操作系统实验报告

实验名称：进程管理 实验地点：图书馆323

组号：56 小组成员：周钰宸 王志远 齐明杰

一、实验目的

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

二、实验过程

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/ffags/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）

回答：

1.1 问题1

alloc_proc函数主要是分配并且初始化一个PCB用于管理新进程的信息。proc_struct 结构的信息如下：

```
struct proc_struct {//进程控制块
    enum proc_state state;                      // 进程状态
    int pid;                                    // 进程ID
    int runs;                                   // 运行时间
    uintptr_t kstack;                           // 内核栈位置
    volatile bool need_resched;                 // 是否需要调度
    struct proc_struct *parent;                 // 父进程
    struct mm_struct *mm;                       // 进程的虚拟内存
    struct context context;                     // 进程上下文
    struct trapframe *tf;                       // 当前中断帧的指针
    uintptr_t cr3;                              // 当前页表地址
    uint32_t flags;                             // 进程
    char name[PROC_NAME_LEN + 1];               // 进程名字
    list_entry_t list_link;                     // 进程链表
    list_entry_t hash_link;
};

```

在alloc_proc中我们对每个变量都进行初始化操作，代码如下：

```
static struct proc_struct *
alloc_proc(void) {

```

```
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;//给进程设置为未初始化状态
        proc->pid = -1;//未初始化的进程，其pid为-1
        proc->runs = 0;//初始化时间片,刚刚初始化的进程，运行时间一定为零
        proc->kstack = 0;//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。
        proc->need_resched = 0;//不需要调度
        proc->parent = NULL;//父进程为空
        proc->mm = NULL;//虚拟内存为空
        memset(&(proc->context), 0, sizeof(struct context));//初始化上下文
        proc->tf = NULL;//中断帧指针为空
        proc->cr3 = boot_cr3;//页目录为内核页目录表的基址
        proc->flags = 0; //标志位为0
        memset(proc->name, 0, PROC_NAME_LEN);//进程名为0
    }
    return proc;
}

```

- state设置为未初始化状态；
- 由于刚创建进程，pid设置为-1；
- 进程运行时间run初始化为0；
- 内核栈地址kstack默认从0开始；
- 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。初始化为0，表示不需要调度；
- 父进程parent设置为空；
- 内存空间初始化为空；
- 上下文结构体context初始化为0；
- 中断帧指针tf设置为空；
- 页目录cr3设置为为内核页目录表的基址boot_cr3；
- 标志位flags设置为0；
- 进程名name初始化为0；

通过如上代码，完成了对分配得到的新进程的PCB的初始化操作。

1.2 问题2

①**context**作用：进程的上下文，用于进程切换。主要保存了前一个进程的现场（各个寄存器的状态）。在uCore中，所有的进程在内核中也是相对独立的。使用context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是在kern/process/switch.S中定义switch_to。

②**tf**：中断帧的指针，总是指向内核栈的某个位置：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。除此之外，uCore内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的trapframe，uCore 在内核栈上维护了 tf 的链。

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。*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。请在实验报告中简要说明你的设计实现过程。
- 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。

回答：

2.1 问题1

根据文档提示，do_fork函数的处理大致可以分为7步，下面我们来按步骤实现该函数：

1. 调用alloc_proc

```
    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()函数申请内存块，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()函数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;
    }
```

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

调用copy_mm()函数，复制父进程的内存信息到子进程。对于这个函数可以看到，进程proc复制还是共享当前进程current，是根据clone_flags来决定的，如果是clone_flags & CLONE_VM（为真），那么就可以拷贝。

本实验中，仅仅是确定了一下当前进程的虚拟内存为空，并没有做其他事。

4. 复制原进程上下文到新进程

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

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

调用copy_thread()函数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
```

```
hash_proc(struct proc_struct *proc) {
    list_add(hash_list + pid_hashfn(proc->pid), &(proc->hash_link));
}
```

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，并把总进程数+1。在添加到进程链表的过程中，我们使用了local_intr_save()和local_intr_restore()函数来屏蔽与打开，保证添加进程操作不会被抢断。

6. 唤醒新进程

```
wakeup_proc(proc);
```

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

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

7. 返回新进程号

```
ret = proc->pid;//返回当前进程的PID
```

返回新进程号。

通过如下7个步骤，我们可以完整的实现do_fork函数创建新进程的功能。

2.2 问题2

我们可以查看实验中获取进程id的函数：**get_pid(void)**

```
// get_pid - alloc a unique pid for process
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;
}
```

这段代码通过维护一个静态变量`last_pid`来实现为每个新fork的线程分配一个唯一的id。让我们逐步分析：

1. `last_pid`是一个静态变量，它会记录上一个分配的pid。
2. 当`get_pid`函数被调用时，首先检查是否`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。
3. 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。
4. 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。

这样，通过这个机制，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 用于将指定的进程切换到 *CPU* 上运行。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？

回答：

3.1 问题1

根据文档的提示说明，我们编写的proc_run()函数如下：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // LAB4:EXERCISE3
        /*
        * 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);
    }
}
```

此函数**基本思路**是：

- 让 current指向 next内核线程initproc；
- 设置 CR3 寄存器的值为 next 内核线程 initproc 的页目录表起始地址 next->cr3，这实际上是完成进程间的页表切换；
- 由 switch_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当 switch_to 函数执行完“ret”指令后，就切换到initproc执行了。

值得注意的是，这里我们使用`local_intr_save()`和`local_intr_restore()`，作用分别是屏蔽中断和打开中断，以免进程切换时其他进程再进行调度，保护进程切换不会被中断。

3.2 问题2

在本实验中，创建且运行了2两个内核线程：

- 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。

4.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：

这两句分别是kern/sync.h中定义的中断前后使能信号保存和退出的函数。

```
#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__ */
```

这两个函数在当时Lab1中有所涉及，主要作用是首先通过定义两个宏函数local_intr_save和local_intr_restore。这两个宏函数会调用两个内联函数`intr_save`和`intr_restore`。

1. `do{...}while(0)`结构

   我们看到local_intr_save的宏定义部分使用了一个`do{...}while(0)`的结构，作用是确保宏在语法上表现得像是单个语句。这样做的目的是为了在代码中使用时，保证宏在语法结构上的一致性，不引起语法错误。我们考虑C++和C语言的编译机制：宏定义是一种文本替换机制，**预处理过程**它将代码中的宏名称替换为预定义的代码片段。如果没有`do{...}while(0)`结构，那么编译器在预处理过程将它展开时可能就会导致语法错误，举一个如下的例子：

   ```
   #define DOSOMETHING()   func1(); func2()

   展开前：
   if(judge == TRUE)
       DOSOMETHING;

   展开后：
   if(judge == TRUE)
       func1();
       func2();
   ```

由于宏展开后没有花括号，所以func1()和func2()就直接被简单的替换到了代码之后，但这时显然出现了**逻辑错误**：无论判断条件是否成立，func2()都会被执行。如果这个if还有else等分支的话，会出现**编译错误**，else匹配不到指定的if。

而如果只考虑加把宏定义中的语句加一对{}来作为整体，那么会出现`;`冗余的问题：

```
#define DOSOMETHING()   { func1(); func2();}

展开前：
if(judge == TRUE)
    DOSOMETHING;

展开后
if(judge == TRUE)
{
    func1();
    func2();
};
```

if最后的一个`}`后的`;`显然是多余的。所以我们最终采用`do{...}while(0)`结构来解决这个问题：

```
#define DOSOMETHING()   do{ func1(); func2();} while(0)

展开前：
if(judge == TRUE)
    DOSOMETHING;

展开后：
if(judge == TRUE)
    do{
        func1();
        func2();
    }while(0);
```

2. `intr_save`和`intr_restore`的具体实现

这两个内联函数分别调用了`intr_disable()`和`intr_enable()`函数来实现中断的禁止与启用，我们查看这两个函数：

```
/* 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); }
```

可以看到，这两个函数又分别调用了`set_csr()`和`clear_csr()`函数来设置状态寄存器的**SIE**标志位，以标识是否允许中断。我们找到这两个宏定义的函数：

```
#define set_csr(reg, bit) ({ unsigned long __tmp; \
  asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })

#define clear_csr(reg, bit) ({ unsigned long __tmp; \
  asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "rK"(bit)); \
  __tmp; })
```

这两个宏定义是用于在 RISC-V 架构的汇编语言中设置和清除控制状态寄存器（CSR）中特定位的值的。我们以`set_csr`宏为例来解析说明。

> 在这里，`%0` 和 `%1` 是内联汇编代码中的操作数占位符，用于表示输出和输入操作数。这些占位符会与约束（constraints）一起使用，将操作数与C代码中的变量进行关联。
>
> 在这两个宏中：
>
> - `%0` 代表输出操作数，即存储结果的变量 `__tmp`。
> - `%1` 代表输入操作数，即传递给内联汇编代码的变量 `bit`。
>
> 这些占位符允许内联汇编代码与C代码进行交互，将C代码中的变量与汇编代码中的寄存器或内存位置相对应。在这种情况下，`%0` 和 `%1` 分别与 `__tmp` 和 `bit` 相关联。

这个宏使用**内联汇编**在执行期间设置控制状态寄存器 `reg` 中的特定位 `bit`。具体步骤如下：

- 声明一个无符号长整型变量 `__tmp`，用于存储操作结果。
- 使用 `asm volatile` 关键字引入**内联汇编**代码。
- 在汇编代码中，使用 `csrrs` 指令（CSR Read and Set）来读取寄存器的当前值，并将特定位设置为1。
- 使用占位符`0%`和**约束** `"=r"(__tmp)` 将 `__tmp` 变量与**输出寄存器**相关联。
- 使用占位符1%和**约束** `"rK"(bit)` 将 `bit` 变量与**输入寄存器**相关联。
- 最后，返回设置后的寄存器值。

其中用到的`csrrs` 指令实际上是一条汇编指令，用于读取控制状态寄存器（CSR）的当前值并将指定的位设置为1。这个指令的格式如下：

```
csrrs rd, csr, rs1
```

- `rd`: 目标寄存器，用于存储读取到的CSR的当前值。本例中就是变量**__temp**。
- `csr`: 控制状态寄存器的标识符，表示要读取的寄存器。本例中就是状态寄存器**sstatus**。
- `rs1`: 一个通用寄存器，其值用于设置CSR的特定位。本例中就是**SSTATUS_SIE**。

所以这句代码就实现了把寄存器sstatus的SSTATUS_SIE位在读取后置为1的功能。

同理，clear_csr()函数中的**csrrc**指令就实现了把SSTATUS_SIE设置为0的功能。由此我们知道，程序最终是通过**汇编指令**来改变状态寄存器的SSTATUS_SIE位来禁止或者启用中断。

最后总结一下， **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。**

三、与参考答案的对比

由于本实验的代码逻辑较为固定，因此我们小组所完成的代码与参考答案差别不大。

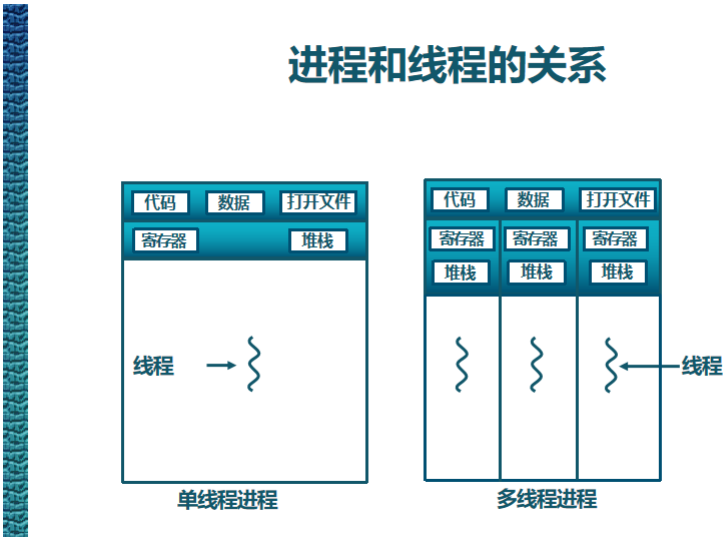
四、实验中的知识点

4.1 进程与线程的关系

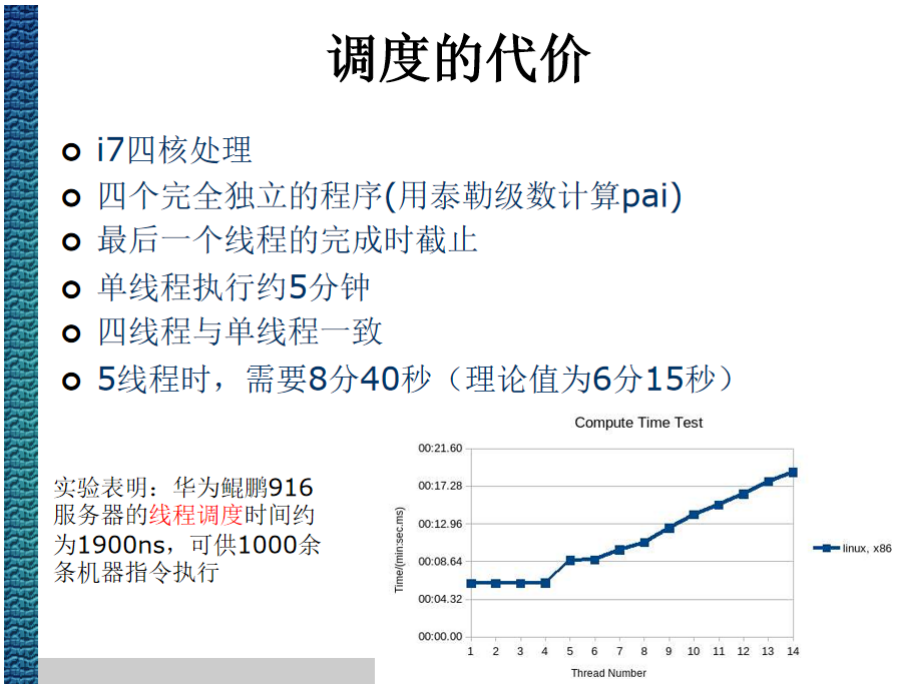
我们平时编写的源代码，经过编译器编译就变成了可执行文件，我们管这一类文件叫做**程序** 。而当一 个程序被用户或操作系统启动，分配资源，装载进内存开始执行后，它就成为了一个 **进程** 。

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。



4.2 进程调度



上OS课时候宫老师提到过，调度的代价是很大的，其中一般涉及：

- 减少上下文切换涉及的寄存器数量
- 减少不必要的权限切换

一些理论上可以处理的方式包括：

- 纤程 Fiber, ucontext
- 协程 coroutine
- 发挥ULT快速切换的优势
- 在编程时提出对程序员的限制，要求他们妥善的设计代码