lab4

2112547张玉硕 2113099祝天智 2111288杜金轩

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(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));//分配空间
    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
     */
```

```
        // 初始化进程状态为 PROC_UNINIT，设置进程为“初始”态
        proc->state = PROC_UNINIT;
        // 初始化进程 ID 为 -1，设置进程pid的未初始化值
        proc->pid = -1;
        // 初始化运行次数为 0
        proc->runs = 0;
        // 初始化内核栈指针为 0
        proc->kstack = 0;
        // 初始化是否需要重新调度为 false
        proc->need_resched = 0;
        // 初始化父进程指针为 NULL
        proc->parent = NULL;
        // 初始化内存管理结构为 NULL
        proc->mm = NULL;
        // 初始化上下文结构
        memset(&proc->context, 0, sizeof(struct context));
        // 初始化中断帧指针为 NULL
        proc->tf = NULL;
        // 初始化 CR3 寄存器值为 boot_cr3?
        proc->cr3 = boot_cr3;
        // 初始化进程标志位为 0
        proc->flags = 0;
        // 初始化进程名字为空字符串，set_proc_name中以实现
        memset(proc->name, 0, PROC_NAME_LEN);
    }
    return proc;
}
```

在`struct proc_struct`中，`struct context context`和`struct trapframe *tf`是用于处理进程上下文切换和中断处理的关键成员。

1. `struct context context`：
    - 该成员用于保存进程在被内核调度之前的上下文信息。
    - 上下文切换是指从一个进程切换到另一个进程时，需要保存当前进程的状态，以便在将来再次执行该进程时能够从切换前的状态继续执行。
    - `struct context`可能包含处理器寄存器的信息，如静态寄存器（一般用于保存在函数调用之间需要保持不变的局部变量或全局变量）、栈指针等，以便在切换回进程时能够还原其执行状态。

```
struct context {// 用于进程或线程的上下文切换
    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;
};
```

2. `struct trapframe \*tf`：
   - 该成员是一个指针，指向一个用于存储中断或异常发生时CPU状态的数据结构，通常称为"trap frame"（陷阱帧）。
   - 在中断或异常发生时，CPU会将当前的执行状态保存到`tf`指向的结构中，包括寄存器的值、各种指针等。
   - 这个结构允许内核捕获和处理中断，然后在中断处理结束后恢复被中断的进程的状态，以确保进程可以从中断发生的地方继续执行。

```
struct trapframe {// 用于保存和表示一个中断或异常发生时的处理状态
    struct pushregs gpr;// 通用寄存器的值
    uintptr_t status;// 保存处理状态寄存器
    uintptr_t epc;// 保存异常程序计数器
    uintptr_t badvaddr;// 保存异常的虚拟地址
    uintptr_t cause;// 用于保存异常的原因或中断号
};
```

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。

```
// 分配并初始化进程控制块（alloc_proc函数）
// 分配并初始化内核栈（setup_stack函数）
// 根据clone_flags决定是复制还是共享内存管理系统（copy_mm函数）
// 设置进程的中断帧和上下文（copy_thread函数）
// 把设置好的进程加入链表
```

```
// 将新建的进程设为就绪态
// 将返回值设为线程id
    // Step 1: Call alloc_proc to allocate a proc_struct
    if ((proc = alloc_proc()) == NULL) {
        goto bad_fork_cleanup_proc;
    }
    // Step 2: Call setup_kstack to allocate a kernel stack for the child process
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    // Step 3: Call copy_mm to duplicate or share memory management
    if (copy_mm(clone_flags, proc) != 0) {// 本实验没有用
        goto bad_fork_cleanup_kstack;
    }

    // Step 4: Call copy_thread to set up the trapframe and context
    copy_thread(proc, stack, tf);

    // Step 5: Call hash_proc to add the child process to the hash list
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        // 生成并设置新的pid
        proc->pid = get_pid();
        // 把proc加入全局线程控制块哈希表
        hash_proc(proc);
        // 把proc加入全局线程控制块双向链表
        list_add(&proc_list, &(proc->list_link));
        nr_process ++;
    }
    local_intr_restore(intr_flag);
    // Step 6: Call wakeup_proc to mark the new child process as RUNNABLE
    wakeup_proc(proc);// PROC_RUNNABLE

    // Step 7: Set the return value using the child process's PID
    cprintf("THIS MY: do_fork proc create over thread: %d! isNULL:%d \n", proc->pid, proc == NULL);
    ret = proc->pid;
    goto fork_out;

```

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。

```
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);// PID 的范围应该大于最大进程数
```

```
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid)
{
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

- **PID的分配方式**：
  - 通过 `last_pid` 和 `next_safe` 变量，`get_pid` 函数找到一个尚未分配的PID。`last_pid` 是上一个分配的PID，而 `next_safe` 用于跟踪下一个安全的PID。当需要分配新的PID时，`get_pid` 会递增 `last_pid` 直到找到一个未被使用的PID，并确保这个PID 小于`next_safe` 。
- **唯一性保证**：
  - 在进程创建过程中，`get_pid` 函数通过遍历进程列表，检查每个进程的PID，确保新分配的PID不与现有进程的PID冲突。这是通过在列表中查找相同PID的进程来实现的。
  - 如果当前分配的PID已经存在（即与某个进程的PID相同），则会递增 `last_pid` 并重新检查，确保分配的PID是唯一的。

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？

两个

创建第 0 个内核线程 idleproc，第0个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行cpu_idle函数开始过退休生活了。

创建第 1 个内核线程 initproc

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？

```
// 保存中断状态并关闭中断
static inline bool __intr_save(void) {
    // 读取 SSTATUS 寄存器，检查 SIE 位（中断使能位）
    if (read_csr(sstatus) & SSTATUS_SIE) {
        // 如果中断已经使能，关闭中断并返回1表示中断在调用函数之前是开启的
        intr_disable();
        return 1;
    }
    // 如果中断未使能，返回0表示中断在调用函数之前是关闭的
    return 0;
}

// 恢复中断状态
static inline void __intr_restore(bool flag) {
    // 根据保存的中断状态 flag 决定是否开启中断
    if (flag) {
        intr_enable();
    }
}
```

  - `local_intr_save`宏通过调用`__intr_save`函数保存中断状态并关闭中断。
  - `local_intr_restore`宏通过调用`__intr_restore`函数根据保存的中断状态决定是否恢复中断。