

# 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/fflags/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：

```
1  struct proc_struct { //进程控制块
2      enum proc_state state;           // 进程状态
3      int pid;                         // 进程ID
4      int runs;                        // 运行时间
5      uintptr_t kstack;                // 内核栈位置
6      volatile bool need_resched;      // 是否需要调度
7      struct proc_struct *parent;      // 父进程
8      struct mm_struct *mm;            // 进程的虚拟内存
9      struct context context;          // 进程上下文
10     struct trapframe *tf;             // 当前中断帧的指针
11     uintptr_t cr3;                    // 当前页表地址
12     uint32_t fflags;                  // 进程
13     char name[PROC_NAME_LEN + 1];    // 进程名字
14     list_entry_t list_link;           // 进程链表
15     list_entry_t hash_link;
16 };
17
```

在`alloc_proc`中我们对每个变量都进行初始化操作，代码如下：

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

```

3     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
4     if (proc != NULL) {
5         proc->state = PROC_UNINIT; //给进程设置为未初始化状态
6         proc->pid = -1; //未初始化的进程, 其pid为-1
7         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
8         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。
9         proc->need_resched = 0; //不需要调度
10        proc->parent = NULL; //父进程为空
11        proc->mm = NULL; //虚拟内存为空
12        memset(&(proc->context), 0, sizeof(struct context)); //初始化上下文
13        proc->tf = NULL; //中断帧指针为空
14        proc->cr3 = boot_cr3; //页目录为内核页目录表的基址
15        proc->flags = 0; //标志位为0
16        memset(proc->name, 0, PROC_NAME_LEN); //进程名为0
17    }
18    return proc;
19 }
20

```

- 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`

```
1 | if ((proc = alloc_proc()) == NULL) {
2 |     goto fork_out;
3 | }
4 | 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

```
1 | if (setup_kstack(proc)) {
2 |     goto bad_fork_cleanup_proc;
3 | }
```

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)

```
1 | if(copy_mm(clone_flags, proc)){
2 |     goto bad_fork_cleanup_kstack;
3 | }
```

```
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 | }
```

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

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

```

1 static void
2 copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe
  *tf) {
3     proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE -
    sizeof(struct trapframe));
4     *(proc->tf) = *tf;
5
6     // Set a0 to 0 so a child process knows it's just forked
7     proc->tf->gpr.a0 = 0;
8     proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;
9
10    proc->context.ra = (uintptr_t)forkret;
11    proc->context.sp = (uintptr_t)(proc->tf);
12 }

```

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

```

1 bool intr_flag;
2 local_intr_save(intr_flag); //屏蔽中断, intr_flag置为1
3 {
4     proc->pid = get_pid(); //获取当前进程PID
5     hash_proc(proc); //建立hash映射
6     list_add(&proc_list, &(proc->list_link)); //加入进程链表
7     nr_process ++; //进程数加一
8 }
9 local_intr_restore(intr_flag); //恢复中断

```

```

1 hash_proc(struct proc_struct *proc) {
2     list_add(hash_list + pid_hashfn(proc->pid), &(proc->hash_link));
3 }

```

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

```

1 wakeup_proc(proc);

```

```

1 void
2 wakeup_proc(struct proc_struct *proc) {
3     assert(proc->state != PROC_ZOMBIE && proc->state != PROC_RUNNABLE);
4     proc->state = PROC_RUNNABLE;
5 }

```

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

```

1 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)`

```
1 // get_pid - alloc a unique pid for process
2 static int
3 get_pid(void) {
4     static_assert(MAX_PID > MAX_PROCESS);
5     struct proc_struct *proc;
6     list_entry_t *list = &proc_list, *le;
7     static int next_safe = MAX_PID, last_pid = MAX_PID;
8     if (++ last_pid >= MAX_PID) {
9         last_pid = 1;
10        goto inside;
11    }
12    if (last_pid >= next_safe) {
13    inside:
14        next_safe = MAX_PID;
15    repeat:
16        le = list;
17        while ((le = list_next(le)) != list) {
18            proc = le2proc(le, list_link);
19            if (proc->pid == last_pid) {
20                if (++ last_pid >= next_safe) {
21                    if (last_pid >= MAX_PID) {
22                        last_pid = 1;
23                    }
24                    next_safe = MAX_PID;
25                    goto repeat;
26                }
27            }
28            else if (proc->pid > last_pid && next_safe > proc->pid) {
29                next_safe = proc->pid;
30            }
31        }
32    }
33    return last_pid;
34 }
```

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：

```
1 void
2 proc_run(struct proc_struct *proc) {
3     if (proc != current) {
4         // LAB4:EXERCISE3
5         /*
6          * Some Useful MACROS, Functions and DEFINES, you can use them in
7          below implementation.
8          *
9          * MACROS or Functions:
10         *   local_intr_save():      Disable interrupts
11         *   local_intr_restore():   Enable Interrupts
12         *   lcr3():                  Modify the value of CR3 register
13         *   switch_to():             Context switching between two
14         processes
15         */
16         bool intr_flag;
17         struct proc_struct *prev = current, *next = proc;
18         local_intr_save(intr_flag);
19         {
20             current = proc;
21             lcr3(next->cr3);
22             switch_to(&(prev->context), &(next->context));
23         }
24         local_intr_restore(intr_flag);
25     }
26 }
```

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。

```
1  #include <defs.h>
2  #include <intr.h>
3  #include <riscv.h>
4
5  static inline bool __intr_save(void) {
6      if (read_csr(sstatus) & SSTATUS_SIE) {
7          intr_disable();
8          return 1;
9      }
10     return 0;
11 }
12
13 static inline void __intr_restore(bool flag) {
14     if (flag) {
15         intr_enable();
16     }
17 }
18
19 #define local_intr_save(x) \
20     do {                    \
21         x = __intr_save(); \
22     } while (0)
23 #define local_intr_restore(x) __intr_restore(x);
24
25 #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`。其中：

#### 1. `intr_save`和`local_intr_save`:

- **`intr_save`**: 通过读取CSR控制和状态寄存器的 `sstatus` 中的值，并对比其是否被设置为了 `SIE=1`，即中断使能位=1。如果中断使能位 `SIE` 是1，那么表示中断是被允许的；为0就是不允许的。因此如果中断本来是允许的，就会调用 `intr.h` 中的 `intr_disable` 禁用中断，否则直接返回，因此本身也不允许。
- **`local_intr_save`**: 在这个函数其中，用 `do-while` 循环可以确保 `x` 变量在 `__intr_save()` 函数调用之后被正确赋值，无论中断是否被禁用。

#### 2. `intr_restore`和`local_intr_restore`:

- **`intr_restore`**: 直接根据 `flag` 标志位是否为0，`intr_enable()` 重新启用中断。
- **`local_intr_restore`**: 与 `save` 不同，无需返回任何值，恢复中断即可。

由此将两个宏定义函数结合起来，`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。如果不这样的就会导致宫老师讲的我和舍友买面包贴标签的尴尬问题。

在 `Kernel_Thread` 函数中，就设置 `SSTATUS` 的 `SPIE`, `SIE` 等标志位实现控制中断开关。

除此之外，我们在练习3中也通过这个方式控制中断前后的切换。

```
1 local_intr_save(intr_flag);
2     {
3         current = proc;
4         lcr3(next->cr3);
5         switch_to(&(prev->context), &(next->context));
6     }
7 local_intr_restore(intr_flag);
```

### 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。

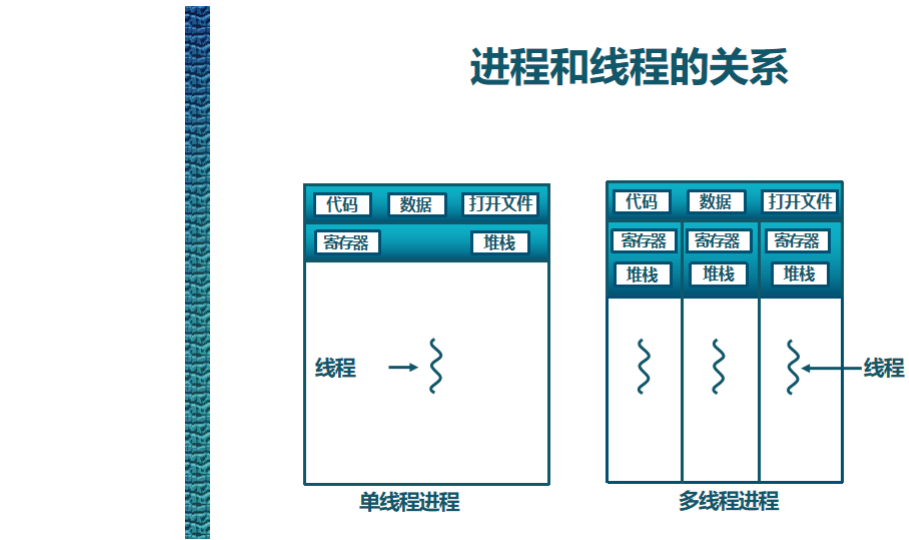
### 四、实验中的知识点

#### 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。



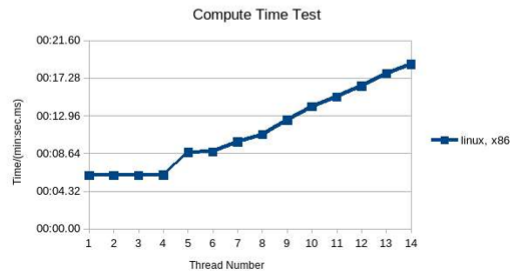
#### 4.2 进程调度



# 调度的代价

- i7四核处理
- 四个完全独立的程序(用泰勒级数计算pai)
- 最后一个线程的完成时截止
- 单线程执行约5分钟
- 四线程与单线程一致
- 5线程时，需要8分40秒（理论值为6分15秒）

实验表明：华为鲲鹏916服务器的线程调度时间约为1900ns，可供1000余条机器指令执行



上OS课时候官老师提到过，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