

Lab4

马浩祎 卢艺晗 陆皓喆

练习1：分配并初始化一个进程控制块（需要编码）

问题

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）

解答

设计实现

我们首先定位到proc_struct结构的位置（proc.h），结构信息如下所示：

```
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
};
```

我们需要做的就是将这些变量进行初始化即可。

根据指导书上的提示，设置进程为“初始”态，我们需要使用PROC_UNINIT；设置进程pid的未初始化值，我们需要使用-1；使用内核页目录表的基址，我们需要使用boot_cr3。所以根据以上，我们的设计如下所示：

```

static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state=PROC_UNINIT;
        proc->pid=-1;
        proc->runs=0;
        proc->cr3=boot_cr3;
        proc->kstack=0;
        proc->need_resched=0;
        proc->parent = NULL;
        proc->mm = NULL;
        proc->tf=NULL;
        proc->flags = 0;
        memset(&proc->name, 0, PROC_NAME_LEN);
        memset(&proc->context,0,sizeof(struct context));
    }
    return proc;
}

```

解释：

- state: 设置进程为初始态，我们需要将其设置为PROC_UNINIT
- pid: 未初始化的进程号我们需要设置为-1
- runs: 初始化时间片，刚刚初始化的进程，运行时间设置为0
- cr3: 使用内核页目录表的基址，我们需要使用boot_cr3
- kstack: 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
- need_resched: 是一个用于判断当前进程是否需要被调度的bool类型变量，为1则需要进行调度。初始化的过程中我们不需要对其进行调度，因此设置为0
- parent: 父进程为空，设置为NULL
- mm: 虚拟内存为空，设置为NULL
- tf: 中断帧指针为空，设置为NULL
- flags: 标志位flags设置为0
- memset(&proc->name, 0, PROC_NAME_LEN): 进程名name初始化为0
- memset(&proc->context,0,sizeof(struct context)): 初始化上下文，将上下文结构体context初始化为0

通过以上的代码，我们就可以完成PCB的分配和初始化！

问题

- `struct context context`: context是保存进程执行的上下文，也就是关键的几个寄存器的值。可用于在进程切换中还原之前的运行状态。在通过 `proc_run` 切换到CPU上运行时，需要调用 `switch_to` 将原进程的寄存器保存，以便下次切换回去时读出，保持之前的状态。

具体分析：

context的结构：

```

struct context {
    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;
};
//ra: 返回地址寄存器，用于保存函数调用后的返回地址。
//sp: 栈指针寄存器，指向当前线程的栈顶。
//s0 到 s11: 这些是保存寄存器（scratch registers），用于保存临时数据，它们在函数调用时不需要
被保存和恢复，因为它们不会被调用者所保留。

```

为什么不全保存？

- 我们知道寄存器可以分为调用者保存（caller-saved）寄存器和被调用者保存（callee-saved）寄存器。因为线程切换在一个函数当中（我们下一小节就会看到），所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码，在实际的进程切换过程中我们**只需要保存被调用者保存寄存器**就好啦！

首先看proc_run函数：

```

void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        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);
    }
}

```

具体，首先将当前的进程赋值为proc，然后更新cr3寄存器的值——它指向页目录表（Page Directory Table, PDT）的基地址

需要调用 `switch_to` 将原进程的寄存器保存，以便下次切换回去时读出，保持之前的状态。

`local_intr_save`和`local_intr_restore`都是控制开关中断，来保证我们中间的函数处理不出问题。

- `struct trapframe *tf`: 保存了进程的中断帧 (32个通用寄存器、异常相关的寄存器)。在进程从用户空间跳转到内核空间时, 系统调用会改变寄存器的值。我们可以通过调整中断帧来使的系统调用返回特定的值。比如可以利用 `s0` 和 `s1` 传递线程执行的函数和参数; 在创建子线程时, 会将中断帧中的 `a0` 设为 0。

```

struct pushregs {
    uintptr_t zero; // Hard-wired zero
    uintptr_t ra;   // Return address
    uintptr_t sp;   // Stack pointer
    uintptr_t gp;   // Global pointer
    uintptr_t tp;   // Thread pointer
    uintptr_t t0;   // Temporary
    uintptr_t t1;   // Temporary
    uintptr_t t2;   // Temporary
    uintptr_t s0;   // Saved register/frame pointer
    uintptr_t s1;   // Saved register
    uintptr_t a0;   // Function argument/return value
    uintptr_t a1;   // Function argument/return value
    uintptr_t a2;   // Function argument
    uintptr_t a3;   // Function argument
    uintptr_t a4;   // Function argument
    uintptr_t a5;   // Function argument
    uintptr_t a6;   // Function argument
    uintptr_t a7;   // Function argument
    uintptr_t s2;   // Saved register
    uintptr_t s3;   // Saved register
    uintptr_t s4;   // Saved register
    uintptr_t s5;   // Saved register
    uintptr_t s6;   // Saved register
    uintptr_t s7;   // Saved register
    uintptr_t s8;   // Saved register
    uintptr_t s9;   // Saved register
    uintptr_t s10;  // Saved register
    uintptr_t s11;  // Saved register
    uintptr_t t3;   // Temporary
    uintptr_t t4;   // Temporary
    uintptr_t t5;   // Temporary
    uintptr_t t6;   // Temporary
};

struct trapframe {
    struct pushregs gpr; // 32个通用寄存器
    uintptr_t status; // 下面是一些处理异常的寄存器
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};

```

- `s0`: `s0` 寄存器保存函数指针
- `s1`: `s1` 寄存器保存函数参数

练习2：为新创建的内核线程分配资源（需要编码）

问题

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。

解答

设计实现

我们按照提示的步骤进行编程设计：

- 调用alloc_proc，首先获得一块用户信息块

```
if ((proc = alloc_proc()) == NULL) {  
    goto fork_out;  
} //调用alloc_proc函数，获取一个新的用户信息块，如果为空的话就跳转到fork_out做  
失败返回处理
```

- 为进程分配一个内核栈

```
if (setup_kstack(proc) != 0) {  
    goto bad_fork_cleanup_proc;  
} //调用setup_kstack函数，分配一个内核栈，如果没有的话就坐失败处理
```

- 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) != 0) {  
    goto bad_fork_cleanup_kstack;  
}
```

调用copy_mm函数

```
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    assert(current->mm == NULL);
    return 0;
}
```

确认了一下当前进程的虚拟内存为空

- 复制原进程上下文到新进程

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

调用copy_thread()函数复制父进程的中断帧和上下文信息

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe
*tf) {
    proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE -
sizeof(struct trapframe)); //分配栈顶
    *(proc->tf) = *tf; //放进来
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;
    proc->context.ra = (uintptr_t)forkret; //进一步forkrets
    proc->context.sp = (uintptr_t)(proc->tf); //中断帧
}
```

- 将新进程添加到进程列表

- `proc->pid=get_pid();`
`hash_proc(proc);`
`list_add(&proc_list, &(proc->list_link));`

首先，获取当前进程PID；然后调用hash_proc函数把新进程的PCB插入到哈希进程控制链表中，然后通过list_add函数把PCB插入到进程控制链表中

- 唤醒新进程

- `wakeup_proc(proc);`

- 返回新进程号

- `ret=proc->pid;`

通过以上的步骤，我们就可以完成为新创建的内核线程分配资源的任务！

问题

我们分析对应的函数get_pid：

```
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);
```

```

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;
}

```

这个函数完成了对于pid的分配。我们分析其逻辑：

last_pid是上一个分配的pid号，当get_pid函数被调用的时候，首先会检查是否 last_pid 超过了最大的pid值（MAX_PID）。如果超过了，将 last_pid 重新设置为1，从头开始分配。

如果 last_pid 没有超过最大值，就进入内部的循环结构。在循环中，它遍历进程列表，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。

综上所述，该函数的主要流程为：遍历每一个pid号，找到第一个当前没有被其他进程使用的pid号，即为我们得到的pid号。这样可以保证我们每个新fork的线程都是唯一的id号。

练习3：编写proc_run 函数（需要编码）

问题

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。

解答

设计实现

下面是我们设计的代码：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        local_intr_save(intr_flag);
        {
            struct proc_struct *curr_proc = current;
            current = proc;
            switch_to(&(curr_proc->context), &(proc->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

首先，完成中断状态的保存，调用了 `local_intr_save` 函数，并将保存的中断状态存储在 `intr_flag` 变量中。

然后，就开始做上下文切换。

- 首先，通过 `struct proc_struct *curr_proc = current;` 保存当前正在运行的进程结构体指针到 `curr_proc` 变量中。这一步很重要，因为后续需要根据当前进程的上下文进行切换操作。
- 然后，将 `current` 指针更新为要运行的进程 `proc`，这表示从现在起，系统将认为 `proc` 是当前正在运行的进程）。
- 最后，调用 `switch_to` 函数，传入当前进程（`curr_proc`）的上下文结构体指针和要运行的进程（`proc`）的上下文结构体指针。

在完成进程切换相关的核心操作后，通过调用 `local_intr_restore` 函数，并传入之前保存的中断状态 `intr_flag`，来恢复系统的中断状态。

问题

在本实验的执行过程中，创建且运行了**2个内核线程**：

- **idleproc**：第一个内核进程，完成内核中各个子系统的初始化，之后立即调度，执行其他进程。
- **initproc**：用于完成实验的功能而调度的内核进程。

扩展练习 Challenge

问题

说明语句 `local_intr_save(intr_flag); ... local_intr_restore(intr_flag);` 是如何实现开关中断的？

解答

这两个语句的相关定义如下所示：

```
#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();
    }
}

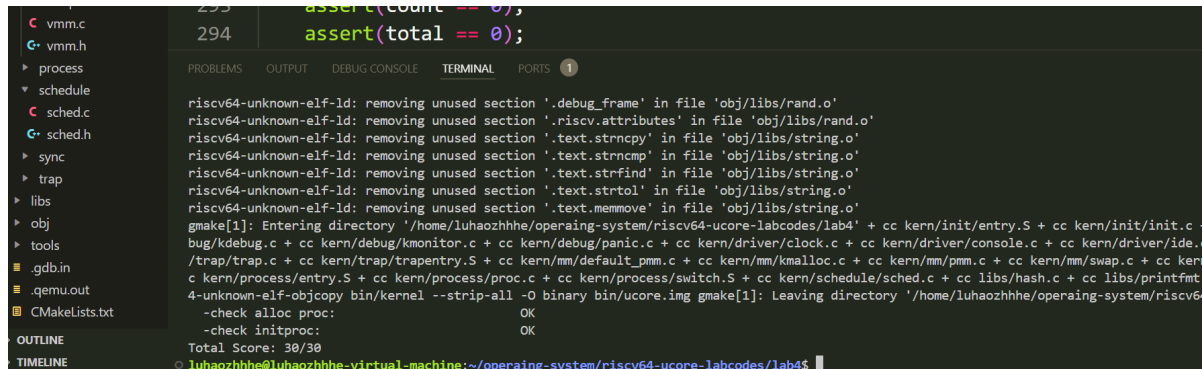
#define local_intr_save(x) \
    do { \
        x = __intr_save(); \
    } while (0)
#define local_intr_restore(x) __intr_restore(x);

#endif /* !__KERN_SYNC_SYNC_H__ */
```

当调用 `local_intr_save` 时，会读取 `sstatus` 寄存器，判断 `SIE` 位的值，如果该位为1，则说明中断是能进行的，这时需要调用 `intr_disable` 将该位置0，并返回1，将 `intr_flag` 赋值为1；如果该位为0，则说明中断此时已经不能进行，则返回0，将 `intr_flag` 赋值为0。这样就可以保证之后的代码执行时不会发生中断。

当需要恢复中断时，调用 `local_intr_restore`，需要判断 `intr_flag` 的值，如果其值为1，则需要调用 `intr_enable` 将 `sstatus` 寄存器的 `SIE` 位置1，否则该位依然保持0。以此来恢复调用 `local_intr_save` 之前的 `SIE` 的值。

实验结果



```
293     assert(count == 0);
294     assert(total == 0);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  1

riscv64-unknown-elf-ld: removing unused section '.debug_frame' in file 'obj/libs/rand.o'
riscv64-unknown-elf-ld: removing unused section '.riscv.attributes' in file 'obj/libs/rand.o'
riscv64-unknown-elf-ld: removing unused section '.text.strncpy' in file 'obj/libs/string.o'
riscv64-unknown-elf-ld: removing unused section '.text.strncmp' in file 'obj/libs/string.o'
riscv64-unknown-elf-ld: removing unused section '.text.strfind' in file 'obj/libs/string.o'
riscv64-unknown-elf-ld: removing unused section '.text.strtol' in file 'obj/libs/string.o'
riscv64-unknown-elf-ld: removing unused section '.text.memmove' in file 'obj/libs/string.o'
gmake[1]: Entering directory '/home/luhaozhhe/operating-system/riscv64-ucore-labcodes/lab4' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/ide.c + cc kern/driver/keyboard.c + cc kern/driver/rtc.c + cc kern/driver/serial.c + cc kern/driver/usb.c + cc kern/mm/kmalloc.c + cc kern/mm/pmm.c + cc kern/mm/swap.c + cc kern/process/entry.S + cc kern/process/proc.c + cc kern/process/switch.S + cc kern/schedule/sched.c + cc libs/hash.c + cc libs/printfmt.c
4-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/luhaozhhe/operating-system/riscv64-ucore-labcodes/lab4'
-check alloc proc: OK
-check initproc: OK
Total Score: 30/30
luhaozhhe@luhaozhhe-virtual-machine:~/operating-system/riscv64-ucore-labcodes/lab4$
```