lab4实验报告

实验名称：进程管理
学号：2211349 2212126 2213603
小组成员：周末 周思洁 李娅琦

1.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：

```
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE  2212126
    /*
     * 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;
        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->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);

    }
    return proc;
}
```

- 给进程设置为未初始化状态
- 未初始化的进程，其pid为-1
- 初始化时间片,刚刚初始化的进程，运行时间一定为零
- 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。
- 最初设置为不需要调度
- 初始化设置父进程为空
- 虚拟内存为空
- 初始化上下文
- 中断帧指针为空
- 页目录为内核页目录表的基址
- 标志位为0
- 进程名为0

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）

- context的作用：在uCore操作系统中，进程的上下文（context）是用于管理进程切换的关键数据结构。它主要保存了进程在被切换前的执行状态，即各个寄存器的值。这样，当内核需要在不同的进程之间进行切换时，可以通过context来保存和恢复进程的执行现场。在uCore中，即使是运行在内核态的进程，它们也是相互独立的。context的存在使得在内核态能够实现进程上下文的快速切换。具体实现上下文切换的函数是在kern/process/switch.S文件中定义的switch_to。

- tf（trapframe）：tf是一个指向中断帧的指针，它始终指向内核栈上的某个特定位置。每当进程从用户空间跳转到内核空间时，中断帧会记录下进程在被中断前的状态信息。当内核处理完毕，需要将控制权交回给用户空间时，会根据tf指向的中断帧来恢复进程的各个寄存器值，以便进程可以从中断点继续执行。此外，uCore内核支持中断的嵌套处理。为了确保在嵌套中断发生时，tf始终能够指向当前的中断帧，uCore在内核栈上维护了一个tf的链表。这样，无论何时，内核都能通过这个链表访问到正确的trapframe

2.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。请回答如下问题：

请在实验报告中简要说明你的设计实现过程。

根据文档中大致实验步骤所述，设计过程如下：

1. 调用alloc_proc，首先获得一块用户信息块。

```
if ((proc = alloc_proc()) == NULL) {
            goto fork_out;
}
proc->parent = current;//将子进程的父节点设置为当前进程
```

调用**alloc_proc()**申请内存块，如果失败，直接返回处理

2. 为进程分配一个内核栈

```
if (setup_kstack(proc)) {
            goto bad_fork_cleanup_proc;
}
```

调用**setup_kstack()**为进程分配一个内核栈

3. 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）

```
if(copy_mm(clone_flags, proc)){
            goto bad_fork_cleanup_kstack;
}
```

调用**copy_mm()**，复制父进程的内存信息到子进程，其中进程proc复制还是共享当前进程current，是根据clone_flags来决定的。

4. 复制原进程上下文到新进程

```
copy_thread(proc, stack, tf);
```

调用**copy_thread()** 复制父进程的中断帧和上下文信息

5. 将新进程添加到进程列表

```
bool intr_flag;
local_intr_save(intr_flag);//屏蔽中断, intr_flag置为1
{
    proc->pid = get_pid();//获取当前进程PID
    hash_proc(proc); //建立hash映射
    list_add(&proc_list, &(proc->list_link));//加入进程链表
    nr_process ++;//进程数加一
}
local_intr_restore(intr_flag);//恢复中断
```

- 调用**hash_proc()** 把新进程的PCB插入到哈希进程控制链表中

- 通过**list_add()**把PCB插入到进程控制链表中
- 并把总进程数+1
- 在添加到进程链表的过程中，使用了**local_intr_save()**和**local_intr_restore()**来屏蔽与打开，保证添加进程操作不会被抢断

6. 唤醒新进程

```
wakeup_proc(proc);
```

调用**wakeup_proc()来把当前进程的state设置为PROC_RUNNABLE**

7. 返回新进程号

```
ret = proc->pid;//返回当前进程的PID
```

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。

查看 **get_pid()** 获取进程id的函数可以知道：

- 使用静态变量`last_pid`记录上一个分配的pid
- 检查是否`last_pid`超过了`MAX_PID`
- 超过将`last_pid`重新设置为1，从头开始分配
- 未超过则遍历进程列表，检查是否有其他进程已经使用了当前的`last_pid`。
- 如果发现有其他进程使用了相同的pid，就将`last_pid`递增，并继续检查。
- 如果没有找到其他进程使用当前的`last_pid`，则说明`last_pid`是唯一的，函数返回该值。

因此每次调用`get_pid`都会尽力确保分配一个未被使用的唯一pid给新fork的线程。

3.练习3：编写proc_run 函数（需要编码）

proc_run的作用是将指定的进程切换到CPU上运行，通过禁用和恢复中断来安全地在两个进程之间进行上下文切换。它的大致执行步骤如下：

- 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。

函数实现

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // LAB4:EXERCISE3 YOUR CODE 2211349
        /*
        * 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
        */
       bool intr_flag;
       struct proc_struct *prev = current, *next = proc;
       local_intr_save(intr_flag);
       {
            current = proc;
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
       }
       local_intr_restore(intr_flag);
    }
}
```

实现思路

1. 判断当前进程是否与目标进程相同

```
if (proc != current)
```

在进行进程切换前，首先需要判断目标进程 proc 是否已经是当前进程 current。如果目标进程与当前进程相同，则不需要进行上下文切换，因为没有必要切换进程。通过 if (proc != current) 来判断，如果是相同的进程就直接跳过。

2. 禁用中断以避免上下文切换时的中断干扰

```
local_intr_save(intr_flag)
```

进程上下文切换的过程中，我们需要确保不会有外部中断或调度中断发生，因为中断可能会打断正在进行的上下文切换，导致数据不一致或切换不完整。为此，在切换开始时禁用中断，避免在切换过程中被打断。

3. 更新当前进程指针

```
current = proc
```

目标进程 proc 是新要运行的进程，因此需要将 current 指针更新为目标进程。这样操作系统就知道，当前正在执行的进程已经是目标进程了。

4. 切换页表（CR3寄存器）

```
lcr3(next->cr3)
```

每个进程都有独立的虚拟内存空间，操作系统通过页表将虚拟地址映射到物理内存地址。为了让目标进程能够正确地访问其虚拟内存空间，需要切换到目标进程的页表。页表的基地址由 CR3 寄存器保存，因此切换进程时需要更新 CR3 寄存器的值。

5. 执行上下文切换

```
switch_to(&(prev->context), &(next->context))
```

进程的上下文是指该进程在执行过程中所需保存的所有状态（例如寄存器的值、程序计数器等）。上下文切换的核心就是保存当前进程的上下文并恢复目标进程的上下文。此时操作系统会将当前进程的寄存器等状态保存到 prev->context 中，并从 next->context 恢复目标进程的状态。

6. 恢复中断

```
local_intr_restore(intr_flag)
```

在上下文切换完成后，恢复中断的状态，使得操作系统能够正常处理外部中断和调度中断。通过 local_intr_restore(intr_flag) 来恢复之前保存的中断状态。

问题：在本实验的执行过程中，创建且运行了几个内核线程?

在实验中创建并运行了两个内核线程：idleproc，initproc。

- idleproc：用于表示空闲进程，主要目的是在系统没有其他任务需要执行时，占用 CPU 时间，同时便于进程调度的统一化。是本次创建的第一个内核进程，完成内核中各个子系统的初始化后立即调度，执行其他进程。
- initproc：idleproc创建后通过调用kernel_thread函数来创建initproc内核线程，用于显示实验的功能体现而调度的内核进程。

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?

回答:

首先我们在kern/sync.h中找到对应的程序片段如下：

```
#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__ */
```

在上述代码中，`local_intr_save` 和 `local_intr_restore` 宏定义用于在函数或代码块中保存和恢复中断状态。以下是这两个宏如何实现开关中断的详细步骤：

local_intr_save(intr_flag);

这个宏调用了 `__intr_save` 函数，并将返回值保存在变量 `intr_flag` 中。以下是 `__intr_save` 函数的执行流程：

1. `read_csr(sstatus) & SSTATUS_SIE`：读取当前的状态寄存器 `sstatus` 并检查其中的中断使能位 `SSTATUS_SIE`。如果该位为1，表示中断当前是使能的。
2. 如果中断是使能的，`intr_disable()` 被调用，它会禁用中断（通过清除 `sstatus` 寄存器中的 `SSTATUS_SIE` 位）。
3. `__intr_save` 函数返回一个布尔值，如果中断原本是使能的，则返回1，否则返回0。这个返回值被保存在 `intr_flag` 变量中。

因此，`local_intr_save(intr_flag);` 的效果是保存当前的中断状态，并禁用中断（如果它们原本是使能的）。

local_intr_restore(intr_flag);

这个宏调用了 `__intr_restore` 函数，并将之前保存的 `intr_flag` 作为参数传递。以下是 `__intr_restore` 函数的执行流程：

1. 如果 `flag` 参数为真（即1），`intr_enable()` 被调用，它会重新使能中断（通过设置 `sstatus` 寄存器中的 `SSTATUS_SIE` 位）。
2. 如果 `flag` 参数为假（即0），则不执行任何操作，因为中断原本就是禁用的。

因此，`local_intr_restore(intr_flag);` 的效果是恢复之前保存的中断状态。如果 `intr_flag` 为真，则重新使能中断；如果为假，则保持中断禁用状态。

由此将两个宏定义函数结合起来， **local_intr_save(intr_flag);....local_intr_restore(intr_flag)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，保证进程切换原子性的目的。**

知识点

进程与线程

- **进程（Process）**：

- 是操作系统进行资源分配和调度的基本单位。
- 拥有独立的地址空间，一个进程崩溃后，在保护模式下不会影响到其他进程。
- 进程间的通信（IPC）较为复杂。

- **线程（Thread）**：
  - 是进程的执行单元，是CPU调度和分派的基本单位。
  - 线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器、一组寄存器和栈），但可以与同属一个进程的其他线程共享进程所拥有的全部资源。
  - 一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以直接通信。

- **区别**： 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。

进程作用

进程的一个重要特点在于其可以调度。在我们操作系统启动的时候，操作系统相当是一个初始的进程。之后，操作系统会创建不同的进程负责不同的任务。用户可以通过命令行启动进程，从而使用计算机。想想如果没有进程会怎么样？所有的代码可能需要在操作系统编译的时候就打包在一块，安装软件将变成一件非常难的事情，这显然对于用户使用计算机是不利的。

如果只让这些计算机服务于一个用户，有时候又有点浪费。有没有可能让一个计算机服务于多个用户呢（哪怕只有一个核心）？分时操作系统解决了这个问题，就是通过时间片轮转的方法使得多个用户可以"同时"使用计算资源。这个时候，引入进程的概念，成为操作系统调度的单元就显得十分必要了。