lab4_team

练习1：分配并初始化一个进程控制块

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，。

在alloc_proc函数的实现中，需要初始化的proc_struct结构中的成员变量包括：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。

实验指导书提出的问题是：

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?

在 `struct proc_struct` 结构体中，维护了如下所示的变量

```
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 resche
    struct proc_struct *parent;                 // the parent process
    struct mm_struct *mm;                       // Process's memory management f
    struct context context;                     // Switch here to run process
    struct trapframe *tf;                       // Trap frame for current interr
    uintptr_t cr3;                              // CR3 register: the base addr c
    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
};
```

其中变量的基本信息都在注释中有一定的描述，我们具体看一下 `struct context context` 和 `struct trapframe *tf` 。

`context`： `context` 中保存了进程执行的上下文，也就是几个关键的寄存器的值。这些寄存器的值用于在进程切换中还原之前进程的运行状态（进程切换的详细过程在后面会介绍），其中包含了 `ra` ， `sp` ， `s0~s11` 共14个寄存器。切换过程的实现在 `kern/process/switch.S`

`tf`： `tf` 里保存了进程的中断帧。当进程从用户空间跳进内核空间的时候，进程的执行状态被保存在了中断帧中（注意这里需要保存的执行状态数量不同于上下文切换）。系统调用可能会改变用户寄存器的值，我们可以通过调整中断帧来使得系统调用返回特定的值。

初始化

```
    // in proc.c allocpage
    memset(&(proc->context),0,sizeof(struct context));
    // tf结构体指针初始化NULL
    proc->tf = NULL;
```

我们在初始化这两个变量时，分别将它们赋值为0和空指针。

```
// in proc.c proc_init
kernel_thread(init_main, "Hello world!!", 0);
```

tf作为函数调用以及状态设置

在创建内核线程时，又调用了 `kernel_thread` 函数， `init_main` 是对应的函数指针，在本次实验中输出一些必要的日志信息。

```
// kernel_thread - create a kernel thread using "fn" function
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));
    tf.gpr.s0 = (uintptr_t)fn;
    tf.gpr.s1 = (uintptr_t)arg;
    tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;
    tf.epc = (uintptr_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

将fn函数的地址转换为uintptr_t类型，并将其赋值给tf.gpr.s0，即保存在tf的寄存器s0中。将arg参数的地址转换为uintptr_t类型，并将其赋值给tf.gpr.s1，即保存在tf的寄存器s1中。设置tf.status字段，将当前read_csr(sstatus)寄存器的值与SSTATUS_SPP、SSTATUS_SPIE进行按位或操作，并将结果与SSTATUS_SIE的反码相与，然后赋值给tf.status。这样设置了线程的特权级（SPP）、中断使能（SPIE）和外部中断使能（SIE）。

将kernel_thread_entry函数的地址转换为uintptr_t类型，并将其赋值给tf.epc，即保存在tf的异常程序计数器（EPC）中。

kernel_thread_entry中，`move a0, s1` 指令将寄存器 `s1` 的值移动到寄存器 `a0` 中。这将把线程函数的参数值传递给寄存器 `a0` ，通常用于传递给线程函数的参数。`jalr s0` 指令将寄存器 `s0` 中的地址作为目标地址，执行间接跳转。这里的 `jalr s0` 指令将执行位于寄存器 `s0` 中地址所指向的指令，即线程函数。

`jal do_exit` 指令将 `do_exit` 函数的地址作为目标地址，执行跳转并跳转到 `do_exit` 函数。`do_exit` 函数通常用于退出线程，执行清理和资源释放等操作。

也就是利用了传入的函数指针和参数组成的函数调用。

```
1 // in entry.S
2 kernel_thread_entry:        # void kernel_thread(void)
3         move a0, s1
4         jalr s0
5
6         jal do_exit
```

最后调用do_fork()函数，传递clone_flags | CLONE_VM和0作为参数，以及指向tf结构体的指针&tf。do_fork()函数用于创建一个新的进程或线程。

将tf写入即将被调度的线程

`do_fork()` 函数接受三个参数：`clone_flags` 表示如何克隆子进程，`stack` 表示父进程的用户栈指针，`tf` 是一个指向 `struct trapframe` 结构体的指针，其中包含了中断帧的信息，将会被复制到子进程的 `proc->tf` 中。

在函数中首先检查当前进程数量是否已经达到了最大进程数（`MAX_PROCESS`）。如果是，直接跳转到 `fork_out` 标签处，返回错误码 `-E_NO_FREE_PROC` 。其余有关的初始化操作详见Exercise 2。

有关tf的地方在于将其通过copythread将信息写入到即将被调度的线程中

```
1 /* do_fork -     parent process for a new child process
2  * @clone_flags: used to guide how to clone the child process
3  * @stack:       the parent's user stack pointer. if stack==0, It means to fork
4  * @tf:          the trapframe info, which will be copied to child process's prc
5  */
```

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc,stack,tf);
}

```

调度中修改`content`

在CPU中一直进行循环，如果当前的线程需要调度，那么就会调用schedule函数进行调度。

```
// in proc.c
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

`schedule()`函数开始时定义了一个布尔类型的变量`intr_flag`，用于保存当前的中断状态，并调用`local_intr_save()`函数将中断状态保存起来，屏蔽中断。这是为了在进行进程调度期间防止被中断打断。当发现需要可以调度的线程那么就将调用调用`proc_run`进行调度

```
// in sche.c
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
```

```
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

在 `proc_run` 中我们进行了调度，其中保存了satp（也就是Windows x86的cr3）寄存器，并调用 `switch_to` 函数对 `content` 进行切换。保存正在运行的现场，恢复之前的状态。

```
// in proc.c
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
       bool x;
       local_intr_save(x);
       struct proc_struct* last = current;
       current = proc;
       lcr3(current->cr3);
       switch_to(&(last->context),&(current->context));
       local_intr_restore(x);
    }
}
```

可以看到这里维护着14个与content有关的寄存器，将当前的14个寄存器值保存到以a0为起始地址的一段地址空间上，又将a1起始的14个值读入到寄存器中，因此这一段汇编代码被包装为switch_to(struct proc_struct* from, struct proc_struct* to)，方便被调用。

```
// in switch.S

.text
# void switch_to(struct proc_struct* from, struct proc_struct* to)
.globl switch_to
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0)
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
```

```
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)

    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)
    LOAD s8, 10*REGBYTES(a1)
    LOAD s9, 11*REGBYTES(a1)
    LOAD s10, 12*REGBYTES(a1)
    LOAD s11, 13*REGBYTES(a1)

    ret
```

练习2：为新创建的内核线程分配资源

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，而是通过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。

我们将实现过程以注释形式写在语句块内部：

```c
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 YOUR CODE
    //    1. call alloc_proc to allocate a proc_struct
    if ((proc = alloc_proc()) == NULL) // 调用alloc_proc，获得一块用户信息块，若失败
    {
        goto bad_fork_cleanup_proc;
    }
    //    2. call setup_kstack to allocate a kernel stack for child process
    if ((ret = setup_kstack(proc)) == -E_NO_MEM)
    // 为进程分配一个两页大小的内核栈。若失败跳转至bad_fork_cleanup_kstack
    {
        goto bad_fork_cleanup_kstack;
    }
    //    3. call copy_mm to dup OR share mm according clone_flag
    copy_mm(clone_flags, proc);    // 复制原进程的内存管理信息到新进程（但内核线程不必
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc, stack, tf); // 复制原进程上下文到新进程

    //    5. insert proc_struct into hash_list && proc_list
    proc->pid = get_pid();
    hash_proc(proc);
     // 将新进程添加到进程列表，包括哈希链表和进程链表。由于链表无法实现随机访问，
     // 因此框架中实现了一个哈希链表（list_entry数组），是为了更方便访问找到对应的节点值
    list_add(&proc_list, &(proc->list_link));
    nr_process++;

     //    6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc); // 唤醒新进程并返回新进程号
    //    7. set ret value using child proc's pid
    ret = pid;

fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
```

```
    goto fork_out;
}
```

指导书提出的问题是：**请说明ucore是否做到给每个新fork的线程一个唯一的id?**

回答是肯定的。这需要分析分配id的get_pid函数的逻辑：

```
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;  //使用static int，相当于
    if (++last_pid >= MAX_PID)
    {
        last_pid = 1; //超出范围，重置为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);//顺着链表查看进程
            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;
            }
```

```
38          }
39      }
40      return last_pid;
41  }
```

内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的ucore内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的用户内存空间。从内存空间占用情况这个角度上看，我们可以把线程看作是一种共享内存空间的轻量级进程。

调用get_pid的fork使用的是proc的相关内容，同时max_pid的定义也和max_process息息相关。

函数维护了两个变量last_pid和next_safe。

last_pid是static int类型，会记录上一次分配的pid，但在调用时，last_pid的值巧妙地发生改变，调整成本次可以分配的pid的值并返回。

next_safe维护的是下一个没有重复pid的安全范围。（实际是大于last_pid且值最小的已占用的pid，在一定程度上能减少探测次数优化程序）

如果是从最初开始线程数目较少时的分配，next_safe即是MAX_PID，那么pid会从1开始依次递增加1，然后返回这个pid给相应线程，此时的pid是唯一的，不会重复；然而，当不断有进程分配和释放后，pid总会超出MAX_PID，此时就需要重新扫描是否有可以分配的pid了。

即函数的作用是，找到一个[last_pid,nextsafe)区间是空闲的，并从中分配一个可用的pid，如果此时pid大于等于next_safe就将进行next_safe变量的维护并重新扫描确定新的last_pid。

next_safe变量的维护主要跟当前存在链表中的线程的pid和MAX_PID二者有关。考虑到next_safe变量的含义和pid是连续递增分配的，因此在proc_list中各进程的pid是局部有序的，而while循环内部是在比对每一个想要分配的pid和proc->pid的值是否相同(即if分支)，当发现一个不是相同的pid时，由于pid分配时是连续分配的，则可以说明，我们找到了一个连续的pid段是空闲的（可能由于上一个pid进程的释放），并将next_safe缩小为这个值(即else if分支)。

这一点意味着我们找到了一个安全的区间可供分配，非常重要。而每次last_pid在比对过程中超过next_safe，意味着下一个值可能被已有的线程占用（之前的判断中为某个proc的pid），将会将

next_safe重置为MAX_PID，last_pid重新比对进程已经占用的pid，寻找合适的区间。

算法中同样能保证至少这样取到的区间至少有一个值last_pid可以作为pid返回。

那当所有的PID均被占用当如何是好呢？看起来函数会重复从头开始尝试分配pid直至有线程最终给出空闲的pid供当前需要分配的进程使用，但是在当前的操作系统中，这个pid是唯一的。实际上在我们的实验系统中，MAX_PID 被定义为MAX_PROCESS的两倍，因此保证可以取得一个可以使用的pid。

至此，可以说明get_pid是为每个进程分配的唯一的pid值。

练习3：编写proc_run 函数

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：

1. 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。
2. 禁用中断。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。
3. 切换当前进程为要运行的进程。
4. 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。
5. 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。
6. 允许中断。

```
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 i
        * 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;
       local_intr_save(intr);  //关中断
```

```
15          struct proc_struct *tmp;
16          tmp = current;
17          current = proc;       //切换
18          lcr3(current->cr3);  //切换页表，使用新进程的地址空间
19          switch_to(&(tmp->context),&(current->context)); //上下文切换
20
21          local_intr_restore(intr);   //开中断
22
23      }
24  }
```

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？

一个IdleProcess

一个initProcess（int_main线程）

1. `idleproc` 是一个在操作系统中常见的概念，用于表示空闲进程。在操作系统中，空闲进程是一个特殊的进程，它的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。

   在init.c::kern_init函数调用了proc.c::proc_init函数。proc_init函数启动了创建内核线程的步骤：

   首先初始化进程链表，随后，调用alloc_proc函数来通过kmalloc函数获得proc_struct结构的一块内存块-，作为第0个进程控制块。并把proc进行初步初始化,把proc_struct中的各个成员变量清零,某一些设置特殊的值（详见Exercise 1）；接下来，proc_init函数对idleproc内核线程进行进一步初始化，设置了它的页表，栈，pid等；

   此后，Idle线程就通过执行cpu_idle函数开始过退休生活；

2. uCore还需创建其他进程来完成各种工作，idleproc内核子线程通过调用kernel_thread函数创建了一个内核线程init_main，用于打印一些字符串。kernel_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do_fork函数，而do_fork函数会调用copy_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。在这个过程中，do_fork函数有很重要的作用，包括分配并初始化进程控制块、内核栈、pid等等（详见Exercise 2）

3. 创建好了两个内核线程：idleproc和initproc后，此时的uCore当前执行的是idleproc，等到执行到init函数的最后一个函数cpu_idle之前，uCore的所有初始化工作就结束了，idleproc将通过执行cpu_idle函数让出CPU，给init_main线程使用。

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?

一些语句需要进行开关中断，这和事务的原子性十分类似，在程序编写者认为需要保证执行这段代码过程中不被中断打断时使用。

与这部分相关的代码如下，在sync/sync.h和intr.c中

```
// in /sync/sync.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);

// in /driver/intr.c
/* intr_enable - enable irq interrupt */
void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }

/* intr_disable - disable irq interrupt */
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

此处语句宏调用的这两个函数，local_intr_save是保存sstatus寄存器中的中断使能位(SIE)信息并禁用中断，而local_intr_restore是根据保存的中断使能位信息来使能中断。

二者分别调用了intr.c中的 intr_enable和intr_disable对sstatus这一csr寄存器的SIE寄存器进行修改：由于RISC-V中sstatus 寄存器的 SIE 位用于控制全局中断使能，当 SIE 位被设置时，允许中断发生，反之当 SIE 位被清除时，中断将被禁止，不再响应中断请求。

在探究中，我们还发现它们在lab中一般成对使用，在函数的具体逻辑里，restore宏接收save宏的结果，并以此为依据判断是否之前禁用过中断并重新启用中断，只有save宏禁用中断并返回1后，restore宏才能重新启用，这也和禁用-恢复的逻辑是吻合的。