

# Operating System Lab4

查科言 (2312189) 禹相祐 (2312900) 董丰瑞 (2311973)

## Operating System Lab4

### 一、练习一：配并初始化一个进程控制块

(1) 问题：

(2) 设计实现

(3) 问题回答

1. `struct context context`: 进程上下文 (内核态切换用)

2. `struct trapframe *tf`: 中断帧 (异常 / 中断处理用)

3. 对比

### 二、练习二：为新创建的内核线程分配资源

(1) 问题：

(2) 设计实现

(3) 问题回答

### 三、练习三：编写proc\_run 函数

(1) 问题：

(2) 设计实现

(3) 问题回答

### 四、实验结果

### 五、扩展练习 Challenge:

(1) 问题一：

(2) 问题二：

1. 代码相似原因

2. `get_pte()` 中查找与创建功能的合并是否合理

### 六、实验知识点对应

(1) 实验与OS原理知识点对应

(2) OS原理重要但是没有覆盖的知识点

## 一、练习一：配并初始化一个进程控制块

### (1) 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。

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)

### (2) 设计实现

`alloc_proc` 函数的核心功能是分配动态分配一个进程控制块 (`struct proc_struct`) 并初始化其所有成员变量，为进程的创建和管理奠定基础。下面结合 `proc_struct` 的结构详细解释其实现逻辑：

#### 1. `proc_struct` 结构回顾

进程控制块（PCB）是操作系统管理进程的核心数据结构，包含进程的所有关键信息。其定义如下（来自 `proc.h`），我们要做的就是将这些变量进行初始化。

```
1 struct proc_struct {
2     enum proc_state state;           // 进程状态
3     int pid;                         // 进程ID
4     int runs;                        // 进程运行次数
5     uintptr_t kstack;               // 进程内核栈地址
6     volatile bool need_resched;     // 是否需要重新调度
7     struct proc_struct *parent;      // 父进程指针
8     struct mm_struct *mm;            // 内存管理结构
9     struct context context;          // 进程上下文（用于切换）
10    struct trapframe *tf;            // 中断帧（保存中断时的现场）
11    uintptr_t pgdir;                // 页目录表基地址
12    uint32_t flags;                 // 进程标志位
13    char name[PROC_NAME_LEN + 1];   // 进程名称
14    list_entry_t list_link;          // 进程链表节点（用于全局进程链表）
15    list_entry_t hash_link;          // 进程哈希表节点（用于按PID快速查找）
16};
```

## 2. alloc\_proc函数实现

```
1 static struct proc_struct *
2 alloc_proc(void)
3 {
4     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
5     if (proc != NULL)
6     {
7         proc->state = PROC_UNINIT; // 初始化为未初始化状态
8         proc->pid = -1;           // 临时无效PID
9         proc->runs = 0;            // 运行次数初始化为0
10        proc->kstack = 0;          // 内核栈地址初始化为0
11        proc->need_resched = 0; // 不需要重新调度
12        proc->parent = NULL;       // 无父进程
13        proc->mm = NULL;           // 内存管理结构为空（内核线程）
14        memset(&proc->context, 0, sizeof(struct context)); // 上下文清零
15        proc->tf = NULL;             // 陷阱帧初始化为空
16        proc->pgdir = 0;              // 页目录初始化为0
17        proc->flags = 0;                // 标志位清零
18        memset(proc->name, 0, PROC_NAME_LEN + 1); // 进程名清零
19        list_init(&proc->list_link); // 初始化进程链表项
20        list_init(&proc->hash_link); // 初始化哈希链表项
21
22    }
23    return proc;
24}
```

### 步骤一：

- 首先通过 `kmalloc(sizeof(struct proc_struct))` 动态分配一块内存用于存储 `proc_struct` 结构。
- 如果分配失败 (`proc == NULL`)，直接返回 `NULL`；否则进入初始化流程。

### 步骤二：初始化成员变量

根据指导书中的内容，每个成员的初始化逻辑如下：

1. `state`: **进程状态**, 初始化为 `PROC_UNINIT` (未初始化状态)。
  - 进程创建时需经历 “未初始化→可运行→运行 / 睡眠 / 僵尸” 的状态转换, 此处为初始状态。
2. `pid`: **进程 ID**, 初始化为 `-1` (无效值)。
  - `PID` 是进程的唯一标识, 需通过 `get_pid()` 函数分配唯一有效值, 此处暂用 `-1` 标记未分配。
3. `runs`: **运行次数**, 初始化为 `0`。
  - 记录进程被调度运行的总次数, 用于调度算法 (如优先级调整)。
4. `kstack`: **内核栈地址**, 初始化为 `0` (无效地址)。
  - 进程的内核栈用于执行内核代码, 需在后续通过 `setup_stack` 等函数分配并设置, 此处我们设置暂为无效值。
5. `need_resched`: **是否需要重新调度**,
  - 布尔类型, 为 `1` 则需要调度, 在进行初始化时我们无需对其进行调度, 设置为 `0`
6. `parent`: **父进程指针**, 初始化为 `NULL`。
  - 进程创建时通常会指定父进程 (如通过 `fork`) , 此处暂为 `NULL`, 后续由创建者设置。
7. `mm`: **内存管理结构**, 初始化为 `NULL`。
  - `mm_struct` 用于管理进程的用户地址空间 (如页表、内存映射)。内核线程无需用户空间, 因此默认 `NULL`; 用户进程会在后续初始化时分配。
8. `context`: **进程上下文**, 通过 `memset` 清零。
  - `context` 存储进程切换时需要保存的寄存器状态 (如 `ra`、`sp`、`s0-s11`) , 初始化为全 `0`, 后续由 `switch_to` 等函数设置。
9. `tf`: **中断帧指针**, 初始化为 `NULL`。
  - `trapframe` 用于保存进程在用户态被中断时的现场 (如通用寄存器、PC 值) , 内核线程默认无需此结构, 用户进程会在中断时设置。
10. `pgdir`: **页目录表基地址**, 初始化为 `0` (无效地址)。
  - 页目录表是虚拟内存的核心结构, 用户进程会在后续通过 `setup_pgdir` 分配, 内核线程共享内核页表, 因此暂为 `0`。
11. `flags`: **进程标志位**, 初始化为 `0`。
  - 用于标记进程的特殊属性 (如是否为内核线程、是否正在退出等) , 后续根据进程类型设置。
12. `name`: **进程名称**, 通过 `memset` 清零。
  - 进程名称用于调试和显示, 长度限制为 `PROC_NAME_LEN` (15 字节) , 后续通过 `set_proc_name` 设置。
13. `list_link` 和 `hash_link`: **链表节点**, 通过 `list_init` 初始化。
  - `list_link` 用于将进程加入全局进程链表 (`proc_list`) , 便于遍历所有进程。
  - `hash_link` 用于将进程加入哈希表 (`hash_list`) , 便于通过 PID 快速查找进程。

**我们至此实现了进程控制块的初始化!**

### 3. 总结

`alloc_proc` 函数的作用是“初始化一个空白的进程控制块”，它为进程的所有成员变量设置合理的初始值，但不会分配实际资源（如 PID、内核栈、内存空间等）。这些资源的分配和设置我们的代码会在后续步骤中完成（如 `proc_init` 初始化进程、`do_fork` 创建子进程等）。

### (3) 问题回答

在实验代码中，`struct proc_struct` 中的 `context` 和 `tf` 成员变量均与进程的执行状态保存和恢复相关。

#### 1. `struct context context`: 进程上下文 (内核态切换用)

含义：

`struct context` 定义了进程在内核态下切换时需要保存的寄存器集合，具体包含：

```
1 struct context
2 {
3     uintptr_t ra; // 保存函数调用后的返回地址
4     uintptr_t sp; // 栈指针 (内核栈顶位置)
5     uintptr_t s0-s11; // s0~s11 保存寄存器 (被调用者保存的寄存器)
6 }
```

这些寄存器是 RISC-V 架构中函数调用约定中需要**被调用者保存**的寄存器，用于保证函数调用后上下文的完整性。

作用：

- **进程切换的核心载体**：当操作系统进行进程调度（如 `switch_to` 函数）时，需要保存当前进程的 `context` 到其 `proc_struct` 中，再加载目标进程的 `context` 以恢复执行。
- **内核线程的初始执行状态**：对于内核线程（如通过 `kernel_thread` 创建的进程），`context` 会被初始化为指向线程入口函数（如 `forkret`），确保切换后从正确位置开始执行。

关键代码关联

- 在 `proc_run` 中，通过 `switch_to(&(prev->context), &(proc->context))` 完成上下文切换。
- 在 `alloc_proc` 中，`memset(&proc->context, 0, sizeof(struct context));`，被初始化为全 0，后续由进程创建逻辑（如 `copy_thread`）设置具体值。

#### 2. `struct trapframe *tf`: 中断帧 (异常 / 中断处理用)

含义：

`struct trapframe` 定义了进程在用户态或内核态发生异常 / 中断时，需要保存的完整寄存器状态，我们在 Lab3 中使用到了，包含：

```
1 struct trapframe {
2     struct pushregs gpr; // 所有通用寄存器 (包括零寄存器、临时寄存器、参数寄存器等)
3     uintptr_t status; // 状态寄存器 (记录中断前的处理器状态)
4     uintptr_t epc; // 异常程序计数器 (中断发生时的指令地址)
5     uintptr_t badvaddr; // 错误地址 (如页故障的访问地址)
6     uintptr_t cause; // 异常原因 (标识中断/异常的类型)
7 }
```

相较于 `context`, `tf` 保存的寄存器更完整, 涵盖了所有可能影响程序执行的状态。

## 作用

- 异常 / 中断现场保护**: 当发生中断或异常 (如时钟中断、系统调用、页故障) 时, 硬件或内核会自动将当前寄存器状态保存到 `tf` 中, 确保中断处理完成后能精确恢复原执行流程。
- 用户态与内核态交互的桥梁**: 对于用户进程, `tf` 记录了用户态下的寄存器状态 (如系统调用前的参数), 内核处理完系统调用后, 通过恢复 `tf` 让进程回到用户态继续执行。
- 进程初始执行状态设置**: 在创建新进程 (如 `kernel_thread`) 时, 会初始化 `tf` 以指定进程的初始入口地址 (`epc`) 和状态 (`status`)。

## 关键代码关联

- 在 `trap` 函数中, `tf` 作为参数传递, 用于处理中断 / 异常 (如 `interrupt_handler` 和 `exception_handler`)。
- 在 `forkrets` 函数中, 通过恢复 `tf` 中的寄存器状态, 让新进程从指定入口点开始执行。
- 在 `kernel_thread` 中, 手动构造 `tf` 以设置内核线程的初始执行环境 (如入口函数 `kernel_thread_entry`)。

## 3. 对比

结构体	<code>struct context context</code>	<code>struct trapframe *tf</code>
使用场景	进程调度时的内核态上下文切换	异常 / 中断发生时的现场保护与恢复
保存内容	最小化的被调用者保存寄存器	完整的通用寄存器 + 状态 / 地址 / 原因寄存器
关联操作	与 <code>switch_to</code> 配合完成进程切换	与 <code>trap</code> 处理函数配合完成中断响应
生命周期	仅在进程切换时更新	在每次中断 / 异常发生时更新

在 `struct context` 中, 仅保存了部分寄存器, 而不是完整的寄存器集合, 这是因为, RISC-V架构将寄存器分为两类, 遵循函数调用约定:

- 调用者保存寄存器** (如 `t0-t6`、`a0-a7` 等) : 由函数调用者负责保存和恢复, 用于传递参数或临时变量, 函数调用后无需保留其值。
- 被调用者保存寄存器** (如 `s0-s11`、`sp`、`ra`) : 由被调用函数负责保存和恢复, 用于维持函数执行的上下文连续性, 其值在函数调用后必须保留。

`struct context` 中保存的恰好是**被调用者保存寄存器**, `context` 设计的目的是实现进程在内核态下的切换, 而进程切换的本质是内核中“调度函数”对“目标进程”的函数调用。根据调用约定:

- 调用者 (调度器) 会**自行保存**其使用的**调用者保存寄存器**, 无需目标进程的 `context` 关心。
- 被调用者 (目标进程) 只需确保其**被调用者保存寄存器**的值正确, 即可从切换前的状态继续执行。

因此, `context` 只需保存被调用者保存寄存器, 即可满足进程切换时的上下文完整性需求, 无需保存全部寄存器。

## 二、练习二：为新创建的内核线程分配资源

### (1) 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。

### (2) 设计实现

`do_fork` 函数是实现进程创建的核心函数，其作用是通过复制父进程（`current`）的资源（如内存空间、执行状态等）创建一个新进程，并完成新进程的初始化。

我们的完整实现如下：

#### 实现步骤

##### 1. 检查创建条件

```
1 | if (nr_process >= MAX_PROCESS) {  
2 |     ret = -E_NO_FREE_PROC;  
3 |     goto fork_out;  
4 | }
```

- 首先检查系统进程数量是否已达上限（`MAX_PROCESS`），若已达上限则返回错误。

##### 2. 分配并初始化进程控制块（PCB）

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

- 我们首先调用 `alloc_proc` 分配 `struct proc_struct` 结构体 (PCB) , 并初始化其成员 (如状态为 `PROC_UNINIT`、`PID` 临时设为 `-1` 等) 。
- 若分配失败 (内存不足) , 返回内存错误。

### 3. 为子进程分配内核栈

```

1 | if (setup_kstack(proc) != 0) {
2 |     goto bad_fork_cleanup_proc;
3 |

```

- 调用 `setup_kstack` 为子进程分配内核栈 (大小为 `KSTACKPAGE` 页) , 并将栈顶地址记录在 `proc->kstack` 中。
- 若分配失败, 释放已分配的 PCB 并返回错误。

### 4. 复制内存管理结构 (MM)

```

1 | if (copy_mm(clone_flags, proc) != 0) {
2 |     goto bad_fork_cleanup_kstack;
3 |

```

- 根据 `clone_flags` 决定内存空间的处理方式:
  - 若 `clone_flags & CLONE_VM` (线程创建) : 子进程与父进程共享内存空间 (`mm_struct` 相同) 。
  - 否则 (普通进程创建) : 复制父进程的内存空间 (包括页表、虚拟内存区域等) 。
- 若复制失败, 释放内核栈和 PCB 并返回错误。

### 5. 复制上下文和陷阱帧

```

1 | 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。

- 初始化子进程的执行状态, 关键是设置内核入口点和栈:
  - 对于内核线程: `tf` 是手动构造的陷阱帧, 包含入口函数和参数。
  - 对于用户进程: 复制父进程的 `tf` (陷阱帧), 确保子进程从父进程的中断点继续执行。
- 设置 `proc->context` (进程上下文) , 使得调度时能正确切换到子进程的内核栈和执行位置 (如 `forkret` 函数) 。

### 6. 设置进程标识与关系

```

1 | proc->pid = get_pid();           // 分配唯一PID
2 | proc->parent = current;         // 父进程为当前进程
3 | nr_process++;                  // 系统进程数加1

```

- 调用 `get_pid` 分配唯一的进程 ID (PID) 。
- 建立父子进程关系 (`proc->parent = current`) 。
- 更新系统进程总数。

### 7. 将新进程加入调度队列

```
1 | list_add(&proc_list, &proc->list_link); // 加入全局进程链表
2 | hash_proc(proc); // 加入PID哈希表（加速查找）
3 | proc->state = PROC_RUNNABLE; // 设为可运行状态
```

- 将新进程添加到全局进程链表 `proc_list` 和哈希表 `hash_list`，便于管理和查找。
- 将进程状态设为 `PROC_RUNNABLE`，使其可被调度器选中执行。

## 8. 返回结果

```
1 | ret = proc->pid; // 成功时返回子进程PID
```

- 若所有步骤成功，返回子进程的 PID；否则返回对应的错误码。

## 9. 错误处理

通过 `goto` 语句实现多级错误处理，确保资源释放的正确性：

```
1 | bad_fork_cleanup_kstack:
2 |     put_kstack(proc);
3 | bad_fork_cleanup_proc:
4 |     kfree(proc);
5 |     goto fork_out;
```

- 若内核栈分配失败 (`bad_fork_cleanup_proc`)：释放 PCB。
- 若内存复制失败 (`bad_fork_cleanup_kstack`)：释放内核栈和 PCB。

通过以上步骤，我们完成了 `do_fork` 函数的所有功能，可以实现为新内核线程分配资源！

## (3) 问题回答

`ucore` 能够保证给每个新 `fork` 的线程分配唯一的 ID (PID)，分析如下：

`ucore` 中负责分配 PID 的函数是 `get_pid()`，他保证了 PID 的唯一性。

```
1 | static int
2 | get_pid(void)
3 |
4 | {
5 |     static_assert(MAX_PID > MAX_PROCESS); // 确保PID的范围足够大
6 |     struct proc_struct *proc; // 进程控制块指针，用于遍历进程
7 |     list_entry_t *list = &proc_list, *le; // 进程链表头和遍历指针
8 |     static int next_safe = MAX_PID, last_pid = MAX_PID; // 下一个安全的PID检查
9 |     // 起点，上次分配的PID，初始值为MAX_PID
10 |    if (++last_pid >= MAX_PID) // 递增PID，若超过最大值则从1重新开始
11 |    {
12 |        last_pid = 1;
13 |        goto inside; // 跳转到内部逻辑，处理循环后的检查
14 |    }
15 |    if (last_pid >= next_safe) // 若当前PID超过下一个安全检查点，需要重新确定检查范围
16 |    {
17 |        inside:
18 |            next_safe = MAX_PID; // 重置下一个安全检查点为最大值
19 |            repeat:// 重复检查标签，用于冲突时重新遍历
20 |                le = list; // 从进程链表头开始遍历
21 |    }
22 | }
```

```

19         while ((le = list_next(le)) != list) // 遍历所有进程，检查当前last_pid是否已被使用
20     {
21         proc = le2proc(le, list_link); //利用宏，从链表项获取进程控制块
22         if (proc->pid == last_pid)// 若当前PID已被占用
23         { // 递增PID，若超过当前安全检查点则重新调整范围
24             if (++last_pid >= next_safe)
25             {
26                 if (last_pid >= MAX_PID)
27                 {
28                     last_pid = 1;
29                 }
30                 next_safe = MAX_PID;// 重置安全检查点
31                 goto repeat;// 重新遍历检查新的PID
32             }
33         } // 记录下一个可能的安全检查点（缩小后续遍历范围）
34         else if (proc->pid > last_pid && next_safe > proc->pid)
35         {
36             next_safe = proc->pid;
37         }
38     }
39 }
40 return last_pid;// 返回找到的唯一PID
41 }

```

- 在我们每次分配 PID 时，会遍历全局进程链表 `proc_list`，检查候选 PID 是否已被其他进程使用。从若已被使用，则递增 PID 并重新检查，直到找到未被使用的 PID。
- 当 PID 达到 `MAX_PID` 时，会从 1 开始重新循环检测，确保在系统进程数未达上限时总能找到可用 PID。

其中，`last_pid` 用于记录上一次成功分配的 PID，其核心作用是作为下一次分配 PID 时的起始候选值，优化 PID 分配效率。初始时把他设为最大 PID 值，每次调用 `get_pid`，将 `last_pid` 递增，若递增后超过 `MAX_PID`，则重置为 1。

递增后的 `last_pid` 会与系统中所有已存在进程的 PID 进行比对（通过遍历 `proc_list`）。如果该值已被占用，则继续递增 `last_pid` 并重新检查，直到找到一个未被使用的 PID。

这样我们在分配 PID 时无需从 1 开始遍历，而是从上一次分配的 PID 附近开始检查，减少了不必要的遍历操作。

## 三、练习三：编写proc\_run 函数

### (1) 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 `tsatp(unsigned int pgdir)` 函数，可实现修改 SATP 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

## (2) 设计实现

proc\_run 函数的核心功能是将指定进程 proc 切换为当前运行的进程，我们的实现流程如下：

```

1 void proc_run(struct proc_struct *proc)
2 {
3     if (proc != current)
4     {
5         struct proc_struct *prev = current;
6         bool intr_flag;           // 声明中断标志变量
7         local_intr_save(intr_flag); // 禁用中断并保存当前中断状态
8         current = proc; // 更新全局当前进程为目标进程
9         lsatp(proc->pgdir); // 切换页表，使用目标进程的地址空间
10        switch_to(&(prev->context), &(proc->context)); // 切换进程上下文
11        local_intr_restore(intr_flag); // 恢复中断状态
12    }
13 }
```

- **参数检查：**，如果目标进程 proc 与当前进程 current 相同，则无需切换，直接返回。
- **保存当前进程：**将当前进程 current 保存到 prev 变量，用于后续上下文切换。
- **禁用中断：**进程切换是内核的关键操作，需要保证原子性，我们将目前中断状态保存在 intr\_flag 中，防止切换过程中被外部中断如时钟中断干扰，导致状态不一致，调用 local\_intr\_save(intr\_flag) 禁用 CPU 中断。
- **更新当前进程：**将全局变量 current 指向目标进程 proc。
- **切换地址空间：**
  - 每个进程有独立的页表 (pgdir)，用于管理虚拟地址到物理地址的映射。
  - 我们采用 lsatp(proc->pgdir) 修改 RISC-V 架构的 satp (超级用户地址转换与保护) 寄存器，加载目标进程的页表，使 CPU 后续访问的虚拟地址映射到 proc 对应的物理内存，实现地址空间的隔离。
- **上下文切换：**
  - 上下文 context 是进程运行的状态快照，包括 CPU 寄存器 (如 ra、sp、s0-s11 等) 的值。
  - switch\_to(&(prev->context), &(proc->context)) 是汇编实现的关键函数，功能是：
    - 保存 prev 进程的上下文到 prev->context (如当前的 ra、sp 等寄存器值)，使其暂停运行时状态不丢失。
    - 从 proc->context 恢复目标进程的上下文到 CPU 寄存器，使 proc 从上次暂停的位置继续运行。
- **恢复中断：**

- 切换完成后，我们还需要通过 `local_intr_restore(intr_flag)` 恢复之前保存的中断状态，避免系统长期无法响应中断。

至此，我们就实现了两个进程之间安全，原子交换！

### (3) 问题回答

我们创建并运行了 2 个内核线程，分别是：

#### 1. idle 进程 (idleproc)

- 系统初始化时创建的第一个进程，用于在没有其他可运行进程时占用 CPU，避免系统无进程可执行的情况。
- 在 `proc_init` 中初始化，最终通过 `cpu_idle` 函数进入运行状态。

#### 2. init 进程 (initproc)

- 系统初始化过程中创建的用户态初始化进程，负责执行初始化逻辑（如输出 "Hello world!!" 等信息）。
- 由内核线程通过 `kernel_thread` 函数创建，并通过调度器切换到运行状态。

## 四、实验结果

我们输入 `make qemu`，可以看到运行成功！输出 Hello world!!

```
Special kernel symbols:
entry 0xc020004a (virtual)
etext 0xc0203df0 (virtual)
edata 0xc0209030 (virtual)
end 0xc020d4e4 (virtual)
Kernel executable memory footprint: 54KB
memory management: default_pmm_manager
physical memory map:
    memory: 0x08000000, [0x80000000, 0x87ffff]. .
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en..., Bye, Bye. :)"
kernel panic at kern/process/proc.c:388:
    process exit!!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
○ keyan@keyan-u22:/opt/riscv/code/labcode/lab4$
```

## 五、扩展练习 Challenge:

### (1) 问题一：

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

这两个宏定义在 `sync.h` 中：

```

1 #ifndef __KERN_SYNC_SYNC_H__
2 #define __KERN_SYNC_SYNC_H__
3
4 #include <defs.h>
5 #include <intr.h>
6 #include <riscv.h>
7
8 static inline bool __intr_save(void) {
9     if (read_csr(sstatus) & SSTATUS_SIE) {
10         intr_disable();
11         return 1;
12     }
13     return 0;
14 }
15
16 static inline void __intr_restore(bool flag) {
17     if (flag) {
18         intr_enable();
19     }
20 }
21
22 #define local_intr_save(x) \
23     do { \
24         x = __intr_save(); \
25     } while (0)
26 #define local_intr_restore(x) __intr_restore(x);
27
28 #endif /* !__KERN_SYNC_SYNC_H__ */

```

- `local_intr_save(x)` 宏
  - 用于禁用中断并保存状态，调用调用 `__intr_save()` 函数，禁用中断，并将中断状态（是否原本允许中断）保存到变量 `x` 中。
- `local_intr_restore(x)` 宏
  - 用于恢复中断状态，调用 `__intr_restore(x)` 函数，根据保存的状态 `x` 决定是否重新启用中断。

涉及到的两个关键函数：

- `__intr_save()` 函数
  - 通过 `read_csr(sstatus)` 读取 RISC-V 架构的 `sstatus` 寄存器（超级用户状态寄存器）。
  - 检查 `sstatus` 中的 `SIE` 位（Supervisor Interrupt Enable，位 1）：若为 1，说明当前允许中断。
  - 若允许中断，则调用 `intr_disable()` 禁用中断，并返回 `1`（标记原状态为“允许”）；否则返回 `0`
- `__intr_restore(flag)` 函数
  - 它根据 `flag` 的值决定是否恢复中断。若 `flag` 为 `1`（表示调用 `local_intr_save` 前中断是允许的），则重新启用中断；否则保持中断禁用状态。
- `intr_enable()` 和 `intr_disable()` 函数

- 这两个函数直接操作寄存器，通过 `set_csr` 和 `clear_csr` 宏操作 `sstatus` 寄存器的 `SIE` 位：

- `SIE=1`：允许 CPU 响应超级用户模式下的中断（如时钟中断、外部设备中断）。
- `SIE=0`：禁止 CPU 响应中断，确保临界区代码不被打断。

他们的核心是通过读写 RISC-V 架构的 `sstatus` 寄存器中控制中断允许的 `SIE` 位，实现了中断的临时禁用和恢复。先保存当前中断状态，再禁用中断执行临界区，最后根据原状态恢复，既保证了临界区原子性，又不干扰系统整体的中断响应逻辑。

## (2) 问题二：

- 深入理解不同分页模式的工作原理（思考题）

`get_pte()` 函数（位于 `kern/mm/pmm.c`）用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作。这在操作系统中的分页机制下，是实现虚拟内存与物理内存之间映射关系非常重要的内容。

### 1. 代码相似原因

`get_pte()` 函数中有两段形式类似的代码，结合 `sv32`, `sv39`, `sv48` 的异同，解释这两段代码为什么如此相像。

RISC-V 的 `Sv32`、`Sv39`、`Sv48` 均为页表分页模式，核心是通过多级页表将虚拟地址映射到物理地址，三者的共性与差异如下：

- 共性：**均采用**多级页表结构**（至少 2 级），虚拟地址被划分为多个索引字段，分别用于索引各级页表，最终通过页表项（PTE）找到物理页帧。
- 差异：**
  - 级数不同：** `Sv32` 用 2 级页表（页目录 `PD1`→页目录 `PD0`→页表 `PT`），`Sv39` 用 3 级，`Sv48` 用 4 级。
  - 虚拟地址划分不同：** 各级索引的位数不同（如 `Sv32` 的 `PD1` 索引 9 位、`PD0` 索引 9 位、页内偏移 12 位），但每级索引均为 9 位（除 `Sv32` 可能有例外，具体取决于实现）。
  - 页表项结构：** 核心标志位（如 `PTE_V` 有效位、权限位）一致，均通过页表项的高位存储物理页号（`PPN`）。

`get_pte` 函数的核心功能是在多级页表中查找虚拟地址 `1a` 对应的页表项（PTE），若 `create` 为 `true` 且对应页表项不存在，则自动创建所需的页表（分配物理页并初始化页目录项）。

两级页表的索引过程：

```

1 pde_t *pdep1 = &pgdir[PDX1(1a)]; // 获取一级页目录项的地址
2 if (!(pdep1 & PTE_V)) {           // 检查一级页目录项是否有效 (PTE_V 位)
3     // 若无效且允许创建，则分配新页作为二级页目录
4     struct Page *page;
5     if (!create || (page = alloc_page()) == NULL) {
6         return NULL; // 不创建或分配失败，返回 NULL
7     }
8     set_page_ref(page, 1);           // 初始化页引用计数
9     uintptr_t pa = page2pa(page);   // 获取新页的物理地址
10    memset(KADDR(pa), 0, PGSIZE); // 清零新页 (初始化二级页目录)
11    // 设置一级页目录项: 物理页号 (PPN) + 权限 (用户可访问 + 有效)
12    *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);
13 }
```

- 一级页目录项（`pdep1`）指向二级页目录的物理页。若该页目录项无效（未映射），则分配新物理页作为二级页目录，并更新一级页目录项使其指向新页。

```

1 // 通过一级页目录项的地址，获取二级页目录的内核虚拟地址，并定位到二级页目录项
2 pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(1a)];
3 if (!(pdep0 & PTE_V)) { // 检查二级页目录项是否有效
4     // 若无效且允许创建，则分配新页作为页表（PT）
5     struct Page *page;
6     if (!create || (page = alloc_page()) == NULL) {
7         return NULL;
8     }
9     set_page_ref(page, 1);
10    uintptr_t pa = page2pa(page);
11    memset(KADDR(pa), 0, PGSIZE); // 清零新页（初始化页表）
12    // 设置二级页目录项：物理页号（PPN）+ 权限（用户可访问 + 有效）
13    *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
14 }
```

- 二级页目录项（`pdep0`）指向页表（PT）的物理页。若无效，则分配新物理页作为页表，并更新二级页目录项使其指向新页。

```

1 // 通过二级页目录项的地址，获取页表（PT）的内核虚拟地址，再定位到具体页表项
2 return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(1a)];
```

- 最终通过页表索引（`PTX`）从页表中找到虚拟地址 `1a` 对应的页表项，并返回其内核虚拟地址。

我们使用**Sv32 分页模式**（32位虚拟地址），虚拟地址划分为三级索引：

- `PDX1`：一级页目录索引（最高位9位）
- `PDX0`：二级页目录索引（中间9位）
- `PTX`：页表索引（低9位）
- 页内偏移：最低12位（`PGOFF`）

代码中使用 `PDX1`（一级索引）和 `PDX0`（二级索引），对应 Sv32 的两级页表。两级索引的处理逻辑完全相同（检查有效性→分配页→设置页表项），因此代码形式相似。

无论 Sv32/Sv39/Sv48，每级页表的索引逻辑一致——通过虚拟地址的对应字段索引页表项，若页表项无效则分配新页作为下一级页表，并设置页表项。这种递归性导致各级索引代码结构高度相似。

## 2. `get_pte()` 中查找与创建功能的合并是否合理

**目前`get_pte()`函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？**

我们认为当前的实现方式确实存在一定优势，其将查找与创建逻辑整合在同一函数中，减少了函数调用带来的额外开销，同时在加锁场景下能减少锁的获取与释放次数，从而保证“查找→不存在则创建”操作的原子性，对于操作系统中常见的“查找页表项时若不存在则自动创建”场景（如进程初始化映射、页错误处理），这种整合方式可简化调用逻辑。

但是我们认为也存在一些设计问题：

1. 一个函数同时承担“查询”和“修改”两种操作，使得代码的可读性和可维护性降低。

- 当存在“仅查找不到创建”的场景（如检查地址是否已映射）时，需通过 `create` 参数进行控制，这不仅增加了函数的参数复杂度，也使得函数内部逻辑分支增多，增加了出错概率。
- 如果在创建过程中有内存分配失败等错误，错误处理与查找逻辑混合在一起，不容易错误定位。

如果进行函数拆分，也是比较合理的：

- 可以将原函数拆分为 `find_pte(pgdир, la)`（仅负责查找，不进行创建操作）和 `create_pte(pgdир, la)`（负责查找并在需要时创建），每个函数仅处理单一任务，逻辑更加清晰，便于理解和后续调试。
- `find_pte` 可独立应用于无需创建页表项的场景（如页表项检查、解除映射等），避免了代码的重复编写，减少了冗余
- 更容易定位错误，内存分配失败等创建过程中的错误可在 `create_pte` 中集中处理，使得错误处理更高效。

## 六、实验知识点对应

### (1) 实验与OS原理知识点对应

#### 1. 进程控制块 (`proc_struct`) 与初始化——对应PCB原理

实验中 `proc_struct` 存储进程状态、`PID` 等核心信息，是 OS 原理 PCB 的简化实现，仅适配内核线程，原理中 PCB 需覆盖用户进程等多场景，字段更完整，但核心都是“通过 PCB 管理进程”。

#### 2. 内核线程创建 (`do_fork` 等) —— 对应进程 / 线程创建原理

实验 `do_fork` 实现资源分配、状态初始化等核心流程，是原理中 `fork` 机制的简化版。我们的实验仅创建内核线程，实际中 `fork` 需处理用户态内存、文件描述符等资源的复制 / 共享。

#### 3. 进程调度 (`schedule`、`switch_to` 等) —— 对应调度与上下文切换原理

我们的实验以 FIFO 策略选进程，通过 `proc_run + switch_to` 完成切换，实现原理中“调度选择→上下文切换”核心逻辑。实验仅简单 FIFO 调度，而实际应用中支持 RR 等复杂算法；实验仅保存内核态寄存器，原理需兼顾用户态现场。

#### 4. 页表查找 (`get_pte`) —— 对应多级页表与虚拟内存原理

实验通过两级页表索引定位 PTE，是原理多级页表映射的具体实现。实验仅适配 Sv32，实际支持多级结构，且含页面置换等扩展机制。

#### 5. 中断控制 (`local_intr_save/restore`) —— 对应临界区保护原理

实验通过屏蔽中断保护进程切换等临界区，是原理“中断屏蔽保护临界区”的直接应用。我们的实验仅用中断屏蔽，实际应该还支持信号量等多处理器适配方案。

### (2) OS原理重要但是没有覆盖的知识点

- 死锁检测与避免
- 页面置换、按需分页
- 多级反馈队列、优先级调度等复杂算法
- 用户态进程与系统调用完整流程
- 进程间通信（管道、消息队列等）
- 虚拟内存页面共享与置换策略