Lab 4：进程管理

小组成员：吴昊然 2312321 袁宇菡 2312554 葛佳琦 2313202

实验目的

- 了解虚拟内存管理的基本结构，掌握虚拟内存的组织与管理方式
- 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

实验内容

在前面的实验中，完成了物理内存管理和基础页表机制的实现，使内核具备了对物理内存页的分配与管理能力，并能够建立起虚拟地址到物理地址的基本映射结构。但当前系统仍然只能以单一执行流的方式运行，无法并发执行多个任务，也尚未体现虚拟内存机制在进程或线程隔离中的作用。

本次实验将在此基础上进一步扩展，完成以下两方面内容：

首先，**进一步完善虚拟内存管理，实现基本的地址空间结构**。通过引入虚拟内存描述结构，管理进程或线程的虚拟地址空间布局，为每个执行实体提供逻辑上的运行空间。与后续实验中的缺页异常和页面置换不同，本实验中的虚拟内存仍采用预映射方式，即在建立地址空间时一次性完成所有需要的页表映射，不涉及按需分配或页面置换。

其次，**引入内核线程机制，实现多执行流并发运行能力**。内核线程是一种特殊形式的“进程”。当一个程序加载到内存中运行时，首先通过 ucore OS 的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用 CPU 来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自

拥有“自己”的 CPU。本实验将实现**线程控制块、上下文切换和调度器**等内容，从而实现多线程并发执行，使内核能够调度多个执行实体轮流使用 CPU 运行。

内核线程与用户进程的区别如下：

比较项	内核线程	用户进程
运行模式	仅在内核态运行	在用户态和内核态之间切换
地址空间	共享内核地址空间	拥有独立的用户虚拟地址空间

通过本次实验，系统将从“单一执行流的内核”发展为“支持多线程调度的内核”，并完成基本的虚拟内存环境框架搭建，为后续实现用户进程、系统调用、缺页异常处理和页面置换等功能奠定基础。需要注意的是，在 ucore 的调度和执行管理中，对线程和进程做了统一的处理。且由于 ucore 内核中的所有内核线程共享一个内核地址空间和其他资源，所以这些内核线程从属于同一个唯一的内核进程，即 ucore 内核本身。

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。

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）

  (1)struct context context

含义：

- 进程的**执行上下文**，保存进程切换时的 CPU 寄存器状态
- 包含：返回地址(ra)、栈指针(sp)、callee-saved 寄存器(s0-s11)，也叫被调用者保存寄存器（按照 RISC-V 调用约定，这些寄存器的值需要在函数调用前后保持不变）

在本实验中的作用：

- **进程切换的核心数据结构**：proc_run() → switch_to()通过 context 实现进程切换，switch_to()函数通过 context 完成两个进程间的切换
- **保存进程执行现场**：当进程被切换出 CPU 时，通过 switch_to()函数将当前进程的寄存器保存到 context 中：

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        struct proc_struct *prev = current;


        // 调用 switch_to 进行上下文切换
        switch_to(&prev->context, &proc->context);
```

- **恢复进程执行现场**：当进程被调度重新运行时，从 context.ra 指向的地址继续执行，从 context 中恢复之前保存的寄存器状态，使进程能够从上次被切换出去的地方继续执行。
- **实现控制流转移**：switch_to() 汇编函数通过操作 context 实现进程间的控制流转移 （在 kern/process/switch.S 中）

```
switch_to:
    # 保存当前进程的寄存器到 from->context
    STORE ra, 0*REGBYTES(a0)
    STORE sp, 1*REGBYTES(a0)
```

（2）struct trapframe *tf

含义：

- tf 是指向**中断帧（Trap Frame）的指针，中断帧保存了进程在发生中断或异常时的完整 CPU 状态**。
- trapframe 保存了**所有通用寄存器**以及一些特殊寄存器，如程序计数器(sepc)、状态寄存器(sstatus)等

在 kern/trap/trap.h 中

```
struct trapframe {
    struct pushregs gpr;    // 所有通用寄存器 (x0-x31)
    uintptr_t status;       // 状态寄存器 sstatus
    uintptr_t epc;          // 异常程序计数器 sepc（发生中断时的 PC）
```

在本实验中的作用：

- **设置新进程的初始状态：**在创建内核线程时，通过构造一个临时的 trapframe 来指定线程的入口函数和参数：

- **在进程创建时复制给新进程：**do_fork() 调用 copy_thread() 将临时 trapframe 复制到新进程的内核栈顶
- **中断处理：**发生时钟中断时，硬件/内核会把当前状态保存到 tf
- **特殊用途：**内核线程创建时，通过设置 tf 来指定线程的入口函数

两者的区别：

对比项	context	trapframe
保存时机	主动进程切换	中断/异常发生时
保存内容	部分寄存器(callee-saved)	完整 CPU 状态(所有寄存器)
使用场景	switch_to()	中断处理、进程创建

设计原因：

context 小而快：只保存必需的寄存器，减少切换开销

trapframe 全而准：保存完整状态，满足中断处理和进程初始化需求

为什么需要两个结构？

效率考虑：进程切换非常频繁，如果每次都保存/恢复所有寄存器会带来很大开销。根据 RISC-V 调用约定，caller-saved 寄存器由调用者负责保存，switch_to 只需处理 callee-saved 寄存器。

功能需求：中断处理和进程创建需要完整的 CPU 状态，仅靠 context 无法满足。

context 像书签：记录读到哪一页，下次打开书可以继续读

trapframe 像快照：完整记录某个时刻的所有信息，可以完全恢复当时的状态

（3）alloc_proc 函数分析

```
static struct proc_struct *alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;          // 状态：未初始化
        proc->pid = -1;                     // PID：-1 表示未分配
        proc->runs = 0;                     // 运行次数：0
```

设计实现过程：

① **分配内存**：使用 kmalloc 分配一个 proc_struct 大小的内存块

② **初始化各字段**：对进程控制块的所有字段进行初始化，主要包括：

·状态信息：state 设为 PROC_UNINIT，表示进程刚分配，尚未完全初始化

·标识信息：pid 设为-1，表示尚未分配有效的进程 ID

·资源信息：kstack 设为 0，内核栈尚未分配；mm 和 parent 设为 NULL

·调度信息：runs 设为 0（未运行过），need_resched 设为 0（不需要调度）

·执行上下文：context 清零，tf 设为 NULL（稍后在进程创建时设置）

·地址空间：pgdir 设为 boot_pgdir_pa，内核线程共享内核页目录

·其他：flags 清零，name 清空

③ **返回指针**：返回初始化好的进程控制块指针；分配失败则返回 NULL

总结：

在 alloc_proc() 函数中：

- 将 context 清零，因为新进程的执行上下文会在 copy_thread()中设置
- 将 tf 设为 NULL，因为中断帧会在 copy_thread() 中分配并初始化

这两个字段是进程管理的核心数据结构，context 负责进程切换，trapframe 负责进程初始化和中断处理，两者配合实现了多进程并发执行。

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。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。

（一）设计实现过程

1. 函数功能概述

do_fork 函数负责创建一个新的内核线程，是 ucore 中进程创建的核心函数。该函数通过复制当前进程的部分状态，并为新进程分配独立的资源（如内核栈、进程控制块等），最终创建出一个可运行的新进程。

2. 完整代码实现

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
{
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    // 检查进程数是否超过上限
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    // 步骤 1：分配进程控制块 PCB
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    // 步骤 2：分配内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    // 步骤 3：复制内存管理信息（kernel thread → 直接返回）
```

```
// 错误处理：逐级释放资源
bad_fork_cleanup_kstack:
    put_kstack(proc);           // 释放内核栈
bad_fork_cleanup_proc:
```

参数说明：

- ✓ clone_flags：控制复制行为的标志位
- ✓ CLONE_VM：共享内存空间（内核线程）
- ✓ 其他标志位用于用户进程（后续实验）
- ✓ stack：用户栈指针（内核线程为 0）
- ✓ tf：中断帧指针，包含新进程的初始寄存器状态

返回值：

成功：新进程的 PID

失败：错误码（负数）

错误处理机制：

- ❖ 如果步骤 2 失败（内核栈分配失败）：

→ bad_fork_cleanup_proc → 释放进程控制块 → 返回错误

- ❖ 如果步骤 3 失败（内存管理复制失败）：

→ bad_fork_cleanup_kstack → 释放内核栈 → bad_fork_cleanup_proc → 释放进程控制块 → 返回错误

（二）问题回答

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。

ucore **能够保证每个新 fork 的线程有唯一的 ID**

do_fork 函数在步骤 5（插入进程管理结构）中的 get_pid()函数通过**递增分配 + 冲突检测 + 安全窗口优化 + 原子性保护**，确保了每个进程都有唯一的 ID。

1. get_pid() 函数实现分析

```
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;
    // 递增分配
    if (++last_pid >= MAX_PID) {
        last_pid = 1;           // 循环回 1（0 保留给 idle 进程）
        goto inside;
    }
    // 快速路径：如果在安全窗口内，直接返回
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        // 遍历所有进程，检查 PID 冲突
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            // 发现冲突，递增并重新检查
            if (proc->pid == last_pid) {
                if (++last_pid >= next_safe) {
```

2. 算法核心思想

算法核心：

- **last_pid 单调递增**
- **达到 MAX_PID 后从 1 重新开始**
- **遍历当前进程链表检查冲突**
- **使用 next_safe 优化，避免每次都遍历全链表，让 get_pid()大多数时候可以 O(1)分配 PID**

基本策略：递增分配 + 安全窗口优化

(1) PID 空间足够大

```
static_assert(MAX_PID > MAX_PROCESS);
```

实际情况：PID 空间至少是进程数的两倍。

只要系统允许创建新进程，则一定存在一个空的 PID。

(2) 快速路径（Fast Path）

```
if (++last_pid >= MAX_PID) {
    last_pid = 1;
    goto inside;
}
if (last_pid >= next_safe) {
```

- **变量说明：**

last_pid（静态变量）：上一次分配的 PID

next_safe（静态变量）：下一个已被占用的 PID

- **快速路径条件：last_pid < next_safe**

含义：在 [last_pid, next_safe) 这个**安全区间**内的 PID 都是空闲的

操作：直接返回 last_pid，无需遍历进程链表

时间复杂度：O(1)

(3)**冲突检测**（Slow Path）

当 last_pid >= next_safe 时，需要遍历所有进程检查冲突：

```
while ((le = list_next(le)) != list) {
    proc = le2proc(le, list_link);
    // 情况 1：发现冲突
    if (proc->pid == last_pid) {
        if (++last_pid >= next_safe) {
            // 超出当前安全窗口，重新扫描
            if (last_pid >= MAX_PID) {
                last_pid = 1;
            }
            next_safe = MAX_PID;
            goto repeat;
```

（4）原子性保证

do_fork 在**关中断的临界区中**执行：（这三步必须在关中断内连在一起做）

```
proc->pid = get_pid(); // 1. 分配 PID
hash_proc(proc);       // 2. 把 PID 登记到哈希表
```

则在此期间不会发生中断，不会有另一个线程执行到 do_fork，不会存在多个线程同时执行 get_pid，不会在 get_pid 和 list_add 之间暂停，相当于把这三步变成了一个不可拆分的原子操作（即**分配 PID + 注册 PID**）

get_pid 必须依赖“当前进程链表”的最新状态，即 get_pid 分配出一个新的 PID 后必须马上把它登记到链表里，否则后来的 get_pid()看到的是不完整的链表，会分到重复的 PID。

步骤	含义
get_pid()	计算一个当前未使用的 PID
hash_proc(proc)	在哈希表中登记这个 PID 已被占用
list_add(proc_list)	在全局链表中登记这个进程存在

总结：ucore 使用以下机制确保 PID 唯一

- **get_pid() 会遍历当前进程链表，确保返回未被占用的 PID。**
- **PID 分配 + 插入链表 + 插入哈希表 在关中断的临界区一次性完成。**

- **临界区保证了这些关键操作不可被打断，防止竞态条件。**

因此 ucore 能够保证在任何并发情况下，新创建的进程一定会得到唯一的 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 lsatp(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

（一）设计实现过程

1. 函数功能概述

proc_run 函数负责**将指定的进程切换到 CPU 上运行**，是操作系统进程调度的核心函数。该函数完成进程切换的全部工作，包括更新当前进程指针、切换页表和

执行上下文切换。

2. 完整代码实现

参数说明： proc 是要切换到的目标进程

功能：将 CPU 的控制权从当前进程转移到目标进程

```
void proc_run(struct proc_struct *proc)
{
    if (proc != current) //只有当目标进程不是当前进程时才需要切换
    {
        bool intr_flag;
        struct proc_struct* prev = current, * next = proc;
        // 步骤 1：禁用中断
        local_intr_save(intr_flag);

        // 步骤 2：切换当前进程指针
        current = proc;

        // 步骤 3：切换页表
        if (next->pgdir != NULL) {
            lsatp(next->pgdir);
```

3. 详细过程分析

- **步骤 1：禁用中断**

local_intr_save 的作用：（这个宏定义在 kern/sync/sync.h 中）

```
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
```

① 读取当前中断状态（RISC-V 的 sstatus.SIE 位）

（SIE = 1 时中断启用；SIE = 0 时中断禁用）

② 将当前中断状态保存到 intr_flag 变量中

③ 清除 sstatus.SIE 位，关闭中断

禁用中断：进程切换是临界区操作，必须保证原子性，不能被中断打断。

- **步骤 2：切换当前进程指针**

```
struct proc_struct* prev = current, * next = proc;
        // 步骤 2：切换当前进程指针
```

作用：保存旧进程指针到 prev，保存新进程指针到 next

更新全局变量 current，指向新进程

current 变量在 kern/process/proc.h 中声明

```
extern struct proc_struct *idleproc, *initproc, *current;

//idle 进程指针（空闲时运行）,init 进程指针（第一个用户态进程）, current 当前运行的进程指针
```

- **步骤 3：切换页表**

每个用户进程有独立的虚拟地址空间，不同进程的相同虚拟地址可能映射到不同的物理地址，必须切换页表，才能正确访问新进程的内存

- **步骤 4：上下文切换**

· 调用汇编函数 switch_to(&prev->context, &next->context)

· 保存：将 prev 的寄存器（ra, sp, s0-s11）保存到 prev->context

· 恢复：从 next->context 恢复寄存器

· 结果：CPU 开始执行 next 进程

- **步骤 5：恢复中断**

使用 local_intr_restore(intr_flag) 恢复之前的中断状态

（二）问题回答：创建并运行了几个内核线程？

分析 proc_init() 函数

本实验的执行过程中，创建并运行了 2 **个内核线程**

1．第 0 个线程：idleproc（PID：0）

创建方式：直接调用 alloc_proc()（不是通过 fork）

作用：系统空闲时运行，不断检查是否需要调度

功能：在 cpu_idle()函数中循环检查 need_resched，触发调度

2．第 1 个线程：initproc （PID：1）

创建方式：通过 kernel_thread() → do_fork()创建

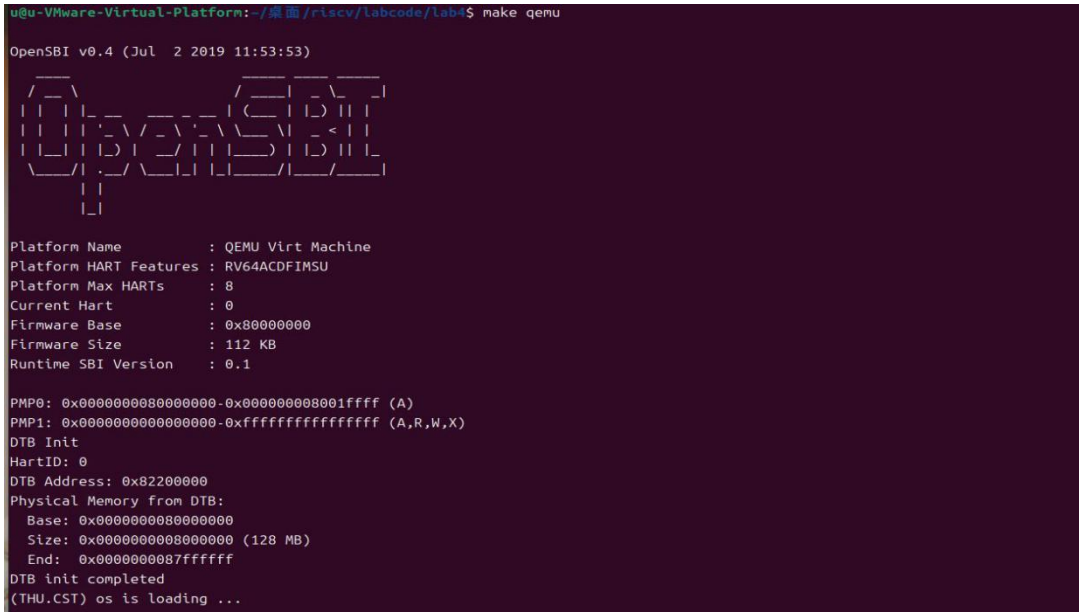
作用：第一个通过 fork 创建的内核线程

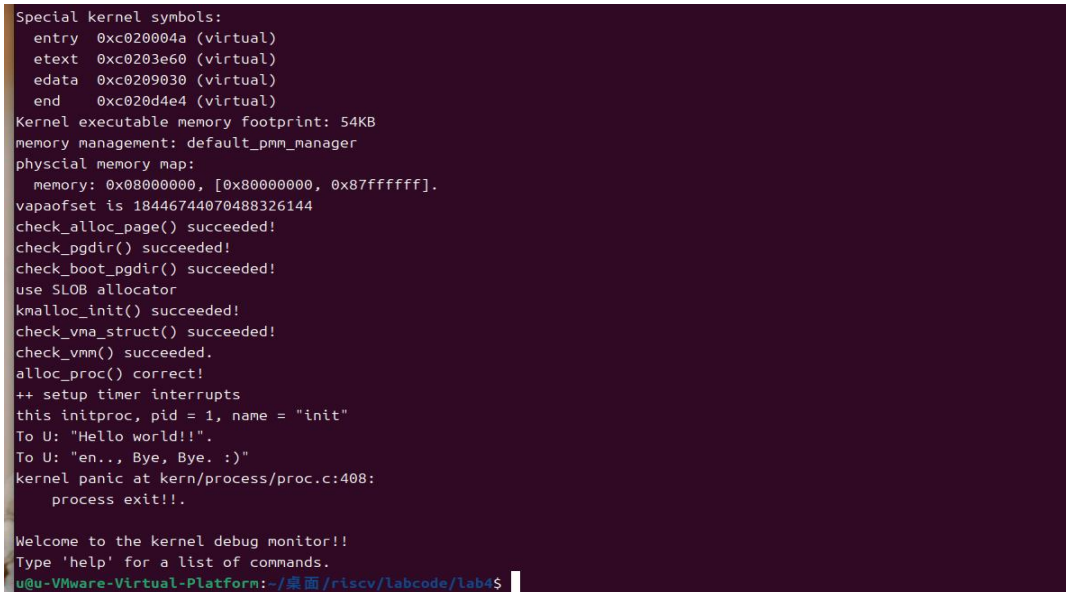
功能：执行 init_main() 函数，打印测试信息

运行流程：

- 系统启动后首先运行 idleproc
- idleproc 检测到 need_resched=1，调用 schedule()调度函数
- 调度器选择 initproc 运行
- initproc 执行完毕后，控制权返回 idleproc
- idleproc 继续循环，等待下一次调度

编译运行成功，截图如下





扩展练习 Challenge：

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？**

在文件 sync.h 中有 local_intr_save(intr_flag);的宏定义，

```
#define local_intr_save(x) \
      do {                 \
          x = __intr_save(); \
      } while (0)
```

它主要是通过__intr_save()来实现功能。在同文件中__intr_save()：

```
static inline bool __intr_save(void) {
      if (read_csr(sstatus) & SSTATUS_SIE) {
          intr_disable();
          return 1;
      }
      return 0;
  }
```

read_csr(sstatus)是读取 sstatus 寄存器（记录当前 CPU 的特权态信息和控制状态），read_csr(sstatus) & SSTATUS_SIE 是读取状态寄存器的 SIE 位，若 SIE=1（表示内核态允许中断），那么就调用 intr_disable()函数，将 SIE 位置零，让内核态不再允许被打断。之后 return 1 表示原来是允许中断的，return 0 表示原来是不允许中断的。

通过以上分析，可以得出，local_intr_save()这个宏定义，是用来实现**关闭允许中断状态**的。

sync.h 中对于 local_intr_restore()的宏定义如下：

```
#define local_intr_restore(x) __intr_restore(x);
```

它主要是通过__intr_restore()来实现功能。

```
static inline void __intr_restore(bool flag) {
      if (flag) {
          intr_enable();
      }
  }
```

它的功能和__intr_save()相对应，是配套使用的。这里传入的 flag 参数就是上面__intr_save(void)的返回值，表示调用前中断是否开启。如果 flag=1 说明原来的状态就是允许中断的，就通过 intr_enable()将 SIE 位置为 1，**恢复原来的允许中断状态**；如果 flag=0 说明原来的状态就是不允许中断的，那么保持现有状态就行了。

综上所述，local_intr_save(intr_flag);用来**保持原有状态并关闭中断**，

local_intr_restore(intr_flag);**恢复到原有状态**，中间就可以进行一些临界区操作。

2.**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。

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。

- **sv32，sv39，sv48 的异同：**

在 RISCV 中，每个页大小固定为 4KB，4KB=2^{12}，所以无论是在哪种分页机制下，虚拟地址的**最低 12 位**都是用来表示**页内偏移**。

sv32 固定用于 **32 位操作系统**，sv39 和 sv48 固定用于 **64 位操作系统**。所以 sv32 的页表项是 32bit，而 sv39 和 sv48 的页表项是 64bit。

那么对于 sv32 来说，每一页可以容纳

$$4KB \div 4B = 1KB = 1024\ 项 = 2^{10}$$

也就是需要 **10bit 来索引页表项**，在 sv32 下，虚拟地址除去最低的 12 位，还剩 20 位，正好可以建立**两级页表**。

同理，对于 sv39 和 sv48，每一页容纳

$$4KB \div 8B = 512\ 项 = 2^{9}$$

除去固定的页表偏移 12 位，分别剩下 27 位、36 位，也就建立对应的**三级页表**和**四级页表**。

回到 get_pte()函数：

```
pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
 {
     pde_t *pdep1 = &pgdir[PDX1(la)];
     if (!(*pdep1 & PTE_V))
```

```
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_
V);
    }
    pde_t *pdep0 = &((pte_t *)KADDR(PDE_ADDR(*pdep1)))[PD
X0(la)];
    if (!(*pdep0 & PTE_V))
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_
V);
    }
    return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
}
```

可以看到其中相似的部分是：

```
if (!(*pdep1 & PTE_V))
    {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL)
        {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_
V);
    }
```

- `if (!(*pdep1 & PTE_V))`检查当前页表项是否有效
- `*pdep1 & PTE_V` 非 0 表示此目录项已经被设置为**指向下一级页表页**；等于 0 则说明**下一级页表页尚未存在**。如果下一级页表页不存在则准备创建页表页。
- `if (!create || (page = alloc_page()) == NULL)`，如果 `create` 为 `false`，则调用者**不允许创建缺失的页表页**，函数直接返回 `NULL`（表示找不到 PTE）；或者如果 `alloc_page()` **分配失败**（例如内存耗尽），也返回 `NULL`。
- 之后，`set_page_ref(page, 1);`将新分配的页的引用计数设为 1（或增加到 1），防止该页被回收或重用。
- `uintptr_t pa = page2pa(page);`把 `struct Page *` 转成对应的**物理地址** `pa`，接下来需要通过内核虚地址访问这页内存（例如 `memset`），所以需要物理地址。
- `memset(KADDR(pa), 0, PGSIZE);`把整页**清零**。目的有两个：
    - 初始化页表页中的所有 PTE 为 0（即无效），**避免旧数据或随机内存值**被当作有效页表项导致错误；
    - 保证内存可见且一致，后续将目录项标记为有效时，其他 CPU 看到的页表页内容是**已经初始化**的。
- `*pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);`把目录项写回到 `pgdir` 中，指示该目录项现在**指向新分配**并**已初始化**的页表页。

之所以这两段代码如此相像，就是因为上述步骤在每一级当中都差不多，即为：

取索引 → 检查是否存在 → 如果不存在就创建 → 进入下一层，每一层的操作**只有索引不同**，结构完全一致。这里实现的应该是二级页表，也就有两段几乎相同的代码。

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？

我觉得是，拆开是更好的设计方案，尤其是在多核系统或复杂内核中，但保留一个封装函数（查找+按需创建）用于方便调用也是合理的。