实验工作文档

11.18 LAB4

操作系统Lab4实验报告

By 叶潇晗（2112120）, 张振铭（2112189），林子淳（2114042）

一、实验练习

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）

编程解答内容：

实现内核线程的第一步是给线程创建进程（ucore中的线程相当于一个不拥有资源的轻量级进程）控制块。
在kern/process/proc.c的alloc_proc函数中，给要创建的进程控制块指针（struct proc_struct *proc）分配了内存空间，设置如下变量：

```
/*       enum proc_state state;                      // Process state
 *       int pid;                                    // Process ID
 *       int runs;                                   // the running times of Proc
 *       uintptr_t kstack;                           // Process kernel stack
 *       volatile bool need_resched;                 // bool value: need to be re
 *       struct proc_struct *parent;                 // the parent process
 *       struct mm_struct *mm;                       // Process's memory manageme
 *       struct context context;                     // Switch here to run proces
 *       struct trapframe *tf;                       // Trap frame for current in
 *       uintptr_t cr3;                              // CR3 register: the base ad
```

```
11 *       uint32_t flags;                               // Process flag
12 *       char name[PROC_NAME_LEN + 1];                 // Process name
13 */
```

各个变量的详细解释如下：

- state：进程状态，proc.h中定义了四种状态：创建（未初始化，UNINIT）、睡眠（SLEEPING）、就绪（RUNNABLE）、退出（ZOMBIE，等待父进程回收其资源）
- pid：进程ID，调用本函数时尚未指定，默认值设为-1
- runs：线程运行总数，默认值0
- need_resched：标志位，表示该进程是否需要重新参与调度以释放CPU，初值0（false，表示不需要）
- parent：父进程控制块指针，初值NULL
- mm：用户进程虚拟内存管理单元指针，由于系统进程没有虚存，其值为NULL
- context：进程上下文，默认值全零
- tf：中断帧指针，默认值NULL
- cr3：该进程页目录表的基址寄存器，初值为ucore启动时建立好的内核虚拟空间的页目录表首地址boot_cr3（在kern/mm/pmm.c的pmm_init函数中初始化）
- flags：进程标志位，默认值0
- name：进程名数组，这里将已有的进程名称对象赋值为0即可

可以写出初始化代码：

```
1  proc->state = PROC_UNINIT;
2  proc->pid = -1;
3  proc->runs = 0;
4  proc->kstack = 0;
5  proc->need_resched = 0;
6  proc->parent = NULL;
7  proc->mm = NULL;
8  memset(&(proc->context), 0, sizeof(struct context));
9  proc->tf = NULL;
10 proc->cr3 = boot_cr3;
11 proc->flags = 0;
12 memset(proc->name, 0, PROC_NAME_LEN);
```

回答问题

> *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?*

context指进程上下文，proc.h中可以看到这一结构体存储了12个无符号指针，这其实是用于保存创建进程时父进程的部分寄存器值：eip, esp, ebx, ecx, edx, esi, edi, ebp，32位和64位均可保存。其他寄存器在切换进程时值不变，故不需要保存。

tf是中断帧的指针，总是指向内核栈的某个位置。此结构再trap.h中定义，包含了状态、错误地址和原因和EPC寄存器四个指针，以及寄存器堆结构。当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值，这就在tf结构中可以完全找到。

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。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

根据题目要求可知，do_fork()函数的实现大致步骤包括七步，我们也根据提示，再参考代码文件当中提供的注释信息，给出do_fork()函数的实现过程如下：

1. 调用alloc_proc()函数申请内存块，如果失败，直接返回处理。

alloc_proc()函数在练习1中实现过，在alloc_proc()函数中，如果分配进程PCB失败，也就是说因为空间不足等原因导致进程一开始就是NULL，那么就不会分配初始化资源，而是返回NULL。因此在do_fork()里面，也要使用相应的if语句判断alloc_proc()是否正确返回初始化资源。

2. 调用setup_kstack()函数为进程分配一个内核栈。

观察此函数的代码：

```
1 static int
2 setup_kstack(struct proc_struct *proc) {
3     struct Page *page = alloc_pages(KSTACKPAGE);
4     if (page != NULL) {
5         proc->kstack = (uintptr_t)page2kva(page);
6         return 0;
7     }
8     return -E_NO_MEM;
9 }
```

从中可以看到，如果页不为空的时候，会return 0，也就是说分配内核栈成功了,否则返回表示没有足够内存空间可供分配的错误信息，即return -E_NO_MEM。因此，在do_fork()函数当中我们调用该函数分配一个内核栈空间，并判断是否分配成功，如果分配失败，则跳转到处理相应错误这部分的代码。

3. 调用copy_mm()函数，复制父进程的内存信息到子进程

对于这个函数，观察它的代码：

```
1 static int
2 copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
3     assert(current->mm == NULL);
4     /* do nothing in this project */
5     return 0;
6 }
```

可以看到，这个函数里面实际上没有做任何的事情，而是之间return 0表示成功。我们可以在本次实验当中的do_fork()里面不调用这个函数，不过为了保证题目需求的完整性，还是添加了该函数的调用及

相应的判断语句。

4. 调用copy_thread()函数复制父进程的中断帧和上下文信息

copy_thread()函数需要传入的三个参数，第一个是比较熟悉，练习1中已经实现的PCB模块proc结构体的对象，第二个参数是一个栈，第三个参数是练习1PCB中的中断帧的指针。以下是该函数的代码：

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf)
{
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct tr
    *(proc->tf) = *tf;

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;

    proc->context.ra = (uintptr_t)forkret;
    proc->context.sp = (uintptr_t)(proc->tf);
}
```

可以得知，copy_thread()函数的作用是在内核中创建一个新线程的副本，包括中断帧、寄存器状态等信息。这通常在创建新的进程或线程时使用，以确保它们具有正确的初始状态并能够正确执行。这个函数还根据传入的参数设置了新线程的一些特殊状态，例如返回值和栈指针。

5. 将新进程添加到进程链表与哈希链表当中

在项目当中不仅有普通的管理进程的链表，还有为查找效率而设立的哈希链表,因此我们需要将新进程分别插入到两种链表当中，使用对应的链表插入函数即可。不过在此之前，由于这个新进程刚刚初始化，还要调用get_pid()函数来为这个进程分配一个PID。最后将表示进程数目的全局变量nr_process加1即可。

此外，我们为了保障程序的数据一致性，即不会出现像哈希链表存储了新进程信息，而进程链表没有新进程信息这样的情况，我们在这一系列操作的前后加上了语句local_intr_save(intr_flag);....local_intr_restore(intr_flag)，保证了插入新进程操作的原子性，从而保障我们程序的数据一致性

6. 调用wakeup_proc()函数将新进程唤醒

观察wakeup_proc()函数的代码：

```
void
wakeup_proc(struct proc_struct *proc) {
    assert(proc->state != PROC_ZOMBIE && proc->state != PROC_RUNNABLE);
    proc->state = PROC_RUNNABLE;
}
```

不难发现，该函数实际上是将进程proc的成员变量state设置为PROC_RUNNABLE，表示这是一个可运行的线程。因为每个进程初始状态都是PROC_UNINIT，即未初始化的状态，因此需要调用wakeup_proc()函数来改变进程状态。

7. 返回新进程号

在do_fork()函数内有局部变量ret，在函数执行完毕后会return ret，那么只需要将新线程的PID赋值给ret即可。

综合以上，我们可以给出do_fork()函数的完整代码：

```
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 2112189

    //1.调用alloc_proc()函数申请内存块
    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

    //2.调用setup_stack()函数为进程分配一个内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
```

```
    }

    //3.调用copy_mm()函数复制父进程的内存信息到子进程
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    //4.调用copy_thread()函数复制父进程的中断帧和上下文信息
    copy_thread(proc, stack, tf);

    //5.将新进程添加到进程的（hash）列表中
    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1
    }
    local_intr_restore(intr_flag);//恢复中断

    wakeup_proc(proc);//6.唤醒新进程

    ret = proc->pid;//7.返回当前进程的PID

fork_out:
    return ret;

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

问题回答：

ucore可以做到给每个新fork的线程一个唯一的id，理由如下：

首先，观察get_pid()函数的代码：

```
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;
}
```

可以看到，在其中使用了两个静态变量last_pid和next_safe。last_pid用于记录上一个进程的进程号，在经过本次处理后也会作为本次返回的PID值，而next_safe用于维护最小的一个不可用进程号。设置变量next_safe的目的是为了缩小查找空间，减少查找操作的开销，提高查找效率。

因此，由于静态变量的性质，每一次进入get_pid后，可以直接从(last_pid,next_safe)这个开区间中直接获得一个可用的进程号，从last_pid+1开始检查，直到这个区间中不存在进程号，也就是last_pid+1==next_safe的时候。此时，通过遍历进程链表，在整个进程号空间中寻找最小可用的进程号，将last_pid更新为该进程号，在循环进程链表的同时将next_safe更新为大于last_pid的最小不可用进程号，从而方便下一次获取进程号，最后返回last_pid。

由于代码当中的last_pid每次遇到了同现有进程相等的PID时，都会自增，从而get_pid()函数返回的PID不会与现有的任何一个进程相同，会为该进程分配一个唯一的PID。

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?

```
1  void
2  proc_run(struct proc_struct *proc) {
3      if (proc != current) {
4          bool intr_flag;// 中断变量
5          struct proc_struct *prev = current, *next = proc;
6          local_intr_save(intr_flag); // 屏蔽中断
7          {
8              current = proc;//当前进程为切换新进程
9              lcr3(next->cr3);//进程间的页表切换
10             switch_to(&(prev->context), &(next->context));//上下文切换
11         }
```

```
        local_intr_restore(intr_flag); // 允许中断
    }
}
```

proc_run 函数是操作系统内核中用于切换当前运行的进程的功能。当需要将CPU的控制从一个进程切换到另一个进程时，就会调用这个函数。

函数首先检查传入的进程 proc 是否与当前正在运行的进程 current 不同。如果它们不同，定义 intr_flag 并调用 local_intr_save(intr_flag) 函数来禁用中断（这里使用了intr_disable的宏定义），以确保在上下文切换过程中不会被中断干扰。接下来将将全局变量 current 更新为新的进程 proc，更新 CR3 寄存器的值，以切换到新进程的页表。同时执行 switch_to 函数进行实际的上下文切换，其保存当前进程 prev 的上下文（如寄存器状态等），并加载新进程 next 的上下文，以便新进程可以从其上次停止的地方继续执行。最后，调用local_intr_restore(intr_flag) 恢复中断（这里使用了intr_enable的宏定义），允许再次发生中断。

本次实验一共运行了两个内核线程，分别是idle和init。

1. idle：idle线程的作用是在没有其他线程可运行时占据CPU。其运行一个简单的循环，当其他线程变为可运行状态就会跳转到该线程中执行。这个实验中idle启动后，设置init为runnable，便将运行权交给init进程。
2. init：init线程由 kernel thread(init main,"Hello world!!"，0)创建，这个线程通过执行 init_main 函数打印“Hello World”信息，表示内核线程已经初始化成功并且可以正常使用。

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？

观察 `sync/sync.h` 当中local_intr_save以及local_intr_restore的定义如下：

```
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);
```

可以看到，local_intr_save宏首先调用__intr_save()函数，而这个函数会通过read_csr()函数来读取CSR寄存器的sstatus的值，来检查当前的中断状态。如果中断是开启的（SSTATUS_SIE位为1），那么它会调用intr_disable()函数来关闭中断并返回1；否则，它会返回0。这个返回值会被保存到在实际使用时传入的intr_flag变量中。

local_intr_restore宏实际上是调用__intr_restore()检查flag参数。如果flag为1表明之前使用过local_intr_save宏，原本中断是被关闭的，那么这时它会通过调用intr_enable()函数开启中断。

所以，通过先调用local_intr_save，后调用local_intr_restore，从而在两者之间形成了临界区，临界区前保存中断位，临界区的代码在中断关闭的状态下运行，并在临界区代码执行完毕后恢复原来的中断状态。它们之间的搭配保证了临界区内所执行操作的原子性，保障了程序的数据一致性。