process++;
    }
    local_intr_restore(intr_flag);
    //    6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc);
    //    7. set ret vaule using child proc's pid
    ret = proc->pid;

fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

回答问题

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 中通过 `get_pid` 函数为每个新进程分配唯一的 PID，该函数通过遍历进程列表来确保 PID 的唯一性 **`get_pid`分配pid逻辑** `get_pid` 使用 `last_pid` 变量来追踪上一个分配的 PID，确保新的 PID 大于上一个 PID。 当 `last_pid` 达到 `MAX_PID` 时，PID 会从 1 重新开始。 函数会遍历进程列表 `proc_list`，确保新分配的 PID 不与当前存在的进程冲突。

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proc_run函数

```
// 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 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
        *   lcr3():                   Modify the value of CR3 register
        *   switch_to():              Context switching between two
processes
        */
        // 禁用中断，保存中断状态
        bool intr_flag;
        local_intr_save(intr_flag);
        // 保存当前进程的上下文，并切换到新进程
        struct proc_struct * temp = current; // 将当前进程保存到临时变量 temp，
以便之后恢复其上下文
        current = proc; // 切换到新进程
        // 切换页表，以便使用新进程的地址空间
        // cause:
        // 为了确保进程 A 不会访问到进程 B 的地址空间
```

```
            // 页目录表包含了虚拟地址到物理地址的映射关系,将当前进程的虚拟地址空间映射关系
切换为新进程的映射关系.
            // 确保指令和数据的地址转换是基于新进程的页目录表进行的
            lcr3(current->cr3); // CR3 寄存器存储当前使用的页目录表(Page Directory
Table, PDT)的物理地址
            // 上下文切换
            // cause:
            // 保存当前进程的信息,以便之后能够正确地恢复到当前进程
            // 将新进程的上下文信息加载到相应的寄存器和寄存器状态寄存器中,确保 CPU 开始执
行新进程的代码
            // 禁用中断确保在切换期间不会被中断打断
            switch_to(&(temp->context),&(proc->context));
            // 恢复中断状态
            local_intr_restore(intr_flag);
        }
    }
```

回答问题

创建了两个内核线程：`idleproc` 和 `initproc`:

idelProc:

- `idleproc`是第一个内核线程（空闲线程），它是操作系统调度器的默认线程，用于在没有其他任务运行时占据 CPU。
- 分配进程控制块，调用`alloc_proc()`为`idelProc`分配一个新的`proc_struct`结构，用于表示一个内核线程。
- 将空闲线程设置为可以调度运行的状态，绑定其内核栈，分配线程名称。设置`PID`0，状态为`PROC_RUNNABLE`（表示可调度状态），具有一个指向内核栈的指针 kstack（绑定内核栈，指向`bootstack`），标志`need_resched`被设置为 1（标记需要立即调度），设置线程名称为 "idle"（用于执行`cpu_idle`函数）。在初始化时，完成新的内核线程创建后进入死循环，用于调度其他进程线程。
- 由于这是操作系统启动后的第一个线程，idleproc 被设置为当前线程。
- `idleproc`在`proc_init`完成时并未主动运行。它的运行依赖于调度器的选择：当没有其他线程需要运行时，调度器会选择`idleproc`。空闲线程的主要职责是维持 CPU 活动（通常是执行空循环或进入低功耗模式）。

initproc:

- `initproc`是第二个内核线程（初始化线程）,它通常负责加载用户进程、初始化设备驱动程序等系统初始化工作。
- 该线程通过`kernel_thread`函数创建，将`init_main`函数保存为该线程的入口函数，同时保存相关参数，主要是为新线程设置好其运行所需的上下文（`trapframe`），然后调用`do_fork`来完成线程的创建。
- 为该线程分配`proc_struct`并将当前线程设置为该线程的父线程，分配内存栈，复制父线程的内存布局（如页表和内存空间）和 CPU 上下文（寄存器状态、堆栈指针等）。
- 分配唯一 PID，将新线程从`PROC_UNINIT`状态唤醒到`PROC_RUNNABLE`状态，使其能够被调度执行。
- 将新线程的`proc_struct`插入进程哈希表`hash_list`，方便快速查找，同时将新线程添加到全局线程链表`proc_list`中。

- 新线程的执行上下文（`trapframe`）被复制，确保其从正确的状态开始运行。调度器选中后立即运行，开始执行`init_main`函数，完成系统初始化任务（如加载用户进程、设备驱动等）。

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？
local_intr_save(intr_flag)中调用的是__intr_save()

```
// 保存当前的中断使能状态，并将中断禁止

static inline bool __intr_save(void) {

if (read_csr(sstatus) & SSTATUS_SIE) //检查终端使能

{

intr_disable();//如果可以，就调用中断不能函数

return 1;

}

return 0;

}

//中断不能函数，就是清除sstatus上的中断使能位

void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }

//清楚中断使能位的具体过程

\#define clear_csr(reg, bit)

({ unsigned long __tmp; \

asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \

__tmp; })

//csrrc使读取、指令 控制和状态寄存器，clear_csr这个宏用于清除指定寄存器中的指定位。
```c

 这段代码通过检查和清除`sstatus`寄存器中的中断使能位来实现中断的保存和禁用`local_intr_save(intr_flag);`：这个宏会调用内联函数 `__intr_save()`，该函数

1. `__intr_save` 函数：

- 该函数用于保存当前的中断使能状态，并立即将中断禁用。
- 首先检查`sstatus`寄存器中的`SSTATUS_SIE`位，这个位用于指示中断是否被使能。
- 如果中断被使能（`SSTATUS_SIE`为1），则调用`intr_disable`函数来禁用中断，并返回`1`表示中断状态已改变。
```
```

```
- 如果中断未被使能（`SSTATUS_SIE`为`0`），则返回`0`表示中断状态未改变。

local_intr_restore(intr_flag);
```

static inline void __intr_restore(bool flag) {

if (flag) {

intr_enable();

}

}

void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }

#define set_csr(reg, bit) ({ unsigned long __tmp; \

asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \

__tmp; })

这段代码提供了中断使能状态的恢复功能。`__intr_restore`函数根据之前保存的状态决定是否恢复中断使能，而`intr_enable`函数和`set_csr`宏则用于实现这一操作。`set_csr`宏通过汇编指令直接操作寄存器，这是为了确保对硬件的低级控制，并且可以在不使用操作系统服务的情况下实现快速的中断使能。